Discovering Likely Program Invariants for Persistent Memory

Zunchen Huang
University of Southern California
Los Angeles, California, USA

Srivatsan Ravi
University of Southern California
Los Angeles, California, USA

Chao Wang
University of Southern California
Los Angeles, California, USA

Abstract

We propose a method for automatically discovering likely program invariants for persistent memory (PM), which is a type of fast and byte-addressable storage device that can retain data after power loss. The invariants, also called PM properties or PM requirements, specify *which objects of the program should be made persistent* and *in what order*. Our method relies on a combination of static and dynamic analysis techniques. Specifically, it relies on static analysis to compute dependence relations between LOAD/STORE instructions and instruments the information into the executable program. Then, it relies on dynamic analysis of the execution traces and counterfactual reasoning to infer PM properties. With precisely computed dependence relations, the inferred properties are *necessary conditions* for the program to behave correctly through power loss and recovery; with imprecise dependence relations, these are likely program invariants. We have evaluated our method on benchmark programs including eight persistent data structures and two distributed storage applications, Redis and Memcached. The results show that our method can infer PM properties quickly and these properties are of higher quality than those inferred by a state-of-the-art technique. We also demonstrate the usefulness of the inferred properties by leveraging them for PM bug detection, which significantly improves the performance of a state-of-the-art PM bug detection technique.

ACM Reference Format:
Zunchen Huang, Srivatsan Ravi, and Chao Wang. 2024. Discovering Likely Program Invariants for Persistent Memory. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695544

1 Introduction

Persistent memory (PM) is a type of emerging storage device with fast and direct access at the granularity of LOAD and STORE instructions. Since it can also retain data in the presence of power failure, it bridges the gap between volatile DRAM and conventional non-volatile storage devices such as solid-state disks. However, writing software code that can utilize PM correctly and efficiently remains a challenging task [18]. The current practice requires the programmers to specify and enforce PM related program invariants. These invariants must identify not only *the objects that should be made persistent* but also *the order in which they should be made persistent*. However, manually specifying such low-level properties can be difficult. While there are PM-specific libraries including Intel PMDK [19] for programmers to enforce data persistency, the library APIs may still be misused, thus leading to PM bugs.

The lack of PM property specifications can negatively affect downstream software engineering tasks such as testing/verification, fault localization, and program repair. As an example, consider existing techniques for PM bug detection [4, 10–14, 24, 33], which generally fall into two categories. The first category consists of techniques that leverage heuristics or known bug patterns to search for violations. The second category consists of techniques that leverage more general and advanced search algorithms, such as symbolic execution [27] and model checking [13]. However, regardless of the underlying algorithms, existing PM bug detection techniques have a common requirement: the intended PM properties must be specified before violations of these properties can be found. There are similar requirements for PM related fault localization and program repair techniques.

The current state of practice, which relies solely on programmers to specify PM properties, has severe limitations. The process is not only tedious and time-consuming but also error-prone. For example, if programmers over-specify the set of persistent objects and/or the ordering constraints, it may lead to degraded performance. In the extreme case where every STORE to PM is immediately flushed from cache and fenced in memory, the caching and buffering optimizations of the CPU and the memory subsystem will be disabled. On the other hand, if programmers under-specify, meaning they miss some persistent objects and/or ordering constraints that should have been included, the program may have abnormal behavior when it goes through power loss and recovery.

Instead of relying solely on programmers to specify PM properties, we propose a method for automatically inferring them. Our method takes the source code of a C program as input, and returns a set of PM properties as output. As shown in Figure 1, our method relies on a combination of static and dynamic analysis techniques. First, static analysis techniques are used to compute the dependence relations of LOAD/STORE instructions inside the LLVM compiler. Then, the dependence relations are instrumented into the executable program, to generate the execution traces. Next, dynamic analysis techniques are applied to the execution traces to first infer the *must-persist-before* (MPB) requirements and then infer the *durability* (DURA) and *must-persist-atomically* (MPA) requirements. Here, $MPB(st_1, st_2)$ requires STORE st_1 to persist before STORE st_2 in all possible executions; $DURA(st_1)$ is a special case of MPB that requires st_1 to persist before the end of the program execution; and $MPA(st_1, \ldots, st_n)$ requires a set of STOREs to persist atomically (either all or none).

To have a more intuitive understanding of these PM properties, consider the example program in Figure 2, which inserts a node to a singly-linked list. With the goal of retaining the list content through power loss and recovery, both the existing nodes (A and

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695544

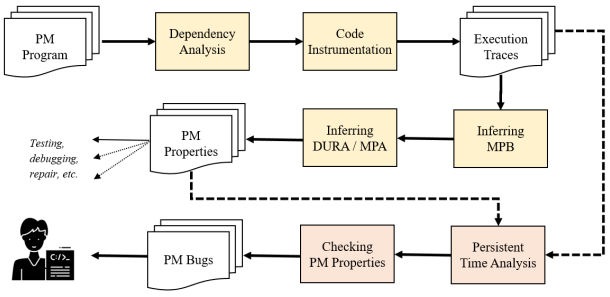


Figure 1: Discovering persistent memory related program invariants (PM properties) and checking them for violations.

C) and the new node (B) must be stored in PM. This is accomplished by the STORE operations in Lines 1, 2 and 6, together with the CLFLUSHOPT and SFENCE instructions shown in green color. Commenting out these green-colored instructions will lead to PM bugs. First, STOREs to `B->data`, `B->next` and `A->next` at Lines 1-2 and 6 must satisfy the durability (DURA) property, meaning they must persist before the end of the program. In addition, STORE to `B->next` at Line 2 must persist before STORE to `A->next` at Line 6. This can be expressed as an MPB property, to ensure that the list is always in a consistent state during node insertion, as shown by Figure 2 (b). Together, the DURA and MPB properties form a *necessary condition* for the list to remain consistent through power loss and recovery. The reason is because power loss may strike at any time during the program execution. If STORE to `A->next` at Line 6 is allowed to persist before STORE to `B->next` at Line 2, as shown by Figure 2 (c), the content of the list may become inconsistent in PM.

In addition to MPB and DURA, sometimes MPA is needed to ensure that a set of STOREs persist atomically. For example, while nodes in a *singly-linked list* are always traversed from left to right following the `next` pointers as shown in Figure 2, in a *doubly-linked list*, nodes may also be traversed from right to left following the `prev` pointers (to be shown in Figure 3). They will create circular requirements that can only be satisfied by updating the `C->prev` and `A->next` pointers atomically. In the context of PM programming, MPA may be enforced by a persistent transaction.

One notable difference between our method and dynamic invariant generation techniques like Daikon [8] is that our method does not rely on correlation; instead, it relies on causation discovered by counterfactual reasoning. In logic, "*if P then Q*" is equivalent to "*if Q is false then P is false*" and Q is called a "necessary" condition for P. Since our method relies on this type of counterfactual reasoning, under the assumption that precise dependence relations are computed, it guarantees that the inferred PM property is a true invariant. In other words, if the property is violated, there exists a concrete execution showing that the program can definitely go wrong during power loss and recovery. Existing techniques like Daikon do not provide such guarantees since they rely on *correlation*, which does not always imply *causation*.

While the inferred PM properties may be useful in many ways, in this work we demonstrate their usefulness in one specific downstream software engineering task, which is PM bug detection. As shown at the bottom of Figure 1, we propose a trace-based analysis algorithm for computing persistent time intervals and then leveraging these intervals to detect violations of the inferred properties. Consider the example program in Figure 2 again. If any of the CLFLUSHOPT instructions is commented out, the corresponding STORE will have a durability (DURA) violation. Furthermore, if the SFENCE instruction at Line 5 is commented out, it will become possible for STOREs at Lines 1-2 to persist after STORE at Line 6, thus leading to MPB violations.

We have implemented our method in a software tool that leverages the LLVM compiler to conduct static analysis and code instrumentation, and relies on Python to implement our method for analyzing the execution traces, inferring properties, and checking properties. Our tool has been evaluated on benchmark programs consisting of eight persistent data structures (lists, array, queues, and ring buffer) and two PM-enabled distributed storage applications (Redis and Memcached). The experimental results show that our method can quickly infer PM properties, and the quality is significantly higher than those inferred by a state-of-the-art technique [10]. We also leveraged the inferred properties for bug detection and found that they significantly improved the performance of a state-of-the-art PM bug detection technique [25].

To summarize, this paper makes the following contributions:

- We propose a method for automatically discovering PM properties based on a combination of static and dynamic analysis techniques and, most notably, counterfactual reasoning.
- To demonstrate the usefulness of the inferred PM properties, we also conduct a trace-based analysis algorithm for checking and detecting violations of these properties.
- We evaluate our method on a number of persistent data structures and two distributed storage applications to demonstrate its advantages over state-of-the-art techniques.

The remainder of this paper is organized as follows. We provide the technical background in Section 2, and present our top-level procedure in Section 3. Then, we present our detailed algorithms for inferring PM properties in Section 4 and for checking these properties in Section 5. We present the experimental results in Section 6. After reviewing the related work in Section 7, we give our conclusions in Section 8.

2 Background

In this section, we review the basics of persistent memory (PM) programs and PM related properties.

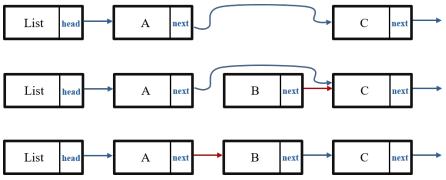
2.1 Persistent Memory

Since persistent memory provides the same *byte-addressable* LOAD and STORE access as volatile DRAM, it may be mapped to the address space of a program just like DRAM. However, the difference is that data written to PM by a STORE instruction alone is not guaranteed to take effect immediately due to CPU hardware optimizations such as caching and buffering. In the presence of power loss, which may strike at any moment during the program execution, data stored in volatile parts of the CPU will be lost permanently.

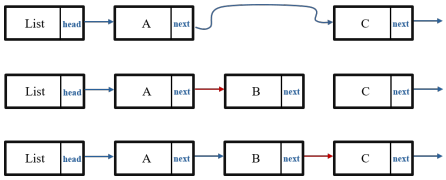
To ensure data persistency, programmers must add special instructions after a STORE to PM, to *explicitly* flush data from cache

```
1  B->data = data;
2  B->next = A->next;
3  //CLFLUSHOPT(&B->data,sizeof(int));
4  //CLFLUSHOPT(&B->next,sizeof(Node*));
5  //SFENCE();
6  A->next = B;
7  //CLFLUSHOPT(&A->next,sizeof(Node*));
8  //SFENCE();
```

(a) persistent memory program



(b) correct persistent order



(c) incorrect persistent order

Figure 2: Inserting a node to a *singly-linked list* stored in persistent memory (PM). To ensure the correct persistent order shown in subfigure (b), all CLFLUSHOPT and SFENCE statements in subfigure (a) are needed. Otherwise, after power failure (which may strike at any moment), the PM state may be inconsistent as shown by the broken list in the middle of subfigure (c).

and insert a memory fence. For Intel CPUs built upon the *persistent x86* architecture [32], the flush and fence instructions are CLFLUSHOPT (optimized cache line flush) and SFENCE (store memory fence), which have been used by the example program in Figure 3 (a).

A typical instruction sequence following a STORE to PM at the address `&v` would be `{v=data;CLFLUSHOPT(&v);SFENCE();}` which first flushes the cache line associated with the address `(&v)` and then inserts the memory fence. Since CLFLUSHOPT is non-blocking, meaning it may return *before* the cache line is written to PM, the subsequent SFENCE is necessary. In other words, missing either CLFLUSHOPT or SFENCE would not guarantee data persistency.

At the same time, it is beneficial to minimize the number of SFENCEs used by a program due to their significant performance overhead. In Figure 3 (a), for example, the SFENCE at Line 5 is shared by the two STOREs at Lines 1 and 2.

2.2 PM Programs

In this work, we are concerned with programs that directly read from and write to PM using LOAD and STORE instructions. Given such a program P, where $STMTs = \{st_1, \ldots, st_n\}$ is the set of all statements, we use $STORES \subseteq STMTs$ to represent the subset that write to PM, and use $LOADs \subseteq STMTs$ to represent the subset that read from PM.

We use *CDEP* to represent the control dependence relation. That is, $CDEP(st, st')$ holds if and only if program statement st is control dependent on program statement st'. While a classic example for $CDEP(st, st')$ would be `if(x>0) y=1; else y=0;` where the STORE to y is control dependent on the LOAD from x, there are more subtle examples.

For instance, in a singly-linked list, there will be a control dependence relation in `{A->next->prev = B;}` since it has an implicit LOAD from `A->next` before the STORE to `A->next->prev`, and the STORE is control dependent on the LOAD. The reason is because if `A->next` is aliased to the object `C`, for instance, the STORE would be directed to `C->prev`; but if `A->next` is aliased to `B`, the STORE would be directed to `B->prev`, as illustrated by the example program in Figure 2.

Similar to *CDEP*, we use *DDEP* to represent the data dependence relation. That is, $DDEP(st, st')$ holds if and only if program statement st is data-dependent on the program statement st'. A classic example for $DDEP(st, st')$ would be `{y = x+1;}` where the value stored to y depends on the value read from x. Both control and data dependence relations are crucial for inferring PM properties.

2.3 PM Properties

PM properties are a special type of correctness requirements concerned with PM LOAD and STORE accesses. They are meant to specify *which objects should be made persistent* and *in what order*.

2.3.1 Durability (DURA). DURA is a unary relation over the set of program statements. $DURA(st)$, where $st \in STORES$, means that the STORE must persist in PM before the end of the program execution. While CPU voluntarily flushes data from cache to PM, the process is highly non-deterministic, and thus should not be relied upon to ensure durability. To ensure durability, programmers must add CLFLUSHOPT and SFENCE instructions to the program both correctly and efficiently.

2.3.2 Must-Persist-Before (MPB). MPB is a binary relation over the set of program statements. $MPB(st, st')$, where $st, st' \in STORES$, means that the first STORE (st) must persist before the second STORE (st'). This property is often needed to ensure that the PM program state remains consistent through potential power loss and recovery. For example, in Figure 2, the STORE to `B->next` must persist before the STORE to `A->next`.

Another example for the MPB property is a program that uses a `flag` to indicate whether a `data` field is valid in PM. When recovering from power loss, the program would read the `flag` first, and then read the `data` field only if the `flag` is properly set. What it requires is that, while writing to PM, the `data` field must always persist before the `flag` field. Otherwise, while reading from PM, the program will get a stale value from the `data` field.

2.3.3 Must-Persist-Atomically (MPA). MPA is a k-ary relation over the set of program statements. That is, $MPA(st_1, \ldots, st_k)$, where $st_1, \ldots, st_k \in STORES$, means that these STOREs must persist atomically. This property may be needed to ensure that a complex data structure is always in a consistent state. Recall the doubly-linked list example mentioned earlier, where the *A->next* and *C->prev* fields must persist atomically. If the MPA property is not satisfied, the list may get into a inconsistent state during power loss and recovery.

To enforce MPA, the notion of a *persistent transaction* is needed. In existing PM libraries, such as Intel's PMDK library, there are dedicated APIs for persistent transactions. Typically, they include `TX_BEGIN()`, `TX_END()` and `TX_ADD()`. As long as all STOREs in a transaction are executed between `TX_BEGIN()` and `TX_END()`,

and their addresses are registered using `TX_ADD(&v)`, the STOREs will persist atomically. We will show an example for persistent transaction in the next section (Figure 3).

The three types of PM properties mentioned above are complete in the sense that they cover all PM properties checked by existing PM bug detection tools. In fact, most of the existing tools can detect *durability bugs*, which are violations of DURA properties. Some tools can detect *crash consistency bugs*, which are violations of MPB properties. Very few can detect *atomicity bugs*, which are violations of MPA properties. As for existing PM bug repair tools, to the best of our knowledge, none of them can repair MPA violations: for example, the Hippocreates tool of Neal et al. [26] can repair DURA violations only, while the PMBugAssist tool of Huang et al. [16] can repair both DURA and MPB violations. Furthermore, in the literature, PM bugs are often classified in a way that seems *ad hoc*. We are the first to propose a unified and graph-theoretic framework for classifying PM bugs (see Section 3.1).

2.4 Invariant Generation

In program analysis, the notion of *program invariants* goes far beyond PM-specific properties concerned in this work. In general, a program invariant may be any logical assertion that always holds during the execution of a program. Existing techniques for discovering program invariants fall into two categories: some rely on static analysis techniques such as abstract interpretation [5], while the others like Daikon [7] rely on dynamic analysis techniques. Both types of techniques tend to focus on values of the program variables. In contrast, our method focuses solely on the ordering of LOAD and STORE accesses of PM.

In parallel systems, *happens-before* relation [21] has been used to define a partial order of concurrent events coming from different threads or processes. However, it differs from our *must-persist-before* relation. The reason is because there are three distinct problems: (a) program order within a sequential program, (b) concurrency control between parallel threads, and (c) data persistency during power loss and recovery. These problems reside in three orthogonal dimensions, in the sense that they may be either separated or combined during program analysis. To the best of our knowledge, our method is the only one for inferring PM-specific properties using counterfactual reasoning.

3 Overview of Our Method

In this section, we first propose our unified graph-theoretic view of PM properties and then present our top-level procedure.

3.1 The Unified View of PM properties

The seemingly isolated DURA, MPB, and MPA properties mentioned in the previous section are all related to each other, if we take a certain graph-theoretic perspective. While this unified view may seem intuitive (as shown by the remainder of this subsection), to the best of our knowledge, it has not appeared elsewhere in the literature.

At the center of this perspective is the MPB relation, which we consider to be the most fundamental building block. In contrast, DURA may be viewed as a special case of MPB, where the second member of the MPB relation is an imaginary program statement modeling program end. MPA may be viewed as a property implied by a set of (otherwise-conflicting) MPB relations.

3.1.1 The Directed Graph. First, we define a directed graph $G = (V, E)$ to represent the set of MPB relations. In this graph, the nodes in V correspond to program statements, while the edges in E correspond to the MPB relations. Specifically, $MPB(st, st')$ is represented by an edge from node st to node st'.

3.1.2 From MPB to DURA. Next, we define DURA as a special case of MPB, i.e., $DURA(st) := MPB(st, st_{last})$, where st_{last} is an imaginary program statement modeling the program end; that is, st_{last} is executed after all other statements in the program. Therefore, $MPB(st, st_{last})$ means that STORE st must persist eventually.

3.1.3 From MPB to MPA. Finally, we show how MPA may be implied by a set of MPBs. Recall that all the MPBs are already represented as edges in the graph G. If there is a strongly connected component (SCC) in G, the corresponding MPBs would represent a set of conflicting requirements (the circular MPBs require a STORE to persist before itself). The only way to reconcile these conflicting requirements is to put all of the involved STOREs in a persistent transaction. The persistent transaction guarantees that these STOREs take effect atomically.

3.1.4 Another Running Example. Consider the example program in Figure 3, which inserts a node in a doubly-linked list. It differs from the singly-linked list example in Figure 2 in the sense that both the `prev` and the `next` pointers are used. As mentioned earlier, following the `next` pointers, MPBs will be generated from left to right. Following the `prev` pointers, MPBs will also be generated from right to left. As a result, there will be two sets of MPBs conflicting with each other, unless the involved STOREs persist atomically, as shown in subfigure (b). This leads to the inference of the MPA shown in subfigure (c), together with the DURA and MPB relations.

3.2 The Top-Level Procedure

Algorithm 1 shows the top-level procedure of our method. The input consists of the program P and a set of test cases for running the program. The output is a set of inferred properties, stored in the tuple $\langle DURAs, MPBs, MPAs \rangle$.

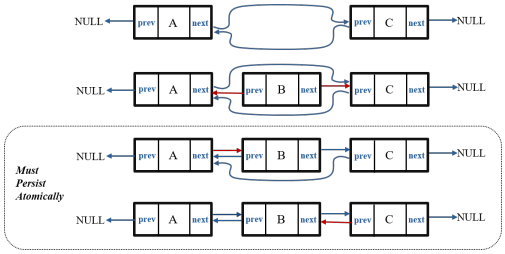
Algorithm 1: Inferring PM Properties.

```
1 Traces ← InstrumentedExecution (P, TestCases)
2 MPBs ← { }
3 foreach T ∈ Traces do
4 |   MPBs ← MPBs ∪ Infer_MPB_Requirements (T)
5 end foreach
6 DURAs ← Infer_DURA_Requirements (MPBs)
7 MPAs ← Infer_MPA_Requirements (MPBs)
8 return ⟨DURAs, MPBs, MPAs⟩
```

Our procedure goes through three steps. First, it conducts static analysis of the program P to compute the dependence relations and then instruments the executable program to add the *self-logging* capability, which means that running the instrumented program with the test cases produces a set of execution traces (Line 1). Next, our procedure conducts dynamic analysis of the execution traces, one at a time, to infer the MPB relations (Lines 2-5). Finally, the

```
1   B->data = data;
2   B->next = A->next;
3   B->prev = C->prev;
4   //CLFLUSHOPT(&B->data,sizeof(int));
5   //CLFLUSHOPT(&B->next,sizeof(Node*));
6   //CLFLUSHOPT(&B->prev,sizeof(Node*));
7   //SFENCE();
8   //TX_BEGIN();
9   //TX_ADD(&A->next);
10  //TX_ADD(&C->next);
11  A->next = B;
12  C->prev = B;
13  //TX_END();
```

(a) persistent memory program



(b) correct persistent order

```
//DURA:
   (B->data=...@line 1), (B->next=...@line 2),
   (B->prev=...@line 3), (A->next=...@line 11),
   (C->prev=...@line 12)
//MPB:
   (B->data=...@line 1, A->next=...@line 11)
   (B->next=...@line 2, A->next=...@line 11)
   (B->prev=...@line 3, A->next=...@line 11)
//MPA:
   (A->next=...@line 11, C->prev=...@line 12)
   ...
```

(c) PM properties

Figure 3: Inserting a node to a *doubly-linked list* stored in persistent memory (PM). To ensure the correct persistent order shown in subfigure (b), the TX_BEGIN(), TX_ADD(), and TX_END() statements in subfigure (a) are needed. Otherwise, after power failure (which may strike at any moment), the PM state may be inconsistent.

inferred MPB relations, stored in *MPBs*, are used to infer the DURA and MPA relations (Lines 6-7).

Algorithm 2: Checking PM Properties.

```
1 Bugs ← { }
2 foreach T ∈ Traces do
3 |   PT ← Compute_PersistentTime_Intervals (T)
4 |   Bugs ← Bugs ∪ Check_DURA_Requirements (T, PT, DURAs)
5 |   Bugs ← Bugs ∪ Check_MPB_Requirements (T, PT, MPBs)
6 |   Bugs ← Bugs ∪ Check_MPA_Requirements (T, PT, MPAs)
7 end foreach
8 return Bugs
```

Once the PM properties are inferred, they may be used in various downstream software engineering tasks. One task explored in this work is to improve PM bug detection, where the inferred PM properties are used as test oracles for detecting property violations. This trace-based analysis technique is shown in Algorithm 2. For each trace $T \in Traces$, we traverse the events in T to compute a persistent time interval $PT[ev]$ for every STORE event ev (Line 3). Then, we leverage the computed PT to detect violations of DURA, MPB, and MPA properties (Lines 4-6).

In the next two sections, we present our detailed algorithms for inferring and checking PM properties.

4 Inferring PM Properties

We first explain our method for static analysis and trace generation, then provide the intuition behind our method for inferring PM properties, and finally present the detailed algorithm.

4.1 Static Analysis and Trace Generation

Our method starts with static analysis of the program to compute the dependence relations. Our static analysis is implemented using the LLVM compiler to leverage its existing APIs for program analysis. LLVM first constructs a program dependency graph for the program, and then uses it to compute the control- and data-dependencies. However, these dependencies are restricted to individual functions; this is because by default LLVM only performs intra-procedural analysis. We have extended LLVM to perform inter-procedural analysis.

Next, we instrument the dependence relations into the executable program, to add the *self-logging* capability. That is, while the instrumented program is executed, it generates an execution trace annotated with the dependence relations, thus providing all of the information our method needs to infer and check PM properties. Specifically, the dependence relations are defined over the program statements, and are stored in a map which takes two program statements st' and st as input and returns *true*, for example, if $CDEP(st', st)$ holds.

While the instrumented program generates the execution trace, it associates each event ev with a field $ev.st$ to represent the program statement that generates the event. Thus, for any two LOAD events (ev' and ev) in the execution trace, we can quickly check whether $CDEP(ev'.st, ev.st)$ holds. It is worth noting that multiple events may be generated by the same program statement, e.g., if the statement is executed more than once. It is also worth noting that the execution trace only contains instructions related to PM LOAD and STORE accesses; DRAM related instructions are ignored.

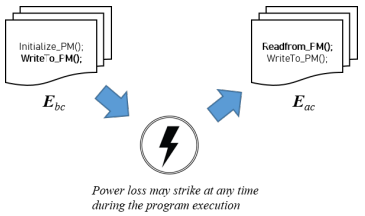
Let $T = ev_1, \ldots, ev_n$ be an execution trace in $Traces$, where each event ev_i has the following fields in addition to $ev_i.st$:

- $ev_i.type$, which may be type LOAD, STORE, CLFLUSHOPT, SFENCE, TX_BEGIN, TX_END, and TX_ADD.
- $ev_i.addr$, which is the starting address for LOAD, STORE, CLFLUSHOPT, and TX_ADD.
- $ev_i.size$, which is the corresponding size for LOAD, STORE, CLFLUSHOPT, and TX_ADD.

Other PM related CPU instructions, such as CLWB (cache line write back) and CLFLUSH (unoptimized version of CLFLUSHOPT) for the *Px86* architecture, may be modeled by CLFLUSHOPT and SFENCE.

4.2 The Intuition behind Our Method

Our method traverses the execution trace $T \in Traces$ to infer the MPB relations first. To understand the intuition behind MPB inference, it is helpful to take a look at the typical life cycle of a PM program, shown in Figure 4. The classic example of a PM program is shown on the right-hand side, which starts by checking a flag to see if it needs to initialize PM for the very first time (Line 4) or recover from power loss by reading data already stored in PM (Line 6). After that, the program proceeds to normal operation, which may write to PM.




```
1  int main() {
2    ...
3    if (!recovering)
4      Initialize_PM();
5    else {
6      ReadFrom_PM();
7    ...
8    WriteTo_PM()
9    ...
10 }
```

Figure 4: The life cycle of a PM program executed before and after power loss (which may strike at any time) and recovery.

With this in mind, we can look at the figure on the left-hand side of Figure 4, where a power loss induced program crash occurs between two executions of the program. We call the execution before the crash E_{bc}, and call the execution after the crash E_{ac}. Note that there may be causal relations between STOREs in E_{bc} and the corresponding LOADs in E_{ac}.

Thus, we regard a PM property as a *contract* between these two executions. Furthermore, our counterfactual reasoning relies on the following observation: ***Unless STOREs in E_{bc} satisfy the contract demanded by LOADs in E_{ac}, LOADs in E_{ac} can definitely go wrong***. Based on this observation, our method focuses on analyzing LOADs in E_{ac}, to derive the contract, and then leveraging the contract to infer MPBs over STOREs in E_{bc}. Conceptually, this is accomplished in the two steps below.

4.2.1 Step 1. Identifying the Contract from LOADs in E_{ac}. In this step, we search for two LOAD events in E_{ac}, denoted ev_{ld} and ev'_{ld}, such that $CDEP(ev_{ld}.st, ev'_{ld}.st)$ holds, meaning ev_{ld} is control dependent on the preceding ev'_{ld}. Together, they demand the corresponding STOREs in E_{bc} to persist in a certain order.

As an example, consider the singly-linked list in Figure 2 where E_{ac} may be any function that reads the list from PM. Since the list nodes can only be traversed from left to right, the two LOADs may come from `{if(A->next!=NULL) node=A->next->next;}` since LOAD from `A->next->next` (which may be aliased to `B->next`) is control dependent on LOAD from `A->next`. Together, they demand that STORE to `A->next->next` always persists before STORE to `A->next`. The reason is because, otherwise, E_{ac} can definitely go wrong since `A->next->next` may return a stale value.

4.2.2 Step 2. Mapping the Contract to STOREs in E_{bc}. In this step, we search for any two matching STORE events, ev_{store} and ev'_{store}, such that $ev_{store}.addr = ev_{ld}.addr$ and $ev'_{store}.addr = ev'_{ld}.addr$. There may be multiple pairs of STOREs for each pair of LOAD events. For each pair of matching STOREs, we infer a property $MPB(ev_{store}.st, ev'_{store}.st)$.

Consider again the singly-linked list example in Figure 2, where E_{bc} may be any function that writes the list to PM, including the `insert()` function. More specifically, the two matching STORE events may come from Lines 2 and 6 of the example program in Figure 2 (a), where `A->next->next` is aliased to `B->next`. Thus, we infer `MPB(B->next=...@line2,A->next=...@line6)`.

While the intuition behind our two-step method for inferring MPBs from E_{bc} and E_{ac} may seem straightforward, we note that it is only a simplified mental picture for ease of presentation. In practice, there may be no clear separation between E_{bc} and E_{ac} in software code; instead, LOADs from PM and STOREs to PM may be mixed together, either within a function or being scattered in multiple functions. Furthermore, STOREs in one function may correspond to LOADs in multiple other functions, and vice versa. Therefore, having a fully automated technique for analysis and inference is important.

4.2.3 Causation versus Correlation. It is important to note that our method relies on the *causal relationship* between STOREs in E_{bc} and the matching LOADs in E_{ac}. This differentiates our method from existing invariant generation techniques based solely on *correlation*. Correlation does not always imply causation, and correlation without causation leads to unsound results.

As an example, consider again the `insert()` function for doubly-linked list in Figure 3. Based solely on the five STOREs shown in Lines 1-3 and 11-12, it is impossible to know with certainty that the two STOREs in Lines 11-12 must persist atomically, but the three STOREs in Lines 1-3 do not need to persist atomically. Specifically, just because two STOREs occur together does not imply that they must persist atomically.

To correctly infer the MPA property for these STOREs, we must analyze the matching LOADs, which are elsewhere in the code base. For the example in Figure 3, in particular, the matching LOADs are outside of the `insert()` function.

In general, even finding the matching LOADs for these STOREs is a difficult problem in program analysis. Our proposed method has made it easy, by generating annotated execution traces where concrete PM addresses associated with the STOREs and LOADs are made readily available. Thus, given any STOREs in $T \in Traces$, finding their matching LOADs in T becomes easy.

4.3 The Inference Algorithm

Depending on the type of PM properties, i.e., whether they are MPBs, DURAs or MPAs, our inference algorithm uses different subroutines.

4.3.1 Inferring the MPBs. Algorithm 3 shows our subroutine for inferring MPBs, which takes an execution trace $T \in Traces$ as input and returns a set of MPBs as output. Internally, it goes through two steps. First, it traverses LOAD events in T to find every pair (ev_{ld}, ev'_{ld}) such that $CDEP(ev_{ld}, ev'_{ld})$ holds. These pairs go into a set named $Contracts$. Then, the subroutine traverses STORE events in T to find the matching pair such that $ev_{store}.addr = ev_{ld}.addr$ and $ev'_{store}.addr = ev'_{ld}.addr$. Each of these matching pairs must satisfy an MPB property.

4.3.2 Inferring DURAs from MPBs. While generating the execution trace T, we add an imaginary LOAD event ev_{first} at the start of the trace, and assume that all subsequent LOAD events are control dependent on it. We also add an imaginary STORE event ev_{last} at the end of the trace such that $ev.st = st_{last}$, representing the program end. With these additions, DURAs are inferred as special cases of MPBs. Thus, inside Algorithm 4, we only need to take them out of $MPBs$ and put them into $DURAs$.

4.3.3 Inferring MPAs from MPBs. Algorithm 5 shows our subroutine for inferring MPAs from MPBs. Internally, it first represents the

Algorithm 3: Infer_MPB_Requirements(T).

```
Contracts ← { }
foreach ev_ld ∈ T and ev'_ld ∈ T do
    if CDEP(ev_ld.st, ev'_ld.st)) then
        Contracts ← Contracts ∪ { (ev_ld, ev'_ld) }
    end if
end foreach
MPBs ← { }
foreach ev_store ∈ T and ev'_store ∈ T do
    foreach (ev_ld, ev'_ld) ∈ Contracts do
        if ev_store.addr = ev_ld.addr ∧ ev'_store.addr = ev'_ld.addr then
            MPBs ← MPBs ∪ { MPB(ev_store.st, ev'_store.st) }
        end if
    end foreach
end foreach
return MPBs
```

Algorithm 4: Infer_DURA_Requirements($MPBs$).

```
DURAs ← { }
foreach MPB(st, st') ∈ MPBs where st' = st_last do
    DURAs ← DURAs ∪ { DURA(st) }
    MPBs ← MPBs \ { MPB(st, st') }
end foreach
return DURAs
```

set of MPBs using a directed graph, and then computes the strongly-connected components (SCCs). Next, for each SCC, it creates an MPA by including every statement st involved in the SCC. At a high level, the MPBs inside each SCC represent a set of conflicting requirements. The conflict can only be resolved by including all these STOREs in a persistent transaction.

Algorithm 5: Infer_MPA_Requirements($MPBs$).

```
MPAs ← { }
let G be the graph where each MPB(st, st') is an edge from st to st'
let SCCs be the set of strongly-connected components in G
foreach SCC ∈ SCCs do
    MPA ← {st | st is a node in SCC }
    MPAs ← MPAs ∪ {MPA}
end foreach
return MPAs
```

4.4 The Correctness Guarantee

To a limited extent, our method guarantees correctness of the inferred properties, i.e., when the control and data dependence relations are precise. This is captured by the following theorem.

Theorem 4.1. *With precisely-computed dependence relations, our inferred PM properties are* necessary conditions *for the program to behave correctly through potential power loss and recovery. With over-approximated dependence relations, they are likely program invariants.*

The reason why this theorem holds is because, based on counterfactual reasoning, our method can always find a concrete execution trace $T \in Traces$ showing that, if an inferred property (over the STOREs in T) is violated, the program (more specifically LOADs of the program in T) can definitely go wrong due to inconsistent PM content.

Precisely computing dependence relations is possible during dynamic analysis, e.g., using Valgrind [28]; in fact, it is routinely performed during taint tracking. However, it is computationally expensive since it requires monitoring each and every instruction in the program (not just PM instructions).

For a more practical implementation, we decide to use LLVM's static dependency analysis, which is orders-of-magnitude fast, but the results are over-approximated, meaning the inferred properties are likely program invariants. Nevertheless, our experimental evaluation (in Section 6) shows that the inferred PM properties are almost always valid in practice.

While the inference of PM properties may be done entirely with static analysis, in practice, it can be challenging for static analysis to precisely compute the causal relationships between STOREs in E_{bc} and LOADs in E_{ac} due to aliased expression. For example, when `A->next` is alised to `B` in the software code, `A->next->next` refers to `B->next`; but when `A->next` is aliased to `C`, `A->next->next` refers to `C->next`. How to combine *fast* static techniques and *accurate* dynamic techniques to precisely compute the dependence relations is a research problem that we leave for future work.

It is also worth noting that, during the experimental evaluation, we use the default unit tests of the benchmark programs to generate program execution traces, from which our method infers the PM properties. Thus, the quality of the inferred properties depend on the quality of the test suite. While incomplete test suite may lead to *missing properties*, it may never lead to *wrong properties*, as stated in the above theorem. The reason is that, for each inferred property, our method will find a concrete execution trace showing that, if the property is violated, the program can definitely go wrong due to some inconsistent data stored in PM.

5 Checking PM Properties

To detect violations of the inferred properties, we first traverse the execution trace to compute the persistent time interval for each STORE event ev, denoted $PT[ev]$, and then check it against the DURA, MPB, and MPA properties.

5.1 Persistent Time Intervals

The notion of a *persistent time interval* [25] is as follows. For a given STORE event ev, the time when it takes effect in PM is represented by a window $[from, to]$. The window starts from $ev.time$, which is the time when the STORE instruction is executed, and ends after the corresponding CLFLUSHOPT and SFENCE instructions are executed. If any of the corresponding CLFLUSHOPT and SFENCE instructions is missing, the window ends after the imaginary $ev_{last}.time$, which is when the program ends.

As an example, consider the STORE events $ev_1 - ev_3$ corresponding to Lines 1-3 of the program in Figure 3. Since the program has only 13 lines of code, we can set the imaginary $ev_{last}.time = 14$. Assuming that all the green-colored CLFLUSHOPT and SFENCE instructions are included in the program, we have $PT[ev_1] = [1, 7]$ meaning the STORE may take effect at any time between Line 1 and Line 7. In other words, $PT[ev_1].from = 1$ and $PT[ev_1].to = 7$. Similarly, we have $PT[ev_2] = [2, 7]$ and $PT[ev_3] = [3, 7]$.

Consider the STORE events $ev_{11} - ev_{12}$ corresponding to Lines 11-12 of the program in Figure 3. Since these STORE events are

protected by a persistent transaction using `TX_ADD()`, the persistent time intervals will be computed differently. Specifically, since the transaction starts at Line 8 and ends at Line 13, we have $PT[ev_{11}] = [8, 13]$ and $PT[ev_{13}] = [8, 13]$. Since the two intervals have the same start time and end time, we say $PT[ev_{11}] = PT[ev_{13}]$.

Assuming for now that all persistent time intervals have been computed and stored in the map PT, we present our property checking algorithm. We will present the algorithm for computing the persistent time intervals at the end of this section.

5.2 The Checking Algorithm

Algorithm 6 shows our subroutine for checking durability properties. It traverses each execution trace $T \in Traces$ and, for each STORE event ev such that $DURA(ev.st) \in DURAs$, checks if the persistent time is always less than $ev_{last}.time$. If $PT[ev].to < ev_{last}.time$ does not hold, it reports a DURA violation. Note that $PT[ev].to = +\infty$ if the corresponding CLFLUSHOPT and SFENCE are missing.

Algorithm 6: CHECK_DURA_REQUIREMENTS($T, PT, DURAs$).

```
1 Bugs ← { }
2 foreach STORE event ev ∈ T do
3     if DURA(ev.st) ∈ DURAs then
4         if ¬(PT[ev].to < ev_last.time) then
5             Bugs ← Bugs ∪ {DURA}
6         end if
7     end if
8 end foreach
9 return Bugs
```

Algorithm 7 shows our subroutine for checking MPB properties. It traverses each execution trace $T \in Traces$ and, for any two STORE events such that $MPB(ev.st, ev'.st) \in MPBs$, meaning that $ev.st$ must persist before $ev'.st$, checks the persistent time intervals $PT[ev]$ and $PT[ev']$. If $PT[ev].to < PT[ev'].from$ does not hold, it reports an MPB violation.

Algorithm 7: CHECK_MPB_REQUIREMENTS($T, PT, MPBs$).

```
1 Bugs ← { }
2 foreach pair of STORE events ev ∈ T and ev' ∈ T do
3     if MPB(ev.st, ev'.st) ∈ MPBs then
4         if ¬(PT[ev].to < PT[ev'].from) then
5             Bugs ← Bugs ∪ {MPB(ev.st, ev'.st)}
6         end if
7     end if
8 end foreach
9 return Bugs
```

Algorithm 8 shows our subroutine for checking MPA properties. It traverses each execution trace $T \in Traces$ and, for any two STORE events ev, ev' appearing in the same MPA, checks the persistent time intervals $PT[ev]$ and $PT[ev']$. If $PT[ev] = PT[ev']$ does not hold, it reports an MPA violation.

5.3 Computing Persistent Time Intervals

Algorithm 9 shows the subroutine for computing persistent time intervals [25]. It takes an execution trace T as input and returns the map PT as output. For each STORE event $ev \in T$, the corresponding $PT[ev] = (from, to)$ represents a time window.

Algorithm 8: CHECK_MPA_REQUIREMENTS($T, PT, MPAs$).

```
1 Bugs ← { }
2 foreach STORE event ev ∈ T and STORE event ev' ∈ T do
3     if ev.st ∈ MPA and ev'.st ∈ MPA and MPA ∈ MPAs then
4         if ¬(PT[ev.st] = PT[ev'.st]) then
5             Bugs ← Bugs ∪ {MPA}
6         end if
7     end if
8 end foreach
9 return Bugs
```

Algorithm 9: COMPUTE_PERSISTENT_INTERVAL(T).

```
1  PT ← {(ev, [−∞, +∞])} ;              // map ev to a time window
2  transaction ← False ;
3  cached_stores ← list() ;          // STOREs outside of transaction
4  tx_stores ← set() ;                  // STOREs inside transaction
5  tx_begin_time ← 0 ;
6  tx_add_addrs ← set() ;
7  foreach event ev ∈ T do
8      if ev.type = STORE then
9          PT[ev].from ← ev.time ;                  // set start time
10         if ¬transaction then
11             cached_stores.add(ev)
12         else
13             tx_stores.add(ev)
14     else if ev.type = CLFLUSHOPT then
15         foreach ev' ∈ cached_stores and ev'.addr = ev.addr do
16             PT[ev'].to ← ev_last.time+1 ;              // set end time
17         end foreach
18     else if ev.type = SFENCE then
19         foreach ev' ∈ cached_stores and PT[ev'].to = ev_last.time+1 do
20             PT[ev'].to ← ev.time ;              // finalize end time
21             cached_stores.remove(st)
22         end foreach
23     else if ev.type = TX_BEGIN then
24         transaction ← True
25         tx_begin_time ← ev.time
26     else if ev.type = TX_ADD then
27         tx_add_addrs.add(ev.addr)
28     else if ev.type = TX_END then
29         transaction ← False
30         foreach ev' ∈ tx_stores where ev'.addr ∈ tx_add_addrs do
31             PT[ev'].from ← tx_begin_time ;    // set start time
32             PT[ev'].to ← ev.time ;              // set end time
33         end foreach
34         tx_stores.clear()
35         tx_add_addrs.clear()
36 end foreach
37 return PT
```

Internally, the subroutine starts by setting $PT[ev]$ to $[-\infty, +\infty]$ for all events. Then, it traverses the events in T sequentially and, depending on the event type, updates the persistent time intervals for the STORE events. At a high level, there are two cases depending on whether a STORE is inside a persistent transaction.

The first case is when the STORE is outside of the persistent transaction (Lines 8-22). In this case, the STORE event is held temporarily in the list named *cached_stores* until it is persisted in PM. This takes three steps, by first setting the $from$ field of the time window when STORE is executed, then setting the *to* filed when SFENCE is executed, and finally changing the *to* field when SFENCE is executed.

The second case is when the STORE is inside the persistent transaction (Lines 23-35). In this case, the STORE event is held

Table 1: Statistics of the benchmark programs (Columns 1-3) and execution traces (Columns 4-9).

Program	LoC	Description	Version	# ST	# LD	# FL	# FE	# TR
cc_slist	2698	singly linked list [30]	bad	166	244	103	64	19
			ok	166	258	102	90	26
cc_list	3208	doubly linked list [30]	bad	1466	3022	401	264	270
			ok	1555	3443	401	300	285
cc_array	2171	array [30]	bad	102	344	130	44	0
			ok	102	344	84	75	20
cc_stack	612	stack [30]	bad	27	84	24	8	0
			ok	27	84	13	13	4
cc_deque	2575	double end queue [30]	bad	181	565	201	104	0
			ok	181	471	57	26	57
cc_queue	576	queue [30]	bad	124	303	128	48	0
			ok	124	271	68	20	24
cc_pqeque	618	priority queue [30]	bad	70	367	0	0	0
			ok	70	367	117	63	0
cc_ringbuf	316	ring buffer [30]	bad	144	443	117	99	0
			ok	144	443	118	118	3
redis	83864	PM Redis [17]	default	367	558	8	8	N/A
memcached	27331	PM Memcached [23]	default	230	1043	3	3	N/A

temporarily in the set named tx_stores until it is persisted in PM. This takes a single step, by setting both $from$ and to fields of the time window when TX_END is executed.

6 Experiments

We have implemented our method in a software tool that leverages the Clang/LLVM [22] compiler to parse the C code of a PM program, compute the dependence relations, and instrument the executable program. Running the executable program with existing test cases will produce annotated execution traces, which are what our method needs to infer and check PM properties. We implement our algorithms for inferring and checking PM properties in Python, by taking the execution traces as input and returning the inferred properties and the detected violations as output, respectively.

6.1 Benchmark Programs

The benchmark programs used in our evaluation fall into two sets. The first set consists of eight persistent data structures [30], implementing various lists, queues, and ring buffers. These data structures come with unit test cases, which we have used to run the instrumented programs to generate execution traces. The second set consists of two PM-enabled distributed storage applications: Redis [17] and Memcached [23]. The execution traces are generated from the servers interacting with clients.

Table 1 shows the statistics of of each program, including the name, the number of lines of code, and a short description of the application in Columns 1-3. Column 4 shows the program version. Each data structure has two versions: one good version, e.g., `cc_list-ok`, in which PM instructions have been properly added and used; and one bad version, e.g., `cc_list-bad`, in which some PM instructions are missing or misused. For Redis and Memcached, only the default version is used. Columns 5-9 show the statistics of the execution traces, including the number of STORE (ST), LOAD (LD), CLFLUSHOPT (FL), SFENCE (FE) events and persistent transaction (TR) blocks.

6.2 Research Questions

Our experiments were designed to answer the following three research questions (RQs):

RQ 1: Can our method infer PM properties automatically?
RQ 2: Are PM properties inferred by our method useful?
RQ 3: Are PM properties inferred by our method of high quality?

To answer RQ 1, we have applied our method to all of the benchmark programs. Our experiments were conducted on a computer with AMD Ryzen 5 5600X CPU and 32GB memory on Ubuntu 20.04 with DAX emulation for Persistent Memory.

To answer RQ 2, we have leveraged the PM properties inferred by our method to detect PM bugs, and compared its performance against the state-of-the-art PM bug detection technique described in [25]. To answer RQ 3, we have compared the quality of our PM properties with those inferred by the state-of-the-art technique described in [10]. To ensure a fair comparison, we have taken the effort to implement both existing techniques within our software tool.

6.3 Experimental Results

We divide the results to three parts, one for each research question.

6.3.1 Results for RQ 1. We present the results for RQ 1 in Table 2, where Column 1 shows the benchmark name, Column 2 shows the time taken by our method to infer PM properties, and Columns 3-5 show the number of inferred DURA, MPB, and MPA properties, respectively. In total, our method took 7.4 seconds to infer 386 DURA properties, 280 MPB properties, and 18 MPA properties. The results show that our method can infer PM properties quickly and automatically.

We have manually inspected all these properties and confirmed that they are correct. Furthermore, for the data structure benchmarks, regardless of which program version (ok or bad) is used, the inferred properties are almost always the same. The only exceptions are `cc_list-bad/ok` and `cc_deque-bad/ok`, which have different MPBs but the difference is small. This is consistent with our expectation, because our inference method focuses on *what PM properties should be enforced* and not on *whether they have been correctly enforced*. In other words, our method relies on causal relations between LOADs and STOREs in the execution traces, and is not affected by whether CLFLUSHOPT, SFENCE, and transaction have been used correctly.

6.3.2 Results for RQ 2. We present the results for RQ 2 in Columns 6-13 of Table 2. Specifically, Columns 6-9 show the time taken by our method to detect property violations, and the number of DURA, MPB, and MPA violations detected. Columns 10-13 show the time taken by the state-of-the-art technique [25], and the number of DURA, MPB, and MPA violations detected. In total, our method took 7.8 seconds to detect 79 DURA violations, 67 MPB violations, and 9 MPA violations. In contrast, the existing technique only detected 79 DURA violations, but no MPB or MPA violations. This result is consistent with our expectation because, when the existing technique is not accompanied by manually specified properties, it can only detect the simplest type of bugs, i.e., DURA bugs.

A closer look at the results shows that, for the *ok* versions of data structures, our method does not report any violation, which is

Table 2: Evaluating our method for inferring properties and checking these properties for violations.

Program Version	Our Method (infer)				Our Method (check)				Existing Method [25] (check)			
	Time (s)	DURA	MPB	MPA	Time (s)	DURA bugs	MPB bugs	MPA bugs	Time (s)	DURA bugs	MPB bugs	MPA bugs
cc_slist-bad	0.1	20	9	0	0.1	6	3	0	0.1	6	0	0
cc_slist-ok	0.1	20	9	0	0.1	0	0	0	0.1	0	0	0
cc_list-bad	2.4	44	45	3	2.5	5	10	1	2.4	5	0	0
cc_list-ok	2.8	44	39	3	3.0	0	0	0	2.9	0	0	0
cc_array-bad	0.1	16	15	1	0.1	3	9	1	0.1	3	0	0
cc_array-ok	0.1	16	15	1	0.1	0	0	0	0.1	0	0	0
cc_stack-bad	0.1	12	1	0	0.1	1	1	0	0.1	1	0	0
cc_stack-ok	0.1	12	1	0	0.1	0	0	0	0.1	0	0	0
cc_deque-bad	0.1	21	14	1	0.1	10	10	1	0.1	10	0	0
cc_deque-ok	0.1	21	13	1	0.1	0	0	0	0.1	0	0	0
cc_queue-bad	0.1	15	10	2	0.1	2	5	2	0.1	2	0	0
cc_queue-ok	0.1	15	10	2	0.1	0	0	0	0.1	0	0	0
cc_pqeque-bad	0.1	9	2	0	0.1	9	2	0	0.1	9	0	0
cc_pqeque-ok	0.1	9	2	0	0.1	0	0	0	0.1	0	0	0
cc_ring_buf-bad	0.1	14	8	0	0.1	2	3	0	0.1	2	0	0
cc_ring_buf-ok	0.1	14	8	0	0.1	0	0	0	0.1	0	0	0
redis	0.4	54	66	3	0.4	13	18	3	0.3	13	0	0
memcached	0.4	30	13	1	0.5	30	6	1	0.4	30	0	0
Total	7.4	**386**	**280**	**18**	7.8	**79**	**67**	**9**	7.4	**79**	**0**	**0**

consistent with expectation, because these properties are enforced by the execution traces. For the *bad* versions of data structures, as well as the default version of Redis and Memcached applications, we have manually inspected the detected violations and confirmed that they are real violations.

While the existing method [25] represents the state-of-the-art, it has the same drawback as other PM bug detection tools. That is, the intended PM properties must be manually specified before violations of these properties can be detected. However, it is a challenging task for developers to manually specify PM properties. For example, even for the small code snippets in Figures 2 and 3, correctly specifying the PM properties is not easy. That is exactly where our contribution is – to automatically discover these PM properties (which specify the objects of a program that should be made persistent, and the order).

Note that, while Redis and Memcached has large code sizes, not all program statements are PM related. Since we only record PM related events in the execution traces and skip DRAM related events, analyzing these traces does not take a long time. PM programs use different memory allocators for different types of memory, e.g., `malloc()` for DRAM and `PMmalloc()` for PM. We use LLVM to instrument the calls to PM-specific memory allocator. At run time, we decide if a LOAD/STORE operation is for PM by checking if the concrete memory address has been allocated by the PM-specific memory allocator.

6.3.3 Results for RQ 3. There are two sets of results for RQ 3. We present the first set in Table 3, which compares the quality of our PM properties with those inferred by an existing technique [10]. While both methods inferred 386 DURA properties, our method inferred significantly fewer MPB properties (280 vs. 1641) and MPA properties (18 vs. 35). We have manually checked the inferred properties and found that many of the MPBs and MPAs inferred by the existing method are not correct.

For some benchmarks, the number of MPBs inferred by the existing method is more than 10 times, e.g., for `cc_stack` and `Memcached`. This is also due to its unsound heuristic rules for generating MPBs. Furthermore, unlike our method, which infers MPAs from conflicting MPBs, the existing method uses yet another set of unsound heuristics rules to infer MPAs. As a result, although the set of MPBs inferred by our method is a strict subset of their MPBs, the MPAs are not as clearly related as the MPBs.

For example, for the data structures, our method inferred MPAs only for `cc_list`, `cc_array`, `cc_deque`, and `cc_queue`. This result is consistent with expectation because only circular MPBs may lead to MPAs, but `cc_slist` (a singly-linked list) does not have circular MPBs, and thus should not have MPAs. However, using unsound heuristic rules, the existing method generated bogus MPAs even for data structures like `cc_slist`.

Finally, we present the remaining experimental results for RQ 3 in Table 4, which compares the detected violations by using the inferred properties as test oracles. Since the two property inference methods lead to the same DURA violations, this table only compares the MPB and MPA violations. Both methods reported 67 MPB violations. However, the existing method missed 8 MPA violations in `cc_array-bad`, `cc_deque-bad`, `cc_queue-bad`, Redis, and Memcached. As for the bogus violations, our method reported none, but the existing method reported 27 bogus MPB violations and 23 bogus MPA violations.

The higher quality of our inferred properties as shown in Table 4 is not surprising because our method focuses on *what PM properties should be enforced* instead of *whether these properties are correctly enforced*. Furthermore, our method will not report the property (or the violation) unless there exists a concrete execution trace that can serve as evidence, based on our counterfactual reasoning. That is, if the inferred property (over PM STOREs) is violated, the program execution (more specifically PM LOADs) can definitely go wrong, due to the use of some stale data stored in PM.

7 Related Work

Our method is the first one for inferring PM properties using a combination of static and dynamic analysis techniques together with counterfactual reasoning. With precisely computed control

Table 3: Comparing the properties inferred by our method and the properties inferred by an existing method [10].

Program Version	Our Method (infer)			Existing[10] (infer)		
	DURA	MPB	MPA	DURA	MPB	MPA
cc_slist-bad	20	9	0	20	81	2
cc_slist-ok	20	9	0	20	81	2
cc_list-bad	44	45	3	44	245	5
cc_list-ok	44	39	3	44	217	5
cc_array-bad	16	15	1	16	50	1
cc_array-ok	16	15	1	16	48	1
cc_stack-bad	12	1	0	12	13	0
cc_stack-ok	12	1	0	12	12	0
cc_deque-bad	21	14	1	21	110	5
cc_deque-ok	21	13	1	21	121	5
cc_queue-bad	15	10	2	15	37	2
cc_queue-ok	15	10	2	15	34	2
cc_pqeque-bad	9	2	0	9	14	0
cc_pqeque-ok	9	2	0	9	12	0
cc_ringbuf-bad	14	8	0	14	23	1
cc_ringbuf-ok	14	8	0	14	23	1
redis	54	66	3	54	357	2
memcached	30	13	1	30	177	1
Total	**386**	**280**	**18**	**386**	**1641**	**35**

Table 4: Comparing the detected violations using properties inferred by our method and an existing method [10].

Program Version	Real Violations				Bogus Violations			
	Ours		Existing[10]		Ours		Existing[10]	
	MPB	MPA	MPB	MPA	MPB	MPA	MPB	MPA
cc_slist-bad	3	0	3	0	0	0	2	2
cc_slist-ok	0	0	0	0	0	0	0	2
cc_list-bad	10	1	10	1	0	0	0	2
cc_list-ok	0	0	0	0	0	0	0	2
cc_array-bad	9	1	9	0	0	0	0	1
cc_array-ok	0	0	0	0	0	0	0	1
cc_stack-bad	1	0	1	0	0	0	0	0
cc_stack-ok	0	0	0	0	0	0	0	0
cc_deque-bad	10	1	10	0	0	0	1	5
cc_deque-ok	0	0	0	0	0	0	0	3
cc_queue-bad	5	2	5	0	0	0	1	2
cc_queue-ok	0	0	0	0	0	0	0	1
cc_pqeque-bad	2	0	2	0	0	0	0	0
cc_pqeque-ok	0	0	0	0	0	0	0	0
cc_ringbuf-bad	3	0	3	0	0	0	0	1
cc_ringbuf-ok	0	0	0	0	0	0	0	1
redis	18	3	18	2	0	0	18	0
memcached	6	3	6	0	0	0	6	1
Total	67	**11**	67	**3**	**0**	**0**	**27**	**23**

and data dependence relations, it guarantees that the inferred properties are correct. Even in a practical implementation that sacrifices the accuracy of dependency analysis for efficiency, it can still infer high-quality properties, as confirmed by our experimental evaluation. In this sense, it is better than techniques that use unsound heuristic rules [10], which belong to a larger body of work on discovering likely invariants, including Daikon [6, 7] and its follow-up work [2, 3, 8, 31].

Our method is related to existing techniques for detecting PM bugs [4, 10–14, 24, 33]. Besides PMTest [25], which is a tool that provides a flexible user interface to take in a wide range of PM properties and can check them for violations, Jaaru [13] is a technique for detecting certain types of persistency bugs such as missing flushes with model checking. Pmfuzz [24] is a recent technique built upon AFL++[9] by incorporating PM related heuristics to generate test cases for PM programs.

There are also techniques for detecting PM bugs in concurrent software. For example, both PMRace [4] and Yashme [14] are able to detect PM bugs caused by thread interleaving, while DURINN [11] is a tool for checking *durable linearizability*, an API-level correctness criterion for concurrent data structures. While these existing PM bug detection techniques are related, they do not directly help programmers specify PM properties.

Another line of related work is automated PM program repair. Hippocrates [26] is a technique for repairing durability (DURA) bugs; it first searches for known syntactic bug patterns and then applies predefined program transforms. PMBugAssist [16] is a more general technique that relies on SMT solver based symbolic program analysis to search for repairs. Therefore, it works for previously unknown bug patterns, and can handle both DURA and MPB violations. However, automated repair techniques still require correctness specifications, and our method for inferring PM properties can provide such specifications.

We infer PM properties using a combination of static and dynamic analyses. Similar *trace-based analysis* techniques have been used in other applications, e.g., for diagnosing concurrency bugs [1, 20, 34] and analyzing side-channel leaks [15]. In some of these cases, symbolic reasoning techniques based on SMT solvers have been used to amplify the coverage of dynamic analysis techniques, e.g., as in symbolic predictive analysis [35].

In general, all dynamic program analysis techniques including ours share a limitation: they require high-quality test cases to generate execution traces as input. One way to overcome this limitation is using automated testing to diversify the test cases and hence the execution traces. Another way to overcome this limitation is using static techniques to verify the dynamically inferred invariants, e.g., as in Nimmer et al. [29]. While static techniques may also directly generate true invariants [36], for PM related properties, we are not aware of such prior work. Nevertheless, these are interesting research problems that we leave for future work.

8 Conclusion

We have presented a method for inferring PM properties from existing software code. Our method relies on a combination of static and dynamic analysis techniques, where static analysis is used to compute dependence relations and instrument the executable program, and dynamic analysis of the program's execution traces is used to infer and check PM properties. Our method leverages a unified graph-theoretic perspective and counterfactual reasoning to generate high-quality PM properties. Our experimental evaluation on eight persistent data structures and two distributed storage applications shows that the method can infer PM properties quickly and automatically. We also demonstrate the usefulness of the inferred PM properties, by leveraging them to significantly improve the performance of a state-of-the-art PM bug detection technique.

Acknowledgments

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

References

[1] Mitra Tabaei Befrouei, Chao Wang, and Georg Weissenbacher. 2016. Abstraction and mining of traces to explain concurrency bugs. *Formal Methods Syst. Des.* 49, 1-2 (2016), 1–32. https://doi.org/10.1007/S10703-015-0240-5

[2] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. 2014. Inferring models of concurrent systems from logs of their behavior with CSight. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 468–479. https://doi.org/10.1145/2568225.2568246

[3] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, Tibor Gyimóthy and Andreas Zeller (Eds.). ACM, 267–277. https://doi.org/10.1145/2025113.2025151

[4] 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*, Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch (Eds.). ACM, 873–887.

[5] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

[6] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 2001. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Trans. Software Eng.* 27, 2 (2001), 99–123. https://doi.org/10.1109/32.908957

[7] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. 2000. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000*, Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf (Eds.). ACM, 449–458. https://doi.org/10.1145/337180.337240

[8] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1-3 (2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015

[9] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, Yuval Yarom and Sarah Zennou (Eds.). USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[10] 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*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 100–115.

[11] 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.

[12] João Gonçalves, Miguel Matos, and Rodrigo Rodrigues. 2023. Mumak: Efficient and Black-Box Bug Detection for Persistent Memory. In *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan (Eds.). ACM, 734–750. https://doi.org/10.1145/3552326.3587447

[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*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). 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*, Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch (Eds.). ACM, 830–845.

[15] 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)*, Einar Broch Johnsen and Manuel Wimmer (Eds.). Springer, 163–183. https://doi.org/10.1007/978-3-030-99429-7_10

[16] Zunchen Huang and Chao Wang. 2024. Constraint Based Program Repair for Persistent Memory Bugs. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 91:1–91:12. https://doi.org/10.1145/3597503.3639204

[17] Intel. 2021. A version of Redis that uses persistent memory. https://github.com/pmem/pmem-redis.

[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. Persistent Memory Development Kit (PMDK). https://https://pmem.io/pmdk/.

[20] Markus Kusano, Arijit Chattopadhyay, and Chao Wang. 2015. Dynamic Generation of Likely Invariants for Multithreaded Programs. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 835–846. https://doi.org/10.1109/ICSE.2015.95

[21] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563

[22] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[23] Lenovo. 2018. Lenovo modifications to Linux memcached for enhanced persistent memory support. https://github.com/lenovo/memcached-pmem.

[24] 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*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 487–502. https://doi.org/10.1145/3445814.3446691

[25] 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*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 411–425.

[26] 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*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 401–414. https://doi.org/10.1145/3445814.3446694

[27] 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. https://www.usenix.org/conference/osdi20/presentation/neal

[28] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation.. In *PLDI*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 89–100. http://dblp.uni-trier.de/db/conf/pldi/pldi2007.html#NethercoteS07

[29] Jeremy W. Nimmer and Michael D. Ernst. 2001. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Workshop on Runtime Verification, RV 2001, in connection with CAV 2001, Paris, France, July 23, 2001*. 255–276. https://doi.org/10.1016/S1571-0661(04)00256-7

[30] Srđan Panić. 2023. Collections-C: A library of generic data structures. https://github.com/srdja/Collections-C.

[31] Jeff H. Perkins and Michael D. Ernst. 2004. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2004, Newport Beach, CA, USA, October 31 - November 6, 2004*, Richard N. Taylor and Matthew B. Dwyer (Eds.). ACM, 23–32. https://doi.org/10.1145/1029894.1029901

[32] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2020. Persistency semantics of the Intel-x86 architecture. *Proc. ACM Program. Lang.* 4, POPL (2020), 11:1–11:31.

[33] 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*, Jaejin Lee, Kunal Agrawal, and Michael F. Spear (Eds.). ACM, 322–336.

[34] 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*, Satnam Singh, Barbara Jobstmann, Michael Kishinevsky, and Jens Brandt (Eds.). IEEE, 99–108.

https://doi.org/10.1109/MEMCOD.2011.5970516

[35] 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)*, Ana Cavalcanti and Dennis Dams (Eds.). Springer, 256–272. https://doi.org/10.1007/978-3-642-05089-3_17

[36] Jingbo Wang and Chao Wang. 2022. Learning to Synthesize Relational Invariants. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 65:1–65:12. https://doi.org/10.1145/3551349.3556942