Zhuque: Failure is Not an Option, it's an Exception

George Hodgkins, *University of Colorado, Boulder;* Yi Xu and Steven Swanson, *University of California, San Diego;* Joseph Izraelevitz, *University of Colorado, Boulder*

https://www.usenix.org/conference/atc23/presentation/hodgkins

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

Zhuque: Failure is Not an Option, it's an Exception

George Hodgkins*
University of Colorado, Boulder

Yi Xu*
University of California, San Diego

Steven Swanson
University of California, San Diego

Joseph Izraelevitz
University of Colorado, Boulder

Abstract

Persistent memory (PMEM) allows direct access to fast storage at byte granularity. Previously, processor caches backed by persistent memory were not persistent, complicating the design of persistent applications and reducing their performance. A new generation of systems with flush-on-fail semantics effectively offer persistent caches, offering the potential for much simpler, faster PMEM programming models.

This work proposes Whole Process Persistence (WPP), a new programming model for systems with persistent caches. In the WPP model, all process state is made persistent. On restart after power failure, this state is reloaded and execution resumes in an application-defined interrupt handler.

We also describe the Zhuque runtime, which transparently provides WPP by interposing on the C bindings for system calls in userspace. It requires little or no programmer effort to run applications on Zhuque.

Our measurements show that Zhuque outperforms state of the art PMEM libraries, demonstrating mean speedups across all benchmarks of 5.24× over PMDK, 3.01× over Mnemosyne, 5.43× over Atlas, and 4.11× over Clobber-NVM. More important, unlike existing systems, Zhuque places no restrictions on how applications implement concurrency, allowing us to run a newer version of Memcached on Zhuque and gain more than 7.5× throughput over the fastest existing persistent implementations.

1 Introduction

Persistent memory (PMEM) exposes fast storage devices as byte-addressable main memory, allowing the processor to access persistent data via load and store instructions. The durability of PMEM enables an application's in-memory data to survive across system reboots and unexpected power failures. It promises to realize a vision of high performance, data persistence, a simple programming interface, and low storage overhead at the same time.

*The first two authors contributed equally to this work

However, building a system that realizes the promise of persistent programming is not simple. The contents of CPU caches do not survive power loss, and, since caches may delay evicting a modified cache line, writes may not reach PMEM in program order. This makes reasoning about the state of memory after a crash extremely challenging.

Programming systems (e.g., libraries, programming models, language support, and compilers) to help address the challenges of persistent memory programming have proliferated over the last decade. Broadly, three families of systems have emerged: each takes a different approach to consistency, and each faces significant challenges which bar widespread adoption.

The first and largest family [50, 63, 68, 71] requires programmers to access persistent state only through well-defined atomic operations (often called transactions). This provides a clean notion of consistency: after recovery from crash, each atomic section has either executed entirely or not at all. However, like all transactional memory models, this approach suffers from serious weaknesses: it is fundamentally incompatible with non-transactional synchronization, and has never gained significant traction in real systems.

The second family of systems [6, 26, 30, 42] uses FASEs, regions of code protected by locks, as atomic regions for PMEM updates. Legacy code can run with minimal changes, but these systems suffer from fundamental weaknesses arising from complex locking schemes and external IO. As we will show, addressing these weaknesses either cripples the system or essentially reduces it to a transaction-based system.

The final family of systems takes the more dramatic step of making *everything* in the system persistent via whole-system persistence (WSP) [47]. WSP provides the conceptually simplest programming model: Nothing much changes and, from the program's perspective, crashes never occur. WSP faces two major challenges: First, making all of memory persistent has until recently been infeasible, because regularly flushing volatile caches to PMEM creates enormous performance overheads. Second, making *everything* persistent would require a far-reaching redesign of many system components, for an unclear benefit.

We think that WSP-style persistence is due for a renaissance: The advent of PMEM devices and platforms supporting *flush-on-fail* semantics (e.g. eADR for NVDIMMs or GPF for CXL devices) allows developers to treat caches as effectively persistent [14, 29], removing the main performance argument against WSP. Further, we believe that limiting the scope of persistence to a process – yielding *Whole Process Persistence (WPP)* – and providing well-defined, application-level semantics for system failures combine to produce a programming model that is fast, flexible enough to support legacy programs and complex locking schemes, and easy for programmers to use and understand.

WPP provides a simple abstraction to the process: its entire memory is persistent and will survive a power outage. If a power outage occurs, the process receives an OS signal after restart notifying it of the crash. The process can install a normal error handler for this signal which cleans up and exits, or performs more complex application-specific recovery; by default, program execution simply continues at the point of failure.

This work makes the following contributions:

- We identify a fundamental limitation of FASE-based PMEM systems.
- We introduce the WPP programming model, which treats power failure as a recoverable exception.
- We build the Zhuque runtime which provides WPP and describe its design and implementation.
- We provide experiments demonstrating the viability of the WPP system and its performance improvements over existing alternatives.

Zhuque is faster than existing PMEM programming systems. It is between 4.7× and 10.14× faster than PMDK [50], Mnemosyne [63], Atlas [6] and Clobber-NVM [68] on STAMP applications. Zhuque is also more flexible than these systems: Since Zhuque is agnostic about the application's locking scheme, it can run the most recent version of memcached, while those systems cannot. As a result, our Zhuque-based persistent memcached is more than 7.5× faster than any similar system. We also demonstrate Zhuque's flexibility by running unmodified Python benchmarks with minimal performance loss.

The rest of this paper is organized as follows. Section 2 provides some background on PMEM and associated software systems. Section 3 describes fundamental limitations of prior art necessitating the WPP model. We discuss the WPP design and musl-based system implementation in Section 4 and Section 5, respectively. Section 6 showcases the performance of WPP. We discuss related work in Section 7 and conclude the paper in Section 8.

2 Background

PMEM has introduced new possibilities for designing storage systems: programs can have byte-addressable access to terabytes of persistent data at near-DRAM latencies. However, utilizing PMEM in a both performant and programmer-friendly manner remains a challenging problem.

This section begins by describing our machine model, and then reviews existing general-purpose persistent memory programming models and their limitations to motivate WPP.

2.1 Machine Model

WPP is designed for a multi-core, cache-coherent machine equipped with PMEM (e.g. Intel DC Persistent Memory [28] or persistent CXL.mem devices [53]), and supporting *flush-on-fail* semantics, meaning that they provide a hardware guarantee that all in-flight and cached writes will reach PMEM in the event of an external power failure (as opposed to a fault in the machine or its onboard power supply). Such guarantees are provided by eADR-compliant platforms and NVDIMMs, and CXL platforms and devices supporting Global Persistent Flush (GPF). eADR and GPF are similar solutions targeting different device interfaces: the primary hardware requirement for both is that the platform must store sufficient energy to allow caches and internal device buffers to be drained to persistence after a power failure [1, 53].

On x86 systems, both eADR and GPF require system firmware to initiate and oversee the drain to persistence in response to a System Management Interrupt (SMI) [1, 13, 14]. Upon receiving this interrupt, the processor retires all in-flight instructions, drains all stores to the cache, and saves architectural state (register file etc.) to a designated per-core memory region before beginning execution of the SMI handler [16]. In both GPF and eADR, this handler first flushes the processor caches (and, for CXL, the caches of any CXL.cache device), and then proceeds to flush the buffers on the PMEM devices (NVDIMMs for eADR, CXL.mem devices for GPF) [1, 14].

2.2 Persistent Programming Models

Most existing persistent programming libraries rely on marking regions as *failure-atomic*, that is, all of the code region's effects will survive a failure or none will. Models differ in whether regions are explicitly marked (*transactional*) or inferred from locks (*FASE-based*). In addition, one work has proposed making the whole system persistent.

2.2.1 Transactional Libraries

Transactional PMEM libraries expect the programmer to explicitly mark failure atomic sections. For concurrency, these libraries either rely on off-the-shelf transactional memory systems or require the use of their own locks. For example, NV-Heaps [9], Mnemosyne [63], and DudeTM [41] are built on existing transactional memory (TM) systems, and implement their failure-atomicity techniques (e.g. redo or undo logging) on top of those systems.

Meanwhile, transactional libraries that rely on locks generally expect transactions to acquire and release locks in a conservative, strong strict two-phase locking pattern [51, 64], that is: transactions acquire all locks at transaction begin, transactions release all locks at transaction commit, and locks are released in the order they are acquired. For example, PMDK [50], Pangolin [71] and Clobber-NVM [68] require applications to follow this lock pattern.

2.2.2 FASE-based libraries

Atlas [6] proposed the concept of failure-atomic sections (FASEs) as an alternative to transactions. A FASE is a failure-atomic operation which begins when a thread acquires its first lock and ends when it holds none — importantly, the final lock held may be different from the first lock. Because this locking scheme allows updates to be visible to other FASEs before a FASE commits, FASE-based libraries are required to track dependencies between threads, and roll back dependent FASEs in case of failure. Because FASEs are dynamically formed at runtime, user annotation is not required for existing lock-based code. NVThreads [26], JUSTDO [30], and iDO [42] follow this model.

2.2.3 Whole System Persistence

Instead of basing persistence on bounded sections of code, whole-system persistence (WSP) [47] focuses on the persistence of the entire system. WSP describes a system substantially similar to eADR and GPF, where an interrupt at power failure triggers the draining of volatile caches/buffers to persistence. This model requires no annotation and avoids the extra work done by transactional or FASE-based systems, but requires that large amounts of state be made persistent at the instant of failure, which until the advent of flush-on-fail systems was not possible.

3 Limitations of Prior Art

In this section, we argue that the existing programming models for persistent memory, namely transactional or FASE-based, necessitate an alternative path, especially when working with legacy code.

Fortunately, the emergence of persistent caches has enabled our efforts to develop a revitalize a model that does not fit either of these directions, namely, whole process persistence, in which all process state is preserved at a power failure.

3.1 Limitations of Transactions

The fact that many failure atomicity libraries leverage transactional memory is not surprising — transactions are commonly leveraged for durability within databases and file systems. When applied to (volatile) multi-threaded code, the transactional memory programming model simplifies concurrency by exporting to the programmer "single global lock" semantics, that is, the programmer should simply protect groups of accesses to shared data as "transactions," each of which are mutually exclusive. The transactional programming model is in theory appealing as programmers need not worry about data races on shared data, multiple locks, or parallel performance. To this transactional programming model, many failure atomicity libraries add persistence: transactions become both visible to other threads, and persistent, upon transaction completion.

In practice, however, despite decades of research and dedicated hardware support, (volatile) transactional memory has failed to become a common programming paradigm for general purpose multi-threaded code. Transactions generally mix poorly with both other synchronization methods (locks, barriers, condition variables, etc.) [4, 70] and IO [45, 52], tend to incur significant performance overhead when compared to fine grained locking [4, 19], and are incompatible with legacy multi-threaded code [52], whose locking discipline is rarely compatible without significant rewriting. Support for transactional memory in C++, for example, remains experimental [45]. There is no indication that persistent transactional memory systems will solve these problems, indeed, they appear to perpetuate them.

Generally, the transactional programming model is exported to the programmer using a scoped transaction, (e.g. `transaction{}`) and the library guarantees transactions will execute mutually exclusively (e.g. PMDK's C++ interface). However, for PMEM, transactional libraries may syntactically decouple mutual exclusion from failure atomicity due to language limitations (e.g. PMDK's C interface). In such an API, the library expects the programmer to first explicitly acquire the necessary locks to gain mutual exclusion before, subsequently, executing the transaction's failure atomic contents.

Despite this apparent separation, a transactional PMEM library's programming model imposes hard limits on the locking discipline - it expects that all transactional updates are mutually exclusive and isolated by the locking discipline. This restriction effectively forces the application to use a limited locking scheme such as single-global-lock or strong strict conservative two phase locking to protect any failure-atomic update. The programming model *explicitly disallows* releasing or acquiring a lock while executing a failure-atomic update.

For more complex locking schemes in which failure-atomic writes are visible to other threads before they are committed, the use of a FASE-based programming model is required, and is often necessary for legacy programs as, in general, their existing synchronization fails to follow the restrictive transactional requirements.

3.2 Limitations of FASEs

Despite being, at first appearances, more compatible with legacy code, we argue the FASE-based model is also fundamentally flawed, or, at the very least, excessively permissive. The FASE model defines a failure-atomic code region as a "contiguous critical section," that is, it defines a failure-atomic code region as stretching from a thread's first lock acquire until the point where it holds no locks. While flexible with respect to locking scheme, this model requires tracking runtime dependencies between concurrently running failure-atomic code regions, which may not be isolated from each other. This permissiveness results in complicated and degenerate scenarios for recovery.

As a contribution of this work, we demonstrate that, for certain adversarial application patterns, any FASE-based system will either fail to recover or collapse into a degenerate case in which literally all program state must be logged for recovery, including volatile data never accessed within failure atomic regions — effectively, the FASE programming model requires whole process persistence for correctness.

Theorem 3.1 (FASE Limitation) *There exist applications for which, in order to consistently recover from a crash, a reasonably permissive FASE-based failure atomicity system requires all volatile program state be available at recovery.*

We prove this theorem by counterexample. This counterexample (Figure 1) can emerge naturally where two threads communicate via shared variables and one executes IO, a common pattern in event-based servers. In these servers, some threads handle the IO socket (thread 2 in example), some threads are application workers (thread 1), and they communicate via shared flags to manage outstanding requests. Detecting this pattern requires detailed reasoning about synchronization, and therefore prevents the blind use of FASEs on applications.

In the remainder of this section, we describe the counterexample and a brief sketch of our proof's reasoning. A full proof by contradiction, formal definitions, and additional discussion incorporating related work can be found in Appendix A.

Figure 1 gives our adversarial application that breaks FASE-based systems. In this example, two threads compute a fixed series of four values for nonvolatile variable x. Thread 1 computes the first value, Thread 2 the second and third, and Thread 1 the final, fourth value.

The two "tricks" of the code are that (1) the long FASE executed by thread 1 (lines 6 through 22) spans the entire example and (2) the third value of x, computed, but not assigned, outside of a FASE (line 39), is dependent on an access to a large volatile array Q.

```
1  lock_t lock0, lock1, lock2;
2  bool cond1 = false, cond2 = false;
3  int Q[] = rand(); // large random volatile array
4  nvm<int> x = 0; // x resides in nvm
```

```
5  void thread1{
6    lock0.lock();
7      x = (int s1=f1(x));
8
9      lock1.lock();
10       cond1 = true;
11     lock1.unlock();
12
13     bool w = true;
14     while(w){
15       lock2.lock();
16         if(cond2)
17           {w = false;}
18       lock2.unlock();
19     }
20
21     x = (int s4=f4(x));
22   lock0.unlock();
23 }
```

```
24 void thread2{
25   bool w = true;
26   while(w){
27     lock1.lock();
28       if(cond1){
29         w = false;
30         x =(int s2=f2(x));
31       }
32     lock1.unlock();
33   }
34
35   int in;
36   printf("x=%d", s2);
37   scanf("%d",&in);
38   /*****/
39   int s3 = f3(s2,in,Q);
40
41   lock2.lock();
42     x = s3;
43     cond2 = true;
44   lock2.unlock();
45 }
```

Figure 1: FASE counterexample

Recovery of this example presents an unsolvable problem. First, we note that Thread 1's long FASE, due to failure-atomicity semantics, forces recovery to recover either to the very beginning of the program or the very end. However, both options are impossible for a crash at line 38, just before x's third value is computed. At this point, thread 2 has already issued IO, so rolling back program state at recovery is inconsistent with the external world. However, rolling forward from this point requires the computation of the third value of x, which is dependent on an arbitrarily sized volatile array (Q). Since Q can be of any size, it can be replaced, without loss of generality, with any or all of the program's volatile state, effectively requiring whole process persistence.

Our proof requires failure atomicity systems to be "reasonably permissive," by which we mean that this counter example can be expressed as valid input for the system. Systems that restrict locking to two-phase-locking (e.g. [50,68]) or a single, semantic, global lock (i.e. transactional memory [63]) avoid this counterexample by prohibiting the locking pattern. Of course, by the same token, this restriction hampers their utility for legacy code, which rarely follows such a strict locking discipline.

The FASE programming model may be fixable by prohibiting situations like the counter-example. Simple (but undesirable) solutions include prohibiting all volatile accesses or all IO in the program. Alternatively, we could try to prohibit the precise counter-example problem by targeting the interplay

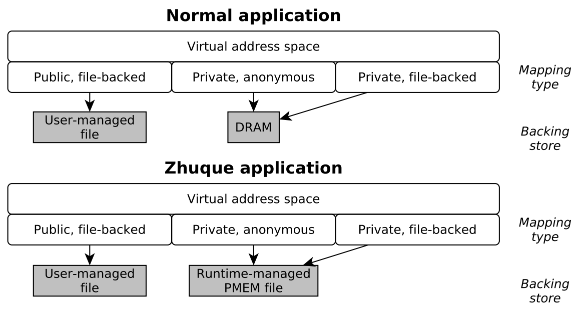

Figure 2: **Virtual memory in Zhuque.** The runtime modifies the backing store based on the mapping type, but the interface presented to the userspace application does not change.

between FASE dependencies, volatile accesses, and IO. One potential approach to achieve this involves a specification that disallows volatile accesses concurrently with a FASE execution. However, formally defining this specification is tricky, and formally verifying the proper use of FASEs is almost certainly undecidable through the halting problem. Notably, the requirement of a transactional locking scheme (e.g. strict, strong conservative 2PL) would also prevent the counterexample by explicitly disallowing its locking discipline.

To our knowledge, all existing FASE-based systems (e.g. [6, 26,30,42]) are “reasonably permissive” and would both accept this code as valid input and fail to recover correctly on it.

4 Design

Whole process persistence (WPP) is our answer to the limitations of transaction- and FASE-based programming models. In WPP, the in-memory state of an individual process is made persistent with, in simple cases, no modification to the application, primarily by interposing on the creation of virtual memory mappings (see Figure 2). WPP is designed for systems with flush-on-fail support, so we expect the contents of the process’s PMEM-backed cache lines to survive a power failure. When the process is restarted after a power failure, it receives an OS signal, which it can ignore or handle with a signal handler. If no signal handler is installed, or if the installed signal handler does not exit the program, each thread continues execution at the point where it was interrupted by the failure.

There are several benefits to this model over transactions and FASEs. First and most importantly, WPP solves the problem described in Section 3 by discarding the concept of a failure-atomic section. The visible effects of an instruction on process state (that is, not including effects on OS state or peripherals) are guaranteed to survive a failure at least from the point at which they are visible to other threads. Second, restarting at the point of failure removes the need to "redo" or "undo" any writes at recovery, and with it the need to keep a persistent log and incur the cost of extra writes to PMEM. Third, no longer needing to define failure-atomic sections either reduces the programmer’s burden directly, compared to manually-annotated failure-atomicity systems, or allows them to design concurrency schemes orthogonal to persistency without incurring overhead, unlike FASE-based systems.

There are two requirements that must be satisfied in order for an application to use WPP. The first is that its threading and virtual memory must be managed using a well-defined API for those purposes (i.e., on POSIX: `mmap()`, `pthread_create()`, etc). Any modern application targeting a POSIX system would have to go out of its way in order to violate this requirement.

The second is that applications must check error returns from system calls and other mechanisms that access non-process-private state, to detect failure-related errors beyond the process boundary, such as an application using a file on a filesystem that was not remounted after system restart. This requirement is more onerous than the first, but in our experience a wide range of applications can be correctly restarted without modification or special handling.

The principal challenge in implementing WPP is preserving process state across a power failure. Continuing execution after failure requires that the process’s virtual address space, volatile architectural state, and relevant kernel-resident state (e.g., the file descriptor table) are a) persistent or b) can be resurrected along with the application.

The remainder of this section introduces Zhuque, our runtime implementing WPP, and describes how it makes process state persistent and restores that state after failure.

4.1 Overview

Zhuque provides WPP functionality by interposing on system calls which allocate resources (memory, file descriptors, threads), and by modifying the application startup process. In order to do this, we modified `libc`, which provides C bindings for system calls and implements the application startup process. Zhuque also requires small changes to the kernel to protect userspace context when failures occur in kernel mode (see Section 5.3 for details).

Interposing on system calls allows Zhuque to ensure that all application state which is normally volatile is instead stored in PMEM, as shown in Figure 2. It also allows Zhuque to track memory mappings and system calls so it can reconstruct the program’s address space and re-create its kernel-resident state after a failure. Remaining volatile architectural state (e.g., the register file) is preserved by writing it to PMEM at failure.

When the application is resurrected after failure, Zhuque restores the application’s address space, respawns its threads, and each thread reloads its architectural state. Execution resumes by calling the program’s power failure signal handler, if it exists, and then resuming execution of each thread at the point interrupted by power failure.

4.2 Ensuring State Persistence

The first requirement that Zhuque must fulfill is ensuring that all state required for continuing correct execution of a program is preserved across power failures. This state can be divided into three categories based on its storage location: architectural state, memory state, and file state.

If a system supports flush-on-fail, it would be possible to modify its firmware to write per-thread architectural state (register file, floating-point configuration, etc.) to PMEM in response to power failure. However, we do not have the ability to modify that firmware, so we emulate it using userspace signals (see heading Power Failure in Section 5.1). We also save architectural state to PMEM on every kernel entry, in case a failure occurs in kernel mode (see Section 5.3).

File state is either inherently persistent, if the file was opened read-only or if changes have been written to disk, or is buffered awaiting being written to disk, in which case it is actually memory state and is handled as described below.

Automatically ensuring memory state is persistent is more complex, and is one of the main innovations of this design. Memory state itself can be divided into dynamic and static memory.

Dynamic memory Programs conjure dynamically allocated (heap) memory and thread stacks by calling anonymous `mmap()` (often via `malloc()`). Zhuque interposes on `mmap()` so that requests for anonymous memory return DAX-mapped persistent memory backed by a runtime-managed PMEM file, making heap and stack memory persistent.

Static memory Before an application binary is executed, the loader uses `mmap()` to create memory regions to hold code and static data (globals) from the application binary and linked dynamic libraries. Zhuque treats these regions differently based on whether they are un/zero-initialized, or initialized to non-zero values. The loader creates un/zero-initialized regions with anonymous `mmap()`, so they are treated as dynamic memory.

Initialized static memory, however, actually takes up space in the binary, and is loaded by mapping that region of the binary into memory as a private mapping. Thus, Zhuque transforms any writable, private mapping backed by a file to a writable, shared mapping that is backed by a PMEM file (see Figure 2), which is populated with the initialization values from the binary.

This mechanism also cleanly handles other outputs of dynamic loading, like relocations of position-independent code and cross-binary symbol resolutions, since they also are stored in writable, file-backed, private mappings.

4.3 Ensuring Correct Restoration

Having persisted the application's state, we also have to ensure it can be restored correctly. Recovery must restore the application address space, restore kernel-resident state, and restore architectural state.

Application address space All of the PMEM-backed memory mappings managed by Zhuque, as well as any other mappings the application created with `mmap()`. Zhuque stores the mapping table in a persistent memory file, and updates it to match any changes to the address space as they occur, so no action is required at failure to ensure this metadata is persistent.

At recovery, restoring the virtual memory map to its previous state must be done first, because all other state to be restored is stored in virtually mapped persistent memory. Restoration consists of re-mapping each virtual memory region with the correct backing store and access permissions. This restoration also replaces dynamic loading.

Kernel-resident state Any data required for continuing execution that resides outside the address space (and architectural state) of the process. The specific data varies depending on operating system and implementation decisions: for instance, Zhuque tracks the state of open Linux file descriptors in PMEM and restores them at restart using system calls. We discuss Zhuque's handling of kernel-resident state in Section 5.

Architectural state Any state stored in the processor itself and directly accessible from software. This state is per-thread, and since it includes the program counter and stack pointer registers, restoring it is equivalent to restarting execution of the thread (so it must be done last).

Zhuque manipulates the saved PC and stack so that the thread resumes as if it had just called the application-defined failure handler (if it exists), and then that handler returns to the point interrupted by execution when it executes a `RET` instruction. To avoid references to a thread which has not yet been recreated, threads wait to restart execution after they are created until all threads have been created.

5 Implementation

Zhuque is based on the musl implementation of the C standard library runtime [46], plus a minor modification to the Linux kernel (Section 5.3). Figure 3 depicts Zhuque's place in the runtime environment, and Figure 4 shows the changes to control flow at initialization and termination. This section describes the life cycle of a Zhuque process, describes how Zhuque handles the userspace-kernel boundary, and finally discusses some limitations of our prototype implementation.

5.1 Process Life Cycle

When Zhuque starts a process, it checks an environment variable for a path to a directory which holds or will hold the persistent state for that process. One file in the directory holds the process's global "process context", a memory map of a C

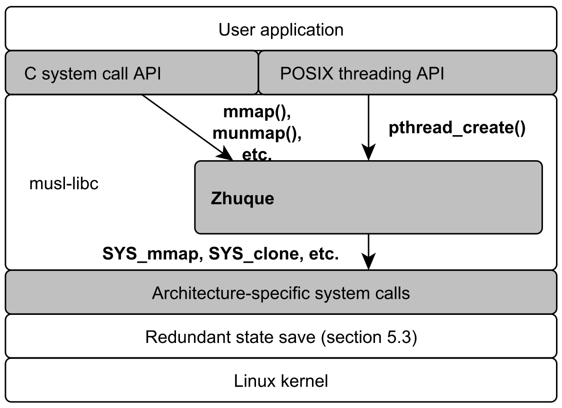

Figure 3: **Zhuque architecture.** User applications link to the C APIs provided by musl libc, and we modify the implementation of the APIs and the arguments passed to the underlying system calls. To protect against failures in kernel mode, we save userspace context to PMEM on entry to the kernel.

structure. The directory also holds all other persistent memory files allocated during the process's life.

Zhuque takes control of the process after the dynamic loader loads its own metadata using information provided by the kernel ("loader bootstrapping"). If the context file is present, then Zhuque takes steps to *restart* the process. If the context file is missing, but the environment variable is set, then it is a newly created Zhuque process (i.e., a *clean start*).

Clean start In the clean start case, Zhuque creates and initializes the process context file. Then, it records the locations of the dynamic loader and the main binary in the mapping table and remaps their static memory sections to memory backed by a persistent file. This retroactive process is necessary because our userspace runtime cannot interpose on mappings created by the kernel.

Next, control returns to the loader and it loads the application's dynamically-linked dependencies. Our code intercepts the loader's calls to `mmap()` and `mprotect()` during this process in order to record the mapping metadata and transform any writable, private mappings into persistent memory regions.

After loading is complete, control returns to Zhuque just before `main()` executes. Zhuque copies `main()`'s arguments into PMEM and runs it in a new thread with a persistent stack.

Power failure To save volatile architectural state (e.g. the register file) to PMEM at failure, we propose repurposing existing functionality. NVDIMM eADR and CXL GPF both rely on a System Management Interrupt (SMI) to implement the flush-on-fail process on x86 systems (see Section 2.1). SMI handling saves volatile architectural state to a designated per-core region (the *SMRAM*) before beginning execution of the handler, and x86 allows the SMRAM to be PMEM-backed [16]. However, the location of the SMRAM is controlled by system firmware. Unfortunately, updates to firmware must be signed by the manufacturer — the firmware uses encryption to prevent modification by the end user [15], so we were unable to make this change for our prototype.

Instead, to test Zhuque's application support, we emulate the SMI's state save using userspace signals. If SIGPWR is delivered while the process is executing, the volatile thread receives it and sends a second signal to each thread. When the kernel interrupts a thread to run the signal handler, it first pushes the register file and other state needed to resume execution onto the persistent thread stack. The handler body saves the current stack pointer and some context not saved by handler entry in PMEM, and then exits the thread directly, preserving the contents of the stack. Thus, at recovery, we have access to a persistent memory region containing a snapshot of volatile architectural state at failure, as if it had been saved by an SMI.

Restart after failure On restart, the runtime opens the context file, re-creates PMEM mappings, re-opens file descriptors, and finally re-maps file-backed memory. If a file descriptor was closed after being used to create a mapping, it is temporarily re-opened while the mapping is restored.

After the virtual memory map and file set are re-established, Zhuque restarts the execution of each thread from the point of failure. Zhuque does this by starting each thread with the same start routine, and the same initial stack pointer, so that the bottom frames of the stack are overwritten with new frames of the same size, and the contents of application frames are preserved. From this entry routine, we use assembly to restore architectural state, including setting the stack pointer and PC to the addresses saved at failure. Execution resumes within the runtime's failure handler, which calls the user-defined failure handler if present. If there is no user-defined handler, or the handler does not exit the program, execution continues at the point interrupted by the signal at failure.

5.2 Kernel-resident State

In order to preserve correctness in a userspace-only implementation, our runtime tracks and restores two pieces of kernel state tied to the process: the file descriptor set and the thread set.

To track the **thread set**, our runtime interposes on calls to `pthread_create()`, wrapping the passed thread entry point and arguments in our own entry point function (which itself is wrapped in the musl entry point function). It also saves both the Linux and pthreads identifiers for the thread; the Linux ID is used at failure to signal each thread individually with `tgkill()`, while the pthreads ID is the address of the thread metadata, and is used at restoration to continue execution at the point of failure. The Linux ID of a thread changes when the process is restarted, while the pthreads ID does not, because we restart threads at recovery with a modified version of `pthread_create()` which uses an existing thread metadata object rather than creating a new one.

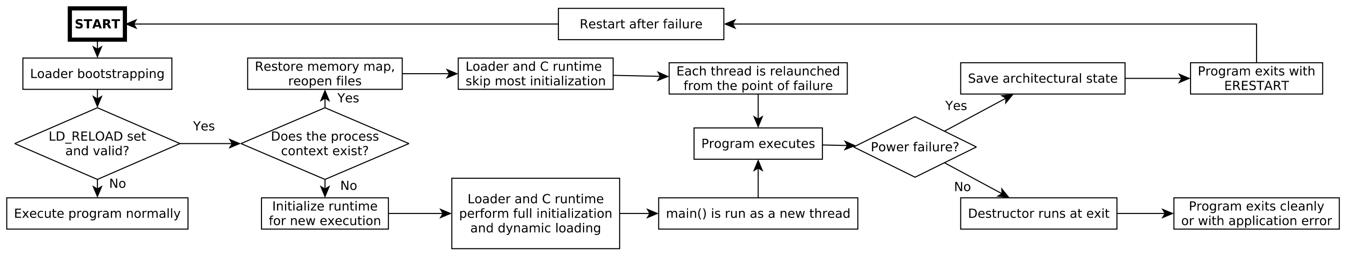
START
Restart after failure
Loader bootstrapping
Restore memory map, reopen files
Loader and C runtime skip most initialization
Each thread is relaunched from the point of failure
Save architectural state
Program exits with ERESTART
Yes
LD_RELOAD set and valid?
Yes
Does the process context exist?
Program executes
Power failure?
Yes
No
No
No
Execute program normally
Initialize runtime for new execution
Loader and C runtime perform full initialization and dynamic loading
main() is run as a new thread
Destructor runs at exit
Program exits cleanly or with application error

Figure 4: **Zhuque runtime control flow.** Zhuque modifies runtime startup and termination; application code is not modified.

To track the **file descriptor set**, Zhuque interposes on calls which assign (e.g. `open()`, `socket()`), modify (e.g. `fcntl()`, `bind()`), or release (i.e. `close()`) file descriptors and replays them at restart, using `dup()` to patch any discrepancy in assigned descriptors. This approach is sufficient for sockets with stateless protocols, `epoll` file descriptors, and simple file accesses.

However, pipes and files require special handling: for regular files, we ensure that they will not be deleted between failure and restoration by creating separate hardlinks to the files and using those to reference the file, deleting them when the program exits cleanly; we also open them without kernel buffering (i.e. with O_SYNC) due to the limits of a userspace implementation. And for pipes, we use the `splice()` family of system calls to save and restore unconsumed contents at failure/restoration. Support for restoring network sockets is best-effort, and a more systematic approach to network support under WPP is an interesting future extension of this work.

5.3 Failures in kernel mode

When an application makes a system call, or is suspended by an interrupt, the kernel will save the application's volatile architectural state on the suspended thread's kernel stack and restore it when application execution resumes. In our implementation the kernel stack is volatile, so we must save this state in persistent memory to allow recovery if power failure interrupts the kernel-mode operation.

To enable this, we added a `prctl()` operation to designate a page as a redundant state save area. We added code on all entries from user- to kernel-mode which checks whether such a page has been provided, and if so saves the state there as well as to the kernel stack. Accesses to the page must not fault: since a page fault is itself an interrupt, a fault in interrupt entry deadlocks the kernel. We found that there is no way (in our test kernel version) to reliably prevent access to a filesystem DAX page from faulting, so we use device DAX to provide the save memory.

5.4 Limitations

There are two notable limitations of our implementation, neither of which is fundamental to the design.

Multi-process applications are not supported. Zhuque currently has no support for persisting multiple processes in the same process tree; if an application under our runtime forks a new process while leaving the `LD_RELOAD` environment variable unchanged, the child process will crash when it attempts to use the same context object as its parent. By the same token, we make no attempt to preserve OS process IDs across failures, so applications that save and retrieve their PID after failure may find it invalid. Our runtime supports unrestricted concurrency schemes, so we believe it would be possible to extend it to support multi-process operation by interposing on the creation and termination of processes, similar to our approach to threads.

Some ASLR is not supported. Address space layout randomization (ASLR) is a security technique which randomizes the address of virtual memory mappings. Random addresses returned by `mmap()` to userspace are not an obstacle, since the randomization only occurs once under Zhuque. However, Zhuque cannot prevent the kernel from mapping libc and the application binary at a random location at restart, which means that their static memory cannot be recreated at the same locations when ASLR is enabled. It would be possible to move those mappings after they are created, but since it does not otherwise affect correctness or performance, and it would add significant complexity to the startup process, we chose not to implement this feature.

6 Evaluation

In this section, we evaluate Zhuque's performance to provide answers to the following questions:

- How much performance improvement does Zhuque provide for persistent applications compared to existing libraries?
- How much performance overhead does Zhuque incur compared to native, volatile execution?
- What benefits does Zhuque provide by enabling zero-

effort persistence?

Our experimental workloads include a set of microbenchmarks, a set of Python benchmarks, and three recoverable applications.

6.1 Evaluation Setup

We compare against native application performance based on musl and the performance of four popular PMEM libraries.

Musl [46] stands for the default (volatile) implementation based on musl libc.

PMDK [50] is Intel's failure atomicity library. It uses hybrid undo-redo log for both failure-atomicity [27] and memory allocation [55]. **Atlas** [6] also uses an undo logging-based mechanism for failure-atomicity. It can automatically infer failure-atomic region boundaries by analyzing lock behaviors in application code. **Clobber-NVM** [68] is a state-of-the-art PMEM library. It records clobber_log and v_log during runtime, and recovers an application by re-executing any interrupted transactions. **Mnemosyne** [63] is a redo-log based system. Instead of relying on locks in user applications, Mnemosyne uses the C++ transactional memory model to parallelize code.

To measure their performance with flush-on-fail semantics, we removed the flushes and fences from all three comparison libraries. These *flush-on-fail (FoF)* versions are used for all experiments described below, unless otherwise noted.

We run the benchmarks on a platform with one 20-core Intel Xeon Gold 6230 processor, running at 2.1 GHz. The platform has a total of 96 GB of DRAM and 768 GB (6 ×128 GB) of Intel Optane DC Persistent Memory directly attached to the processor [28]. We configured our test machine such that Optane DCPMM is in 100% App Direct mode [2]. In this mode, software has direct, byte-granularity access to the Optane DCPMM. Zhuque uses DAX-mapped [40] `Ext4` files for all PMEM allocations except the kernel state save area, which uses device DAX for the reasons described in Section 5.3. Unless otherwise noted, all Zhuque experiments are run on the modified kernel with the state save area activated, while all comparison software is run on an unmodified kernel of the same version (Ubuntu kernel 4.15.0-169). Each data point reported is the average over five runs.

We observed variations in memory usage across different benchmarks, ranging from 20 MB to 1 GB. For all but two of the Python benchmarks, the memory usage exceeds the Last Level Cache (LLC) capacity of 30.25 MB.

As discussed in Section 5.1, we were unable to modify platform firmware to direct the SMI state save to PMEM, because firmware updates must be signed by the manufacturer [15]. However, since it only affects events at failure, we expect that making this change would produce no measurable changes in the steady-state performance results we report below.

We verified that, with Zhuque enabled, all benchmark applications restarted and ran to completion correctly after randomly-timed simulated power failures.

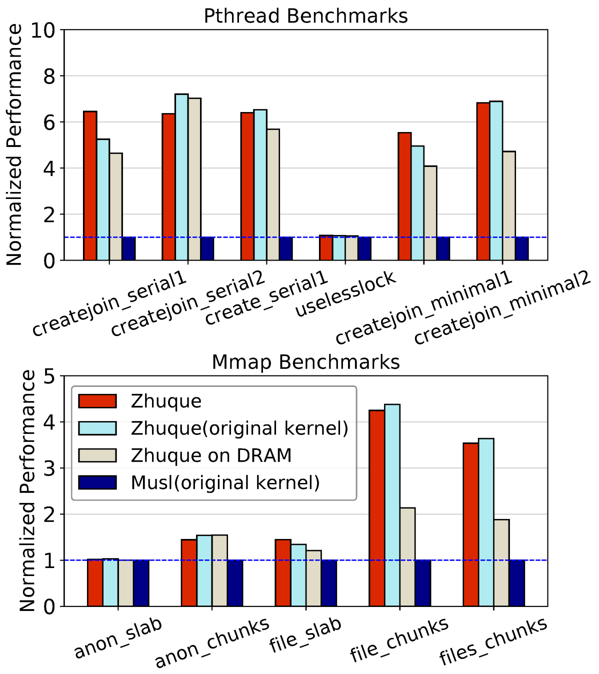

Figure 5: **Measuring the overhead of Zhuque on basic operations.** Performance values are normalized to the Musl(original kernel) value.

6.2 Microbenchmarks

In our first experiment, we compare Zhuque's performance with the default (volatile) musl libc on a set of microbenchmarks from the libc-bench [37] benchmark suite which tests the operations modified by Zhuque: thread creation and virtual memory mapping. We implemented the latter within the libc-bench test harness. Descriptions of the `pthread_create()` benchmarks can be found in the libc-bench documentation [37]. The `mmap()` benchmarks are described below.

anon_slab creates a 16 MB anonymous mapping, writes to every page, and then removes the mapping. **anon_chunks** creates, writes to every page, and then removes 128 128 kB anonymous mappings. **file_slab** creates a 16 MB file-backed mapping, writes to every page, and then removes the mapping. **file_chunks** creates, writes to every page, and then removes 128 128 kB mappings backed by a 16 MB file.
files_chunks creates, writes to every page, and then removes 128 128 kB mappings each backed by a separate 128 kB file.

We ran these benchmarks on four implementations: **Zhuque** is Zhuque using DAX-mapped PMEM as the backing store, running on modified kernel. **Zhuque(original kernel)** is Zhuque without kernel modification. **Zhuque on DRAM** uses non-PMEM files (loads and stores access the DRAM page cache) as a backing store, but also saves kernel

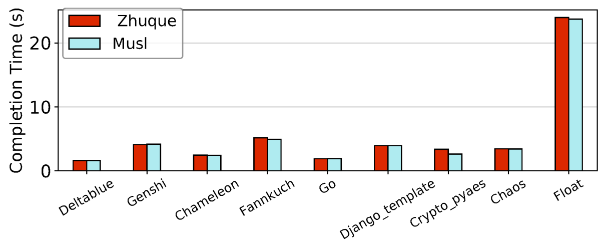

Figure 6: **Measuring the overhead of different Python benchmarks**: Zhuque matches native performance on some benchmarks

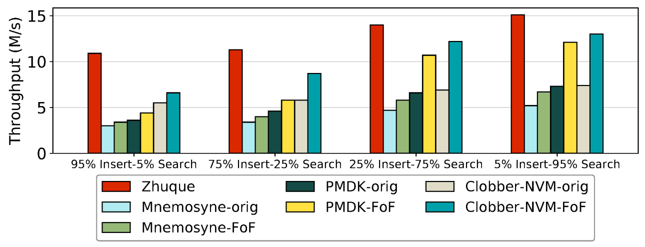

Figure 7: **Porting Existing PMEM Libraries to Flush-on-Fail Machines Provides up to 1.76× Improvement**

states. **Musl(original kernel)** is unmodified musl libc with unmodified kernel. We report the results in Figure 5.

We observe that Zhuque introduces significant overheads to modified operations, especially `mmap()` and `munmap()` – the bottleneck for all of the poorly-performing microbenchmarks is the allocation of new anonymous memory. The overhead is per-operation, and does not depend on the size of the mapping. This is not surprising: our modified versions still perform the original operations, and are also required to modify userspace data structures maintained by Zhuque in persistent memory, often including a search whose time is linear in the number of created mappings. As demonstrated by the results below, these overheads do not translate to a significant slowdown on macrobenchmarks, because modifications of the virtual memory map are rarely on an application's critical path. In addition, these overheads are a result of implementation choices, not the fundamental design. We anticipate they would decrease substantially in a kernel-based implementation of WPP.

6.3 Python Benchmarks

In this experiment, we ran nine Python benchmarks on the musl and Zhuque configurations using the CPython interpreter. It demonstrates that Zhuque can run a wide range of unmodified Python applications. We chose the first nine benchmarks (out of 42), in alphabetical order, from version 1.0.0 of the Pyperformance benchmark [21] suite. Descriptions of the benchmarks can be found in the Pyperformance documentation [21]. We ran them in our own benchmark framework, but did not modify the benchmarks themselves. We also added dynamic thread stack support, not present in vanilla musl, in order to support the large stack sizes required by the interpreter. We report the results in Figure 6.

Most of the Python benchmarks perform competitively with the musl versions. Those that perform worse generally incur overhead from frequent random-access reads and writes to data structures too large to fit in the cache, causing thrashing and exposing the higher access latency of persistent memory compared to DRAM.

6.4 Memcached

Memcached [44] is a widely-deployed key-value store. Early versions (1.2.*) have been ported to Mnemosyne, PMDK, and Clobber-NVM. We ran memcached-1.2.5 on Zhuque unmodified.

We evaluate memcached performance with four types of workloads: insertion-intensive (95% insertion / 5% search), insertion-mostly (75% insertion / 25% search), search-mostly (25% insertion / 75% search), and search-intensive (5% insertion / 95% search). We use memslap [39] to generate a stream of uniformly distributed memcached requests with 16-byte keys and 64-byte values. As shown in Figure 7, Mnemosyne, PMDK and Clobber-NVM can perform up to 1.76× faster with flushes and fences removed. This result indicates that the flush-on-fail semantics can benefit existing PMEM libraries.

Figure 8 presents the results of these experiments, using Musl-based memcached-1.2.5 performance on DRAM as baseline. We find Zhuque can always provide nearly 80% of musl memcached performance. Across different thread configurations, Zhuque provides up to 3.58× Mnemnosyne's throughput, 1.81× of PMDK's throughput and 1.71× of Clobber-NVM's throughput.

Poor scalability is a well-known problem with early versions of memcached [20, 42]. Memcached went through a rewrite of the synchronization framework to use fine-grained locking across seven years of development and over thirty versions [31]. Many current (transactional) PMEM libraries have strict requirements for applications' concurrency schemes. These requirements make converting recent versions of memcached to run on PMEM a complicated and difficult process. Zhuque places no restrictions on the locking scheme, so the newest version (1.6.17) can run unmodified on Zhuque. By simply running the newest version on Zhuque, we can provide more than 7.5× performance of the best performant older version of persistent memcached on the same workload, with the same thread count.

6.5 Vacation and Yada

Furthermore, we evaluated the performance of the Vacation and Yada applications from the STAMP benchmark suite [8],

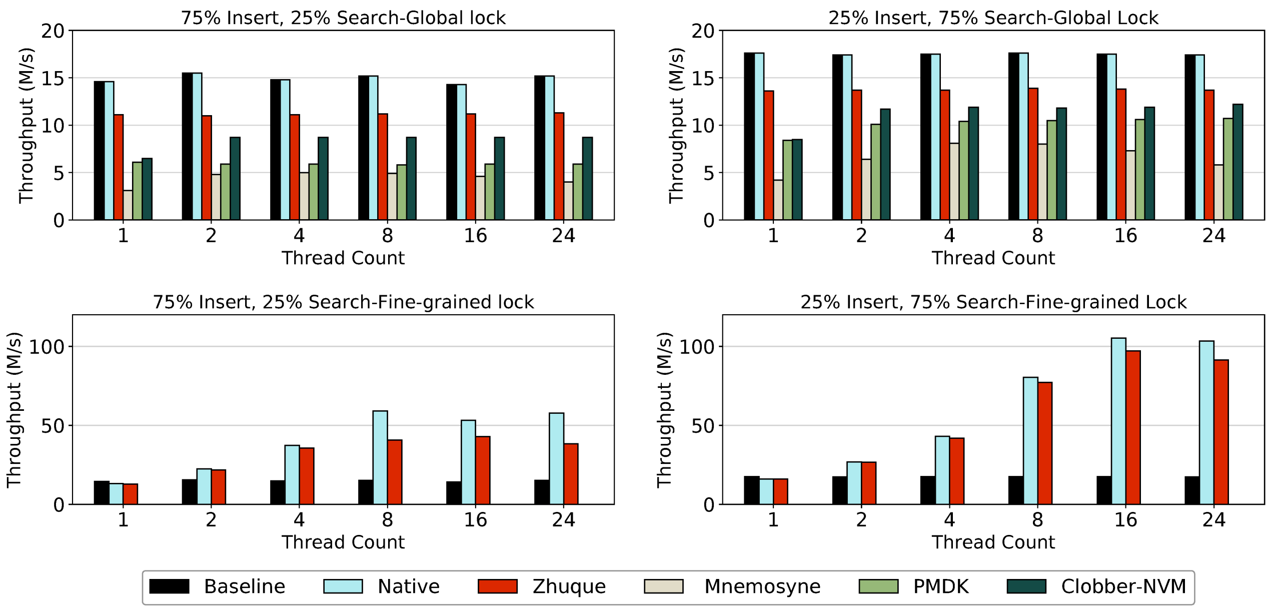

Figure 8: **Zhuque Enables Newest Version of Memcached to Run on PMEM, Provides Significantly Better Performance**

each targeting common PMEM applications such as, respectively, KV-stores and graph workloads. Prior works [25,63,68] provide readily available implementations of these applications built on top of existing PMEM libraries, with the exception of Yada - Mnemosyne.

We compare Zhuque with Musl, Mnemosyne, PMDK, Atlas and Clobber-NVM Vacation performance. Vacation is a travel reservation system, consisting of tables updated concurrently using transactions that span multiple tables.

Prior implementations [25, 63, 68] persist the tables in PMEM, and leave the client side in volatile memory. Zhuque persists the entire vacation application, including both the server tables and the client threads.

Figure 9 shows that Zhuque performs 10.8×, 4.8×, 3.7× and 3.6× faster on Vacation than Mnemosyne, PMDK, Atlas and Clobber-NVM, respectively. The performance gain comes from fewer logging writes and more efficient memory management: Zhuque uses vanilla `malloc()`, while the other libraries use either PMDK's transactional allocator (PMDK, Clobber-NVM) or a hand-written allocator (Mnemosyne, Atlas). The memory management efficiency problem is more prominent on Yada, which implements Ruppert's algorithm for Delaunay mesh refinement [54].

7 Related Work

PMEM Software & Hardware. In the past decade, researchers have designed many highly-optimized PMEM data structures [7, 10, 22, 48, 62, 69]. They are designed to reduce the cost of persistent updates while ensuring failure-atomicity. Because they are carefully designed by experts to cope with PMEM characteristics, they usually provide good performance. However, using and developing these data structures takes significant programming effort.

Because PMEM's bandwidth is significantly higher than traditional secondary storage, PMEM file systems [11, 33, 65, 67] aim to expose raw PMEM performance as much as possible. It is easy for existing applications to use files on these PMEM file systems, but they are not designed to solve the same problem that Zhuque targets (failure-resilience of the application's in-memory data).

The architecture community has also sought better hardware support for PMEM, often by allowing more permissive (and thus performant) store ordering at the memory controller [3,18,24,32,34–36,59]. Many of these systems demonstrate dramatic performance gains in simulation, but none that we are aware of are available in production.

General-purpose PMEM libraries. General-purpose PMEM libraries aim to ensure failure-atomicity for applications which directly access PMEM, with low overhead and minimal code changes.

Traditional undo-log systems [6, 9, 50] and redo-log systems [23, 63] write to a log alongside every visible update, at least doubling each write. On non-flush-on-fail PMEM machines, undo-logging usually requires expensive memory fences at the end of each log write, while redo-logging needs to redirect loads even if the machine supports flush-on-fail.

To avoid expensive synchronization between threads, some undo/redo systems buffer writes in "shadow" copies of PMEM data [5, 17, 41, 43, 66], at least doubling the amount of PMEM required by the application. These systems still incur the write amplification and read redirection costs of conventional log-

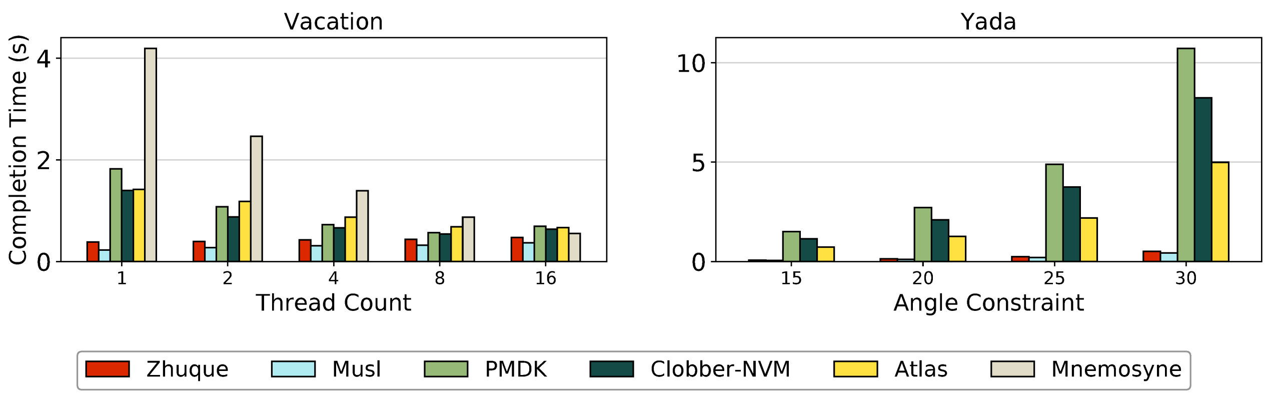

Figure 9: **Vacation and Yada performance on different PMEM libraries and Zhuque**

ging, and with flush-on-fail semantics the synchronization they are designed to reduce is no longer required at all. Compared to these systems, Zhuque does not amplify or redirect memory accesses, nor increase the size of the working set.

All these systems rely on either lock-inferred failure atomic sections (FASEs) [6, 26, 30, 42, 66], classical transactions [5, 41, 43, 63], or programmer delineated transaction boundaries with a restricted lock scheme [23, 50, 68] to identify failure-atomic operations. In contrast, Zhuque is not concerned with synchronization: it uses the same concurrency model as the original application.

JUSTDO [30], iDO [42], and Clobber-NVM [68] recover by resuming execution of the interrupted failure-atomic section or re-executing interrupted transactions. These systems are similar to Zhuque in that they also resume execution at the point of failure, but they all restrict concurrency and require manual annotation of atomic sections.

Single Level Stores. WPP transparently makes processes persistent by providing continuous checkpointing, durability guarantees for external observers of application IO (known as *external synchrony* [49]), and POSIX compatibility. Single level stores (SLS) [12, 38, 56–58, 60, 61] also provide persistent address spaces to applications, but they suffer from high overhead, rarely support external synchrony, and are often hard to use due to custom APIs [57, 60].

The high overhead arises from checkpointing and, for some systems, the enforcement of external synchrony. Checkpointing overhead is high because traditional storage devices are far slower than DRAM, and SLSes usually amplify writes by tracking memory modification at page granularity. To achieve external synchrony, SLS systems must delay IO until data is safely persisted. If checkpoints are too frequent, the communication delay can cause high overheads.

Because of the performance penalty of providing external synchrony, most SLSes choose not to enforce it. The state-of-the-art SLS system Aurora [61] points out the value of external synchrony, but does not support it. Zhuque shows that flush-on-fail semantics allow continuous checkpointing, making external synchrony feasible and performant.

8 Conclusion

This paper has described Whole Process Persistence (WPP), a programming model that treats power failure as a recoverable exception. Zhuque, implementing WPP, transparently makes applications failure-resilient by interposing on the POSIX-specified APIs. Zhuque ensures process state survives power failures, and allows for resumption of execution.

Compared with existing solutions, Zhuque greatly simplifies the programming model. Our evaluation shows that Zhuque significantly improves performance compared to state-of-the-art, and tends to match original volatile application performance on certain workloads.

Acknowledgments

This work was supported in part by the NSF/Intel Foundations of Microarchitecture Program awards 2011213 and 2011212. We would like to thank our shepherd Gaël Thomas and the anonymous reviewers for their insightful feedback.

References

[1] eadr characteristics at failure. https://groups.google.com/g/pmem/c/K35X70fzAMw/m/5qEhhzb8AAAJ, 2021.

[2] Alper Ilkbahar. Intel Optane DC Persistent Memory Operating Modes Explained, 2018.

[3] Miao Cai, Chance C Coats, and Jian Huang. Hoop: Efficient hardware-assisted out-of-place update for non-volatile memory. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 584–596. IEEE, 2020.

[4] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Sid-

dhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Commun. ACM*, 51(11):40–46, November 2008.

[5] Daniel Castro, Paolo Romano, and João Barreto. Hardware transactional memory meets memory persistency. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 368–377, 2018.

[6] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud 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, pages 433–452. ACM, 2014.

[7] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[8] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46, 2008.

[9] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven 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 '11, pages 105–118, New York, NY, USA, 2011. ACM.

[10] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. Fine-grain checkpointing with in-cache-line logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 441–454. Association for Computing Machinery, 2019.

[11] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[12] George Copeland, Michael Franklin, and Gerhard Weikum. Uniform object management. In *International Conference on Extending Database Technology*, pages 253–268. Springer, 1990.

[13] Intel Corporation. Asynchronous event handling. In *CXL Type 3 Memory Device Software Guide*, page 65. June 2021. Revision 1.0.

[14] Intel Corporation. Gpf sequence. In *CXL Type 3 Memory Device Software Guide*, page 121. June 2021. Revision 1.0.

[15] Intel Corporation. Microcode update facilities: Update signature and verification. In *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3, chapter 10.11, pages 10–36–10–37. March 2023. Order No. 325462-079US.

[16] Intel Corporation. System management mode: Smram. In *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3, chapter 32.4, pages 32–4–32–9. March 2023. Order No. 325462-079US.

[17] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 271–282. Association for Computing Machinery, 2018.

[18] Mahesh Dananjaya, Vasilis Gavrielatos, Arpit Joshi, and Vijay Nagarajan. Lazy release persistency. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 1173–1186. Association for Computing Machinery, 2020.

[19] Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. Adaptive integration of hardware and software lock elision techniques. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 188–197. Association for Computing Machinery, 2014.

[20] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing numa locks. *ACM Trans. Parallel Comput.*, 1(2), February 2015.

[21] Python Software Foundation. The python performacne benchmark suite, 2021.

[22] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 28–40. Association for Computing Machinery, 2018.

[23] Ellis R Giles, Kshitij Doshi, and Peter Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2015.

[24] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M Chen, Satish Narayanasamy, and Thomas F Wenisch. Relaxed persist ordering using strand persistency. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 652–665. IEEE, 2020.

[25] Swapnil Haria, Mark D. Hill, and Michael M. Swift. Mod: Minimally ordered durable datastructures for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, AS-

PLOS '20, pages 775–788. Association for Computing Machinery, 2020.

[26] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 468–482. Association for Computing Machinery, 2017.

[27] Intel Corporation. Pmdk issues: introduce hybrid transactions, 2017.

[28] Intel Corporation. Intel Optane DC Persistent Memory, 2019.

[29] Intel Corporation. eADR: New Opportunities for Persistent Memory Applications, 2021.

[30] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 427–442, New York, NY, USA, 2016. ACM.

[31] Joseph Izraelevitz, Lingxiang Xiang, and Michael L Scott. Performance improvement via always-abort htm. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 79–90. IEEE, 2017.

[32] Jungi Jeong and Changhee Jung. Pmem-spec: persistent memory speculation (strict persistency can trump relaxed persistency). In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 517–529, 2021.

[33] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 494–508, 2019.

[34] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 399–411. Association for Computing Machinery, 2016.

[35] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. Delegated persist ordering. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

[36] Kunal Korgaonkar, Joseph Izraelevitz, Jishen Zhao, and Steven Swanson. Vorpal: Vector clock ordering for large persistent memory systems. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 435–444. Association for Computing Machinery, 2019.

[37] Eta Labs. libc-bench, 2021.

[38] Charles R Landau. The checkpoint mechanism in keykos. In *[1992] Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pages 86–91. IEEE, 1992.

[39] libMemcached.org. libMemcached, 2011.

[40] Linux Kernel Organization. Direct Access for Files, 2020.

[41] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 329–343. Association for Computing Machinery, 2017.

[42] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270. IEEE, 2018.

[43] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 499–512. Association for Computing Machinery, 2017.

[44] Memcached. http://memcached.org/.

[45] Transactional memory study group (SG5). Technical specification for c++ extensions for transactional memory iso/iec ts 19841:2015, 2015.

[46] musl libc, 2021. https://musl.libc.org/.

[47] Dushyanth Narayanan and Orion Hodson. Whole-system persistence with non-volatile memories. In *Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*. ACM, March 2012.

[48] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B Morrey III, Dhruva R Chakrabarti, and Michael L Scott. Dalí: A periodically persistent hash map. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[49] Edmund B Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. Rethink the sync. *ACM Transactions on Computer Systems (TOCS)*, 26(3):1–26, 2008.

[50] pmem.io. Persistent Memory Development Kit, 2017. http://pmem.io/pmdk.

[51] Yoav Raz. The principle of commitment ordering, or

guaranteeing serializability in a heterogeneous environment of multiple autonomous resource mangers using atomic commitment. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 292–312. Morgan Kaufmann Publishers Inc., 1992.

[52] Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. Transactionalizing legacy code: An experience report using gcc and memcached. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 399–412. Association for Computing Machinery, 2014.

[53] Andy Rudoff, Chet Douglas, and Tiffany Kasanicky. Persistent memory in cxl. In *Proceedings of the 2021 SNIA Persistent Memory + Computational Storage Summit*, April 2021.

[54] J. Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms*, 1995.

[55] Steve Scargall. *PMDK Internals: Important Algorithms and Data Structures*, pages 313–331. Apress, 2020.

[56] Jonathan S Shapiro and Jonathan Adams. Design evolution of the eros single-level store. In *USENIX Annual Technical Conference, General Track*, pages 59–72, 2002.

[57] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. Eros: a fast capability system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 170–185, 1999.

[58] Eugene Shekita and Michael Zwilling. Cricket: A mapped, persistent object store. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1990.

[59] Seunghee Shin, James Tuck, and Yan Solihin. Hiding the long latency of persist barriers using speculative execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 175–186. Association for Computing Machinery, 2017.

[60] Frank G Soltis. *Fortress Rochester: The Inside Story of the IBM iSeries*. System iNetwork, 2001.

[61] Emil Tsalapatis, Ryan Hancock, Tavian Barnes, and Ali José Mashtizadeh. The aurora operating system: revisiting the single level store. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 136–143, 2021.

[62] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 5. USENIX Association, 2011.

[63] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS '11: Proceeding of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2011. ACM.

[64] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., 2001.

[65] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11, New York, NY, USA, 2011. ACM.

[66] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel 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, pages 623–637. Association for Computing Machinery, 2020.

[67] Jian Xu and Steven Swanson. {NOVA}: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pages 323–338, 2016.

[68] Yi Xu, Joseph Izraelevitz, and Steven Swanson. Clobber-nvm: Log less, re-execute more. In *To appear in the Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[69] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 167–181. USENIX Association, 2015.

[70] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the tires of software transactional memory: Why the going gets tough. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 265–274. Association for Computing Machinery, 2008.

[71] Lu Zhang and Steven Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 897–912, 2019.

A Proof

In this appendix, we provide a proof of theorem 3.1:

Theorem 3.1 (FASE Limitation) *There exist applications for which, in order to consistently recover from a crash, a* *reasonably permissive FASE-based failure atomicity system requires all volatile program state be available at recovery.*

A.1 Definitions

We begin by defining terms. By *application* we mean a multi-threaded program, executed as a *process*. The process's *internal state* consists of all its data, including heap, globals, and stack. Some memory locations are designated *nonvolatile*, their contents (the *nonvolatile state*) survive a power outage; the remainder are *volatile*, and their contents (the *volatile state*) are lost. The process may perform IO operations — we term the set of IO operations performed by an executing process its *external state*. The process, being multi-threaded, contains code regions that execute while a lock is held, these are termed *critical sections*.

If power is lost during process execution, its volatile state is lost. The purpose of a *failure atomicity system* is to provide *consistent recovery* from a power outage. For consistent recovery, the system selects a point in execution, termed the *recovery point*. The recovery point is *consistent* with the external state; process execution from initialization through the recovery point would generate the observed IO. For failure atomicity, the recovery point also lies outside all critical sections. Consistent recovery of a process consists of selecting a valid recovery point and restoring the persistent state's contents to its values as of this point. If the power failure interrupts a critical section, consistent recovery will involve, for failure atomicity, chosing a recovery point outside the critical section and undoing or redoing changes made within the section.

We assume a powerful failure atomicity system which is free, during pre-crash execution, to intercept the process at any point and log data in nonvolatile memory. After a crash, the system has access both to these logs and the process's nonvolatile state — its task is to ensure that the nonvolatile state is restored to a recovery point; consistent with IO operations and outside any critical section. The failure atomicity system must be *reasonably permissive* with respect to its programming model — we require the system's programming model to support our adversarial example. To all our knowledge, all existing FASE-based systems are "reasonably permissive".

Figure 1 gives our counterexample. The "trick" is that the long FASE executed by thread 1 (lines 6 through 22) is dependent on non-FASE code executed by thread 2 that contains both IO and accesses to large volatile data (lines 36 through 39).

A.2 Proof Sketch

We prove Theorem 3.1 by contradiction. We consider a process executing the code sample in Figure 1 and suffering a power failure on line 38. Suppose, for contradiction, there exists a FASE system for which, given this situation, could restore the program's nonvolatile state to a recovery point consistent with the external state and outside any critical section. As thread 1, by construction, executes a critical section (FASE) for its duration, our recovery point for thread 1 must lie at line 6 or line 22 — all other points violate failure atomicity. We consider both options.

Suppose the recovery point lies at line 6 (i.e. recover `x` to 0), it is inconsistent with the external process state due to the IO executed before the failure on lines 36 and lines 37, which indicate that thread 2 (and therefore thread 1) have progressed beyond this recovery point, leading to a contradiction.

Suppose the recovery point lies at line 22 (i.e. recover `x` to `s4`). First we note that the value `s4` has a true dependence (read-after-write) on `s3`, and `s3` has a true dependence on both the inputed seed `in` and large volatile array `Q`. Since `s3` cannot be computed before `in` is known, `s3` must be computed after the `scanf` on line 37 is executed. Since the failure can interrupt the computation of `s3` after the `scanf`, all inputs to `f3` must be preserved in nonvolatile storage for recovery. However, since `Q` is an arbitrarily sized volatile array, Q can be of any size and can be replaced, without loss of generality, with any or all of the program's volatile state, requiring the failure atomicity system to preserve all volatile process state and leading to a contradiction.

B Artifact Appendix

B.1 Abstract

This appendix describes the artifact submitted with this paper. The artifact contains files to build a Docker image with Zhuque installed as the system `libc`, and containing all benchmarks and comparison PMEM systems evaluated in Section 6. It also contains our patch against the Linux kernel necessary for correct resumption, described in Section 5.3.

B.2 Scope

The artifact allows verification of the following claims:

- All performance results from Section 6, for both Zhuque and comparison systems.
- Zhuque can successfully restart programs after an simulated asynchronous failure, as described in Section 5.1.
- The kernel modification correctly saves userspace architectural state to the redundant state save area, as described in Section 5.3.

The artifact does not verify the following claims:

- The kernel modification is sufficient to protect against a failure in kernel mode (we cannot simulate this type of failure).
- The formal claims made in Section 3 about FASE-based systems.

B.3 Contents

The artifact is organized into these key directories (see README for detailed listing):

- `musl-src`: Source code of Zhuque-musl.
- `musl-src/src/psys`: Zhuque core implementation.
- `clobber-pmdk`: Source code for comparison PMEM systems and their versions of application benchmarks.
- `apps`: Zhuque/native implementations of application benchmarks.
- `pigframe`: Materials to build and test our kernel modification.

B.4 Hosting

This artifact is hosted in a Github repository at https://github.com/georgehodgkins/Zhuque_artifact. The commit ID for the current version is `ffc033972bb36adc23b7a4b8c8b2cc6d736bff53`. See README for build instructions.

B.5 Requirements

The only software required for the artifact is Docker on a Linux kernel; the build process bootstraps all other dependencies. Zhuque and most comparison applications can be run on a system without PMEM, but only a system with PMEM can fully reproduce the reported results. Zhuque was mostly developed against a rather old kernel version (4.15.18), and we have sometimes observed unexpected behavior when running on newer kernels.

We built and tested the artifact on the evaluation machine described in Section 6. Our kernel modification targets the Ubuntu kernel fork at version 4.15.0-169. The Docker image is based on Alpine Linux 3.14.