TÉCNICO LISBOA

HawkSet: An Automatic, Agnostic, and Efficient Concurrent PM Bug Detection Tool

João Miguel Ferreira Oliveira

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Miguel Ângelo Marques de Matos

Examination Committee

Chairperson: Prof. João Pedro Faria Mendonça Barreto
Supervisor: Prof. Miguel Ângelo Marques de Matos
Member of the Committee: Prof. José Orlando Roque Nascimento Pereira

November 2023

Acknowledgments

This work was partially supported by Fundação para a Ciência e a Tecnologia (FCT) under grants PTDC/CCI-COM/4485/2021 (Ainur) project, and UIDB/50021/2020.

Resumo

A memória persistente (PM, do inglês *Persistent Memory*) é endereçável ao byte e durável, com tempos de acesso comparáveis aos de DRAM tradicional. Estas características permitem o desenvolvimento de aplicações com estado persistente, sem a utilização de operações de E/S custosas. No entanto, após uma falha arbitrária, o estado da aplicação pode ficar incoerente devido à falta de uma cache persistente, e à reordenação e atraso de operações. Os programadores de aplicações em PM, precisam de ter este facto em consideração, utilizando instruções de baixo nível próprias e mecanismos de recuperação específicos para garantir coerência à falha. Estes requisitos são complexos, e como tal motivaram a criação de várias ferramentas para a deteção de erros de PM. No entanto, as poucas ferramentas que procuram ativamente por erros concorrentes de PM, focam-se em domínios específicos, como as bases de dados chave-valor, e necessitam de artefactos complexos próprios para a deteção. Nesta tese, propomos o HawkSet, uma ferramenta para a deteção de erros concorrentes de PM de forma automática, agnóstica, e eficiente. O HawkSet, utiliza *lockset* analysis para detetar erros concorrentes de PM, e introduz uma heurística capaz de reduzir os falsos positivos e a utilização de recursos. Os resultados da nossa avaliação demonstram que o HawkSet é escalável, reportando os mesmos erros que o estado-da-arte, em menos tempo. Além disso, o HawkSet deteta um novo bug.

Palavras-chave: Memória Persistente, Concorrência, Deteção de Erros

Abstract

Persistent memory (PM) provides developers with byte-addressable and durable storage, with performance comparable to that of DRAM. This allows for the development of applications with persistent state, without employing costly HDD/SSD based IO operations. However, after a crash, the application state may become inconsistent due to the lack of a persistent cache or the reordering and stalling of operations. Taking this into consideration, developers must either prevent or fix inconsistent states, using low-level instructions and specific recovery mechanisms. Failure to do so can result in data loss or corruption, subsequent crashes or undefined behavior. These requirements incur a heavy toll on the developer, who must reason about possible incoherent application states and how to recover from them. This motivated the creation of several PM bug detection tools. However, the few tools that actively search for concurrent PM bugs focus on specific domains, such as key-value stores, and require the development of complex artifacts for debugging. In this thesis, we propose HawkSet, an automatic, application agnostic, and efficient concurrent PM bug detection tool. HawkSet uses *lockset* analysis to detect concurrent PM bugs, and employs a heuristic that is able to reduce false positives and resource consumption. The results of our evaluation show that HawkSet is scalable, reporting the same bugs as the state of the art, in less time. On top of that, we were able to detect a novel bug.

Keywords: Persistent Memory, Concurrency, Bug Detection

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Persistent Memory (PM) provides developers with byte-addressable and durable storage, with performance comparable to that of DRAM. This allows for the development of applications with persistent state, without employing costly HDD/SSD based IO operations.

Currently, there are a plethora of PM applications, which fall into a wide array of categories. Some examples are key-value stores [1, 2], hash tables [3, 4, 5], trees [6, 7], caches [8], file systems [9, 10], and other frameworks [11, 12]. These applications require a specialized design in order to properly support PM. Data is only persisted once it reaches PM, since the cache is generally not persisted. In order to guarantee that a particular write is persisted, a developer must issue a flush instruction, whose job is to move data from the cache to memory at a cache-line granularity. Modern CPUs create an additional complication, as they can stall writes and flushes by using a store buffer, and can even reorder them according to specific rules [13]. In single-threaded volatile applications, these stalls and reorders are completely invisible to the application, and do not pose a threat to consistency. However, in PM applications, this is not the case, since stores are only persisted after they reach PM. For example, assume that an application performs two stores, and it assumes they are executed in that order and therefore persisted in that order. The CPU may reorder these stores, performing the second one before the first. In this example, if the application crashes after the second store has been persisted, but before the first one is, then the post-crash state will be incoherent. Furthermore, in multi-threaded applications, stalls and reorders may cause other threads to observe incoherent states. For example, when accessing a shared memory region, even with a mutual exclusion mechanism, a store to a shared variable may be stalled by the CPU. In this example, when other threads access this value, they might observe the old value. To solve these problems, applications use fence instructions, which force the CPU to complete all the pending memory operations, before continuing, effectively ensuring stores before the fence are not stalled or reordered against those

that come after. In the context of multi-threaded applications, most synchronization mechanisms include memory fences, to ensure stores are visible to other threads. In the context of PM, developers must also create special procedures, recovery programs, that can detect and correct incoherent post-crash states that were not mitigated by use of low-level instructions.

Failure to properly use these mechanisms can result in data loss or corruption, subsequent crashes or undefined behavior. These requirements incur a heavy toll on the developer, who must reason about possible incoherent application states and how to recover from them. This motivated the development of a wide range of PM bug detection tools [14, 15, 16, 17, 18, 19, 20, 21, 22, 23].

We can see a parallel between the use of PM and concurrency: when used correctly concurrency allows for performance gains, however, it can lead to complex bugs, that are hard to reason about since they may depend on specific thread interleavings. With PM, we observe a similar scenario: its use enables performance gains, but it also leads to complex bugs. Given its relevancy, the literature about concurrency bug detection is vast [24].

The parallel between PM and concurrency does not end here, because, in fact, we can combine both for applications with high performance requirements, however, this combination creates a new complex programming challenge. To correctly implement a concurrent PM application, the developer must reason about the persistent state, the stalling and reordering of memory operations, the synchronization of shared memory accesses, and the impact that different thread interleavings may have in the correctness of the system. This increased challenge resulted in the recent proposals for concurrent bug detection in PM applications [22, 23]. However, these tools have certain limitations, namely, they are not application agnostic, since both focus on key-value stores (or equivalent applications), nor automatic, because they require the development of custom artifacts for debugging each application.

To address this, we propose HawkSet, a new tool that uses lockset analysis to detect concurrent PM races, in an automatic and application agnostic fashion. Lockset analysis allows HawkSet to detect races without needing to observe them, unlike PMRace [23], which needs to explore the exact interleaving that causes a particular bug, possibly even needing to explore it several times. We are also able to detect races without prior knowledge of the application, or any complex artifacts, unlike Durinn [22], which can only analyze key-value stores (or equivalent applications), and requires a complex driver. Even though lockset analysis is known to scale poorly for large amounts of memory accesses, our key insight is that the fraction of PM accesses amongst regular accesses, is small, approximately 4% [25]. Thus making this technique suitable for PM programs.

Contributions The work outlined in this thesis constitutes the following contributions:

- A lockset analysis algorithm tailored for race detection in PM applications
- A heuristic capable of addressing commonly faced problems in lockset analysis
- HawkSet, a full implementation of our proposal
- An empiric evaluation of HawkSet's ability to debug various PM applications
- The report of one previously unknown bug

Publications Part of the work described in this document has been published in João Oliveira, Miguel Matos and João Gonçalves. (2023). Deteção de Erros de Concorrência em Programas de Memória Persistente Utilizando Análise de Locksets. INForum.

The rest of this document is structured as follows: Chapter 2 presents vital background in PM and concurrency. In Chapter 3 we characterize the existing state-of-the-art in PM bug detection and outline some techniques used for concurrency bug detection. Chapter 4 further explains the problem, our solution and its implementation details. Chapter 5 describes the experimental results of our tool and compares it to the state-of-the-art. Finally, Chapter 6 concludes the document.

Chapter 2

Background

In this chapter, we lay out the main concepts behind our work: persistent memory and concurrency, including the specific semantics and bugs related to each discipline.

2.1 Persistent Memory Semantics

PM combines the byte-addressability of memory and the durability of storage while being able to maintain a performance on-par with traditional DRAM. Commercially available solutions, such as Intel's Optane DC memory, have allowed for increased adoption of PM. As is to be expected, the increased adoption of PM comes with an increased interest in guaranteeing correctness in PM systems, which in turn, motivates research in PM bug detection. To fully understand how to detect bugs in PM applications, we must first study its semantics and how to properly develop such applications.

Some applications, such as Hemem [26], use PM as a cheap extension to DRAM without taking advantage of its persistency guarantees. Those applications do not use PM to maintain a persistent state and therefore none of the concerns outlined in this document apply.

To properly maintain a persistent state, an application must guarantee that it is crash-consistent, in other words, it is able to successfully recover from an arbitrary crash at any point in the execution [27]. A successful recovery depends on the semantics of the application, but in general it must return the system to a valid state from which it can continue executing.

When discussing PM applications, it is useful to logically divide the execution in two parts. The first is the execution that occurs before the crash, the pre-crash execution, and the second is the one that comes after the crash, the post-crash execution, usually involving some recovery mechanism. It is also common to talk about the post-crash state, this is the state of the application after recovering from a crash.

The use of PM creates additional implementation difficulties that are generally not thought of in applications build on top of DRAM. To successfully achieve crash-consistency, PM developers need to take into account the persistency of stores and the order in which they occur since they are not guaranteed to execute in program order [13].

Figure 2.1 is a graphical representation of the semantics of PM. We say that stores are persisted, if they are in the persistent domain, in other words, they are guaranteed to be visible to the post-crash execution and, for the purposes of this document, in persistent memory. Everything else, belongs in the volatile domain, where data is not persisted and can be lost after a crash (i.e.: power failure), usually stored in the cache or volatile memory.

To control the stalling and reordering of stores in PM programs, hardware vendors provide several instructions for developers to take advantage of [28]. They can differ in name between architectures however, there are a few generalizations we can make regarding them:

Stores are commonly used to instruct the CPU to write in memory. Performing writes is extremely expensive, therefore, stores usually stay in the cache, which may be arbitrarily written-back to memory depending on the cache policy. However, modern CPUs take this optimization even farther, and can delay and reorder stores using the store buffer. This means that a program might have executed one or more store instructions while the cache and memory remain the same. Over the course of this document we use store and write interchangeably.

Non-temporal stores are a subset of store instructions that bypass the cache, being written to memory, however they can still be delayed like regular stores.

Flushes instruct the CPU to flush the data from cache to memory with cache-line granularity, however, CPUs may reorder these flushes with respect to other instructions, for example, another flush. It is important to note that these instructions are not the only way stores can be persisted, the cache might write-back data to memory depending on the cache eviction policy. Flushes ensure that data is written-back explicitly instead of arbitrarily.

Fences guarantee that all memory operations before them are visible after their execution, effectively forcing stalled stores to be executed and prohibiting reorders with respects to instructions that come after the fence. For example, 2 flushes separated by a fence instruction guarantees that the CPU executes the first flush instruction before the second one.

To prevent an inconsistent state after an arbitrary crash the developer must explicitly flush cache-lines, and order such flushes using fences. This process is not simple to implement, because it requires a considerable understanding of PM semantics, the cache, and of the application itself. The incorrect usage of PM can lead to performance and correctness bugs.

To aid in the development of PM applications, programmers can use libraries designed to en-

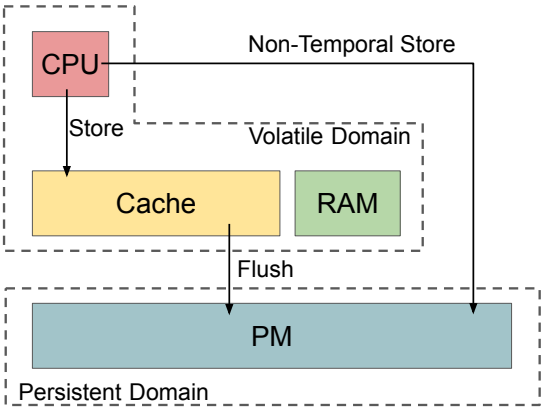


Figure 2.1: PM instructions and their effects on the domain of data. Each box denotes a storage medium. Arrows accompanied by the respective instruction denote the flow of data between storage mediums and domains.

sure crash-consistency, without requiring the developer to engage directly with low-level instructions. These libraries use different mechanisms, such as logging, to guarantee crash-consistency, if used correctly. Ensuring the correct use of such libraries can be as much of a challenge as ensuring the correct use of PM instructions. Not only that, but the libraries themselves can have bugs possibly affecting every other application that relies on them.

2.1.1 Note on Intel x86's Instruction Set

In Intel x86's instruction set there are three distinct types of flushing instructions [13]: `clflushopt` which is comparable to the flush primitive described above, `clflush` which cannot be reordered with respect to other flushes, `clwb` that might not invalidate cache-lines after the write-back. Over the course of this report we will refrain from using specialized names from a particular architecture, opting instead to follow the naming conventions described above: store, flush and fence.

2.1.2 Note on eADR Technology

Intel has released a feature in recent processors, called extended Asynchronous DRAM Refresh (eADR), that extends the persistent domain to the cache [29], effectively ensuring all stores are immediately made durable once they reach the cache. However, eADR does not solve all issues described later in this document: The CPU can stall stores from reaching the cache, which can cause PM bugs. Furthermore, semantic based violations are still possible even with a persistent cache (see Section 2.2.2). Not only that, but for the foreseeable future, eADR will likely not be available in every platform, and applications will not be able to rely on its existence. For these reasons we believe the study of bug reporting in PM programs will remain useful.

2.2 Persistent Memory Bugs

As discussed before, the adoption of persistent memory technology goes hand in hand with the surfacing of novel bugs. In this section we provide a consolidated bug taxonomy extrapolated from the state-of-the-art in persistent memory debugging, accompanied by examples.

2.2.1 Persistency Bugs

Missing Flush - these bugs occur when a store is never flushed. There are two possible reasons for this, either the flush is missing, and the store is not guaranteed to be persisted, therefore it results in a persistency bug or, the flush is not necessary because it is transient data, therefore it constitutes a performance bug (this will be discussed later in this section). To the best of our knowledge, there is no tool in the current state-of-the-art of PM debugging that can confidently and automatically distinguish between the two.

2.2.2 Semantic Bugs

Semantic bugs include all PM bugs that cause an inconsistent post-crash state after recovery due to a failure to adhere to the semantic requirements of the applications. In particular ordering and atomicity between store operations.

Algorithm 2.1: Ordering violation

```
1 value←VALUE;
2 key←KEY;
3 flush(&value);
4 flush(&key);
5 fence();
```

Algorithm 2.2: Atomicity violation

```
1 value←VALUE;
2 flush(&value);
3 fence();
4 key←KEY;
5 flush(&key);
6 fence();
```

The algorithms described above are examples of semantic bugs in persistent memory programs. All variables are assumed to be backed by PM.

Algorithm 2.1 represents a typical insertion on a key-value store, in it we can observe a potential ordering violation when the crash happens between lines 2 and 5. If `key` has been written-back to PM, but `value` has not, the post-crash state will be incoherent, an entry in the data structure points to a garbage value. This can happen because caches may write-back cache lines arbitrarily due to their eviction policy. This is an example of an ordering violation, when an application has a semantic requirement that is not guaranteed.

A simple way to fix this violation is to guarantee an order between both writes (and respective flushes), Algorithm 2.2 implements this fix. Since the fence instruction only completes when all

pending memory operations are executed, we guarantee that the second write will only occur after the first one has been persisted.

Now assume that our key-value store program has a new, stricter requirement, keys and values must be persisted atomically. The only valid states are ones before and after the insertion operation.

In Algorithm 2.2 we can observe a potential atomicity violation if the application crashes between lines 3 and 6. After recovery, the post crash-state will only observe the new value of `key`, which is not allowed by the requirement described above.

Algorithm 2.3: Atomic Operation

```
1 c← 0; flush(&c); fence();
2 backup_key_value();
3 key← KEY;
4 value← VALUE;
5 flush(&key);
6 flush(&value);
7 fence();
8 c← 1; flush(&c); fence();
```

Algorithm 2.4: Recovery Procedure

```
1 if c = 0 then
2     key← OLD_KEY;
3     value← OLD_VALUE;
```

To fix the atomicity violation a developer could take a similar approach to the one implemented in Algorithm 2.3, where the variable `c` (usually called commit variable) tells the recovery procedure described in Algorithm 2.4 whether the operation was fully completed. In this example, when the recovery procedure detects an incomplete operation, it resets the state to the initial value. Using this technique we can guarantee that, after recovery, no intermediate states are visible.

2.2.3 Performance Bugs

Transient Data in PM - PM tends to have a higher store latency than volatile memory, therefore storing data in a slower medium when it is not necessary can decrease performance. Currently, this bug is detected by finding memory regions in PM that are written to but never flushed, however, as detailed above, it is not trivial to distinguish this from a missing flush bug.

Extra Flush - flushing the same cache-line without any previous store to that region constitutes a performance bug, as the second flush is not necessary and will have no effect.

Extra Fence - fence instructions guarantee that all previous flushes are executed, if there is a fence instruction without pending flushes then the fence has no effect and could be removed.

Flush on Volatile Addresses - flushing a volatile memory address, in the context of persistent memory, is a pointless endeavor since by the end of the flushing operation the data is

still in the volatile domain.

2.3 Concurrency

Concurrent programs are able to harness more performance from the CPU, by either taking advantage of parallel execution or using dead time (i.e: waiting for IO) for useful operations. However, with great power comes great responsibility: a big challenge of concurrent programming involves the correct use of shared resources, by using several synchronization techniques [30].

Threads - a thread is a logical unit of execution that can run concurrently with other threads. In practice, programs use threads to achieve concurrency, distributing the workload among them. A thread has its own local storage and communicates with other threads using shared memory, it is here that caution must be taken to avoid races.

Races - a race condition occurs when two concurrent executions access shared variables concurrently and at least one of those accesses is a write.

Critical sections - to avoid data races, enforce atomicity, and ensure ordering between operations when handling shared variables, it is necessary to make sure that different threads do not access the same memory region at the same time. These sections of code where shared memory is accessed is the critical section.

Synchronization - there are several synchronization mechanisms that usually fall into one of the following categories: A `mutex` represents exclusivity, ensuring that only one thread may hold the mutex at any given point. It is commonly used to dictate whether a thread can access a shared memory region. A `semaphore` provides the ability to allow access to a particular critical section to a set number of threads, it maintains an atomic counter that is either incremented or decremented as threads release or acquire it, if a thread tries to acquire a semaphore whose counter is 0 it must wait until another thread releases it. A `condition variable` allows threads to execute only after some pre-condition is achieved. While a thread is waiting and another thread signals the condition variable, the waiting thread will awake and continue execution. If there are multiple threads waiting and another thread broadcasts the condition variable, all the waiting threads will awake. This synchronization mechanism creates a happens-before relationship from the thread that signals the condition variable to the threads that wait. A `barrier` allows the synchronization of threads, when an execution reaches a barrier it stops until all other threads reach it, at which point all threads are awoken and continue executing. To guarantee coherency, these synchronization mechanisms are usually implemented using hardware-provided atomic instructions.

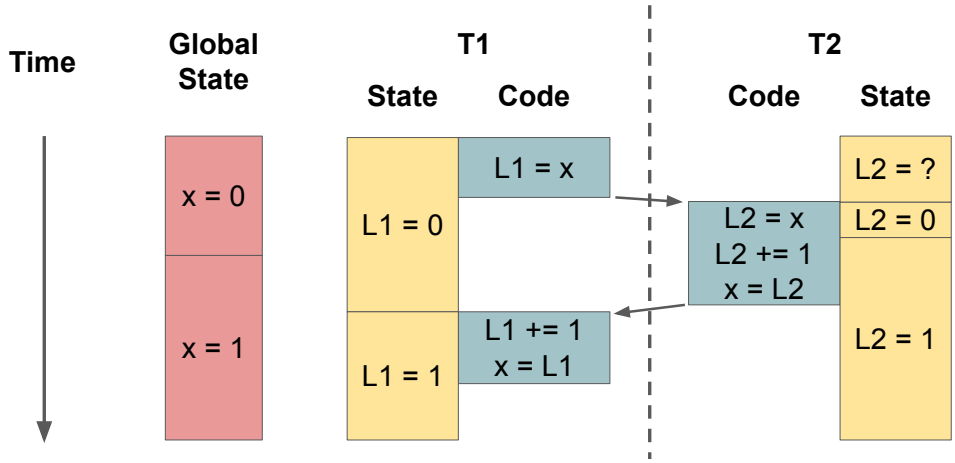


Figure 2.2: Read-modify-write race

2.4 Concurrency Bugs

In this section we describe three types of concurrency bugs, which are bugs that specifically occur due to the properties of concurrent programs.

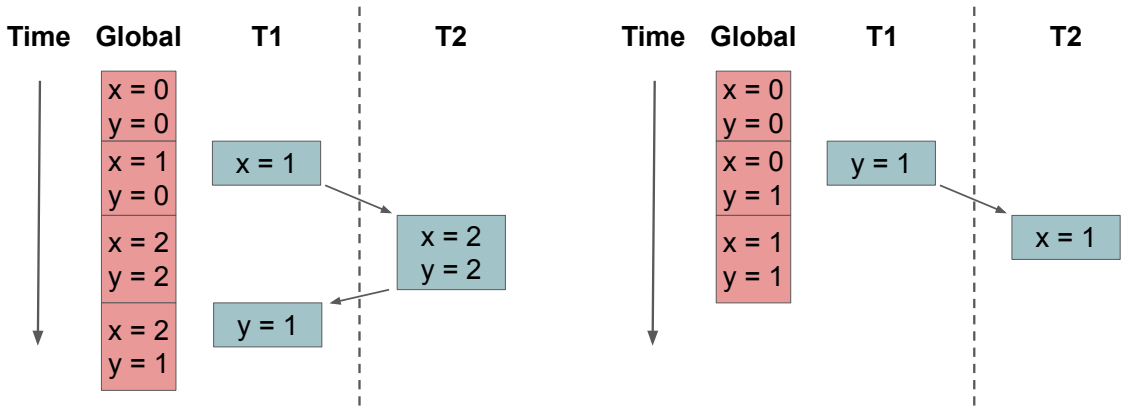
2.4.1 Read-modify-write Race

Races occur when there are two concurrent accesses to a shared variable and at least one of those accesses is a write.

In Figure 2.2 we can see an example of a race in action. The variable `x` is a shared variable, initialized to 0, `L1` and `L2` are the corresponding local variables for each thread. We can see in the example that `T1` stores the shared value in a local variable then its execution is halted (this can happen arbitrarily) and `T2` resumes reading, incrementing and storing the new value. At this point the global variable has the new value (1), after that `T1` uses its old local copy of the shared value to calculate the increment and store it. In this example two threads try to increment a variable initialized to 0 resulting in 1. We would expect the resulting outcome to be 2, however that may not always be the case as expressed by the example above. To fix this issue, a developer can use a mutex to ensure mutual exclusivity for the entire increment operation.

2.4.2 Semantic Violations

Atomicity violations occur when memory regions that should be updated atomically (with respect to other updates and reads) are not. In Figure 2.3a we can observe this violation. The variables `x` and `y` are shared variables that should be modified atomically. In this application each thread is tasked with writing a different value (1 and 2) to `x` and `y`, the only acceptable states by the end of the execution are either both variables with a value of 1 or both variables



(a) Atomicity Violation

(b) Ordering Violation

Figure 2.3: Concurrent semantic violations

with a value of 2 (depending on which thread executed last). In this example we can see that this might not always happen. `T1` starts by writing to `x` and immediately gets swapped with `T2` that updates both variables, then `T1` finishes its second write operation by writing to `y`. By the end of the execution each variable has a different value, which is an indication that atomicity was broken. One possible way to fix this violation is to employ a mutex around both assignments.

Ordering violations occur when operations happen out of order. It is easy to imagine how that can occur in a multithreaded environment if the proper synchronization is not implemented. Figure 2.3b provides a simple example of such a violation. Assume that it describes an application where the shared variable `x` must be updated before `y`. In this example, due to the lack of synchronization, that requirement is not met, and therefore we have an ordering violation. One possible way to fix this violation is to employ a condition variable between both assignments.

2.5 Concurrency Bug Detection Techniques

In this section we outline the rationale behind three approaches to concurrency bug detection [24]. These constitute the techniques we considered while building our proposal.

2.5.1 Delay Injection

This family of approaches attempts to force concurrent bugs by injecting delays in memory accesses. They can fall between two extremes: totally random where the duration and location of the delays is randomized, or informed by a multitude of heuristics and other techniques. RaceFuzzer [31], for example, requires several executions to be able to provide good coverage since its efficacy is dependent on the interleaving observed in any given run. TSVD [32] focuses

on large scale testing, and is able to achieve a relatively low overhead. However, to accomplish this, TSVD is specifically designed to find bugs within the first few runs, sacrificing coverage. Overall, delay injection increases the probability that a concurrent bug is observed, but still requires multiple executions if we want to achieve proper coverage of interleavings.

2.5.2 Happens-Before Relation

The happens-before relation was first outlined in [33] as a method to determine a partial order between events in distributed systems. Each event is ordered based on a logical clock, which is incremented based on messages, specific events that establish an order between processes. Using the happens-before relation, shared memory accesses can be analyzed in order to determine whether they can occur concurrently and subsequently reported. Several works have improved the technique over time. To the best of our knowledge, it was first described in [34] and further improved upon in [35, 36].

2.5.3 Lockset Analysis

According to our research, lockset analysis was first introduced in Eraser [37], which attempts to uncover concurrent accesses to shared memory that are not correctly protected by locks. This approach assumes all accesses to the same memory region must be consistently protected by the same lock, and if that is not observed then a race can occur. Each access registered by Eraser further refines the lockset. To compute this, the locksets of each access to the same variable are intersected, the result constitutes in the set of locks that protect a given variable. If at any point of the execution the variable is not consistently protected by at least one lock, it issues a report. This approach is susceptible to two type of false positive: when variables are correctly initialized without a lock, because their address has not yet been made public to other threads, or when a variable is read-only after initialization, therefore does not necessitate protection. To accommodate for these programming practices Eraser only registers accesses made to a variable after it is published, which is when it is visible to other threads. However, there is no trivial way to know when that occurs. To approximate this decision, a heuristic is used, stating that a variable is published when a second thread performs a write on it. Eraser also reports the use of annotations to remove specific false positives caused by benign races, private locking mechanisms, and memory reuse. Lockset analysis is often paired with the aforementioned happens-before relationship [36]. Overall lockset analysis provides a way to detect concurrent accesses to shared variables without the need to observe the specific execution where they occur. This removes the need to run the analysis several times in the hope of exploring different interleavings.

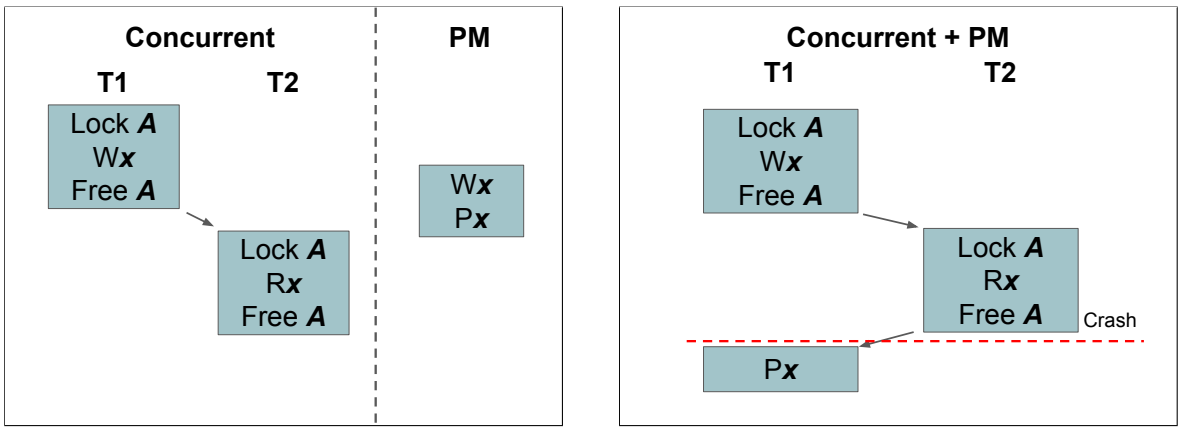


Figure 2.4: Persistent Race

2.6 Concurrent PM Bugs

Typical concurrent bugs involve the incorrect synchronization of accesses to shared memory regions, to detect them, we must take into account the possible thread interleavings or synchronization primitives used to protect them. PM bugs are caused by the incorrect use of low-level instructions, or the lack of recovery mechanisms, and to detect them, we must reason about the semantics of PM. When using both concurrency and PM, novel bugs are introduced, concurrent PM bugs, and, to detect them, both matters must be taken into account. In this section we describe a taxonomy consisting of three classes of concurrent PM bugs, including how they correlate to concurrent bugs as well as PM bugs, and what programming errors can cause then.

2.6.1 Persistent Race

We consider that the persistent race is a logical consequence of introducing PM in concurrent applications, or vice-versa. To understand its origin we start by analyzing Figure 2.4a, which represents two different programs. On the left there is a concurrent program that correctly accesses a shared variable `x`, using locking primitives on the same lock `A`. On the right lies a simple PM application that correctly writes and persists variable `x`. Both these applications are correct within their respective contexts, however, when combining both PM and concurrency on a singular program the developer may make a mistake resulting in the implementation in Figure 2.4b. In this application, the accesses to the shared variable `x` is protected by the same lock `A`, however, the persistency is not. If the application crashes after the read operation in `T2` creates a side effect, then that side effect will be visible in the post-crash state but the original store it depended on will not. This constitutes a bug. In this relatively simple example, the bug can be fixed by putting the persistency operation inside the section protected by the lock `A`.

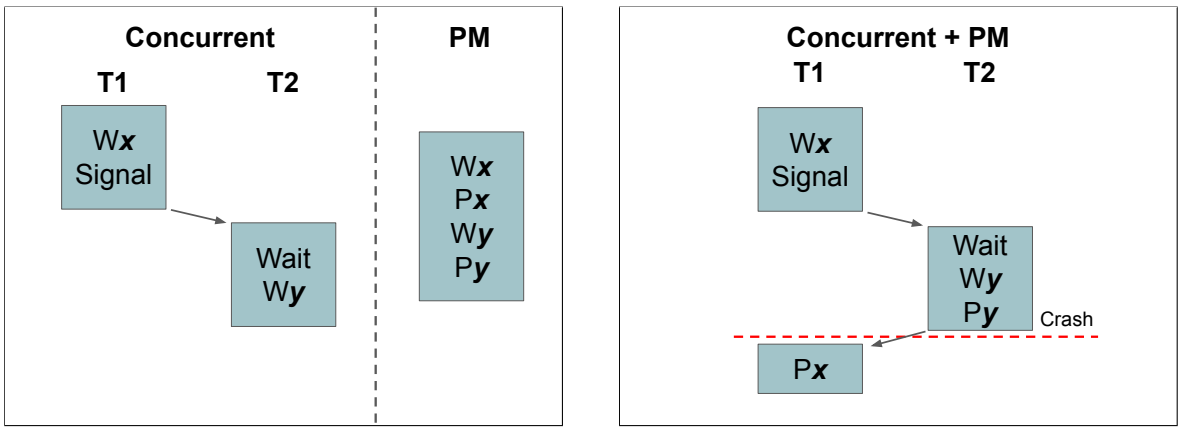


Figure 2.5: Persistent Ordering Violation

2.6.2 Persistent Ordering Race

The persistent ordering race occurs when, due to the concurrent nature of an application, the order in which data is persisted is not guaranteed. Consider the example applications described in Figure 2.5a. On the left there is a concurrent application that correctly enforces ordering between writes using a signal/wait synchronization primitive. On the right we describe a PM application that correctly employs a strict persistency ordering between writes. Both these applications are correct within their domains, however, when combining both versions in a single program, the developer may erroneously produce the implementation in Figure 2.5b. In this example, despite the write operations being correctly ordered by the signal/wait primitives, the persistency is not. If the application crashes after the second write, the post-crash may observe the second write but not the first, breaking that ordering requirement. In this relatively simple example, the bug can be fixed by putting the persistency operation before the signal.

2.6.3 Persistent Atomicity Race

The third concurrent PM bug is the persistent atomicity race. Figure 2.6a represents two distinct applications. On the left there is a concurrent application that atomically updates a set of variables `x` and `y`. On the right there is a PM application that uses a commit variable `C` to ensure the set of variables is update atomically in case of a crash. Both these applications are correct, however, merging them can result in the program in Figure 2.6b. Despite the use of correct synchronization primitives to ensure thread-atomicity, including the persistency of each write, and the use of commit variables to ensure failure-atomicity, this application can still result in an error if it crashes after thread `T2` persists some side effects from its read operation. After the crash, the atomic update is rolled back, since the operation was not marked as complete

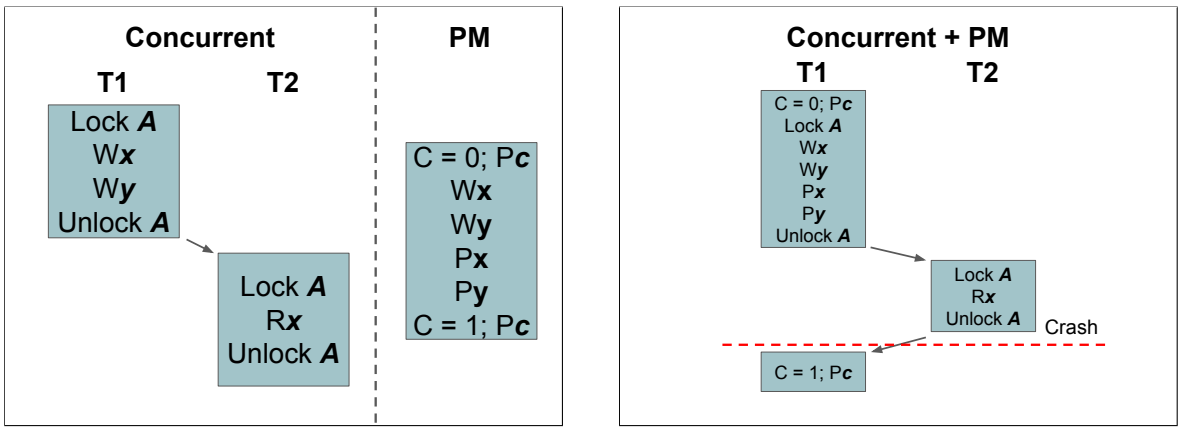


Figure 2.6: Persistent Atomicity Violation

with the commit variable, however the side effect is still visible. This constitutes a bug. In this relatively simple example, the bug can be fixed by ensuring the commit variable updates are included inside the critical section.

2.7 Summary

To summarize, the combination of PM and concurrency leads to a new set of bugs that requires specific tools to be detected. In the next chapter we discuss the state-of-the-art of PM bug detection tools.

Chapter 3

Related Work

In this chapter we lay out our analysis of the current state-of-the-art for PM bug detection. In Section 3.1 we report the state-of-the-art in PM bug detection tools. In Section 3.2 we further analyze the tools that focus on concurrent PM bug detection. Finally, in Section 3.3, we provide an overview of the state-of-the-art and identify the areas in which we can make our contribution.

3.1 Persistent Memory Bug Detection Tools

Over the course of this section we describe how each tool functions, and characterize them in terms of being automatic and application agnostic. We define automation as a subjective measure of the work required to debug a particular PM application using the tool, and define an application agnostic tool as being able to analyze any type of application. Our interest in these properties will become evident later.

PM bug detection tools need to decide whether a particular application has a bug, however, this is not a trivial decision due to the existence of semantic bugs. It is straightforward to decide whether a missing flush/fence constitutes a bug, but it is a different task to decide whether any two variables must be persisted in a specific order or atomically. To solve this issue, tools approximate this decision using oracles, as a way to decide failure. We find that the oracle used is heavily correlated with how automatic and application agnostic a tool is (although not an exact match), and based on this observation, decided to group tools depending on the oracle used.

3.1.1 Recovery/Verification Program

The use of a verification program is one of the first crash state validating techniques presented in the literature.

The first tool proposed for PM bug detection is Yat [14], which divides its analysis into multiple stages: running a workload while generating a trace of memory primitives (store, flush, and fence) related to persistent memory, computing the set of all states based on the fact that PM operations can be delayed and reordered, and then, for each state, executing a file system checker (verification program) reporting a bug in case of a failure. As previously mentioned, Yat eagerly enumerates all states, whose number grows exponentially with the number of stores. According to the evaluation presented in the original paper, the analysis of a workload of 54k stores, 14k flushes and 6k fences would take a theoretical 5.2 years of runtime to complete [14]. To overcome this challenge, Yat proposes a threshold, that limits the number of states explored for each segment of PM instructions between `fences`. With an arbitrarily chosen threshold of 250 combinations per segment it takes 3 days to execute the previous test, reducing the total number of states from 789M to 1M. This threshold does not take into consideration any information about the program, arbitrarily pruning the amount of states in the search space. Although a pioneer of PM bug detection, Yat makes one thing clear, a complete exploration of all possible crash-states is not a feasible solution for debugging PM programs.

Jaaru [15] improves on this matter by exploring possible states in a lazy fashion. It uses model checking on top of a persistent memory simulation to realistically infer the possible states without repetition, considerably pruning the search space. In the best case scenario, the number of possible crash states grows quadratically with the number of stores, instead of exponentially like in Yat. Jaaru achieves this by using constraint-refinement, a technique which is very effective in programs that use commit variables (a common PM technique according to the authors of the paper, however it is not backed by a formal study). Constraint refinement only enumerates crash states if a value is read on the post-crash, unlike Yat where even if a variable is never read on the post-crash it still enumerates all the possible combinations stores to that address. Jaaru is still exhaustive in its testing, all crash-states pruned by constraint refinement are either equivalent to another state that is tested or impossible to be observed in a real execution. The improvement brought by Jaaru constitutes a significant shift in performance of systematic crash state validation, however its model checking approach does not support certain components commonly used in applications, such as sockets, without considerable modification.

Yashme [16] builds on Jaaru by introducing a concept they call `persistency races`, a particular type of atomicity violation caused by compiler optimizations. Store tearing (when a store operation is divided into multiple stores) and store inventing (when an extra store instruction is added to an operation) are techniques implemented by the compiler that, coupled with PM semantics, have the potential to introduce bugs in the application. For example, the

compiler may divide the store of an eight byte value, such as a pointer, into two separate store instructions, which means that under specific circumstances (where only one of the stores is persisted right before the application crashes) the post-crash state will be inconsistent, resulting in a failure. It is worth noting, that a program like Yat that does not employ model checking would be able to catch this type of bug, since it instruments each store separately and should reorder/delay them to reveal an inconsistent state.

Mumak [38] focuses on achieving scalability, while also maintaining an automatic and application agnostic approach. To accomplish this, Mumak foregoes testing all possible post-crash states, opting instead to evaluate the memory operations in program order. This would obviously result in missed bugs, however, the authors report that this approach detects 90% of the bugs detected by the previous tools in the state-of-the-art. Mumak divides its approach in two distinct phases: fault injection and trace analysis. The fault injection stage injects failures in PM operations and runs the recovery program, if it fails then a bug is reported. The second stage, reports several types of PM bugs by analyzing the trace and applying tailor-made patterns. Most importantly, it attempts to compensate for the fact that the previous stage does not test all possible post-crash states. To do this, it reports any fences that act on more than one flush. Such occurrences imply that the data is persisted in a non-deterministic order, which can lead to a PM bug. In short, Mumak's ability to scale is much greater than previous tools that rely on the recovery procedure, however, it achieves this by sacrificing completeness. Despite its black-box approach, Mumak performs a significant optimization to the instrumentation that requires the application to be deterministic.

Yat provides PM bug detection for file systems, which means it is not application agnostic, while Jaaru, Yashme, and Mumak support most applications. In practice, Jaaru, and Yashme are prototypes that make several implementation decisions which hinder their automation and application agnosticism. Namely, their model checking does not support certain commonly used features, such as sockets. However, Mumak makes application agnosticism a priority, and puts considerable weight on its black-box approach. Yat, Jaaru, Yashme, and Mumak are fairly automatic, because they use the verification/recovery program as an oracle, which most (if not all) PM applications already have, requiring no extra artifacts for the purpose of debugging. Furthermore, despite all efforts made to achieve scalability, Yat, Jaaru, and Yashme simply do not scale as desired. Mumak is able to scale for large workloads, however it only considers the states in program-order, which leads to missing bugs.

PMRace [23] is a bug detector for concurrent PM applications. It uses the recovery procedure as its oracle, we will discuss it further in Section 3.2.

3.1.2 Operation's Semantic Information

Another type of oracle employed by PM bug detection tools is the semantic information of the operations, namely, these tools make assumptions regarding on how the application under test functions.

Witcher [17] is one such tool, it employs likely-correctness conditions, which are heuristics that are used to find points in the code where if a failure would happen, the chance of it resulting in an invalid state would be higher. These conditions were created based on commonly used PM programming patterns, and inferred from a dynamically created graph which contains data and control dependencies between instructions. Witcher focuses on testing durable linearizability (preserves linearizability even in the face of arbitrary crashes) at the operation level. To achieve this, it needs semantic information about each operation in the form of a driver, and requires the application to have an API equivalent to that of key-value stores (insertion, removal, and reading). Witcher uses output equivalence checking to test a particular failure point. It runs the application twice without a failure, once where the operation completes fully, and another where the operation is skipped, registering both outputs. It then runs the application again, while injecting a failure in the operation (according to the heuristics mentioned above), and if the output differs from both of the previously mentioned outputs it means that the application did not follow all-or-nothing semantics and does not satisfy durable linearizability. To be able to successfully compare the output of different executions Witcher requires a deterministic application (or at least an application whose output is deterministic).

Durinn [22] builds on top of Witcher with concurrency bug support, we will discuss it further in §3.2.

Vinter [39] is a full system PM debugger that leverages full virtualization with qemu [40] to debug kernel code by applying instrumentation at the emulation layer. Vinter requires a specialized program that extracts the semantic information from the post-crash state. Since the applications evaluated are file systems, the extractor is a file system serializer. Vinter uses hypercalls to distinguish between different operations, these calls must be manually inserted by the developer in the workload between operations. Vinter proposes two types of crash-consistency guarantees: Single Final State (SFS), where an operation may have several in-between states, but after returning it must have only one state, and Atomicity, where an operation can only have one possible state. All operations that satisfy Atomicity also satisfy SFS by definition. Vinter reports an operation's failure to adhere to either guarantee, and the user must manually decide which reports are useful, resulting in a very large number of potential bugs reported and increasing the burden on the tool's users. Overall Vinter's biggest strength is the support for

kernel-side code, its claims of testing full systems fall short in practice since full systems tend to be more complex than file systems and their operations might not be trivially mapped to either of the proposed guarantees.

Witcher, Durinn, and Vinter make heavy assumptions about the application under test, and require extra artifacts to be created specifically for the purpose of the debugging process. Namely, Witcher and Durinn support key-value stores or equivalent applications and require complex drivers that map the applications operations to each tool, while Vinter supports file-systems whose operations must follow specific guarantees and requires a program that extracts the semantic information of a post-crash state.

3.1.3 Annotations

Annotation based oracles require the programmer to define the intended semantics in the program itself.

PMTest [18] uses assert-like annotations in the code to transmit ordering and atomicity information. There are two basic annotations (low-level checkers), `isPersist`, that checks whether a particular store is persisted, and `isOrderedBefore`, that checks whether write A is persisted before write B. PMTest computes the flushing and persistence intervals at any given point using PM semantics, and when the checking engine finds an annotation it tests it and, if it is not true, reports a bug. These low-level checkers are enough to ensure correct ordering and atomicity between writes, however they can be very repetitive to write and error-prone. With this in mind, the authors propose a series of high-level checkers, that are automatically inserted in PMDK transactions and check whether variables have been correctly persisted. The use of annotations in PMTest requires deep knowledge of the application's semantics as well as PM semantics, incurring a great cognitive load on the programmer. On top of that, the misuse of annotations can lead to false positives, as well as false negatives.

XFDetector [20] leverages the commit variable PM programming pattern to infer the consistency of memory regions. The programmer registers commit variables, and what memory regions are controlled by each variable, then XFDetector keeps track of each memory region's consistency and persistency state. The persistency state is obtained via tracing, by instrumenting store, flush and fence operations, and the consistency state is obtained from updates to the registered commit variable for that memory region. When a read happens on the post-crash execution and the location read is not persisted or not consistent XFDetector reports bugs. Commit variables are meant to be read whether they were successfully persisted or not. To avoid false positives, XFDetector ignores these accesses when reporting bugs. XFDetector also allows the

annotation of start and ending of regions of interest, which are the slices of execution that get analyzed. The annotation based approach leads to a similar problem as PMTest, although the burden can be considered smaller, since only commit variables need to be annotated. However, it can still lead to misuse of annotations.

PMDebugger [19] implements a novel way to organize the bookkeeping metadata based on a study of PM programs that yielded the following patterns: (1) most stores are persisted by the nearest fence, (2) multiple writes to a particular cache line are persisted by a single flush instruction, and (3) store instructions are more frequent than flush or fence. With these patterns in mind, PMDebugger uses two data structures: an array, for the most recent stores, that capitalizes on the first pattern due to its fast insertion/removal, and an AVL-tree for longer lived stores, that takes longer to insert/remove, but it is amortized in faster searching. The second pattern is used to guide the structure of the metadata itself, persistency information can and should be maintained per cache line. The third pattern also justifies the array for storing the most recent store operations since insertion is faster and store instruction constitute an insertion. Unlike PMTest or XFDetector that use inline annotations, PMDebugger's paper describes a configuration file, external to the application used to define ordering constraints. We believe this detachment between annotations and code can be counterproductive and lead to more mismatches between annotation and implementation due to forcing the developer to maintain two different locations while changing the code. In reality, PMDebugger's implementation uses inline annotations for ordering information.

Overall the disadvantage of annotations is not just error proneness, it is also the effort required of the programmer, their use requires that developers spend a considerable amount of time creating specific artifacts for debugging, unlike some previous examples that use the recovery/validation program, where any time spent creating a recovery program contributes towards debugging and production. It is clear that annotations hinder the level of automation provided by these tools, however, when it comes to application agnosticism, the approaches of PMTest, XFDetector, and PMDebugger should provide support to any application, although the high level checkers in PMTest only support PMDK transactions.

3.1.4 Persistency Model

The tools discussed in this section take advantage of clear-cut persistency models to guide their debugging process. There are three proposed models for persistency: strict, epoch, and strand. The strict persistency model dictates all stores are sequential and synchronous and must be followed by a flush and a fence. The epoch persistency model slightly relaxes these restrictions.

It divides the execution in epochs and all stores within an epoch can be reordered with each other, however, stores in different epochs cannot be reordered with respect to each other. The last persistency model is strand persistence. A strand is an interval of execution (similar to an epoch), where stores can be reordered with respect to other strands but not the same strand.

DeepMC [41] uses static analysis to determine whether an application follows the intended persistency model. It generates a data structure graph (DSG) using data structure analysis, and then it uses the DSG together with a trace to ensure the program follows the persistency model.

Another tool that has been mentioned above, PMDebugger, uses the persistency model to test for bugs, particularly the ones that do not involve ordering or atomicity.

Both of these tools have one major drawback, most applications, including simple synthetic benchmarks, follow a mixed persistency model, neither fully strict, epoch, nor strand persistency model, which leads to a lot of false positives. Similarly to Yashme and Jaaru, DeepMC's approach is automatic and application agnostic on paper, however, in practice, the static analysis implementation may not support all features of an application, but it should be extendable.

3.1.5 Symbolic Oracles

Agamotto [21] builds the symbolic state of persistent memory variables by driving the execution towards regions of code with higher number of PM-modifying instructions, which are therefore more likely to contain PM bugs. Agamotto supplies symbolic oracles for trivially identifiable PM bugs. These oracles receive the symbolic state at key points of the execution (flush, fence and end of the execution), and decide whether it constitutes a bug. However, it requires the creation of custom oracles for semantic bug detection, pushing that burden to the developers who need to reason about their program's semantics in the context of persistent memory and a symbolic state, which is not a trivial task. Symbolic execution is oblivious to the implementation details of a compiled program, meaning that any bug caused by compiler optimizations or architecture cannot be checked by the oracles in the current state of the tool. Agamotto is not automatic since it requires the creation of custom oracles that check for semantic bugs in the symbolic state, nor application agnostic since these oracles cannot be reused between applications. On top of that, commonly used programming constructs (i.e: sockets) may not be supported by the underlying symbolic execution engine, requiring modification.

3.2 Concurrent PM Bug Detection

We now characterize the state-of-the-art in terms of its ability to deal with concurrent programs and discover concurrent PM bugs. We define a concurrent PM bug as a PM bug that manifests due to concurrency and divide the state-of-the-art in a tool's ability to search for such bugs, passively and actively.

Agamotto uses a symbolic execution engine which is known to not handle concurrency [42], for this reason, it is left out of this discussion.

3.2.1 Passive Search

Tools that do not explore different thread interleavings but support concurrent programs can, by chance, discover concurrent PM bugs. Yat, Jaaru, Yashme, Mumak, Witcher, PMTest, PMDebugger, and XFDetector's techniques are, in principle, thread safe, therefore if a particular execution sees a race between 2 threads which causes a PM bug these tools should be able to detect it. However, these tools do not make an active effort to search for these bugs, and would not be able to distinguish them from regular PM bugs.

3.2.2 Active Search

Tools that actively explore interleavings are better equipped to detect concurrency PM bugs.

PMRace [23] detects what it calls PM inter-thread inconsistencies, when a thread persists side effects based on non-persisted data from another thread. If the program crashes after this, there will be persisted data that depends on data that was lost. These side effects are therefore considered tainted and, if not overwritten by the recovery, constitute a bug in the application. PMRace defines two ways in which these side effects may occur, the persisted data was read from a non-persisted write, or, the target address of the persisted data was read from a non-persisted write. This taint analysis approach is not foolproof and can fall short under specific circumstances: First, even overwritten data may still be inconsistent if the new value depends on another tainted value, leading to a false negative (i.e.: the recovery program increments a tainted memory region). Second, there can be regions marked as tainted by the end of the recovery that do not actually constitute a bug (i.e.: a recovery program that works in a lazy fashion). PMRace attempts to force persistent side effects based on unpersisted data, essentially, it needs to observe a store and a read to the same address executing concurrently. To achieve this, it injects waits between writes and reads, these waits are properly engineered to avoid deadlocks, and reduce wasted time, but still constitute a noticeable slowdown. These waits are targeted to shared PM variables with a focus on the most accessed addresses. The authors introduce

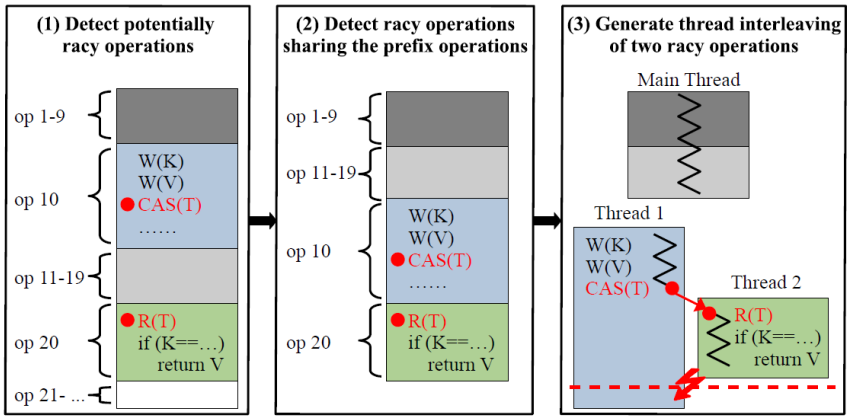


Figure 3.1: Detection of concurrent PM bugs with Durinn (Source: [22])

fuzzing to generate workloads in order to increase coverage. PMRace's approach to fuzzing encompasses a series of strategies that target the workload based on a seed, performing mutations to operation's parameters, adding or deleting operations, and shuffling the operations. These strategies prioritize similar keys in order to increase the likelihood of concurrent accesses. If the strategies fail to increase coverage, PMRace populates the workload with insertion operations in order to trigger resizing mechanisms that usually involve a lot of PM operations that can potentially contain bugs. Finally, PMRace also swaps out seeds if the above strategies are not fruitful. The approach adopted by PMRace for concurrent PM detection should, in theory, be fairly automatic and application agnostic. The fuzzing engine is made with key-value store (or equivalent) applications in mind and requires non-trivial artifacts be created specifically for the debugging process. The authors claim that coverage is an orthogonal problem, however, it is not clear if the approach would even be feasible without the use of the fuzzing engine. Therefore, we believe that PMRace is not automatic nor application agnostic.

Durinn describes linearization points (LP) as points in the program where data becomes visible to other threads and durability points (DP) as points in the program where data is persisted. Concurrent PM bugs occur when the crash happens between these points. For example, let's say thread A performs a store that is visible to other threads (after LP), while thread B reads and returns that new value and the application crashes before thread A persists the store (before DP). After recovery, reading from the memory region may yield the old value, breaking linearizability. DPs are flush/fence instructions, therefore, are easy to find and instrument. To infer LPs Durinn exploits 3 common programming practices: (1) Atomic instructions, used to perform atomic operations, (2) guarded-protection, which is the use of guard variables to ensure that non-atomic operations are only visible after they are completed, and (3) publish-after-initialization, used for atomic allocation and initialization, where the pointer to a newly allocated memory region only becomes visible to other threads after the initializa-

tion is complete. Atomic instructions are trivially inferred from the binary and guard variables are inferred based on read patterns in conditional branch instructions, which are used to find likely LPs for further analysis. Publish-after-initialization allows Durinn to ignore allocation and initialization as LPs, pruning the total number of LPs without missing possible bugs. After inferring likely-LPs and DPs, Durinn takes a 3-step approach described in Figure 3.1. First, it generates a synchronous trace of the application under test and finds pairs of potentially racy operations, that write-write or write-read on a synchronization variable (from likely-LPs). Each pair of operations is not necessarily racy when ran in parallel, that is because they might not have the same prefix of operations. To mitigate this, the second step of Durinn's bug finding strategy is to rerun the application, while reordering the operations to make sure they both have the same prefix. If the operations remain racy, then it is time for the third and final step of Durinn's strategy, it executes both operations concurrently (with the required prefixes), forcing a concurrent execution of the racy instructions and crashing the application before the DP is reached. Durinn achieves this by controlling the interleaving of both threads using breakpoints, setting them on the racy operations. First thread A runs until it reaches the breakpoint, second thread B runs until it reaches its breakpoint and then continues until the end of the operation where the application crashes. After recovery, Durinn executes application specific validation operations that check whether the state is consistent and reports a bug if it is not. Just like Witcher, Durinn makes assumptions about the application under test and requires a complex driver and application specific validation operations which heavily reduce its automation and application agnosticism.

3.3 Discussion

The state-of-the-art of persistent memory debugging provides us with several oracles with differing advantages and drawbacks. Annotations increase the programmer's burden since they require artifacts that can only be used for testing but generally require a single execution of the application. Symbolic oracles suffer from similar problems as annotations requiring users to be able to develop custom oracles. The persistency model provides a great degree of confidence in the debugging if and only if the applications strictly follow a single model, which is usually not the case. Semantic information of operations require a design heavily dependent on the targeted application (key-value stores or file systems) and the creation of a complex artifact for debugging. Recovery/validation programs are able to automatically enumerate and test a large amount of states which can take a lot of time, however, the artifact required for the debugging process is useful for production.

Tool	Oracle	Automation	Application Agnostic	Supports Concurrency	Actively Searches for Concurrent PM Bugs
Yat [14]	RVP	High	No	Yes	No
Jaaru [15]	RVP	High	No*	Yes	No
Yashme [16]	RVP	High	No*	Yes	No
Mumak [16]	RVP	High	Yes‡	Yes	No
Witcher [17]	OSI	Medium	No	Yes	No
Durinn [22]	OSI	Medium	No	Yes	Yes
Vinter [39]	OSI	Medium	No	Yes	No
PMTest [18]	A	Low	Yes	Yes	No
XFDetector [20]	A	Low	Yes	Yes	No
PMDebugger [19]	A / PMod	Low	Yes	Yes	No
DeepMC [41]	PMod	High	Yes	Yes	No
PMRace [23]	RVP	Medium†	No†	Yes	Yes
Agamotto [21]	SO	Low	No*	No	No
HawkSet		High	Yes	Yes	Yes

Table 3.1: Summary of key characteristics for each PM debugging tool in the state-of-the-art. We use the following notation: **RVP**: Recovery/Validation Program **OSI**: Operation's Semantic Information **A**: Annotations **PMod**: Persistency Model **SO**: Symbolic Oracles *****: Not application agnostic due to implementation details †: Heavily optimized for key value stores ‡: Requires deterministic execution

On the concurrency side of the state-of-the-art there is much less variety. PMRace explores multiple executions while fuzzing the inputs in the hopes of increasing its coverage of useful interleavings. Durinn actively tries to create scenarios where the initially flagged interleavings are executed.

Table 3.1 summarizes the key aspects of each tools in the state-of-the-art for bug detection in PM applications. We categorize these tools in terms of the degree of automation, and on whether they are application application agnostic. Finally, we characterize the tools in their ability to analyze concurrent applications and whether they actively search for concurrency PM bugs.

The level of automation is given based on the steps required to set up the tool to debug a particular application. We can see that the type of oracle used, roughly correlates with the degree of automation of the tool. Tools that rely on the recovery procedure are highly automatic since that procedure already exists, while tools that rely on annotations or symbolic oracles require the creation of extra artifacts for debugging.

Application agnostic tools are the ones that can effectively debug any PM application. Jaaru, Yashme and Agamotto do not support any arbitrary application because their implementation relies on techniques like model checking and symbolic execution. Durinn, Vinter and Witcher do not support arbitrary applications because their approach is made with a particular type of

application in mind.

PMRace is application agnostic, however, for performance reasons, the fuzzing algorithm is optimized for key-value stores and without this optimization, the tool might not be usable in a production environment. Also, due this optimization, PMRace requires that a driver be provided, meaning it is not highly automatic.

When it comes to actively searching for concurrent PM bugs, there is a significant drop in variety compared to PM bug detection in general. Only two tools in the state-of-the-art analyzed for this document satisfy this condition. However, none of these tools guarantee an automatic and application agnostic approach.

In the following section we present HawkSet, an automatic and application agnostic tool, that actively searches for concurrent PM bugs in a useful amount of time.

Chapter 4

Approach

Developing PM applications is a difficult endeavor. In order to correctly prevent an invalid post-crash state, low level instructions and various abstractions can be employed, accompanied by a recovery program that detects and corrects invalid states, such as corrupted or missing data. Adding to this problem, concurrent applications that use PM, also require the synchronization of accesses to shared PM regions. Finding bugs in these applications is a complex challenge even for experienced engineers. They must reason about the impact of an arbitrary crash at any point of the execution, how the stalling and reordering of memory operations can affect the post-crash state, and how the different shared memory accesses may impact the correctness of the system. To alleviate this pressure, we propose HawkSet, an automatic and application agnostic tool for systematic concurrent PM bug detection.

4.1 Revisiting The Problem

In Section 2.6 we outline three distinct concurrent PM bugs: i) The persistent race, which consists of two racing accesses to unpersisted data; ii) the persistent ordering race, which occurs when persistency ordering guarantees are not respected due to faulty orchestration of threads; and iii) the persistent atomicity race, which comprises a racy PM access to persisted data protected by a commit variable that is then rolled back after a crash. Our solution focuses on persistent races, and we leave the detection of other concurrent PM bugs as future work.

After the careful analysis of the state-of-the-art in Chapter 3, we adopted additional constraints into our approach. Namely, our tool must be automatic, application agnostic and efficient. This is a complex challenge because these goals can, at times, be diametrically opposed. On one hand, an application agnostic tool must make no assumptions about the application under test, and an automatic tool cannot request any additional information or code artifacts from

the user. On the other hand, making assumptions about the application and requesting additional artifacts from the user can allow for specific optimizations. For example, some annotation based tools, request information about the persistent state of the application and therefore don not need to compute all possible post crash states, allowing for much faster testing.

4.2 HawkSet

HawkSet is our proposal for an automatic, application agnostic and efficient tool for concurrent PM bug detection. It leverages a lockset analysis technique, tailored for PM applications, to detect persistent races, and applies a heuristic capable of ignoring common false positives often found with lockset analysis. Furthermore, application agnosticism and automation are key aspects of our implementation. We do not require any annotations, drivers, or similar artifacts for debugging and support any application in binary form.

Lockset analysis allows the detection of races without actually observing them in real-time, since we can instead infer where their occurrence is possible, assuming the workload provided has sufficient coverage. This eliminates the need to execute the application several times in order to explore all possible interleavings, which drastically reduces execution time. Lockset analysis is known to scale poorly when dealing with large amounts of memory accesses, however, the key insight that allows for its use in PM applications, is that the ratio of memory accesses to PM is very small, 4% [25]. Thus allowing an otherwise expensive technique to scale well in PM programs.

4.3 Architecture

To detect the previously outlined concurrent PM races, we want to be able to report pairs of PM accesses, that are not correctly synchronized by a lock. To do this, we need to monitor synchronization primitives and PM accesses, apply our heuristic, and finally use lockset analysis to detect persistent races.

HawkSet is logically divided into three stages, represented in Figure 4.1. The first stage, which we collectively call instrumentation ①, is responsible for collecting information about the execution. The second stage applies the initialization removal heuristic ② to detect race likely points (RLPs). Finally, the analysis stage ③ applies our lockset analysis algorithm aided by a happens-before analysis.

Although these stages are logically distinct, in practice, the heuristic is applied during instrumentation.

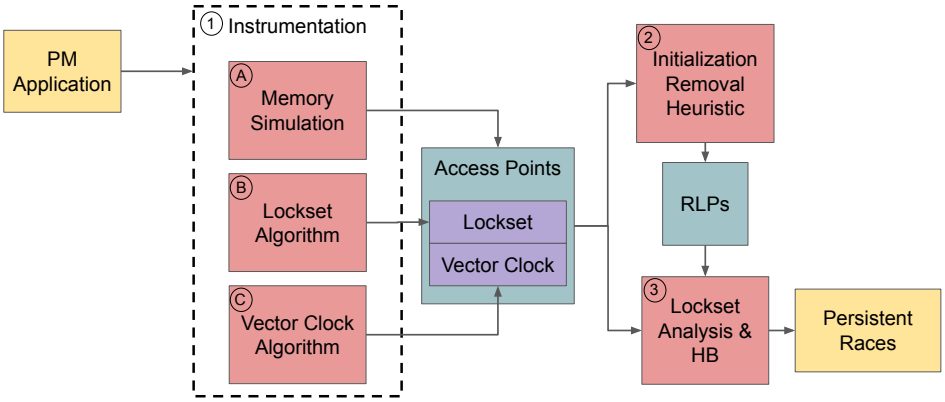


Figure 4.1: HawkSet's pipeline

4.3.1 1st Stage: Instrumentation

The first stage of HawkSet has three different responsibilities. The **memory simulation** Ⓐ determines in which points of the execution data is persisted and registers important events. To this end, we maintain the state of the cache, simulating the execution of write, flush, and fence instructions. In the case of HawkSet, we are particularly interested in knowing the moment when data is guaranteed to be persisted. To achieve this, we simulate a worst-case cache that only persists a cache-line if a flush is issued and where all stores are delayed until a fence (or equivalent instruction) is executed. PM accesses are stored alongside the current lockset and vector clock given by the lockset Ⓑ and vector-clock Ⓒ algorithms (described below).

4.3.2 2nd Stage: Heuristic

Before HawkSet analyzes each PM access for potential races, it is important to select the ones that are likely to reveal races (RLPs), as we do not want to report false positives or spend time analyzing accesses that cannot result in a race.

The first intuition would be to compare every PM access, flagging those that are not correctly synchronized by a common lock. However, in concurrent applications, it is correct, and generally more performant, to initialize freshly allocated variables without holding a lock, since the memory region has not yet been made public to other threads. To avoid the incorrect report of these accesses as a persistent race, we extend the heuristic provided in Eraser [37] to PM applications. We call this the initialization removal heuristic. It aims to ignore memory accesses to memory regions that have not yet been made public. For PM applications we must extend it to take persistency into account. For example, if a thread allocates and initializes a PM region, publishes the pointer, and only then persists the data, the original heuristic will discard those

stores. However, if a second thread comes along and accesses that memory region, the original store has been discarded and therefore no persistent race is reported. To fix this problem, we must extend the definition to include persistency, ignoring all stores whose persistency occurs before the variable is public.

4.3.3 3rd Stage: Lockset Analysis & Happens-Before

In this section we will incrementally build our solution, describing how a traditional lockset analysis algorithm can be extended in order to detect concurrent PM races, starting with a possible lockset analysis outlined in Algorithm 4.1.

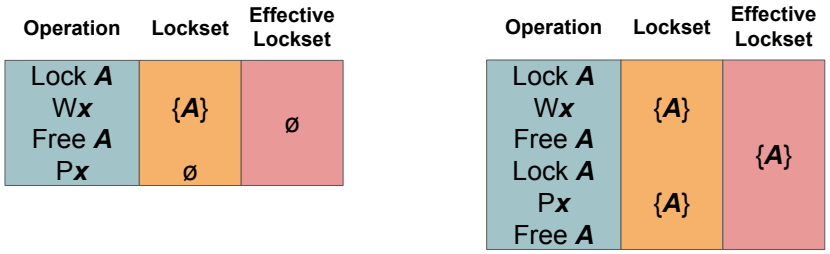
PM-unaware Lockset Analysis

Remember that a lockset is a set of all locks acquired at any given point of the execution, namely, the lockset of a memory access is the set of all locks acquired when the access took place. The result of intersecting two locksets is all the locks that were acquired when both accesses were executed.

Algorithm 4.1 works by comparing the lockset of each write to the locksets of each read to the same memory region, and reports a bug if the resulting set is empty. This algorithm does not take into account the moment when a store is persisted. We have not been able to find instances of write-write races resulting in malign side effects in the context of PM. Although traditional lockset analysis algorithms can and do check for write-write races, we did not represent it in Algorithm 4.1 since we plan on expanding it towards persistent races.

Algorithm 4.1: PM-unaware Lockset Analysis

```
   input: writes - The set of all writes
          reads - The set of all reads
1  foreach address ∈ writes do
   |  /* tid - thread ID */
2  |  foreach (tid, write) ∈ writes[address] do
3  |  |  foreach (tid', read) ∈ reads[address] where tid ≠ tid' do
4  |  |  |  if write.lockset ∩ read.lockset = ∅ then
5  |  |  |  |  report(write, read)
6  |  |  |  end
7  |  |  end
8  |  end
9  end
```



(a) (b)

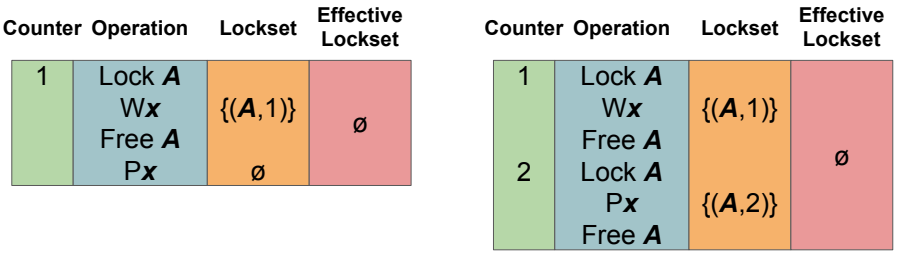
Figure 4.2: Locksets extended for PM accesses. The first column represents each operation executed. The second column represents the lockset of each PM access. Finally, the third column corresponds the effective lockset of the write operation.

PM-Aware Lockset Analysis

As previously hinted at, the first extension required for the accurate report of concurrent PM races, is to take into account the persistency of the store. We want to check whether the write and the respective persistency are protected by the same lock. To achieve this, we compute the intersection of locksets for the write and persistency, which we call the effective lockset. It represents the locks that protect both instructions. Figure 4.2a demonstrates this extension, the write operation is inside the critical section, however the persistency is not. Under the original lockset analysis algorithm we would not be able to detect these bugs however, if we compute the effective lockset then it correctly tells us there is no common lock protecting the write and the persistency.

This extension can discover a good chunk of persistent races, however, if the lock is released and reacquired between the write and the persistency, there is room for a racy access that is not detected. Figure 4.2b demonstrates this problem. The effective lockset is not empty even though we can intuitively observe that a race is possible. To account for this situation, we extend the locksets with a per thread logical clock that is incremented when a lock is acquired. This allows us to guarantee that the lock protecting the write and the persistency were acquired at the same time. We can see how this correctly computes the effective lockset by looking at Figure 4.3b. In this example, both the write and the persistency are protected by lock `A`, however, this synchronization is incorrect, because the lock is released and reacquired between both operations. The introduction of logical clock, correctly results in an empty effective lockset. Figure 4.3a is the extension of Figure 4.2a with the logical clock, we can see that it does not affect this already working example.

Finally, we intersect the effective lockset of the write operation with the lockset of the read operation. The logical clock associated with each lock in the effective lockset is discarded for this



(a) (b)

Figure 4.3: Locksets extended for PM accesses with the logical clock. The first column represents the logical clock.

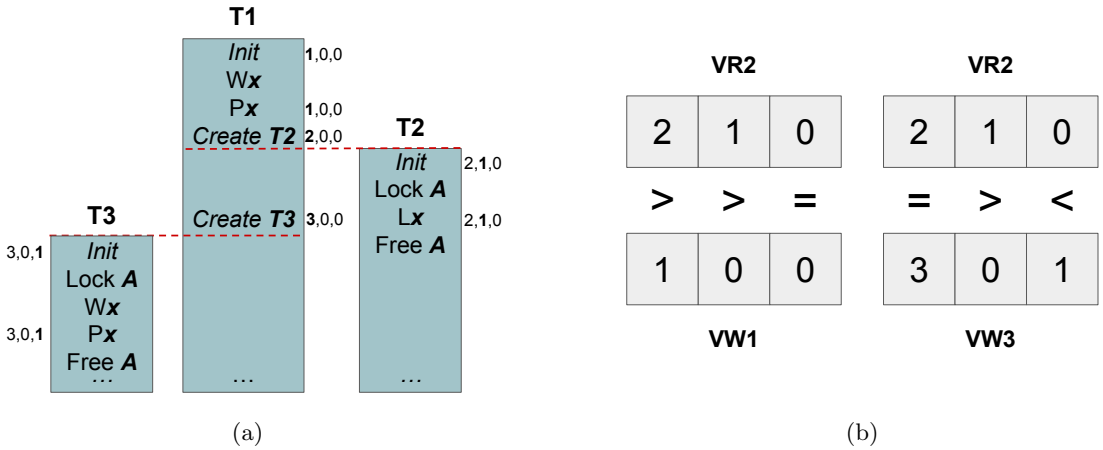
intersection since it is meaningless among accesses from different threads. As per the original algorithm, if the intersection corresponds to an empty set we report a concurrent PM race.

Happens-Before Analysis

The final extension reduces the number of false positives by leveraging the happens-before relation of events between threads. Figure 4.4a represents the execution of a multithreaded application, it performs several memory accesses that can be analyzed using our algorithm. However, some of these pairs of accesses can never run concurrently. For example, the unprotected write in `T1`, can never run concurrently with the read in `T2`, because `T2` is only created after the first access is performed. The write on `T3` can in fact be executed concurrently with the read on `T2`. Even thought we can assess this intuitively, we need a systematic way to approach this problem. We can solve it by using vector clocks. Given two operations with their respective vector clocks `V1` and `V2`, the operations are concurrent if there is at least one index `i` where `V1[i] < V2[i]` and another index `j` where `V1[j] > V2[j]`. Figure 4.4b represents the computations required to check the previous definition. On the left side, we compare the vector clocks of the write by `T1`, `VW1`, with the vector clock associated with the read by `T2`, `VR2`. There is no index where `VW1 > VR2` therefore these operations are not concurrent. However, for `VW3` (the vector clock for the write performed by `T3`) and `VR2`, we can see that the condition is true, which implies that these operations execute concurrently, and thus we can apply lockset analysis. We can see that this definition follows our previously stated intuitions for each pair of accesses.

Summary

Algorithm 4.2 is the result of the previously outlined extensions to a PM-unaware lockset analysis algorithm. It starts by pairing every RLP (store instructions that pass the initialization



(a) (b)

Figure 4.4: Multithreaded execution, accompanied by the respective vector clocks of each event

removal heuristic), with every read operation from a different thread that can occur concurrently according to the happens-before relation. This is represented by lines 2 and 3 of the algorithm, and the happens-before analysis described above is represented by the `isConcurrent` function. Then, in line 4, we compute the effective lockset for the store operation by intersecting the locksets of both the store and the persistency, while using the logical clock to ensure atomicity between both. The logical clocks are per thread, and therefor are subsequently ignored when the effective lockset is intersected with the read lockset in line 5. This is because the read is executed in a different thread than the store, and intersecting said logical clocks would be meaningless. Finally, in line 6, if the aforementioned intersection is an empty set, a race is reported.

Algorithm 4.2: PM-aware Lockset Analysis, with happens-before relation

```
   input: RLPs - The set of RLP
          reads - The set of all reads
 1 foreach address ∈ writes do
   |  /* tid - thread ID */
 2 |  foreach (tid, write, persist) ∈ RLPs[address] do
 3 |  |  foreach (tid', read) ∈ reads[address] ∧ isConcurrent(tid, write, tid', read) do
 4 |  |  |  efective_lockset ← write.lockset ∩ persist.lockset;
   |  |  |  // The logical clock is ignored for the final intersection
 5 |  |  |  if efective_lockset.locks ∩ read.lockset = ∅ then
 6 |  |  |  |  report(write, read)
 7 |  |  |  end
 8 |  |  end
 9 |  end
10 end
```

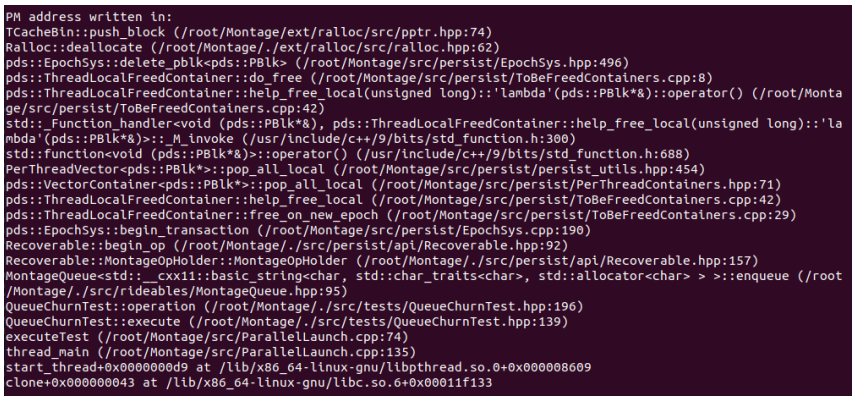
4.4 Implementation

In this section we discuss the implementation details of HawkSet, our automatic, application agnostic and efficient tool for concurrent PM bug detection.

HawkSet is implemented in C++ as a PIN tool, a binary instrumentation framework [43]. Using PIN's infrastructure, we inject instrumentation for PM instructions (stores, reads, flushes, fences and read-modify-write). To differentiate between PM accesses and regular accesses we record calls to `mmap` for PM files, and then compare the target address of the operation with the previously recorded PM regions. To achieve this, the PM files paths are required as a command line argument.

Our instrumentation requires that the application uses the pthread library's synchronization primitives, or provides an extremely simple configuration file with its custom version. HawkSet accepts one or more configuration files as a command line argument. These configuration files have a simple YAML structure and only need to supply the name of the synchronization primitive and, in the case of `try_locks`, must supply the value correlated with a successful acquisition. We believe this does not break our initial goal of automation for the following reasons: (1) most applications already use pthread, or build their abstractions on top of pthread's primitives, and (2) unlike other tools that require a driver that exposes the semantics of the application, which can be constantly changing and must be maintained, HawkSet only requires simple information about the synchronization primitives, which constitute a small set of functions, that have been solidified over years of research and do not change nearly as often. For example, during the evaluation of our tool, we analyzed an application that used `pmemobj`'s version of synchronization primitives. After a few minutes, we produced a configuration file that is now usable for all applications that use pmemobj's locking mechanisms.

Over the course of this chapter, we talked about accesses to memory regions. In Algorithm 4.2, the base address of each access is used as an identifier for each memory region. However, in reality, there can be two accesses to different base addresses that still access the same memory region, or at least, part of it. For example, an eight byte access to the address `0x1000`, should be compared against another access in address `0x1004`, since these regions overlap partially. If we use the base address has the identifier of the access, we would ignore the previous example. Our implementation takes this into account by breaking each access in byte sized chunks, ensuring that accesses to the same memory region, but with different base addresses, are still analyzed as expected.



(a) Write

(b) Flush

(c) Read

Figure 4.5: Write, Flush, and Read responsible for a persistent race

4.4.1 Ergonomics

The ergonomics of a debugging tool is always an important aspect of its development. Although we consider this an orthogonal problem, some considerations have been made when it comes to ergonomics in HawkSet. Namely, we output reports grouped in a fashion that we believe is easier to use, although no formal study was conducted. The output itself is also cleaned for better readability, although its efficacy may vary depending on the language in which the application was written. Figure 4.5 displays a report for the Montage framework, we present the backtrace for both the write and flush responsible for the persistent race in question. Alongside it, we present a list of all backtraces where a read operation can concurrently access the store. We believe that the backtrace information is crucial when fixing and verifying certain hard to reach bugs. Finally, it is containerized using Docker, which allows for an easy setup.

4.5 Summary

HawkSet is an automatic, application agnostic, and efficient concurrent PM bug detector. It leverages lockset analysis, the happens-before relation, and a specific heuristic, to detect persistent races. Even thought lockset analysis and happens-before relations based techniques are known to not scale well, we are able to rely on them because PM accesses constitute a small fraction of all memory accesses. All features described in this chapter are currently implemented as a PIN tool in C++, fully containerized using Docker.

In the next chapter, we present our experimental evaluation of HawkSet.

Chapter 5

Evaluation

In this chapter we discuss the experimental results of our evaluation of HawkSet. We evaluate its resource footprint, bug finding ability, its limitations, and compare it to the state-of-the-art. Finally, we discuss its applicability in a real software development environment.

5.1 Research Questions

HawkSet's evaluation focuses on three areas, namely: (1) automation and application agnosticism, (2) resource usage, and (3) coverage. To evaluate HawkSet we must answer the following research questions:

- How automatic is HawkSet?
- How application agnostic is HawkSet?
- What is the resource consumption of HawkSet?
- What persistent races can be detected by HawkSet?
- What is the impact of our heuristic in the races reported?
- What is the impact of our heuristic in resource consumption?
- How does HawkSet compare to the state-of-the-art?

5.2 Experimental Setup

To answer these questions we performed a series of experiments in a machine with an Intel(R) Xeon(R) Gold 6338N CPU @ 2.20GHz, 256 GB of RAM and 128 cores, on top of a 1 TB Intel DCPMM in App Direct mode.

Application	Version	Synchronization Method	Supported by	
			Durinn	PMRace
Fast-Fair [6]	0f047e8	Lock/Lock-Free	Yes	Yes
P-CLHT [4]	70bf21c	Lock	Yes	Yes
TurboHash [3]	2d7d7b3	Lock/Lock-Free	Yes	Yes
Montage-Queue [11]	3384e50	Lock	Yes	Yes
Montage-HashTable [11]	3384e50	Lock	Yes	Yes
Montage-Graph [11]	3384e50	Lock	No	No
MadFS [9]	7514a78	Lock-Free	No	No

Table 5.1: PM applications tested using HawkSet

5.2.1 Target Applications

We evaluated a total of 5 PM applications, outlined in Table 5.1. The first column contains the name of the application, the second column contains the git version used for our analysis, the third column depicts the locking mechanisms used by the applications. Finally, the last two columns say whether the applications could be debugged using either Durinn or PMRace. To better understand the results discussed in this chapter, we offer a short description of each application:

Fast-Fair is a PM backed B+-Tree. It leverages the cache-line ordering constraints of PM to perform atomic insertions without the need for a recovery process that fixes inconsistencies. `Fast-Fair`'s synchronization methods are complex. It mixes lock-based concurrency control with lock-free methods to synchronize its shared PM accesses. Insertions and deletions performed on the leaf nodes of the tree are protected by a lock, while the same operations occur freely for internal nodes. This makes `Fast-Fair` an interesting challenge to debug using our lockset based approach. `Fast-Fair` has multiple implementations (some of which do not use real PM), for the purpose of our analysis, we used the concurrent version that uses real PM, supported by PMDK.

P-CLHT [4], and **TurboHash** [3] are PM backed hash tables. `P-CLHT` restricts the size of each bucket to that of a cache-line, as this guarantees that updates to each bucket can be persisted atomically. It uses bucket-specific locks to synchronize insertions and updates, and a global lock for rehashing. Read operations occur in a lock free manner. `TurboHash` brings forth improvements in different areas: it performs efficient out-of-place updates, it minimizes long-distance linear probing, and exploits hardware features, such as Intel's AVX registers. `TurboHash` has three different implementations. For the purpose of our analysis, we used the version that uses real PM, supported by PMDK.

Montage [11] is a PM framework that provides support for building buffered persistent data structures. The framework does not provide linearization guarantees to the persistent data

structures on its own, only the building blocks to achieve it. We tested three different data structures, provided by the authors of `Montage`, namely: a queue, a hash table, and a graph. Our application agnostic analysis covered both the framework and the structures built on top of it.

MadFS [9] is a PM backed file system. It maintains a mapping of all virtual blocks it manages, via a compact, crash-consistent log, where entries are 8 bytes long and therefore updated atomically. It manages some file metadata in user-space, in the form of a log, updated atomically and delegates security related operations to the kernel's underlying file system.

5.2.2 Workloads

Due to the variety of the applications evaluated, we are not able to use a single benchmark. In turn, we divide our evaluation in three groups: (1) key-value stores or equivalent, (2) `Montage`, and (3) `MadFS`. All experiments were run with 8 threads.

Key-Value Stores

`P-CLHT`, `Fast-Fair`, and `TurboHash` support similar operations (insert, update, get, and delete), and as such, we use the same workload to better compare their results. To measure the scalability of HawkSet, we ran four separate experiments for each of these applications. Each workload was generated using YCSB [44] with an initial load phase of 1k insertions, followed by 1k, 5k, 10k, or 50k operations comprised of 30% insertions, 30% updates, 30% gets, and 10% deletes.

Montage

`Montage`'s internal benchmarks run for a fixed amount of time, instead of executing a set number of operations. Each implementation, (queue, hash table, and graph) was executed four times with an increasing amount of time, namely 10, 100, and 600 seconds. Before each experiment there is a single-threaded load phase. In the case of `Montage-Queue` and `Montage-HashTable` this comprised of 1k enqueues and insertions, respectively, whilst in the case of `Montage-Graph` this comprised creating 5k vertices, each with an average of 5 edges.

MadFS

`MadFS` provides some benchmarks out-of-the box. The benchmark used performs 4kb write operations in a 1Mb shared file among all threads. The target offset of the operation is randomized and follows a zipfian distribution. We ran the benchmark four times, with increasing workload, with 1k, 5k, 10k, and 50k operations.

5.3 Automation and Application Agnosticism

As described in Chapter 3, automation and application agnosticism constitute an important aspect in the design of HawkSet and, as such, evaluating these characteristics is fundamental. Given the nature of these metrics, the evaluation is qualitative and not quantitative.

5.3.1 Automation

To evaluate automation, we want to outline the impact that our tool has on its users, in terms of modifications to the source code, creating artifacts for the debugging process, and other hindrances. Our tool requires the instrumentation of synchronization primitives and, as expected, the only alterations made to the original applications were aimed at extracting this information. We support pthread and pmemobj's synchronization primitives out of the box, and any configuration file built for one particular library can be reused.

`P-CLHT` implements its concurrency control in-place, using `CAS` instructions directly in the source code. To instrument these operations, we extracted the concurrency controls, by hand, to functions, and subsequently created a configuration file that covers them. Overall the extraction process took less than an hour, and the configuration file was created in a few minutes. We believe that for the developers of `P-CLHT`, this process would take much less time, since they would be familiar with the source code.

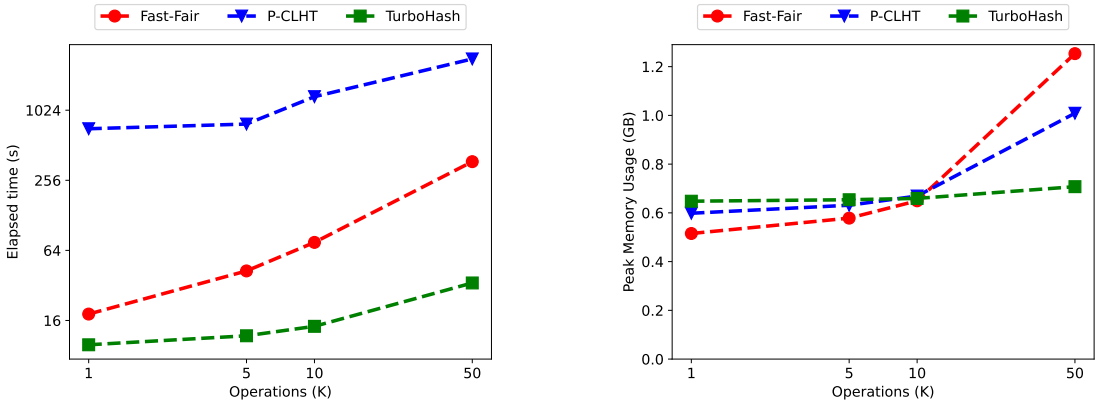
`TurboHash` uses custom concurrency control primitives, we created a configuration file to instrument them. Overall, there were 3 primitives, which took a few minutes to find and enumerate.

For `MadFS`, `Montage`'s data structures, and `Fast-Fair`, no modifications were required for debugging under HawkSet. This is because `MadFS` does not use synchronization primitives, and `Montage` and `Fast-Fair` use pthread's synchronization libraries, which we support implicitly.

Overall, we believe HawkSet is extremely automatic, providing concurrent PM debugging whilst requiring minimal effort from the user.

5.3.2 Application Agnosticism

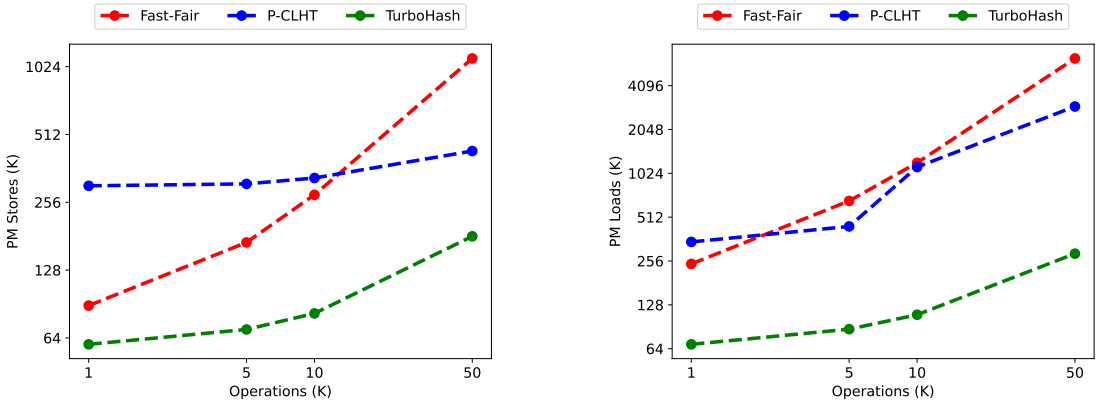
We measure application agnosticism by the variety of PM applications the tool can debug. We were able to evaluate HawkSet using key-value stores, a framework for persistent data structures, including a graph, and a file system. Furthermore, HawkSet does not require a specific version of PMDK, or even that PMDK is used at all. When it comes to application agnosticism, HawkSet provides very ample support. As it does not make any assumptions about the semantics of the system under testing during the analysis.



(a) Time elapsed. Note that the y-axis is logarithmic

(b) Peak memory.

Figure 5.1: Time elapsed and peak memory usage for P-CLHT, Fast-Fair, and TurboHash. Note that the x-axis is logarithmic



(a) PM store instructions.

(b) PM load instructions.

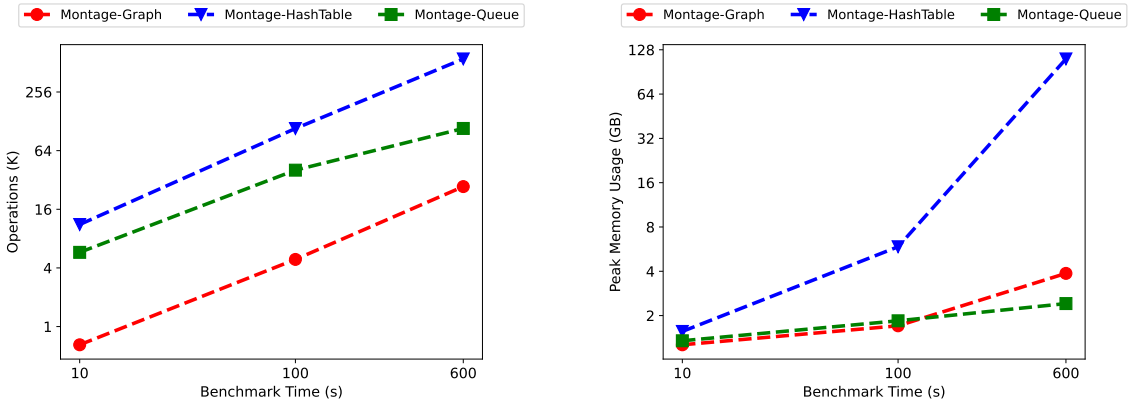
Figure 5.2: PM instructions processed by HawkSet while analyzing P-CLHT, Fast-Fair, and TurboHash. Note that both axes are logarithmic

5.4 Resource Consumption

Resource consumption is an important metric when deciding which tools to use, and when to use them. To determine HawkSet's time and memory footprints we conducted a series of experiments whose results are represented in Figure 5.1 for the key-value stores, Figure 5.3 for `Montage`'s implementations, and Figure 5.4 for `MadFS`.

5.4.1 Key-Value Stores

Figure 5.1a shows the duration of HawkSet's analysis for `P-CLHT`, `Fast-Fair`, and `TurboHash`. The duration of the analysis grows linearly with the number of operations. We observe an initial cost for `P-CLHT`, for all workloads. We believe this is due to initialization performed by the application. To justify this belief, we counted the instructions processed by HawkSet during



(a) Time elapsed

(b) Peak memory

Figure 5.3: Time elapsed and peak memory usage for Montage-Queue, Montage-HashTable, and Montage-Graph. Note that both axes are logarithmic

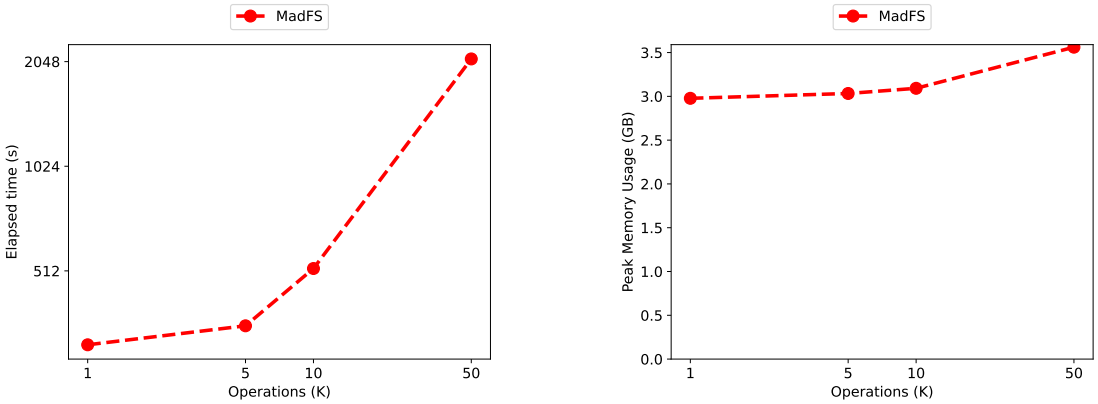
the analysis, Figure 5.2 contains the results. We can see that for `P-CLHT` there is a significant amount of PM stores, even for the smallest work. This correlates to the initialization of the PM pool, that `P-CLHT` memsets to zero. All of these instructions are instrumented which increases the duration of the analysis. We can also see a jump in PM load operations between the 5K and 10K workloads. This is most likely caused by a rehashing operation that must copy the old hash table into the newer, bigger hash table. We believe this is the cause for the small bump in analysis time between the same workloads. For the other applications, `Fast-Fair` and `TurboHash`, the instructions grow linearly, without irregularities, following the same trend as the analysis time.

Figure 5.1b depicts the maximum resident memory usage while debugging the key-value stores. This demonstrates that peak memory usage grows linearly, although with a considerably high initial cost (approximately 0.6 GB).

Although we are not sure of the reason for it, we believe that it is caused by a combination of the following factors: (1) PIN's library , (2) the injected instrumentation, and (3) registering PM accesses of the initial initialization (i.e: pool allocation, initial memsets, etc...) and prefill phase. We leave as future work studying the concrete sources of this behavior and finding ways to optimize it.

5.4.2 Montage

Figure 5.3a shows the number of operations that could be completed within a set amount of time for `Montage`'s different data structures. We can see that the number of operations increases linearly with the benchmark duration. It is interesting to note that, even though these applications were built on top of the same framework, they appear to have varying performance. For



(a) Time elapsed. Note that the y-axis is logarithmic

(b) Peak memory. Represented on the y-axis

Figure 5.4: Resource usage in the analysis of MadFS. Note that the x-axis is logarithmic

example, `Montage-HashTable` performs around 5 times more operations than `Montage-Queue` on the same amount of time, which in turn performs around 4 times more operations than `Montage-Graph`. Due to time constraints no further experiments were conducted to understand this behavior, but we believe this to be due to a combination of the following factors: (1) `Montage-HashTable` is more paralelizable than it's counterparts, since it is synchronized at the bucket level, which allows for multiple operations on different buckets, and (2) the implementation of `Montage-Queue`'s operations are much simpler than that of the other data structures.

Figure 5.3b depicts the peak memory usage of HawkSet during the analysis of `Montage`. For `Montage-Queue` and `Montage-Graph`, the memory usage follows a similar trend to that of the previously mentioned key-value stores. However, `Montage-HashTable` does not, and appears to grow super-linearly. We are not sure as to why, and while no further experiments were conducted due to time constraints, our suspicion is that some expensive rehashing operation is being run which leads to a super-linear growth in PM operations that HawkSet must process.

5.4.3 MadFS

Figure 5.4a shows the duration of HawkSet's analysis of `MadFS`. Similarly to the results presented for the key-value stores, the growth appears to be linear, which is a strong indicator of HawkSet's scalability.

Figure 5.4b represents the peak memory usage of HawkSet during the analysis. As with previous applications, we can see a large initial memory cost, after that, memory usage grows linearly.

5.4.4 Discussion

Overall, we see three patterns in HawkSet's resource consumption: (1) analysis time grows linearly with the size of the workload, (2) there is a considerable initial memory cost, and (3) memory usage increases linearly.

Analysis time grows linearly with the size of the workload because we only observe the execution once, instrumenting each PM related instruction along the way. In the context of PM debugging, it is common to see tools whose analysis time grows quadratically or even exponentially. This is a strong indication of the efficiency of HawkSet, and its ability to scale for larger workloads.

Figure 5.2 outlines PM store and load instructions processed by HawkSet for the key-value stores. We can see these instructions grow linearly as the workload increases. We believe that memory usage increases linearly because HawkSet keeps most of the PM accesses registered throughout the execution until the end, when it finally analyzes them for persistent races. This means that the memory footprint grows with the PM accesses, which, as we saw previously, tend to grow linearly with the size of the workload. A garbage collection mechanism could remedy this situation, however, deciding which accesses can be discarded, merged, or otherwise reduced is not trivial. We leave studying the feasibility of such a mechanism for future work.

In short, HawkSet provides a very scalable approach to concurrent PM bug detection. It is able to scale for large workloads, both in time and memory usage, which grow linearly with respect to the size of the workload.

5.5 Persistent Races

In this section we discuss the persistent races reported by HawkSet, argue their validity, and present some interesting examples. We divide reports into three distinct categories, namely:

- Malign Persistent Races - Pairs of PM accesses that can execute concurrently and can cause an adverse effect in the system
- Benign Persistent Races - Pairs of PM accesses that can execute concurrently, but the application is designed to tolerate them
- False Positives - Pairs of PM accesses that can never execute concurrently

5.5.1 False Positives

In this section, we provide an example of a false positive. We define false positives, as pairs of accesses, (write-read), that can never execute concurrently.

One such example is the case when variables are initialized. In general, applications want to protect their shared accesses using locks, in order to prevent races. However, this constitutes an expensive overhead and, there are situations when such synchronization is not necessary. During initialization, the thread that allocates a certain memory region is the only one who has access to it, since the pointer as not yet been made visible to the other threads. This means that applications do not need to worry about protecting these accesses with expensive synchronization primitives. Reporting these accesses as a persistent race would constitute a false positive, since they can never execute concurrently.

5.5.2 Benign Races

Benign races are a class of persistent race, where the accesses can execute concurrently, but the design of the application tolerates or corrects the side effects. As far as we know, there is no way to differentiate between benign and malign races using lockset analysis, specially in the context of PM, where the recovery program can mitigate the effects of some races. In this section we describe two such races detected by HawkSet in `TurboHash` and `MadFS`.

`TurboHash` uses a snapshot mechanism to ensure thread-safety for lock-free get operations. It first finds the bucket associated with the entry and saves a local copy of the metadata of the bucket. This metadata contains, among other things, a version number that is incremented when the bucket is changed. It then traverses the bucket until it finds the specific entry and, if the version number remains the same, then there has been no concurrent update to the bucket, which means it can return the found value. If the version number has changed, it restarts the whole process. Reading the bucket and entry is clearly a race, since no synchronization primitives protect these accesses. However, this race cannot result in an incoherent state, which makes it benign.

`MadFS` uses `CAS` operations to update its block allocation log atomically. This update is executed without holding a lock which means accesses made to the log are reported as a race by HawkSet. If thread `A` performs an IO write call that forces `MadFS` to allocate a block, the metadata written with that `CAS` is not guaranteed to be persisted until flushed. `MadFS` flushes its log when `fsync` calls are issued by the application. This means that, if thread `B` reads from the memory region just written by thread `A` before it calls `fsync`, we have a persistent race. However, a file system has different thread-safety requirements than that of a general PM

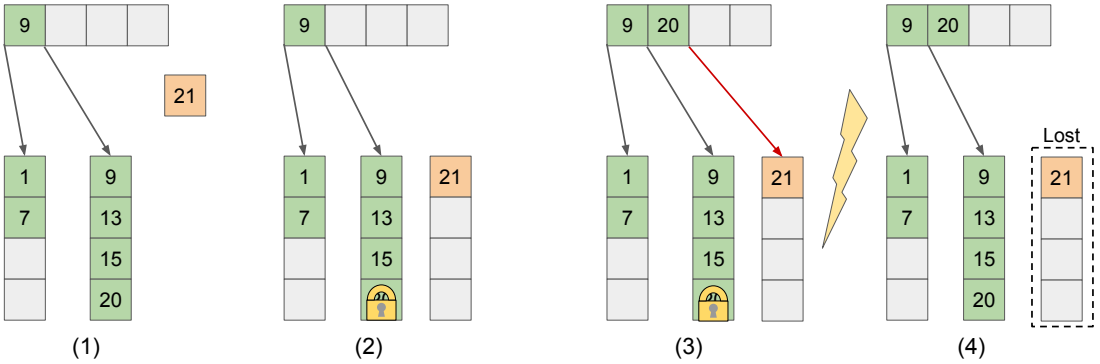


Figure 5.5: Fast-Fair insertion crash. Boxes represent entries in each node of the B+-Tree, while arrows represent pointers. A red arrow represents an unpersisted pointer.

application. Namely, writes to a file are only guaranteed to be flushed to storage after a `fsync` call. Which means, even though there is a persistent race, it does not violate the requirements of `MadFS`. Instead, it would be an error of the application built on top of `MadFS` to not synchronize its accesses to the file. In the context of `MadFS`, this constitutes a benign persistent race.

5.5.3 Malign Persistent Races

HawkSet detected 3 malign races: one in `Fast-Fair` and another in `P-CLHT` that were previously reported by the state of the art, and one in `TurboHash` which, to the best of our knowledge, constitutes a new bug.

Fast-Fair

The bug detected in `Fast-Fair` occurs whenever the tree grows. In `Fast-Fair`, the internal nodes of the tree are accessed in a lock-free manner, while the leaf nodes are locked. Take into consideration the example provided in Figure 5.5. We start with a B+-Tree, where one of its leaf nodes is full (1). Assume that a new entry is about to be inserted. When attempting to insert a new entry in a full node, a sibling node is created (2), which carries with it the new entry which did not fit in the previously full node. Then, the sibling node is inserted in the parent node (3), which is not protected, since the parent node is internal. At this moment, the new entry is visible to all threads, however, the pointer from the parent node to the new child node is not persisted. If the application crashes, after recovery, the new node will not be visible, and its data would have been lost (4). This alone, does not constitute a bug, since if an operation crashes before terminating, the data created by that operation can be lost. However, assume that a different thread accesses the new entry between the insertion of the new sibling node and

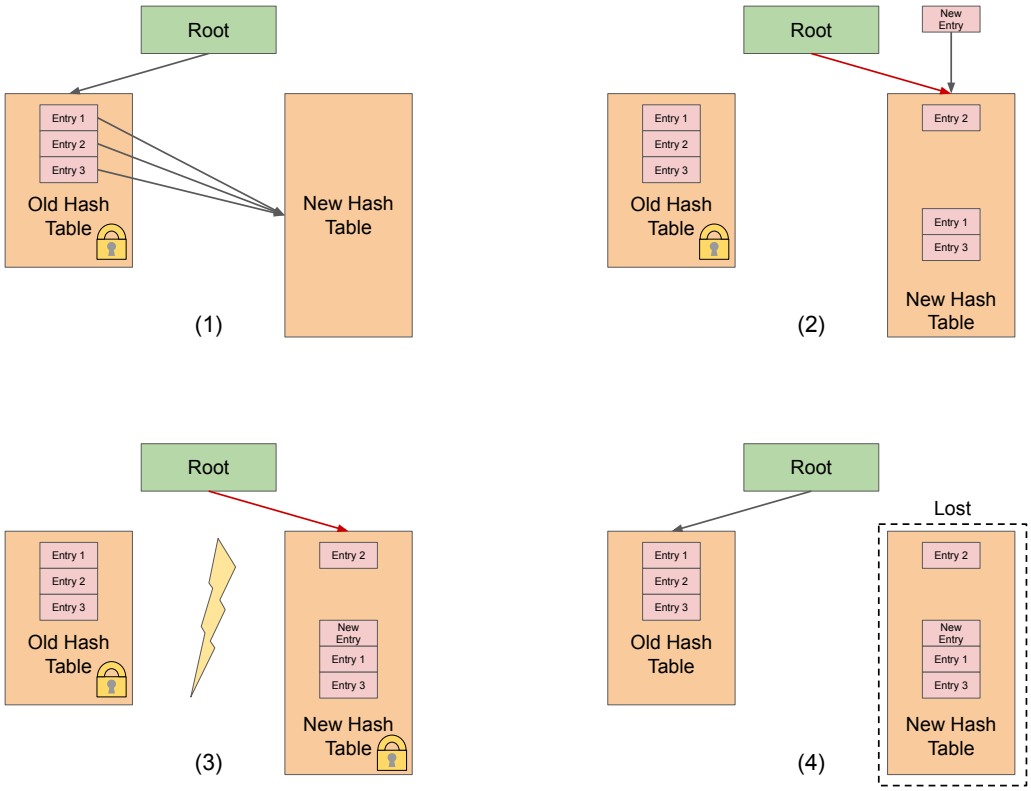
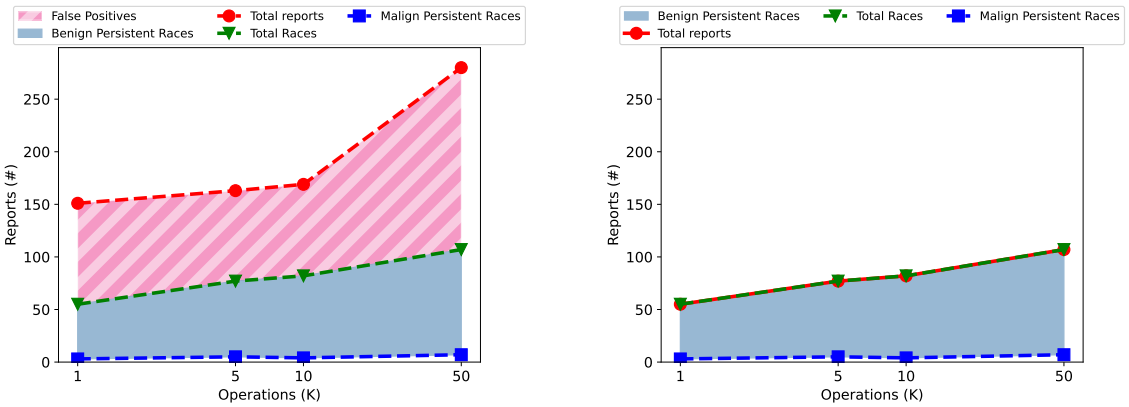


Figure 5.6: P-CLHT persistent race. Boxes represent entries in each node of the B+-Tree, while arrows represent pointers. A red arrow represents an unpersisted pointer.

the crash, and then performs a side effect. If this happens, after the crash, the side effect will be visible, however the original entry that caused it will not. This constitutes a malign race.

P-CLHT

The bug detected in `P-CLHT` is caused by a rehashing operation. Figure 5.6 provides a visual representation of the persistent race in action. `P-CLHT` atomically rehashes its hash table by locking it, copying all the entries to a new hash table (1). Then it atomically swaps the root pointer to the new hash table (2). Before the root pointer is persisted, a different thread inserts a new entry into the new hash table (3). This insertion is able to acquire the lock of the new hash table, because only the old hash table was locked during rehashing. If the application crashes after the insertion, but before the rehashing operation fully completes, then the data inserted by the new thread is lost (4). This constitutes a malign race. It is important to note that the persistent race occurs between the write that swaps the root pointer, and the read in the insertion operation that accesses the root pointer.



(a) Without the initialization removal heuristic

(b) With the initialization removal heuristic

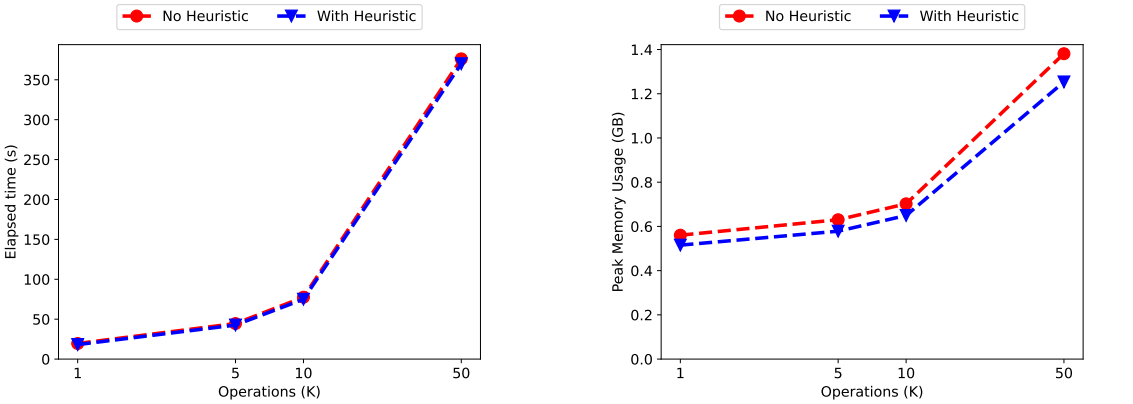
Figure 5.7: Breakdown of HawkSet's reporting of Fast-Fair. Note that the x-axis is logarithmic

TurboHash

`TurboHash`'s insertion operation writes an entry into the bucket, and it then performs some metadata manipulation, and flushes it. This operation is protected by a lock, meaning no persistent race should be possible. However, `TurboHash`'s buckets are larger than a cache-line, and therefore, when the entry is too far from the metadata, such that it falls on a different cache-line, the update is not persisted. HawkSet is able to detect this bug because the update is eventually persisted by the delete operation on a different entry in the same cache-line. In this case, the store and the persistence are not protected by a common lock and therefore HawkSet reports it as a persistent race. Although it is caused by a missing flush, it is still a persistent race, because a thread can read the newly inserted entry and perform some side effects. If the application crashes before the original entry is persisted, the side effects that depend on that entry will be present after the crash, but the original entry will not. This bug can only be detected as the buckets start to fill, which means it requires larger workloads to detect. HawkSet was only able to report it with the biggest workload we evaluated. This is a strong indicator that HawkSet's ability to debug large workloads is important when detecting concurrent PM bugs. This persistent race is caused by a missing flush, which means, it should be detected by tools that do not focus on current PM bugs.

5.6 Initialization Removal Heuristic

In this section we study the impact of the initialization removal heuristic, in resource consumption and bug reporting. To evaluate the heuristic, we ran an experiment on `Fast-Fair`. It is equivalent to the one described in Section 5.2.2, except that we ran it twice, with and without the initialization removal heuristic.



(a) Analysis duration in function of operations

(b) Peak memory usage in function of operations

Figure 5.8: Impact of the initialization removal heuristic in resource consumption

Figure 5.7 summarizes the results of this analysis. False positives, benign races, and malign races were classified manually. Figure 5.7a represents a breakdown of all the persistent races reported by HawkSet when analyzing `Fast-Fair` without the heuristic, while Figure 5.7b represents the reports while using the heuristic. The ● line represents all reports by HawkSet, the ▼ line represents all races that can execute concurrently, and the ■ line represents malign races. Logically, the dashed pink area represents the false positives, and the filled light blue area represents the benign persistent races. For `Fast-Fair`, the initialization removal heuristic is able to detect and discard all false positives reported by HawkSet without the heuristic. This constitutes an extremely positive result. With that being said, the heuristic is not foolproof, since there are certain uncommon cases where false positives can still be reported. For example, if two threads perform the initialization, then part of those accesses will not be discarded, and therefore will show up as false positives on the report. Due to time constraints we were not able to fully analyze all reports from `P-CLHT`, however, we were able to detect at least one report that suffers from this issue. We attempted to fix this issue by generalizing the heuristic for `N` threads, but this did not improve our results. We leave the study of these and other false positives, and how to mitigate them, as future work.

Figure 5.8 displays the impact of the initialization removal heuristic in the resource usage of HawkSet. Remember that the initialization removal heuristic is responsible for discarding PM accesses that are commonly associated with the initialization of unpublished memory regions, which might induce false positives.

When it comes to the duration of the analysis, represented in Figure 5.8a, we see that the initialization removal heuristic has a very small impact, (a reduction of approximately 1% for the largest workload). The heuristic is able to reduce the accesses that must be processed with lockset analysis algorithm, which greatly reduces the duration of that final analysis step. How-

ever, most of the cost of HawkSet's execution consists of instrumenting PM accesses, simulating the cache, and registering accesses, while only small portion is spent performing the lockset analysis algorithm itself.

In the case of peak memory usage, we can see a considerable improvement, (approximately 10%). This is because PM accesses are discarded by the heuristic and are never registered, which reduces the memory footprint of HawkSet.

Overall, the initialization removal heuristic performs a great job at removing false positives.

5.7 State-of-the-art

In this section we outline the results of a comparative evaluation of HawkSet with the state-of-the-art in concurrent PM bug detection.

5.7.1 Automation and Application Agnosticism

To compare HawkSet to the state-of-the-art, when it comes to automation, we provide examples of two applications that were debugged by all three tools: Durinn, PMRace, and HawkSet.

`Fast-Fair` was evaluated both by Durinn, PMRace, and HawkSet, which provides a very useful comparison among all three tools. Durinn's evaluation required the creation of a specific test driver written in C++, consisting of 191 LOC. For the evaluation of PMRace, complex changes to the Makefile and test driver were made, 329 lines according to git's diff tool. The Makefile was completely overhauled to support PMRace's fuzzing driver, and its custom instrumented libraries. The test driver was modified to support the fuzzing engine. It is important to note that if the semantics of the application change at any point of the development cycle, these artifacts would need to be remade. HawkSet requires no modification to the original source code, and no additional artifacts, to debug `Fast-Fair`.

Another application that is evaluated by all three tools is `P-CLHT`. As expected, Durinn required a complex driver consisting of 181 LOC, the manual removal of non-deterministic behaviors, and the extraction of inline synchronization primitives into functions. PMRace also required a specialized driver with 189 LOC. Unlike with `Fast-Fair`, HawkSet requires some minor modifications, similarly to Durinn, synchronization primitives must be extracted into separate functions. Even then, we believe HawkSet offers a much higher degree of automation, since extracting synchronization primitives can be considered a trivial task compared to creating a specialized driver.

Regarding application agnosticism, HawkSet is able to analyze all types of applications. According to Table 5.1, HawkSet can analyze two applications that the state-of-the-art cannot:

`Montage-Graph` and `MadFS`, because they do not follow the semantics of key-value stores or equivalent. Furthermore, Durinn and PMRace assume the usage of PMDK for the development of PM applications, while HawkSet does not.

In short, when it comes to automation and application agnosticism, HawkSet contrasts with Durinn and PMRace, as it requires minimal to no modification of the original applications it debugs and supports a wide range of application types.

5.7.2 Effectiveness

In this section we compare HawkSet to PMRace, in their ability to find bugs effectively. We would have liked to compare HawkSet to Durinn, however due to time constraints it was not possible. We performed this evaluation with `Fast-Fair` as our target application, since it can be evaluated by both PMRace and HawkSet.

To reiterate, PMRace's approach is divided into two separate stages. The first one uses fuzzing and specialized delay injection to find races in PM applications. The second stage uses the recovery program to check whether those side effects were resolved.

In this evaluation we are interested in the first phase of PMRace, since HawkSet shares the same end goal. However, these tools work in very different ways and, as such, they cannot be directly compared.

PMRace starts off with a small workload, which it calls the seed, and then executes the application. On subsequent executions it mutates the original seed, and executes again. This process only stops when PMRace decides that a seed is not worth exploring anymore, and at that time, it gets a new seed and starts the process all over again. This way, PMRace's evaluation theoretically never ends. In practice, the authors provide 240 different seeds for `Fast-Fair`, which means that the analysis ends eventually.

HawkSet, runs an application, extracts all the PM accesses, and reports all the possible races within the execution it observed. As long as the target application terminates, so does HawkSet.

The design differences imply that we cannot directly compare the results of these tools. To remedy this situation, we executed PMRace for 10 minutes, with a set seed, and extracted all the mutated workloads it ran during that time. We chose 10 minutes with a specific seed because that is the setup recommended by the authors to replicate their analysis of `Fast-Fair`.

This resulted in 76 different workloads, each with an average of 549 operations, with varying distributions. We then executed HawkSet 76 different times, each time with a different workload.

PMRace was able to find the previously described persistent race with three of the 76 workloads. HawkSet was able to find the race in all 76 workloads. With an average analysis of

approximately 9 seconds for each workload. It took a total of 12 minutes. This is an extremely positive result, that demonstrates that HawkSet is very consistent in its persistent race finding abilities.

Even though coverage is an orthogonal problem in concurrent PM bug detection, this result shows that PMRace's success is much more sensitive to the workload, than HawkSet's. The delay injection analysis not only requires that the workload provides sufficient code coverage, which lockset analysis also requires, but it must provide good interleaving coverage as well. For example, if a workload consists of 1000 operations, where the first and last are racy, PMRace will most likely never find the bug, because it would require massive waits that considerably degrade the performance of the system. In the case of HawkSet, as long as it observes these accesses in different threads, it reports the race.

To further test this hypothesis we ran each of the 240 seeds provided by the authors of PMRace in separate, and only 9 were able to detect the persistent race in `Fast-Fair` within 10 minutes. This result shows that PMRace is very sensitive to the workload provided.

5.8 Discussion

To conclude this section, HawkSet presents a promising approach to concurrent PM bug detection. It is able to provide an automatic and application agnostic approach, supporting most applications with minimal or no modifications. It detects persistent races, with relatively small workloads, such as the persistent race in `Fast-Fair` reported in approximately 18 seconds. The initialization removal heuristic, despite not being foolproof, was able to remove all false positives when analyzing `Fast-Fair`. HawkSet is able to detect the same bugs detected by the state-of-the-art in the applications we evaluated, and it also found a new, previously unreported bug in `TurboHash`. When compared to PMRace's analysis of `Fast-Fair`, HawkSet is able to detect the persistency race much faster and more consistently. HawkSet is effective at finding persistent races with small workloads, able to scale for larger workloads, which increases coverage and allows detecting harder to reach bugs. This means that HawkSet can be used in a day-to-day production environment, possibly constituting a short build step, as well as part of more in-depth "overnight" bug detection efforts.

Chapter 6

Conclusion

Current state-of-the-art concurrent PM bug detection tools focus on specific types of applications, such as key-value stores or equivalent. They are not automatic, because they require additional complex artifacts for debugging and modifications to the original source code. Additionally, they cannot support the full range of PM applications, current and future, since they focus on key-value stores.

In this thesis we present HawkSet, an automatic, application agnostic, and efficient concurrent PM bug detector. It leverages lockset analysis, the happens-before relation, and an initialization removal heuristic to detect persistent races. We expanded a traditional PM-unaware lockset analysis algorithm in order to detect concurrent accesses to PM. The results of our evaluation demonstrate that HawkSet is effective at finding persistent races. We believe it can be a useful tool in all aspects of debugging.

Alongside our main contribution, we present a streamlined description of PM semantics and PM bug taxonomy, concurrency, and a concurrent PM bug taxonomy. Furthermore, we provide an in-depth analysis of the state-of-the-art of PM debugging, which impacted HawkSet's design goals.

6.1 Limitations and Future Work

Even though HawkSet is able to detect the same bugs as the state-of-the-art, in less time and more consistently, our approach does not contemplate a strong verification of each bug reported. This can increase the burden on the developer, that must process them by hand, in order to filter out benign races. As future work, we want to research automatic and application agnostic techniques that are able to confirm persistent races, comparable to PMRace's verification phase.

Bibliography

[1] S. Scargall and S. Scargall, "pmemkv: A persistent in-memory key-value store," *Programming Persistent Memory: A Comprehensive Guide for Developers*, pp. 141–153, 2020.

[2] S. Han, D. Jiang, and J. Xiong, "SplitKV: Splitting IO paths for different sized Key-Value items with advanced storage devices," in *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association, Jul. 2020. [Online]. Available: https://www.usenix.org/conference/hotstorage20/presentation/han

[3] X. Zhao, C. Zhong, and S. Jiang, "Turbohash: A hash table for key-value store on persistent memory," in *Proceedings of the 16th ACM International Conference on Systems and Storage*, 2023, pp. 35–48.

[4] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "Recipe: Converting concurrent dram indexes to persistent-memory indexes," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 462–477. [Online]. Available: https://doi.org/10.1145/3341301.3359635

[5] M. Nam, H. Cha, Y.-r. Choi, S. H. Noh, and B. Nam, "{Write-Optimized} dynamic hashing for persistent memory," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019, pp. 31–44.

[6] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in {Byte-Addressable} persistent {B+-Tree}," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018, pp. 187–200.

[7] W.-H. Kim, R. M. Krishnan, X. Fu, S. Kashyap, and C. Min, "Pactree: A high performance persistent range index using pac guidelines," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 424–439. [Online]. Available: https://doi.org/10.1145/3477132.3483589

[8] J. Yang, B. Li, and D. J. Lilja, "Heuristicdb: A hybrid storage database system using a non-volatile memory block device," in *Proceedings of the 14th ACM International Conference on Systems and Storage*, ser. SYSTOR '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3456727.3463774

[9] S. Zhong, C. Ye, G. Hu, S. Qu, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Swift, "{MadFS}:{Per-File} virtualization for userspace persistent memory filesystems," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, 2023, pp. 265–280.

[10] M. Dong, H. Bu, J. Yi, B. Dong, and H. Chen, "Performance and protection in the zofs user-space nvm file system," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 478–493. [Online]. Available: https://doi.org/10.1145/3341301.3359637

[11] H. Wen, W. Cai, M. Du, L. Jenkins, B. Valpey, and M. L. Scott, "A fast, general system for buffered persistent data structures," in *Proceedings of the 50th International Conference on Parallel Processing*, ser. ICPP '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3472456.3472458

[12] A. Lefort, Y. Pipereau, K. Amponsem, P. Sutra, and G. Thomas, "J-nvm: Off-heap persistent objects in java," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 408–423. [Online]. Available: https://doi.org/10.1145/3477132.3483579

[13] A. Raad, J. Wickerson, G. Neiger, and V. Vafeiadis, "Persistency semantics of the intel-x86 architecture," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, dec 2019. [Online]. Available: https://doi.org/10.1145/3371079

[14] P. Lantz, S. Dulloor, S. Kumar, R. Sankaran, and J. Jackson, "Yat: A validation framework for persistent memory software," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 433–438. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/lantz

[15] H. Gorjiara, G. H. Xu, and B. Demsky, "Jaaru: Efficiently model checking persistent memory programs," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS

'21. New York, NY, USA: Association for Computing Machinery, 2021, p. 415–428. [Online]. Available: https://doi.org/10.1145/3445814.3446735

[16] ——, "Yashme: detecting persistency races," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds. ACM, 2022, pp. 830–845. [Online]. Available: https://doi.org/10.1145/3503222.3507766

[17] X. Fu, W.-H. Kim, A. P. Shreepathi, M. Ismail, S. Wadkar, D. Lee, and C. Min, "Witcher: Systematic crash consistency testing for non-volatile memory key-value stores," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 100–115. [Online]. Available: https://doi.org/10.1145/3477132.3483556

[18] S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan, "Pmtest: A fast and flexible testing framework for persistent memory programs," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 411–425. [Online]. Available: https://doi.org/10.1145/3297858.3304015

[19] B. Di, J. Liu, H. Chen, and D. Li, "Fast, flexible, and comprehensive bug detection for persistent memory programs," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 503–516. [Online]. Available: https://doi.org/10.1145/3445814.3446744

[20] S. Liu, K. Seemakhupt, Y. Wei, T. Wenisch, A. Kolli, and S. Khan, "Cross-failure bug detection in persistent memory programs," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1187–1202. [Online]. Available: https://doi.org/10.1145/3373376.3378452

[21] I. Neal, B. Reeves, B. Stoler, A. Quinn, Y. Kwon, S. Peter, and B. Kasikci, "AGAMOTTO: How persistent is your persistent memory application?" in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 1047–1064. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/neal

[22] X. Fu, D. Lee, and C. Min, "DURINN: Adversarial memory and thread interleaving for detecting durable linearizability bugs," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 195–211. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/fu

[23] Z. Chen, Y. Hua, Y. Zhang, and L. Ding, "Efficiently detecting concurrency bugs in persistent memory programs," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 873–887. [Online]. Available: https://doi.org/10.1145/3503222.3507755

[24] H. Fu, Z. Wang, X. Chen, and X. Fan, "A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques," *Software Quality Journal*, vol. 26, pp. 855–889, 2018.

[25] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 135–148. [Online]. Available: https://doi.org/10.1145/3037697.3037730

[26] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter, "Hemem: Scalable tiered memory management for big data applications and real nvm," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 392–407.

[27] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Crash consistency," *Commun. ACM*, vol. 58, no. 10, p. 46–51, sep 2015. [Online]. Available: https://doi.org/10.1145/2788401

[28] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 91–104, 2011.

[29] "eADR: New Opportunities for Persistent Memory Applications," 2021. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html

[30] A. Tanenbaum, *Modern operating systems*. Pearson Education, Inc.,, 2009.

[31] K. Sen, "Race directed random testing of concurrent programs," *SIGPLAN Not.*, vol. 43, no. 6, p. 11–21, jun 2008. [Online]. Available: https://doi.org/10.1145/1379022.1375584

[32] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye, "Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 162–180. [Online]. Available: https://doi.org/10.1145/3341301.3359638

[33] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, p. 558–565, jul 1978. [Online]. Available: https://doi.org/10.1145/359545.359563

[34] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. Netzer, "Detecting data races on weak memory systems," *ACM SIGARCH Computer Architecture News*, vol. 19, no. 3, pp. 234–243, 1991.

[35] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai, "Toward integration of data race detection in dsm systems," *Journal of Parallel and Distributed Computing*, vol. 59, no. 2, pp. 180–203, 1999. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731599915745

[36] E. Pozniansky and A. Schuster, "Multirace: efficient on-the-fly data race detection in multithreaded c++ programs," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 3, pp. 327–340, 2007.

[37] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 391–411, 1997.

[38] J. a. Gonçalves, M. Matos, and R. Rodrigues, "Mumak: Efficient and black-box bug detection for persistent memory," in *Proceedings of the Eighteenth European Conference on Computer Systems*, ser. EuroSys '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 734–750. [Online]. Available: https://doi.org/10.1145/3552326.3587447

[39] S. Kalbfleisch, L. Werling, and F. Bellosa, "Vinter: Automatic Non-Volatile memory crash consistency testing for full systems," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 933–950. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/werling

[40] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, 2005, p. 41.

[41] B. Reidys and J. Huang, "Understanding and detecting deep memory persistency bugs in nvm programs with deepmc," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 322–336. [Online]. Available: https://doi.org/10.1145/3503221.3508427

[42] C. Cadar and M. Nowack, "Klee symbolic execution engine in 2019," *International Journal on Software Tools for Technology Transfer*, vol. 23, no. 6, pp. 867–870, Dec 2021. [Online]. Available: https://doi.org/10.1007/s10009-020-00570-3

[43] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[44] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154. [Online]. Available: https://doi.org/10.1145/1807128.1807152