MadFS: Per-File Virtualization for Userspace Persistent Memory Filesystems

Shawn Zhong*, Chenhao Ye*, Guanzhou Hu, Suyan Qu
Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Michael Swift

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

* Equal contribution

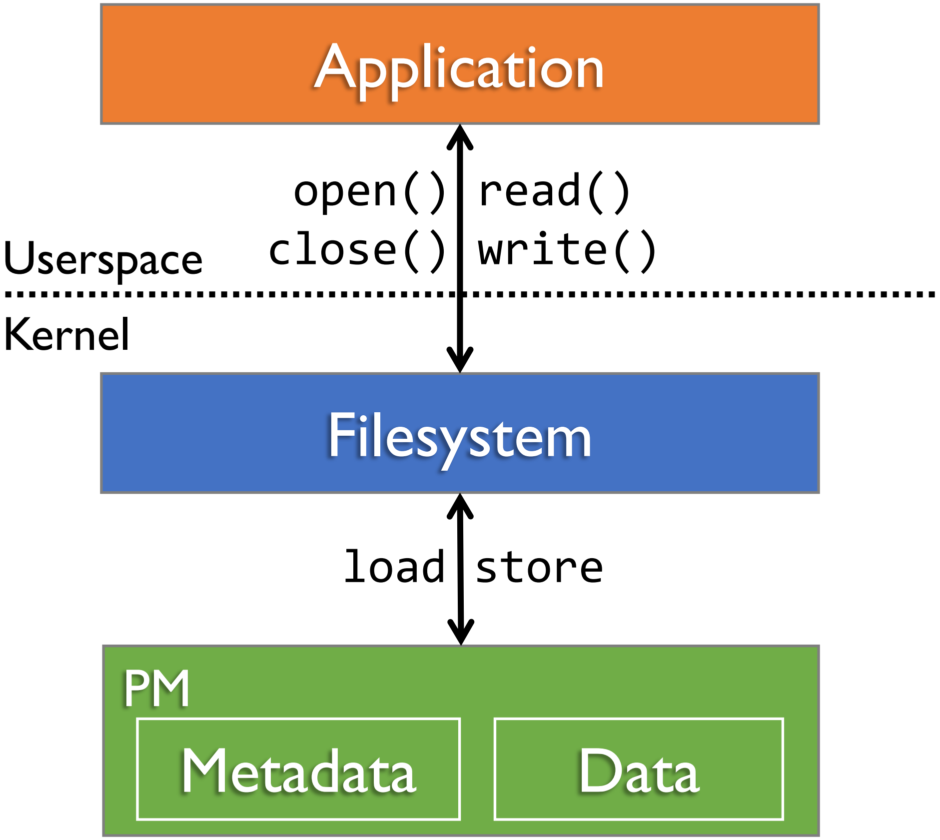
Background: Persistent Memory

Fast
sub-us latency

Byte-Addressable
accessed via CPU instructions

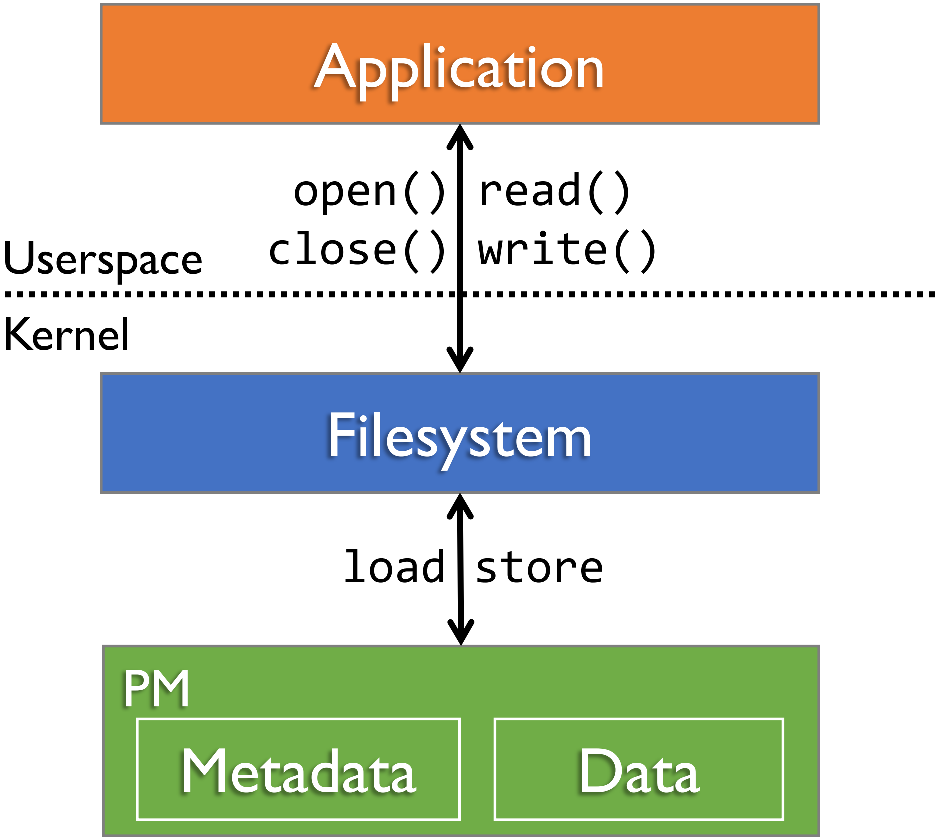
Non-Volatile
retain data without power

Background: Kernel Filesystems for PM



Kernel FS manages both data & metadata

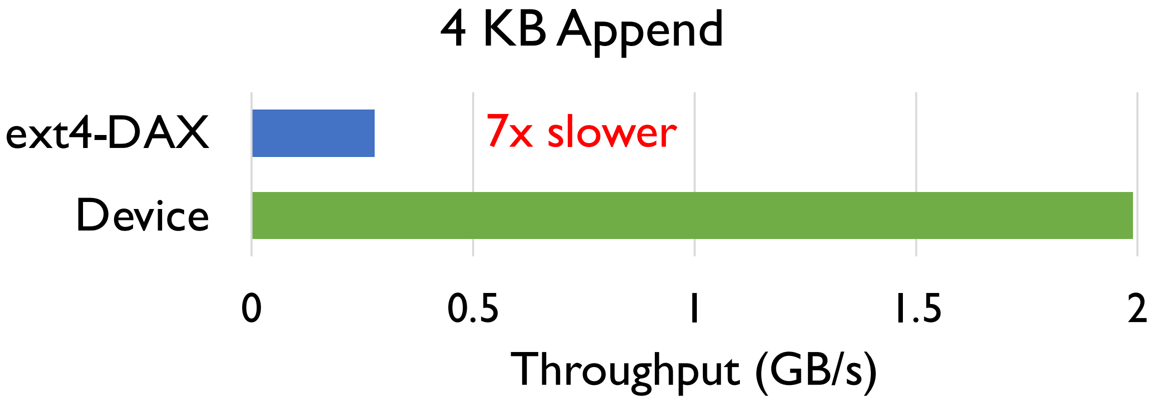
Background: Kernel Filesystems for PM



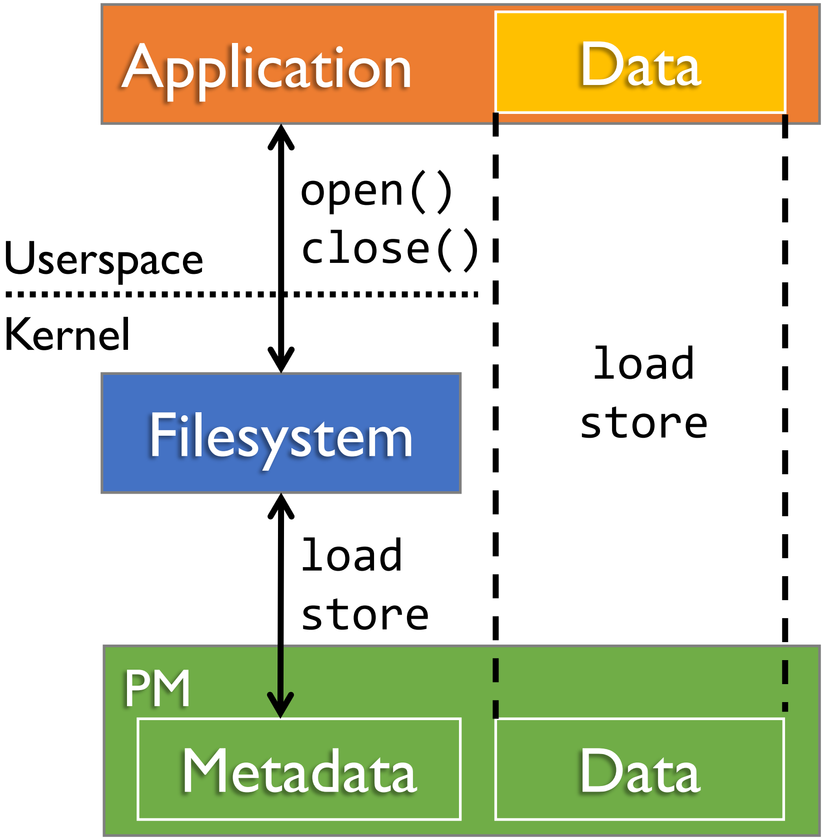
Kernel FS manages both data & metadata

Overhead for append in ext4-DAX

- System call
- VFS (e.g., inode locking)
- Metadata journaling in block granularity



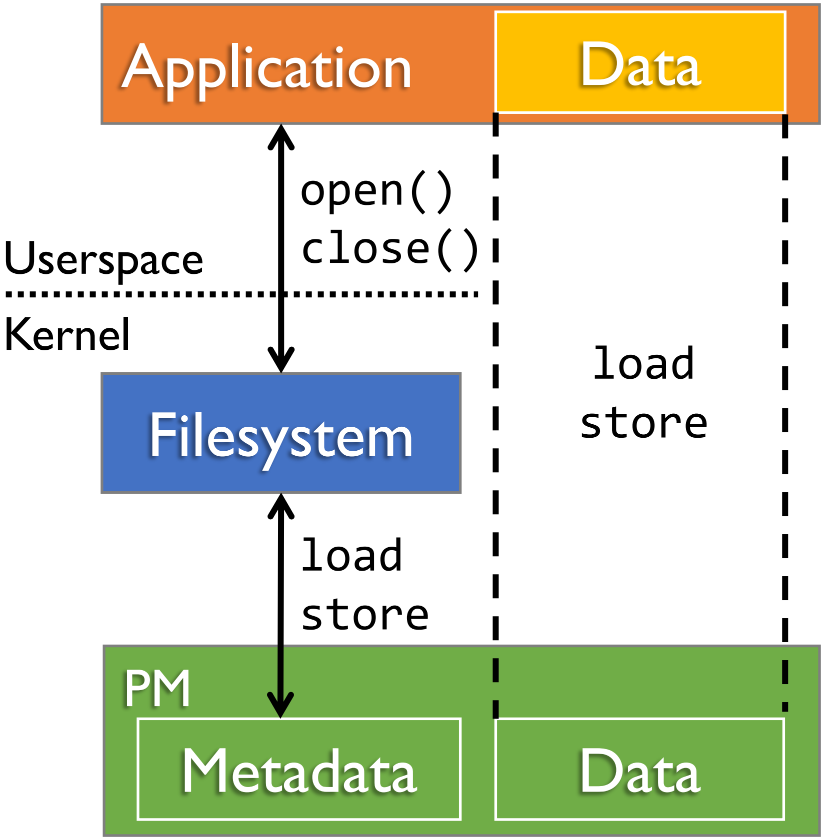
Background: Userspace Filesystems for PM



Userspace FS bypass kernel for data ops

- Memory-map file data on open
- Handle read/write in userspace via load/store

Background: Userspace Filesystems for PM



Userspace FS bypass kernel for data ops

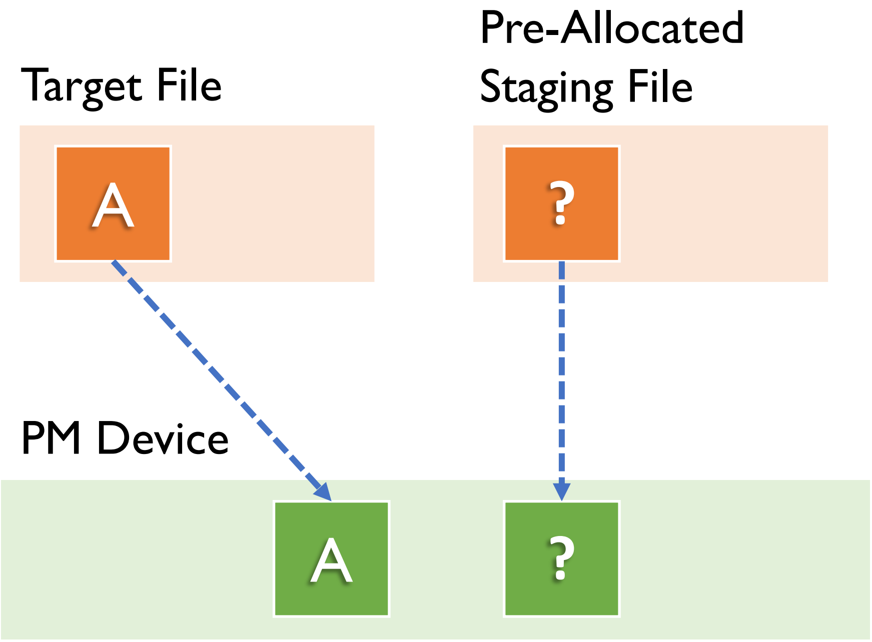
- Memory-map file data on open
- Handle read/write in userspace via load/store

Metadata still managed by kernel

Issue: Data ops coupled metadata updates

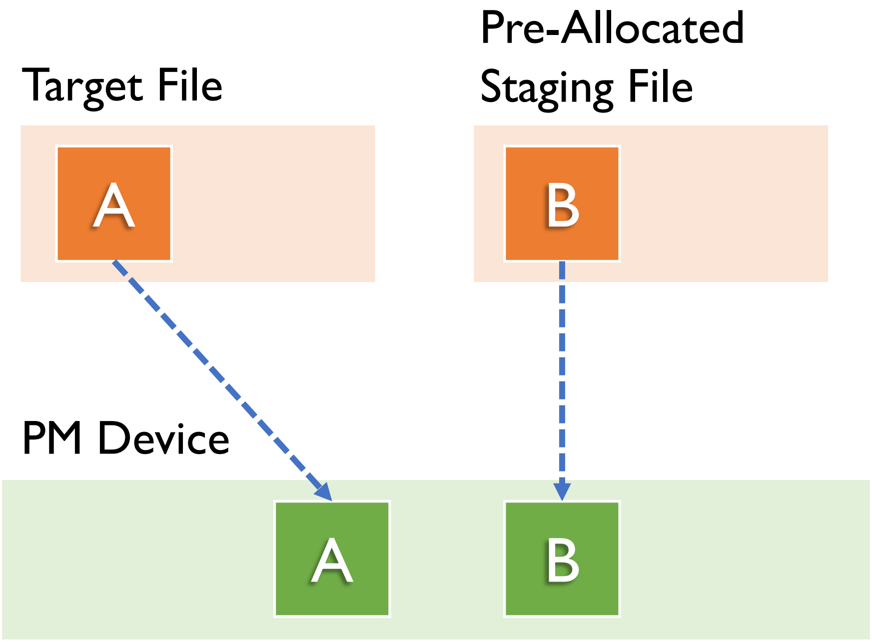
Example: Append + Fsync in SplitFS [SOSP '19]

Example: Append + Fsync in SplitFS



Append "B": Userspace Data Operation

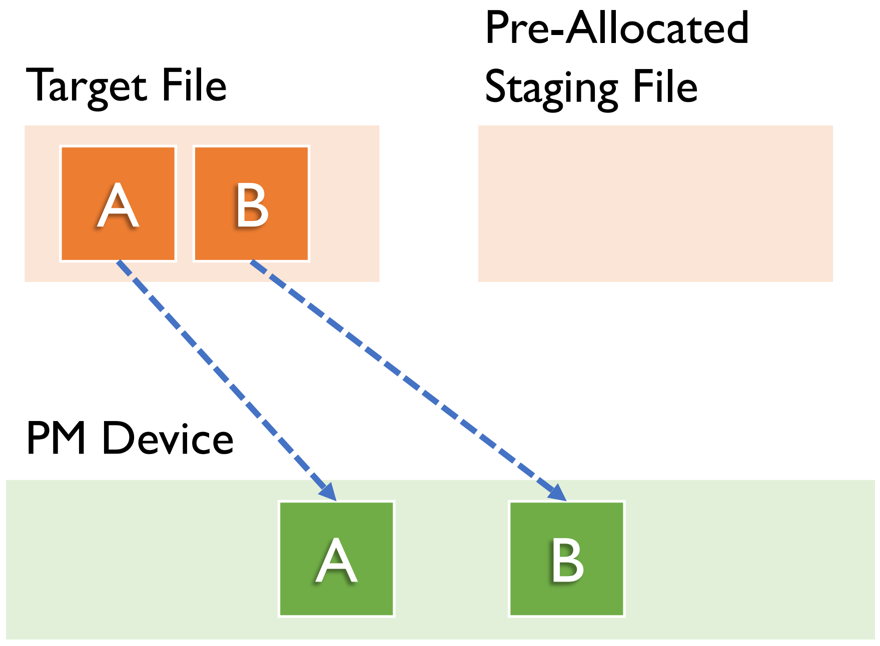
Example: Append + Fsync in SplitFS



Append "B": Userspace Data Operation

- Write data to pre-allocated file

Example: Append + Fsync in SplitFS



Append "B": Userspace Data Operation

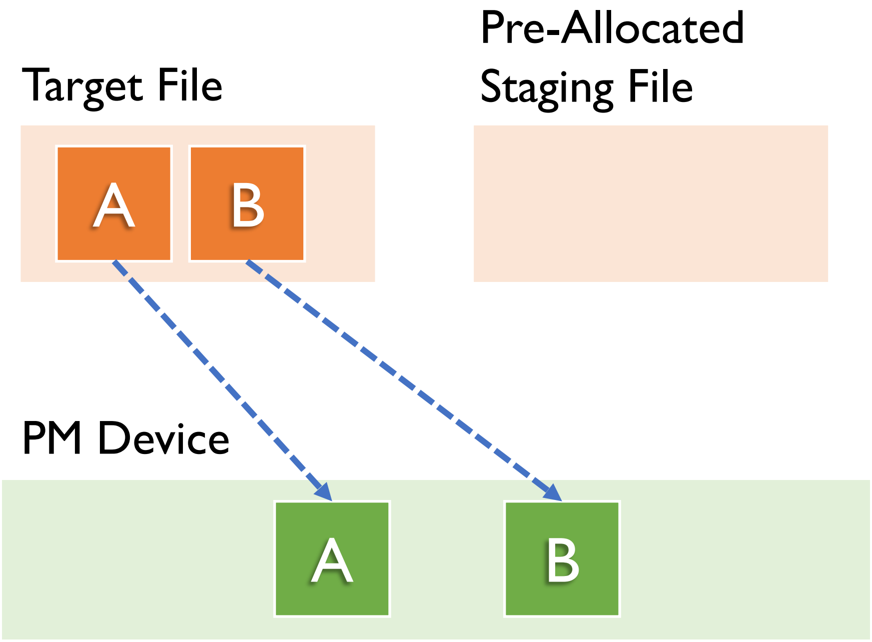
- Write data to pre-allocated file

Fsync: Kernel Metadata Operation

- Remap data to target file for visibility
- Update block map in inode

memory map / page table

Example: Append + Fsync in SplitFS



Append "B": Userspace Data Operation

- Write data to pre-allocated file

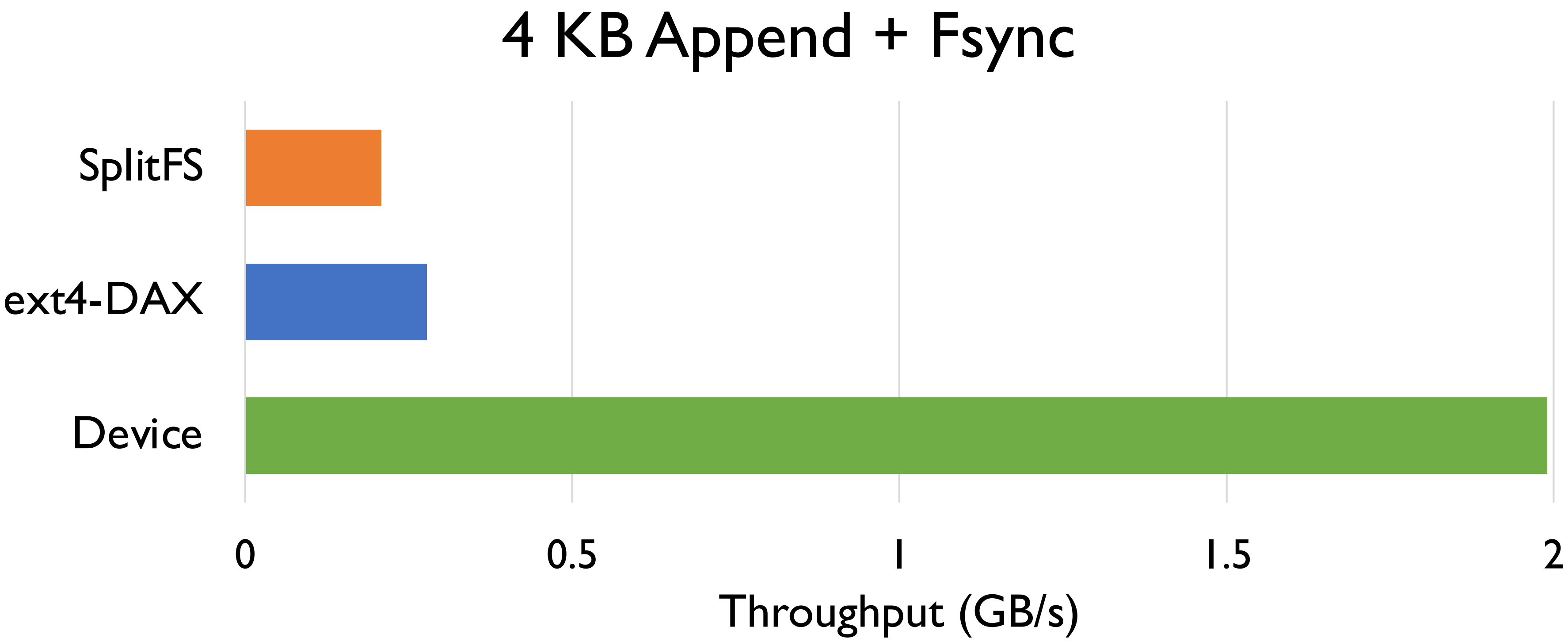
Fsync: Kernel Metadata Operation

- Remap data to target file for visibility
- Update block map in inode: kernel I/O stack

  memory map / page table: TLB

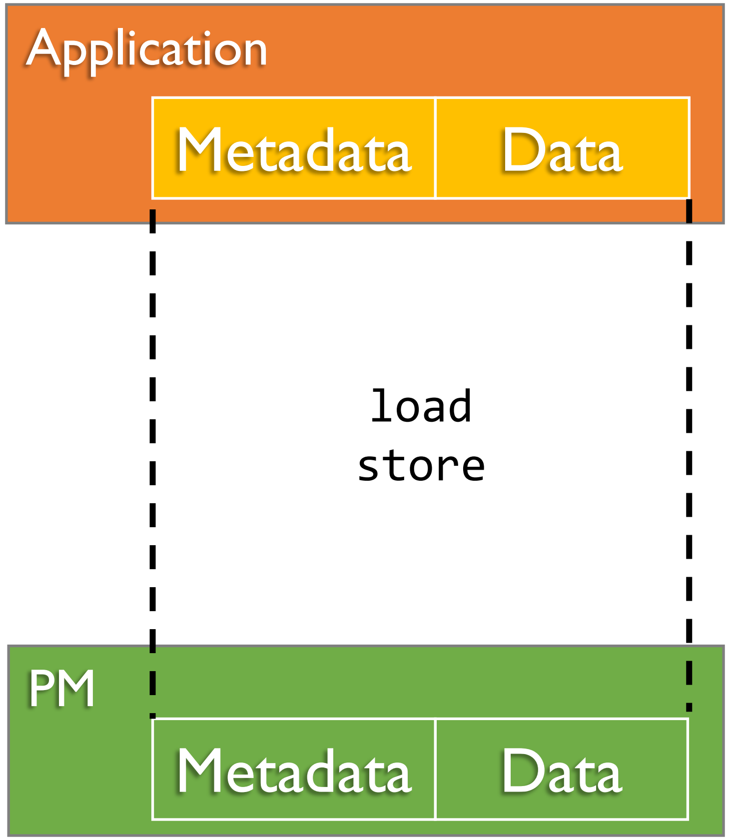
Kernel metadata operation is expensive

Example: Append + Fsync in SplitFS



Result: Worse performance compared to kernel FS 😢

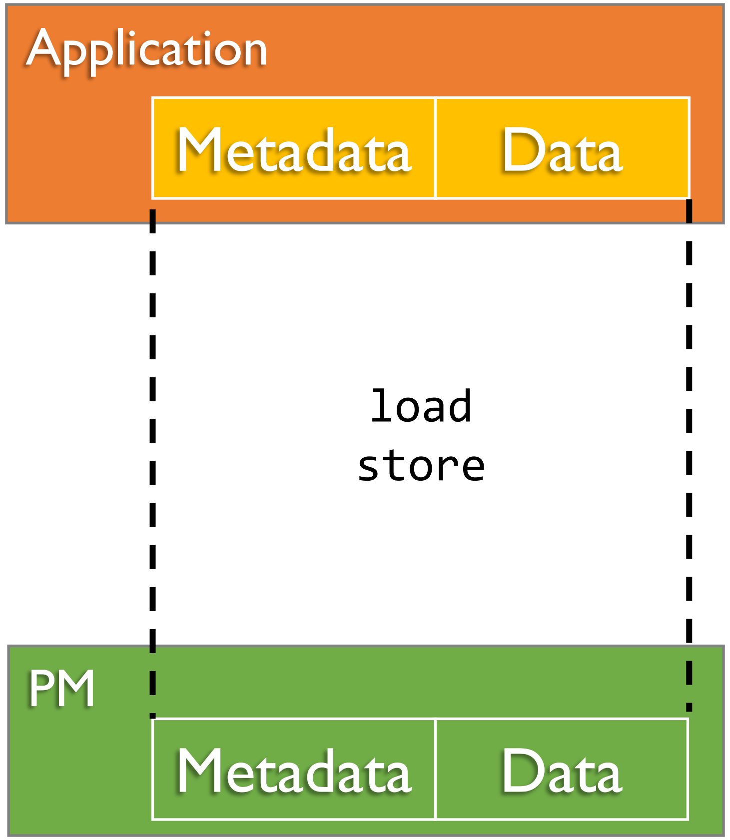
Background: Userspace Filesystems for PM



Expensive to modify kernel-managed metadata

Can we manage all metadata in userspace?

Background: Userspace Filesystems for PM



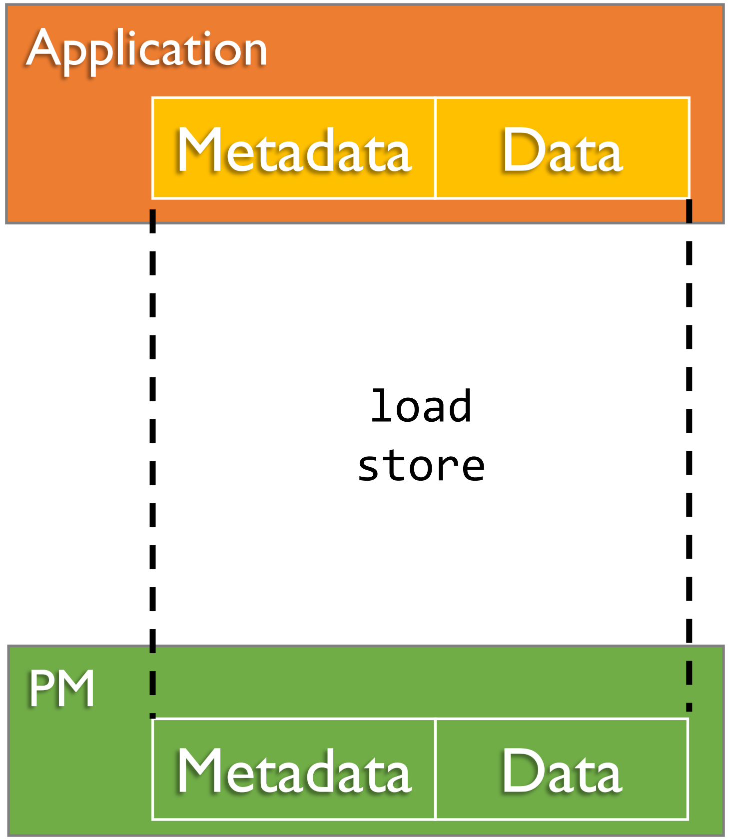
Expensive to modify kernel-managed metadata

Can we manage all metadata in userspace?

Unfortunately, no: applications are untrusted

Example: malicious user changes permission

Background: Userspace Filesystems for PM



Expensive to modify kernel-managed metadata

Can we manage all metadata in userspace?

Unfortunately, no: applications are untrusted

Example: malicious user changes permission

What about only the metadata coupled with data operation

Observation:

Some file metadata share the same protection domain as data

Insight:

Embed these metadata into data

for efficient userspace management

without sacrificing security

Metadata Embedding

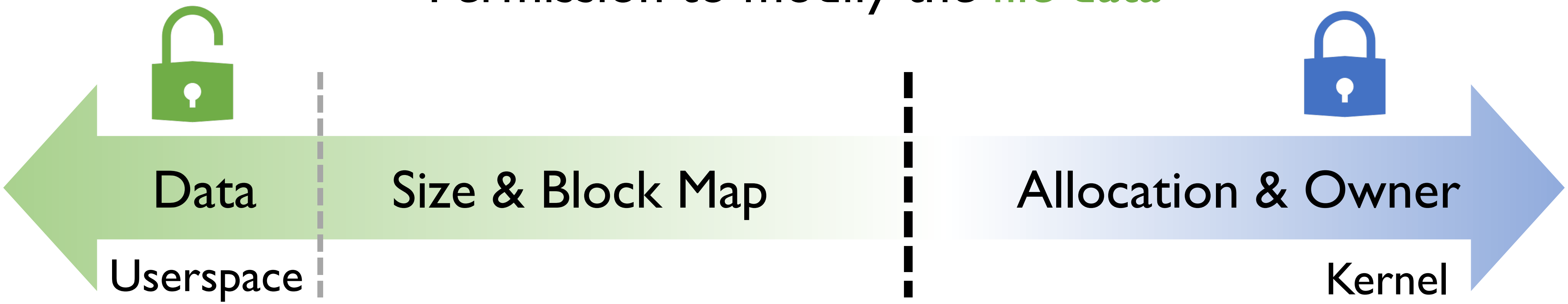
Observation: Some metadata share the same protection domain as data

Example: Block map

Permission to swap two block pointers within a file

≈

Permission to modify the file data



Data

Size & Block Map

Allocation & Owner

Userspace

Kernel

Metadata Embedding

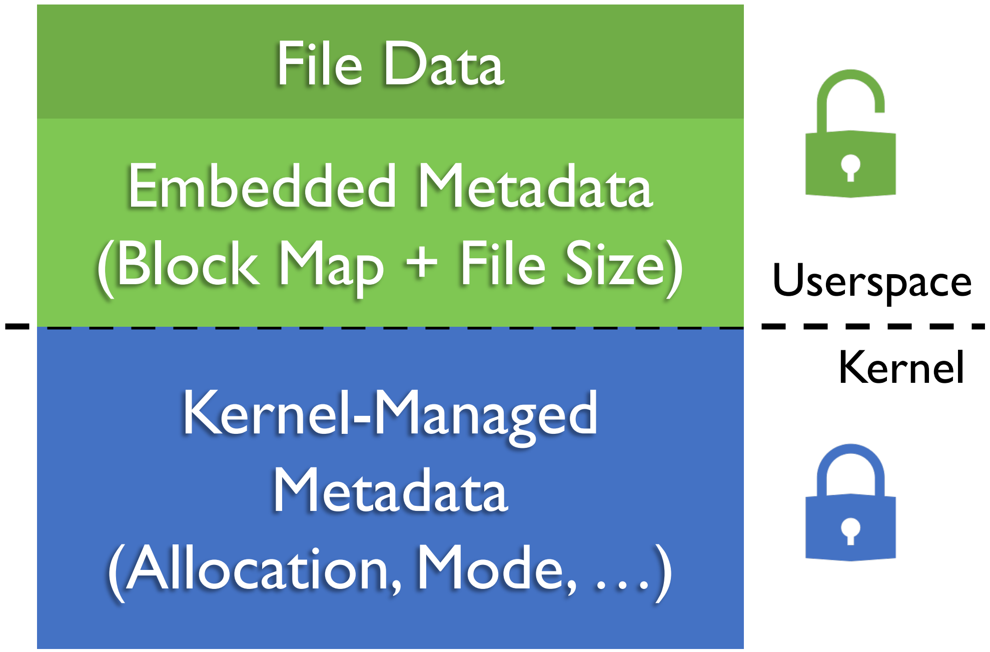
Insight: Embed metadata coupled with data ops into file data

Efficient metadata operations

- No kernel I/O stack involvement

Equivalent security guarantees

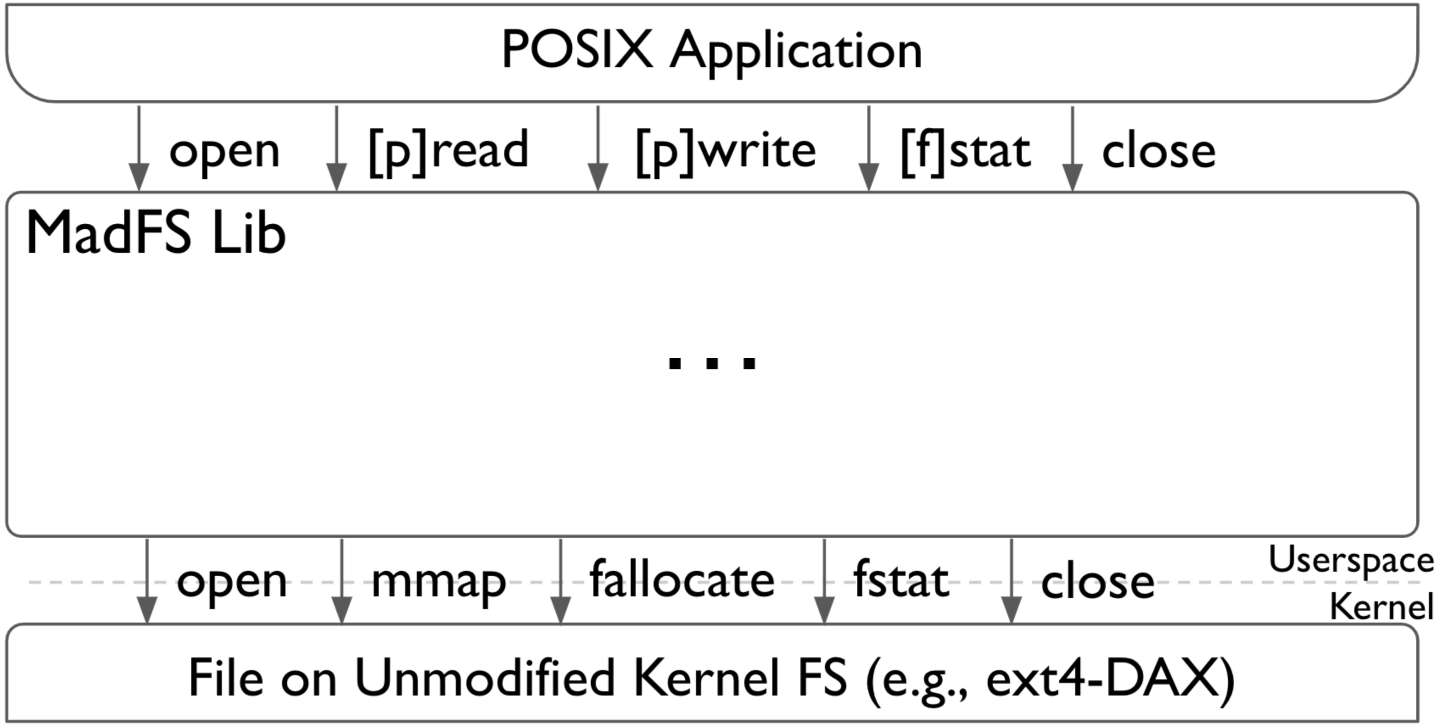
- Require write permission to modify embedded metadata



MadFS: Metadata Embedded Filesystem

Userspace library filesystem for PM

- Memory mapped I/O
- Data & most metadata ops in userspace
- Data crash consistency via copy-on-write


POSIX Application

open [p]read [p]write [f]stat close

MadFS Lib

. . .

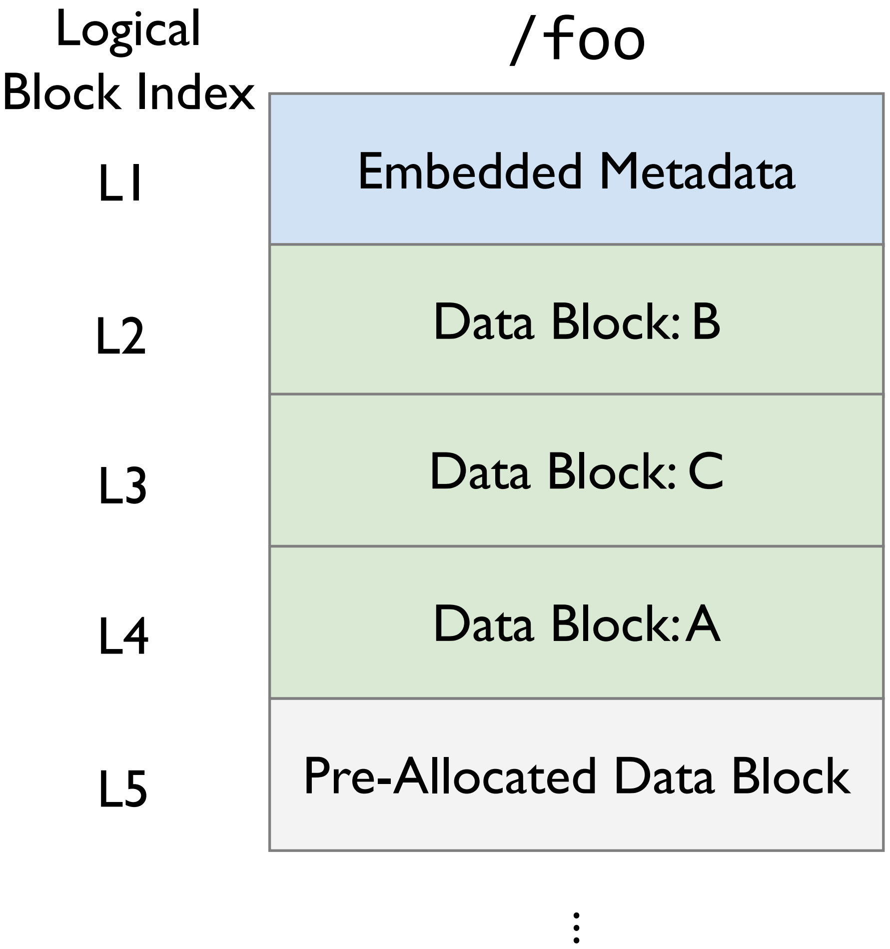
open mmap fallocate fstat close

Userspace

Kernel

File on Unmodified Kernel FS (e.g., ext4-DAX)

MadFS: Simplified Design

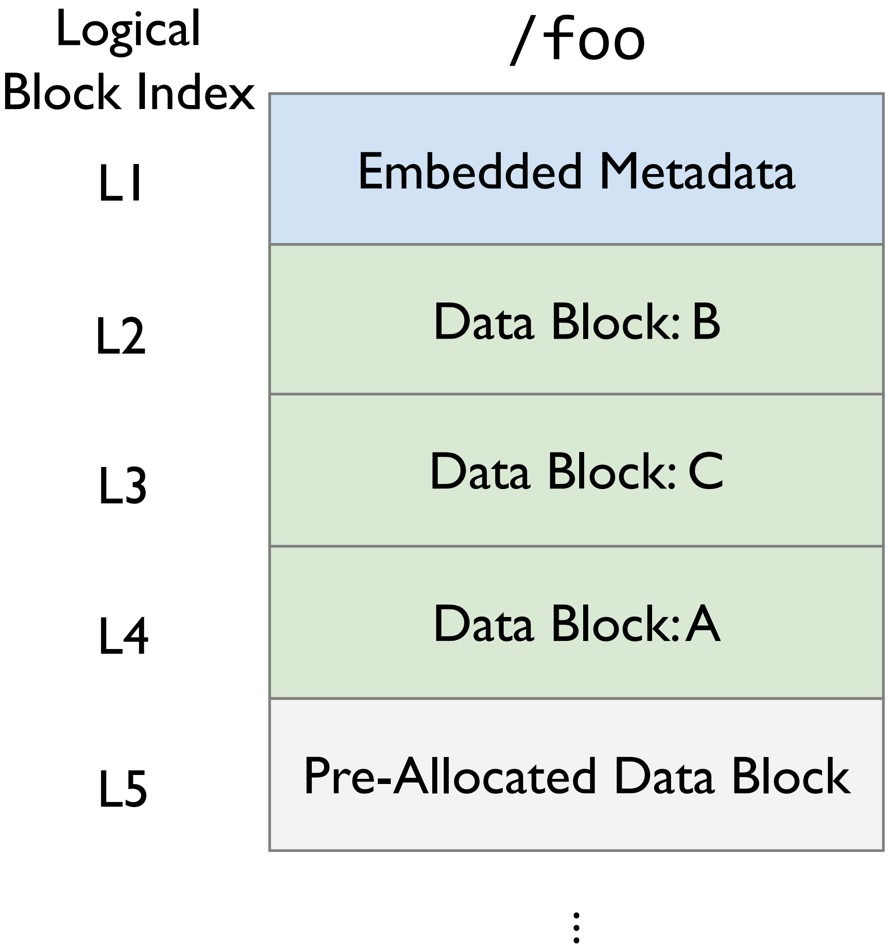


Logical Block Index	`/foo`
L1	Embedded Metadata
L2	Data Block: B
L3	Data Block: C
L4	Data Block: A
L5	Pre-Allocated Data Block
	⋮

<u>Logical</u> blocks: stored on the underlying FS

<u>Virtual</u> blocks: seen by the application

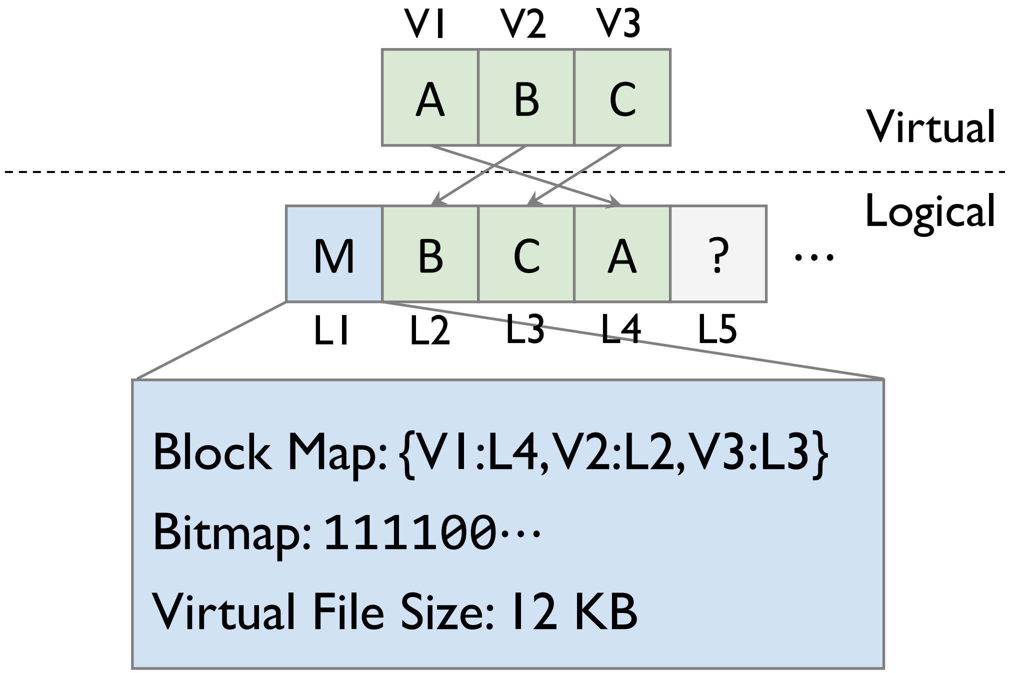
MadFS: Simplified Design



Logical Block Index	`/foo`
L1	Embedded Metadata
L2	Data Block: B
L3	Data Block: C
L4	Data Block: A
L5	Pre-Allocated Data Block
	⋮

Logical blocks: stored on the underlying FS

Virtual blocks: seen by the application



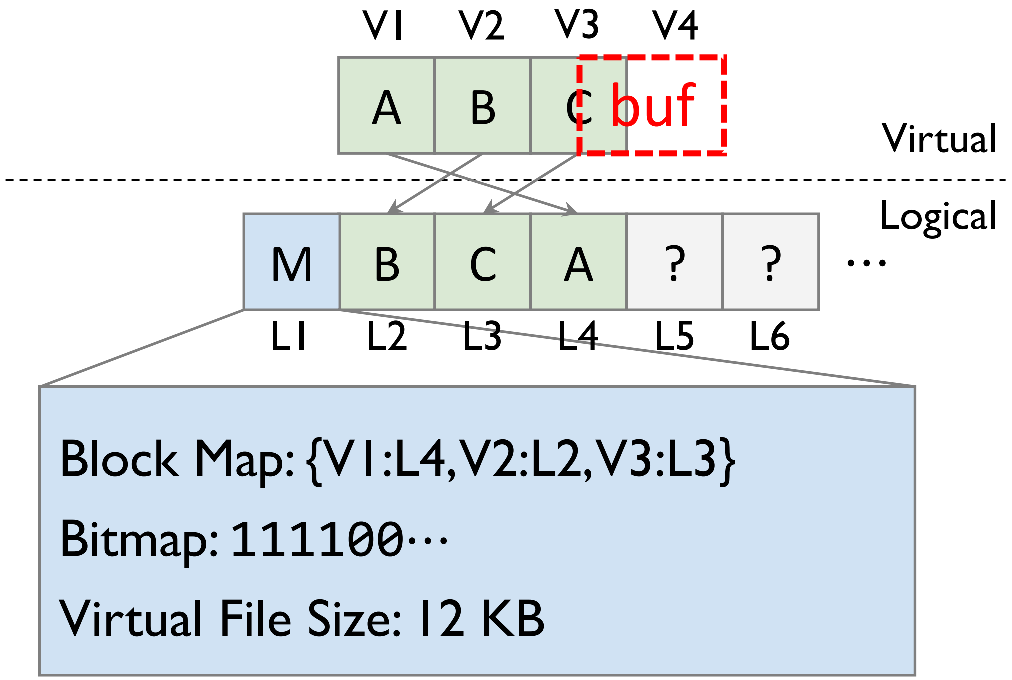
Block Map: {V1:L4,V2:L2,V3:L3}

Bitmap: 111100…

Virtual File Size: 12 KB

MadFS: Simplified Design

Example: `pwrite(fd, buf, count=6KB, offset=10KB)`



Block Map: {V1:L4, V2:L2, V3:L3}

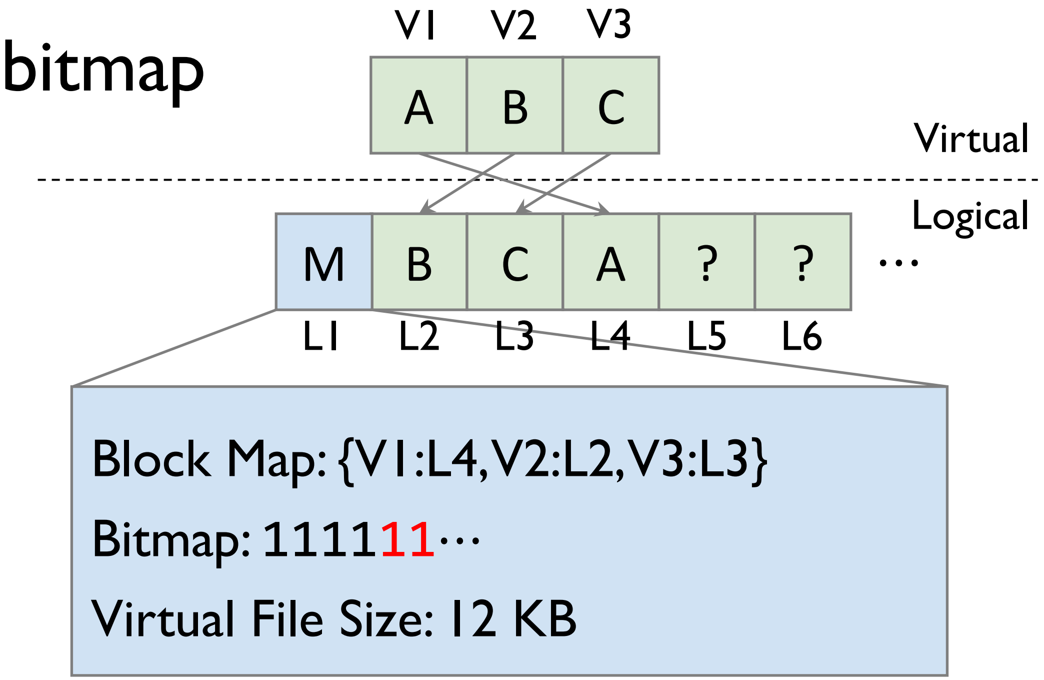
Bitmap: `111100···`

Virtual File Size: 12 KB

MadFS: Simplified Design

Example: `pwrite(fd, buf, count=6KB, offset=10KB)`

1. Allocate 2 logical blocks from the bitmap



Block Map: {V1:L4, V2:L2, V3:L3}

Bitmap: 111111···

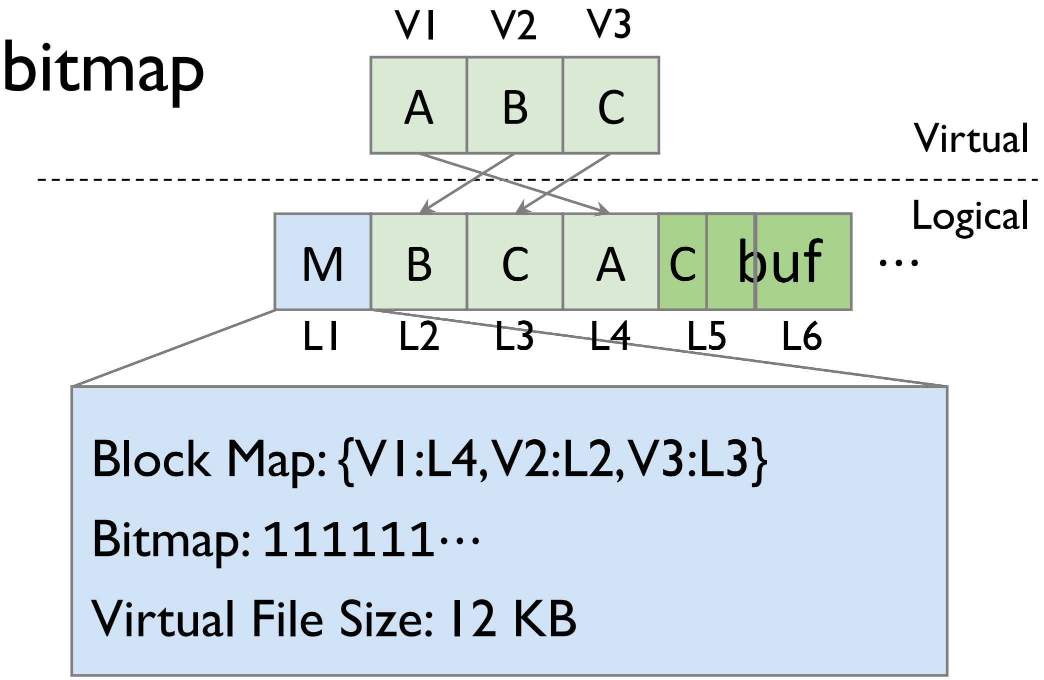
Virtual File Size: 12 KB

MadFS: Simplified Design

Example: `pwrite(fd, buf, count=6KB, offset=10KB)`

1. Allocate 2 logical blocks from the bitmap

2. Copy buffer and unaligned data



Block Map: {V1:L4, V2:L2, V3:L3}

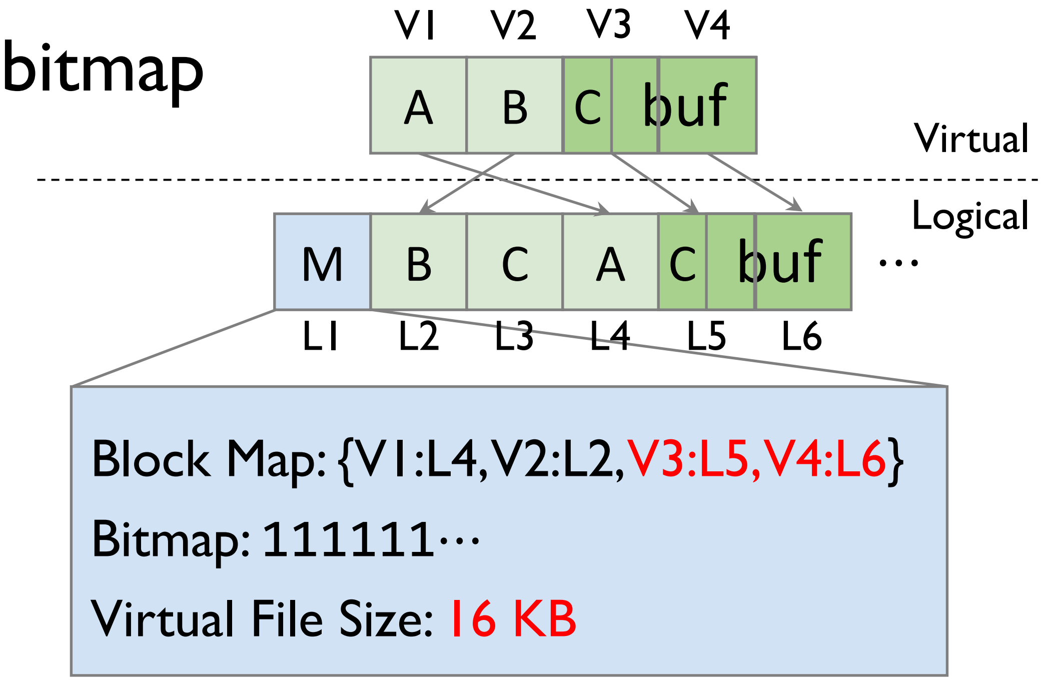
Bitmap: 111111…

Virtual File Size: 12 KB

MadFS: Simplified Design

Example: `pwrite(fd, buf, count=6KB, offset=10KB)`

1. Allocate 2 logical blocks from the bitmap
2. Copy buffer and unaligned data
3. Update block map and virtual size



Block Map: {V1:L4,V2:L2,V3:L5,V4:L6}

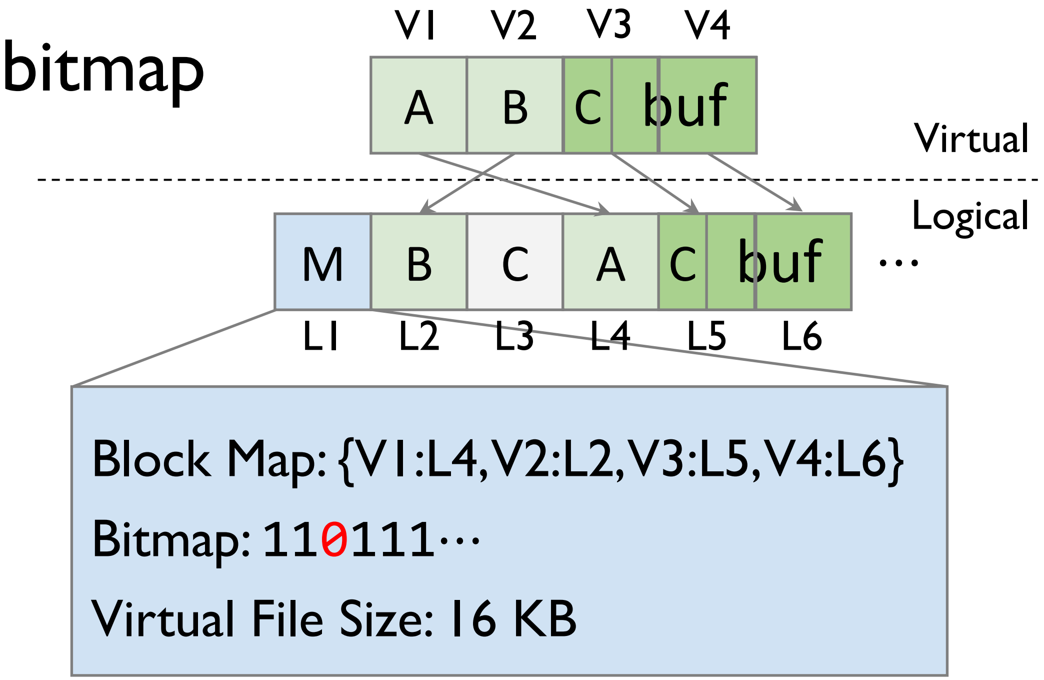
Bitmap: 111111⋯

Virtual File Size: 16 KB

MadFS: Simplified Design

Example: `pwrite(fd, buf, count=6KB, offset=10KB)`

1. Allocate 2 logical blocks from the bitmap
2. Copy buffer and unaligned data
3. Update block map and virtual size
4. Deallocate old blocks



Block Map: {V1:L4,V2:L2,V3:L5,V4:L6}

Bitmap: 110111…

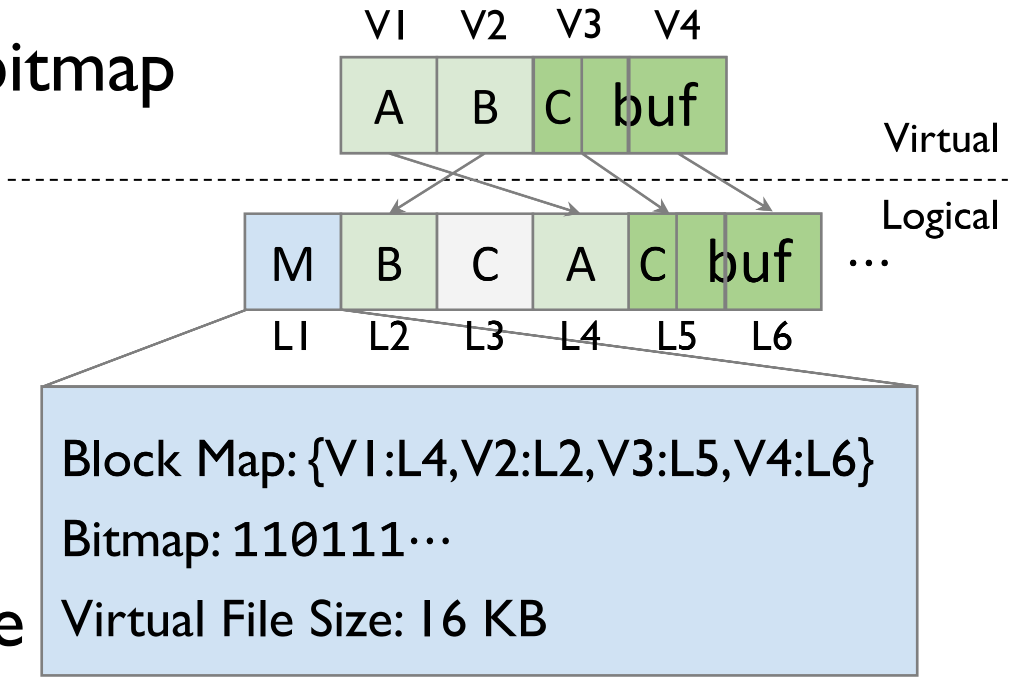
Virtual File Size: 16 KB

MadFS: Simplified Design

Example: `pwrite(fd, buf, count=6KB, offset=10KB)`

1. Allocate 2 logical blocks from the bitmap
2. Copy buffer and unaligned data
3. Update block map and virtual size
4. Deallocate old blocks

Copy-on-write & append in userspace



Block Map: {V1:L4,V2:L2,V3:L5,V4:L6}

Bitmap: `110111`…

Virtual File Size: 16 KB

MadFS: Metadata Management

Embedded Metadata

- Virtual-to-logical map
- Virtual file size
- Logical blocks bitmap

Allows efficient data ops without expensive kernel involvement

Kernel-Managed Metadata

MadFS: Metadata Management

Embedded Metadata

- Virtual-to-logical map
- Virtual file size
- Logical blocks bitmap

Allows efficient data ops without expensive kernel involvement

Kernel-Managed Metadata

- Logical-to-physical map
- Logical file size
- Physical blocks bitmap
- File permission

Updated on pre-allocation (infrequent)

Provides coarse-grained allocation and protection

MadFS: Per-File Virtualization

A userspace virtualization layer implements a complete set of file functionalities, including metadata management, crash consistency, and concurrency control, on a per-file basis

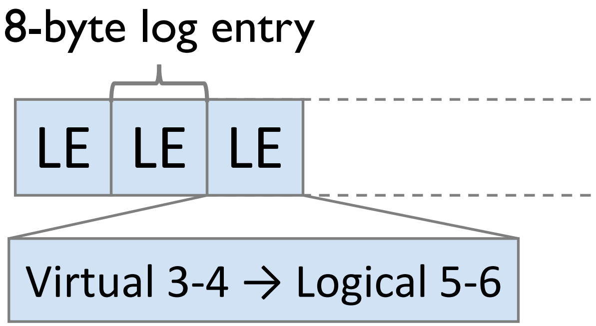
Metadata Management

Crash Consistency

Concurrency Control

MadFS: Full Design (Details in Paper)

- 💥 Metadata Crash Consistency
  - Log-structured metadata with 8-byte log entries



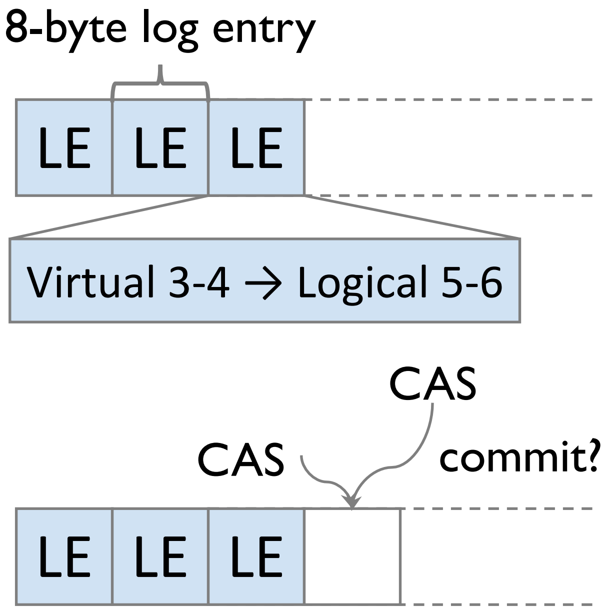
MadFS: Full Design (Details in Paper)

💥 Metadata Crash Consistency

- Log-structured metadata with 8-byte log entries

🔀 Lock-Free Optimistic Concurrency Control

- Commit log entry via compare-and-swap (CAS)
- Safe in presence of process crashes
- Better scalability with concurrent data ops



MadFS: Full Design (Details in Paper)

💥 Metadata Crash Consistency

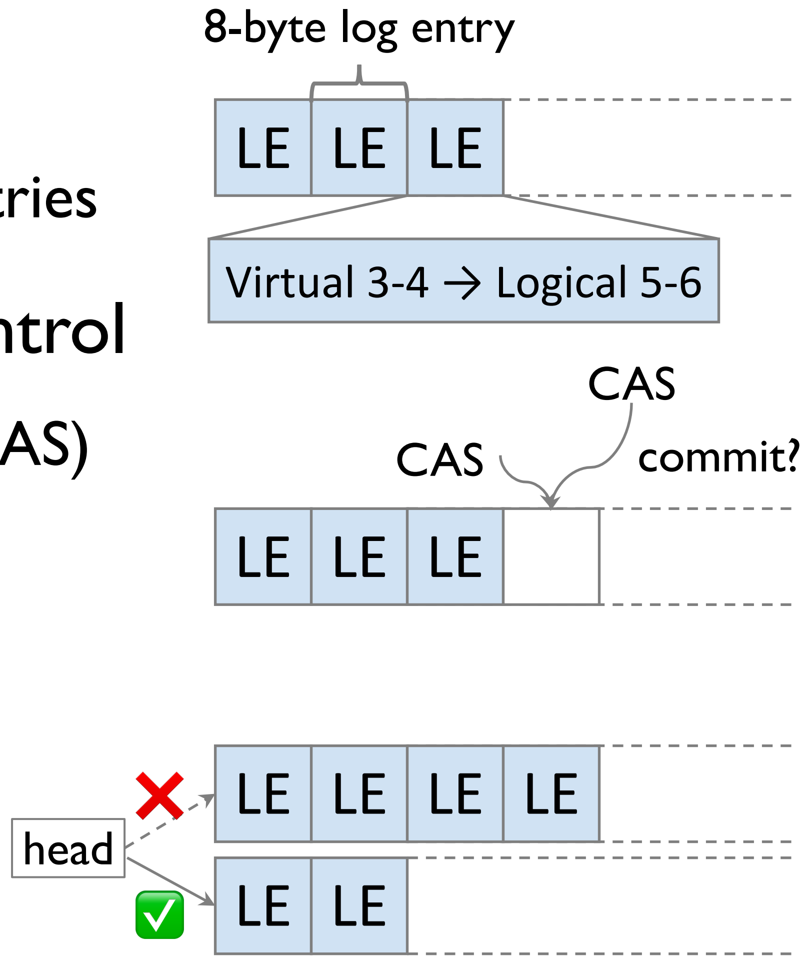
- Log-structured metadata with 8-byte log entries

🔀 Lock-Free Optimistic Concurrency Control

- Commit log entry via compare-and-swap (CAS)
- Safe in presence of process crashes
- Better scalability with concurrent data ops

♻️ Non-Blocking Garbage Collection

- Read-Copy Update w/o tail latency impact



Evaluation

Questions:

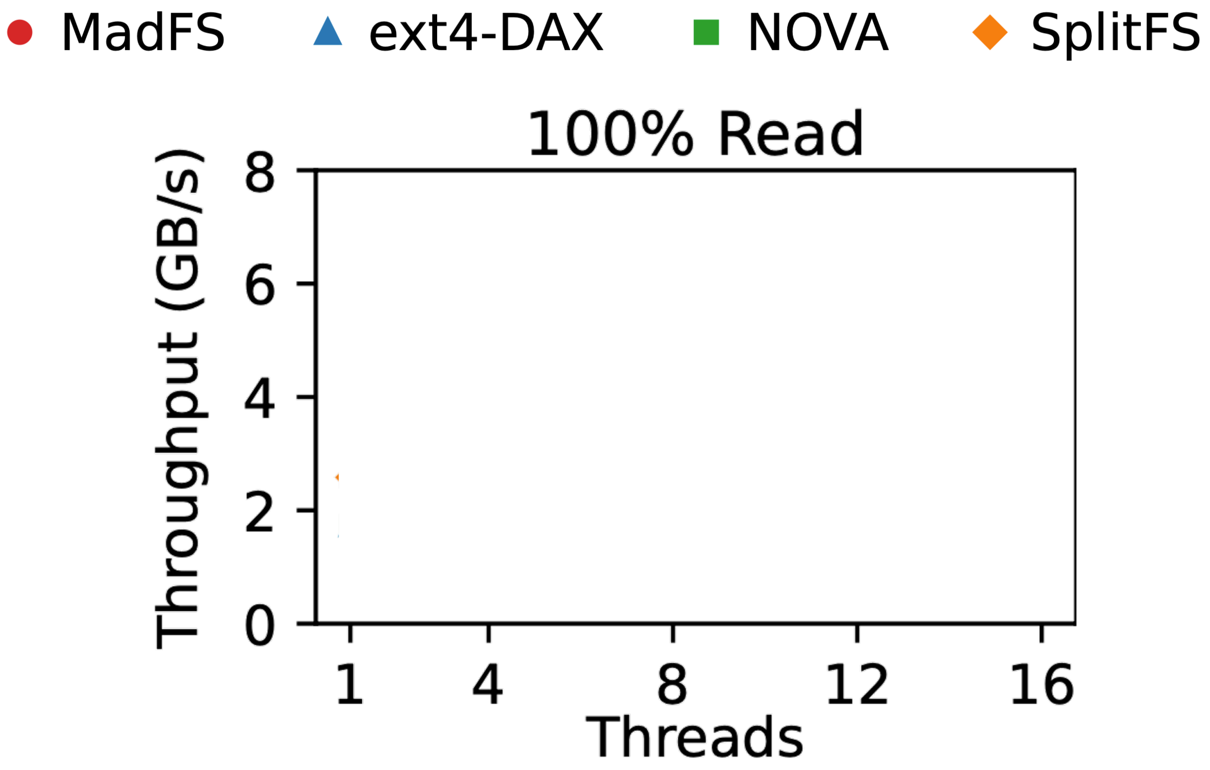
- How does MadFS perform on microbenchmarks?
- How does MadFS perform on real-world applications?

Compare MadFS running on ext4-DAX with

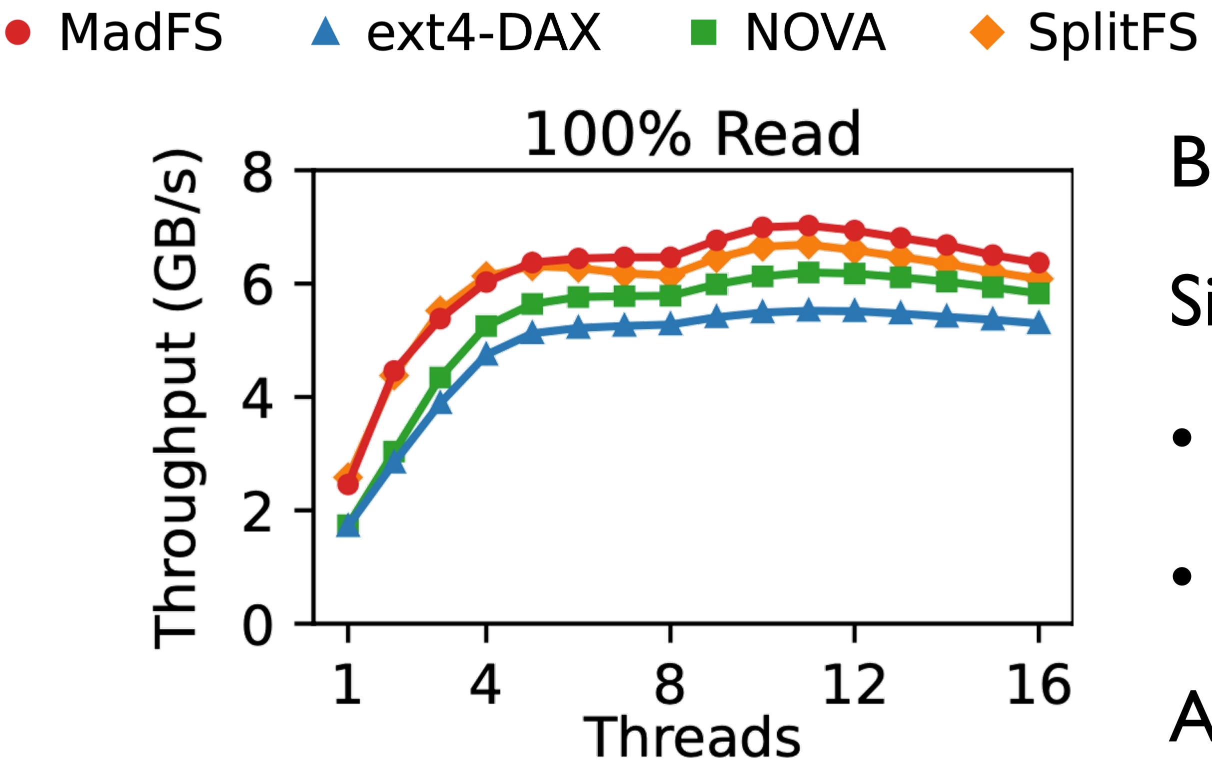
- ext4-DAX, NOVA [FAST '16], SplitFS [SOSP '19]

Hardware: 8-core Intel Xeon 4215R CPU
1 × 128GB Intel Optane PM

Evaluation: Concurrent 4 KB Random Read



Evaluation: Concurrent 4 KB Random Read



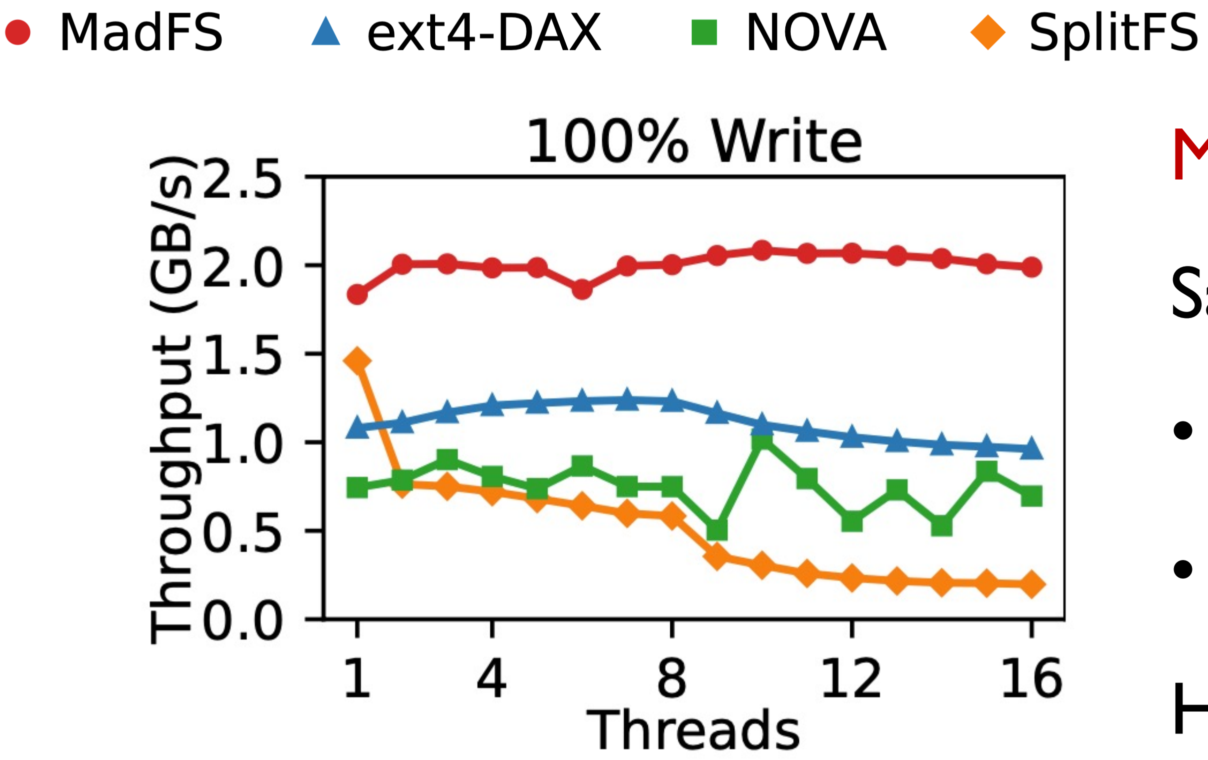
Best performance: MadFS

Single thread

- 43% faster than ext4-DAX
- 41% faster than NOVA

All FS scale well: no writes

Evaluation: Concurrent 4 KB Random Overwrite



MadFS doesn’t update kernel metadata

Saturates device bandwidth w/ 1 thread

- 26% faster than SplitFS
- 70% faster than ext4-DAX

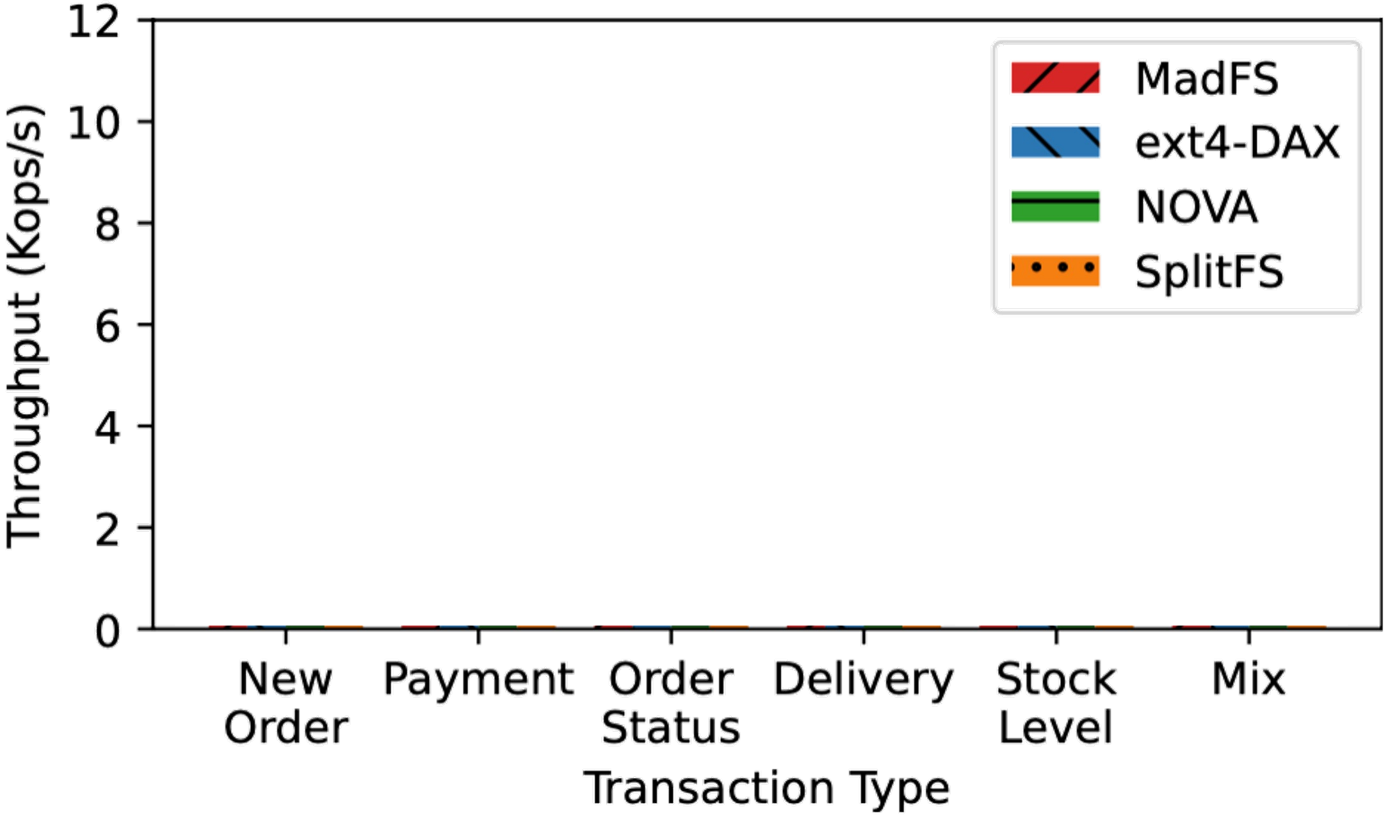
High throughput w/ more threads

- Lock-free concurrency control

Evaluation: TPC-C on SQLite

Transaction processing benchmark on relational database

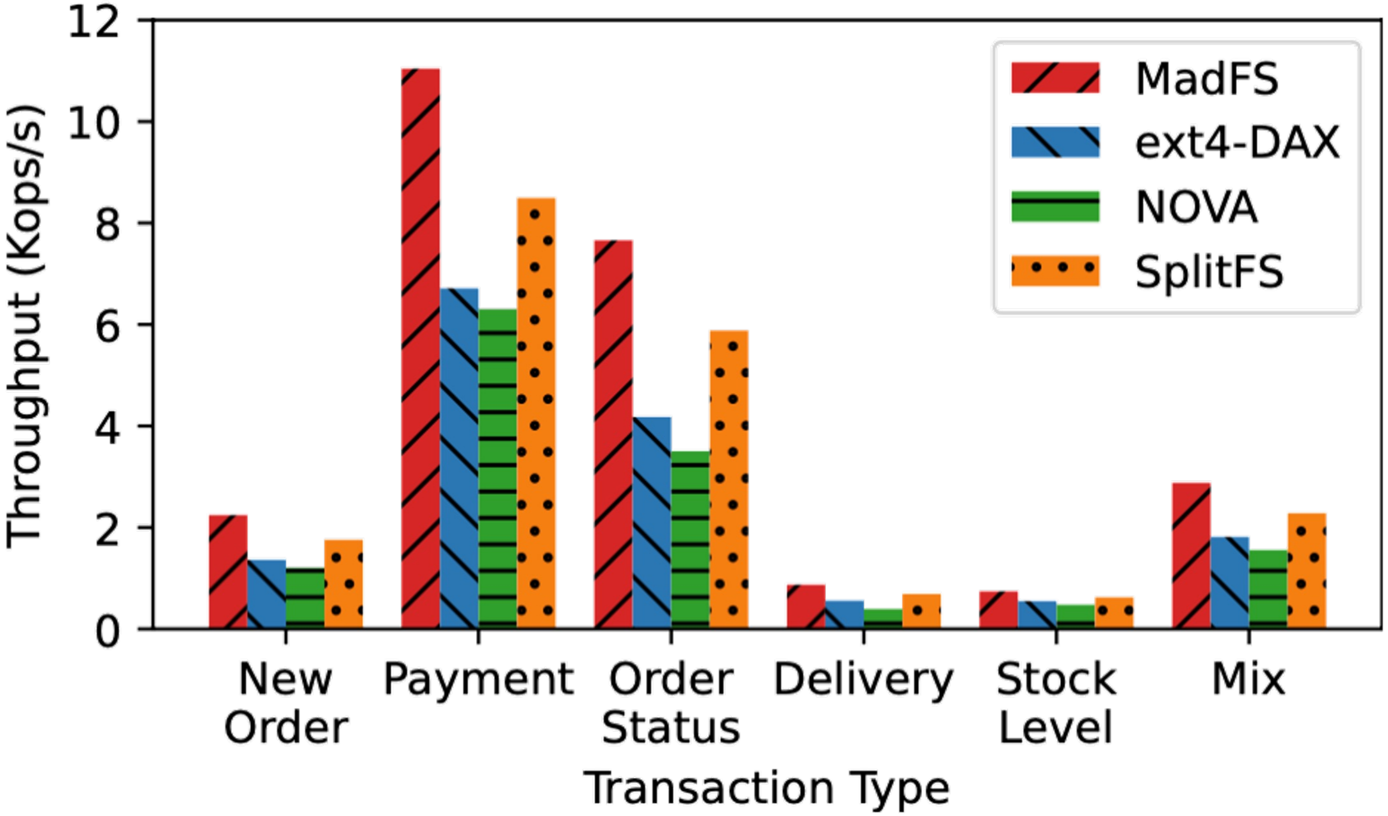
Characteristic: block-aligned writes followed by fsync



Evaluation: TPC-C on SQLite

Result: MadFS outperforms other filesystems

Mix: 26% faster than SplitFS, 58% faster than ext4-DAX



Evaluation: More in Paper

Multi-threaded benchmarks

- Contended concurrent write
- Concurrency control comparison

Metadata operations

- Open latency
- Garbage collection

Macro-benchmarks

- YCSB on LevelDB

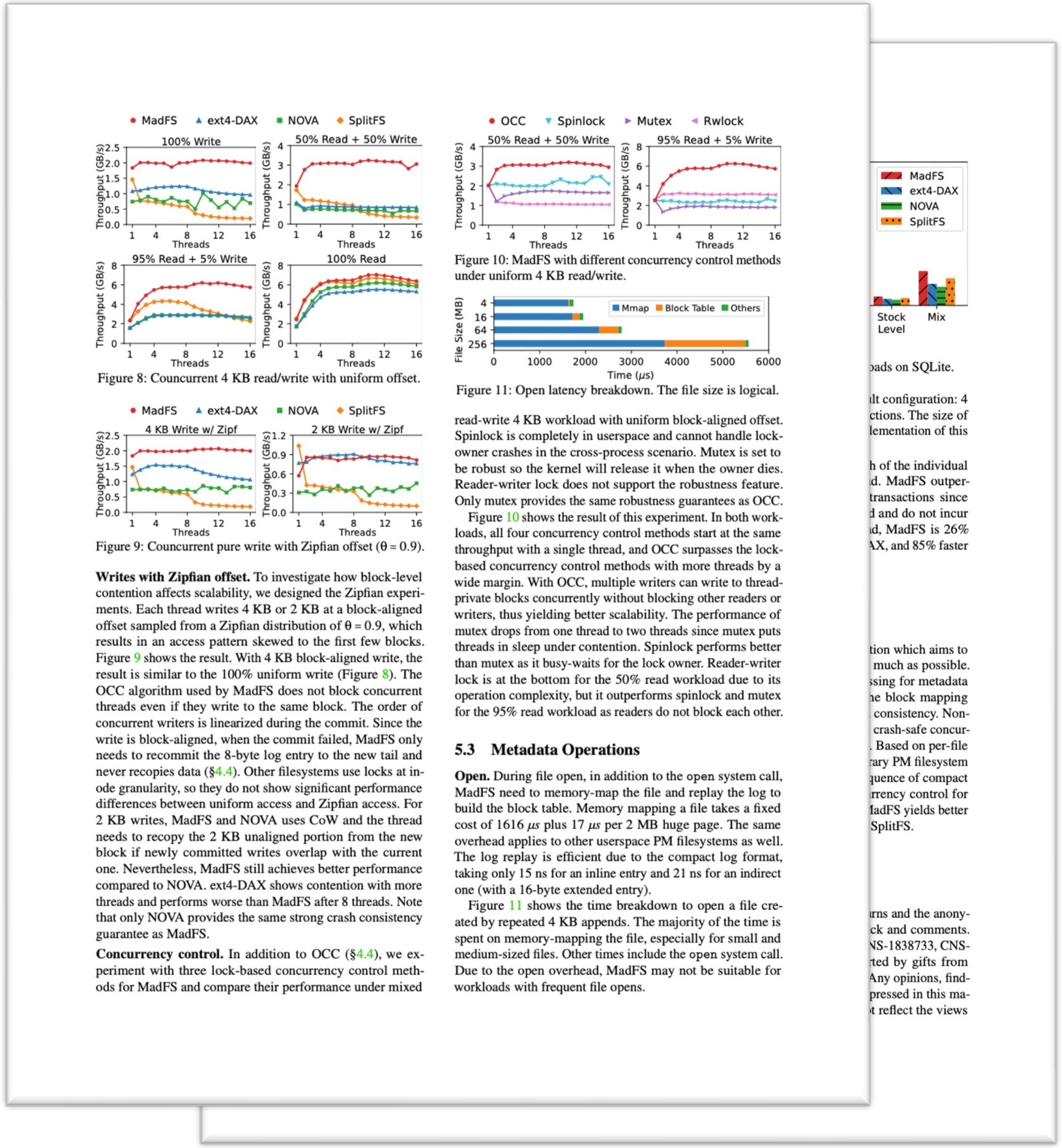


Figure 8: Concurrent 4 KB read/write with uniform offset.

Figure 9: Councurrent pure write with Zipfian offset (θ = 0.9).

Writes with Zipfian offset. To investigate how block-level contention affects scalability, we designed the Zipfian experiments. Each thread writes 4 KB or 2 KB at a block-aligned offset sampled from a Zipfian distribution of θ = 0.9, which results in an access pattern skewed to the first few blocks. Figure 9 shows the result. With 4 KB block-aligned write, the result is similar to the 100% uniform write (Figure 8). The OCC algorithm used by MadFS does not block concurrent threads even if they write to the same block. The order of concurrent writers is linearized during the commit. Since the write is block-aligned, when the commit failed, MadFS only needs to recommit the 8-byte log entry to the new tail and never recopies data (§4.4). Other filesystems use locks at inode granularity, so they do not show significant performance differences between uniform access and Zipfian access. For 2 KB writes, MadFS and NOVA uses CoW and the thread needs to recopy the 2 KB unaligned portion from the new block if newly committed writes overlap with the current one. Nevertheless, MadFS still achieves better performance compared to NOVA. ext4-DAX shows contention with more threads and performs worse than MadFS after 8 threads. Note that only NOVA provides the same strong crash consistency guarantee as MadFS.

Concurrency control. In addition to OCC (§4.4), we experiment with three lock-based concurrency control methods for MadFS and compare their performance under mixed read-write 4 KB workload with uniform block-aligned offset. Spinlock is completely in userspace and cannot handle lock-owner crashes in the cross-process scenario. Mutex is set to be robust so the kernel will release it when the owner dies. Reader-writer lock does not support the robustness feature. Only mutex provides the same robustness guarantees as OCC.

Figure 10 shows the result of this experiment. In both workloads, all four concurrency control methods start at the same throughput with a single thread, and OCC surpasses the lock-based concurrency control methods with more threads by a wide margin. With OCC, multiple writers can write to thread-private blocks concurrently without blocking other readers or writers, thus yielding better scalability. The performance of mutex drops from one thread to two threads since mutex puts threads in sleep under contention. Spinlock performs better than mutex as it busy-waits for the lock owner. Reader-writer lock is at the bottom for the 50% read workload due to its operation complexity, but it outperforms spinlock and mutex for the 95% read workload as readers do not block each other.

Figure 10: MadFS with different concurrency control methods under uniform 4 KB read/write.

Figure 11: Open latency breakdown. The file size is logical.

5.3 Metadata Operations

Open. During file open, in addition to the open system call, MadFS need to memory-map the file and replay the log to build the block table. Memory mapping a file takes a fixed cost of 1616 µs plus 17 µs per 2 MB huge page. The same overhead applies to other userspace PM filesystems as well. The log replay is efficient due to the compact log format, taking only 15 ns for an inline entry and 21 ns for an indirect one (with a 16-byte extended entry).

Figure 11 shows the time breakdown to open a file created by repeated 4 KB appends. The majority of the time is spent on memory-mapping the file, especially for small and medium-sized files. Other times include the open system call. Due to the open overhead, MadFS may not be suitable for workloads with frequent file opens.

Conclusion

Metadata Embedding

- Many data ops coupled with metadata updates ⇒ expensive kernel I/O stack
- Embed metadata into file data for efficient userspace management

Per-File Virtualization

- Push file functionalities into userspace as much as possible

MadFS: Metadata Embedded Filesystem

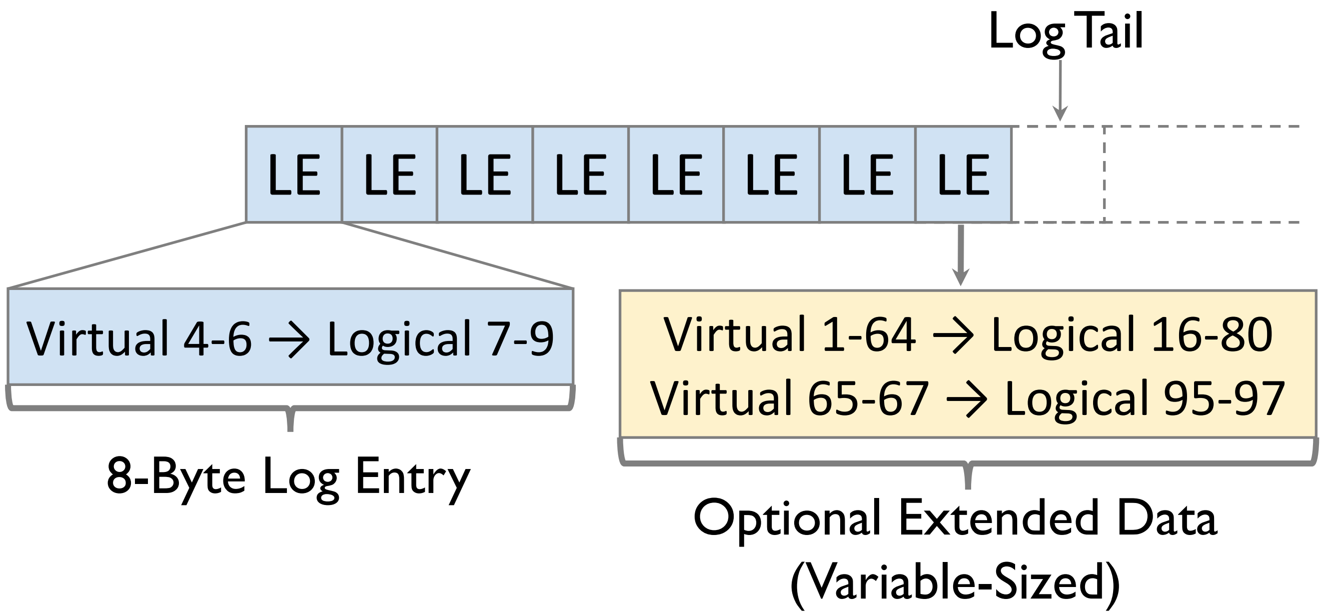
- Highly-scalable userspace PM filesystem with strong crash consistency

Questions

Backup Slides

MadFS: Log-Structured Metadata

Fix-sized 8-byte log entries + optional extended data

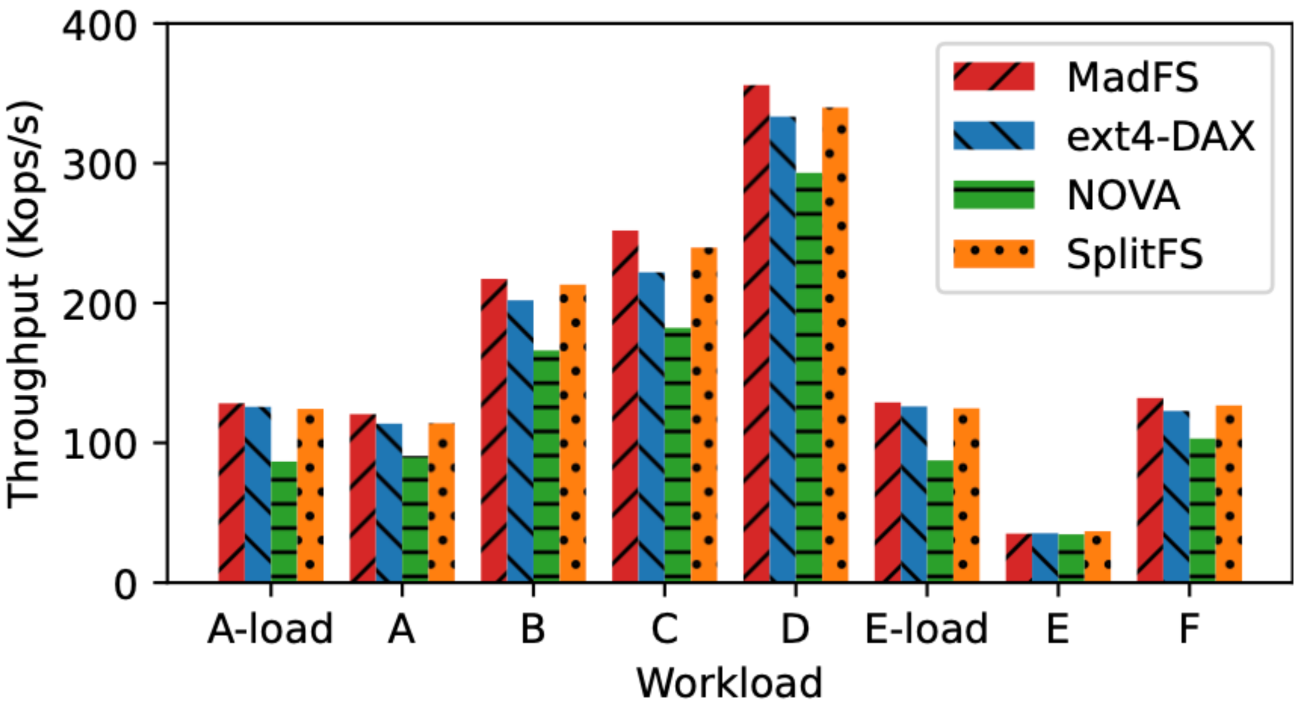


MadFS: Lock-Free Optimistic Concurrency Control

Use Compare-and-Swap to commit 8-byte log entry

1. *Begin*
   Record Starting Log Tail
2. *Execute*
   Copy-on-Write
3. *Validate*
   “Compare” if Log Tail still Points to an Empty Slot
4. *Commit*
   “Swap” the Log Entry to the Tail

Evaluation: YCSB on LevelDB



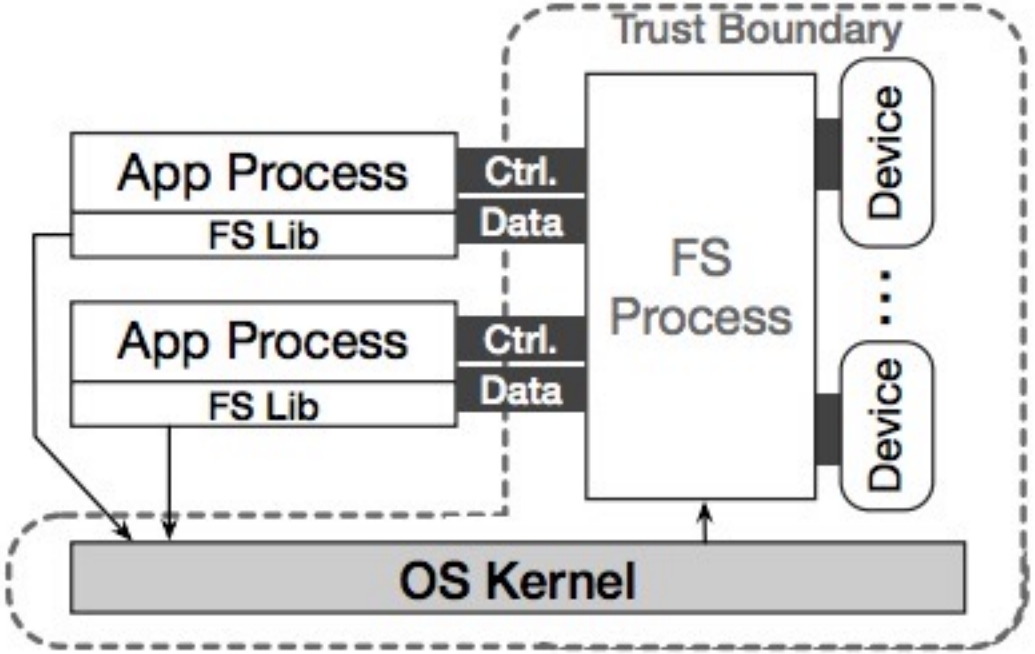
Workload C

- 5% faster than SplitFS
- 12% faster than ext4-DAX

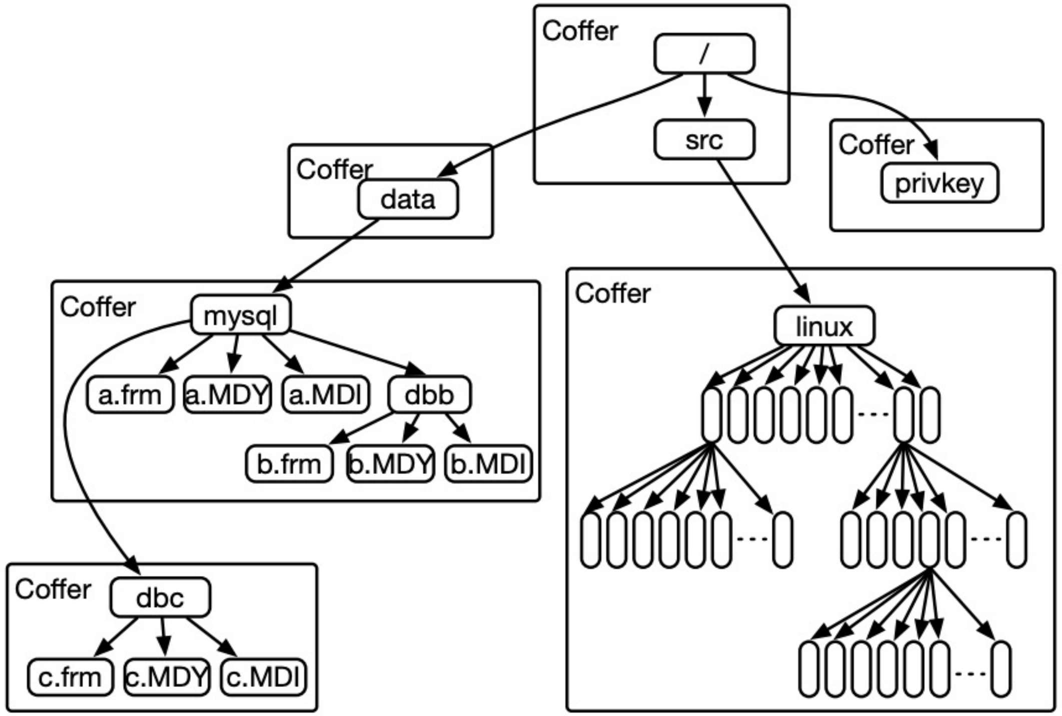
Workload F

- 4% faster than SplitFS
- 7% faster than ext4-DAX

Related Work



Aerie



ZoFS