File Systems

Jiacheng Wu and Matthew Arnold

Motivation–Technological Ecosystem

- Technological performance improvements circa 1990
  - Processor speed increasing exponentially
  - Main memory speed and size increasing exponentially
  - Disk speed not increasing as fast
- Caching
  - Absorbs read requests, expect writes to dominate disk traffic
  - Can cache large portions of data and write it all at once to optimize transfer bandwidth
- Typical use case
  - Lots of small file accesses

Motivation–Unix FFS

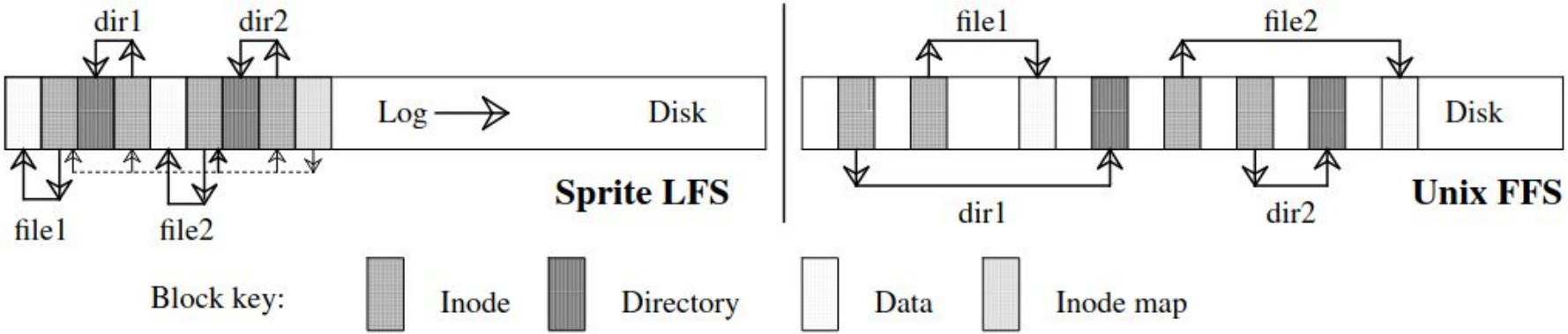
- Disk bandwidth underutilized on Unix FFS
  - Particularly with small files
  - Because of lots of disk I/O operations/seeks
- Synchronicity of disk operations
  - FS metadata written synchronously
  - Couples application performance to disk
  - With caching disk operations should be made asynchronous

Goals

- Improve write performance of disk
  - Particularly for small files
  - Cache writes and send them all at once to disk in single operation
  - Maximize disk utilization for actually writing
- Write to cache to may be synchronous, but write to disk asynchronous
  - Application performance no longer linked to slow disk operations

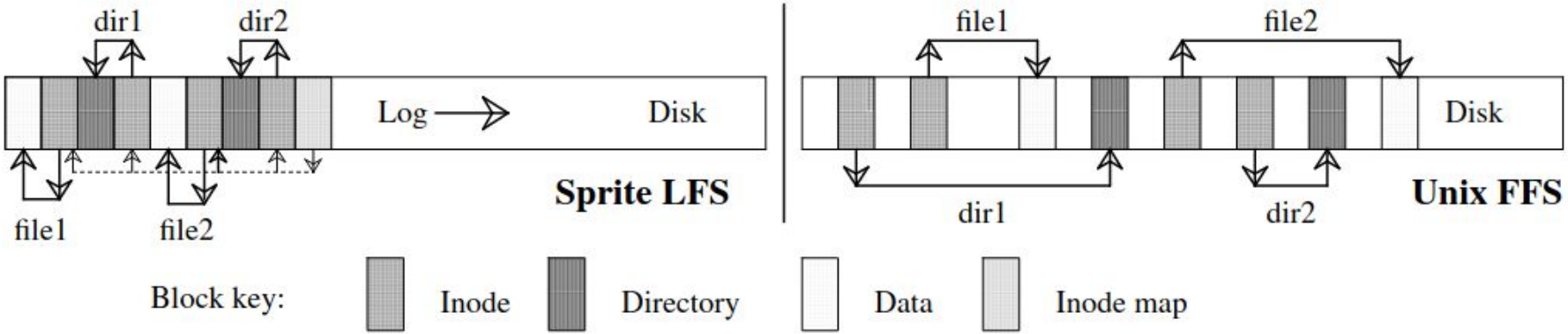
LFS–General Idea

- All data stored directly in log
  - Files temporally located vs Unix FFS, where files logically located
- Keep appending all data and organization mechanisms in log
  - Indirect blocks, inodes, etc.
  - Block “segments” memory unit of operation
- Clean segments when space more space needed



Mechanisms–Locating Files

- Use same mechanisms as Unix FFS
  - inode map→inode →(optional) indirect block→file segment(s)
  - Inode map stored in fixed memory location–checkpoint region
- More compact structure than Unix FFS
- Read operation similar number of disk operations compared to Unix FFS

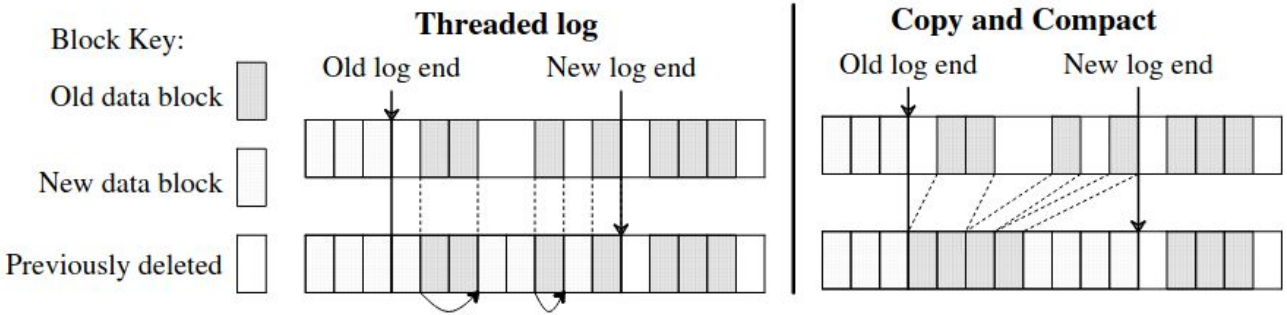


Mechanisms–Disk Space Management (1)

- Data stored in **segments**
  - Unit that disk management techniques operate upon/reason about
- Segment size–chosen s.t. seek operation much longer than transfer time for reading/writing a segment
  - Attempts to maximize disk usage

Mechanisms–Disk Space Management (2)

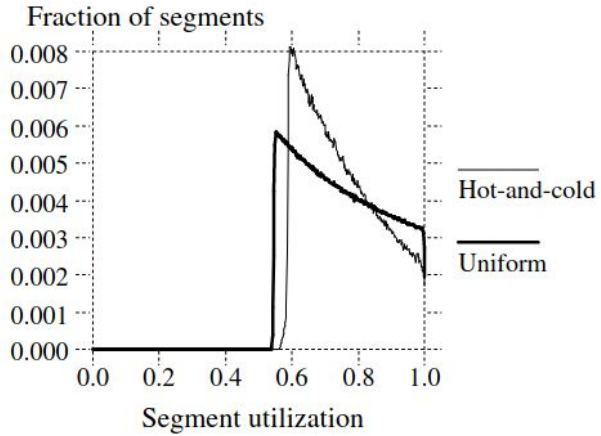
- Log starts with large and completely free extent to write to
- What happens when you get to the end of the memory you can log to?
  - Goal: reclaim space that is no longer being used
  - Possibilities: Threading or Copying, or both
- Threading impossible because of extreme fragmentation in segments
- LFS uses combination of threading and copying
  - Thread at segment level, coalesce fragmented data to smaller number of blocks



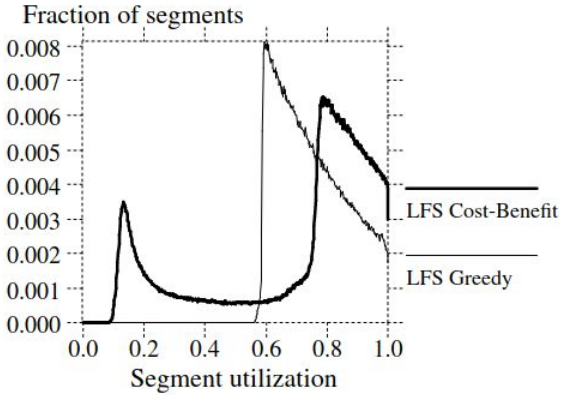
Mechanisms–Segment Cleaning (1)

- Liveness–to clean a block you must know if data inside is still referenced
- Cleaning policies influenced by the notion of “write cost”
  - How busy disk is per byte of written data
  - Write cost **inversely proportional** to liveness of segment being cleaned aka utilization
  - Intuitively–if you have many segments that are mostly live, a lot of work is required to make room for new data
- Policies described work by cleaning segments that dropped below a particular utilization
  - We want utilization to be low to reduce write cost, but also don’t want to spend too much time cleaning
- So what cleaning policy should be chosen?
  - Greedy–clean least utilized segments
  - Cost-Benefit–clean hot segments at lower utilization than cold segments b/c free space in cold segments more valuable than in hot segments

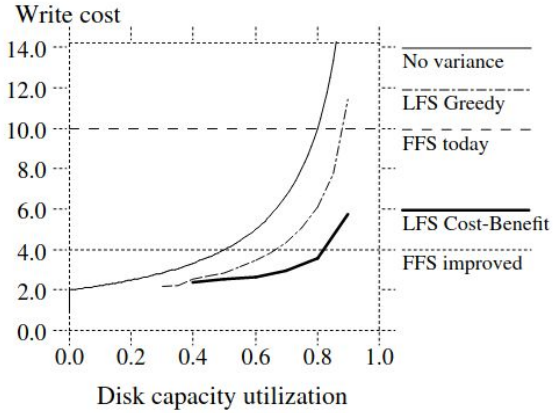
Mechanisms–Segment Cleaning (2)



Original greedy policy



Cost-Benefit approach–can clean segments early to maintain desired bi-modal segment distribution



Cost-Benefit performs better than Greedy

Mechanisms–Logging for Crash Recovery

- General idea–create checkpoint, then roll forward to last checkpoint
- Similar to database system write-ahead logging, but use of log is different
  - Log viewed as “truth” about state of disk, but DBMS doesn’t use log as final location of data, so no cleaning mechanism needed
  - Sprite LFS reclaims space during cleaning after logging changes written to final location, DBMS doesn’t care about this

Claims

- Permits 65-75% of bandwidth for writing new data (rest time spent cleaning), vs Unix systems, which only utilize 5-10% of raw bandwidth for writing (rest spent seeking)
- More efficient than SunOS–10x faster for create, delete
  - Reading pretty much the same, aside from re-reading after random write due to additional seeking to find data
- Predicted speedup because CPU saturated but disk usage was low

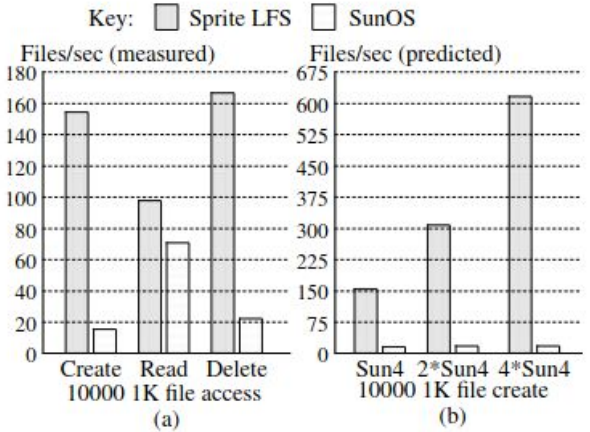


Figure 8 — Small-file performance under Sprite LFS and SunOS.

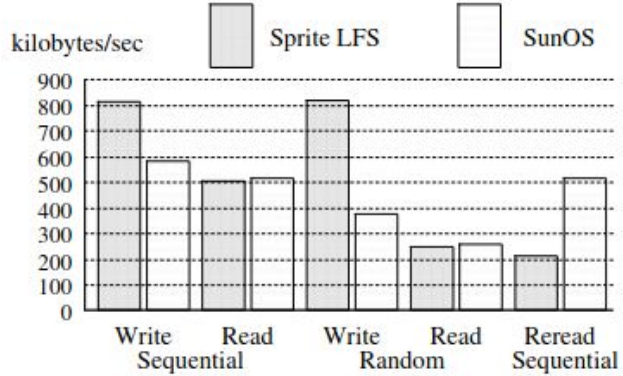


Figure 9 — Large-file performance under Sprite LFS and SunOS.

Do you agree with the claims?

Disadvantages

Why don’t we exclusively use log-structured file systems?

Assume that caches are ever increasing/infinite

- On media where seeks take a long time, reading a fractured file would take a long time due to a significant amount of seeking to read something
- Doesn’t work well when disk full
- Doesn’t work well when writes are random b/c forces dead space and frequent cleaning
- Why not used on disks today?
  - Works well when segment cleaning done in background when file system not busy doing other stuff but…
  - When considering real world utilization, performance degrades significantly (40%) compared to results found in paper
  - Segment cleaning problem was never solved to the point where LFS was better than in-place file systems

LFS with SSDs

- Assumption that reads cheap and writes expensive fits into the model of flash
  - reads can be granular, but writes done in large contiguous blocks (clear block, then set individual bits in the block)
- It is assumed that SSDs typically have some sort of LFS-like internal to the system, externally a non-LFS used on top/outside of the SSD
- SSD work is private/commercialized and you can’t actually see the firmware manufacturers deploy on SSDs, so this is mostly guesswork/external observation
- They could be implementing some other data management system that behaves similar to LFS

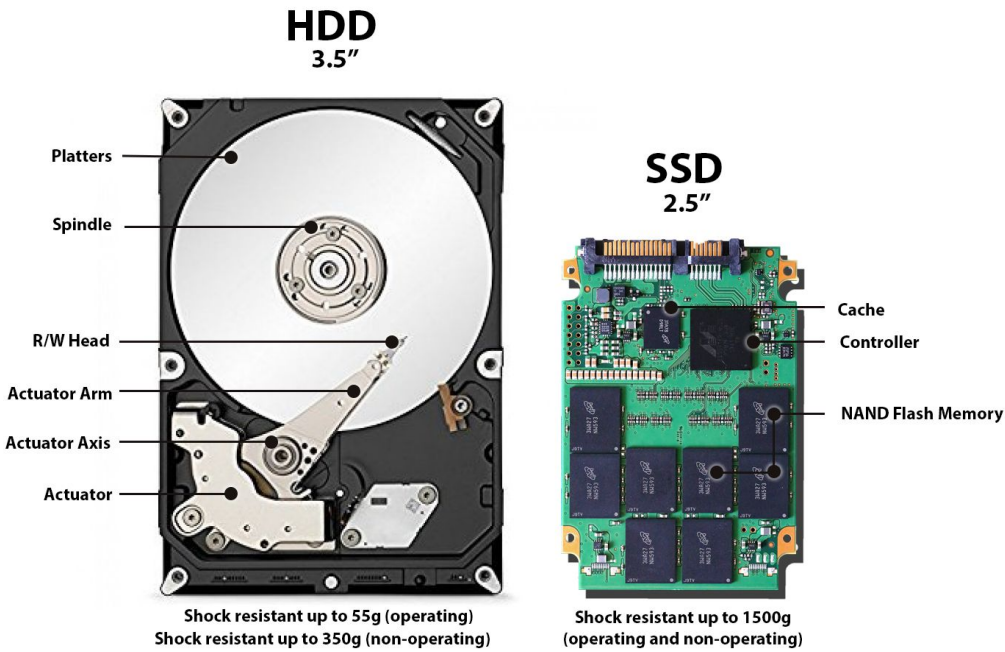
Questions related to the paper

- Suppose log FSes weren't designed in the 1990s…would motivation remain the same for someone to design them today?
- Are techniques from Sprite LFS used today?
  - Flash memory systems use techniques (writes expensive/done in big blocks), reads cheap
- How important is reducing fragmentation in the LFS?
  - I.e. since copy and compact is costly can we reduce how much of it we do to make LFS more efficient?
  - Is this dependent on the type of nonvolatile memory system being used

Modern Day File Systems

Characteristic of SSD/Flash

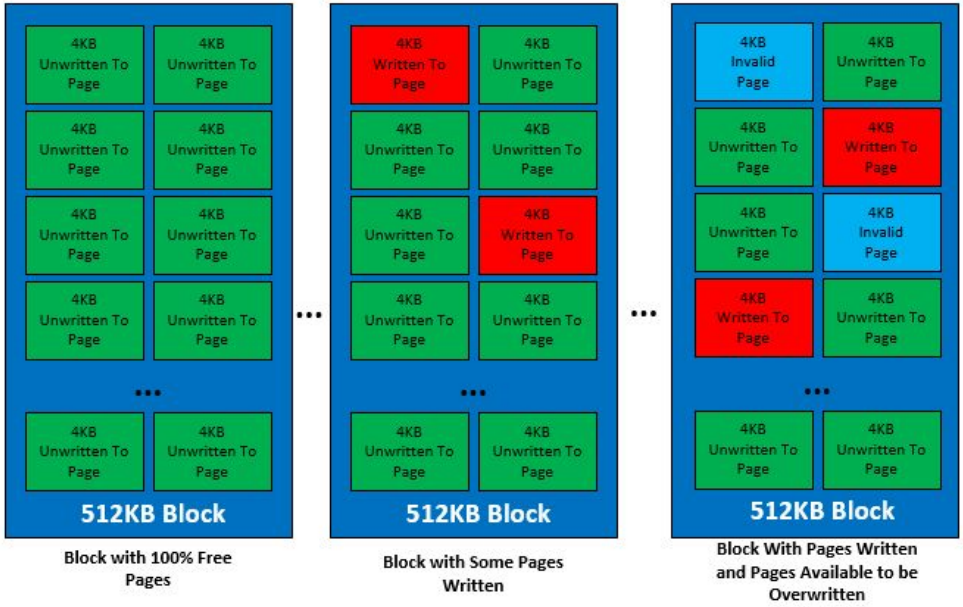
- SSD, most of SSD use Flash
- SSD performs random R/W well
  - It use electronic no mechanical
  - An order of magnitude than HDD
- SSD supports simultaneous R/W
  - SATA controllers operate sequentially
  - NVMe controller can execute parallel commands using multiple PCIe interface lanes.



Problems on SSD/Flash

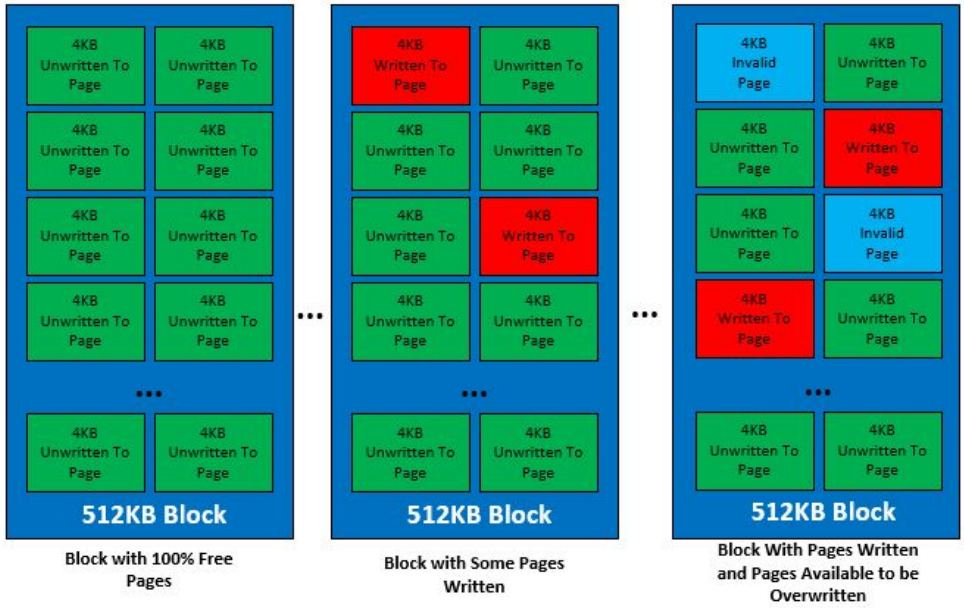
- SSDs - Blocks >> Pages >> Cells
  - Data is read and written at the page level
  - But Data Erasing only be at the block level
- **Will Fragmentation influence**

  the performance of RW on SSD?



Write Amplification on SSD/Flash

- SSDs - Blocks >> Pages >> Cells
  - Data is read and written at the page level
  - But Data Erasing only be at the block level
- Append on SSD is just page level
- Update on SSD is the block level
  - Once you need update a page/cell
  - First copy whole block to buffer
  - Erase the whole block
  - Move backup with updated to block
- Cell has limited times to write/erase
  - Write Amplification reduce the life of SSD



F2FS - Flash Storage [Lee et al. FAST15]

- Based on LFS
  - Considering If the characteristic of Flash
  - Radom Write -> ↑ Fragmentation, ↓ Performance
- Flash-Friendly Layout
  - Segment (cleaning units), Section, Zone
- Multi-Head Logging
  - Leverage Flash Parallelism
  - Multiple active logging segments
- Adaptive Logging
  - Write new data in free space of dirty seg
  - Thread Logging at high utilization

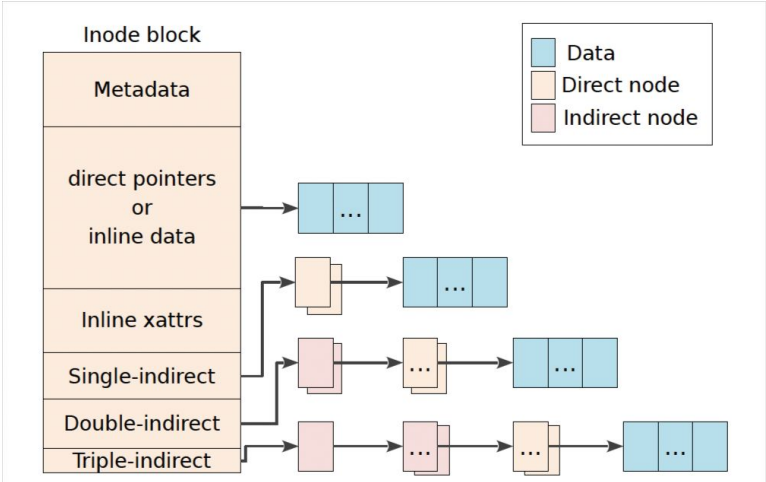
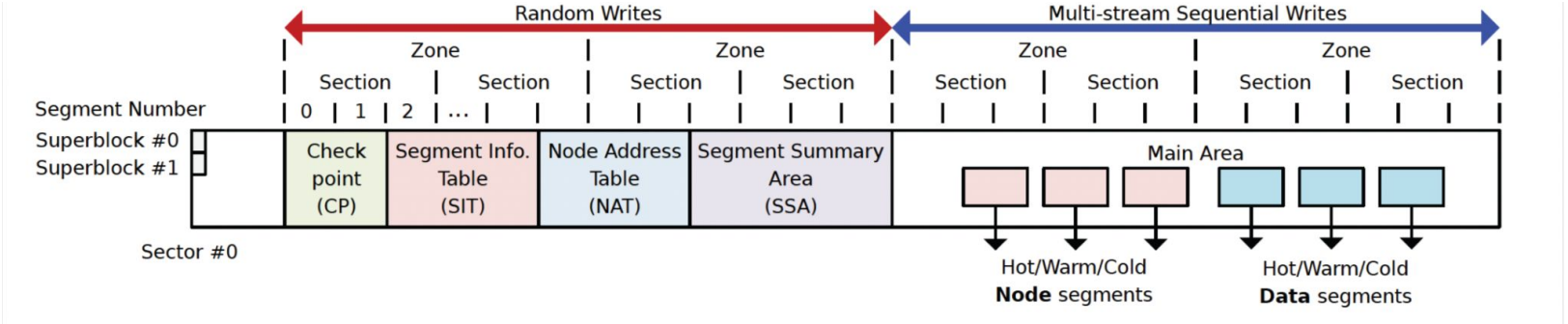
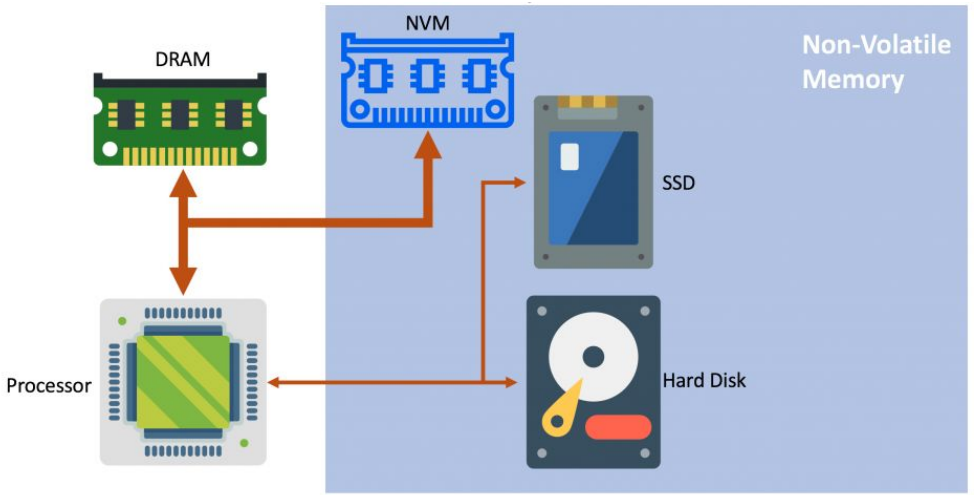


Figure 2: File structure of F2FS.



Non-Volatile Memory

- Just behave as memory
  - Byte-addressable
  - High performance for R/W
  - Processor directly access by PCIE
  - But is non volatile
- To enforce the data stored on NVM
  - Write to the NVM address range
  - Flush the Cache (CLWB)
  - Add Memory Barrier (SFENCE)
- Intel Octane NVM
  - Phase Change Memory
  - No Write Amplification!!



Discussion

- What problems might be traditional file system on NVM
  - Just like all data fits into the whole memory?
  - But some part of memory is non-volatile
- Is necessary to Log Data in NVM File System?

NOVA - Hybrid V/NV Memory [Xu et al. FAST16]

- Challenges
  - Managing, Accessing NVM
  - Maintaining Consistency
    - CPU reorder stores/Memory Cache
    - Maintain are costly
- Adapt LSFS to fast random write
  - Index in DRAM, Logs in NVM
  - Separate Log for each node (Concurrency)
    - Store Log as link list
    - Lightweight journing for atomic
  - Store file data outside log
    - Recovery is fast
    - GC is fast
    - Do not log data
- Keep Complex DS in DRAM
  - Provide (meta)data atomicity

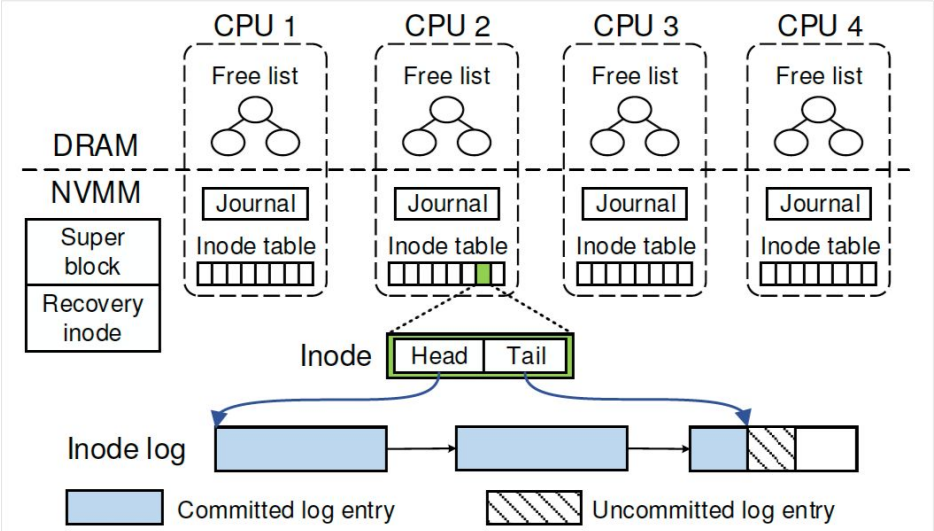
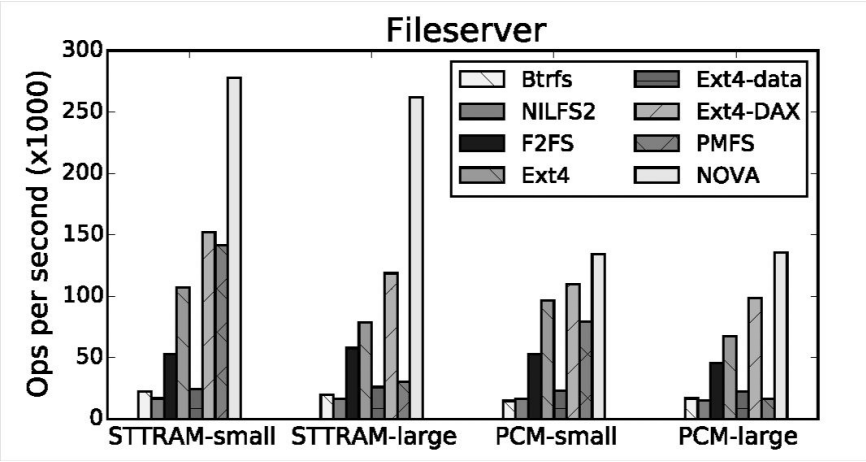
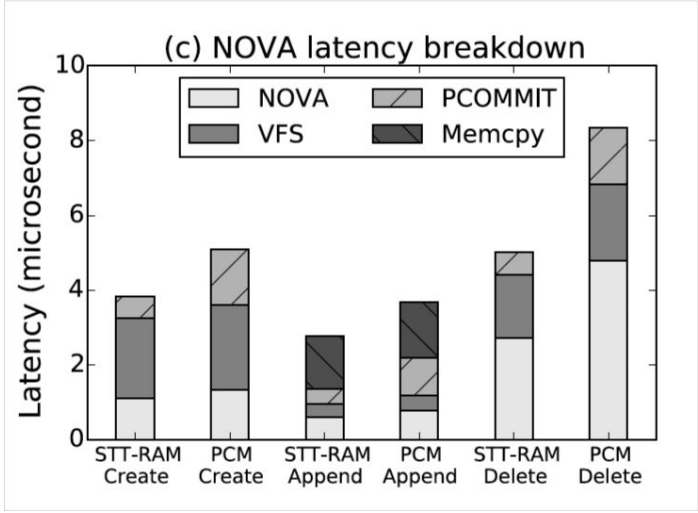


Figure 1: NOVA data structure layout. NOVA has per-CPU free lists, journals and inode tables to ensure good scalability. Each inode has a separate log consisting of a singly linked list of 4 KB log pages; the tail pointer in the inode points to the latest committed entry in the log.

NOVA - Hybrid V/NV Memory [Xu et al. FAST16]





SplitFS - Persistent Memory [Kadekoli et al. SOSP19]

- Problem: Software Overhead
  - Allocation, Logging and Updating Metadata
  - The Overhead is >= 80%
- Main Idea: Split
  - Userspace FS -> data ops
  - Kernel PM FS(ext4 DAX) -> metadata ops
- UserSpace for Read/Overwrite
  - Intercept POSIX calls
  - Map memory to underlie files
  - FS R/W -> Processors Loads/Stores
- Kernel for Meta/Append/Atomic Data Ops
  - Relink primitive optimizations

https://github.com/utsaslab/splitfs

Technique	*Benefit*
Split architecture	Low-overhead data operations, correct metadata operations
Collection of memory-mmaps	Low-overhead data operations in the presence of updates and appends
Relink + Staging	Optimized appends, atomic data operations, low write amplification
Optimized operation logging	Atomic operations, low write amplification

Table 4. Techniques. The table lists each main technique used in SplitFS along with the benefit it provides. The techniques work together to enable SplitFS to provide strong guarantees at low software overhead.

KucoFS - Kernel Userspace Collaboration

[Chen et al. FAST21]

- Main Goal: Scalability
  - Kernel Ops is not scalable
  - VFS layer, Centralized Components
- Offload time-consuming Tasks
  - PathName resolution / CC
  - From Kernel to Space
  - Using 3 techniques
    - Collaborative Index (Ulib/KFS)
    - Two-Level Locking
    - Version Reads

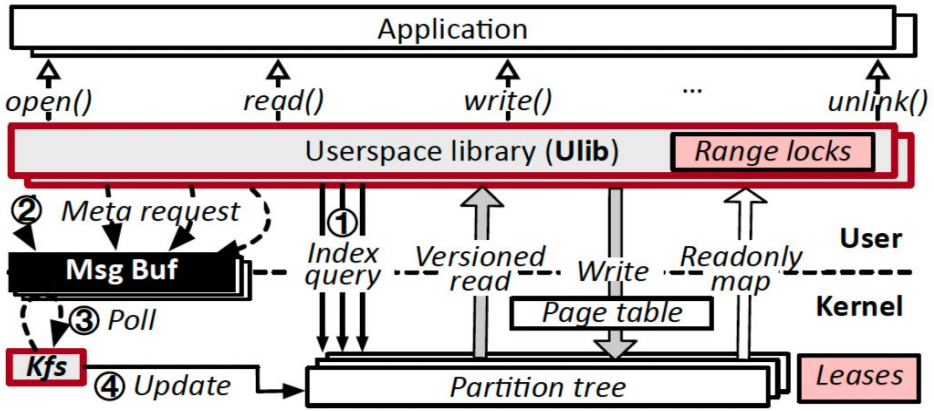


Figure 2: The KUCO architecture. metadata updates (①-④): Ulib interacts with Kfs via *collaborative indexing*; read: direct access via *versioned read*; write: direct access based on a *three-phase write protocol* and *two-level locking* for concurrency control.

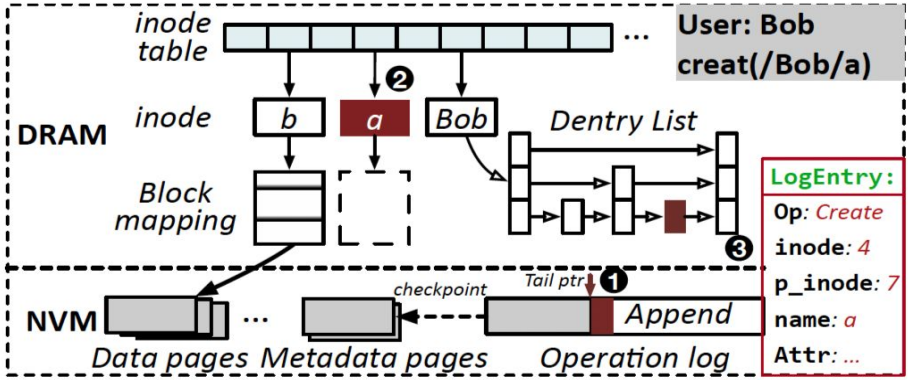


Figure 6: Data layout of a partition tree in KUCOFS. `creat` operation with three steps is also shown.

ctFS - Hardware Memory Translation [Li et al. FAST22]

- Motivation
  - Costly Block Address Lookup
  - Build/Update Complex Index
  - File Offsets -> PM address (> 50%)
- Contiguous File System
  - Files <- contiguous Virtual Memory
  - Offsets is Offsets !!!
  - Leverage just Hardware MMU
  - No Software Maintained Index
- Challenge:
  - How files are allocated
  - How resizing is managed

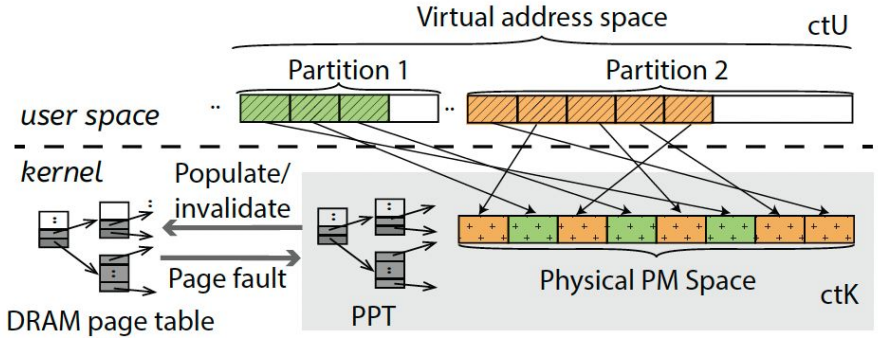


Figure 2: **Architecture of ctFS.** Each box represents a page. Two partitions are shown. The file allocated in partition 1 uses 3 pages (green), and the file in partition 2 uses 5 pages. ctK maintains virtual-to-physical page mappings in the PPT.

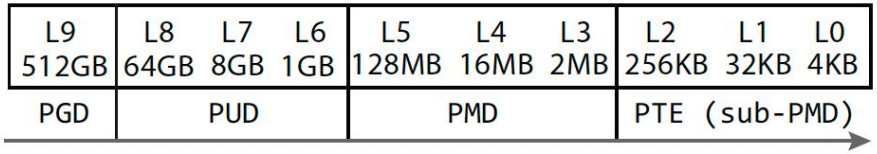
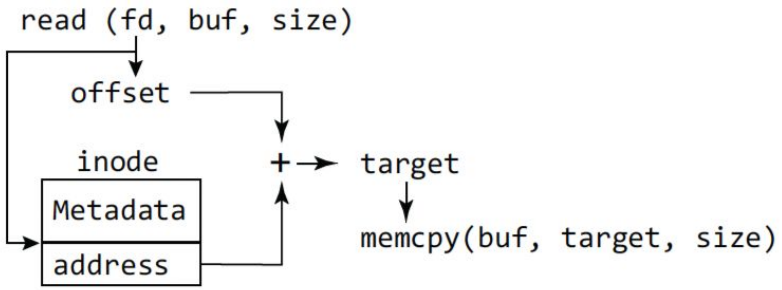


Figure 3: **Size of partitions at levels L0 to L9.** PGD, PUD, PMD, and PTE refer to the four levels of page tables in Linux (from highest to lowest). An L9 partition aligns with PGD, i.e., its starting address has zero in all of the lower level page tables (PUD, PMD, PTE); Similarly, L6-L8 partitions align with PUD, whereas L3-L5 partitions align with PMD.

ctFS - Hardware Memory Translation [Li et al. FAST22]



1

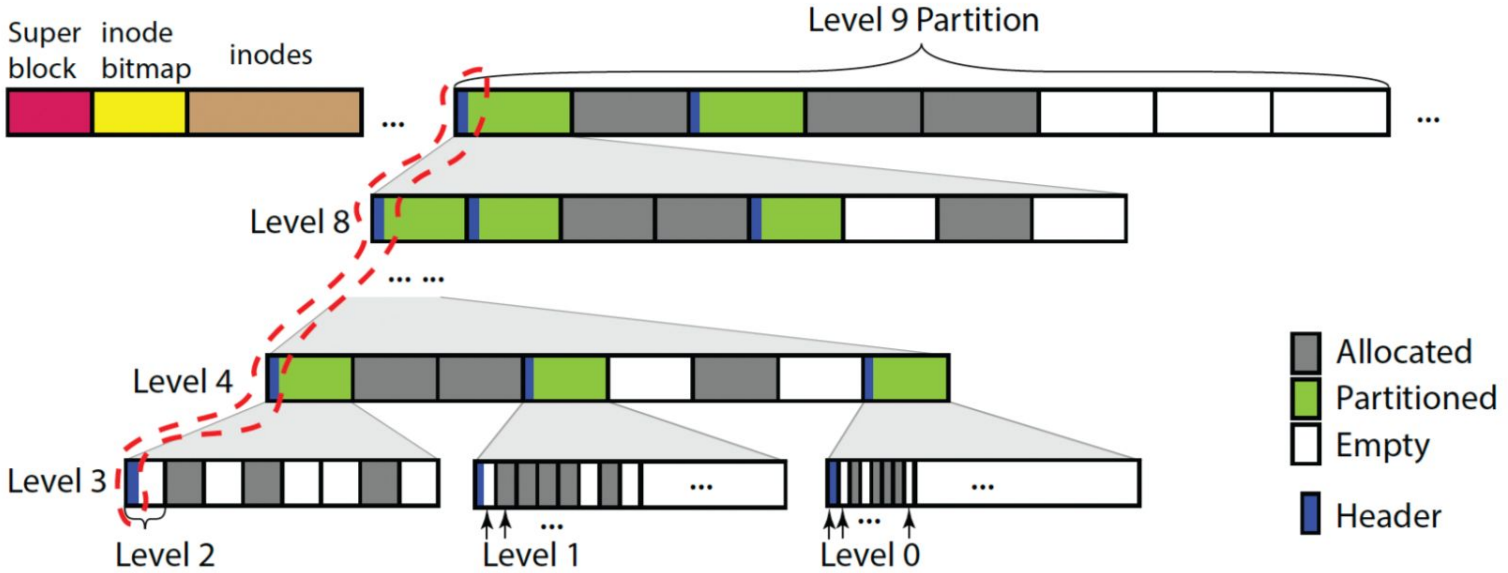


Figure 4: **Layout of ctFS in the *virtual* address space (VAS).** The space of an entire partition is reserved in VAS, whereas the physical PM space is allocated on-demand based on actual usage. Headers circled in the dashed-line reside on the same page.

Discussion

1. **SSDs**: General consensus in LFS is a good starting point
   - Writing chunks of data ideal, so don't want too much fragmentation
   - How do we address wearing issue?
2. **Tape**: locality important (logical locality, probably not temporal locality)
   - Preprocessing important and OK to take a long time doing this
3. **In-Flight Network**: (Store the data on the LINK!!)
   - ?? Consensus and Synchronization between Disk and Network IO
   - High latency to access data, somewhat nonvolatile and potentially high message loss, need robustness
4. **Disk fits in memory**: reads are "free" and writes asynchronous, which fits LFS model
   - Lots of writes in the background to minimize CPU usage

ANY QUESTIONS?