The Design and Implementation of a Log-Structured File System

Mendel Rosenblum and John K. Ousterhout

Presented by Wenhao Tang, Xinzhao Wang

2020/12

Motivation

- CPU is fast, while **disk access is slow**.
- Most of the **disk bandwidth is wasted** by the current file system, especially in the case of many small-file accesses.
- Assumption: Main memories are used as caches for disk. → Disk read is relatively fast.
  - Disk traffic is dominated by **WRITE**

Problems of Current File System

- **Too many small accesses.**
  - Unix FFS physically separates different files.
  - Files are separate from metadata (attributes/inode, directory) corresponding to it.
- **Write synchronously.**
  - The application must wait for the write to complete.
- **Recovering from crash is slow.**
  - FS needs to scan the entire disk.
  - (Modern FS uses journaling.)

Main Idea

- Use main memory as a write buffer.
- Increase write performance by eliminating seeks.
  1. Buffering a sequence of file system changes in the file cache.
  2. Writing all the changes to disk sequentially in a single disk write operation (including files and metadata).

➡️ Convert the many small synchronous random writes of traditional file systems into large asynchronous sequential transfers. (*Data is written as logs.*)

Two Issues

- **[READ] How to retrieve information from the log?**
  - The position of data will be changed after writting.
- **[WRITE] How to manage the free space on disk?**
  - We want to maintain large extents of free space for writing new data.
  - Delete and change will cause fragments.

File location

- For each ***file*** there is an ***inode*** structure containing its attributes and address.
  - In Unix FFS the location of inode is fixed; but in LFS they are written to the log.
- LFS uses an ***inode map*** to maintain the location of inode. (Usually cached.)
- Each disk has a ***fixed checkpoint region*** to save the location of all inode map blocks.
- The process:

fixed checkpoint region

➡️ inode map (use the identifying number of a file)

➡️ inode

➡️ file

File location

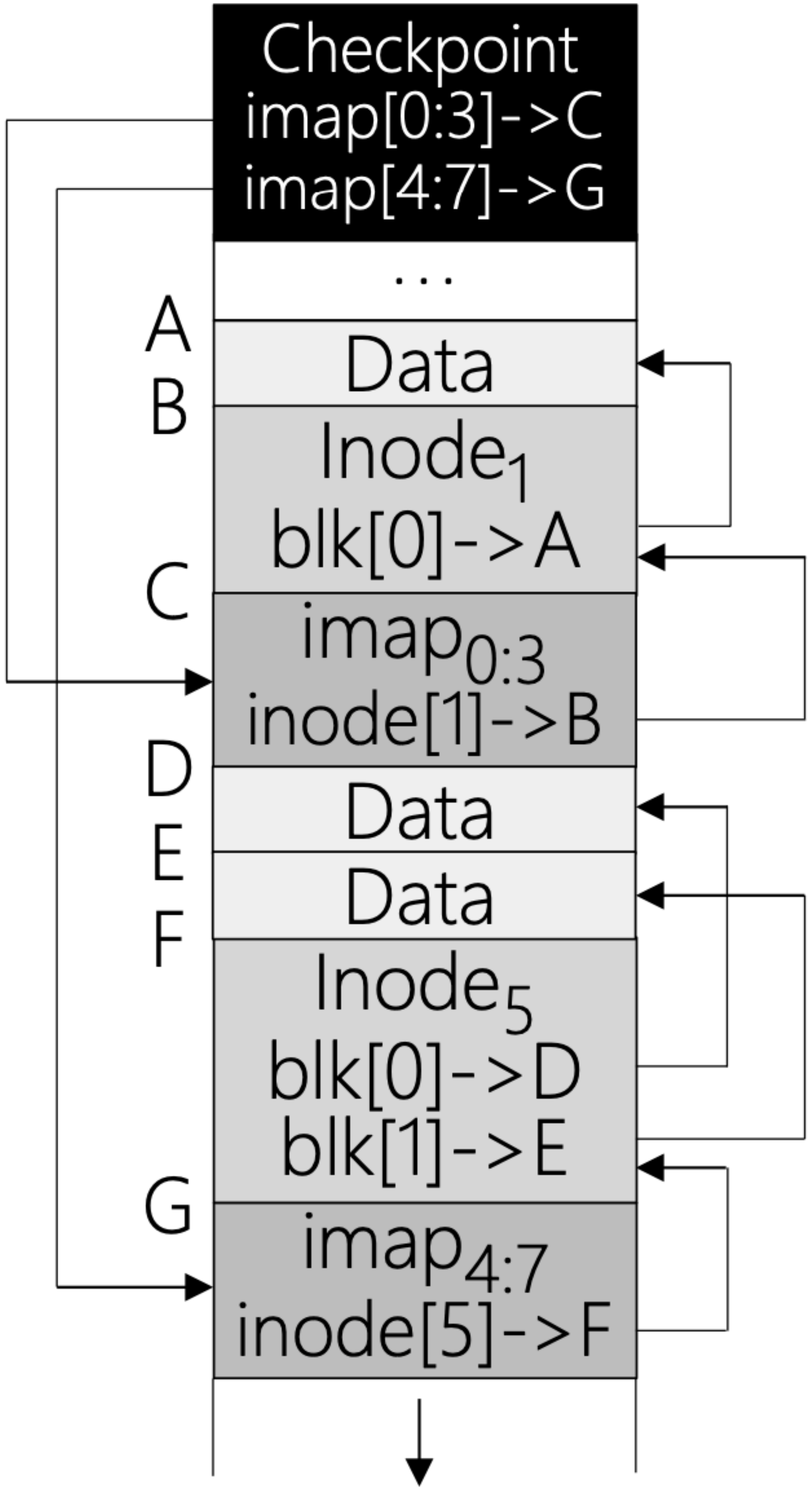
- For each ***file*** there is an ***inode*** structure containing i
  - In Unix FFS the location of inode is fixed; but in LFS th
- LFS uses an ***inode map*** to maintain the location of i
- Each disk has a ***fixed checkpoint region*** to save the l blocks.
- The process:

fixed checkpoint region

➡️ inode map (use the identifying number of a file)

➡️ inode

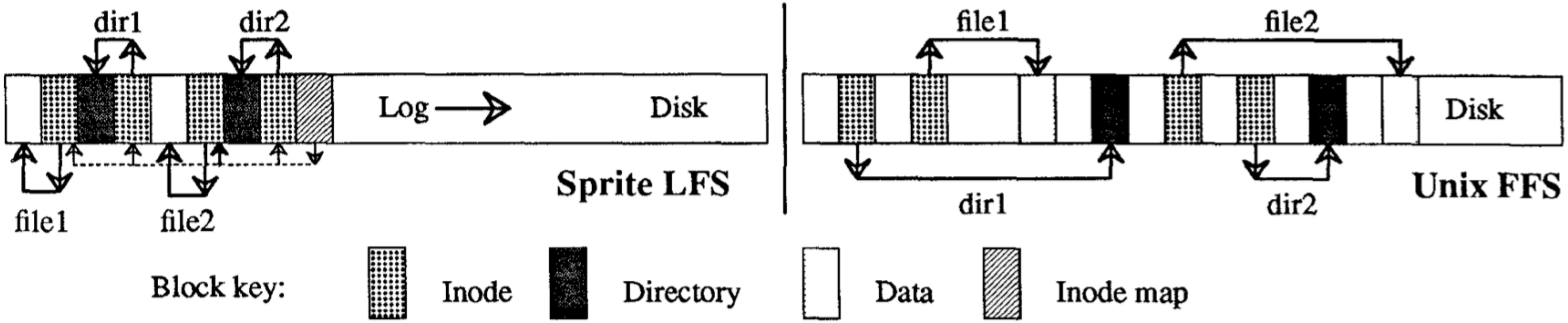
➡️ file



File location

Example

- Example: creating two new files in different directories.



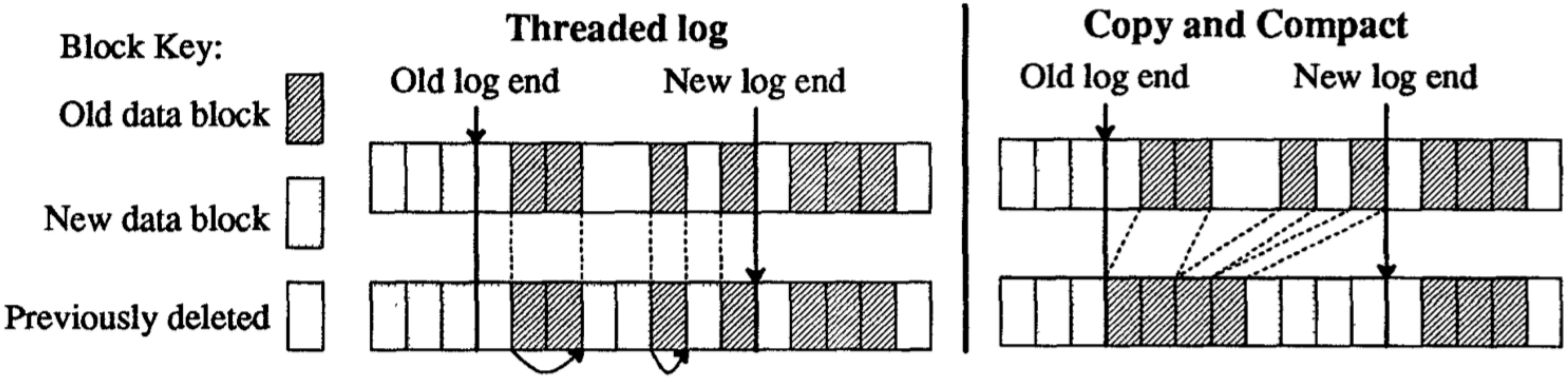
File location

Question

- **The address of inode will be changed after writing. Should directories be changed when writing files?**
  - No.
  - Directories store a mapping from file name to a unique file id (inode number). LFS uses this id to get the entry in the inode map.

Free Space Management

- **How to deal with fragments?**
  - **Theading**: thread the log through the free fragments
  - **Copying**: copy live data out of the log
- LFS uses a combination of threading and copying.



Free Space Management

Segments

- The disk is divided into *large fixed-size extents* called **segments**.
  - Copying in segments.
  - Threading between segments.
- The segment size is chosen large enough that the transfer time to read or write a whole segment is much greater than the cost of a seek to the beginning of the segment

Free Space Management

Segment cleaning mechanism

- The process of copying live data out of a segment is called **segment cleaning**.
- Process:
  - Read segments into memory
  - Identify live data and move them into clean segments
  - Mark the previous segments clean.
- LFS uses a **segment summary block** to identify the contents of segment.
- How to know which block is dead?
  - keeping a version number in the inode map entry for each file and in the segment summary block for each block;
  - increment whenever the file is deleted or truncated to length zero.

Free Space Management

Segment cleaning policies

- Four Problems:
    1. When should the segment cleaner execute? (A fixed threshold)
    2. How many segments should it clean at a time? (A fixed threshold)
    3. Which segments should be cleaned?
    4. How should the live blocks be grouped?
- Use **write cost** to compare cleaning policies.

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

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

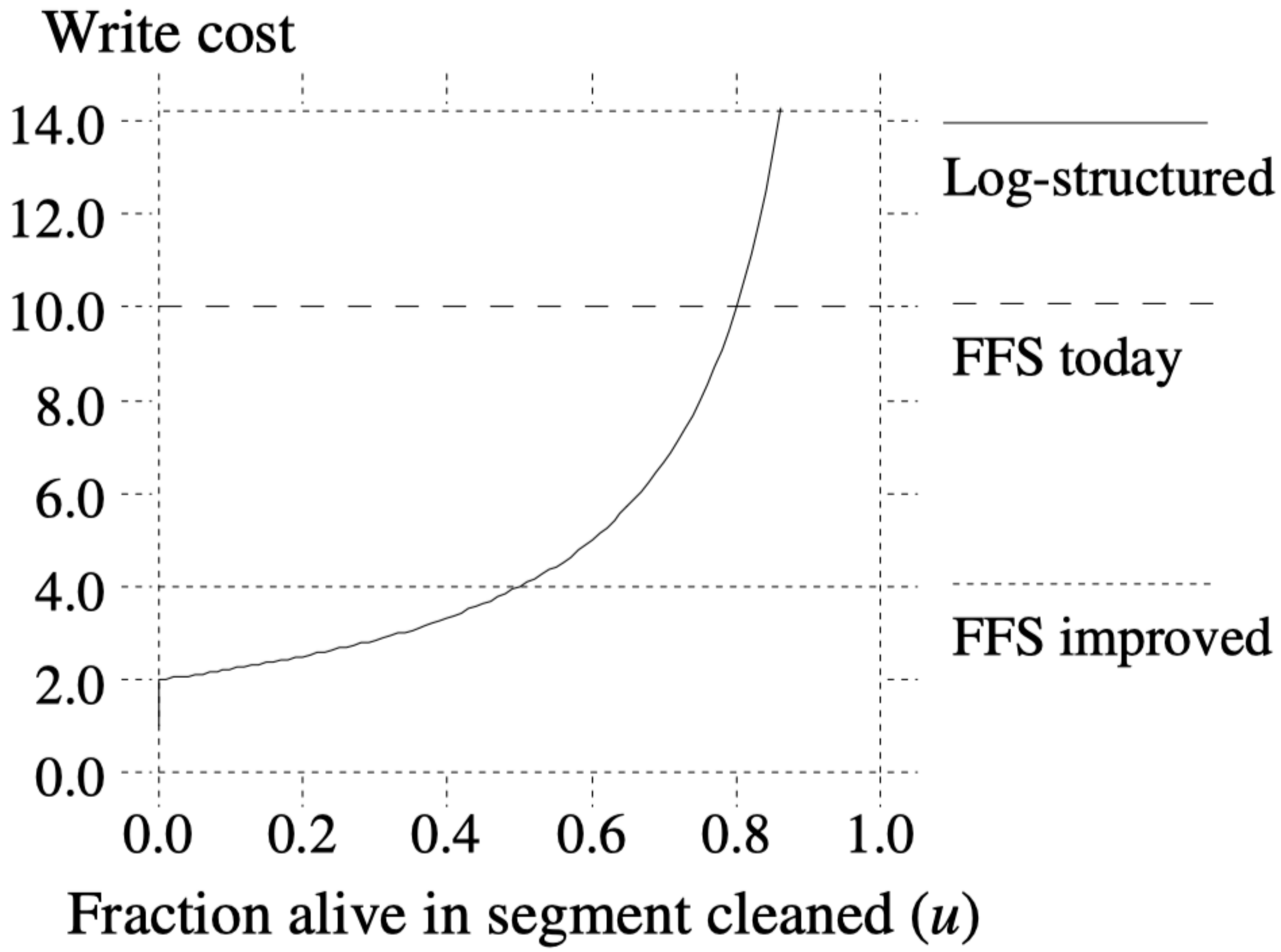
$$= \frac{N + N^{*}u + N^{*}(1-u)}{N^{*}(1-u)} = \frac{2}{1-u}$$

u is the utilization of the segments

Free Space Management

Segment cleaning policies

- cost-performance tradeoff
  - Write cost increases as utilization increases.



FFS with logging, delayed writes, and disk request sorting

Simulation Results

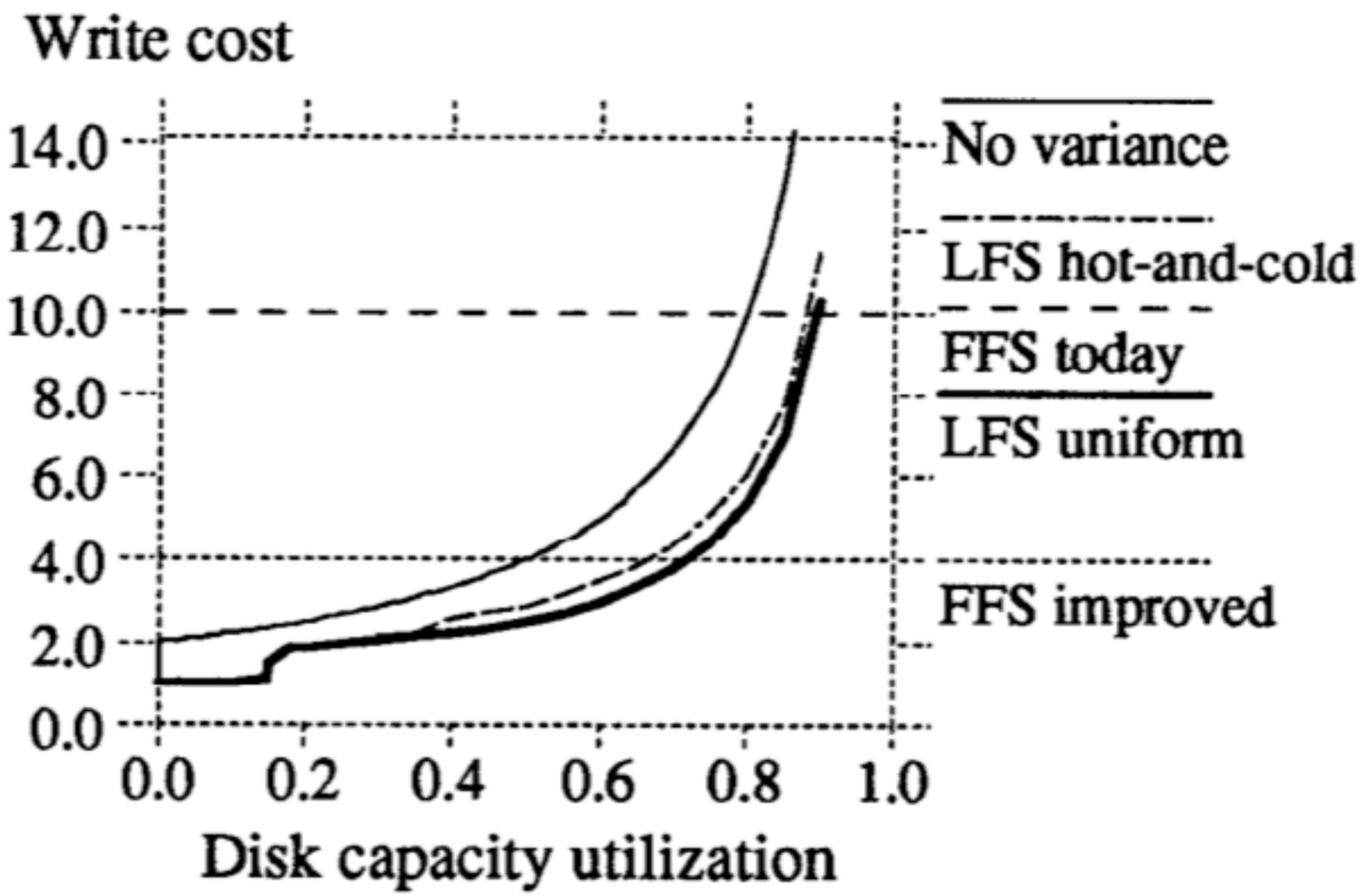
Simulation Results

- The simulator models a file system as a fixed number of 4-kbyte files, with the number chosen to produce a particular overall disk capacity utilization
- Two pseudo-random access patterns
  - Uniform: Each file has equal likelihood of being selected in each step
  - Hot-and-Cold: Files are divided into two groups. Files in “hot group” are accessed frequently and files in “cold group” are rarely accessed
  - No read traffic is modeled

Simulation Results

Simulation Results

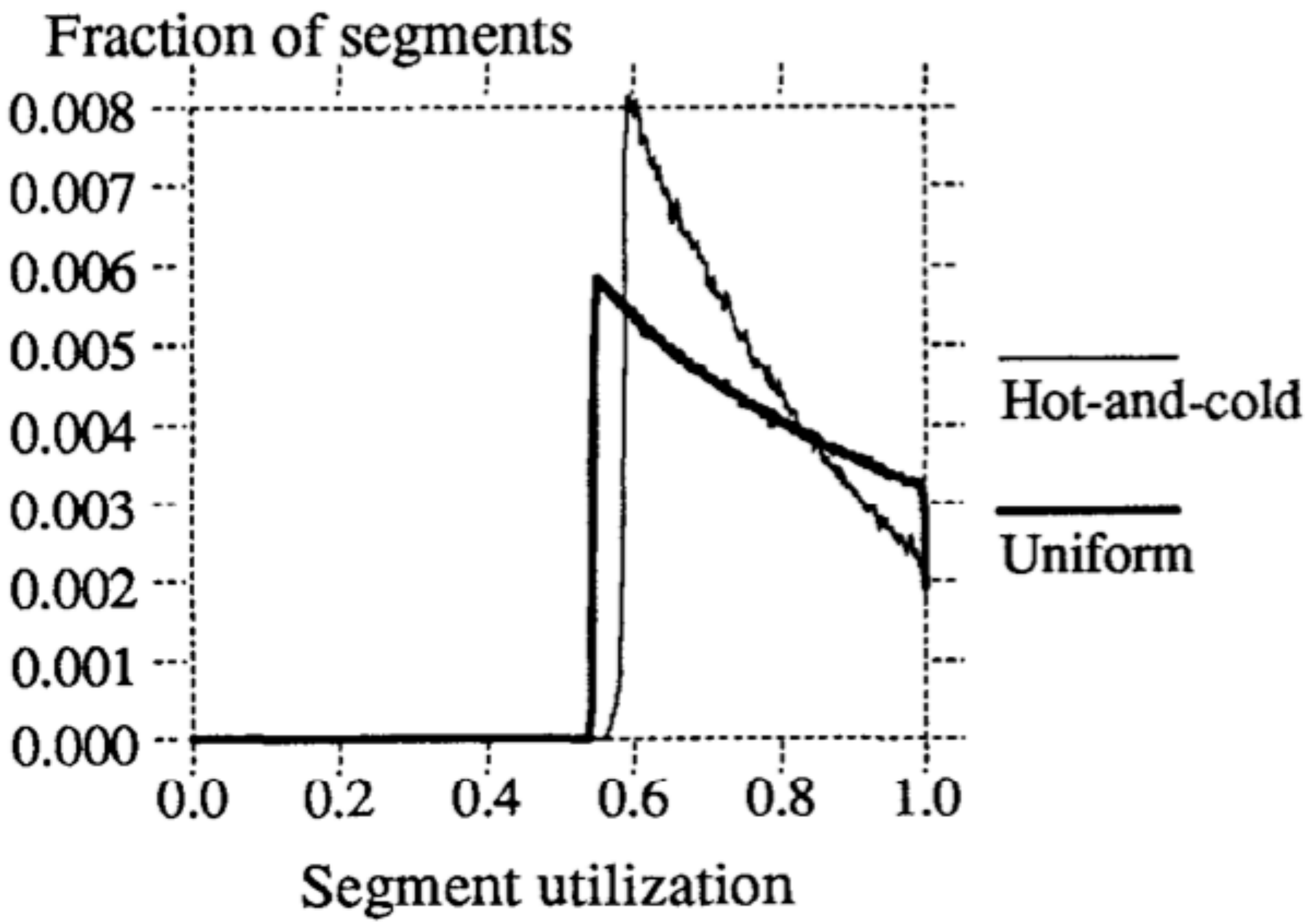
- Why better locality leads to worse performance?



Simulation Results

Simulation Results

- Why better locality leads to worse performance?



Simulation Results

Cost-Benefit Policy

- Stability need to be considered
- Use age of data to estimate stability

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

- **Segment Usage Table**: For each segment, the table records the number of live bytes in the segment and the most recent modified time of any block in the segment

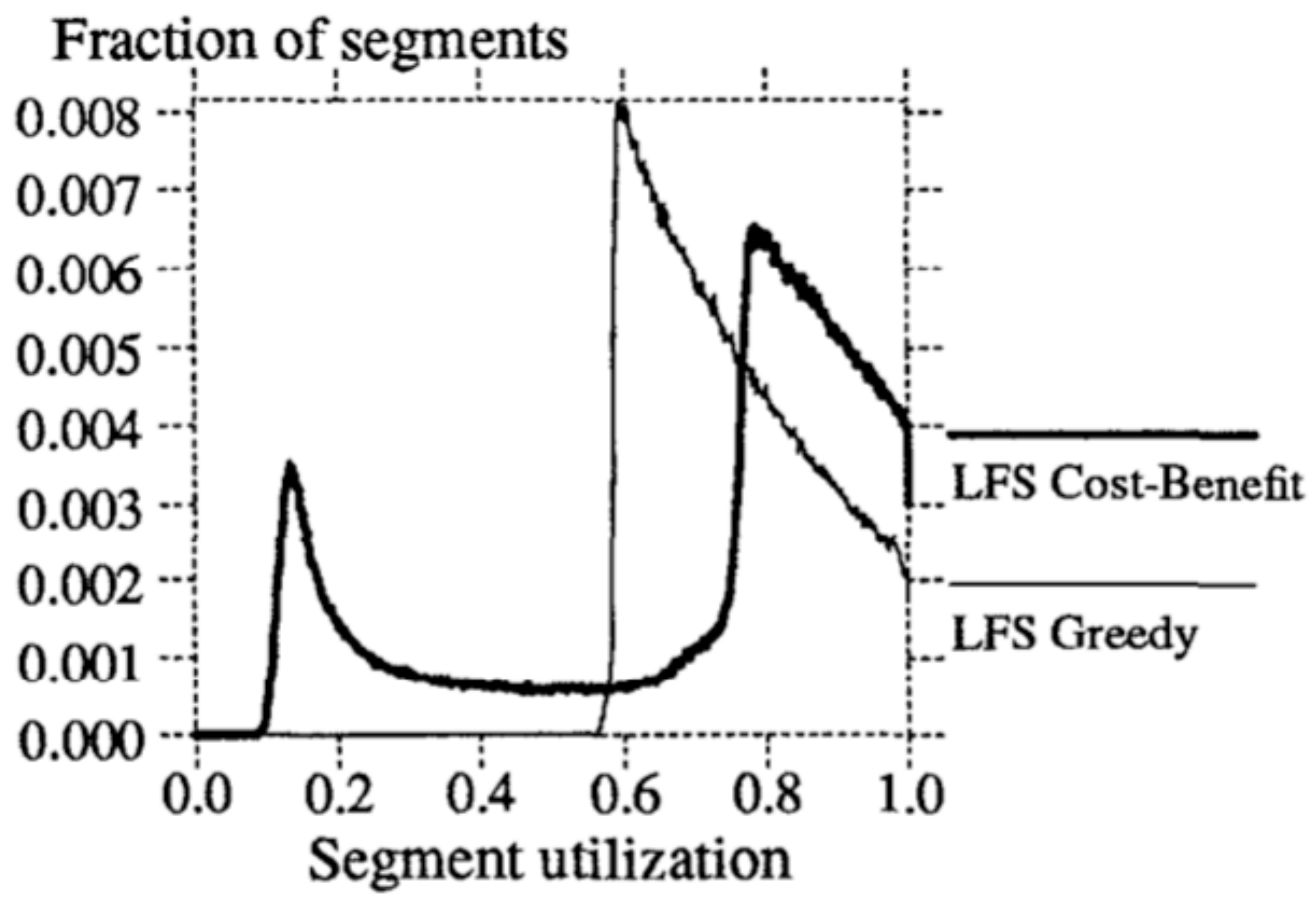


Figure 6 — Segment utilization distribution with cost-benefit policy.

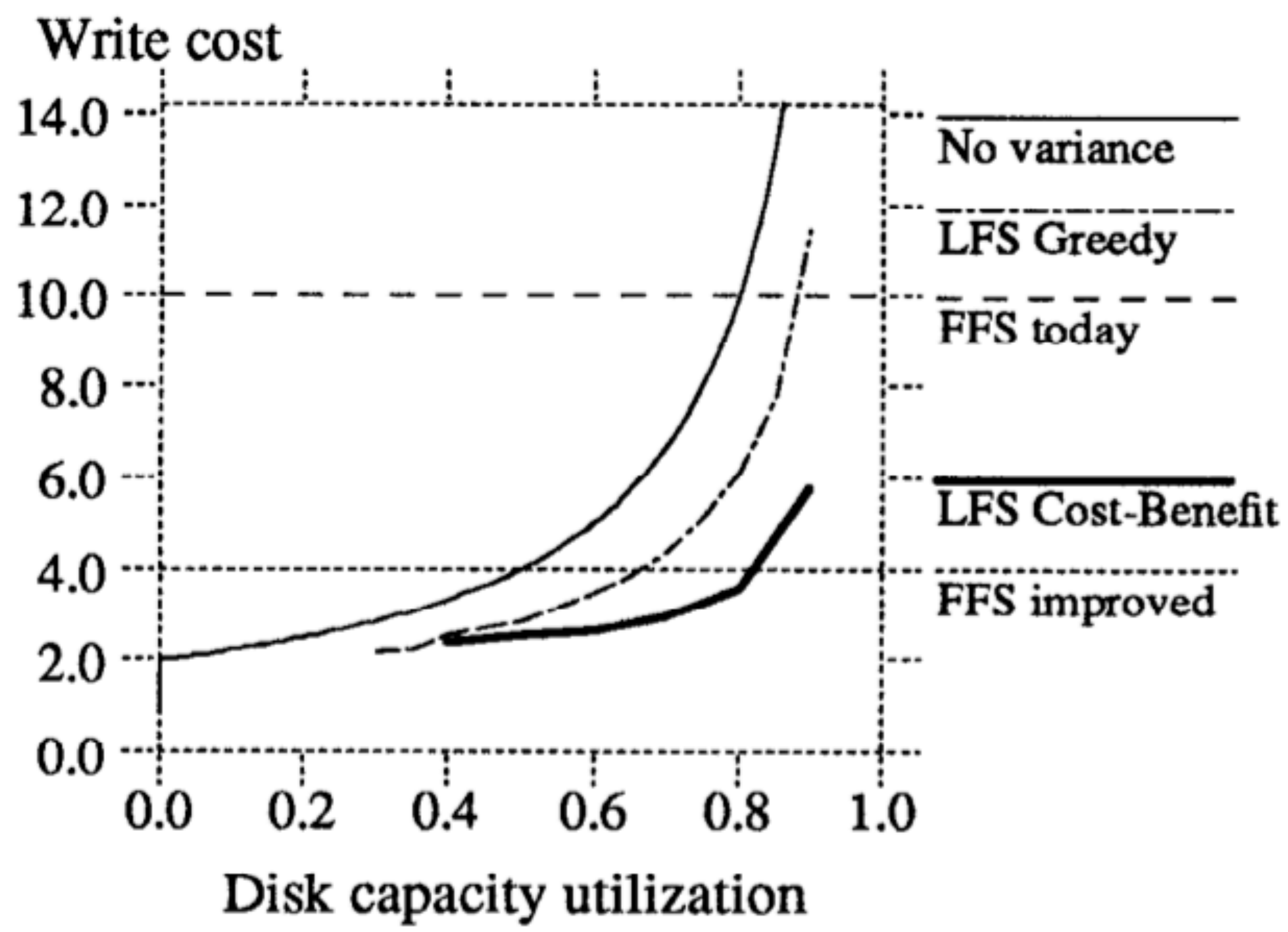


Figure 7 — Write cost, including cost-benefit policy.

Crash Recovery

Crash Recovery

- **Crash**: When a system crash occurs, the last few operations performed on the disk may have left it in an inconsistent state
- **Crash Recovery**: Operating system must review these operations in order to correct any inconsistencies
- **In Traditional File System**: In traditional Unix file systems without logs, the system cannot determine where the last changes were made, so it must scan all of the metadata structures on disk to restore consistency
- **In Log File System**: In a log-structured tile system the locations of the last disk operations are easy to determine they are at the end of the log.

.

Crash Recovery

Checkpoint

- **Checkpoint:** Checkpoint is a position in the log at which all of the file system structures are consistent and complete
- Two-phase process to create a checkpoint:
    1. Write **ALL** modified information to the log
    2. Write a checkpoint region to a special fixed position on disk(it also contains a pointer to the last segment written)
- During reboot, Sprite LFS reads the checkpoint region and uses that information to initialize its main memory data structures

Crash Recovery

Roll Forward

- How to implement crash recovery using checkpoint?
  - Read the latest checkpoint and discard any data in the log after that checkpoint
  - Data written since the last checkpoint would be lost
- Roll Forward
  1. Inode: Update inode map
  2. Data Block: Ignore data blocks without an inode
  3. Update segment usage table
  4. Directory Operation Log: Guarantee directory operation log entry appears in the log before the corresponding directory block or inode
  5.

Thanks