Advanced file systems: LFS and Soft Updates

Ken Birman
(based on slides by Ben Atkin)

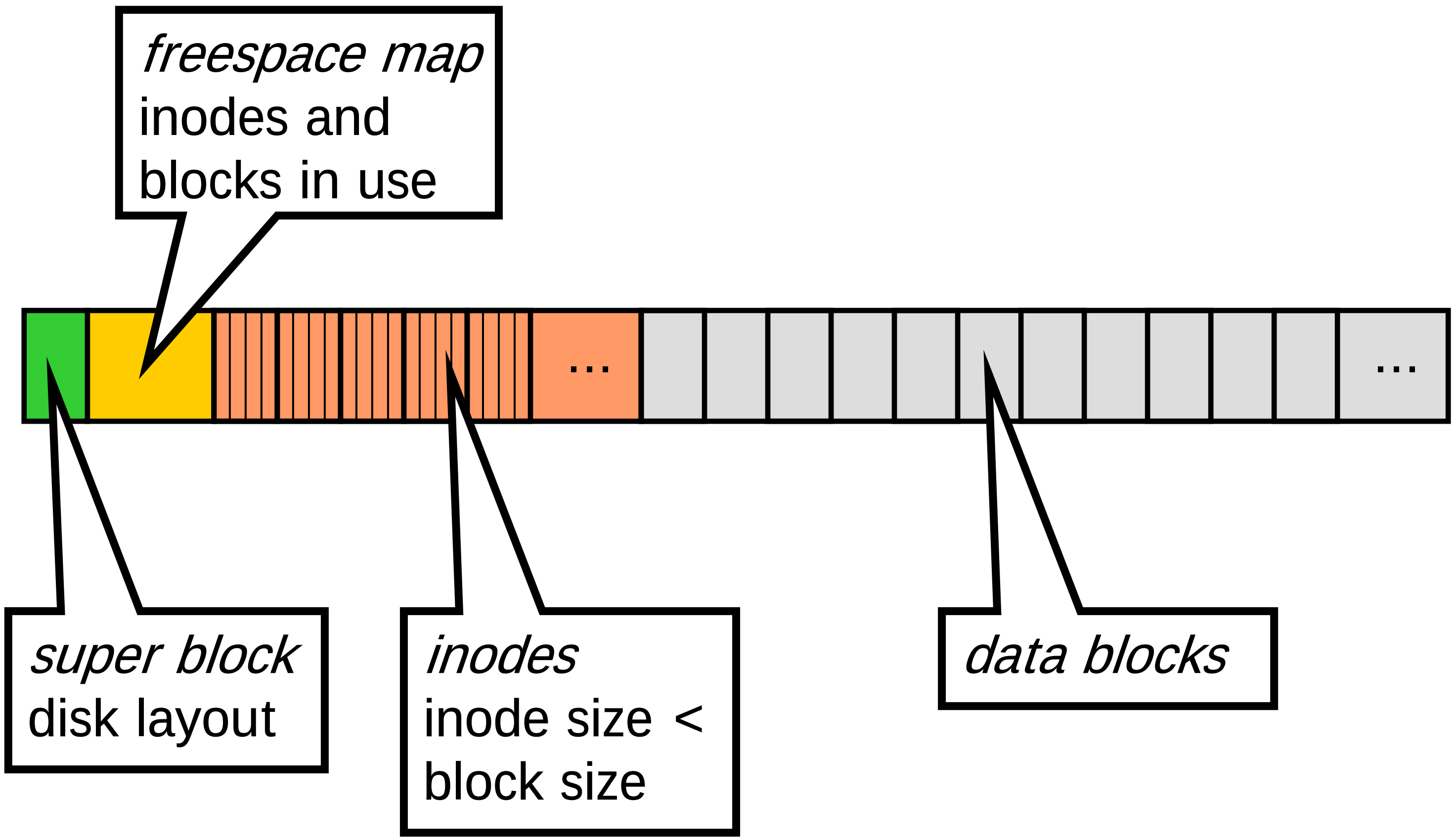
Overview of talk

- Unix Fast File System
- Log-Structured System
- Soft Updates
- Conclusions

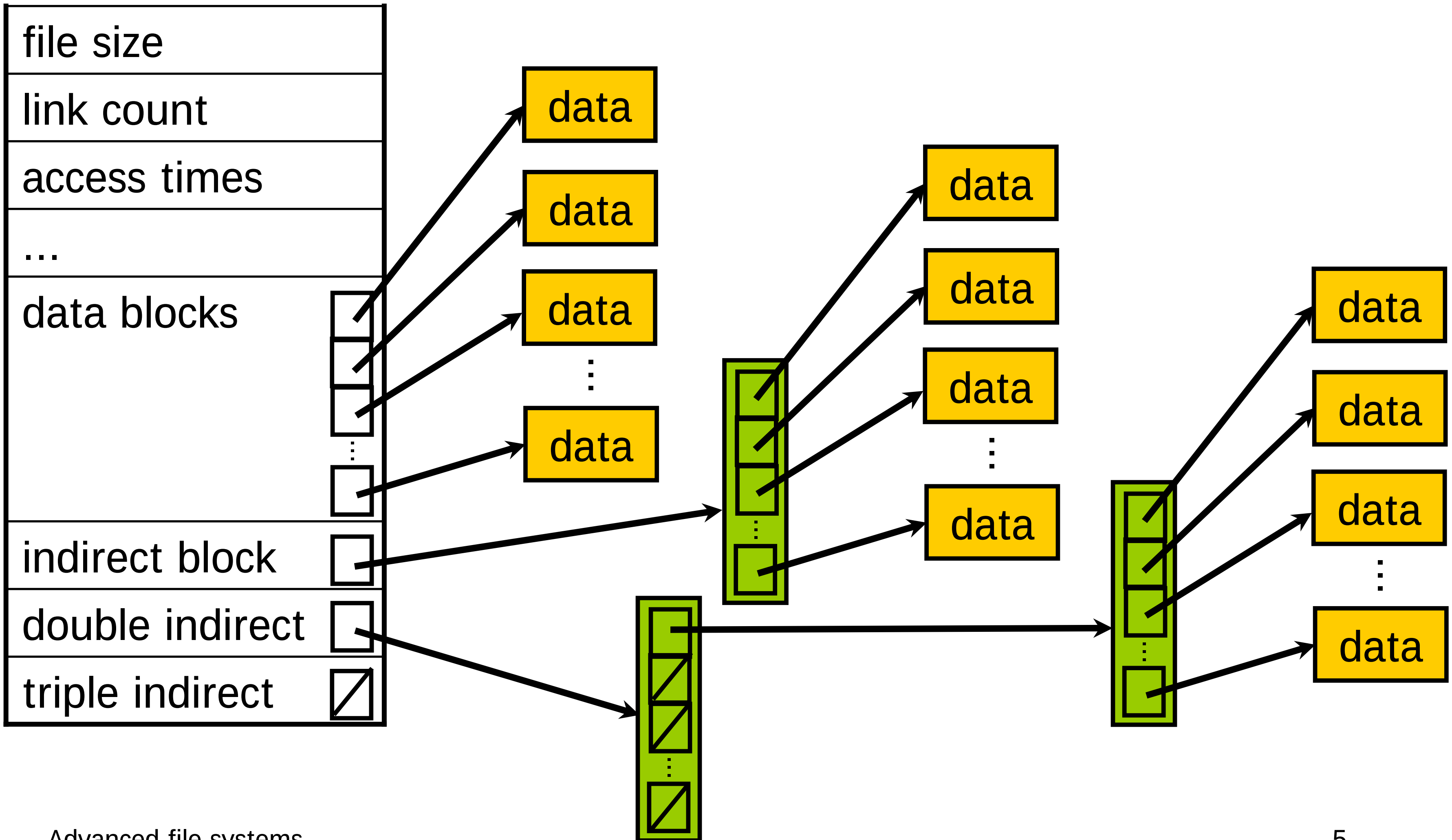
The Unix Fast File System

- Berkeley Unix (4.2BSD)
- Low-level index nodes (inodes) correspond to files
- Reduces seek times by better placement of file blocks
  - Tracks grouped into cylinders
  - Inodes and data blocks grouped together
  - Fragmentation can still affect performance

File system on disk



File representation



Inodes and directories

- Inode doesn't contain a file name
- Directories map files to inodes
  - Inode can be in multiple directories
  - Low-level file system doesn't distinguish files and directories
  - Separate system calls for directory operations

FFS implementation

- Most operations do multiple disk writes
  - File write: update block, inode modify time
  - Create: write freespace map, write inode, write directory entry
- Write-back cache improves performance
  - Benefits due to high write locality
  - Disk writes must be a whole block
  - Syncer process flushes writes every 30s

FFS crash recovery

- Asynchronous writes are lost in a crash
  - **Fsync** system call flushes dirty data
  - Incomplete metadata operations can cause disk corruption (order is important)
- FFS metadata writes are synchronous
  - Large potential decrease in performance
  - Some OSes cut corners

After the crash

- **Fsck** file system consistency check
  - Reconstructs freespace maps
  - Checks inode link counts, file sizes
- Very time consuming
  - Has to scan all directories and inodes

Overview of talk

- Unix Fast File System
- Log-Structured System
- Soft Updates
- Comparison and conclusions

The Log-Structured File System

- CPU speed increases faster than disk speed
- Caching improves read performance
- Little improvement in write performance
  - Synchronous writes to metadata
  - Metadata access dominates for small files
  - e.g. Five seeks and I/Os to create a file

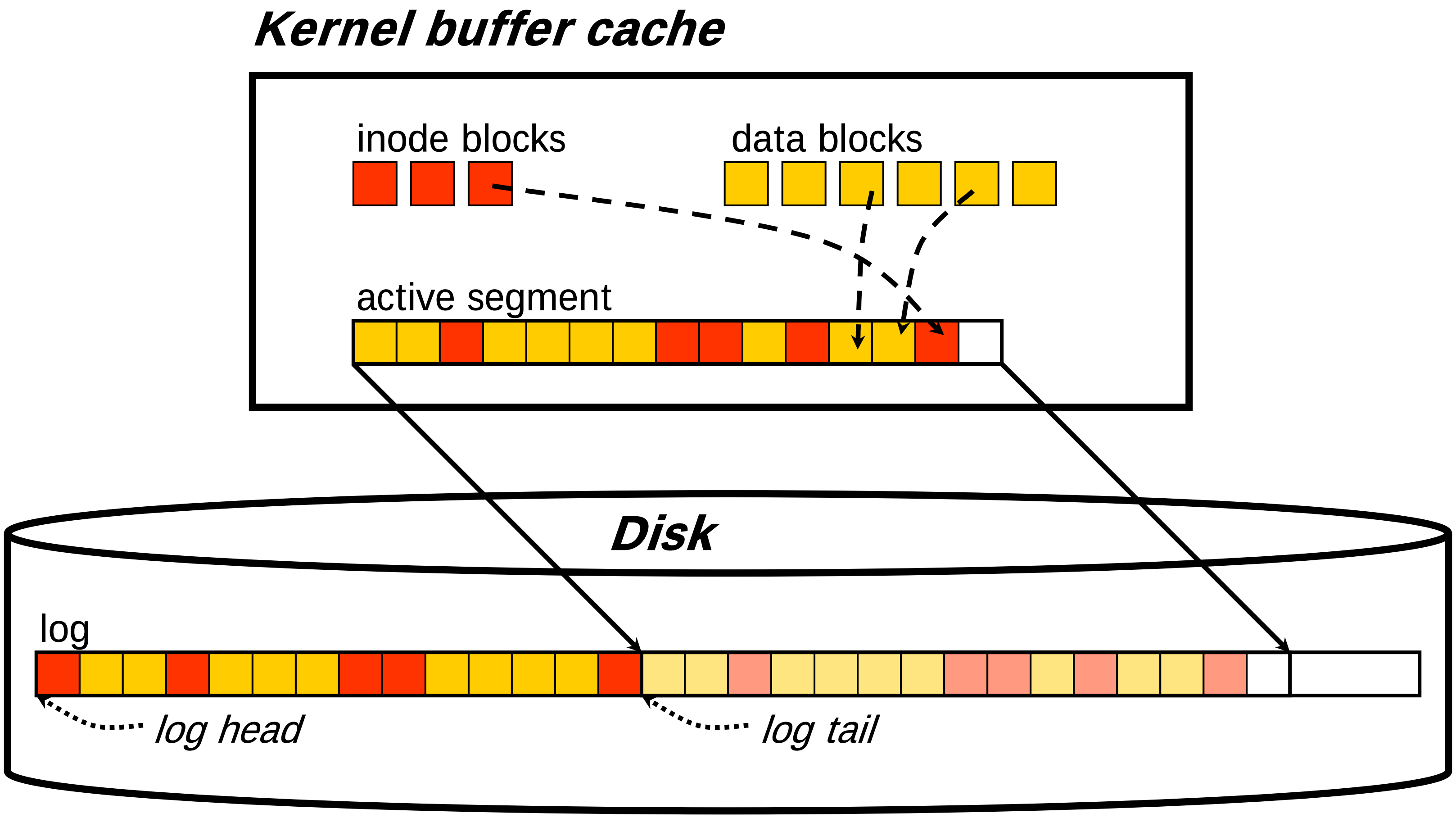
LFS design

- Increases write throughput from 5-10% of disk to 70%
  - Removes synchronous writes
  - Reduces long seeks
- Improves over FFS
  - "Not more complicated"
  - Outperforms FFS except for one case

LFS in a nutshell

- Boost write throughput by writing all changes to disk contiguously
  - Disk as an array of blocks, append at end
  - Write data, indirect blocks, inodes together
  - No need for a free block map
- Writes are written in *segments*
  - ~1MB of continuous disk blocks
  - Accumulated in cache and flushed at once

Log operation



Locating inodes

- Positions of data blocks and inodes change on each write
  - Write out inode, indirect blocks too!
- Maintain an inode map
  - Compact enough to fit in main memory
  - Written to disk periodically at *checkpoints*

Cleaning the log

- Log is infinite, but disk is finite
  - Reuse the old parts of the log
- Clean old segments to recover space
  - Writes to disk create holes
  - Segments ranked by "liveness", age
  - Segment cleaner "runs in background"
- Group slowly-changing blocks together
  - Copy to new segment or "thread" into old

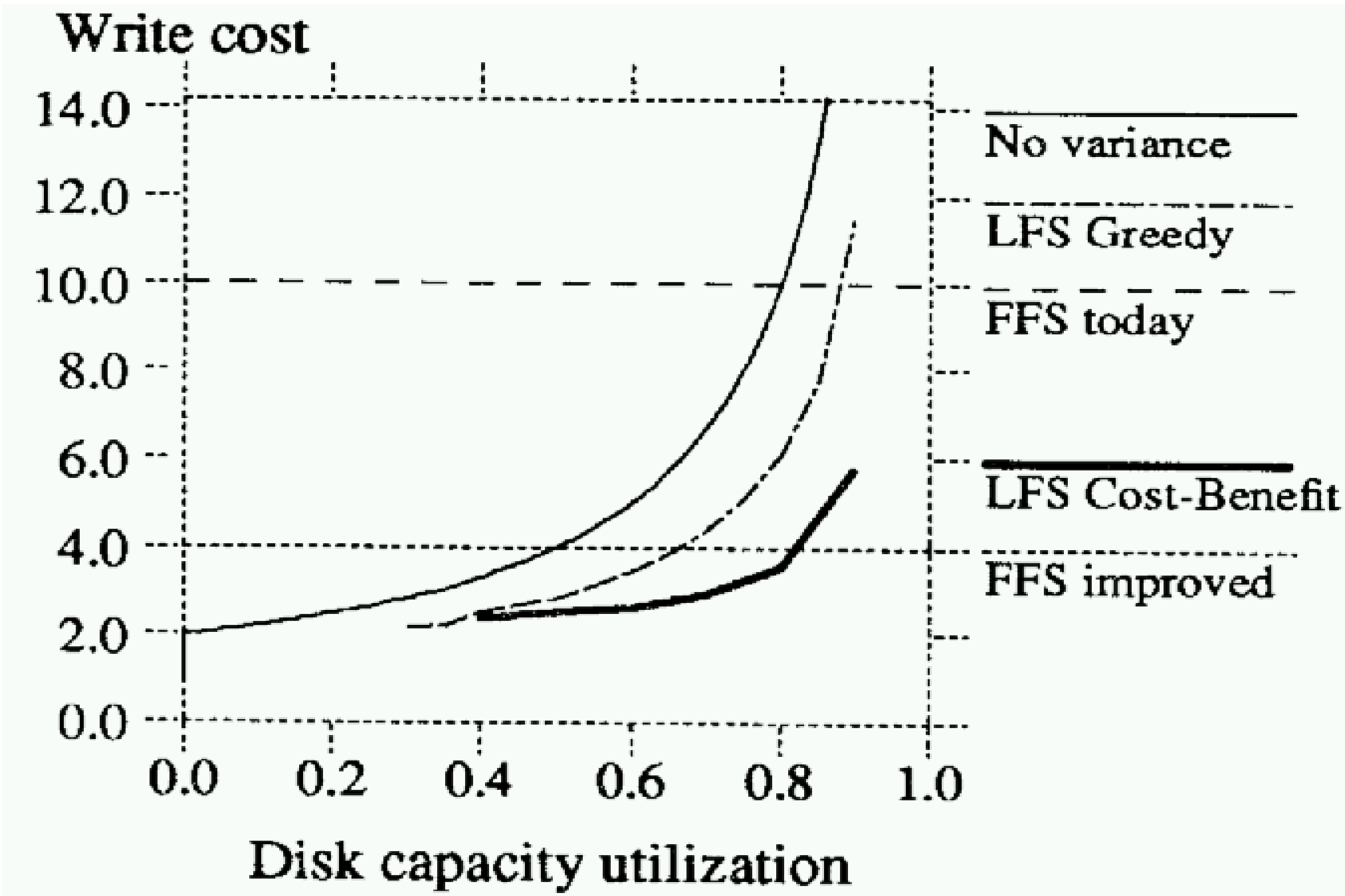
Cleaning policies

- Simulations to determine best policy
  - Greedy: clean based on low utilisation
  - Cost-benefit: use age (time of last write)

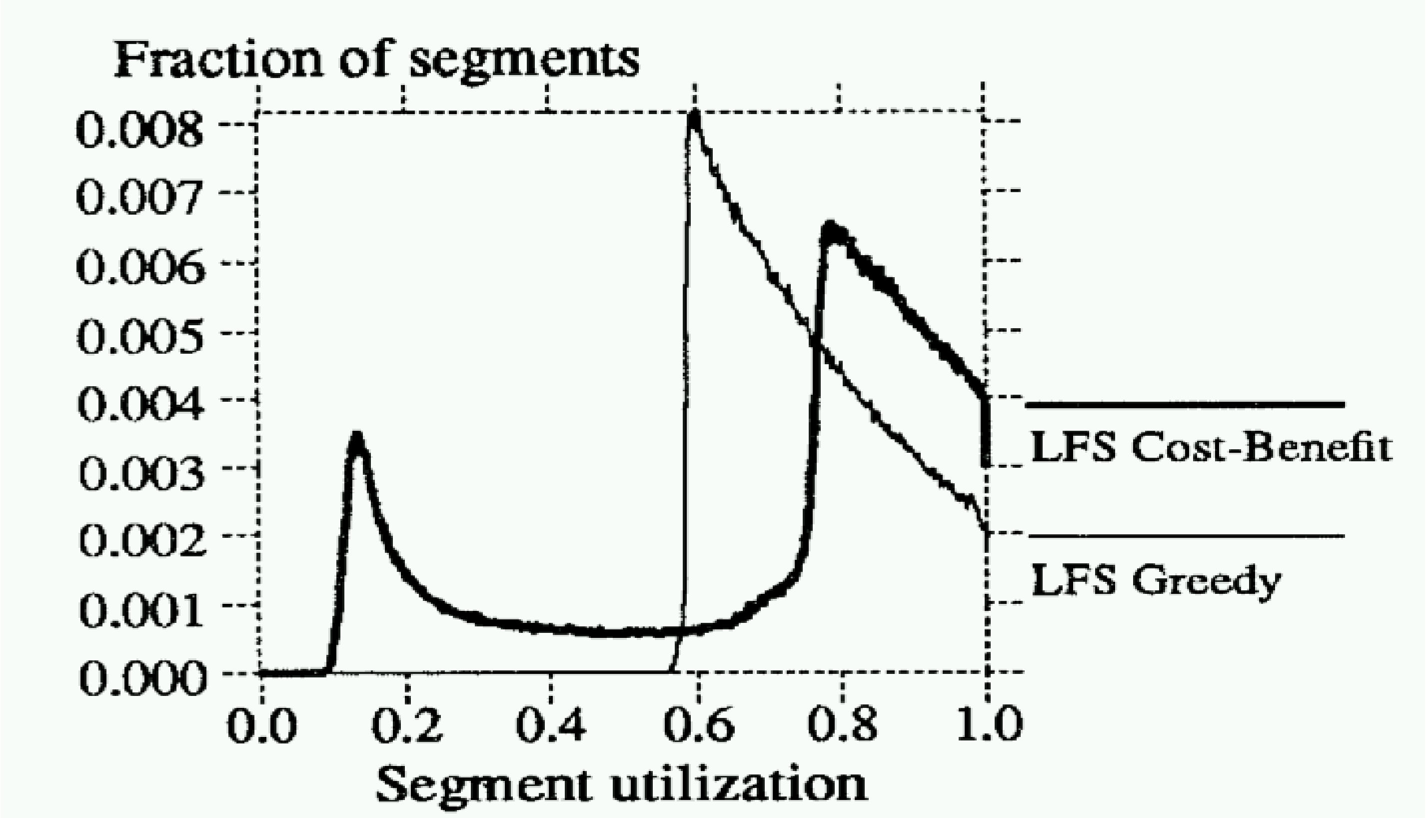
$$\frac{\text{benefit}}{\text{cost}} = \frac{\text{(free space generated)*(age of segment)}}{\text{cost}}$$

- Measure *write cost*
  - Time disk is busy for each byte written
  - Write cost 1.0 = no cleaning

Greedy versus Cost-benefit



Cost-benefit segment utilisation



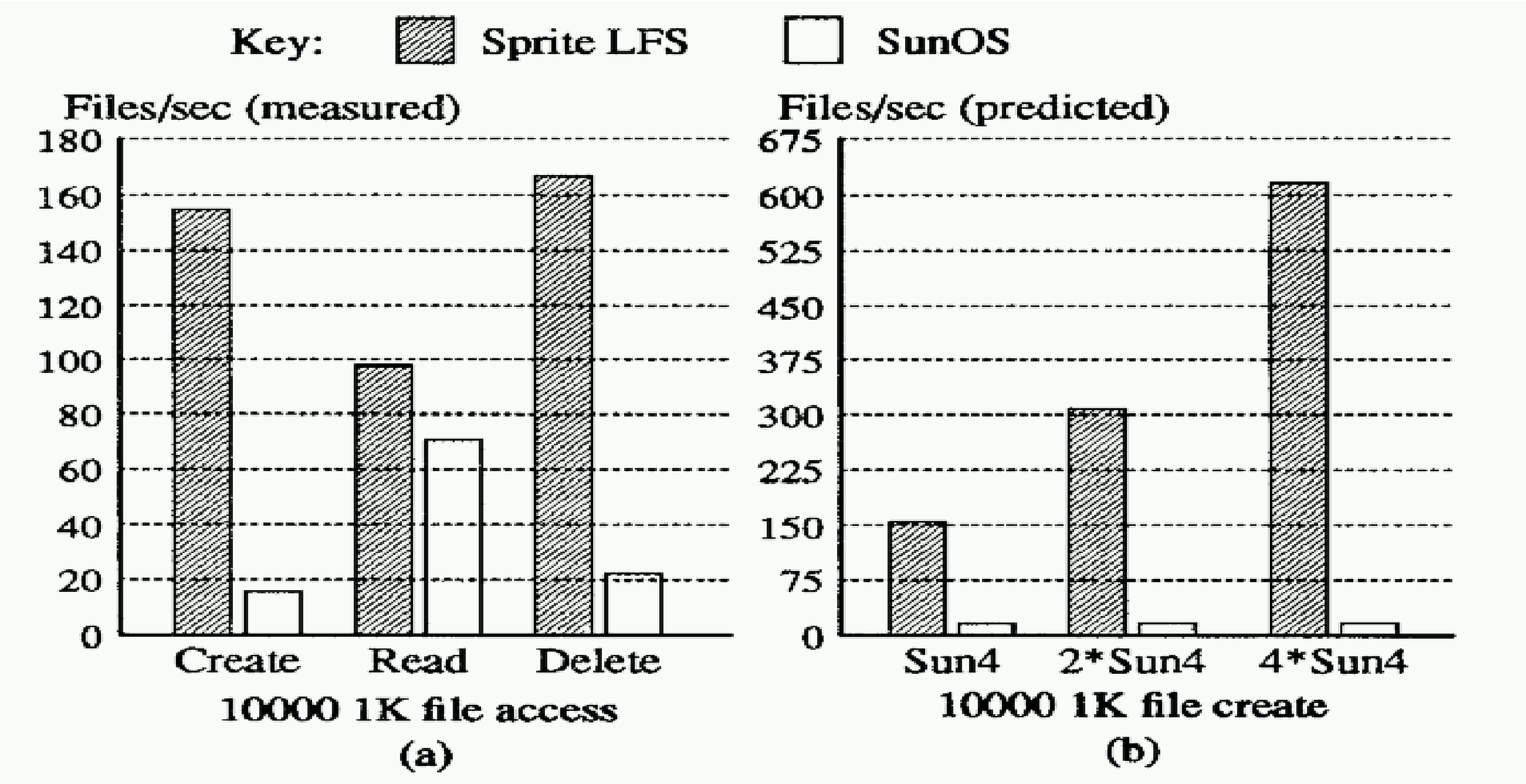
LFS crash recovery

- Log and checkpointing
  - Limited crash vulnerability
  - At checkpoint flush active segment, inode map
- No `fsck` required

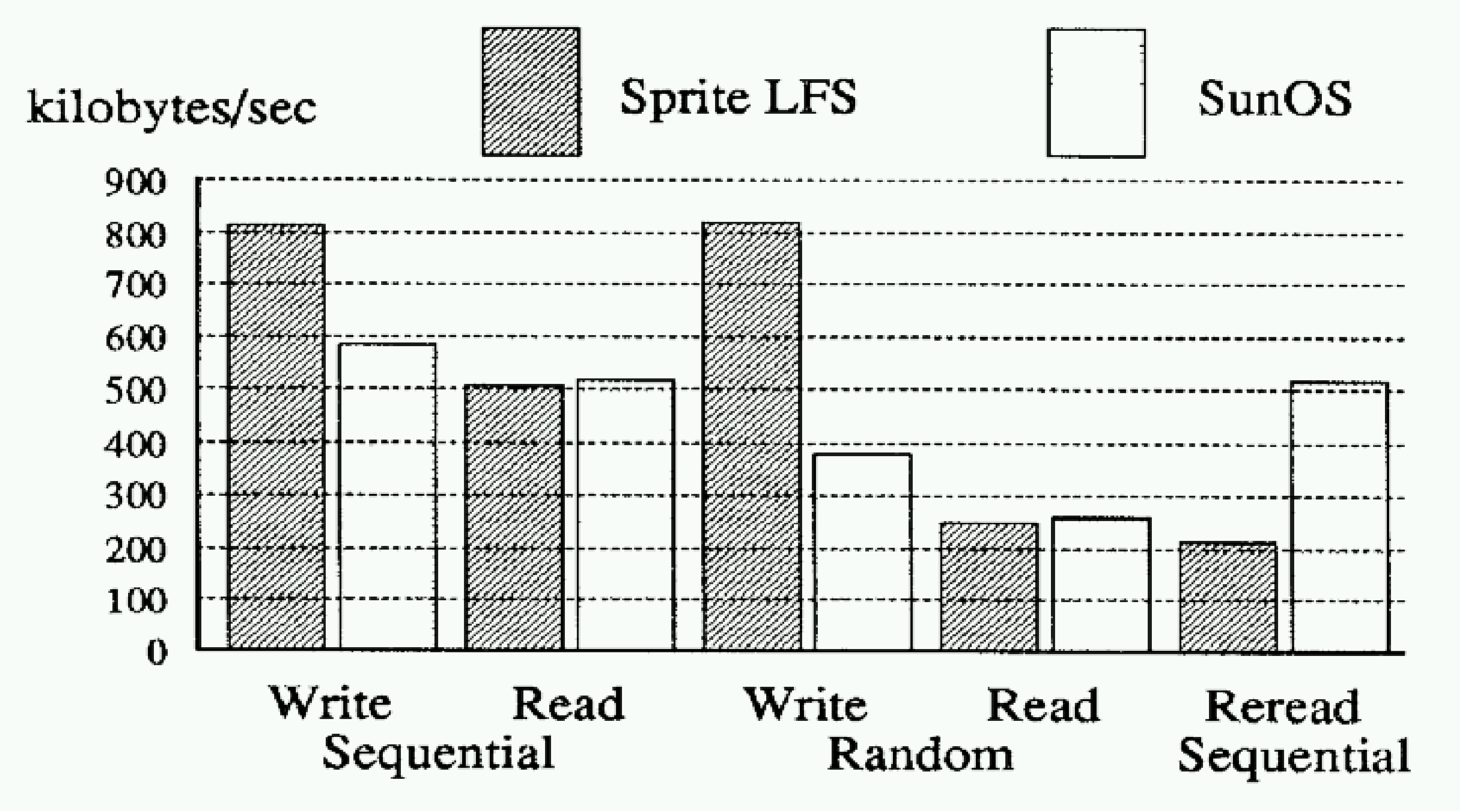
LFS performance

- Cleaning behaviour better than simulated predictions
- Performance compared to SunOS FFS
  - Create-read-delete 10000 1k files
  - Write 100-MB file sequentially, read back sequentially and randomly

Small-file performance



Large-file performance



Overview of talk

- Unix Fast File System
- Log-Structured System
- Soft Updates
- Conclusions

Soft updates

- Alternative mechanism for improving performance of writes
  - All metadata updates can be asynchronous
  - Improved crash recovery
  - Same on-disk structure as FFS

The metadata update problem

- Disk state must be consistent enough to permit recovery after a crash
  - No dangling pointers
  - No object pointed to by multiple pointers
  - No live object with no pointers to it
- FFS achieves this by synchronous writes
  - Relaxing sync. writes requires update sequencing or atomic writes

Design constraints

- Do not block applications unless **`fsync`**
- Minimise writes and memory usage
- Retain 30-second flush delay
- Do not over-constrain disk scheduler
  - It is already capable of some reordering

Dependency tracking

- Asynchronous metadata updates need ordering information
  - For each write, pending writes which precede it
- Block-based ordering is insufficient
  - Cycles must be broken with sync. writes
  - Some blocks stay dirty for a long time
  - False sharing due to high granularity

Circular dependency example

directory

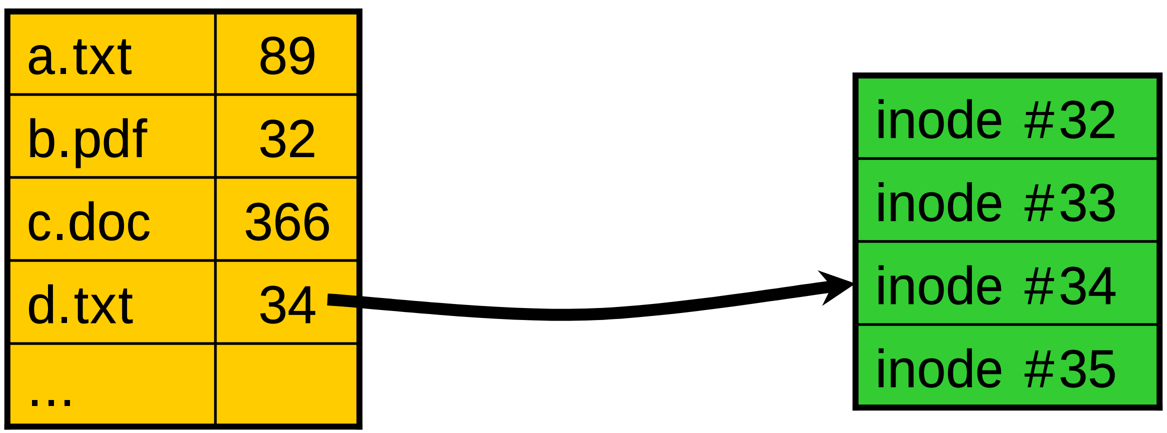
a.txt	89
b.pdf	32
c.doc	366
...	

inode block

inode #32
inode #33
inode #34
inode #35

Circular dependency example

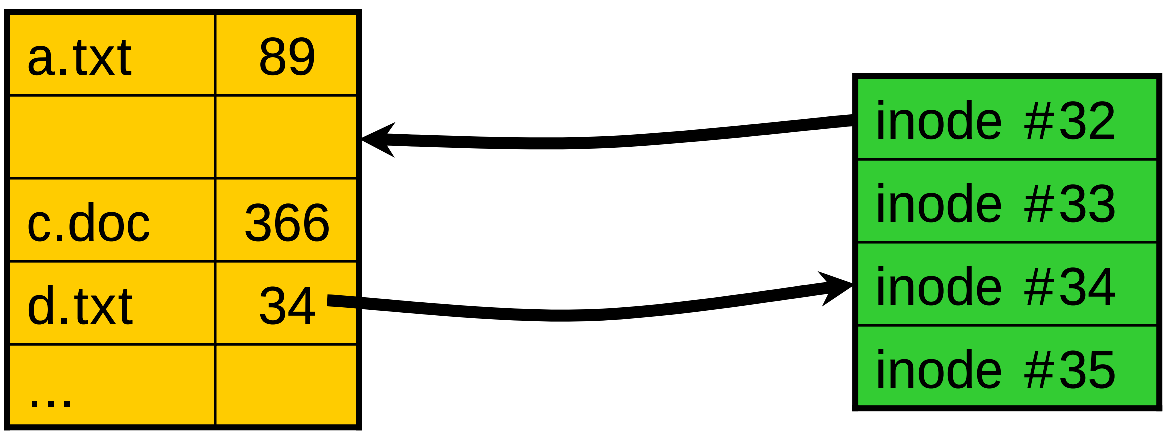
create file d.txt



Inode must be initialised before directory entry is added

Circular dependency example

remove file b.pdf

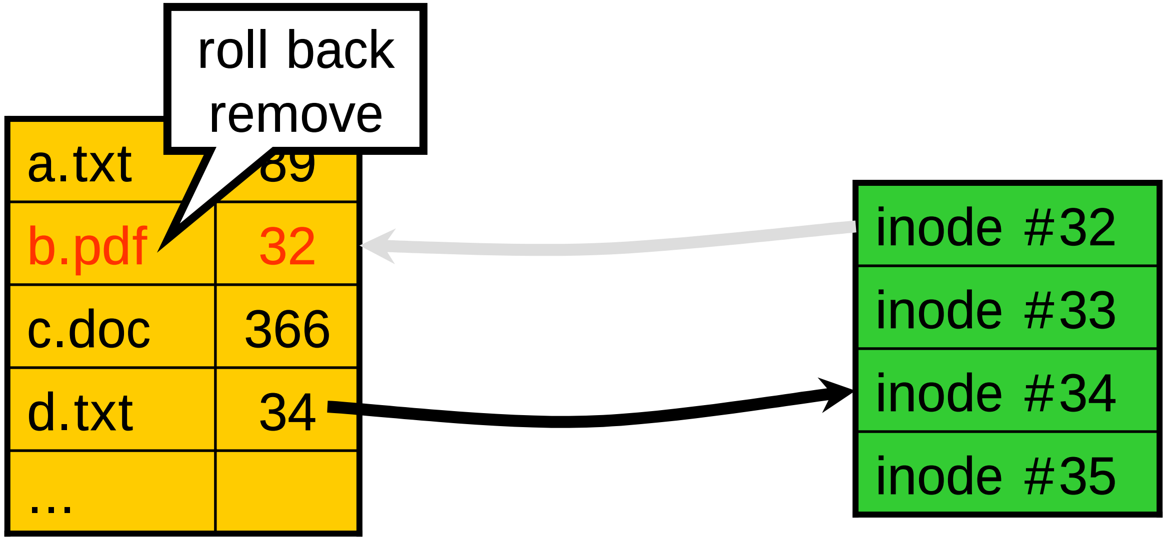


Directory entry must be removed before inode is deallocated

Update implementation

- Update list for each pointer in cache
  - FS operation adds update to each affected pointer
  - Update incorporates dependencies
- Updates have "before", "after" values for pointers
  - Roll-back, roll-forward to break cycles

Circular dependency example



Rollback allows dependency to be suppressed

Soft updates details

- Blocks are locked during roll-back
  - Prevents processes from seeing stale cache
- Existing updates never get new dependencies
  - No indefinite aging
- Memory usage is acceptable
  - Updates block if usage becomes too high

Recovery with soft updates

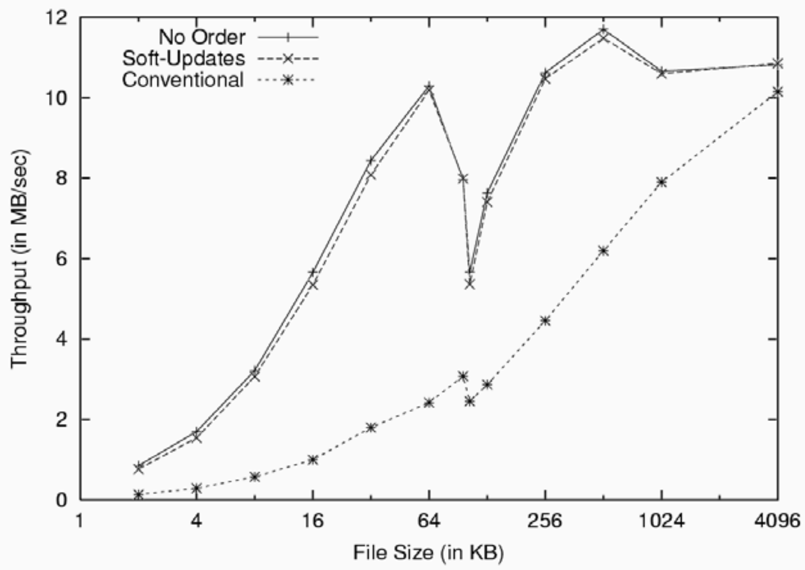
- "Benign" inconsistencies after crashes
  - Freespace maps may miss free entries
  - Link counts may be too high
- **Fsck** is still required
  - Need not run immediately
  - Only has to check in-use inodes
  - Can run in the background

Soft updates performance

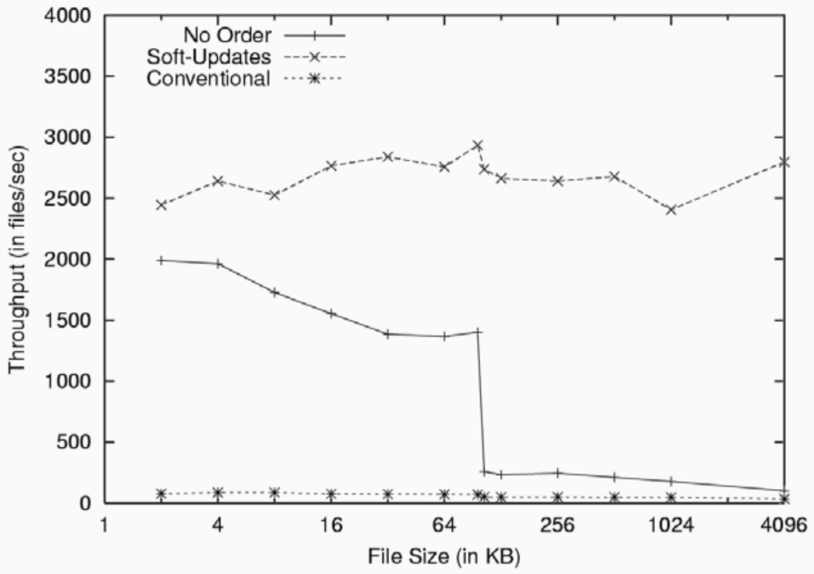
- Recovery time on 76% full 4.5GB disk
  - 150s for FFS **`fsck`** versus 0.35s ...
- Microbenchmarks
  - Compared soft updates, async writes, FFS
  - Create, delete, read for 32MB of files
- Soft updates versus update logging
  - **`Sdet`** benchmark of "user scripts"
  - Various degrees of concurrency

Create and delete performance

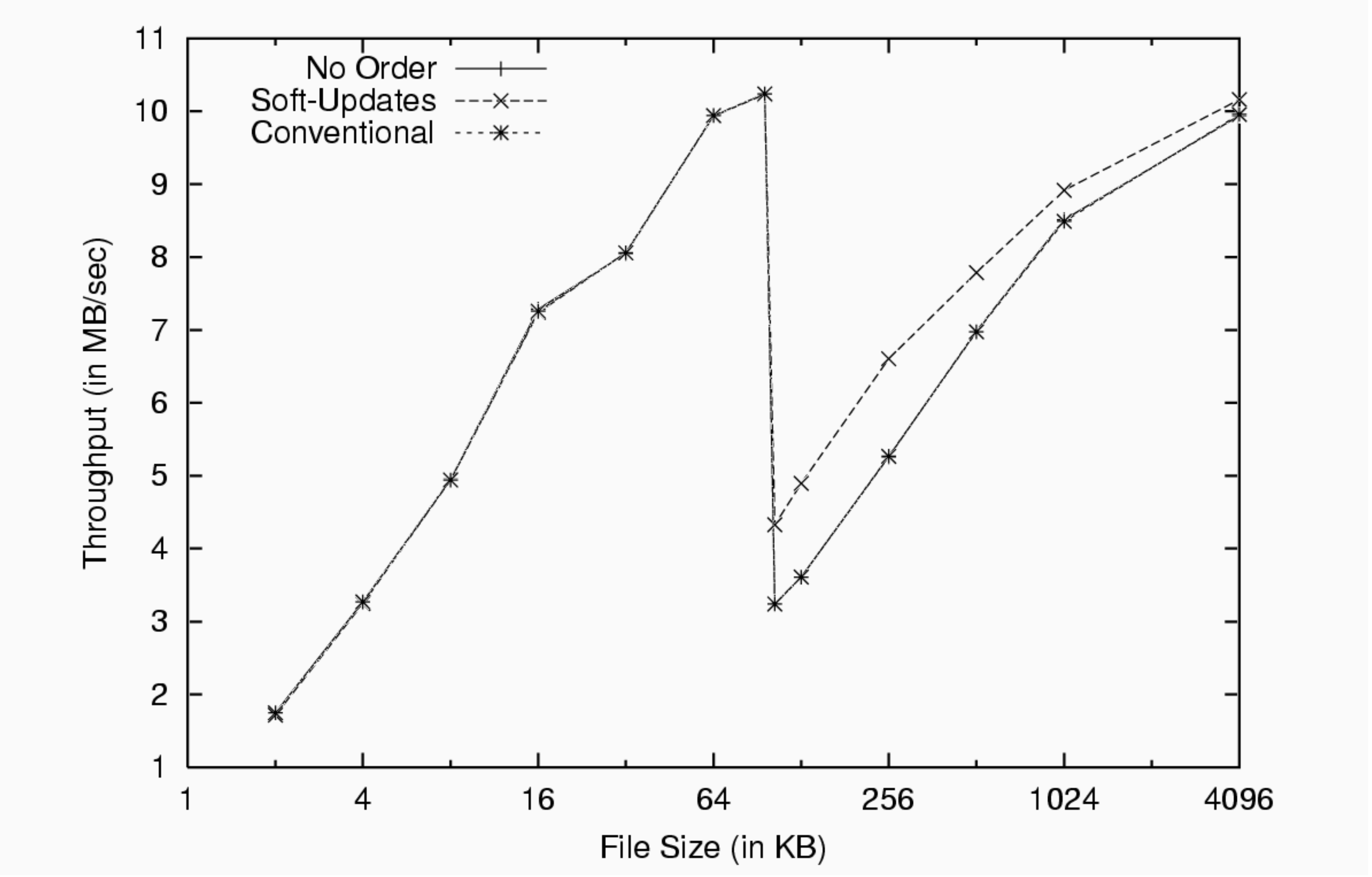
Create files



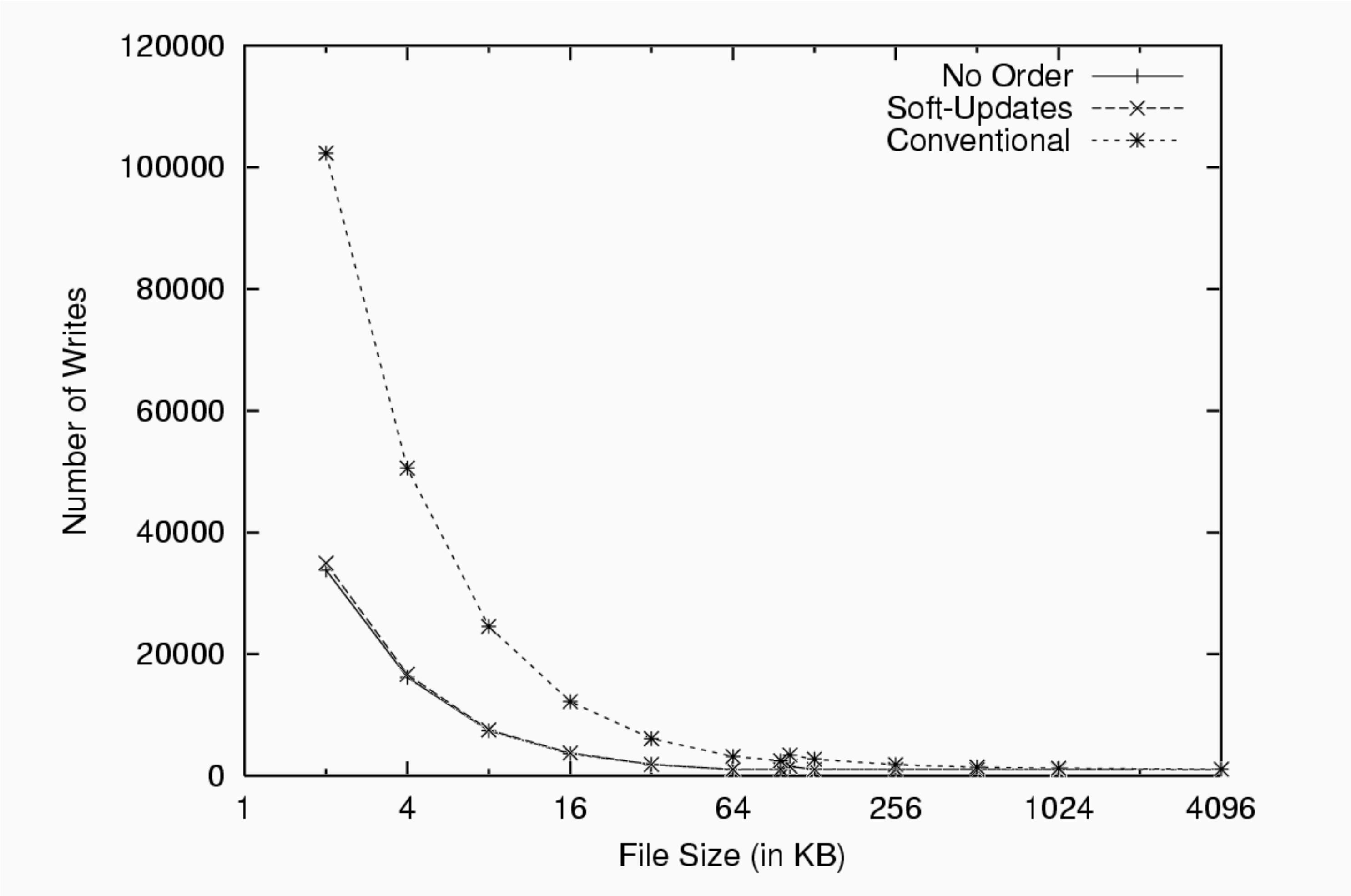
Delete files



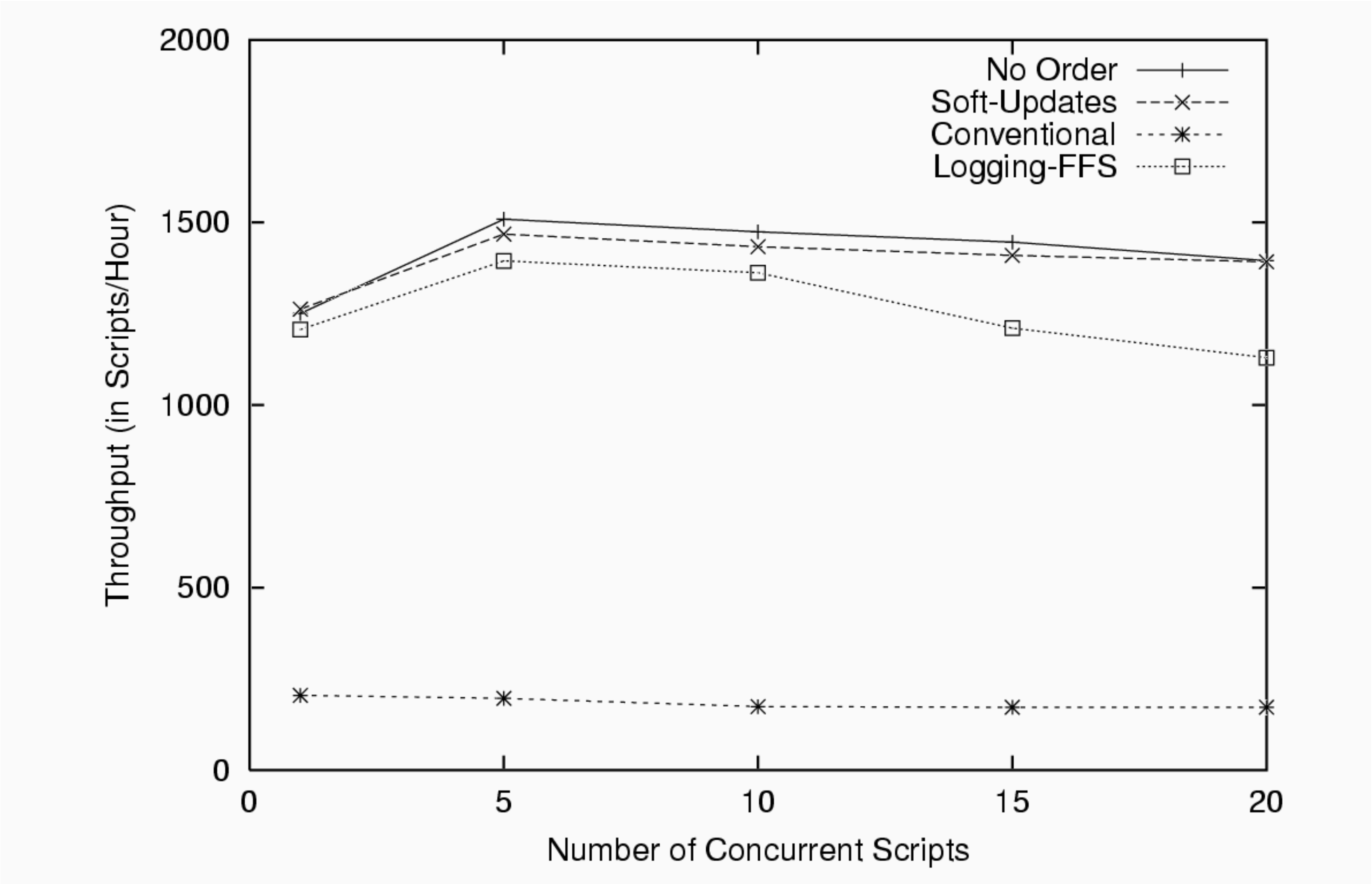
Read performance



Overall create traffic



Soft updates versus logging



Conclusions

- Papers were separated by 8 years
  - Much controversy regarding LFS-FFS comparison
- Both systems have been influential
  - IBM Journalling file system
  - Ext3 filesystem in Linux
  - Soft updates come enabled in FreeBSD