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

1. Motivation

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 allows an application's in-memory state to survive across system reboots and unexpected power failures, but leveraging this capability is not simple. The contents of traditional 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.

Programming systems to help address the challenges of persistent memory programming have proliferated over the last decade. Unfortunately, due to volatile caches, most current solutions impose significant performance overhead and are based on fundamentally limited programming models.

However, the advent of PMEM devices supporting flush-on-fail semantics (such as eADR for NVDIMMs or GPF for CXL devices) means that caches are, in properly configured systems, effectively persistent [7]. Our work demonstrates the potential these systems offer for much simpler, faster PMEM programming models.

2. Limitations of the State of the Art

Broadly, three families of persistent memory programming models have emerged, and each takes a different approach to what consistency is and how the system achieves it.

The first and largest family requires programmers to access persistent state only through well-defined atomic operations, often called transactions. This provides a clean notion of consistency, but transactional approaches have fundamental incompatibilities [2] with existing legacy multithreaded code and have never gained significant traction in real systems.

The second family of systems is based on FASEs [3], regions of code protected by locks, as atomic regions for PMEM updates. Legacy code can run with minimal changes, but FASE-based system have to deal with arbitrary locking schemes. This leads to fundamental weaknesses arising from external IO. Addressing these weaknesses either cripples the system or reduces it essentially to a transaction-based system.

The third family of systems takes the more dramatic step of making *everything* in the system persistent via whole system persistence (WSP) [9]. WSP provides the conceptually simplest programming model: nothing needs to be rewritten and, from the program's perspective, crashes never occur. WSP has faced two challenges: First, making all of memory persistent has until recently been infeasible, because regularly flushing volatile caches to PMEM create enormous performance overheads. Second, making the *whole* system persistent would require a far-reaching redesign of many system components, for an unclear benefit.

3. Whole Process Persistence

However, we think that WSP-style persistence is due for a renaissance. New PMEM devices and platforms support automatically flushing the cache hierarchy upon power failure [7, 5]. For these systems, the caches are effectively persistent, removing the main performance argument against WSP. Then, to address the second argument, we narrow the scope to a single process, yielding *Whole Process Persistence (WPP).*

WPP provides a simple abstraction to the process: its entire memory is PMEM and will survive a power outage. When the process is restarted after a power failure, it receives a signal, which it can ignore or handle with a signal handler. If the signal is ignored, or if the signal handler does not exit, each thread continues execution at the point where it was interrupted by the failure.

There are several benefits to this model over previous work. Most importantly, WPP avoids the problems with FASE- and transaction-based models by discarding the concept of a failure-atomic section. To flush caches at failure, GPF and eADR both use a System Management Interrupt (SMI) which respects architectural semantics, so the effects of an instruction on a process's in-memory state are guaranteed to survive a failure from the point at which they are visible to other threads [6, 1, 4].

Further, 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. Moreover, 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 lock-inferred systems.

4. Zhuque Runtime

To implement WPP, we developed Zhuque, a modified version of musl-libc. libc provides C bindings for system calls that allocate system resources including memory, file

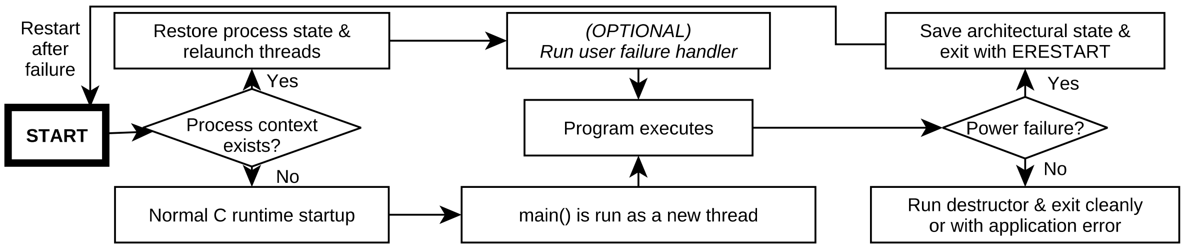
Restart after failure
START
Process context exists?
Yes
No
Restore process state & relaunch threads
Normal C runtime startup
(OPTIONAL) Run user failure handler
Program executes
main() is run as a new thread
Power failure?
Yes
No
Save architectural state & exit with ERESTART
Run destructor & exit cleanly or with application error

Figure 1: Life cycle of a Zhuque process.

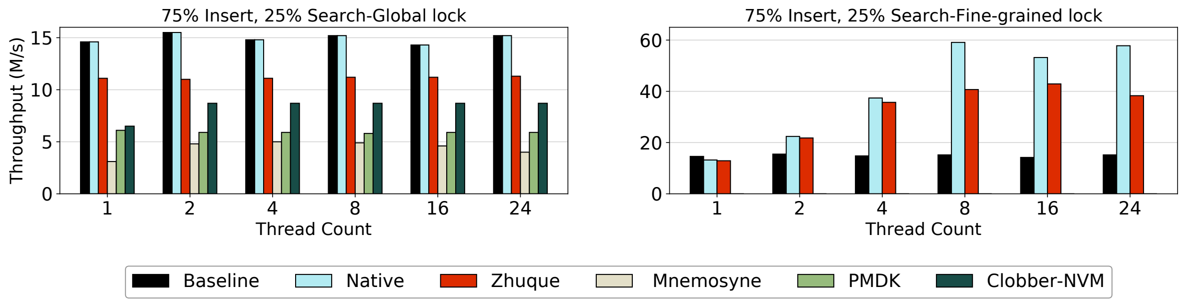

Figure 2: Zhuque Enables Newer Version of Memcached to Run on PMEM, Provides Significantly Better Performance

descriptors, and threads; Zhuque interposes on these bindings in userspace, and modifies the dynamic loader (part of libc), in order to provide WPP functionality. Figure 1 shows the process lifecycle under Zhuque.

These changes allow Zhuque to ensure that all of the application's state is stored in persistent memory, and 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 persistent memory at failure. This step could be achieved with a simple change to the SMI setup (pointing the SMBASE to a PMEM region) [6], but since we do not have access to the signing key required to install modified platform firmware, we are unable to make that change and are required to emulate this capability with userspace signal handlers.

5. Preliminary Results

We evaluate Zhuque's performance against other state-of-the-art PMEM libraries on several applications, and we evaluate its usability by making Python programs persistent without modification. We removed all cache flushes from the other libraries, to better model their performance on a flush-on-fail system. Although Zhuque emulates the state save at failure, we believe that an implementation with a non-emulated save would have substantially similar performance to our results, because the emulation only affects events at failure, not during normal execution.

Figure 2 shows the performance of two versions of memcached on Zhuque and other PMEM libraries, compared to a volatile baseline. Memcached's synchronization framework was rewritten to use fine-grained locking across seven years of development and over thirty versions [8]. Most PMEM libraries have strict requirements for the underlying concurrency strategy, making converting recent versions of memcached to run on PMEM a complicated 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 (right-side), we can provide 7.5× the throughput of the older version (left-side) of persistent memcached with the same workload and thread count.

References

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

[2] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Commun. ACM*, 51(11):40–46, November 2008.

[3] 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, New York, NY, USA, 2014. ACM.

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

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

[6] Intel Corporation. System management mode. In *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3, chapter 31.4, pages 31–4–31–9. April 2022. Order No. 325462-077US.

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

[8] 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.

[9] 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.