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

George Hodgkins*, Yi Xu*, Steven Swanson, Joseph Izraelevitz

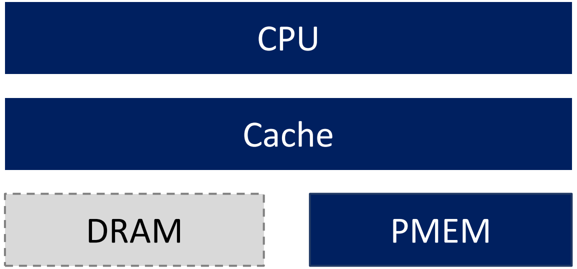
*co-first authors

University of Colorado, Boulder

University of California, San Diego

Overview

```
$ CRASH_RESISTANT=1 ./mycomputation
```



CPU

Cache

DRAM

PMEM

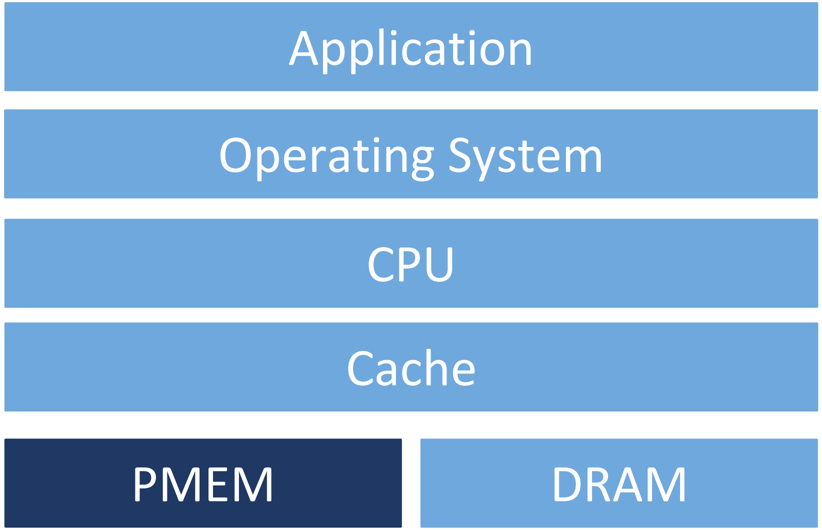
Whole Process Persistence

- **Simple PMEM programming model** for systems with flush-on-fail support (eADR, GPF)
- Our implementation, Zhuque, requires **little or no modification** to native applications
- **>3x mean speedup** over prior works, after removing their cache flushes

Persistent Memories (PMEMs)

PMEM

- **Persistent** across power failures.
- **Byte-addressable** interface.
- **DRAM-class latency and bandwidth.**



Application

Operating System

CPU

Cache

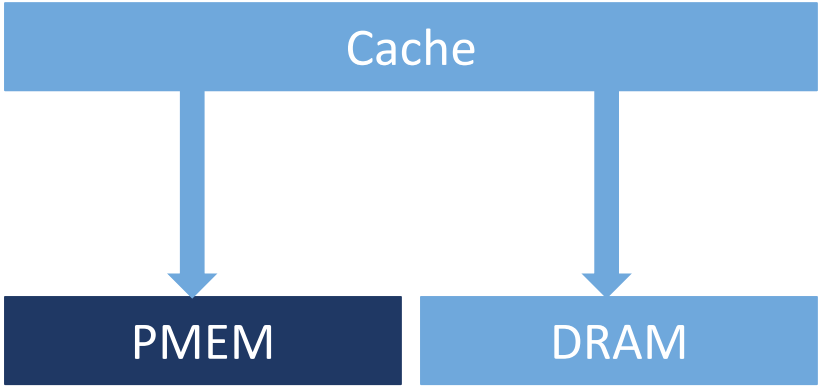
PMEM

DRAM

The challenge

Cache

- The cache has been volatile.
- **Cached updates will be dropped** after a power loss.

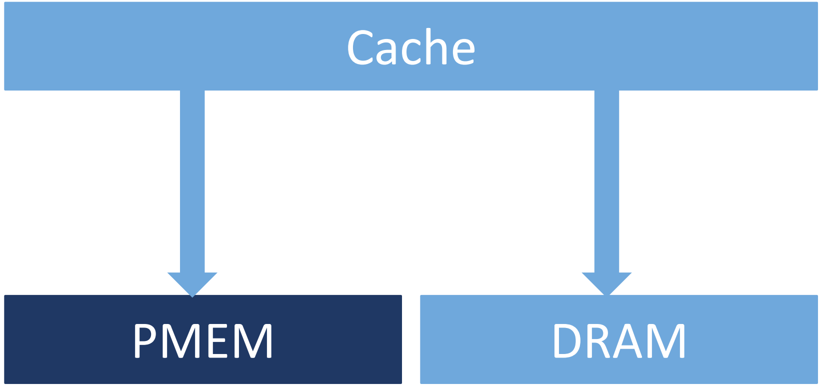


Applications need to explicitly evict cachelines to provide crash consistency.

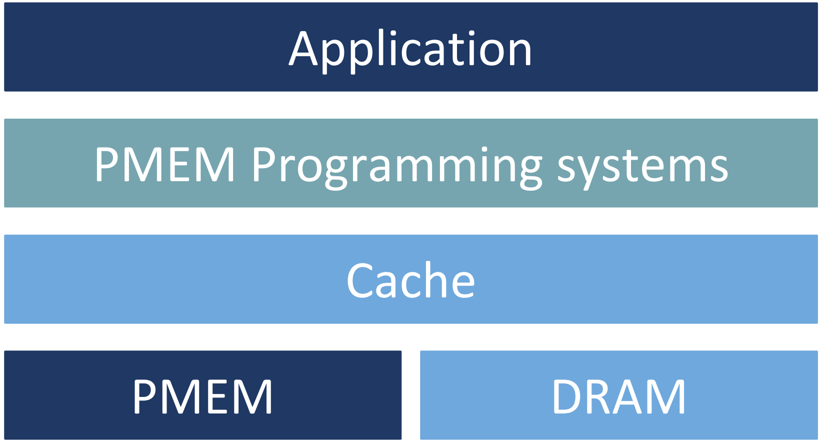
The consequences

Explicit cache flushes

- Explicit flushes **amplify writes to PMEM**
- Correctly placing flushes requires **extra programming effort**
- Required memory barriers incur **pipeline stalls** and **synchronization overhead**



Persistent Memory Programming



Application

PMEM Programming systems

Cache

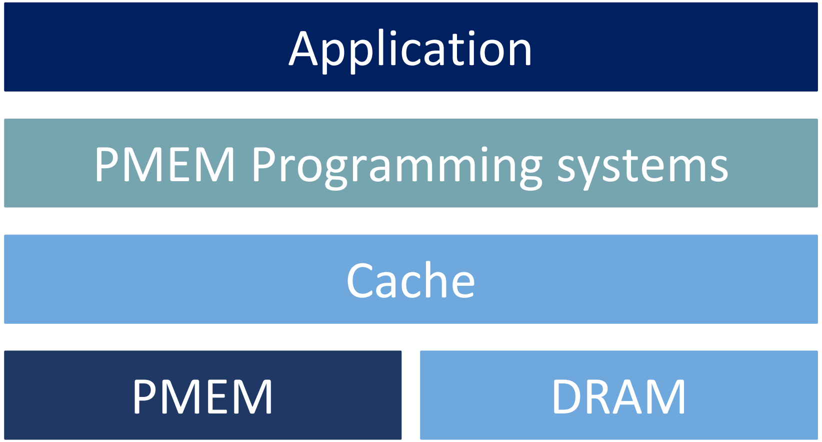
PMEM

DRAM

PMEM programming systems

- Tools to make PMEM programming easier and faster.
- Most are based on a ***"failure-atomic section“*** model.
- After a crash, each section’s writes are either all persistent, or none are.

Persistent Memory Programming



PMEM Programming systems

1. Transaction-based.
2. FASE-based.
3. Whole system persistence (WSP).

1: Transactional Models

Bank transaction example

```
void transfer (src_account, dest_account, amount)
{
        src_account.lock();
        dest_account.lock();

        src_account.balance -= amount;
        dest_account.balance += amount;

        dest_account.unlock();
        src_account.unlock();
}
```

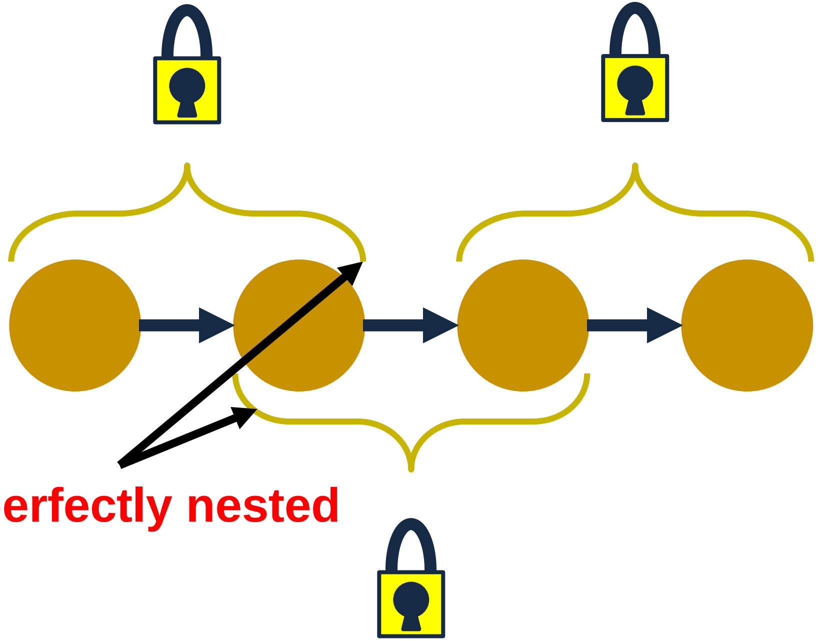
Transactional code

All writes to PMEM are performed as traditional ACID transactions.

Often use locks to mark transactions, since **transactions restrict locking semantics**

Lock acquire/release **must nest perfectly**, and PMEM can only be accessed when all locks are held

Transaction Limitations

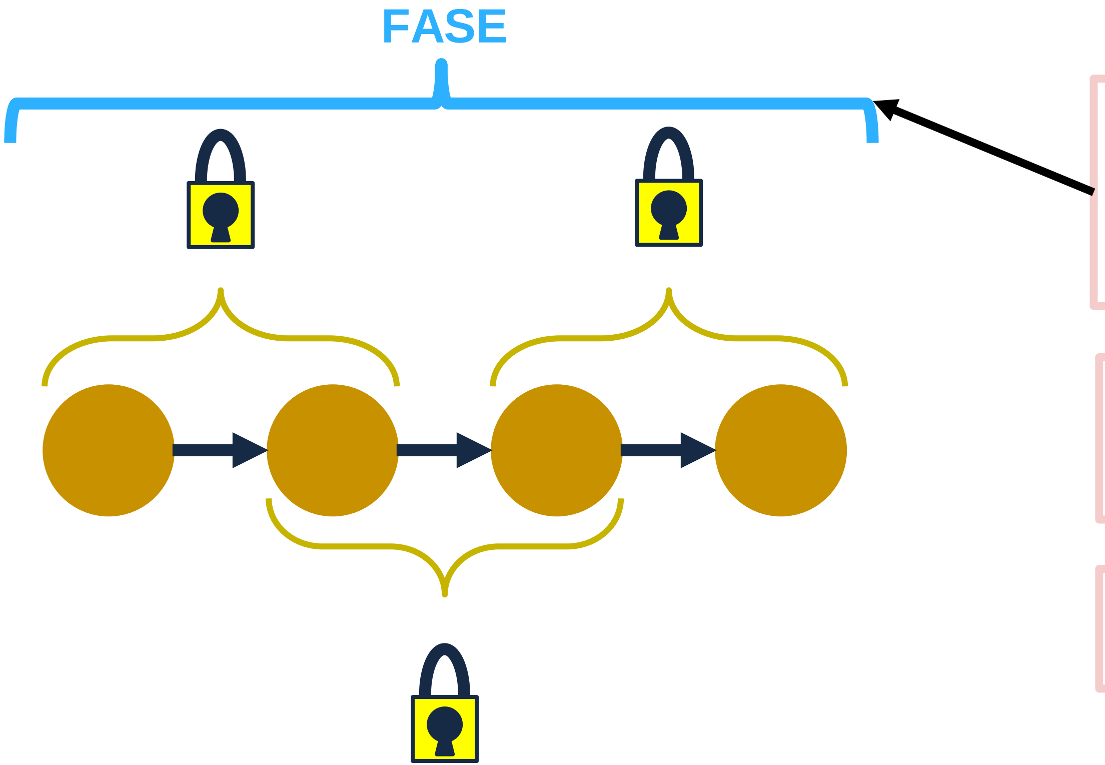


Not perfectly nested

Locking the next item in a pointer chain before releasing the previous one **violates transactional locking.**

This pattern is common in multithreaded graph applications with fine-grained synchronization.

2: FASE Models

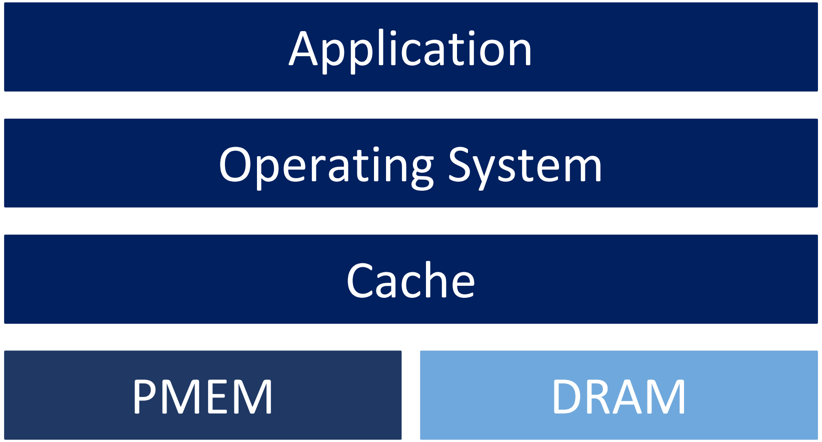


Allows arbitrary locking schemes. **A FASE is a failure-atomic operation protected by its outermost locks.**

Supports any locking scheme; **compatible with legacy code.**

Requires runtime tracking of **dependencies between threads.**

3: Whole system persistence



Application

Operating System

Cache

PMEM

DRAM

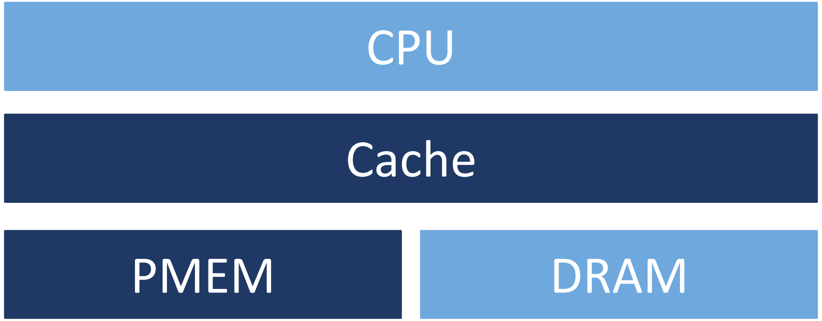
Whole System Persistence

- Everywhere DRAM would normally be used, it is replaced with PMEM.
- Only explicitly flush the cache if a failure occurs (***flush-on-fail***).

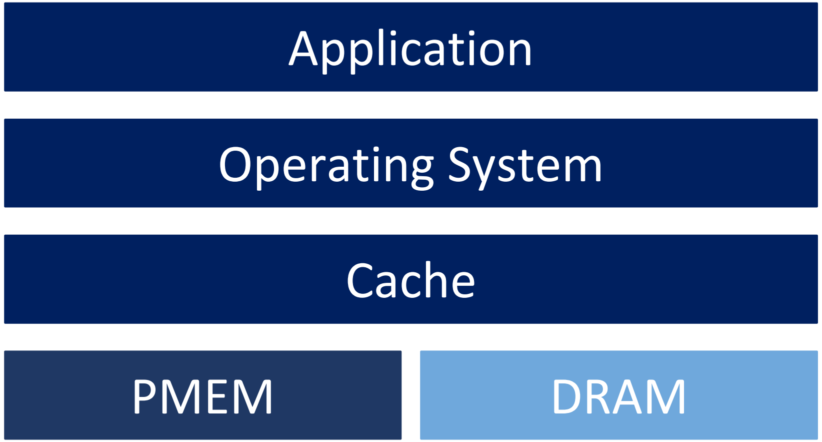
Flush-on-Fail Hardware

Flush-on-fail

- Manufacturers have developed systems with flush-on-fail support (CXL **GPF**, NVDIMM **eADR**)
- These systems guarantee that the **caches will be flushed by a low-level interrupt** if a power failure occurs.
- **Caches are effectively persistent.**



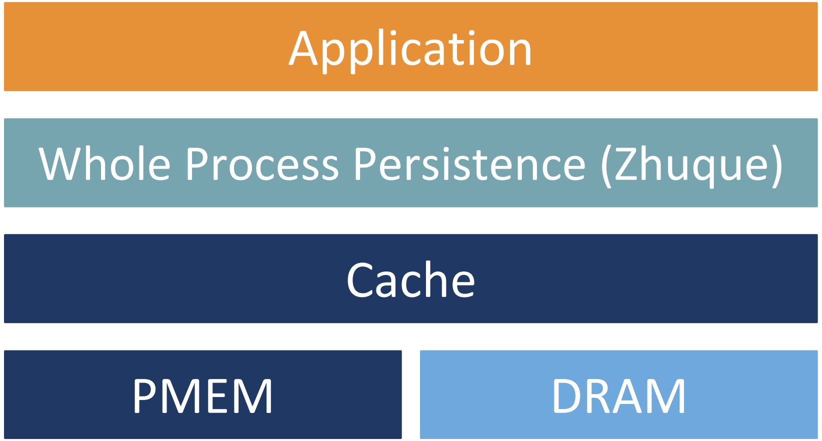
Limitations of Whole System Persistence



WSP limitations

- Only preserves memory contents; **applications are responsible for implementing recovery**
- The **whole** system doesn’t need to be persistent – just important applications.

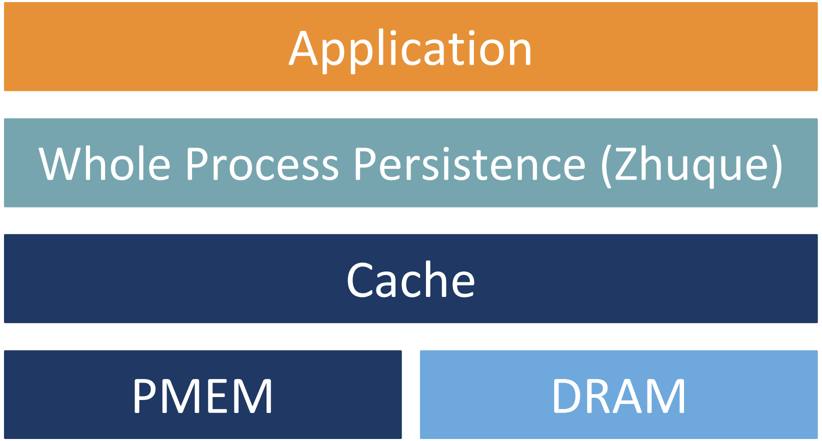
Whole Process Persistence



Whole Process Persistence

- Transform **all memory allocated by a process** into PMEM
- If a power failure occurs, the process is **signaled by the OS at time of recovery**
- **Execution continues** from the point interrupted by failure

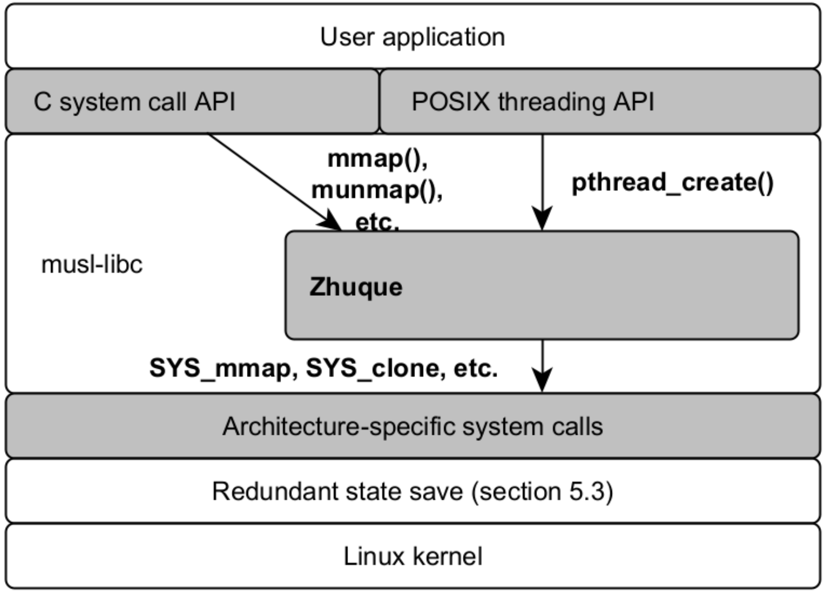
Whole Process Persistence



Whole Process Persistence

- Easy to use:
  - No restrictions on locking or I/O
  - Binary and source-compatible with native applications
- Low overhead:
  - No explicit cache flushes
  - No write amplification

Zhuque

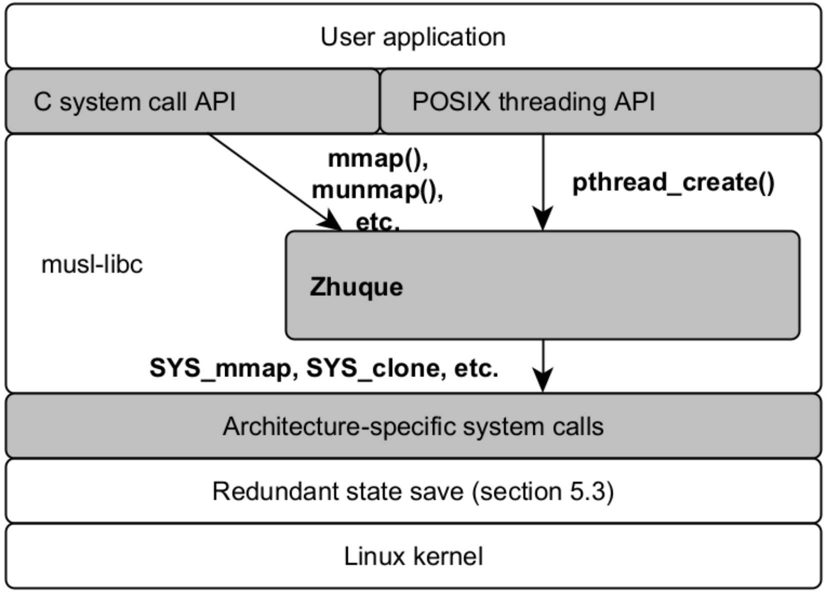


Zhuque

- **Modified version of `libc`** which implements WPP
- **Intercepts and transforms API calls** for memory, thread, and file management
- **Transparent to the application** – just set an environment variable

```
$ CRASH_RESISTANT=1 ./mycomputation
```

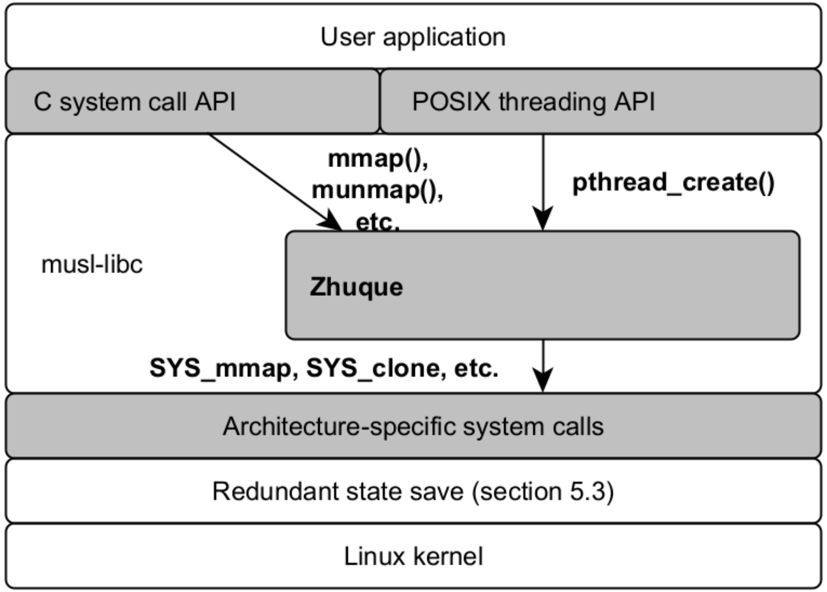
During normal execution



Zhuque

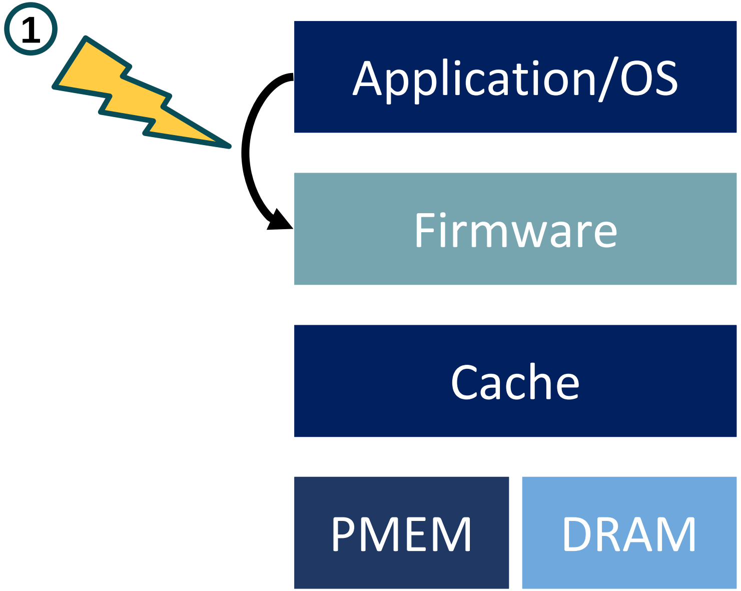
- Dynamic memory: return PMEM for **anonymous** `mmap()`.
- (Initialized) static memory: transform **private, writable file mappings** to PMEM.
- Save architectural state (register file, etc) to PMEM **on kernel entry**.

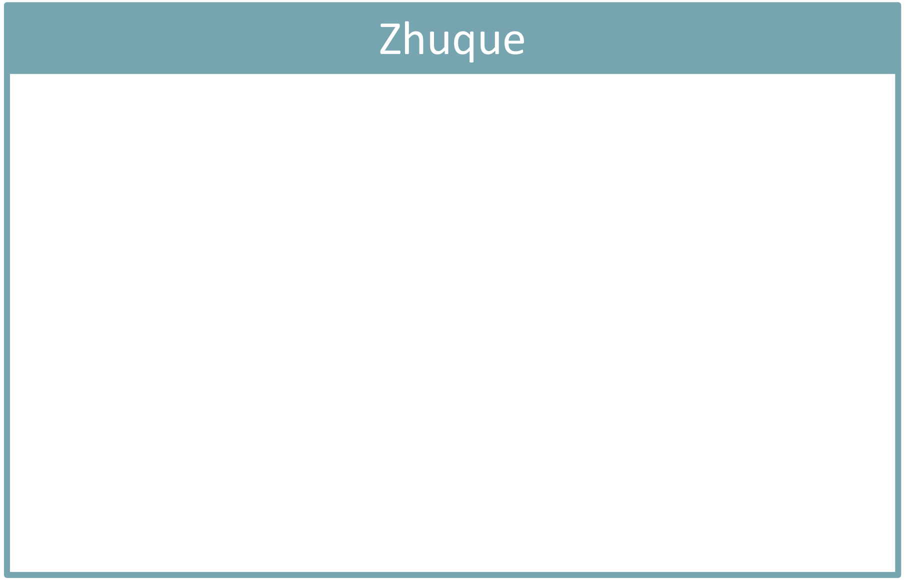
At a crash



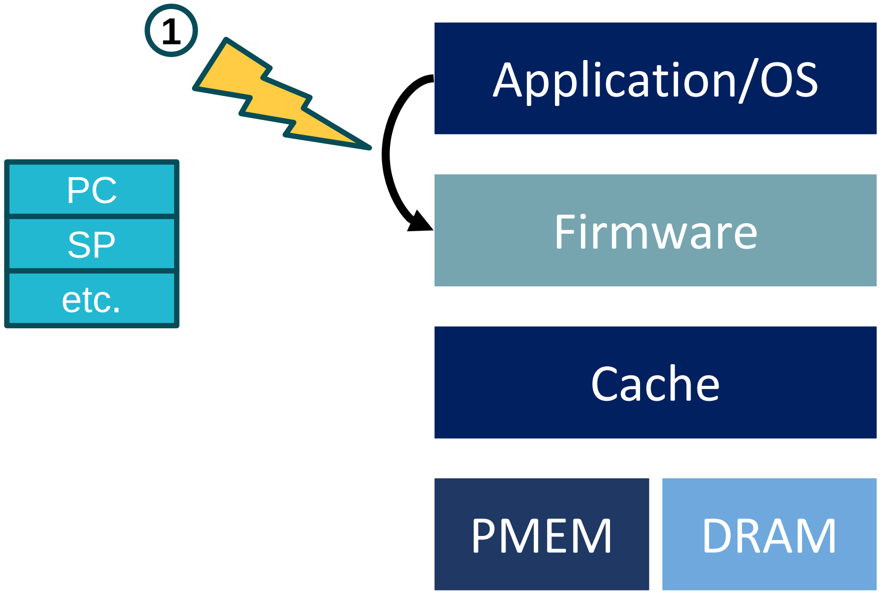
Zhuque

At a crash





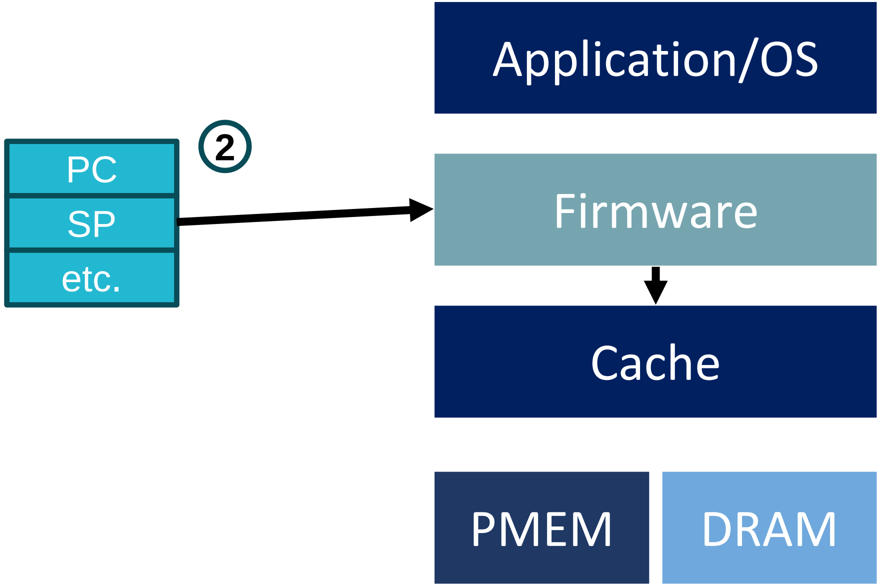
At a crash



Zhuque

- We need to save **volatile architectural state**: register file, FP/vector context, etc

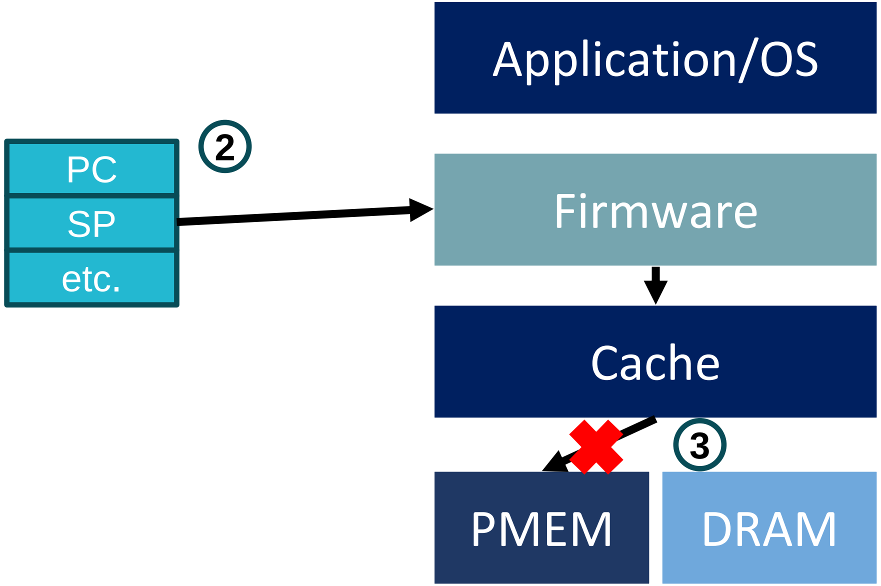
At a crash



Zhuque

- We need to save **volatile architectural state**: register file, FP/vector context, etc
- On x86, the firmware interrupt that flushes the caches saves this state to memory…

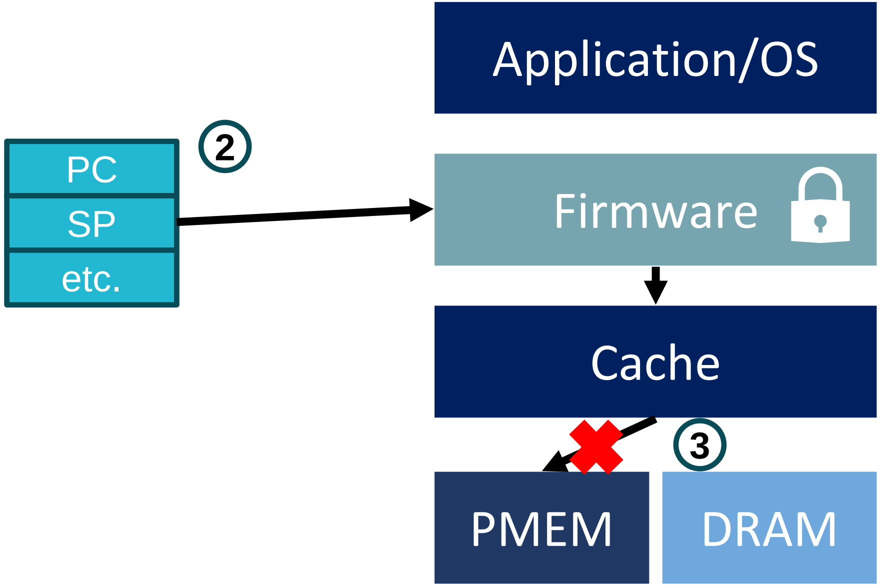
At a crash



Zhuque

- We need to save **volatile architectural state**: register file, FP/vector context, etc
- On x86, the firmware interrupt that flushes the caches saves this state to memory… **but not to PMEM**

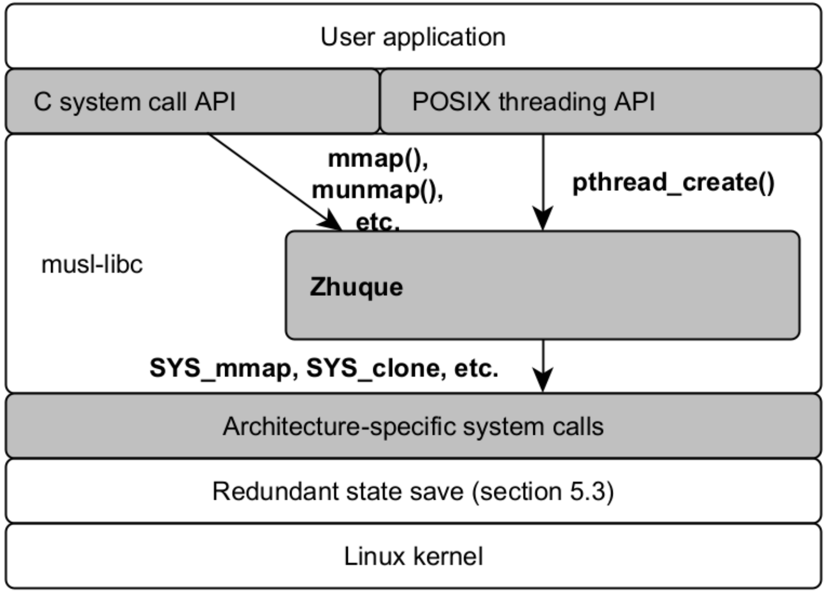
At a crash



Zhuque

- We need to save **volatile architectural state**: register file, FP/vector context, etc
- On x86, the firmware interrupt that flushes the caches saves this state to memory… **but not to PMEM**
- Saving to PMEM should work, but **we cannot replace firmware** without the platform manufacturer’s signing key

At recovery



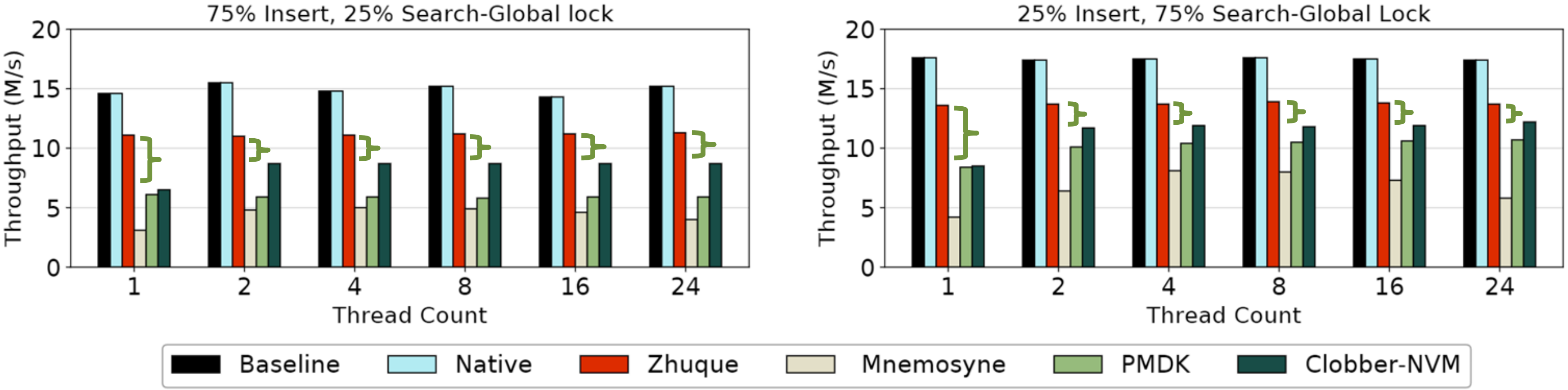
Whole Process Persistence

1. Restore the **address space**.
2. Restore **OS-specific state**: Zhuque tracks threads and file descriptors and recreates them at restart.
3. Restore the **architectural state** (including stack pointer and program counter). This is <u>equivalent to restarting execution</u>.
4. If the application provided a **failure handler**, run it before continuing execution.

Zhuque -- Requirement to applications

- Threading, FDs, virtual memory must be managed through libc (**no inline syscalls**)
- Applications must check error returns from system calls which interact with **components outside the process**

Performance - memcached 1.2.5



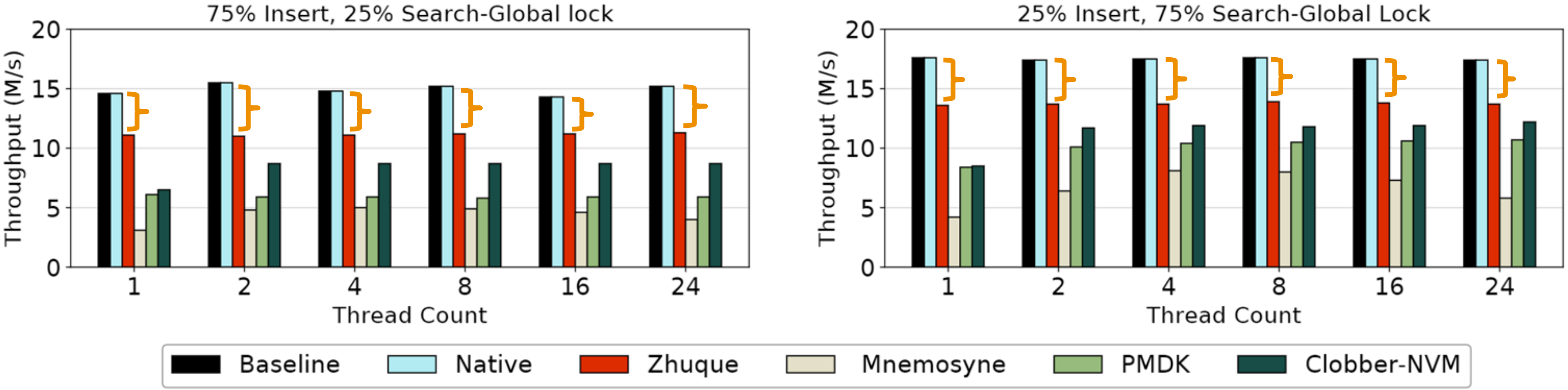
Zhuque outperforms prior work, with flushes and fences removed, on old memcached

UP IS BETTER

Y-AXIS RANGE: 0-20 M/s

} = **improvement over prior work**

Performance - memcached 1.2.5



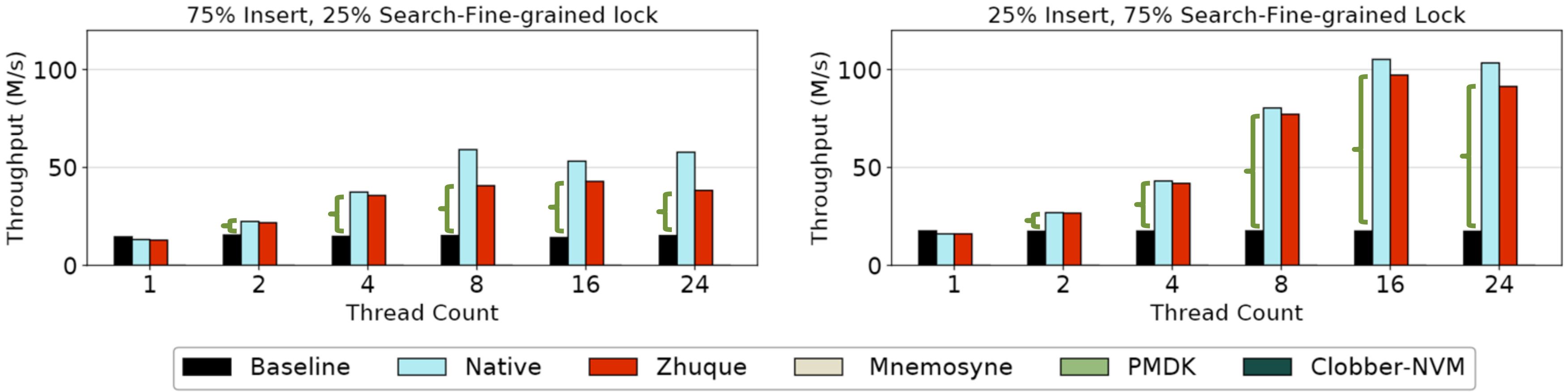
Zhuque outperforms prior work, with flushes and fences removed, on old memcached

UP IS BETTER

Y-AXIS RANGE: 0-20 M/s

} = **overhead vs. native**

Performance - memcached 1.6.10



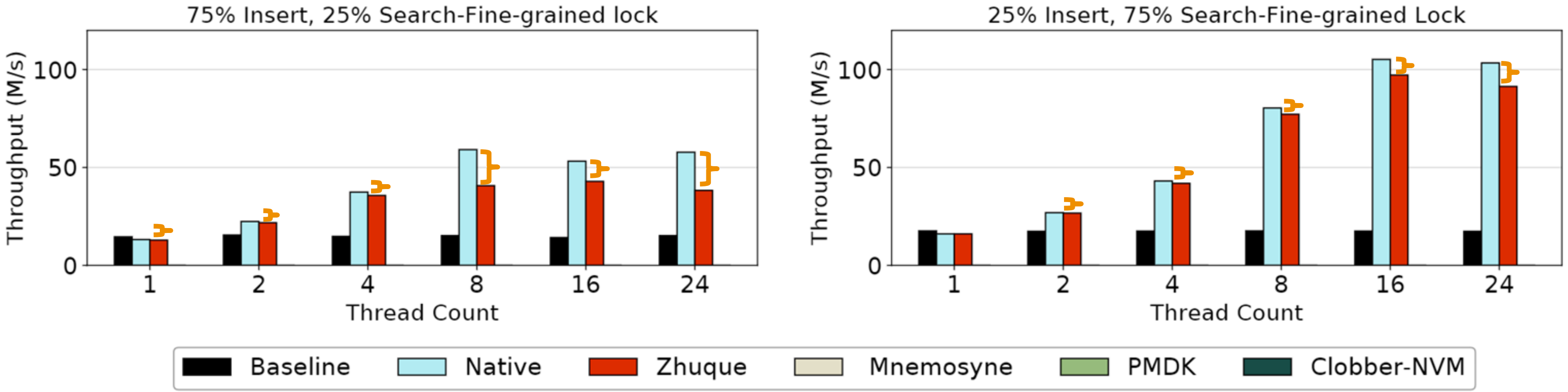
<u>Unlike prior work</u>, Zhuque can run a new version of Memcached and take advantage of better scaling

UP IS BETTER

Y-AXIS RANGE: 0-100 M/s

{ = **improvement vs old Memcached**

Performance - memcached 1.6.10



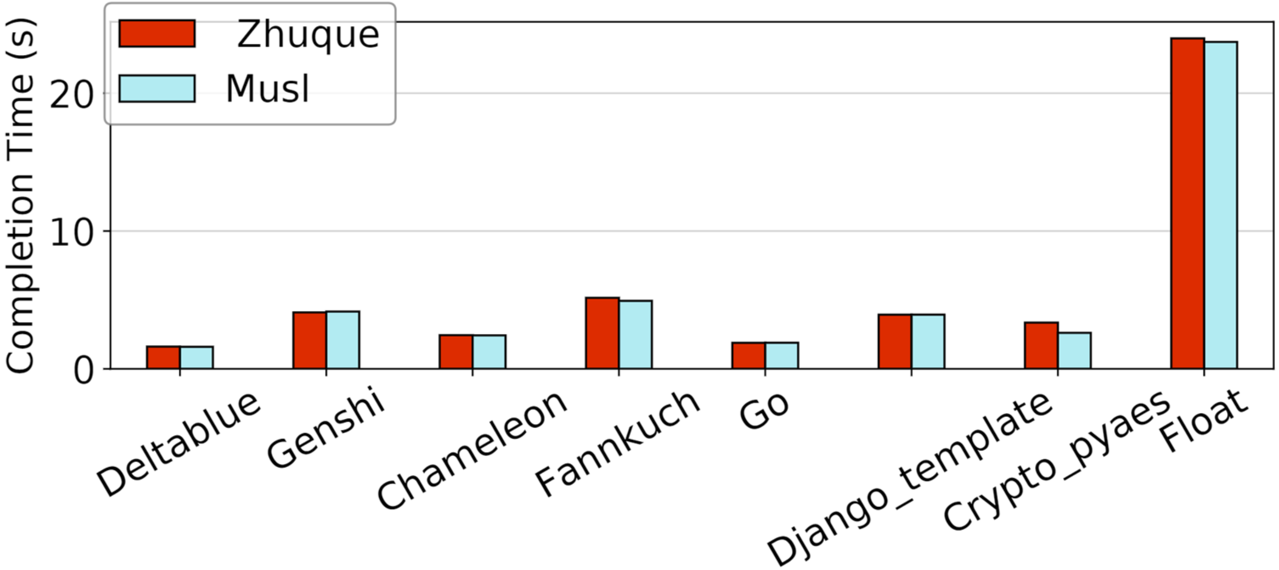
<u>Unlike prior work</u>, Zhuque can run a new version of Memcached and take advantage of better scaling

UP IS BETTER

Y-AXIS RANGE: 0-100 M/s

} = **overhead vs. native**

Performance – CPython / Pyperformance



LOWER IS BETTER

Zhuque can run unmodified Python programs with minimal overhead

Summary of Contributions

Contributions

- Introduced the Whole Process Persistence programming model for flush-on-fail systems
- Built and tested a libc-based prototype implementation, called Zhuque
- We found that Zhuque outperforms state-of-the-art PMEM programming libraries, without cache flushes
- We found that Zhuque can run a wider range of applications than prior work, without modifying or recompiling them