# Symbolic Analysis for Repairing Bugs in Concurrent Persistent-Memory Programs

Tooba Khan, Srivatsan Ravi, Chao Wang
*University of Southern California*
Los Angeles, USA
{tooba.khan, sravi, wang626}@usc.edu

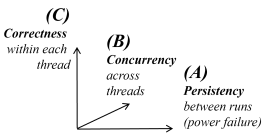
***Abstract*—The availability of byte-addressable persistent memory (PM) has sparked increased efforts toward designing software based on *persistent* and *concurrent* data structures capable of recovering from system crashes due to power failures. However, it can be challenging for programmers to ensure data persistence and guarantee the correct order of commits, thus leading to persistent-memory bugs. We present the first symbolic analysis method based on the use of an SMT solver to repair such bugs. Unlike existing methods, which focus exclusively on repairing bugs in a sequential computation, our method is capable of repairing *inter-thread inconsistency* PM bugs that are unique to concurrent programs. To this end, we propose a *two-step approach to symbolic analysis*, which separates the repair of bugs within each thread from the repair of bugs across threads, to drastically reduce the cost of invoking the SMT solver. Our experimental evaluation demonstrates that our method effectively repairs all of the three most common PM bugs, namely durability, crash-consistency, and inter-thread inconsistency PM bugs. This makes it the first solution to repairing all these bugs in concurrent persistent-memory programs. We demonstrate the efficacy of our method by repairing PM bugs across five real-world applications, including PM-enabled distributed storage applications such as Memcached and P-CLHT.**

***Index Terms*—Program repair, verification, program synthesis, persistent memory, concurrency, SMT solver**

## I. Introduction

Persistent memory (PM) combines the speed of DRAM with the durability of storage, but correctness depends on explicitly flushing and fencing cache lines to ensure consistency. Even small mistakes in these persist-ordering operations can corrupt data after crashes, making PM programming error-prone. Persistent-memory bugs, or PM bugs, may have severe consequences. For example, if a complex data structure like a tree or a graph is partially updated but interrupted (by a power failure) before completion, the data stored in PM may be left in an inconsistent state. To repair such bugs, we must guarantee that all PM updates are persisted in the correct order, which requires careful use of cache-line flushes and memory barriers, which can be intricate and error-prone to manage.

While there are many existing methods for detecting PM bugs [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], methods for automatically repairing PM bugs are still lacking. To the best of our knowledge, there are only two existing repair methods. The first method is Hippocrates [15], which relies on matching known bug patterns and then applying predefined code transformations, but is limited to simple durability violations. The second method is PMBugAssist [16], which uses the SMT solving to repair both durability and crash-consistency violations. However, both of them are restricted to PM bugs in a sequential computation. Since they do not model *multi-threading*, they cannot repair *inter-thread inconsistency* bugs that are unique to concurrent programs. This is the gap that we aim to fill in this paper.




(a) three dimensions

| Repair Problem | Program Type | Bug Type | Existing Methods |
|---|---|---|---|
| *(C)* | sequential | correctness bug | many |
| *(B + C)* | concurrent | concurrency bug | many |
| *(A + C)* | sequential, PM | PM bug | two |
| *(A + B + C)* | concurrent, PM | inter-thread PM bug | none |

(b) four repair problems

Fig. 1. Our method covers all three dimensions (left) to solve the repair problem that is unique to concurrent PM programs (right).

As illustrated in Fig. 1, persistent-memory (PM) bugs differ fundamentally from both thread-consistency issues in DRAM-based programs and correctness issues in sequential programs. Let $A$ denote PM data *persistency* across crashes, $B$ denote inter-thread *consistency* within a run, and $C$ denote sequential *correctness* of each thread. Existing techniques address $(C)$, $(B{+}C)$, and $(A{+}C)$, but no method repairs bugs in the full $(A{+}B{+}C)$ space.

Repairing inter-thread inconsistency bugs is challenging because threads may concurrently access and update shared PM data, leading to race-like conditions. If one thread reads data while another partially persists an update, an unexpected crash may leave the PM state inconsistent or corrupted. This interplay among persistency, concurrency, and sequential correctness defines a new bug class that existing repair methods cannot handle.

To address this problem, we develop the first method for repairing PM bugs in concurrent programs. The key difficulty lies in reasoning about all possible persist orders and thread interleavings. Within each thread, PM store operations may persist nondeterministically due to hardware reordering; across threads, their interleavings further complicate recovery semantics. Since explicit enumeration of all possible scenarios is practically infeasible, our approach performs SMT-based symbolic analysis that jointly reasons about both crash scenarios and interleavings to automatically generate correct repairs.

Fig. 2 shows our repair method, which takes $(\mathcal{T}, \mathcal{B})$ as input and returns $\mathcal{R}$ as output. Here, $\mathcal{T}$ is the erroneous execution

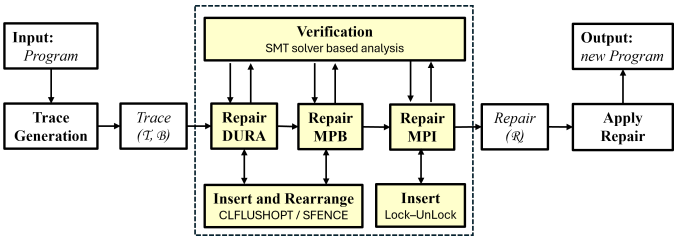



Fig. 2. Our method for repairing bugs in concurrent PM programs.

trace and $\mathcal{B}$ is the set of violated properties. In practice, $\mathcal{T}$ and $\mathcal{B}$ may be generated by any existing PM bug detection tools, including PMRace [1] which is the most recent tool. The output $\mathcal{R}$ is a set of constraints to be improved on the erroneous execution $\mathcal{T}$ such that the resulting execution, denoted $\mathcal{T}'$, no longer violates any of the PM properties in $\mathcal{B}$.

Internally, our method employs an SMT-based symbolic analysis to validate the correctness of the repaired execution $\mathcal{T}'$. If the repair $\mathcal{R}$ is incomplete, the solver returns a counterexample showing how an event interleaving in $\mathcal{T}$ violates properties in $\mathcal{B}$. Unlike PM bug-finding tools, which only detect violations, our SMT formulation reasons about event orderings and derives repair constraints that eliminate them.

As shown in Fig. 2, the repair proceeds in two steps. In *Step 1*, we analyze thread-local events to repair durability and crash-consistency bugs (also known as DURA and MPB) by reordering or adding `CLFLUSHOPT`/`SFENCE` instructions. In *Step 2*, we repair inter-thread inconsistency (MPI) bugs by inserting `Lock`–`UnLock` pairs to form PM-specific critical sections, ensuring atomic persist-time behavior and eliminating erroneous interleavings. Together, these two steps enable automatic repair of all three major classes of PM bugs.

At the core of our method lies an SMT formulation specialized for concurrent PM programs. While SMT solvers have been used to detect and repair sequential or concurrent bugs [17], [18], [19], [20], [21], [22], [23], [24], none addresses the full $(A{+}B{+}C)$ space of Fig. 1. A naive monolithic encoding for concurrent PM proved infeasible: our prototype scaled only to $< 10$ lines of C. We make it practical with a novel *two-step formulation*, which separates intra-thread and inter-thread repair, decomposing a single intractable solver call into smaller, tractable ones. This decomposition guarantees correctness of the repaired trace while preserving feasible interleavings. In doing so, it avoids exhaustive testing while ensuring the realizability and soundness of the repair.

We implemented our method as a tool built on the LLVM compiler [25] and the Z3 SMT solver [26]. LLVM is used to extract and analyze erroneous execution traces, while Z3 performs the symbolic reasoning and repair synthesis. We evaluated the tool on multi-threaded C programs including eleven PM-enabled data structures and five distributed storage applications. Compared with existing repair tools Hippocrates [15] and PMBugAssist [16], our method repairs all three categories of PM bugs and uniquely fixes all *inter-thread inconsistency* violations that prior methods could not.

TABLE I
*Px86* SEMANTICS [29] — WILL THE CPU HARDWARE GUARANTEES THAT TWO INSTRUCTIONS $(I_i, I_j)$ TAKE EFFECT IN THE EXECUTION ORDER?

| First Instruction ($I_i$) | Second Instruction ($I_j$) | | | |
|---|---|---|---|---|
| | LOAD | STORE | SFENCE | CLFLUSHOPT |
| LOAD | ✔ | ✔ | ✔ | ✔ |
| STORE | ✘ | ✔ | ✔ | **CL** |
| SFENCE | ✘ | ✔ | ✔ | ✔ |
| CLFLUSHOPT | ✘ | ✘ | ✔ | ✘ |

It also produces practical repairs for large systems such as *Memcached* [27] and *P-CLHT* [28].

In summary, this paper makes the following contributions:

- We propose the first symbolic analysis method for repairing *inter-thread inconsistency* bugs in concurrent PM programs.
- We propose a two-step approach that scales symbolic repair from toy traces to 60 K-LoC systems.
- We demonstrate the effectiveness of our method on a diverse set of benchmark programs.

## II. Background

We now review the basics of PM bugs and explain the challenges of repairing PM bugs in multi-threaded programs.

### A. Persistent Memory Semantics

From the perspective of a programmer, persistent memory provides the exact same LOAD and STORE accesses as the volatile main memory. The only difference is that data written to PM is not guaranteed to persist, e.g., in the presence of power failures, unless the programmer explicitly flushes the cache lines and adds memory fences. Different types of CPU architectures have different instructions for flushing the cache lines and adding memory fences.

For *persistent x86 (Px86)* architecture [29], `CLFLUSHOPT` (cache line flush, optimized) and `SFENCE` (store-only memory fence) are two PM-related instructions. Since modern *x86* CPUs implement the TSO (total-store order) memory model, STORE instructions executed by the CPU are first sequentialized in a *store buffer* before they take effect in memory. In contrast, LOAD instructions take effect immediately. This performance optimization makes it possible to violate the program order, e.g., for the sequence `STORE x; LOAD y`, the LOAD may take effect before the STORE.

Table I shows all possible ways to violate the program order, i.e., whether $I_i$ always take effect before $I_j$ for every pair $(I_i, I_j)$ where $I_i$ is executed before $I_j$. The symbol ✔ means they must take effect in order, while ✘ means they may take effect in reverse order. The symbol **CL**, which is a short-hand for *Cache-Line*, means that they must take effect in order only if $I_i$ and $I_j$ access memory blocks mapped to the same cache line. For brevity, we do not include other instructions such as RMW (read modify write), CLFLUSH (cache line flush, unoptimized), and CWB (equivalent to CLFLUSHOPT), since they may be simulated using the instructions shown in Table I.

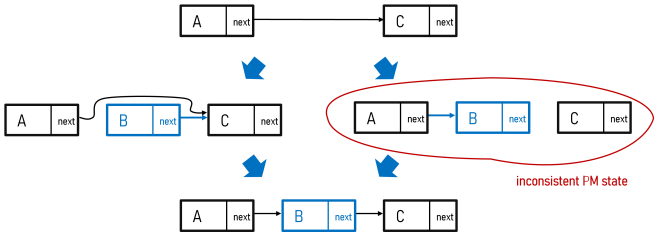
In a correctly written PM program, after executing a STORE to address `&v`, both CLFLUSHOPT and SFENCE must be

```
//Insert node 'B' in a singly-linked
//list containing nodes 'A' and 'C'
B->next = C;
A->next = B;
//Assume that 'A', 'B' and 'C' are
//all stored in PM
```

(a) code in original program

```
1  B->next = C;
2  CLFLUSHOPT(&B->next,sizeof(Node*));
3  SFENCE();
4  A->next = B;
5  CLFLUSHOPT(&A->next,sizeof(Node*));
6  SFENCE();
```

(b) code in repaired program




(c) an inconsistent PM state in the original (but not the repaired) program

Fig. 3. A buggy sequential program in (a), the repaired program in (b), and the erroneous PM state in (c).

used to ensure that the value is written to PM eventually, as in {`STORE(&v,data)... CLFLUSHOPT(&v)... SFENCE()`}. Here, CLFLUSHOPT flushes the cache line, while SFENCE adds the memory fence. Together, they guarantee that the value is written to PM by the time SFENCE finishes its execution. If either instruction is missing, there is no guarantee that the value will ever be written to PM.

### B. PM Bugs in a Sequential Thread

Among the three most common types of PM bugs studied in the literature, two are in sequential programs. The example program in Fig. 3 (a) illustrates these two types.

The assignments are meant for inserting a new node, B, to a singly-linked list consisting of nodes A and C. Assuming that the list is stored in PM, the commit order in Fig. 3 (a) is important, since power failure may occur at any moment. Ensuring that `B->next = C` takes effect before `A->next = B` means that the list remains in a consistent state, regardless of when power failure occurs.

Unfortunately, Fig. 3 (a) cannot guarantee the above property—it cannot even guarantee that either store takes effect because both CLFLUSHOPT and SFENCE are missing. Consequently, the list stored in PM may end up in any of the four states shown in Fig. 3 (c). One of the states is not acceptable since the corresponding list is broken; feeding it to a list traversal routine, for example, may lead to a crash since `B->next` may hold a garbage value.

To avoid the inconsistent PM state, a possible repair is to add the blue-colored statements in Fig. 3 (b). In particular, `CLFLUSHOPT` at Line 2 flushes the cache line that holds the written value of `B->next`, while `SFENCE` at Line 3 forces the store to take effect. Similarly, `CLFLUSHOPT` at Line 5 flushes the cache line that holds the written value of `A->next`, while `SFENCE` at Line 6 forces the store to take effect. Together, they force the store of `B->next` to persist before the store of `A->next`.

The four blue-colored statements in Fig. 3 (b) also form a *minimal* repair set. If any of them is removed, there will be either DURA or MPB violations, defined below.

**Durability (DURA) Violations:** Durability violations are the first type of common PM bugs. By *durability*, we mean that every store to a PM address `&var` *takes effect eventually*, i.e., before the program ends. This property must be enforced by adding both `CLFLUSHOPT(&var)` and `SFENCE`, following the store. For example, in Fig. 3 (b), if either `CLFLUSHOPT` at Line 5 or `SFENCE` at Line 6 is missing, the store of `A->next` at Line 4 may never take effect in PM.

While every store must be followed by its own `CLFLUSHOPT`, `SFENCE` may be shared by multiple stores. Sharing improves performance because `SFENCE` is an expensive CPU instruction. However, sharing is allowed only if two stores do not have to persist in a specific order. Otherwise, there will be a must-persist-before (MPB) violation.

**Must-Persist-Before (MPB) Violations:** MPB violations are the second type of common PM bugs. In the literature, they are also called crash-consistency bugs [16], [5]. The MPB property requires that, for two stores targeting the addresses `&var1` and `&var2`, respectively, the store to `var1` must always persist before the store to `&var2`. For the example program in Fig. 3 (b), this property is enforced by adding a pair of `CLFLUSHOPT` and `SFENCE` after each store.

If `SFENCE` is removed from Line 3, store of `B->next` will not guarantee to take effect before store of `A->next`, because `CLFLUSHOPT` itself does not force the cache line to flush immediately; thus, nothing will prevent the store of `A->next` to take effect before `B->next`, leading to the inconsistent PM state shown in Fig. 3 (c).

There are many existing methods for detecting PM bugs, but they do not work well for concurrent programs. A notable exception is PMRace [1], which is designed specifically to detect inter-thread inconsistency bugs in concurrent programs.

### C. PM Bugs in Concurrent Programs

In this paper, we refer to inter-thread inconsistency bugs [1] also as Must-Persist-Instantaneously (MPI) violations.

**Must-Persist-Instantaneously (MPI) Violations:** These are the third type of common PM bugs. They violate the requirement that each PM store and its `CLFLUSHOPT` and `SFENCE` instructions within the same thread must be executed atomically, i.e., free of undesired thread interleaving.

We illustrate MPI violations using the program in Fig. 4 (a), which has two threads. While store of `*y` in `thread2` is followed by `CLFLUSHOPT` and `SFENCE`, store of `*x` in `thread1` is not. Therefore, both DURA and MPB violations exist in the execution trace in Fig. 4 (c). Even after adding `CLFLUSHOPT` and `SFENCE` in Fig. 4 (b), without the `Lock-UnLock` pairs, there can still be bugs.

Assume that the initial state is {`*x`$\mapsto$`0,*y`$\mapsto$`0`}. Since store of `*y` in `thread2` depends on store of `*x` in `thread1`, the resulting PM state should not be {`*x`$\mapsto$`0,*y`$\mapsto$`42`} regardless of how the two threads interleave. However, as illustrated by Fig. 4 (d), it is possible to get into the undesired


(a) code with DURA/MPB violations

(c) execution of code in (a)

(b) code with MPI violations

(d) execution of code in (b)

Fig. 4. Two buggy concurrent programs in (a) and (b), and their erroneous executions in (c) and (d), respectively.

PM state. This occurs when the store of $\star$y in thread2 has taken effect in PM but store of $\star$x in thread1 has not. If power failure occurs at this moment, the value 42 of $y$ has persisted but the value of $x$ has not, thus leading to $\{\star x \mapsto 0, \star y \mapsto 42\}$.This is an *out-of-the-blue* sate that should never occur. Since $\star y \mapsto 42$ causally depends on $\star x \mapsto 42$, it is logically impossible for $\star y \mapsto 42$ to occur when $\star x \mapsto 0$.

By adding Lock–UnLock pairs shown in Fig. 4 (b), which disallow the erroneous thread interleaving in Fig. 4 (d), we avoid the inconsistent PM state and thus repair the MPI violation. When Lock–UnLock pairs are inserted to code regions of multiple threads, only one of these code regions may be executed at any moment in time.

Note that, while existing methods such as Hippocrates [15] and PMBugAssist [16] can repair DURA and MPB violations, they cannot repair MPI violations. DudeTx [30] is another method for identifying and fixing DURA bugs, but is also limited to non-concurrent settings.

For readers who are familiar with PM-based programs, we emphasize that the MPI property is different from a persistent transaction, e.g., the one enforced by TX_BEGIN() and TX_END() of the Intel PMDK library [31]. While persistent transaction ensures that multiple stores persist in an *all-or-none* fashion, it is not thread-safe and cannot enforce the MPI property illustrated by the example program in Fig. 4.

## III. Our Method

In this section, we present an overview of the repair method. Detailed algorithms of the subroutines used in the method will be presented in the subsequent sections.

### A. The Top-level Procedure

The input of our method is $(\mathcal{T}, \mathcal{B})$, where $\mathcal{T}$ is the erroneous execution trace of a program and $\mathcal{B}$ is the set of violated properties (PM bugs). Recall that Fig. 4 (c) and (d) show the example execution traces generated from the example programs in Fig. 4 (a) and (b), respectively.

**The Execution Trace:** $\mathcal{T} = e_1, \ldots, e_n$ is a sequence of events where each $e_i$ $(1 \leq i \leq n)$ represents an instance of a PM-related instruction. The event type may be LOAD, STORE, CLFLUSHOPT, SFENCE, or one of the basic operations of a persistent transaction [31] such as TX_BEGIN and TX_END. In practice, $\mathcal{T}$ and $\mathcal{B}$ may be produced by any external PM bug-detection tool (e.g., *PMRace* [1]). Generating $\mathcal{T}$ or $\mathcal{B}$ is orthogonal to our work; this paper focuses exclusively on repair.

**The Set of Properties:** $\mathcal{B} = \{b_1, \ldots, b_m\}$ is a set where each $b_i \in \mathcal{B}$ $(1 \leq i \leq m)$ represents a bug in the following syntax: the first element of $b_i$ indicates the bug type, followed by more elements, which contain information of the relevant store. The three types of common PM properties are as follows:

- DURA:[$\mathtt{I}_i$], where the STORE of instruction $\mathtt{I}_i$ must satisfy the durability property,
- MPB:[$\mathtt{I}_i$,$\mathtt{I}_j$], where the STORE of instruction $\mathtt{I}_i$ must persist before the STORE of instruction $\mathtt{I}_j$, and
- MPI:[$\mathtt{I}_i$,$\mathtt{I}_j$], where the STORE of instruction $\mathtt{I}_i$ must persist instantaneously, and in particular, before the STORE of instruction $\mathtt{I}_j$ from another thread.

Inside our symbolic analysis procedure, each property $b_i \in \mathcal{B}$ translates to a logical constraint that precisely captures the condition under which the property is violated.

**Algorithm 1** Repairing PM bugs in $\mathcal{B}$ manifested by trace $\mathcal{T}$.

1: **for all** $t \in$ *Threads* **do**
2: $\quad \mathcal{T}_t \leftarrow$ ProjectTraceToThread $(\mathcal{T}, t)$
3: $\quad \mathcal{R}_t \leftarrow$ RepairBugsWithinThread $(\mathcal{T}_t, \mathcal{B})$ *//Step 1: Algorithm 2*
4: **end for**
5: Let $\mathcal{T}'$ be the result of applying $\mathcal{R}_t$ $\mathcal{T}$ for all $t \in$ *Threads*;
6: $\mathcal{R} \leftarrow$ RepairBugsAcrossThreads $(\mathcal{T}', \mathcal{B})$ *//Step 2: Algorithm 3*
7: **return** $\mathcal{R}$;

Algorithm 1 shows our top-level procedure. Internally, it relies on two subroutines, corresponding to Algorithms 2 and 3 to be presented in the subsequent sections. For now, it suffices to say that our method first repairs DURA and MPB violations by analyzing the *thread-local* events in $\mathcal{T}_t$ for each $t \in$ *Threads*. Here, $\mathcal{T}_t$ is the execution trace $\mathcal{T}$ projected to the $t$-th thread, and $\mathcal{R}_t$ is the corresponding repair. More specifically, $\mathcal{R}_t$ is a set of *commit-order* constraints that must be enforced within a thread by adding/reordering CLFLUSHOPT and SFENCE instructions.

Next, our method repairs the MPI violations by analyzing events from all threads at the same time. Here, $\mathcal{T}'$ is the partially repaired trace of applying $\mathcal{R}_t$ to $\mathcal{T}$ for all $t \in$ *Threads*. Given $\mathcal{T}'$ and $\mathcal{B}$, we compute the repair $\mathcal{R}$, which is a set of *thread-interleaving* constraints that must be enforced by adding Lock–UnLock pairs to threads involved in the MPI violations.

While throughout this work we assume that the CPU implements the *Px86-TSO* semantics, by changing the symbolic encoding (to be presented in Section IV), our method can be applied to other CPU architectures and memory models. However, this is orthogonal to our core repair technique and thus is left for future work.

### B. The Two Repair Steps

A key innovation of our method is decomposing the repair computation into two steps. While a monolithic symbolic

approach could, in principle, repair all persistent-memory (PM) bugs in one step, our experiments showed that such an encoding scales only to traces from tiny programs (fewer than ten lines of C code). In contrast, our *two-step symbolic analysis* efficiently repairs large systems such as Memcached [27] and ClevelHashing [32], containing over 20K and 60K lines of code, respectively.

**Step 1: Repairing Bugs within Each Thread.** Given a thread-local trace $\mathcal{T}_t$, our method fixes DURA and MPB violations by reordering and, when necessary, inserting `CLFLUSHOPT`/`SFENCE` instructions to produce the local repair $\mathcal{R}_t$. The process is *greedy* (each new instruction is added only when reordering existing ones cannot eliminate the violation) and terminates as soon as all local bugs are removed. Empirically, this yields compact, high-quality repairs, typically requiring only a few new instructions per thread.

**Step 2: Repairing Bugs across Threads.** After applying all $\mathcal{R}_t$ to obtain $\mathcal{T}'$, only MPI violations remain. To repair them, we construct a logical encoding that captures the persist-time behavior and interleavings of all threads. If the SMT solver finds a model violating any MPI property, our method blocks the erroneous interleaving by inserting `Lock-UnLock` pairs in the affected threads, forming minimal PM-specific critical sections. They ensure that a *write* and its subsequent *flush/fence* operations execute atomically with respect to any concurrent *read*, guaranteeing that the write's effects are durably visible before the read occurs.

### C. The Running Example

Putting it all together, we now apply the repair method to programs in Fig. 4 (a) and Fig. 4 (b) whose erroneous executions are shown in Fig. 4 (c) and Fig. 4 (d), respectively.

The trace in Fig. 4 (b) is $\mathcal{T}$ = `Thread_1:[STORE x], Thread_2: [STORE y], Thread_2:[CLFLUSHOPT &y], Thread_2:[SFENCE]`. The value stored to `*y` in the second thread depends on `*x`, which in turn depends on the value stored to `*x` in the first thread. Since the first thread has no `CLFLUSHOPT` and `SFENCE` at all, there is a DURA violation for the store of `*x`. Furthermore, there is an MPI violation for the stores of `*x` and `*y`.

In Step 1, our method repairs DURA and MPB violations by inserting the necessary `CLFLUSHOPT` and `SFENCE` instructions after the store of `*x` in `thread1`. We encode the program as a symbolic formula whose satisfiability corresponds to the existence of a DURA violation. When the SMT solver returns a model, we extract the erroneous commit order and block it by adding or rearranging flush and fence instructions. This *check-and-then-block* process repeats until all DURA and MPB violations are eliminated.

Step 2 targets MPI violations across threads. We insert `Lock`/`UnLock` pairs to form atomic regions so that, at any time, only one such region executes, thereby preventing erroneous interleavings. Again, we encode the program symbolically: a satisfying model indicates the presence of an MPI violation. From that model, we extract the erroneous interleaving and block it by inserting the minimal lock/unlock pair. This *check-and-then-block* process continues until no MPI violations remain.

## IV. The Verification Procedure

Before presenting details of our two-step repair method in Section V, we present the verification procedure in this section. This is because the verification procedure is a fundamental building block of the repair method.

### A. The Symbolic Formula ($\Phi \wedge \neg\Phi_{\mathcal{B}}$)

Given an execution trace $\mathcal{T}$ and a set $\mathcal{B}$ of properties, the verification procedure checks if it is possible for events in $\mathcal{T}$ to interleave and commit in such an order that violates some properties in $\mathcal{B}$. This is accomplished by constructing a logical formula that is satisfiable *if and only if* there exists a total order of events under which some properties in $\mathcal{B}$ are violated.

While verification has a similar functionality to a bug-finding tool, there are two differences. First, verification produces important diagnostic information, i.e., a counterexample in the form of a total order of events in $\mathcal{T}$ under which a property in $\mathcal{B}$ is violated. This provides feedback on the subsequent repair method regarding how to rearrange the `CLFLUSHOPT/SFENCE` instructions to avoid the property violation. Second, verification presented here is conducted symbolically in that it covers all possible execution scenarios for the events in $\mathcal{T}$, including both thread interleavings and commit orders of the PM `STOREs`.

Our method differs from prior work on using SMT solvers to analyze both PM bugs [16] and concurrency bugs [17], [18], [19], [20], [21]. As mentioned earlier, these methods correspond to either (A+C) or (B+C) shown in Fig. 1, whereas our method presented in this section corresponds to (A+B+C).

### B. Constructing Subformula $\Phi$

Let $\Phi := \Phi_{PC} \wedge \Phi_{Seq} \wedge \Phi_{PT}$, where

- $\Phi_{PC}$ encodes the program order of instructions in $\mathcal{T}$,
- $\Phi_{Seq}$ encodes the sequential order of `STOREs`, and
- $\Phi_{PT}$ encodes the persistent time of each `STORE`.

To construct these subformulas, we need to define the following two sets of symbolic variables:

- **Program Counter ($PC_{i_t}$)** denotes the time when the $i$-th instruction ($I_i$) is executed by thread $t \in \mathit{Threads}$.
- **Persistent Time ($PT_{i_t}$)** denotes the time when a `STORE` of the instruction $I_i$ takes effect in PM.

The values of $PC_{i_t}$ and $PT_{i_t}$ are non-negative integers. With these definitions, we construct the subformulas of $\Phi$.

**Subformula** $\Phi_{PC}$ requires that the value of each $PC_{i_t}$ is unique, and is in the range between 1 and length($\mathcal{T}$). Here, length($\mathcal{T}$) represents the number of events in $\mathcal{T}$.

**Subformula** $\Phi_{Seq}$ requires that the `STOREs` within a thread cannot be re-ordered (it will change the program semantics); however, `CLFLUSHOPT, SFENCE` and `Lock-UnLock` can be re-ordered, to capture possible ways of repairing the program.

$\Phi_{PC} := (PC_{1_{t1}} < 2) \wedge (PC_{1_{t1}} > 0) \wedge (PC_{1_{t2}} < 4) \wedge (PC_{1_{t2}} > 0) \wedge PC_{2_{t2}} < 4) \wedge (PC_{2_{t2}} > 0) \wedge (PC_{3_{t2}} < 4) \wedge (PC_{3_{t2}} > 0) \wedge UNIQUE(PC_{1_{t2}}, PC_{2_{t2}}, PC_{3_{t2}})$

$\Phi_{Seq} := True$

$\Phi_{PT} := (PT_{1_{t1}} \geq PC_{1_{t1}}) \wedge (PT_{1_{t2}} \geq PC_{1_{t2}}) \wedge (PT_{1_{t2}} \leq PC_{3_{t2}})$

$\neg\Phi_B := (PT_{1_{t1}} \geq \texttt{PT\_MAX}) \vee ((PC_{1_{t1}} < PC_{1_{t2}}) \wedge (PT_{1_{t1}} > PT_{1_{t2}}))$

Fig. 5. Example sub-formulas for the trace in Fig. 4 (c).

**Subformula** $\Phi_{PT}$ requires that the $PT$ value of each `STORE` is set between when `STORE` is executed and when the first FENCE is executed. Let `PT_MAX` = length($\mathcal{T}$). For instruction $I_i$ executed by $t \in Threads$, in particular, $PT_{i_t} \geq$ `PT_MAX` means that the `STORE` in $I_i$ is not yet flushed to the PM.

### C. Constructing Subformula $\neg\Phi_{\mathcal{B}}$

Since $\Phi_B$ encodes the condition under which properties in $\mathcal{B}$ hold, $\neg\Phi_B$ means that some of these properties are violated.

**Subformula** $\Phi_B$ encodes the condition under which properties in $\mathcal{B}$ hold, based on the following property types:

- For `DURA:[I`$_{i_t}$`]`, it encodes $(PT_{i_t} <$ `PT_MAX`$)$, which requires the `STORE` instruction $i_t$ to persist before the end of the program.
- For `MPB:[I`$_{i_t}$`,I`$_{j_t}$`]`, it encodes $(PT_{i_t} < PT_{j_t})$.
- For `MPI:[I`$_{i_{t1}}$`,I`$_{j_{t2}}$`]`, where the instructions are executed by $t1, t2 \in Threads$, it encodes $(PC_{i_{t1}} < PC_{j_{t2}}) \implies (PT_{i_{t1}} \leq PT_{j_{t2}})$.

The MPI formula above, in particular, means that if the `STORE` instruction $j_{t2}$ reads a value written by the `STORE` instruction $i_{t1}$, denoted by $(PC_{i_{t1}} < PC_{j_{t2}})$, then $i_{t1}$ must not persist after $j_{t2}$, denoted by $(PT_{i_{t1}} \leq PT_{j_{t2}})$.

### D. Example Formulas

We illustrate our method by constructing the formulas for programs in Fig. 4. Assuming that $\mathcal{T}$ is the trace in Fig. 4 (c), consists of an event from thread $t1$ (denoted $1_{t1}$) followed by three events from thread $t2$ (denoted $1_{t2}$, $2_{t2}$ and $3_{t2}$), Fig. 5 shows the subformulas.

For brevity, in $\Phi_{PC}$, $UNIQUE(PC_{1_{t2}}, PC_{2_{t2}}, PC_{3_{t2}})$ is a short-hand notation for the pair-wise inequality constraint $(PC_{1_{t2}} \neq PC_{2_{t2}} \wedge PC_{2_{t2}} \neq PC_{3_{t2}} \wedge PC_{1_{t2}} \neq PC_{3_{t2}})$. It means that, within $t2 \in Threads$, the three events have different $PC$ values.

The reason why $\Phi_{Seq} := True$ is because, in each of the two threads, there is only one `STORE` operation.

Inside $\neg\Phi_B$, we encode both the DURA violation for $1_{t1}$ (the 1st event of thread $t1$) and the MPB violation between $1_{t1}$ and $1_{t2}$ (the 1st event of thread $t2$).

Since the formula $\Phi$ defined according to Fig. 5 is satisfiable, the SMT solver may return a total order: $T_1$`:[STORE x]`,$T_2$`:[STORE y]`,$T_2$`:[CLFLUSHOPT &y]`,$T_2$`:[SFENCE]`. It is an erroneous scenario where the `STORE` of `*x` by thread $T_1$ is never explicitly flushed and fenced to PM, causing a DURA violation. Another violation also manifested in this total order is the MPI violation associated with the `STORE`s of `*x` and `*y`.

Thus, the solution returned by the SMT solver provides important diagnostic information to be used by our method in the next section to repair PM bugs.

## V. The Two-Step Repair Approach

Our repair method relies on invoking the verification procedure as a subroutine. In step 1, verification is applied to each thread-local trace $\mathcal{T}_t$ to compute the repair $\mathcal{R}_t$ for all $t \in Threads$. Applying $\mathcal{R}_t$ to $\mathcal{T}$ produces a partially repaired trace $\mathcal{T}'$. In step 2, verification is applied to $\mathcal{T}'$ to compute the repair $\mathcal{R}$.

### A. Repairing Bugs in Each Thread

The goal is to leverage the verification routine to repair DURA and MPB violations within each thread. Algorithm 2 shows the procedure, which takes the thread-local trace $\mathcal{T}_t$ and $\mathcal{B}$ as input and returns the thread-local repair $\mathcal{R}_t$ as output. $\mathcal{R}_t$ is a set of *commit-order* constraints that, when imposed on $\mathcal{T}_t$, eliminates all DURA and MPB violations in $\mathcal{T}_t$.

**Algorithm 2** Repair DURA/MPB bugs in $\mathcal{T}_t$ for $t \in Threads$.

```
1: while checkSAT (Φ ∧ ¬Φ_B) do  //call SMT — still have bugs?
2:     R_t ← True
3:     while checkSAT (Φ ∧ R_t ∧ ¬Φ_B) do  //call SMT — bad execution?
4:         ψ_bad ←getRelativeOrdering (Φ ∧ R_t ∧ ¬Φ_B)
5:         R_t ← R_t ∧ ¬ψ_bad
6:     end while
7:     if checkSAT (Φ∧R_t) then      //call SMT — exists a good execution?
8:         return R_t;
9:     end if
10:    T_t ←addFlushFenceInstructions(T_t, R_t)
11: end while
```

Initially, $\mathcal{R}_t$ is set to $True$, meaning that there are no constraints. Inside Algorithm 2, our method first constructs the formula $(\Phi \wedge \neg\Phi_{\mathcal{B}})$, but only using the thread-local $\mathcal{T}_t$ and $\mathcal{B}$, and then checks if the formula is satisfiable. When $\mathcal{R}_t = True$, the formula $(\Phi \wedge \mathcal{R}_t \wedge \neg\Phi_{\mathcal{B}})$ at Line 3 reduces to $(\Phi \wedge \neg\Phi_{\mathcal{B}})$.

During the subsequent iterations, as long as the bad executions have not been eliminated, the formula $(\Phi \wedge \mathcal{R}_t \wedge \neg\Phi_{\mathcal{B}})$ remains satisfiable, and the solution returned by the SMT solver corresponds to an erroneous commit-order.

This erroneous commit-order is used to generate a *relative ordering* $\psi_{bad}$, representing some bad permutation of instructions in $\mathcal{T}_t$ that must be blocked, using the blocking constraint $\neg\psi_{bad}$. The repair $\mathcal{R}_t$ can be regarded as a conjunction of all these blocking constraints. By blocking the *relative orders* generated in all iterations of the while-loop, our method guarantees that no bad execution exists when the procedure reaches Line 7 of Algorithm 2.

The relative order $\psi_{bad}$ is computed as follows. First, from a satisfying solution returned by the SMT solver, we extract the concrete values of the $PC$ variables for all instructions. Then, we leverage these concrete $PC$ values to figure out the relative ordering of all pairs of instructions, $I_i$ and $I_j$, where $i \neq j$. To construct the blocking constraint $\neg\psi_{bad}$, we

**Iteration 1:**
**order:** $T_1$:[CLFLUSHOPT &x], $T_1$:[SFENCE], $T_1$:[STORE x], $T_2$:[STORE y], $T_2$:[CLFLUSHOPT &y], $T_2$:[SFENCE]
**relative order:** $\psi_{bad} = (PC_{2_{t1}} < PC_{1_{t1}}) \wedge (PC_{3_{t1}} < PC_{1_{t1}})$
**Iteration 2:**
**order:** $T_1$:[CLFLUSHOPT &x], $T_1$:[STORE x], $T_1$:[SFENCE], $T_2$:[STORE y], $T_2$:[CLFLUSHOPT &y], $T_2$:[SFENCE]
**relative order:** $\psi_{bad} = (PC_{2_{t1}} < PC_{1_{t1}}) \wedge (PC_{1_{t1}} < PC_{3_{t1}})$
**Iteration 3:**
**order:** $T_1$:[SFENCE], $T_1$:[STORE x], $T_1$:[CLFLUSHOPT &x], $T_2$:[STORE y], $T_2$:[CLFLUSHOPT &y], $T_2$:[SFENCE]
**relative order:** $\psi_{bad} = (PC_{3_{t1}} < PC_{1_{t1}}) \wedge (PC_{1_{t1}} < PC_{2_{t1}})$
**Iteration 4:**
UNSAT
**Repair**
$\mathcal{R}_t := \neg((PC_{3_{t1}} < PC_{1_{t1}}) \wedge (PC_{1_{t1}} < PC_{2_{t1}})) \wedge \neg((PC_{2_{t1}} < PC_{1_{t1}}) \wedge (PC_{1_{t1}} < PC_{3_{t1}})) \wedge \neg((PC_{2_{t1}} < PC_{1_{t1}}) \wedge (PC_{3_{t1}} < PC_{1_{t1}}))$
**Valid & Bug free Execution (Line 7):**
**order:** $T_1$:[STORE x], $T_1$:[CLFLUSHOPT &x], $T_1$:[SFENCE], $T_2$:[STORE y], $T_2$:[CLFLUSHOPT &y], $T_2$:[SFENCE]

Fig. 6. Blocking constraints generated while repairing the DURA and MPB bugs in the running example.

only include the relative orderings of STORE–CLFLUSHOPT and STORE–SFENCE pairs because, while repairing a PM program, we only intend to rearrange these PM-specific flush/fence instructions.

Finally, we check if a good execution still exists in $\mathcal{T}$ after it is repaired using $\mathcal{R}_t$. This is important because, sometimes, blocking constraints in $\mathcal{R}_t$ are too strong; they kill not only bad executions but also all of the good executions. In such a case, the repair $\mathcal{R}_t$ is *unrealizable*, so we add more flush/fence instructions to $\mathcal{T}_t$ (Line 10 of Algorithm 2) and try again.

Checking if $\mathcal{R}_t$ is *unrealizable* (Line 7) is performed by feeding $(\Phi \wedge \mathcal{R}_t)$ to the Z3 SMT solver. This formula differs from the verification formula in that it excludes $\neg\Phi_{\mathcal{B}}$. We return $\mathcal{R}_t$ as a valid repair only if $(\Phi \wedge \mathcal{R}_t)$ is satisfiable.

**An Example for Algorithm 2:** We now apply Algorithm 2 to the execution trace in Fig. 4 (c). Recall that, in this step, $\Phi_B$ includes only the DURA and MPB violations. Another observation is that, as shown in the execution trace, initially no CLFLUSHOPT and SFENCE instructions exist for the STORE of ⋆x. Hence, we must add new CLFLUSHOPT and SFENCE instructions (to the end of the trace) and then run Algorithm 2, until a valid repair $\mathcal{R}_t$ is computed. This extends the trace $\mathcal{T}$ with CLFLUSHOPT(x) and SFENCE in thread 1.

With the extended trace, Fig. 6 shows the repair computation. Specifically, the while-loop at Line 3 of Algorithm 2 runs for 3 iterations before returning UNSAT in the $4^{th}$ iteration. During each iteration, the erroneous order returned by the SMT solver and the blocking constraint (relative order) computed by our method are shown in Fig. 6.

Once the SMT solver returns UNSAT at Line 3 of Algorithm 2, we check if a good execution exists (Line 7). In this example, the check at Line 7 returns SAT, implying that a good execution still exists. This is shown in the last two rows of Fig. 6.

### B. Repairing Bugs across Threads

Assuming that DURA and MPB bugs are repaired in Step 1, we explain how to perform Step 2 of our repair method. The goal is to leverage the same verification routine to repair MPI violations. Algorithm 3 shows the procedure, which takes the modified execution trace $\mathcal{T}'$ and $\mathcal{B}$ as input and returns the repair $\mathcal{R}$ as output. As mentioned earlier, $\mathcal{T}'$ is the result of applying to $\mathcal{T}$ all of the thread-local repairs $\mathcal{R}_t$ for $t \in Threads$.

**Algorithm 3** Repair MPI bugs in partially repaired trace $\mathcal{T}'$.

```
1: R ← True
2: Φ ← constructFormula(T', B, R)
3: while checkSAT(Φ ∧ ¬Φ_B) do        //call SMT — bad interleaving?
4:     R ← addLocks(R, Φ)
5:     Φ ← constructFormula(T', B, R)
6: end while
7: return R;
```

Inside Algorithm 3, our method begins by initializing $\mathcal{R}$ and then constructing the formula $\Phi$. Since the MPI bugs in $\mathcal{B}$ are *inter-thread* bugs, while constructing $\Phi$ (Line 2), we consider events executed by all threads. There is a significant difference from how the verification routine was used in Algorithm 2 to repair DURA and MPB bugs. That is, for MPI bugs, we add another subformula $\Phi_{lock}$, which encodes the semantics of Mutex locks. In other words, the formula is $\Phi := \Phi_{PC} \wedge \Phi_{Seq} \wedge \Phi_{PT} \wedge \Phi_{Lock}$.

**Subformula** $\Phi_{Lock}$ encodes the requirement that, once a lock has been acquired by a thread, no other threads should acquire the same lock, until the lock is released by the first thread. More formally, let $M$ be the set of all locks and, for each lock $m \in M$, let $lock_k^m$ and $unlock_k^m$ be one pair of Lock–UnLock operations, and $lock_l^m$ and $unlock_l^m$ be another pair of Lock–UnLock operations. Let $K$ be the set of Lock–Unlock pairs such that $k, l \in K$ and $k \neq l$.

Let $PC_{lock_k^m}$ be the time when $lock_k^m$ is executed and let $PC_{unlock_k^m}$ be the time when $unlock_k^m$ is executed. $[PC_{lock_k^m}, PC_{unlock_k^m}]$ is the time interval for the corresponding critical section.

Since any two such time intervals from different threads, denoted $[PC_{lock_k^m}, PC_{unlock_k^m}]$ and $[PC_{lock_l^m}, PC_{unlock_l^m}]$, must be non-overlapping, we know that either $(PC_{unlock_k^m} < PC_{lock_l^m})$ holds or $(PC_{lock_k^m} > PC_{unlock_l^m})$ holds.

Thus, for all lock $m \in M$ and for all Lock–Unlock pairs $k, l \in K$, we define the subformula $\Phi_{Lock} :=$

$$\bigwedge_{\forall m} \bigwedge_{\forall k} \bigwedge_{\forall l} (PC_{unlock_k^m} < PC_{lock_l^m}) \vee (PC_{lock_k^m} > PC_{unlock_l^m})$$

With this updated definition of formula $\Phi$, our method checks at Line 3 of Algorithm 3 if an MPI violation exists. The check returns true *if and only if* an MPI violation exists. This is when the repair $\mathcal{R}$ computed so far has not yet eliminated all of the erroneous executions.

At Line 4 of Algorithm 3, our method repairs by adding Lock–UnLock pairs inside the subroutine addLocks($\mathcal{R}, \Phi$). The result is an updated repair $\mathcal{R}$. To minimize $\mathcal{R}$, it adds

**Iteration 1:**
**order:** $T_1$:[STORE x], $T_2$:[STORE y], $T_2$:[CLFLUSHOPT &y], $T_2$:[SFENCE], $T_1$:[CLFLUSHOPT &x], $T_1$:[SFENCE]
**violated property:** $((PC_{1_{t1}} < PC_{1_{t2}}) \wedge (PT_{1_{t1}} < PT_{1_{t2}}))$
**repaired trace:** $T_1$:[Lock &l], $T_1$:[STORE x], $T_1$:[CLFLUSHOPT &x], $T_1$:[SFENCE], $T_1$:[UnLock &l], $T_2$:[Lock &l], $T_2$:[STORE y], $T_2$:[CLFLUSHOPT &y], $T_2$:[SFENCE], $T_2$:[UnLock &l]
**Iteration 2:**
UNSAT

Fig. 7. Blocking constraints generated in Step 2 of our method for the running example in Fig. 4.

Lock–UnLock one at a time. With the updated repair $\mathcal{R}$, our method constructs the updated formula $\Phi$ (Line 5) and to check for MPI violations again. This process continues until all the MPI bugs are repaired.

**An Example for Algorithm 3:** We now apply Algorithm 3 to the execution trace in Fig. 4 (d). Recall that, in this case, all DURA and MPB bugs have been repaired. However, the Lock–UnLock pairs shown in Fig. 4 (c) have not been added. As a result, there exists an MPI violation.

Fig. 7 shows the repair process. During the first iteration of the while-loop in Algorithm 3, the SMT solver returns an erroneous order. By adding Lock–UnLock pairs to block it, as shown in the repaired trace, we make the updated formula $\Phi$ in the second loop iteration UNSAT. Thus, Algorithm 3 terminates after two iterations.

**Ensuring Deadlock Freedom:** While repairing MPI bugs, our method adds Lock–UnLock pairs to enforce atomic persist-time behavior. To ensure that these additions never introduce deadlocks, we rely on the following structural properties of the generated critical sections. (1) Each new atomic region encloses only the STORE operations involved in the MPI violation and their associated CLFLUSHOPT and SFENCE instructions, without nesting or overlapping with other critical sections. (2) All locks introduced by our repair are independent of the program's existing synchronization primitives and are acquired according to a fixed global order, thereby eliminating the possibility of circular waiting. Since our construction of the repair is verified, the resulting execution trace $\mathcal{T}'$ preserves feasible interleavings while guaranteeing that no cyclic dependencies exist in the lock acquisition graph. Consequently, the repaired program is provably free of deadlocks.

## VI. Experiments

We implemented our method as a tool built on the LLVM compiler [25] and the Z3 SMT solver [26]. We leverage LLVM to generate instrumented binaries that record execution traces at runtime, and the Z3 Python API to solve all symbolic constraints. Our trace generator transforms the original program to an instrumented program that can generate its own execution trace at runtime. Each execution trace captures PM STORE/LOAD operations, FLUSH/FENCE instructions, and calls to PMDK libraries [31], consistent with prior PM bug detectors such as PMRace [1]. Using this setup, we successfully reproduced all PM bugs reported in prior work [1].

TABLE II
Statistics of the benchmark programs.

| Name | Description | LoC | PM Variables |
|---|---|---|---|
| Motiv | Two-threaded PM-enabled program | 28 | 2 |
| DQueue | Multi-threaded PM-enabled double-ended queue | 202 | 36 |
| Heap | Multi-threaded PM-enabled heap | 210 | 13 |
| Queue | Multi-threaded PM-enabled queue | 119 | 102 |
| List | Multi-threaded PM-enabled singly-linked list | 286 | 84 |
| Hash | Multi-threaded PM-enabled hash table | 261 | 180 |
| PriorityQueue | Multi-threaded PM-enabled priority queue | 190 | 66 |
| DoublyList | Multi-threaded PM-enabled doubly-linked list | 201 | 66 |
| Graph | Multi-threaded PM-enabled graph | 176 | 92 |
| Set | Multi-threaded PM-enabled set | 169 | 22 |
| Stack | Multi-threaded PM-enabled stack | 174 | 44 |
| Memcached | High-performance event-based key-value store [27] | 23,175 | 15,241 |
| P-CLHT | Crash-consistent hash table [28] | 6,403 | 11 |
| Fast-Fair | B+-Tree with failure-atomic shift / re-balancing [33] | 1,378 | 588 |
| ClevelHashing | Lock-free hashing index with async. resizing [32] | 60,303 | 377 |
| CCEH-PMDK | Cache line conscious extensible hashing [34] | 1,278 | 44 |

### A. Experimental Setup

We evaluated our method on two benchmark suites, whose statistics are shown in Table II. The first suite includes eleven PM-enabled concurrent data structures extending traditional lock-based designs to persistent memory. The second suite includes five larger applications from prior work [1], including distributed storage systems such as *Memcached* [27] and *P-CLHT* [28]. For each program, Table II lists its description, the number of lines of code (LoC), and the number of PM-stored variables.

As mentioned earlier, our repair method takes an erroneous execution trace $\mathcal{T}$ and $\mathcal{B}$ as input. During the experiments, PM bugs ($\mathcal{B}$) exposed by these execution traces ($\mathcal{T}$) are all confirmed. For the five larger applications, in particular, the PM bugs directly came from the prior work [1] and they had been confirmed by developers of the most recent PM bug detection tool[1].

While we only report results of experiments conducted for programs with 2 threads, our method has no problem of handling more threads. Our method also has no issue handling long execution traces or large code bases. For example, our method successfully handled *P-CLHT* whose execution trace has 394K events, and *Memcached* which has more than 15K PM variables.

We ran all experiments on a computer with an Intel Xeon W-2245 CPU, 128 GB RAM, and Ubuntu 20.04. Our experiments were designed to answer the following research questions (RQs):

- RQ1. Does our method advance the state of the art? In particular, how does it compare to Hippocrates [15] and PMBugAssist [16], the two existing PM bug repair methods?
- RQ2. Does our method use the SMT solver efficiently? In particular, does it avoid the potentially long running time? To answer this question, we looked at the detailed statistics of our two-step repair approach.
- RQ3. Do the repairs computed by our method look reasonable? To answer this question, we manually inspected

TABLE III
COMPARISON WITH HIPPOCRATES [15] AND PMBUGASSIST [16]. OUR METHOD REPAIRED 207 PREVIOUSLY UNFIXABLE MPI BUGS ACROSS 16 BENCHMARKS, ACHIEVING HIGHER COVERAGE AND FASTER RUNTIME.

| Name | Bugs Repaired by Hippocrates [15] | | | Bugs Repaired by PMBugAssist [16] | | | Bugs Repaired by Our New Method | | |
|---|---|---|---|---|---|---|---|---|---|
| | DURA | MPB | MPI | DURA | MPB | MPI | DURA | MPB | MPI |
| Motiv | 2 | 0 | 0 | 2 | 0 | 0 | 2 | 0 | 1 |
| DQueue | 14 | 0 | 0 | 14 | 6 | 0 | 14 | 6 | 17 |
| Heap | 25 | 0 | 0 | 25 | 20 | 0 | 25 | 20 | 17 |
| Queue | 14 | 0 | 0 | 14 | 5 | 0 | 14 | 5 | 41 |
| List | 14 | 0 | 0 | 14 | 9 | 0 | 14 | 9 | 32 |
| Hash | 10 | 0 | 0 | 10 | 4 | 0 | 10 | 4 | 12 |
| PriorityQueue | 10 | 0 | 0 | 10 | 3 | 0 | 10 | 3 | 13 |
| DoublyList | 14 | 0 | 0 | 14 | 8 | 0 | 14 | 8 | 29 |
| Graph | 29 | 0 | 0 | 29 | 17 | 0 | 29 | 17 | 13 |
| Set | 10 | 0 | 0 | 10 | 5 | 0 | 10 | 5 | 21 |
| Stack | 4 | 0 | 0 | 4 | 1 | 0 | 4 | 1 | 3 |
| Memcached | 11 | 0 | 0 | 11 | 8 | 0 | 11 | 8 | 6 |
| P0CLHT | 25 | 0 | 0 | 25 | 10 | 0 | 25 | 10 | 1 |
| Fast0Fair | 6 | 0 | 0 | 6 | 0 | 0 | 6 | 0 | 1 |
| ClevelHashing | 15 | 0 | 0 | 15 | 8 | 0 | 15 | 8 | 0 |
| CCEH0PMDK | 16 | 0 | 0 | 16 | 7 | 0 | 16 | 7 | 0 |

TABLE IV
DETAILED PERFORMANCE INFORMATION OF OUR TWO-STEP APPROACH FOR COMPUTING THE REPAIR.

| Name | Repair Time (s) | | | Calls to SMT Solver | | | Instructions Added | | |
|---|---|---|---|---|---|---|---|---|---|
| | Step 1 | Step 2 | Total | Step 1 | Step 2 | Total | clflush | sfence | lk-unlk |
| Motiv | 0.01 | 0.03 | 0.04 | 0 | 2 | 2 | 2 | 3 | 2 |
| DQueue | 4.49 | 1.67 | 6.16 | 25 | 13 | 38 | 14 | 17 | 17 |
| Heap | 63.69 | 21.28 | 84.97 | 97 | 4 | 101 | 25 | 10 | 5 |
| Queue | 25.28 | 16.80 | 42.08 | 25 | 23 | 48 | 14 | 20 | 23 |
| List | 9.46 | 4.13 | 13.59 | 29 | 16 | 45 | 14 | 23 | 22 |
| Hash | 4.76 | 2.17 | 6.93 | 20 | 11 | 31 | 10 | 13 | 14 |
| PriorityQueue | 2.94 | 2.73 | 5.67 | 10 | 11 | 21 | 10 | 14 | 14 |
| DoublyList | 16.15 | 4.95 | 21.20 | 37 | 16 | 53 | 14 | 23 | 22 |
| Graph | 97.41 | 14.03 | 111.44 | 99 | 5 | 104 | 29 | 15 | 5 |
| Set | 51.81 | 9.50 | 61.31 | 21 | 12 | 33 | 10 | 20 | 13 |
| Stack | 0.26 | 0.17 | 0.43 | 5 | 4 | 9 | 4 | 7 | 5 |
| Memcached | 103.28 | 3.53 | 106.81 | 44 | 7 | 51 | 11 | 11 | 11 |
| P-CLHT | 2.29 | 1.80 | 4.09 | 54 | 2 | 56 | 24 | 10 | 2 |
| Fast-Fair | 8.25 | 41.24 | 49.49 | 1 | 2 | 3 | 6 | 1 | 2 |
| ClevelHashing | 142.78 | 0.01 | 142.79 | 38 | 0 | 38 | 15 | 9 | 0 |
| CCEH-PMDK | 60.90 | 0.01 | 60.91 | 15 | 0 | 15 | 15 | 6 | 0 |

all repairs, including the *inter-thread inconsistency bugs* (MPI) bugs in the five large applications.

### B. Experimental Results for RQ1

To answer RQ1, we experimentally compared our method with Hippocrates [15] and PMBugAssist [16] on all benchmark programs to see if our method can repair more PM bugs. The results are presented in Table III, which shows the number of repaired DURA, MPB and MPI bugs by each method.

As we expected, Hippocrates repaired only DURA bugs, and PMBugAssist repaired only DURA and MPB bugs; this should not come as a surprise because these two existing methods were designed to repair only these bugs. They were not designed to handle multi-threaded programs.

In contrast, our method is designed to repair all PM bugs, especially the MPI bugs that are unique to multi-threaded programs. As a result, our method repaired 207 MPI bugs that could not be repaired by the existing methods, in addition to all of the DURA and MPB bugs that could be repaired by the existing methods.

### C. Experimental Results for RQ2

To answer RQ2, we measured the time taken by our method to repair PM bugs. The results are presented in Table IV, which shows the breakdown of the repair time into the two steps, as well as the number of calls to the SMT solver, and the number of instructions added. Recall that our method relies on a two-step approach to add `CLFLUSHOPT`s and `SFENCE`s in Step 1, and then add `Lock-UnLock` pairs in Step 2. As shown by Column 4 of Table IV, the total repair time ranges from a few seconds to a few minutes, indicating that our method is reasonably fast even for the largest applications, *Memcached* and *ClevelHashing*, which have 32,175 and 60,303 lines of C/C++ code, respectively. Compared to what is typically needed to manually repair PM bugs, the time taken by our method is negligible.

Our repair method was efficient primarily because of the *two-step* symbolic analysis. Furthermore, in general, Step 1 took more time than Step 2. For *Heap*, in particular, Step 1 took 63.69 seconds whereas Step 2 took 21.28 seconds. This is because Step 1 tends to make more calls to the SMT solver as shown by breakdown of the calls in Columns 5-7. For *Heap*, in particular, Step 1 made 97 calls to the SMT solver whereas Step 2 made only 4 calls.

However, a call to the SMT solver made in Step 2 may be more expensive than a call made in Step 1, since it has to encode the interactions among multiple threads. By separating the repair computation for each thread in Step 1, our method significantly reduces the cost of the repair computation in Step 2. Finally, Columns 8-10 show a breakdown of the added instructions, which come from all three categories, indicating that the repair computed by our method is non-trivial.

### D. Experimental Results for RQ3

To answer RQ3, we manually analyzed all of the repairs returned by our method, to check how much change was made to the program. While the analysis was conducted for all benchmark programs, for brevity, we only show details in Table V for the larger real-world applications taken from the prior work [1]. To summarize, our method successfully repaired all of the 6 MPI bugs in *Memcached* [27], 1 MPI bug in *P-CLHT* [28], and 1 MPI bug in *Fast-Fair* [33]. Table V does not have *Clevel hashing* [32] and *CCEH* [34] because no bugs exist in them as shown by the last column of Table III.

In Table V, Columns 1-2 show the benchmark name and the bug type. Columns 3-4 show the source code locations of the read and write operations corresponding to each MPI bug. For example, `do_item_unlink_q_465` refers to Line 465 of the function named `do_item_unlink_q()`. Column 5 shows the time (in seconds) taken to repair each MPI violation. Columns 6-7 show the number of added `SFENCE`s and `Lock-UnLock` pairs.

It should not come as a surprise that only 0-1 `SFENCE` and 1-2 `Lock-UnLock` pairs were added in Step 2 of our method, because more `CLFLUSHOPT`s and `SFENCE`s had been added in Step 1. Furthermore, our method minimizes the number of `CLFLUSHOPT`s and `SFENCE`s added in Step 1 by first re-ordering the existing ones to repair the bug, before adding new ones.

TABLE V
INFORMATION OF THE REPAIRS COMPUTED FOR THE LARGER APPLICATIONS, WHERE ALL OF THE MPI BUGS WERE DETECTED AND REPORTED IN A PRIOR WORK [16].

| Name | Bug type | Read instruction / Write instruction | Step 2 time | sfence added | lk-unlk added |
|---|---|---|---|---|---|
| Memcached | MPI | do_item_unlink_q_465 / do_item_link_q_423 | 0.38 s | 1 | 2 |
| | MPI | do_slabs_free_550 / do_slabs_alloc_414 | 0.39 s | 1 | 2 |
| | MPI | do_item_update_624 / do_item_update_628 | 0.41 s | 1 | 2 |
| | MPI | _store_item_copy_data_2808 / do_add_delta_4295 | 0.48 s | 1 | 2 |
| | MPI | do_item_get_1097 / _store_item_copy_data_2825 | 0.65 s | 1 | 2 |
| | MPI | _store_item_copy_data_2808 / do_add_delta_4292 | 0.63 s | 0 | 1 |
| P-CLHT | MPI | clht_put_417 / ht_resize_pes_785 | 1.80 s | 1 | 2 |
| Fast-Fair | MPI | linear_search_876 / insert_key_560 | 41.24 s | 1 | 2 |

As mentioned earlier, our method creates critical sections by adding at most one lock-unlock pair for each shared PM variable. Our analysis of the computed repair shows that the added lock-unlock pairs do not substantially increase the size of the critical section. Specifically, the critical section contains only PM STOREs and their corresponding CLFLUSHOPTs and SFENCEs. This is important because smaller critical sections reduce the contention among concurrently running threads.

### *E. Discussion and Threats to Validity*

Our method currently depends on erroneous execution traces generated by external detectors such as PMRace [1]. While this limits completeness to observed behaviors, it is consistent with how practical PM bug repair tools operate. Importantly, our empirical results in Section VI show that the computed repairs generalize across unobserved executions and remain valid under multiple crash and interleaving scenarios. Another potential threat arises from hardware-specific persistency models; we mitigate this by encoding PM semantics symbolically in the solver, making the analysis independent of specific instruction reorderings. Our benchmark suite spans both micro-benchmarks and large systems (e.g., Memcached, P-CLHT), demonstrating robustness across diverse workloads. Finally, future work will integrate static reasoning to extend coverage to unobserved traces and alternative memory models, further improving completeness and portability.

## VII. RELATED WORK

To the best of our knowledge, there are two closely-related works on repairing PM bugs in existing software code, namely Hippocrates [15] and PMBugAssist [16]. Hippocrates [15] is limited to repairing durability violations and, since it is based on matching known bug patterns, it cannot handle previously unseen bug patterns. PMBugAssist [16] is limited to repairing durability and crash-consistency violations in a sequential computation. Neither of these two methods is able to repair inter-thread inconsistency bugs that are unique to concurrent programs. Besides the above two methods, DUDETX [30] is another method for identifying and fixing durability issues, but it is also limited to non-concurrent settings.

Our method take an erroneous execution trace of the PM program as input. In principle, the trace may be generated by any PM bug detection tool. While there are many such tools, the most closely-related one is PMRace [1], which can detect durability and crash-consistency violations as well as inter-thread inconsistency bugs. They are exactly the same set of PM bugs that can be handled by our repair method. In fact, PMRace is the bug detection tool that inspired our work on automated repair.

Besides PMRace, there are other trace-based PM bug detection methods and tools [6], [7], [8], [9], [10], [11], [12], [13], [14]. Earlier tools include Persistence Inspector [3] and PMREORDER [35], which are testing tools for PM programs originated from Intel for detecting simple PM bugs. PMEMCHECK [4], which is based on Valgrind [36], can also detect PM bugs. YAT [2] is a hypervisor-based framework for systematically testing PM-enabled applications. PMTEST [37] and PMDEBUGGER [38] recent tools that leverage user specified checking rules to detect PM bugs. More recently, Huang et al. [5] propose a method for automatically discovering and checking PM properties based on trace-based program analysis techniques and counter-factual reasoning.

Instead of repairing PM bugs in existing software code, PMROBUST [39] avoids PM bugs by leveraging a compiler to automatically insert flush and fence operations, to ensure that software code using persistent memory is free from missing flush and fence bugs. Its focus is primarily on performance optimizations using static analysis.

Beyond PM bugs, there is a large body of work on analyzing sequential and concurrency bugs in volatile memory based programs, some of which also rely on constraint solving to conduct trace-based analysis [20], [21], [18], [17], [19], [40], [41], [42]. While most of these methods are for bug detection, some of them focus on repair. For example, ConcBugAssist [24] employs bounded model checking and constraint-solving techniques to diagnose and repair concurrency bugs. BUGASSIST [22], [23] uses a SAT solver to repair assertion failures in a sequential program. EXTRACTFIX [43] uses an execution trace and a crash-free constraint as input to generate candidate patches that satisfy the constraint.

There are many methods for repairing concurrency bugs [44], [45], [46], [47], [48], [49], [50], [51], [52]. For example, Alglave et al. [47] use static analysis to insert synchronization operations, to enforce proper memory visibility in concurrent programs. PFIX [53] repairs concurrency bugs by utilizing memory access patterns, effectively targeting race conditions in multi-threaded programs. Meshman et al. [48] synthesize synchronization mechanisms to adhere to the C++ memory model, while Cerny et al. [49] enhance concurrency through transformations that maintain program semantics. However, none of these methods can repair PM bugs.

## VIII. Conclusion

We present the first symbolic analysis method for efficiently repairing *inter-thread inconsistency* bugs in concurrent PM programs. The method relies on an SMT solver based symbolic verification routine to precisely encode the set of possible thread interleavings and commit orders with respect to the PM. It computes the repair by repeatedly invoking the verification routine to detect erroneous commit orders and then blocking them by adding cache line flush and memory fence instructions as well as lock-unlock pairs. To reduce the computational cost, it relies on a novel two-step approach that separates the repair of DURA/MPB bugs within each thread from the repair of MPI bugs across multiple threads. Our experimental evaluation using a diverse set of benchmark programs shows that the method is effective in repairing all three types of most common PM bugs, including MPI bugs which are unique in concurrent persistent-memory programs.

## Data Availability

Our software artifacts are made publicly available at https://zenodo.org/records/14015317.

## Acknowledgments

This research was supported in part by the U.S. National Science Foundation (NSF) under grant CCF-2220345. We thank the anonymous reviewers for their constructive feedback.

## References

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

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

[3] Intel, "How to detect persistent memory programming errors - intel.com," https://www.intel.com/content/www/us/en/developer/articles/technical/detect-persistent-memory-programming-errors-with-intel-inspector-persistence-inspector.html, [Accessed 26-10-2024].

[4] Intel, "Discover Persistent Memory Programming Errors with Pmemcheck," https://www.intel.com/content/www/us/en/developer/articles/technical/discover-persistent-memory-programming-errors-with-pmemcheck.html, 2018, [Online; accessed 26-Oct-2024].

[5] Z. Huang, S. Ravi, and C. Wang, "Discovering likely program invariants for persistent memory," in *IEEE/ACM International Conference on Automated Software Engineering, ASE 2024*. ACM, 2024.

[6] H. Gorjiara, G. H. Xu, and B. Demsky, "Jaaru: efficiently model checking persistent memory programs," in *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*. ACM, 2021, pp. 415–428.

[7] B. Reidys and J. Huang, "Understanding and detecting deep memory persistency bugs in NVM programs with DeepMC," in *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*. ACM, 2022, pp. 322–336.

[8] H. Gorjiara, G. H. Xu, and B. Demsky, "Yashme: detecting persistency races," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. ACM, 2022, pp. 830–845.

[9] H. Gorjiara, W. Luo, A. Lee, G. H. Xu, and B. Demsky, "Checking robustness to weak persistency models," in *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. ACM, 2022, pp. 490–505.

[10] Z. Chen, Y. Hua, Y. Zhang, and L. Ding, "Efficiently detecting concurrency bugs in persistent memory programs," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. ACM, 2022, pp. 873–887.

[11] X. Fu, W. Kim, A. P. Shreepathi, M. Ismail, S. Wadkar, D. Lee, and C. Min, "Witcher: Systematic crash consistency testing for non-volatile memory key-value stores," in *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. ACM, 2021, pp. 100–115.

[12] X. Fu, D. Lee, and C. Min, "DURINN: Adversarial memory and thread interleaving for detecting durable linearizability bugs," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, Carlsbad, CA, Jul. 2022, pp. 195–211.

[13] S. Liu, S. Mahar, B. Ray, and S. M. Khan, "PMFuzz: test case generation for persistent memory programs," in *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*. ACM, 2021, pp. 487–502.

[14] S. Liu, K. Seemakhupt, Y. Wei, T. F. Wenisch, A. Kolli, and S. M. Khan, "Cross-failure bug detection in persistent memory programs," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. ACM, 2020, pp. 1187–1202.

[15] I. Neal, A. Quinn, and B. 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*, ser. ASPLOS '21. New York, NY, USA: ACM, 2021, p. 401–414.

[16] Z. Huang and C. Wang, "Constraint based program repair for persistent memory bugs," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 91:1–91:12.

[17] J. Huang, P. O. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. ACM, 2014, pp. 337–348. [Online]. Available: https://doi.org/10.1145/2594291.2594315

[18] J. Huang, C. Zhang, and J. Dolby, "CLAP: recording local executions to reproduce concurrency failures," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. ACM, 2013, pp. 141–152.

[19] M. Said, C. Wang, Z. Yang, and K. A. Sakallah, "Generating data race witnesses by an smt-based analysis," in *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, vol. 6617. Springer, 2011, pp. 313–327.

[20] C. Wang, S. Kundu, M. K. Ganai, and A. Gupta, "Symbolic predictive analysis for concurrent programs," in *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, ser. Lecture Notes in Computer Science, vol. 5850. Springer, 2009, pp. 256–272.

[21] C. Wang, R. Limaye, M. K. Ganai, and A. Gupta, "Trace-based symbolic analysis for atomicity violations," in *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, ser. Lecture Notes in Computer Science, vol. 6015. Springer, 2010, pp. 328–342.

[22] M. Jose and R. Majumdar, "Bug-Assist: Assisting fault localization in ANSI-C programs," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, 2011, pp. 504–509.

[23] M. Jose and R. Majumdar, "Cause clue clauses: error localization using maximum satisfiability," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, 2011, pp. 437–446.

[24] S. Khoshnood, M. Kusano, and C. Wang, "ConcBugAssist: constraint solving for diagnosis and repair of concurrency bugs," in *Proceedings*

*of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*. ACM, 2015, pp. 165–176. [Online]. Available: https://doi.org/10.1145/2771783.2771798

[25] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[26] L. De Moura and N. Bjørner, "Z3: an efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 337–340.

[27] Lenovo, "Memcached-pmem," https://github.com/lenovo/memcached-pmem, 2018.

[28] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "RECIPE: Converting concurrent DRAM indexes to persistent-memory indexes," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, Ontario, Canada, October 2019.

[29] A. Raad, J. Wickerson, G. Neiger, and V. Vafeiadis, "Persistency semantics of the intel-x86 architecture," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 11:1–11:31, 2020.

[30] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "DudeTx: Durable transactions made decoupled," *ACM Trans. Storage*, vol. 14, no. 1, Apr. 2018. [Online]. Available: https://doi.org/10.1145/3177920

[31] Intel, "Persistent memory development kit (PMDK)," https://https://pmem.io/pmdk/, 2022.

[32] Z. Chen, Y. Hua, B. Ding, and P. Zuo, "Lock-free concurrent level hashing for persistent memory," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 799–812. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/chen

[33] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in Byte-Addressable persistent B+-Tree," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA: USENIX Association, Feb. 2018, pp. 187–200. [Online]. Available: https://www.usenix.org/conference/fast18/presentation/hwang

[34] M. Nam, H. Cha, Y. ri Choi, S. H. Noh, and B. Nam, "Write-Optimized dynamic hashing for persistent memory," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 31–44. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/nam

[35] Intel, "PMREORDER - performs a persistent consistency check using a store reordering mechanism," 2022. [Online]. Available: https://pmem.io/pmdk/manpages/linux/master/pmreorder/pmreorder.1/

[36] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, p. 89–100, Jun. 2007. [Online]. Available: https://doi.org/10.1145/1273442.1250746

[37] S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. M. 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 2019, Providence, RI, USA, April 13-17, 2019*. ACM, 2019, pp. 411–425.

[38] B. Di, J. Liu, H. Chen, and D. Li, "Fast, flexible, and comprehensive bug detection for persistent memory programs," in *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*. ACM, 2021, pp. 503–516.

[39] Y. Guo, W. Luo, and B. Demsky, "Automated insertion of flushes and fences for persistency," in *IEEE/ACM International Conference on Automated Software Engineering*, 2025.

[40] A. Sinha, S. Malik, C. Wang, and A. Gupta, "Predictive analysis for detecting serializability violations through trace segmentation," in *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Cambridge, UK, 11-13 July, 2011*. IEEE, 2011, pp. 99–108.

[41] A. Sinha and S. Malik and C. Wang and A. Gupta, "Predicting serializability violations: SMT-based search vs. DPOR-based search," in *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 7261. Springer, 2011, pp. 95–114. [Online]. Available: https://doi.org/10.1007/978-3-642-34188-5_11

[42] Z. Huang and C. Wang, "Symbolic predictive cache analysis for out-of-order execution," in *Fundamental Approaches to Software Engineering - 25th International Conference, FASE 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, ser. Lecture Notes in Computer Science, E. B. Johnsen and M. Wimmer, Eds., vol. 13241. Springer, 2022, pp. 163–183. [Online]. Available: https://doi.org/10.1007/978-3-030-99429-7_10

[43] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury, "Beyond tests: Program vulnerability repair via crash constraint extraction," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, pp. 14:1–14:27, 2021.

[44] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: program repair via semantic analysis," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. IEEE Computer Society, 2013, pp. 772–781.

[45] S. Mechtaev, J. Yi, and A. Roychoudhury, "DirectFix: Looking for simple program repairs," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. IEEE Computer Society, 2015, pp. 448–458.

[46] M. Z. Malik, K. Ghori, B. Elkarablieh, and S. Khurshid, "A case for automated debugging using data structure repair," in *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*. IEEE Computer Society, 2009, pp. 620–624.

[47] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl, "Don't sit on the fence: A static analysis approach to automatic fence insertion," *ACM Trans. Program. Lang. Syst.*, vol. 39, no. 2, pp. 6:1–6:38, 2017. [Online]. Available: https://doi.org/10.1145/2994593

[48] Y. Meshman, N. Rinetzky, and E. Yahav, "Pattern-based synthesis of synchronization for the C++ memory model," in *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*. IEEE, 2015, pp. 120–127. [Online]. Available: https://doi.org/10.1109/FMCAD.2015.7542261

[49] P. Cerný, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach, "Efficient synthesis for concurrency by semantics-preserving transformations," in *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, vol. 8044. Springer, 2013, pp. 951–967. [Online]. Available: https://doi.org/10.1007/978-3-642-39799-8_68

[50] V. Kahlon and C. Wang, "Lock removal for concurrent trace programs," in *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, ser. Lecture Notes in Computer Science, vol. 7358. Springer, 2012, pp. 227–242.

[51] Y. Cai, L. Cao, and J. Zhao, "Adaptively generating high quality fixes for atomicity violations," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. ACM, 2017, pp. 303–314. [Online]. Available: https://doi.org/10.1145/3106237.3106239

[52] Y. Cai and L. Cao, "Fixing deadlocks via lock pre-acquisitions," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. ACM, 2016, pp. 1109–1120. [Online]. Available: https://doi.org/10.1145/2884781.2884819

[53] H. Lin, Z. Wang, S. Liu, J. Sun, D. Zhang, and G. Wei, "PFix: fixing concurrency bugs based on memory access patterns," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. ACM, 2018, pp. 589–600. [Online]. Available: https://doi.org/10.1145/3238147.3238198