File Systems
LFS

Emmett Witchel

CS380L

LFS faux quiz (any 2, 5 min):

1. Why would anyone optimize a file system for writes?
2. Why is/isn’t an imap + “mobile” inodes better than a fixed array?
3. Why are segments better than threading or compaction?
4. What workloads will be slower for LFS than FFS?
5. Why clean hot and cold segments at different thresholds?
6. How do crash recovery techniques differ between LFS and a journaling FS?
7. Compare and contrast FFS and LFS from a mechanical sympathy perspective.
8. FreeBSD and LFS deal with multiple allocation sizes (superpages/segments). How are the problems and solutions similar and/or different?
9. Why doesn’t LFS have to completely replay the log at initialization time?
10. How does LFS handle a crash that occurs during a checkpoint. Is it always guaranteed to have a consistent checkpoint?

Crash Consistency—refresher

- Crash consistency:
  - File system is in a “consistent” state after crash
  - File system is in a “recoverable” state?
  - User data is consistent?
- Difficulty: multiple meta-data updates must appear atomic

The three **consistency commandments**

NEVER:

- … point to a structure before it has been initialized.
- … reuse a resource before nullifying all previous pointers to it.
- … reset last pointer to live resource before new pointer is set.

(Adapted from soft updates [McKusick et al.])

3.6. Dependency Tracking for new Indirect Blocks

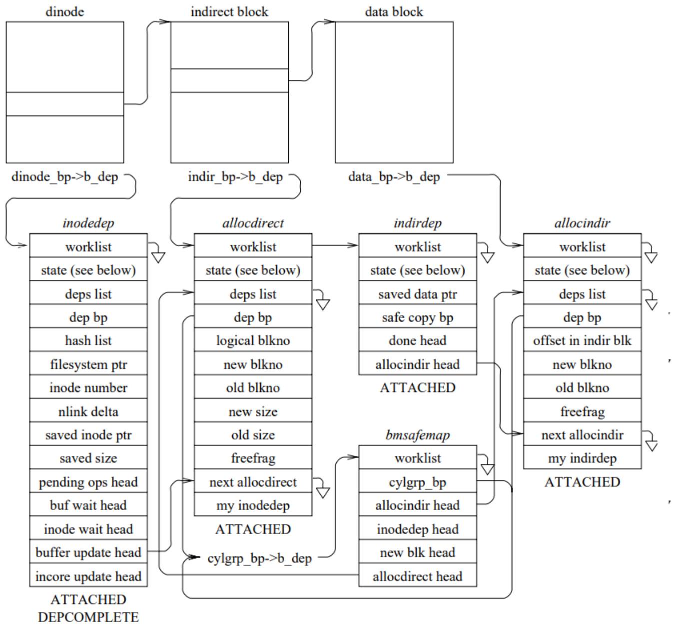


Figure 6: *Dependencies for a File Expanding into an Indirect Block*

3.7. New Directory Entry Dependency Tracking

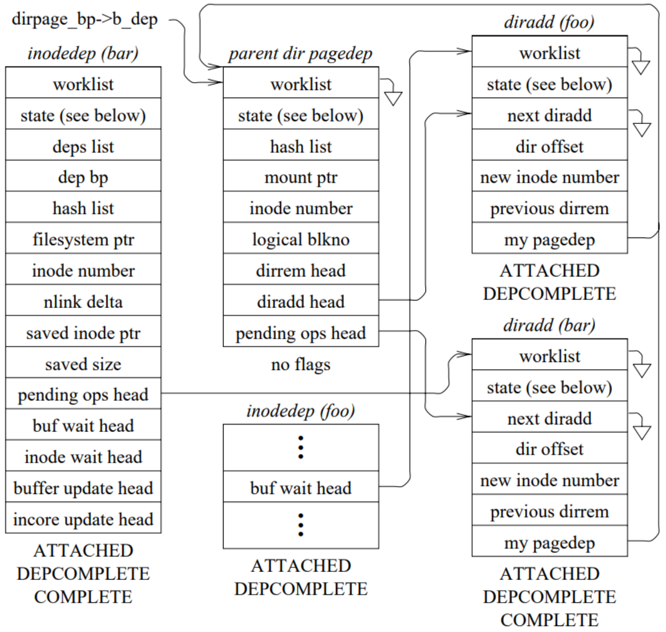


Figure 7: *Dependencies Associated with Adding New Directory Entries*

LFS: …why?

- Technology trends
  - Growing DRAM
  - RAID, network RAID, transfer bandwidth/access time relative to CPU

- Implications
  - All reads served from cache
  - Can’t we serve writes from cache?
  - Most disk traffic is writes
  - RAID5 makes small writes s*&k

LFS: Some important questions

- Why is an imap necessary for LFS? Is it clearly better?
- Why doesn’t LFS compact segments based on “age-sort” alone? What does it do instead?
- For what operations will LFS be faster/better than FFS? Vice-versa?
- How does LFS deal with the consistency challenges above? Does LFS do the kind of logging we saw in the previous slides?

Motivation: creating two files

:> echo “quack” > dir1/file1

:> echo “quack again” > dir2/file2

What are the basic file system structures that get updated?

How would FFS allocate disk space for this?

Motivation: creating two files

:> echo “quack” > dir1/file1

:> echo “quack again” > dir2/file2

What are the basic file system structures that get updated?

1. Inodes for dir1, dir2 updated to include pointers to blocks for file1, file2 dentries
2. Data blocks created for file1, file2 data
3. Inodes for file1, file2 created, point to datablocks.

How would FFS allocate disk space for this?

Motivation: creating two files

:> echo “quack” > dir1/file1

:> echo “quack again” > dir2/file2

What are the basic file system structures that get updated?

1. Inodes for dir1, dir2 updated to include pointers to blocks for file1, file2 dentries
2. Data blocks created for file1, file2 data
3. Inodes for file1, file2 created, point to datablocks.

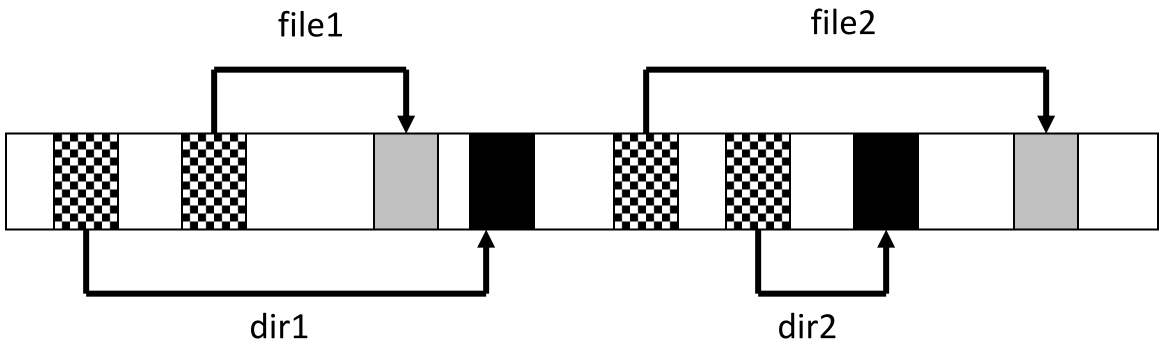
How would FFS allocate disk space for this?

Using heuristics to preserve locality (e.g. cylinder groups, etc.)

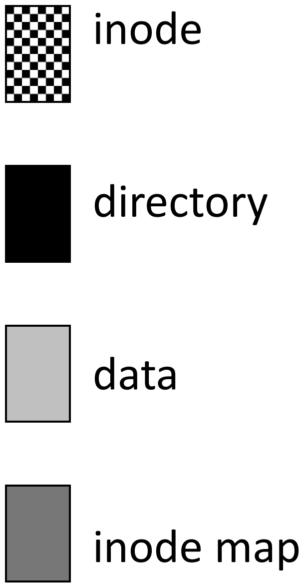
:> echo “quack” > dir1/file1

:> echo “quack again” > dir2/file2

LFS Motivation: FFS



Unix File System

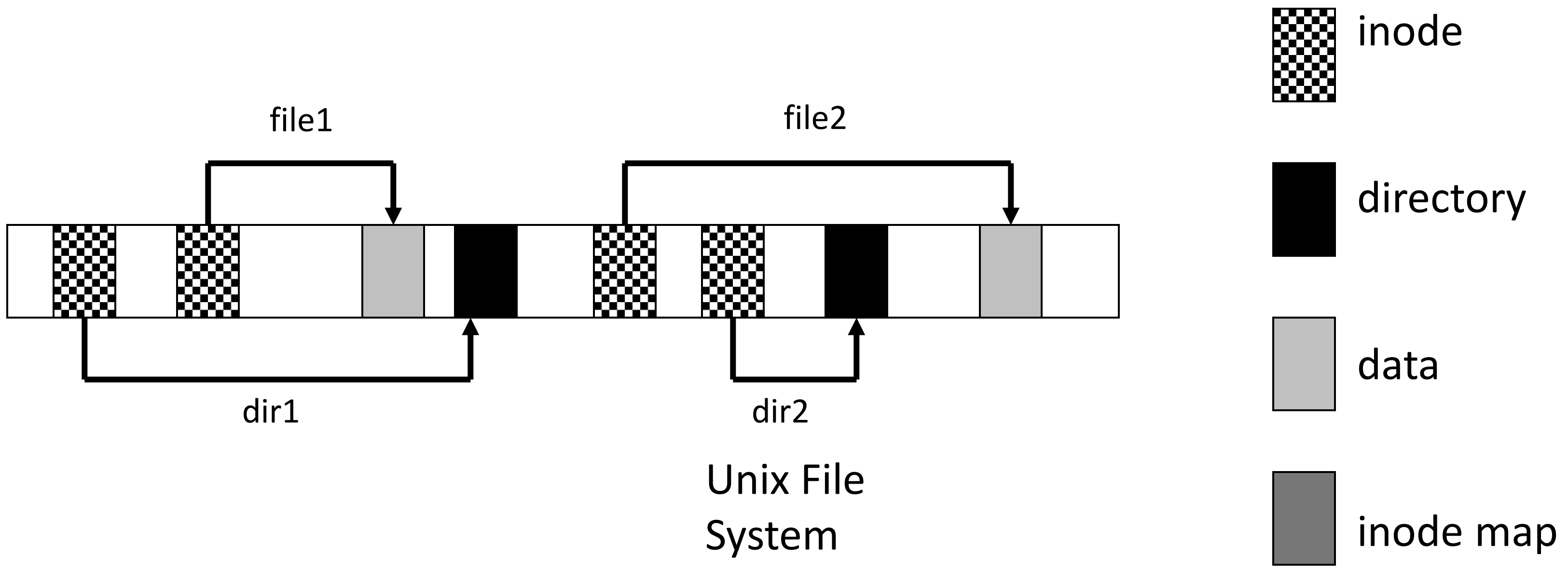


How many seeks in FFS?

:> echo “quack” > dir1/file1

:> echo “quack again” > dir2/file2

LFS Motivation: FFS

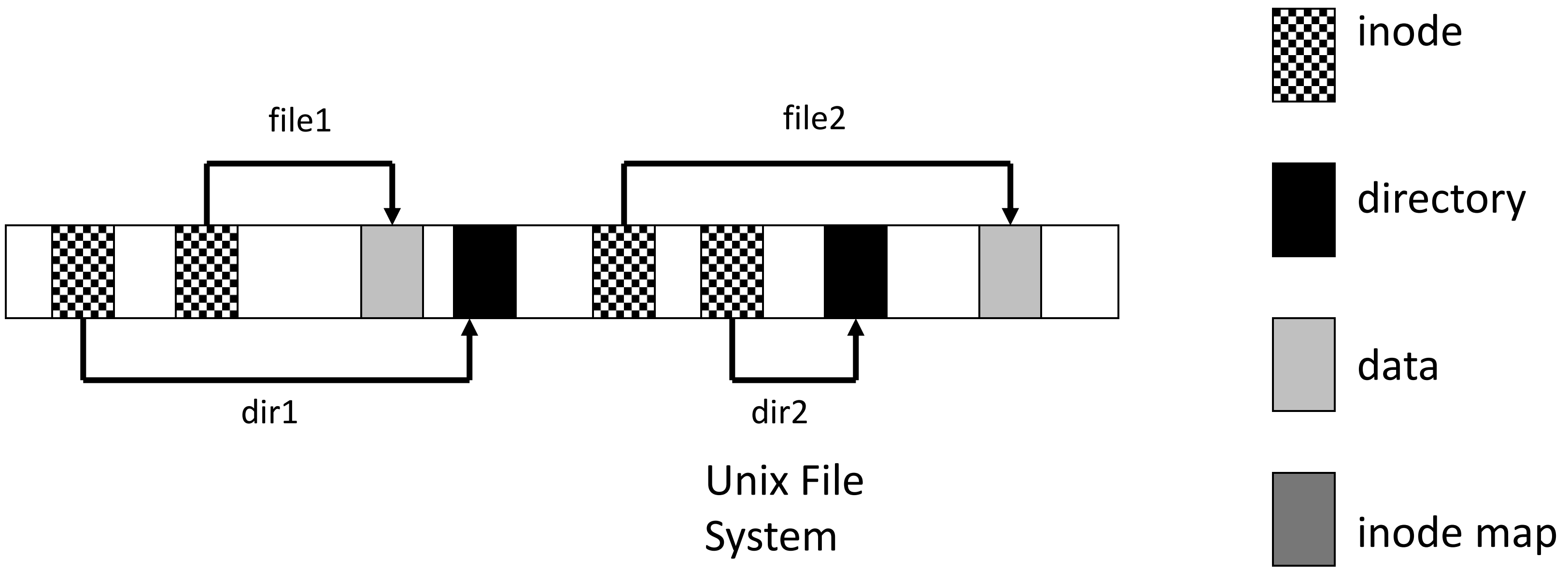


How many seeks in FFS?
(Yes, it depends)
So…worst case?

:> echo “quack” > dir1/file1

:> echo “quack again” > dir2/file2

LFS Motivation: FFS



Unix File System

How many seeks in FFS?
(Yes, it depends)
So…worst case?

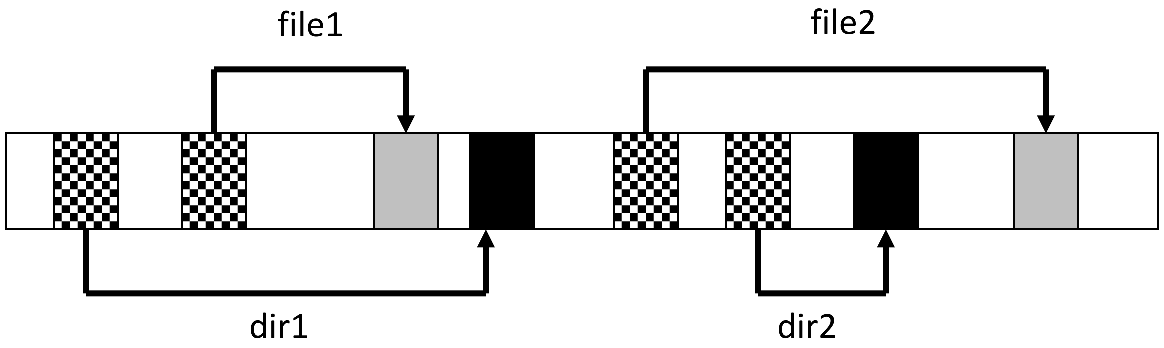
5 disk I/Os per create:

- 2 access to file attrs (inode)
  - why 2?
- Data block
- Dentry block
- Dir attrs

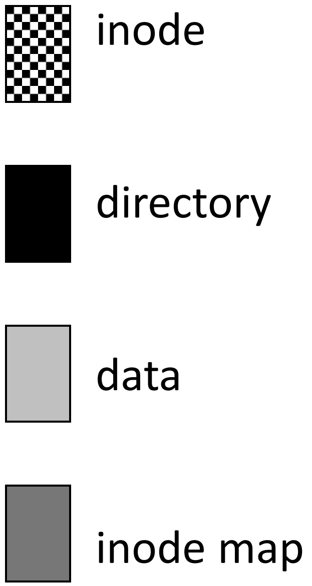
:> echo “quack” > dir1/file1

:> echo “quack again” > dir2/file2

LFS Motivation



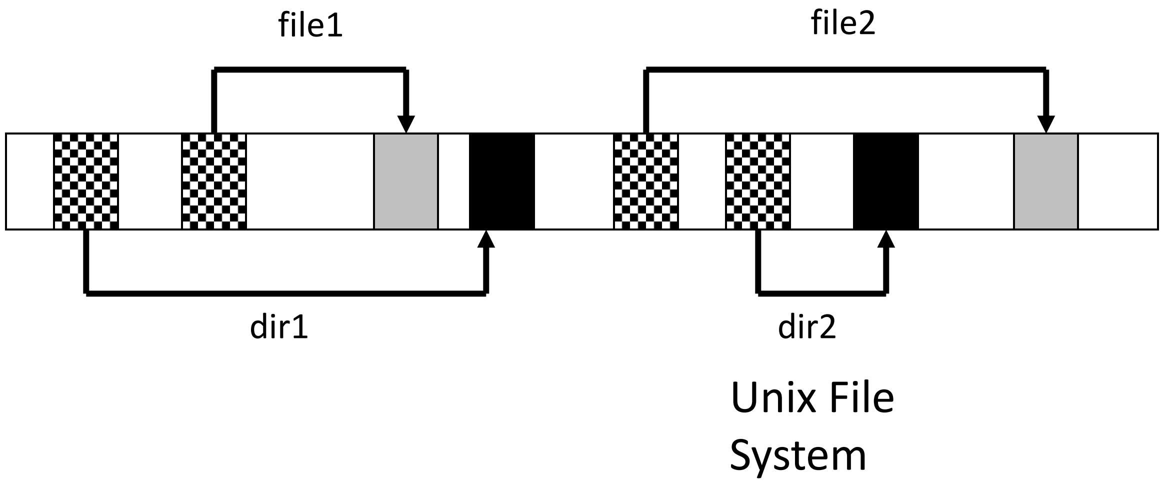
Unix File System

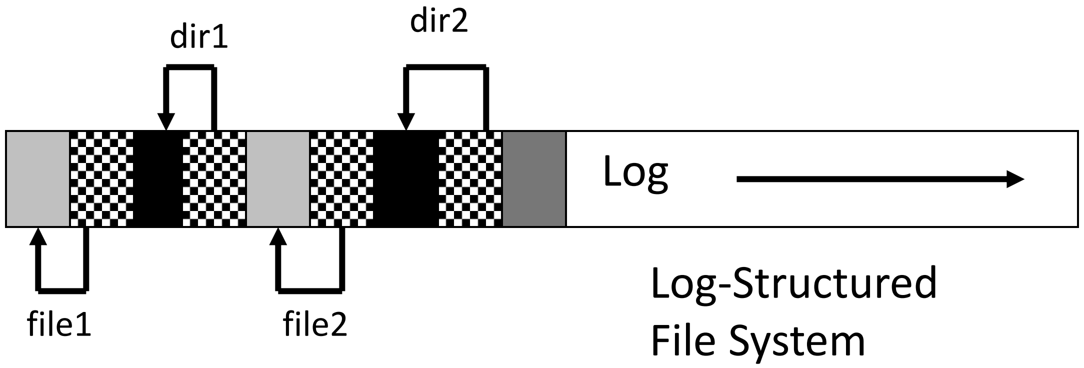


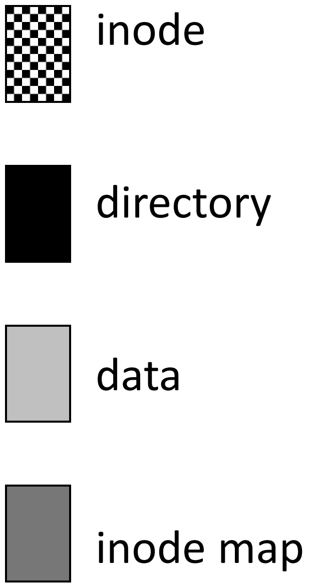
:> echo “quack” > dir1/file1

:> echo “quack again” > dir2/file2

LFS Motivation



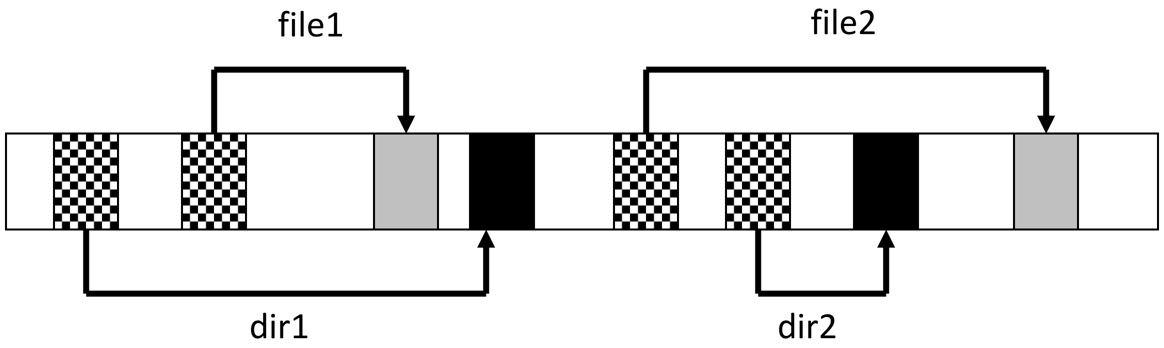




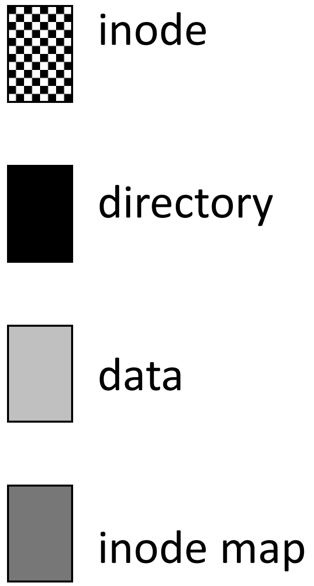
:> echo “quack” > dir1/file1

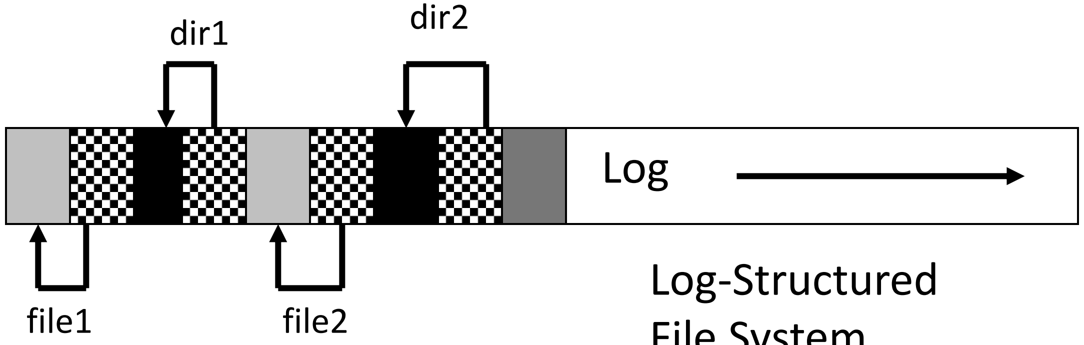
:> echo “quack again” > dir2/file2

LFS Motivation



Unix File System





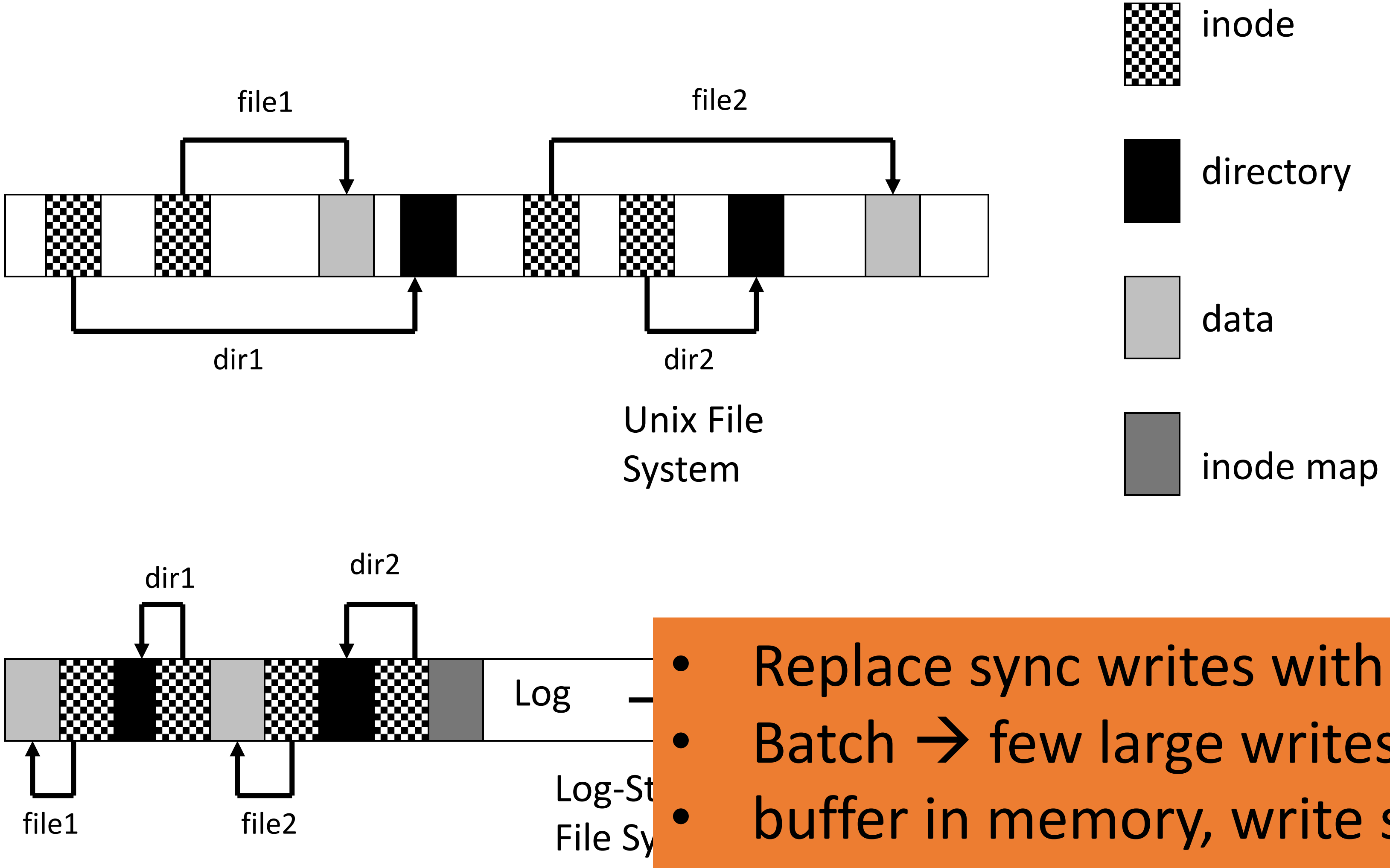
Log-Structured File System

How many I/Os in LFS?

:> echo “quack” > dir1/file1

:> echo “quack again” > dir2/file2

LFS Motivation



- Replace sync writes with async
- Batch → few large writes
- buffer in memory, write segs to disk
- append only, no overwrite in place

LFS Challenges

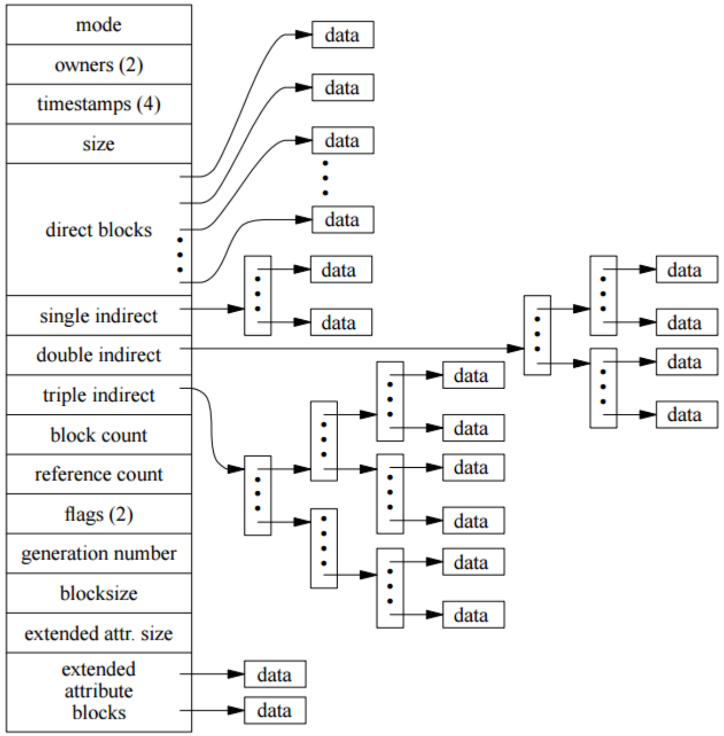
- Metadata design
  - No update in place
  - Nothing has a permanent home
  - How do we find anything?
- Free space management
  - ***We need large extents of free space***
  - ***How do we ensure we always have it?***

OK then…how do we find things?

- How are FS metadata organized in FFS?
  - How do we find an inode?
  - From mkfs.ext4
    - -N number-of-inodes
    - Overrides the default calculation of the number of inodes that should be reserved for the filesystem (which is based on the number of blocks and the bytes-per-inode ratio). This allows the user to specify the number of desired inodes directly.
- How do we find inodes in LFS?
- ***I-node map*** maintains the location of all i-node blocks
  - I-node map blocks stored on the log
    - Along with data blocks/i-node blocks
  - Active blocks cached in main memory
- Fixed ***checkpoint region***
  - on each disk
  - contains the addresses of all ***i-node map*** blocks
    - at checkpoint time (when is that?)

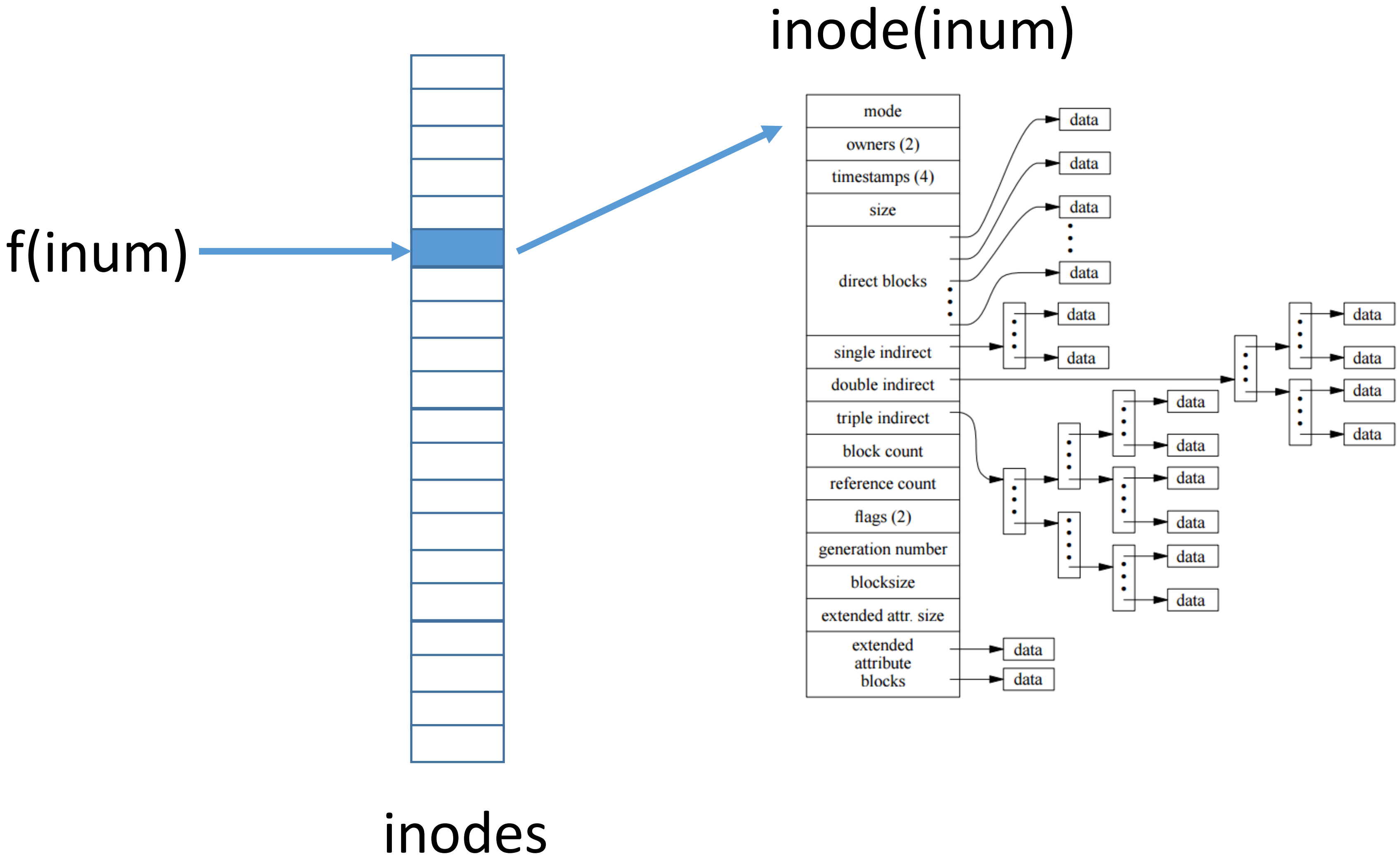
Index structures: FFS

inode(inum)

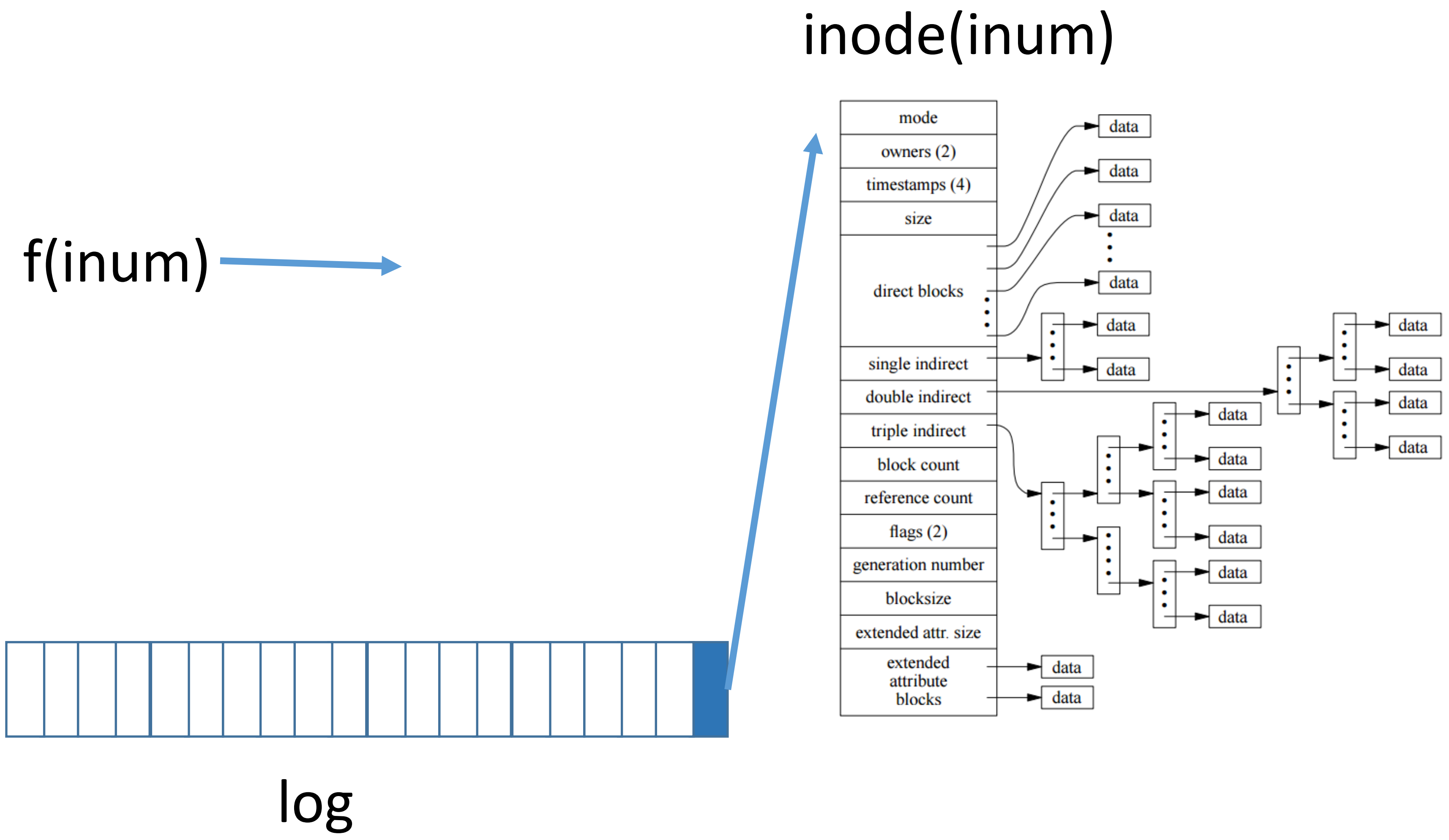


inodes

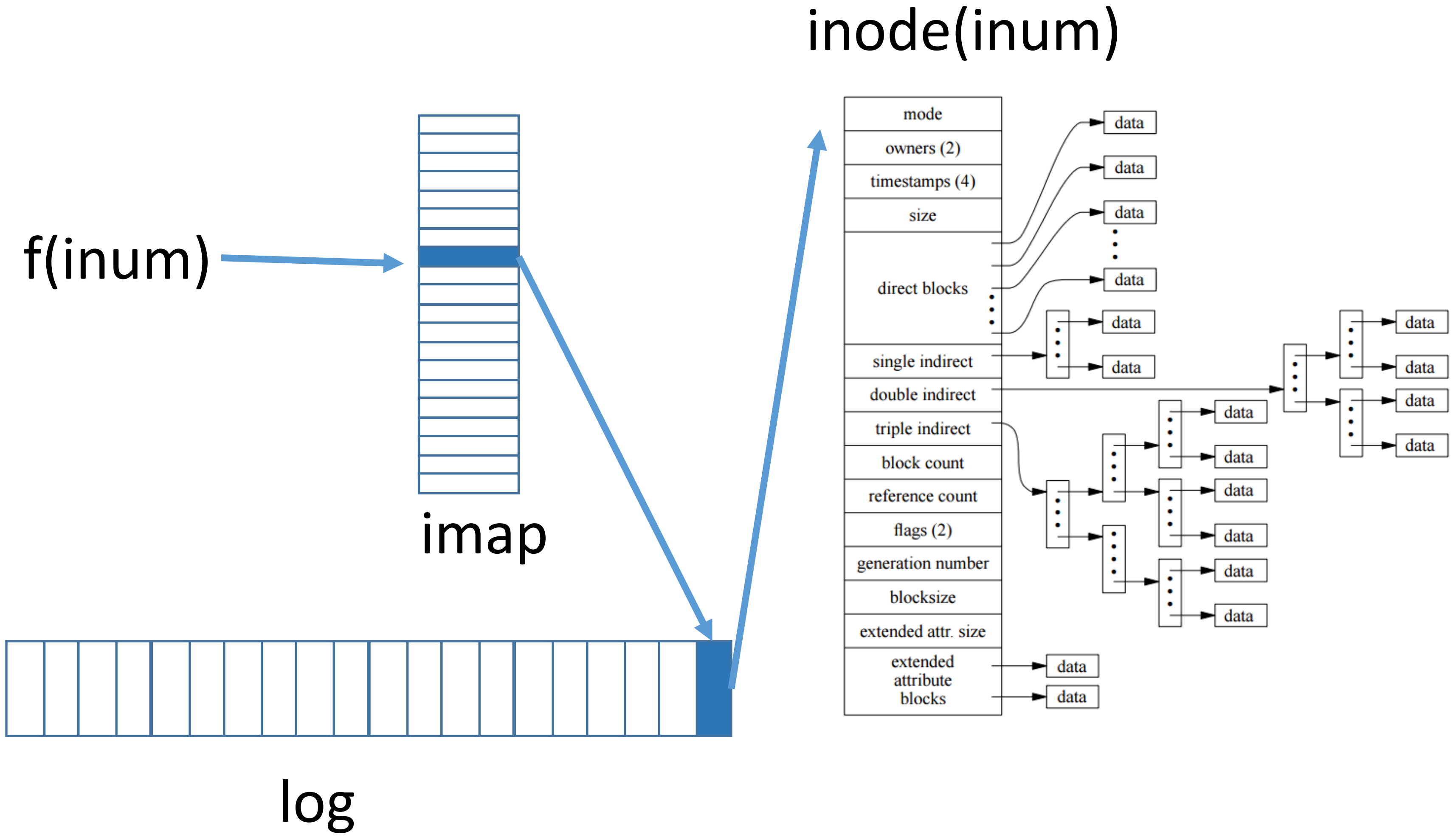
Index structures: FFS



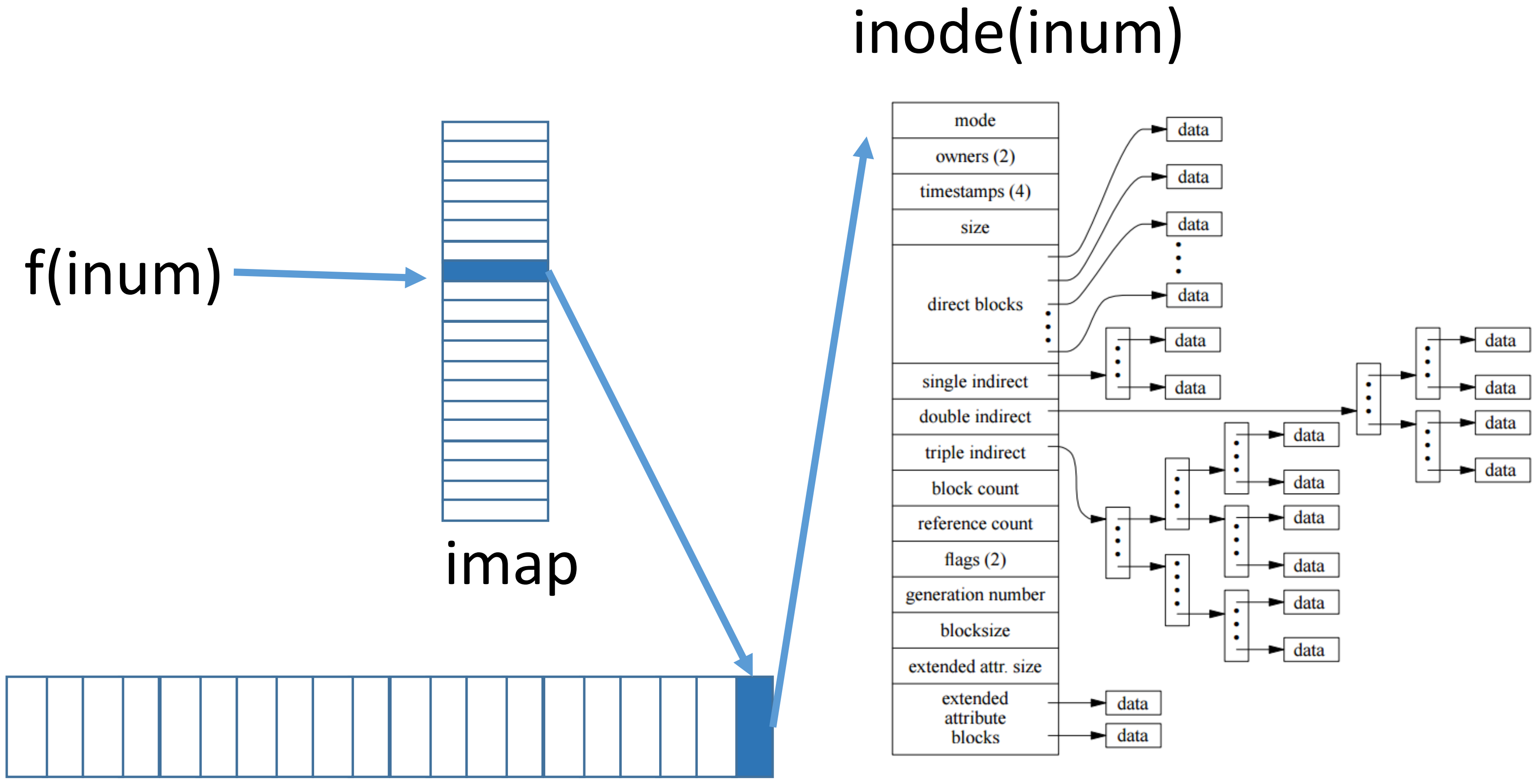
Index structures: LFS



Index structures: LFS



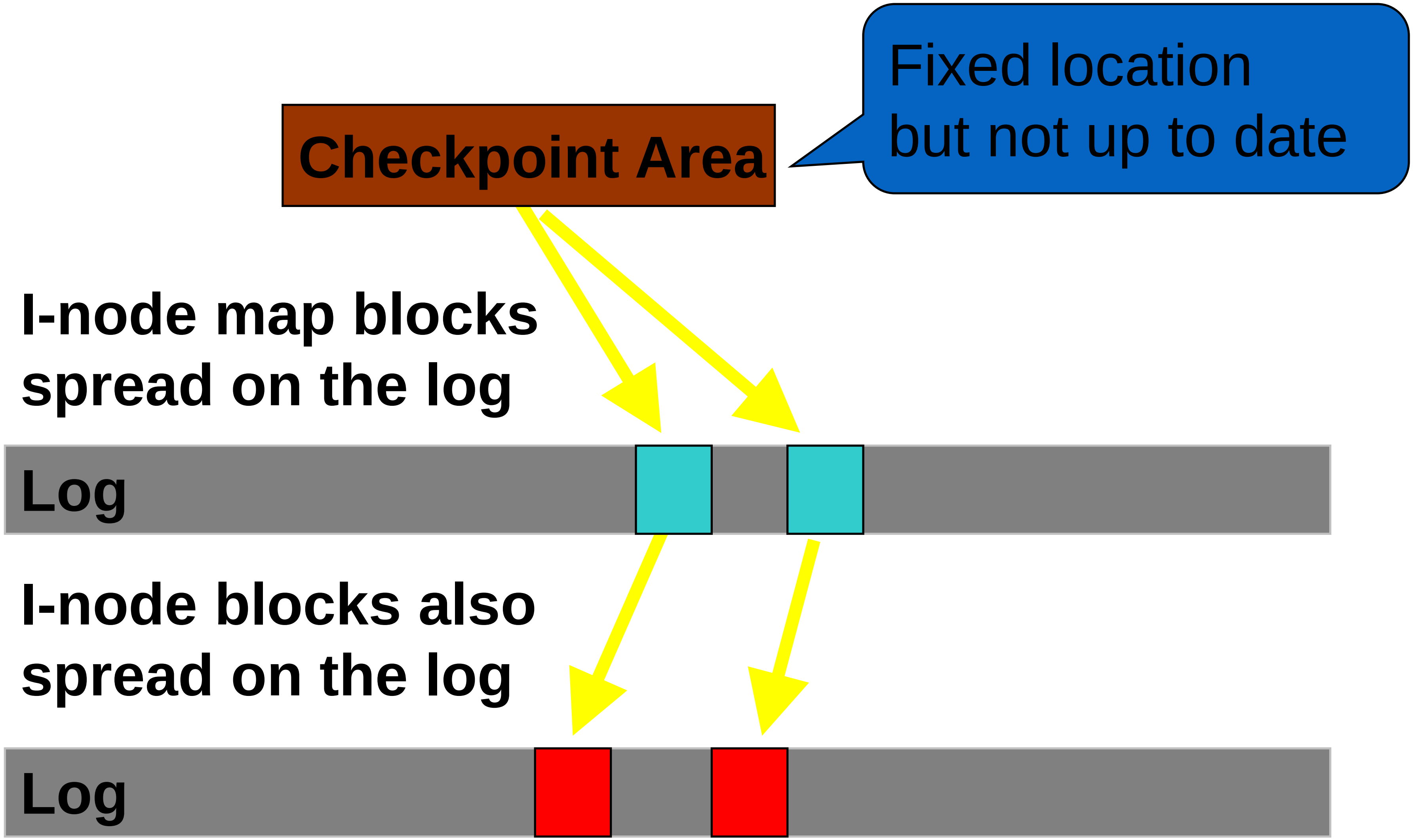
Index structures: LFS



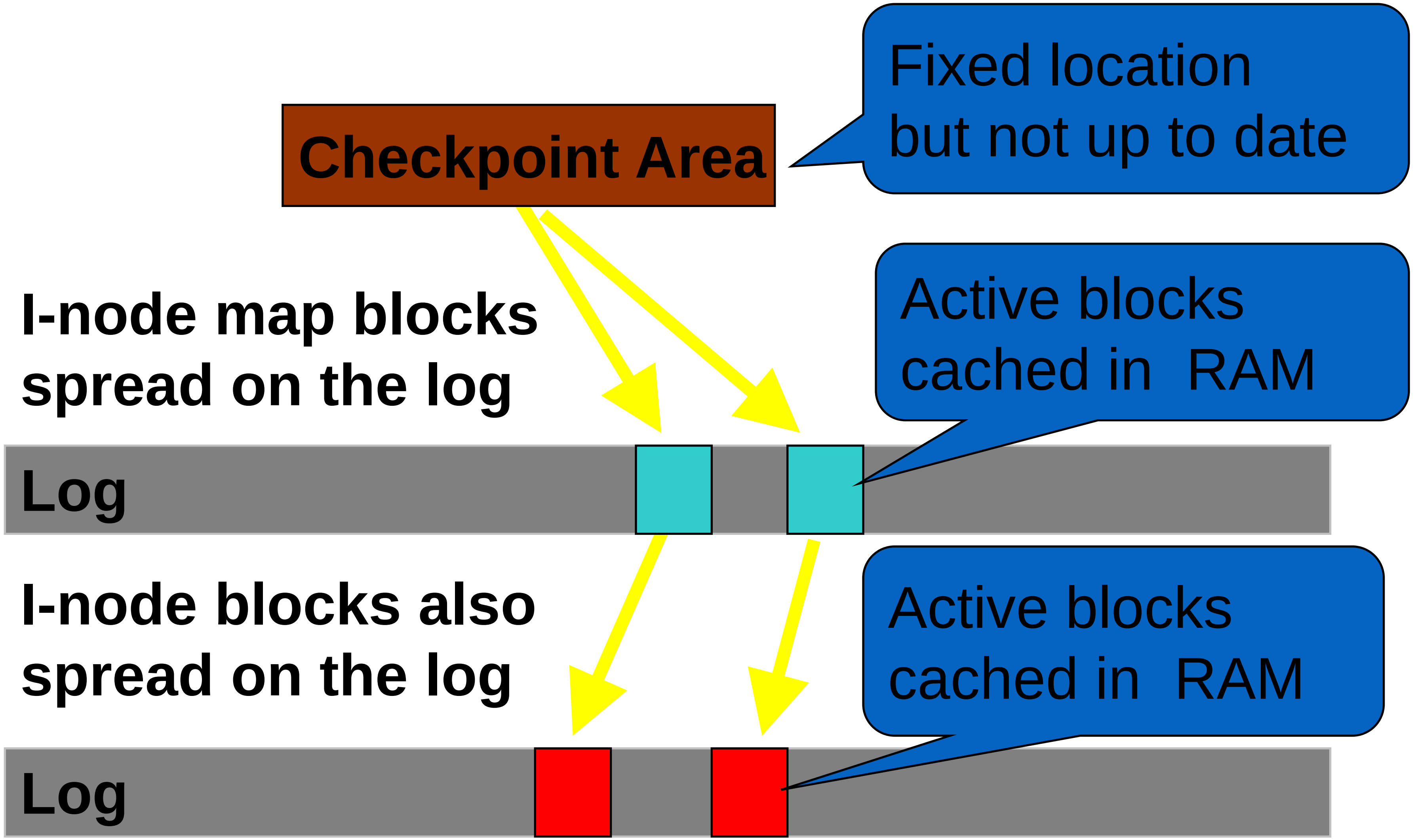
Is this obviously better?

Why doesn’t this break sequential writes?

Accessing an i-node

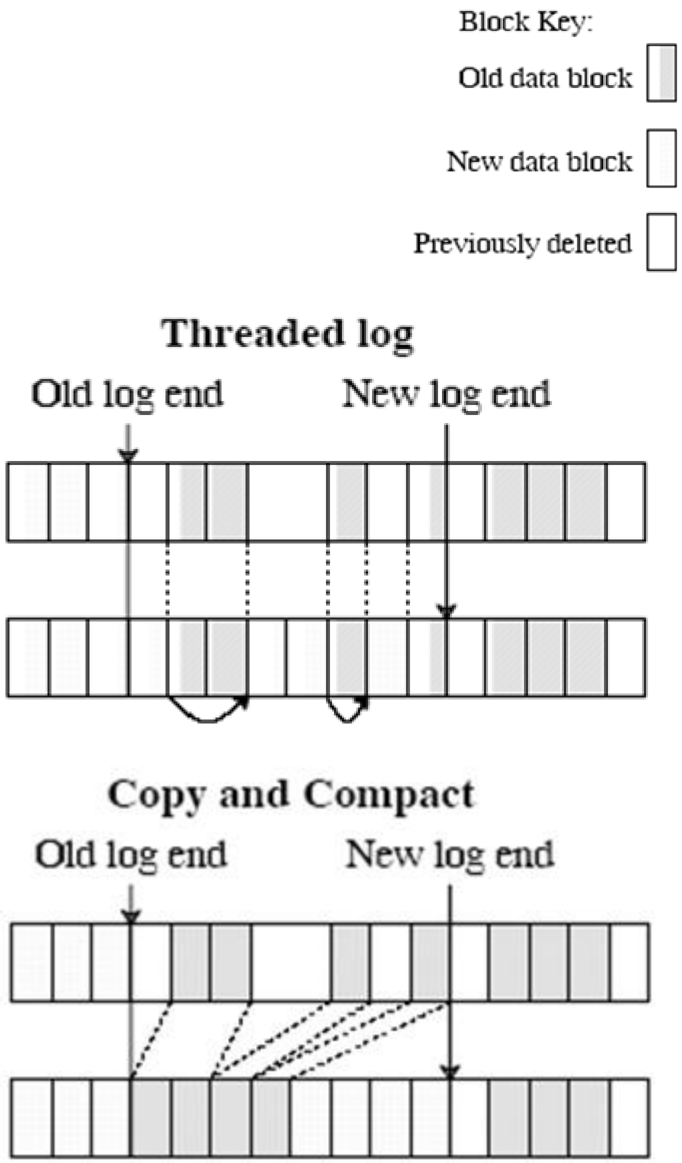


The way it works



Cleaning

- Option 1: threading
  - Put new blocks wherever there are holes
  - Blocks point to next block
  - Pro: doesn't waste time R/W live data
  - Con: storage system entropy: fragmentation!
- Option 2: compact/copy
  - Move live blocks to smaller area
  - Pro: create large extents reliably
  - Con: RW same data over and over



Problem with threaded log—fragmentation
Problem with copy and compact—cost of copying data

Segments: benefits of both

- Chop disk into large segments
- When to use compaction?
  - Compact within segments
- When to use threading?
  - Thread among segments
- Always write to current clean segment before moving on
- How to deal with finite-ness of log?
- Needs a “segment cleaner”

Segment cleaning

- Old segments contain
  - ***live data***
  - "dead data" → files overwritten/deleted
- Segment cleaning → compact/write out live data
- Segment summary block → per-segment metadata

Algorithm:

Read segments into memory
Identify the live data
Write live data (contiguously) to clean segments

Key issues: where/when to write?

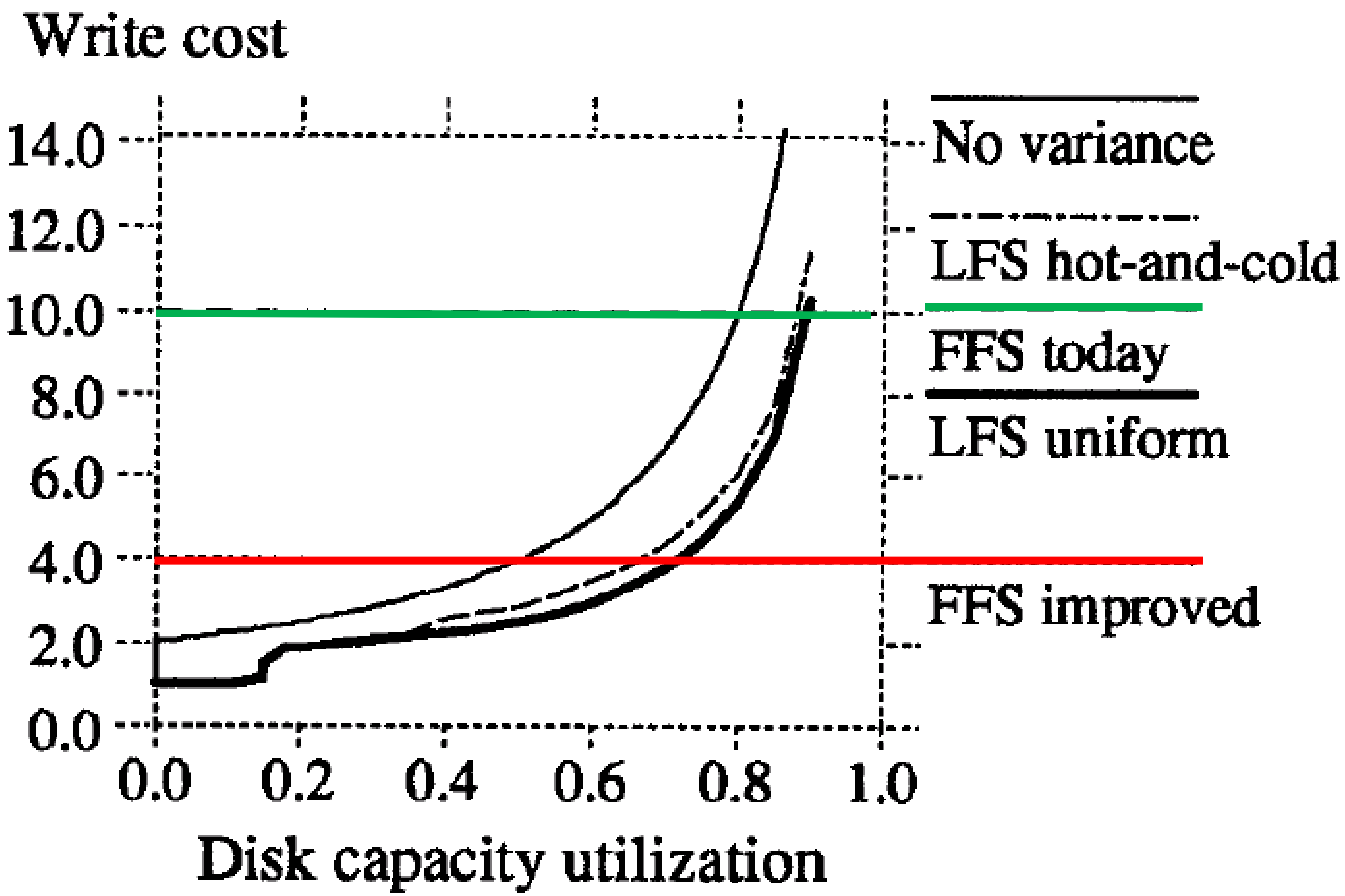
- *Want to avoid repeated moves of stable files*
- *Minimize overhead for writes: "write cost"*

$$\text{write cost} = \frac{\text{total bytes read and written}}{\text{new data written}}$$

$$= \frac{\text{read segs} + \text{write live} + \text{write new}}{\text{new data written}}$$

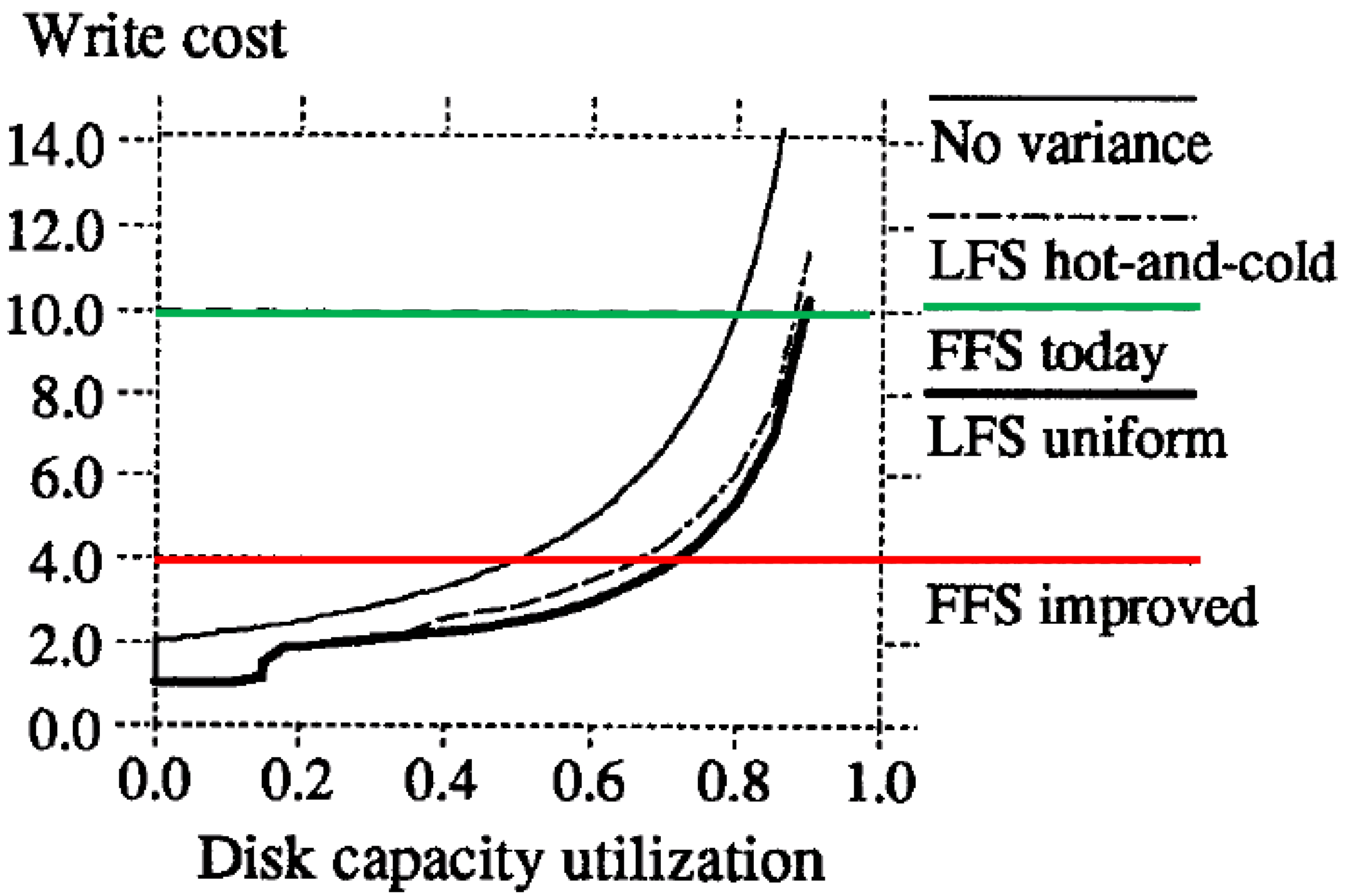
$$= \frac{N + N^{*}u + N^{*}(1-u)}{N^{*}(1-u)} = \frac{2}{1-u}$$

Write Cost and Cleaning Policy



- ***No variance*** → write cost computed with formula (all segments have same *u* (?!))
- ***LFS uniform*** → greedy policy (always clean least util)
- ***LFS hot-and-cold*** → greedy policy + sorts blocks by age
- ***FFS improved*** → estimate of best possible FFS performance

Write Cost and Cleaning Policy

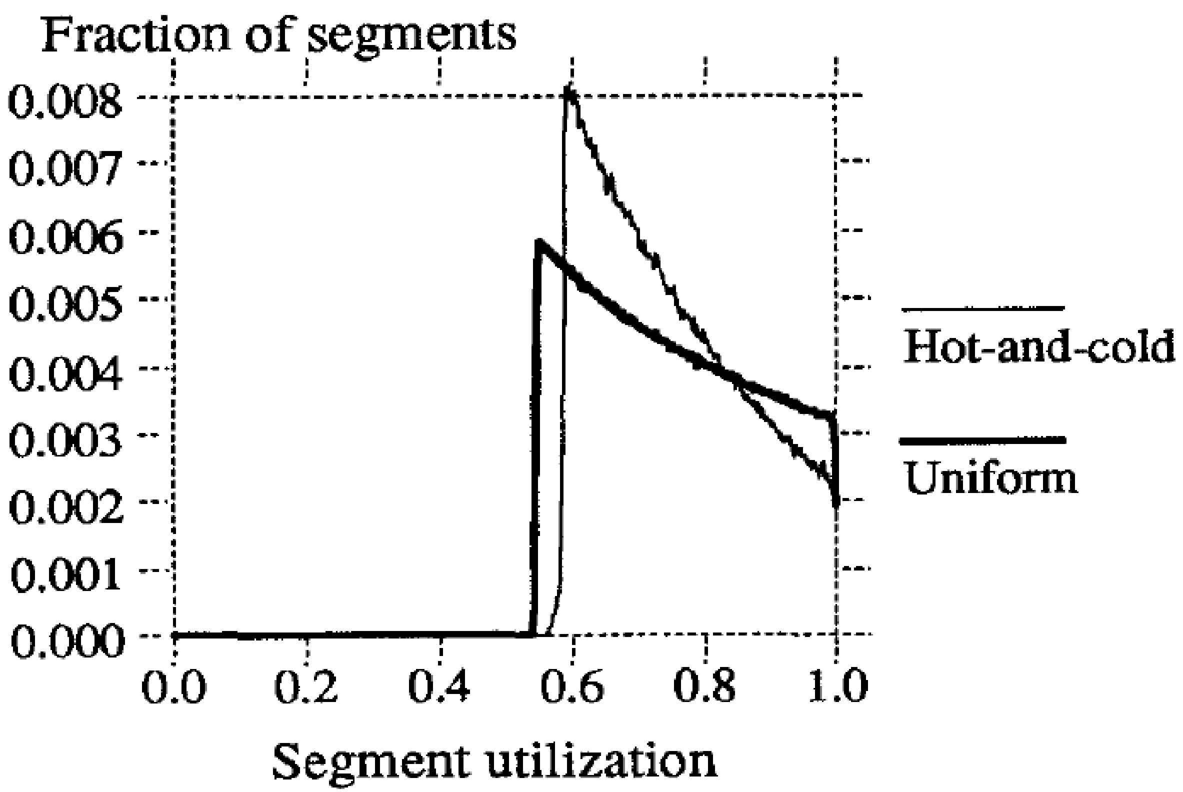


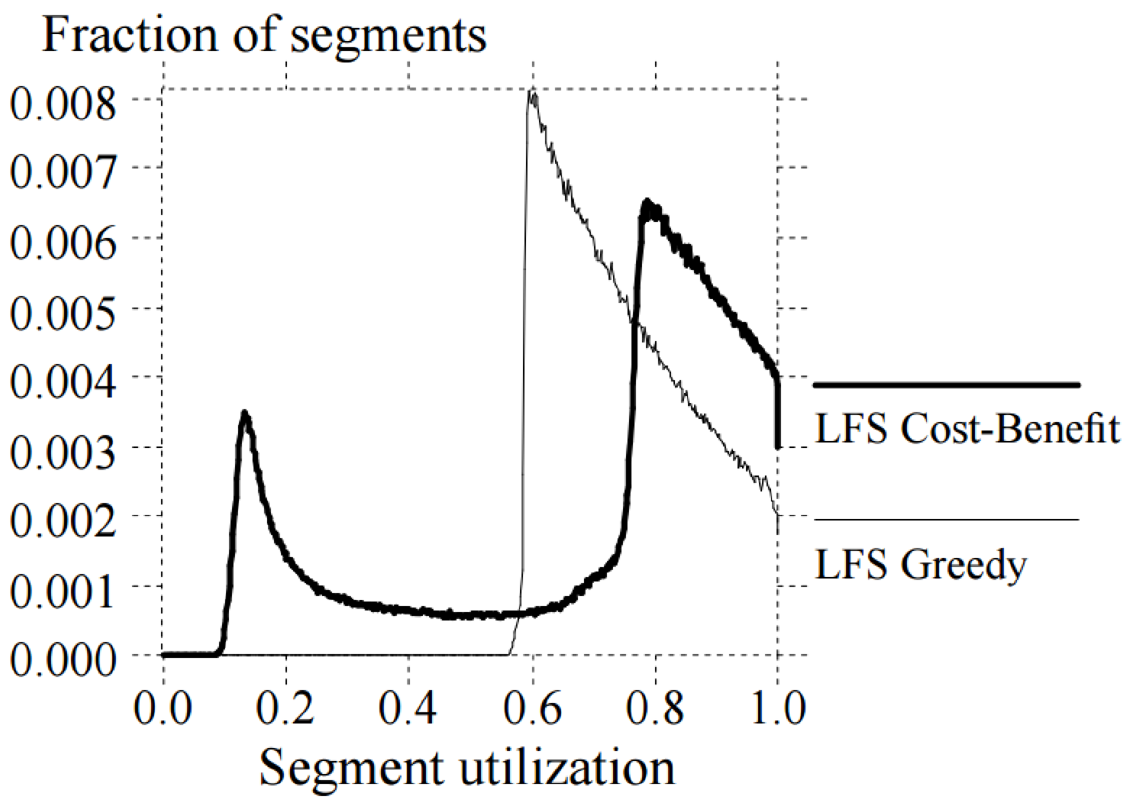
- ***No variance*** → write cost computed with formula (all segments have same *u* (?!))
- ***LFS uniform*** → greedy policy (always clean least util)
- ***LFS hot-and-cold*** → greedy policy + sorts blocks by age
- ***FFS improved*** → estimate of best possible FFS performance

Observations:

- *Write cost very sensitive to u*
  - *High disk util → in more frequent cleaning*
- *Free space*
  - *valuable in cold segments*
  - *Not valuable hot segments*
  - *Value depends on stability of live blocks in segment*

Segment cleaning policies (II)





$$\frac{\text{benefit}}{\text{cost}} = \frac{\text{free space generated*age of data}}{\text{cost}} = \frac{(1 - u)\text{*age}}{1 + u}$$

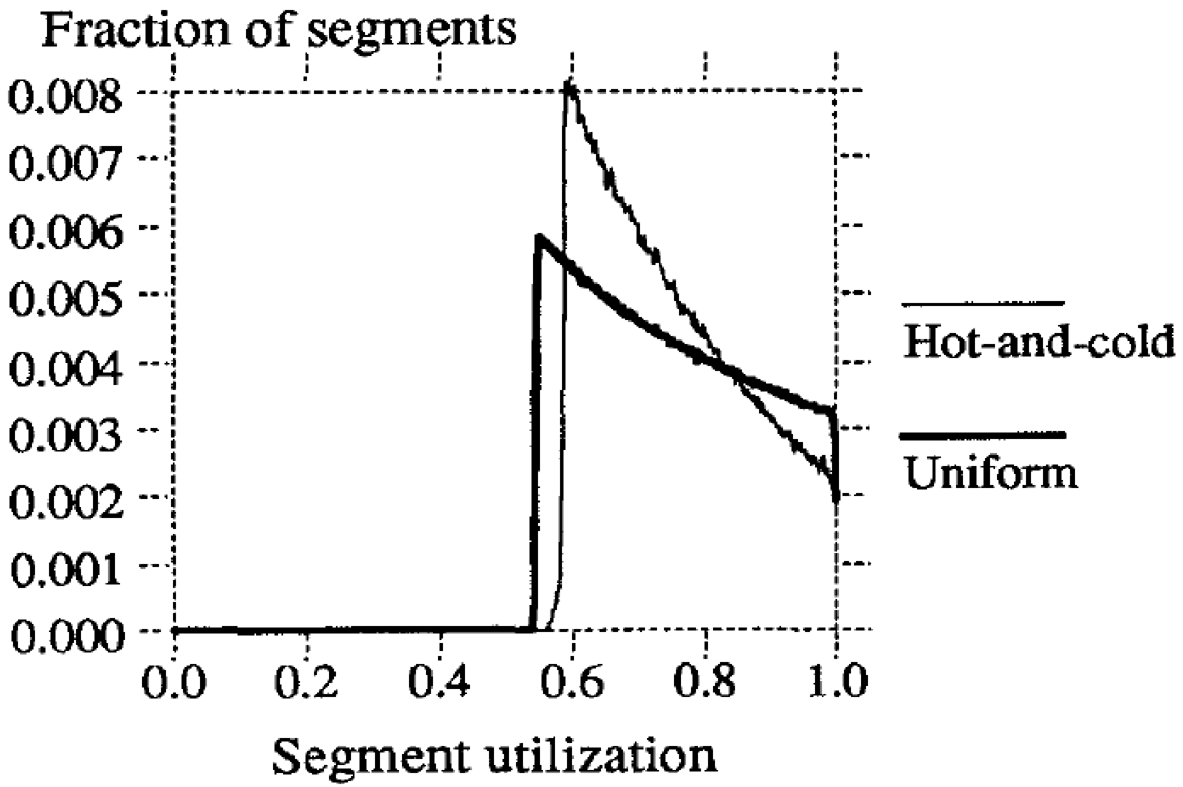
Segment cleaning policies (II)

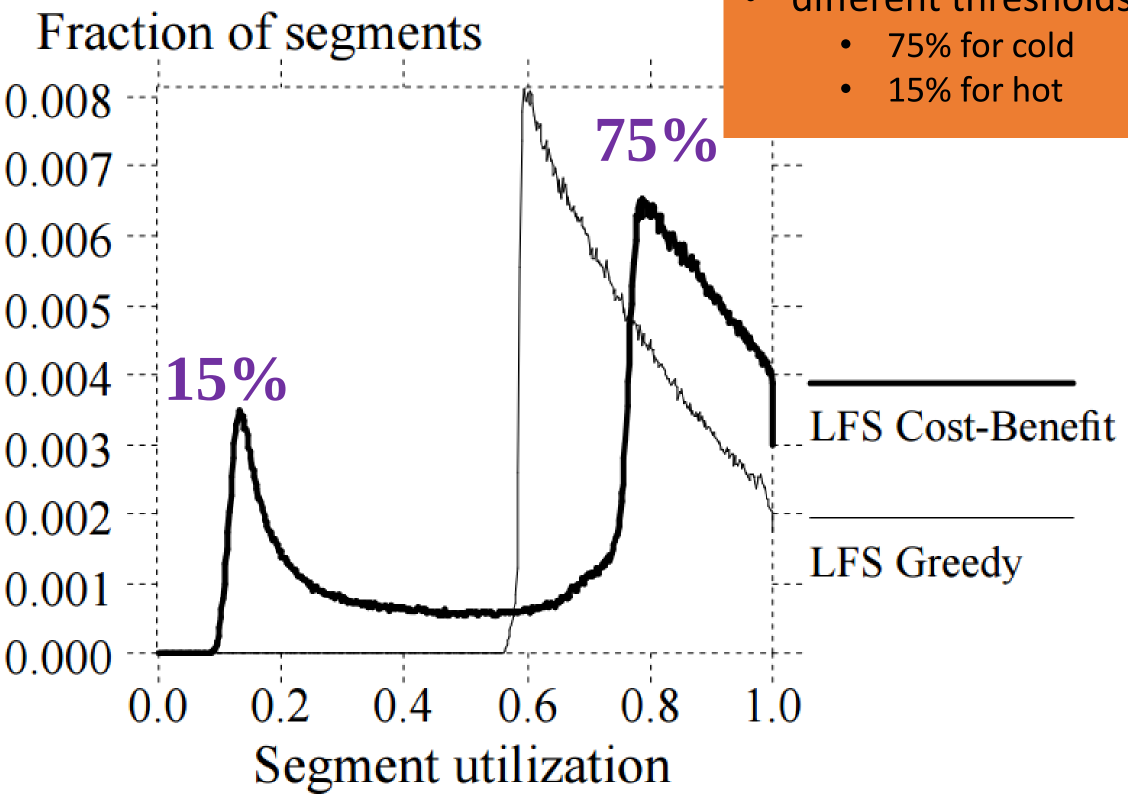
Without cost-benefit:

- Locality skews distribution toward cleaning
- Segments cleaned at higher utilizations than optimal

With cost-benefit:

- different thresholds
  - 75% for cold
  - 15% for hot





$$\frac{\text{benefit}}{\text{cost}} = \frac{\text{free space generated*age of data}}{\text{cost}} = \frac{(1-u)\text{*age}}{1+u}$$

Crash-recovery: Checkpoints/Roll-forward

Checkpoint = log position s.t. all FS metadata consistent

- Create:
  - 1. Write out all dirty info to log, including metadata
  - 2. Write *checkpoint region* to special place on disk
- On reboot:
  - read checkpoint region to init in-memory data structures
  - 2 checkpoints handles checkpoint write crash

Roll-Forward: *try to recover as much data as possible*

- Look at segment summary blocks
  - if new inode and data blocks, but no inode map entry → update inode map
  - if only data blocks, ignore
- Need special record for directory change
  - avoids problems with inode but *not* directory written
  - appears before the corresponding directory block or inode
  - again, roll-forward

Crash-recovery: Checkpoints/Roll-forward

Checkpoint = log position s.t. all FS metadata consistent

- Create:
  - 1. Write out all dirty info to log, including metadata
  - 2. Write *checkpoint region* to special place on disk
- On reboot:
  - read checkpoint region to init in-memory data structures
  - 2 checkpoints handles checkpoint write crash

Roll-Forward: *try to recover as much data as possible*

- Look at segment summary blocks
  - if new inode and data blocks, but no inode map entry → update
  - if only data blocks, ignore
- Need special record for directory change
  - avoids problems with inode but *not* directory written
  - appears before the corresponding directory block or inode
  - again, roll-forward

- Crash in UNIX is a mess
  - disk DS maybe inconsistent
  - fsck slow
- A mess in LFS?
  - find end of log
  - scan backward to last consistent state

When is LFS better?

- LFS wins, **relative to FFS**
  - metadata-heavy workloads
    - small file writes
    - deletes

  (metadata requires an additional write, and FFS does this synchronously)
- LFS loses, **relative to FFS**
  - many files are partially over-written in random order, then read
    - file gets spread throughout the log
- LFS vs. JFS
  - JFS is “robust” like LFS but data must eventually be written back “where it came from” so disk bandwidth is still an issue

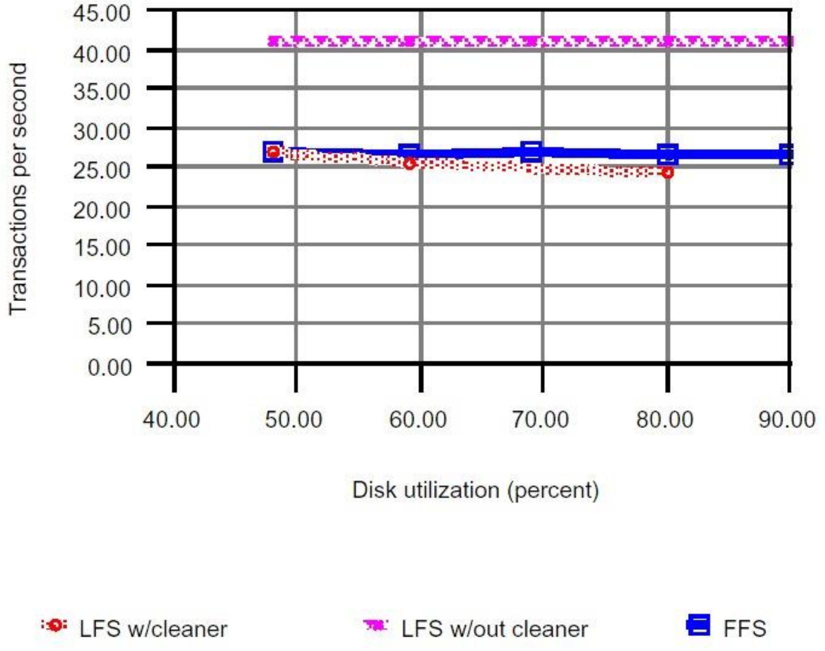
LFS: key takeaways

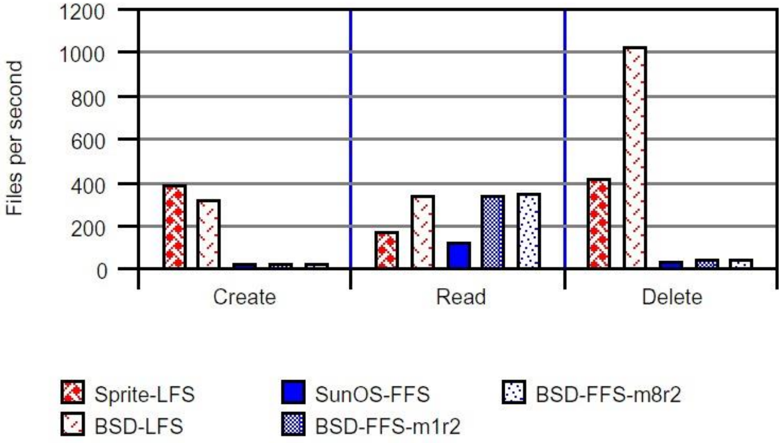
- Big memory → reads served from cache, optimize for writes
- Take journaling to logical extreme
- Hard problems:
  - Find data in the log
  - Cleaning
- Key ideas:
  - Log your writes, log is ground truth
  - Indexing: imap → data can live anywhere

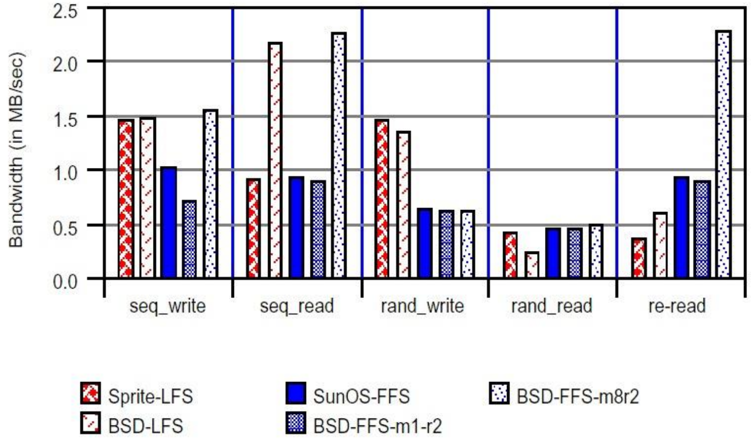
Seltzer v Ousterhout: what a kerfuffle!

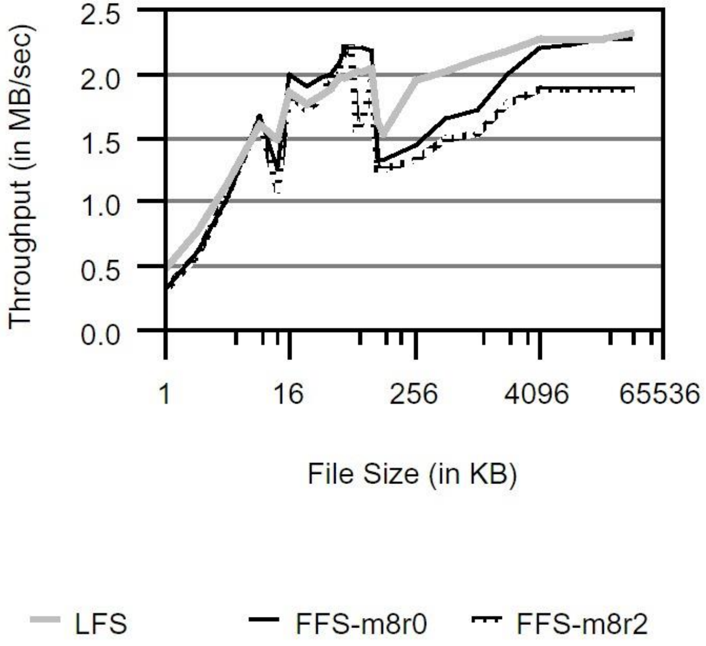
- What did you think?
- Could you extract what the controversy was?
  - Why it occurred?
- Seltzer papers: intentional challenge of LFS hypothesis?
  - Implement LFS in BSD
  - Validate by comparing against sprite-LFS
  - Explore file size / access pattern (seq/rand) vs. perf
  - Characterize disk fullness impact on LFS
  - Characterize fragmentation impact on FFS

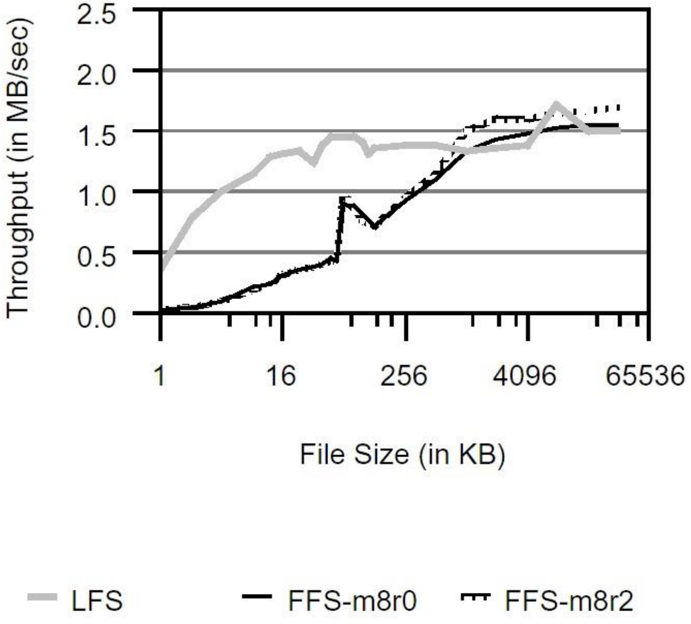
FFS v LFS


Transactions per second
45.00
40.00
35.00
30.00
25.00
20.00
15.00
10.00
5.00
0.00
40.00
50.00
60.00
70.00
80.00
90.00
Disk utilization (percent)
LFS w/cleaner
LFS w/out cleaner
FFS

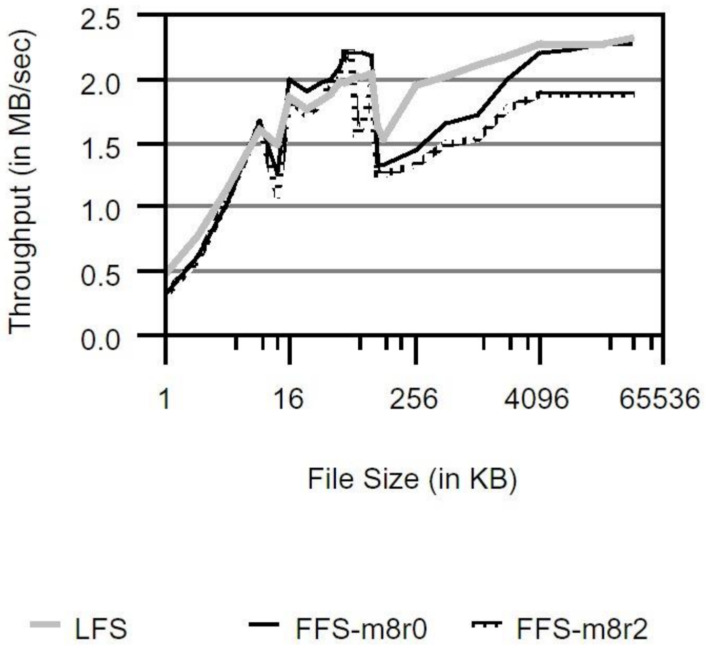

Files per second
1200
1000
800
600
400
200
0
Create
Read
Delete
Sprite-LFS
BSD-LFS
SunOS-FFS
BSD-FFS-m1r2
BSD-FFS-m8r2


Bandwidth (in MB/sec)
2.5
2.0
1.5
1.0
0.5
0.0
seq_write
seq_read
rand_write
rand_read
re-read
Sprite-LFS
BSD-LFS
SunOS-FFS
BSD-FFS-m1-r2
BSD-FFS-m8r2

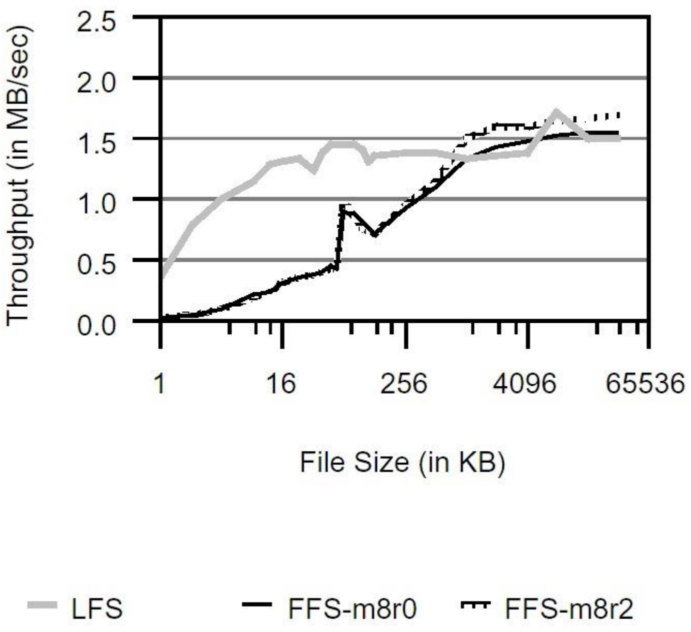

Throughput (in MB/sec)
2.5
2.0
1.5
1.0
0.5
0.0
1
16
256
4096
65536
File Size (in KB)
LFS
FFS-m8r0
FFS-m8r2


Throughput (in MB/sec)
2.5
2.0
1.5
1.0
0.5
0.0
1
16
256
4096
65536
File Size (in KB)
LFS
FFS-m8r0
FFS-m8r2

FFS v LFS



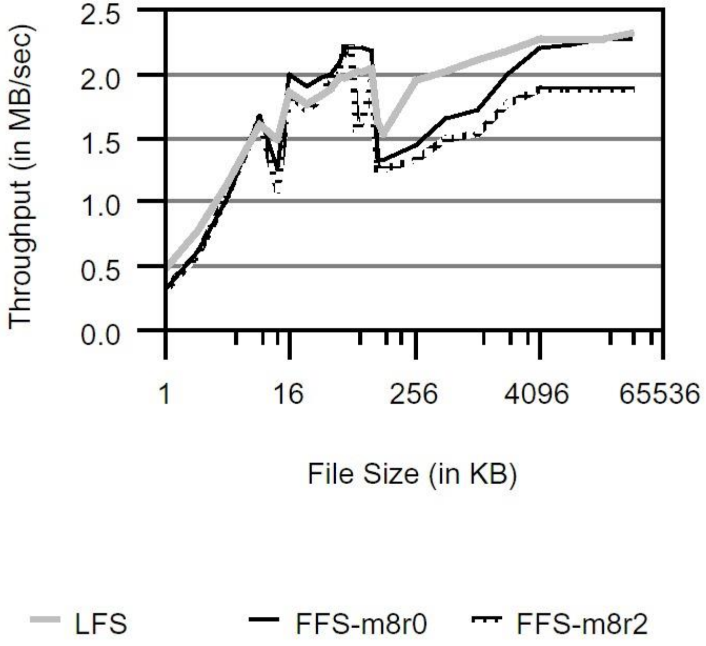
Throughput (in MB/sec)
2.5
2.0
1.5
1.0
0.5
0.0
1
16
256
4096
65536
File Size (in KB)
LFS
FFS-m8r0
FFS-m8r2



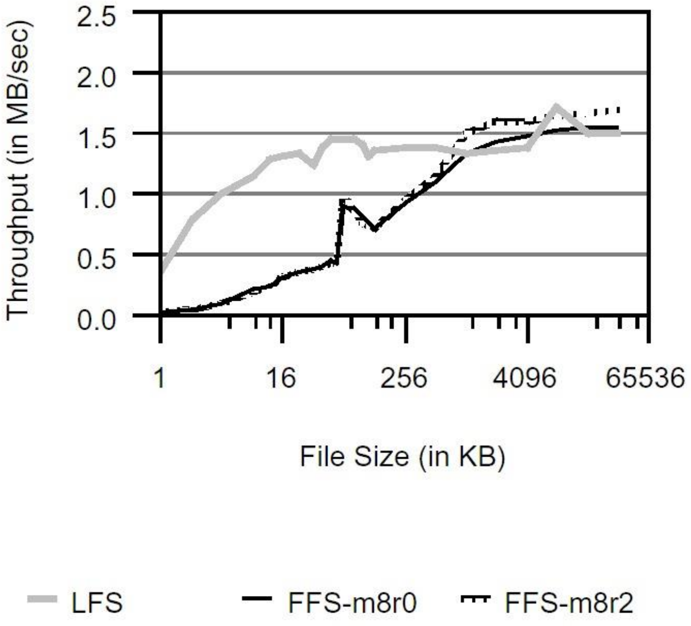
Throughput (in MB/sec)
2.5
2.0
1.5
1.0
0.5
0.0
1
16
256
4096
65536
File Size (in KB)
LFS
FFS-m8r0
FFS-m8r2

FFS v LFS

Read

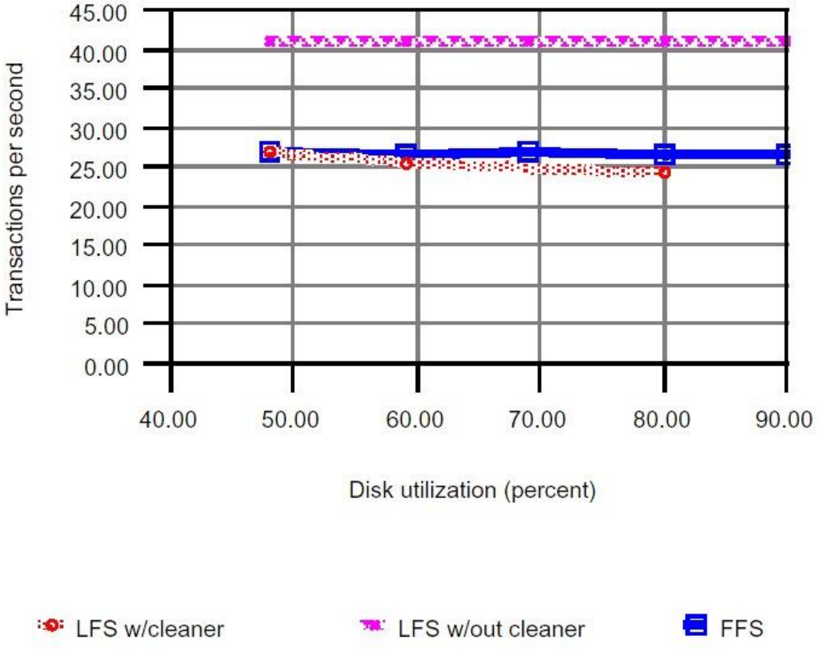


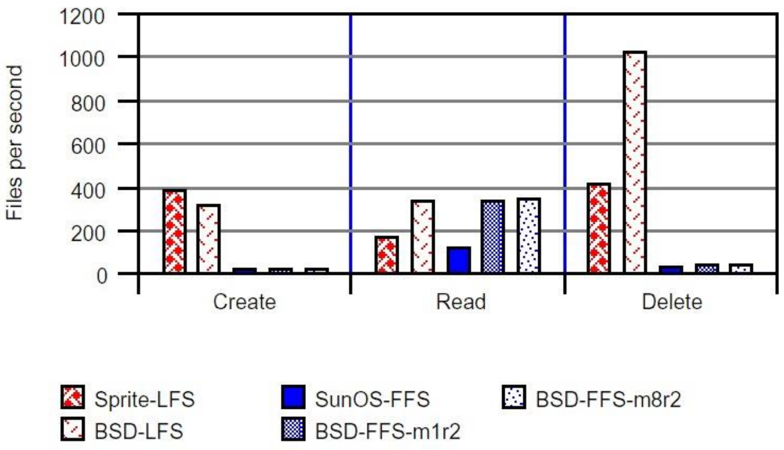
Create

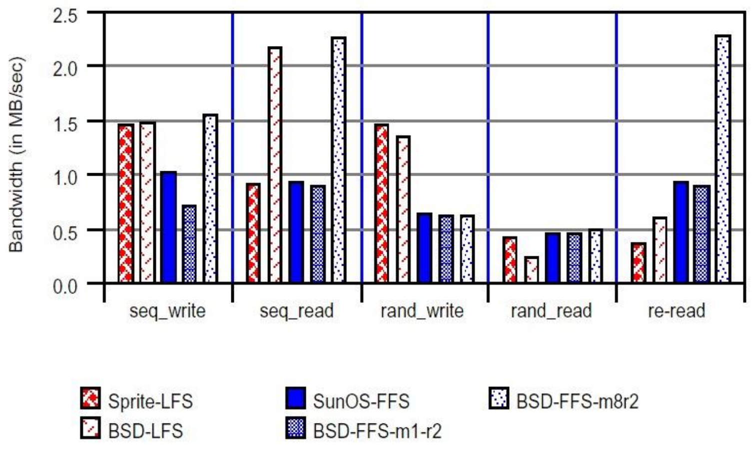


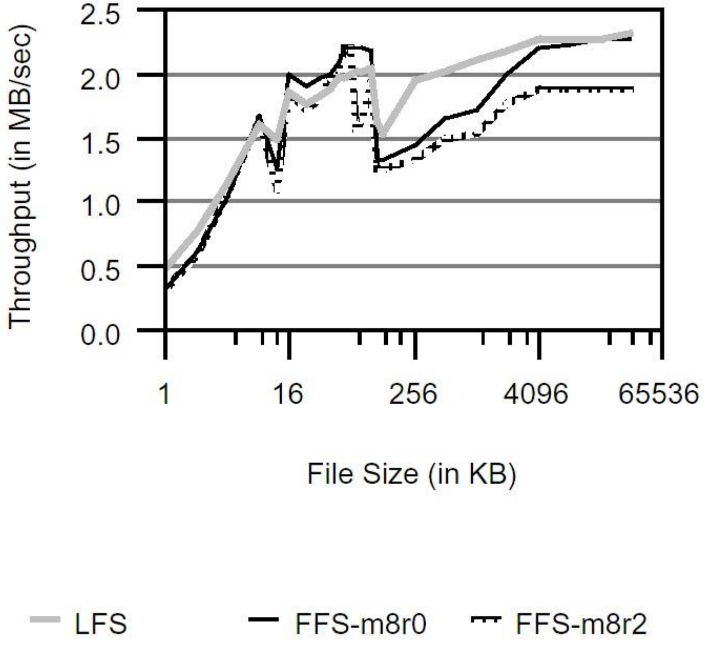
"m" == maxcontig, "r" == rotdelay

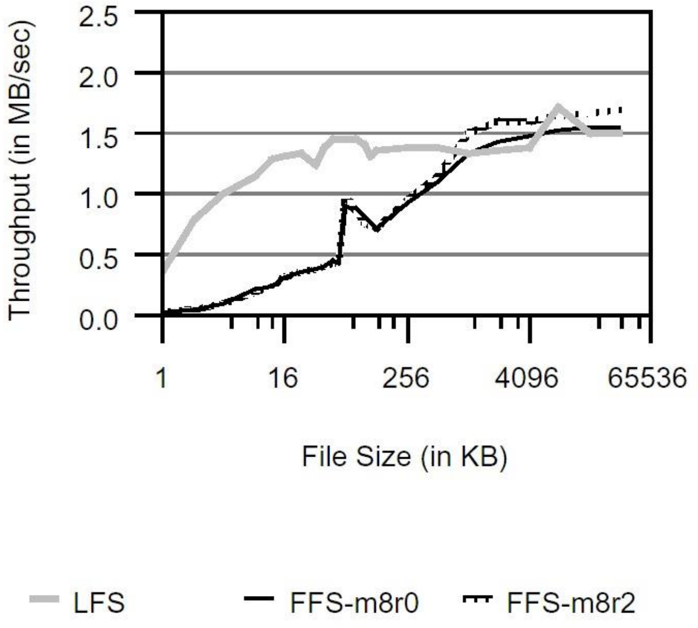
FFS v LFS


Transactions per second
45.00
40.00
35.00
30.00
25.00
20.00
15.00
10.00
5.00
0.00
40.00
50.00
60.00
70.00
80.00
90.00
Disk utilization (percent)
LFS w/cleaner
LFS w/out cleaner
FFS

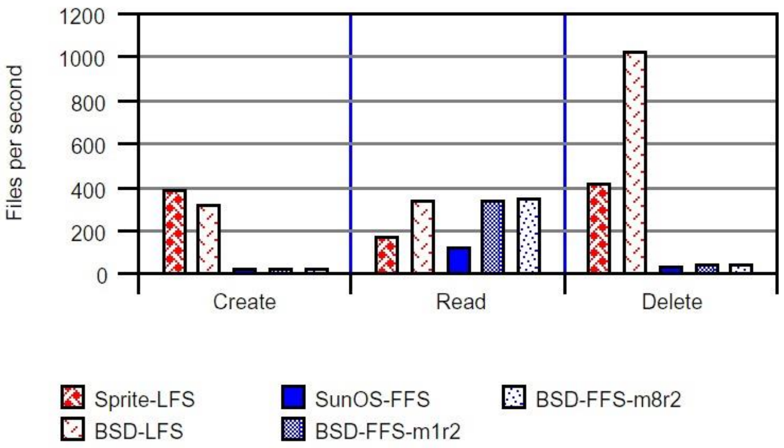

Files per second
1200
1000
800
600
400
200
0
Create
Read
Delete
Sprite-LFS
BSD-LFS
SunOS-FFS
BSD-FFS-m1r2
BSD-FFS-m8r2


Bandwidth (in MB/sec)
2.5
2.0
1.5
1.0
0.5
0.0
seq_write
seq_read
rand_write
rand_read
re-read
Sprite-LFS
BSD-LFS
SunOS-FFS
BSD-FFS-m1-r2
BSD-FFS-m8r2

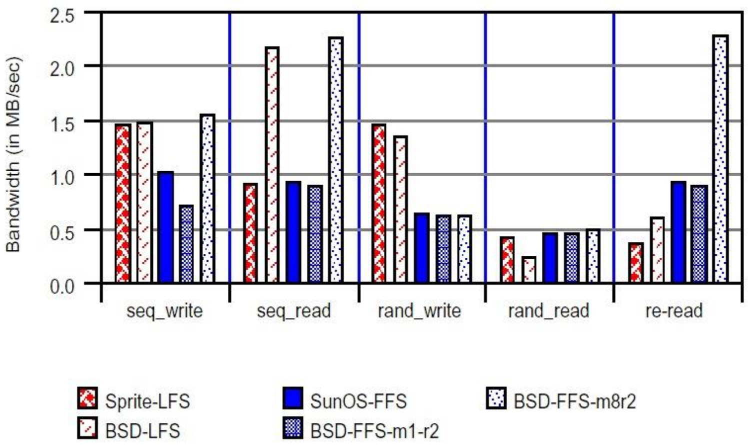

Throughput (in MB/sec)
2.5
2.0
1.5
1.0
0.5
0.0
1
16
256
4096
65536
File Size (in KB)
LFS
FFS-m8r0
FFS-m8r2


Throughput (in MB/sec)
2.5
2.0
1.5
1.0
0.5
0.0
1
16
256
4096
65536
File Size (in KB)
LFS
FFS-m8r0
FFS-m8r2

FFS v LFS

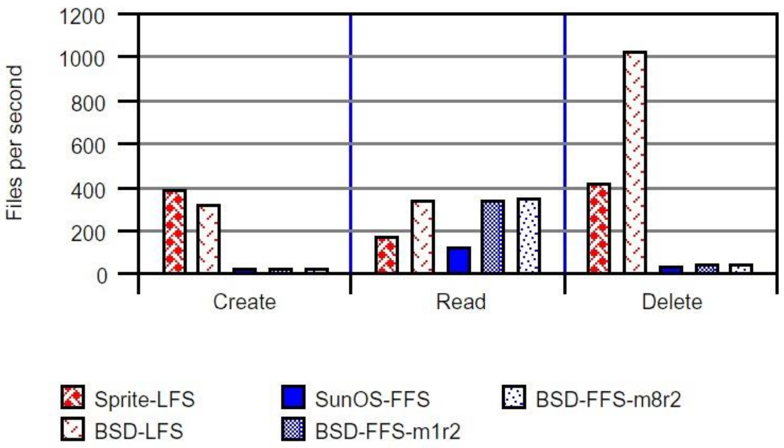


Files per second
1200
1000
800
600
400
200
0
Create
Read
Delete
Sprite-LFS
BSD-LFS
SunOS-FFS
BSD-FFS-m1r2
BSD-FFS-m8r2

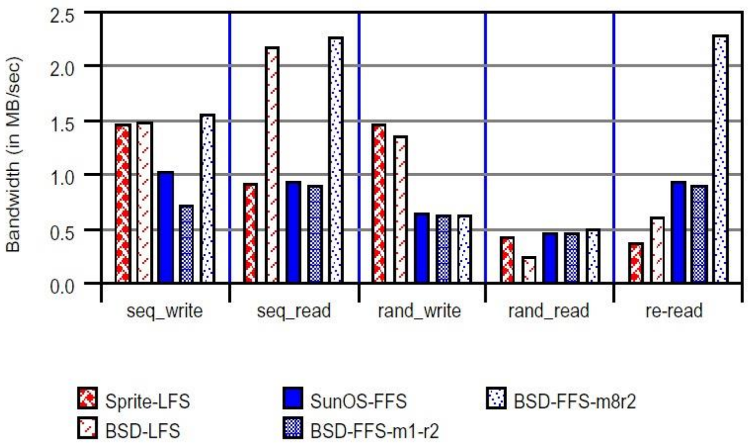


Bandwidth (in MB/sec)
2.5
2.0
1.5
1.0
0.5
0.0
seq_write
seq_read
rand_write
rand_read
re-read
Sprite-LFS
BSD-LFS
SunOS-FFS
BSD-FFS-m1-r2
BSD-FFS-m8r2

FFS v LFS



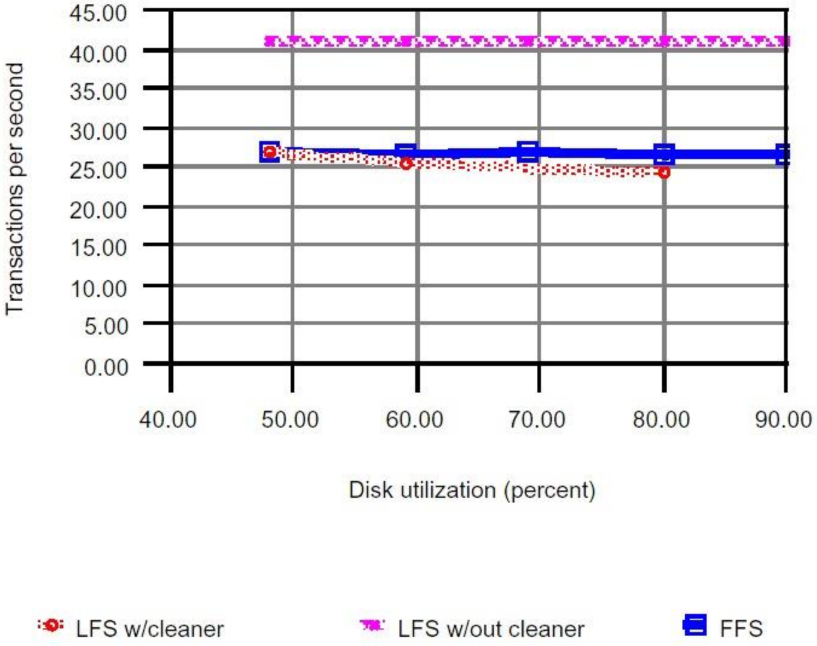
Small files



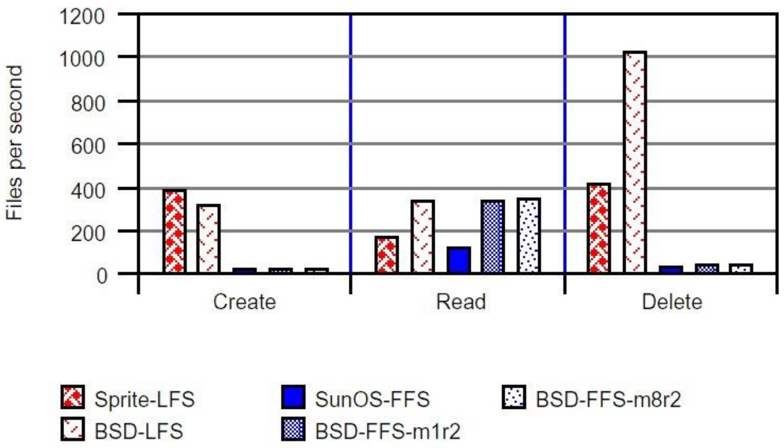
Large Files

“m” == maxcontig, “r” == rotdelay

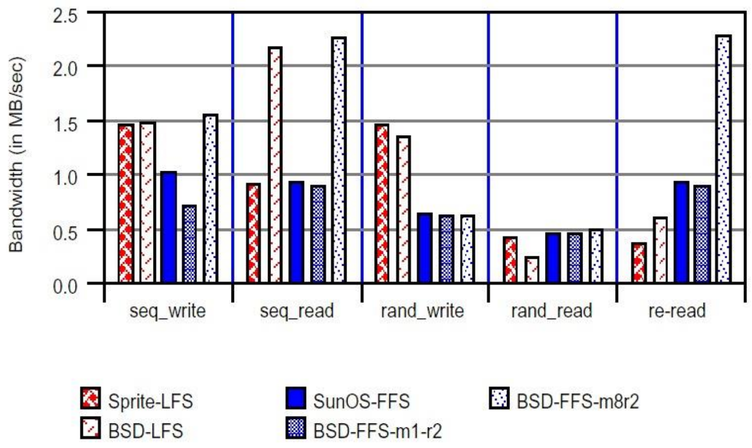
FFS v LFS



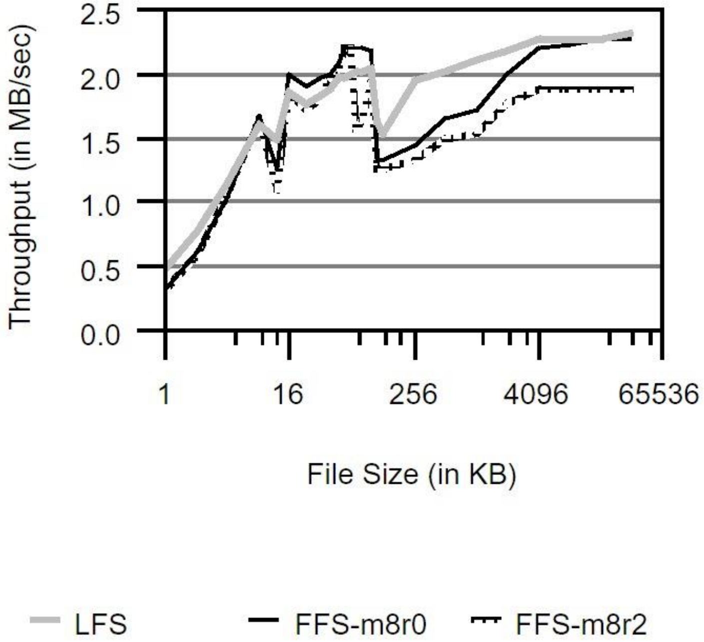
45.00
40.00
35.00
30.00
25.00
20.00
15.00
10.00
5.00
0.00
Transactions per second
40.00
50.00
60.00
70.00
80.00
90.00
Disk utilization (percent)
LFS w/cleaner
LFS w/out cleaner
FFS



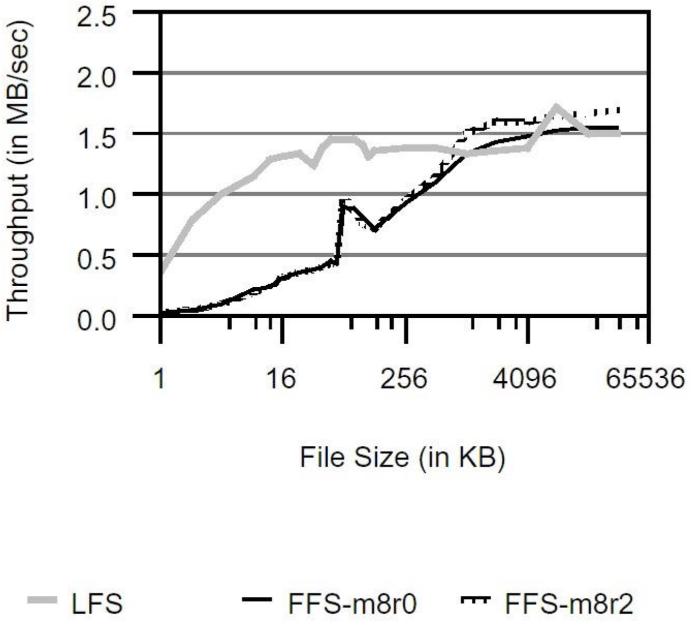
1200
1000
800
600
400
200
0
Files per second
Create
Read
Delete
Sprite-LFS
BSD-LFS
SunOS-FFS
BSD-FFS-m1r2
BSD-FFS-m8r2



2.5
2.0
1.5
1.0
0.5
0.0
Bandwidth (in MB/sec)
seq_write
seq_read
rand_write
rand_read
re-read
Sprite-LFS
BSD-LFS
SunOS-FFS
BSD-FFS-m1-r2
BSD-FFS-m8r2

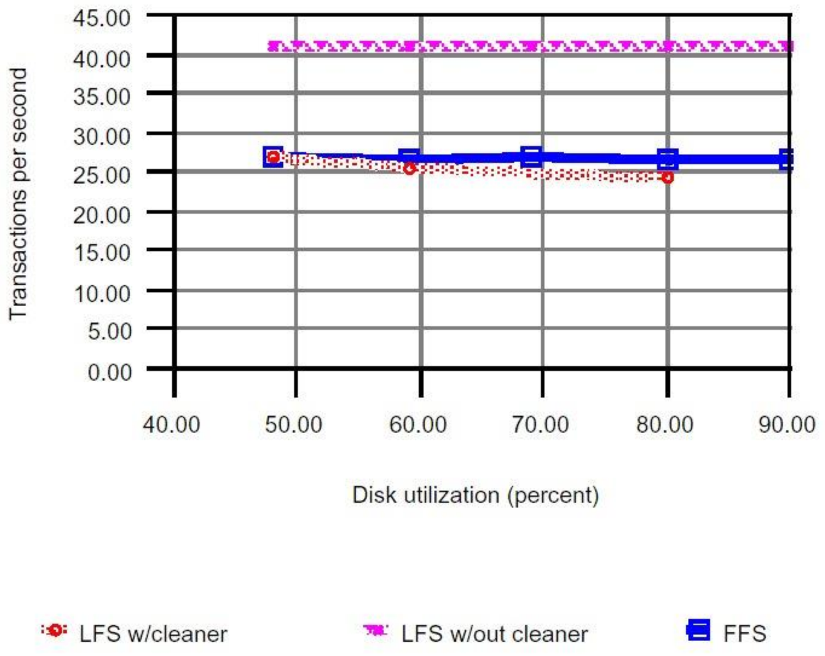


2.5
2.0
1.5
1.0
0.5
0.0
Throughput (in MB/sec)
1
16
256
4096
65536
File Size (in KB)
LFS
FFS-m8r0
FFS-m8r2



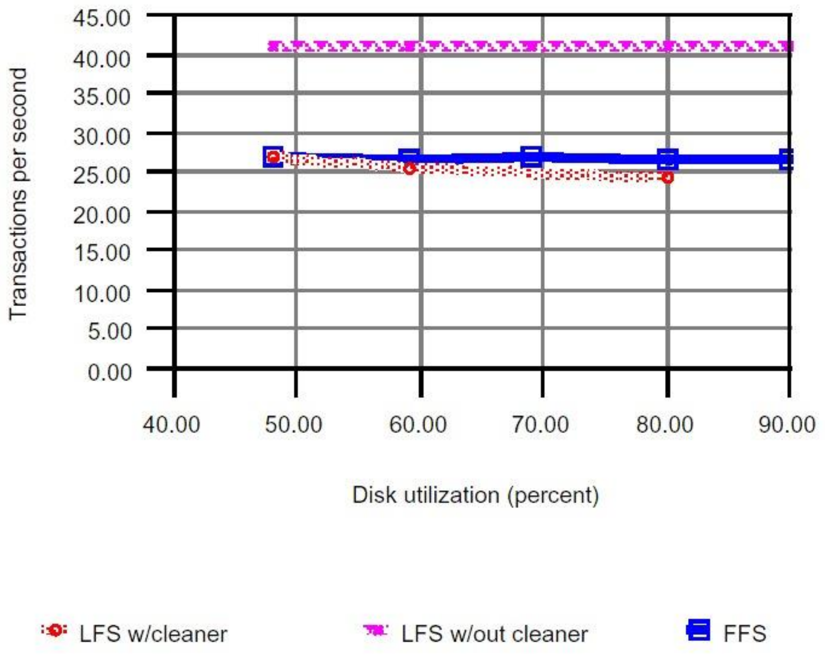
2.5
2.0
1.5
1.0
0.5
0.0
Throughput (in MB/sec)
1
16
256
4096
65536
File Size (in KB)
LFS
FFS-m8r0
FFS-m8r2

FFS v LFS



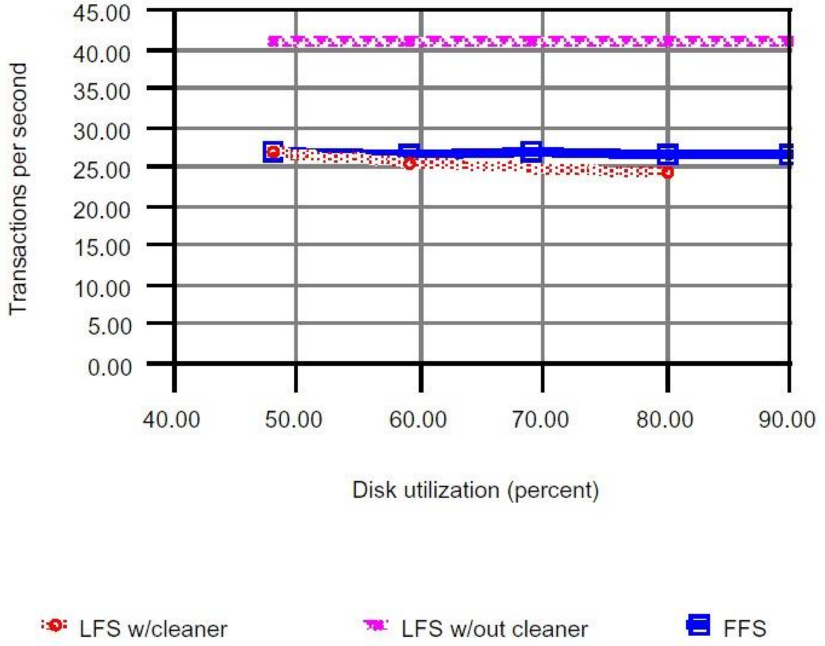
45.00
40.00
35.00
30.00
25.00
20.00
15.00
10.00
5.00
0.00
Transactions per second
40.00
50.00
60.00
70.00
80.00
90.00
Disk utilization (percent)
LFS w/cleaner
LFS w/out cleaner
FFS

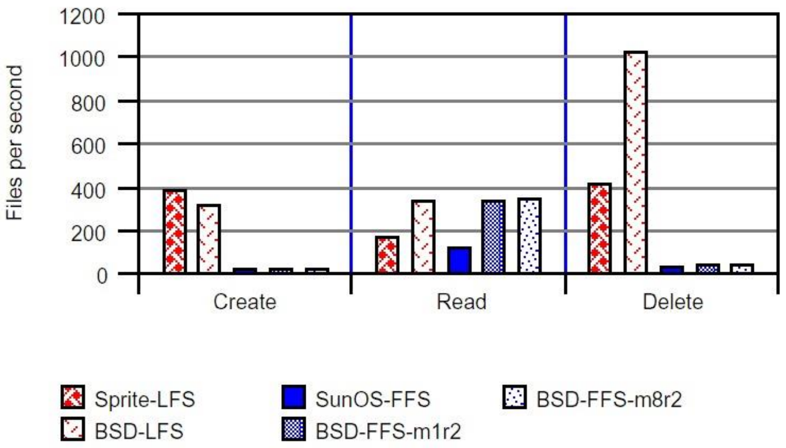
FFS v LFS

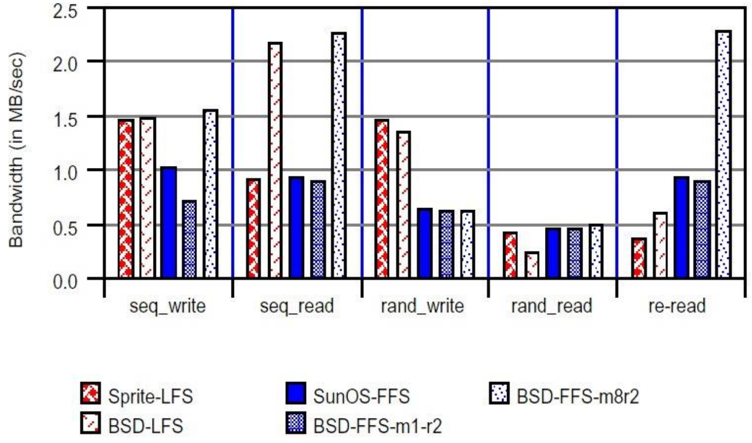


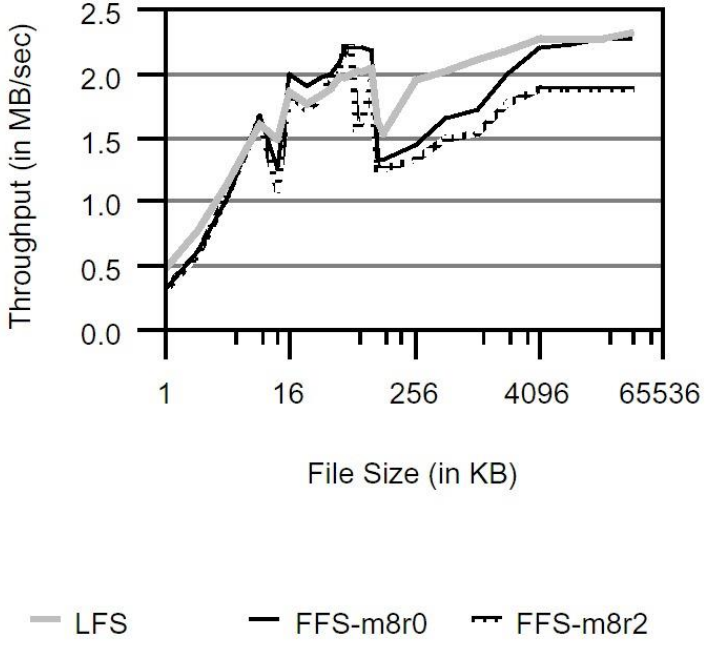
Transaction Processing:
TPC-B, database over FS

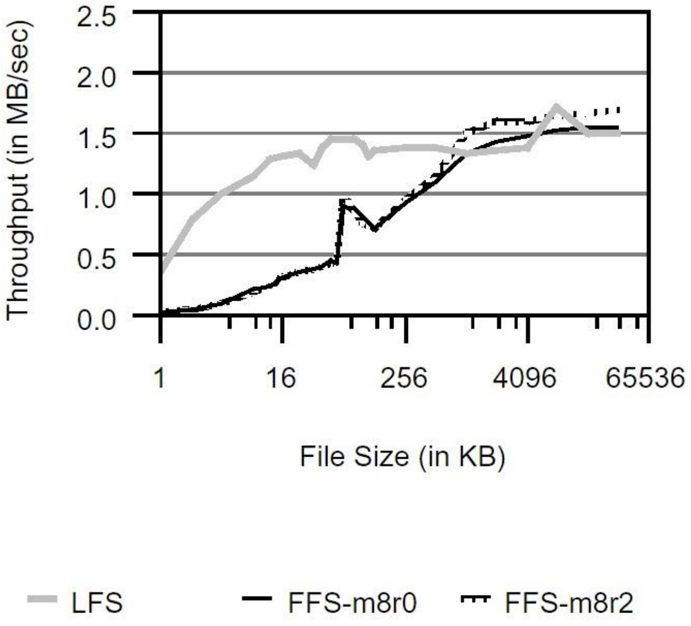
FFS v LFS


Transactions per second
45.00
40.00
35.00
30.00
25.00
20.00
15.00
10.00
5.00
0.00
40.00
50.00
60.00
70.00
80.00
90.00
Disk utilization (percent)
LFS w/cleaner
LFS w/out cleaner
FFS


Files per second
1200
1000
800
600
400
200
0
Create
Read
Delete
Sprite-LFS
BSD-LFS
SunOS-FFS
BSD-FFS-m1r2
BSD-FFS-m8r2

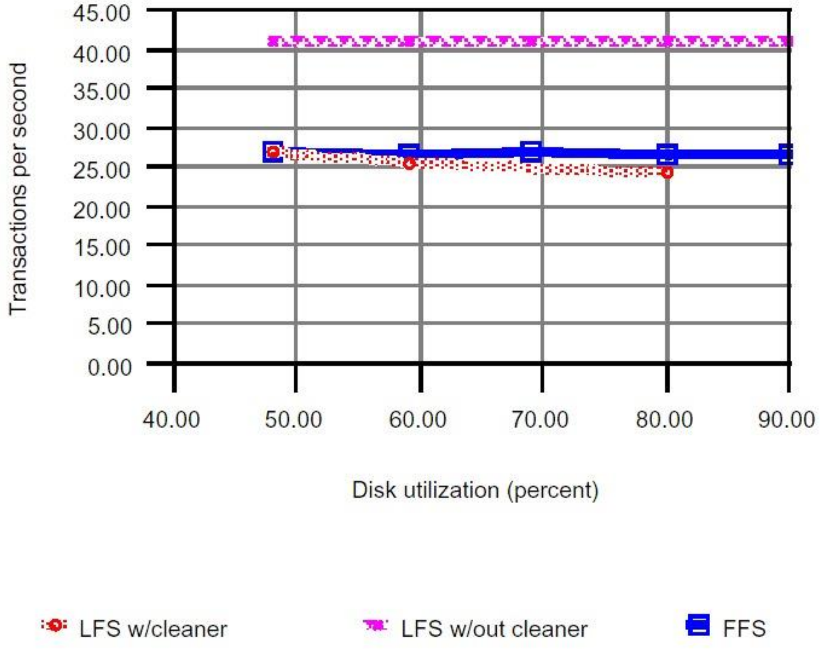

Bandwidth (in MB/sec)
2.5
2.0
1.5
1.0
0.5
0.0
seq_write
seq_read
rand_write
rand_read
re-read
Sprite-LFS
BSD-LFS
SunOS-FFS
BSD-FFS-m1-r2
BSD-FFS-m8r2

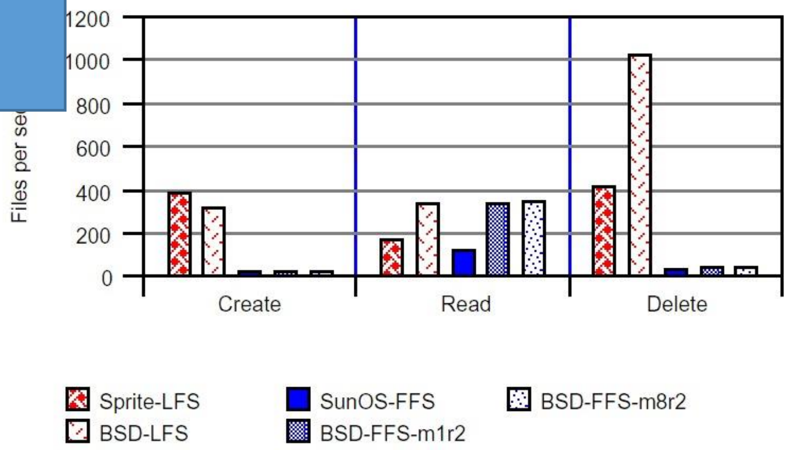

Throughput (in MB/sec)
2.5
2.0
1.5
1.0
0.5
0.0
1
16
256
4096
65536
File Size (in KB)
LFS
FFS-m8r0
FFS-m8r2

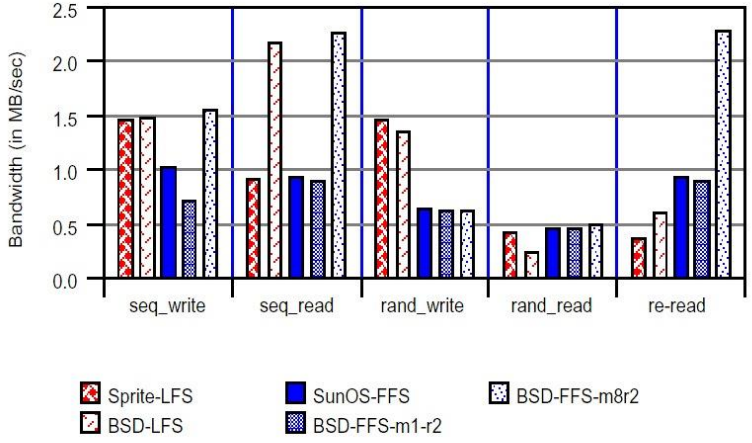

Throughput (in MB/sec)
2.5
2.0
1.5
1.0
0.5
0.0
1
16
256
4096
65536
File Size (in KB)
LFS
FFS-m8r0
FFS-m8r2

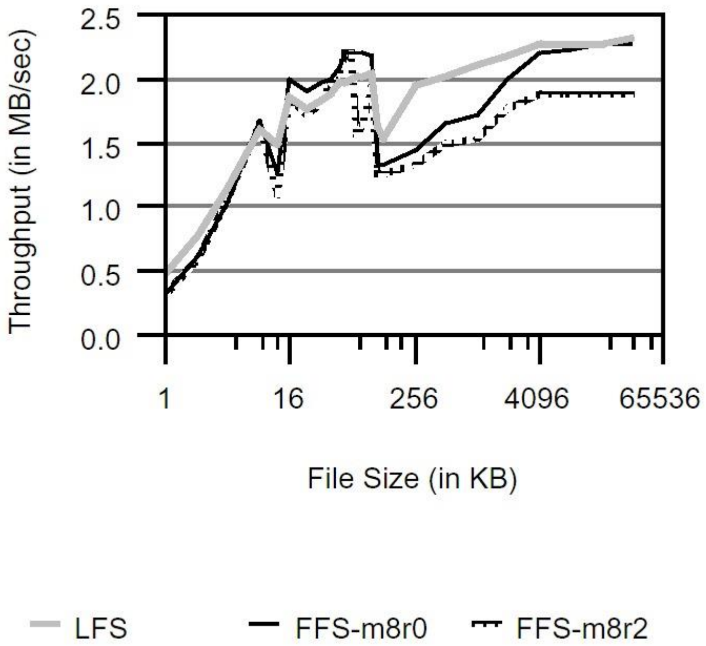
FFS v LFS

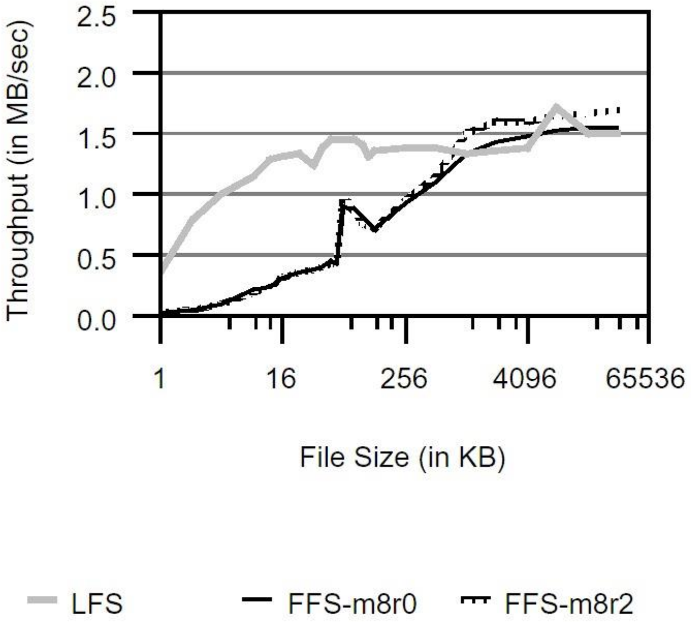
- LFS order of magnitude faster (small creates/del)
- LFS+FFS comparable on large file create (>= .5MB).
- LFS+FFS comparable on reads (<=64 KB).
- LFS read faster [64KB..4MB]
- LFS+FFS comparable on reads >= 4MB.
- LFS write superior (<=256KB)
- FFS write superior (>256KB)





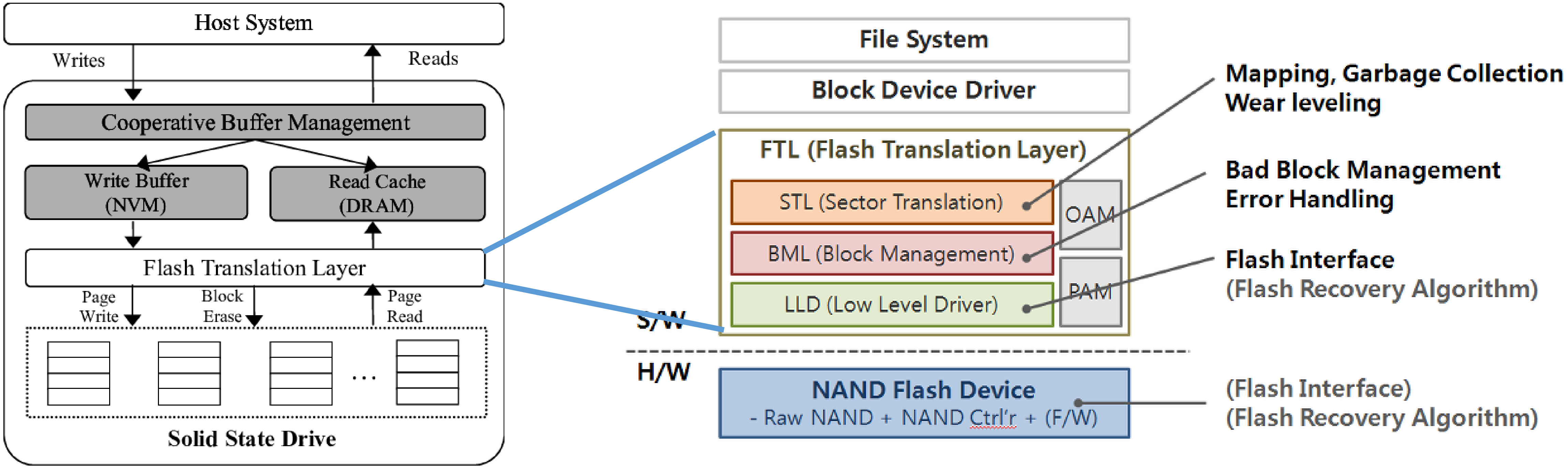






LFS Legacy: SSDs and FTLs

(and many other acronyms…)



Discussion:

- What workloads will be slower for LFS than FFS?
- Compare and contrast FFS and LFS from a mechanical sympathy perspective.
- FreeBSD and LFS deal with multiple allocation sizes (superpages/segments). How are the problems and solutions similar and/or different?