The Design and Implementation of a Log-Structured File System

Mendel Rosenblum
John K. Ousterhout

Electrical Engineering and Computer Sciences, Computer Science Division, University of California Berkeley

Contents

- Introduction
- Log-Structued File Systems
  - How to retrieve information from the log
  - How to manage the free space on disk
    - Cleaning Mechanism
    - Cleaning Policy
- Evaluation
- Conclusion

Introduction

- **CPU speeds have increased dramatically**
  - Applications become disk-bound
- **Memory sizes are growing**
  - Read performance is getting better
  - Disk traffic consists of writes
- **Growing gap between random small file access and sequential large file access Performance**
  - Disk transfers bandwidth has increased
  - Disk access times have evolved much more slowly
- **Existing file systems perform poorly on small random disk I/O**
  - They spread information around the disk in a way that causes too many small accesses
  - They tend to write synchronously

Log-Structured File Systems

- **Basic structures are identical to those used in Unix FFS**
  - –For each file, there exists a data structure called inode
- **The fundamental idea is to improve write performance**
  - –Buffering a sequence of file system changes in the file cache
  - –Writing all the changes to disk sequentially in a single disk write operations
- **Two key issues must be resolved to achieve the benefits of the logging**
  - –How to retrieve information from the log
  - –How to manage the free space on disk
    - Large extent of free space must be always available for writing new data

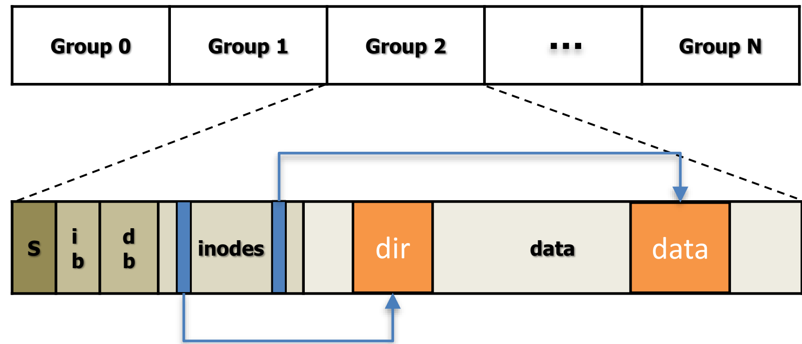
How to retrieve information from the log

- **Goal is to match or exceed the read performance of Unix FFS**
- **Sprite LFS doesn't place inodes at fixed positions**
  - They are written to the log
  - *inode map* must be indexed to determine the disk address of the inode

#**Segment**: Large fixed-size extents for logging

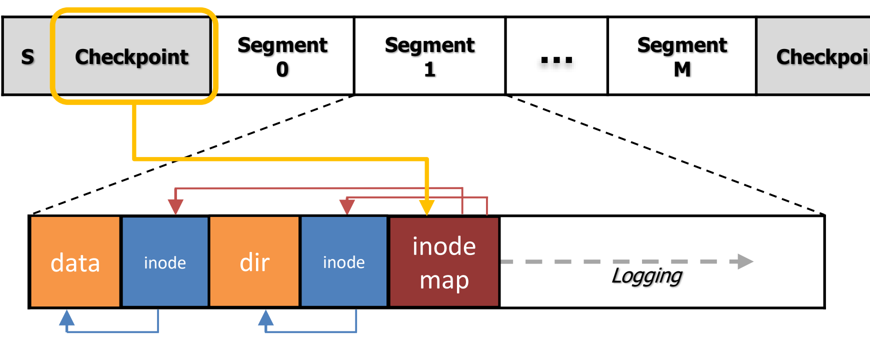
(transfer time >>>> cost of seek to the segment)

FFS Disk Structure



Each inode is at a fixed location on disk

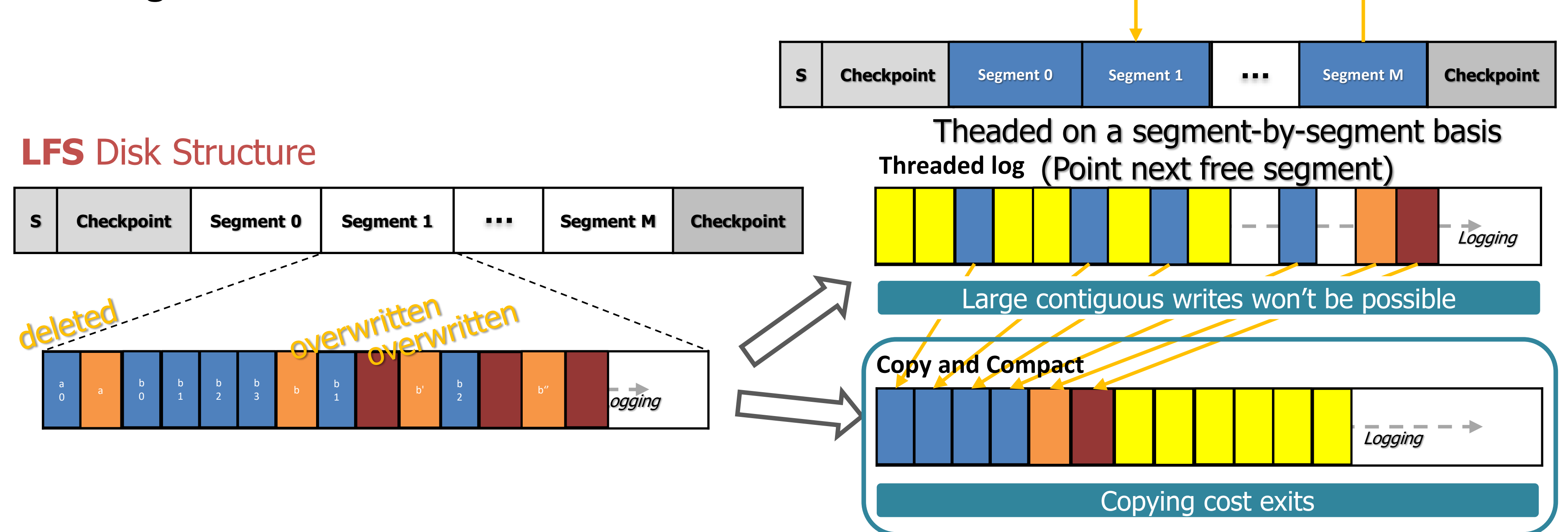
LFS Disk Structure



inode map maintain the location of each inode

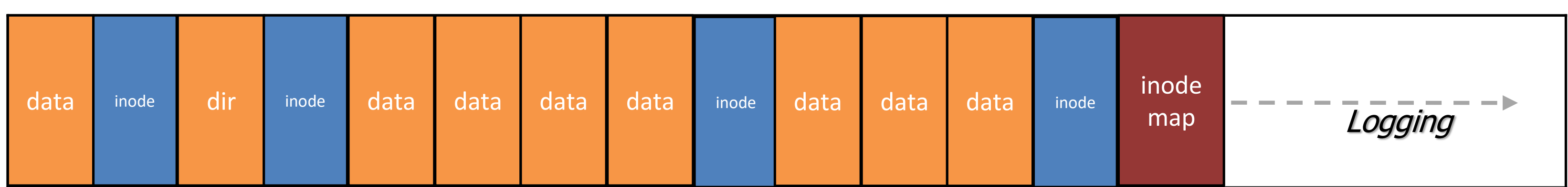
How to manage the free space on disk

- **Goal is to maintain large free extents for writing new data**
- **Free space will be fragmented into many small extents**
  - Corresponding to the files that were deleted or overwritten
- **Copy and Compact the live data, but the log is threaded on a segment-by-segment basis**



Cleaning mechanism

- **The process of copying live data out of a segment is called segment cleaning**
  - Step 1. Read a number of segments into memory
  - Step 2. Identify the live data
  - Step 3. Write the live data back to a smaller number of clean segments
- **Following information must be identified *to update the file's inode to point to the new location of the block***
  - Which data is live ?
  - Which file does each block belongs to?
  - Where is the block located in the file?



Cleaning mechanism (Cont.)

- **Sprite LFS solves both of these problems by writing a *segment summary block* as part of each segment.**

- **Segment summary block identifies each piece of information that is written in the segment**
  - For each file data block the summary block contains *the file number(inode number)* and *block number(offset)* of the block
  - It is also used to distinguish live blocks from those that have been overwritten or deleted

Is this block live?

[3] See where the 2th block of this file is on disk

[1] Read the block's inode number(N

d where N lives

Not Same! It's not live block

Segment Summary Block | data | inode | dir | inode | data [0] | data [1] | data [2] | data [3] | Inode N | data [1]' | inode map | data [2]' | data [3]' | Inode N' | inode map'

invalidated

Segment Cleaning Policies

- **Policies Issues**

1. Which segments to clean?
   - Greedy / Cost-benefit cleaning policy

2. When to clean?
   - When the number of free segments falls below a certain threshold

3. How many segments to clean?
   - Few tens of segments at a time, more segments cleaned at a time, the more opportunities to rearrange

4. How to group live blocks be grouped while cleaning?
   - Sort the blocks by the time they were last modified and group blocks of similar age into new segments

Segment Cleaning Policies – Cleaning cost Metric

- **Implementation**
  - The Write-cost Metric - A way of comparing cleaning policies

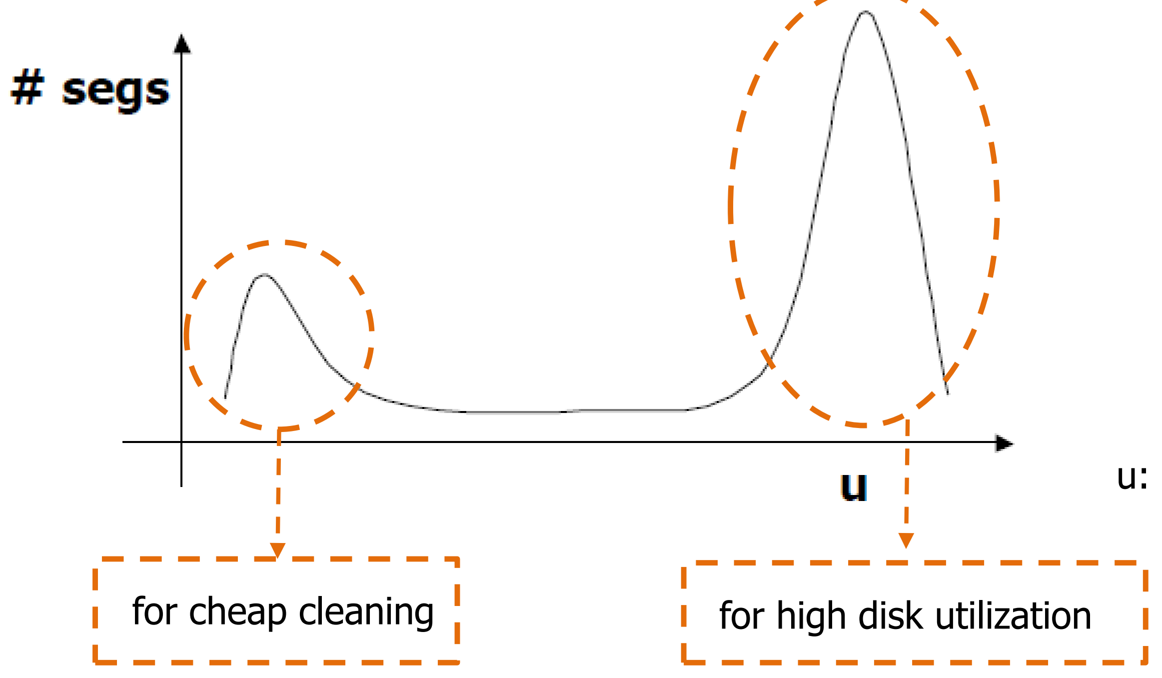
  - Write-cost : $\frac{read\ segs + write\ live + write\ new}{new\ data\ written} = \frac{\cancel{N+N*u+N*(1-u)}}{\cancel{new\ data\ written}} = \frac{2}{1-u}$

u : utilization of the segments and $0 \le u < 1$
N : # of segments participate in cleaning

- **LFS provide a cost-performance trade-off**
  - If disk capacity utilization is increased, storage costs are reduced but so is performance

Cleaning Goal

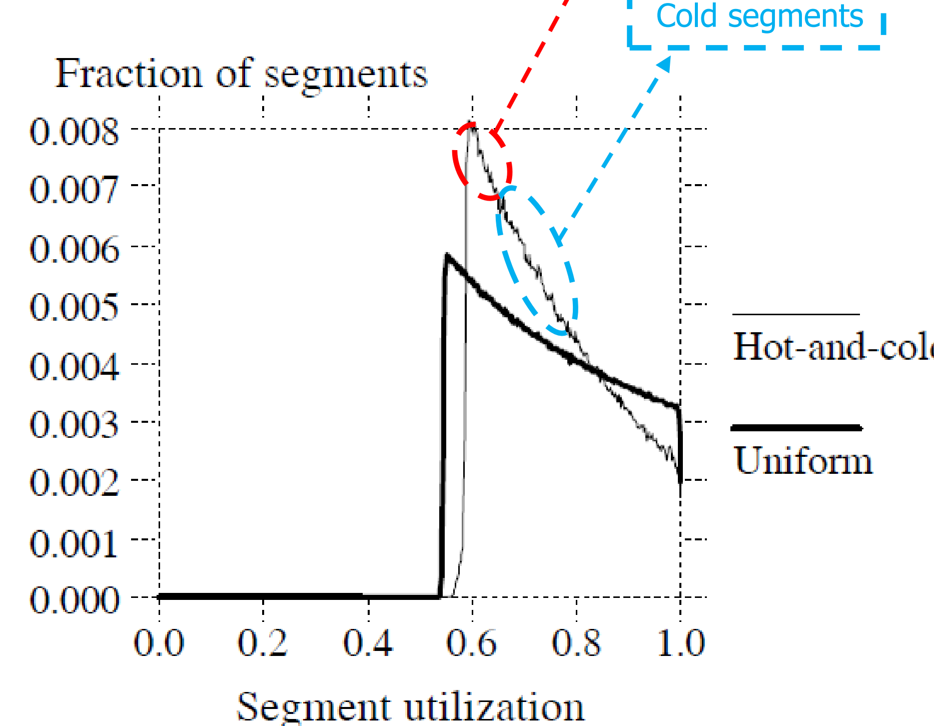
- **The Key to achieve high performance at low cost**
  - Small # of low-utilized segments : cleaner can always find easy segments to clean
  - Large # of high-utilized segments : disk is well utilized



u: utilization of segment, $0 \le u < 1$

Segment Cleaning Policy : Greedy Policy

- **Greedy policy :**
  - Always chose the least-utilized segments to clean
- **LFS uniform :**
  - Uniform pick random files to overwrite
- **LFS hot-and-cold:**
  - Hot-cold workload(90% of the updates to 10% of files)



- **Greedy is not creating bimodal distribution!**

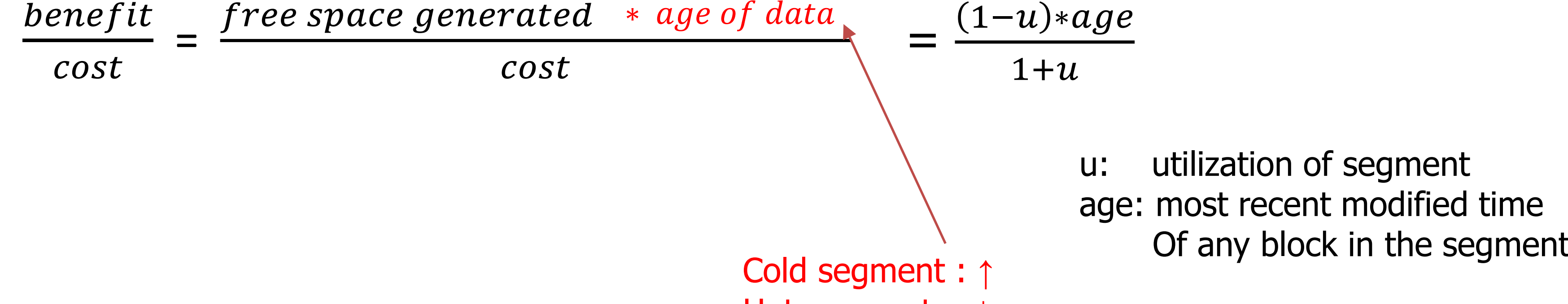
why so clustered?

1. **Hot segments join in cleaning again soon after cleaning**
2. **Cold segments tend to linger just above the cleaning point**
   Cold segments is not cleaned until low utilization

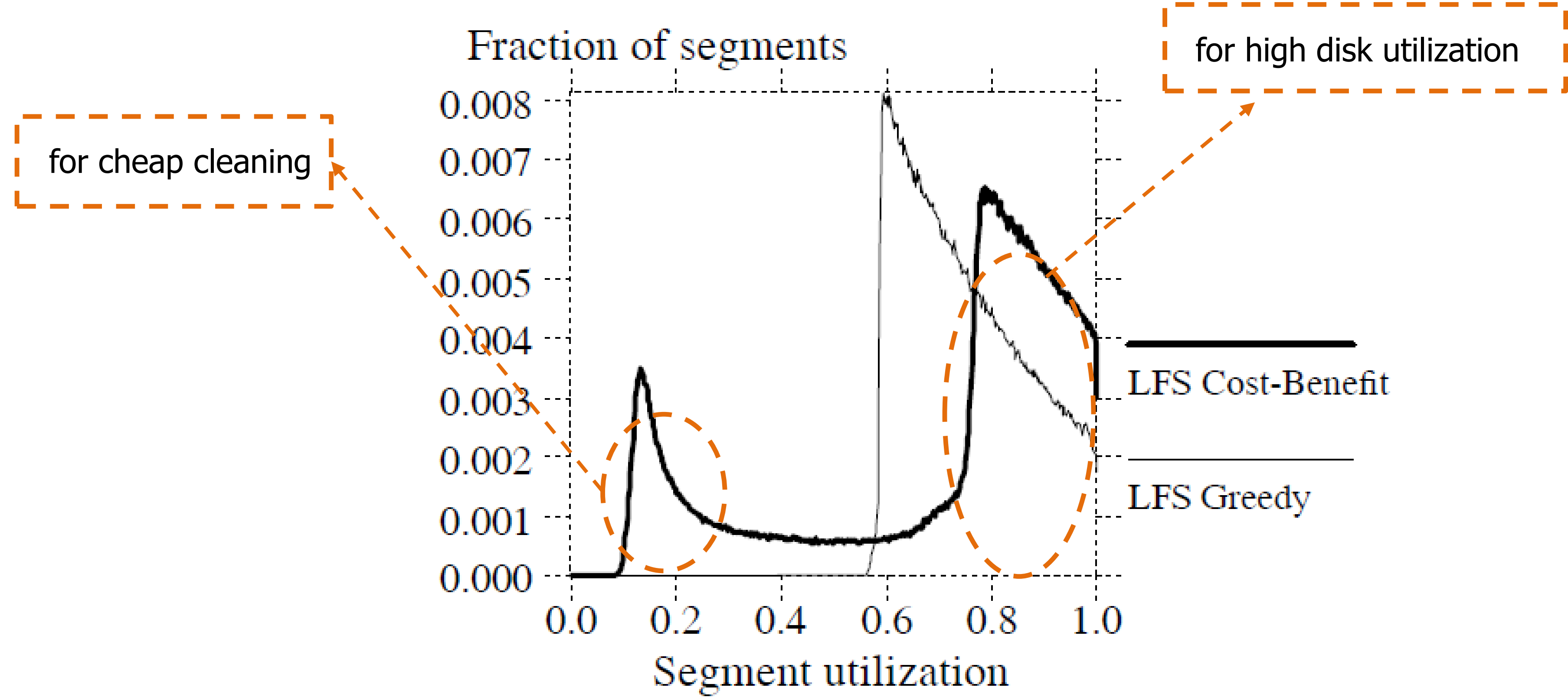
Segment Cleaning Policy : Cost-benefit

- **Goal**
  - Allow cold segments to be cleaned at a much higher utilization than hot segments

- **Cleaner policy :** $\boldsymbol{cost\text{-}benefit}$



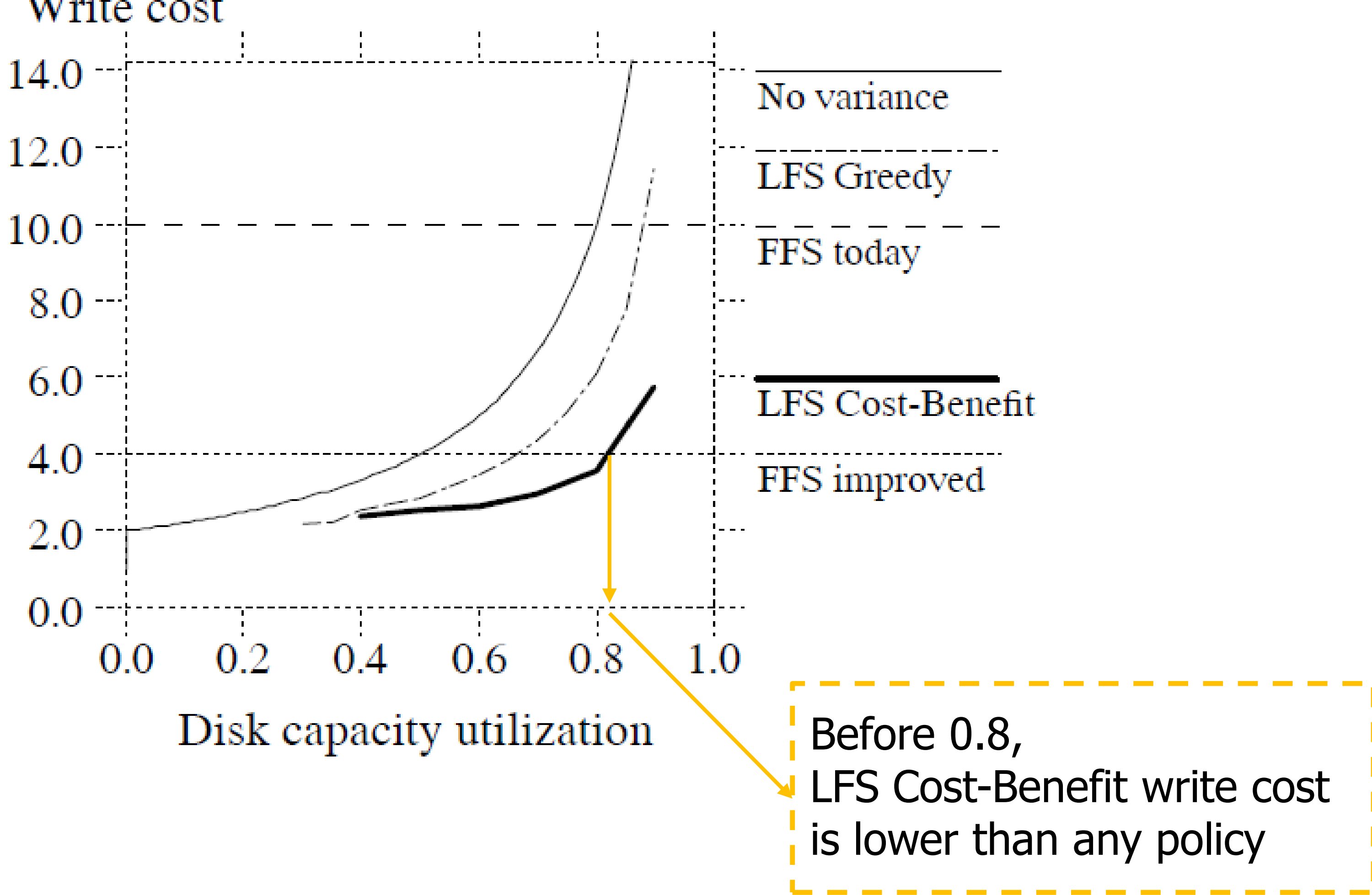
Effect of Cost/Benefit Policy



Prohibit from Only-cleaning of host segments

- Cold segments cleaned at around 75% utilization
- Hot segments cleaned at around 15% utilization

Cost-benefit Analysis



Before 0.8,
LFS Cost-Benefit write cost
is lower than any policy

Crash Recovery

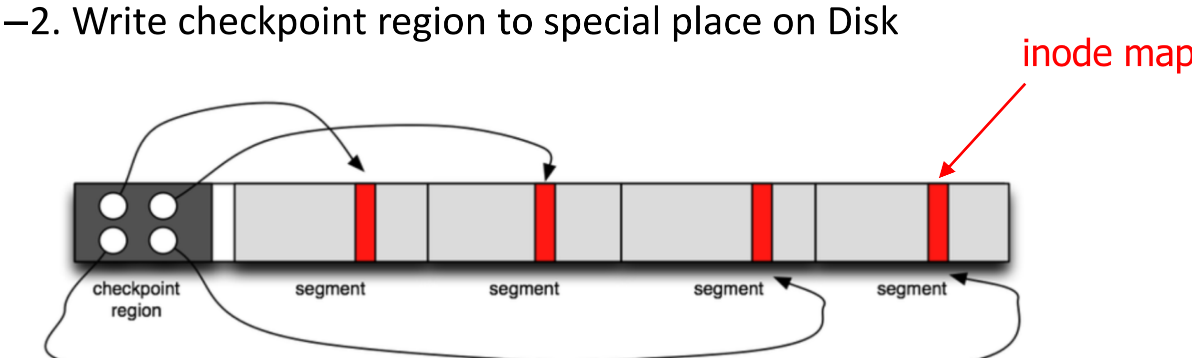
- **Crash in other FS of UNIX**
  - Disk may be in inconsistent state
    - File created but directory is not updated

- **In LFS**
  - Just look at end of log
  - 1. Checkpoint(define consistent state of file system)
  - 2. Roll-forward(to recover information written since the last checkpoint)

Checkpoints

- **A checkpoint is a position in the log where all file systems structures are consistent**

- **Creation of a checkpoint:**
  - –1. Write out all modified info to log, including metadata
  - –2. Write checkpoint region to special place on Disk



- **On reboot, read checkpoint region to initialize main-memory data structures(ex. Inode)**

Roll-forward

- **Goal : try to recover as much data as possible**
  - Recovering to latest checkpoint would result in loss of too many recently written data
- **During roll-forward Sprite LFS:**
  - 1. Uses information in segment summary blocks to recover recently-written file data.
  - 2. When summary block indicates presense of a new I-node, update I-node map read from checkpoint
  - 3. Incorporate the file's new data blocks into recovered file system

Evaluation

- **Use a collection of small benchmark programs to measure the best-case performance of Sprite LFS**
  - Sprite LFS vs SunOS 4.0.3

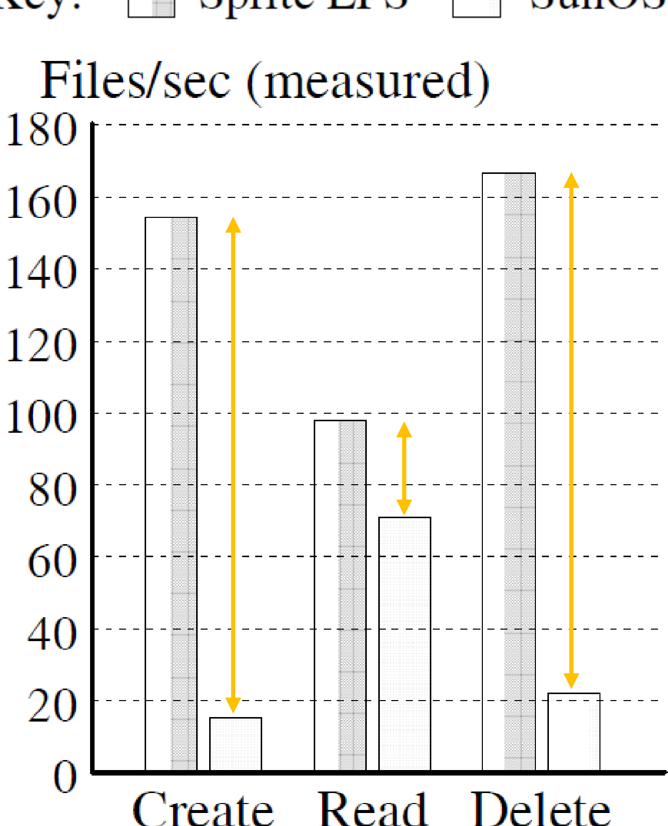
	Sprite LFS	SunOS
Workstation	Sun-4/260	
Main memory	32MB	
Disk	Capacity: 300MB Bandwidth: 1.3MB/s Seek time: 17.5ms (avg.)	
Block size	4KB	8KB
Segment size	1MB	X

Small-File Performance with no cleaning

- Step 1. Create 10000 1KB files
- Step 2. Read them back in the same order as created
- Step 3. Delete them

Speed Metric : the number of files per second

Sprite LFS is much better

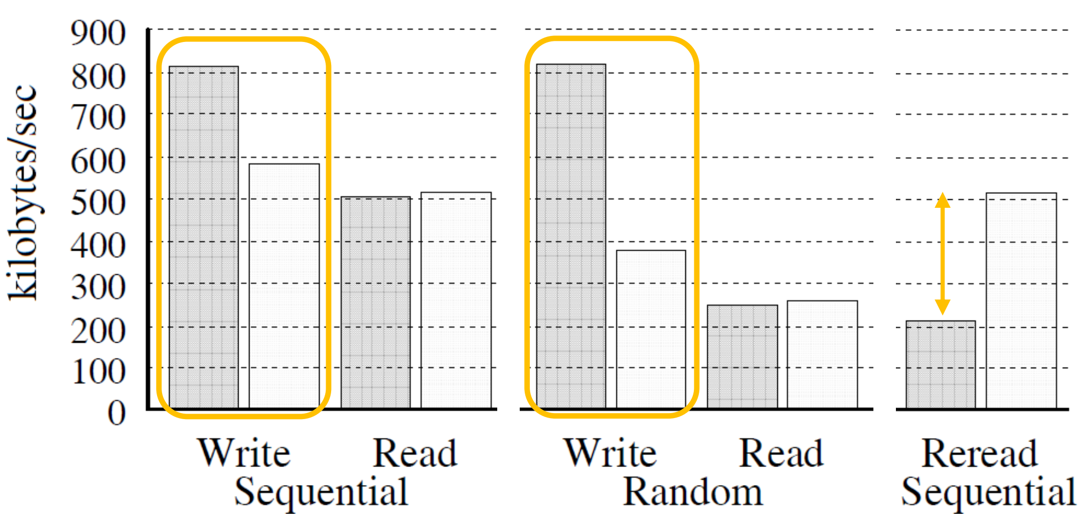


Large-File Performance with no cleaning

- **Step 1. Create 100MB file with sequential writes**
- **Step 2. Read the file back sequentially**
- Step 3. Writes 100MB randomly to the existing file
- Step 4. Read 100MB randomly from the file
- Step 5. Read the file sequentially again

Corrupt file block's order

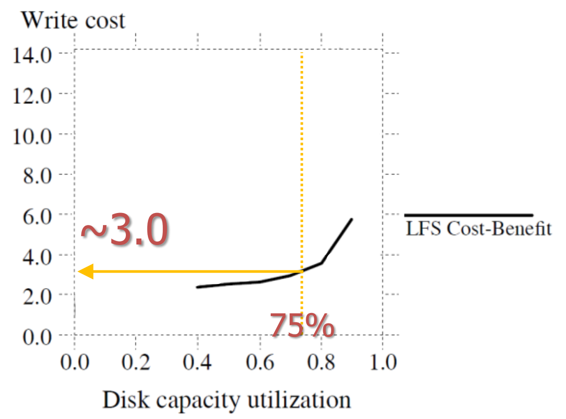
Sprite LFS requires more seeks than **SunOS**
Because the order of the file blocks is more chaotic



Sprite LFS SunOS

Cleaning overheads

- **Record statistics about <u>production LFS</u> over a period of 4 months**
  - Start-up effects is eliminated



Cleaning overheads limit the long-term write performance to about 70% of the maximum sequential write bandwidth

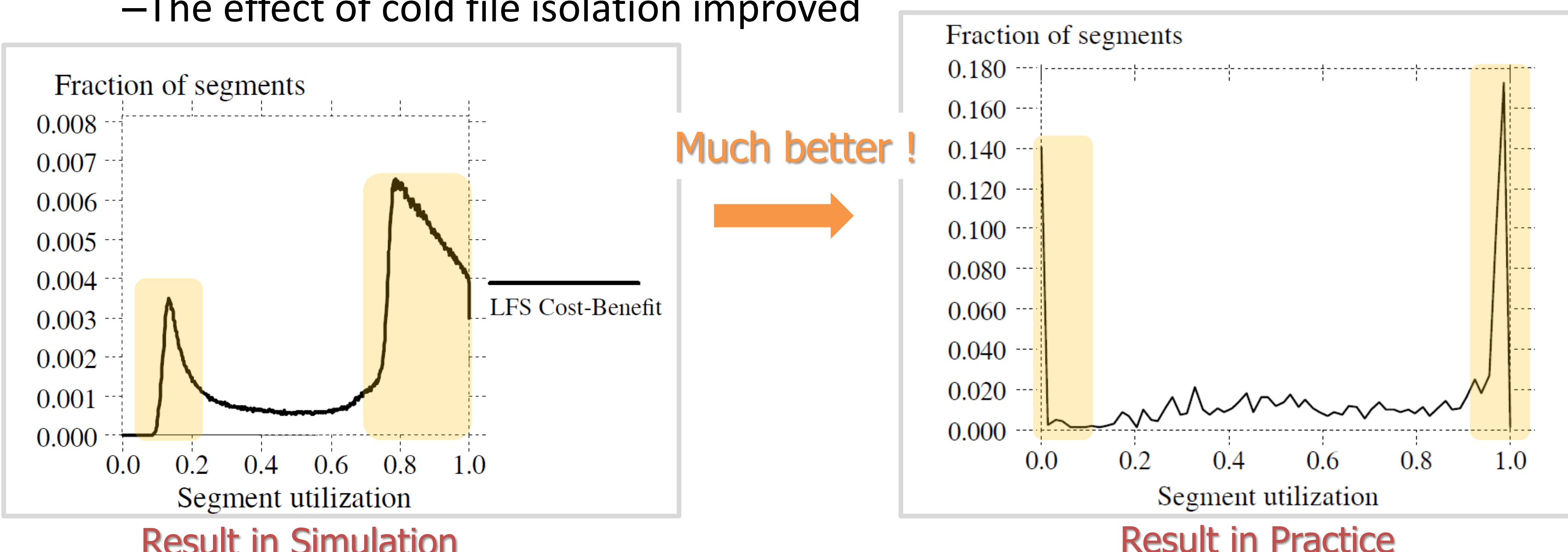
Better than simulation result

Write cost in Sprite LFS file systems

File system	Disk Size	Avg File Size	Avg Write Traffic	In Use	Segments Cleaned	Segments Empty	u Avg	Write Cost
/user6	1280 MB	23.5 KB	3.2 MB/hour	75%	10732	69%	.133	1.4
/pcs	990 MB	10.5 KB	2.1 MB/hour	63%	22689	52%	.137	1.6
/src/kernel	1280 MB	37.5 KB	4.2 MB/hour	72%	16975	83%	.122	1.2
/tmp	264 MB	28.9 KB	1.7 MB/hour	11%	2871	78%	.130	1.3
/swap2	309 MB	68.1 KB	13.3 MB/hour	65%	4701	66%	.535	1.6

Cleaning overheads (Cont.)

- **In practice, there are a substantial number of longer files**
  - They tend to be written and deleted as a whole
  - In the best case, where a file is much longer than a segment, deleting the file will produce one or more totally empty segments
- **In practice, cold segments are much colder than in the simulations**
  - The effect of cold file isolation improved

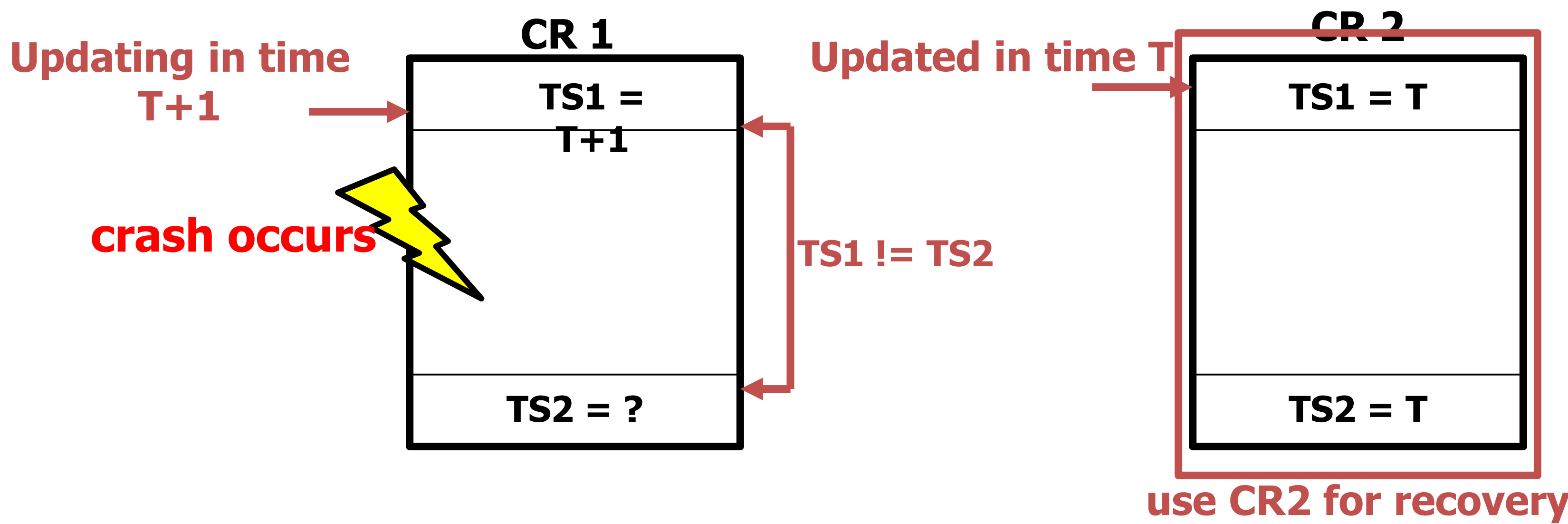


Conclusion

- **LFS introduces a new approach to updating the disk**
  - Writes to an unused portion of the disk, then later reclaims that old space
- **This approach enables highly efficient writing**
  - Gather all updates into an in-memory segment and write them out together
- **It works well for small files accesses as well as large files accesses**
- **LFS make it possible to take advantage of faster processors**

Crash Recovery

- **Crashes when write to the Checkpoint Region**
  - LFS keeps two CRs, writes to them alternately
  - LFS implements a careful protocol when updating
    1. Write out a header(with timestamp)
    2. Write out the body of the CR(information including imap)
    3. Finally writes one last block (with timestamp)



Data structure	Purpose	Location	Section
Inode	Locates blocks of file, holds protection bits, modify time, etc.	Log	3.1
Inode map	Locates position of inode in log, holds time of last access plus version number.	Log	3.1
Indirect block	Locates blocks of large files.	Log	3.1
Segment summary	Identifies contents of segment (file number and offset for each block).	Log	3.2
Segment usage table	Counts live bytes still left in segments, stores last write time for data in segments.	Log	3.6
Superblock	Holds static configuration information such as number of segments and segment size.	Fixed	None
Checkpoint region	Locates blocks of inode map and segment usage table, identifies last checkpoint in log.	Fixed	4.1
Directory change log	Records directory operations to maintain consistency of reference counts in inodes.	Log	4.2

Crash Recovery

Sprite LFS recovery time in seconds			
File Size	File Data Recovered		
	1 MB	10 MB	50 MB
1 KB	1	21	132
10 KB	< 1	3	17
100 KB	< 1	1	8

Other overheads

Sprite LFS /user6 file system contents		
Block type	Live data	Log bandwidth
Data blocks*	98.0%	85.2%
Indirect blocks*	1.0%	1.6%
Inode blocks*	0.2%	2.7%
Inode map	0.2%	7.8%
Seg Usage map*	0.0%	2.1%
Summary blocks	0.6%	0.5%
Dir Op Log	0.0%	0.1%