HawkSet: Automatic, Application-Agnostic, and Efficient Concurrent PM Bug Detection

João Oliveira
IST Lisbon & INESC-ID
Portugal

João Gonçalves
IST Lisbon & INESC-ID
Portugal

Miguel Matos
IST Lisbon & INESC-ID
Portugal

Abstract

Persistent Memory (PM) enables the development of fast, persistent applications without employing costly HDD/SSD-based I/O operations. Since caches are volatile and CPUs may reorder and stall memory accesses for performance, developers must use low-level instructions to ensure a consistent state in case of a crash. Failure to do so can result in data corruption, data loss, or undefined behavior. In concurrent executions, this exposes a new class of bugs.

HawkSet is an automatic, application-agnostic, and efficient tool to detect concurrent PM bugs. HawkSet uses lockset analysis, and automatic binary instrumentation to find all the bugs detected by the state-of-the-art tools and 7 previously unknown bugs. This is achieved without requiring application-specific knowledge or models, nor specialized debugging artifacts or guided executions. Compared to the state-of-the-art, HawkSet offers up to a 159× speedup, and consistently detects harder-to-reach bugs, where a rare interleaving is required.

CCS Concepts: • **Hardware → Emerging technologies**; • **Software and its engineering → Software testing and debugging**.

Keywords: Persistent Memory, Concurrency, Bug Detection

1 Introduction

Persistent Memory (PM) provides developers with byte-addressable and durable storage with performance comparable to that of DRAM [39]. Initially introduced to the general market by Intel Optane, which has since be discontinued in favor of the up-and-coming CXL 3 standard [6], PM offers an appealing combination of performance and durability enabling applications to maintain a persistent state without incurring in costly HDD/SSD-based IO operations. This sparked novel designs in domains such as key-value stores [19, 35], hash tables [24, 32, 41], trees [20, 22, 28, 37], caches [40], file systems [9, 42], databases [5] and programming frameworks [25, 38].

Achieving both correctness and high performance in a PM application is challenging. In particular, since the cache is volatile and orders of magnitude faster than PM, stores are not immediately persisted and the hardware may reorder or stall memory accesses for performance. Hence, developers must carefully use low-level flush and fence instructions to ensure persistency and ordering guarantees.

Failure to meet these requirements can have catastrophic consequences, including data loss or corruption, subsequent crashes, or undefined behavior. To ameliorate these issues, the research community has developed a wide range of PM bug detection tools [7, 14, 16–18, 23, 26, 27, 33]. While some of these tools may detect bugs in concurrent PM programs, none actively search for them.

The reason is that the combination of PM and concurrency introduces a novel class of bugs, unique to concurrent PM programs, that requires specialized tools [2, 15] to be detected. We call this novel class of bugs *persistency-induced races* and precisely define them in §2.2. As with other forms of concurrency bugs, detecting *persistency-induced races* is challenging due to the large search space. Existing tools tackle this challenge by leveraging application-specific semantics, namely a key-value store interface, and guiding the execution to points more likely to expose bugs [2, 15]. While these approaches successfully detect *persistency-induced races*, they lack generality and cannot be applied to applications other than ones with key-value stores semantics. Moreover, they require observing specific interleavings to detect a *persistency-induced race* which, in turn, requires controlling or guiding the execution schedule, resulting in long testing times and poor efficiency.

This paper introduces HawkSet, an automatic, application-agnostic, and efficient tool to detect *persistency-induced races*. HawkSet eliminates the need to directly observe a race by building upon lockset analysis [34]. Lockset analysis tracks the set of locks that protect a given memory region. If different threads access the same region with disjoint locksets, then the region is unprotected, and it can *potentially* be accessed concurrently, even if such an access is not observed in a concrete execution. However, lockset analysis poses some challenges as it is known to scale poorly with the number of memory accesses, cannot be directly applicable to PM programs due to PM-specific semantics, and might yield False Positives. To address the scalability challenge, we observe that PM accesses are a small fraction of all memory accesses [31], making lockset analysis suitable for PM programs. We tackle the second challenge by introducing the concept of *effective lockset*, which discerns PM's distinctive

phases for store visibility and persistence, not captured by traditional lockset analysis. Finally, we propose a heuristic to vastly reduce the number of False Positives.

HawkSet automatically instruments the application binary to identify PM accesses and concurrency primitives, and does not require specialized client drivers or debugging artifacts [2, 15]. When combined with the ability to detect *persistency-induced races* even if those are not directly observed in a given execution, it reduces developer effort both when preparing the application for testing and during testing itself. Overall, HawkSet detected all *persistency-induced races* reported by PMRace [2], a state-of-the-art concurrent PM bug detection tool, in a fraction of the time. In fact, for a particular set of workloads, HawkSet's average time to find a specific race in `Fast-Fair` [20] is ≈159× smaller than PMRace's. Since HawkSet does not rely on direct observation for *persistency-induced race* detection, it consistently detects bugs, even hard-to-reach ones, in a single execution (provided with enough coverage), without the need for multiple executions, greatly reducing testing time.

Contributions. This paper makes the following contributions:

- The design of a PM-Aware Lockset Analysis algorithm for *persistency-induced race* detection.
- The design and implementation of HawkSet, an automatic, application-agnostic, and efficient concurrent PM bug detection tool.
- An empirical evaluation showcasing HawkSet's scalability and effectiveness in large workloads.
- The detection of seven new bugs.

2 Background

This section provides background on Persistent Memory semantics (§2.1) and defines *persistency-induced races* (§2.2).

2.1 Persistent Memory Semantics

When programming with PM, developers must be aware of store visibility, persistency and ordering. We say that stores are persisted, if they are in the persistent domain, which for the purpose of this paper means they are in PM. Persisted stores are guaranteed to be visible to the post-crash execution. Everything else is in the volatile domain (usually consisting of the CPU registers, the store buffer, the cache, and volatile memory) and can be lost after a crash (e.g.: a power failure). This model brings forth three problems that, in conjunction, can cause PM bugs.

Persistency. The existence of a volatile cache between the CPU and PM means that a store may have been executed, but the data remains in the cache, meaning it will be lost after a crash. To counteract this, CPUs offer special `flush` instructions that force the data in a cache line to be written to PM. It is important to note that data may be arbitrarily flushed to PM by the cache-policy algorithm, but the programmer generally has no control over this.

Stalling and Reordering. Modern CPUs include a store buffer, an optimization that amortizes store latency by stalling and reordering memory operations, including the aforementioned store and flush families of instructions. This means, that even if a flush instruction has been executed, there is no guarantee that the data is persisted, since it can be stalled, while the application resumes execution. The `fence` family of instructions guarantee that all pending memory operations have been completed, before resuming the execution. Note that even though fences are used in many implementations of synchronization primitives to ensure that data is visible to other threads, in the context of PM, to guarantee persistency a flush must be followed by a fence. The exception is non-temporal stores which bypass the cache and do not need to be flushed but require a fence to ensure order.

Visibility. As previously mentioned, data is considered persistent only when it reaches PM, but it is already visible to other threads while in the cache. As we will see next, this distinction is the root cause of *persistency-induced races*.

Note on eADR and CXL. Some Intel processors have a feature called extended Asynchronous DRAM Refresh (eADR) that extends the persistent domain to the cache [10], effectively ensuring all stores are persisted once they reach the cache. However, it is crucial to note that eADR requires additional hardware support which is not widely available and hence developers cannot make assumptions about eADR ubiquity across all platforms. Consequently, applications should not depend on the availability of eADR and need to be designed with this consideration in mind.

Additionally, the recent Compute Express Link v3 (CXL) [6] aims to provide PM support. Given the large performance gap between the volatile and persistent domains, it is anticipated that CXL will provide familiar PM semantics, including explicit instructions for managing the transition from the volatile to the persistent domain. While our current implementation is based on the PM semantics of Intel's Optane DC memory, which until recently was the predominant PM technology available, we are confident that our techniques, designed to be agnostic of application semantics and libraries, will translate well to CXL.

2.2 Persistency-Induced Races

A *persistency-induced race* can happen only due to the combination of concurrency and PM and is different from classical concurrency bugs and non-concurrent PM bugs. To illustrate this, let us consider the examples in Figure 1. Figure 1a shows a program with two threads `T1` and `T2` that concurrently access a shared variable `X`. Both accesses are protected by the same lock `A`, and hence this program is correct from a concurrency perspective. Figure 1b illustrates a PM application that

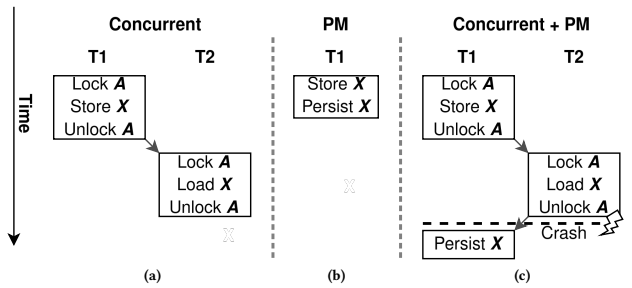


Figure 1. Concurrency and PM. `Lock` and `Unlock` denote synchronization primitives. `Store`, `Load`, `Persist` represent different PM accesses. The arrows correspond to a thread interleaving. `X` is a PM variable and `A` is a mutex.

correctly stores and persists variable `X`. Both programs are correct within their respective contexts. However, a naive combination of PM and concurrency can result in incorrect programs, as illustrated in Figure 1c. In this example, the accesses in both threads to the shared variable `X` are protected by the same lock `A`, however, the persistency is outside the critical section. If the application crashes after the load in thread `T2` and before `X` is persisted in thread `T1`, side effects from the load might be observable in the post-crash state while the store might not be observable. The side effect could be, for instance, a reply to a client that saw the most recent value of the store which is lost after the crash, or the load could cause some modification to the state based on the value which is lost after the crash, resulting in an inconsistent state. This incorrect behavior typifies a *persistency-induced race*, which we define as:

Definition 1. ***Persistency-induced race.*** *If a thread* `T2` *loads a value modified by another thread* `T1` *and that value is not guaranteed to be persisted at the time of the access by* `T2`*, then we are in the presence of a* persistency-induced race.

This definition captures the notion that values can be visible to other threads but not yet persisted, i.e. they are in the CPU cache and hence are accessible to other threads, but they are not yet in PM. Note that our definition of *persistency-induced race* aligns with the notions of *visible-but-not-durable* of Durinn [15] and *PM Inter-thread Inconsistency* of PMRace [2]. Note that despite the similar name, *persistency-induced race* is different from the *persistency race* described in Yashme [18]. The latter is unrelated to concurrency and describes issues that arise due to compiler induced store tearing.

3 HawkSet

In this section, we present HawkSet's design. We start by discussing the challenges of applying lockset analysis to PM and then iteratively construct our PM-Aware Lockset Analysis algorithm to solve these challenges (§3.1). Last, we detail how HawkSet's pipeline achieves our goals of efficiency, automatism and application-agnosticism (§3.2).

3.1 Lockset Analysis

Due to the large search space, it is crucial to detect *persistency-induced races* in an efficient and timely manner.

Lockset analysis partially addresses this challenge since it allows the detection of a race even if that race did not manifest itself in an observed interleaving. This substantially reduces the need to perform multiple executions of the application, as we show in the evaluation (§5.2), and avoids resorting to execution serialization techniques which lead to long testing times. However, lockset analysis is not without its shortcomings. First, it is known to scale poorly with the number of memory accesses. Second, it treats regular memory accesses and PM accesses indiscriminately and ignores the persistency semantics. Finally, lockset analysis is also known for yielding a non-negligible amount of False Positives, which can impair its usability.

The key observation to tackle the scalability challenge is that the ratio of PM accesses is very small ($\approx$ 4% [31]) when compared to the total memory accesses, which allows an otherwise expensive technique to scale well in PM programs. In the rest of this section we address the remaining challenges of traditional lockset analysis, following a constructive approach, and build a PM-Aware Lockset Analysis algorithm that is efficient and scalable to large workloads.

3.1.1 Traditional Lockset Analysis.

A lockset is the set of all locks held at a given point in a thread's execution. The lockset of a memory access is the set of all locks held by the issuing thread when the access takes place. The intersection of the locksets of two memory accesses by different threads contains all the locks that were held when both accesses were executed. If the intersection of two locksets is the empty set, it implies that no common lock protects both accesses and hence they can occur concurrently.

A typical lockset analysis algorithm [34] starts by intersecting the lockset of each store with the lockset of each load to the same memory region and reports a race if the resulting set is empty. While conceptually simple, this algorithm cannot be directly applied to PM since it does not consider the persistency operations. Let us revisit Figure 1c to see why. In this example, the intersection of the locksets for the load and store of `X` is lock `A` and therefore both accesses are correctly synchronized. However, according to the definition of a *persistency-induced race* (Definition 1), thread `T2` loads a value that is not guaranteed to be persisted, and therefore we are in the presence of a *persistency-induced race*. Hence, a naive approach is not suitable for detecting *persistency-induced races*.

Before proceeding, it is important to note that although traditional lockset analysis algorithms can and do check for store-store races, HawkSet does not. This is because,

persistency-induced races result in bugs due to the causal dependency between the side effect of the load, and the original store that led to that effect but which can be lost. This dependency does not happen between two stores and therefore store-store races cannot lead to *persistency-induced races.*

3.1.2 PM-Aware Lockset Analysis. The first step towards correctly detecting *persistency-induced races* is to take into account the lockset for the full lifetime of the unpersisted value, instead of just the moment it is stored. This starts at the store instruction that makes the data visible, and lasts until the persistency, or the point where it is overwritten by another store. To capture this, we employ an expanded version of the lockset, the *effective lockset* defined as: the effective lockset of a store X by a thread T1 is given by the intersection between the lockset of the store with the lockset of its explicit persistency via fence, or overwrite via store. Intuitively, the effective lockset represents the set of locks that protect both instructions. Figure 2a illustrates this concept by showing the locksets observed by thread *T1* from Figure 1c — the store operation is protected by lock A, however, the persistency is not. As we saw earlier, traditional lockset analysis does not account for this persistency. By computing the effective lockset, we can assert that a *persistency-induced race* is possible since the effective lockset is empty and hence a thread performing the load could do so concurrently with the thread performing the store and the persistency.

A naive implementation of the effective lockset can still miss *persistency-induced races*, as illustrated in Figure 2b. In this example, the effective lockset is not empty, but we can observe that lock A does not create an atomic section encompassing the store and the persistency. If the lock is released and reacquired between the store and the persistency, there is room for a racy access that would remain undetected.

We address this problem by extending the lockset with a timestamp given by a thread-local logical clock that is incremented whenever a lock is acquired. This simple yet effective approach allows us to determine whether the store and persistency belong to the same atomic section. Figure 2d depicts this in action. As before, both the store and the persistency are protected by lock A, however, because the lock is released and reacquired between both operations, the logical timestamp is increased. Because the effective lockset now considers both the set of locks and the timestamps when computing the intersection, the result is the empty effective lockset which means a *persistency-induced race* can happen. Figure 2c is the extended version of Figure 2a.

To detect *persistency-induced races* we compute the intersection of the effective lockset among the different threads that access a given PM location. When computing the inter-thread intersection, the timestamp of the effective lockset is ignored since it is only meaningful in the thread-local context. If this intersection is the empty set we report a *persistency-induced race.*

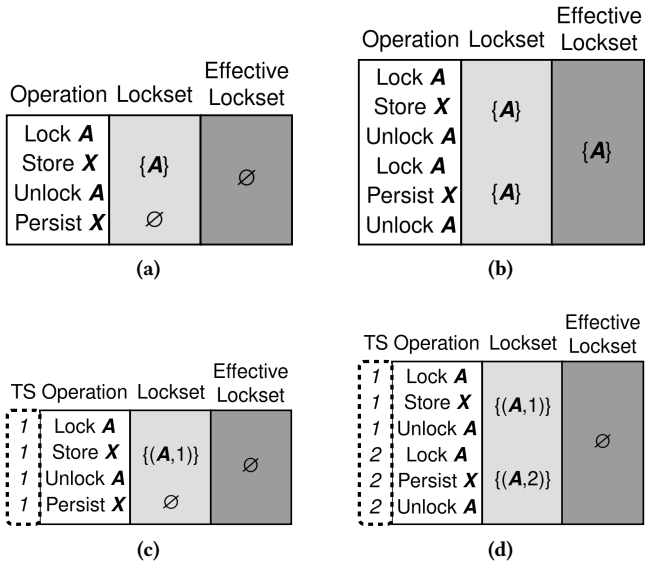


Figure 2. HawkSet's effective lockset. The TS column represents the logical clock at each point of the execution. The Operation column depicts the program's PM and synchronization instructions. The Lockset column represents the lockset assigned to each PM access while the Effective Lockset column denotes the effective lockset for the store+persistency.

Inter-Thread Happens-Before Analysis. To eliminate, from the lockset analysis, PM accesses that can never happen concurrently, HawkSet leverages the happens-before relationship between thread operations. This has been exploited in past works in general race detectors [21], here we present our adaption to the PM domain.

Before presenting our approach, let us first analyze the motivating example depicted in Figure 3 consisting of three different threads. It is easy to see that the store+persistency to X performed by thread T1, even though it is not protected by any lock, can never be concurrent with the accesses done to X by thread T2 and thread T3. This is because there is an implicit *inter-thread happens-before* relationship between the access done by thread T1, and the subsequent creation of the other threads. An analysis oblivious to this relationship would incorrectly report a concurrent access between T1 and the accesses done by both thread T2 and thread T3, resulting in a False Positive. Following the same rationale, there is no implicit *inter-thread happens-before* relationship between the accesses done by thread T2 and thread T3, therefore they can run concurrently.

We express this *inter-thread happens-before* relationship through vector clocks with a logical counter per thread [12]. Each thread maintains its own local vector clock which is

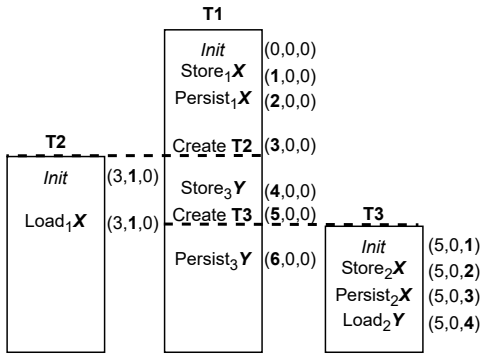


Figure 3. Multithreaded execution. Each column represents a different thread. Dashed lines represent thread creation. Assume `X` and `Y` are PM backed variables that fall on separate cache-lines. Tuples represent the vector-clock associated with each operation. Bold indexes in vector clocks represent updates.

updated as follows: i) thread creation increments the counter of the parent thread by one, sets the vector clock of the child to that of the parent and further increases the child's counter by one; ii) PM accesses increment the counter of the thread in which they occur by one; iii) thread join updates the vector clock of the waiting thread with that of the joining thread following standard vector clock procedures [12].

With this in place, we can now determine whether two accesses are concurrent by comparing the vector clocks of their respective threads at the time of the access. As a reminder, 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]`.

Revisiting Figure 3, thread `T1` starts the execution with an initially zeroed vector clock, which is updated to `(3,0,0)` after the creation of thread `T2`. Upon starting, thread `T2` sets its vector clock to that of its parent, and increases its local counter resulting in a vector clock of `(3,1,0)`. Similarly, when starting thread `T3`, thread `T1` increases its vector clock to `(5,0,0)` and thread `T3` in turn sets its vector clock to `(5,0,1)`. With this information, we can verify that Store_1 in thread `T1` is not concurrent with the loads of either thread `T2` and thread `T3` since there is no index `i` and `j` that satisfies the condition for the respective vector clocks to be concurrent. Hence, we can remove those pairs of accesses from our lockset analysis and eliminate an otherwise False Positive. Similarly, we can see that the vector clocks from thread `T2` and thread `T3` are concurrent, and therefore we need to perform lockset analysis on those accesses.

Note that, for this to work correctly in a PM application, we must take into account the vector clock of the persistency. We can see why this is the case by following Store_3 and Persist_3 done by thread `T1` which have vector clocks of `(4,0,0)` and `(6,0,0)` respectively. By applying the previous method, we learn that Store_3 in thread `T1` cannot execute concurrently with Load_2 on thread `T3`, however, that does not mean a race is impossible. Persistency races can occur up until the moment of persistence, in this case Persist_3, which means that after thread `T3` is created, there is the possibility that it loads the still unpersisted `X` which can lead to a persistency race. By using the vector clock of Persist_3, the analysis correctly predicts that these accesses can execute concurrently, and hence those accesses should be considered for lockset analysis.

3.1.3 Initialization Removal Heuristic.

In concurrent applications, it is a common pattern to initialize freshly allocated variables without holding a lock. This is correct since the memory region has not yet been made public to other threads, therefore, no other thread can access the data concurrently. However, the lockset analysis algorithm would blindly compare these accesses with others, resulting in False Positives. This was first noticed and exploited by Eraser [34], and our evaluation in §5.4 further demonstrates this is also a common pattern in PM applications.

An ideal implementation would determine the point at which the address becomes public to other threads and ignore all the memory accesses done up to that point. However, and to the best of our knowledge, there is no application-agnostic and automatic method to identify this point. In our approach, we approximate it by discarding all explicitly persisted stores to a PM address done by a thread `T1` before a second thread `T2` accesses that address.

To understand why persistency must be taken into account, consider a scenario where a thread `T1` performs an allocation, initializes the memory, and then publishes the pointer without persisting the data. Next, a second thread `T2` might perform a load for that same address which, since it is not guaranteed to be persisted, is a *persistency-induced race*. Using a naive approach, we would exclude the first store performed by `T1`, since it occurred during initialization, missing the race. By excluding only stores that have been explicitly persisted before being published, we eliminate this problem.

3.2 Pipeline

We now present HawkSet's pipeline which is divided into the three stages, shown in Figure 4. We assume that a workload with sufficient coverage is provided.

The first stage, called Instrumentation ①, runs the application with the provided workload and collects a trace of the execution, namely: PM accesses, thread creation and joining, and the synchronization primitives. The second stage applies

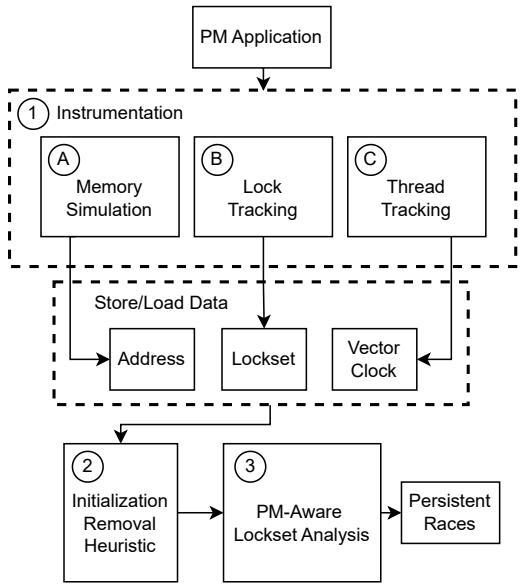


Figure 4. HawkSet's pipeline.

the Initialization Removal Heuristic ② to eliminate accesses that are not visible to other threads. Finally, the Analysis stage ③ applies our PM-Aware Lockset Analysis algorithm.

1st Stage: Instrumentation. The Memory Simulation component Ⓐ maintains the state of the cache, simulating the execution of store and persistency (flush and fence) instructions to PM addresses. Because we are interested in knowing the moment when data is *guaranteed* to be persisted, we simulate a worst-case cache that only persists a cache line if a flush and fence are explicitly executed.

The Lock Tracking component Ⓑ tracks lock acquisition and release in order to compute the current lockset. When a lock is acquired, the current thread's lockset is updated to include it, and when one is released, it is removed.

The Thread Tracking component Ⓒ maintains the thread local vector clock that tracks thread creation and joining.

When a PM access occurs, the Memory Simulation registers that information in a tuple composed of: target address, current lockset, current vector clock, and a thread identifier. For PM loads, this data will be directly used in the lockset analysis stage ③. For PM stores the Memory Simulation saves the access in a store lockset and marks that memory region as dirty in cache, and once it is explicitly persisted (via flush and fence instructions) it saves the persistency lockset. Both the store and persistency locksets are then used to compute the effective lockset.

2nd Stage: Initialization Removal Heuristic. This stage eliminates accesses done during the initialization of a memory region. To achieve this, we track which threads have accessed each address. When a second thread accesses an address, we consider that that address has become visible.

Algorithm 1: PM-Aware Lockset Analysis Algorithm. The ‖ operand denotes concurrent accesses.

```
   input: stores - The set of all stores
          loads - The set of all loads
 1 Vec⟨Int⟩ as VectorClock
 2 Set⟨Lock⟩ as Lockset
 3 Struct StoreData
 4     Int tid /* thread ID */
 5     Lockset effective_set
 6     VectorClock vec
 7     Int addr /* PM address */
 8 Struct LoadData
 9     Int tid /* thread ID */
10     Lockset set
11     VectorClock vec
12     Int addr /* PM address */
13 foreach StoreData st ∈ stores do
14     foreach LoadData ld ∈ loads ∧
15     st.addr = ld.addr ∧
16     st.tid ≠ ld.tid ∧
17     st.vec ‖ ld.vec do
18         if st.effective_set ∩ ld.set = ∅ then
19             report(st, ld)
```

All subsequent accesses after this point, and any unpersisted store prior to it, are passed to the next stage.

3rd Stage: PM-Aware Lockset Analysis. Finally, in this stage, we apply our PM-Aware Lockset Analysis algorithm, depicted in Algorithm 1. The algorithm starts by pairing every store (line 13) and load (line 14) to the same address (line 15), from different threads (line 16) which may execute concurrently according to the *inter-thread happens-before* analysis (denoted by ‖ in line 17). Then, using the effective lockset of the store, we compute its intersection with the lockset of the load (line 18). If the intersection is the empty set, we report a *persistency-induced race* (line 19). Note that, for simplicity, the matching of addresses is presented as being based in the starting address. In reality, HawkSet performs more complex calculations that take into account the size of the PM access, and is able to detect partially overlapping races.

3.3 Discussion

Our PM-Aware Lockset Analysis algorithm is able to detect *persistency-induced races* even if they are not directly observed in a concrete execution of the application: it suffices to find a pair of PM accesses to the same address whose lockset intersection is empty. This allows HawkSet to be efficient

since it does not require running the application exhaustively in the hope of directly observing a race. By relying only on binary instrumentation, HawkSet is agnostic of the application semantics and libraries used (further implementation details are provided in §4). It is also automatic since it does not require special annotations, drivers, or dedicated artifacts for conducting the analysis.

The *persistency-induced races* reported by HawkSet can be classified in three categories: i) Malign Persistency-induced races: genuine races that consist of pairs of PM accesses that can execute concurrently and can cause an unexpected behavior in the application; ii) Benign Persistency-induced races: genuine races that consist of pairs of PM accesses that can execute concurrently, but which can be tolerated by the application's design. The mechanism that tolerates a race is application-specific and cannot be detected by application-agnostic approaches such as HawkSet; iii) False Positives: pairs of PM accesses that can never execute concurrently.

In the evaluation (§5.4) we study the distribution of the *persistency-induced races* detected by HawkSet across these categories, and discuss the impact of the Initialization Removal Heuristic in this distribution.

4 Implementation

HawkSet is implemented in C++ in ≈ 2600 lines of code and uses Intel PIN [29] for binary instrumentation. In detail, we instrument the PM instructions (stores, loads, flushes, fences), atomic instructions and synchronization primitives (such as lock prefixes, and CAS), as well as functions responsible for thread creation and joining. To differentiate between PM accesses from regular memory accesses we record calls to `mmap` for PM files, and then compare the target address of the operation with the previously recorded PM regions. For simplicity, we require the paths to the PM files paths to be passed as command-line arguments, but this could be extended to capture the opening of files in the PM device. Regarding the synchronization primitives, we provide builtin support for `pthread`. Other primitives can be supported through a simple configuration file.

We believe this simple configuration does not break our initial goal of automation since: i) HawkSet only requires simple information about the synchronization primitives, namely, the name of functions that have acquire-and-release semantics, and in the case of tentative acquires (similar to `pthread_mutex_trylock`) the value specifying a successful acquisition; ii) the set of primitives is usually very small; iii) many applications already use `pthreads` or abstractions built on top of `pthread` primitives; and iv) once a configuration file is created for a synchronization library it can be reused by other applications using the same library.

The implementation roughly follows the pipeline depicted in Figure 4 except for the Initialization Removal Heuristic ② which is applied alongside the Instrumentation stage ①.

To help developers pinpoint the cause of *persistency-induced races*, HawkSet provides the backtraces of the PM accesses that caused the race. PIN provides a built-in method to obtain detailed backtraces, `PIN_Backtrace`, but we experimentally observed that this method is prohibitively expensive, accounting for an overhead of up to 90% in earlier versions of HawkSet. To overcome this, we instrument call and return instructions to create the call stack. This method is substantially faster than the one provided by Intel PIN, with minimal loss of information. Over the course of our experimentation, this improved approach has proved to be robust enough for our use case, with minor loss of trace information compared to `PIN_Backtrace`.

The PM-Aware Lockset Analysis is described in Algorithm 1 for clarity rather than performance, and hence it is obviously inefficient and performs several unnecessary iterations. In the implementation, we group PM accesses by thread id and address, and short-circuit several iterations based on empty or equal locksets.

We also implemented several techniques to reduce testing time and memory usage. Locksets and vector clocks are shared across PM accesses since we found experimentally that the number of accesses far outnumbers the amount of locksets and vector clocks, by several orders of magnitude. Furthermore, vector clocks are not incremented at every PM access. Instead, only the first PM access after a thread creation/joining increments the local vector clock. Logically, we batch all sequential PM accesses done by a single thread as one when it comes to the inter-thread happens-before analysis, which drastically reduces management overhead. Since pairs of accesses from the same thread are ignored by the lockset analysis, this optimization does not impact correctness. Moreover, backtraces, locksets, and vector clocks are unique and identifiable by a unique integer, which allows for several optimization opportunities throughout the code, such as direct comparison (i.e: for lockset analysis), fast hashing (i.e: for hashtable lookups), and memory usage.

5 Evaluation

In this section, we experimentally answer the following questions: i) how effective is HawkSet in detecting *persistency-induced races* (§5.1), ii) how does HawkSet compare with the state-of-the-art (§5.2), iii) how efficient is HawkSet (§5.3). iv) what is the impact of the Initialization Removal Heuristic (§5.4), and v) how HawkSet fares in achieving its automation and application-agnosticism (§5.5) goals. All the experiments were performed in a machine with two Intel(R) Xeon(R) Gold 6338N CPUs @ 2.20GHz, comprising 128 cores, 256 GB of RAM, on top of a 1 TB Intel DCPMM in App Direct mode.

Target Applications. We evaluated a total of 9 PM applications, outlined in Table 1. In §5.5 we further discuss the criteria for selecting these applications. To contextualize our experimental results, we briefly describe each application.

Table 1. PM applications tested.

Application	Version	Synchronization Method	Supported by	
			Durinn	PMRace
Fast-Fair [20]	0f047e8	Lock/Lock-Free	Yes	Yes
TurboHash [41]	2d7d7b3	Lock/Lock-Free	Yes	Yes
P-CLHT [24]	70bf21c	Lock	Yes	Yes
P-Masstree [24]	5b4cf3e	Lock/Lock-Free	Yes	Yes
P-ART [24]	5b4cf3e	Lock/Lock-Free	Yes	Yes
MadFS [42]	7514a78	Lock-Free	No	No
Memcached-pmem [5]	0208b53	Lock-Free	Yes	Yes
WIPE [37]	029cfc4	Lock	Yes	Yes
APEX [28]	5aee22a	Lock	Yes	Yes

`Fast-Fair` [20] 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` mixes lock-based concurrency control with lock-free methods to synchronize its shared PM accesses. It has multiple implementations, for the purpose of our analysis, we used the concurrent version that uses real PM, supported by Intel's PMDK [4]. `TurboHash` [41] and `P-CLHT` [24] are PM-backed hash tables. `TurboHash` is a typical hash table focused on improving performance in a few areas: performs efficient out-of-place updates, 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 Intel's PMDK. `P-CLHT` restricts the size of each bucket to that of a cache line, uses bucket-specific locks to synchronize insertions and updates, and a global lock for rehashing. Get operations occur in a lock-free manner. Similarly to `Fast-Fair`, `P-Masstree` is a trie-like concatenation of B+Tree nodes backed by PM. It performs put, scan and delete operations using locks while get operations are lock-free. `P-ART` is a crash-consistent radix tree that varies the sizes of its nodes adapting its memory footprint to the provided workload. To enable comparison of results, we use the custom versions of `P-Masstree` and `P-ART` provided by Durinn [15]. `MadFS` [42] is a file system backed by PM. 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 file metadata in user space, in the form of a log that is updated atomically. `Memcached-Pmem` [5] is a PM enabled fork of the well known Memcached [30], an in-memory key-value store. `WIPE` [37] and `APEX` [28] are learned indexes, data structures that use machine learning for data retrieval. `APEX` is a PM and concurrency enabled extension of Microsoft's ALEX [8].

Workloads. Due to the variety of the applications evaluated, we could not use a single benchmark. All experiments were run with eight threads and consisted of an initial load phase followed by 1k, 10k, or 100k operations of various types depending on the application. The only exception is `P-ART`, which hangs for workloads larger than 1k operations. All presented results are the average of five executions.

For `P-CLHT`, `Fast-Fair`, `TurboHash`, `P-Masstree`, `P-ART`, `WIPE`, and `APEX`, the workloads were generated using YCSB [3] with a load phase of 1k insertions, and a main phase with 30% insertions, 30% updates, 30% gets, and 10% deletes. Note that, in the case of `Fast-Fair`, `TurboHash`, and `P-Masstree` inserts and updates are treated as the same operation. `MadFS`'s benchmark performs 4kb write operations in a shared file amongst all threads. The target offset of the operation is randomized following a zipfian distribution. `Memcached-pmem`'s benchmark performs an initial load phase of 1000 set operations, followed by a range of set, get, add, replace, append, prepend, CAS, delete, increment, and decrement operations over a zipfian workload. HawkSet requires code coverage to provide meaningful reports, we consider the generation of workloads that provide this coverage an orthogonal problem. While generating the previous workloads, we aimed to exercise all available operations for each system.

5.1 Persistency-Induced Races

HawkSet detected 20 *persistency-induced races*, 7 of which were previously unknown and have been reported to the developers. Namely, it detected one bug in `Fast-Fair`, one in `P-CLHT`, and 6 in `Memcached-pmem` that were previously reported by PMRace [2] while another bug in `Fast-Fair`, one in `TurboHash`, three in `WIPE`, and two in `APEX` were previously unknown. Furthermore, we detect three bugs in `P-Masstree`, and two in `P-ART` that we believe have been detected by Durinn [15]. However, since Durinn reports bugs by the key-value store operation, instead of the PM access, we cannot confidently claim that these are the same races. The races reported by HawkSet fall on the same operations of those reported by Durinn, which might indicate that they are the same operation. Table 2 summarizes the results.

Overall, HawkSet detected all *persistency-induced races* reported by the state-of-the-art for the applications evaluated. Note that even though Durinn analyzed `Fast-Fair` and `P-CLHT`, those are not the reference implementations but rather custom modified versions where Durrin's authors manually added PM support to the original, non-PM, implementations.

The known bug detected in `Fast-Fair` (#1) occurs whenever the tree grows which implies creating a new node and inserting it in the tree. While performing this operation, another thread can perform an insertion on the new node, whose pointer is not yet persisted. In case of a crash, the inserted value is lost, resulting in an inconsistent state. The new bug detected in `Fast-Fair` (#2), shown in Figure 5, has a similar pattern but occurs in a program branch that is much less likely to be executed. Crucially, this highlights the importance of detecting *persistency-induced races* even when they are not directly observable and supporting large

Table 2. Persistency-induced races detected using HawkSet. The * denotes races in the operations reported by Durinn.

Application	#	New	Store Access	Load Access	Description
Fast-Fair	1	×	btree.h:560	btree.h:878	load unpersisted pointer
	2	✓	btree.h:571	btree.h:878	load unpersisted pointer
TurboHash	3	✓	turbo_hash_pmem_pmdk.h:2238	turbo_hash_pmem_pmdk.h:2546	load unpersisted value
P-CLHT	4	×	clht_lb_res.c:785	clht_lb_res.c:431	load unpersisted pointer
P-Masstree	5	*	masstree.h:822	masstree.h:1883	load unpersisted value
	6	*	masstree.h:1387	masstree.h:1883	load unpersisted value
	7	*	masstree.h:1425	masstree.h:1953	unpersisted removal
P-ART	8	*	N4.cpp:22,N16.cpp:13,N256.cpp:17	N4.cpp:56,N16.cpp:61,N256.cpp:39	load unpersisted value
	9	*	N4.cpp:67,N16.cpp:76	N4.cpp:56,N16.cpp:61	load unpersisted value
Memcached-pmem	10	×	memcached.c:4292	memcached.c:2805	load unpersisted value
	11	×	memcached.c:4293	memcached.c:2805	load unpersisted value
	12	×	items.c:423	items.c:464	load unpersisted value
	13	×	slabs.c:549	slabs.c:412	load unpersisted pointer
	14	×	items.c:1096	memcached.c:2824	load unpersisted metadata
	15	×	items.c:627	items.c:623	load unpersisted metadata
WIPE	16	✓	pointer_bentry.h:1771,1799	pointer_bentry.h:1606	load unpersisted key
	17	✓	pointer_bentry.h:1550,1772	pointer_bentry.h:1601	load unpersisted value
	18	✓	letree.h:393	letree.h:228	load unpersisted pointer
APEX	19	✓	apex_nodes.h:3479,3798	apex_nodes.h:2915,2933	load unpersisted value
	20	✓	apex_nodes.h:3480,3606	apex_nodes.h:962	load unpersisted key

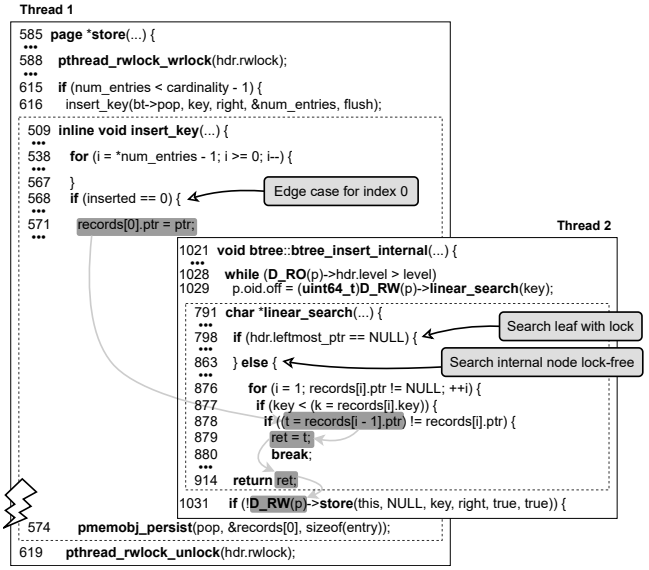


Figure 5. The new bug found in `Fast-Fair`.

workloads that explore uncommon code paths. We further study this in §5.2.

The new bug detected in `TurboHash` (#3) occurs due to a missing persistency in the insertion operation. When an operation inserts an entry into the bucket, it performs some metadata manipulation, and flushes said metadata. When the entry is towards the end of the bucket, such that it falls on a different cache line, the update is not persisted. 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. We stress that this bug can only be detected as the buckets start to fill, which again illustrates the criticality of using large workloads and an efficient testing tool. As a matter of fact, this bug manifested only in the largest workload we tested (100k operations).

The `P-CLHT` bug (#4) is caused by a rehashing operation, which allocates a new hash table and swaps the root pointer. Before the root pointer is persisted, a different thread inserts a new entry into the new hash table. If the application crashes after the insertion, but before the rehashing operation fully completes, then the data inserted by the new thread is lost.

`P-Masstree` and `P-ART`'s bugs result from each application's lock-free get operations. Bugs #5, #6, and #8 occur when a get operation reads a yet unpersisted insertion and bug #7 and #9 occur when the get operation fails to find a deleted key, whose deletion has not been fully persisted.

`Memcached-pmem`'s bugs #10 and #11 occur when a new node is created from an old, unpersisted node. Bugs #12, #13, #14, #15 are caused by missing persistence. Since these bugs have all previously been reported by PMRace [2], in the interest of space, a more in-depth explanation can be found in that paper.

We found several *persistency-induced races* in `MadFS` (further detailed in Table 4) but after further inspection we concluded that these are tolerated by the relaxed guarantees of `MadFS` which contrasts with those of the other applications. MadFS is a file system, and as such has different concurrency and crash-consistency guarantees (such as requiring an explicit `fsync`) than the other systems. However, we believe our reporting is still relevant, because we show that HawkSet

Table 3. Comparison with PMRace using 240 seeds.

	Bug	Executions	Racy Executions	Avg. Time per Execution (s)	Avg. Time to Race (s)
PMRace	#1	240	9	600.00	69900.00
HawkSet			110	6.65	439.19
PMRace	#2		0	600.00	∞
HawkSet			115	6.65	422.55

is able to detect these races when MadFS is incorrectly used in a crash-consistent application.

WIPE's bugs #16 and #17 occur when the lock-free get operations access unpersisted data. Bug #18 occurs during node expansion. In WIPE, nodes start small, and grow as the data structure fills up. During this node resizing operation, a new, larger, persistent memory buffer is allocated which replaces the older buffer via an atomic pointer swap. Although the data in the buffer is persisted, the pointer itself is not. This means that subsequent modifications to the buffer, for example via a put operation, may be lost in the event of a crash.

Finally, APEX's bug #19 and #20 occur when a search operation races with either the insert, erase or update operations. Although the latter operations are protected via mutex, and correctly persisted, the lock-free search can still observe an unpersisted value.

5.2 Comparison with PMRace

We now compare HawkSet's efficiency in detecting *persistency-induced races* with PMRace [2]. Due to space constraints, we limit the comparison to Fast-Fair. Since HawkSet and PMRace have different designs, below we briefly describe PMRace's approach (more details in §6) and the steps we took towards designing a fair comparison of both tools.

PMRace's approach is divided into two separate stages. The first stage uses fuzzing and specialized delay injection to find *persistency-induced races* (called *PM Inter-thread Inconsistency* by the authors). The second stage uses the recovery program to check whether potential side effects that might affect consistency are resolved during recovery which helps remove False Positives. HawkSet uses lockset analysis to find *persistency-induced races*, however, it does not perform further validations. Due to this difference, we focus our comparison on the effectiveness of the *persistency-induced race* detection mechanism, i.e. PMRace's first stage.

PMRace starts with an initial workload, called the seed, and then executes the application with that workload. Each workload has an average of 400 operations. On subsequent executions, it mutates the workload and executes again. This process continues until it is deemed that the workload is not worth exploring. At this point, a new seed is obtained, and the process starts all over again. This means that conceptually PMRace can execute indefinitely. In practice, the authors cap the execution time and execute the system with a restricted set of seeds. For Fast-Fair there are 240 provided seeds. Note that these seeds are workloads, that are then mutated by the fuzzing engine.

To compare PMRace to HawkSet, we ran Fast-Fair under PMRace and HawkSet with each of the 240 workloads provided. The goal was to determine the effectiveness of each tool by measuring the average time to find a *persistency-induced race*. HawkSet takes 6.65 seconds per workload while PMRace uses the full allotted 10 minutes as suggested by the authors. Table 3 presents the results. For bug #1, PMRace reported the race on 9 of the 240 workloads, while HawkSet reported it on 120 out of 240 workloads. The `avg. time to race` reported in Table 3 captures the expected time, on average, for each tool to find the race if the workloads are selected at random and without replacement.

More precisely:

$$\frac{\sum_{i=0}^{E} \binom{E}{i} \times S \times T \times (i+1)}{\sum_{i=0}^{E} \binom{E}{i} \times S}$$

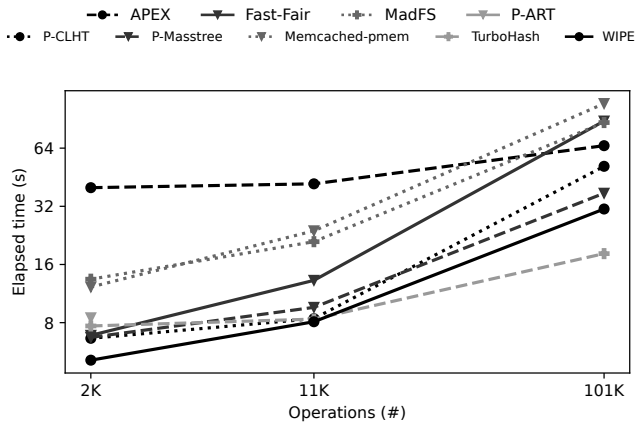
where: E represents the number of workloads where the tool cannot find the race; S represents the number of workloads where the tool finds the race; and T represents the average execution time for one workload. This metric takes into account both the execution time and the ability for each tool to find races in the same workloads. Overall this results in a speedup of 159x compared to PMRace.

Furthermore, the new bug (#2) found in Fast-Fair is reported in 83 out of 240 workloads by HawkSet, while PMRace is unable to detect it. This is because bug #2 is much less likely to occur than bug #1, both require an insertion that leads to a resizing operation splitting a node and inserting the new node in their respective parent, however, bug #2 only occurs when that insertion falls on a specific edge case, shown in Figure 5. By removing the need of directly observing the concrete interleaving, HawkSet is able to detect this *persistency-induced race* reasonably consistently even under a small workload, unlike PMRace, which does not report it. Note that, in the case of HawkSet, the workloads for which no race was reported lack the necessary operations to cause it, therefore, they did not provide coverage to the racy operations.

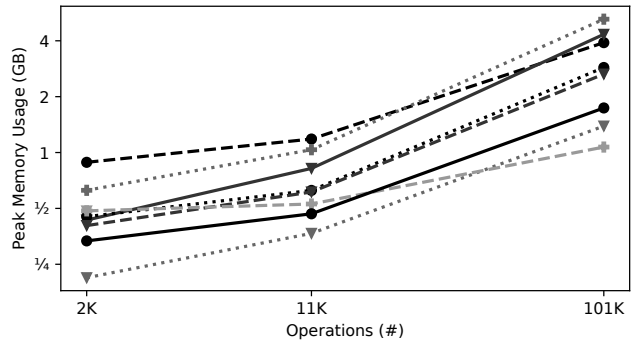
These results show that HawkSet brings substantial efficiency improvements: not only does it find more bugs than existing approaches, it also accomplishes it in a fraction of the time.

5.3 Performance and Cost

We evaluate HawkSet's performance and cost by measuring testing time and peak memory usage for the different systems and workloads. The results are presented in Figure 6, the standard deviation is negligible and therefore not displayed. The testing time, shown in Figure 6a, grows sublinearly with the workload size (note the different logarithmic basis for each axis) taking a little over three minutes for the largest



(a) Time elapsed



(b) Peak memory usage

Figure 6. Breakdown of HawkSet's metrics across all applications. Note that both axes are logarithmic.

experiment with 100k operations. We believe small testing times allow developers to run HawkSet often as part of the development process and hence catch PM concurrency bugs early on, leading to more robust PM software. The peak memory usage, shown in Figure 6b, is relatively small, peaking at around 4GB which shows that HawkSet can efficiently maintain the necessary bookkeeping information even for large workloads.

5.4 Initialization Removal Heuristic

We now study the effectiveness of the Initialization Removal Heuristic (IRH) in removing False Positives. In order to do this, we re-executed all the experiments with the IRH disabled and manually classified the reported races (Malign races, Benign races and False Positives) as discussed in §3.3. The results are presented in Table 4 for the workloads with 100k operations (the results for the other workload sizes follow similar trends and are omitted for brevity). Note that all reports pruned by the IRH were False Positives.

We can draw a few conclusions from the results. For `Fast-Fair`, `MadFS`, `P-Masstree`, and `P-ART` the initialization removal heuristic was able to eliminate all False Positives. For `P-CLHT`, `TurboHash`, `WIPE`, and `ALEX` we can see that the heuristic is able to prune the vast majority of False Positives. Finally, for the case of `Memcached-pmem`, we see that most False Positives have not been pruned by the IRH. We believe this is due to memory reuse, where PM is reallocated and reinitialized safely but without holding a lock. The IRH has previously marked this memory as published to other threads, however, at this moment, it is not. Overall, despite this limitation, the IRH removed a large fraction of False Positives without removing any Malign Persistency-induced races and hence is effective without any noticeable downsides.

Table 4. Breakdown of reports across all applications. Results under "Manual" represent the manual breakdown of *persistency-induced races*, between Malign races (MR), Benign races (BR), and False Positives (FP). Results under "Automatic" represent the reports filtered automatically via the Initialization Removal Heuristic (IRH).

	Manual			Automatic	
Application	MR	BR	FP	After IRH	Reported Races
Fast-Fair	2	21	0	23	53
TurboHash	2	3	1	6	10
P-CLHT	1	6	1	8	40
P-Masstree	3	13	0	16	46
P-ART	11	40	0	51	91
MadFS	0	5	0	5	7
Memcached-pmem	6	37	24	67	68
WIPE	6	19	2	27	47
APEX	10	15	3	28	106

5.5 Automation and Agnosticism

We now discuss the automatic and application-agnostic properties of HawkSet. Given the nature of these properties, our analysis below is necessarily qualitative, not quantitative.

Automatism. HawkSet does not require the creation of any artifacts, code annotations, or changes to the build process.

Evaluating `MadFS`, `Fast-Fair`, `P-Masstree`, `WIPE`, and `Memcached-pmem` required no modifications or artifacts.

`TurboHash` and `P-ART` use custom concurrency control primitives, and hence we had to create a configuration file describing those primitives in order for HawkSet to instrument them. Overall, this process took a few minutes to find and enumerate all concurrency control operations.

`P-CLHT` and `APEX` implement their concurrency control using `CAS` instructions. To instrument these operations, we implemented wrapper functions, and 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` and `APEX`, this process would take much less time since they would be familiar with the source code. It is very important

to note that this is an exception, within the 9 applications evaluated, only `P-CLHT` and `APEX` required this modification.

Application Agnosticism. We selected the target applications used in the evaluation according to the following criteria: The selection of target applications aimed to show HawkSet's broad applicability. We used the following criteria: i) applications which were tested by PMRace and Durinn; ii) wide range of application types, namely various data structures, key-value stores, a database and a file system; iii) recently published PM applications, namely WIPE from 2024, and TurboHash from 2023; iv) applications that use PM frameworks other than Intel's PMDK, such as MadFS.

Overall, we believe HawkSet achieves our automatic and application-agnostic goals and is able to provide developers with a tool for concurrent PM debugging that supports a wide range of PM applications while requiring minimal to no additional effort.

5.6 Discussion

HawkSet is able to detect *persistency-induced races* even if they are not directly observed in a concrete execution thanks to its PM-Aware Lockset Analysis algorithm. This obviates the need of requiring multiple (guided) executions since the conditions in which a race can be detected are much more relaxed. In turn this leads to two major benefits: i) decrease testing times since the number of executions required to find a bug is much smaller than approaches that require a direct observation of the race and/or guided execution, and ii) increased bug detection since it can detect races that happen infrequently. This is supported by our evaluation results where HawkSet detected all the bugs detected by PMRace and Durinn in the applications in common, in a fraction of PMRace's time. HawkSet is effective at finding *persistency-induced races* with small workloads, and it also scales to larger workloads, which increases coverage and allows for detecting harder-to-reach bugs, namely the new bug found in `TurboHash` and `Fast-Fair` which only occur in very specific conditions.

One downside of traditional lockset analysis is the large number of False Positives. The IRH was able to remove all False Positives when analyzing `Fast-Fair`, `MadFS`, and `P-Masstree` while greatly reducing those in `P-CLHT`, without impacting the detection of Malign races.

6 Related Work

In this section, we provide an analysis of the state-of-the-art in concurrent PM bug detection.

6.1 Concurrent Bug Detection

The literature covering the detection of concurrency bugs is vast. Many techniques have been proposed and thoroughly tested throughout the years, such as, but not limited to, delay injection [36], lockset analysis (dynamic [34] and static [11]) or happens-before analysis [1]. For a survey, we refer the interested reader to the work of Fu et al. [13]. It is important to note that none of these techniques can be directly applied to PM programs since they fail to take into account PM semantics, in particular the point of the execution where stores are guaranteed to be persisted.

6.2 PM Bug Detection

Most of the work for detecting bugs in PM programs is oblivious to concurrency [7, 14, 16–18, 23, 26, 27, 33]. Works such as Agamotto [33] rely on symbolic execution models which are known to not support concurrent executions adequately. Other works [7, 14, 16–18, 23, 26, 27] while not actively searching for *persistency-induced races*, might be able to find some bugs provided that: i) the concrete observed interleaving exposes a race, and ii) this race can be reduced to the models they support. Note however that this is highly unlikely and even if such a 'race' is reported, these tools are unable to provide meaningful feedback, as they do not reason about concurrency.

6.3 Concurrent PM Bug Detection

Tools to detect *persistency-induced races* must be aware of the semantics of both PM and concurrency primitives. To the best of our knowledge, PMRace [2] and Durinn [15] are the only tools that fall into this category. Next, we discuss their design in light of three criteria: efficiency, application-agnosticism and automation.

Efficiency. Given that the search space is extremely large and that some *persistency-induced races* might only manifest themselves in a few specific interleavings, the efficiency of the search for *persistency-induced races* is a deciding factor for a tool's testing time. Durinn searches for *persistency-induced races* between key-value store operations. This is achieved by serializing the execution into a single thread and extracting potentially racy operations for further testing. For each pair of operations, Durinn adds breakpoints at the relevant points in an attempt to force the interleaving required to observe the race. While this approach works well for small workloads, it quickly becomes impractical for large workloads. PMRace uses fuzzing campaigns to increase interleaving coverage, combined with delay injection techniques to improve the chance of observing interleavings that constitute a *persistency-induced race*. Still, the search space is very large and PMRace needs to directly observe a persistent race to report it, which might lead to long testing times.

Application-Agnostic. An application-agnostic tool supports any PM application regardless of its concrete semantics and underlying libraries. Durinn's operation-level approach is limited to key-value stores and relies on the concrete semantics of these operations to guide the execution. Hence, it is unclear how it could be extended for other applications. PMRace's fuzzing engine only supports key-value stores or

applications with equivalent semantics, and therefore, similarly to Durinn it is also limited to these types of applications. The fuzzing engine could arguably be replaced with a different one, specialized for different application semantics, but it is unclear how this would affect the overall design of the tool. Furthermore, both PMRace and Durinn assume that the target application is built on top of Intel's PMDK [4] which precludes testing applications that do not use PMDK such as MadFS [42].

Automation. The automation level of a tool determines the amount of work that the developer needs to perform to analyze the application. Durinn [15] requires extending a driver that maps the high-level operations of the target application (such as gets and puts) to the hook points expected by the tool, and a battery of application-specific tests for the analysis. PMRace [2]'s relies on a fuzzing engine which in turn requires modifications to the source code of the application to comply with the fuzzing requirements and, similarly to Durinn, it also requires a driver that maps the application's operations to the internal hook points. Overall, these modifications and extensions are not trivial and require a good understanding of the inner workings of each tool.

7 Limitations

We now discuss two practical limitations of HawkSet and how we expect them to impact developers using the tool.

First, applications using a lock-free mechanism for shared data synchronization yield a substantial number of Benign races. While this is a fundamental limitation of approaches based on lockset analysis, which requires manual inspection to distinguish between Malign and Benign races, we argue that this process aligns well with the expertise of the developers who implement these advanced concurrency control mechanisms in the first place. In practice, these developers should be well-equipped to efficiently classify race conditions based on their deep understanding of the synchronization patterns they employ.

Second, some applications might yield a significant amount of False Positives as we saw in the case of `memcached-pmem`. To the best of our knowledge, this happens when the application reuses PM regions. While HawkSet could theoretically address this by marking newly allocated memory as unpublished, doing so would require instrumenting all the PM allocation primitives used by the application. Unlike synchronization primitives, which have mature and standardized interfaces, PM allocation primitives are still evolving and vary across implementations. We chose not to implement this approach to maintain HawkSet's key strengths namely its application-agnostic nature and automated operation. This design decision allows HawkSet to remain robust and widely applicable across the PM application ecosystem, even as PM allocation interfaces continue to evolve.

8 Conclusion

Current state-of-the-art concurrent PM bug detection tools are limited in scope, focusing primarily on specific application types like key-value stores, or specific libraries and frameworks such as Intel's PMDK. Furthermore, they are not automatic since they require additional complex artifacts for debugging and modifications to the original source code. Finally, they require multiple executions of the target applications which negatively impacts testing time.

In this paper, we introduce HawkSet, an automatic, application-agnostic, and efficient concurrent PM bug detector. HawkSet extends a traditional lockset analysis algorithm to the persistent domain to detect *persistency-induced races*. This is complemented with a heuristic to prune the vast majority of False Positives that commonly plague these techniques. Our results demonstrate that HawkSet is efficient at finding *persistency-induced races*, achieving a speedup of up to $\approx 159\times$ when compared with existing approaches. Furthermore, HawkSet is able to detect hard-to-reach bugs reliably and efficiently. HawkSet characteristics enable developers to integrate it into their development workflow which, we argue, contributes to the creation of more reliable concurrent PM applications.

References

[1] Sarita V Adve, Mark D Hill, Barton P Miller, and Robert HB Netzer. 1991. Detecting data races on weak memory systems. *ACM SIGARCH Computer Architecture News* 19, 3 (1991), 234–243.

[2] Zhangyu Chen, Yu Hua, Yongle Zhang, and Luochangqi Ding. 2022. 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* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 873–887. https://doi.org/10.1145/3503222.3507755

[3] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[4] Intel Corporation. 2024. *Persistent Memory Development Kit.* https://pmem.io/pmdk/

[5] Lenovo Corporation. 2018. Memcached-pmem. https://github.com/lenovo/memcached-pmem.

[6] Peter Desnoyers, Ian Adams, Tyler Estro, Anshul Gandhi, Geoff Kuenning, Mike Mesnier, Carl Waldspurger, Avani Wildani, and Erez Zadok. 2023. Persistent Memory Research in the Post-Optane Era. In *Proceedings of the 1st Workshop on Disruptive Memory Systems* (Koblenz, Germany) *(DIMES '23)*. Association for Computing Machinery, New York, NY, USA, 23–30. https://doi.org/10.1145/3609308.3625268

[7] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. 2021. 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* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 503–516. https://doi.org/10.1145/3445814.3446744

[8] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke,

Donald Kossmann, David Lomet, and Tim Kraska. 2020. *ALEX: An Updatable Adaptive Learned Index*. Technical Report MSR-TR-2020-12. Microsoft. https://www.microsoft.com/en-us/research/publication/msr-alex-techreport/ SIGMOD 2020 | June 2020.

[9] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and Protection in the ZoFS User-Space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 478–493. https://doi.org/10.1145/3341301.3359637

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

[11] Dawson Engler and Ken Ashcraft. 2003. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 237–252. https://doi.org/10.1145/945445.945468

[12] C. J. Fidge. 1988. Partial Orders for Parallel Debugging. *SIGPLAN Not.* 24, 1 (nov 1988), 183–194. https://doi.org/10.1145/69215.69233

[13] Haojie Fu, Zan Wang, Xiang Chen, and Xiangyu Fan. 2018. A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques. *Software Quality Journal* 26 (2018), 855–889.

[14] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. 2021. Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 100–115. https://doi.org/10.1145/3477132.3483556

[15] Xinwei Fu, Dongyoon Lee, and Changwoo Min. 2022. DURINN: Adversarial Memory and Thread Interleaving for Detecting Durable Linearizability Bugs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 195–211. https://www.usenix.org/conference/osdi22/presentation/fu

[16] João Gonçalves, Miguel Matos, and Rodrigo Rodrigues. 2023. Mumak: Efficient and Black-Box Bug Detection for Persistent Memory. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 734–750. https://doi.org/10.1145/3552326.3587447

[17] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2021. Jaaru: Efficiently Model Checking Persistent Memory Programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 415–428. https://doi.org/10.1145/3445814.3446735

[18] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2022. 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*, Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch (Eds.). ACM, 830–845. https://doi.org/10.1145/3503222.3507766

[19] Shukai Han, Dejun Jiang, and Jin Xiong. 2020. 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. https://www.usenix.org/conference/hotstorage20/presentation/han

[20] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. 187–200.

[21] Ali Jannesari, Kaibin Bao, Victor Pankratius, and Walter F Tichy. 2009. Helgrind+: An efficient dynamic race detector. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 1–13.

[22] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 424–439. https://doi.org/10.1145/3477132.3483589

[23] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. 2014. Yat: A Validation Framework for Persistent Memory Software. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 433–438. https://www.usenix.org/conference/atc14/technical-sessions/presentation/lantz

[24] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 462–477. https://doi.org/10.1145/3341301.3359635

[25] Anatole Lefort, Yohan Pipereau, Kwabena Amponsem, Pierre Sutra, and Gaël Thomas. 2021. J-NVM: Off-Heap Persistent Objects in Java. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 408–423. https://doi.org/10.1145/3477132.3483579

[26] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 2020. 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1187–1202. https://doi.org/10.1145/3373376.3378452

[27] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. 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* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 411–425. https://doi.org/10.1145/3297858.3304015

[28] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: a high-performance learned index on persistent memory. *Proc. VLDB Endow.* 15, 3 (Nov. 2021), 597–610. https://doi.org/10.14778/3494124.3494141

[29] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[30] memcached. 2009. Memcached. https://github.com/memcached/memcached.

[31] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. 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* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 135–148. https://doi.org/10.1145/3037697.3037730

[32] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 31–44.

[33] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. 2020. AGAMOTTO: How Persistent

is your Persistent Memory Application?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1047–1064. https://www.usenix.org/conference/osdi20/presentation/neal

[34] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411.

[35] Steve Scargall and Steve Scargall. 2020. pmemkv: A Persistent In-Memory Key-Value Store. *Programming Persistent Memory: A Comprehensive Guide for Developers* (2020), 141–153.

[36] Koushik Sen. 2008. Race Directed Random Testing of Concurrent Programs. *SIGPLAN Not.* 43, 6 (jun 2008), 11–21. https://doi.org/10.1145/1379022.1375584

[37] Zhonghua Wang, Chen Ding, Fengguang Song, Kai Lu, Jiguang Wan, Zhihu Tan, Changsheng Xie, and Guokuan Li. 2024. WIPE: a Write-Optimized Learned Index for Persistent Memory. *ACM Transactions on Architecture and Code Optimization* 21, 2 (2024), 1–25.

[38] Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L. Scott. 2021. A Fast, General System for Buffered Persistent Data Structures. In *Proceedings of the 50th International Conference on Parallel Processing* (Lemont, IL, USA) *(ICPP '21)*. Association for Computing Machinery, New York, NY, USA, Article 73, 11 pages. https://doi.org/10.1145/3472456.3472458

[39] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 169–182.

[40] Jinfeng Yang, Bingzhe Li, and David J. Lilja. 2021. 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* (Haifa, Israel) *(SYSTOR '21)*. Association for Computing Machinery, New York, NY, USA, Article 16, 12 pages. https://doi.org/10.1145/3456727.3463774

[41] Xingsheng Zhao, Chen Zhong, and Song Jiang. 2023. TurboHash: A Hash Table for Key-value Store on Persistent Memory. In *Proceedings of the 16th ACM International Conference on Systems and Storage*. 35–48.

[42] Shawn Zhong, Chenhao Ye, Guanzhou Hu, Suyan Qu, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Michael Swift. 2023. MadFS:Per-File Virtualization for Userspace Persistent Memory Filesystems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 265–280.

Acknowledgments

We thank our shepherd, Horst Schirmeier, and the anonymous reviewers for their feedback and help in improving the paper. This work was supported by Fundação para a Ciência e a Tecnologia (FCT) under plurianual grant UIDB/50021/2020, PhD scholarships 2021.07401.BD and 2024.01891.BD, and research project grant PTDC/CCI-COM/4485/2021 (Ainur). This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No 952226, project BIG (Enhancing the research and innovation potential of Tecnico through blockchain technologies and design Innovation for social Good) where we are researching the use of these techniques and persistent memory technologies to improve the performance and reliability of blockchain systems. In the same vein, this work was also supported by a research grant from Composable Foundation under project ScalableCosmosConsensus.

A Artifact Appendix

This artifact includes HawkSet's implementation, accompanied by the experiments used to evaluate the tool, as seen in §5. Furthermore, it includes the necessary artifacts for the comparison between HawkSet and PMRace.

A.1 Abstract

The HawkSet artifact is divided in two components, i) a HawkSet implementation as described in this paper, and ii) a suite of applications (including the required configuration files, patches, workloads and scripts for its evaluation).

A.2 Description & Requirements

A.2.1 How to access. The artifact is available at https://github.com/Jonyleo/HawkSet-exp and as a DOI at https://doi.org/10.5281/zenodo.14917473.

A.2.2 Hardware dependencies. The evaluation of this artifact depends on the use of a machine equipped with an Intel x86 processor with support for clwb, clflushopt, clflush and sfence instructions.

A.2.3 Software dependencies.

- Linux (Tested for Ubuntu 22.04.5 LTS, kernel version 5.15.0)
- Docker (Tested for version 27.5.1)
- Vagrant (Tested for version 2.4.3)

A.2.4 Benchmarks. None.

A.3 Set-up

To download HawkSet and all experiments, run:

```
git clone https://github.com/Jonyleo/HawkSet-exp
cd HawkSet-exp && git submodule update --init
```

To build HawkSet and all experiments, run:

```
./build_hawkset.sh && ./build_pmrace.sh
```

The full process takes about 1 hour and 15 minutes. To leave the build process running in the background, we suggest using tmux.

A.4 Evaluation workflow[1]

A.4.1 Major Claims.

- *(C1): [Bugs detected] HawkSet detects the bugs described in Table 2. The results should match the table, with the exception that the last column is omitted, and the exact line numbers differ. This is because Table 2 reports line numbers for the original source code of each application, while the line numbers reported by HawkSet's analysis are subject to slight changes in the source code as described in §5.5.*
- *(C2): [Scalability] HawkSet scales for large workloads. The results should match Figure 6's linear trend, while exact values might change due to variation in hardware.*
- *(C3): [False Positives] HawkSet's Initialization Removal Heuristic prunes a vast majority of False Positives commonly associated with lockset analysis. The results should be similar to what is shown in Table 4, except the* `Manual` *column, which cannot be automated as is reported in the paper.*
- *(C4): [Performance] HawkSet achieves a very high speedup compared with PMRace. The results should approach the ones in Table 3, and the overall speedup should reach 159x as reported in § 5.*

A.4.2 Experiments. *Experiment (E1): [Bugs detected] [5 human-minutes + 2.5 compute-hour]: This experiment analyses every application under hawkset, the results are used to generate Table 2, as well as Figure 6, Table 4 in later experiments. We recommend that you run this experiment under tmux.*

[Preparation] None.

[Execution] To run HawkSet, use:

```
./analyze_all_applications.sh
```

[Results] First, launch the `hawkset-exp` container:

```
cd ..
./launch.sh hawkset-exp
cd artifact_evaluation
```

Finally, run the following to display the table:

```
python3 disp_bug_table.py ../output/reports/
```

A table containing all the bugs reported in this paper should be displayed in the terminal.

Experiment (E2): [Scalability] [1 human-minute + 1 compute-minute]: This experiments analyses HawkSet's performance as workload sizes increase. It is used to generate Figure 6. Altough the exact number may vary slightly, the result should be a linear trend between the workload size and each metric (time and memory).

[Preparation] None.

[Execution] None.

[Results] First, if you haven't done so yet, launch the `hawkset-exp` container:

```
cd ..
./launch.sh hawkset-exp
cd artifact_evaluation
```

Finally, run the following to display the table:

```
python3 python3 gen_graphs.py ../output/reports/ ../output/graphs
```

Two figures, analogous to Figure 6 should have been created in ../output/graphs. To view them, you can use scp to copy them into your machine, or mount the remote machine via sftp and access the images from there.

Experiment (E3): [False Positives] [1 human-minute + 1 compute-minute]: This experiment compares the Initialization Removal Heuristic's impact in the reporting of persistency-induced races. It is used to generate Table 4. Note that the `Manual` *column is not generated by the experiment, since, as is described in the paper, this work must be done manually.*

[Preparation] None.

[Execution] None.

[Results] First, if you haven't done so yet, launch the `hawkset-exp` container:

```
cd ..
./launch.sh hawkset-exp
cd artifact_evaluation
```

Finally, run the following to display the table:

```
python3 disp_irh_comparison.py ../output/reports/
```

A table containing the breakdown of reports with and without our Initialization Removal Heuristic should be displayed in the terminal.

Experiment (E4): [Performance] [5 human-minutes + 48 compute-hours]: This experiment compares HawkSet's performance to that of a State-of-the-art tool, PMRace. The key metric evaluated is the average time to race or, in other words, given sequence of random workloads, how long would it take, on average, for each tool to find a specific persistency-induced race. This experiment is used to generate Table 3.

[Preparation] None.

[Execution] First, make sure you have exited the `hawkset-exp` container.

To run the comparison, use:

```
cd artifact_evaluation
./exp_pmrace_comparison.sh
```

This should prompt for a confirmation, ensure you are using tmux, and proceed.

[Results] First, launch the `hawkset-exp` container:

```
cd ..
./launch.sh hawkset-exp
cd artifact_evaluation
```

[1] Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://sysartifacts.github.io/eurosys2025/.

Finally, run the following to display the table:

```
python3 disp_pmrace_comparison.py ffair ../pmrace_results \
../output/pmrace_seeds/
```

The output should resemble Table 3, and the speedup should reach the 159x as reported in the paper. Note that some variation is expected due to the uncertain nature of these tools.

A.5 Notes on Reusability

To evaluate a PM application under HawkSet, a developer must ensure three things:

Synchronization primitives. HawkSet needs to instrument synchronization primitives, such as the ones offered by the pthread library. Our tool does this automatically for pthread, and libpmemobj's primitives. If the application uses custom synchronization primitives, the developers must provide a simple configuration file. Some examples are provided in the `config` directory of the artifact.

Persistent Memory Map. HawkSet detects PM mapped memory via the directory from which that memory was mapped. By default HawkSet looks for the `PM_MOUNT` environment variable to tell it what directory should be considered PM. Make sure to set this variable such that all PM, and only PM, is allocated from files in it. This is usually `/mnt/pmem`.

Coverage. HawkSet is able to detect persistency-induced races only for PM accesses that are executed. It is up to the developers to provide workloads with sufficient coverage.