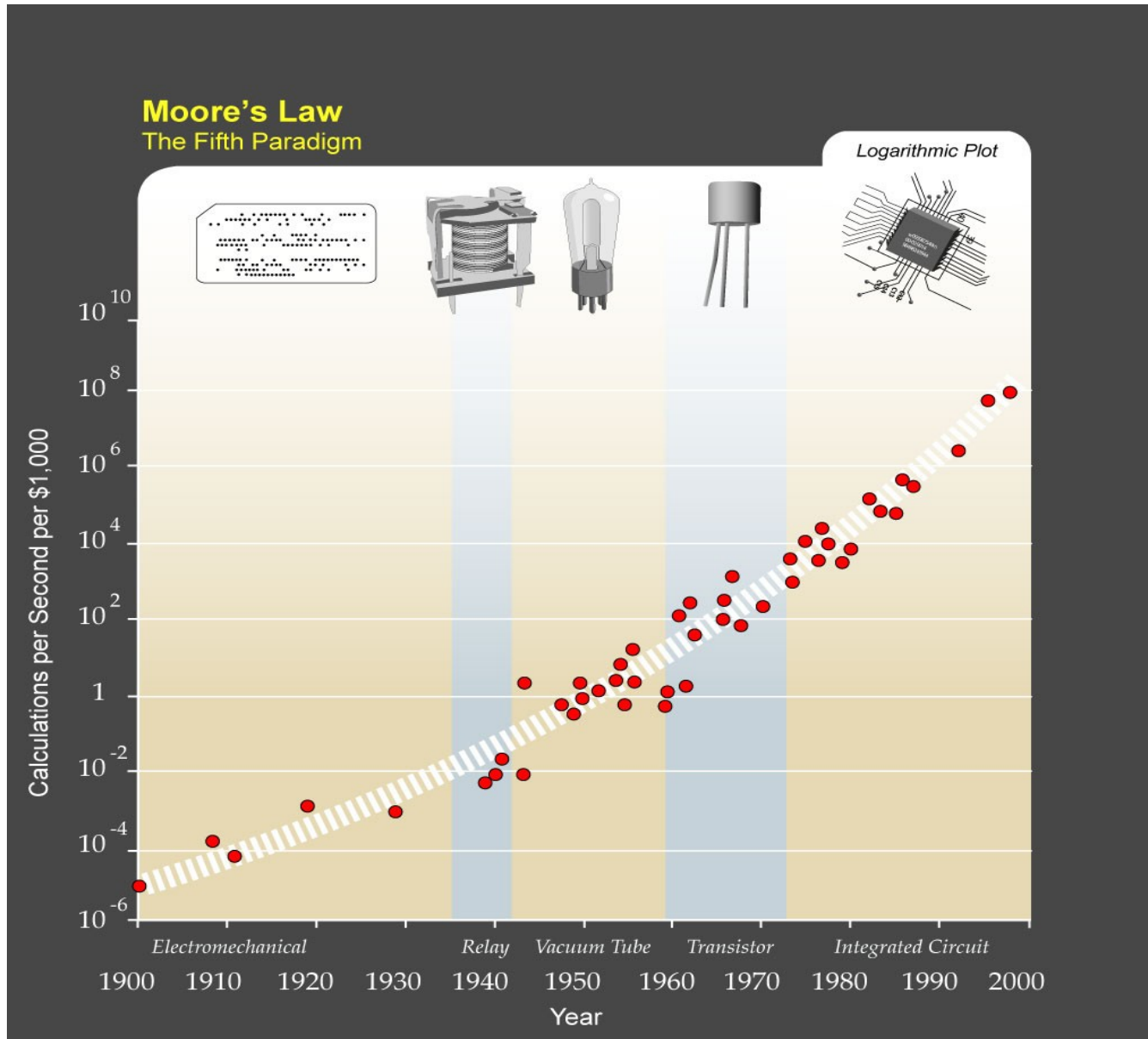


# The Design and Implementation of a Log-Structured File System

Mendel Rosenblum and John K. Ousterhous

Presented by Ian Elliot

# Processor speed is getting faster...



... A lot faster,  
and quickly!

# Hard disk speed?

- Transfer speed vs. sustainable transfer speed vs. access speed (seek times)
- Seek times are especially problematic...
- They're getting faster, even potentially exponentially, but by a very small constant relative to processor speed.

# Main memory is growing

- Makes larger file caches possible
- Larger caches = less disk reads
- Larger caches  $\neq$  less disk writes (more or less)
  - This isn't quite true.. The more write data we can buffer, the more we may be able to clump writes to require only disk access...
  - Doing so is severely bounded, however, since you must dump the data to disk in a somewhat timely manner for safety

- Office and engineering applications tend to access many small files (mean file size being “only a few kilobytes” by some accounts)
- Creating a new file in recent file systems (e.g. Unix FFS) requires many seeks
  - Claim: When writing small files in such systems, less than 5% of the disk's potential bandwidth is used for new data.
- Just as bad, applications are made to wait for certain slow operations such as inode editing

# (ponder)

- How can we speed up the file system for such applications where
  - files are small
  - writes are as common (if not more common) than reads due to file caching
- When trying to optimize code, two strategies:
  - Optimize for the common case (cooperative multitasking, URPC)
  - Optimize for the slowest case (address sandboxing)

Good news / Bad news

# Good news / Bad news

- The bad news:
  - Writes are slow

# Good news / Bad news

- The bad news:
  - Writes are slow
- The good news:
  - Not only are they slow, but they're the common case (due to file caching)

# Good news / Bad news

- The bad news:
  - Writes are slow
- The good news:
  - Not only are they slow, but they're the common case (due to file caching)

( Guess which one we're going to optimize... )

# Recall soft timers...

- Ideally we'd handle certain in-kernel actions when it's convenient
- What's ideal or convenient for disk writes?

# Ideal disk writes

- Under what circumstances would we ideally write data?
  - Full cluster of data to write (better throughput)
  - Same track as the last disk access (don't have to move the disk head, small or no seek time)

# Ideal disk writes

- Under what circumstances would we ideally write data?
  - Full cluster of data to write (better throughput)
  - Same track as the last disk access (don't have to move the disk head, small or no seek time)

**Make it so!**



( ... Number One )

- Full cluster of data? Buffering writes out is a simple matter
  - Just make sure you force a write to disk every so often for safety
- Minimizing seek times? Not so simple...

( idea )

- Sequential writing is pretty darned fast
  - Seek times are minimal? Yes, please!
- Let's always do this!

# ( idea )

- Sequential writing is pretty darned fast
  - Seek times are minimal? Yes, please!
- Let's always do this!
- **What could go wrong?**
  - Disk reads
  - End of disk

# Disk reads

- Writes to disk are always sequential.
  - That includes inodes
- Typical file systems
  - inodes in fixed disk locations
- inode map (another layer of indirection)
  - table of file number → inode disk location
  - we store disk locations of inode map “blocks” at a fixed disk location (“checkpoint region”)
- Speed? Not too bad since the inode map is usually fully cached

# Speaking of inodes...

- This gives us flexibility to write new directories and files in potentially a single disk write
  - Unix FFS requires ten (eight without redundancy) separate disk seeks
  - Same number of disk accesses to read the file
- Small reminder:
  - inodes tell us where the first ten blocks in a file are and then reference indirect blocks

# End of disk

- There is no vendor that sells Turing machines
- Limited disk capacity
- Say our hard disk is 300 “GB” (grumble) and we've written exactly 300 “GB”
  - We could be out of disk space...
  - Probably not, though. Space is often reclaimed.

# Free space management

- Two options
  - Compact the data (which necessarily involves copying)
  - Fill in the gaps (“threading”)
- If we fill in the gaps, we no longer have full clusters of information. Remind you of file segmentation, but at an even finer scale? (Read: Bad)

# Compaction it is

- Suppose we're compacting the hard drive to leave large free consecutive clusters...
- Where should we write lingering data?
- Hmm, well, where is writing fast?
  - Start of the log?
  - That means for each revolution of our log end around the disk, we will have moved all files to the end, even those which do not change
  - Paper: (cough) Oh well.

# Sprite LFS

- Implemented file system uses a hybrid approach
- Amortize cost of threading by using larger “segments” (512KB-1MB) instead of clusters
- Segment is always written sequentially (thus obtaining the benefits of log-style writing)
  - If the segment end is reached, all data must be copied out of it before it can be written to again
- Segments themselves are threaded

# Segment “cleaning” (compacting) mechanism

- Obvious steps:
  - Read in  $X$  segments
  - Compact segments in memory into  $Y$  segments
    - Hopefully  $Y < X$
  - Write  $Y$  segments
  - Mark the old segments as clean

# Segment “cleaning” (compacting) mechanism

- Record a cached “version” counter and inode number for each cluster at the head of the segment it belongs to
- If a file is deleted or its length set to zero, increase the cached version counter by one
- When cleaning, we can immediately discard a cluster if its version counter does not match the cached version counter for its inode number
- Otherwise, we have to look through inodes

# Segment “cleaning” (compacting) mechanism

- Interesting side-effect:
  - No free-list or bitmap structures required...
  - Simplified design
  - Faster recovery

# Compaction policies

- Not so straightforward
  - When do we clean?
  - How many segments?
  - Which segments?
  - How do we to group live blocks?

# Compaction policies

- Clean when there's a certain threshold of empty segments left
- Clean a few tens of segments at a time
- Stop cleaning we have “enough” free segments
  
- Performance doesn't seem to depend too much on these thresholds. Obviously you wouldn't want to clean your entire disk at one time, though.

# Compaction policies

- Still not so straightforward
  - ~~When do we clean?~~
  - ~~How many segments?~~
  - Which segments?
  - How do we to group live blocks?

# Compaction policies

- Segments amortize seek times and rotation latency. That means where the segments are isn't much of a concern
- Paper uses unnecessary formulas to say the bloody obvious:
  - If we try to compact segments with more live blocks, we'll spend more time copying data and achieving achieving free segments
  - That's bad. Don't do that.

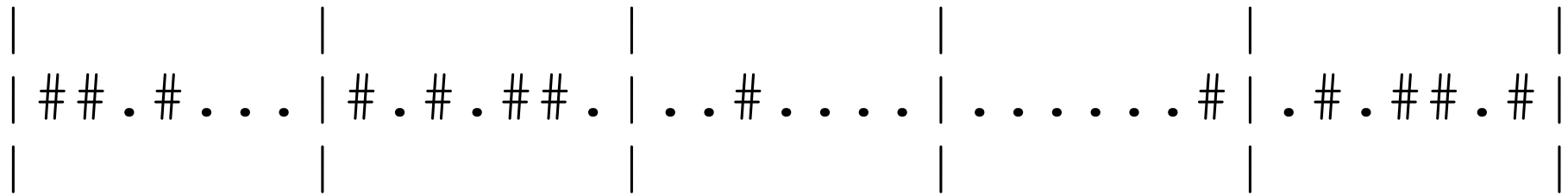


# An example:





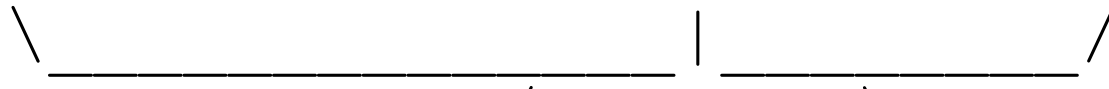
# An example:



Read

Read

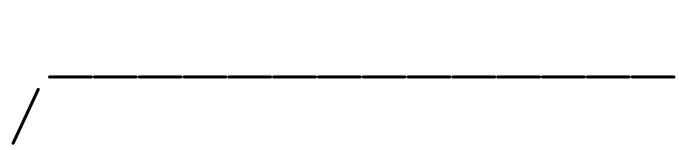
Read



/Compact\



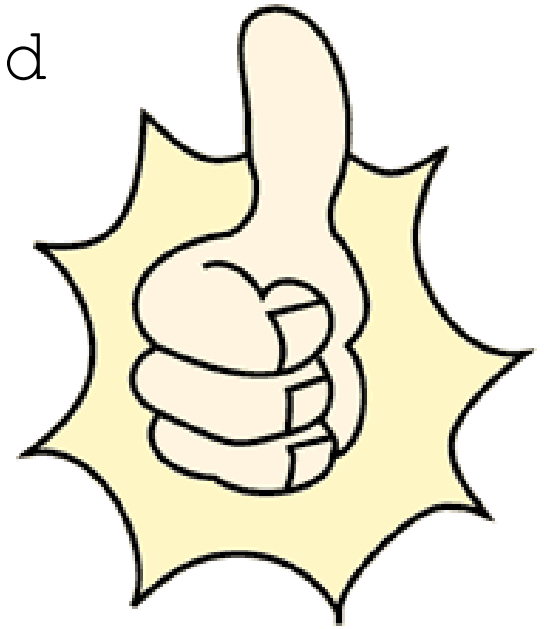
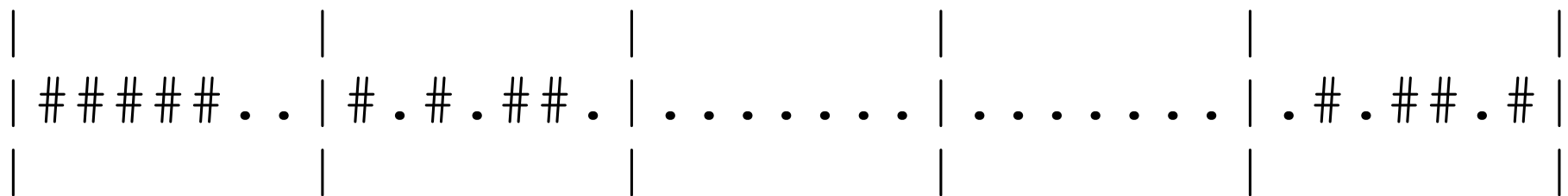
#####..



Write

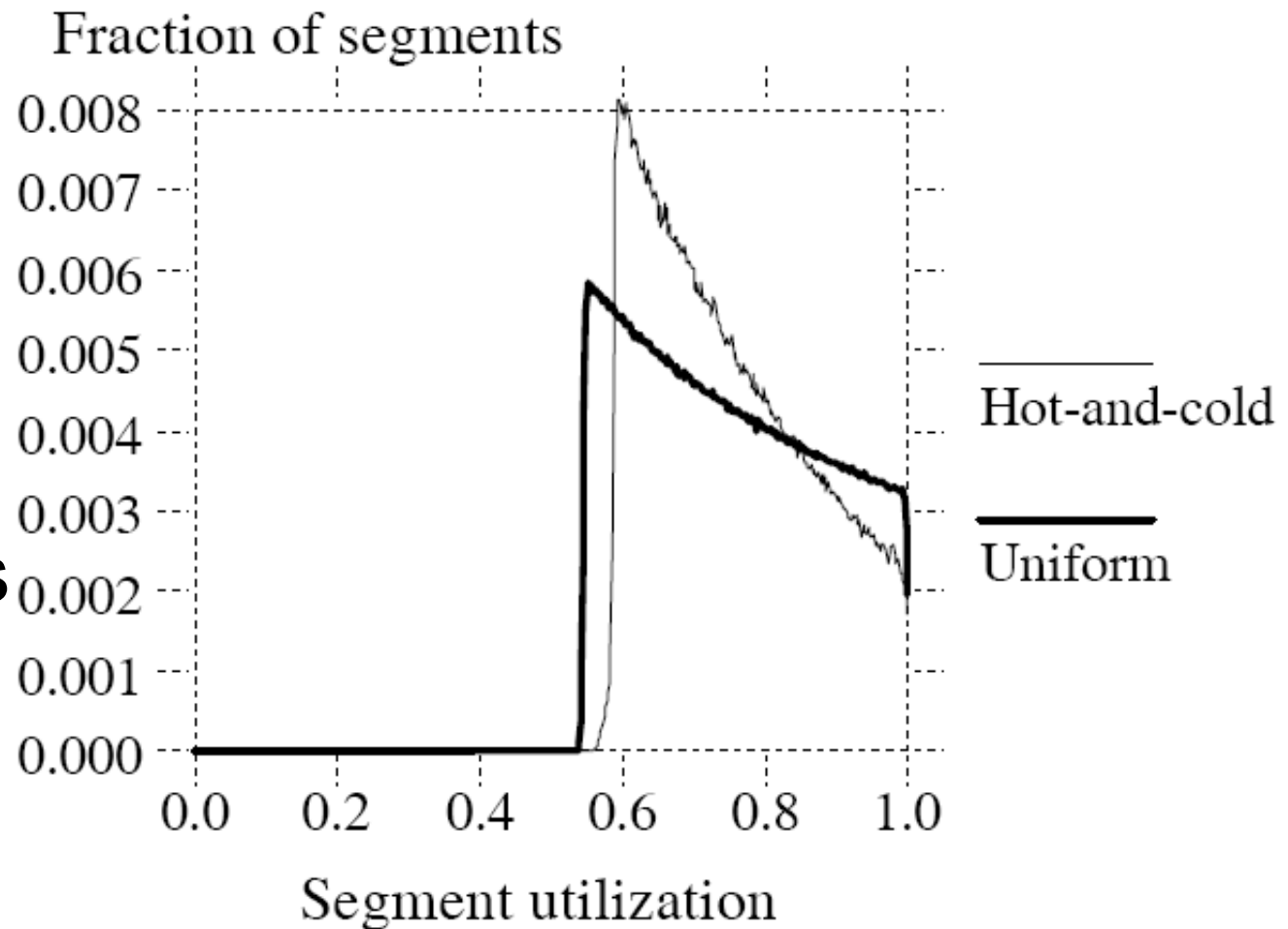
Free

Free



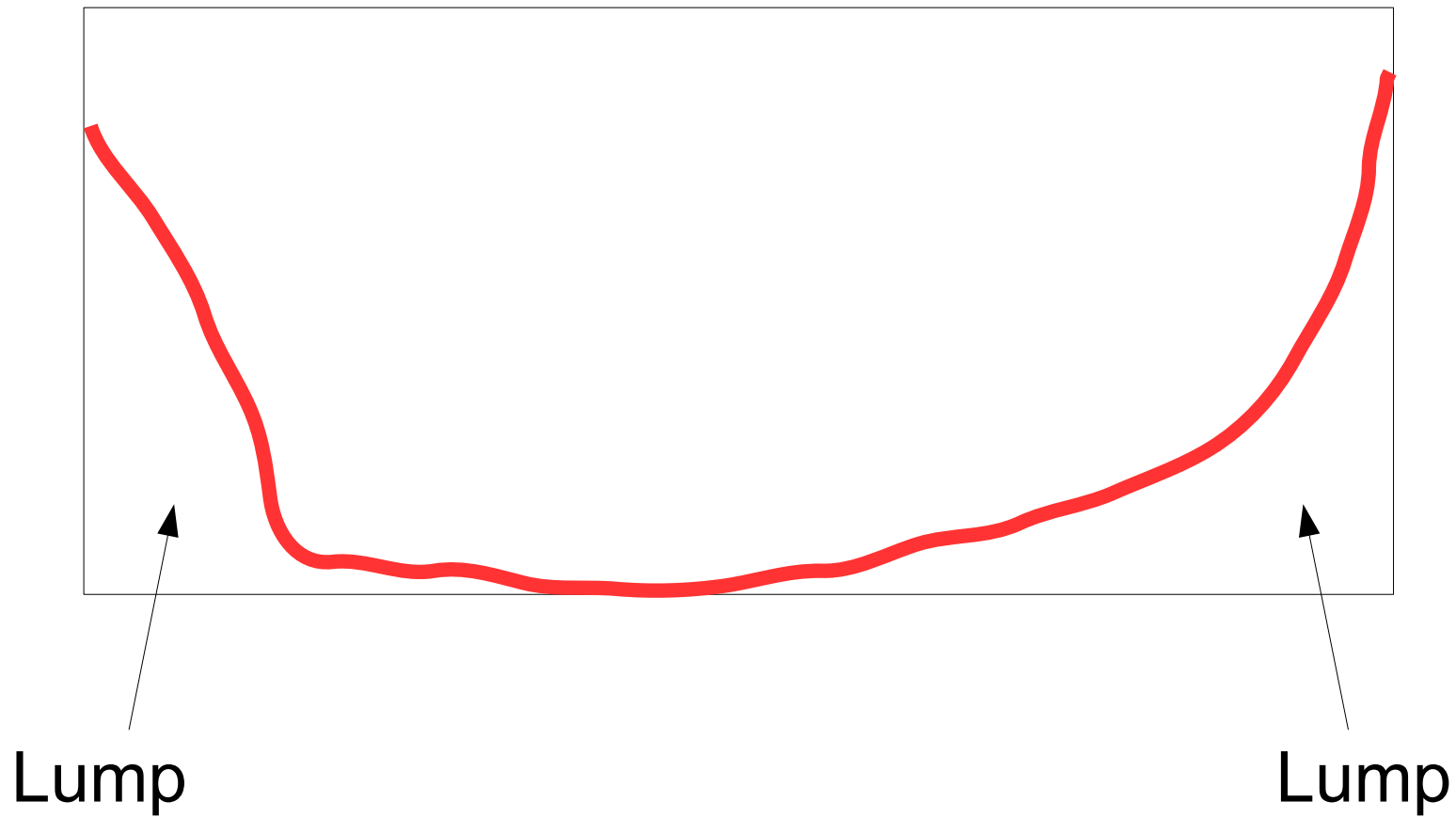
# Compaction policies

- This suggests a greedy strategy: choose lowest utilized segments
- Interesting simulation results with localized accesses
- Cold segments tend to linger near lowest utilization



# Compaction policies

- What we really want is a bimodal distribution:



# Compaction policies

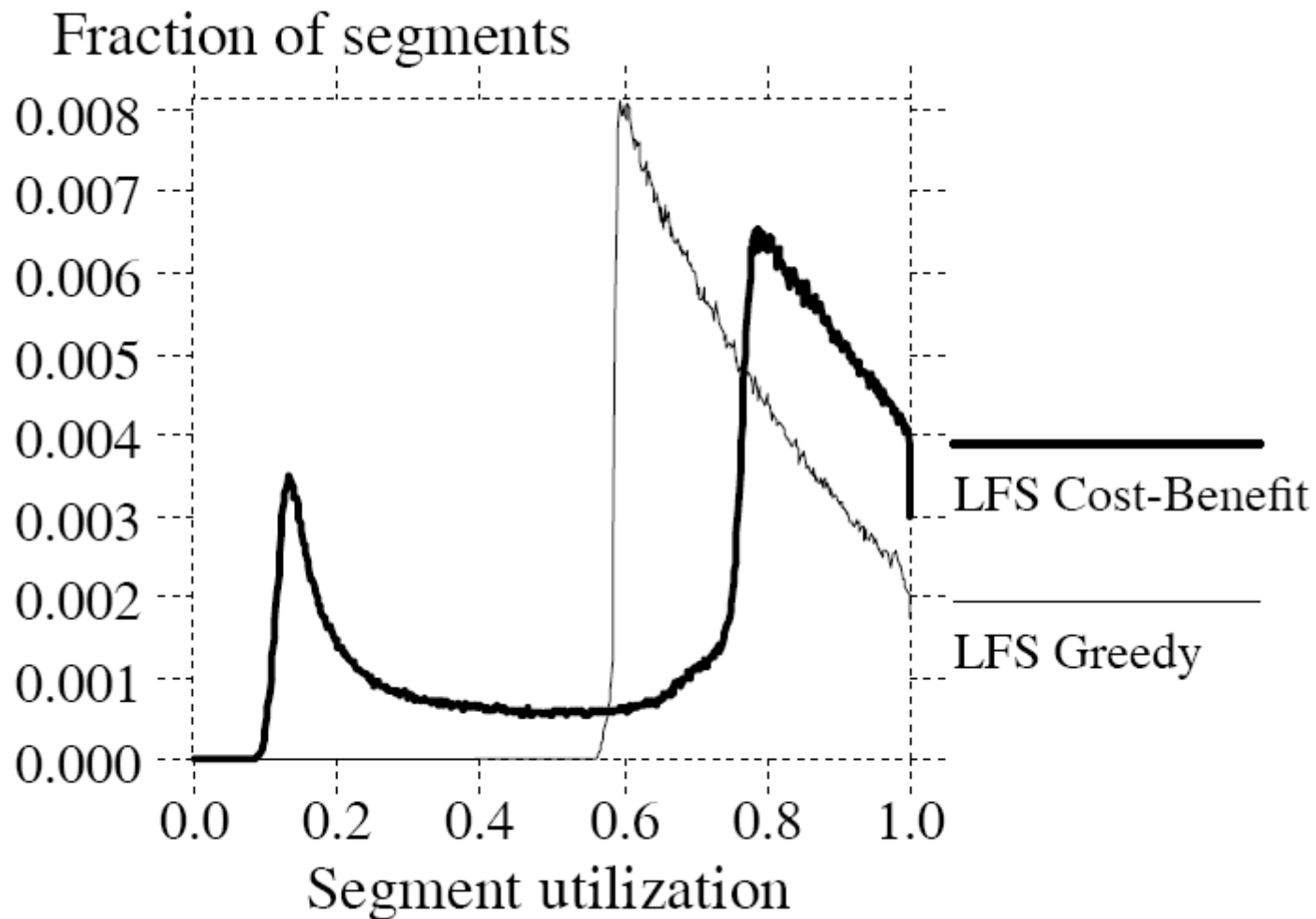
- How do we prevent lingering data?
- Reclaiming space from cold segments is actually more valuable than reclaiming space from hot segments.
- Cold segments contain unchanging data... If we compact relatively unchanging data, we will not have to compact it again soon.

# Compaction policies

- Impossible to predict future changes, so we use a simple heuristic:
  - If it hasn't been changed for a while, it's not likely to be changed again soon
- In general, the older a cluster is, the less likely it is to change.
- Good things: Free space, age of data
- Bad things: Reading/writing
- Formula: 
$$\frac{\text{benefit}}{\text{cost}} = \frac{(\text{free space}) * \text{age}}{\text{reading} + \text{writing}}$$

# Compaction policies

- New simulation results:

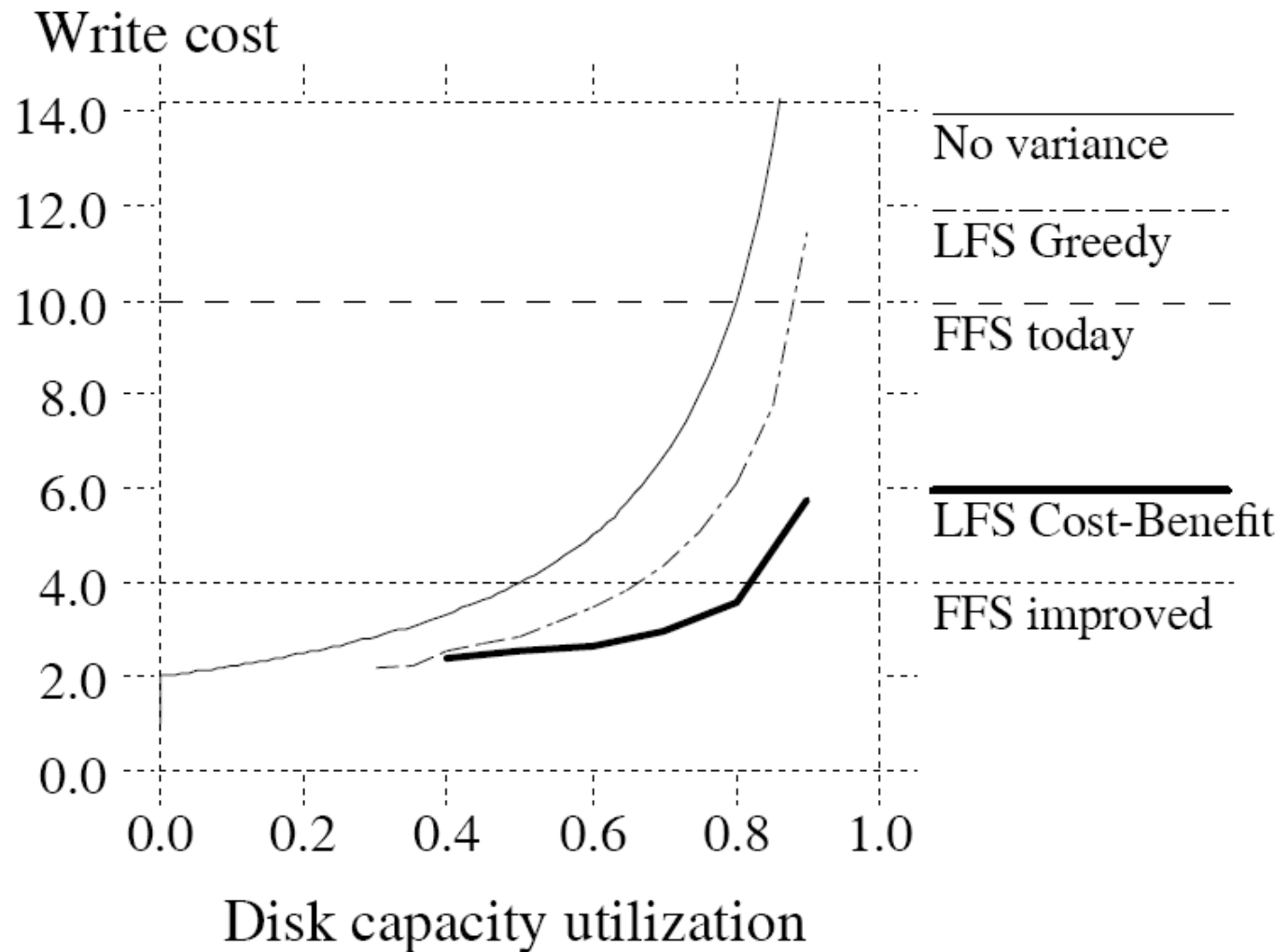


# Compaction policies

- Greedy approach was comparable to Unix FFS

- The Cost-Benefit approach is remarkably better

- Actual implementation uses this approach



# Crash recovery

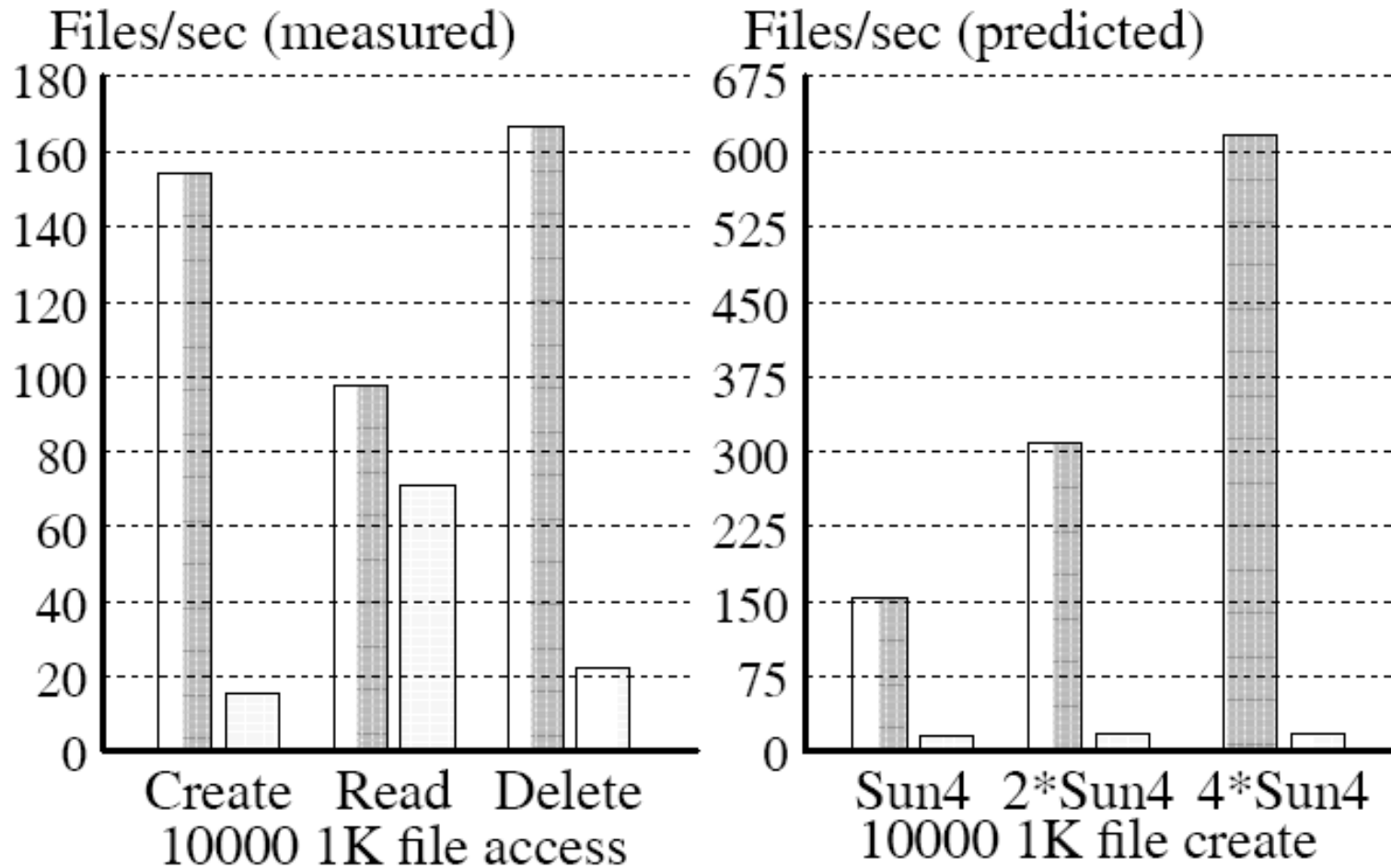
- Log structure allows crash recovery to be fast since the location of all recent entries are consolidated
- Checkpoints are places in the log where the file system structure is consistent and complete
- Done periodically or when unmounted
- Recording a checkpoint includes writing the entire inode map, segment usage table (used for the compaction policy), the current time, and a pointer to the last segment written.
- Quick recovery from check-point

# Crash recovery

- We can do better
- Short version:
  - Examine post-checkpoint segments
  - Reconstruct modified clusters
  - Reconstruct inodes and directory entries

# More results

Key:  Sprite LFS  SunOS

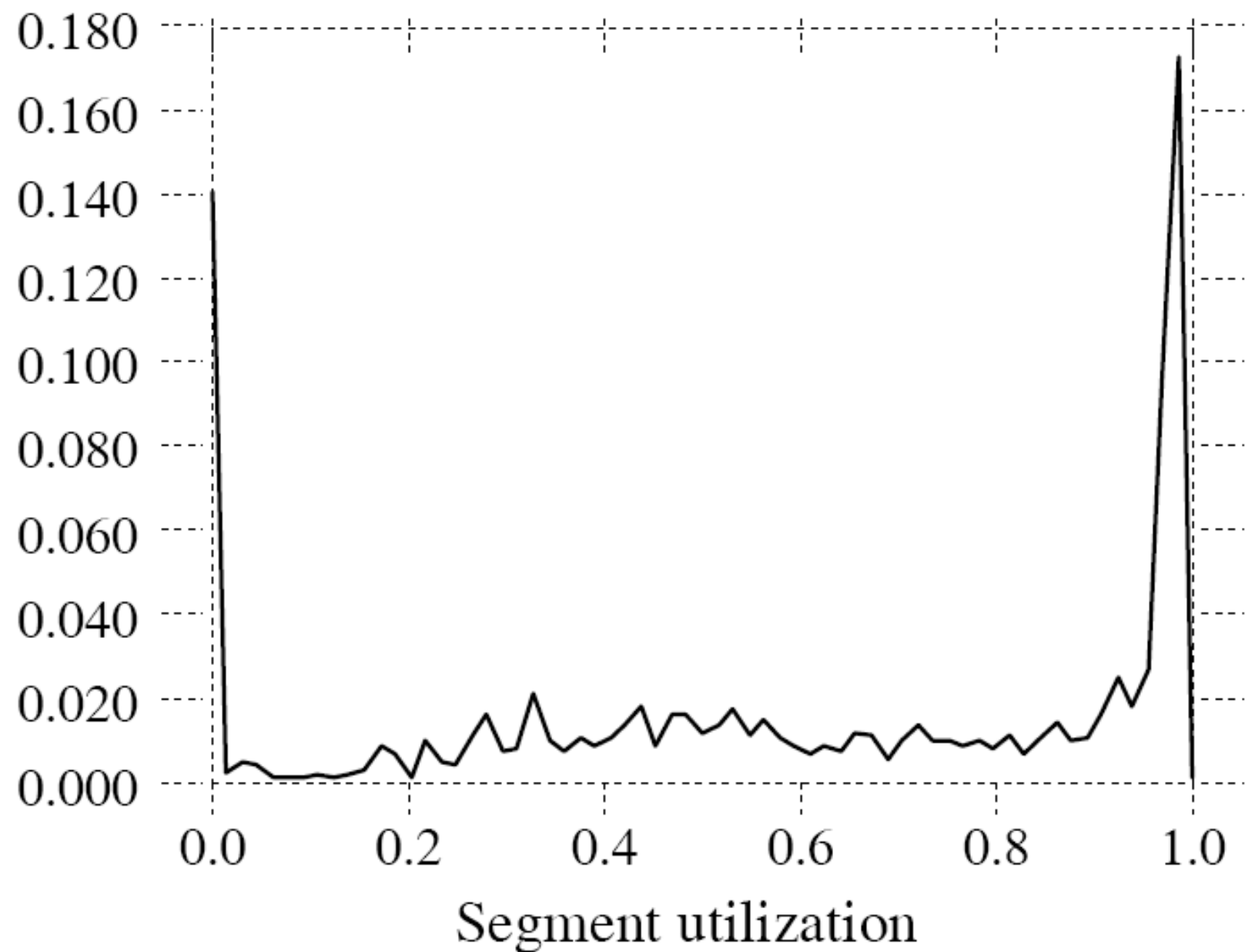


Sun4: An entire 16.67Mhz of Fujitsu goodness

# More results

- This figure is skewed because it doesn't include compaction times...
- In reality, compaction overhead is 70% of sequential-write performance.
- Performance was actually better than expected with a better bimodal distribution. Authors attribute this to super-cold segments and multi-cluster files enhancing intra-segment locality.

Fraction of segments



# Conclusion

- Cache writes, write in a single disk access
  - Complicated by need to free data
  - Log-structure works effectively, even for large files (large-file deletes are nearly free!)
- Only one real problem noted in paper:
  - Random writes and then sequential reads expose the obvious weakness: physical location of blocks depends on when you wrote them, not where you said to write them.
  - Not really a problem in real applications

# Additional comments

- I suspect they didn't store any video files on those Sun-4's. This file system may not cope well with the new computer uses of today.

# Wikipedia strikes again!

- <http://upload.wikimedia.org/wikipedia/commons/c/c5/PPTMooresLawai.jpg>

