Vardalith: Hybrid Detection of Persistent Memory Concurrency Bugs

João Gonçalves
IST Lisbon / INESC-ID, Portugal

José Fragoso Santos
IST Lisbon / INESC-ID, Portugal

Rodrigo Rodrigues
IST Lisbon / INESC-ID, Portugal

Miguel Matos
IST Lisbon / INESC-ID, Portugal

Abstract

Persistent Memory offers byte-addressable persistence but exposes developers to new concurrency bugs – *persistency-induced races* – where a thread might read unpersisted data, potentially leading to inconsistencies after crashes. Existing tools face important practical limitations: they either require exhaustive exploration of thread interleavings, depend on application-specific semantics or specialized testing drivers, or report many interleavings that do not correspond to real persistency-induced races. This paper introduces a hybrid approach for detecting persistency-induced races that overcomes these limitations. Our method operates without application-specific knowledge and does not require observing the exact racy interleaving during testing. Instead, by precisely extending the detection window around persistent memory accesses, we can infer the existence of racy interleavings whenever conflicting executions are observed. Our evaluation across multiple applications found 26 bugs (7 new) demonstrating that our approach provides a principled and practical foundation for detecting persistency-induced races.

2012 ACM Subject Classification Hardware → Emerging technologies; Software and its engineering → Software testing and debugging

Keywords and phrases persistent memory, concurrency, crash consistency

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2026.9

Supplementary Material *Software (Source Code)*: `https://github.com/task3r/mumak` [29] archived at `swh:1:dir:8c7962c00482101579ad30f09cfeec6ee0a864bd`
Software (ECOOP 2026 Artifact Evaluation approved artifact): `https://doi.org/10.4230/DARTS.12.1.24`

Funding This work was supported by Fundação para a Ciência e a Tecnologia, I.P. (FCT) under grants 2021.07401.BD, PTDC/CCI-COM/4485/2021 (Ainur), UID/50021/2025 (`https://doi.org/10.54499/UID/50021/2025`), UID/PRR/50021/2025 (`https://doi.org/10.54499/UID/PRR/50021/2025`), 2023.16986.ICDT (`https://doi.org/10.54499/2023.16986.ICDT`), and the bilateral project PLD2.

Acknowledgements We thank the anonymous reviewers for their feedback.

1 Introduction

Persistent Memory (PM) is a recent technology that brings together byte-addressability and data persistence, combined with performance comparable to DRAM. This convergence of properties offers programmers the opportunity to explore innovative designs that can bring substantial performance gains [41, 56, 78, 5, 77, 69, 104, 91, 94, 40, 39, 57, 38, 99, 51]. While the inceptive Intel Optane [15] technology has been discontinued, the core principles of PM continue to gain traction through the emerging CXL standard [12, 16, 87], which has earned

© João Gonçalves, José Fragoso Santos, Rodrigo Rodrigues, and Miguel Matos;
licensed under Creative Commons License CC-BY 4.0

significant interest from both academia and industry in recent years [55, 101, 103, 35, 36]. In particular, CXL Type 3 [87] devices support persistent byte-addressable accesses aligned with existing PM semantics. Furthermore, several new devices with these capabilities are being announced for the near future [82, 63, 70], underscoring the industry demand for PM and its related technologies.

Despite the good performance of PM, CPU caches are orders of magnitude faster. Consequently, and similarly to DRAM, regular stores to PM still go through the cache to optimize performance. However, CPU caches are volatile, and here lies the catch: once a PM store reaches the cache it becomes visible to other threads, but it can still be lost if a crash or power failure occurs. PM stores are only guaranteed to be persisted once flushed from the cache. To control when this occurs, developers must explicitly issue flush and fence instructions. This gap between visibility and persistence exposes a specific class of bug: if one thread reads another thread's unpersisted store and a crash occurs, the system can reach an inconsistent state, leading to data corruption or loss. In this work, we refer to such bugs as *persistency-induced races* (PIRs).

Detecting PIRs requires reasoning about the gap between visibility and persistence, which creates non-persistency windows that do not exist for volatile data races. Incorrect behavior may only manifest under specific thread interleavings within this narrow temporal window, where the concurrent load happens between the store and its explicit persistence. All state-of-the-art approaches for detecting PIRs [11, 25, 72] are based on dynamic program analysis and suffer from important limitations. PMRace [11] performs exhaustive interleaving exploration by injecting delays into conflicting PM accesses in an attempt to force one thread to read another's unpersisted store. On its own, this core technique struggles to scale to real systems, thus requiring application-specific fuzzers to work effectively. Durinn [25] leverages the target application's semantics to generate adversarial workloads that induce PIRs. However, this requires prior knowledge of the applications' high-level operations that interact with PM and the implementation of application-specific drivers used for testing. Finally, HawkSet [72] uses lockset analysis [83] to detect PIRs. This approach inherently produces a considerable number of reports for pairs of accesses that cannot occur concurrently in practice, in part because it treats locks as the only synchronization primitive. In sum, no state-of-the-art tool offers a practical solution for PM concurrency bug detection.

Our insight is that, rather than searching for the proverbial needle in the haystack – a load from one thread that occurs between another thread's store and its explicit persistence – we define larger windows around PM accesses such that observing *any* overlapping execution between two threads within those regions guarantees the existence of a PIR. We call these *Persistent Interference-Free Regions* (PIFRs), as they draw inspiration from prior work on Interference-Free Regions [19, 20]. The latter were previously used to detect conventional data races efficiently, and we redesigned them to specifically capture PM semantics. Unlike IFRs, PIR detection must reason about the persistence actions and non-persistency windows that have no analogue in volatile memory, changing both the race definition and the analysis complexity. Our region-based approach substantially expands the detection window in a precise manner, increasing bug discovery without exhaustive interleaving exploration, reliance on application-specific semantics, or producing reports for pairs of accesses that cannot occur concurrently in practice. We present a formal definition of PIFRs (§ 3.2), along with a theoretical result establishing that if two conflicting PIFRs overlap, then a PIR exists.

Leveraging PIFRs, we implemented Vardalith, a tool that detects PIRs following a hybrid approach that combines static and dynamic analysis in a new way. A purely dynamic approach would be inefficient because it is difficult to characterize PIFRs using

dynamic information only, as every acquire action must conservatively be treated as the start of potentially many candidate PIFRs until PM accesses are observed, forcing the system to track a large set of potential PIFRs even though only a small fraction would actually access PM [66]. In contrast, a purely static approach cannot precisely determine PIFR ends because mapping each store instruction to its explicit persistency is, in general, extremely challenging, as persistence instructions are frequently executed in locations entirely separate from where the actual PM accesses occurred. Our hybrid design addresses both issues: an intraprocedural static component under-approximates PIFR starts and instruments the binary, while a dynamic component observes execution to detect overlaps and precisely delimit PIFR ends, achieving both precision and scalability. Another important design choice is to operate at the binary level, as it enables precisely tracking PM accesses across abstraction layers, maintaining applicability across different PM hardware implementations and software libraries, and analyzing closed-source components. We evaluate Vardalith on twelve existing PM applications, uncovering 26 PIRs (7 of which were previously unknown).

In summary, this paper makes the following contributions:

- A formalization of PIFRs and a proof that a PIFR overlap implies a PIR;
- Vardalith, a tool that implements PIFRs to efficiently detect PIRs in PM programs;
- A comprehensive evaluation demonstrating Vardalith's effectiveness by discovering 26 bugs, across multiple PM applications, including 7 previously unknown bugs.

The rest of the paper is organized as follows: § 2 provides some background, § 3 defines PIRs and PIFRs, § 4 presents Vardalith, § 5 details the implementation, § 6 evaluates Vardalith, § 7 presents the related work, § 8 discusses Vardalith's approach and its impact, and § 9 concludes the paper.

2 Background

Persistent Memory (PM) is a storage media offering a unique combination of byte-addressability through CPU load/store instructions, performance comparable to DRAM, and persistence. Both the discontinued Intel Optane [15] and CXL Type 3 [87] offer support for PM with multiple implementations currently in development [82, 63, 70].

The x86 architecture follows a relaxed, buffered persistency model [75]: when a store occurs, it is first queued in a store buffer before reaching the volatile CPU caches, where it can remain indefinitely. To guarantee persistency, the store must reach PM. This can happen non-deterministically, as the cache evicts lines to load new data. Control over store persistency and ordering requires special instructions: flush instructions, which asynchronously write cache lines to memory, and fence instructions, which ensure the completion of the flushes issued prior to them, guaranteeing that they occur before any subsequent instruction. Visibility follows TSO, meaning store order is preserved for each thread, but stores become visible to other threads only after leaving the issuing core's store buffer and reaching the shared cache hierarchy, and may be arbitrarily delayed due to buffering.

Intel's enhanced Asynchronous DRAM Refresh (eADR) [14] uses specialized hardware and batteries to guarantee that CPU caches are flushed upon a crash. However, this is costly and not expected to be universally available, and therefore PM applications should tolerate the classic ADR domain [84].

3 Persistent Interference-Free Regions

Informally, given two different threads t_1 and t_2, we say that a *persistency-induced race* (PIR) happens if thread t_2 accesses an unpersisted store made by t_1. This definition is consistent with prior work [11, 25, 72][1], and assumes stores are persisted by the thread that issued them, meaning cross-thread persistence patterns, such as flush-on-read, are outside this scope (§ 8).

As stated earlier, such occurrences are difficult to detect in practice. Our insight is that, instead of searching for a specific interleaving within a narrow temporal window, we consider larger detection windows which we call Persistent Interference-Free Regions (PIFRs). We define PIFRs such that, if there is an overlap of two threads executing PIFRs for the same PM address, then there must necessarily be a concrete interleaving that constitutes a PIR.

More concretely, a *Load PIFR* is a region surrounding a load from persistent memory starting from the immediately preceding acquire to the immediately succeeding release, exclusively. An acquire is an action that synchronizes with earlier actions in the execution (such as a mutex lock or a thread join), and a release is an action that synchronizes with subsequent actions in the execution (such as a mutex unlock or a thread create) [2]. A *Store PIFR* is a region surrounding a store to persistent memory starting from the immediately preceding acquire to the immediately succeeding release *after the explicit persistence of the store*, exclusively. We say that two PIFRs *overlap* if part of their executions happen simultaneously, and that they are *conflicting* if they belong to different threads and access the same PM address, one performing a load and the other performing a store. Our main result establishes that if a *Store PIFR* for a given memory address overlaps with a conflicting *Load PIFR*, then there is a thread interleaving whose execution forms a PIR. We formalize this result in § 3.2.

3.1 PIFRs in action

To illustrate the advantages of PIFRs, consider Figure 1, which shows a program, inspired by a few of the bugs detailed in § 6.2, with two threads (left) and a lattice of all possible interleavings (right).

Thread t_1 begins by getting a pointer to a PM variable x (line A:1). The value of x is determined dynamically by the result of the function call, but for the sake of the example, we assume x is the same for both threads. Then, t_1 checks an unlikely condition (line A:2), for instance, to trigger a data structure resizing operation. If the condition is true, it performs an acquire over resource b, computes value y, stores it in x, performs a call to an auxiliary function that eventually persists x, and releases resource b (lines A:3–7). Otherwise, if the condition is false, it performs an acquire over resource a, computes value z, stores it in x, releases resource a, persists x, and performs some cleanup (lines A:9–13). Concurrently, thread t_2 acquires resource a, loads the value of x, releases resource a, and returns the value (lines B:1–5).

Note that in the lattice, and for simplicity, we represented all lines as atomic operations, although they are split across multiple machine instructions.

Let us first consider the case where the unlikely condition is true. In this case, there is an interleaving that triggers a PIR if t_2 executes line B:3 after t_1 executes line A:6 but before line A:7, since t_2 observes the store done by t_1 before its explicit persistence. In this case, there is also a conventional data race, detectable by regular tools, since both threads access the same address without proper synchronization.

[1] This pattern should not be confused with the *persistency races* presented in [32], which consider store tearing and inventing in single-threaded executions such that, conceptually, a timely crash makes a post-failure execution race with the pre-failure execution. This pattern is outside the scope of the paper.

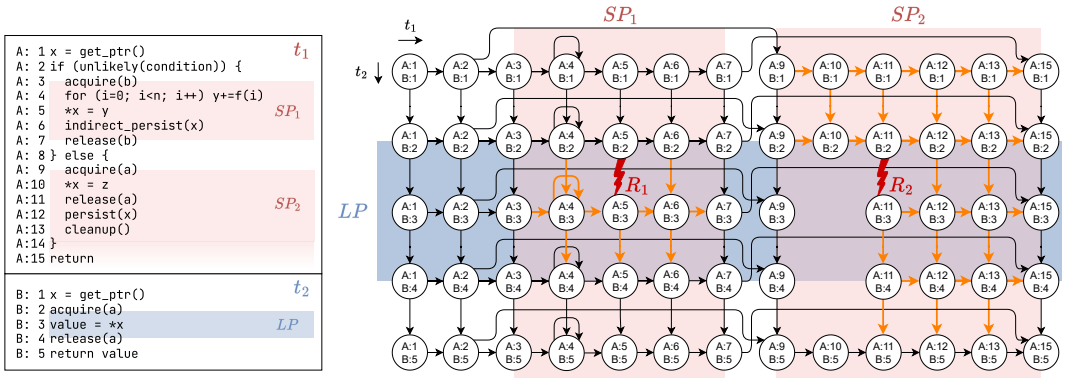

Figure 1 A buggy program that has two PIRs. The left side shows the code for two concurrent threads, t_1 and t_2, and the right shows all the possible interleavings as a lattice. The red bolts represent the two concrete interleavings that result in PIRs. The blue region LP represents the *Load PIFR* of t_2 (for the access in line B:3) and the regions SP_1 and SP_2 represent the *Store PIFRs* of t_1 (for lines A:5 and A:10, respectively). The orange arrows represent, in addition to the red bolts, all the interleavings that guarantee VARDALITH reports a PIR. For simplicity, we considered each code line as an atomic operation, although in practice they are split across multiple machine instructions.

Existing approaches [11, 25] must observe this specific interleaving (denoted with a red bolt R_1) to detect the race. The key advantage of our approach is that we are able to detect the race in a substantially larger set of interleavings. In particular, as long as we observe an execution where the *Load PIFR* of t_2, LP, overlaps with the *Store PIFR* of t_1, SP_1, denoted with orange arrows, we report a race. This means that there are multiple interleavings that do not trigger the race, and thus are not detected by existing tools, but are detected by VARDALITH and guarantee the existence of the PIR in at least some interleaving.

Next, we analyze the case where the unlikely condition is false (the common case). In this case, both accesses are synchronized correctly, however the persistency occurs outside the synchronized region. Similarly, there is a specific interleaving (denoted with red bolt R_2) that triggers a PIR if t_2 executes line B:3 after t_1 executes line A:10 but before line A:12. Since line A:10 and B:3 are properly synchronized, this also represents a very narrow window. Consequently, existing tools must either repeatedly execute these code segments in the hope of observing this specific interleaving, or steer the execution to force it to happen. This results in an undesirable trade-off between execution time and coverage. In contrast, since the PIFRs SP_2 and LP overlap in both cases, VARDALITH reports a PIR in both. Moreover, SP_2 never closes because t_1 does not perform any release action after the store is explicitly persisted. Hence, this PIFR will extend until t_1 performs another release action, or until the end of the program, further widening the detection window. As such, this means that any interleaving that includes one of the transitions marked in orange will inevitably be detected by VARDALITH as a PIR. Note that this is not a conventional data race and therefore existing volatile data race detectors [20, 21, 23] would not detect a race in any interleaving where t_1 takes the else branch since the memory accesses are properly synchronized, highlighting a key semantic difference between detecting PIRs and conventional data races.

3.2 Formal Model

We formally define PIRs and PIFRs over execution traces and prove our first main result establishing that if a *Store PIFR* for a thread t_1 overlaps with a *Load PIFR* for a thread t_2, then threads t_1 and t_2 can execute in a way that causes a persistency-induced race.

Before formalizing the result and its proof, we must first establish the program execution model. We assume the PM Intel x86-TSO memory model [75]: memory visibility follows x86-TSO consistency, while durability is decoupled from visibility. This means that writes may become visible without being persistent, and persistence is guaranteed only after explicit flush and ordering fence instructions. Since release actions imply a memory fence, we focus on flushes. Additionally, we assume stores are explicitly persisted by the thread that issued them. As such, we do not model cross-thread persistence (*i.e.*, one thread explicitly persisting another thread's store), which we treat as out of scope for this formalization (see § 8.1 for a discussion on this design choice).

Following previous work [20], we model program executions as triples $(A, \leq_{sb}, \leq_{sw})$, where:

- A is a set of actions, with each action $a \in A$ consisting of a triple (t, k, u), written $\mathbf{k}^t(u)$ for short, where: **(1)** k is the type of action, **(2)** t is a thread identifier, and **(3)** u is the unique action identifier. We do not detail all the types of actions that occur during the execution, but assume that there are: *persistent memory stores*, $\mathbf{s}_x$, *persistent memory loads*, $\mathbf{l}_x$, and *persistency instructions*, $\mathbf{p}_x$, where x denotes the target address.
- The *sequenced-before* relation, $\leq_{sb}$, totally orders actions that happen within the same thread.
- The *synchronizes-with* relation, $\leq_{sw}$, orders synchronization actions; $u_1 \leq_{sw} u_2$ implies that u_1 is the identifier of a release action and u_2 is the identifier of an acquire action for the same synchronization primitive.

The standard *happens-before* relation, $\leq_{hb}$, is simply the reflexive-transitive closure of the union of $\leq_{sb}$ and $\leq_{sw}$; put formally: $\leq_{hb} \triangleq (\leq_{sb} \cup \leq_{sw})^*$.

In order to formally define the concept of *PIR*, we need to first formalize the concept of *non-persistency window*. Intuitively, a non-persistency window corresponds to the fragment of the execution that spans from a persistent memory store to its explicit persistency. Formally, two actions $\mathbf{s}_x^t(u_s)$ and $\mathbf{p}_x^t(u_p)$ are said to form a non-persistency window, written as $\mathbb{W}_x^t\langle u_s, u_p\rangle$, if $u_s <_{sb} u_p$ and there is no action $\mathbf{p}_x^t(u'_p)$ such that $u_s <_{sb} u'_p <_{sb} u_p$. Given that, a PIR occurs when a load for a given persistent memory address x can occur during a non-persistency window for that memory address belonging to another thread. Put formally:

▶ **Definition 1** (PIR). *A non-persistency window $\mathbb{W}_x^{t_1}\langle u_s, u_p\rangle$ and a load action $\mathbf{l}_x^{t_2}(u_l)$ are said to form a persistency-induced race (PIR) if: $t_1 \neq t_2$, $u_p \not\leq_{hb} u_l$, and $u_l \not\leq_{hb} u_s$.*

Next, we formalize *Load PIFRs* and *Store PIFRs* over ordered sequences of actions:

▶ **Definition 2** (PIFR). *A* Load PIFR*, written as $\mathbb{LP}_x^t\langle u_b, u_e\rangle$, is a region that starts at u_b, ends at u_e, and surrounds a load from persistent memory address x, corresponding to some action identifier u_l, such that: $u_b <_{sb} u_l <_{sb} u_e$, there is no acquire action between u_b and u_l, and there is no release action between u_l and u_e. A* Store PIFR*, written as $\mathbb{SP}_x^t\langle u_b, u_e\rangle$, is a region that starts at u_b, ends at u_e, and surrounds a non-persistency window, $\mathbb{W}_x^t\langle u_s, u_p\rangle$, such that: $u_b <_{sb} u_s$, $u_p <_{sb} u_e$, there is no acquire action between u_b and u_s, and there is no release action between u_p and u_e.*

We say that a *Store PIFR*, $\mathbb{SP}_x^{t_1}\langle u_b, u_e\rangle$ overlaps with a *Load PIFR*, $\mathbb{LP}_x^{t_2}\langle u'_b, u'_e\rangle$, if there exists a scheduling of threads t_1 and t_2 that does not force one of the regions to end before the other begins; put formally: if $u'_e \not<_{hb} u_b$ and $u_e \not<_{hb} u'_b$. Finally, we prove our first main result: if a *Store PIFR* for a thread t_1 overlaps with a *Load PIFR* for a thread t_2, then threads t_1 and t_2 can execute in a way that causes a PIR to occur. Theorem 3 captures this intuition.

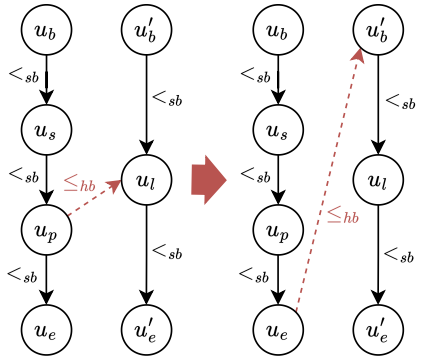

Figure 2 Contradiction for the proof of Case I, Theorem 3.

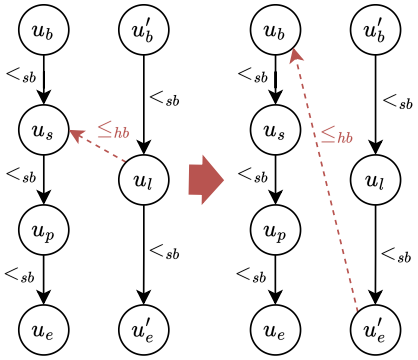

Figure 3 Contradiction for the proof of Case II, Theorem 3.

▶ **Theorem 3** (Dynamic Detection). *Let* $\mathbb{SP}_x^{t_1}\langle u_b, u_e\rangle$ *and* $\mathbb{LP}_x^{t_2}\langle u'_b, u'_e\rangle$*, be overlapping PIFRs for memory address* x *and execution* $(A, \leq_{sb}, \leq_{sw})$*, then there are some action identifiers* $u_s, u_p, u_l \in A$ *such that* $\mathbb{W}_x^{t_1}\langle u_s, u_p\rangle$ *and an action* $\mathbf{l}_x^{t_2}(u_l)$ *form a PIR.*

Sketch Proof. Let us assume that $\mathbb{SP}_x^{t_1}\langle u_b, u_e\rangle$ and $\mathbb{LP}_x^{t_2}\langle u'_b, u'_e\rangle$ are overlapping PIFRs. Further, assume the hypothesis that, despite overlapping, these do not constitute a PIR. For this to hold, either **(i)** the load occurs after the store's explicit persistence, or **(ii)** the load occurs before the store. We will show that each case leads to a contradiction, thereby proving our result.

Case I. $u_p \leq_{hb} u_l$, *i.e.* the load occurs after the persistence. For this to be true, then $u_p \leq_{hb} u'_b$, implying $u_e \leq_{hb} u'_b$, as there is no acquire action between u'_b and u_l and no release action between u_p and u_e. This would mean that $\mathbb{SP}_x^{t_1}\langle u_b, u_e\rangle$ ends before $\mathbb{LP}_x^{t_2}\langle u'_b, u'_e\rangle$ begins, contradicting the hypothesis that these are overlapping PIFRs, as shown in Figure 2.

Case II. $u_l \leq_{hb} u_s$, *i.e.* the load occurs before the store. For this to be true, then $u_l \leq_{hb} u_b$, implying $u'_e \leq_{hb} u_b$, as there is no acquire action between u_b and u_s and no release action between u_l and u'_e. This would mean that $\mathbb{SP}_x^{t_1}\langle u_b, u_e\rangle$ begins after $\mathbb{LP}_x^{t_2}\langle u'_b, u'_e\rangle$ ends, contradicting the hypothesis that these are overlapping PIFRs, as shown in Figure 3. ◀

A fully detailed version of the proof of the above theorem is provided in the appendix. Next, we show how Vardalith exploits PIFRs in practice to detect PIRs.

4 Vardalith

In this section, we present the design of Vardalith, a tool that leverages the concept of PIFRs to detect PIRs. By construction, and following Theorem 3, all Vardalith reports correspond to true PIRs, but delimiting PIFRs in practice raises a few fundamental challenges. On the one hand, a purely dynamic approach that considers every acquire action as a potential PIFR start would not scale, since only a small fraction of memory accesses target PM [66]. On the other hand, a purely static analysis cannot, in general, precisely determine the start and end of a PIFR for a given PM access. For example, PIFR ends are often delegated to auxiliary functions or libraries, thus requiring precise interprocedural analysis [4], which is provably expensive and, in many cases, even impossible [79].

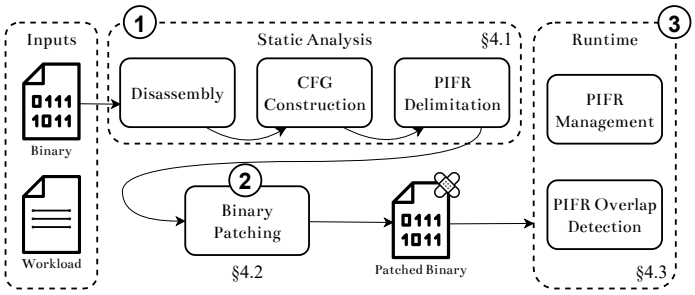

Figure 4 VARDALITH pipeline.

We address this tension between precision and scalability with a hybrid design that combines static and dynamic analysis. First, to avoid considering every acquire action as a potential PIFR start, we employ an intraprocedural static component that under-approximates PIFR starts and instruments the binary to track relevant PM operations. Second, to delimit PIFR ends without costly and potentially imprecise interprocedural static analysis, we make use of a dynamic component that monitors execution to detect PIFR overlaps and precisely define PIFR ends. This combination allows us to achieve both precision and scalability.

Next, we give an overview of VARDALITH. We defer the details of the static analysis algorithm and correctness proof to § 4.2.

4.1 Vardalith Overview

VARDALITH's pipeline is divided into three stages, as depicted in Figure 4. The first stage takes the program binary as input and uses static analysis to determine the PIFR starts (§ 4.1.1). The second stage instruments the binary with calls to our runtime whenever the program enters a PIFR or performs a call with release semantics, which may delimit PIFR ends (§ 4.1.2). Finally, the third stage takes a user-provided workload as input and performs a series of testing runs of the instrumented binary (§ 4.1.3).

4.1.1 Static Analysis Stage

The purpose of this stage is to precisely determine, for each PM access, the instruction address that marks the start of its enclosing PIFR. Note that we want to define PIFRs as large as possible to increase the chances of detecting overlaps at runtime. However, to maintain both precision and scalability, we restrict ourselves to intraprocedural analysis, meaning that each PIFR start will be at most at the beginning of the function where the corresponding PM access occurs.

Informally, given a PM access a targeting memory address x, we want to find the earliest instruction s that satisfies the following constraints: (i) the execution of s always precedes the inevitable execution of a, (ii) there is no acquire action between s and a, and (iii) s falls within the scope of the address x accessed by a. The first two constraints come directly from the definition of PIFR presented in § 3. The third constraint is more subtle. In particular, in the dynamic analysis component, we need to know the value of the PM address x at the PIFR start s. At the source code level, this can be understood as the standard notion of scope (*i.e.* s belongs to the scope of x). However, as variable scope is not known at the binary level, we rely on a strictly narrower definition of scope: for s to be inside the scope of

x we need to be able to generate a static expression based on registers and constants that evaluates to the value of x when computed at instruction s (explained in more detail in § 5). As registers are often reused by the compiler, this notion of scope might be narrower than the variable scope at the source code level. Although some techniques work around this issue by statically aliasing memory accesses to memory regions, such as value-set analysis [7, 6] or alias pairing [102, 42], they are not precise and would compromise the inherent properties of our approach. Note that it is always possible to find an instruction that satisfies all criteria – in the worst case it is the PM access itself.

Our static analysis for computing PIFR starts is detailed in § 4.2. Despite our practical considerations in choosing intraprocedural static analysis for scalability and precision, our approach still delimits large regions effectively, as shown in the evaluation (§ 6.3). The output of this stage is a set of instruction addresses corresponding to PIFR starts and respective target expressions, which are passed to the patching stage.

4.1.2 Patching Stage

The patching stage rewrites the binary to inject calls to our runtime: (i) `start_pifr(type, target)` placed at the PIFR starts determined by the previous stage, where `type` is *load* or *store*, and `target` is the memory address accessed by the instruction; (ii) `flush(target)`, placed before all flush instructions (both in the binary and library dependencies); and (iii) `end_pifrs()`, placed before all instructions with release semantics. Note that `end_pifrs()` is placed before any instruction that *may* end a PIFR, but it is the runtime that actually determines which PIFRs to close.

This phase also deduplicates start calls that share the same `type`, `target`, and start instruction. This is only possible thanks to the static analysis phase and further reduces the instrumentation cost relative to a purely dynamic approach.

4.1.3 Dynamic Analysis Stage

This stage consists of submitting the patched binary to a series of testing runs and searching for PIFR overlaps using the runtime calls injected previously.

To track potentially conflicting accesses, the runtime keeps a map of open PIFRs indexed by the target memory address. Whenever a PIFR opens, it is added to the map with a state associated, which can be `dirty` (*i.e.* an unpersisted store operation) or `clean` (*i.e.* a load or an explicitly persisted store). As such, *Store PIFRs* (resp. *Load PIFRs*) are opened in a dirty (resp. clean) state.

To dynamically detect PIFR overlaps, whenever a PIFR opens, the runtime queries the map to determine if there are open PIFRs for the same target address. If so, it checks each of them against the newly opened PIFR. For each pair of PIFRs, and following Theorem 3, if they were opened by different threads, one is a *Store PIFR*, and the other is a *Load PIFR*, then we report a PIR.

To keep track of PIFR state, whenever a flush operation takes place, the runtime scans the map for *Store PIFRs* opened by the issuing thread, within the flushed range, and changes their state to `clean`. Since a flush acts on an entire cache line, often containing multiple individual store operations, the runtime must check for all addresses within the same cache line to ensure correctness.

Finally, to precisely delimit PIFRs, immediately before the program performs a release, Vardalith closes all open *Load PIFRs* or `clean` *Store PIFRs* of the issuing thread. This dynamic tracking of PIFR closure ensures that we close PIFRs as late as possible, and hence maximize the possibility of overlaps without incurring costly interprocedural static analysis.

Note that synchronization primitives typically perform atomic updates and thus have fence semantics. Hence, we do not instrument fence instructions (which avoids unnecessary overhead) and assume that, in the worst case, flushed stores are fenced when the release primitive is executed.

4.2 Vardalith Algorithm

In this section, we describe VARDALITH 's static analysis algorithm. We first introduce the necessary preliminaries (§ 4.2.1), then present the algorithm itself (§ 4.2.2), walk through an example execution (§ 4.2.3), and conclude with a proof of correctness (§ 4.2.4).

4.2.1 Preliminaries

We model control flow graphs (CFGs) [3, 90, 71] as directed graphs where nodes represent basic blocks and edges represent jumps or fallthrough between two basic blocks within a function. We consider a basic block a contiguous sequence of instructions with a single entry and a single exit point. If there is an edge from block b to block b', then we say that b is a predecessor of b' and b' is a successor of b. A CFG has an entry point, *i.e.*, a block without predecessors, and one or more exit blocks, *i.e.* blocks without successors. A block b dominates b' – denoted as b **dom** b' – if all paths from the start of the CFG to b' must pass through b. A block trivially dominates itself. Conversely, block b postdominates block b' – denoted as b **pdom** b' – if all paths from b' to exit blocks must pass through block b. Note that the two relations are not symmetric.

4.2.2 Algorithm

Given a PM access a in block b we want to find the starting point s of a's PIFR. Revisiting the constraints introduced in § 4.1.1, this starting point s is the earliest instruction in some block b' in the CFG that satisfies all of the following constraints: (i) no path from s to a performs an acquire action, (ii) b **pdom** b', and (iii) s falls within the scope (to be defined shortly) of a's target address. Note that since a might be reachable through more than one execution path, there might be more than one PIFR start for a given access that satisfy all the above constraints.

To detect the earliest possible PIFR start for a given PM access a we perform a backward data-flow analysis, starting from the block b that a belongs to, going as far back as possible and halting the process when it either: (C1) encounters an acquire instruction, (C2) reaches the beginning of the CFG, (C3) exits the scope of the target address, or (C4) cannot guarantee postdominance by b. Condition C4 is essential to guarantee that executions that reach the computed PIFR start will inevitably result in the execution of the PM access that it delimits.

Algorithm 1 represents our static analysis, using a worklist approach [3]. The main logic is performed in function `FindPIFRStart` (lines 1–14), which takes as input the instruction corresponding to the PM access, i, and the expression that determines its target address, e. In the initialization (lines 2–4), we create a worklist, $\mathcal{W}$, containing the original access instruction, i. Then, at each iteration, we take an instruction i from $\mathcal{W}$ (line 6), and process it (line 7) to determine if its block has an instruction that delimits the PIFR start (lines 15–21). To this end, this logic checks if conditions C1 and C3 hold while reverse iterating over the block's instructions.

Once the block is processed, we check if we have found the PIFR start and, if so, add it to set $\mathcal{S}$ (line 8). Otherwise, we have not found a PIFR start and the analysis must continue. To do so, we compute the set of walkable predecessors of b and add them to $\mathcal{W}$ (lines 10–12).

Algorithm 1 Static delimitation of PIFR start.

```
1  fn FindPIFRStart(i, e)
2      b† ← block(i) ▷ block that contains original access.
3      S ← ∅ ▷ set of possible PIFR starts.
4      W ← {i} ▷ worklist of instructions.
5      while W do
6          i ← pop(W)
7          foundStart, i' ← RevProcessBlock(i, e)
8          if foundStart then S ← S ∪ i'
9          else
10             preds ← RevWalkCFG(block(i), b†, e)
11             for b' ∈ preds do
12                 W ← W ∪ end(b')
13             if preds = ∅ then S ← S ∪ i'
14     return S
15 fn RevProcessBlock(i, e)
16     b ← block(i)
17     while i ∈ b do
18         if i is acquire ∨ i ∉ scope(e) then
19             return ⟨true, next(i)⟩
20         i ← prev(i)
21     return ⟨false, entry(b)⟩
22 fn RevWalkCFG(b, b†, e)
23     walkable ← ∅
24     for b' ∈ predecessors(b) do
25         if ¬ b† pdom b' ∨ b' ∩ scope(e) = ∅ ∨
             (∃ p, i ∈ p : b' ⇝_p b† ∧ i is acquire) then
26             return ∅
27         if ¬ b† dom b' then
28             walkable ← walkable ∪ b'
29     return walkable
```

▷ $a \rightsquigarrow_p b$ represents a sequence of instructions p that reaches b starting from a.

▷ `entry()` and `end()` return the first and last instruction from a block, respectively. `prev()` and `next()` return the instruction immediately before or after, respectively.

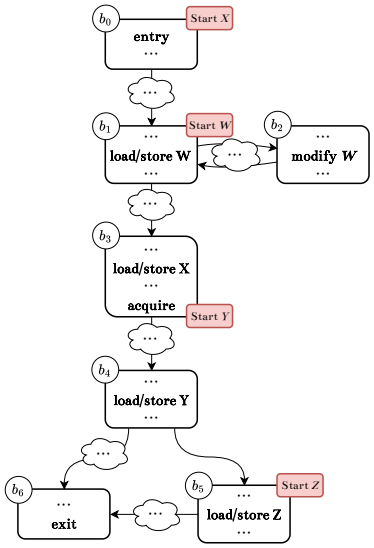

Figure 5 Example CFG. `load/store [x]` represents a PM load/store on PM address x. `modify a` represents a modification of value a, which is used as a target address for a PM access. The red shaded boxes denote PIFR starts. Each ellipsis represents an undetermined number of non-acquire instructions or blocks, assuming postdominance is maintained.

The walkable predecessors are the subset of predecessors of the current block that satisfy conditions C1 – C4, as computed by `RevWalkCFG` (lines 22–29). This function takes as input the current block b, the original block $b^\dagger$, and the target address e, and iterates over the predecessors of block b to determine if they are walkable (lines 24–28). More concretely, it checks if (line 25): (i) all the predecessors are postdominated by the block containing the original PM access, $b^\dagger$ (C4), (ii) all the predecessors are inside the scope of e (C3), and (iii) none of the predecessors can reach the original block $b^\dagger$ via a path that performs an acquire action (C1). If any of these conditions fails, then none of the predecessors is walkable. Additionally, if b does not have any predecessors, we have reached the start of the CFG (C2) and found the PIFR start. Finally, a predecessor is only added to the walkable set if it is not dominated by $b^\dagger$ (lines 27–28), since otherwise it would indicate that the algorithm would be moving forward in the CFG. Once $\mathcal{W}$ is empty (line 5), we have found all PIFR starts.

4.2.3 Example

Figure 5 exemplifies some of the cases that Algorithm 1 needs to handle. This CFG performs accesses on PM addresses W, X, Y, and Z. The red boxes denote the PIFR start of each access. Access W has a PIFR start in b_1 because blocks b_1 and b_2 are part of a loop and b_2 modifies W. This causes the target address of the access performed on b_1 to potentially

change in each iteration of the loop. Therefore, the scope of W is delimited by the loop, since we cannot determine a static expression that computes the memory addresses targeted by W before each iteration of the loop. For that reason (C3), W's PIFR can stretch at most to the beginning of the loop. Access X in b_3 has a PIFR start in b_0. This represents the ideal case, since X is not modified and there are no acquire actions before the access, allowing us to start the PIFR at the beginning of the CFG (C2). Access Y in b_4 has a PIFR start in b_3, since b_3 performs an acquire action (C1). Finally, access Z in b_5 has a PIFR start in b_5 because it does not postdominate its only predecessor (C4).

4.2.4 Algorithm Correctness

In this section, we prove the correctness of our algorithm for computing static PIFR starts. We prove this in two steps. First, we prove that our criteria to define static PIFR starts under-approximates the dynamic definition presented in § 3. Second, we prove that Algorithm 1 computes correct static PIFR starts.

In the following, we write $\mathsf{instr}(u) = i$ to mean that i is the instruction executed with identifier u and $\mathsf{block}(i) = b$ to mean that instruction i belongs to block b. An instruction i_1 is said to immediately precede instruction i_2 in CFG g, $g \vdash i_1 \rightarrow i_2$, where $\mathsf{block}(i_1) = b_1$ and $\mathsf{block}(i_2) = b_2$, if: these are consecutive instructions in the same block or i_1 is the last instruction of b_1 and i_2 is the first instruction of subsequent block b_2. Starting at instruction i_1, it is possible to reach instruction i_2, via path p in CFG g, written $g \vdash i_1 \rightsquigarrow_p i_2$, if there is a sequence of immediately preceding instructions between them. Based on these constructs, an execution $(A, \leq_{sb}, \leq_{sw})$ is said to be consistent with a CFG g if for all actions $\mathbf{k}_1^t(u_1), \mathbf{k}_2^t(u_2) \in A$, where $\mathsf{instr}(u_1) = i_1$ and $\mathsf{instr}(u_2) = i_2$, it holds that immediate precedence respects the *sequenced-before* relation: $u_1 <_{sb} u_2 \implies g \vdash i_1 \rightsquigarrow_p i_2$.

We now formalize static PIFR starts and prove how they always correctly under-approximate the corresponding dynamic PIFR starts.

▶ **Definition 4** (Static PIFR Start)**.** *Given a CFG g, an instruction i_1 is said to be a* static PIFR start *for instruction i_2 that accesses a persistent memory address via target expression e, written i_1* **pifr start** *i_2, if:* $\mathsf{block}(i_2)$ **pdom** $\mathsf{block}(i_1)$ *in g; there is no acquire instruction in all paths that connect i_1 to i_2 in g; and e is such that the computed value at i_1 coincides with the persistent memory address accessed by i_2.*

▶ **Theorem 5** (Static PIFR Start vs. Dynamic PIFR Start)**.** *Given a CFG g, let i_1* **pifr start** *i_2 and let $(A, \leq_{sb}, \leq_{sw})$ be an execution consistent with g. If it holds that $\mathsf{instr}(u_1) = i_1$ for a given action identifier u_1 and the execution terminates, then there must exist an action identifier u_3 s.t. $u_3 \leq_{sb} u_1$ and u_3 is the start of a dynamic PIFR.*

Proof. First, we note that the hypothesis that execution terminates together with the definition of *static PIFR start* imply that there is an action identifier u_2 such that $\mathsf{instr}(u_2) = i_2$. This is because the definition of *static PIFR start* requires i_1 to be postdominated by i_2; therefore all executions that go through i_1 must also go through i_2, provided that they terminate.

There are now two cases to consider: **(1)** the persistent memory access corresponding to u_2 is contained in a PIFR that started immediately after an acquire action; **(2)** there is no acquire action that precedes u_2, in which case the start of the region coincides with the beginning of the execution. Here, we only consider case **(1)** as the property trivially holds for case **(2)**.

Assuming that there is an acquire action u_3, where $\mathsf{instr}(u_3) = i_3$, after which a dynamic PIFR starts, we only have to prove that $u_3 \leq_{sb} u_1$. Assume, by contradiction, that $u_3 \not\leq_{sb} u_1$. Since $\leq_{sb}$ is a total order, it must be the case that $u_1 <_{sb} u_3 <_{sb} u_2$. Given that the execution is consistent with the CFG, it follows that: $i_1 \rightsquigarrow i_3 \rightsquigarrow i_2$. However, this contradicts the definition of static PIFR start which requires that there is no acquire instruction between i_1 and i_2. ◀

Finally, we prove the correctness of Algorithm 1 by showing that all the *static PIFR starts* computed by the algorithm are valid, meaning that they satisfy the conditions established by our definition.

▶ **Theorem 6** (Correctness of Static PIFR Start Computation). *All the* static PIFR starts *computed by Algorithm 1 are valid. Formally: if* $i_1 \in \mathsf{FindPIFRStart}(i_2, e)$*, where* i_2 *is a PM access to a memory address computed via target expression* e*, then* i_1 *is a* static PIFR start *for instruction* i_2*.*

Proof. In order to prove the theorem, we establish that the main loop (lines 5–13) maintains the following invariant: for all $i_1 \in \mathcal{W} \cup \mathcal{S}$, it must hold that i_1 **pifr start** i_2.

The invariant trivially holds when the execution reaches the loop since $\mathcal{S} = \emptyset$ and $\mathcal{W}$ contains only the target instruction i_2. The invariant holds for the set $\{i_2\}$ since: i_2 trivially postdominates itself, i_2 is not an acquire, and e is in scope at i_2. To show that the invariant is maintained by the loop, we observe that:

- Only the last instructions of blocks returned by RevWalkCFG are added to $\mathcal{W}$ (line 11), and it only returns a block b after checking that: **(1)** $\mathsf{block}(i_2)$ **pdom** b, **(2)** the value of e is accessible from b, and **(3)** there is no path from b to $\mathsf{block}(i_2)$ that performs an acquire (line 25);
- Only instructions returned by RevProcessBlock are added to the set of PIFR starts $\mathcal{S}$, and it only returns i_1 after checking that: **(1)** i_1 is not an acquire, **(2)** i_1 is in scope of e (line 18). Since RevProcessBlock only processes instructions inside a single block at a time in reverse order, and all instructions in $\mathcal{W}$ respect postdominance of i_2, the same holds for all the instructions in $\mathcal{S}$.

When the main loop finishes, $\mathcal{W} = \emptyset$ and the invariant holds for all the instructions in $\mathcal{S}$. ◀

5 Implementation

VARDALITH's implementation[2] consists of $\approx 7k$ lines of code (C/C++, Python, and Rust). The implementation closely follows the pipeline in Figure 4, with a minor modification. Before performing the static analysis, the application is executed with the provided workload, to gather which loads and stores are PM accesses. This is done with lightweight instrumentation using Intel PIN [62] to determine, for each access, if its target address belongs to the range assigned to PM. This range is determined by instrumenting `mmap` system calls and analyzing their arguments and return values. We use the same method during testing to ignore DRAM accesses. Note that even though the prototype uses tracing to gather the PM accesses, the approach itself is not dependent on it, and other techniques such as static points-to analysis [34] applied in other works [68] could be used instead. For the static analysis we use Zydis [106] to disassemble the binary, after which we reconstruct the control-flow graph for each function and delimit the PIFRs (§ 4.1.1). This analysis is custom and operates directly on binaries, rather than on LLVM IR or another external analysis framework. To identify calls to synchronization primitives, we use a list of known acquire and release calls, including

[2] `https://github.com/task3r/mumak`

popular libraries such as `pthreads`. To support custom synchronization methods, the user can extend the list with the names of the relevant calls (typically few and well-known), which does not imply manually annotating the code. Next, we use e9patch [18] to rewrite the binary with calls to the runtime (§ 4.1.2), as well as add logic to dynamically load the runtime library at the start of the execution. We provide symbolized backtraces to help developers during debugging. To achieve this, we implemented a custom mechanism that traverses frame pointers to obtain backtrace addresses, which are then symbolized offline. This is necessary because binary rewriting can break symbols in binaries and, although our approach does not rely on symbols to find PIRs, symbol information remains valuable for debugging.

5.1 x86 Variable Scope

Working at the binary level has advantages, such as generality and practicality, but also implementation challenges and drawbacks, most notably the concept of target address and scope. Approaches that work over source code or intermediate representations, such as LLVM IR [49], have variable names with scopes defined. In contrast, in compiled binaries, accesses are performed on memory operands. For x86, these operands are represented as `base_register + (index_register * scale) + displacement`, with some parameters being optional. The same expression might refer to different memory locations within the same function, especially since compilers reuse registers when possible. To address this, for each PM access we capture its destination operand (if the access is a store) or source operand (if the access is a load) and save the respective expression, together with the registers it depends upon. For example, if the original access is `mov [rax], rdx`, which stores the value of `rdx` in the address stored in `rax`, then the target expression is a single register, `rax`. During backwards analysis, for each processed instruction that writes to any of the relevant registers, we update the expression accordingly so that we can recompute the original address.

5.2 Sampling and Scheduling

The base design of Vardalith does not actively steer the execution towards specific interleavings. Instead, it executes the program under test and detects overlapping PIFRs dynamically. To complement the base design, we implemented two additional optional features in the prototype: sampling and scheduling. Sampling offers a configurable trade-off between execution time and coverage. Although Vardalith is generally fast, certain testing scenarios can take more time than desired, for instance if Vardalith is included as part of the build process to check for PIRs as the code is developed. Vardalith can be configured with a sampling rate of $1/n$, in which case it only keeps track of 1 out of n PIFRs. This reduces bookkeeping costs and testing time but since fewer PIFRs are open at any given time, it also reduces coverage. We study this trade-off in § 6.4.

We also implemented a delay injection-based scheduling component, whose goal is twofold. First, we want to assess the extent to which PIFR's larger windows can still benefit from guiding the execution to specific interleavings. Second, we use it as a baseline, to compare the effects of enlarging the detection windows with PIFRs versus doing so through delay injection to force specific interleavings. Following previous works on race detection [54, 88], we implemented an active delay injection mechanism paired with a probability decay. Each time we reach a release action and before closing the respective PIFR, we inject a delay with probability p (starting at 100%). After the delay, we assess whether any races were detected during that period. If no races were found, we decrease the probability of injecting a delay at that point by a configurable constant. This tries to minimize the delays at ineffective

locations, and maximize the chances of finding PIRs while minimizing the testing time required to do so. We evaluate the effectiveness of active delay injection, both standalone and in combination with PIFRs in § 6.3 and § 6.4.

5.3 Implementation Limitations

Our implementation assumes that synchronization is used explicitly, as the result of function calls (e.g., a call to a mutex lock function, semaphore, or a conditional variable). These functions can sometimes be inlined for performance, but debug builds generally do not perform this inlining and therefore suffice for Vardalith. Nevertheless, the use of C/C++ macros to define synchronization primitives can be problematic since these are inlined even in debug builds. In such cases, the programmer needs to wrap/replace the macros with inlined function calls, so that these calls are present on debug binaries but optimized in release builds. This is a simple change that we performed manually on P-CLHT [50] by modifying less than 20 lines.

6 Evaluation

In this section, we evaluate Vardalith. The main takeaway is that we found 26 bugs, 7 of which are new (§ 6.2). We demonstrate the effectiveness of PIFRs (§ 6.3), evaluate Vardalith's performance (§ 6.4), and compare it with PMRace (§ 6.5) and ThreadSanitizer (§ 6.6).

6.1 Experimental Setup

We used a dual-socket 64 core Intel Xeon Gold 6338N, with 256 GB of RAM, and 1 TB Intel DCPMM, running Ubuntu 22.04 with Linux kernel 5.15.

We analyzed 12 programs: P-CLHT [50], FAST-FAIR [37], Memcached [52], Apex [61], and WIPE [96] using the authors' implementation [89, 45, 53, 60, 95]; SEPH [92], Level Hashing [105], Clevel Hashing [10] and CCEH/CCEH-CoW [67], using SEPH's benchmark and implementations [64]; and P-ART and P-Masstree [50] using the implementations provided by Durinn [46], as was done in [72]. All experiments ran using 8 threads and workloads of 1k, 10k, and 100k operations, with the exception of P-ART since this implementation does not work with workloads larger than 1k operations [72]. The P-CLHT and SEPH benchmarks use a YCSB [13] workload with a single-threaded load phase, followed by 50% gets and 50% updates. FAST-FAIR uses its own benchmark, which performs inserts/updates/gets/deletes. Memcached uses a Python client that performs all the operations available in the protocol in round-robin fashion. All workloads access keys following a zipfian distribution.

6.2 Bug Detection

Vardalith found 26 bugs, 7 of which are new. For comparison, PMRace [11] found 8 bugs (4 new), labeled as *PM inter-thread inconsistency* in the original paper, Durinn [25] found 10 bugs (10 new), labeled as *visible-but-not-durable bugs* in the original paper, and HawkSet [72] found 20 bugs (7 new). The results are summarized in Table 1, where we also report when multiple read/write accesses trigger the same bug. Considering the intersection of evaluated systems, Vardalith finds all bugs reported by PMRace on P-CLHT, FAST-FAIR, and Memcached, and all bugs reported by HawkSet on FAST-FAIR, P-CLHT, P-Masstree, P-ART, Memcached, WIPE, and APEX, *i.e.*, all bugs listed in those tools' evaluation tables for the overlapping systems. Durinn works at the operation level and hence reports bugs at the operation declaration rather than at the PM access as done by Vardalith, PMRace, and HawkSet. Therefore, these reports are not directly comparable on a one-to-one basis.

Table 1 Bugs detected by VARDALITH. The load/store columns identify some accesses that triggered the bug.

Target	#	New	Load access	Store access	Description	Impact
P-CLHT	1		clht_lb_res.c:417 clht_lb_res.c:431 clht_lb_res.c:514 clht_lb_res.c:528 clht_lb_res.c:561	clht_lb_res.c:785 clht_lb_res.c:785 clht_lb_res.c:785 clht_lb_res.c:785 clht_lb_res.c:785	Resize operation is not correctly synchronized and this leads to reading unflushed pointer and inserting/updating/removing data	Data Loss
	2	✓	clht_gc.c:186	clht_gc.c:199	Unpersisted garbage collection pointer can result in a double-free bug after a crash	Data Corruption
P-ART	3		N4.cpp:56 N16.cpp:61 N256.cpp:39	N4.cpp:22 N16.cpp:13 N256.cpp:17	Lock-free get operation returns unpersisted inserted value	Causality Violation
	4		N4.cpp:56 N16.cpp:61	N4.cpp:67 N16.cpp:76	Lock-free get operation fails to find unpersisted deleted value	Causality Violation
P-Masstree	5		masstree.h:1883	masstree.h:822	Lock-free get operation returns unpersisted inserted value	Causality Violation
	6		masstree.h:1883	masstree.h:1387	Lock-free get operation returns unpersisted inserted value	Causality Violation
	7		masstree.h:1953	masstree.h:1425	Lock-free get operation fails to find unpersisted deleted value	Causality Violation
Level Hashing	8	✓	level.hpp:347 level.hpp:347	level.hpp:386 level.hpp:412	Insert checks for existing values without proper synchronization and this leads to reading unflushed token and not inserting new value	Causality Violation
CCEH	9	✓	cceh.hpp:561 cceh.hpp:758 cceh.hpp:761 cceh.hpp:761	cceh.hpp:681 cceh.hpp:681 cceh.hpp:665 cceh.hpp:666	Directory doubling is not correctly synchronized and this leads to reading unflushed pointer and inserting data	Data Loss
CCEH CoW	10	✓	cceh_cow.hpp:737 cceh_cow.hpp:740 cceh_cow.hpp:740 cceh_cow.hpp:740 cceh_cow.hpp:740	cceh_cow.hpp:660 cceh_cow.hpp:640 cceh_cow.hpp:641 cceh_cow.hpp:644 cceh_cow.hpp:645	Directory doubling is not correctly synchronized and this leads to reading unflushed pointer and inserting/updating data	Data Loss
	11	✓	cceh_cow.hpp:947	cceh_cow.hpp:852	Get operation is performed lock-free and observes unpersisted update	Causality Violation
FAST-FAIR	12		btree.h:878	btree.h:560	Split operation is not correctly synchronized and this leads to reading unflushed pointer and inserting data	Data Loss
	13		btree.h:878	btree.h:571	Split operation is not correctly synchronized and this leads to reading unflushed pointer and inserting data	Data Loss
Memcached	14	✓	memcached.c:2823	memcached.c:4292	Store item function copies unpersisted value from prepend	Causality Violation
	15	✓	memcached.c:2823	memcached.c:4293	Store item function copies unpersisted value from prepend	Causality Violation
	16		memcached.c:2805	memcached.c:4292	Store item function copies unpersisted value from append	Causality Violation
	17		memcached.c:2805	memcached.c:4293	Store item function copies unpersisted value from append	Causality Violation
	18		items.c:464	items.c:423	Item unlink function updates pointer based on unpersisted index	Inconsistent Index
	19		slabs.c:412	slabs.c:549	Allocate slab based on unpersisted pointer	Inconsistent Index
	20		memcached.c:2824	items.c:1096	Store item function copies value based on unpersisted flag	Inconsistent Data
	21		items.c:623	items.c:627	Item update function updates cache based on unpersisted index	Inconsistent Index
WIPE	22		pointer_bentry.h:1606 pointer_bentry.h:1606	pointer_bentry.h:1771 pointer_bentry.h:1799	Lock-free get operation returns unpersisted data	Causality Violation
	23		pointer_bentry.h:1601 pointer_bentry.h:1601	pointer_bentry.h:1550 pointer_bentry.h:1779	Lock-free get operation returns unpersisted data	Causality Violation
	24		letree.h:228	letree.h:393	Node resize is not atomically persisted leading to lost updates	Data Loss
APEX	25		apex_nodes.h:2915 apex_nodes.h:2933	apex_nodes.h:3479 apex_nodes.h:3798	Lock-free get operation returns unpersisted insert	Causality Violation
	26		apex_nodes.h:3480 apex_nodes.h:3606	apex_nodes.h:962 apex_nodes.h:962	Lock-free get operation returns unpersisted update	Causality Violation

Nonetheless, VARDALITH detects races on the same operations reported by DURINN in P-Masstree and P-ART, so we follow the same best-effort reasoning presented in [72] and consider that VARDALITH also finds the 5 PIRs reported by DURINN in those systems We reported all the new bugs to the developers, and one has already been confirmed. Next, for each program, we briefly discuss the new bugs found by VARDALITH. A detailed description of each new bug is provided in the appendix.

P-CLHT [50] is a persistent cache-friendly hash table. Bug #2 is caused by two stores during garbage collection that are not persisted, leading to a double-free after a crash and potential data corruption. Note that this bug stems from an unpersisted store and therefore also manifests in single-threaded executions.

Level Hashing [105] is a PM write-optimized hashing index. Bug #8 occurs when two threads concurrently insert the same key. The first thread completes the insert but before it is persisted, the second thread, without proper synchronization, observes the inserted entry and reports its existence to the client. Following a crash, this can result in a causality violation.

CCEH [67] is a variant of extendible hashing for PM. Bug #9 results from incorrect synchronization during a directory doubling operation, allowing concurrent inserts to store data based on an unpersisted pointer, leading to data loss after a crash. CCEH-CoW is a variant that uses *copy on write*. Since it shares part of the CCEH implementation, it exhibits

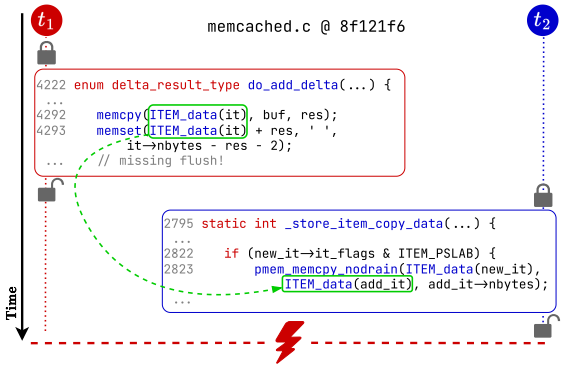

Figure 6 New bugs found in Memcached [52] by VARDALITH (Bug #14 and Bug #15).

the same issue (Bug #10). Bug #11 occurs because the implementation of a lock-free search in CCEH-CoW allows a get operation to return an unpersisted value. Following a crash, this can result in a causality violation.

Finally, we use the new bugs found by VARDALITH in Memcached [52] as a case study for our approach. Bug #14 and Bug #15 are triggered by similar executions, depicted in Figure 6. Briefly, Thread t_1 receives an increment/decrement request, acquires the item lock, and performs the operation by issuing calls to `memcpy` and `memset`, which store the new value of the item. It then releases the lock without explicitly persisting those stores. Subsequently, Thread t_2 receives a request to perform a prepend operation on the same key. This involves acquiring the lock and copying the existing data to a new item. A subsequent crash results in inconsistent data and client observations. Since the stores performed in lines 4292 and 4293 were never explicitly persisted, the *Store PIFRs* remain open indefinitely. Therefore, any subsequent prepend operation on the same key opens a *Load PIFR* on the same address, resulting in a report. These bugs are similar to Bug #16 and Bug #17, previously reported by PMRace [11], which occur when the append operation occurs concurrently with an increment/decrement on the same key.

6.3 PIFR Effectiveness

We now assess the effectiveness of PIFRs. We start by comparing VARDALITH with a *baseline* implementation. The baseline shares the same code base of VARDALITH but, rather than using PIFRs, the regions just enclose the PM accesses mimicking what is done in approaches such as PMRace. We test both variants with different scheduling parameters (§ 5) and workloads of 1k, 10k and 100k operations. Additionally, we also tested VARDALITH with sampling. We ran VARDALITH five times (which took 90 minutes for the largest workload) and counted the number of unique PIRs found. Then, for the baseline, we ran it until it either found the same number of unique races or until a timeout of one week. Figure 7 presents the results.

While we ran this experiment for other programs, we only present the results for FAST-FAIR, since it has a large number of PIRs (as detailed in § 6.4) and thus represents a favorable case for the baseline approach.

There are a few takeaways from these results. First, VARDALITH detects the unique races substantially faster than the baseline. For the 1k workload, it takes 3 days for the baseline approach to find the same number of unique PIRs that VARDALITH finds in 5 testing runs, each run taking, even in the most unfavorable case, less than 2 seconds on average (§ 6.4). For larger workloads, we timed out the baseline execution after a week, and by then it found less than half the PIRs that VARDALITH found in 5 runs. Furthermore, even in the most

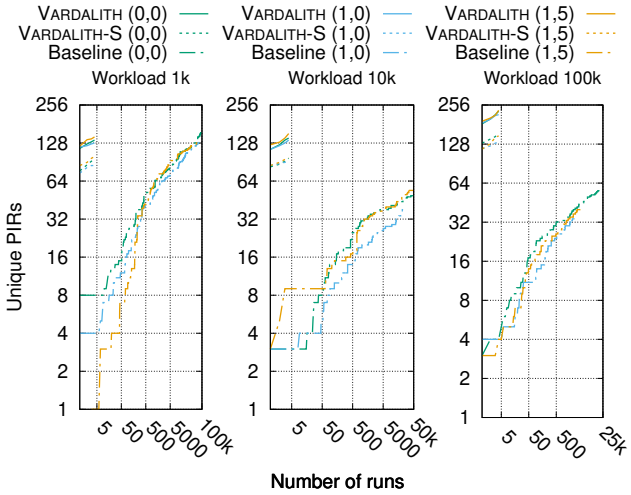

Figure 7 Unique PIRs detected by Vardalith and a baseline approach in FAST-FAIR, under different configurations. Tuple (x, y) represents a delay of xms, and a decay of $y\%$. Vardalith-S has a sample rate of 10%.

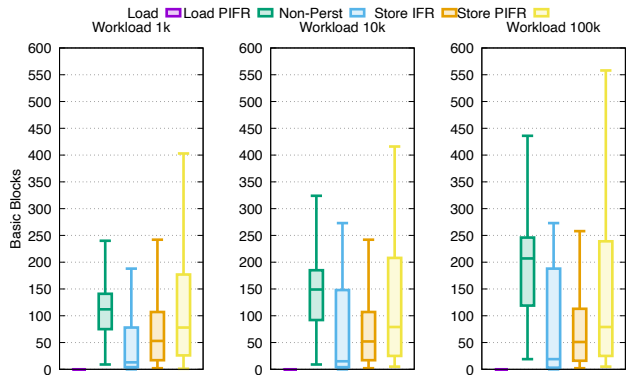

Figure 8 Size of PIFR regions in comparison with the classic race detection window and IFRs in FAST-FAIR. A Load PIFR is equivalent to a Load IFR.

unfavorable case, each of these 5 runs took on average 1.2 and 13.75 minutes for 10k and 100k workloads, respectively. Second, Vardalith scales well, detecting more PIRs as the workload size increases. These additional races stem from the need to perform rebalancing operations that are only triggered for large workloads and hence can only be tested with scalable tools. Interestingly, the baseline performance degrades with increasing workloads. We attribute this to the fact that conflicts become sparser as the workload increases which reduces the probability of detection given the baseline's small detection window. Finally, and as detailed in § 6.4, sampling affects the number of unique races detected, but even with sampling Vardalith is much more effective than the baseline.

Next, to understand why PIFRs are more effective, we compared their size, in terms of encompassed basic blocks, with two baselines: IFRs [20], which encompass the region from the immediately preceding acquire to the immediately succeeding release for each memory access, regardless of persistency; and the *non-persistency windows* (Non-Perst), which represent the region between a store and its explicit persistency (as defined in § 3.2). The results are shown in Figure 8. For brevity, we only present results for FAST-FAIR, but the other programs follow a similar trend. The key observation is that PIFRs are significantly larger than the

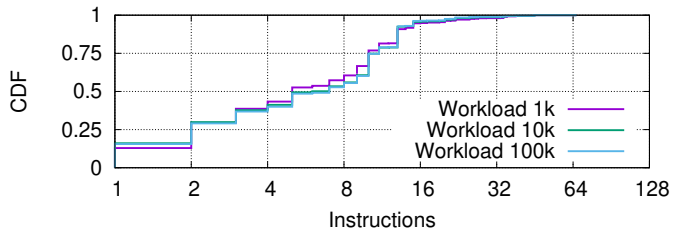

Figure 9 CDF of the number of instructions in a basic block in FAST-FAIR.

alternatives. Interestingly, as the workload increases, the size of Store IFRs becomes slightly smaller than the regions defined by stores and their explicit persistency. This is because some stores are not explicitly persisted within their respective IFRs, underscoring a fundamental distinction to PIFRs.

Figure 9 shows the number of instructions per basic block belonging to at least one PIFR. While some blocks are small, roughly half encompass a few dozen instructions. These results illustrate how PIFRs, by defining larger regions (10s–100s of instructions) that are more likely to overlap, provide a more efficient detection mechanism than existing approaches that require overlap with a single load instruction, as depicted in Figure 1.

6.4 Performance

We now investigate the trade-offs of the different sampling and scheduling configurations (§ 5). We varied the following parameters: no delay, 1ms delay without decay, and 1ms delay with 5% decay. These are tested without sampling and with a 10% sampling rate, for a total of 6 configurations. Table 2 displays the average and standard deviation of 3 executions. Due to the large number of experiments (6 configurations, 3 workloads, 12 programs, and 3 runs, for a total of 648 executions), we capped the execution time of each experiment at 90 minutes. The executions that did not finish within this time are identified with a "—" in Table 2.

The main conclusion is that sampling presents an interesting trade-off: a sample rate of 10% results in 3 to 15× faster executions and detects around 30% to 50% of the PIRs when compared with sampling disabled. Notably, the vast majority of the bugs reported in Table 1 were detected with the 10% sampling rate. This suggests that developers can use sampling as part of the build process to spot PIRs early on, and disable sampling in the nightly tests and builds. In § 8 we discuss how developers can use the output of Vardalith to narrow down the bugs.

Manipulating the thread scheduling with delay injection substantially increases testing time, as expected. The results also show that using constant delays has diminishing returns, and we attribute this to the fact that PIFRs already result in large detection windows and hence the effect of simultaneously delaying multiple concurrent threads is not beneficial. However, we note that the use of active delay injection still yields more races at the cost of increased testing time.

To illustrate that multiple PIRs can trigger the same bug, and since Bug #1 was already known, we ran a version of P-CLHT where this bug was fixed. The results show an average decrease of 5 races reported per test, which is consistent with our observation that Bug #1 was triggered by 5 different PIRs (Table 1). The takeaway is that a single fix can have a noticeable impact in the number of PIRs reported.

Table 2 PIRs detected using different Vardalith configurations. The results show the average and standard deviation of 3 testing runs. Configurations that did not complete within 90 minutes are denoted by "—".

Target	Workload	Sampling=10%						Sampling=OFF					
		Delay=0ms		Delay=1ms, Decay=0%		Delay=1ms, Decay=5%		Delay=0ms		Delay=1ms, Decay=0%		Delay=1ms, Decay=5%	
		Time (s)	Races	Time (s)	Races	Time (s)	Races	Time (s)	Races	Time (s)	Races	Time (s)	Races
P-CLHT	1k	0.7 ± 0.1	0 ± 0	1.8 ± 0.2	0 ± 0	1 ± 0.5	0 ± 0	1.5 ± 0.6	4.3 ± 1.5	1.8 ± 0.4	4 ± 1	1.6 ± 0.4	4 ± 2.6
	10k	5.4 ± 0.9	3.3 ± 1.5	7.2 ± 0.3	5 ± 2	4.7 ± 1.1	6 ± 2.6	7.7 ± 0.9	21 ± 2	9.3 ± 0.1	21 ± 1.7	7.2 ± 0.7	21 ± 2.6
	100k	42 ± 13.4	6.3 ± 2.1	63.8 ± 1.6	7 ± 1	48.4 ± 1.4	11.3 ± 3.5	78.8 ± 9.5	27.3 ± 2.1	94.2 ± 6.9	25.7 ± 3.5	87.3 ± 2.5	32 ± 1.7
P-ART	1k	0.3 ± 0	210 ± 2.2	0.3 ± 0	220 ± 3.7	0.3 ± 0	211.7 ± 5.9	0.5 ± 0	251 ± 7.3	0.5 ± 0	259.3 ± 5.2	0.5 ± 0	258.7 ± 8.3
P-Masstree	1k	0.6 ± 0	26 ± 2.2	2.2 ± 0.3	15 ± 1.4	1.1 ± 0.1	18.7 ± 2.5	0.7 ± 0	51.7 ± 5.2	2.5 ± 0	42 ± 7	1.4 ± 0.1	50 ± 1.6
	10k	9.2 ± 0.1	33.7 ± 2.9	10 ± 0	27.7 ± 1.7	9.9 ± 0.1	28 ± 2.2	3.4 ± 0.5	71 ± 3	13.8 ± 0.2	66.7 ± 5.2	9.8 ± 4.2	70 ± 3
	100k	79.6 ± 0.4	47 ± 2.2	81.9 ± 1.4	43 ± 2.2	81 ± 2	49 ± 5.7	122 ± 3.6	114.7 ± 2.6	119.8 ± 2.5	113 ± 3.3	124.9 ± 2.2	118.3 ± 1.7
SEPH	1k	37.2 ± 0.1	1 ± 0	37.2 ± 0.1	1 ± 0	37.2 ± 0.1	1 ± 0	34.3 ± 0.1	7 ± 0	34.3 ± 0.2	6 ± 0	34.3 ± 0.2	6.7 ± 0.6
	10k	37.3 ± 0.1	2 ± 0	37.3 ± 0.1	2.7 ± 0.6	37.4 ± 0.1	2.7 ± 0.6	44.1 ± 0.6	12 ± 0	43.1 ± 0.1	12 ± 0	43.5 ± 0.7	11.3 ± 0.6
	100k	115.3 ± 1	5.7 ± 0.6	115.6 ± 1	6.7 ± 0.6	115.3 ± 1	6 ± 0	1838.9 ± 7.5	14 ± 1	1843.1 ± 6.4	14 ± 0	1843.9 ± 3.2	14 ± 1
Level Hashing	1k	50.9 ± 0.3	4 ± 1	51.9 ± 1	5.7 ± 2.5	50.5 ± 0.2	5 ± 1	57.1 ± 0.7	14.7 ± 1.5	58.2 ± 0.8	14 ± 1	53.9 ± 0.7	15.3 ± 2.1
	10k	97.5 ± 0.5	17 ± 1	100.6 ± 0.9	14.7 ± 2.1	96.2 ± 0.6	16 ± 2.6	175.6 ± 7.5	18.3 ± 2.1	191.8 ± 5.6	18.3 ± 4.2	166.3 ± 2.8	22.3 ± 1.5
	100k	561.6 ± 9.9	29.7 ± 3.1	661.4 ± 6.6	29.3 ± 1.2	543 ± 2.6	33.7 ± 2.3	—	—	—	—	—	—
Clevel Hashing	1k	43.8 ± 0.3	5.7 ± 1.2	44.3 ± 0.8	4.3 ± 0.6	43.5 ± 0.4	4.7 ± 1.2	45.1 ± 0.4	22 ± 0	45.3 ± 0.9	22.3 ± 0.6	44.7 ± 0.1	22 ± 0
	10k	65.7 ± 0.8	11.3 ± 3.8	65.3 ± 0.2	10 ± 4.6	64.9 ± 0.2	7 ± 0	343.2 ± 1.8	30 ± 0	342.6 ± 1.7	30 ± 0	345.2 ± 4.4	30.3 ± 0.6
	100k	4022.7 ± 18.1	8 ± 0	4021.5 ± 4.7	7.7 ± 0.6	4029 ± 8.4	7.7 ± 1.2	—	—	—	—	—	—
CCEH	1k	40.9 ± 0.4	4.7 ± 1.5	41.6 ± 0.6	3 ± 1	40.6 ± 0.6	2.7 ± 0.6	39.9 ± 0.1	12 ± 0	40.5 ± 0	12 ± 0	40 ± 0.3	12 ± 0
	10k	50.9 ± 0.3	5.7 ± 1.5	57.1 ± 0.2	3.7 ± 1.2	51.2 ± 0.3	4 ± 1	56.8 ± 0.1	16 ± 0	59.8 ± 0.6	15 ± 0	56.9 ± 0.1	15 ± 0
	100k	148 ± 5.2	13 ± 2	210.7 ± 1.2	13 ± 3.6	153 ± 0.5	16.3 ± 2.1	226.5 ± 0.9	42 ± 1.7	268.9 ± 2.1	28 ± 4.6	227.1 ± 0.7	39.7 ± 1.5
CCEH CoW	1k	40.4 ± 0.4	5.7 ± 1.5	41.6 ± 0.2	5 ± 0	40.7 ± 0.6	5 ± 0	39.7 ± 0.2	12 ± 0	40.3 ± 0.2	12.3 ± 0.6	39.9 ± 0	12 ± 0
	10k	49.2 ± 0.5	6.3 ± 1.5	53.5 ± 0.6	6.3 ± 0.6	49.1 ± 0.1	6 ± 0	53.3 ± 0.1	13.7 ± 1.5	56.7 ± 0	15 ± 1	53 ± 0.5	13.7 ± 0.6
	100k	131.8 ± 0.4	14 ± 2	176.1 ± 0.3	12.7 ± 1.5	134.5 ± 0.6	15 ± 1	191.4 ± 0.5	34.7 ± 2.5	231.7 ± 1.3	24.3 ± 2.1	192.2 ± 0.8	34.3 ± 1.5
FAST-FAIR	1k	1.2 ± 0.3	69 ± 4.4	5.3 ± 0.3	67.3 ± 4	1.1 ± 0.1	78 ± 4.4	3.3 ± 0.8	120.3 ± 8.7	6.3 ± 0.1	112.7 ± 4	2.7 ± 0.7	132.7 ± 4
	10k	10.2 ± 1.9	80.7 ± 1.5	51.6 ± 0.8	80.3 ± 5.1	26.6 ± 0.8	77.7 ± 2.5	28.1 ± 13.9	125.3 ± 2.5	71.4 ± 0.6	118.3 ± 3.8	44 ± 4.2	131 ± 3.5
	100k	146.1 ± 20.9	125.3 ± 5.9	523.8 ± 7.9	110 ± 3	302.7 ± 3.2	114.3 ± 5.7	557.7 ± 23	186.7 ± 3.1	825.4 ± 65.3	175 ± 3	622.1 ± 40.3	183.7 ± 6.8
Memcached	1k	3.3 ± 0.1	317 ± 39.2	14 ± 0.3	269.7 ± 9.2	4.2 ± 0.4	335.3 ± 47.1	4.6 ± 0.8	435 ± 41.7	17 ± 0.1	363.3 ± 22.4	5.2 ± 0.7	422.3 ± 6.4
	10k	23.2 ± 2.4	434.7 ± 37.6	164.5 ± 2.1	434.7 ± 8.6	26.6 ± 0.1	444.3 ± 17.9	53 ± 2.7	533.7 ± 9.3	210.2 ± 5.1	544 ± 10.1	54.4 ± 1.8	536.3 ± 5.5
	100k	208 ± 2.4	570.7 ± 5	2097.4 ± 7.5	549.3 ± 6.4	213.7 ± 1.8	586.7 ± 3.2	668.1 ± 4.5	610 ± 5.3	2625.9 ± 17.8	581.3 ± 3.2	679.5 ± 7.4	615.7 ± 7.2
WIPE	1k	0.8 ± 0.1	22.7 ± 2.9	8.7 ± 0.1	14.3 ± 5.2	1.1 ± 0	20 ± 1.4	1.2 ± 0	35 ± 2.2	9.3 ± 0	32 ± 0.8	1.5 ± 0	30.3 ± 3.3
	10k	3.8 ± 0	32.3 ± 0.9	39.6 ± 0.2	22.3 ± 2.6	4.5 ± 0.1	30.7 ± 0.9	7.2 ± 0.2	42 ± 1.6	43.3 ± 0.2	37.3 ± 1.7	7.7 ± 0.1	38 ± 5
	100k	24.8 ± 1.9	82 ± 1.6	232.8 ± 57.5	57 ± 5.9	28.7 ± 4.1	66.7 ± 9.2	58.8 ± 5.7	123.7 ± 8.1	239.2 ± 37	109.7 ± 11.6	58.9 ± 9.6	124.3 ± 10.5
Apex	1k	2.7 ± 0	3 ± 2.2	3.1 ± 0.1	6.3 ± 0.9	2.8 ± 0	4 ± 1.4	2.9 ± 0	9.3 ± 1.2	3.1 ± 0	5.7 ± 3.3	2.9 ± 0	19.3 ± 1.9
	10k	3.5 ± 0	17.7 ± 2.6	5.2 ± 0	18.3 ± 2.9	3.8 ± 0.3	20 ± 3.7	4.5 ± 0	48.3 ± 7.5	8.5 ± 1.5	47.7 ± 4.5	4.7 ± 0.1	46.7 ± 5.4
	100k	12 ± 0.7	88.7 ± 3.8	15.5 ± 1.8	80 ± 6.5	12.3 ± 0.2	89.3 ± 6	28.2 ± 0.8	221.3 ± 7	39.8 ± 4.3	213.7 ± 12.7	32 ± 0.4	213 ± 8.6

Table 3 Comparison between Vardalith and PMRace. PMRace was configured with timeouts of 1, 10, and 30 minutes. Vardalith ran workloads of 1k, 10k, and 100k operations. Each experiment ran 3 times.

Tool	P-CLHT			FAST-FAIR			Memcached								
	Time (s)	Detection		Time (s)	Detection		Time (s)	Detection							
		Bug #1	Bug #2		Bug #12	Bug #13		Bug #14	Bug #15	Bug #16	Bug #17	Bug #18	Bug #19	Bug #20	Bug #21
Vardalith	1.5 ± 0.6	0/3	3/3	3.3 ± 0.8	3/3	1/3	4.6 ± 0.8	3/3	3/3	3/3	3/3	0/3	0/3	3/3	0/3
	7.7 ± 0.9	0/3	3/3	28.1 ± 13.9	3/3	0/3	53 ± 2.7	3/3	3/3	3/3	3/3	1/3	0/3	3/3	0/3
	78.8 ± 9.5	3/3	3/3	557.7 ± 23	3/3	0/3	668.1 ± 4.5	3/3	3/3	3/3	3/3	1/3	0/3	3/3	1/3
PMRace	60	0/3	0/3	60	2/3	0/3	60	0/3	0/3	2/3	2/3	2/3	2/3	2/3	0/3
	600	0/3	0/3	600	2/3	0/3	600	0/3	0/3	3/3	3/3	2/3	3/3	3/3	0/3
	1800	0/3	0/3	1800	3/3	0/3	1800	0/3	0/3	3/3	3/3	3/3	3/3	3/3	0/3

6.5 Comparison with PMRace

We compared Vardalith with PMRace [11] using P-CLHT, FAST-FAIR and Memcached. The goal of this direct comparison is to shed light on the effectiveness of PIFRs versus traditional interleaving exploration techniques. We only considered PMRace's first phase, making it an apples-to-apples comparison with Vardalith, as we detail in § 7. Following the authors' instructions, we configured PMRace with 4 workers, the full set of seeds, and timeouts of 1, 10 (the default), and 30 minutes. As for Vardalith, we used the same experiments as before, with active delay injection and sampling disabled (corresponding to the sixth column of Table 2). We ran each experiment 3 times and compared the time taken and the detection rate of each known bug.

The results are shown in Table 3. There are two main observations. First, Vardalith is more consistent than PMRace at detecting PIRs. Considering P-CLHT and FAST-FAIR, the only instances where Vardalith missed bugs were either in smaller workloads that did not exercise the buggy code paths (P-CLHT, 1k and 10k operations), and Bug #13 which is known to be hard to reach [72]. As for Memcached, Vardalith still compares favorably with PMRace: PMRace failed to detect 3 of the 8 bugs in all 9 runs, while Vardalith

only failed to detect Bug #19 in this configuration (though the bug was detected in other configurations shown in Table 2). Second, VARDALITH was significantly faster while using fewer resources than PMRace, which used four parallel workers exploring the search space.

6.6 Comparison with ThreadSanitizer

We ran all programs from Table 2 with ThreadSanitizer [86], a widely used dynamic data race detector. ThreadSanitizer uses a hybrid algorithm combining happens-before and lockset analysis to detect concurrent, unsynchronized accesses to shared memory. It tracks memory access and synchronization events (locks, unlocks, happens-before arcs) and flags pairs of accesses that are neither ordered by the happens-before relation nor protected by a common lock. However, ThreadSanitizer has no notion of PM semantics: it does not track persistence operations such as flushes and fences, and therefore cannot detect cases where accesses are properly synchronized from a concurrency standpoint but persistence operations fall outside critical sections. As expected, ThreadSanitizer did not report any of the bugs that stem from incorrectly synchronized persistence, as these do not constitute classic data races (Figure 1). In contrast, VARDALITH reports all of the PM data races found by ThreadSanitizer, in addition to the 9 PIRs that ThreadSanitizer misses.

7 Related Work

In this section, we discuss the related work in PM bug detection (§ 7.1), data race detection (§ 7.2), and interleaving exploration (§ 7.3).

7.1 PM Bug Detection

The improper utilization or absence of flush and fence instructions can result in durability, ordering, atomicity, and performance bugs, which led to the development of tools to detect them in single-threaded executions [28, 59, 58, 68, 17, 48, 31, 32, 24]. Most of these tools either lack support for concurrent executions or passively uncover PIRs by chance. Some approaches [76, 30, 33] establish stricter persistency models, flagging and/or correcting deviations from them. Robustness-based approaches [30, 33] require that every weak-persistency execution correspond to *some* strictly persistent execution, flagging any execution whose post-crash state could not have arisen from a program-order-consistent pre-crash history. These definitions, however, do not subsume our notion of PIR. In particular, PIRs additionally capture genuine bugs that robustness checks miss: cases in which a program observes unpersisted data and subsequently produces an externally visible effect other than a persistent store. Others [25, 11, 72], like VARDALITH, actively search for PM concurrency bugs.

PMRace [11] focuses on narrower PIRs (termed inter-thread inconsistencies) where the reading of non-persisted data has persistent side effects, whereas VARDALITH considers any side effects. It works in two phases. It uses PM-aware fuzzing tailored to key-value store semantics, along with interleaving exploration that selects pairs of accesses to the same target and injects delays to ensure that the read operation takes place right after the store but before it is persisted. This is similar to the scheduling strategy we discussed in § 5 and evaluated in § 6.3. This approach requires many runs to find PIRs, hindering the scalability of the system to large workloads. The second phase validates if the PIRs detected correspond to bugs, or if the behaviour is tolerated by the program, by checking if PIR-causally dependent PM stores are overwritten by the recovery. However, this technique only covers some recovery mechanisms, so PMRace still reports recoverable races in many instances [11].

Durinn [25] reasons about key-value store operations rather than PM accesses. The approach involves defining 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. It works by trying to force data to be read while the writer is between the LP and the DP for a given operation. To that end, Durinn generates a synchronous trace from a predefined workload and identifies pairs of potentially racy operations. It then reorders the operations in the workload to force one thread to read another's unpersisted data, crashes the application, and checks for consistency using an approach similar to output equivalence checking [24]. Despite being highly effective, Durinn is tied to key-value store semantics and requires hand-written test drivers, limiting its applicability to other types of PM applications.

HawkSet [72] is a recent work that uses lockset analysis [83] to detect PIRs. While fast and scalable, HawkSet inherits the usual limitations of lockset analysis: it relies on locks as the only synchronization mechanism and can flag pairs of accesses that cannot actually occur concurrently in practice. Vardalith, in turn, follows a principled approach that does not depend on any specific synchronization mechanism and is formally proven correct, ensuring it only reports genuine races.

7.2 Data Race Detection

The literature in data race detection is vast and spans static [8, 74] and dynamic [83, 23, 86, 81, 20, 43, 80] approaches. Static detectors predict races without running the application but lack precision whereas dynamic approaches observe execution traces to detect actual races. Happens-before analysis [47, 23, 86] imposes a strict partial order on events based on program and synchronization order, which is conservative. However, HB analyses do not reason about persistency actions, and thus cannot capture non-persistency windows that define PIRs. Partial-order-based predictive search [80, 43, 81] achieves a greater detection rate with comparable performance. IFRit [20] uses Interference-Free Regions to extend the race detection window and thus increases the probability of detecting races. However, none of these approaches can detect PIRs that result from persisting outside the critical section.

7.3 Interleaving Exploration

Systematic testing [65, 27] explores all possible thread interleavings but does not scale. Partial-order reduction [26, 1, 44] soundly reduces the number of interleavings to explore by skipping equivalent executions. Randomized testing [9, 100] employs probabilistic scheduling but sacrifices completeness and tends to explore equivalent interleavings. Delay injection influences scheduling without controlling it. Some approaches [22] perform lightweight analysis and inject delays probabilistically at many locations, while others perform extensive analysis to select few locations [85, 73]. Active delay injection [54, 88] offers a middle ground by selecting many locations at the start and dynamically adjusting their probabilities based on their effectiveness. However, in Vardalith, since PIFRs already extend the race detection window, the impact of these techniques is reduced, as illustrated in § 6.3.

8 Discussion

In this section, we discuss the current limitations of Vardalith (§ 8.1), how developers can efficiently use Vardalith's output (§ 8.2), the fundamental differences to classical race detection tools (§ 8.3), and Vardalith's expected impact as PM technology evolves (§ 8.4).

8.1 Limitations

All races reported by VARDALITH are true PIRs– that is, VARDALITH guarantees that whenever it reports a bug, there exists an interleaving in which a non-persisted write is read by a concurrent thread. However, the application under analysis may include recovery mechanisms that restore a consistent state after a crash, depending on its semantics. Existing tools do not handle this well: HawkSet [72] simply ignores recovery, with the added problem that lockset analysis can flag pairs of accesses that cannot even occur concurrently in practice; PMRace [11] accounts for some recovery mechanisms but still reports benign races in many instances [11]. VARDALITH explores a different point in the design space: it takes care of the hard part – proving that a concurrent interleaving witnessing the race exists – and leaves the developer with the easier job of checking whether recovery covers it.

We assume threads persist their own stores and do not consider the *flush-on-read* pattern [93], where a thread that reads from PM flushes the loaded value before performing any operation that depends on it. Our choice is consistent with most PM applications we are aware of, including all programs we and the state-of-the-art analyze [37, 67, 92, 50, 52, 96, 61, 105, 10], which rely on writers to flush. One likely reason is cost: having readers flush can dominate runtime when readers vastly outnumber writers, even though recent work proposes mitigations [97]. Alternative persistency models, such as epochs [98], propose writing to intermediate structures that are asynchronously persisted in the background. Similarly to the state-of-the-art [72, 25, 11], we do not consider these alternative models.

8.2 From Detection to Bug Resolution

VARDALITH outputs two artifacts: a list of overlapping PIFR identifiers and detailed PIFR backtraces. Based on our experience in analyzing the PM programs from § 6, we developed the following methodology to translate these outputs into actionable bug fixes:

1. *Prioritization:* Sort PIFR pairs to identify the most prevalent race patterns, which often indicate systemic issues rather than isolated bugs.
2. *Filter Benign Cases:* Exclude known benign races that represent intentional design choices. For example, in Memcached, $\approx 60\%$ of the races involve unpersisted stores to the LRU cache fields and bitwise operations on flags that are rebuilt after crashes.
3. *Focus on Critical Paths:* Analyze remaining races in important code sections, particularly those involving resizing of data structures, which is often linked to data loss bugs, as exemplified by Bugs #1, #9, #10, #12, #13, and #24 from Table 1.

This approach proved effective despite our initial unfamiliarity with the codebases. Developers with domain knowledge should be able to identify critical issues even more efficiently.

8.3 Beyond Traditional Race Detection

While some PIRs are also traditional data races, many critical PM bugs occur despite proper synchronization, when persistence operations happen outside critical sections. In such cases, traditional data race detectors like ThreadSanitizer [86] are ineffective, as shown in § 6.3. This gap is fundamental: PIRs arise from the interaction between concurrency and persistency semantics, which traditional detectors simply do not model. Of the 26 bugs VARDALITH detected, 9 are not traditional data races and would be missed by such tools.

Similarly, even though PIFRs draw inspiration from Interference-Free Regions (IFRs) [19], the differences between volatile and PM semantics make IFRs inadequate for PIR detection. Computing PIFRs is substantially harder than computing IFRs. An IFR surrounds a single

memory access delimited by the nearest acquire and release. A *Store PIFR*, in contrast, must additionally reason about the persistency operation that pertains to the enclosed store, since the region extends until the first release after explicit persistence. This requires value-level reasoning to match flushes to prior stores by address range, which is already non-trivial within a single function and is compounded by the fact that persistency operations are frequently encapsulated in auxiliary functions or separate procedures, demanding interprocedural analysis that is provably expensive or even impossible in some cases [79], as discussed in § 4. Our hybrid design addresses this by under-approximating PIFR starts intraprocedurally via static analysis and deferring the precise determination of PIFR ends to the dynamic component.

8.4 Relevance in Emerging PM Technologies

Despite Intel Optane's discontinuation [15], PM remains highly relevant as the industry moves towards CXL-based solutions [12, 16, 87]. CXL is rapidly gaining traction in both academia and industry [55, 101, 103], with CXL Type 3 [87] introducing explicit support for PM. Multiple vendors have announced upcoming PM devices for the near future [82, 63, 70], indicating industry commitment to these technologies.

Based on the available information, we expect CXL-based PM devices to present similar semantic challenges to Optane. In particular, given the intrinsic differences in access latency and capacity among the cache, RAM, and PM: i) any memory-mapped persistent storage solution must rely on CPU caches for performance, and ii) the hardware must continue to provide explicit primitives to control the transitions between volatile and persistent domains. As a result, the decoupling between visibility and durability remains, meaning PIRs can still occur. Therefore, developers must still handle these transitions carefully, as with Optane.

At the same time, our paper assumes a crash-consistency model and does not explicitly model the partial failures that may arise in some CXL settings. We view these directions as complementary: Vardalith focuses on efficiently detecting PIRs, while other approaches can extend the failure model to reason about partial failures more explicitly [31].

Overall, Vardalith's approach is particularly well-suited to this evolving landscape. By operating at the binary level and relying only on a few simple and fundamental primitives common across implementations, our approach can adapt to diverse vendor-specific devices and libraries.

9 Conclusion

This paper presents Vardalith, a tool to detect persistency-induced races in PM applications. Our key innovation lies in the concept of Persistent Interference-Free Regions (PIFRs), which significantly expand the detection window around memory accesses to precisely identify races even when they are not directly observable in a particular execution. This addresses the fundamental limitations of existing approaches that require exhaustive interleaving exploration, application-specific knowledge, or yield many false positives – all of which limit their practical applicability.

We propose a hybrid approach that combines static and dynamic analysis and formally prove its correctness. Our experimental evaluation shows Vardalith's effectiveness in discovering 26 bugs (7 previously unknown) across several PM applications.

References

1. Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, POPL '14, pages 373–384, New York, NY, USA, January 2014. ACM. doi:10.1145/2535838.2535845.
2. S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996. doi:10.1109/2.546611.
3. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986. URL: https://www.worldcat.org/oclc/12285707.
4. Frances E. Allen. Interprocedural data flow analysis. In Jack L. Rosenfeld, editor, *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, pages 398–402. North-Holland, 1974.
5. Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and availability via client-local NVM in a distributed file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1011–1027. USENIX Association, November 2020. URL: https://www.usenix.org/conference/osdi20/presentation/anderson.
6. Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. Codesurfer/x86—a platform for analyzing x86 executables. In *International conference on compiler construction*, pages 250–254. Springer, 2005. doi:10.1007/978-3-540-31985-6_19.
7. Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*, pages 5–23. Springer, 2004. doi:10.1007/978-3-540-24723-4_2.
8. Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. RacerD: compositional static race detection. *Proc. ACM Program. Lang.*, 2(OOPSLA):144:1–144:28, 2018. doi:10.1145/3276514.
9. Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In James C. Hoe and Vikram S. Adve, editors, *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, ASPLOS XV, pages 167–178, New York, NY, USA, 2010. ACM. doi:10.1145/1736020.1736040.
10. Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 799–812. USENIX Association, July 2020. URL: https://www.usenix.org/conference/atc20/presentation/chen.
11. Zhangyu Chen, Yu Hua, Yongle Zhang, and Luochangqi Ding. Efficiently detecting concurrency bugs in persistent memory programs. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, ASPLOS '22, pages 873–887. ACM, 2022. doi:10.1145/3503222.3507755.
12. CXL Consortium. Compute Express Link. Accessed April 2026. URL: https://computeexpresslink.org/.
13. Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC'10, pages 143–154, 2010. doi:10.1145/1807128.1807152.
14. Intel Corporation. eADR: New opportunities for persistent memory applications. Accessed April 2025. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html.

15 Intel Corporation. Intel optane dc persistent memory product brief. Accessed April 2026. URL: https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf.

16 Debendra Das Sharma, Robert Blankenship, and Daniel Berger. An introduction to the compute express link (cxl) interconnect. *ACM Comput. Surv.*, 56(11), July 2024. doi:10.1145/3669900.

17 Bang Di, Jiawen Liu, Hao Chen, and Dong Li. Fast, flexible, and comprehensive bug detection for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 503–516, 2021. doi:10.1145/3445814.3446744.

18 Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, PLDI 2020, pages 151–163, New York, NY, USA, 2020. ACM. doi:10.1145/3385412.3385972.

19 Laura Effinger-Dean, Hans-Juergen Boehm, Dhruva R. Chakrabarti, and Pramod G. Joisha. Extended sequential reasoning for data-race-free programs. In Jeffrey S. Vetter, Madanlal Musuvathi, and Xipeng Shen, editors, *Proceedings of the 2011 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with PLDI '11, San Jose, CA, USA, June 5, 2011*, MSPC '11, pages 22–29, New York, NY, USA, 2011. ACM. doi:10.1145/1988915.1988922.

20 Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-Juergen Boehm. Ifrit: interference-free regions for dynamic data-race detection. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, volume 47(10) of *OOPSLA '12*, pages 467–484, New York, NY, USA, October 2012. ACM. doi:10.1145/2384616.2384650.

21 Dawson R. Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In Michael L. Scott and Larry L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, volume 37(5), pages 237–252, New York, NY, USA, 2003. ACM. doi:10.1145/945445.945468.

22 John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, OSDI'10, pages 151–162, USA, 2010. USENIX Association. URL: http://www.usenix.org/events/osdi10/tech/full_papers/Erickson.pdf.

23 Cormac Flanagan and Stephen N Freund. FastTrack: Efficient and precise dynamic race detection. *Communications of the ACM*, 53(11):93–101, 2010. doi:10.1145/1839676.1839699.

24 Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, SOSP '21, pages 100–115. ACM, 2021. doi:10.1145/3477132.3483556.

25 Xinwei Fu, Dongyoon Lee, and Changwoo Min. DURINN: Adversarial memory and thread interleaving for detecting durable linearizability bugs. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 195–211. USENIX Association, 2022. URL: https://www.usenix.org/conference/osdi22/presentation/fu.

26 Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 1996. doi:10.1007/3-540-60761-7.

27 Patrice Godefroid. Model checking for programming languages using verisoft. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, POPL '97, pages 174–186, New York, NY, USA, 1997. ACM Press. doi:10.1145/263699.263717.

28 João Gonçalves, Miguel Matos, and Rodrigo Rodrigues. Mumak: Efficient and black-box bug detection for persistent memory. In Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan, editors, *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, EuroSys '23, pages 734–750, New York, NY, USA, 2023. ACM. doi:10.1145/3552326.3587447.

29 João Gonçalves. task3r/mumak. Software, swhId: swh:1:dir:8c7962c00482101579ad30f09cfeec6ee0a864bd (visited on 2026-06-11). URL: https://github.com/task3r/mumak, doi:10.4230/artifacts.26595.

30 Hamed Gorjiara, Weiyu Luo, Alex Lee, Guoqing Harry Xu, and Brian Demsky. Checking robustness to weak persistency models. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, PLDI 2022, pages 490–505, New York, NY, USA, 2022. ACM. doi:10.1145/3519939.3523723.

31 Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Jaaru: efficiently model checking persistent memory programs. In Tim Sherwood, Emery D. Berger, and Christos Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, ASPLOS '21, pages 415–428. ACM, 2021. doi:10.1145/3445814.3446735.

32 Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Yashme: detecting persistency races. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, ASPLOS '22, pages 830–845. ACM, 2022. doi:10.1145/3503222.3507766.

33 Zhilei Han and Fei He. Robustness verification for checking crash consistency of non-volatile memory. In Lieven Eeckhout, Georgios Smaragdakis, Kaitai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach, editors, *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, ASPLOS '25, pages 955–969, New York, NY, USA, 2025. ACM. doi:10.1145/3669940.3707269.

34 Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, PLDI '07, pages 290–299, New York, NY, USA, June 2007. ACM. doi:10.1145/1250734.1250767.

35 Yibo Huang, Haowei Chen, Newton Ni, Yan Sun, Vijay Chidambaram, Dixin Tang, and Emmett Witchel. Tigon: A distributed database for a CXL pod. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, Boston, MA, July 2025. USENIX Association. URL: https://www.usenix.org/conference/osdi25/presentation/huang-yibo.

36 Yibo Huang, Newton Ni, Vijay Chidambaram, Emmett Witchel, and Dixin Tang. Pasha: An efficient, scalable database architecture for CXL pods. In *15th Conference on Innovative Data Systems Research, CIDR 2025, Amsterdam, The Netherlands, January 19-22, 2025*. www.cidrdb.org, 2025.

37 Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in Byte-Addressable persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, Oakland, CA, February 2018. USENIX Association. URL: https://www.usenix.org/conference/fast18/presentation/hwang.

38 Zhicheng Ji, Kang Chen, Leping Wang, Mingxing Zhang, and Yongwei Wu. Falcon: Fast OLTP engine for persistent cache and non-volatile memory. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, SOSP '23, pages 531–544, New York, NY, USA, 2023. ACM. doi:10.1145/3600006.3613141.

39 Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. Winefs: a hugepage-aware file system for persistent memory that ages gracefully. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, SOSP '21, pages 804–818, New York, NY, USA, 2021. ACM. doi:10.1145/3477132.3483567.

40 Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: reducing software overhead in file systems for persistent memory. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, SOSP '19, pages 494–508, New York, NY, USA, 2019. ACM. doi:10.1145/3341301.3359631.

41 Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for nonvolatile memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, Boston, MA, July 2018. USENIX Association. URL: https://www.usenix.org/conference/atc18/presentation/kannan.

42 Sun Hyoung Kim, Dongrui Zeng, Cong Sun, and Gang Tan. Binpointer: Towards precise, sound, and scalable binary-level pointer analysis. In *CC 2022 - Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, CC 2022, pages 169–180. Association for Computing Machinery, Inc, 2022. doi:10.1145/3497776.3517776.

43 Dileep Kini, Umang Mathur, and Mahesh Viswanathan. Dynamic race prediction in linear time. In *PLDI 2017 - Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 157–170. Association for Computing Machinery, 2017. doi:10.1145/3062341.3062374.

44 Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. Truly stateless, optimal dynamic partial order reduction. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. doi:10.1145/3498711.

45 SKKU/UNIST Data Intensive Computing Lab. Dicl/fast_fair: Endurable transient inconsistency in byte-addressable persistent b+-tree. Accessed April 2026. URL: https://github.com/DICL/FAST_FAIR/tree/0f047e88eb5005afcd02ccb2a7c1289e17702cc3.

46 VT COSMOSS Lab. cosmoss-jigu/durinn: Osdi'22 durinn artifact, recipe implementation. Accessed April 2026. URL: https://github.com/cosmoss-jigu/durinn/tree/d5aaa1cdeeae50f6b8d828fd93e991ba9eefb995/third_party/RECIPE.

47 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978. doi:10.1145/359545.359563.

48 Philip Lantz, Dulloor Subramanya Rao, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In Garth Gibson and Nickolai Zeldovich, editors, *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014, Philadelphia, PA, USA, June 19-20, 2014*, ATC '14, pages 433–438. USENIX Association, 2014. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/lantz.

49 Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004. doi:10.1109/CGO.2004.1281665.

50 Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 462–477. ACM, 2019. doi:10.1145/3341301.3359635.

51 Sekwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K. Aguilera, Kimberly Keeton, and Vijay Chidambaram. Dinomo: An elastic, scalable, high-performance key-value store for disaggregated persistent memory. *Proc. VLDB Endow.*, 15(13):4023–4037, 2022. doi:10.14778/3565838.3565854.

52 Lenovo. Lenovo: Lenovo modifications to linux memcached for enhanced persistent memory support. Accessed April 2026. URL: https://github.com/lenovo/memcached-pmem.

53 Lenovo. Lenovo: Lenovo modifications to linux memcached for enhanced persistent memory support - version 8f121f6. Accessed April 2026. URL: https://github.com/lenovo/memcached-pmem/tree/8f121f6cef6b79560be60bd59ad77f78ed75f034.

54 Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, SOSP '19, pages 162–180, New York, NY, USA, 2019. ACM. doi:10.1145/3341301.3359638.

55 Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, ASPLOS '23, pages 574–587, New York, NY, USA, 2023. ACM. doi:10.1145/3575693.3578835.

56 Ruibin Li, Xiang Ren, Xu Zhao, Siwei He, Michael Stumm, and Ding Yuan. ctFS: Replacing file indexing with hardware memory translation through contiguous file allocation for persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 35–50, Santa Clara, CA, February 2022. USENIX Association. URL: https://www.usenix.org/conference/fast22/presentation/li.

57 Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In Yunji Chen, Olivier Temam, and John Carter, editors, *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, ASPLOS '17, pages 329–343, New York, NY, USA, 2017. ACM. doi:10.1145/3037697.3037714.

58 Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 1187–1202, 2020. doi:10.1145/3373376.3378452.

59 Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 411–425, 2019. doi:10.1145/3297858.3304015.

60 Baotong Lu. APEX: high-performance, concurrent indexes for persistent memory. Accessed April 2026. URL: https://github.com/baotonglu/apex/tree/5aee22aa6a6059e161aa2aca6e4080118aa246e9.

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

62 Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, PLDI '05, pages 190–200. ACM, 2005. doi:10.1145/1065010.1065034.

63 Chris Mellor. Numemory reinvents optane storage-class memory. Accessed April 2026. URL: https://blocksandfiles.com/2024/10/07/numemory-reinvents-optane-storage-class-memory/.

64 Memory and The Chinese University of Hong Kong Storage Systems Research Group. cuhk-mass/seph: Scalable, efficient, and predictable hashing on persistent memory. Accessed April 2026. URL: https://github.com/cuhk-mass/SEPH/tree/258121e1c317798216335c2f27705a4993b75b68.

65 Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, OSDI'08, pages 267–280, USA, 2008. USENIX Association. URL: http://www.usenix.org/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf.

66 Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with WHISPER. In Yunji Chen, Olivier Temam, and John Carter, editors, *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, ASPLOS '17, pages 135–148, New York, NY, USA, April 2017. ACM. doi:10.1145/3037697.3037730.

67 Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. Write-Optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, Boston, MA, February 2019. USENIX Association. URL: https://www.usenix.org/conference/fast19/presentation/nam.

68 Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, OSDI '20, pages 1047–1064. USENIX Association, 2020. URL: https://www.usenix.org/conference/osdi20/presentation/neal.

69 Ian Neal, Gefei Zuo, Eric Shiple, Tanvir Ahmed Khan, Youngjin Kwon, Simon Peter, and Baris Kasikci. Rethinking file mapping for persistent memory. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 97–111. USENIX Association, February 2021. URL: https://www.usenix.org/conference/fast21/presentation/neal.

70 Neumonda. Neumonda and ferroelectric memory company collaborate in the commercialization of non-volatile dram. Accessed April 2026. URL: https://www.neumonda.com/neumonda-and-ferroelectric-memory-company-collaborate-in-the-commercialization-of-non-volatile-dram/.

71 Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999. doi:10.1007/978-3-662-03811-6.

72 João Oliveira, João Gonçalves, and Miguel Matos. Hawkset: Automatic, application-agnostic, and efficient concurrent PM bug detection. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, EuroSys '25, pages 1092–1108, New York, NY, USA, 2025. ACM. doi:10.1145/3689031.3717477.

73 Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In Mary Lou Soffa and Mary Jane Irwin, editors, *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, ASPLOS XIV, pages 25–36, New York, NY, USA, 2009. ACM. doi:10.1145/1508244.1508249.

74 Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: Practical static race detection for c. *ACM Trans. Program. Lang. Syst.*, 33(1), January 2011. doi:10.1145/1889997.1890000.

75 Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency Semantics of the Intel-x86 Architecture. *Proc. ACM Program. Lang.*, 4(POPL):11:1–11:31, 2020. doi:10.1145/3371079.

76 Benjamin Reidys and Jian Huang. Understanding and detecting deep memory persistency bugs in NVM programs with deepmc. In Jaejin Lee, Kunal Agrawal, and Michael F. Spear, editors, *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, PPoPP'22, pages 322–336. ACM, 2022. doi:10.1145/3503221.3508427.

77 Jinglei Ren, Jishen Zhao, Samira Manabi Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. Thynvm: enabling software-transparent crash consistency in persistent memory systems. In Milos Prvulovic, editor, *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, MICRO-48, pages 672–685, New York, NY, USA, 2015. ACM. doi:10.1145/2830772.2830802.

78 Yujie Ren, Changwoo Min, and Sudarsun Kannan. CrossFS: A cross-layered Direct-Access file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 137–154. USENIX Association, November 2020. URL: https://www.usenix.org/conference/osdi20/presentation/ren.

79 Thomas Reps. On the sequential nature of interprocedural program-analysis problems. *Acta Informatica*, 33:739–757, 1996. doi:10.1007/BF03036473.

80 Jake Roemer, Kaan Genç, and Michael D Bond. High-coverage, unbounded sound predictive race detection. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '18, pages 374–389. Association for Computing Machinery, 2018. doi:10.1145/3192366.3192385.

81 Jake Roemer, Kaan Genç, and Michael D Bond. SmartTrack: Efficient predictive race detection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 747–762, 2020. doi:10.1145/3385412.3385993.

82 Samsung. Samsung demonstrates new cxl capabilities and introduces new memory module for scalable, composable disaggregated infrastructure at memcon 2024. Accessed April 2026. URL: https://semiconductor.samsung.com/news-events/news/samsung-demonstrates-new-cxl-capabilities-and-introduces-new-memory-module-for-scalable-composable-disaggregated-infrastructure-at-memcon-2024/.

83 Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997. doi:10.1145/265924.265927.

84 Steve Scargall. *Persistent Memory Architecture*, pages 11–30. Apress, Berkeley, CA, 2020. doi:10.1007/978-1-4842-4932-1_2.

85 Koushik Sen. Race directed random testing of concurrent programs. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, PLDI '08, pages 11–21, New York, NY, USA, 2008. ACM. doi:10.1145/1375581.1375584.

86 Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer - Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1791194.1791203.

87 Debendra Das Sharma and Ishwar Agarwal. Compute Express Link 3.0. Technical report, CXL Consortium, August 2022. URL: https://www.computeexpresslink.org/_files/ugd/0c1418_a8713008916044ae9604405d10a7773b.pdf.

88 Bogdan Alexandru Stoica, Shan Lu, Madanlal Musuvathi, and Suman Nath. WAFFLE: exposing memory ordering bugs efficiently with active delay injection. In Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan, editors, *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, EuroSys '23, pages 111–126, New York, NY, USA, 2023. ACM. doi:10.1145/3552326.3567507.

89 UT Systems and Storage Lab. utsaslab/recipe: high-performance, concurrent indexes for persistent memory. Accessed April 2026. URL: https://github.com/utsaslab/RECIPE/tree/70bf21c6240327f4f8fac343aba708f194fe19f4/P-CLHT.

90 Robert Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974. doi:10.1137/0203006.

91 Lukas Vogel, Alexander van Renen, Satoshi Imamura, Jana Giceva, Thomas Neumann, and Alfons Kemper. Plush: A write-optimized persistent log-structured hash-table. *Proc. VLDB Endow.*, 15(11):2895–2907, July 2022. doi:10.14778/3551793.3551839.

92 Chao Wang, Junliang Hu, Tsun-Yu Yang, Yuhong Liang, and Ming-Chang Yang. SEPH: Scalable, efficient, and predictable hashing on persistent memory. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 479–495, Boston, MA, July 2023. USENIX Association. URL: https://www.usenix.org/conference/osdi23/presentation/wang-chao.

93 Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 461–472, 2018. doi:10.1109/ICDE.2018.00049.

94 Yu Chen Wang, Angela Demke Brown, and Ashvin Goel. Integrating non-volatile main memory in a deterministic database. In Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan, editors, *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, EuroSys '23, pages 672–686, New York, NY, USA, 2023. ACM. doi:10.1145/3552326.3567494.

95 Zhonghua Wang. WIPE implementation. Accessed April 2026. URL: https://github.com/olemon111/WIPE/tree/029cfc481b4a9fda02f418c1551494d56f0ef537.

96 Zhonghua Wang, Chen Ding, Fengguang Song, Kai Lu, Jiguang Wan, Zhihu Tan, Changsheng Xie, and Guokuan Li. WIPE: A write-optimized learned index for persistent memory. *ACM Trans. Archit. Code Optim.*, 21(2), February 2024. doi:10.1145/3634915.

97 Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch, and Erez Petrank. Flit: a library for simple and efficient persistent algorithms. In Jaejin Lee, Kunal Agrawal, and Michael F. Spear, editors, *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, PPoPP '22, pages 309–321, New York, NY, USA, 2022. ACM. doi:10.1145/3503221.3508436.

98 Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L. Scott. A fast, general system for buffered persistent data structures. In *50th International Conference on Parallel Processing*, ICPP'21. ACM, 2021. doi:10.1145/3472456.3472458.

99 Fangnuo Wu, Mingkai Dong, Gequan Mo, and Haibo Chen. Treesls: A whole-system persistent microkernel with tree-structured state checkpoint on NVM. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, SOSP '23, pages 1–16, New York, NY, USA, 2023. ACM. doi:10.1145/3600006.3613160.

100 Xinhao Yuan, Junfeng Yang, and Ronghui Gu. Partial order aware concurrency sampling. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 317–335, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-96142-2_20.

101 Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, SOSP '23, pages 658–674, New York, NY, USA, 2023. ACM. doi:10.1145/3600006.3613135.

102 Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. Bda: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi:10.1145/3360563.

103 Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing memory tiers with cxl in virtualized environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, OSDI '24, pages 37–56, Santa Clara, CA, July 2024. USENIX Association. URL: `https://www.usenix.org/conference/osdi24/presentation/zhong-yuhong`.

104 Diyu Zhou, Vojtech Aschenbrenner, Tao Lyu, Jian Zhang, Sudarsun Kannan, and Sanidhya Kashyap. Enabling high-performance and secure userspace NVM file systems with the trio architecture. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, SOSP '23, pages 150–165, New York, NY, USA, 2023. ACM. `doi:10.1145/3600006.3613171`.

105 Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'18, pages 461–476. USENIX Association, 2018. URL: `https://www.usenix.org/conference/osdi18/presentation/zuo`.

106 Zyantific. Fast and lightweight x86/x86-64 disassembler and code generation library. Accessed November 2023. URL: `https://github.com/zyantific/zydis`.

A New Bugs

Table 4 Extended description of the new bugs detected by VARDALITH and an explanation for why the recovery procedures do not solve the issue.

Target	#	Detailed Description	Recovery Mechanisms
P-CLHT	2	In `clht_gc_collect_cond`, GC advances the persistent list by setting `hashtable->ht_oldest` and `hashtable->version_min` after freeing old tables (`clht_gc.c:186-199`). These pointer/metadata updates are regular stores without the corresponding explicit persistence. A crash after `clht_gc_free` (which frees the old table and its buckets) but before `ht_oldest` is persisted leaves the root object pointing to a freed table version. After restart, the next GC walk starts from that stale `ht_oldest`, re-frees the same table/buckets, and corrupts allocator metadata (double-free) and potentially neighboring objects.	Recovery on startup only reinitializes locks and reuses the persisted root (`clht_create` in `clht_lb_res.c:251-272`), it does not rebuild the version list or validate `ht_oldest`.
Level Hashing	8	`LevelHashing::Insert` scans both level arrays for an existing key by reading `token[j]` and `slot[j].key` without holding the corresponding bucket lock for the second index (`level.hpp:328-352`). A concurrent insert sets `token[j]=1` after persisting the slot (`level.hpp:381-388` and `level.hpp:408-414`), but the persistence is delayed, meaning another thread can observe the `token==1` and conclude the key already exists, returning `-1`, but if a crash occurs before the `token` flush, the key is absent on recovery, yet the failed insert was already rejected. This results in an inconsistent observation pre and post-crash.	The recovery routine only cleans up resizing artifacts (`level.hpp:149-165`) and does not rescan or repair tokens or revalidate uniqueness. Thus the missed insert is permanent after restart.
CCEH	9	Lookups/inserts read the directory pointer without a directory lock (`old_sa = dir->sa` at `cceh.hpp:561` and `cceh.hpp:758`) and only retry if the pointer changes before the actual operation is performed. During directory doubling, a new segment array is built and `dir->sa` is swapped inside a PMDK transaction (`cceh.hpp:658-685`). Since PMDK transactions do not guarantee opacity or isolation, the window between the pointer becoming visible in cache and the new directory being fully persisted allows readers to use an unpersisted directory to select a segment and insert/update data. After a crash, if the swap is not persisted, the directory will not contain those entries, leading to lost inserts/updates.	Recovery (`cceh.hpp:600-634`) only resets locks, frees `new_sa`, and reconstructs the directory mapping from whatever `dir->sa` points to. It does not fall back to the old directory nor validate that the new directory is fully persisted, so the lost updates are not detected or repaired.
CCEH CoW	10	In the CoW variant, inserts read `dir->sa` without a directory lock (`cceh_cow.hpp:737-741`). Directory doubling writes a new array, assigns split segment pointers, then swaps `dir->sa` in a transaction (`cceh_cow.hpp:633-663`). As with CCEH, readers can observe the new `dir->sa` before the new array is fully persisted and use it to route inserts/updates. A crash in this window leaves the persistent directory missing the entries, resulting in permanent data loss.	Recovery (`cceh_cow.hpp:574-608`) only clears logs, frees `new_sa`, and re-derives patterns from the existing directory. It does not validate the durability of the directory swap, and therefore, it cannot recover inserts routed through an unpersisted directory.
	11	`Update` writes `dir_->_[slot].value = v_ptr` and then persists the pair (`cceh_cow.hpp:852-854`). `Get` is lock-free and returns the value directly (`cceh_cow.hpp:930-949`). A `Get` concurrent with an `Update` can observe the updated value before it is persisted, meaning that if a crash occurs before the flush, the durable state still contains the old value, but the `Get` returned the new value. This violates causality and can lead to stale data after restart.	The recovery logic does not check versioning or roll back partially persisted updates. It only resets directory metadata and log buffers (`cceh_cow.hpp:574-608`), thus it cannot detect or undo the transient value that was observed.
Memcached	14	For a `prepend` operation, `_store_item_copy_data` copies the payload with `pmem_memcpy_nodrain` (`memcached.c:2823-2824`), which reads from the in-memory value of `old_it`. A concurrent `incr/decr` can update the same item in-place via `memcpy` into `ITEM_data(it)` without explictly flushing (`memcached.c:4292`). If `prepend` runs after that in-place update but before it has been persisted, it will copy the transient bytes and then persist the new item on link (`items.c:514-522`), making a non-durable update durable via `prepend`. After a crash, the value reflects an update that never reached PM through its own operation.	Recovery for persistent slabs (`pslab_do_recover` in `pslab.c:173-268`) only walks items and relinks those marked `ITEM_LINKED`. It does not validate or recompute item payloads or reconcile them with logged operations. Since `prepend` persisted the copied bytes, recovery treats them as authoritative and cannot detect the causality violation.
	15	Same interleaving as Bug #14, but the unpersisted bytes are the padding region written by the in-place `memset` in `incr/decr` (`memcached.c:4293`). The `prepend` path reads this in-memory data and persists it in the new item (`memcached.c:2823-2824`, `items.c:514-522`), so a crash can leave durable data that depends on an update which never reached PM.	Recovery uses the same slab walk and relink logic (`pslab.c:173-268`) and does not rederive or validate the value bytes, so it cannot fix the persisted but non-durable padding written via the transient `memset`.