Understanding and Dealing with Hard Faults in Persistent Memory Systems

Brian Choi
Johns Hopkins University

Randal Burns
Johns Hopkins University

Peng Huang
Johns Hopkins University

Abstract

The advent of Persistent Memory (PM) devices enables systems to actively persist information at low costs, including program state traditionally in volatile memory. However, this trend poses a reliability challenge in which multiple classes of *soft* faults that go away after restart in traditional systems turn into *hard (recurring) faults* in PM systems. In this paper, we first characterize this rising problem with an empirical study of 28 real-world bugs. We analyze how they cause hard faults in PM systems. We then propose Arthas, a tool to effectively recover PM systems from hard faults. Arthas checkpoints PM states via fine-grained versioning and uses program slicing of fault instructions to revert problematic PM states to good versions. We evaluate Arthas on 12 *real-world* hard faults from five large PM systems. Arthas successfully recovers the systems for all cases while discarding 10× *less* data on average compared to state-of-the-art checkpoint-rollback solutions.

ACM Reference Format:
Brian Choi, Randal Burns, and Peng Huang. 2021. Understanding and Dealing with Hard Faults in Persistent Memory Systems. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–28, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3447786.3456252

1 Introduction

Modern systems maintain a vast amount of state. Persisting state allows important information to be saved across runs. Yet, persistence has costs, so developers have to make a careful decision on what program state to persist and when. For performance, many systems store a substantial amount of (or all) state in volatile DRAM, which will be lost or have to be re-constructed upon restart. The advent of low-latency, byte-addressable Persistent Memory (PM) alleviates this tension and enables developers to persist much more information at low cost [39]. For example, developers can store previously volatile cache data structures in PM, which allows applications to restart quickly with a warm cache [48].

From a reliability perspective, however, PM leads to an emergent challenge we call the *soft-to-hard fault transformation* problem. Without PM, process restart is often a simple but effective method to mitigate production failures because the issues are "soft"[1], *i.e.*, they only affect volatile state and go away upon restart. But the integration with PM can cause buggy state to persist after restart, *i.e.*, becoming "hard" faults.

The soft-to-hard fault transformation produces classes of bugs that have much more severe effects in PM systems than the same bugs in traditional systems. For example, a rare race condition or CPU bit flip in traditional systems can corrupt a volatile pointer and cause a system crash upon dereference, but the crash would disappear after restart. In PM systems, this fault could cause repeated crashes. As another example, a memory leak that previously would be cleaned up after restart could become permanently leaked space in PM.

While hard faults also occur in traditional systems if a bad state is written to storage (e.g., a file) and retrieved after restart, this problem is particularly acute in PM systems. This is because, developers can afford to store much more state in persistent memory, and/or persist states more frequently. As a result, PM systems are much more susceptible to the danger of bad persistent state.

Prior works [23, 25, 38, 43, 45, 46, 50, 57, 65] have extensively investigated the crash consistency problem in PM systems, which is caused by partial or reordered PM writes leading to inconsistent state *after* crashes. The hard fault problem significantly impacts PM system reliability with a variety of common bugs *during* the program executions. Unfortunately, the problem has received less attention, nor has it been formally classified.

To understand this emergent problem's characteristics, we first conduct an empirical study. We collect 8 real-world hard fault bugs from five new PM systems. One challenge is that existing new PM systems do not have a rich development history yet and have a limited number of reported issues. Fortunately, there are growing interests in adapting [43] and porting [17, 22, 30, 48, 61] mature systems that have long histories to PM. We thus propose a forward-looking methodology that uses historical bugs to study their effect in the ported

[1] We adopt the terms from a classic paper [36] by Jim Gray, which conjectures that most production software faults are soft (transient) while most "hard" faults (always fail on retry) in software are eliminated before production.

EuroSys '21, April 26–28, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8334-9/21/04…$15.00
https://doi.org/10.1145/3447786.3456252

PM versions. We collect 20 real-world bugs in Redis and Memcached and reproduce them in their persistent versions.

We summarize the common practices of persisting program state (Section 2.2) that may contribute to the rise of the hard fault problem. We then analyze individual collected cases. We find that the root causes of the studied cases are diverse (6 types). Even after multiple restarts, these bugs cause the PM systems to continue to experience deterministic failures, or even worse deteriorating failures. In addition, the majority (68%) of the cases involve bad state propagation across volatile and persistent variables in different code locations.

When a PM system experiences a hard failure in production, it is critical to get the system functional as soon as possible. For hard faults, however, typical mitigation actions such as process restart or node reboot will not make the failure symptom disappear. Developers may then be forced to shutdown the system, debug the issue offline, pinpoint the root cause, fix the bug, and re-deploy the system to production. Unfortunately, the challenge remains in this scenario—even after the code bug is found and fixed, the underlying bad persistent states still exist, so the system will likely fail again in the re-deployment. Essentially, to deal with hard failures, we need a solution to restore the PM system state to a point in which the system is operational again.

We propose a tool, Arthas, to help users systematically restore PM systems from hard failures, during either online failure mitigation or offline diagnosis and fixing. Guided by our study, we observe that when persistent memory objects are assigned with bad values the persistence point (root cause) often occurs long before the failure point, and the bad states may have already been propagated to multiple volatile and persistent variables along the execution flow. Properly recovering from a hard failure must revert these bad persistent values. We design Arthas to use checkpoint/rollback of dependent states that carefully eliminates bad values in the PM.

The central challenge Arthas addresses is to ensure recoverability while minimizing the amount of discarded data due to rollback. Current checkpoint-rollback solutions operate on coarse-grained, point-in-time memory snapshots. They are not effective in recovering PM systems. Even if the PM system is temporarily recovered, the hard failures could occur soon because contaminated states are not properly rolled back. In addition, these solutions tend to incur significant data loss by rolling-back states that are independent of the failure, *e.g.*, they would undo many successful key/value operations that are independent of the failure.

To address this challenge, Arthas designs PM-aware fine-grained check-pointing by versioning PM states at program memory object level. To accurately identify "just enough" PM states to roll back for a failure, Arthas takes a novel approach with static analysis and lightweight runtime tracing techniques. In particular, the Arthas analyzer computes the Program Dependency Graph (PDG) of a PM system offline. It further instruments tracing code to the target system, which will emit PM addresses and the instruction sources at runtime. Upon detecting a suspected hard fault, the Arthas reactor leverages the PDG, fault instruction, and PM trace to compute the set of PM states related to the failure. Based on this information, Arthas reverts only dependent bad PM states.

	CCEH	Dash	PMEMKV	LevelHash	RECIPE	Memc.	Redis
Cases	1	1	2	2	2	9	11
Type	New	New	New	New	New	Port	Port

Table 1. Collected hard fault bugs in new and ported PM systems.

For evaluation, we reproduce 12 hard fault bugs from five large PM systems (Redis, Memcached, Pelikan, PMEMKV and CCEH). Arthas successfully mitigates the failures in all cases. In achieving the recoverability, Arthas only discards an average 3.1% of the PM state updates. For comparison, we enhance the state-of-the-art process-level checkpointing solution, CRIU [1], to support PM checkpointing. CRIU mitigates 9 cases deterministically and 2 cases with some probability. It discards an average of 56.5% of the persistent states. Arthas only incurs up to 4.8% runtime overhead.

The main contributions of this work are:

- We study the emergent hard fault problem in PM systems.
- We propose a novel solution, Arthas, to effectively mitigate hard faults in PM systems while minimizing the data loss.
- We evaluate Arthas with hard fault bugs in real-world PM systems to demonstrate its effectiveness.

2 Empirical Study on PM Hard Faults

We present an empirical study for the hard fault problem in PM systems. We aim to understand the characteristics of hard faults and leverage the insights to guide our solution design.

2.1 Definition and Study Methodology

In this paper, we define a hard fault to be "bad" values in persistent memory that originate from software bugs or transient hardware errors and cause the PM systems to experience recurring failures across runs. By "bad", we mean a PM location holding a value that is incorrect and affects subsequent program behavior (trigger crash, assertion, leak, wrong result, etc.). If a PM location holds an incorrect value but it does not influence subsequent program behavior, it is a benign fault.

We examine several new PM systems [3, 40, 43, 47, 66, 68] to search for bugs related to PM programming and check if the bug leads to hard faults. One challenge is that most new PM systems only have a few reported bugs due to their short development history. We have to inspect the commit messages to complement the bug trackers. We found 8 hard fault bugs from five new PM systems (Table 1).

To enrich our study dataset, we observe the growing interests from developers and researchers in porting mature systems to support PM. New bugs can be introduced in the ported versions. In addition, mature systems have a rich development history with various bugs. Some of the historical bugs affect the volatile variables or data structures that are now stored in PM. Were these bugs re-introduced in the PM

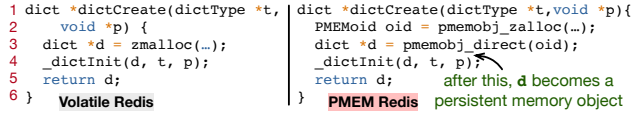


Figure 1. Original Redis vs. PMEM-Redis.

versions, they can potentially result in hard faults. We study two widely-used, in-memory key-value stores, Redis [13] and Memcached [5]. There are several efforts that port them to support PM [44, 48, 50, 59], including the effort from the official Memcached developers [30]. We manually go through bug trackers of the two systems and inspect existing bugs that affect PM related code in the ported versions. In total, we collect 11 bugs from Memcached and 9 bugs from Redis, and reproduce these bugs on the persistent versions.

2.2 Volatile vs. Persistent States

We first summarize the common practices used in making program state persistent in the studied systems.

Usually, traditional systems only persist critical data or adopt in-memory designs for performance. For example, all states in Redis by default are volatile. Redis does provide a feature [15] that allows users to take periodic snapshots to disks. But even with this feature, only the key-value items are written to disk infrequently.

PM enables developers to save not only critical data but also various "progress" state. More active persistence allows a PM system to restart quickly and pick up the progress.

One category of program state that is usually stored in DRAM in traditional systems but can be stored in PM is auxiliary data structures. Take key-value stores as an example. Their indexing structures like hashtables are usually *not* persisted, because persisting their updates hurts performance. Instead, upon restart, the hashtables are re-constructed by re-inserting all items into newly allocated hashtables. A PM implementation can keep the hashtables in persistent memory besides the items. This reduces the recovery time because the system can directly use the hashtables upon restart.

Another category is cache-related data structures. For example, in the RocksDB PM version [17], developers persist the LRU cache. Volatile states may also be persisted as a by-product. This occurs because persistent and volatile objects can have complex dependencies [48]. For example, in N-Store [20], by persisting the core table structure, developers inevitably end up with persisting several other transient data structures due to their dependencies [10]. Developers may also put volatile states in PM to avoid complexity. For example, in the persistent Memcached [30], developers directly persist the entire `item` data structure for the purpose of simplicity, even though it includes some fields that one could classify as "transient". Last but not least, we observe in several cases in which developers also persist queues or asynchronous processing data structures.

Adding PM persistence is natural because the PM programming model is similar to DRAM's. The snippet in Figure 1 shows how the original Redis and PMEM-Redis create the `dict` object. After line 3, the two versions are indistinguishable. We also observe some studied systems add PM persistence through using a global persistent memory allocator or a memory mapped file in PM device. This makes it easy to use PM since any program objects allocated or referring to the mapped region will be backed by PM. But the implication is that a large portion of a system's code may touch or interact with persistent objects and introduce bad values.

2.3 Bug Examples

We explain several bug cases in detail, and then describe the general patterns in the following sections.

CCEH directory doubling bug. CCEH [51] is a new dynamic hashing scheme designed for PM. It has a bug reported by the authors of RECIPE [43]. CCEH doubles the size of its directory structure when the number of buckets grows large, which modifies several pieces of metadata. If an untimely crash occurs before the global depth is updated, the insertion operations will get stuck in an infinite loop. We reproduced this bug and confirmed the failure recurs across restarts. Just fixing the bug does not make the previous failure symptom go away. We must also correct the bad persistent metadata.

PMEMKV asynchronous free leaks. PMEMKV [3] is PM key-value database designed by Intel and supports multiple storage engines. When handling the client delete requests, for performance, PMEMKV sets the hashtable to be unlinked and uses an asynchronous thread to free the key-value items later. The bug will cause a persistent memory leak if PMEMKV crashes before the asynchronous thread frees the objects [12].

Memcached corrupt hashtables. Memcached tracks reference counts for key-value items. It has a bug [7] where the refcount is incremented without checking for integer overflow. Memcached checks for any items with refcount 0 and frees them. It assumes that items with refcount 0 have been unlinked from the hashtables. Since the overflow prematurely sets the refount to 0 *without* unlinking this item, its address is still in the hashtables. If the same key is reinserted, it will reclaim the same memory address. As a result, there are two key-value items of the same address in the hashtable. The item's hashtable next pointer will point to itself. Upon a GET request, Memcached gets stuck in an infinite loop:

```
while (it) {
  if (!memcmp(key, it->key, n)) return it;
  it = it->h_next; // next point to itself
}
```

In the original version, the key-value item and the hashtables were volatile. So the bad refcount and contaminated hashtable are gone after restart. But in the PMEM-Memcached, developers would naturally persist the entire `item` structure [8] including an item's refcount. Even if the refcount is not persisted, the corrupt hashtables are persisted. Consequently, Memcached would hang again after restart.

Redis listpack buffer overflow. Redis' listpack structure stores lists of encoded strings. The encoding function has

a logic error bug [14] that results in buffer overflow for listpacks of size larger than 4096 bytes. Even though the listpack insertion request succeeds, the listpack size value gets corrupted. Subsequently, when a client makes a request to read that listpack, Redis encounters a segmentation fault. In the original Redis with default options, the corrupt listpack will be gone after the restart. However, in a PMEM Redis, since the corrupt listpack is immediately written to PM, Redis will experience repeated segfaults after restart.

2.4 Root Causes

We categorize the studied hard faults by their root causes.

Logic Errors. These bugs involve temporary variables that are assigned with problematic values. In regular systems, such variables are volatile, and restart typically recovers the system. In PM systems, the bad values can get persisted and incur permanent failures. An example is a bug [6] in Memcached where calling `flush_all` at a future time incorrectly begins removing valid items in the LRU before the time comes.

Buffer Overflow. These are bugs in which the system receives an unexpected input and misbehaves. This includes the Redis listpack issue described earlier, which causes corrupt items. In the original version, the unexpected input usually only affects volatile variables. In the PM version, some persistent variables can be affected as well, *e.g.*, the bogus length is persisted, and introduce persistent faults.

Concurrency Bugs. These issues are known for behaving non-deterministically. That is, they often disappear in re-executions. So restart is usually an effective approach to mitigate them. But in persistent systems, race conditions or resource contention can lead to permanent wrong results or deadlocks when locks or shared variables are made persistent.

Hardware Faults. Defects in hardware can corrupt program variables. This can occur in CPU (bitflip), DRAM, or network messages. Typically such hardware faults are rare and transient, *i.e.*, after restart or retry the faults will be gone. With persistent memory, the affected system states can remain corrupted even after restart. The Memcached hashtable expansion flag discussed previously is one such example.

Memory Leaks. Volatile heap objects that are not properly freed can lead to performance degradation. But after restart, the memory leak is gone. In PM systems, however, the memory leaks will be permanent, causing a cascading amount of performance and storage issues after each run.

Figure 2 shows the root cause distribution. We can see that logic errors make up the largest percentage (46%). The second largest contributor (18%) are race condition errors.

2.5 Bug Consequences

We analyze the consequences of these studied bugs. Figure 3 categorizes the outcomes and shows the frequencies of occurrences for each consequence. For ported PM systems, all bugs cause more severe impact than the consequences in the original versions, such as repeated crashes, permanent corruption, persistent leak, and repeated hang. Repeated crashes are the most common (32%) consequence.

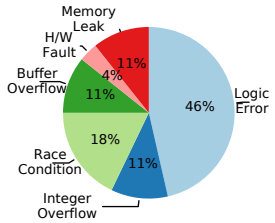


Figure 2. Root cause of studied persistent failures.

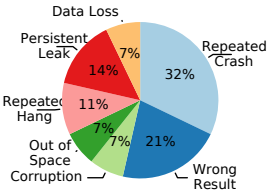


Figure 3. Consequence of studied persistent failures.

Another interesting comparison is PM versus disk-based persistence. Take Redis and the listpack bug (Section 2.3) as an example. Redis allows users to configure periodic snapshots to disk. If users enable this feature, Redis will store its listpacks to files. The bug consequence will then depend on the snapshot frequency: if Redis crashes before the snapshot of corrupt listpack, no future crash will occur; if Redis crashes after the snapshot, the segfault will recur (assuming the `GET` request is still issued). In other words, even with this persistence option, vanilla Redis can still nondeterministically avoid this problem. However, in the PM version of Redis, this failure will *deterministically* recur after restart.

2.6 Fault Propagation Patterns

Besides root causes and consequences, we also classify these bugs into three different programming patterns based on which type of memory the root cause resides in and how the fault propagates through the system during runtime:

- Type I: A program variable backed by persistent memory has bad value that directly causes the failure. Example: corrupt persistent pointer resulting in segmentation fault.
- Type II: A program variable or data structure gets assigned a bad value. The bad value propagates across the system and indirectly causes the system failure. Some variable along the propagation chain is written to persistent memory. Example: an incorrect persistent flag causes the program to unexpectedly execute a function and deadlock.
- Type III: Persistent variables cause undesirable behavior, *not* due to bad values, but because of other programming mistakes. Example: forget to free persistent objects and lead to persistent memory leak.

Among the 20 bugs we studied, the majority (68%) of them are of Type II, i.e., they involve bad state propagation. 18% of them belong to Type I, *i.e.*, some bad persistent state directly causes system failures. The remaining 14% are of Type III.

2.7 Implications

Our study shows that several classes of bugs would incur much more severe consequences in PM systems than the same bugs in systems without PM. Even after multiple restarts, a PM system continues to experience the same failure or even deteriorating failures. These issues occur in new PM systems as well as in traditional systems adapted to PM.

This motivates the need for more effective solutions other than process restart to mitigate failures for production PM systems. Our insight from the study is that, despite its diverse root causes and manifestations, the essence of the soft-to-hard fault problem is that some volatile states with bad values are now persistent. Mitigating these faults thus requires getting rid of the bad persistent states in the PM systems.

A natural approach is to checkpoint PM states and rollback them upon failures. However, traditional checkpoint solutions are not PM-aware and could not effectively rollback PM hard faults. In addition, they typically perform periodical, coarse-grained snapshots of the entire memory, which would incur significant data loss during rollback. Moreover, as Section 2.6 shows, the majority of hard failures involve bad state propagation among volatile and persistent variables in different functions. Consequently, even when one bad PM state is rolled back, the PM system could still quickly hit the same failure if the root cause of the bad state is not reverted. The Memcached refcount overflow bug is such an example. Even if the Memcached states are reverted to prior to creation of the problematic persistent hashtable, it soon encounters the same infinite loop because the bad `refcount` still exists.

3 Overview of Arthas

Based the insights from our study in Section 2, we propose a PM-aware, fine-grained checkpoint-rollback mechanism and design a toolchain called Arthas to aid PM systems recover from hard faults. The design goals of Arthas are: (1) to bring a PM system experiencing hard faults back to normal quickly, (2) to minimize the amount of data discarded during the rollback, and (3) to incur small runtime overhead.

3.1 Basic Idea and Workflow

The basic idea of Arthas is to checkpoint PM state updates at variable/address level and record the program data flow. The latter is used to accurately identify which PM states are affected during a failure for effective rollback with minimal data loss. However, tracking such information dynamically can be expensive. To address this challenge, Arthas takes a novel approach that uses program analysis to statically analyze the PM system's data dependencies, and then applies lightweight address tracing and checkpointing at runtime.

Figure 4 shows an overview of the Arthas workflow. The *analyzer* takes input the source code of a target PM system and performs static analysis. It is responsible for two tasks. First, it identifies program variables that may potentially reside in persistent memory. For potential persistent variables, the analyzer instruments a lightweight tracing API call in the program that will record the memory addresses of these variables at runtime. Second, the analyzer computes a Program Dependence Graph (PDG) to track the dependency information. This static PDG will later be used in the rollback process. Arthas also provides a *checkpointing library* to *transparently*

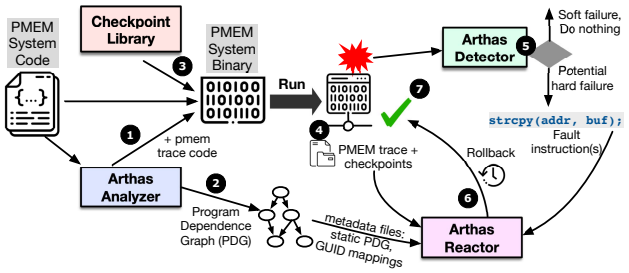


Figure 4. Overview of Arthas design.

checkpoint persistent memory updates in the target PM system. Arthas links the checkpoint library and tracing code to the target system and produce an enhanced executable.

As the enhanced PM system executes, it emits the PM address trace, and Arthas versions PM state updates in a persistent checkpoint region. Arthas *detector* monitors the PM system and determines if a potential hard failure occurs. If so, the detector will identify the faulty instruction and invoke the *reactor* component. The reactor analyzes the faulty instruction, computes the dependent instructions based on the static PDG and dynamic PM address trace. The reactor then computes a list of states that should be reverted and the order. This rollback plan is presented to operator for inspection. When confirmed, Arthas automatically executes this plan.

3.2 Supported PM Systems

Arthas currently supports two kinds of PM systems: (1) systems written with the popular PMDK framework [11]; (2) systems written with persistence instructions such as `clwb` and `sfence`. For PMDK-based systems, we particularly support two PMDK libraries—`libpmem`, which provides low-level primitives such as `pmem_persist`, `pmem_map _file`, etc., and `libpmemobj`, which provides high-level transaction interfaces.

These two types—library persistence and native persistence [50]—cover many existing PM systems. We choose PMDK to represent the library approach because it is widely-used, and most systems we study use it, including the PM-ports of Redis and Memcached. Our static analysis scheme, checkpoint, and rollback designs are not limited to PMDK.

To support additional PM frameworks such as NVHeaps [28], Mnemosyne [62] and Memkind [9], Arthas needs to intercept specific calls in order to enact checkpointing. Arthas's underlying checkpoint data structure design and functions are the same across different frameworks. Therefore, the main effort of incorporating such frameworks would involve identifying which functions/instructions in a framework should be intercepted. The main functionalities to consider are: (a) PM initialization or allocation function, (b) flush/fence/commit function, (c) free function, (d) misc functions such as re-alloc. Typically, PM frameworks only provide a handful of interfaces. Thus, the support effort is small.

3.3 Use Cases and Limitations

Arthas activates in response to a persistent fault in which the system becomes inoperable. In this case, *some state/data loss*

is inevitable to recover the system, and the amount inherently depends on the specific fault and how the fault propagates dynamically in the specific system execution. Arthas performs a more fine-grained recovery than traditional checkpoint-rollback solutions to minimize the state/data loss.

Arthas' approach is suitable for PM applications that have many concurrent clients. A driving concern for them is that a fault arising in handling one client can cause loss of state updates from many other concurrent clients. Arthas leverages static analysis to isolate the dependent operations from independent operations and avoid unnecessary state reversions. In contrast, a traditional checkpoint/rollback approach reverts that entire state of the system. In doing so, it will lose many independent state changes from concurrent clients.

Example applications that have to manage concurrent clients and avoid state-loss impact on innocent clients include directory services, web apps that store user state and preferences, recommendation systems, content-distribution networks, and feature storage for machine learning. Clients of these applications conduct individual read and write requests with the expectation that reads are repeatable and writes are persistent. Most clients rely on the state persistence for correctness. Arthas's recovery minimizes the state loss damage to the many concurrent clients that (1) have completed and are no longer available or (2) do not implement recovery logic for completed operations. Thus, it allows the target system to maintain repeatable reads and persistence for as many completed operations as possible.

Arthas focuses on the analysis and recovery within a single component. It is unable to detect or respond to external dependencies. Consequently, it do not match the needs of distributed PM applications. It may also introduce inconsistency if a PM application has a client that makes multiple semantically related requests that Arthas cannot detect as dependent. However, Arthas's ability to isolate the impact on other clients and minimize state loss may still be desirable.

4 Arthas Design

In this Section, we describe the design details for the core Arthas components as shown in Figure 4.

4.1 Static Analysis and Instrumentation

Locating PM Variables and Instructions. The Arthas analyzer scans the entire PM system code to systematically identify instructions that create or access program variables backed by PM. We build the dataflow analysis on top of the LLVM framework [42]. The analyzer first locates instructions that call APIs of common PM libraries. If these API calls return a value that is stored in a variable, e.g., `pmemobj_create`, `pmemobj_direct`, we identify the initial set of PM variables.

For PM systems using low-level persistence instructions like `clwb`, the analyzer extracts PM variables from the instruction operands. The analyzer also supports identifying PM variables from mmapped-style APIs such as `pmem_map_file`. We treat the return value variables (e.g., `ptr=pmem_map_file(...)`) as well as variables whose values come from the return variables (e.g., `fptr = ptr+10;`) as PM variables.

Starting from these initial PM variables, the analyzer computes the transitive closure of all instructions that use the PM variables using the *def-use* chain analysis and add the variables into the PM variable/instruction set. In real-world PM systems, persistent variables often involve pointers and are passed across functions. To properly handle such cases, our analysis is inter-procedural and incorporates field-sensitive, context-sensitive pointer alias analysis [64].

Instrumenting Tracing Code (❶). Once the instructions that create or access PM variables are identified, we would ideally track their dependencies with other instructions during execution using techniques like dynamic taint analysis [54, 58]. However, such tracking can be intrusive and adds significant overhead to the PM systems. Instead, Arthas's approach is to compute dependencies statically and enhance the static dependencies with lightweight runtime information.

The most critical piece of information Arthas needs from the system runtime execution is the dynamic addresses of the PM variables and program locations of the associated instructions. To get this information, the Arthas analyzer assigns a Globally Unique Identifier (GUID) for each identified PM instruction and generates a metadata file that records the mappings of `<GUID,source_location,instruction>`. Arthas then instruments an API call to a lightweight runtime tracing library just before the PM instruction. The tracing code will emit a trace of `<GUID,pmem_address>` at runtime. To reduce the tracing overhead, we inline the tracing code, buffer address traces in memory, and asynchronously flush the traces to a file when the buffer is full or the system stops execution.

Constructing Program Dependence Graph (❷). To tackle the challenge of complex bad state propagation (Section 2.7), Arthas statically analyzes the dependency information by computing a Program Dependence Graph (PDG) [32]. Nodes in the PDG are LLVM IR instructions and edges represent dependencies among the instructions. We analyze two types of dependencies. Data dependencies represents data flow relationship. Control dependencies occur when an instruction determines if another instruction should be executed or not. Our dependency analysis is inter-procedural and flow-sensitive. It handles pointers, including function pointers (the callgraphs contain edges to potential targets in the points-to set).

The PDG and the GUID mappings are static metadata of the target program. This metadata is stored in regular files. They will be read as input by the Arthas reactor later in production. As long as the target program code does not change, the dependency metadata is consistent with the application code.

4.2 Eager Checkpointing of PM States with Versioning

To rollback a PM system, we need to checkpoint PM states. Arthas performs checkpointing through *versioning* PM states

```
typedef struct {
  const void *address;
  uint64_t offset;
  int version;
  int data_type;
  void *data[MAX_VERSIONS];
  size_t sizes[MAX_VERSIONS];
  int seq_nums[MAX_VERSIONS];
  uint64_t old_entry;
  uint64_t new_entry;
} checkpoint_log_entry;
```

Figure 5. Checkpoint log entry definition.

at the granularity of program variables/addresses. This differs from existing solutions such as Flashback [60], CRIU [1] and ThyNVM [57] that perform periodic snapshots at a coarse granularity, e.g., entire process state or memory pages.

The Arthas checkpointing functionalities are implemented as a generic library. During the compilation phase, this library is linked with the target program (❸).

When the PM system starts, the checkpoint library initializes a checkpoint log in persistent memory. When the system performs an update to PM address `r` of size `n`, the library transparently stores the update into the checkpoint log based on `r` and records `n` (❹). Each log entry holds a maximum number (default 3) of versions of data for that address. In addition, the checkpoint library maintains an atomic **sequence number** to order PM updates by logical time, which is used during rollback. Figure 5 shows the checkpoint log entry.

The Arthas library performs checkpointing *eagerly* on each update rather than periodically. But it respects the program's persistence points to avoid premature checkpointing if the data is still in DRAM or partially persisted. It ensures so by intercepting all the well-defined durability APIs such as `pmemobj_persist`, `sfence`, which are called by the target program to ensure an address range is stored durably in PM. We only checkpoint when these calls succeed. Besides using explicit durability APIs, a PM system may use transaction APIs like those in the `libpmemobj` library to implicitly flush updates to PM at the end of a successful transaction. Such libraries already keep an undo log (or some form of logging/tracking) that records all update operations within the transaction. We modify the internal transaction commit function to copy each updated address range in the undo log to our checkpoint log.

In summary, both the granularity and the timing of our checkpointing are *consistent* with how the target system updates its persistent states and makes them durable in its code. For multi-threaded PM systems, we assume their persistence program points are properly synchronized, so Arthas would not checkpoint PM updates with data races.

The library does *not* checkpoint program states in volatile DRAM. PM systems usually reconstruct volatile states based on persistent states upon restart, which we rely on to handle volatile states. We could use Arthas together with traditional checkpointing techniques to checkpoint volatile states and reduce the state reconstruction time. But the rollback of persistent state must be done carefully together with volatile states rollback to respect their complex dependencies. Reconstructing volatile states is a simpler and more robust choice.

Some PM programs may resize a persistent memory block, which can present challenges during reversion. We add a field `old_entry` in the checkpoint log entry to connect entries

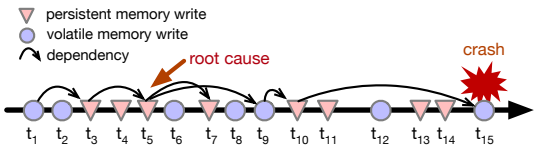


Figure 6. Timeline of volatile and persistent memory writes.

that have reallocation relationship. Our technical report [27] describes the solution in more detail.

4.3 Detecting Hard Failures

The Arthas detector monitors the target PM system in production for various failures such as crash, assertion failure, hang or memory leak (❺). It also supports user-defined checks (*e.g.,* inserted key-value items exist). When a failure is detected, the detector use heuristics to assess whether the issue might be a potential hard failure. It retrieves the faulting instruction, exit code, stack trace, memory usage, etc., and compares if the symptom is similar to the previously recorded failure (e.g., having the same exit code, fault instruction, loosely the same stack trace). The heuristics are imperfect and may flag issues that turn out to be a soft failure or are not caused by bad PM states. This does not cause problems because Arthas reactor is able to prune these false alarms in the mitigation stage (§ 4.5).

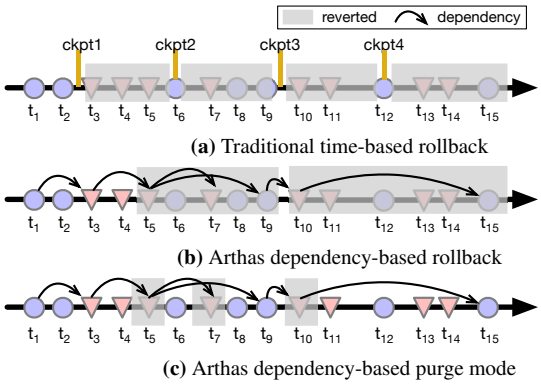
4.4 Rollback Strategies

Once the detector suspects a potential hard failure, Arthas invokes the reactor to recover the system (❻). The reactor reverts PM states at fine granularity. It also leverages the PDG from the static analyzer to guide the mitigation.

Arthas takes a dependency-based rollback approach, while the traditional rollback approach is typically time-based. Figure 6 shows an abstract example of the volatile and persistent memory write timeline in a crash, which we use to compare the rollback approaches. The arrows represent data dependencies among the updates. The immediate crash site is at time t_{15} and involves a volatile variable, *e.g.*, `strcpy(addr, buf);`, but the root cause is a buggy update to a persistent variable at time t_5 that later tainted the volatile variable `addr` or `buf`.

Note that the root cause point is unknown to a rollback solution. It has to be deduced post-failure. In addition, a rollback solution judges and reverts buggy states from an operational point of view—if it recovers a system to be operational again, it may not further search for those benign buggy states that do not influence system execution success.

Time-based vs. Dependency-based Rollback. In traditional time-based checkpoint-rollback (Figure 7a), program states including persistent memory states are checkpointed at periodic intervals. When the failure occurs, the program states are reverted to the previous checkpoint, i.e., `ckpt4` at time t_{12}. Unfortunately, the program will crash again because the volatile `addr` or `buf` are again assigned by the bad persistent value at time t_{10}. The program is further reverted to `ckpt3` at time t_{10}. While in this snapshot, the bad persistent value at t_{10} is gone, the failure is not over because the bad value at t_5



(a) Traditional time-based rollback

(b) Arthas dependency-based rollback

(c) Arthas dependency-based purge mode

Figure 7. Three rollback strategies for Figure 6.

pollutes t_{10} and then `addr` or `buf` again. So the program has to be reverted to `ckpt2` and then finally `ckpt1`.

In Arthas's dependency-based rollback (Figure 7b), the Arthas reactor analyzes the updates that a program variable depends on and reverts by following the dependency chain. It finds that the crash site t_{15} depends on t_{10} and thus directly reverts the program states back to be before t_{10}. This rollback does not fix the failure, because t_{10} propagates the bad value from t_5. The reactor identifies t_{10} has a dependency on t_9, a volatile variable that in turn has a dependency on t_5. Thus, Arthas further reverts the program states to be before t_5.

Compared to the time-based rollback, the dependency-based rollback has two advantages. First, the dependency information enables it to jump to the functional point with fewer trials (e.g., 2 versus 4). Second, time-based rollback has to revert an entire checkpoint at a time, thus easily incurring unnecessary state loss (e.g., t_3 and t_4), whereas dependency-based approach can revert to a precise point.

Dependency-based Rollback vs. Purge. Besides dependency-based rollback, the Arthas reactor supports a more aggressive reversion operation–*purging*. Rollback strictly follows time order. If the system is reverted to time t_i, any state updates after t_i are reverted. In purging mode, the reactor only reverts selected PM objects to their old versions. Updates to other objects after the reverted time points may not be reverted if they do not depend on the selected objects. Figure 7c shows that the reactor identifies dependency between t_{15} and t_{10} and reverts t_{10} to the old value. While t_{11}, t_{13} and t_{14} occur after t_{10}, they do not create dependencies to t_{15} and thus are not reverted. Because t_{10} is not the root cause, the crash still recurs; the reactor then reverts dependent update from t_5.

Compared to dependency-based rollback (Figure 7b), purging has the advantage of reducing the amount of unnecessary state loss. But it may introduce inconsistencies. To reduce this risk, when operating in the purging mode, the Arthas reactor runs a second pass to identify whether there are *forward* dependencies from the reverted states and then purges them as well. For example, even though t_7 does not directly affect the crash site, after we revert the direct dependent update in t_5, the reactor analyzes that the t_5 influences t_7 and reverts t_7 as well to maintain the consistency between them.

In practice, we observe that for request-based server programs, such as key-value stores, the state dependencies are typically partitioned based on requests. The system failure is usually caused by a specific request that only pollutes states related to that request. In this case, purging removes only those bad states. In comparison, rollback may have to get rid of states from many other requests that are not responsible for or influenced by the fault.

4.5 Rollback Workflow

For a given fault instruction, the Arthas reactor first derives a reversion plan based on the recorded checkpoint log (§ 4.2), the static PDG (§ 4.1), the PM address trace file, and the static metadata file of trace GUIDs (§ 4.1). The reversion plan is a candidate list of sequence numbers in the checkpoint log that might need reversions. The reactor first computes the *backward slices* [63] of the fault instruction based on the PDG. A backward slice for an instruction A includes all instructions that may affect the values in A. We only retain instructions that have persistent variables operands.

With the backward slices, we use the dynamic address trace to check whether a node of a slice generates a PM update. If so, we find the checkpoint log entries that have the same PM address and add the sequence number (§ 4.1) of the log entries along with the slice node into a list. In the end, we apply a policy function to obtain the final candidate list of sequence numbers. Our default policy function sorts and de-duplicates the supplied sequence numbers. A more complex function could arrange the sequence numbers based on DFS or BFS order of the slices and enforce a maximum distance with the fault instruction to filter excessive sequence numbers.

It is possible that the final reversion plan ends up being empty. This could occur due to an inaccuracy from the detector (§ 4.3), *i.e.*, the failure is not caused by bad values in PM. The reactor then safely aborts and resorts to simple restart.

If the reversion plan is not empty, the reactor proceeds to reversions (❻). It keeps a current version `v` and tries the selected sequence number in order. To revert a sequence number `s`, the reactor copies the data from version `v-1` to the PM address associated with `s`. After reversion, the reactor invokes a *re-execution* script that re-runs the target program with the same arguments as the prior run. The re-execution status is checked to see if the PM fault has been fixed. If the failure persists, the reactor moves on to try reverting the next sequence number in the candidate list. When all sequence numbers in the list are reverted and the failure still persists, the reactor retries reversion to an older version `v-2` until the max versions are exhausted. For the purge mode, if re-execution triggers assertion checks in the target program's recovery function or a configurable maximum number of tries have reached, the reactor switches to the conservative rollback mode.

We make the design decision of multi-attempt rollback for the goal of minimizing data loss. Since we do not know where the root cause bad state is, we revert in a smaller granularity and check if the system is in a good state. If the root cause is not in that small segment, we would have to rollback further. Other traditional checkpoint-rollback solutions such as Rx [56] also perform rollback in multiple iterations. Another factor is that the inaccuracies in static analysis could cause a single-attempt rollback ineffective, multiple attempts would become necessary to address the inaccuracies.

4.6 Rollback Consistencies

As described in Section 4.2, the granularity and timing of Arthas checkpointing strictly respects the developers' choice of persistence granularity and timing in the PM program.

For example, for a persistent data structure `item` that has many fields, if developers choose to update one field of `item` and then call `pmem_persist` or `sfence`, Arthas will create a checkpoint entry just for that field. Or, if developers choose to update multiple fields of `item` and then call a durability API, Arthas will create one checkpoint entry for the persisted fields right after the durability API call.

Because of this, the rollback in Arthas preserves the update consistency of the target PM program. It will not revert the PM states to intermediately updated PM states. In addition, Arthas's reversion strategies carefully follow the dependency chains extracted from the PM program's source code. This ensures a sequence number (PM update) in the checkpoint will not be reverted before its dependent sequence numbers (PM updates). For PM programs that use transaction interfaces, Arthas's checkpoint library inserts special entries for transaction starts and commits. Later, if the Arthas reactor rollbacks some checkpoint entry within a transaction, the reactor will rollback other checkpoint entries in the transaction. This preserves the transaction-level consistency.

If a PM program is written in a buggy way, e.g., some PM objects were persisted individually when they should be persisted atomically, Arthas may rollback the program to a semantically inconsistent point. However, such inconsistencies are *not* caused by the rollback but the program bugs. Such bugs are exactly one common root cause for the hard failure. To Arthas, if the reactor reverts a program to such a buggy point, the PM program likely encounter a hard failure again; the reactor just needs to further rollback the program.

4.7 Mitigating Persistent Memory Leak

Persistent memory leaks are a challenging type of hard failure to mitigate for two reasons. First, the fault instruction—when the system runs out of PM space or is stopped by a PM usage monitor—is often not connected to the root cause. Second, we need to revert the states far back to before the leaked persistent variables are created. Traditional volatile memory leak detection tools such as Valgrind are ineffective (Intel implements a PM Valgrind [2, 16] and a persistent inspector [55], but they detect bugs like unnecessary flushes, not persistent leaks). Tracing reachability of PM objects during an execution is also not enough, because a PM program may not call free on persistent variables (values in PM locations may be used after restart).

We observe that typically a PM program's recovery function retrieves almost all PM data structures before proceeding. At the same time, the Arthas checkpoint component keeps track of all PM variable updates. So our idea is to compare the PM variables that are accessed during recovery and the PM variable updates that are recorded in the Arthas checkpoint log. To capture the PM variable accesses in the recovery function, we provide two simple APIs `pmem_recover_begin` and `pmem_recover_end` that can be used to annotate the recovery function. The Arthas reactor then discovers all PM variables that are not freed in the checkpoint log *and* not accessed in the recovery function. This mitigation approach is simple and effective. To be safe, the Arthas reactor outputs the suspected leak PM variables and only frees them after confirmation.

5 Implementation

We implement Arthas in C/C++, with a total of 7,800 lines of code. The Arthas static analysis and PM trace instrumentation are built on top of LLVM [42]. The program dependency graph (PDG) construction uses an existing library `dg` [4, 24].

Computing the static PDG and pointer analysis can take a long time for large programs. Also, the PM trace can grow so large that it takes a substantial time to parse. These costs delay mitigation. To reduce delay, we implement the Arthas reactor in a client-server architecture. The Arthas reactor server starts once the target system code is available. It computes the PDG in the background. Once the PDG is computed, it can be re-used until the target system code changes. The reactor server also creates a thread to incrementally parse the PM trace file. When Arthas detects a suspected persistent failure, it invokes the reactor using an RPC client. The reactor server can compute the mitigation plan quickly.

6 Evaluation

We evaluate Arthas to answer several key questions: (1) Can Arthas mitigate hard faults in real-world PM systems? (2) How fast is the mitigation? (3) How much data loss is incurred in the rollback? (4) How do different reversion strategies compare? (5) What is the performance overhead?

6.1 Experimental Setup

We evaluate Arthas on 5 large PM systems, CCEH, PMEMKV, Memcached, Redis, and Pelikan. Their SLOC are 2.6K, 14K, 24K, 94K, and 20K, respectively. The first two are new PM systems, whereas the last three are mature systems adapted to add PM support. The experiments are conducted on a server with one 8-core CPU (2.50GHz), 94 GB DRAM, and two 128 GB Intel Optane DC Persistent Memory DIMMs.

No.	System	Fault	Consequence
f1	Memcached	Refcount overflow	Deadlock
f2	Memcached	`flush_all` logic bug	Data loss
f3	Memcached	Hashtable lock data race	Data loss
f4	Memcached	Integer overflow in append	Segfault
f5	Memcached	Rehashing flag bit flip	Data loss
f6	Redis	Listpack buffer overflow	Segfault
f7	Redis	Logic bug in refcount	Server panic
f8	Redis	slowlogEntry leak	Persistent leak
f9	CCEH	directory doubling bug	Infinite loop
f10	Pelikan	Value length overflow	Segfault
f11	Pelikan	Null stats response	Segfault
f12	PMEMKV	Asynchronous lazy free	Persistent leak

Table 2. List of persistent faults reproduced for evaluation

Failure Dataset. We collect and reproduce 12 persistent fault bugs from the five systems. Seven bugs are from our study dataset described in Section 2. Table 2 lists the 12 cases.

Methodology. For each case, we run the target system for 5 minutes. In 10 out of the 12 cases, the bug triggering condition for that case can be externally controlled (e.g., when a special request or workload is issued or a particular command is executed). We apply this triggering condition around half-way (2.5 minutes) of the system execution to trigger the bug. For two cases (f3 and f8), their triggering conditions are difficult to be externally introduced and instead happen naturally at some point as the system runs. We let failures in these two cases manifest on their own. After the 5-minute workload finishes or whenever the bug is detected, we begin fault mitigation using either Arthas or the baselines.

Baselines. We use a state-of-the-art checkpoint-rollback system CRIU [1] to compare with Arthas. CRIU is designed for traditional systems and takes a coarse-grained approach to periodically checkpoint the system states (snapshot the entire process states). It does not handle PM states. We enhance it to take a snapshot of the PM pools of a target system during checkpointing, which we refer to as *pmCRIU*.

In addition, we evaluate a PM-aware checkpointing solution based on Arthas, which only keeps the checkpoint related functionality of Arthas and disables the analyzer involvement. We refer to this baseline as *ArCkpt*. In ArCkpt, the reactor only considers the existing checkpoints and follows strict time order to rollback, like the rollback algorithm in pmCRIU. Note that ArCkpt is designed as a fine-grained rollback solution that leverages Arthas's information to perform reversion of individual checkpoint entries. It should be perceived more as a facet of Arthas, not as an alternative.

6.2 Effectiveness of Mitigation

Recoverability. We first evaluate how effective Arthas is at mitigating the persistent faults compared to pmCRIU and ArCkpt. We consider a case recovered when (a) the failure symptom (e.g., crash or deadlock) no longer appears, and (b) the system has at least some persistent states left. Arthas reverts PM states based on the computed slices and sequence numbers. pmCRIU dumps a snapshot image every minute. ArCkpt reverts PM states one at a time only based on sequence numbers. Each solution is configured with a timeout so that if it cannot mitigate the failure within 10 minutes, it will abort.

Solution	Fault Id.											
	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12
pmCRIU	✓	✓	✗	✓	1/10*	✓	✓	4/10	✓	✓	✓	✓
ArCkpt	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗
Arthas	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 3. Recoverability in mitigating the evaluated failures. *: 10% probability of success.

	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12
pmCRIU	✓	✓	n/a	✓	1/10	✓	✓	4/10	✓	✓	✓	✓
ArCkpt	n/a	n/a	n/a	✓	n/a	n/a	n/a	n/a	n/a	✓	n/a	n/a
Arthas (pg)	✓	✓	✓	8/10	✓	✓	✗	✓	✓	✓	✓	✓
Arthas (rb)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 4. Checking if the recovered system is in a semantically consistent state. Arthas (pg): purge mode; Arthas (rb): rollback mode.

Table 3 shows the results. Arthas successfully recovers the target systems for all 12 cases. This notable success can be attributed to the facts that (1) Arthas performs fine-grained checkpointing and maintains multiple old versions for an entry, which preserves the previous good states; (2) Arthas's rollback follows the dependency chains, which naturally matches how the root cause bug contaminates the PM states at runtime.

In comparison, pmCRIU mitigates 9 cases, and 2 cases with a probability. The probabilistic successes occur because the bugs for the two failures have a chance to be triggered in the first 1 minute, before pmCRIU has taken the first snapshot.

ArCkpt only successfully mitigates 2 cases. This is because ArCkpt rolls back at an individual checkpoint entry granularity. As a result, since we revert one entry at a time and try re-execution, most of the bugs whose root-causes are triggered far earlier end up timing out. ArCkpt works better for bugs that immediately crash the system.

Consistency. We further evaluate whether a successfully recovered system is in a semantically consistent state: (1) we run sanity checks on the persistent memory file with tools such as `pmempool-check`, which catch bad PM blocks; (2) we run the rollbacked system for 20 minutes and use multiple clients to issue a mix of requests; (3) if a system has stability test cases, we run these test cases; (4) we add basic consistency checks based on domain knowledge, e.g., the number of items should be equal to the hashtable size.

Table 4 shows that pmCRIU's and ArCkpt's results match with the basic recoverability. Arthas can preserve consistencies for most of the cases it recovered. This is because of its careful checkpoint designs and dependency analyses (§4.6). Two cases are the exceptions under Arthas's purge mode (§4.4). When reverting f7, Arthas only reverts the bad keys but not the values, leading to a semantic inconsistency when the client issues a `GET` command for that key. In f4, the system occasionally aborts in `do_slabs_free` in 2 out of 10 runs.

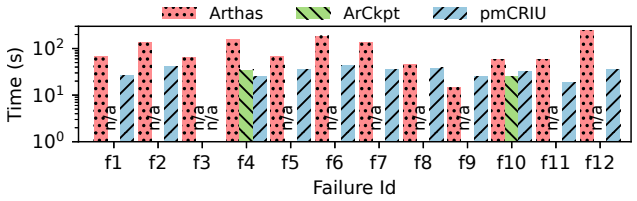


Figure 8. Time to mitigate the failures (including re-execution).

	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12
pmCRIU	1	3	X	1	3*	3	3	3*	1	3	1	1
ArCkpt	T	T	T	1	T	T	T	T	T	1	T	T
Arthas	12	5	1	12	10	8	8	1	1	11	46	1

Table 5. Attempts of rollback during mitigation.

Although reversion in each individual slice maintains consistency by following the dependency order, reverting multiple slices—because reverting one slice does not resolve the issue—can lead to subtle inconsistencies. In this case, one reverted slice is about the `slab` class slice and another is about some metadata. We plan to fix this issue in Arthas. In Arthas's rollback mode, the reverted system does not experience inconsistencies because it conservatively reverts all changes between two dependent updates.

6.3 Efficiency of Mitigation

Next, we measure the time the three solutions take to mitigate the failures. Figure 8 shows the result. On average, Arthas can mitigate the failures within 103.55 seconds. pmCRIU's average mitigation time is 32.33 seconds. ArCkpt's average mitigation time is 30.19 seconds. For each reversion, we need to invoke a re-execution script that restarts the target system and waits for it to finish initialization and pass the bug check. This re-execution delay (typically 3–5 seconds) consumes a significant portion of the mitigation time.

We further measure the number of rollback attempts for each failure. Table 5 shows the result. The median rollback attempts for Arthas is 8. pmCRIU checkpoints every minute and thus results in fewer rollback attempts, with a median of 3 attempts. But this coarse-grained checkpointing incurs significant data loss as we show later. Finally, ArCkpt manages to quickly recover when the bad PM updates cause immediate failures (rollback attempt of 1). However, when the bad PM values do not immediately cause problems, ArCkpt times out.

Interestingly, Table 5 reveals that, even though Arthas's checkpointing is fine-grained, the median rollback attempts is still relatively small. This is because Arthas follows the dependency order, rather than just sequence orders. In addition, our tested PM systems generally apply transactions with larger ranges of data. Arthas respects transaction units in the program when storing checkpoint log entries (Section 4.2). With smaller-size transactions,Arthas's rollback attempts can increase significantly. For example, for bug f1 with smaller transactions, the rollback attempts increase from 12 to 28.

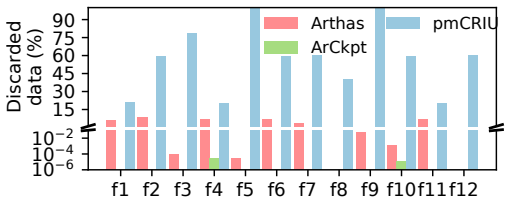


Figure 9. Data discarded in rollback by different solutions.

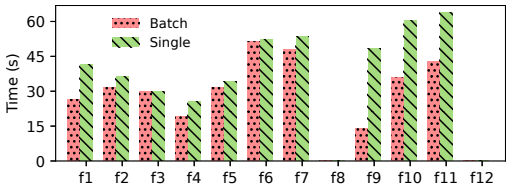


Figure 10. Mitigation time of Arthas using batch and one-by-one reversion. f8 and f12 do not fall under these reversion schemes.

6.4 Amount of Discarded State

Rollback solutions inevitably discard state changes made to a target system. A key requirement is to minimize the amount of discarded changes. For Arthas and ArCkpt, we track the number of reverted checkpoint entries; for pmCRIU (which has no concept of checkpoint entries), we measure the number of inserted key-value pairs right before the failure minus the number in the rollbacked system.

Figure 9 shows the results. Arthas on average only discards 3.1% of items and a minimum of 3.1e-5%. Note that the discarded data remains in the checkpoint log instead of being lost. In two cases, Arthas only discards one item. This is due to the fact that Arthas carefully decides "just enough" PM states to revert to based on the slicing analysis of fault instructions. For both of the persistent memory leak cases, f8 and f12, Arthas properly mitigates them using the checkpoint comparison mechanism (Section 4.7), and does not discard any good item: it precisely discards only the leaked objects.

In comparison, while pmCRIU mitigates 11 cases, it achieves so at the expense of discarding significant data. On average, pmCRIU incurs data loss of 56.5%. ArCkpt fails to properly mitigate most bugs. But for the two immediate crash cases that it mitigates, it only incurs a data loss of 1 item.

Note that in several cases, even with dependency analysis and one-by-one reversion, Arthas still reverts thousands of items (the ratio over total items is still small). This is because our dependency analysis is static. One dependent instruction in a slice may be invoked many times while only some invocations are bad. In the PM trace, an instruction such as this aliases to many sequence numbers that we cannot distinguish. The reactor conservatively reverts these sequence numbers. Our technical report [27] describes a binary search algorithm that reduces the sequence number set that we have to revert.

6.5 Reversion Strategy

Arthas leverages dependency analysis to compute the candidate list of logical sequence numbers (checkpoint log entries) to revert. We evaluate two main design choices regarding how to revert given the candidate list.

	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12
Batch	10	5	5	5	1	5	5	n/a*	2	15	15	n/a*
One-by-one	6	2	1	2	1	2	2	n/a*	1	11	12	n/a*

Table 6. Number of discarded items using a batch and one-by-one reversion strategy. *: f8 and f12 are memory leaks that do not fall under these reversion schemes.

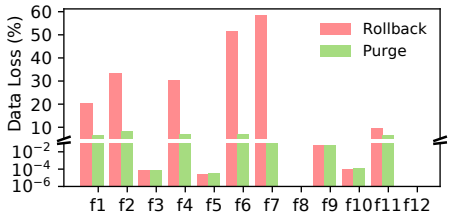


Figure 11. Discarded changes with rollback and purging modes.

	Fault Id.											
	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12
Detectable	✓	✗	✗	✓	✗	✓	✗	✗	✗	✓	✗	✗

Table 7. Detect the hard failures using common invariant checks.

Batching vs One-by-One Reversion. We by default revert sequence numbers in the candidate list one by one to minimize the discarded data. Another strategy is to revert the candidate list in batches. To evaluate these two strategies, we got several key bugs from Memcached and Redis and used a reduced workload. This was done in order to avoid influence from having slice nodes that alias to multiple sequence numbers (described in the previous Section). We use a batch limit of 5 sequence numbers and only revert when we either reach the end of the candidate list or pass the threshold.

Compared to the one-by-one strategy, using batch reversion reduces the number of re-execution attempts by 2.67× on average. However, due to the fact that the one-by-one strategy attempts re-execution after each reversion, the number of attempts will scale up depending on the max versions in the checkpoint. In Figure 10, we can see that one-by-one reversion is slower than the batch strategy due to the increased number of re-execution attempts. However, Table 6 shows that one-by-one reversion discards much less data than the batch strategy, because it reverts at a finer granularity.

Rollback vs Purging. Another decision is whether to rollback or purge (§4.4). Purging reverts only those in the candidate list. Rollback additionally reverts all the checkpoint log entries that are higher than the chosen sequence number. We compare the two modes by the data loss they incur for the evaluated failures. Figure 11 shows the result. Rollback introduces an average of 16.9% data loss while purging introduces an average of 3.6%. Rollback is more conservative than purging and is less likely to cause semantic inconsistencies.

6.6 Checksum and Invariant Approaches

An alternative to checkpoint-rollback solutions is to use checksums. Checksums are effective for catching value corruption and are widely used in storage systems. In the PM context, developers need to compute checksums for a PM state, store it, and update it if the PM state changes. To be comprehensive, each important PM state should be checksummed, which can be expensive. In addition, bad PM states in hard failures are much more than just value corruption. The checksum of an incorrect value (e.g., due to logic error) cannot be used to validate the correctness of the value. Indeed, only one of the 12 failures we evalute (f5) can be caught by checksums.

Another solution is invariant checking. Developers figure out and maintain a maintain a comprehensive set of invariants. For traditional systems that have well-known invariants, this is doable, e.g., `fsck`. For PM systems, this can be more challenging because they have many fine-grained PM states with application-specific invariants that are difficult to identify. In practice, developers do check common invariants, e.g., the number of key-value items must be equal to the hashtable size. But their comprehensiveness is rather limited. Table 7 shows that only 4 failures can potentially be detected by common invariant checks. More importantly, invariant checking only helps with *detection*. Fixing the inconsistent PM states remains a challenge, which Arthas is designed to address.

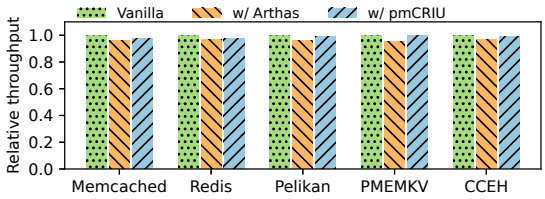


Figure 12. System throughput (op/s) relative to Vanilla.

	Memcached	Redis	Pelikan	PMEMKV	CCEH
Vanilla	62.36K	46.96K	36.02K	25.24K	146.76K
w/ Checkpoint	60.22K	46.44K	34.82K	24.10K	142.96K
w/ Instru.	61.62K	45.73K	35.94K	25.34K	143.94K

Table 8. Average system throughput (op/s), with Arthas checkpoint and with Arthas instrumentation.

6.7 Overhead

We measure the runtime overhead that Arthas introduces to the target systems and compare it with pmCRIU. For Redis and Memcached, we use the YCSB benchmark configured with 4 threads and a workload of 3 million operations (50% writes and 50% reads). We use a custom benchmark for PMEMKV and Pelikan with 6 million insertions, and a custom benchmark for CCEH with 1 million insertions. Figure 12 shows the average throughput for the vanilla systems, with Arthas, and with pmCRIU. Arthas introduces an overhead of between 2.9% to 4.8%. pmCRIU introduces an overhead of between 0.2% to 2.7%. pmCRIU's overhead comes from freezing the process and dumping the image. Given a one minute snapshot frequency, it incurs relatively low overhead.

Arthas's overhead comes from the PM state checkpointing and the instrumentation for tracing PM addresses. We measure their costs individually. Table 8 shows the result. We can see that the checkpointing contributes the majority of Arthas's

	Memcached	Redis	Pelikan	PMEMKV	CCEH
Static Analysis	468.80	256.07	225.23	52.98	165.90
Instrumentation	10.23	10.37	17.76	6.01	13.64
Slicing	0.38	0.54	0.04	0.05	0.59

Table 9. Time taken (in seconds) for Arthas to analyze and instrument the evaluated systems.

overheads (3–5%) The instrumentation introduces negligible overhead. Its low cost comes from buffering and asynchronous flushing. We also measure the latency overhead Arthas introduces to the target systems. Arthas incurs an average of 6% overhead for reads and 7% for writes.

6.8 Performance

We measure the time it takes for Arthas to analyze the target systems, instrument them and slice a given fault instruction. Table 9 shows the result. With a timeout enabled for the reaching definitions and pointer analysis, the Arthas static analysis takes from 53 to 469 seconds. Arthas is implemented in the client-server architecture (§5), so the analysis time does not influence the mitigation time. When the target PM system fails, the Arthas server already has the PDG computed. The main latency is the slicing to compute the candidate list for reversion. This depends on the fault instruction. As Table 9 shows, with the PDG available, the slicing finishes quickly.

7 Discussion and Limitations

Hard fault is a challenging problem for emerging PM systems. Arthas is a first step towards tackling this problem. It has several limitations to be addressed in future work.

Scalability Arthas uses static analysis to identify dependencies among PM states for efficient rollback. Static analysis in general faces scalability challenges. Our overall experience is that for existing large PM systems, static analysis can handle them well. Indeed, although Arthas analyzer did not use the most advanced static analysis algorithms, we have successfully applied Arthas to analyze Redis, Memcached and other popular PM systems that contain tens of thousands of SLOC. Scaling static analysis to very large systems (millions of lines of code) is actively researched (e.g., [37]). These advances can be incorporated into Arthas.

Analysis Accuracy False positives in the static analysis (e.g., alias analysis) can cause unnecessary dependencies. The main consequence is that Arthas would revert more than necessary and/or take more time to rollback. False negatives are more concerning to Arthas. The worst-case consequence would be missing dependencies, which can lead to inconsistencies. The timeouts that we add in the static analysis could create possible false negatives. From our observations, adding some prioritization/filtering often helps Arthas correctly capture the dependencies before the timeout. Future work could explore using techniques such as dynamic program slicing [18] to improve the analysis accuracy while addressing the challenge of reducing their high runtime overhead.

Consistency Semantic inconsistencies could arise in Arthas's purge mode. From our experience, though, such inconsistencies are rare (in 2 out of 12 evaluated cases) because of several designs. Arthas's checkpoint respects the program's choice of persistence granularity, so it will not revert a program to an intermediate state. In reverting multiple PM updates, it carefully follows the dependency chain. If a program uses transactions, Arthas reverts updates within the programmer-defined transaction. Most consistency requirements can be captured correctly with these mechanisms. We assume the remaining inconsistencies can cause noticeable anomalies (e.g., triggers assertions) that Arthas can be enacted to further revert the PM updates until the anomalies are gone. If semantic inconsistencies are intolerable to the target application, Arthas's more conservative rollback mode is a better option.

Distributed Application Arthas only works on standalone PM systems. It cannot handle distributed applications. Arthas also does not capture dependencies that are created by external communication outside this system. For example, a client first sends a request `r1` and then sends another request `r2` based on the `r1` result. Their dependency may *not* be reflected in the PM system's code. Our studied hard fault bugs do not involve such indirect dependencies.

If such a dependency does occur in a hard fault, currently under the purge mode, Arthas may revert PM states associated with `r1` (assuming it triggers the root cause bug) and may *not* revert PM states associated with `r2`, which cause an inconsistency for that client. Under the rollback mode, Arthas should be able to revert `r2`-related PM state updates as well, because they occur after the `r1` state updates (c.f., Figure 7b).

Hard faults in distributed PM systems that are concerned about external dependencies are interesting problems for future work. When recovering a distributed PM system, a distributed checkpoint-rollback solution might be needed. We could have each component checkpoint PM states locally, and add a global coordinator that runs a special rollback-recovery protocol [31]. We can expose the Arthas metadata in each component to the coordinator for determining an effective recovery plan. For external dependencies created by clients, if the client is stateful, it may need to be involved in the recovery. For instance, the PM system and client can maintain vector clocks [33]; after the PM system successfully rollbacks to a particular point, the client will then be notified to rollback its events with vector clocks after that point.

8 Related Work

Crash Consistency: The crash consistency problem has been extensively studied in prior work in the context of traditional file systems [25, 26, 34, 49] and PM systems [57, 62]. PM systems typically ensure crash consistency by requiring programmers to carefully order writes with persistence primitives, such as cache line flushes or the use of transactions.

Several solutions [23, 35, 38, 65] aim to provide transparent crash consistency/failure atomicity for PM systems. For example, PMThreads [65] maintains a shadow DRAM page for each persistent page allocated to a program, along with a working copy, and a consistent copy in NVM. NVThreads [38] buffers intermediate changes and commits shared data at the end of a critical section to ensure it is consistent after crashes.

Our work investigates a different type of reliability issues in PM systems—hard faults caused by traditionally "soft" bugs such as race condition, memory leak, random bit flip, etc. These issues occur during regular program execution. They are not state inconsistencies induced by crashes. Arthas is therefore complimentary to crash consistency solutions.

Finding Bugs in PM Systems: Liu et al. propose PMTest [46], a testing framework to detect crash consistency bugs in persistent memory systems. Pmemcheck [2] is a framework based on Valgrind [53] to check persistent memory programming errors such as store operations not being persisted, memory being added to two different transactions, and unnecessary flushes. XFDetector [45] is recently proposed to detect crash consistency bugs that concern the recovery and re-execution after the crash. A post-failure execution may read data that is not persisted during pre-failure execution. Or a post-failure execution may read from older committed data. These bugs still belong to crash-consistency bugs. AGAMOTTO [52] uses symbolic execution to discover bugs in PM systems related to misuse of persistent memory.

We investigate a different spectrum of bugs—soft-to-hard faults—in PM systems. From our study (Section 2), the errors that lead to hard faults are outside of the detection scope of these prior solutions. In addition, our work aims to mitigate hard faults instead of finding bugs.

Checkpointing and Rollback: Flashback [60] proposes checkpointing and rollback through a shadow process. Flashback periodically checkpoints a running process' execution state through copying its process structure and logging I/O interactions. When a failure occurs, the process is rolled back to previous state in the shadow process. CROCHET [21] proposes fine-grained checkpoints an individual variable granularity within unmodified commodity JVMs by using a lazy heap traversal algorithm that models a page-fault like checkpoint mechanism. Arthas targets PM systems and checkpoints at fine granularity (PM variables) in an eager fashion (as soon as the PM system reaches persistence points).

ThyNVM [57] proposes a hardware-assisted mechanism that supports checkpointing of persistent memory at both cache block and page granularities. Compared to ThyNVM, Arthas checkpointing does not require hardware modification and is not bound by periodic epochs. Also importantly, ThyNVM's checkpointing is a system-wide mechanism for ensuring crash consistency, but Arthas's checkpointing is for mitigating hard faults in different PM applications. Because of this, the Arthas checkpoint is tailored to each target application: it matches the exact granularity that the application issues PM updates at the code level.

Cohen et al. [29] propose a new language extension and runtime system called NVMReconstruction that takes an object-oriented approach to reconstruct persistent objects' states during restart. It requires developers to annotate the PM programs and focuses on avoiding inconsistencies between transient and persistent data during restart. Arthas focuses on transparently helping PM systems recover from hard faults that are caused by persistent bad PM states.

Elnozahy et al. [31] surveys rollback recovery protocols in distributed systems. At a high level, Arthas takes a similar dependency tracking approach, but with a different goal in a drastically different context. We track dependency through static analysis of instructions in a PM system and use the dependency information to revert the PM data to a good state with minimal data loss.

Coping with State Corruptions: Many of our studied hard faults in PM systems arise due when some persistent states become "bad" due to a logic error, race condition, integer overflow, etc. SafeNVM [41] uses a thread-based protection schema with hardware changes to combat against stray writes and offer data safety for non-volatile memory. File systems have tackled the issue of data integrity [19, 67], using a variety of checks to detect and avoid integrity violations. The hard faults Arthas target have a broader scope and are specific to PM applications. Generic integrity checks like checksums can only prevent a small portion of these hard faults.

9 Conclusion

This paper investigates the hard-fault reliability challenge in persistent memory systems. We analyze 28 bugs in popular PM systems to understand how they incur severe consequences that cannot be mitigated with restarts. We propose Arthas, a tool to mitigate hard faults in PM systems. Arthas designs PM-aware fine-grained checkpointing, program analysis and tracing to rollback persistent states to a working version while minimizing the discarded data. We evaluate Arthas on 12 hard faults from five large PM systems. Arthas successfully mitigates all cases and discards 10× less state on average when compared to traditional rollback solutions.

The source code of Arthas is publicly available at:
https://github.com/OrderLab/Arthas

Acknowledgments

We would like to thank our shepherd Yang Wang and the anonymous EuroSys reviewers for their valuable feedback. We thank the members of the OrderLab and Parv Saxena for their feedback during discussion and assistance on the project. Brian Choi is supported by an NSF Graduate Research Fellowship (No. DGE-1746891). This work was supported by NSF grant CNS-1942794.

References

[1] Checkpoint/Restore In Userspace, or CRIU. https://criu.org/Main_Page.

[2] Discover persistent memory programming errors with Pmemcheck. https://software.intel.com/en-us/articles/discover-persistent-memory-programming-errors-with-pmemcheck.

[3] Key/value datastore for persistent memory. https://github.com/pmem/pmemkv.

[4] LLVM DependenceGraph. https://github.com/mchalupa/dg.

[5] Memcached. https://memcached.org.

[6] Memcached flush_all bug. https://code.google.com/archive/p/memcached/issues/249.

[7] Memcached gets a dead loop in func assoc_find. https://github.com/memcached/memcached/issues/271.

[8] Memcached release notes for 1.5.18 (support for persistent memory). https://github.com/memcached/memcached/wiki/ReleaseNotes1518.

[9] Memkind - user extensible heap manager. http://memkind.github.io/memkind.

[10] N-store persists transient data structures due to dependencies. https://github.com/snalli/nstore/commit/a2f1997f855764196ccef2bf6d36d3e750ea3c86.

[11] Persistent memory development kit. https://pmem.io/pmdk.

[12] Pmemkv lazy free causes persistent memory leak. https://github.com/pmem/pmemkv/issues/7.

[13] Redis. https://redis.io.

[14] Redis crashes inside lpnext for large values. https://github.com/antirez/redis/issues/4349.

[15] Redis persistence options. https://redis.io/topics/persistence.

[16] Valgrind: an enhanced version for pmem. https://github.com/pmem/valgrind.

[17] A version of rocksdb that uses persistent memory. https://github.com/pmem/pmem-rocksdb.

[18] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, page 246–256, White Plains, New York, USA, 1990.

[19] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. ffsck: The Fast File System Checker. In *Proceedings of the 11th Conference on File and Storage Technologies (FAST '13)*, San Jose, California, February 2013.

[20] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 707–722, Melbourne, Victoria, Australia, 2015.

[21] J. Bell and L. Pina. CROCHET: Checkpoint and rollback via lightweight heap traversal on stock JVMs. In T. Millstein, editor, *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, volume 109 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:31. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

[22] E. Berrocal. Making NoSQL databases persistent-memory-aware: The Apache Cassandra example. https://software.intel.com/en-us/articles/making-nosql-databases-persistent-memory-aware-the-apache-cassandra-example, 2018.

[23] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, page 433–452, Portland, Oregon, USA, 2014.

[24] M. Chalupa. Slicing of LLVM bitcode. Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2016.

[25] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 228–243, Farminton, Pennsylvania, 2013.

[26] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 9, San Jose, CA, 2012.

[27] B. Choi, R. Burns, and P. Huang. Understanding and dealing with hard faults in persistent memory systems (technical report). https://orderlab.io/paper/arthas-tech-report.pdf, April 2021.

[28] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 105–118, Newport Beach, California, USA, 2011.

[29] N. Cohen, D. T. Aksun, and J. R. Larus. Object-oriented recovery for non-volatile memory. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), Oct. 2018.

[30] Dormando. The volatile benefit of persistent memory. https://memcached.org/blog/persistent-memory, 2019.

[31] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, Sept. 2002.

[32] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[33] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Communications*, ACSC '88, pages 56–66, 1988.

[34] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt. Soft updates: A solution to the metadata update problem in file systems. *ACM Trans. Comput. Syst.*, 18(2):127–153, May 2000.

[35] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch. Persistency for synchronization-free regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 46–61, Philadelphia, PA, USA, 2018.

[36] J. Gray. Why do computers stop and what can be done about it? Technical Report TR-85.7, June 1985.

[37] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, page 289–298. IEEE Computer Society, 2011.

[38] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster. NVthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 468–482, Belgrade, Serbia, 2017.

[39] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv e-prints*, page arXiv:1903.05714, Mar 2019.

[40] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 494–508, Huntsville, Ontario, Canada, 2019.

[41] P. Kumar and H. H. Huang. SafeNVM: A non-volatile memory store with thread-level page protection. In *2017 IEEE International Congress on Big Data (BigData Congress)*, pages 65–72. IEEE, 2017.

[42] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, CGO '04,

pages 75–, Palo Alto, California, 2004.
[43] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram. Recipe: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 462–477, Huntsville, Ontario, Canada, 2019.
[44] Lenovo. memcached-pmem. https://github.com/lenovo/memcached-pmem, 2018.
[45] S. Liu, K. Seemakhupt, Y. Wei, T. Wenisch, A. Kolli, and S. Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1187–1202, Lausanne, Switzerland, 2020.
[46] S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan. PMTest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 411–425, Providence, RI, USA, 2019.
[47] B. Lu, X. Hao, T. Wang, and E. Lo. Dash: Scalable hashing on persistent memory. *Proc. VLDB Endow.*, 13(10):1147–1161, Apr. 2020.
[48] V. J. Marathe, M. Seltzer, S. Byan, and T. Harris. Persistent Memcached: Bringing legacy code to byte-addressable persistent memory. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'17, pages 4–4, Santa Clara, CA, 2017.
[49] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 33–50, Carlsbad, CA, USA, 2018.
[50] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton. An analysis of persistent memory use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 135–148, Xi'an, China, 2017.
[51] M. Nam, H. Cha, Y. ri Choi, S. H. Noh, and B. Nam. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, Boston, Massachusetts, USA, 2019.
[52] I. Neal, B. Reeves, B. Stoler, A. Quinn, Y. Kwon, S. Peter, and B. Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 1047–1064, Banff, Alberta, Canada, Nov. 2020.
[53] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 89–100, San Diego, California, USA, 2007.
[54] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 2005 Network and Distributed System Security Symposium*, NDSS '05, San Diego, California, USA, 2005.
[55] K. P. O'Leary. How to Detect Persistent Memory Programming Errors Using Intel® Inspector - Persistence Inspector. https://software.intel.com/content/www/us/en/develop/articles/detect-persistent-memory-programming-errors-with-intel-inspector-persistence-inspector.html. Accessed: 2021-03-22.
[56] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 235–248, Brighton, United Kingdom, 2005.
[57] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 672–685, Waikiki, Hawaii, 2015.
[58] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, page 317–331. IEEE Computer Society, 2010.
[59] Snalli. Redis pmem. https://github.com/snalli/redis, 2016.
[60] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, page 3, Boston, MA, 2004.
[61] Y. I. Takashi Menjo. Introducing PMDK into PostgreSQL: Challenges and implementations towards PMEM-generation elephant. https://www.pgcon.org/2018/schedule/events/1154.en.html, 2018.
[62] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, Newport Beach, California, USA, 2011.
[63] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 439–449, San Diego, California, USA, 1981.
[64] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, page 1–12, La Jolla, California, USA, 1995.
[65] Z. Wu, K. Lu, A. Nisbet, W. Zhang, and M. Luján. PMThreads: Persistent memory threads harnessing versioned shadow copies. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 623–637, London, UK, 2020.
[66] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST '16, page 323–338, Santa Clara, CA, 2016.
[67] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. End-to-end data integrity for file systems: A zfs case study. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, page 3, San Jose, California, 2010.
[68] P. Zuo, Y. Hua, and J. Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, pages 461–476, Carlsbad, CA, USA, 2018.

A Artifact Appendix

A.1 Abstract

Arthas is a tool that aims to properly recover persistent memory systems by mitigating hard faults. Arthas uses a combination of instrumentation and dynamic checkpointing in conjunction with static analysis program slicing to rollback Persistent Data while minimizing data loss. Arthas is publically available on github and there are testing scripts to verify the workflow of Arthas as described in the paper. Arthas requires access to either emulated or real PM Hardware. The artifact is publicly available at: https://github.com/OrderLab/arthas.

A.2 Artifact Check-list

- **Hardware:** One 8-core CPU (2.50GHz), 94 GB DRAM, and two 128 GB Intel Optane DC Persistent Memory DIMMs.
- **Run-time environment:** Ubuntu 18.04.4 LTS
- **Public link:** https://github.com/OrderLab/arthas

- **Artifact Instructions:** https://github.com/OrderLab/Arthas/blob/master/artifactREADME.md
- **Code licenses:** Apache License 2.0.

A.3 Description

A.3.1 Arthas's Components. Arthas comprises of four primary components:

- Checkpoint Component: Uses dynamic checkpointing to store old versions of Persistent Data. Used alongside a modified version of PMDK, which intercepts PMDK functions to call checkpoint functions
- Analyzer: Uses llvm and dg to form a dependency graph of the analyzed system and detect PM variables. The analyzer is also responsible for inserting instrumentation points in a target system to obtain runtime information such as the dynamic address of the persistent variable in a slice
- Detector: Used to detect hard PM failures
- Reactor: The runtime meta environment that reacts on a hard fault and brings the system back to a normal state using repeated reversion and re-execution

Arthas first uses the analyzer to instrument the executable of the target PM system with Arthas's checkpoint support. The target system runs while dynamically checkpointing PM data and then upon detection of a fault, Arthas's reactor server and client are enacted to react to the hard fault and revert the target system to a normal state.

A.4 Installation

A.4.1 Dependencies.

- Hardware: Either emulated or real PM hardware
- The Arthas analyzer requires LLVM 3.9: https://github.com/llvm/llvm-project
- wllvm
- PMDK: https://github.com/pmem/pmdk
- Custom PMDK: https://github.com/OrderLab/Arthas-PMDK.git
- We also need to install PMDK's dependencies
- Other dependencies that need to be installed include `cmake` (3.4+), `protobuf` (3.11), and `grpc` (1.28.1)

```
pip install wllvm
sudo apt install autoconf automake pkg-config libglib2.0-dev
libfabric-dev pandoc libncurses5-dev cmake
```

A.5 Experiment Deployment

In this experiment we will see Arthas mitigate bug f1: the Memcached refcount bug described in the paper.

A.5.1 Setting Up Environment Variables. To use the wllvm wrapper for compiling a target system, set the following environment variables:

```
export LLVM_COMPILER=clang
export LLVM_HOME=/opt/software/llvm/3.9.1/dist
export LLVM_COMPILER_PATH=$LLVM_HOME/bin
export PATH=$LLVM_COMPILER_PATH:$PATH
```

The LLVM_HOME path should be replaced appropriately.

A.5.2 Testing Arthas: Minimal Interaction.

```
git clone https://github.com/OrderLab/Arthas.git
cd Arthas
scripts/artifact_test.sh
```

The test script will build Arthas, custom PMDK, vanilla PMDK, target system Memcached, and finally run the Arthas analyzer on the Memcached to instrument it.

If successful, you should see a Memcached bitcode file in *eval-sys/memcached/memcached.bc* and a Arthas hooks metadata file in *experiment/memcached/memcached-hook-guids.map*.

You can further test Arthas by a real bug in Memcached:

```
scripts/experiment_memcached_refcount.sh
```

This demo script will do the following things:

1. Start a buggy version of Memcached server (instrumented).
2. Insert some workload to Memcached.
3. Invoke another script to trigger the bug (refcount overflow) and cause Memcached to fail.
4. Start Arthas reactor server.
5. Run the Arthas reactor client to mitigate the failure.

Note that in practice, Arthas's reactor server (step 4) is typically started along with the target system (step 1).

If successful, a message of "Recovery finished" will be printed.

A.5.3 Expected Result. We should see the lines "done with binary reversion [num]" where the binary value of num (0 or 1) will tell you if Arthas was successful in mitigating a bug or not. We should also see "total reverted items is [num]" which will tell you the total number of items reverted.

A.6 Full Usage

To see a walkthrough of the Arthas workflow and the detailed instructions for using the Arthas components, please refer to the READMEs in Arthas's public repository.