Crash-Consistent File Systems for Persistent Memory

Hayley LeBlanc
January 31, 2023

Overview

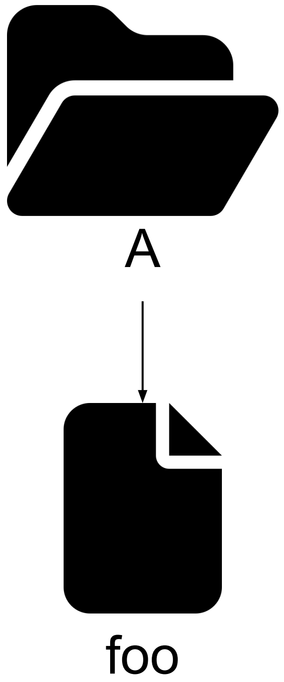
1. Background and motivation
2. Chipmunk
3. Current work

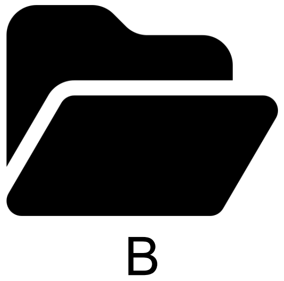
Crash consistency: data can be correctly recovered after power loss or system crash

What actually can be **recovered**?

What does it mean to recover **correctly**?

Example: moving a file

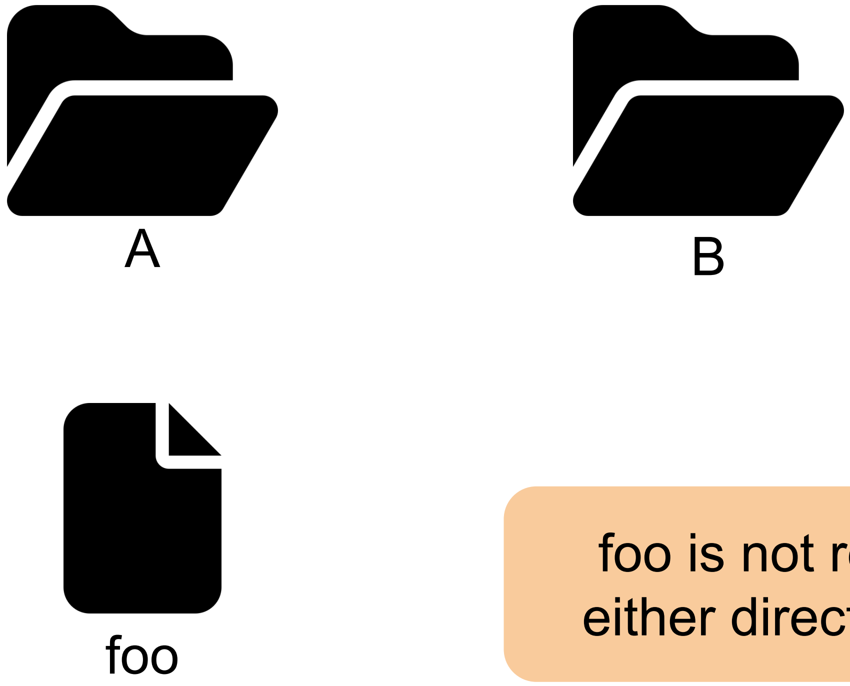




Goal: move foo from A to B

1. Delete pointer to foo in A
2. Create pointer to foo in B

Example: moving a file



Goal: move foo from A to B

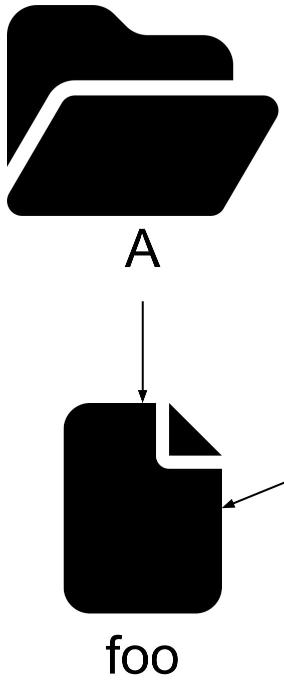
1. Delete pointer to foo in A

— CRASH —

2. Create pointer to foo in B

foo is not reachable from either directory - incorrect!

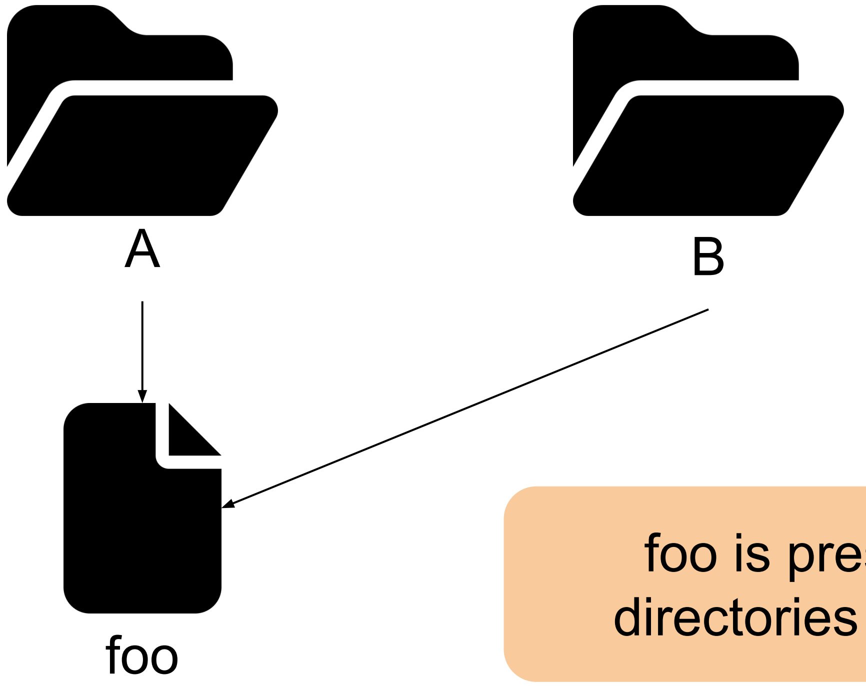
Example: moving a file



Goal: move foo from A to B

1. Create pointer to foo in B
2. Delete pointer to foo in A

Example: moving a file



Goal: move foo from A to B

1. Create pointer to foo in B

— CRASH —

2. Delete pointer to foo in A

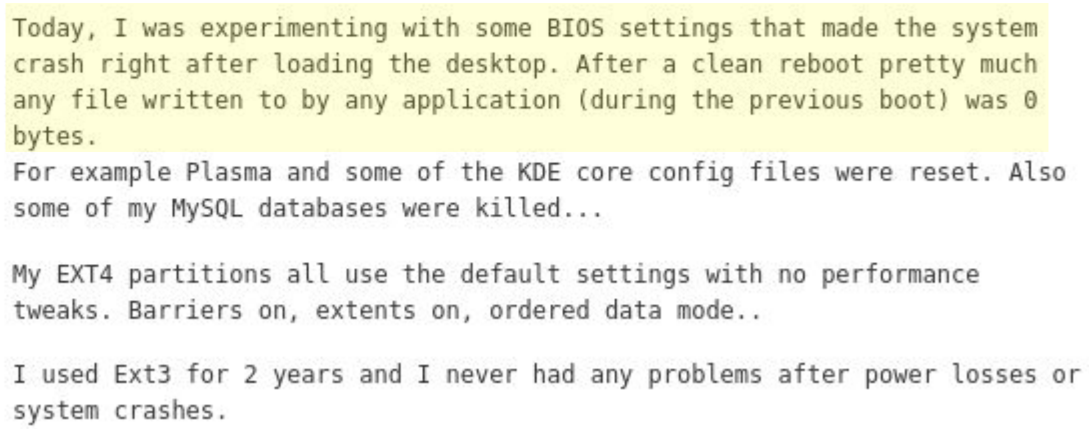
foo is present in *both* directories - incorrect??

Traditional file systems

- Hard drives and SSDs are SLOW
- File systems are ASYNCHRONOUS
  - User does not have to wait for written data to be flushed
  - Need for asynchrony was recognized early - first Unix file system buffered writes to reduce # of I/O calls
- Coarse-grained updates can be delayed and reordered
- As shown in example: order of updates is *very* important for crash consistency!
- Inherent tradeoff between performance and reliability
  - The longer updates can be delayed, better performance will be - but more likely to lose data and confuse users

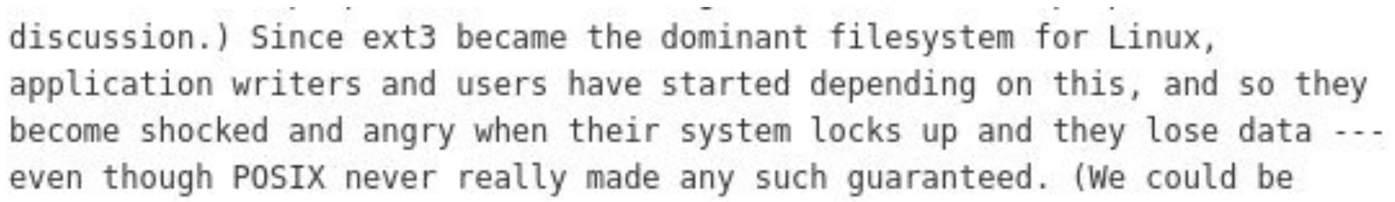
Crash consistency today

- Primary goal: performance
- Crash-consistency properties are often confusing, poorly specified
- Bugs in file system can break these properties…
  - But it is hard to determine if something is *actually* a bug, or an intended behavior that users don’t like
- Today’s FSes are mature and well tested, but burden is often on developers to understand guarantees and write apps to achieve necessary level of reliability
- E.g. 2009 ext4 data loss issue


Today, I was experimenting with some BIOS settings that made the system crash right after loading the desktop. After a clean reboot pretty much any file written to by any application (during the previous boot) was 0 bytes.
For example Plasma and some of the KDE core config files were reset. Also some of my MySQL databases were killed...

My EXT4 partitions all use the default settings with no performance tweaks. Barriers on, extents on, ordered data mode..

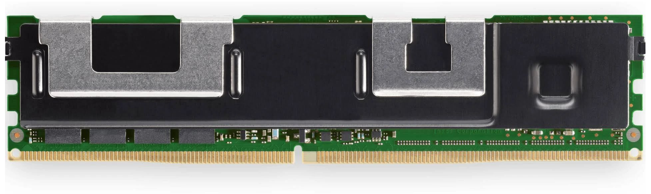
I used Ext3 for 2 years and I never had any problems after power losses or system crashes.


discussion.) Since ext3 became the dominant filesystem for Linux, application writers and users have started depending on this, and so they become shocked and angry when their system locks up and they lose data --- even though POSIX never really made any such guaranteed. (We could be

Persistent memory

Traditional storage media (HDD, SSD)

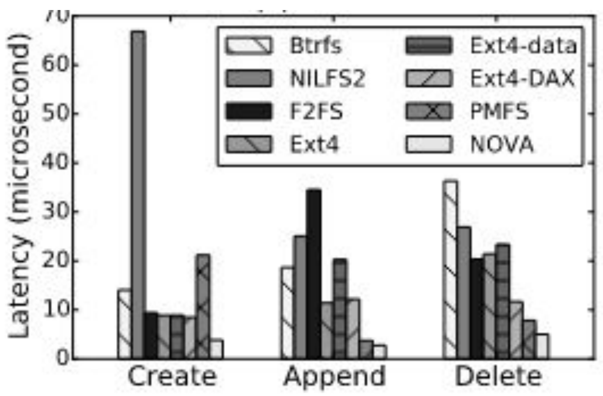
- Slow access latency (4ms-100µs)
- Block-addressable

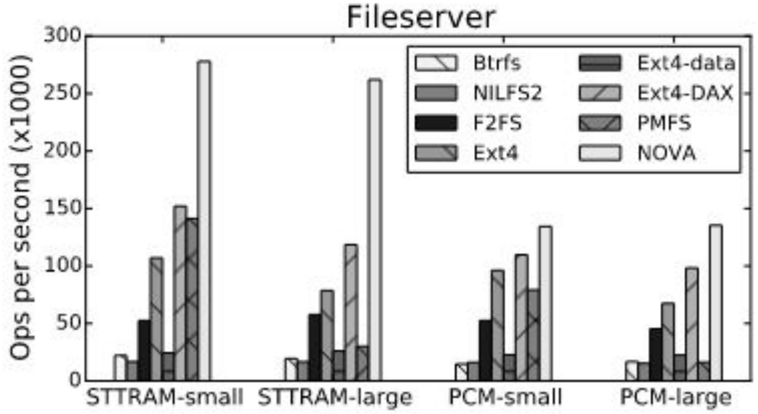


Persistent memory (PM)

- Fast access latency (~300ns)
- Byte-addressable
- Accessible via memory loads/stores

Advantages of PM file systems



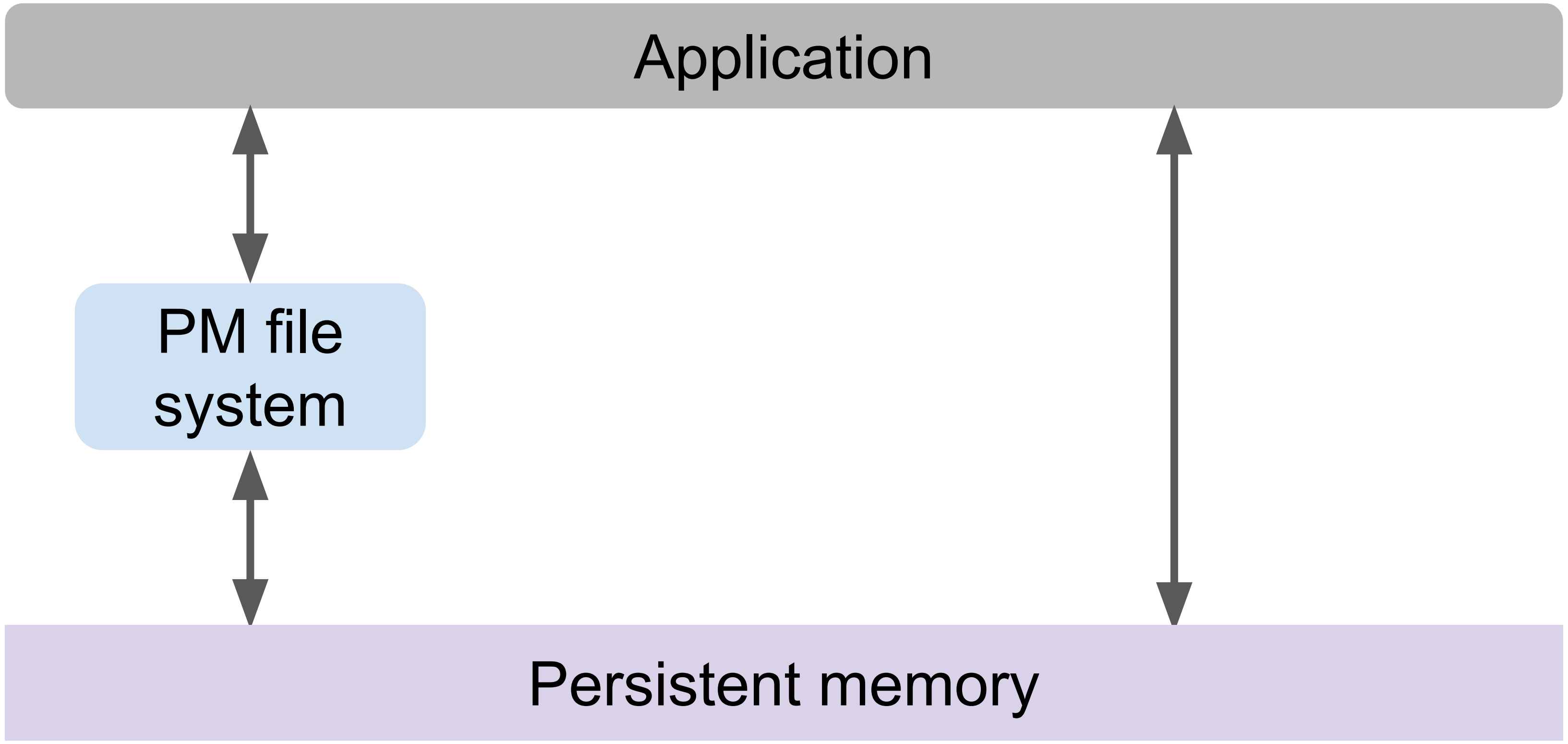


Jian Xu and Steven Swanson. “NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories,” FAST ‘16

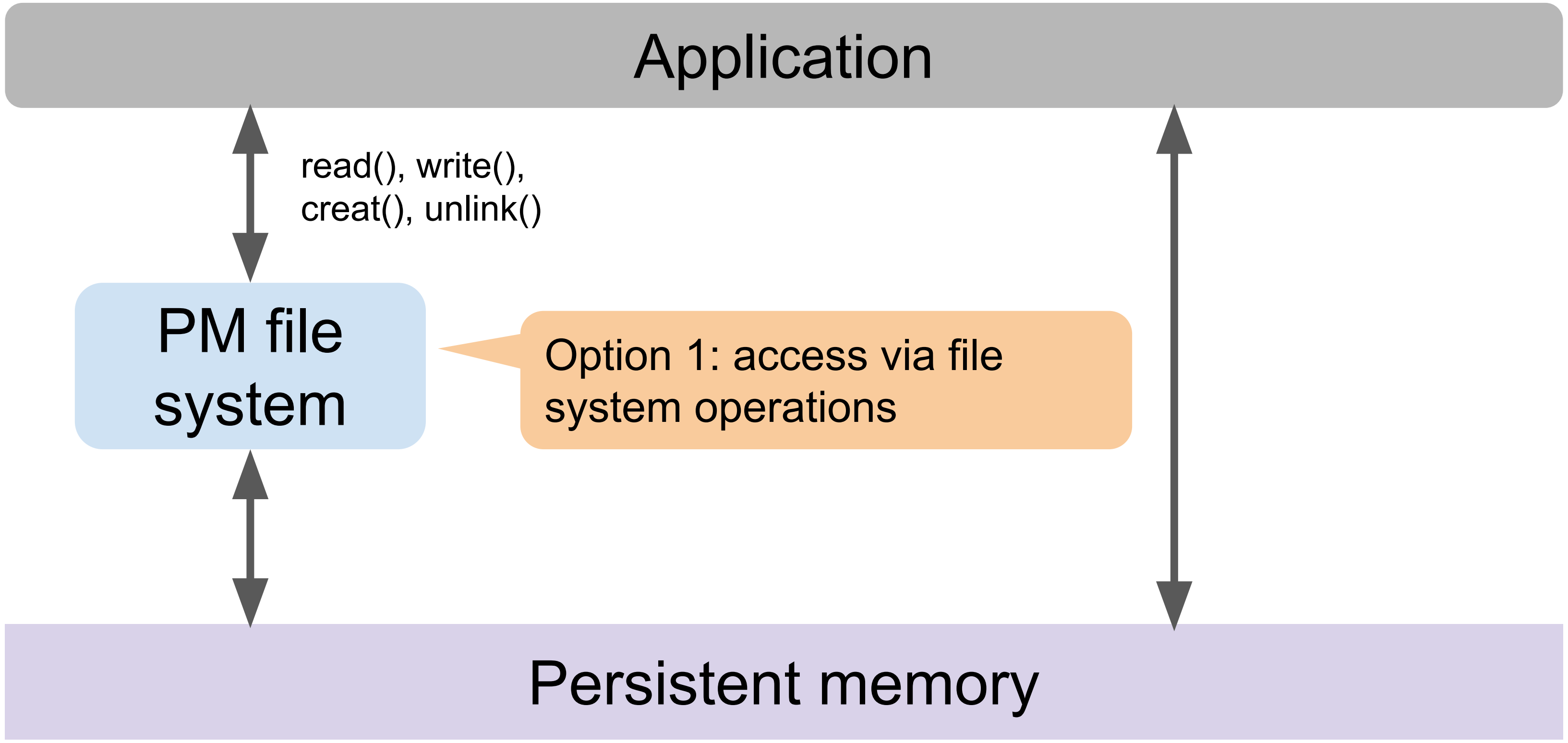
Advantages of PM file systems

- PM file systems DO NOT need to be asynchronous to achieve good performance!
- New PM file systems are synchronous
  - All updates are flushed to storage by the time each system call completes
  - Memory fences prevent updates associated with one system call from being reordered with updates from the next one
- Reliability and clear crash-consistency guarantees are easier to achieve when writes cannot be significantly delayed or reordered

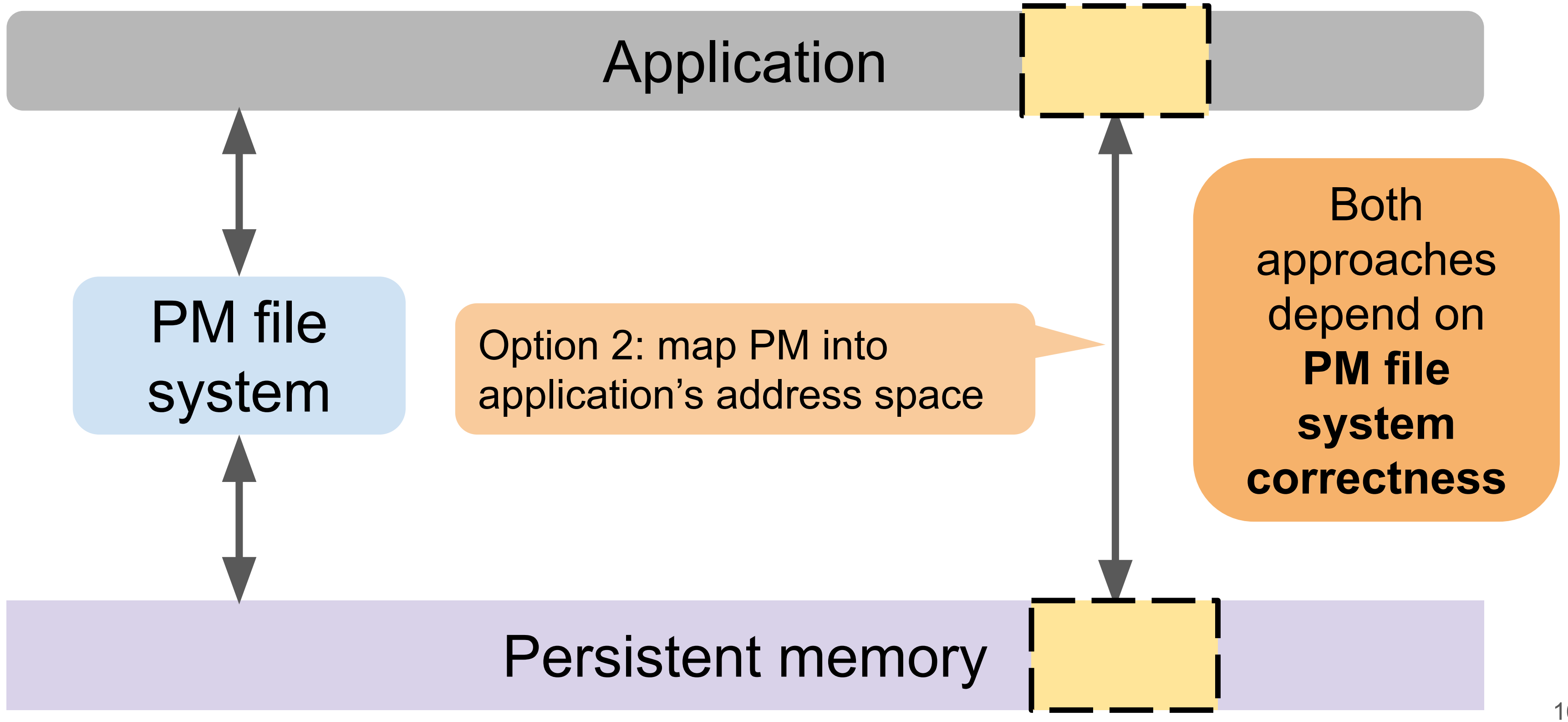
PM access modes



PM access modes

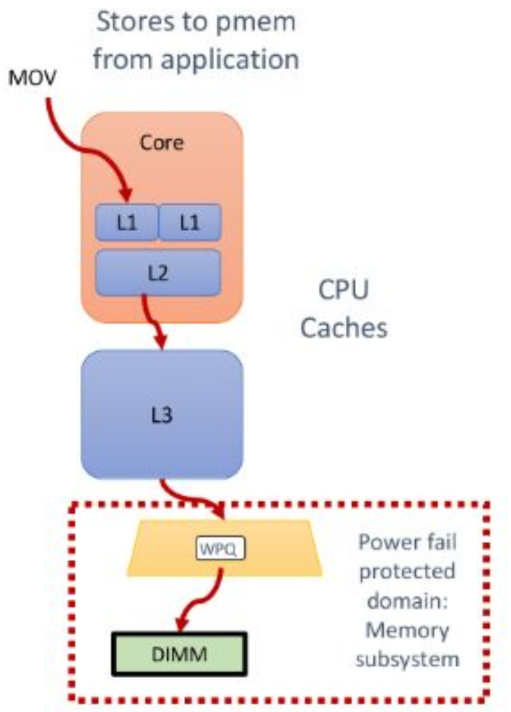


PM access modes


Application
PM file system
Option 2: map PM into application's address space
Both approaches depend on **PM file system correctness**
Persistent memory

Crash consistency challenges with PM

- CPU caches are volatile - contents will be lost in a crash
- Cache line eviction can reorder writes to PM
- **Finer-grained updates can still be delayed and reordered by hardware**
- Developers need to use store fences, non-temporal stores, and explicit cache line flushes to make programs crash safe


Stores to pmem from application
MOV
Core
L1
L1
L2
CPU Caches
L3
WPQ
Power fail protected domain: Memory subsystem
DIMM

Andy Rudoff. “Persistent Memory Programming,” *;login;* Vol. 42, No. 2.

Crash consistency bugs in PM applications

- Prior work focuses on low-level PM management errors
  - And have found many bugs this way!
- But is that the only source of bugs?
- PM file systems present similar interfaces to traditional systems…
- But their implementations are VERY different
- New (and untested) optimizations for PM usage

Chipmunk

- Tool for testing PM file systems for crash consistency bugs
- Goal: find both **high-level logic bugs** and **low-level PM and cache management errors**

Chipmunk found **23 new bugs** in 5 PM file systems!

Hayley LeBlanc, Shankara Pailoor, Om Saran K R E, Isil Dillig, James Bornholt, Vijay Chidambaram. “Chipmunk: Investigating Crash-Consistency in Persistent-Memory File Systems.” EuroSys ‘23.

Chipmunk overview

- Chipmunk records updates to PM, then replays this record to generate file system images representing possible crash states
- Challenges
    - Recording writes
    - Managing the space of crash states
    - Checking images against new crash-consistency semantics

Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, Vijay Chidambaram. “Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing,” OSDI ‘18

Recording writes

- Traditional FS testing tools use kernel block layer to record block-sized I/O
- PM testing tools record individual mov, movnt, clwb, fences
    - Require special hypervisor support, manual code annotation, etc.
    - High overhead
    - How to determine which are relevant to us and which aren’t?
- Our solution: function-based recording
- PM FSes use small set (~4) of **persistence functions** to flush and order persistent updates
- One call to a persistence function → 10’s or 100’s of individual persistence instructions
- Kprobes kernel utility for automatic instrumentation

Managing crash state space

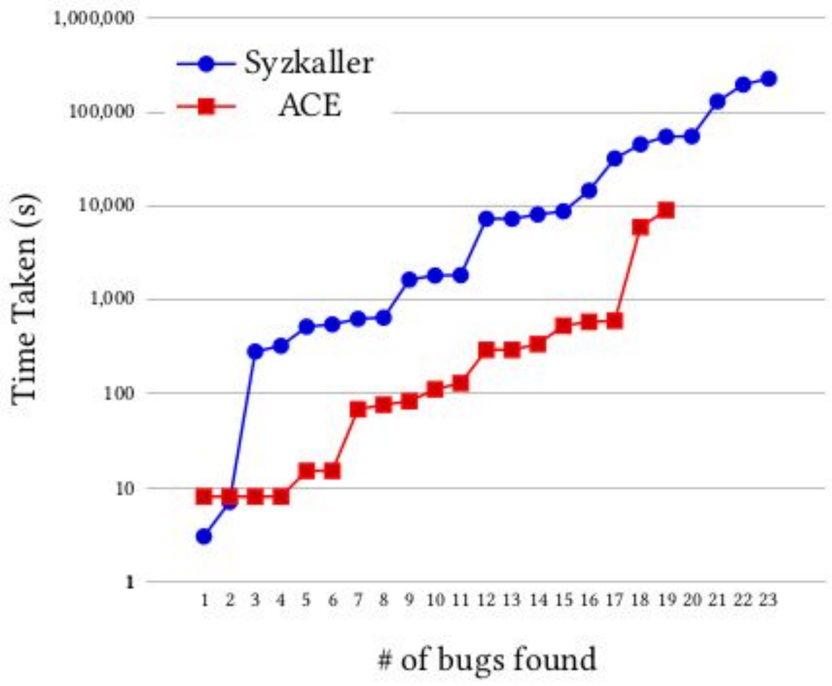
- Could examine every permutation of 8-byte (or smaller) updates between each pair of fences - MASSIVE state space
    - Yat: 1200 sys calls take 5 years to check
- We coalesce logically-related writes into larger chunks
    - Data updated flushed via cache line write back: 64 byte chunks
    - Data updated via non-temporal store: up to 4KB chunks
    - Function-based instrumentation provides info on which writes are logically related
- Only check permutations of larger chunks
- Observation: average of 3 chunks between fences; max 10 in tested workloads
    - Possible to brute-force check

Checking crash states

- PM FSes promise stronger guarantees, but don’t always clearly define them
- Older FS testing tools only check crash states after fsync, but we want to look at more states
- We check:
  - All sys calls are synchronous (effects are all persistent by the time the call returns)
  - All sys calls are atomic (except for write, sometimes)
- Matches implicit spec and expected behavior

Workload generation

- Automatic Crash Explorer (ACE) vs. Syzkaller
- ACE tests find 19/23 bugs in <3 hours
- Syzkaller finds 4 more bugs
- Lightweight testing with ACE, longer-running checks with Syzkaller



Results

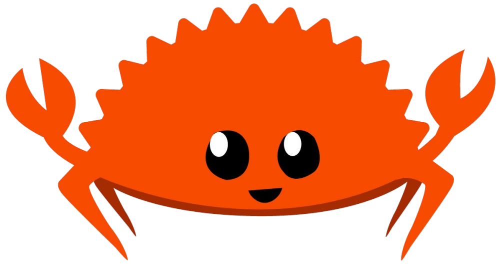
- 23 new bugs total in 5 PM file systems
- Bugs found in both kernel- and user-space file systems
- Bugs have significant consequences
  - Prevent FS mounting after crash
  - Break rename atomicity
  - Lose data
- Most (19/23 bugs) are logic errors, not PM programming mistakes
  - Optimizations for PM - new logging protocols, maintaining state in DRAM, in-place updates - caused many significant bugs
- Short workloads sufficed to expose most (19/23) bugs

Crash-consistent file systems

- How can we be confident that file systems for PM are truly crash consistent?
- Testing cannot find every bug
- Verification can make correctness guarantees, but requires specialized knowledge, takes significant extra time, and impacts performance
- Other techniques (model checking, etc.) are incomplete and/or introduce significant overhead

Rust language

- Compiler enforces memory safety without garbage collection and prevents data races
- Similar performance to C → increasingly popular for systems programming
- Recently merged into the Linux kernel!
- Key idea: **ownership**
  - Each value has a single owner (variable)
  - When the owner goes out of scope, the value is dropped and freed
  - No mutable aliases
- How can Rust be used for PM?



Related work: Corundum

- Library for managing PM in user-space applications
- Uses various language features to statically check PM applications for some of the same types of bugs we found with Chipmunk
    - E.g.: Corundum prevents persistent values from being updated outside of transactions

Morteza Hoseinzadeh and Steven Swanson. “Corundum: statically enforced persistent memory safety,” ASPLOS ‘21.

Rust for PM file systems

- In order for a file system to be crash consistent, the **order** of updates must be carefully managed
    - Journaling, logging make this easier
    - Soft updates: obtain consistency by carefully ordering updates with no additional data structures
- Can Rust ensure that data is written to PM in the correct order? Yes!
- Enter: **typestate analysis**
    - Each object has both a *type* and a *state*
    - Operations may transition objects from one state to another
    - An object’s typestate defines the set of legal operations that can be performed on it

Typestate analysis for file systems

- We can give persistent file system structures (e.g., inodes) typestate indicating their **persistence state** and their **operation state**
  - Persistence state: is the object guaranteed to be persistent yet?
  - Operation state: what is the last operation performed on this object?
- Each operation on a persistent object can only be called if the object has the right persistence and operation state

Typestate analysis for file systems

```
impl Dentry<Clean, Alloc> {
    fn set_inode_pointer(self, inode: Inode<Clean, Init>) ->
    (Dentry<Dirty, Complete>, Inode<Clean, Complete>)
    {
            self.dentry.ino = inode.get_ino();
            return (Dentry::new(self),Inode::new(inode));
    }
}
```

Conclusion

- Persistent memory introduces exciting new opportunities for storage system design
- But we need tools and techniques to build CORRECT storage systems with PM
- Chipmunk: test PM file systems for crash-consistency bugs
- Current work: build a PM file system with statically-checked crash-consistency guarantees

Typestate example: C

```
int a = 10;
char buf[10];


close(a);
read(a, buf, 10);
close(a);
```

- GCC will compile this code…
- But it doesn’t really make any sense

Typestate example: Rust

```
struct File<State> { filename: String, ..}

impl File<Open> {

    fn read(...) {...}

    fn close() -> File<Closed> {...}

}

impl File<Closed> {

    fn open() -> File<Open> {...}

}
```