Automated Insertion of Flushes and Fences for Persistency

Yutong Guo
University of California, Irvine
yutong4@uci.edu

Weiyu Luo
University of California, Irvine
weiyu7@uci.edu

Brian Demsky
University of California, Irvine
bdemsky@uci.edu

***Abstract*—CXL shared memory and persistent memory allow the contents of memory to persist beyond crashes. Stores to persistent or CXL memory are typically not immediately made persistent; developers must manually flush the corresponding cache lines to force the data to be written to the underlying storage. Correctly using flush and fence operations is known to be challenging. While state-of-the-art tools can find missing flush instructions, they often require bug-revealing test cases. No existing tools can ensure the absence of missing flush bugs.**

In this paper, we present PMRobust, a compiler that automatically inserts flush and fence operations to ensure that code using persistent memory is free from missing flush and fence bugs. PMRobust employs a novel static analysis with optimizations that target newly allocated objects. We have evaluated PMRobust on persistent memory libraries and several persistent memory data structures and measured a geometric mean overhead of 0.26% relative to the original benchmarks with hand-placed flush and fence operations.

I. Introduction

In several different contexts, hardware designers have developed systems in which the contents of memory can survive system crashes. Compute Express Link (CXL) is a new open standard that enables cache coherent shared memory across a network. CXL also supports memory interface to persistent storage [43]. Finally, the contents of battery-backed NVDIMM's can survive power outages.

In all of these contexts, the in-memory data manipulated by a computation can survive machine crashes, and it can be desirable to ensure that key data structures are crash consistent so that they can be safely accessed after crashes. Achieving crash consistency is complicated by the volatility of CPU caches, necessitating explicit flush and fence instructions to persist data. [1] Software developers must use special flush and fence instructions to force data to be written back to the underlying memory.

There is a body of work [20], [22], [37] on finding bugs in persistent memory programs that utilize flushes and fences. These tools range from model checkers [30], [21] to various dynamic bug finding tools [36], [29], [39]. The prevalence of such bugs, with 183 new bugs reported by various tools [21], [20], [16], [13], [36], [37], [39] in a small expert-written program set, underscores the difficulty of manual management and motivates automating flush/fence insertion via compilers.

[1]CXL can optionally use energy storage to implement flush on failure, though this requires support by both the device and system components. This is not a panacea; many failure modes would still result in lost cache lines.

We developed a new tool that automatically generates the necessary flush and fence operations. Strict persistency [40] ensures that the "persistency memory order is identical to volatile memory order". The observation here is that bugs caused by missing flush and fence operations can be eliminated by making stores become persistent in the same order that they become visible to other threads.

Enforcing strict persistency is expensive as it requires threads to stall frequently to wait for previous stores to be persisted. Fortunately, we can further loosen the requirements for strict persistency. If it is impossible for the program to observe that stores were persisted in a different order, we can permit those stores to be reordered, and partly recover from the performance penalties of strict persistency. Robustness leverages this additional degree of freedom and ensures that any execution of a program under a weak persistency model is equivalent to some execution of the program under the strict persistency model [20]. This suffices to ensure the correct usage of flush and fence operations as additional flush and fence operations will not alter the set of possible post-crash program executions. As missing flushes and fences are a major source of bugs in persistent memory program, a tool that eliminates these concerns greatly simplifies persistent memory programming.

Robustness to persistency models has been explored before in the context of bug finding. PSan uses a dynamic analysis combined with random execution or model checking to check robustness. PSan suffers from the same limitations as all dynamic analysis—it requires test cases and may miss bugs that are not revealed by the test cases. Thus, the analysis used by PSan cannot be the foundation of a compiler that automatically inserts flush and fence operations.

PMRobust is applicable to CXL shared memory and all non-volatile memory types. We collectively refer to all of these memory types as persistent memory (PM) throughout the paper for convenience.

This paper makes the following contributions:

- **Static Analysis:** It presents the first static analysis that can efficiently analyze full programs for missing flushes or fences using robustness as a correctness criteria.
- **PM Analysis:** It presents the first static analysis that can determine which operations will only modify volatile or local memory, reducing analysis and annotation overhead.

- **Automated Flush Insertion:** It presents the first static tool that can both repair missing flush and fence bugs as well as insert any missing fences, freeing the developer from this task.
- **Eliminates Missing Flush Bugs:** It presents the first tool that ensures the absence of missing flush bugs.
- **Evaluation:** It evaluates PMRobust on a wide range benchmarks, incurring minimal runtime overhead.

We have made PMRobust's source code, benchmarks, and scripts for reproducing evaluation results available at: https://github.com/uci-plrg/PMRobust-docker

II. Underlying Hardware and Failure Model

Persistency semantics are important for a range of new memory hardware. A well-studied example is Px86 [42], the semantics of persistent memory for x86 processors, which adds a global *persistent buffer* that holds stores after they exited the thread-local store buffers. Stores in the persistent buffer are written back to memory in arbitrary order at cache line granularity, and those not written back are lost upon a crash. To avoid inconsistent state caused by crashes under persistency semantics, fence and flushes instructions are often needed to impose persistency ordering between stores.

While the problem has received much attention for persistent memory, the same problem exists for CXL shared memory. Figure 1 presents a graphical overview of CXL shared disaggregated memory. In this setting, processing nodes and memory nodes are connected via CXL networking. Memory can be coherently shared between multiple hosts — memory nodes contain directories that track which cache lines have been requested by processing nodes. CXL on x86 is intended to provide TSO ordering guarantees across machines.

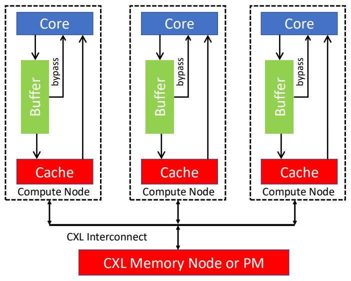


Fig. 1: CXL Memory System

Failures of compute nodes pose a consistency problem. Compute nodes cache their writes to CXL shared memory and the contents of these caches can be written back in an arbitrary order. The CXL standard provides for global persistent flush (GPF). When GPF is supported, it uses energy reserves to write the contents of the CPU cache back to the underlying storage upon a crash or system shutdown. However, GPF does not cover a wide range of failure modes, *e.g.*, failed cables, failed CXL transceivers, failed CPUs, failed motherboards, etc, and therefore does not provide a complete solution to the consistency problem.

If even a single machine that is updating a given CXL memory region fails, the data in that machine's cache is corrupted for all machines that access the CXL memory region. In our email discussions with Intel engineers, it is not yet decided how CXL will expose such failures to the software layer. One option is to report those cache lines as poisoned and throw exceptions on access and another is to allow software to access the copy of the data that resides in the CXL memory node and let software manage flushing data with flush and fence operations in a manner similar to PM systems.

PM can be viewed as a special case of the CXL shared memory model with the differences that: (1) there is only one processing node and (2) after a crash, the node can return having lost the state of its local cache.

The x86 architecture provides the following instructions to force the cache to write data back to persistent storage: (1) the `clflush` and `clflushopt` instructions that flush (and evict) a cache line and (2) the `clwb` instruction that writes back a cache line potentially without eviction. Each of these instructions takes as input the address to flush. The `clflush` instruction flushes a cache line immediately while the `clflushopt` and `clwb` instructions are not guaranteed to flush or write back a cache line until the thread executes a fence instruction.

III. PMRobust

This section discusses the key ideas behind our approach to inserting flush and fence operations.

A. Robustness

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

(a) Pre-crash execution

```
r1 = x;
r2 = y;
```

(b) Post-crash execution

Fig. 2: Assume `x = y = 0` initially. If the post-crash execution observes `r2 = 1`, strict persistency requires that `r1 = 1`.

Figure 2 presents an example that illustrates the requirements of strict persistency. Strict persistency requires that the persistency order for stores respects the happens-before relation. This means that the store `x = 1` must be persisted before the store `y = 1`, forbidding an execution in which `r1 = 0` and `r2 = 1`.

It may initially appear that robustness would require placing flushes after every memory access to PM. However, robustness only requires that the persistency order for stores respects the happens-before relation when an execution can potentially observe a violation of strict persistency. For example if post-crash executions only read from `y`, then the program is robust even if `y = 1` is made persistent while `x = 1` is not. This observation is most relevant for newly created persistent objects that have not yet become reachable from persistent data structures. Thus, it suffices to delay flushing stores to newly created persistent objects until right before they are inserted into persistent data structures.

We next present a sufficient set of requirements on flush and fence operations to ensure robustness. Figure 3 presents a finite state machine that captures how to check robustness for the x86-TSO persistency model. We refer to a state in this finite state machine as an *escape persistency state*. The finite state machine captures the set of legal transitions for cache lines through escape persistency states. If there is an object that is both escaped and non-clean and the program writes to an escaped object, then this is a *robustness violation*. A key insight is differentiating between (1) memory locations that are *captured* by the local thread and thus stores to the memory location would not be visible if the program crashed and (2) memory locations that have *escaped* to become reachable from the roots of persistent data structures, and thus stores would be visible if the program crashed. With this distinction, captured objects do need to flushed according to the volatile order, because they cannot be read from in a post-crash execution.

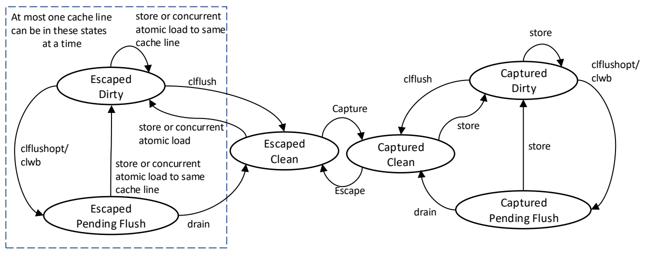

At most one cache line can be in these states at a time
store or concurrent atomic load to same cache line
Escaped Dirty
clflush
store or concurrent atomic load
clflushopt/ clwb
store or concurrent atomic load to same cache line
Escaped Pending Flush
drain
Escaped Clean
Capture
Escape
Captured Clean
clflush
store
Captured Dirty
store
clflushopt/ clwb
store
drain
Captured Pending Flush

Fig. 3: Using Flush & Drain Operations to Ensure Robustness

Captured Objects: Memory locations are captured when there exists no path from the persistent data structure roots to the memory location. Stores to captured memory locations are not visible after a crash, and thus it is safe to delay flushes until immediately before the memory location escapes via insertion into a persistent data structure. This can have several benefits—first, it becomes possible to use optimized flushes like `clflushopt` or `clwb` on several cache lines and amortize the cost of the fence operation across multiple flushes. Second, it is sufficient to handle multiple stores to the same cache line with a single flush.

Escaped Objects: Memory locations have escaped if they are reachable from a persistent data structure. Escaped memory locations require a more expensive flush insertion approach. If consecutive stores happen to be to the same cache line, their persistency order is enforced automatically due to cache coherence. If they are to different cache lines, it is necessary to flush the first store before performing the second store.

Atomic loads require extra care to ensure robustness. Consider the example in Figure 4. If the pre-crash execution crashes before the `clflush` in Thread 1 completes, but after the store to `y` in Thread 2 has been made persistent, it is possible for the post-crash execution to observe `r2 = 0` and `r3 = 1`, violating strict consistency. Robustness in this case requires a `clflush` to the cache line of `x` after Thread 2 reads from `x`. This example also has an implication for all stores to escaped

Thread 1:

```
x = 1;
clflush(&x);
```

Thread 2:

```
r1 = x;
y = r1;
```

(a) Pre-crash execution

```
r2 = x;
r3 = y;
```

(b) Post-crash execution

Fig. 4: A non-robust program that is missing a flush on load. Assume that `x = y = 0` initially and all accesses are atomic, it is possible that `r2 = 0` and `r3 = 1`

memory locations inside of critical sections—the store must be persisted before the lock is released.

B. Analyzing Robustness

```
1  struct Node {
2    int data;
3    struct Node * next;
4  };
5
6  struct Stack {
7    struct Node * top;
8  };
9
10 void push(struct Stack *s, int val) { //s: <esc, clean>
11   struct Node * head = s->top;
12   struct Node * n = pmalloc(sizeof(struct Node));
13   //n: <cap, <clean, clean>>
14   n->data = val;
15   n->next = head;
16   //n: <cap, <dirty, dirty>>
17   s->top = n;
18   //s: <esc, dirty>, n: <esc,<dirty, dirty>>
19 }
```

Fig. 5: A Persistent Stack. esc = escaped, cap = captured

We begin with an example to illustrate key concepts in our approach to analyzing PM code. Figure 5 presents a single-threaded persistent stack. The `push` method adds a new value to the top of the stack. It calls `pmalloc` to allocate a new stack node, stores the value `val` to the node, and updates the node's `next` field. Then it flushes the new node and updates the top of the stack to reference the new node. Finally, it flushes the update to the top of the stack.

In this example, the stack `s` and node `n` are persistent variables and have one of the states in Figure 3. We assume `s` and `n` are on different cache lines. The stack `s` is the root of the persistent data structure and is escaped initially. When the node `n` is created, both of its fields have the state $\langle captured, clean\rangle$ initially. Node `n` has the state $\langle captured, dirty\rangle$ for both fields after the stores at lines 14 and 15. The commit store at line 17 makes `n` *escaped* as `n` is reachable from the persistent data structure root, and the state of `s->top` transitions to $\langle escaped, dirty\rangle$. This is a violation of robustness as both `n` and `s` are *escaped* and *dirty*. Thus, we must insert a flush before line 17.

1) Basic Approach: We next discuss our approach to analyzing PM programs. Our approach builds on the finite state formulation for ensuring robustness from Section III-A.

Our analysis is structured as a standard fixed-point dataflow analysis. The basic idea is to use a static analysis to compute at each program point a mapping from memory locations to a set of potential escape persistency states. The transfer function implements the finite state machine from Figure 3. We apply this machine to each of the potential escape persistency states for a given memory location to generate a new output set of escape persistency states.

The analysis checks several correctness properties. The first is that the program does not take a forbidden transition that would violate robustness such as having multiple escaped and non-clean cache lines. We compute summaries of the effect of method calls by extending escape persistency states to tracks the corresponding initial persistency states when the method was first called. When the analysis of the method is complete, the static analysis has determined how the method changes each of the possible escape persistency states.

Loads pose a challenge because they allow another thread to observe a store before it is made persistent, and that thread may later store a value that was derived from the value returned by the load. The later store can potentially be persisted before the initial store, and thus a crash can leave the PM in an inconsistent state. For example, in Figure 4, it is possible for `y = r1` to be persisted before `x = 1`, and then the post-crash execution would read `r2 = 0` and `r3 = 1`.

This problem can be solved by inserting a flush immediately after every load, but this incurs an overhead. We consider two cases for loads:

1) **Non-atomic Loads:** Here we assume programs are data race free, as they otherwise have undefined semantics in languages such as C/C++. In the case of a non-atomic load, we require that non-atomic stores be persisted before any release operation such as an unlock operation. Then there is no issue because store is persisted before its mutex is released and thus before it can be read.
2) **Atomic Loads:** Atomic operations allow multiple threads to access memory without acquiring a lock. As a result, we cannot assume the flush instruction after the corresponding store has completed before the atomic load. For atomic loads, we use FliT [46] to ensure the corresponding store is flushed. In the analysis, we address this issue by having atomic loads change the persistency state to dirty. This forces the thread to flush the data before performing other visible stores.

C. Detecting References to Persistent Memory

PMROBUST uses a CFL-reachability-based alias analysis introduced by Zheng and Rugina [47] to distinguish PM locations from non-persistent locations. First, the analysis requires a set of user-configured PM allocators, and the pointers returned by these allocators are identified as the initial set of PM pointers. The aliases of known PM pointers are iteratively computed and added to the set until a fixed point is reached. The CFL-reachability-based formulation of the aliasing relation is precise and enables a demand-driven algorithm, which finds all pointers that may reach persistent memory. This helps PMROBUST avoid analyzing a large number of volatile memory pointers.

Our analysis is adapted from an existing implementation from the LLVM-8 codebase [15] that computes aliases between all pairs of pointers. We modified the analysis to only explore aliases of PM pointers following the original demand-driven formulation [47]. We also added type-based field-sensitivity to the analysis to track whether each offset of a struct type is a PM pointer, and treat all objects of the same type uniformly. This approach intentionally sacrifices some precision compared to full field sensitivity because PM programs likely use specialized data types for PM as we observe in the PM benchmarks.

IV. INTRAPROCEDURAL ANALYSIS

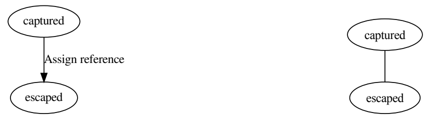
In this section, we first discuss the core intraprocedural analysis. Later, in Section VI we will extend this analysis to the interprocedural context and to handle arrays. The intraprocedural part is implemented as a standard forward dataflow analysis. The algorithm maintains a program state at each instruction.

A. Preliminaries

The instructions that we analyze are atomic and non-atomic loads and stores, atomic RMWs, assignments, and flushes and fences. An object can occupy multiple cache lines and thus an object reference $r \in R$ can be used depending on the field to access one of several different cache lines. Objects are by default not aligned to cache lines, and thus the static analysis may not know whether two different fields reside on the same cache line. Thus, we model a memory location $m \in M$ as the combination $l = \langle r, n \rangle \in \mathcal{L}$ of a reference (variable that references a memory location) $r \in R$ and a non-negative offset $n \in \mathbb{Z}^{0+}$ from that reference.

We next describe our core analysis approach. For each PM location, our analysis must compute: (1) whether a reference to that memory location may have escaped to PM, and (2) whether all stores to the memory location have been flushed to PM. Although the original finite state machine in Figure 3 combines both properties into a single finite state machine, we separated the two properties into two finite state machines to simplify the presentation.

Figure 6a presents a finite state machine that captures whether a memory location has escaped at a given program point, and Figure 6b presents a lattice for our escape analysis. We use an escape state $e \in \mathcal{E}$ to represent one of the two escape values, $\{\textit{captured}, \textit{escaped}\}$, from the escape analysis lattice. We use a may-escape analysis to conservatively catch all cases where a reference might have escaped. Thus, we have the *escaped* value lower in the lattice, and a merge of an *escaped* value with a *captured* value yields the *escaped* state. The analysis computes a map $\mathcal{G}_{\mathcal{E}} \subseteq R \times \mathcal{E}$ from memory locations to escape states at each point in the program. The meet operator $\sqcap : \mathcal{E} \times \mathcal{E} \to \mathcal{E}$ is defined by $e_1 \sqcap e_2 = \text{lower}(e_1, e_2)$, which returns the lower of the two lattice values. We write $e_1 \succeq e_2$ if e_1 is higher than or equal to e_2 in the lattice.

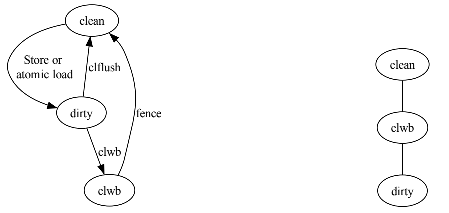


(a) Finite State Machine (b) Lattice

Fig. 6: Lattice and FSM for Escape Analysis

Figure 7a presents a finite state machine that summarizes the semantics for persistency state of a memory location, and Figure 7b presents the corresponding lattice. We use a persistency state $p \in \mathcal{P}$ to represent one of the three persistency state values, $\{clean, clwb, dirty\}$. The lattice is ordered in this fashion, because we need to know whether a memory location may require a fence (*clwb*) or whether it may require a fence and flush (*dirty*). For example, if a reference is *clean* on one path to a node and *clwb* on a different path to the node, the analysis must conservative assume it is *clwb* at the merge point.

The core analysis computes a map $\mathcal{G_P} \subseteq \mathcal{L} \times \mathcal{P}$ from memory locations to persistency states. The meet operator $\sqcap : \mathcal{P} \times \mathcal{P} \to \mathcal{P}$ and the ordering operator $\succeq$ for $\mathcal{P}$ are defined similar to the ones for $\mathcal{E}$.



(a) Finite State Machine (b) Lattice

Fig. 7: Lattice and FSM for Persistency State Analysis

B. Checking Whether Objects are Captured

We take a simple approach to escape analysis — once a reference to a newly allocated struct or array is stored to any place other than a variable, we assume it has escaped. The key ideas of the analysis are that a newly allocated object starts in the captured state. For example, the statement `x=new` would result in the analysis computing that `x` is in the captured state at the program point immediately after this statement. This is stored in the map $\mathcal{G_E}$. The analysis then computes the sets of variables that may reference the same object. After the statement `x=new`, the analysis would compute that `x` is the only variable to reference the memory it references. The analysis stores this information in the alias map $\mathcal{G_A} \subseteq R \times \mathbb{P}(R)$ from references to their aliases, where $\mathbb{P}(R)$ denotes the power set of R. If the value in a variable `x` is stored to some heap location, the variable `x` and all variables that may reference the same heap location are marked as *escaped*.

Figure 8 present the transfer functions of our escape analysis. We use the form $\mathcal{G}'_\mathcal{E} = (\mathcal{G_E} - \text{KILL}) \cup \text{GEN}$ and use the * symbol to match arbitrary states. We next discuss the transfer functions for key statements:

statement	$\mathcal{G}'_\mathcal{E} = (\mathcal{G_E} - \text{KILL}) \cup \text{GEN}$
`y=x`	$U = \mathcal{G_A}(\text{x}) \cup \{\text{y}\}$ $\mathcal{G}'_\mathcal{A} = \{\langle r, S\rangle \mid \langle r, S\rangle \in \mathcal{G_A} \wedge r \notin U\} \cup$ $\{\langle r, U\rangle \mid r \in U\}$ $\text{GEN} = \{\langle \text{y}, e\rangle \mid \langle \text{x}, e\rangle \in \mathcal{G_E}\}$ $\text{KILL} = \{\langle \text{y}, *\rangle \mid \langle \text{x}, e\rangle \in \mathcal{G_E}\}$

`*y=x`
or
`*y=&x->f`
	$U = \mathcal{G_A}(\text{x})$
	$\text{GEN} = \{\langle \text{y}, escaped\rangle \mid \langle \text{x}, e\rangle \in \mathcal{G_E} \wedge r \in U\}$
	$\text{KILL} = \{\langle \text{y}, *\rangle \mid \langle \text{x}, e\rangle \in \mathcal{G_E} \wedge r \in U\}$

`y=*x`	$U = \mathcal{G_A}(\text{y}) \setminus \{\text{y}\}$ $\mathcal{G}'_\mathcal{A} = \{\langle r, S\rangle \mid$ $\langle r, S\rangle \in \mathcal{G_A} \wedge r \notin \mathcal{G_A}(\text{y})\} \cup$ $\{\langle \text{y}, \{\text{y}\}\rangle\} \cup \{\langle r, U\rangle \mid r \in U\}$ $\text{GEN} = \{\langle \text{y}, escaped\rangle\}$ $\text{KILL} = \{\langle \text{y}, *\rangle\}$

Fig. 8: Transfer Functions for Escape Analysis, where `x` and `y` point to PM locations

statement	$\mathcal{G}'_\mathcal{P} = (\mathcal{G_P} - \text{KILL}) \cup \text{GEN}$
`x->f=v`	$\text{GEN} = \{\langle\langle \text{x}, \text{offset}(\text{f})\rangle, dirty\rangle\}$ $\text{KILL} = \{\langle\langle \text{x}, \text{offset}(\text{f})\rangle, *\rangle\}$
`y=x->f` where `f` is an atomic field	$\text{GEN} = \{\langle\langle \text{x}, \text{offset}(\text{f})\rangle, dirty\rangle\}$ $\text{KILL} = \{\langle\langle \text{x}, \text{offset}(\text{f})\rangle, *\rangle\}$
`flush(&x->f)`	$\text{GEN} = \{\langle\langle \text{x}, \text{offset}(\text{f})\rangle, clean\rangle\}$ $\text{KILL} = \{\langle\langle \text{x}, \text{offset}(\text{f})\rangle, *\rangle\}$
`clwb(&x->f)`	$\text{GEN} = \{\langle l, clwb\rangle \mid \langle l, dirty\rangle \in \mathcal{G_P} \wedge l = \langle \text{x}, \text{offset}(\text{f})\rangle\}$ $\text{KILL} = \{\langle l, *\rangle \mid \langle l, dirty\rangle \in \mathcal{G_P} \wedge l = \langle \text{x}, \text{offset}(\text{f})\rangle\}$
`fence`	$\text{GEN} = \{\langle l, clean\rangle \mid \langle l, clwb\rangle \in \mathcal{G_P}\}$ $\text{KILL} = \{\langle l, *\rangle \mid \langle l, clwb\rangle \in \mathcal{G_P}\}$

Fig. 9: Transfer Functions for Persistency State Analysis

Assignments: When there is an assignment `y=x`, we replace the alias set of `y` (and all other aliases of `x`) with the aliases of `x` plus `y` itself. At the same time, we update $\mathcal{G_E}$ to assign `y` to have the same escape state as `x`.

Stores: When a store `*y = x` or `*y = &x->f` stores to the address `x` or the address of one of its fields `&x->f` to any location, we consider `x` and its aliases as escaped. As an example, line 17 in Figure 5 makes `n` *escaped*.

Loads: When a load `y = *x` reads from `x`, and `y` points to persistent memory, the analysis marks `y` as *escaped* as the loaded value comes from a dereference rather than directly from a variable. Since `y` is overwritten, we set the alias set of `y` to only include itself, and remove `y` from the alias sets of its previous aliases. Note that since `y` is escaped, correct alias information is not necessary.

C. Analyzing Persistency States

We next describe our persistency state analysis. Figure 9 presents the transfer functions for the persistency state analysis, where we assume `x` is a PM location. We express transfer functions using GEN and KILL sets. The reader may note that variables may alias, but this analysis does not track aliasing information. The key observation is that aliasing does not violate soundness—it simply means that the same variable that is used to perform a store must be used to flush the value. The lack of must alias information may result in false positives in cases where one alias is used to perform a store and another is used to flush the store. In Figure 9, we use $\langle \texttt{x}, \text{offset}(\texttt{f})\rangle$ to denote the memory location of `x->f`.

Store/Atomic Store: When an atomic or non-atomic store writes to a field `x->f`, the persistency state of `x->f` (*i.e.*, $\langle \texttt{x}, \text{offset}(\texttt{f})\rangle$) is updated as *dirty*, indicating that the location needs to be flushed.

Atomic Load: As we mentioned in Section III-B1, an atomic load changes the state of the loaded variable or field to *dirty*. The analysis removes the old persistency state of `x->f` and marks it as *dirty*.

Atomic RMW: An atomic RMW is a combination of a fence, an atomic load, and an atomic store, so we apply the transfer functions for a fence, atomic load and store in sequence.

An atomic CAS is an atomic RMW if successful and an fence and atomic load, otherwise. Since our transfer functions have the same effects when storing to a field `x->f` and when performing an atomic load from `x->f`, we consider atomic CASs the same as atomic RMWs.

Flush: When flushing the address of a field `x->f` with the stronger `clflush`, the persistency state of `x->f` becomes clean. Flushing the address of a field `x->f` with `clwb` or `clflushopt` changes its state to `clwb` if it was *dirty*, indicating it will become *clean* at the next fence.

Fence: For fences, the transfer function leaves the persistency states untouched unless they are *clwb*. The locations whose persistency states are *clwb* are changed to the *clean* state.

D. Intraprocedural Violation Detection

PMROBUST consider three categories of violations. The first two categories are 1) unflushed PM locations at function exits; and 2) a store to an escaped PM location when a different PM location is already escaped and non-clean. The third category involving arrays will be discussed in Sections VI-B.

The first category of violations is checked at function exits. When we complete the analysis of a function, we get the program states at each function exit and take a union of the states by using meet operators. If the state of any PM location that is not a function parameter or the return value is escaped and non-clean, we report it as a violation.

The second category of errors is checked at every program point. If a PM location is escaped and non-clean and we perform a store to an escape PM location, this is a violation. To address violations involving multiple threads, when there is an escaped and non-clean location, a release operation to a non-PM location (atomic store release or unlock) is reported as a violation. Recall from Section IV-A, a PM location l is a pair $\langle r, n\rangle$ of a reference and an offset from the reference, so an escaped PM object with two or more fields being non-clean is also reported as a violation.

V. DURABILITY

In addition to robustness violations, the first category of violation from the previous section also includes all durability bugs, i.e. any escaped PM objects that is not made durable by the end of the program. This follows from a simple argument: the only objects that may be escaped and non-clean are the return values and parameters of the top-level main function, but none of these objects can be PM objects.

VI. INTERPROCEDURAL ANALYSIS

In this section, we first discuss extending the core analysis to be interprocedural. Then we discuss how we detect bugs that involve objects reachable from function parameters and bugs that involve multiple functions. Lastly, we describe our array support and how we detect references to PM.

A. Context Sensitivity

We handle function calls in a context sensitive manner using function summaries. Each function has a *function summary table* that maps calling contexts to summarized results.

For a function with n parameters, its *calling context* has the form $C \in (\mathcal{E} \times \mathcal{P})^n$, containing the abstract escape and persistency states for each function parameter. To reduce the possible number of calling contexts, for a function parameter with m fields, we collapse the m persistency states to a single abstract persistency state with the abstraction:

$$\mathit{Abs}(\langle e, p_1, ..., p_m\rangle) = \langle e, \text{lowest}(p_1, ..., p_n)\rangle$$

In other words, we use a single state (the lowest one) for all fields of a given parameter.

Each calling context is then mapped to a *summarized result* in the function summary table. For a function F with n parameters, a summarized result for a calling context C has the form $R_{\langle F,C\rangle} \in (\mathcal{E} \times \mathcal{P})^{n+1}$, containing abstract states for each function parameter and the return value. The last element may be ignored for functions with no return values. The abstraction can be reverted to get back program states by approximating every field with the same persistency state:

$$\mathit{AbsRev}(\langle e, p\rangle) = \langle e, (p, ..., p)\rangle$$

This allows summarized results to be used to update the program state after a function call.

To extend our alias analysis to be interprocedural, we also store the aliasing information between function parameters and the return value to summarized results. The summarized result additionally uses a *markObjDirEsp* bit to record if there are any escaped objects that become non-clean in the callee, potentially causing violations with other escaped non-clean objects in the caller, but are not captured by the summarized result because they become clean again before the callee returns. These extensions are straightforward and are omitted in the representation here.

The interprocedural analysis uses a worklist algorithm operating on pairs $\langle F, C \rangle$ of function F and calling context C. The worklist initially includes all functions with the calling contexts of all parameters having the lowest state $\langle$*captured*, *clean*$\rangle$. When we complete the analysis of a function with a calling context, we update the function summary table, and every time F's summarized results are updated, we push all callers of F with their calling contexts to the worklist.

To speed up the convergence of the algorithm, we allow summarized results for a function F to be approximated from results for higher contexts. For two calling context $C = \langle\langle e_1, p_1\rangle, ..., \langle e_n, p_n\rangle\rangle$ and $C' = \langle\langle e'_1, p'_1\rangle, ..., \langle e'_n, p'_n\rangle\rangle$, we say that C is higher than C' if $e_i \succeq e'_i$ and $p_n \succeq p'_n$ for all i. The meet operator $\sqcap : (\mathcal{E} \times \mathcal{P})^{n+1} \rightarrow (\mathcal{E} \times \mathcal{P})^{n+1}$ for summarized results is also defined as a point-wise meet. There are then three cases when processing a pair $\langle F, C \rangle$:

- **Case 1:** If we have already analyzed F with the calling context C, Then the summarized result $R_{\langle F,C \rangle}$ is used to approximate the state of the parameters and the return value of F after the call site.
- **Case 2:** Otherwise, if we have only analyzed F with calling contexts higher that C, we can take the summarized results $R_{\langle F,C' \rangle}$ for all calling contexts C' higher than C, and use the merged result of this set of summarized results via the meet operator as an approximation, and push the pair $\langle F, C \rangle$ to the worklist. This choice ensures monotonicity when processing call sites and thus preserves the termination guarantee for dataflow analysis.
- **Case 3:** If we have not analyzed F with C or any calling context higher than C before, we approximate all parameters and the return value of F as having the state $\langle$*captured*, *clean*$\rangle$ and push the pair $\langle F, C \rangle$ to the worklist.

B. Handling Arrays

To handle arrays, we abstract array writes as a pair of an array reference and an index. Formally, we model an array element $l = \langle r, n \rangle \in \mathcal{L}_a$ as a reference (variable) $r \in R$ and a index $n \in \mathbb{Z}^{0+}$ from that reference. We abstract the array index using the variable that provided the value for the array dereference operation. Thus, our abstraction is only able to track dirty array elements as long as the original index variable exists. To ensure soundness, when a function writes to some PM array, we require the written element to be flushed before the function returns or the index variable is changed/lost.

We conservatively assume all array elements have escaped and only compute a map $\mathcal{G}_{\mathcal{P}_a} \in \mathcal{L}_a \times \mathcal{P}$ from array elements to persistency states. The transfer functions for the array persistency analysis can be straightforward obtained from Figure 9 by replacing the addresses being stored to and loaded from with the array index variable. At a function exit, we report warnings if the map $\mathcal{G}_{\mathcal{P}_a}$ contains any element whose persistency state is not clean.

C. Interprocedural Violation Detection

The robustness violation detection mechanism for interprocedural analysis is the same as the intraprocedural one, except that robustness violations that involve multiple functions are also reported. Our treatment of pointer arithmetic conservatively reports violations when there are stores to or loads from PM addresses computed by pointer arithmetic.

D. Relaxing Strict Persistency: An Escape Hatch

Robustness violations are not always bugs; design patterns like *link-and-persist* [11], *pointer tagging* [33], and checksums can cause false positives by allowing observable low-level robustness violations without compromising high-level safety.

For example, the RECIPE benchmarks sometimes use atomic loads to read data from an atomic variable that is never mutated. This is done because the data structure packs both mutable and immutable state into the same atomic variable. Flushing such loads can incur high overheads.

PM programs sometimes write data and a checksum, and then persist both the data and the checksum. When accessing the data, the PM program first verifies the checksum before using the data. While this pattern is safe, it can yield executions that are not equivalent to any execution under strict consistency and thus violate robustness.

PMROBUST provides strict persistency by default. However, if the developer knows that it is safe to escape from strict persistency, PMROBUST provides annotations to ignore blocks that are exempt from strict persistency. When applied to either a store or an atomic load, this annotation tells PMROBUST that no flush or fence operation is needed.

E. Termination and Correctness

All transfer functions in the analysis are monotonic and the lattices are of finite height. Thus, the dataflow analysis terminates. **The correctness proofs are presented in the appendix of the full version of this paper [23].**

VII. TRANSFORMATION

This section details PMROBUST's approach to automatically inserting flush and fence operations to ensure programs are free from missing flush/fence bugs.

A. Flush Insertion

When PMROBUST's analysis detects a violation, PMROBUST fixes it by immediately flushing the PM locations after they become *dirty* using `clwb`, which changes their state to *clwb* in our analysis. Our experience shows that conservatively flushing on atomic loads can incur a significant overhead. To alleviate this problem, we implement the FliT transformation [46] for atomic memory accesses, which uses counters to signal whether all stores to a PM location have been flushed. By reading this counter after an atomic load, we can determine whether the store may not have completed its flush and thus the load must help. This improves the performance of many atomic-load-heavy benchmarks, as can be seen in the evaluation results of Section VIII.

The flush instruction can be chosen from `clflush`, `clwb`, and `clflushopt`. We choose `clwb` as it has weaker ordering properties than `clflush` and may retain the cache line on some architectures as compared to `clflushopt`, potentially leading to better performance.

If the length of dirty bytes in the object is not constant, which could occur when modifying an object using functions like as `memcpy`, a call to a variable-sized flush function is inserted that issues `clwb` instructions over the entire range.

B. Fence Insertion

After flushes are inserted, PMRobust uses the detected robustness violations to perform fence insertions. We insert a `sfence` before the instruction where a violation is reported, eliminating the violation.

Specifically, fences are inserted for the two types of PMRobust robustness violations as follows:

1) A fence is inserted at function exit when there are objects in the *clwb* state.
2) When there are two escaped and non-clean objects at the same time, a fence is inserted right before the second object becomes escaped and non-clean.

Figure 10 provides an example of the second case, showing a block of code with PM-pointers `x` and `y`, which are escaped and clean at the beginning. In the unmodified version, the address of `x` is written first, making it dirty, and the address of `y` is written later, causing a violation. To fix the violation, PMRobust inserts a `clwb` instruction on `x` after the write, making its state *clwb*, and before the address of `y` is written, it inserts a fence, removing the violation.

```
//x: <esc,clean>
*x = 1;
//x: <esc,dirty>
...
//y: <esc,clean>
*y = 1;
//x: <esc,dirty>,
//y: <esc,dirty>
//violation!!
```

(a) Before insertions

```
//x: <esc,clean>
*x = 1;
//x: <esc,dirty>
clwb(x);
//x: <esc,clwb>
...
fence();
//x: <esc,clean>
*y = 1;
//y: <esc,dirty>
//no violation
```

(b) After insertions

Fig. 10: Example Insertion of Flushes/Fences. esc = escaped

Cases that PMRobust does not analyze precisely, such as pointer arithmetic on PM addresses, involve only one object and are handled by conservatively inserting a fence. The correctness of this approach follows from our proof for the error detection mechanism—the inserted fences turn objects that trigger violations to *clean* before the violation, no violation remains after the fence insertion procedure completes.

Atomic loads turn object states to *dirty* in our analysis due to potentially unflushed stores from other threads. However, flushes of previous writes to PM only need to be serialized before the next store to PM and not the next atomic load. In other words, there is no need for fences between multiple atomic loads. Thus, fences for atomic loads are deferred until the next store to PM to avoid inserting redundant fences.

VIII. Evaluation

In this section, we present the evaluation of PMRobust on several benchmarks in order to answer the following research questions:

- RQ1 Applicablility: Does PMRobust analyze PM programs in a reasonable time?
- RQ2 Performance: Do programs transformed by PMRobust have performance similar to the original programs?
- RQ3 Developer Burden: How much developer involvement does PMRobust require to relax the persistency requirements?

We start by describing our system setup, the settings of PMRobust, and the benchmarks. Finally, we discuss how our research questions can be answered by our evaluation findings.
System Setup: While one motivation for this work is CXL shared memory, it is not commercially available yet. Thus, we have evaluated PMRobust on Intel Optane PM, which requires the same use of flush and fence instructions.

PMRobust is implemented as a transformation pass in LLVM. All benchmarks are compiled with clang/clang++ and LLVM with optimization level O3. We used a Ubuntu 22.04.4 machine with a 16 core 2.4 GHz Intel Xeon Silver 4314 processor, 256 GB of RAM, and 256 GB of Optane PM.
PMRobust Settings: We evaluate PMRobust with three optimization settings to understand the contribution of each optimization. *PMRobust*$_{base}$ uses our alias analysis to insert a flush and a fence immediately after every store and atomic load to PM. *PMRobust*$_{opt}$ adds our escape and persistency state analysis to delay insertion of fences, as described in Section VII. *PMRobust*$_{flit}$ adds FliT on top of *PMRobust*$_{opt}$.

A. Benchmarks

Our evaluation includes RECIPE [32], a set of high-performance concurrent index data structures modified to be crash-consistent for persistent memory. RECIPE uses PMDK [27]'s libvmmalloc to convert all dynamic memory allocations to PM allocations. Of the data structures in RECIPE, P-HOT, WOART, and LevelHash fail to compile with LLVM, and we evaluate the remaining six (P-ART, P-BwTree, P-CLHT, P-Masstree, FAST&FAIR, CCEH) using the YCSB benchmark [9]. During evaluation, we found timing-related bugs in CCEH and FAST&FAIR and discard those runs when they occur. These bugs also occur in the unmodified programs. By comparison with the transformed version, we also discovered a number of missing flushes and fences in the functions `getChild()` and `getChildren()` in P-ART's node classes, which allow the client to retrieve values that are not yet persisted in the index. If the client then writes to PM locations based on the retrieved value and a crash happens, the later update could be persisted while its source in the index might not, *i.e.*, a crash consistency bug. We added these flushes and fences to P-ART.

We also include Memcached [10], a popular distributed memory object caching system. Specifically, we choose a version of Memcached that supports persistent memory via PMDK's libpmem. We use the memaslap load testing tool from libMemcached [5] for benchmarking.

Lastly, we evaluate our tool on PMDK [27], the most widely-used open-source libraries for programming persistent memory, using a data store implementation which is provided as one of its example programs. The data store has a swappable backend that allows choosing from among seven map data structures: `btree`, `rbtree`, `ctree`, `hashmap_atomic`, `hashmap_tx`, `hashmap_rp`, and `skiplist`.

For each benchmark, we show the average throughput of the original and the transformed program over 5 runs for RECIPE and over 10 runs for memcached and PMDK's data store, and display the standard deviation as error bars. In the transformed programs, we first removed the existing flushes and fences from the original programs and then our tool transforms them by inserting flushes and fences.

B. RQ1: Applicability

To answer RQ1, we measure the execution time of PMROBUST pass and under the *PMRobust$_{opt}$* setting for each of the benchmarks. We then compare them against the execution time of the PM bug repair tool PMBugAssist [26], which uses SMT solving to generate targeted fixes for PM bugs contained in the execution trace, but is not able to guarantee the repaired program is free of further PM bugs. Although PMBugAssist belongs to a separate paradigm, there is currently no other PM bug repair tool that takes a purely static approach like PMROBUST does. We use the comparison to demonstrate that PMROBUST is able to run in reasonable time on the same programs as PMBugAssist, while providing stronger correctness guarantees. Thus, in addition to previously mentioned benchmarks, we include measurements on PMDK test programs evaluated by PMBugAssist. The times are presented in Table I along with the code size of the benchmarks. The time reported for PMBugAssist is the total time it takes on our machine to fix all bugs provided for each benchmark in the original evaluation.

Table I shows that PMROBUST's execution time roughly correlates with the code size, whereas the execution time of PMBugAssist varies widely, ranging from less than a second to close to an hour, depending on the length of execution traces, the number of bugs, and the SMT solving process. These measurements support the claim that PMROBUST is applicable to programs targeted by prior tools, but with more consistent analysis times and stronger guarantees.

C. RQ2: Performance

Here we explore RQ2 by comparing the performance of PMROBUST's transformed programs with the originals.

RECIPE: We follow RECIPE's evaluation procedure [32] by testing the ordered indexes (P-ART, P-BwTree, P-Masstree, FAST&FAIR) on two types of keys—**randint** (8 byte random integer keys) and **string** (24 byte YCSB string keys), all uniformly distributed, with YCSB workloads A, B, C, and E, and the unordered index P-CLHT and CCEH on workloads A, B, and C with uniformly distributed randint keys. In both cases, we set the thread count to 16 and first populate the index with 64M keys using LoadA, and then run the respective workloads that insert and/or read the keys.

Benchmark	PMR Time(s)	PBA Time(s)	Code Size (KLOC)
data_store	20.8	N/A	95.2*
Memcached	199.1	1861.49	23
RECIPE	64	5.94	40.3
obj_constructor	17.9	0.1	95.4*
obj_first_next	17.7	0.3	95.5*
obj_mem	18.1	149.5	95.3*
obj_memops	28.3	0.3	95.8*
obj_toid	18.1	0.1	95.3*
rpmemd_db	12.2	0.1	25.3*
pmemspoil	14.8	0.1	47.8*
pmem_memcpy	11.9	0.3	46.7*
pmem_memmove	11.9	0.2	46.7*
pmem_memset	11.9	198.9	46.6*
pmreorder_simple	10.3	4810.0	46.6*
pmreorder_flushes	10.4	5911.8	46.6*
pmreorder_stack	10.4	1.2	46.6*

* including sublibraries of PMDK

TABLE I: Runtime of PMROBUST vs. PMBugAssist. PMR stands for PMROBUST, PBA stands for PMBugAssist. N/A means not included in PBA's original evaluation

We discovered a bottleneck when running YCSB workload E with the transformed P-BwTree that heavily uses an iterator data structure to traverse retrieved values. An iterator and the data it contains is never reused after a crash, yet its data is allocated in PM. This unnecessary usage of PM occurs because the RECIPE benchmarks, as research prototypes, do not use separate memory allocators for PM and DRAM allocations, but simply use libvmmalloc to perform all dynamic allocations using PM. In a proper implementation, iterator data should be allocated in DRAM. Thus, we eliminated the flush insertion on iterator contexts with 5 annotation pairs, where each annotation pair ignores exactly only one program statement. Figure 11 shows the resulting throughputs.

Memcached: We evaluate Memcached's throughput with 16 threads and 100K operations using memaslap. The workload uses 16-byte keys and 1024-bytes values. The proportion of get and set operations is 90/10. Figure 12 reports the throughputs.

PMDK: During the data store benchmark, 100K randomly generated 8-byte integer keys are inserted into the data store and then removed. Figure 13 presents the results of the seven map data structures.

First, we see from the evaluation results that the base transformation achieves performance on-par with the original programs on a number of benchmarks, including P-CLHT, P-Masstree, Fast&Fair, CCEH, and Memcached. This suggests that in these benchmarks, extra flushes and fences inserted by the base approach do not appear on performance-critical paths. For the other benchmarks that have a noticeable gap between the performance of *PMRobust$_{opt}$* and the original program, *PMRobust$_{opt}$* is able to either reduce, or in some cases completely eliminate the performance gaps. The FliT

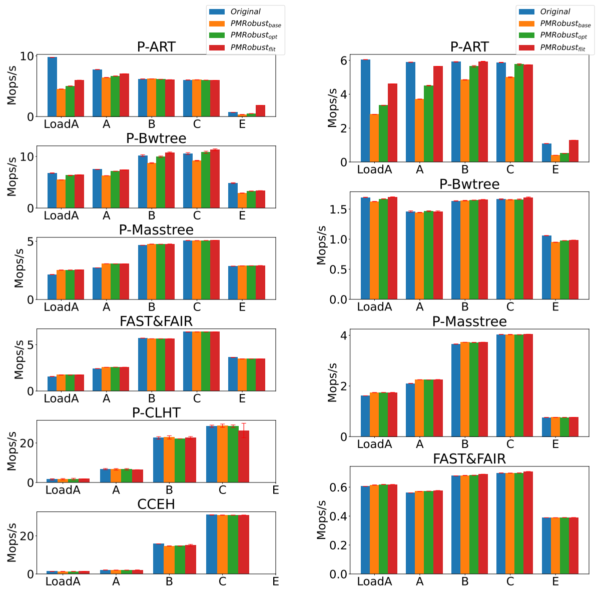

(a) Integer keys (b) String keys

Fig. 11: Throughput of RECIPE indexes. Larger is better.

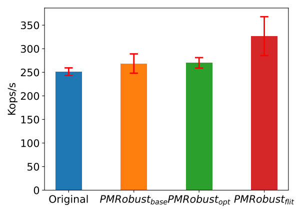


Fig. 12: Throughput of Memcached. Larger is better.

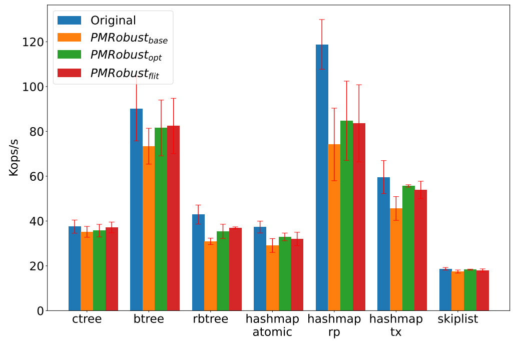


Fig. 13: Throughput of PMDK's data store. Larger is better.

transformation of *PMRobust$_{flit}$* further reduces the gap on many workloads for P-ART and P-BwTree due to their frequent atomic loads. The improvement is most notable for P-ART, which performs many atomic loads during its tree traversal on each key insertion. *PMRobust$_{flit}$* outperforms the original program significantly on workload E as it eliminates most of the overhead from the originally missing flushes mentioned in Section VIII-C. It also outperforms the original Memcached by approximately 20%, which can be explained by the fact that PMRobust inlines flush instructions whereas the original benchmark called a flush function.

Across all benchmarks, the geometric mean overhead over the original programs is 11.21% for *PMRobust$_{base}$*, and 6.41% for *PMRobust$_{opt}$*, and only 0.26% for *PMRobust$_{flit}$*.

Overall, the performance results show that PMRobust's automatic flush and fences insertion is able to match the performance of the original programs on most benchmarks using our dataflow analysis and FliT, while only using a few user annotations. This answers the research question in the affirmative—PMRobust is able to produce programs with performance close to the originals,

D. RQ3: Developer Burden

In this section we answer RQ3 regarding the extent users of PMRobust need to be involved to relax the persistency requirement using manual annotations. As we noted in VIII-C, there was only one instance of significant bottleneck that we removed with an annotation during the performance evaluation. We identified the source of the bottleneck by profiling.

Recall that we noted that this one example was unusual, and due to the fact that RECIPE does not implement a reasonable memory allocation strategy. We would not expect to need annotations for this same reason in non-research software. Based on our experience, PMRobust does not impose too much burden on developers as bottlenecks should be rare, and fixing them only requires familiarity with profilers, which would be reasonable to expect from developers working on low-level software such as PM programs.

E. Threats to Validity

PMRobust currently does not implement support for function pointers and may produce imprecise results for them. This can be addressed by ensuring that all objects are clean before making a call using a function pointer and ensuring that if a function has its address taken, that all objects must be clean at exit. This could also potentially be handled by pointer analysis.

Our current implementation uses whole-program analysis. As a result, we perform our analysis and transformation passes after all translation units are linked together. This requirement can be removed by treating interactions with code outside of the current compilation unit conservatively.

IX. Related Work

There is work on checking/testing PM programs to find bugs. In particular, XFDetector [36] uses a finite state machine to track the consistency and persistency of persistent data. PMTest [37] lets developers annotate a program with checking rules to infer the persistency status of writes and ordering constraints between writes. Pmemcheck [29] checks how many stores were not made persistent and detects memory overwrites using binary rewriting. Yat [30] model checks PM programs. Agamotto [39] finds bugs in PM programs by using symbolic execution. An algorithm by Huang et al. [25] infers invariants from PM programs that are then used to check for bugs. Although these tools are able to find many bugs, none of these tools can assure the absence of flush/fence bugs like PMROBUST can. POG [41] and Pierogi [3] provide logics that can be used to manually reason about program behaviors.

A line of work [6], [17], [18], [34] uses (software or hardware) transactions to provide (failure and thread) atomicity. NVL-C provides language and compiler support for the use of transactions to access non-volatile memory [12]. While NVL-C can provide crash consistency, it does so by incurring the overheads of using transactions to provide crash consistency. Another line of work [4], [7], [24], [28], [35] advocates use of locks or synchronization-free regions [19]. Memento [8] provides detectable checkpointing—it extends standard checkpoint with support to allow the system to be able to detect the status of in flight operations when the crash occurred. These approaches typically incur large overheads to support the necessary logging.

StaticPersist [2] is a static analysis to determine which objects must be allocated in PM. The idea is to use annotations to declare a set of durable roots, and the analysis determines which objects are reachable from the roots. AutoPersist [45] is a Java extension to support NVM. Developers specify durable roots and when an object becomes reachable from a durable root, AutoPersist moves it to PM. Both StaticPersist and AutoPersist provide a higher-level API for programming PM, while PMROBUST targets a lower-level model by automatically inserting flush and fence operations.

Hippocrates [38] and PMBugAssist [26] insert flushes and fences to repair PM bugs. Hippocrates chooses the placement of flushes and fences using a reduction procedure, whereas PMBugAssist uses a SMT solver. They both focus on repairing specific PM bugs given as program input and rely on other PM bug detection tools to produce bug traces, which is different from our task of exhaustively detecting potential PM bugs and fixing them at the same time.

Our work is related to a line of work on static fence insertion for concurrent programs to ensure sequential consistency [1], [31], [14]. The static approach to fence insertion has so far not been applied to PM programs, a gap which we bridge. In terms of techniques, previous work such as musketeer [1] and pensieve [14] focused on variants of delay set analysis [44] combined with flow-insensitive escape analysis to minimise fence insertions, whereas we make use of a flow-sensitive escape analysis to avoid redundant fence insertions.

FliT [46] presents a technique that uses counters to eliminate flush operations on atomic loads, which we applied in PMROBUST to further reduce the overhead it introduces.

X. Conclusion

Correctly using flush and fence operations is notoriously hard. PMROBUST employs compiler analysis and transformation to automatically insert flush and fence instructions, providing strong persistency via robustness, a sufficient condition for the correct usage of flush and fence operations in PM programs. PMROBUST ensures the absence of flush and fence bugs and eliminates the time-consuming and error-prone task of manually inserting these operations.

Acknowledgements

We would like to thank the anonymous reviewers for their thorough and insightful comments that helped us improve the paper. This work is supported by National Science Foundation grants CCF-2006948, CCF-2102940, and CCF-2220410.

References

[1] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence: A static analysis approach to automatic fence insertion. *ACM Trans. Program. Lang. Syst.*, 39(2), May 2017.

[2] Sorav Bansal. Staticpersist: Compiler support for pmem programming. In Cezara Dragoi, Michael Emmi, and Jingbo Wang, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 44–65, Cham, 2023. Springer Nature Switzerland.

[3] Eleni Vafeiadi Bila, Brijesh Dongol, Ori Lahav, Azalea Raad, and John Wickerson. View-based Owicki–Gries reasoning for persistent x86-TSO. In *Programming Languages and Systems: 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings*, page 234–261, Berlin, Heidelberg, 2022. Springer-Verlag.

[4] Hans-J. Boehm and Dhruva R. Chakrabarti. Persistence programming models for non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, pages 55–67, New York, NY, USA, 2016. Association for Computing Machinery.

[5] Brian Aker. libmemcached. https://libmemcached.org/libMemcached.html, 2011.

[6] Daniel Castro, Paolo Romano, and João Barreto. Hardware transactional memory meets memory persistency. In *2018 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '18, pages 368–377, Vancouver, BC, Canada, 2018. Institute of Electrical and Electronics Engineers.

[7] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. Association for Computing Machinery.

[8] Kyeongmin Cho, Seungmin Jeon, Azalea Raad, and Jeehoon Kang. Memento: A framework for detectable recoverability in persistent memory. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[9] 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*, pages 143–154, 2010.

[10] Inc. Danga Interactive. Memcached. https://github.com/lenovo/memcached-pmem, November 2018.

[11] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 373–385, USA, 2018. USENIX Association.

[12] Joel E. Denny, Seyong Lee, and Jeffrey S. Vetter. NVL-C: Static analysis techniques for efficient, correct programming of non-volatile main memory systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, page 125–136, New York, NY, USA, 2016. Association for Computing Machinery.

[13] 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 2021, pages 503–516, New York, NY, USA, 2021. Association for Computing Machinery.

[14] Xing Fang, Jaejin Lee, and Samuel P Midkiff. Automatic fence insertion for shared memory multiprocessing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 285–294, 2003.

[15] LLVM Foundation. Llvm-8. https://github.com/llvm/llvm-project/tree/release/8.x, August 2019.

[16] 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 *Proceedings of the 28th ACM Symposium on Operating Systems Principles*, SOSP 2021, pages 100–115, New York, NY, USA, 2021. Association for Computing Machinery.

[17] Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. Crafty: Efficient, HTM-compatible persistent transactions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 59–74, New York, NY, USA, 2020. Association for Computing Machinery.

[18] Ellis Giles, Kshitij Doshi, and Peter Varman. Continuous checkpointing of HTM transactions in NVM. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2017, pages 70–81, New York, NY, USA, 2017. Association for Computing Machinery.

[19] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for synchronization-free regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 46–61, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Hamed Gorjiara, Weiyu Luo, Alex Lee, Guoqing Harry Xu, and Brian Demsky. Checking robustness to weak persistency models. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 490–505, New York, NY, USA, 2022. Association for Computing Machinery.

[21] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Jaaru: Efficiently model checking persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, pages 415–428, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Yashme: Detecting persistency races. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 830–845, New York, NY, USA, 2022. Association for Computing Machinery.

[23] Yutong Guo, Weiyu Luo, and Brian Demsky. Automated insertion of flushes and fences for persistency. https://arxiv.org/abs/2509.19459, 2025.

[24] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 468–482, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Zunchen Huang, Srivatsan Ravi, and Chao Wang. Discovering likely program invariants for persistent memory. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 1795–1807, New York, NY, USA, 2024. Association for Computing Machinery.

[26] Zunchen Huang and Chao Wang. Constraint based program repair for persistent memory bugs. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[27] Intel Corporation. Persistent memory development kit. https://pmem.io/pmdk/, 2020.

[28] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 427–442, New York, NY, USA, 2016. Association for Computing Machinery.

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

[30] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 433–438, Philadelphia, PA, June 2014. USENIX Association.

[31] Jaejin Lee and D.A. Padua. Hiding relaxed memory consistency with compilers. In *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00622)*, pages 111–122, 2000.

[32] 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, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Nan Li and Wojciech Golab. Brief announcement: Detectable sequential specifications for recoverable shared objects. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 557–560, New York, NY, USA, 2021. Association for Computing Machinery.

[34] 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 *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 329–343, New York, NY, USA, 2017. Association for Computing Machinery.

[35] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pages 258–270, Virtual Event , Greece, 2018. Institute of Electrical and Electronics Engineers.

[36] 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 Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 1187–1202, New York, NY, USA, 2020. Association for Computing Machinery.

[37] 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 Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 411–425, New York, NY, USA, 2019. Association for Computing Machinery.

[38] Ian Neal, Andrew Quinn, and Baris Kasikci. Hippocrates: Healing persistent memory bugs without doing any harm. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, pages 401–414, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Ian Neal, Ben Reeves, Ben Stoler, and Andrew Quinn. AGAMOTTO: How persistent is your persistent memory application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1047–1064, Banff, Alberta, November 2020. USENIX Association.

[40] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 265–276, Minneapolis, MN, USA, 2014. Institute of Electrical and Electronics Engineers.

[41] Azalea Raad, Ori Lahav, and Viktor Vafeiadis. Persistent Owicki-Gries reasoning: A program logic for reasoning about persistent programs on Intel-x86. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[42] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the Intel-x86 architecture. *Proceedings of the ACM on Programming Languages*, 4(POPL), December 2020.

[43] Memory-semantic SSD. https://samsungmsl.com/ms-ssd/, 2023.

[44] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, April 1988.

[45] Thomas Shull, Jian Huang, and Josep Torrellas. Autopersist: an easy-to-use java nvm framework based on reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 316–332, New York, NY, USA, 2019. Association for Computing Machinery.

[46] Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch, and Erez Petrank. FliT: A library for simple and efficient persistent algorithms. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 309–321, New York, NY, USA, 2022. Association for Computing Machinery.

[47] Xin Zheng and Radu Rugina. Demand-driven alias analysis for C. page 197–208, 2008.