Other techniques

- 1. Track-aligned Extents
- 2. Exploiting Disk Bandwidth for Small Files
- 3. Disk Shuffling
- 4. Log Approach

Other tehniques

- **1. Track-aligned Extents**

1. Track-aligned Extents

- **Based on disk-specific knowledge about disk data layout**

- **allocating** and **accessing**
- **related data**
- on **disk track boundaries**

- **system avoids most**
  - **rotational latency**
  - and
  - **track crossing overheads**.

- **Avoiding these overheads**
  - **can increase disk access efficiency**
  - **by up to 50**% for **mid-sized requests** (100–500 KB).

Head Switch time

- **Head switch.** A **head switch occurs**
  - **when a single request accesses a sequence of LBNs**
  - **whose on-disk locations span two tracks.**

Disk	Year	RPM	Head Switch	Avg. Seek	512B Sectors per Track	Number of Tracks	Capacity
HP C2247	1992	5400	1 ms	10 ms	96–56	25649	1 GB
Quantum Viking	1997	7200	1 ms	8.0 ms	216–126	49152	4.5 GB
IBM Ultrastar 18 ES	1998	7200	1.1 ms	7.6 ms	390–247	57090	9 GB
IBM Ultrastar 18LZX	1999	10000	0.8 ms	5.9 ms	382–195	116340	18 GB
Quantum Atlas 10K	1999	10000	0.8 ms	5.0 ms	334–224	60126	9 GB
Seagate Cheetah X15	2000	15000	0.8 ms	3.9 ms	386–286	103750	18 GB
Quantum Atlas 10K II	2000	10000	0.6 ms	4.7 ms	528–353	52014	9 GB

Table 1: **Representative disk characteristics.** Note the small change in head switch time relative to other characteristics.

- **Even compared to other disk characteristics,**
- **head switch**
- **time has improved little in the past decade.**

Zero-latency access

- **With zero-latency access support**, **disk firmware can read** the **N sectors, from** the **media into its buffers, in any order**
- **In the best case of reading exactly one track**,
  - the **head can start reading data as soon as the seek is completed**;
  - **no rotational latency is involved**
  - **because all sectors on the track are needed**.

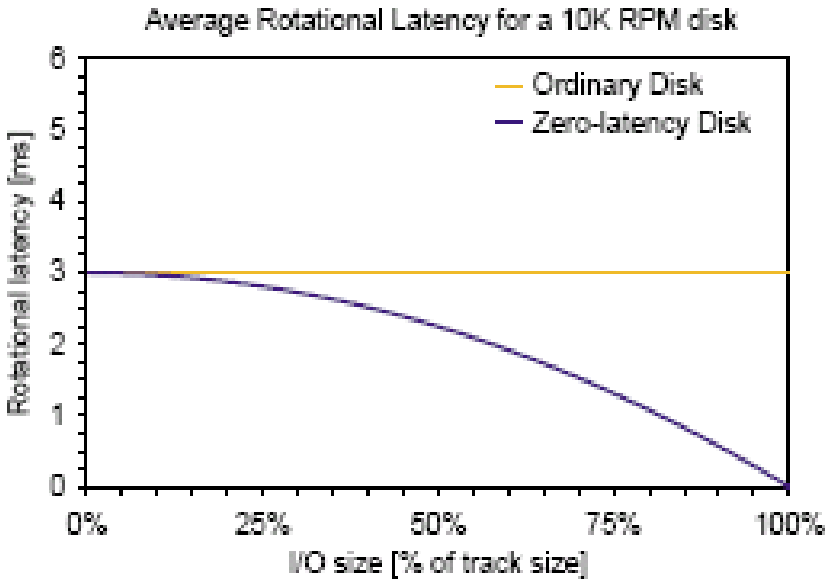


Figure 3: **Average rotational latency for ordinary and zero-latency disks as a function of track-aligned request size.** The request size is expressed as a percentage of the track size.

Timing

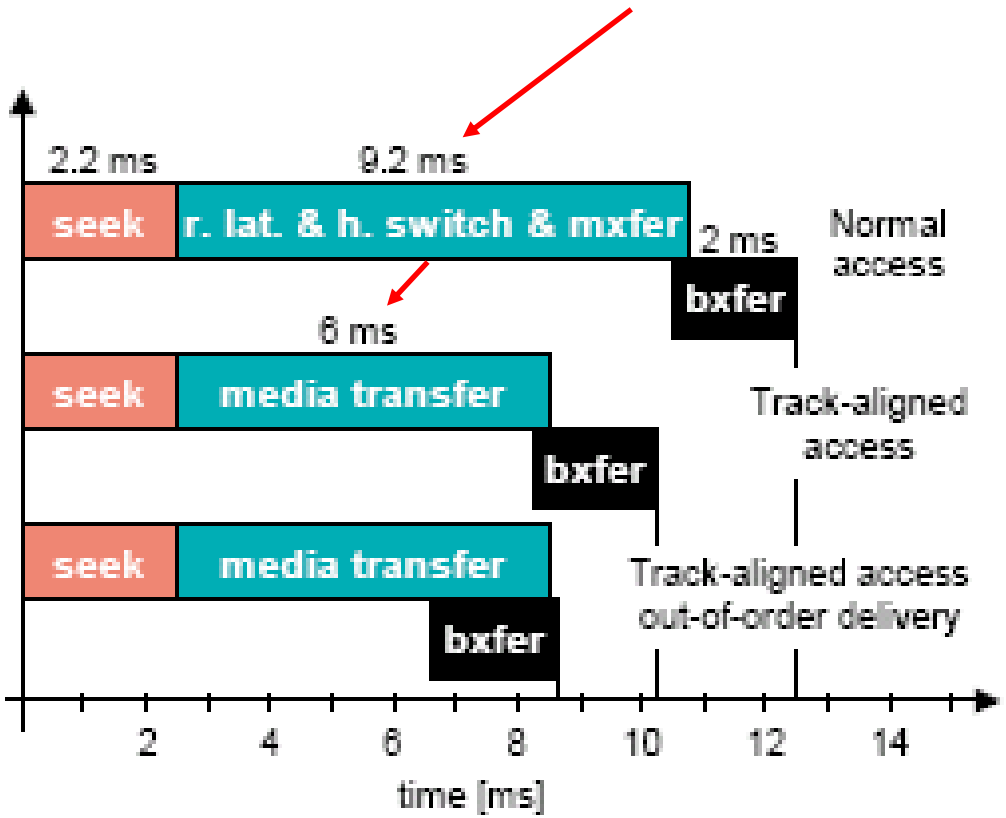


Figure 7: **Breakdown of measured response time for a zero-latency disk.** "Normal access" represents track-unaligned access, including seek, rotational latency (*r.lat.*), head switch, media transfer (*mxfer*), and bus transfer (*bxfer*). For track-aligned access, the in-order bus transfer does not overlap the media transfer. With out-of-order bus delivery, overlap of bus and media transfers is possible.

Implementation

- **Detecting track boundaries**.
  - This **task is more difficult than might be expected**, because of:
    1. **zoned recording**
    2. **media defect management**

- **set of changes needed**
- **to use track boundary knowledge**
- in an **existing file system.**
  - **Grouping** (for block-based FS, like ext3)
  - **Allocation of track-aligned extents**
  - **Very convenient for extent-based FS** (like XFS, NTFS)

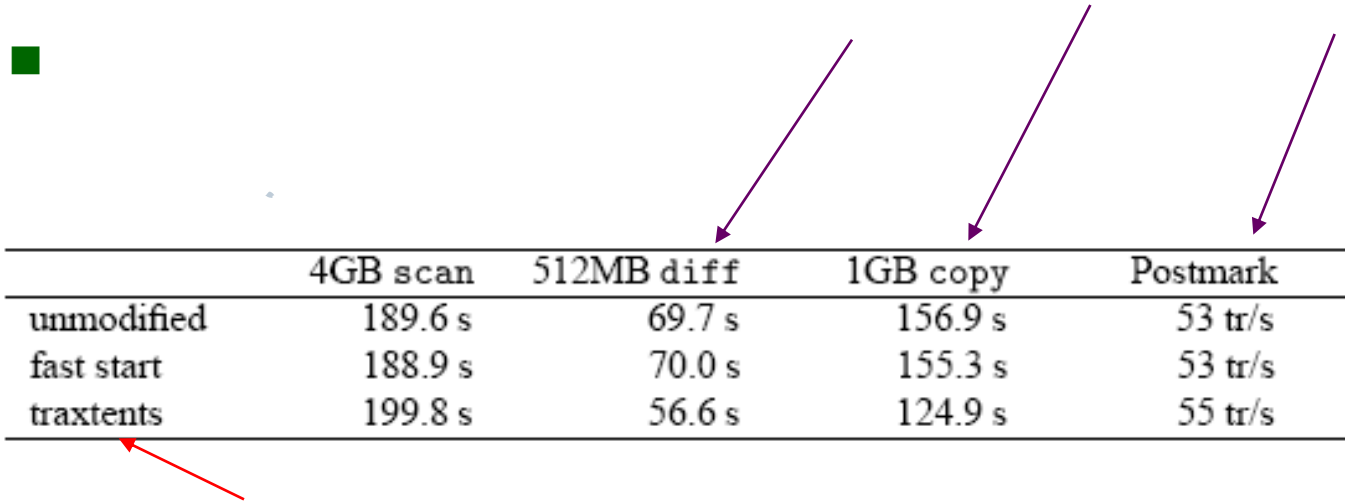
Performance improvements

- **For large-file workloads,**
  - a **version of FreeBSD's FFS** implementation
  - that **exploits traxtents**
  - **reduces application run times by up to 20%**
  - **compared to** the **original version.**

- A **video server** using **traxtent-based requests**
  - can support **56%** more **concurrent streams**
  - at the **same startup latency** and **buffer space**.

- **For LFS,**
  - **44% lower overall write cost**
  - for track-sized segments
  - **can be achieved**

Results



	4GB `scan`	512MB `diff`	1GB `copy`	Postmark	SSH-build	`head` *
unmodified	189.6 s	69.7 s	156.9 s	53 tr/s	72.0 s	4.6 s
fast start	188.9 s	70.0 s	155.3 s	53 tr/s	71.5 s	5.5 s
traxtents	199.8 s	56.6 s	124.9 s	55 tr/s	71.5 s	5.2 s

Table 2: **FreeBSD FFS results.** All but the `head` * values are an average of three runs. The individual run times deviate from their average by less than 1%. The `head` * value is an average of five runs and the individual runs deviate by less than 3.5%. Postmark reported the same number of transactions per second in all three runs for the respective FFS, except for one run of the unmodified FFS that reported 54 transactions per second.

Other techniques

- 2. Exploiting Disk Bandwidth
-
- for
- Small Files

2. Exploiting Disk Bandwidth for Small Files

- C-FFS (Co-locating Fast File System),
- **introduces 2 techniques,**
- **for exploiting disk bandwidth** for **small files** and **metadata**
  - **1. embedded inodes**
  - **2. explicit grouping**

- **1. embedded inodes**
  - **inodes for most files**
    - **are stored in the** directory
    - **with the corresponding name**, (in FCB)
  - removing a physical level of indirection
    - without sacrificing the logical level of indirection.

- **2. explicit grouping**
  - **data blocks of multiple small files named** by a **given directory are**
    - **allocated adjacently** and
    - **moved to and from** the **disk**
    - **as a unit in most cases**.

Embedded inodes

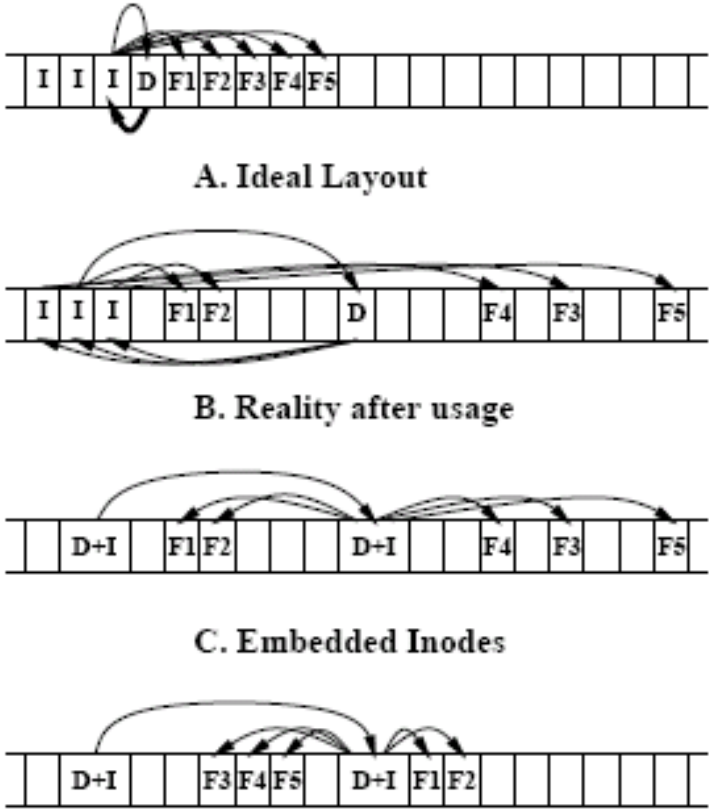


Figure 1: Organization and layout of file data on disk. This figure shows the on-disk locations of directory blocks (marked 'D'), inode blocks ('I') and the data for five single-block files ('F1' – 'F5') in four different scenarios: (A) the ideal conventional layout, (B) a more realistic conventional layout, (C) with the addition of embedded inodes, and (D) with both embedded inodes and explicit grouping (with a maximum group size of four blocks).

F1-F5 blocks from same file

em

Explicit grouping

- **Explicit grouping**
  - places the **data blocks** of **multiple files**
  - at **adjacent disk locations**
  - **accesses them as** a **single unit most of** the **time**

- **To decide** which **small files to co-locate**,
- **C-FFS exploits** the **inter-file relationships**
- indicated by the name space.

- Specifically, **C-FFS groups files**
  - **whose inodes are embedded**
  - **in** the **same directory**.

- As a result,
  - **explicit grouping has the potential**
  - **to improve small file performance**
  - **by an order of magnitude**
  - over **conventional file system implementations.**

Performances

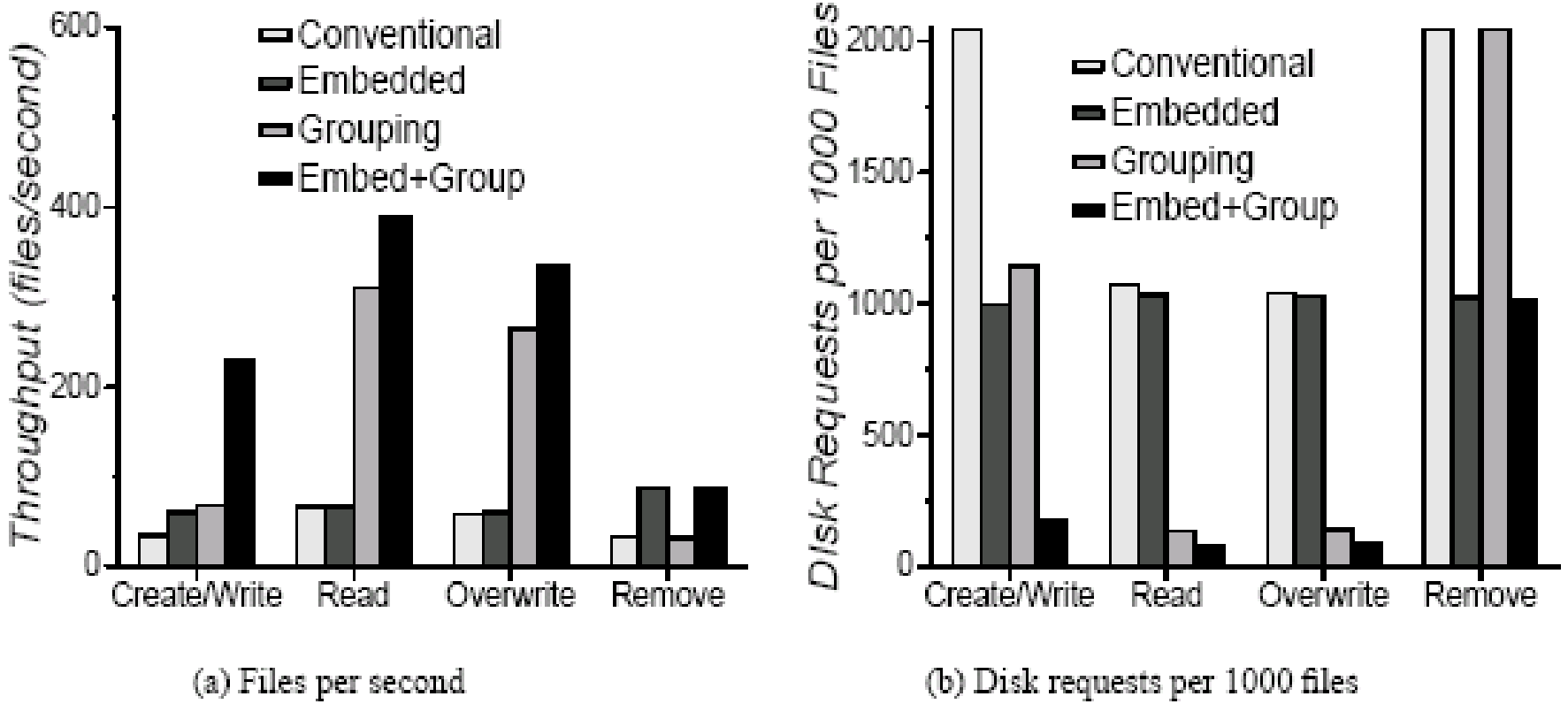


Figure 5: Small file throughput when using synchronous writes for metadata integrity.

embeded

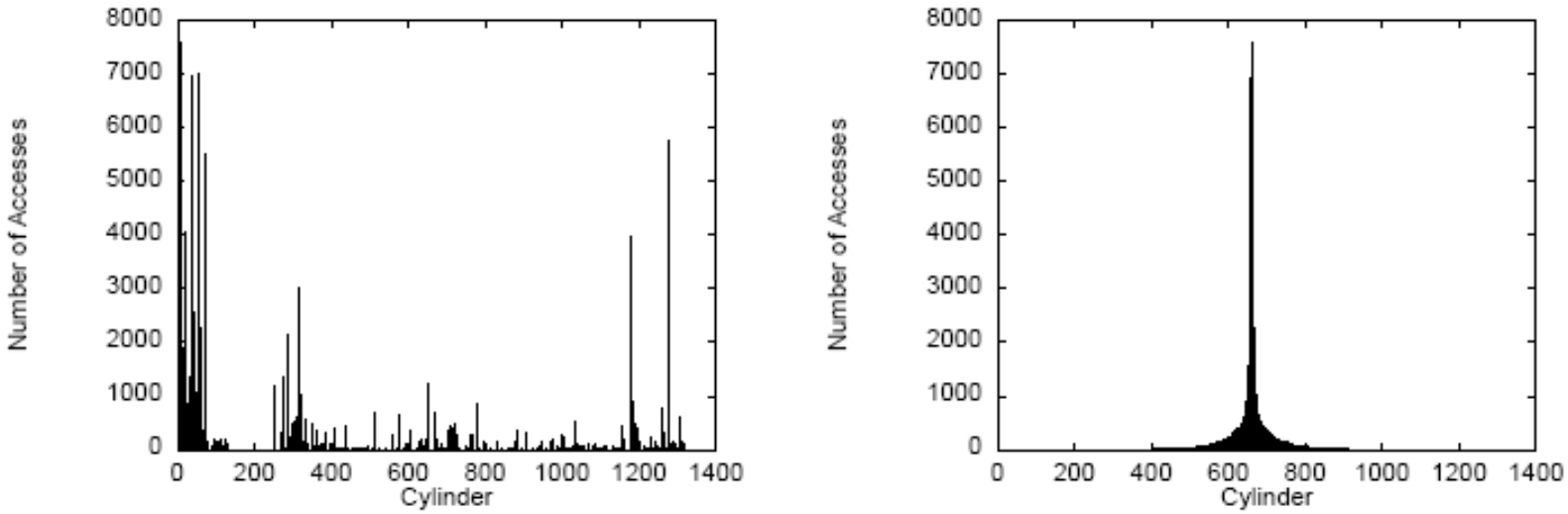
Other techniques

- **3. Disk Shuffling**

03. Disk Shuffling

- Adapting disk layouts to observed, rather than predicted, access patterns can result in faster I/O operations. In particular, a **technique called disk shuffling**, **which moves frequently-accessed data into the center of a disk, can substantially reduce mean seek distances**. We report here on extensions to and (partial) validation of earlier work on this approach at the University of Maryland.
- Starting from disk access traces obtained during normal system use of 4.2BSD-based file systems, we established a repeatable simulation environment across a range of workloads and disk types for comparing different shuffling algorithms. We explored several of these, including how often to make layout changes, how large a unit of data to shuffle, and some mechanisms to exploit sequentiality of reference.
- Our conclusions: some of the originally identified benefits are real, but sometimes performance is worse rather than better unless access interdependencies are considered.
- Most of the benefit can be obtained from infrequent (weekly) shuffling. Smaller quanta generally produce better results, at the expense of needing more working storage.
- Overall, the benefits are small to moderate, but are likely to be much larger with file systems that do not do such a good job of initial data placement.

Disk Shufling



a. As measured on the original layout.

b. After rearrangement into an organ pipe.

Figure 1. Cylinder access distribution over a twenty-four hour period. Measured on an HP7935 disk attached to *red*, a timesharing system running HP-UX.

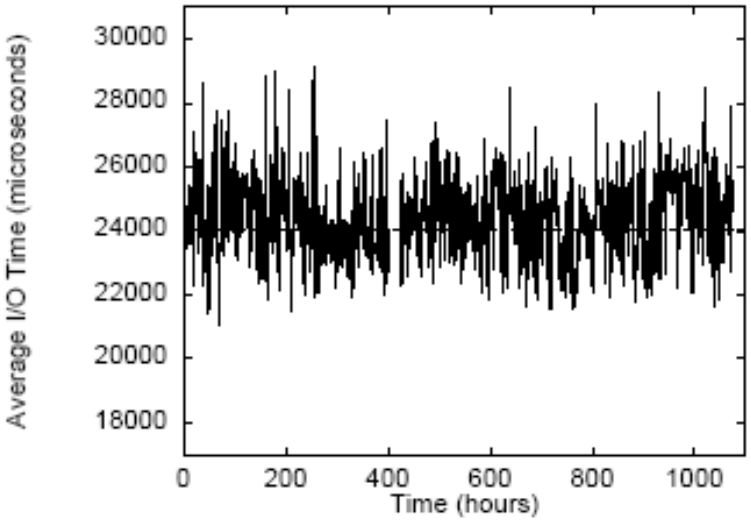
Principal

- Grouping the most frequently-accessed data blocks together at the center of the disk can **reduce** the average seek distance substantially. A good way to do this is the *organ pipe arrangement*, formed by placing the most frequently accessed cylinder in the middle of the disk, the next most frequently accessed cylinders on either side of the middle cylinder, and so on. This arrangement is provably optimal for independent disk accesses [Wong83]. The effect of applying it to the disk shown in Figure 1a is displayed in Figure 1b.

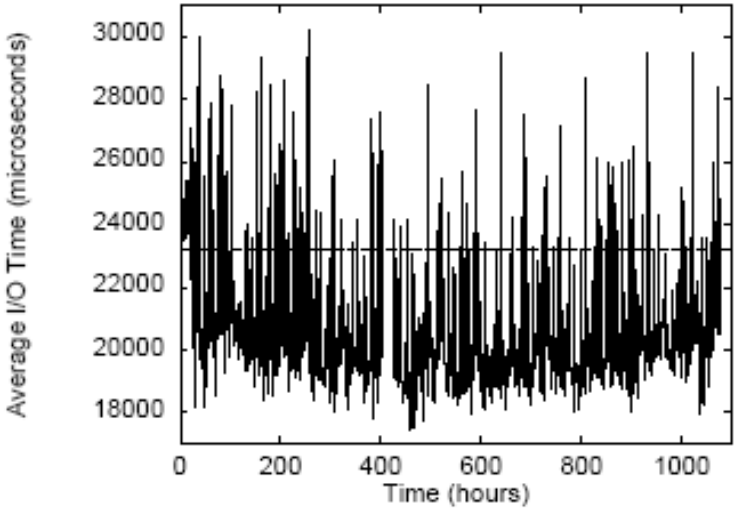
- Moving the data around to achieve such an arrangement is known as *disk shuffling*. The algorithm is sufficiently straightforward that it can be done **inside** the disk controller, or inside the device driver.

- In all cases, a count is kept of the number of requests directed to each *shuffling quantum* (e.g. cylinder) over a period of time, and these counts are used to drive the rearrangement.

Results



b. Average physical I/O time before shuffling (mean = 24.1 ms).



c. Average physical I/O time after shuffling (mean = 23.2 ms).

Other techniques

- **4. Log Approach**

04. LFS

- The **fundamental idea of** a **log-structured file system**
  - is to **improve write performance** by
  - **buffering a sequence of file system changes in** the **file cache** and
  - then **writing all** the **changes to disk sequentially**
  - **in** a **single disk write operation.**
- The **information written to disk in** the **write operation includes:**
  - **file data blocks**
  - **attributes**
  - **index blocks**
  - **directories**
  - **almost all the other information used to manage** the **file system**.
- **For workloads that contain many small files,**
  - a **log-structured file system converts**
  - **many small synchronous random writes of traditional file systems**
  - **into large asynchronous sequential transfers**
  - **that can utilize nearly 100% of** the **raw disk bandwidth.**

3. LFS

- whole disk = log (append only log)
- called log-structured file system

- LFS basic concept
  - Collect large number of written data in the cache
  - Put in the log in large sequential access

- As files are modified,
  - both file data and header information
  - are appended to the log
  - in a sequential stream
  - without seeks.

- if individual files are small
  - they can be collected into large blocks
  - before being written to the log.

3. LFS Benefits

- Fast recovery
- Temporal locality
- Versioning

3. LFS difficult issues

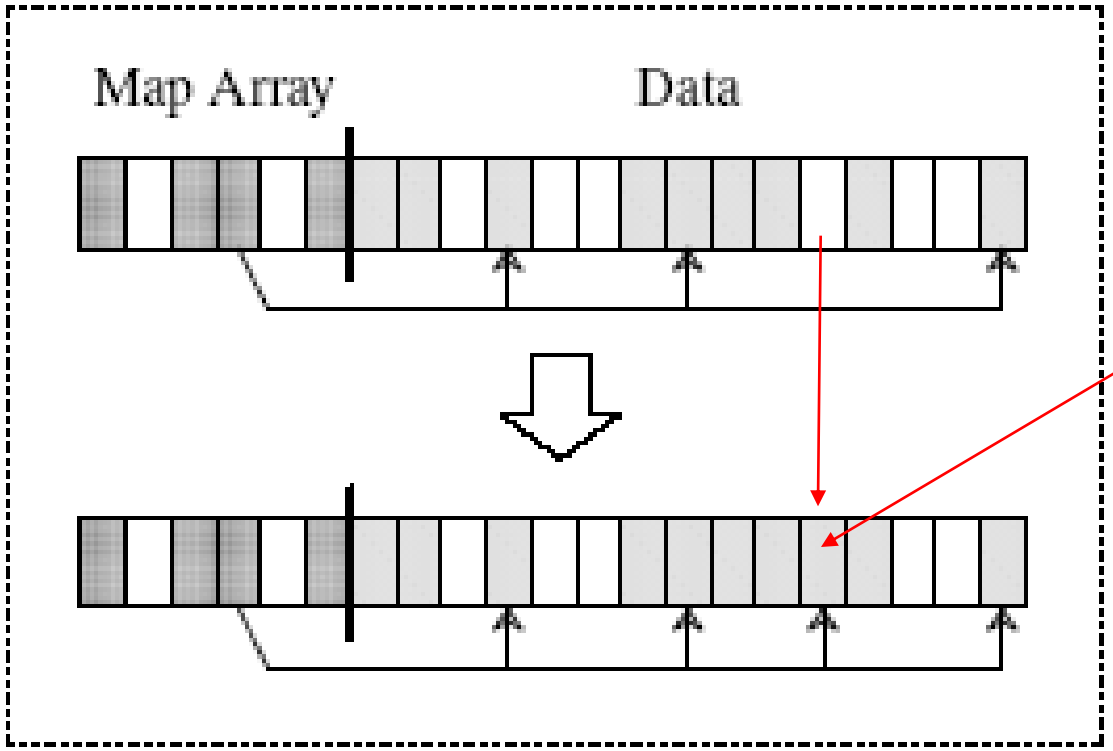
- **There are 3 difficult issues**
  - that
  - **must be resolved**
  - **to make log-structured file systems practical**.

- **I. how to handle the occasional retrievals (reading)**
  - **that will be required from** the **log**

- **II. how to manage log wrap-around**;

- **III. how to achieve efficient disk space utilization**

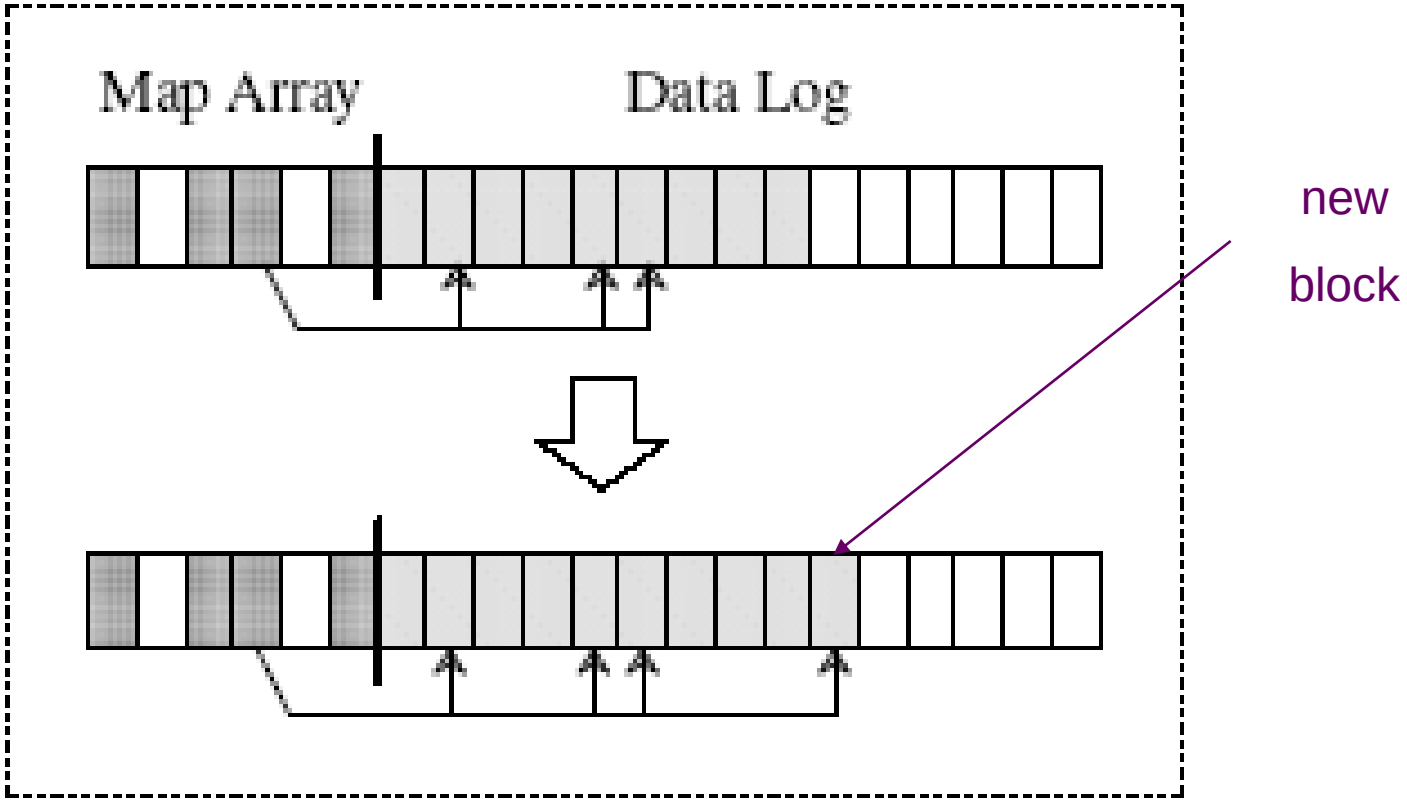
Classical FS (adding new block)



(a)

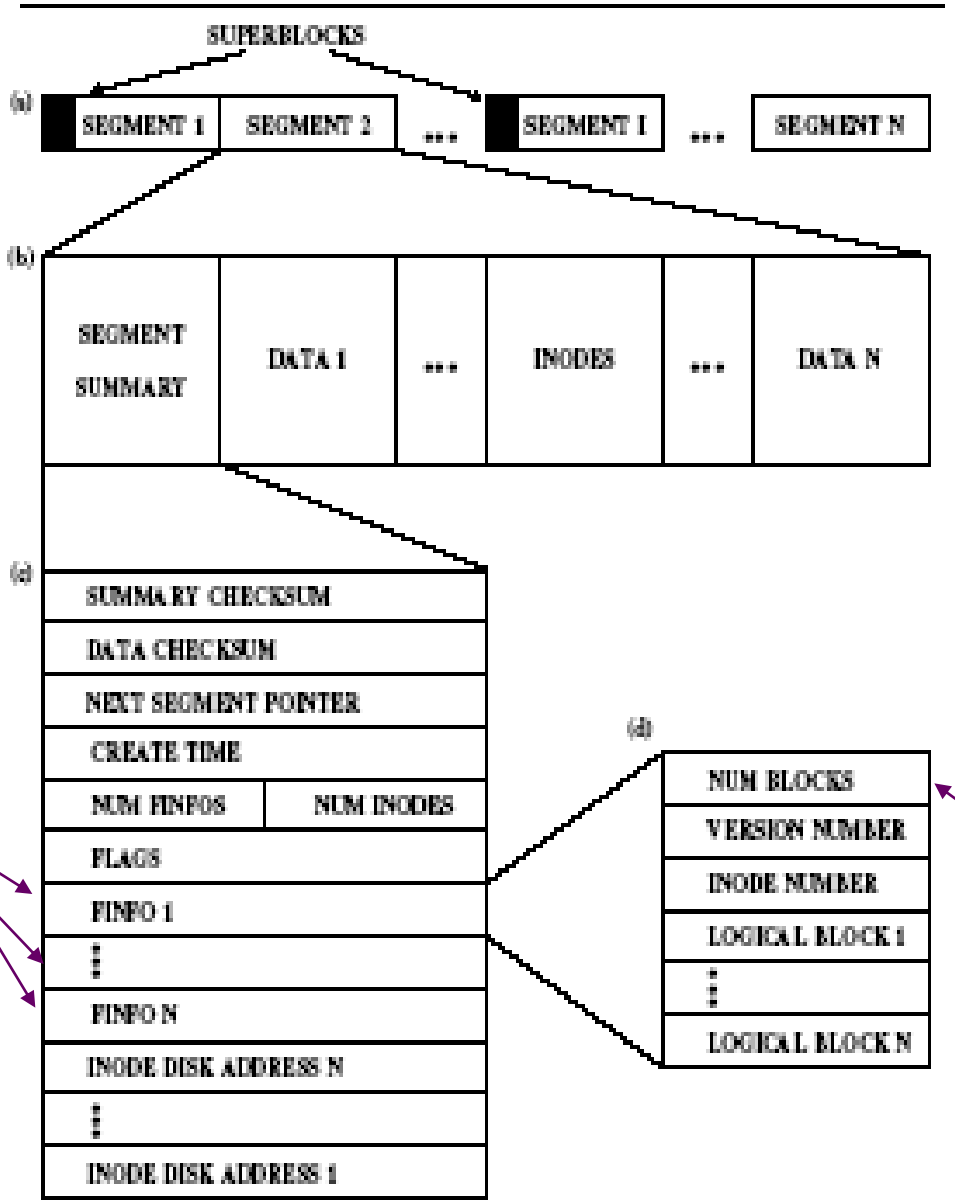
- **shows a traditional file system**
- **with separate map** and **data areas**;
- a **new data block is allocated** and
- the **map is updated in place**.

Log DATA (adding new block)



- the **data area has been made into a log**:
- **each new data block gets added at the end of** the **log,**
- **but map entries are still updated in place**.

UNIX LFS



files

File info

LFS Segments under UNIX

- A Log-Structured File System.
- A **file system is composed** of **segments** as shown in Figure (a).

- **Each segment consists of**
  - **a summary block followed** by
  - **data blocks** and **inode blocks** (b).

- The **segment summary** contains
  - c**hecksums** to validate both the segment summary and the data blocks,
  - a t**imestamp**,
  - a **pointer to the next segment**, and
  - **information that describes**
    - **each file and inode that appears in the segment** (c).

- **Files are described** by **FINFO structures**
  - **that identify the inode number** and
  - **version of the file** (as well as each block of that file)
  - **located in** the **segment** (d).

Wrap-around

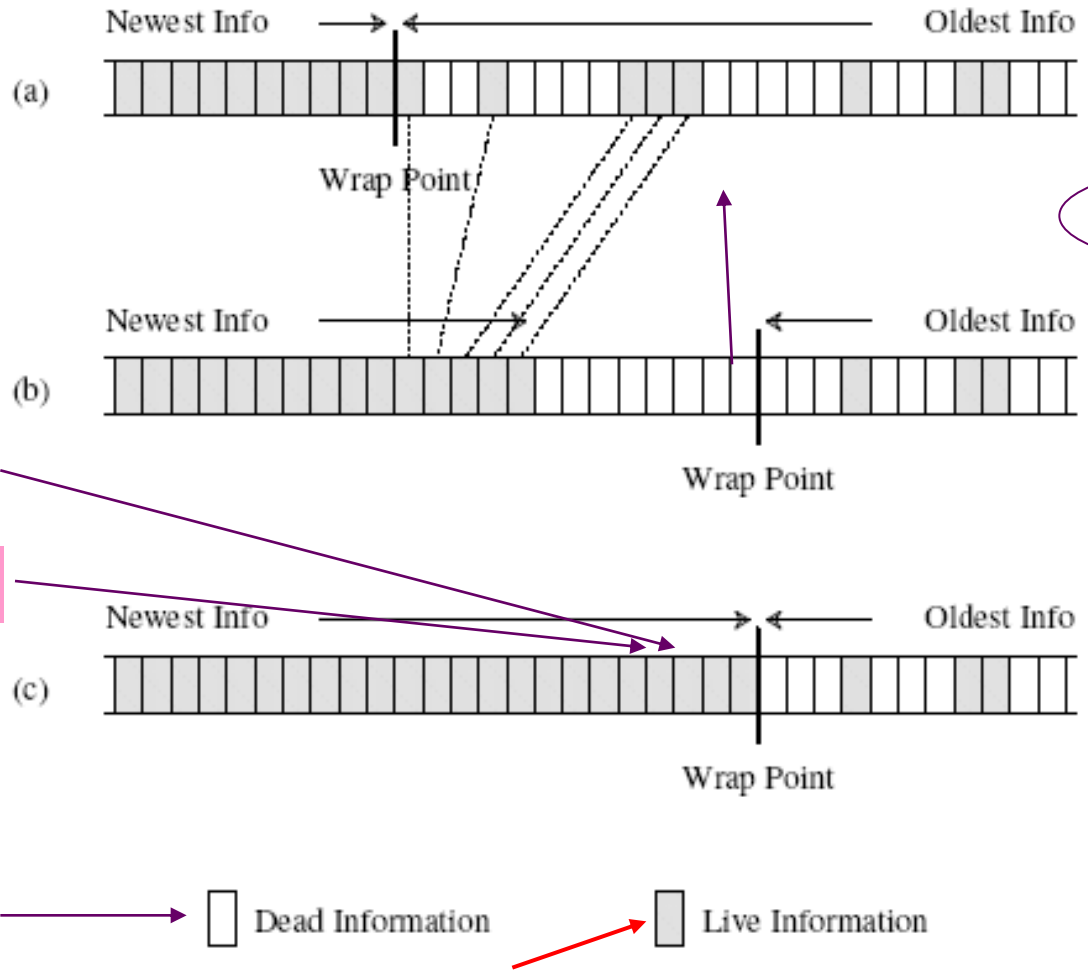


Figure 2. Incremental compaction in a log-structured file system. New data is appended to the log from left-to-right. In (a) the log has just filled up; among the oldest blocks in the log, only a few are still alive. In (b) the live information is compacted to the head of the log, leaving empty space for new log information. In (c) new information is appended to the log, regenerating the situation in (a).

Performance Comparisons

- LFS **will outperform**
- **other file systems**
- **for writes**.
- **In looking for weaknesses** of **LFS approach,**
  - it thus makes most sense
  - to look **at reads** that **miss in the file cache**.
- **For small files**,
  - a log-structured file system
  - **will have read performance**
  - **at least as good as today's file systems**
- **In the worst case**, one **seek will be required** for the **file map** and **one for the file data**.
  - **With a little cleverness in the log management**,
  - **it should be possible to write** the **file map**
  - **close to the file data**
  - so **they can both be retrieved with a single seek.**
  - This would result in **2x better** performance than **current file systems**.

Large file reading (file written at once)

- For **large-file reads**, there are **2 cases to consider**.
- 1. The **simplest case** is **files** that are **written all-at-once**.
- **These files will** be **contiguous in the log**,
  - **which allows them to be read**
  - **at least as efficiently as today's best file systems**
  - (particularly if the file map is written next to the data in the log).
- **Random-access reads** to such a file **will require seeks**,
  - but **no more** in a log-structured file system
  - **than in** a **traditional file system**.

Large access (file written piece-wise)

- 2. The **second case** for **large files consists** of **those that are written piece-wise**, either
  - **by gradually appending to the files or**
  - **by updating them** in **random-access mode.**
- The **logging approach permits such piece-wise writes** and
  - **does not require** the **whole file to be rewritten**,
  - but the **new data for the file will go** at the **end of the log**.
- **This will not be adjacent on disk**
  - **to other data written to the file previously**.
- **If the file is later read sequentially**
  - **from one end to the other,**
  - **many seeks will be required**.
- **In comparison,**
  - a more **traditional file system can keep**
  - the file's **data contiguous on disk**
  - **even under this sort of access pattern**.