Constraint Based Program Repair for Persistent Memory Bugs

Zunchen Huang
University of Southern California
Los Angeles, USA

Chao Wang
University of Southern California
Los Angeles, USA

ABSTRACT

We propose a constraint based method for repairing bugs associated with the use of *persistent memory (PM)* in application software. Our method takes a program execution trace and the violated property as input and returns a suggested repair, which is a combination of inserting new PM instructions and reordering these instructions to eliminate the property violation. Compared with the state-of-the-art approach, our method has three advantages. First, it can repair both *durability* and *crash consistency* bugs whereas the state-of-the-art approach can only repair the relatively-simple *durability* bugs. Second, our method can discover new repair strategies instead of relying on repair strategies hard-coded into the repair tool. Third, our method uses a novel symbolic encoding to model PM semantics, which allows our symbolic analysis to be more efficient than the explicit enumeration of possible scenarios and thus explore a large number of repairs quickly. We have evaluated our method on benchmark programs from the well-known Intel PMDK library as well as real applications such as Memcached, Recipe, and Redis. The results show that our method can repair all of the 41 known bugs in these benchmarks, while the state-of-the-art approach cannot repair any of the *crash consistency* bugs.

ACM Reference Format:
Zunchen Huang and Chao Wang. 2024. Constraint Based Program Repair for Persistent Memory Bugs. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639204

1 INTRODUCTION

Persistent memory (PM) is a type of non-volatile random-access memory with the capability of retaining data after the loss of electrical power. It has become commercially viable in the past few years. In modern computer architecture, PM may serve as the intermediate layer between volatile DRAM and non-volatile storage such as solid-state disks or replace part of the DRAM-based main memory. This will lead to a drastic reduction in latency and power consumption of the computing systems, and an increase in robustness against frequent and unpredictable power interruptions. This is why PM is used in more and more applications as commercial PM devices [20] come close to DRAM in terms of speed but with a significantly larger capacity. However, software developers are required to write PM related software code in order to unleash the full power of these PM devices [44].

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04
https://doi.org/10.1145/3597503.3639204

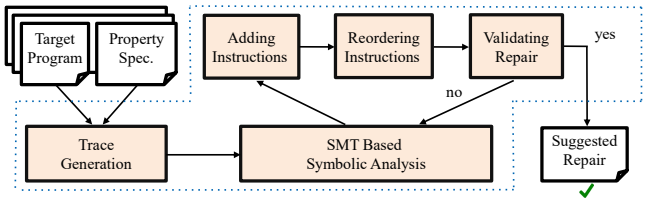

Figure 1: PMBugAssist: the proposed PM bug repair method.

Unfortunately, it is a challenging task to use PM instructions and APIs correctly and efficiently. The reason is because, due to performance concerns, PM instructions are often designed to have weaker persistency/consistency models than volatile memory instructions. Thus, what is considered as a correct behavior for volatile memory may no longer be correct for persistent memory. Since the persistency/consistency models are far from being intuitive, unless developers have a deep understanding of both software and the PM semantics associated with hardware, it will be difficult to use these PM instructions and APIs correctly and efficiently.

Although a large number of program analysis techniques have been developed to help detect PM bugs [6, 8–10, 14, 28, 31–33, 38, 42] or prove their absence [13, 27, 40], little has been done on automated diagnosis and repair of PM bugs. In fact, the only existing repair technique that we are aware of is the Hippocrates tool developed by Neal et al. [37]. Unfortunately, Hippocrates only repairs one type of relatively simple PM bugs called *durability* bugs; these bugs are simple in that fixing them requires only the addition of missing PM instructions. There are more complex PM bugs, often called *crash consistency* bugs in the literature, that Hippocrates cannot repair; fixing them requires some of the existing instructions to be reordered. Furthermore, Hippocrates uses *syntactic-level* pattern-matching, which means if a bug matches a known pattern, the tool will be able to repair it by applying a pre-defined code transformation. However, if the bug does not match any known pattern hard-coded into the repair tool, the bug cannot be repaired.

To fill the gap, we propose a constraint based method for automatically computing repairs for a broader class of PM bugs. Unlike the syntactic-level pattern-matching based approach of Neal et al. [37], our method relies on a semantical analysis of PM instructions to compute repairs. By symbolically encoding the PM-related program behavior and the correctness property as a set of logical constraints, and then leveraging an off-the-shelf SMT solver to solve these constraints, our method is able to search for novel repair strategies in a large solution space. As a result, our method is able to repair *durability* and *crash consistency* bugs of arbitrary form, even if these bugs do not match any of the known syntactic-level bug patterns hard-coded into Hippocrates.

Fig. 1 shows an overview of our method. The input consists of the program and the violated PM property, and the output is the suggested repair. Internally, our method first leverages a Valgrind based software tool to instrument the program and generate the execution trace. The traces generated at the end of this step may be fed to any existing PM bug detection tool [18, 19, 22] to confirm the property violation. To compute a repair, our method uses an SMT solver to symbolically encode the solution space. As shown in Fig. 1, it symbolically checks possible repairs in the solution space to find a valid repair. In this context, a repair can be thought of as a modification of the program through a combination of inserting new PM instructions and reordering PM instructions. Our search for a repair is an iterative process, involving multiple calls to the SMT solver for both finding the repair candidate and validating it. Only valid repairs are returned to the user.

At the center of our method is the SMT solver based symbolic analysis for two reasons. First, symbolic analysis allows us to explore a large number of possible solutions quickly. Second, symbolic analysis is able to model various types of PM instructions and properties not only accurately but also uniformly, meaning that during symbolic encoding, everything boils down to a set of logical constraints. Since these constraints are expressed in a fragment of the SMT-LIB format, i.e., linear integer arithmetic (LIA), they can be solved efficiently using any off-the-shelf SMT solver.

We have implemented the method in a tool named PMBugAssist. During experimental evaluation, we focused on comparing our method with Hippocrates [37]. This is because our focus is on automated repair, for which Hippocrates represents the state of the art. In contrast, prior work on detecting PM bugs [8, 27, 31, 33, 40] and verifying their absence [13, 27, 40] is less relevant; instead, they are complementary to our method.

Our benchmarks include programs from the well-known Intel PMDK library [21] as well as real applications such as Memcached [4], Recipe [30] and Redis [3]. According to prior works on PM bug detection, these benchmarks have 41 known bugs in total, including 23 *durability* bugs and 18 *crash consistency* bugs. Our experimental results show that the new method can repair all of these 41 bugs, whereas Hippocrates cannot repair any of the *crash consistency* bugs. We also evaluated the runtime performance of the new method, and found that, for all benchmark programs, it can finish the repair computation quickly.

To summarize, we make the following contributions:

- We propose the first constraint based method for repairing a broader class of PM bugs. Compared with the state-of-the-art approach, our method can repair PM bugs that do not match any known bug pattern.
- We formalize PM bug repair as a special case of the *syntax-guided synthesis (SyGuS)* [2] problem, through which we discuss the soundness and decidability of our method.
- We implement and evaluate the method on a large number of benchmark programs to demonstrate its advantages over state-of-the-art (Hippocrates).

The remainder of this paper is structured as follows. In Section 2, we review the technical background. In Section 3, we present the top-level procedure of our method. This is followed by our SMT solver based symbolic analysis in Section 4, our repair algorithm in Section 5, and discussion of correctness and optimizations in Section 6. We present the experimental results in Section 7 and review related work in Section 8. Finally, we give our conclusions in Section 9.

Table 1: Persistency order of *Px86* for instructions ($I_i < I_j$).

Instruction Pair (I_i, I_j)		Second Instr. I_j						
		LOAD	STORE	RMW	mfence	sfence	clflushopt	clflush
First Instr. I_i	LOAD	✔	✔	✔	✔	✔	✔	✔
	STORE	✘	✔	✔	✔	✔	CL	✔
	RMW	✔	✔	✔	✔	✔	✔	✔
	mfence	✔	✔	✔	✔	✔	✔	✔
	sfence	✘	✔	✔	✔	✔	✔	✔
	clflushopt	✘	✘	✔	✔	✔	✘	CL
	clflush	✘	✔	✔	✔	✔	CL	✔

2 BACKGROUND

2.1 Persistent Memory (PM) Semantics

We focus on Intel's *persistent x86 (Px86)* model as published by Raad et al. [40]. In the standard *x86* architecture, STORE instructions executed by the CPU are sequentialized in a *store buffer* before taking effect in memory, while LOAD instructions are allowed to take effect immediately. This allows a fast LOAD to take effect before a slow STORE, provided that they have no control/data dependency, while preserving the semantic equivalence of the program.

In the *Px86* architecture, a *persistent buffer* is added after the *store buffer* to further sequentialize the STORE instructions, before the written values show up in persistent media. While the CPU still preserves the sequential program behavior during normal (crash-free) execution, when a program crashes due to power failure, the order in which the written values show up in persistent media may be significantly different. This may lead to PM bugs.

2.1.1 The Persistency Table. Table 1, which is taken from Raad et al. [40], characterizes an important aspect of *Px86* that is relevant to our work: the order in which instructions take effect in persistent memory. Given a pair of instructions, (I_i, I_j), where I_i is executed before I_j by the CPU, the corresponding table entry shows whether *Px86* guarantees that I_i persists before I_j using the symbols ✔ (yes) and ✘ (no). The third symbol, **CL**, means that I_i persists before I_j only when the two instructions access memory address blocks mapped to the same cache line.

For example, `(STORE x, LOAD y)` may persist in reverse order according to the ✘ symbol in Table 1 when the CPU chooses to execute the fast `LOAD y` before the slow `STORE x` for performance reasons. However, according to the table, `(LOAD y, STORE x)` must persist in the same order as they appear in the program, due to a possible control/data dependency. That is, since these two instructions may come from either the code snippet `if(y>0) {x=1;}` (with control dependency) or the code snippet `{reg=y; x=1;}` (without dependency), to be safe, the CPU would have to disallow the reordering based optimization.

2.1.2 PM-related Instructions. In this work, we are concerned with the following PM-related instructions besides `LOAD`, `STORE`, and `RMW` (read-modify-write) instructions.

- `clflush`, which stands for *cache-line-flush*, is a synchronous operation of the CPU that results in flushing the cache line

associated with addr immediately. Since this legacy instruction is blocking and slow, it is rarely used.
- clflushopt, which stands for *cache-line-flush-optimized*, is an asynchronous operation that may postpone flushing to a convenient future time. It is fast, but the exact persistency time is less predictable.
- mfence, which stands for *memory-fence*, is a memory barrier for both STORE and LOAD instructions.
- sfence, which stands for *store-fence*, is a memory barrier for STORE instructions only.
- Following Raad et al. [40], we treat clwb (cache-line-write-back) the same as clflushopt since the two instructions are semantically equivalent.

While the legacy instruction CLFLUSH is semantically equivalent to CLFLUSHOPT followed by SFENCE or MFENCE or RMW according to Intel's user manual, in terms of performance, the fastest and most-frequently-used combination is CLFLUSHOPT followed by SFENCE. Thus, we focus on this combination in this paper.

2.2 Persistent Memory (PM) Bugs

We are concerned with two common types of PM bugs, called *durability* bugs and *crash consistency* bugs in the literature, which can be generated by many existing PM bug detection tools such as PMemCheck [19] and PMTest [33].

2.2.1 Durability Bugs. Here, durability means that a value written by STORE eventually shows up in persistent media. However, this is not automatically guaranteed. Fig. 2 shows an example code snippet adapted from Intel's website, where the value written to header->counter may never show up in persistent media. This is because the program does not force the CPU to flush the corresponding cache line and, as a result, the written value (temporarily stored in the volatile part of the CPU) may be lost permanently if a power failure occurs while writer() is executed. After crash recovery, reader() may not have access to the values written by writer(), for example, due to the incorrect value of header->counter.

To make STORE instructions durable, __mm_clflushopt() and __mm_sfence() must be used to force the CPU to flush the cache line; these API calls correspond to CLFLUSHOPT and SFENCE. This is how values written to the name and addr fields of records[i] are made durable in Fig. 2 (Lines 12-14 and 20 for the THEN-branch, and Lines 18 and 20 for the ELSE-branch).

Note that neither instruction in the CLFLUSHOPT+SFENCE combination may be omitted; otherwise, durability is not guaranteed.

2.2.2 Crash Consistency Bugs. When a program crashes due to power failure, it is possible that some (but not all) of the written values have been stored in persistent media. To prevent the persistent media from entering an inconsistent state, the program must use CLFLUSHOPT+SFENCE correctly, to force the STORE instructions to take effect in a certain order. The *persistency order*, in general, is determined by the reader() executed during crash recovery.

The reader() in Fig. 2 uses header->counter to decide whether to read records[i], and then uses the value of records[i].valid to decide whether to read records[i].name and records[i].addr. Thus, the correct persistency order, which must be enforced by writer(), is that both records[i].name and records[i].addr

```
1  // both 'header' and 'records' are mapped to the persistent memory
2  struct record_t { char name[64],addr[64]; char valid; } records[32];
3  struct header_t { uint32_t counter;  uint8_t reserved[63]; } header;
4  ...
5  // writer() -- code executed before crash
6  for (int i=0; i<NUM_RECORDS; i++) {
7    header->counter++;
8    if (rand()%2==0) {  //store a valid record
9      snprintf( records[i].name, 64, ... );
10     snprintf( records[i].addr, 64, ... );
11     records[i].valid = 1;
12     __mm_clflushopt( &records[i].valid );
13     __mm_clflushopt( records[i].name );
14     __mm_clflushopt( records[i].addr );
15   }
16   else {
17     records[i].valid = 0;
18     __mm_clflushopt( &records[i].valid );
19   }
20   __mm_sfence();
21 }
22 ...
23 // reader() -- code executed after crash
24 for (int i=0; i<header->counter; i++) {
25   if (records[i].valid==1) {
26     cout << "name ="  << records[i].name << "\n";
27     cout << "addr ="  << records[i].addr << "\n";
28   }
29 }
```

Figure 2: An example program with several PM bugs.

```
1  for (int i=0; i<NUM_RECORDS; i++) {
2    if (rand()%2==0) {  //store a valid record
3      snprintf( records[i].name, 64, ... );
4      snprintf( records[i].addr, 64, ... );
5      __mm_clflushopt( records[i].name );
6      __mm_clflushopt( records[i].addr );
7      __mm_sfence();
8      records[i].valid = 1;
9      __mm_clflushopt( &records[i].valid );
10   }
11   else {
12     records[i].valid = 0;
13     __mm_clflushopt( &records[i].valid );
14   }
15   __mm_sfence();
16   header->counter++;
17   __mm_clflushopt( &header->counter );
18   __mm_sfence();
19 }
```

Figure 3: The repaired writer() in the example program.

persist before records[i].valid, and records[i].valid persists before header->counter.

In existing bug detection tools, such as PMemCheck [19] and PMTest [33], the durability and *must-persist-before* properties are typically specified by the user and then checked for violations automatically. Such tools would be able to detect property violations in Fig. 2. For header->counter, the written value is not made durable at all using CLFLUSHOPT+SFENCE. As for records[i], there is a property violation since the reader() may read value 1 for records[i].valid from persistent media, and then expect records[i].name and records[i].addr to be available in persistent media, but end up with uninitialized or partially initialized values.

2.3 Detecting PM Bugs

Existing tools for detecting PM bugs (e.g., PMemCheck [19] and PMTest [33]) are based on analyzing the execution traces. Fig. 4

```
// trace for executing the THEN-branch
Inst I1: STORE       0x4a3c000   //STORE records[i].name
Inst I2: STORE       0x4a3c080   //STORE records[i].valid
Inst I3: clflushopt  0x4a3c080   //clflushopt records[i].valid
Inst I4: clflushopt  0x4a3c000   //clflushopt records[i].name
Inst I5: sfence
assert( PTime(I1) < PTime(I2) ) //crash-consistency bug
```

```
// trace for executing the ELSE-branch
Inst I1: STORE       0x4a3c0C0   //STORE header->counter
Inst I2: STORE       0x4a3c080   //STORE records[i].valid
Inst I3: clflushopt  0x4a3c080   //clflushopt records[i].valid
Inst I4: sfence
assert( PTime(I1) < TMAX )      //durability bug
assert( PTime(I2) < PTime(I1) ) //crash-consistency bug
```

Figure 4: Execution traces of the program in Fig. 2, with a *durability* bug in ELSE-branch (write to `header->counter` may never show up in PM) and a *crash consistency* bug in THEN-branch (write to `records[i].name` may not persist before write to `records[i].valid`).

shows two example traces for branches of the loop body in Fig. 2. For simplicity, we only show the STORE, CLFLUSHOPT, and SFENCE instructions relevant to the violated property assertions.

The first assertion violated by the ELSE-branch represents a durability property. Assume that all the STOREs in Fig. 2 are expected to persist in PM media. For the STORE I_1, its persistency time is denoted `PTime(`I_1`)`. Assuming that `TMAX` is the upper bound of the persistency time (bounded by the number of executed instructions in this program), we can express durability as `PTime(`I_1`)<TMAX`. The assertion is violated because `clflushopt 0x4a3c0C0` is not used to force the CPU to flush the written value from cache to persistent media.

The assertion violated by the THEN-branch captures a crash consistency property. Here, the expectation is that the value written by I_1 always persists before the value written by I_2, as shown in `PTime(`I_1`)<PTime(`I_2`)`. The assertion is violated because the CPU allows two CLFLUSHOPT instructions to take effect in reverse order, as shown by the ✘ symbol in Table 1.

Note that, even if we swap the execution order of the two instructions (I_3 and I_4) in the program, the assertion will still be violated. Fig. 5 illustrates the reason. Here, the solid edges represent the execution order, while the dashed edges represent the persistency order imposed by *Px86*. Since the dashed edges remain the same (before and after swapping the execution order of I_3 and I_4), the requirement that I_1 always persists before I_2 is still not satisfied.

2.4 Repairing PM Bugs

Hippocrates [37] is the only existing method for repairing PM bugs, with two limitations. First, it only repairs the relatively simple *durability* bugs, such as the one shown in the ELSE-branch of Fig. 4, but not the more complex *crash consistency* bugs. Second, it only repairs bugs that syntactically match the patterns hard-coded into the repair tool. For bugs that do not have a syntactical match, Hippocrates would not know how to repair them. For example, if repairing a bug requires reordering some instructions, then Hippocrates cannot do it.

In contrast, our method can repair both *durability* and *crash consistency* bugs, and can repair bugs that do not syntactically match any of the known patterns hard-coded into Hippocrates. This is because our method has the ability to analyze the semantics of the PM instructions, and thus repair PM bugs through a combination of inserting new PM instructions and reordering instructions. We illustrate the technical challenges using examples in Fig. 6.

Fig. 6 shows two possible repairs of the bug in the THEN-branch of Fig. 4. The first attempt, based solely on reordering the existing instructions of the execution trace, is not a complete repair. The reason is because, by moving I_4 and I_5 before I_2 and I_3, the new version of the program guarantees that `records[i].name` persists before `records[i].valid`. However, reordering also introduces a new durability bug for I_2: without a subsequent SFENCE instruction, the value written by I_2 is no longer guaranteed to show up in persistent media, e.g., if the program crashes in the middle of the execution due to power failure.

Fig. 6 highlights the fact that, sometimes, it is impossible to repair a *crash consistency* bug solely by reordering instructions; we also need to add new PM instructions. We shall explain in the remainder of this paper how our method finds out that, by adding SFENCE in I_6, we can completely repair the *crash consistency* bug.

To summarize, for the buggy `writer()` in Fig. 2, the repaired version is shown in Fig. 3. Through a combination of inserting new PM instructions and reordering instructions, the repaired version in Fig. 3 guarantees both the durability of `header->counter` and the crash consistency requirement that `records[i].valid` always persists before `header->counter`. Note that, to satisfy the second requirement, we not only have to add CLFLUSHOPT+SFENCE for `header->counter`, but also have to move `header->counter++` (Line 7 in Fig. 2) after the IF-ELSE statement (Line 16 in Fig. 3).

3 OVERVIEW OF OUR METHOD

Our method takes an existing PM bug as input. Besides the PM bug, which is an execution trace $\mathcal{T}$ that violates a property assertion $\mathcal{A}$, no other input or constraint needs to be provided by the user. The PM bug may be produced by any existing bug detection tools such as PMemCheck [19] and PMTest [33]. Specifically, the trace $\mathcal{T} = \{I_1, \ldots, I_N\}$ is a sequence of instructions, each of which has an instruction type specified in Table 1.

The assertion $\mathcal{A}$ may be of the form $PT(I_i) < TMAX$ (durability) or $PT(I_i) < PT(I_j)$ (crash consistency) as shown in Fig. 4. Here, $TMAX$ is the upper bound of the persistency time. Thus, if there exists a way of satisfying $PT(I_i) \geq TMAX$, there exists a durability violation where I_i has not yet taken effect in persistent media at the end of the execution.

Algorithm 1: Our method $\mathcal{R} \leftarrow$ PMBugAssist($\mathcal{T}, \mathcal{A}$)

```
while BugIsFound(T, A) do
    R ← ComputeRepair(T, A)
    if RepairIsValid(T, R) then
        return R as repair
    end if
    T ← AddInstructions(T, A, R)
end while
```

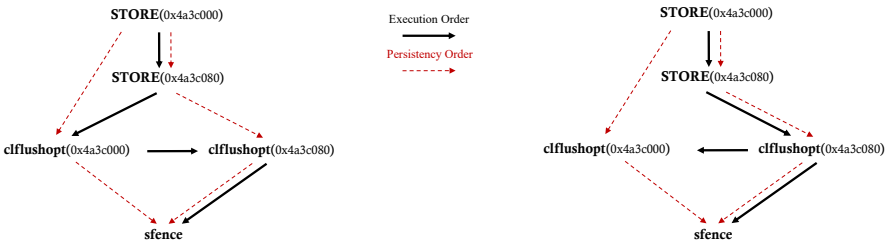

Figure 5: Ordering constraints for THEN-branch: modifying the program by swapping the two `clflushopt` instructions will not fix the crash consistency bug.

```
// trace for executing the THEN-branch
Inst I1: STORE       0x4a3c000   //STORE records[i].name
Inst I4: clflushopt  0x4a3c000   //clflushopt records[i].name
Inst I5: sfence                  //sfence
Inst I2: STORE       0x4a3c080   //STORE records[i].valid
Inst I3: clflushopt  0x4a3c080   //clflushopt records[i].valid
//This is an incomplete repair -- it introduces a new durability bug

Inst I6: sfence                  //fence
//This is a complete repair -- must also add this 'sfence'
```

Figure 6: Two repairs for the bug in THEN-branch of Fig. 4: The first repair is incomplete since it adds a new *durability* bug for I_2; the second repair is complete because it removes the new *durability* and original *crash consistency* bugs.

Algorithm 1 shows the top-level procedure. Since we only invoke the procedure on a buggy execution trace, the first call to the subroutine BugIsFound$(\mathcal{A}, \mathcal{T})$ always returns *true*. Next, we use ComputeRepair$(\mathcal{A}, \mathcal{T})$ to compute a potential repair. It guarantees that, after applying the repair $\mathcal{R}$ to the given trace $\mathcal{T}$, the assertion violation no longer exists. However, this is not yet enough to guarantee that $\mathcal{R}$ is a valid repair.

There are two possibilities. One possibility is that $\mathcal{R}$ indeed is a *valid* repair: by permuting the instructions in $\mathcal{T}$, $\mathcal{R}$ removes all the bad executions and retains only the good executions. The other possibility is that $\mathcal{R}$ is a *vacuous* repair in that, by creating a contradiction between $\mathcal{R}$ and $\mathcal{T}$, it artificially removes all valid executions of the instructions in $\mathcal{T}$. Since there is no longer any valid execution, by definition, the SMT solver cannot detect any violation (which must be a valid, and yet buggy, execution).

To find out whether the repair $\mathcal{R}$ is valid or vacuous, we use the subroutine RepairIsValid$(\mathcal{A}, \mathcal{R})$ to check, after applying $\mathcal{R}$ to $\mathcal{T}$, whether any valid execution exists. If the answer is yes, then $\mathcal{R}$ is a valid repair, and thus is returned to the user. Otherwise, we use AddInstructions$(\mathcal{A}, \mathcal{T}, \mathcal{R})$ to add more SFENCE and CLFLUSHOPT instructions to $\mathcal{T}$, and try again.

There is a distinction between the normal program behavior and PM-related behavior, only the latter of which can be affected by CLFLUSHOPT/SFENCE instructions. Since our method only inserts and reorders CLFLUSHOPT/SFENCE instructions, it will not change the normal program behavior. As for the PM-related behavior, due to the use of the verification subroutine BugIsFound$(\mathcal{T}, \mathcal{A})$ in Line 1 of Algorithm 1, our method guarantees to eliminate the violation of the property assertion $\mathcal{A}$ in the given trace $\mathcal{T}$.

Our method explicitly considers the efficiency of the computed repair by adding SFENCE and CLFLUSHOPT instructions iteratively on a "need-to" basis. As soon as enough instructions are added, the while-loop in Algorithm 1 will terminate. In this sense, it minimizes the number of added instructions, but without using an "optimizing solver" such as MAXSMT in DirectFix [35].

4 SYMBOLIC ANALYSIS OF THE PM BUG

In this section, we present our SMT based method for analyzing the PM bug symbolically. It is the foundation of not only the subroutine BugIsFound$(\mathcal{T}, \mathcal{A})$ but also the subroutines ComputeRepair$(\mathcal{T}, \mathcal{A})$ and RepairIsValid$(\mathcal{T}, \mathcal{R})$ in Algorithm 1.

4.1 The Satisfiability Problem

Given the trace $\mathcal{T}$ and the assertion $\mathcal{A}$, whether there exists a valid execution of the instructions in $\mathcal{T}$ that violates $\mathcal{A}$ can be formulated as a satisfiability (SAT) problem. Toward this end, we construct a logical formula $\Phi := \Phi_{program} \wedge \Phi_{persistency} \wedge \neg\Phi_{assertion}$, where $\Phi_{program}$ encodes the program order, $\Phi_{persistency}$ encodes the persistency order, and $\Phi_{assertion}$ encodes the assertion. Thus, Φ is satisfiable if and only if there exists a valid execution of the instructions in $\mathcal{T}$ that violates $\mathcal{A}$.

We express Φ in a fragment of the SMT-LIB format that allows only integer variables (such as x and y) and Boolean compositions of linear integer arithmetic (LIA) constraints of the form $(x < y)$. Thus, the satisfiability of Φ can be efficiently decided using any off-the-shelf SMT solver.

Before presenting our method for constructing Φ, we define the two sets of variables used to encode Φ as follows:

The PC_I_i Variables. For each instruction $I_i \in \mathcal{T}$, where $i = 1, \ldots, N$, we define a variable *PC_I_i* whose value may be any integer in the interval $[0, N)$; it stands for the execution time, i.e., when the instruction I_i is executed by the CPU. Inside Φ, we will constrain *PC_I_i* variables to allow only valid permutations of $\mathcal{T}$.

The PT_I_i Variables. For each instruction $I_i \in \mathcal{T}$ of the STORE type, we define a variable *PT_I_i* whose value may be any integer in the interval $[0, N + 1]$; it stands for the persistency time of I_i, i.e., when the value written by I_i is actually stored in persistent media.

$$
\begin{aligned}
\Phi_{pc} &:= \bigwedge_{1\le i\le N}(0 \le PC\_I_i < N) \wedge \bigwedge_{1\le i<j\le N}(PC\_I_i \neq PC\_I_j)\\
\Phi_{so} &:= \bigwedge_{I_i\in Stores \wedge I_j\in Stores \wedge i<j}(PC\_I_i < PC\_I_j)\\
\Phi_{fs} &:= \bigwedge_{I_j\in Flushes} \bigvee_{I_i\in Stores \wedge SameCacheL(I_i,I_j)}(PC\_I_i < PC\_I_j)\\
\Phi_{fo} &:= \bigwedge_{I_i\in Fences \wedge I_j\in Fences \wedge i<j}(PC\_I_i < PC\_I_j)\\
\Phi_{mo} &:= \bigwedge_{I_i,I_j\in Stores \wedge SameCacheL(I_i,I_j) \wedge i<j} \bigvee_{I_k\in Flushes \wedge SameCacheL(I_k,I_i)} (PC\_I_i < PC\_I_k < PC\_I_j)\\
\Phi_{pti} &:= \bigwedge_{I_i\in Stores}(-1 \le PT\_I_i \le N+1)\\
\Phi_{pts} &:= \bigwedge_{I_i\in Stores}(PC\_I_i \le PT\_I_i)\\
\Phi_{fi} &:= \bigwedge_{I_i\in Stores \wedge I_j\in Flushes \wedge SameCacheL(I_i,I_j) \wedge I_k\in Fences} (PC\_I_i < PC\_I_j < PC\_I_k) \implies (PT\_I_i \le PC\_I_k)\\
\Phi_{du} &:= \bigwedge_{I_i\in Stores}(PT\_I_i < N)\\
\Phi_{cc} &:= \bigwedge_{I_i,I_j\in Stores \wedge assert(PTime(I_i)<PTime(I_j))}(PT\_I_i < PT\_I_j)
\end{aligned}
$$

Figure 7: Our symbolic encoding of the subformulas in $\Phi_{program} := \Phi_{pc} \wedge \Phi_{so} \wedge \Phi_{fs} \wedge \Phi_{fo} \wedge \Phi_{mo}$, $\Phi_{persistency} := \Phi_{pti} \wedge \Phi_{pts} \wedge \Phi_{fi}$ and $\Phi_{assertion} := \Phi_{du} \wedge \Phi_{cc}$.

4.2 Using $\Phi_{program}$ to Encode Execution Order

Let $\Phi_{program} := \Phi_{pc} \wedge \Phi_{so} \wedge \Phi_{fo} \wedge \Phi_{fs} \wedge \Phi_{mo}$ be a set of constraints on PC_I_i variables such that, for every satisfying assignment to $\Phi_{program}$, the values of PC_I_i variables correspond to a valid permutation of $\mathcal{T}$.

4.2.1 Subformula Φ_{pc}. This *program-counter (pc)* constraint restricts each PC_I_i to $[0, N)$ to model the time when I_i is executed. The execution time starts from 0 and is bounded by N, the total number of instructions in $\mathcal{T}$. We also require each PC_I_i variable to have a unique value. The definition of Φ_{pc} is presented in Fig. 7.

4.2.2 Subformula Φ_{so}. This *store-order (so)* constraint requires the STORE instructions in $\mathcal{T}$ to execute in the same order as they appear in the trace. This is because *Px86* has a single *store-buffer* for all STORE instructions; thus, reordering of two STORE instructions (I_i, I_j) is not allowed, as shown by ✔ in Table 1. The definition of Φ_{so} is also presented in Fig. 7.

While computing the repair, we may choose to relax Φ_{so} in certain cases, to allow some of the STORE instructions to reorder. This is because some PM bugs cannot be repaired unless some STORE instructions are allowed to reorder in the program. We discussed an example at the end of Section 2, and we will discuss details of this relaxation in Section 6.

4.2.3 Subformula Φ_{fs}. This *flush-store (fs)* constraint requires that, for each CLFLUSHOPT (I_j), its execution time must be after at least one of the STORE (I_i) that it can flush. This requires that I_i and I_j are mapped to the same cache line, i.e., $SameCacheL(I_i, I_j)$ holds.

4.2.4 Subformula Φ_{fo}. This *fence-order (fo)* constraint requires multiple SFENCE instructions to be executed in the same order as they appear in the trace.

4.2.5 Subformula Φ_{mo}. This *memory overwrite (mo)* constraint says that two STORE instructions (I_i, I_j) cannot write the same address without a CLFLUSHOPT (I_k) inserted in between, to avoid memory overwrite. Memory overwrites must be avoided because, by definition, it violates the durability property.

```
1  //Program order constraints:
2  (0 ≤ PC_I1 ≤ 5) ∧ (0 ≤ PC_I2 ≤ 5) ∧ (0 ≤ PC_I3 ≤ 5) ∧ (0 ≤ PC_I4 ≤ 5)∧
3    (0 ≤ PC_I5 ≤ 5) ∧ (0 ≤ PC_I6 ≤ 5)∧
4  (PC_I1 ≠ PC_I2) ∧ (PC_I1 ≠ PC_I3) ∧ (PC_I1 ≠ PC_I4) ∧ (PC_I1 ≠ PC_I5)∧
5    (PC_I1 ≠ PC_I6) ∧ (PC_I2 ≠ PC_I3) ∧ (PC_I2 ≠ PC_I4) ∧ (PC_I2 ≠ PC_I5)∧
6    (PC_I2 ≠ PC_I6) ∧ (PC_I3 ≠ PC_I4) ∧ (PC_I3 ≠ PC_I5) ∧ (PC_I3 ≠ PC_I6)∧
7    (PC_I4 ≠ PC_I5) ∧ (PC_I4 ≠ PC_I6) ∧ (PC_I5 ≠ PC_I6)∧
8  (PC_I1 < PC_I2) ∧ (PC_I1 < PC_I4) ∧ (PC_I2 < PC_I3) ∧ (PC_I5 < PC_I6)
9  //Persistency time constraints:
10 (−1 ≤ PT_I1 ≤ 7) ∧ (−1 ≤ PT_I2 ≤ 7) ∧ (PC_I1 ≤ PT_I1) ∧ (PC_I2 ≤ PT_I2)∧
11 (PC_I1 < PC_I4 < PC_I5 ⟹ PT_I1 ≤ PC_I5)∧
12   (PC_I1 < PC_I4 < PC_I6 ⟹ PT_I1 ≤ PC_I6)∧
13   (PC_I2 < PC_I3 < PC_I5 ⟹ PT_I2 ≤ PC_I5)∧
14   (PC_I2 < PC_I3 < PC_I6 ⟹ PT_I2 ≤ PC_I6)
15 //Assertion violation constraints:
16 ¬((PT_I1 < 6) ∧ (PT_I2 < 6) ∧ (PT_I1 < PT_I2))
```

Figure 8: Encoding for the THEN-branch of Fig. 6 with both durability and crash consistency assertions.

4.3 Using $\Phi_{persistency}$ to Encode Persistency Order

Let $\Phi_{persistency} := \Phi_{pti} \wedge \Phi_{pts} \wedge \Phi_{fi}$ be a set of constraints on PT_I_i variables such that, for every satisfying assignment to $\Phi_{persistency}$, the values of the PT_I_i variables correspond to a valid persistency order of instructions in $\mathcal{T}$. These subformulas are defined in Fig. 7.

4.3.1 Subformula Φ_{pti}. This *persistency time initialization (pti)* constraint requires that, for each STORE instruction $I_i \in Stores$, the value of PT_I_i is in the interval $[-1, N+1]$. Besides $[0, N)$, here, -1 means I_i has not been executed, N means I_i has been flushed but not yet fenced, and $N+1$ means I_i has not even been flushed yet at the end of the execution.

4.3.2 Subformula Φ_{pts}. This *persistency time store (pts)* requires the persistency time of each $I_i \in Stores$ to be no earlier than the execution time of I_i, i.e., the value of PC_I_i.

4.3.3 Subformula Φ_{fi}. This *fence interval (fi)* constraint requires that, for each $I_i \in Stores$, matching $I_j \in Flushes$, and $I_k \in Fences$, the persistency time of I_i is no later than the execution time of I_k.

4.4 Using $\Phi_{assertion}$ to Encode the Assertion

Let $\Phi_{assertion} := \Phi_{du} \wedge \Phi_{cc}$, where Φ_{du} represents the set of *durability* conditions and Φ_{cc} represents the set of *crash consistency* conditions. Both of them are defined in Fig. 7.

Recall that for each $I_i \in Stores$, the value written by I_i is expected to be stored in persistent media at the end of the execution ($TMAX = N$). Thus, if $(PT\_I_i \ge N)$ is satisfiable, there exists a durability bug. Similarly, given two instructions $I_i, I_j \in Stores$, if I_i is expected to always persist before I_j, then the satisfiability of $(PT\_I_i \ge PT\_I_j)$ means there exists a crash consistency bug.

4.5 An Example for Our Encoding Method

Fig. 8 shows the constraints in Φ constructed by our method for the THEN-branch of Fig. 6, after the new SFENCE instruction I_6 has been added to the end of the trace.

Algorithm 2: $\mathcal{R} \leftarrow$ ComputeRepair$(\mathcal{T}, \mathcal{A})$

```
1 Φ ← Φ_program ∧ Φ_persistency ∧ ¬Φ_assertion
2 R ← true
3 while Satisfiable(Φ ∧ R) do
4 |   ψ_sat ← ExtractSatConstraint(Φ ∧ R)
5 |   R ← R ∧ ¬ψ_sat
6 end while
7 return R
```

Algorithm 3: RepairIsValid$(\mathcal{T}, \mathcal{R})$

```
1 Ψ ← Φ_program ∧ Φ_persistency
2 if Satisfiable(Ψ ∧ R) then
3 |   return true
4 else
5 |   return false
6 end if
```

Specifically, Lines 2-7 encode Φ_{pc}, which requires each PC_I_i to have a unique value in $[0, 5]$. Here, $N = 6$ is the total number of instructions in the extended execution trace $\mathcal{T}$.

Line 8 encodes the program order. In particular, $PC\_I_1 < PC\_I_2$ encodes Φ_{so}, which requires the two STORE instructions to execute in order. $PC\_I_1 < PC\_I_4$ and $PC\_I_2 < PC\_I_3$ encode Φ_{fs}, which requires each CLFLUSHOPT to execute after a corresponding STORE. $PC\_I_5 < PC\_I_6$ encodes Φ_{fo}, which requires the two SFENCE instructions to execute in the same order as in the trace.

Line 10 encodes Φ_{pti} and Φ_{pts}, where Φ_{pti} requires each PT_I_i to have a value in $[-1, 7]$, and Φ_{pts} requires each PT_I_i to be no earlier than the corresponding PC_I_i.

Lines 11-14 encode Φ_{fi}. In particular, $PC\_I_1 < PC\_I_4 < PC\_I_5 \implies PT\_I_1 \leq PC\_I_5$ means that, whenever the *STORE* and CLFLUSHOPT instructions for $0x4a3c000$ execute before the SFENCE instruction I_5, the persistency time for $0x4a3c000$ is guaranteed to be no later than the execution time of I_5.

Finally, Line 16 encodes the conditions under which assertion may be violated.

Since the set of constraints (Φ) in Fig. 8 is satisfiable, an SMT solver may return a solution corresponding to the permutation $\mathcal{T}' = I_1, I_4, I_2, I_3, I_5, I_6$. This is a valid permutation of $\mathcal{T}$ because, according to the **CL** symbol in Table 1, CLFLUSHOPT (I_4) is allowed to reorder before I_2 and I_3. However, it violates the crash consistency property because I_2 may persist before I_1. In the next section, we present our method for repairing this violation.

5 COMPUTING THE REPAIR

Algorithm 2 shows our method for computing a repair $\mathcal{R}$ when the formula Φ is satisfiable. Our method first uses the subroutine ComputeRepair$(\mathcal{T}, \mathcal{A})$ to compute a candidate $\mathcal{R}$, and then uses the subroutine RepairIsValid$(\mathcal{T}, \mathcal{R})$ to check if $\mathcal{R}$ is a valid repair.

5.1 Subroutine ComputeRepair$(\mathcal{T}, \mathcal{A})$

The repair $\mathcal{R}$ is represented by a conjunction of *blocking* constraints, each of which, denoted $\neg\psi_{sat}$, removes a subset of permutations of $\mathcal{T}$ allowed by Φ. Recall that Φ allows only *valid and yet buggy* permutations. Thus, we want to compute a set of blocking constraints that remove all *valid and yet buggy* permutations.

In Algorithm 2, $\mathcal{R}$ is initialized to *true*, which represents an empty repair. Then, as long as $\Phi \wedge \mathcal{R}$ remains satisfiable (Line 3), we compute a constraint ψ_{sat} from the satisfying assignment (*sol*) to the formula $\Phi \wedge \mathcal{R}$. Here, ψ_{sat} is a conjunction of *happens-before* constraints, $(PC\_I_i < PC\_I_j)$, extracted from the antecedents of the subformula Φ_{fi} such that all these *happens-before* constraints are satisfied by the assignment (*sol*).

Since ψ_{sat} captures a set of valid-and-yet-buggy permutations of $\mathcal{T}$, by adding $\neg\psi_{sat}$ to $\mathcal{R}$, we remove them (Line 5). Inside the while-loop of Algorithm 2, we keep adding $\neg\psi_{sat}$ until $\Phi \wedge \mathcal{R}$ is no longer satisfiable.

Within each call to ExtractSatConstraint$(\Phi \wedge \mathcal{R})$, we compute a minimal set of constraints to be included in ψ_{sat} based on the satisfying assignment (*sol*) returned by the SMT solver. This is accomplished using the greedy algorithm as follows:

First, we extract the concrete values of the PC_I_i variables from the assignment (*sol*), and use these concrete values to decide, for each $(PC\_I_i < PC\_I_j)$ constraint in the antecedents of Φ_{fi}, whether the constraint is satisfied. All the satisfied $(PC\_I_i < PC\_I_j)$ constraints are added to ψ_{sat}. Thus, the negation of ψ_{sat} will eliminate permutations associated with the assignment (*sol*).

Before adding $\neg\psi_{sat}$ to $\mathcal{R}$, we remove the obviously-redundant constraints from ψ_{sat}. These are constraints that are implied by other constraints in ψ_{sat}. For example, if ψ_{sat} contains both $(PC\_I_1 < PC\_I_2)$ and $(PC\_I_2 < PC\_I_3)$, then we remove $(PC\_I_1 < PC\_I_3)$ from ψ_{sat} since it is redundant.

5.2 Subroutine RepairIsValid$(\mathcal{T}, \mathcal{R})$

Algorithm 3 shows our method for validating the repair candidate $\mathcal{R}$ in two steps. First, we define a new formula $\Psi := \Phi_{program} \wedge \Phi_{persistency}$ to capture the set of valid permutations of $\mathcal{T}$. Note that Ψ is a subformula of Φ because $\Phi := \Psi \wedge \neg\Phi_{assertion}$. Next, we check if the combined formula $(\Psi \wedge \mathcal{R})$ is satisfiable; we say that $\mathcal{R}$ is a valid repair only if $(\Psi \wedge \mathcal{R})$ is satisfiable.

Fig. 9 illustrates why we check the validity of the repair in this way. Here, formulas $\neg\Phi_{assertion}$ and Ψ can be thought of as *filters* of permutations of the trace $\mathcal{T}$: red ones are buggy and black ones are non-buggy. In this sense, Ψ retains only the valid permutations of $\mathcal{T}$, and the repair candidate $\mathcal{R}$ filters out the valid-and-yet-buggy permutations. If $\mathcal{R}$ retains at least some non-buggy permutation (black arrow), we say that $\mathcal{R}$ is a valid repair. But if $\mathcal{R}$ does not retain any non-buggy permutation at all, it is a vacuous repair.

The existence of some (valid and non-buggy) permutations means that the constraints imposed by $\mathcal{R}$ is realizable.

5.3 An Example for Our Repair Method

We use the example in Fig. 8 to illustrate the repair computation and validation process. Fig. 10 shows the corresponding steps.

First, recall that the constraints (Φ) shown in Fig. 8 are satisfiable. From the first solution to Φ returned by the SMT solver, our method identifies the *happens-before* constraints in the antecedents of the

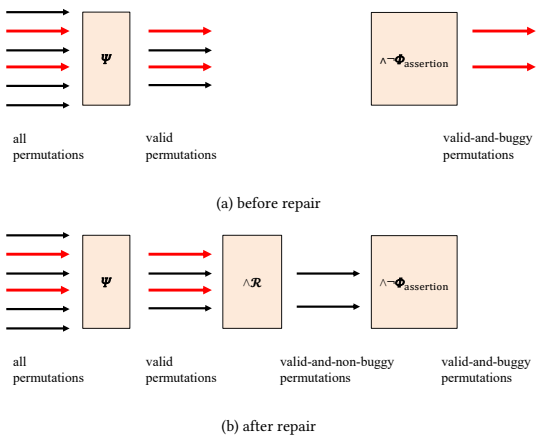

Figure 9: Formulas act as "filters" of the trace permutations, including buggy (red) and non-buggy (black) permutations.

```
//First iteration  -- Satisfiable
From solution 𝒯′ = I1, I2, I3, I4, I5, I6, we extract ψsat as follows:
   (PC_I1 < PC_I4 < PC_I5) ∧ (PC_I2 < PC_I3 < PC_I5)∧
   (PC_I1 < PC_I4 < PC_I6) ∧ (PC_I2 < PC_I3 < PC_I6)

//Second iteration -- Satisfiable
From solution𝒯′ = I5, I1, I2, I3, I4, I6, we extract ψsat as follows:
   (PC_I5 < PC_I1 < PC_I4) ∧ (PC_I5 < PC_I2 < PC_I3)∧
   (PC_I1 < PC_I4 < PC_I6) ∧ (PC_I2 < PC_I3 < PC_I6)

//Third iteration -- Unsatisfiable
Return ℛ as a potential repair

//Validation -- Satisfiable
Found a valid permutation of 𝒯 by solving (Ψ ∧ ℛ)
   Inst I1: STORE       0x4a3c000
   Inst I4: clflushopt  0x4a3c000
   Inst I5: sfence
   Inst I2: STORE       0x4a3c080
   Inst I3: clflushopt  0x4a3c080
   Inst I6: sfence
```

Figure 10: Illustrating the repair computation and validation.

fence interval (fi) subformula Φ_{fi}. This corresponds to Line 4 of Algorithm 2. While there are four antecedents in Φ_{fi} as shown in Fig. 8, only two of them end up in ψ_{sat}, as shown in Fig. 10.

By adding $\neg\psi_{sat}$ to $\mathcal{R}$, our method removes the buggy permutation where instructions I_1 and I_2, together with their CLFLUSHOPT instructions, execute before the first SFENCE instruction in I_5.

Next, we check if $\Phi \wedge \mathcal{R}$ is satisfiable (Line 3 of Algorithm 2). Since the answer is yes, from the second solution to Φ returned by the SMT solver, our method computes another ψ_{sat} and then uses $\neg\psi_{sat}$ to remove the buggy permutation where instructions I_1 and I_2, together with their CLFLUSHOPT instructions, are moved in between instructions I_5 and I_6.

At this moment, the only remaining permutation is as follows: I_1 and its CLFLUSHOPT are before I_5, while I_2 and its CLFLUSHOPT are between I_5 and I_6. Since this permutation does not violate the assertion, our method exists the while-loop in Algorithm 2 and returns $\mathcal{R}$ as a potential repair.

Finally, our method uses Algorithm 3 to validate the repair by checking the satisfiability of $\Psi \wedge \mathcal{R}$. Since $\Psi \wedge \mathcal{R}$ is satisfiable, the SMT solver returns a solution that corresponds to the permutation $\mathcal{T}' = I_1, I_4, I_5, I_2, I_3, I_6$.

This permutation of $\mathcal{T}$ shows exactly how to reorder instructions in the extended execution trace to avoid the assertion violation. Thus, by mapping the reordered instructions from $\mathcal{T}$ back to the original program, we obtain the repaired software code shown in the THEN-branch of Fig. 3.

6 CORRECTNESS AND OPTIMIZATIONS

In this section, we first discuss the correctness of our repair method by treating it as a special case of the well-known *syntax-guided synthesis (SyGuS)* problem [2]. Then, we discuss two optimizations.

6.1 Relating to SyGuS

Our repair problem can be viewed as deciding the existence of a relation $\mathcal{R}$ such that $\Psi(x,y) \wedge \mathcal{R}(x) \implies \Psi_{assertion}(y)$ must be valid (for all x and y) and, at the same time, $\Psi(x,y) \wedge \mathcal{R}(x)$ must be satisfiable (for some x and y). Here, x denotes the set of PC_I_i variables and y denotes the set of PT_I_i variables.

$$\exists \mathcal{R}. \quad (\forall x, y.\ \Psi(x,y) \wedge \mathcal{R}(x) \implies \Phi_{assertion}(y)) \ \wedge \\ (\exists x, y.\ \Psi(x,y) \wedge \mathcal{R}(x))$$

This is the well-known SyGuS problem [2].

In our method, since the *validity* of $A \wedge B \implies C$ is equivalent to the *unsatisfiability* of the negated formula $A \wedge B \wedge \neg C$, we rewrite the problem as follows:

$$\exists \mathcal{R}. \quad \neg(\exists x, y.\ \Psi(x,y) \wedge \mathcal{R}(x) \wedge \neg\Phi_{assertion}(y)) \wedge \\ (\exists x, y.\ \Psi(x,y) \wedge \mathcal{R}(x))$$

This allows use to use off-the-shelf SMT solvers to decide the two satisfiability subproblems. The first one says that $\Psi \wedge \mathcal{R} \wedge \neg\Phi_{assertion}$ must be unsatisfiable, and the second one says that $\Psi \wedge \mathcal{R}$ must be satisfiable. They are the foundations of our method for computing and validating the repair in Algorithms 2 and 3.

The link to SyGuS allows us to understand the complexity of the repair problem. Since quantification is applied to the relation $\mathcal{R}$, the problem is expressed as a formula in second-order logic, which is known to be undecidable in general. That is why practical solutions to the SyGuS problem tend to be sound (and yet incomplete) solutions. In our repair method, we adopt the same approach.

Our Method Is Guaranteed to Be Sound with Respect to the Given Trace. That is, the repair $\mathcal{R}$ computed by our method is guaranteed to be correct. This is because, by definition, $\mathcal{R}$ is able to make $\Psi \wedge \mathcal{R} \wedge \neg\Phi_{assertion}$ unsatisfiable, as shown in Algorithm 2. At the same time, it is able to make $\Psi \wedge \mathcal{R}$ satisfiable, as shown in Algorithm 3. Thus, $\mathcal{R}$ can always eliminate the failed assertion.

Our method is not necessarily complete, meaning that even if there exists a valid repair, in theory, our method may not find it. We do not attempt to make the method complete for efficiency reasons, even if this may be achieved by restricting the search to a decidable solution subspace. Instead, we will demonstrate through experimental evaluation (Section 7) that, in practice, our repair method can always find a valid repair.

6.2 Adding New Instructions to $\mathcal{T}$

So far, our analysis assumes that the set of instructions in the execution trace $\mathcal{T}$ is fixed. Sometimes, however, the PM bug cannot be fixed merely by permuting $\mathcal{T}$; in addition, new CLFLUSHOPT and SFENCE instructions must be added. This is the reason why there is a while-loop in Algorithm 1 and whenever the PM bug cannot be repaired using instructions in given execution trace $\mathcal{T}$, we use AddInstructions (Line 6 in Algorithm 1) to add instructions to $\mathcal{T}$, and try again.

Which instructions to add first depends on the violated assertion. If the violated assertion is $PT\_I_i < PT\_I_j$, our strategy is to add a CLFLUSHOPT instruction whose address is the same as the address of I_i or I_j. If the violated assertion is $PT\_I_i < N$ (a durability bug), our strategy is to add a CLFLUSHOPT instruction first and then check if a valid repair exists; if the violation still exists, we add an SFENCE instruction and check again.

Fig. 6 shows an example. Prior to adding the instruction I_6, the last violated assertion represents the durability of the value written by I_2. Thus, we add an SFENCE instruction. The reason why there is no need to add the CLFLUSHOPT instruction for I_2 is because such an instruction already exists in the given execution trace.

6.3 Relaxing the Subformula Φ_{so}

So far, our analysis assumes that STORE instructions in the given trace $\mathcal{T}$ are executed in the same order as they appear in the program. This is codified in the subformula Φ_{so}. However, enforcing Φ_{so} may prevent some bugs from being repaired.

An example has been shown in the ELSE-branch of Fig. 4. In addition to the durability property ($PT\_I_1 < N$), the user also wants to satisfy the crash consistency property ($PT\_I_2 < PT\_I_1$). However, since the *must-persist-before* constraint ($PT\_I_2 < PT\_I_1$) contradicts with the *happens-before* constraint ($PC\_I_1 < PC\_I_2$) in Φ_{so}, it is impossible to repair the bug. If we assume that $\Phi_{assertion}$ correctly expresses the intended behavior, then we must relax the *happens-before* constraints in Φ_{so}.

In our repair method, the solution is to enforce the subformula Φ_{so} first. However, if this does not lead to a valid repair, we relax it. Toward this end, we first check if Φ_{so} contains a constraint ($PC\_I_i < PC\_I_j$) that contradicts the transitive closure of the *must-persist-before* constraints imposed by the crash consistency requirement Φ_{cc}. If the answer is yes, then we remove the conflicting constraint from Φ_{so}, and try again.

To summarize, whenever the *must-persist-before* constraints in $\Phi_{assertion}$ contradict with the *happens-before* constraints in Φ_{so}, we assume that $\Phi_{assertion}$ is the intended behavior, and relax Φ_{so}.

7 EXPERIMENTS

We implemented our method by using Z3 [7] to conduct the symbolic analysis described in Algorithms 1, 2 and 3. Our method takes an execution trace and a failed assertion as input and returns the repair as output. The known-to-be-buggy execution traces are generated using PMemCheck [19], although many other existing PM bug detection tools [18, 19, 22] can also be used to generate traces.

Table 2: Statistics of the benchmark programs.

Name	LoC	Description	PM Bug Type
obj_constructor	186	Object constructor test [37]	durability
obj_first_next	314	POBJ_FIRST macro test [37]	durability
obj_mem	68	pmemobj copy, move and set tests [37]	durability
obj_memops	654	basic memory operations tests [37]	durability
obj_toid	83	TOID macros test [37]	durability
pmem_memcpy	174	memcpy test [37]	durability
pmem_memmove	223	memmove test [37]	durability
pmem_memset-1	103	memset from libpmemset [37]	durability
pmem_memset-2	103	memset from libpmemset [37]	durability
pmemspoil	1,324	pmempool spoil test [37]	durability
rpmemd_db	653	pool set database [37]	durability
Recipe (2 bugs)	39,581	convert DRAM index to PM index [30]	durability
Memcached (10 bugs)	23,032	key/value cache store in distributed sys [4]	durability
pmreorder_1	141	pmreorder script test [21]	crash consistency
pmreorder_2	141	pmreorder script test [21]	crash consistency
pmreorder_3	141	pmreorder script test [21]	crash consistency
pmreorder_4	141	pmreorder script test [21]	crash consistency
pmreorder_5	141	pmreorder script test [21]	crash consistency
pmreorder_6	141	pmreorder script test [21]	crash consistency
pmreorder_7	141	pmreorder script test [21]	crash consistency
pmreorder_8	141	pmreorder script test [21]	crash consistency
pmreorder_stack_1	123	functional test of pmreorder stack [21]	crash consistency
pmreorder_stack_2	123	functional test of pmreorder stack [21]	crash consistency
pmreorder_flushes_1	155	store reordering with flushes test [21]	crash consistency
pmreorder_flushes_2	155	store reordering with flushes test [21]	crash consistency
Redis (2 bugs)	75,249	distributed, in-memory key–value database [3]	crash consistency
Memcached (4 bugs)	23,032	key/value cache store in distributed sys [4]	crash consistency

7.1 Benchmarks

Table 2 shows the benchmark statistics, including the name, the number of lines of C code (LoC), a short description, and the known PM bug type. These benchmark programs fall into two sets. The first set consists of programs with *durability* bugs. The first ten programs come from the Intel PMDK library. The last two programs are real applications: Memcached [4] is a high-performance object caching system, and Recipe [30] is a set of durable concurrent data structures for fast indexing. The *durability* bugs in these programs have been confirmed by prior work [37]. The second set consists of programs with *crash consistency* bugs. The first twelve are unit-testing programs for durable data structures implemented in the Intel PMDK library. These unit tests are created by Intel developers to illustrate various scenarios under which crash consistency bugs occur. The last two program are two real applications, including Memcached as well as Redis [3], which is a distributed key-value database. All of these crash consistency bugs have been confirmed by the developers.

7.2 Experimental Set-up

Since the only prior work on repairing PM bugs is Hippocrates [37], we focus on comparing our tool, PMBugAssist, with Hippocrates on all benchmark programs. Our experiments were designed to answer the following research questions.

- RQ 1: Is PMBugAssist more effective than Hippocrates in repairing the PM bugs?
- RQ 2: Is PMBugAssist efficient enough for computing repairs for the benchmark programs?
- RQ 3: Does PMBugAssist correctly compute repairs for the benchmark programs?

The experiments were conducted on a computer with AMD Ryzen 5 5600X CPU and 32GB memory, running Ubuntu 20.04.

Table 3: Results of the experimental evaluation.

Name	Trace Length	PMBugAssist (our method)				Hippocrates		
		time(s)	# inst. added	inst. reorder	repaired	repaired	# inst. added	time(s)
obj_constructor	445,254	0.1	1+1	0	✔	✔	1+1	1.4
obj_first_next	559,572	0.3	2+0	0	✔	✔	2+0	6.7
obj_mem	566,129	18.4	11+0	0	✔	✔	210+0	47.1
obj_memops	565,899	0.3	2+0	0	✔	✔	2+0	20.9
obj_toid	419,186	0.1	3+0	0	✔	✔	3+0	1.6
pmem_memcpy	17,008	0.3	4+0	0	✔	✔	4+0	5.8
pmem_memmove	624	0.1	2+0	0	✔	✔	2+0	1.0
pmem_memset-1	194	183.52	1+1	0	✔	✔	1+2	2.6
pmem_memset-2	4,440	0.1	1+0	0	✔	✔	1+0	1.1
pmemspoil	36	0.1	0+1	0	✔	✘	0+1	0.9
rpmemd_db	8,993	0.1	1+1	0	✔	✔	1+0	0.8
Recipe (2 bugs)	500,415	4.5	3+1	0	✔	✔	3+1	0.3
Memcached (10 bugs)	200,939	1,790.2	9+1	0	✔	✔	10+6	0.3
pmreorder_1	8	0.1	1+1	1	✔	✘	N/A	N/A
pmreorder_2	8	0.1	1+1	2	✔	✘	N/A	N/A
pmreorder_3	10	7.9	2+2	4	✔	✘	N/A	N/A
pmreorder_4	10	7.9	2+2	4	✔	✘	N/A	N/A
pmreorder_5	8	4.8	2+2	1	✔	✘	N/A	N/A
pmreorder_6	8	22.9	2+2	4	✔	✘	N/A	N/A
pmreorder_7	10	22.1	2+2	4	✔	✘	N/A	N/A
pmreorder_8	12	3559.12	3+3	5	✔	✘	N/A	N/A
pmreorder_stack_1	26	0.3	0+0	1	✔	✘	N/A	N/A
pmreorder_stack_2	26	0.6	0+0	2	✔	✘	N/A	N/A
pmreorder_flushes_1	35	3857.8	0+0	5	✔	✘	N/A	N/A
pmreorder_flushes_2	35	539.0	0+0	6	✔	✘	N/A	N/A
Redis_1	10,577	0.2	2+2	3	✔	✘	N/A	N/A
Redis_2	10,577	0.2	2+2	3	✔	✘	N/A	N/A
Memcached_1	63,133	0.1	2+2	1	✔	✘	N/A	N/A
Memcached_2	63,133	0.1	2+2	1	✔	✘	N/A	N/A
Memcached_3	63,133	0.1	2+2	1	✔	✘	N/A	N/A
Memcached_4	63,133	0.1	2+2	1	✔	✘	N/A	N/A

7.3 Results for Answering RQ 1

First, we present the experimental results that answer RQ 1. They are shown in the last two columns of Table 3. Here, the first two columns show the benchmark name and the length of the original execution trace $\mathcal{T}$. The last two columns show the effectiveness of the two repair methods: PMBugAssist with Hippocrates. Here, the symbol ✔ means that the method can repair the bug, whereas the symbol ✘ means that the method cannot repair the bug. For each suggested repair generated, we manually inspect and compare it with the developers' fix and verify their correctness.

The first twelve rows of Table 3 are benchmark programs with 23 confirmed *durability* bugs. The last fourteen rows are benchmark programs with 18 confirmed *crash consistency* bugs. The results in Table 3 shows that PMBugAssist was able to repair all of the 41 bugs, while Hippocrates was able to repair 22 of the 23 durability bugs and none of the 18 crash consistency bugs.

We also show, in Table 3, the CLFLUSHOPT+SFENCE instructions added and the time taken by the two methods. Overall, our method added either the same number of instructions or fewer instructions. For *obj_mem*, our method used significantly fewer CLFLUSHOPT instructions than Hippocrates (11+0 versus 210+0) because multiple STORE operations share the same cache line. For *Memcached*, our method used fewer instructions (9+1 versus 10+6) because SFENCE may be shared by multiple STORE operations. For *pmemspoil*, our manual inspection shows that Hippocrates's repair is actually incorrect—at least one CLFLUSHOPT must be added.

While our method takes more time since it conducts the additional semantic analysis of the modified program, this is needed to discover new repair strategies; in contrast, Hippocrates only applies the predefined repair strategy for *durability* bugs bug cannot repair *crash consistency* bugs. For *pmem_memset-1*, our method had a longer running time because the erroneous STORE residing in a loop showed up in the trace many times and thus slowed down our symbolic analysis. Overall, the time taken by our method is reasonable when compared to the alternative of relying on programmers to manually repair the bugs.

7.4 Results for Answering RQ 2

Now, we present the experimental results that answer RQ 2. There are two parts. The first part is shown in Column 2 of Table 2, which reports the program size. It shows that PMBugAssist is able to handle programs with reasonably large code sizes. For example, both Memcached and Recipe have more than 20K lines of C code. The second part is shown in Column 2 of Table 3, which reports the length of the execution trace. It shows that PMBugAssist is able to handle reasonably long execution traces.

Note that neither code size nor trace length is a reliability indicator of how hard the repair problem is. For example, although the majority of durability bugs have traces with more than 100K instructions, the repair problems are often simple, because each ($PT\_I_i < N$) constraint involves only one STORE instruction I_i, and many instructions in the trace are unrelated and thus may be ignored during the analysis. In contrast, while the crash consistency bugs have shorter traces, they have more complex interactions between the PC_I_i and PT_I_i variables and, as a result, have significantly larger search spaces.

For example, even with 10 to 30 instructions in the trace $\mathcal{T}$, the total number of possible repairs in the solution space can be astronomically large (10! to 30!). This means that it is impossible for developers to enumerate the possible repairs manually. This is also the reason why SMT based symbolic analysis is needed.

Column 3 of Table 3 shows that our SMT based symbolic analysis is efficient in computing repairs. Except for *obj_memops*, all durability bugs were repaired in a few seconds. This is the case even for applications such as *Memcached* and *Recipe*, for which our repair method finished within 10 seconds. For *obj_memops*, it took 15 minutes because the program has a very large number of PM accesses and thus requires many SMT solver calls. For crash consistency bugs, our method finished within seconds except for *pmreorder_8*, *pmreorder_flushes_1* and *pmreorder_flushes_2*. For *pmreorder_8*, our method took longer because it went through more iterations in the while-loop, while adding 6 new PM instructions to the original execution trace (shown in Column 4) and reordering 4 instructions in the extended execution trace. For the last two benchmarks, *pmreorder_flushes_1* and *pmreorder_flushes_2*, the reason is because there are more relevant instructions in the traces and more of these instructions need to be reordered to repair the bugs.

Our method also minimizes the number of SFENCE/CLFLUSHOPT instructions added (Section 3). For durability bugs, the results are as efficient as the repairs generated by Hippocrates. For crash consistency bugs (which cannot be handled by Hippocrates), the efficiency of our repairs is shown in Column 4 of Table 3.

7.5 Results for Answering RQ 3

To answer RQ 3, we inspected the repairs computed by our method to see if they are also correct for other traces. Recall that, since each repair is computed from a single trace, in theory, there is no

Table 4: The type of code block that our repair belongs to.

Bug Type	Number of Bugs	Sequential	Branch In Scope	Branch Out of Scope
Durability	23	20 (86%)	3 (14%)	0 (0%)
Crash Consistency	18	16 (89%)	2 (11%)	0 (0%)

guarantee that the repair is correct also for other traces. However, our results show that for all the benchmarks in Table 3, our repairs are correct also for other traces.

The reason is that our repair almost always resides in a local code block, such that the code block (basic block) is either executed in its entirety by a trace, or not executed at all by the trace. An example would be the THEN-branch (or the ELSE-branch) of an If-statement. It is extremely rare for the STORE instructions and the corresponding CLFLUSHOPT and SFENCE instructions to be separated into different code blocks. As a result, a trace executes either all or none of the instructions involved in our repair.

Table 4 shows how often this easy-to-check *sufficient condition* is satisfied in practice. Here, a repair is called *Sequential* when all instructions fall into a straight-line code block, called *Branch In Scope* when all instructions fall into a branch of an If-statement, and called *Branch Out of Scope* when some are in a branch but others are outside of the branch. For *Sequential* and *Branch In Scope*, correctness of the repair is guaranteed for all traces.

Table 4 shows that, for durability, 20 of the 23 repairs (86%) are *Sequential* and only 3 (14%) are *Branch In Scope*. For crash consistency, 16 of the 18 repairs (89%) are *Sequential* and only 2 (11%) are *Branch In Scope*. Whether a repair is *Sequential* or *Branch In Scope* can be checked automatically using static program analysis.

8 RELATED WORK

As we have mentioned earlier, Hippocrates [37] is the only existing PM bug repair tool, but is limited to repairing durability bugs. Our method, in contrast, can also repair crash consistency bugs.

Our work is complementary to existing, trace based PM bug detectors [6, 9, 10, 12–14, 31, 42]. This includes, for example, PMemCheck [19] and Persistence Inspector[18], which are trace based PM bug detection tools from Intel, Pmreorder [22], which is an extension of the Intel tools for explicitly generating trace permutations, Yat [28], which is a framework based on hypervisor for testing persistency bugs on POSIX-compliant file system (PMFS [41]), and Chipmunk [29], which is a framework for testing PM file systems for crash-consistency bugs.

PMTest [33] is a tool that leverages user specified checking rules to compute the persistency time interval of STORE instructions, to decide if there are persistency violations. XFDetector [32] is a tool that automatically injects failures into the program and then replays the execution traces before and after failure, to detect cross-failure bugs. PMDebugger [8] is a also tool that leverages user-specified constraints to detect PM bugs. In addition, there are techniques for verifying the absence of PM bugs [27, 40].

At a high level, our repair method is also related to techniques for repairing other software bugs. They include ExtractFix [11], which is a constraint-based semantic repair approach that leverages an execution trace and a crash-free constraint as input to generate candidate patches that satisfy the constraint, BugAssist [23, 24], which repairs assertion failures in a sequential program, and ConcBugAssist [26], which repairs failures in a multi-threaded program. Other similar repair techniques include SemiFix [39], DirectFix [35], and the method proposed by Malik et al. [34] for repairing data structures. There are also techniques for synthesizing and optimizing fences and synchronization primitives for concurrent programs [1, 5, 25, 36]. However, none of these existing techniques can repair PM bugs.

Our SMT solver based symbolic analysis is related to techniques used by existing tools for traced-based analysis to detect concurrency bugs such as data races and atomicity violations [16, 43, 45–47], as well as symbolic analysis techniques for detecting information leaks through side channels [15, 17]. However, these techniques were designed exclusively for programs that use volatile memory, and thus cannot be used to detect or repair PM bugs.

9 CONCLUSIONS

We have presented a method for automatically repairing both durability and crash consistency bugs in application software that leverages byte-addressable persistent memory. Our method relies on a novel SMT based symbolic analysis to first identify the valid and yet buggy executions allowed by the program, and then remove these executions through iterative addition of blocking constraints. Due to the efficiency of the symbolic analysis over explicit enumeration, our method is able to explore possible repairs in a large solution space quickly. Our experiments on a diverse set of benchmark programs show that the proposed method is significantly more effective in repairing PM bugs than the state-of-the-art approach.

ACKNOWLEDGMENTS

This work was partially funded by the U.S. National Science Foundation grants CNS-1702824 and CCF-2220345.

REFERENCES

[1] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2017. Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion. *ACM Trans. Program. Lang. Syst.* 39, 2 (2017), 6:1–6:38.

[2] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013.* 1–8.

[3] Brad Fitzpatrick et al. 2021. https://github.com/pmem/pmem-redis

[4] Brad Fitzpatrick et al. 2022. https://www.memcached.org

[5] Pavol Cerný, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. 2013. 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 (Lecture Notes in Computer Science, Vol. 8044).* Springer, 951–967.

[6] Zhangyu Chen, Yu Hua, Yongle Zhang, and Luochangqi Ding. 2022. 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, 873–887.

[7] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963).* Springer, 337–340.

[8] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. 2021. 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, 503–516.

[9] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. 2021. Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. ACM, 100–115.

[10] Xinwei Fu, Dongyoon Lee, and Changwoo Min. 2022. DURINN: Adversarial Memory and Thread Interleaving for Detecting Durable Linearizability Bugs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA, 195–211.

[11] Xiang Gao, Bo Wang, Gregory J. Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. *ACM Trans. Softw. Eng. Methodol.* 30, 2 (2021), 14:1–14:27.

[12] Hamed Gorjiara, Weiyu Luo, Alex Lee, Guoqing Harry Xu, and Brian Demsky. 2022. 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, 490–505.

[13] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2021. 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, 415–428.

[14] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2022. Yashme: detecting persistency races. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. ACM, 830–845.

[15] Shengjian Guo, Meng Wu, and Chao Wang. 2018. Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 377–388.

[16] Jeff Huang, Charles Zhang, and Julian Dolby. 2013. 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, 141–152.

[17] Zunchen Huang and Chao Wang. 2022. 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 (Lecture Notes in Computer Science, Vol. 13241)*. Springer, 163–183.

[18] Intel. 2022. 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

[19] Intel. 2022. How to detect persistent memory programming errors using Intel Inspector. https://www.intel.com/content/www/us/en/developer/articles/technical/detect-persistent-memory-programming-errors-with-intel-inspector-persistence-inspector.html

[20] Intel. 2022. Intel Optane Memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html

[21] Intel. 2022. Persistent Memory Development Kit (PMDK). https://https://pmem.io/pmdk/

[22] Intel. 2022. pmreorder - performs a persistent consistency check using a store reordering mechanism. https://pmem.io/pmdk/manpages/linux/master/pmreorder/pmreorder.1/

[23] Manu Jose and Rupak Majumdar. 2011. 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*. 504–509.

[24] Manu Jose and Rupak Majumdar. 2011. 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*. 437–446.

[25] Vineet Kahlon and Chao Wang. 2012. Lock Removal for Concurrent Trace Programs. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science, Vol. 7358)*. Springer, 227–242.

[26] Sepideh Khoshnood, Markus Kusano, and Chao Wang. 2015. 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, 165–176.

[27] Michalis Kokologiannakis, Ilya Kaysin, Azalea Raad, and Viktor Vafeiadis. 2021. PerSeVerE: persistency semantics for verification under ext4. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29.

[28] Philip Lantz, Dulloor Subramanya Rao, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. 2014. Yat: A Validation Framework for Persistent Memory Software. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. 433–438.

[29] Hayley LeBlanc, Shankara Pailoor, Om Saran K. R. E, Isil Dillig, James Bornholt, and Vijay Chidambaram. 2023. Chipmunk: Investigating Crash-Consistency in Persistent-Memory File Systems. In *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*. 718–733.

[30] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. ACM, 462–477.

[31] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Manabi Khan. 2021. PM-Fuzz: 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, 487–502.

[32] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas F. Wenisch, Aasheesh Kolli, and Samira Manabi Khan. 2020. 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, 1187–1202.

[33] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Manabi Khan. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. ACM, 411–425.

[34] Muhammad Zubair Malik, Khalid Ghori, Bassem Elkarablieh, and Sarfraz Khurshid. 2009. 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, 620–624.

[35] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. 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, 448–458.

[36] Yuri Meshman, Noam Rinetzky, and Eran Yahav. 2015. 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, 120–127.

[37] Ian Neal, Andrew Quinn, and Baris Kasikci. 2021. Hippocrates: healing persistent memory bugs without doing any harm. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*. ACM, 401–414.

[38] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. 2020. AGAMOTTO: How Persistent is your Persistent Memory Application?. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 1047–1064.

[39] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. 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, 772–781.

[40] Azalea Raad, Ori Lahav, and Viktor Vafeiadis. 2020. Persistent Owicki-Gries reasoning: a program logic for reasoning about persistent programs on Intel-x86. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 151:1–151:28.

[41] Dulloor Subramanya Rao, Sanjay Kumar, Anil S. Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*. ACM, 15:1–15:15.

[42] Benjamin Reidys and Jian Huang. 2022. 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, 322–336.

[43] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem A. Sakallah. 2011. 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 (Lecture Notes in Computer Science, Vol. 6617)*. Springer, 313–327.

[44] Steve Scargall. 2020. *Programming Persistent Memory A Comprehensive Guide for Developers*. Apress, Berkeley, CA.

[45] Arnab Sinha, Sharad Malik, Chao Wang, and Aarti Gupta. 2011. 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, 99–108.

[46] Chao Wang, Sudipta Kundu, Malay K. Ganai, and Aarti Gupta. 2009. Symbolic Predictive Analysis for Concurrent Programs. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5850)*. Springer, 256–272.

[47] Chao Wang, Rhishikesh Limaye, Malay K. Ganai, and Aarti Gupta. 2010. 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 (Lecture Notes in Computer Science, Vol. 6015)*. Springer, 328–342.