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:

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].

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.

## 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, \((i_j, i_k)\), where \(i_j\) is executed before \(i_k\) by the CPU, the corresponding table entry shows whether Px86 guarantees that \(i_j\) persists before \(i_k\) using the symbols \(\checkmark\) (yes) and \(\times\) (no). The third symbol, CL, means that \(i_j\) persists before \(i_k\) only when the two instructions access memory address blocks mapped to the same cache line.

For example, \((\text{STORE } x, \text{LOAD } y)\) may persist in reverse order according to the \(\times\) 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, \((\text{LOAD } y, \text{STORE } x)\) 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 \(i f (y>0)\) \(\{x=1; \} \) (with control dependency) or the code snippet \(\{ \text{reg} = y; x = 1; \} \) (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 RW (read-modify-write) instructions:

- \text{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 persistence 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 persistence 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 persistence order, which must be enforced by writer(), is that both records[i].name and records[i].addr 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
then-branch (write to records[i].name before and after swapping the execution order of clflushopt 0x4a3c0C0. The assertion is violated because consistency bugs, and can repair bugs that do not syntactically match. 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 I4 and I5 before I2 and I3, 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 I2: without a subsequent SFENCE instruction, the value written by I2 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 I6, 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 T that violates a property assertion 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 [53]. Specifically, the trace T = [I1,...,IN] is a sequence of instructions, each of which has an instruction type specified in Table 1.

The assertion A may be of the form PT(Ii) < TMAX (durability) or PT(Ii) < PT(Ij) (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(Ii) ≥ TMAX, there exists a durability violation where Ii has not yet taken effect in persistent media at the end of the execution.

#### Algorithm 1: Our method R ← PMBUGASSIST(T, A)

1. while BugIsFound(T, A) do
   2.   R ← COMPUTEREPAIR(T, A)
   3.   if RepairIsValid(T, R) then
       4.       return R as repair
   5.   end if
   6.   T ← ADDINSTRUCTIONS(T, A, R)
5. end while

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

There are two possibilities. One possibility is that $R$ indeed is a valid repair: by permuting the instructions in $\mathcal{T}$, $R$ removes all the bad executions and retains only the good executions. The other possibility is that $R$ is a vacuous repair in that, by creating a contradiction between $R$ 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 $R$ is valid or vacuous, we use the subroutine RepairIsValid($\mathcal{A}, R$) to check, after applying $R$ to $\mathcal{T}$, whether any valid execution exists. If the answer is yes, then $R$ is a valid repair, and thus is returned to the user. Otherwise, we use AddInstructions($\mathcal{A}, \mathcal{T}, R$) 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 BugsFound($\mathcal{T}, \mathcal{A}$) 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 BugsFound($\mathcal{T}, \mathcal{A}$) but also the subroutines ComputeRepair($\mathcal{T}, \mathcal{A}$) and RepairIsValid($\mathcal{T}, R$) 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 $\Phi = \Phi_{\text{program}} \land \Phi_{\text{persistency}} \land \neg \Phi_{\text{assertion}}$, where $\Phi_{\text{program}}$ encodes the program order, $\Phi_{\text{persistency}}$ encodes the persistency order, and $\Phi_{\text{assertion}}$ encodes the assertion. Thus, $\Phi$ is satisfiable if and only if there exists a valid execution of the instructions in $\mathcal{T}$ that violates $\mathcal{A}$.

We express $\Phi$ 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 $\Phi$ can be efficiently decided using any off-the-shelf SMT solver.

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

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

- The $PT_{-}Pi$ Variables. For each instruction $i \in \mathcal{T}$ of the STORE type, we define a variable $PT_{-}Pi$ whose value may be any integer in the interval $[0, N + 1]$; it stands for the persistency time of $i$, i.e., when the value written by $i$ is actually stored in persistent media.
4.2 Using $\Phi_{\text{program}}$ to Encode Execution Order
Let $\Phi_{\text{program}} := \Phi_{pc} \land \Phi_{so} \land \Phi_{fso} \land \Phi_{mo}$. $\Phi_{\text{program}}$ is a set of constraints on $PC_{li}$ variables such that, for every satisfying assignment to $\Phi_{\text{program}}$, the values of $PC_{li}$ variables correspond to a valid permutation of $T$.

4.2.1 Subformula $\Phi_{pc}$. This program-counter (pc) constraint restricts each $PC_{li}$ 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 $T$. We also require each $PC_{li}$ variable to have a unique value. The definition of $\Phi_{pc}$ is presented in Fig. 7.

4.2.2 Subformula $\Phi_{so}$. This store-order (so) constraint requires the STORE instructions in $T$ to execute in the same order as they appear in the trace. This is because $Pc8s$ 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 $\varphi$ in Table 1. The definition of $\Phi_{so}$ is also presented in Fig. 7.

While computing the repair, we may choose to relax $\Phi_{so}$ 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 $\Phi_{fso}$. This flush-store (fs) constraint requires that, for each CLEFLUSHOPT ($I_i$), its execution time must be 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., $\text{SameCache}(I_i, I_j)$ holds.

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

4.3 Using $\Phi_{\text{persistency}}$ to Encode Persistency Order
Let $\Phi_{\text{persistency}} := \Phi_{pti} \land \Phi_{pts} \land \Phi_{fi}$. $\Phi_{pti}$ be a set of constraints on $PT_{li}$ variables such that, for every satisfying assignment to $\Phi_{\text{persistency}}$, the values of $PT_{li}$ variables correspond to a valid persistency order of the instructions in $T$. These subformulas are defined in Fig. 7.

4.3.1 Subformula $\Phi_{pti}$. This persistency time initialization (pti) constraint requires that, for each STORE instruction $I_i \in \text{Stores}$, the value of $PT_{li}$ 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 fanned, and $N + 1$ means $I_i$ has not even been flushed yet at the end of the execution.

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

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

4.4 Using $\Phi_{\text{assertion}}$ to Encode the Assertion
Let $\Phi_{\text{assertion}} := \Phi_{du} \land \Phi_{ce}$, where $\Phi_{du}$ represents the set of durability conditions and $\Phi_{ce}$ represents the set of crash consistency conditions. Both of them are defined in Fig. 7.

Recall that for each $I_i \in \text{Stores}$, the value written by $I_i$ is expected to be stored in persistent media at the end of the execution ($T\text{MAX} = N$). Thus, if ($PT_{li} \geq N$) is satisfiable, there exists a durability bug. Similarly, given two instructions $I_i, I_j \in \text{Stores}$, if $I_i$ is expected to always persist before $I_j$, then the satisfiability of ($PT_{li} \geq PT_{lj}$) means there exists a crash consistency bug.

4.5 An Example for Our Encoding Method
Fig. 8 shows the constraints in $\Phi$ constructed by our method for the THEN-branch of Fig. 6, after the new SFENCE instruction $I_k$ has been added to the end of the trace.
Specifically, Lines 2-7 encode \( \Phi_{pc} \), which requires each \( PC_{I_1} \) to have a unique value in \([0, 5]\). Here, \( N = 6 \) is the total number of instructions in the extended execution trace \( T \).

Line 8 encodes the program order. In particular, \( PC_{I_1} < PC_{I_2} \) encodes \( \Phi_{sat} \), which requires the two STORE instructions to execute in order. \( PC_{I_1} < PC_{I_4} \) and \( PC_{I_2} < PC_{I_5} \) encode \( \Phi_{fs} \), which requires each CLFLUSHOPT to execute after a corresponding STORE. \( PC_{I_5} < PC_{I_6} \) encodes \( \Phi_{fs} \), which requires the two SFENCE instructions to execute in the same order as in the trace.

Line 10 encodes \( \Phi_{pti} \) and \( \Phi_{pts} \), where \( \Phi_{pti} \) requires each \( PT_{I_1} \) to have a value in \([-1, 7] \), and \( \Phi_{pts} \) requires each \( PT_{I_1} \) to be no earlier than the corresponding \( PC_{I_1} \).

Lines 11-14 encode \( \Phi_{fi} \). In particular, \( PC_{I_1} < PC_{I_4} < PC_{I_5} \Rightarrow PT_{I_1} \leq PC_{I_5} \) means that, whenever the STORE and CLFLUSHOPT instructions for \( 0x4a3c00 \) execute before the SFENCE instruction \( I_5 \), the persistency time for \( 0x4a3c00 \) 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 (\( \Phi \)) in Fig. 8 is satisfiable, an SMT solver may return a solution corresponding to the permutation \( T' = I_1, I_3, I_2, I_4, I_5, I_6 \). This is a valid permutation of \( T \) because, according to the CL symbol in Table 1, CLFLUSHOPT (\( I_5 \)) 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 \( R \) when the formula \( \Phi \) is satisfiable. Our method first uses the subroutine ComputeRepair(\( T, A \)) to compute a candidate \( R \), and then uses the subroutine RepairIsValid(\( T, R \)) to check if \( R \) is a valid repair.

5.1 Subroutine ComputeRepair(\( T, A \))

The repair \( R \) is represented by a conjunction of blocking constraints, each of which, denoted \( \neg \psi_{sat} \), removes a subset of permutations of \( T \) allowed by \( \Phi \). Recall that \( \Phi \) 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, \( R \) is initialized to \( true \), which represents an empty repair. Then, as long as \( \Phi \land R \) remains satisfiable (Line 3), we compute a constraint \( \psi_{sat} \) from the satisfying assignment (sol) to the formula \( \Phi \land R \). Here, \( \psi_{sat} \) is a conjunction of happens-before constraints, \( (PC_{I_1} < PC_{I_2}) \), extracted from the antecedents of the subformula \( \Phi_{fi} \) such that all these happens-before constraints are satisfied by the assignment (sol).

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

Within each call to ExtractSatConstraint(\( \Phi \land R \)), we compute a minimal set of constraints to be included in \( \psi_{sat} \). This is accomplished using the greedy algorithm as follows:

First, we extract the concrete values of the \( PC_{I_1} \) variables from the assignment (sol), and use these concrete values to decide, for each \( (PC_{I_1} < PC_{I_2}) \) constraint in the antecedents of \( \Phi_{fi} \), whether the constraint is satisfied. All the satisfied \( (PC_{I_1} < PC_{I_2}) \) constraints are added to \( \psi_{sat} \). Thus, the negation of \( \psi_{sat} \) will eliminate permutations associated with the assignment (sol).

Before adding \( \neg \psi_{sat} \) to \( R \), we remove the obviously-redundant constraints from \( \psi_{sat} \). These are constraints that are implied by other constraints in \( \psi_{sat} \). For example, if \( \psi_{sat} \) 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 \( \psi_{sat} \) since it is redundant.

5.2 Subroutine RepairIsValid(\( T, R \))

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

Fig. 9 illustrates why we check the validity of the repair in this way. Here, formulas \( \neg \phi_{assertion} \) and \( \Psi \) can be thought of as filters of permutations of the trace \( T \): red ones are buggy and black ones are non-buggy. In this sense, \( \Psi \) retains only the valid permutations of \( T \), and the repair candidate \( R \) filters out the valid-and-yet-buggy permutations. If \( R \) retains at least some non-buggy permutation (black arrow), we say that \( R \) is a valid repair. But if \( R \) 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 \( R \) 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 (\( \Phi \)) shown in Fig. 8 are satisfiable. From the first solution to \( \Phi \) returned by the SMT solver, our method identifies the happens-before constraints in the antecedents of the

\begin{algorithm}[H]
\caption{ComputeRepair(\( T, A \))}
\begin{algorithmic}[1]
\State \( \Phi \leftarrow \Phi_{program} \land \Phi_{persistency} \land \neg \phi_{assertion} \)
\State \( R \leftarrow true \)
\While {\text{Satisfiable}(\( \Phi \land R \))}
  \State \( \psi_{sat} \leftarrow \text{ExtractSatConstraint}(\Phi \land R) \)
  \State \( R \leftarrow R \land \neg \psi_{sat} \)
\EndWhile
\State return \( R \)
\end{algorithmic}
\end{algorithm}

\begin{algorithm}[H]
\caption{RepairIsValid(\( T, R \))}
\begin{algorithmic}[1]
\State \( \Psi \leftarrow \Phi_{program} \land \Phi_{persistency} \)
\If {\text{Satisfiable}(\( \Psi \land R \))}
  \State return \text{true}
\Else
  \State return \text{false}
\EndIf
\end{algorithmic}
\end{algorithm}
permutations

Algorithm 2. While there are four antecedents in $\mathcal{I}$ checking the satisfiability of $\Psi$ assertion, our method exists the while-loop in Algorithm 2 and

between instructions $\mathcal{I}_1$ and $\mathcal{I}_2$, together with their CLFLUSHOPT instructions, are moved in

and $\mathcal{I}_2$, together with their CLFLUSHOPT instructions, are moved in

the same time, it is able to make

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 $\Psi(x, y) \land \mathcal{R}(x) \implies \Phi_{\text{assertion}}(y)$ must be valid (for all $x$ and $y$) and, at the same time, $\Psi(x, y) \land \mathcal{R}(x)$ must be satisfiable (for some $x$ and $y$). Here, $x$ denotes the set of $\text{PC}_{\mathcal{I}_1}$ variables and $y$ denotes the set of $\mathcal{PT}_{\mathcal{I}_1}$ variables.

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

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

This allows us to use off-the-shelf SMT solvers to decide the two satisfiability subproblems. The first one says that $\Psi \land \mathcal{R} \land \neg \Phi_{\text{assertion}}$ must be unsatisfiable, and the second one says that $\Psi \land \mathcal{R}$ 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 process. 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 $\Psi \land \mathcal{R} \land \neg \Phi_{\text{assertion}}$ unsatisfiable, as shown in Algorithm 2. At the same time, it is able to make $\Psi \land \mathcal{R}$ 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 $PT \cdot I_i < PT \cdot I_j$, 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 $PT \cdot I_i < N$ (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 $I_k$, the last violated assertion represents the durability of the value written by $I_k$. Thus, we add an SFENCE instruction. The reason why there is no need to add the CLFLUSHOPT instruction for $I_k$ is because such an instruction already exists in the given execution trace.

6.3 Relaxing the Subformula $\Phi_{so}$

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 $\Phi_{so}$. However, enforcing $\Phi_{so}$ 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 ($PT \cdot I_i < N$), the user also wants to satisfy the crash consistency property ($PT \cdot I_i < PT \cdot I_j$). However, since the must-persist-before constraint ($PT \cdot I_i < PT \cdot I_j$) contradicts with the happens-before constraint ($PC \cdot I_j < PC \cdot I_j$) in $\Phi_{so}$, 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 $\Phi_{so}$.

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

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

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>
<thead>
<tr>
<th>Name</th>
<th>Loc</th>
<th>Description</th>
<th>PM Bug Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj_constructor</td>
<td>146</td>
<td>Object constructor test [57]</td>
<td>durability</td>
</tr>
<tr>
<td>obj_first_new</td>
<td>314</td>
<td>PMB BugFix macro tests [17]</td>
<td>durability</td>
</tr>
<tr>
<td>obj_memmove</td>
<td>66</td>
<td>partially copy, move and set tests [20]</td>
<td>durability</td>
</tr>
<tr>
<td>obj_mmap</td>
<td>654</td>
<td>basic memory operations tests [57]</td>
<td>durability</td>
</tr>
<tr>
<td>obj_load</td>
<td>83</td>
<td>HVM macros test [17]</td>
<td>durability</td>
</tr>
<tr>
<td>pmem_memcpy</td>
<td>378</td>
<td>memcpy test [91]</td>
<td>durability</td>
</tr>
<tr>
<td>pmem_memcpymove</td>
<td>223</td>
<td>memmove test [39]</td>
<td>durability</td>
</tr>
<tr>
<td>pmem_mremap</td>
<td>103</td>
<td>memremap from libpmemset [37]</td>
<td>durability</td>
</tr>
<tr>
<td>pmem_mremap_move</td>
<td>103</td>
<td>memremap from libpmemset [37]</td>
<td>durability</td>
</tr>
<tr>
<td>pmem_mremap_1</td>
<td>1,24</td>
<td>pmemremap agp test [13]</td>
<td>durability</td>
</tr>
<tr>
<td>pmem_memcpy</td>
<td>655</td>
<td>pmem_memcpy database [57]</td>
<td>durability</td>
</tr>
<tr>
<td>Recipe (2 bugs)</td>
<td>123</td>
<td>convert DREAM index to PM index [38]</td>
<td>durability</td>
</tr>
<tr>
<td>Memcached (10 bugs)</td>
<td>25,032</td>
<td>key-value cache store in distributed sys [4]</td>
<td>durability</td>
</tr>
<tr>
<td>pmemreorder_1</td>
<td>141</td>
<td>pmemreorder script test [21]</td>
<td>crash consistency</td>
</tr>
<tr>
<td>pmemreorder_2</td>
<td>141</td>
<td>pmemreorder script test [21]</td>
<td>crash consistency</td>
</tr>
<tr>
<td>pmemreorder_3</td>
<td>141</td>
<td>pmemreorder script test [21]</td>
<td>crash consistency</td>
</tr>
<tr>
<td>pmemreorder_4</td>
<td>141</td>
<td>pmemreorder script test [21]</td>
<td>crash consistency</td>
</tr>
<tr>
<td>pmemreorder_5</td>
<td>141</td>
<td>pmemreorder script test [21]</td>
<td>crash consistency</td>
</tr>
<tr>
<td>pmemreorder_6</td>
<td>141</td>
<td>pmemreorder script test [21]</td>
<td>crash consistency</td>
</tr>
<tr>
<td>pmemreorder_7</td>
<td>141</td>
<td>pmemreorder script test [21]</td>
<td>crash consistency</td>
</tr>
<tr>
<td>pmemreorder_8</td>
<td>141</td>
<td>pmemreorder script test [21]</td>
<td>crash consistency</td>
</tr>
<tr>
<td>pmemreorder_stack_1</td>
<td>123</td>
<td>functional test of pmemreorder stack [21]</td>
<td>crash consistency</td>
</tr>
<tr>
<td>pmemreorder_stack_2</td>
<td>123</td>
<td>functional test of pmemreorder stack [21]</td>
<td>crash consistency</td>
</tr>
<tr>
<td>pmem_reorder</td>
<td>145</td>
<td>store reordering with flush test [94]</td>
<td>crash consistency</td>
</tr>
<tr>
<td>pmem_reorder_1</td>
<td>145</td>
<td>store reordering with flush test [94]</td>
<td>crash consistency</td>
</tr>
<tr>
<td>pmem_reorder_2</td>
<td>145</td>
<td>store reordering with flush test [94]</td>
<td>crash consistency</td>
</tr>
<tr>
<td>Redis (2 bugs)</td>
<td>35,249</td>
<td>distributed, in-memory key-value database [1]</td>
<td>crash consistency</td>
</tr>
<tr>
<td>Memcached (4 bugs)</td>
<td>25,032</td>
<td>key-value cache store in distributed sys [4]</td>
<td>durability</td>
</tr>
</tbody>
</table>

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 as well as Redis [30], and Recipe [37], which confirms the 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Trace Length</th>
<th>PMBugAssist (new method)</th>
<th>Hippocrates</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (s)</td>
<td>Time (s)</td>
<td>Time (s)</td>
</tr>
<tr>
<td></td>
<td>Instructions added</td>
<td>Instructions added</td>
<td>Instructions added</td>
</tr>
<tr>
<td>obj_constructor</td>
<td>160.246</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>obj_rectifier</td>
<td>100.375</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>obj_mem</td>
<td>166.129</td>
<td>18.4</td>
<td>3±10</td>
</tr>
<tr>
<td>obj_memops</td>
<td>169.899</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>obj_memon</td>
<td>154.136</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>pmem_memopy</td>
<td>17.068</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>pmem_memories</td>
<td>624</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>pmem_memset-1</td>
<td>118</td>
<td>153.3</td>
<td>1</td>
</tr>
<tr>
<td>pmem_memset-2</td>
<td>118</td>
<td>153.3</td>
<td>1</td>
</tr>
<tr>
<td>pmem_reorder</td>
<td>4,440</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>pmemspoil</td>
<td>36</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>pmemspoil_1</td>
<td>593</td>
<td>0.1</td>
<td>1</td>
</tr>
<tr>
<td>pmcontrol_1</td>
<td>1,041</td>
<td>4.3</td>
<td>3±1</td>
</tr>
<tr>
<td>Memcached (10 bugs)</td>
<td>208.919</td>
<td>1,796.2</td>
<td>0±1</td>
</tr>
<tr>
<td>pmreorder_1</td>
<td>8</td>
<td>0.1</td>
<td>1±1</td>
</tr>
<tr>
<td>pmreorder_2</td>
<td>8</td>
<td>0.1</td>
<td>1±1</td>
</tr>
<tr>
<td>pmreorder_3</td>
<td>10</td>
<td>7.5</td>
<td>2±4</td>
</tr>
<tr>
<td>pmreorder_4</td>
<td>10</td>
<td>7.5</td>
<td>2±4</td>
</tr>
<tr>
<td>pmreorder_5</td>
<td>8</td>
<td>4.8</td>
<td>2±1</td>
</tr>
<tr>
<td>pmreorder_6</td>
<td>8</td>
<td>22.3</td>
<td>2±1</td>
</tr>
<tr>
<td>pmreorder_stack_1</td>
<td>10</td>
<td>22.3</td>
<td>2±1</td>
</tr>
<tr>
<td>pmreorder_stack_2</td>
<td>12</td>
<td>1595.12</td>
<td>3±1</td>
</tr>
<tr>
<td>pmreorder_stack_3</td>
<td>36</td>
<td>1851.0</td>
<td>3±1</td>
</tr>
<tr>
<td>pmreorder_flushes_1</td>
<td>10</td>
<td>319.5</td>
<td>3±1</td>
</tr>
<tr>
<td>pmreorder_flushes_2</td>
<td>10</td>
<td>319.5</td>
<td>3±1</td>
</tr>
<tr>
<td>pmreorder_flushes_3</td>
<td>10</td>
<td>319.5</td>
<td>3±1</td>
</tr>
<tr>
<td>pmreorder_flushes_4</td>
<td>10</td>
<td>319.5</td>
<td>3±1</td>
</tr>
<tr>
<td>pmreorder_stack_4</td>
<td>10</td>
<td>319.5</td>
<td>3±1</td>
</tr>
<tr>
<td>pmreorder_7</td>
<td>10</td>
<td>319.5</td>
<td>3±1</td>
</tr>
<tr>
<td>pmreorder_8</td>
<td>10</td>
<td>319.5</td>
<td>3±1</td>
</tr>
<tr>
<td>Memcached_1</td>
<td>63.133</td>
<td>0.1</td>
<td>2±2</td>
</tr>
<tr>
<td>Memcached_2</td>
<td>63.133</td>
<td>0.1</td>
<td>2±2</td>
</tr>
<tr>
<td>Memcached_3</td>
<td>63.133</td>
<td>0.1</td>
<td>2±2</td>
</tr>
<tr>
<td>Memcached_4</td>
<td>63.133</td>
<td>0.1</td>
<td>2±2</td>
</tr>
</tbody>
</table>

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 $T$. 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 $T$, 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
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 CLFUSHOPT 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, HIPCRATES [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, PM-MEMCHECK [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 also a 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 includeExtractFix [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 ConBlaze [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