School of Computer Science & Engineering

COMP3891/9283 Extended Operating Systems

2026 T2 Week 05

Log-Structured File Systems

Gernot Heiser

Copyright Notice

Learning Outcomes

- An understanding of the performance of i-node-based files systems when writing small files.
- An understanding of how a log structured file system can improve performance and increase reliability via improved consistency guarantees without the need for file system checkers.
- An understanding of “cleaning” and how it might detract from performance.

Log-Structured File Systems

Motivation (1992):

Memory size is growing at a rapid rate

⇒ Growing proportion of file system reads will be satisfied by file system buffer cache

⇒ Writes will increasingly dominate reads

Mendel Rosenblum and John K. Ousterhout:
"The Design and Implementation of a Log-Structured File System"
ACM Transactions on Computer Systems 1992 10(1), 1992, pages 26-52

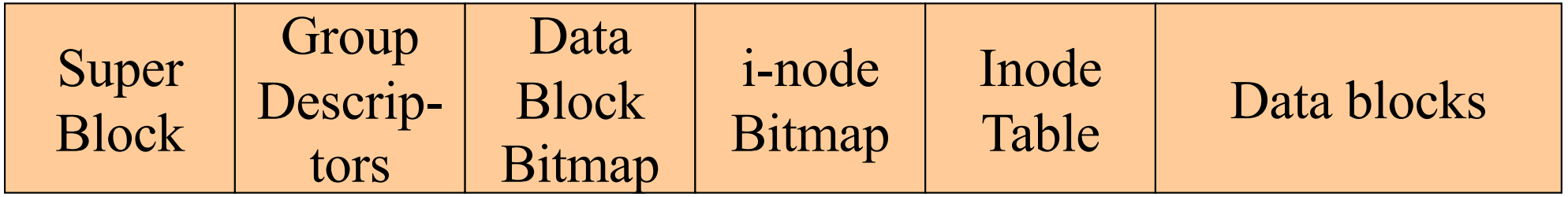
Motivating Observations

Metadata writes dominate cost!

- Creation/modification/deletion of small files form the majority of a typical workload (eg doing `make`)
- Workload poorly supported by traditional i-node-based file system (e.g. BSD FFS, ext2fs)
- Example: create 1KiB file results in:
  - 2 writes to the file i-node
  - 1 write to data block
  - 1 write to directory data block
  - 1 write to directory i-node

5 writes scattered within group!

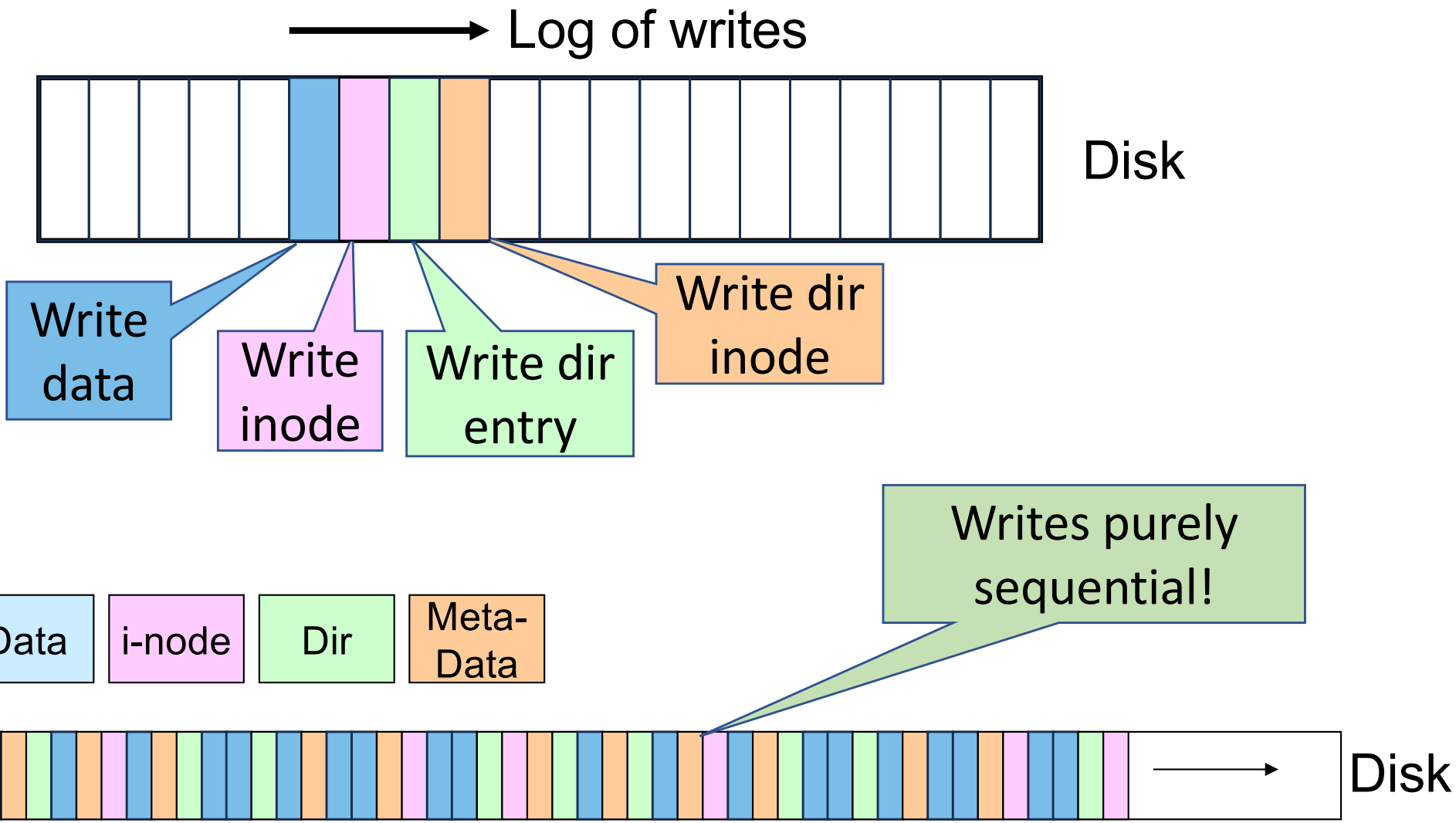
Syncronous writes for consistency!



Motivating Observations

- Consistency checking required for ungraceful shutdown due to potential for sequence of updates to have only partially completed.
- File system consistency checkers are time consuming for large disks.
- Unsatisfactory boot times where consistency checking is required.

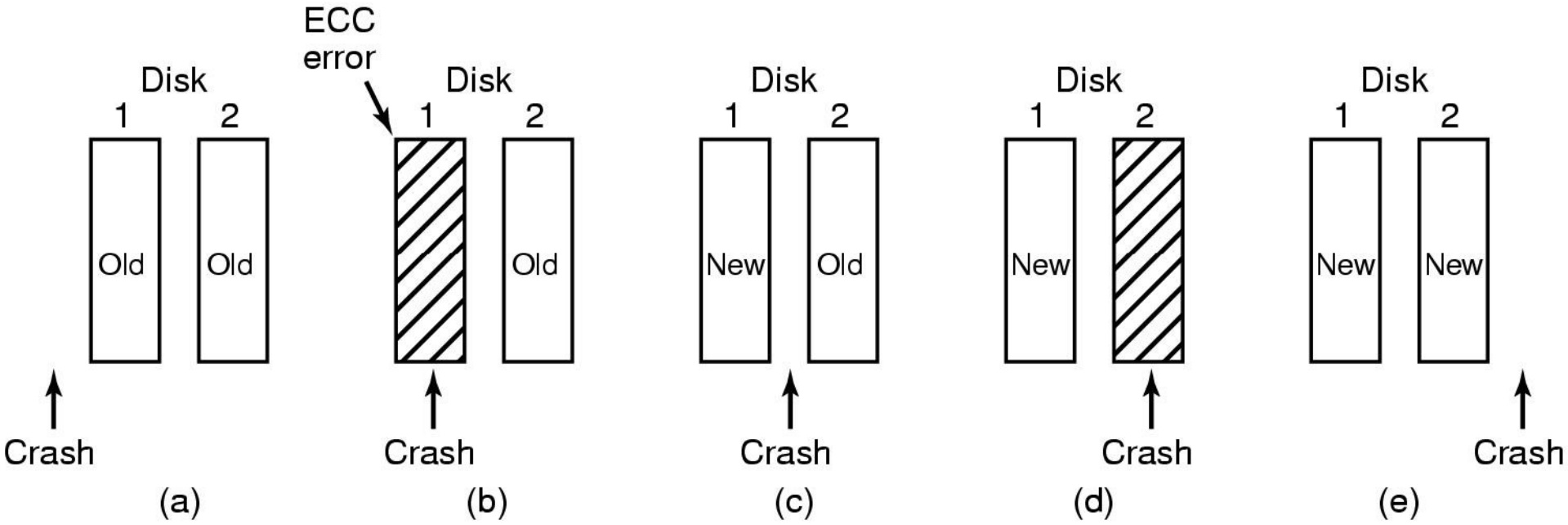
Idea: Buffer Updates, Write Sequentially



How to locate i-nodes scattered on disk?

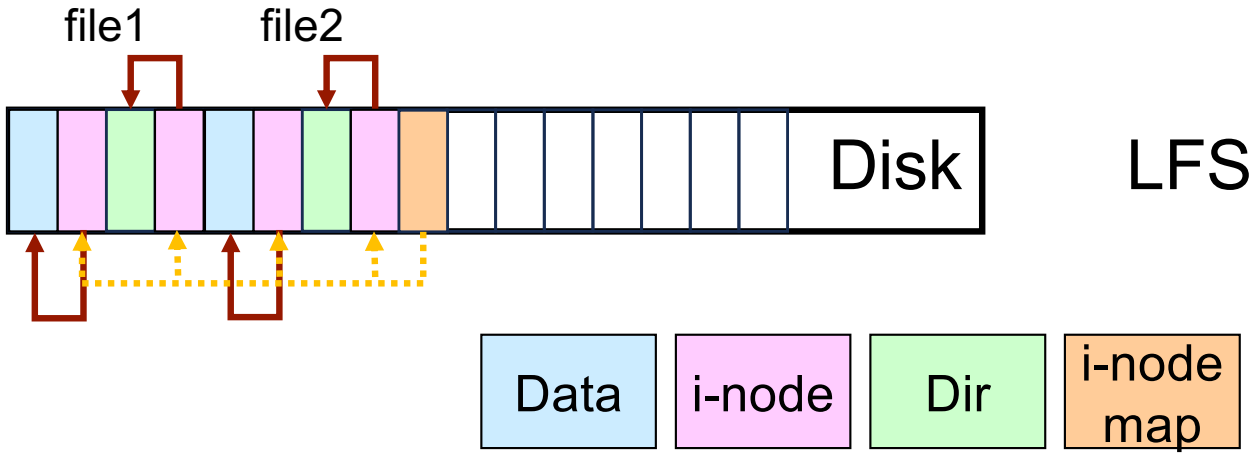
- Keep a map of i-node locations
  - i-node map is also “logged”
  - Assumption is i-node map is heavily cached and rarely results in extra disk accesses
- How find i-node map?
  - Alternate between two fixed locations pointing to the inode map
    - Allows recovery if crash during updating map.

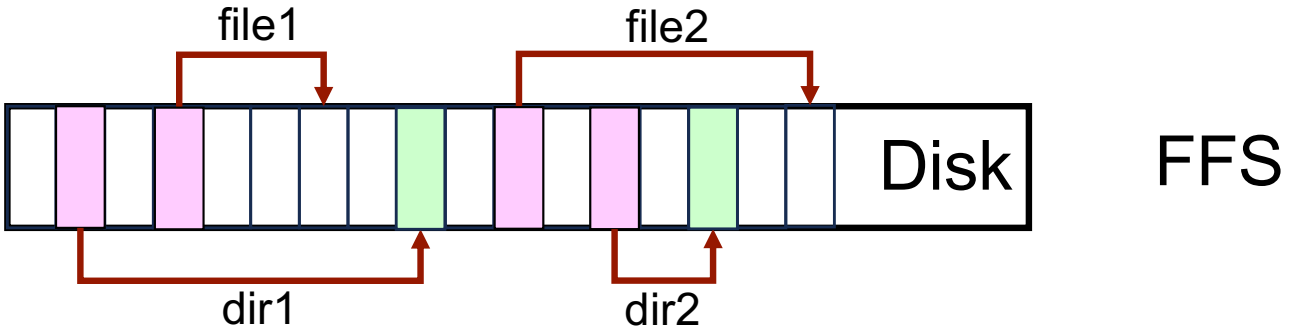
Implementing Stable Storage: Challis



- Use two disks to implement stable storage
  - Problem is when a write (update) corrupts old version, without completing write of new version
  - Solution: Write to one disk first, then to second after completion of first
  - Time stamp tells which is newer
- Can do the same with disk blocks.

LFS versus FFS: Creating Two Small Files





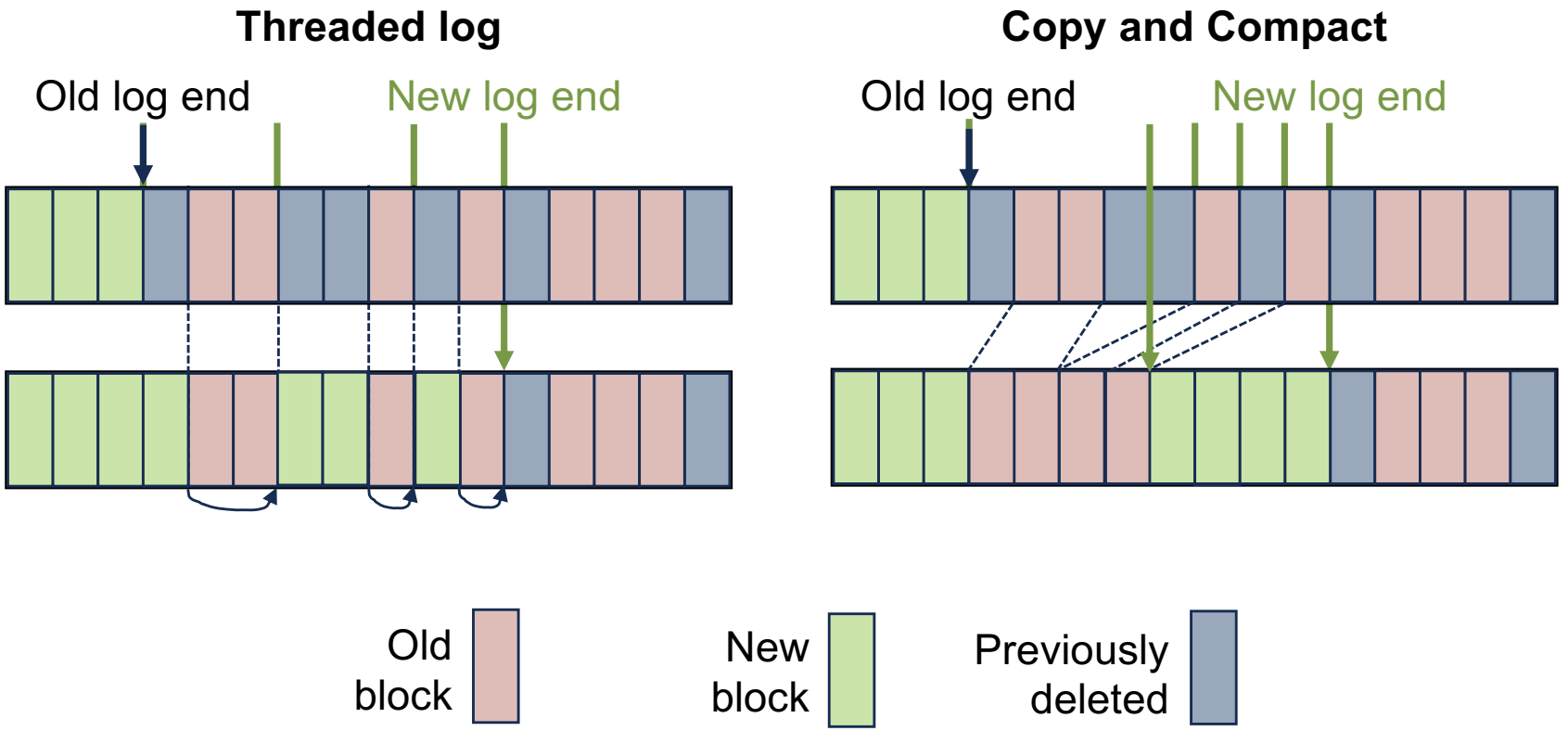
Issue: Disks are Finite in Size

Need file system “cleaner” running in background

- Recovers blocks that are no longer in use by consulting current inode map
  - Identifies unreachable blocks
- Compacts remaining blocks on disk to form contiguous segments for improved write performance

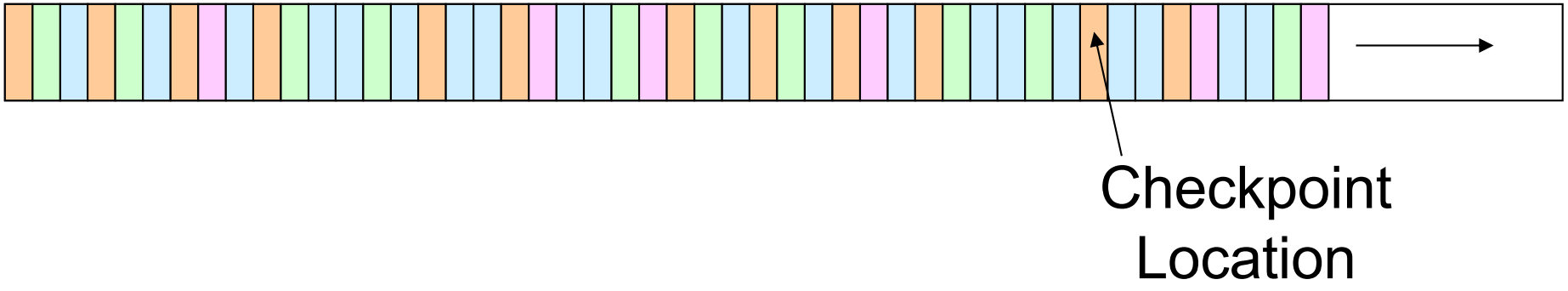
Cleaner

Uses a combination of threaded log and copy and compact



Recovery

- File system is check-pointed regularly which saves
  - A pointer to the current head of the log
  - The current i-node-map blocks
- On recovery, simply restart from previous checkpoint.
  - Can scan forward in log and recover any updates written after previous checkpoint
  - Writes update the log (no update in place), so previous checkpoint is always consistent



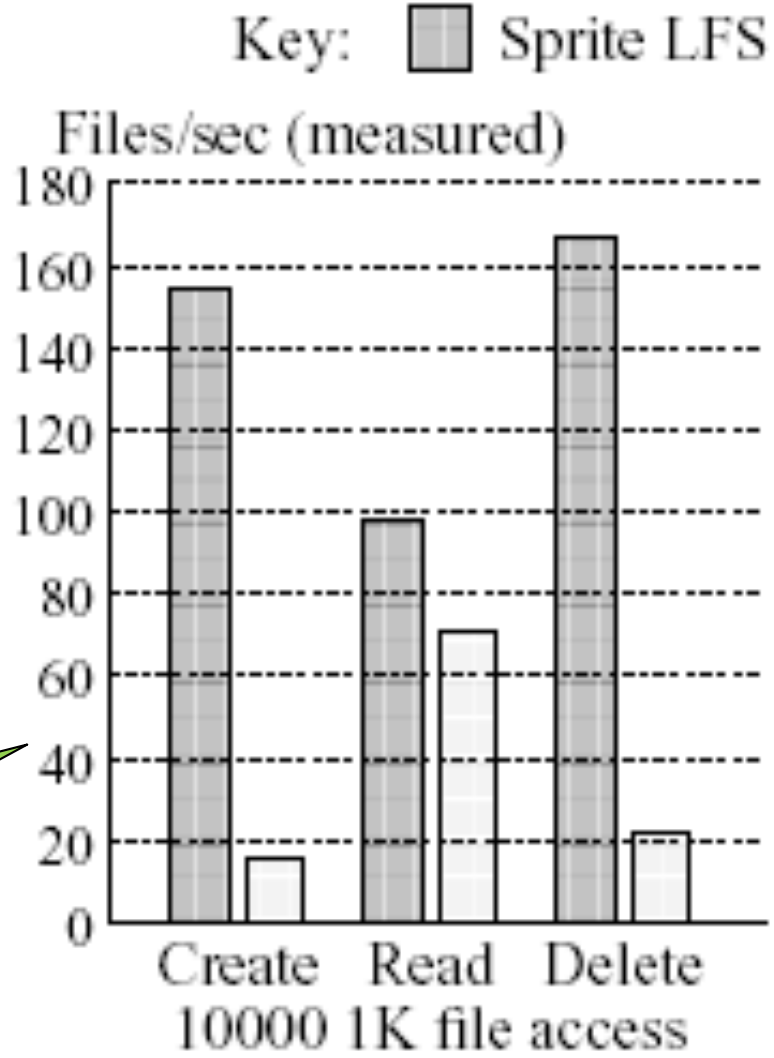
Reliability

- Updated data is written to the log, not in place.
- Reduces chance of corrupting existing data.
  - Old data in log always safe.
  - Crashes only affect recent data (written since last checkpoint)
    - As opposed to updating (and corrupting) the root directory.

Performance: Small-File Create-Read-Delete

Comparing LFS and SunOS FS

- Create 10,000 1-KiB files
- Read them (in order)
- Delete them



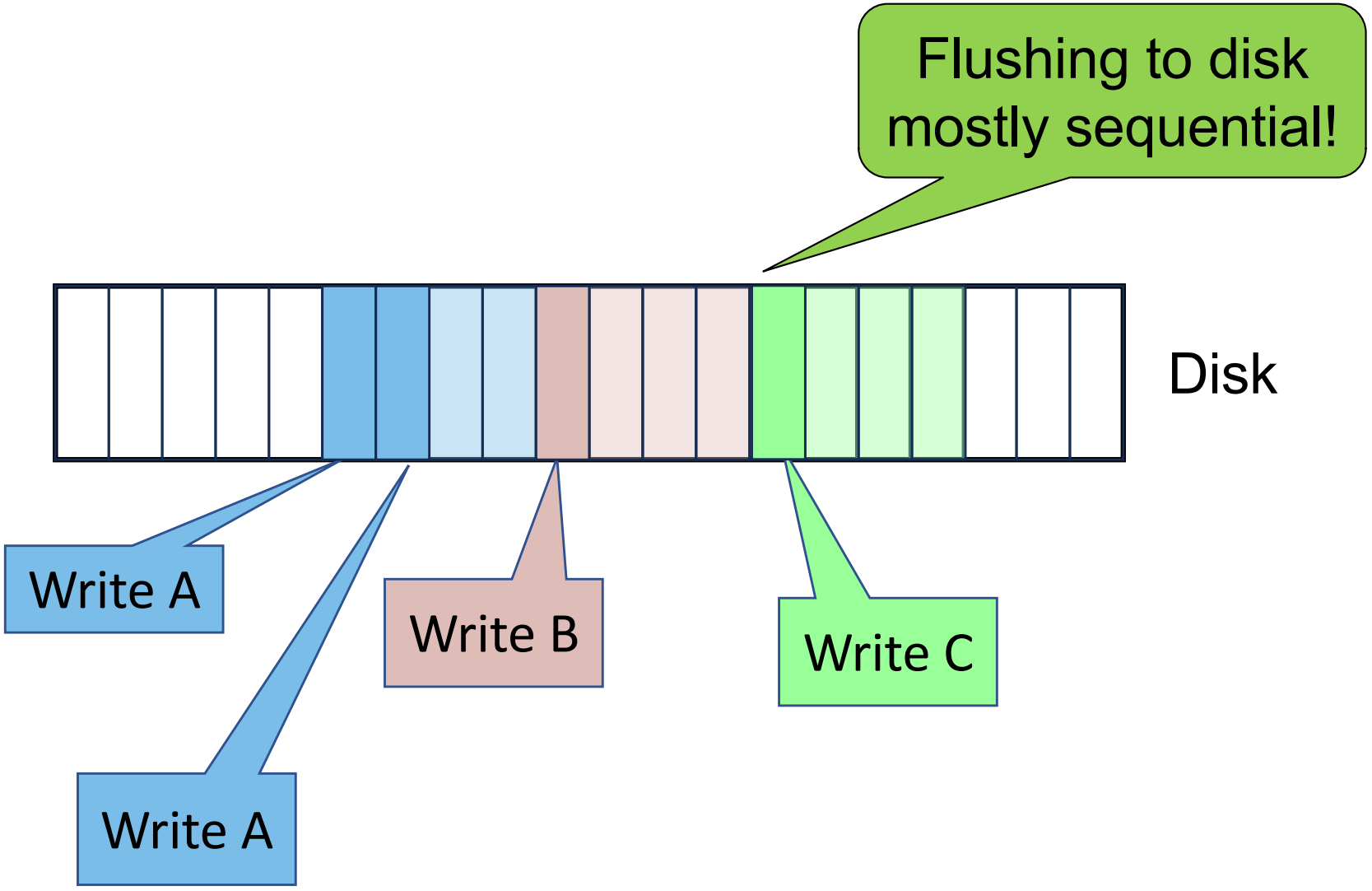
Order-of-magnitude performance improvement for small writes

LFS a clear winner?

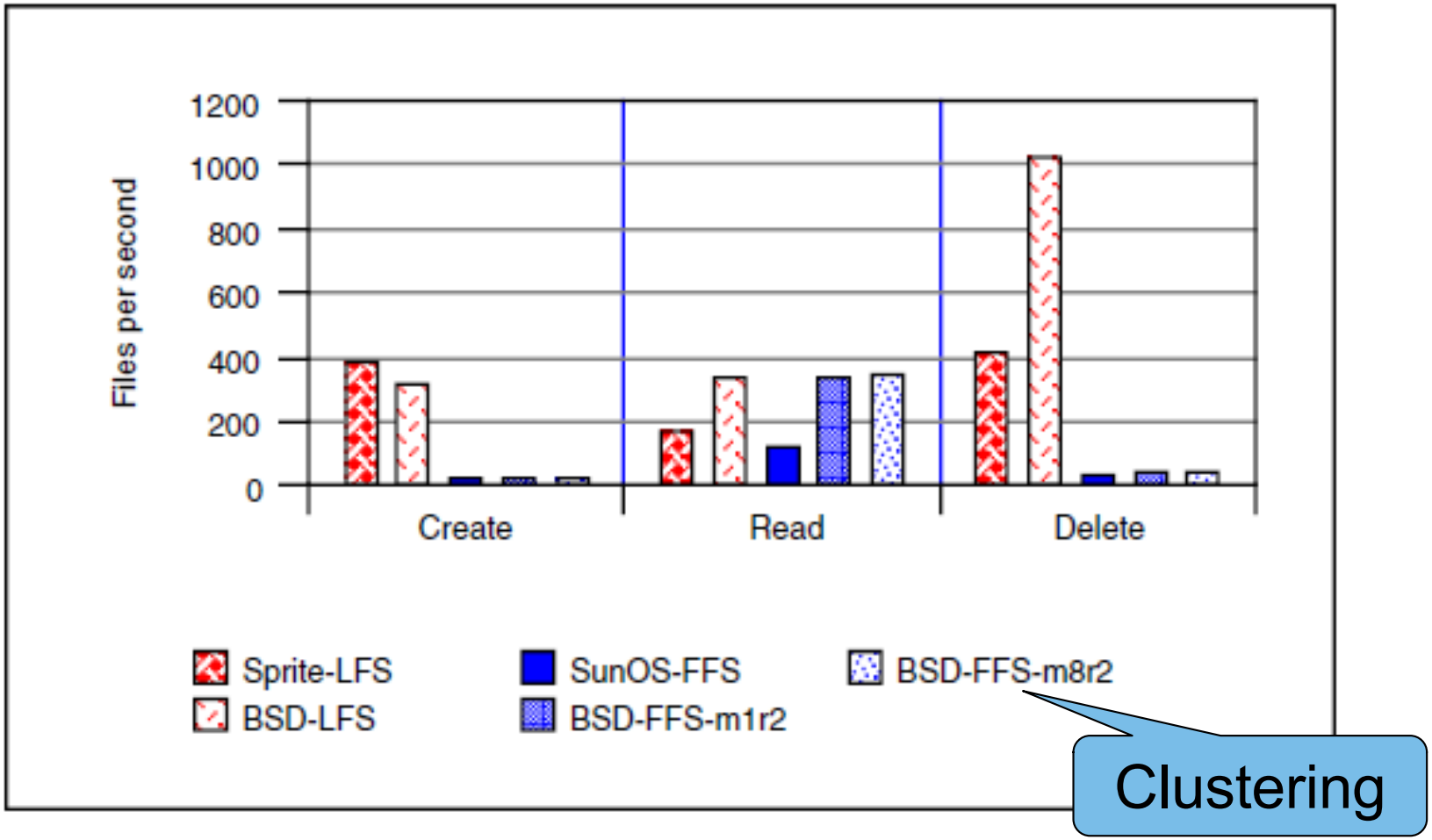
Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara Mcmains and Venkata Padmanabhan:
"File System Logging Versus Clustering: A Performance Comparison". USENIX ATC. 1995.

- Authors involved in BSD-LFS
  - log structured file system for BSD 4.4
  - enable direct comparison with BSD-FFS
    - including recent clustering additions
- Include a critical examination of cleaning overhead

Clustering: Pre-allocate Sequential Blocks



Original Sprite-LFS Benchmarks: Small File

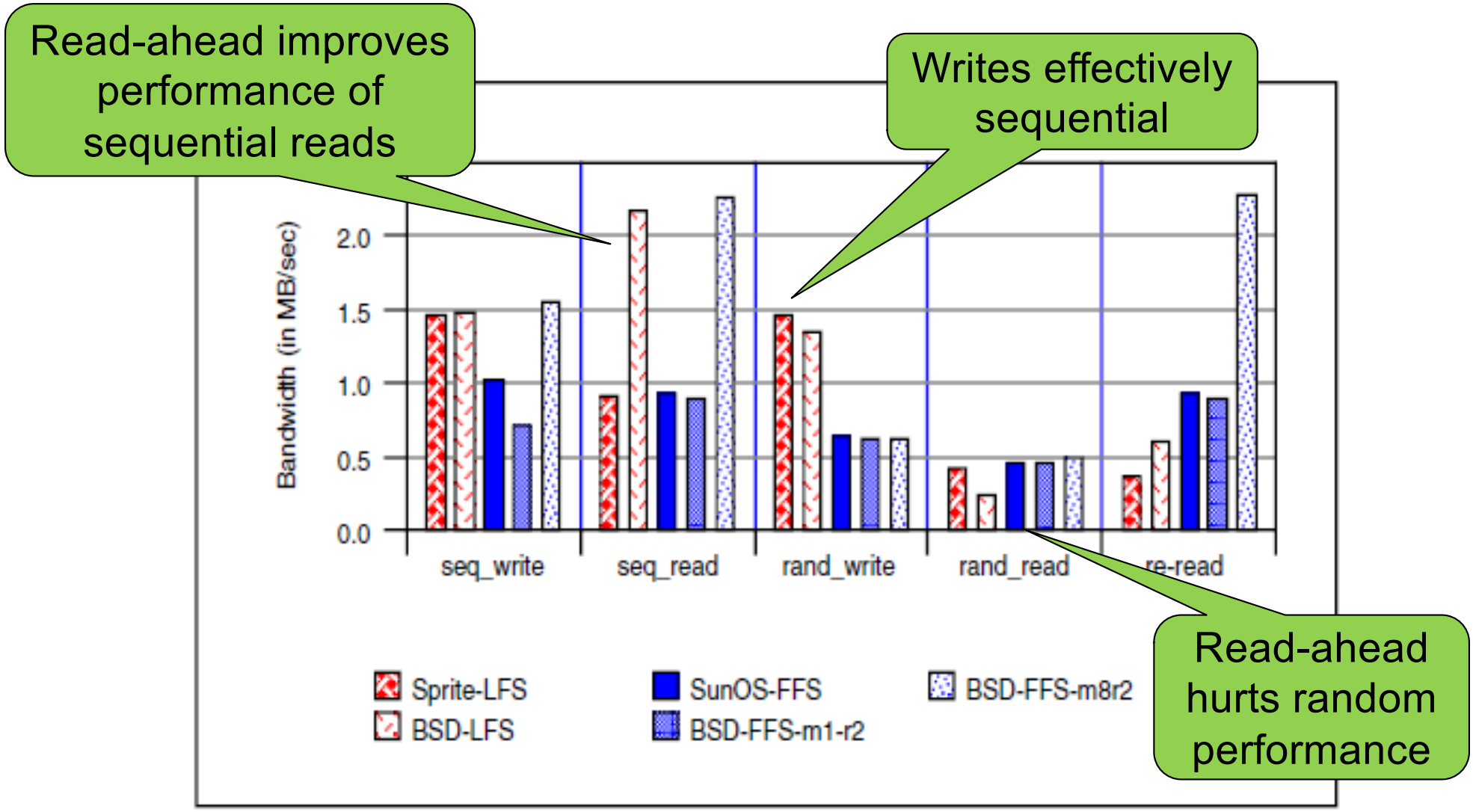


Large File Performance: 100 MiB file

Benchmarks:

- Create the file by sequentially writing 8 KiB units.
- Read the file sequentially in 8 KiB units.
- Write 100 KiB of data randomly in 8 KiB units.
- Read 100 KiB of data randomly in 8 KiB units.
- Re-read the file sequentially in 8 KiB units

Large File Performance: 100 MiB file

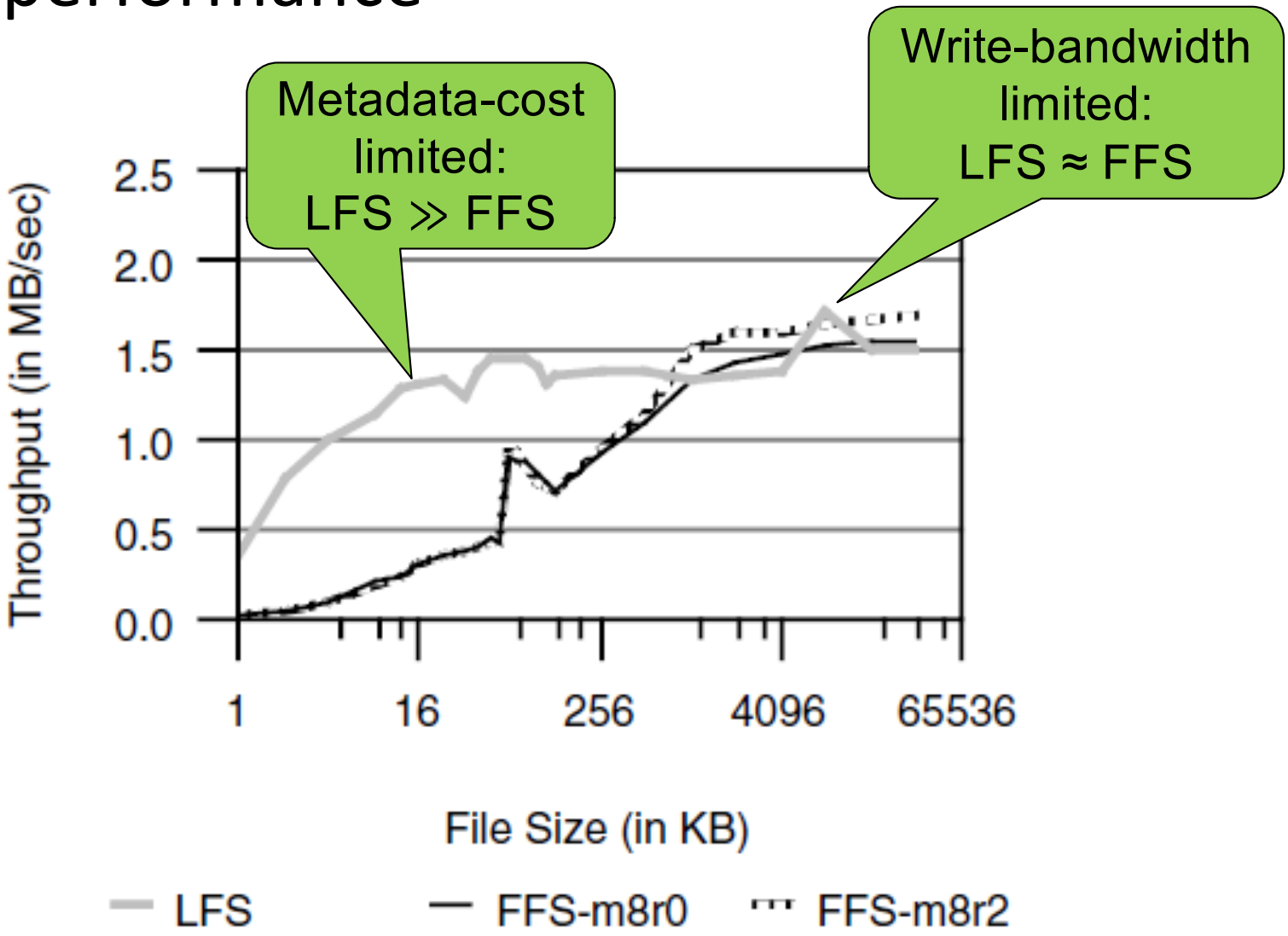


Observations

Read-ahead helps in BSD sequential case, but hurts in random.

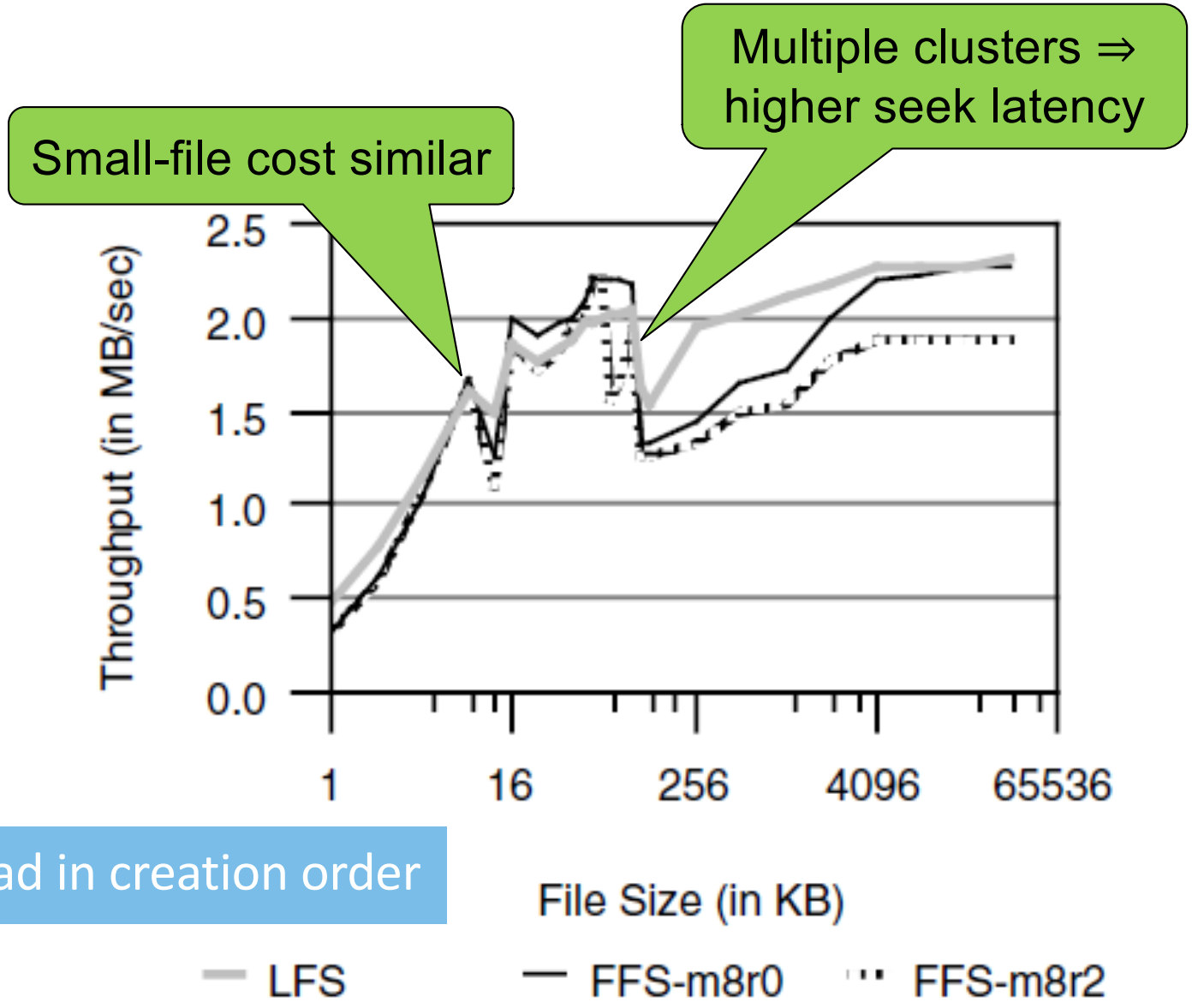
- Read ahead algorithm is triggered on successful read-ahead on sequential, turned off on a miss. Worst case for 8KiB reads with 4KiB blocks.

Create performance



files = 32MiB/file size

Read Performance



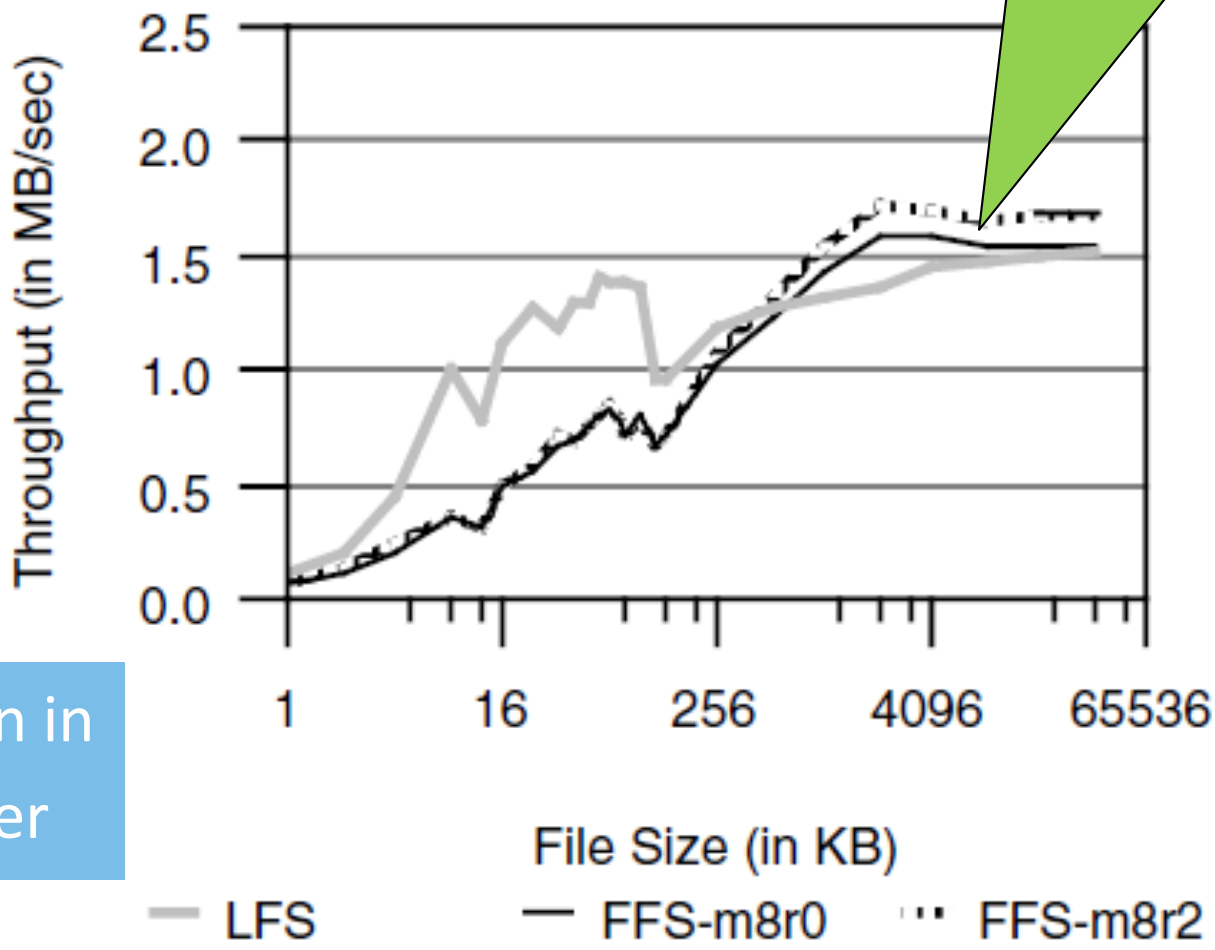
Files read in creation order

Observations

- For files of less than 64 KiB, performance is comparable in all the file systems.
- At 64 KiB, files are composed of multiple clusters and seek penalties rise.
- In the range between 64 KiB and 2 MiB, LFS performance dominates
  - because FFS is seeking between cylinder groups to distribute data evenly.

Write Performance

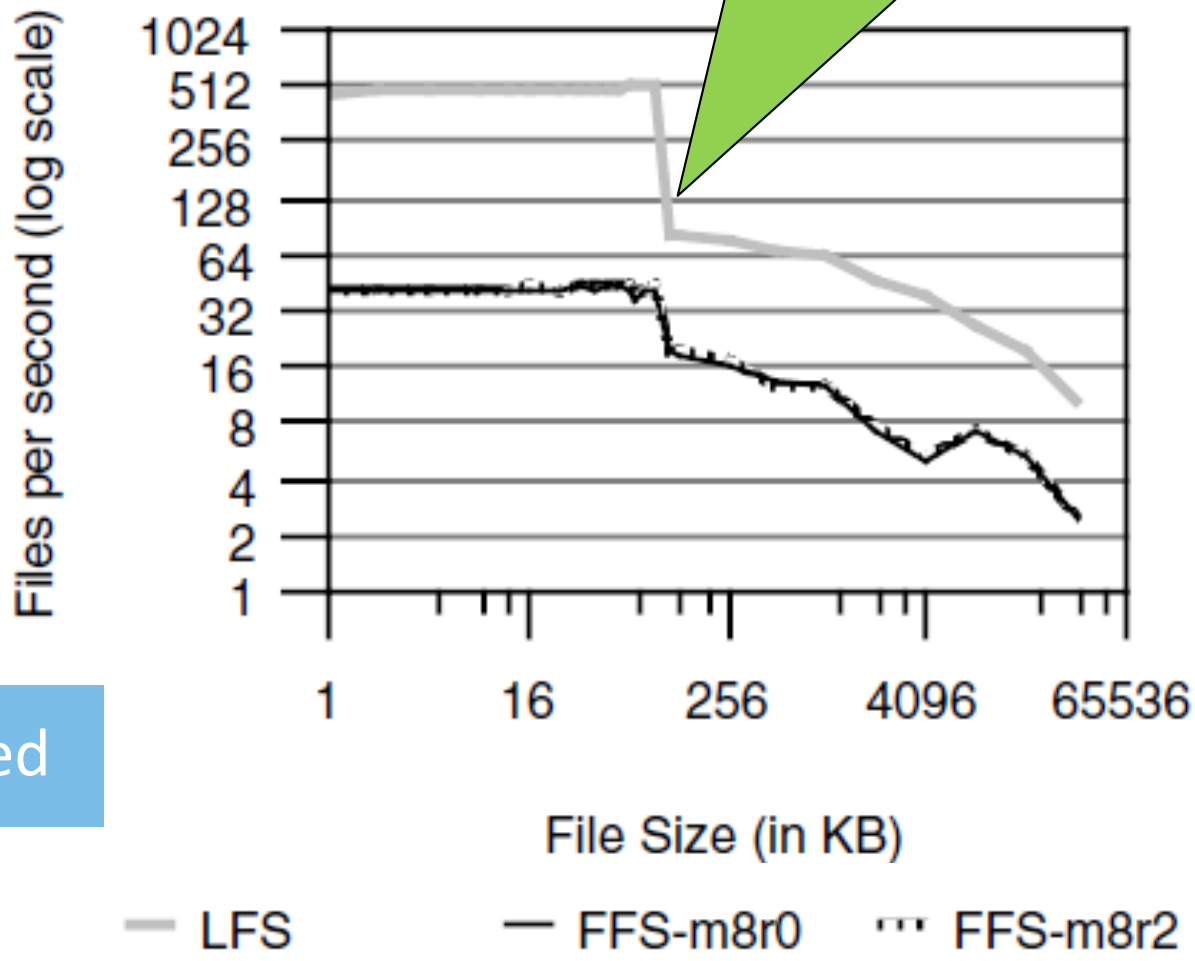
FFS: no synchronous writes
LFS must invalidate
dead blocks



Files re-written in creation order

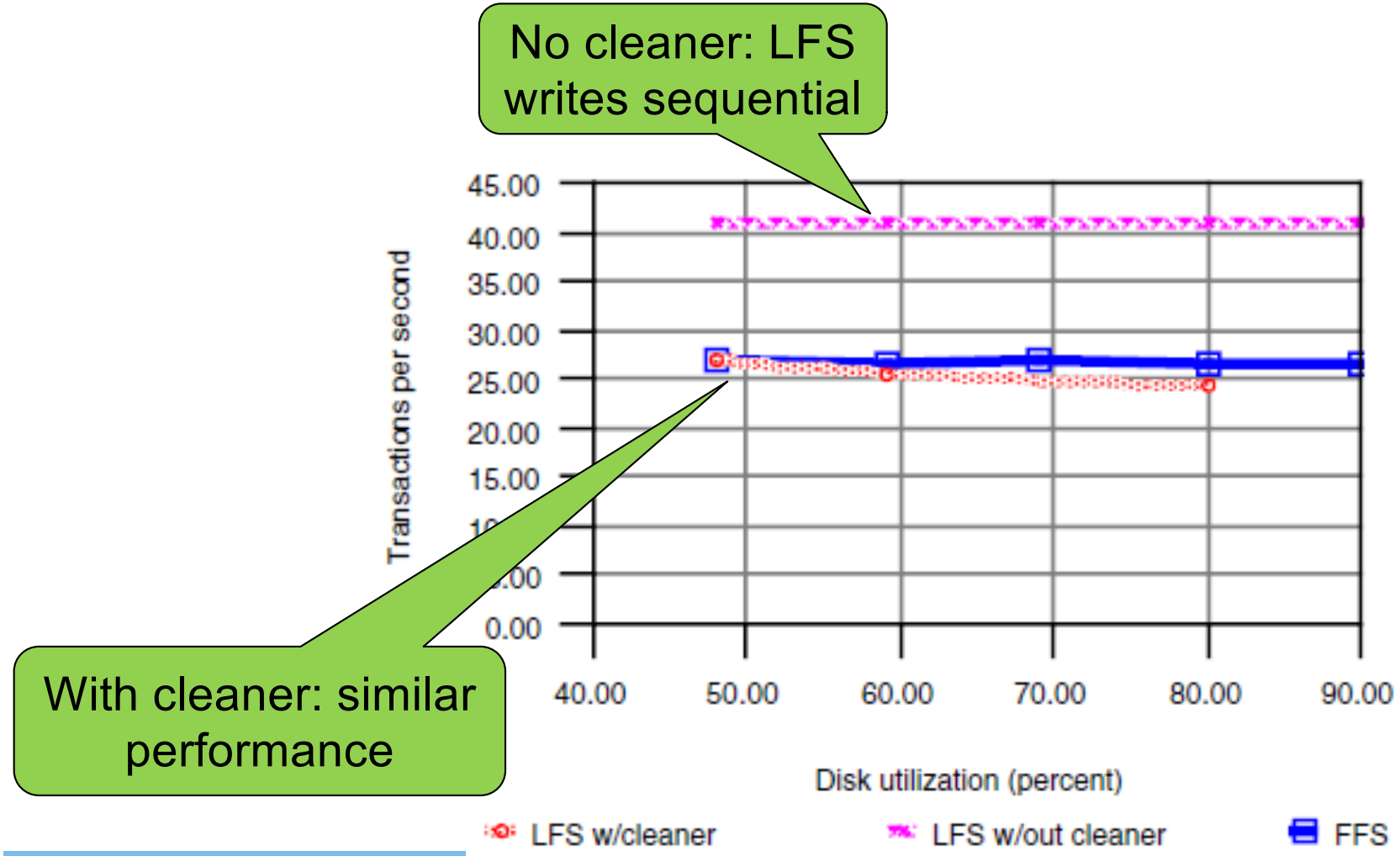
Delete Performance

Performance dominated by meta-data updates



All files deleted

Transaction processing performance.



Random access

Not Clear Winner!

- No cleaner (LFS), fresh (unfragmented) file system: each dominates in some regions:
  - Small file creates and deletes: LFS ≫ FFS
  - Large-file (>0.5 MiB) creates: LFS ≈ FFS
  - Read: LFS > FFS for 64 KiB – 4 MiB
  - Write: LFS > FFS for ≤ 256 KiB; FFS > LFS for ≥ 256 KiB
- Cleaning overhead significantly degrades LFS performance
- Fragmentation degrades FFS performance over time

Take-aways

- When meta-data operations are the bottleneck, LFS wins.
- Cleaning overhead degrades LFS performance significantly as utilisation rises.
- LFS ideas live on in more recent “snapshot”-base file systems.
  - E.g., ZFS and BTRFS

Journaling file systems

- Hybrid of
  - i-node based file system and
  - Log-structured file system (journal)
- Two variations
  - log only meta-data to journal (default)
  - log-all to journal
- Need to write twice: copy from journal to i-node based files)
- Example – ext3
  - Main advantage is guaranteed meta-data consistency