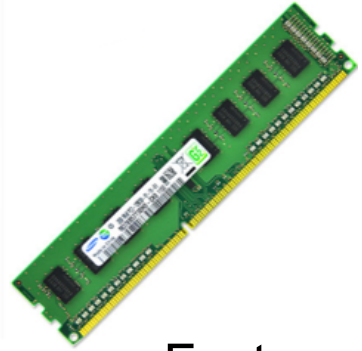
SDC 20
BY Developers FOR Developers
Storage Developer Conference
September 22-23, 2020

SplitFS: Reducing Software Overhead in File Systems for Persistent Memory

Vijay Chidambaram
University of Texas at Austin

Persistent Memory (PM)

Non-volatile

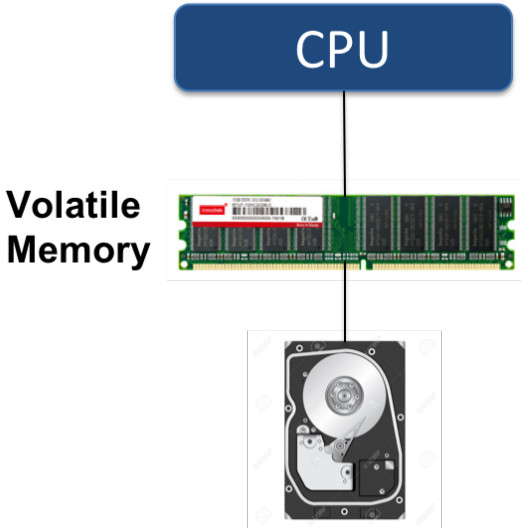


Fast

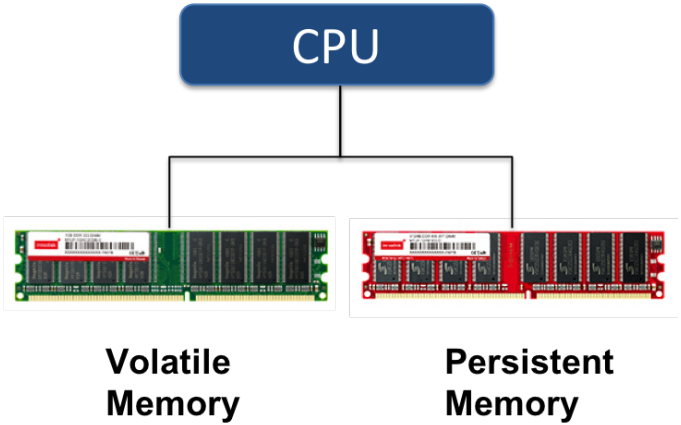
Optane DC PM	Latency	Bandwidth
Loads	300 ns	1/3 of DRAM
Stores	100 ns	1/6 of DRAM

Storage Hierarchy

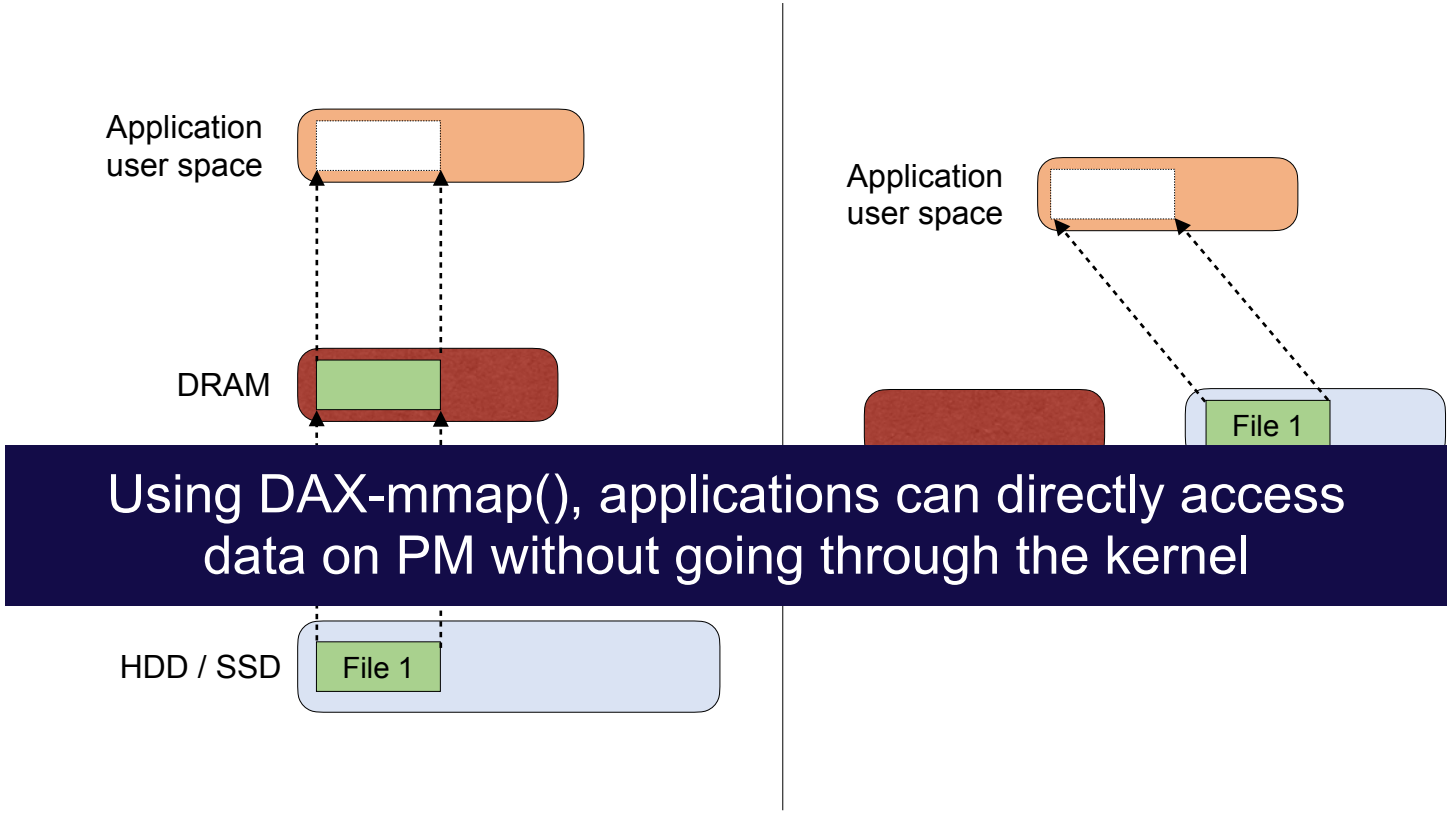
Conventional



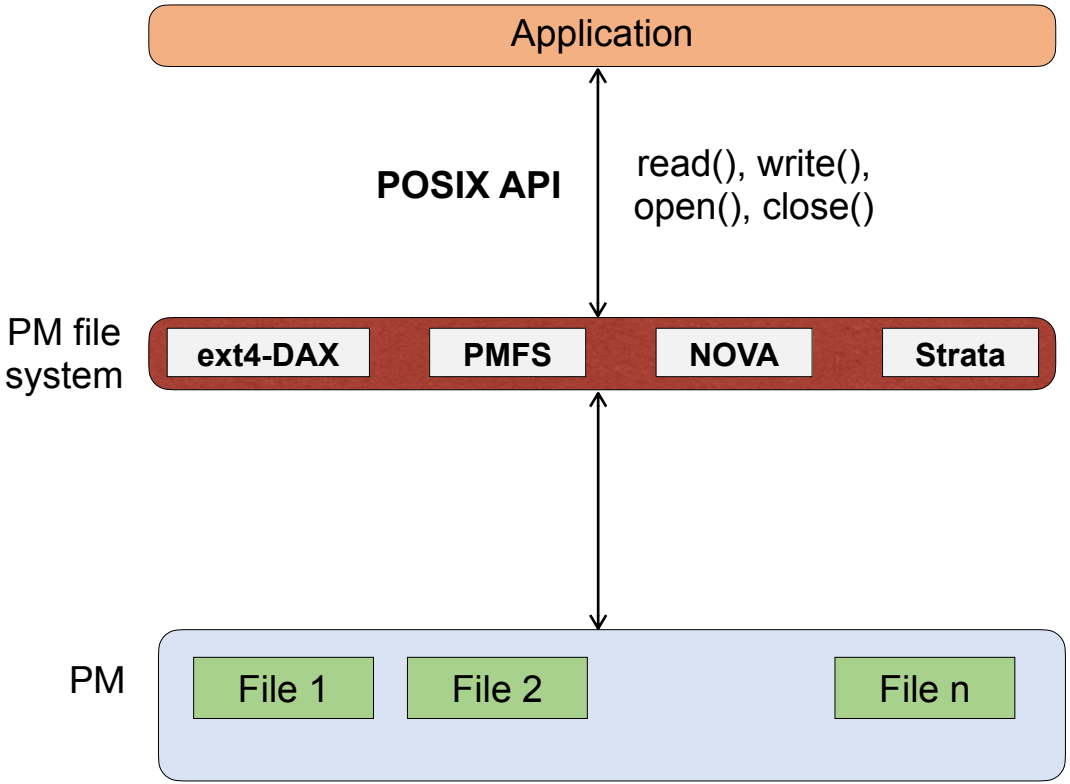
With Persistent Memory



DAX



PM File Systems



POSIX

Commonly used API by applications for accessing data in files

Data operations: read(), write()

Metadata operations: open(), unlink(), rename(), etc

File system guarantees:

- metadata atomicity
- data atomicity

ext4-DAX

Modification of the ext4 file system for Persistent Memory

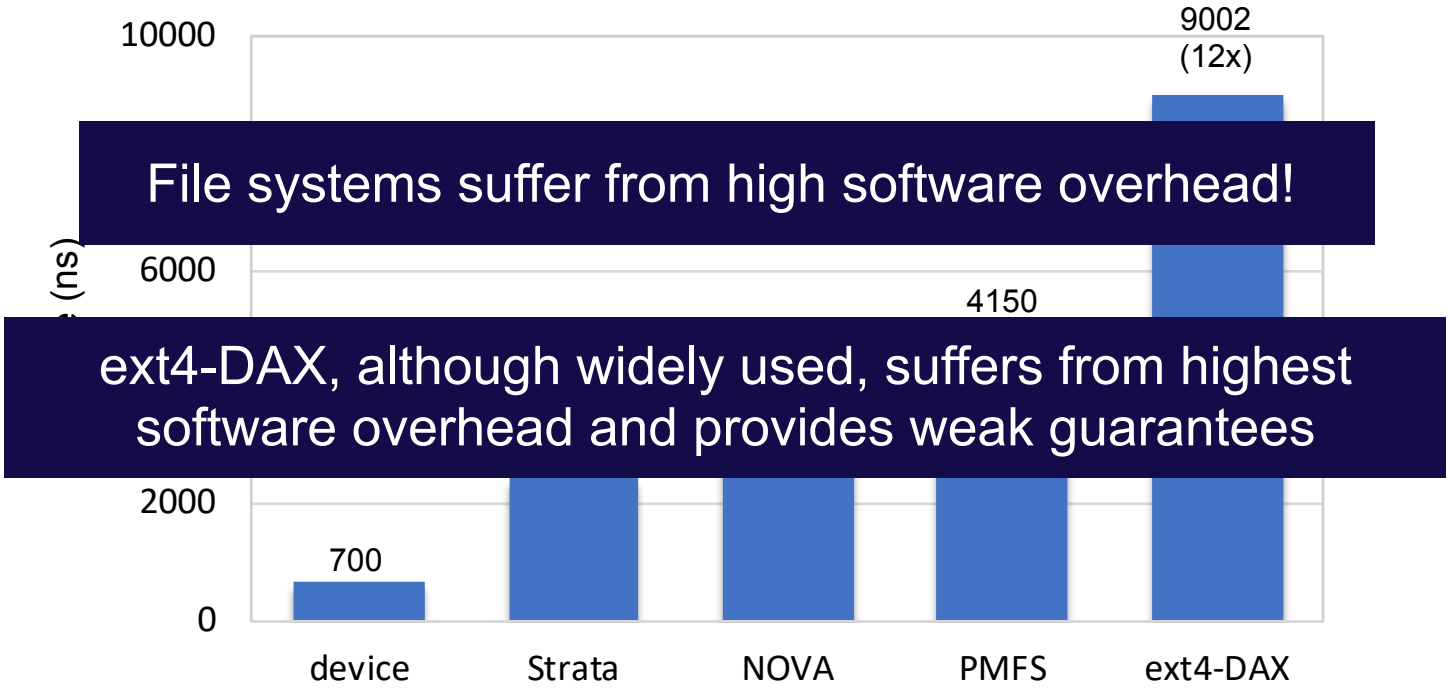
Works with modern Linux kernels

Under active development by the ext4 community

Only PM file system that is widely used

Software overhead in File Systems

- Append 4KB data to a file
- Time taken to copy user data to PM: ~700 ns



Goals

- Low Software Overhead
- Strong Consistency Guarantees
- Leverage the maturity and active development of ext4-DAX

SplitFS

SDC 20

POSIX file system aimed at reducing software overhead for PM

SplitFS serves data operations from user space and metadata operations using the ext4-DAX kernel file system

Provides strong guarantees such as atomic and synchronous data operations

Reduces software overhead by up to 17x compared to ext4-DAX

Improves application throughput by up to 2x compared to NOVA

Outline

- Target Usage Scenario
- High-level design
- Handling data operations
- Consistency Guarantees
- Evaluation

Outline

- Target Usage Scenario
- High-level design
- Handling data operations
- Consistency Guarantees
- Evaluation

Target Usage Scenario

SplitFS is targeted at POSIX applications which use read() / write() system calls in order to access their data on Persistent Memory.

SplitFS does not optimize for the case when multiple processes concurrently access the same file

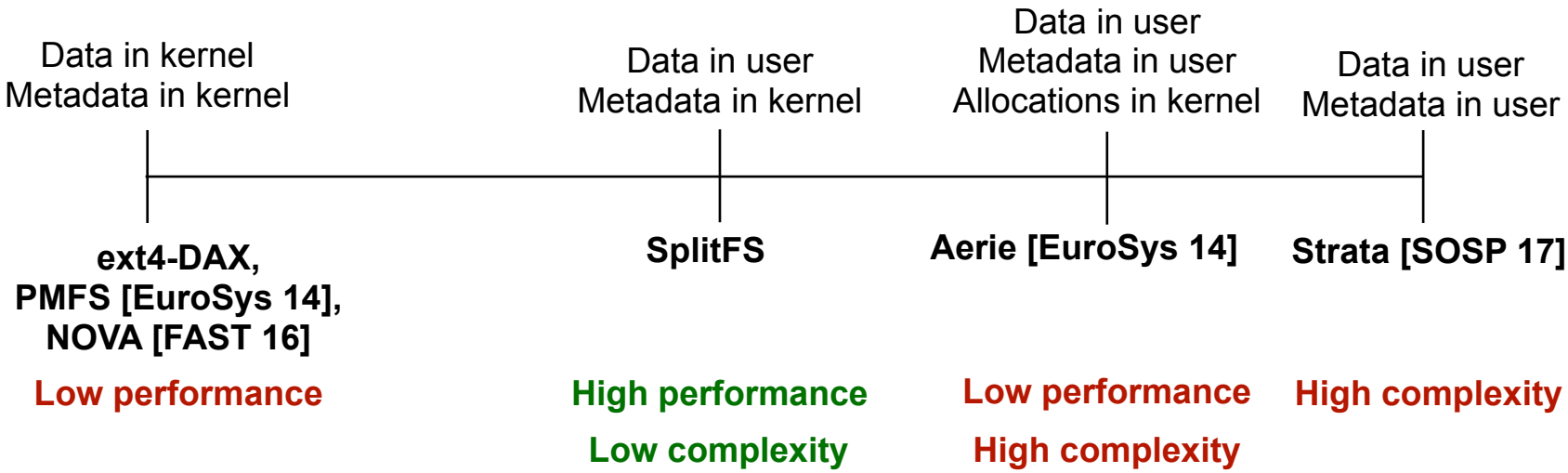
Outline

- Target Usage Scenario
- High-level design
- Handling data operations
- Consistency Guarantees
- Evaluation

High-level Design

SplitFS lies both in user space as well as in the kernel.

- Data operations are served from user space
- Metadata operations are served from ext4-DAX



High-level Design

High performance

Accelerate data operations from user space

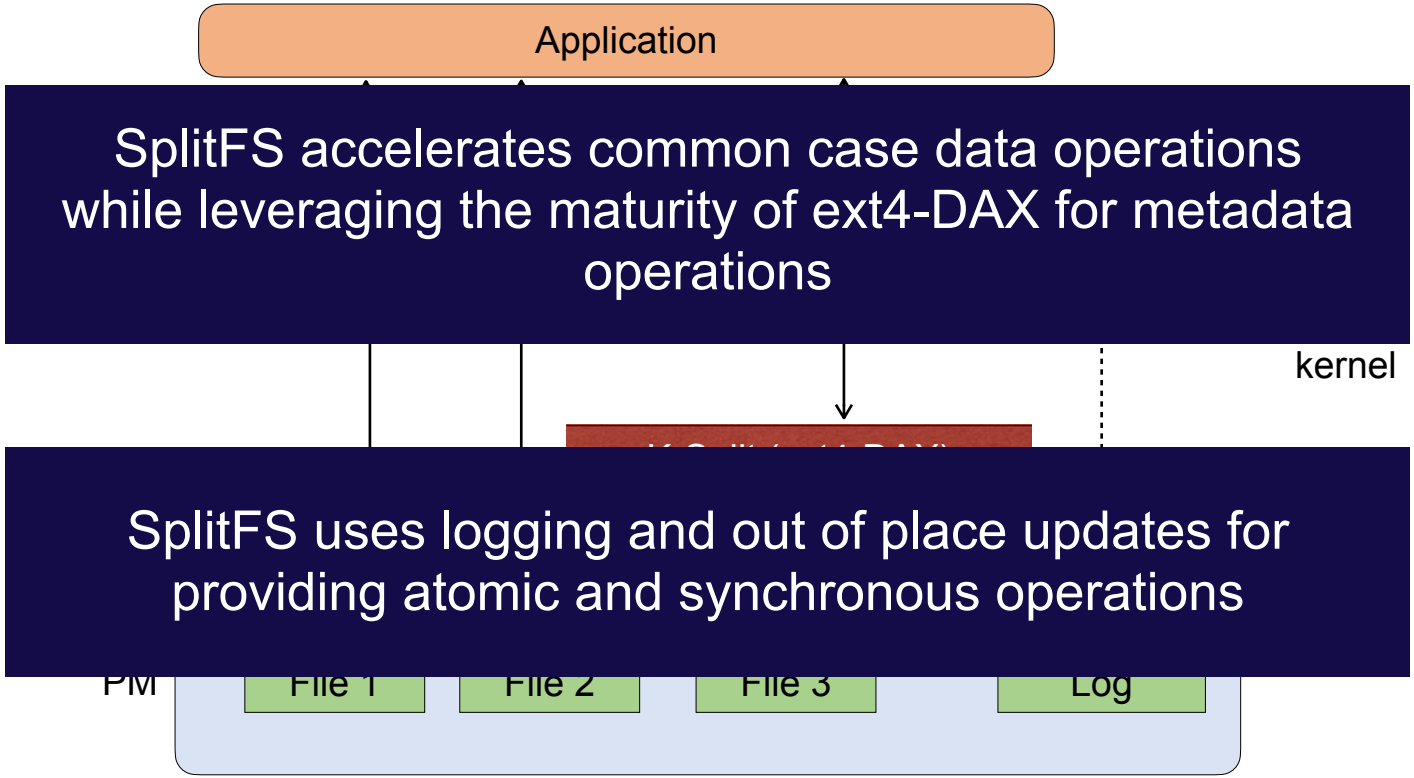
- Data operations are common and simple

Low complexity

Use ext4-DAX for metadata operations

- Metadata operations are rare and complex
- POSIX has many complex corner-cases

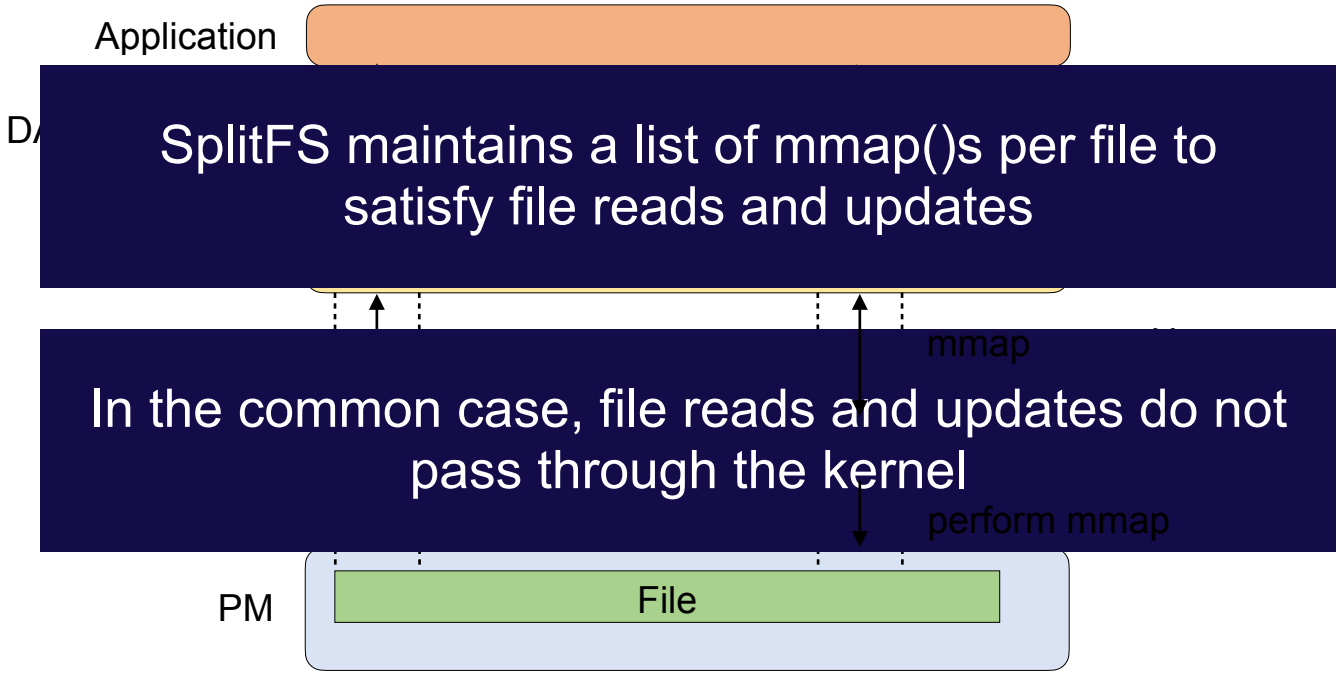
High-level Design



Outline

- Target Usage Scenario
- High-level design
- Handling data operations
  - Handling file reads & updates
  - Handling file appends
- Consistency Guarantees
- Evaluation

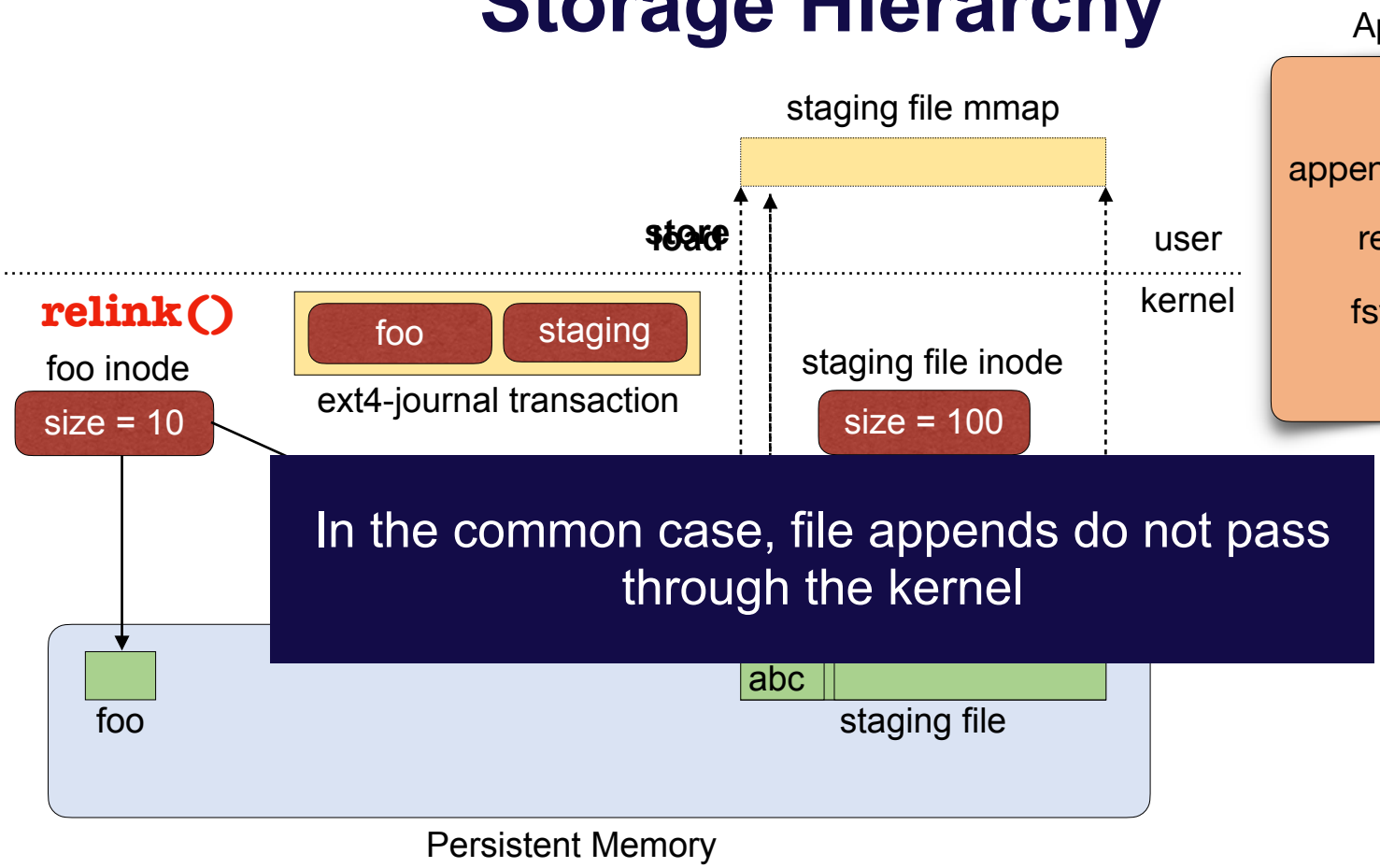
Handling reads & updates



Outline

- Target Usage Scenario
- High-level design
- Handling data operations
  - Handling file reads & updates
  - Handling file appends
- Consistency Guarantees
- Evaluation

Storage Hierarchy



Staging & Relink

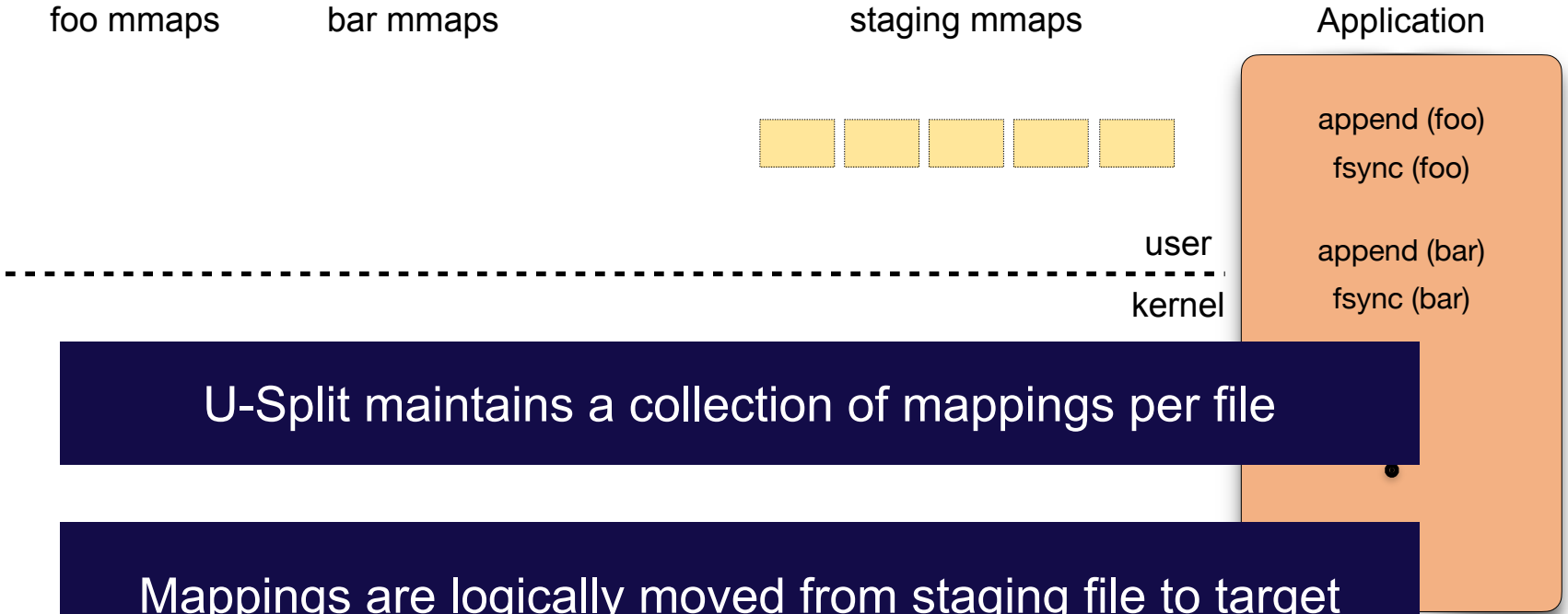
Configurable number of staging files per application

Create and map more staging files in the background as they are used up

Do not unmap staging file regions on relink. Re-use the same mapping for the target files

Maintain metadata for a collection of memory mappings per file

Collection of memory mappings



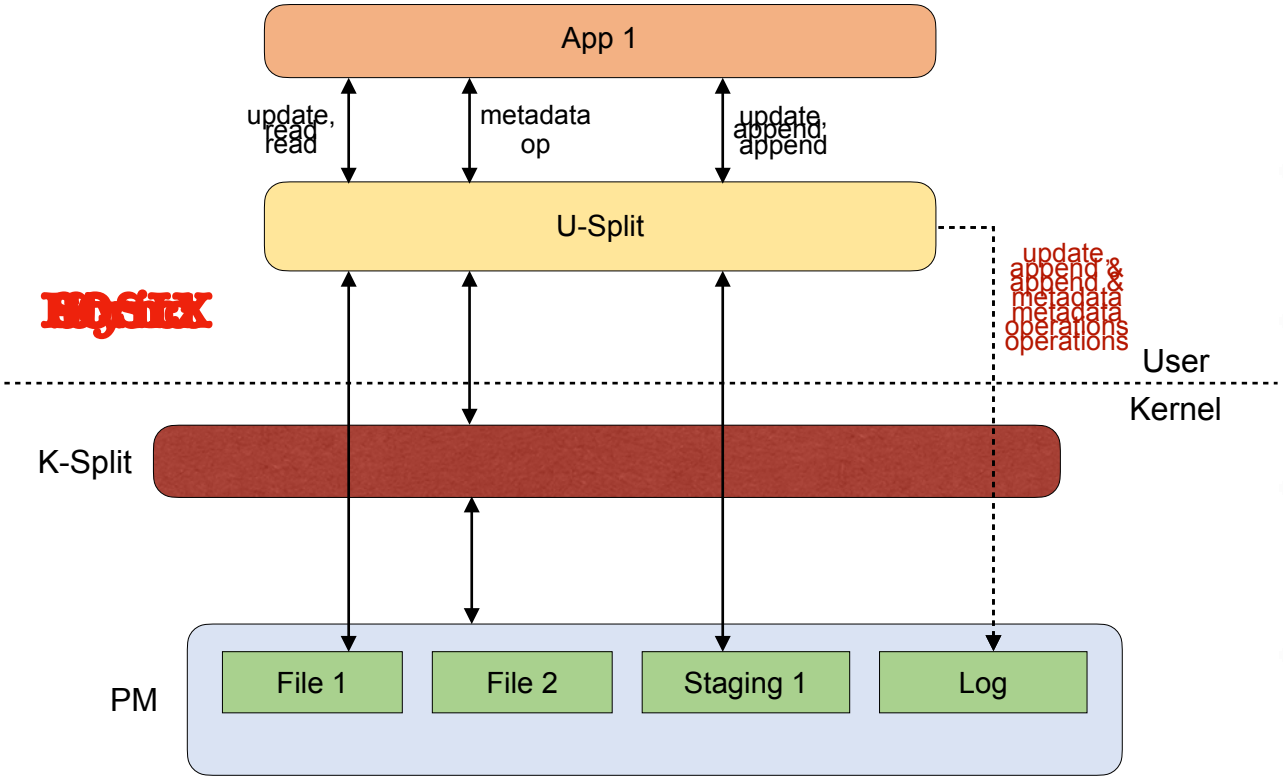
U-Split maintains a collection of mappings per file

Mappings are logically moved from staging file to target files without unmapping

Outline

- Target Usage Scenario
- High-level design
- Handling data operations
  - Handling file reads & updates
  - Handling file appends
- Consistency Guarantees
- Evaluation

SplitFS modes



- Atomic data
- Sync data
- Atomic metadata
- Sync metadata

Optimized Logging

SplitFS employs a per-application log in sync and strict mode, which logs every logical operation

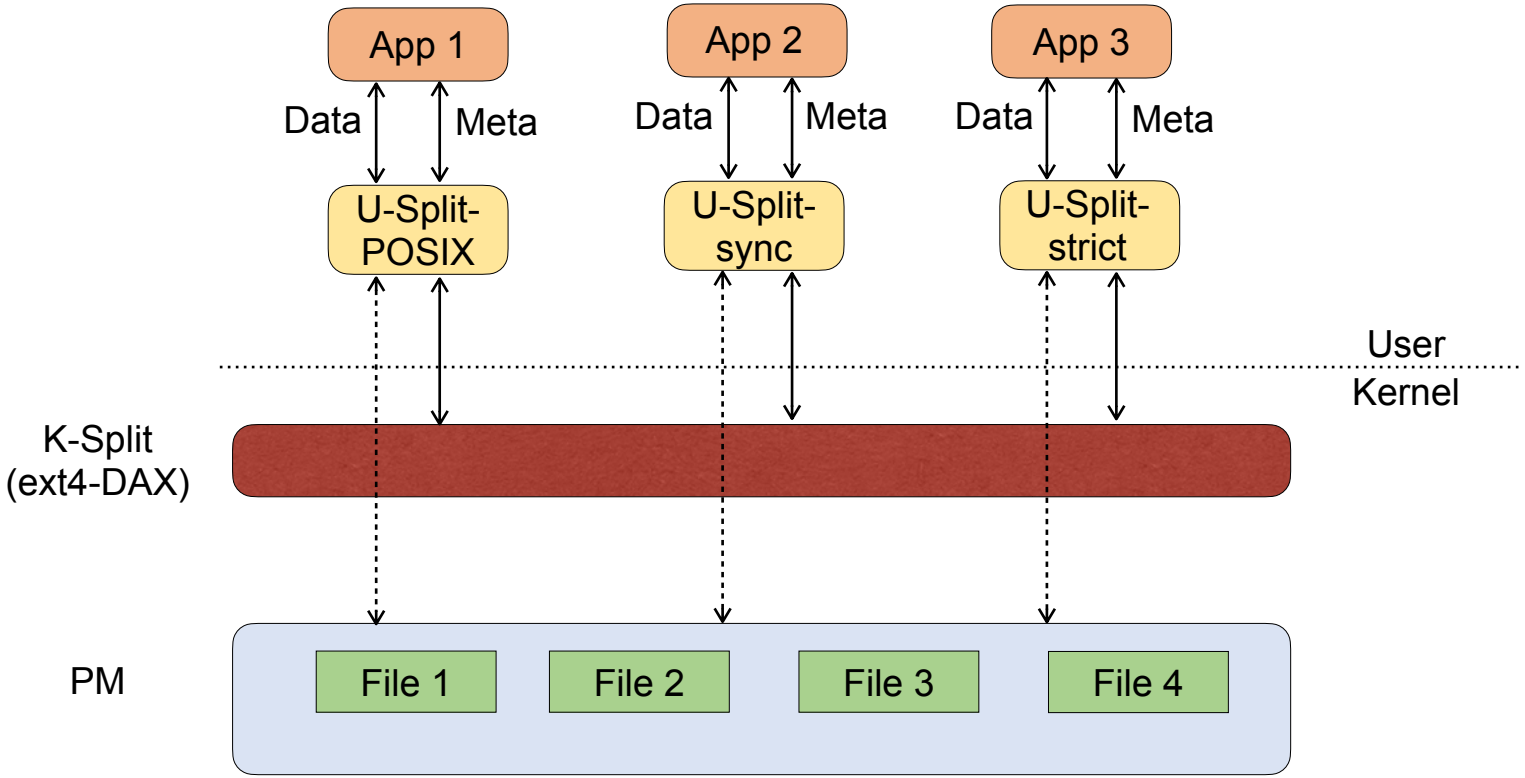
In the common case, each log entry fits in one cache line (64 bytes)

Each log entry is synchronously written to PM using DAX-mmap

Consistency Guarantees

Mode	Metadata Atomicity	Synchronous Operations	Data Atomicity	File System
POSIX	✓	✗	✗	ext4-DAX, SplitFS-POSIX
Sync	✓	✓	✗	PMFS, SplitFS-Sync
Strict	✓	✓	✓	NOVA, Strata, SplitFS-Strict

Flexible SplitFS



Visibility

When are updates from one application visible to another?

- All metadata operations are immediately visible to all other processes
- Writes are visible to all other processes on subsequent fsync()
- Memory mapped files have the same visibility guarantees as that of ext4-DAX

SplitFS Techniques

Technique	Benefit
SplitFS Architecture	Low-overhead data operations, Correct metadata operations
Staging + Relink	Optimized appends, No data copy
Optimized Logging + out-of-place writes	Stronger guarantees

Implementation

9K LOC in user-space
500 LOC in kernel — added a new system call

Supports 35 common file-system related glibc calls:
(open, close, read, write, etc)

Supports multithread applications using fine-grained reader-writer locks

Supports fork(), execve() for multi-process applications such as git, tar

Outline

- Target Usage Scenario
- High-level design
- Handling data operations
  - Handling file reads & updates
  - Handling file appends
- Consistency Guarantees
- Evaluation

Evaluation

Setup:
1-socket 48-core machine with 32 MB LLC
768 GB Intel Optane DC PMM, 192 GB DRAM

File systems compared:
ext4-DAX, PMFS, **NOVA**, Strata

Storage Hierarchy

Does SplitFS reduce software overhead compared to other file systems?

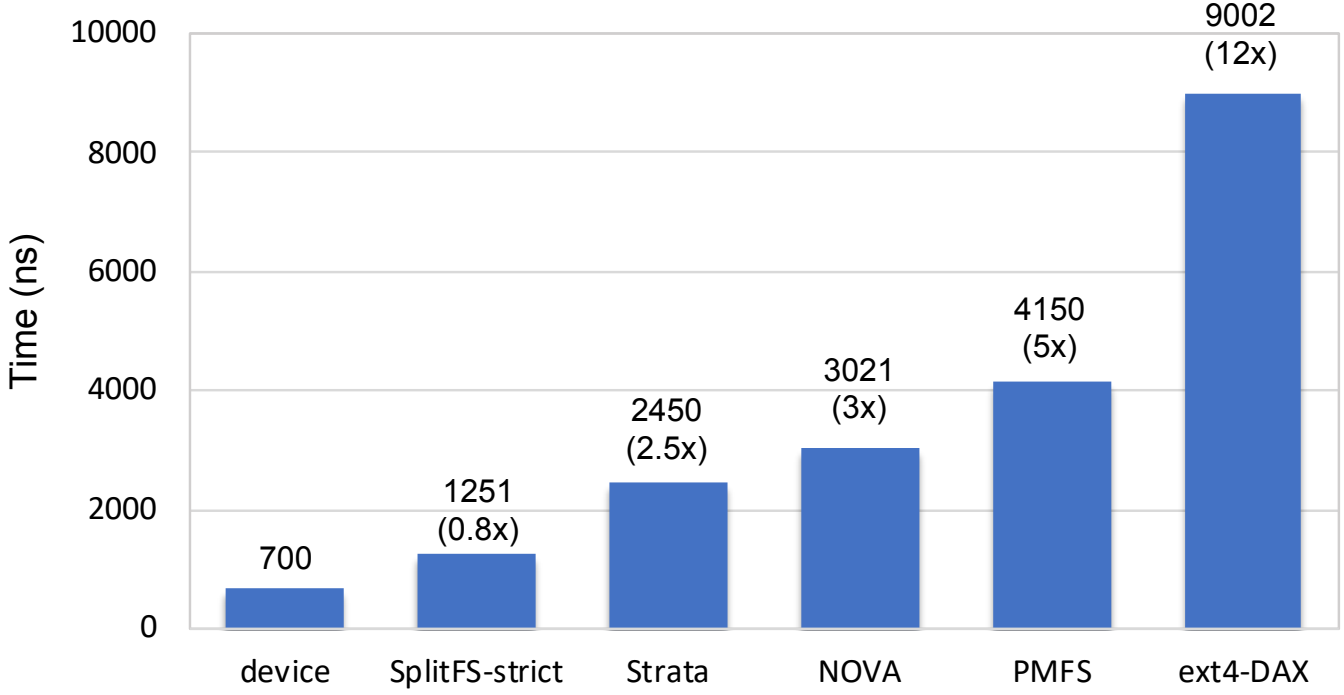
How does SplitFS perform on data intensive workloads?

How does SplitFS perform on metadata intensive workloads?

< 15% overhead

Storage Hierarchy

- Append 4KB data to a file
- Time taken to copy user data to PM: ~700 ns



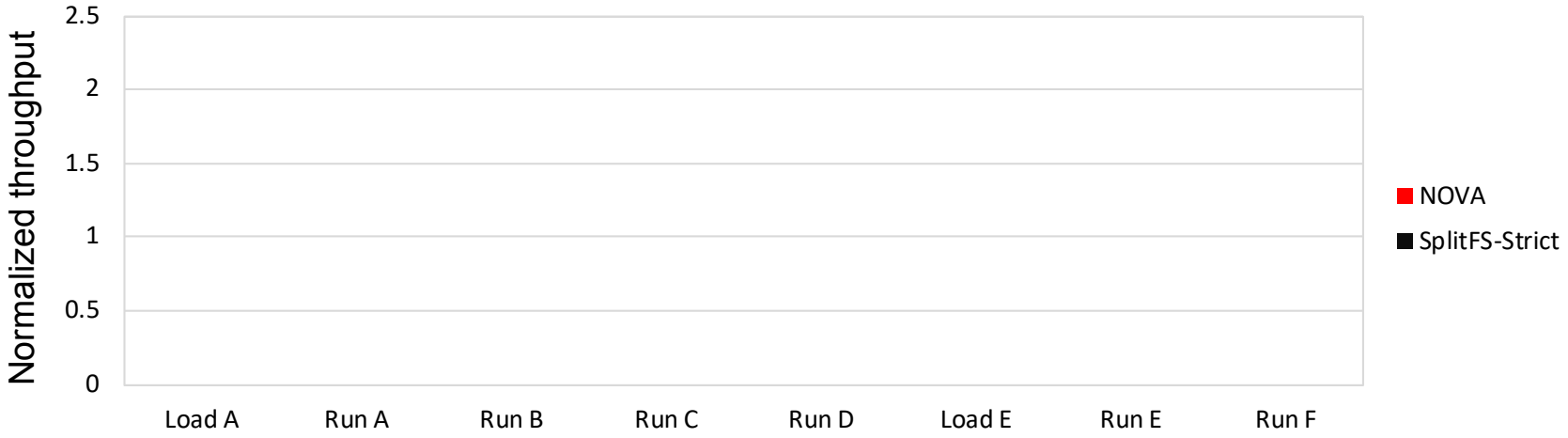
Workloads

- Microbenchmarks
  - Seq reads
  - Seq writes
  - Appends
  - Rand reads
  - Rand writes
- Data intensive
  - YCSB on LevelDB
  - Redis
  - TPCC on SQLite
- Metadata intensive
  - Untar
  - Git
  - Rsync

YCSB on LevelDB

Yahoo! Cloud Serving Benchmark - Industry standard macro-benchmark

Insert 5M keys. Run 5M operations. Key size = 16 bytes. Value size = 1K

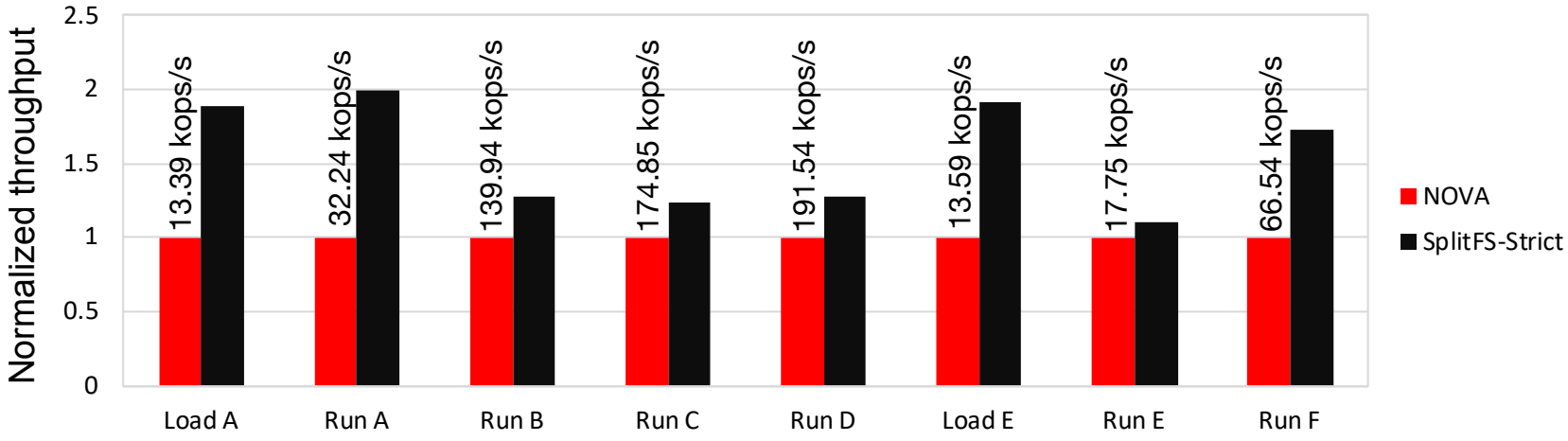


Load A - 100% writes
Run A - 50% reads, 50% writes
Run B - 95% reads, 5% writes
Run C - 100% reads
Run D - 95% reads (latest), 5% writes
Load E - 100% writes
Run E - 95% range queries, 5% writes
Run F - 50% reads, 50% read-modify-writes

YCSB on LevelDB

Yahoo! Cloud Serving Benchmark - Industry standard macro-benchmark

Insert 5M keys. Run 5M operations. Key size = 16 bytes. Value size = 1K



Load A - 100% writes

Run A - 50% reads, 50% writes

Run B - 95% reads, 5% writes

Run C - 100% reads

Run D - 95% reads (latest), 5% writes

Load E - 100% writes

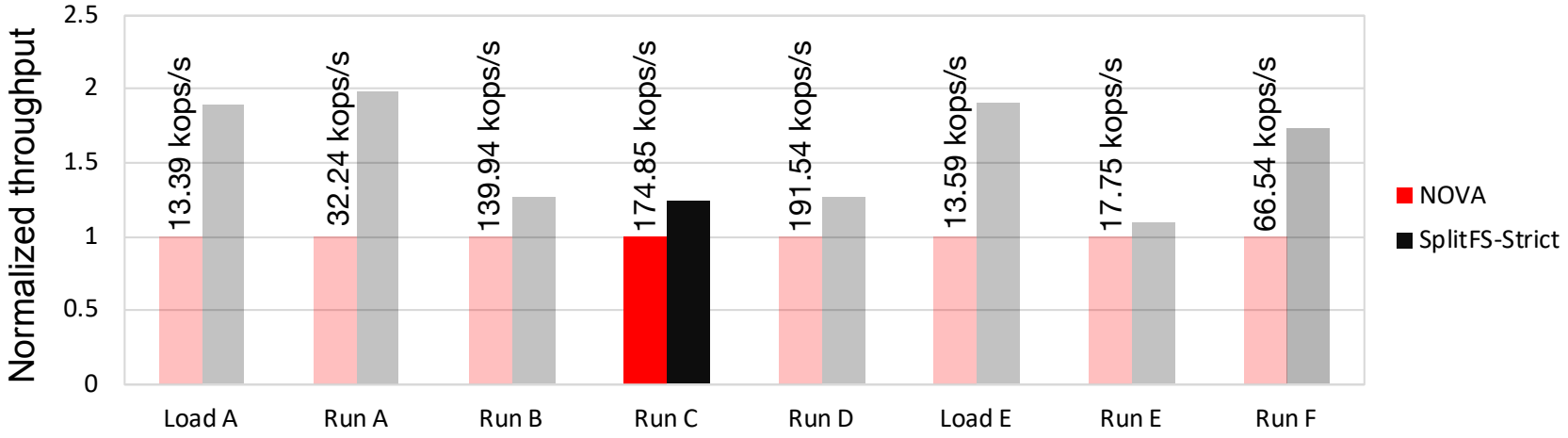
Run E - 95% range queries, 5% writes

Run F - 50% reads, 50% read-modify-writes

YCSB on LevelDB

Yahoo! Cloud Serving Benchmark - Industry standard macro-benchmark

Insert 5M keys. Run 5M operations. Key size = 16 bytes. Value size = 1K

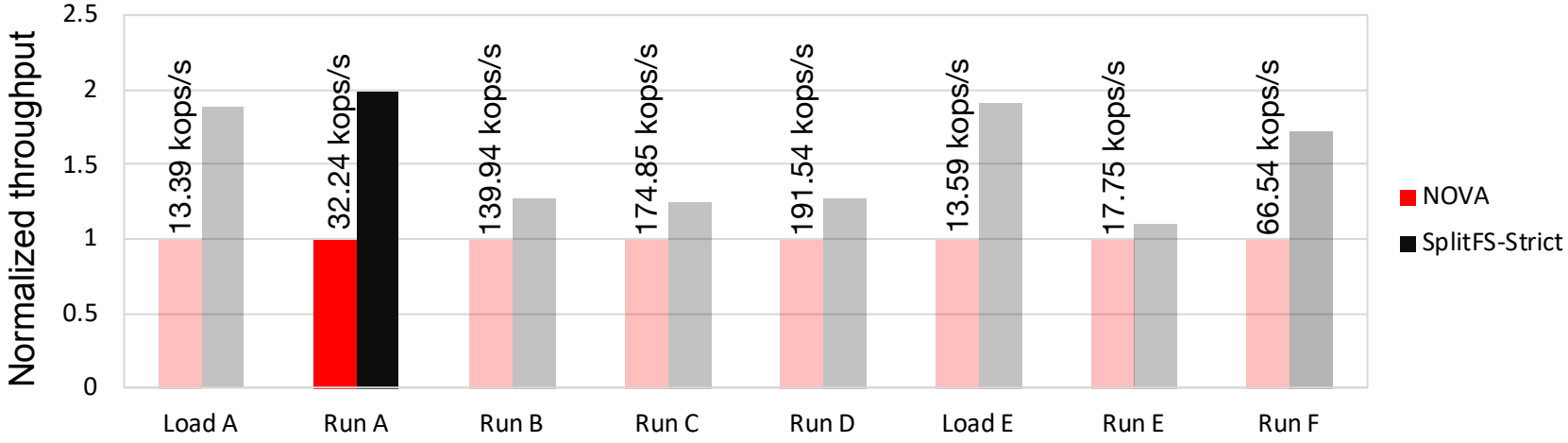


Read-heavy workloads optimized because of converting reads to loads

YCSB on LevelDB

Yahoo! Cloud Serving Benchmark - Industry standard macro-benchmark

Insert 5M keys. Run 5M operations. Key size = 16 bytes. Value size = 1K



Write-heavy workloads optimized because of staging and relink

Limitations

File accessed and modified times are not reflected immediately

SplitFS incurs overheads of ext4-DAX on all metadata operations.

SplitFS does not optimize for applications accessing memory mapped files

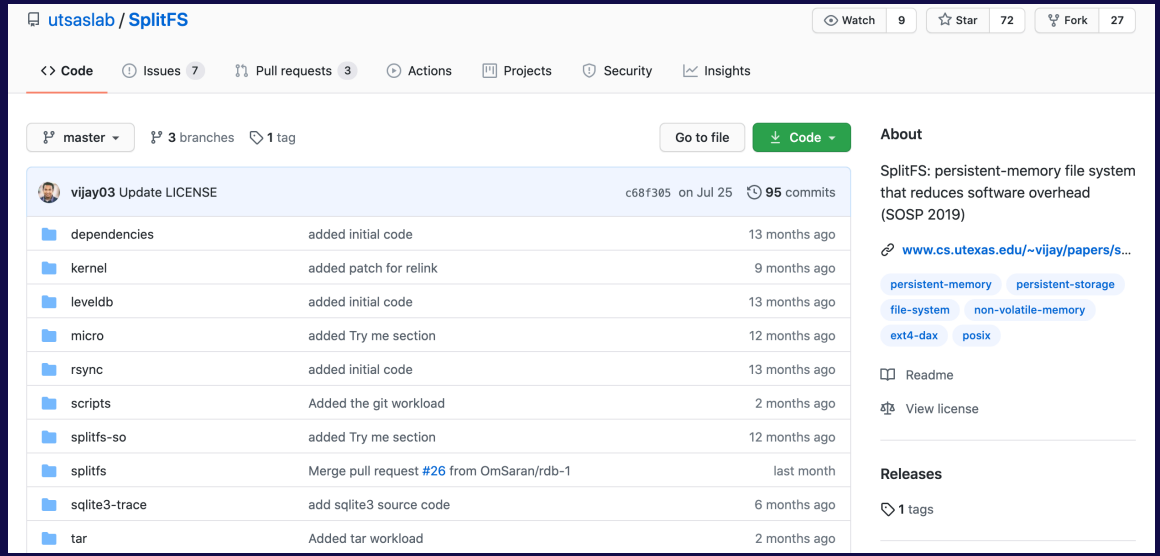
SplitFS

New architecture for building PM file systems that…

reduces software overhead,

provides strong guarantees,

and leverages the widely-used ext4-DAX



https://github.com/utsaslab/splitfs