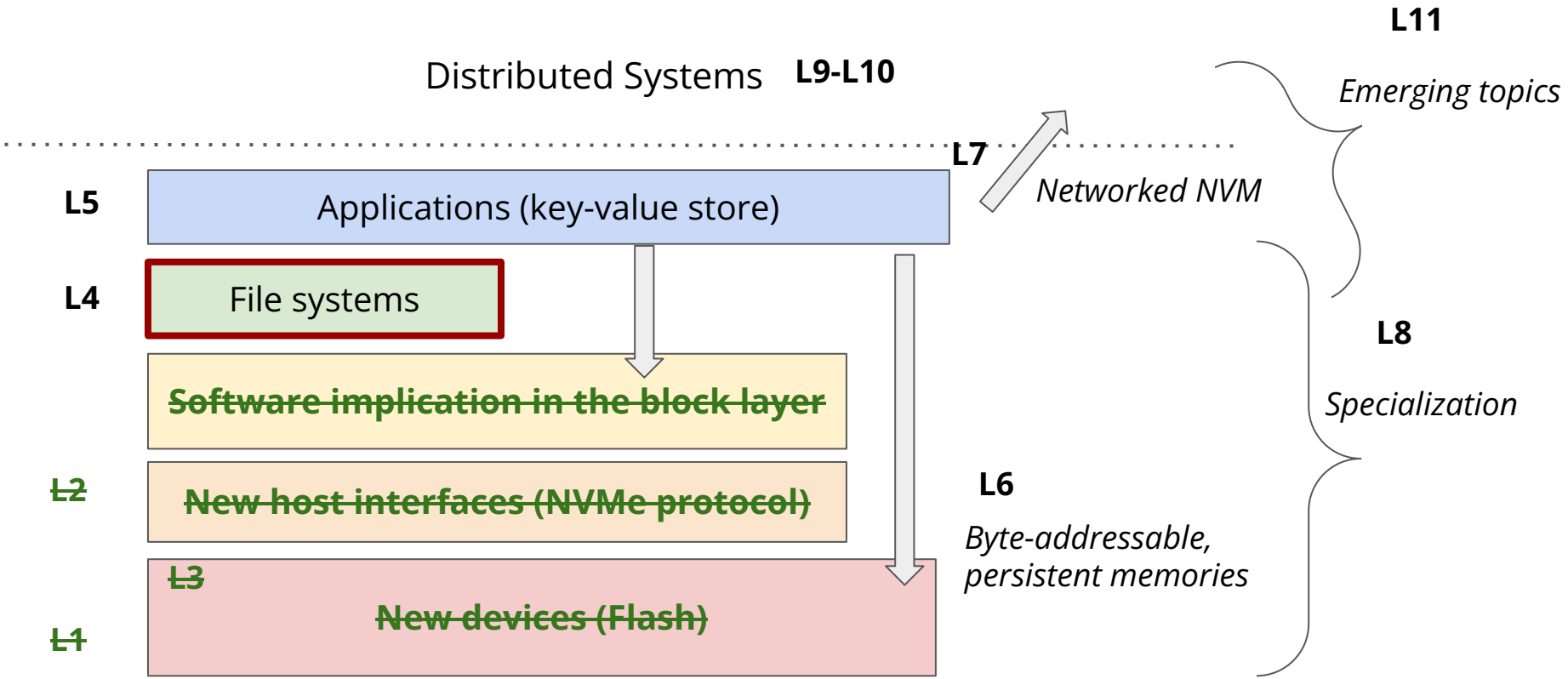
Storage Systems (StoSys)
XM_0092

Lecture 4: Flash-based File Systems

Animesh Trivedi
https://stonet-research.github.io/
Autumn 2023, Period 1

The layered approach in the lectures



Recap: File System (FS)

FS is responsible for

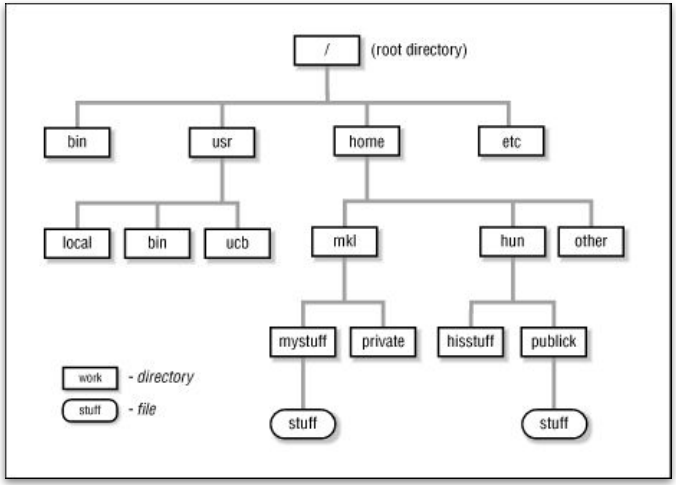
1. storing hierarchical directories and files on a flat disk;
2. translating user read/write to disk addresses

File systems have (1) data ; (2) metadata

1. **Data:** user data
2. **Metadata:** user and fs informations (name, creation time, storage location) etc.

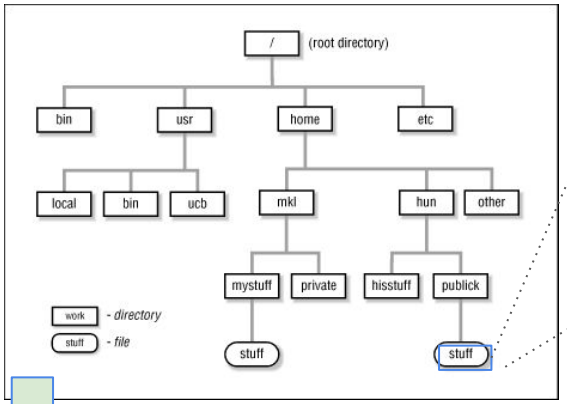
Important data structures: **inode** (stores file system metadata, and location of data)

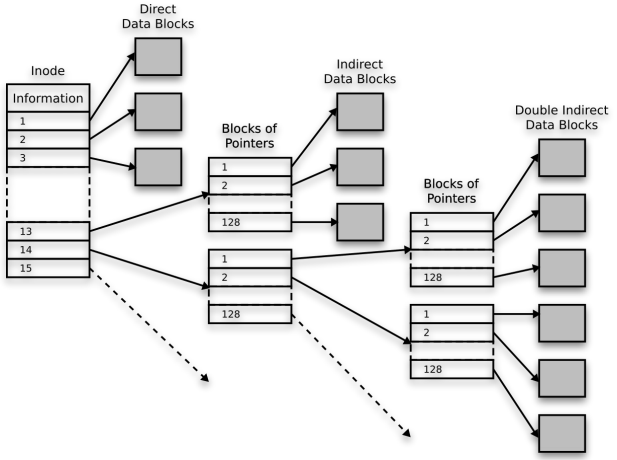
More: free bitmaps, extent maps, superblock, etc.

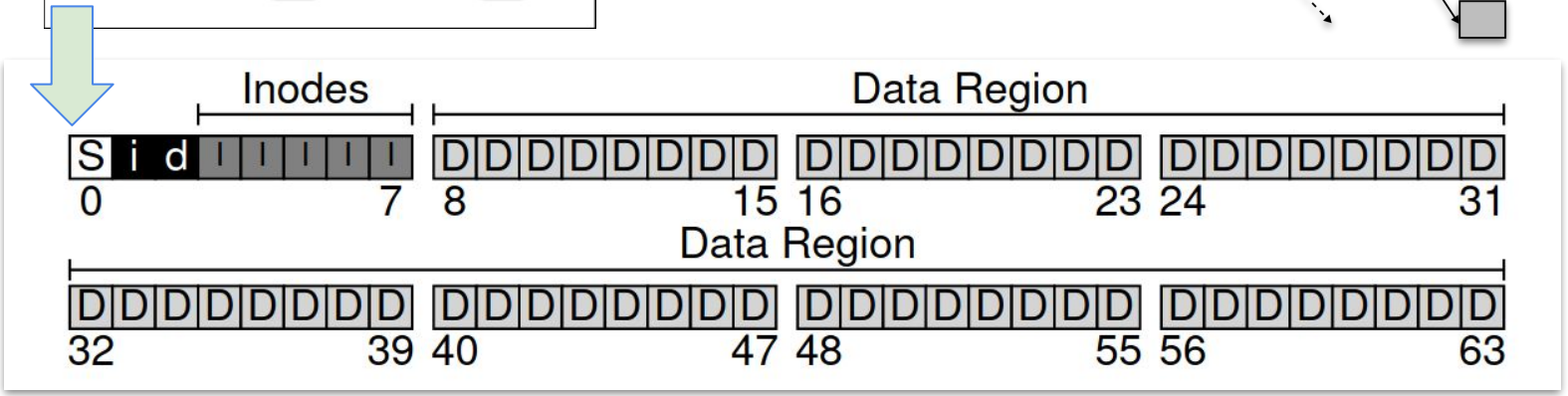


Background reading: chapter 40, File System Implementation, http://pages.cs.wisc.edu/~remzi/OSTEP/file-implementation.pdf

Recap: File System (FS)





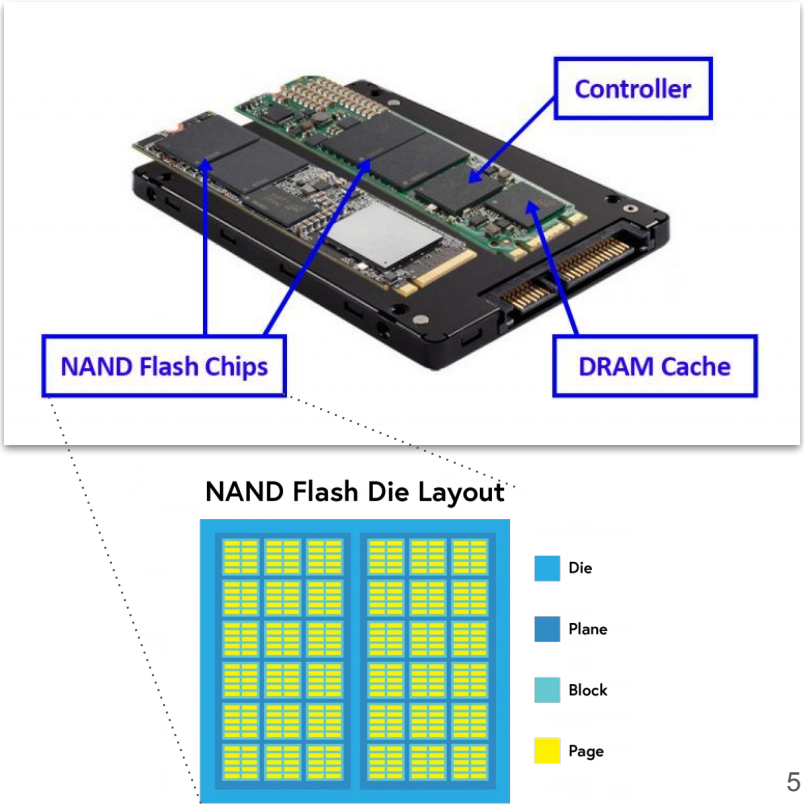


For example, ext2, https://wiki.osdev.org/Ext2 and https://piazza.com/class_profile/get_resource/il71xfllx3l16f/inz4wsb2m0w2oz

Why Do we really need a new file system?

NAND Flash SSD, even though “semantically” is like HDD (read/write sectors), internally it has:

- Mixed performance spectrum:
  - Very good sequential performance
  - Good random read performance
  - Poor random write performance
  - Very poor small random write performance
- FTL implementation
- GC interference
- Chip-die-plane parallelism
- Wear-leveling
- Error handling



What happens if we just ignore it

Technically we can just run any file system. Sure it will work, but

1. Poor degraded performance
2. Unpredictable performance (*cloud providers do not like this!*)
3. Poor reliability during related failures (e.g., wearing metadata areas)
4. Poor device lifetime

Bad things will happen :(Let's do our best try to avoid bad things.

Recall: we talked about how a "log" is a perfect match for flash-based I/O

- Immutable, sequential, transactional → perfect for flash !

Interestingly enough …

The Design and Implementation of a Log-Structured File System

MENDEL ROSENBLUM and JOHN K. OUSTERHOUT
University of California at Berkeley

This paper presents a new technique for disk storage management called a *log-structured file system*. A log-structured file system writes all modifications to disk sequentially in a log-like structure, thereby speeding up both file writing and crash recovery. The log is the only structure on disk; it contains indexing information so that files can be read back from the log efficiently. In order to maintain large free areas on disk for fast writing, we divide the log into *segments* and use a *segment cleaner* to compress the live information from heavily fragmented segments. We present a series of simulations that demonstrate the efficiency of a simple cleaning policy based on cost and benefit. We have implemented a prototype log-structured file system called Sprite LFS; it outperforms current Unix file systems by an order of magnitude for small-file writes while matching or exceeding Unix performance for reads and large writes. Even when the overhead for cleaning is included, Sprite LFS can use 70% of the disk bandwidth for writing, whereas Unix file systems typically can use only 5–10%.

Categories and Subject Descriptors: D 4.2 [**Operating Systems**]: Storage Management—*allocation/deallocation strategies; secondary storage*; D 4.3 [**Operating Systems**]: File Systems Management—*file organization, directory structures, access methods*; D.4.5 [**Operating Systems**]: Reliability—*checkpoint/restart*; D.4.8 [**Operating Systems**]: Performance—*measurements, simulation, operation analysis*; H.2.2 [**Database Management**]: Physical Design—*recovery and restart*; H.3 2 [**Information Systems**]: Information Storage—*file organization*

General Terms: Algorithms, Design, Measurement, Performance

Additional Key Words and Phrases: Disk storage management, fast crash recovery, file system organization, file system performance, high write performance, logging, log-structured, Unix

A Log-structured file system (**SpriteFS**) was investigated back in the early 1990s

Highly influential work

Can you guess why such a design would make sense back in 1992 for a HDD based fs?

Mendel Rosenblum and John K. Ousterhout. **1992**. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.* 10, 1 (Feb. 1992), 26–52. DOI:https://doi.org/10.1145/146941.146943

Why Log-Structured file system (LFS) in 1992?

1. The amount of system DRAM was increasing
   a. More opportunity to cache data and serve “read” requests from DRAM
   b. DRAM is random access, hence, good “read” performance

2. Access to disk will dominated by “writes”
   a. Writes can be sequential and random
   b. Writes can be **small (metadata)** and **large (data)** - large writes are OK, *but small writes are really bad. Plus random writes for metadata updates --- really really bad*

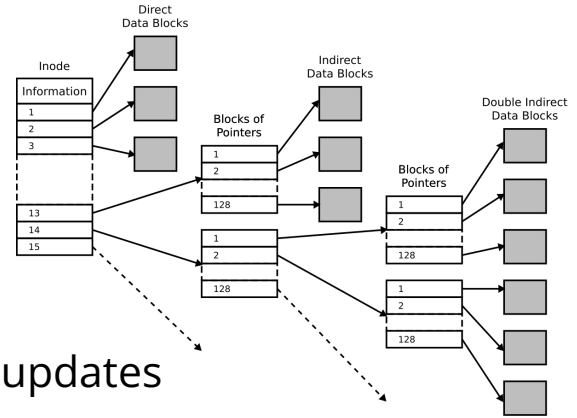
3. Hence, use a log-structured file system optimized for servicing fast writes
   a. Random “read” (metadata) not so much → must be served from the buffer cache

It turned out that “log” is a very useful data structure for write-once media as well (like NAND flash). *But how do you make a working file system on a log? How to do you find inodes? and what happens when a log is full?*

The basic idea of an LFS

With an LFS, there cannot be a single known location where inodes are stored, the location changes every time an inode is updated

LFS's goal is to optimize inode metadata lookups -- *why*?



All new writes are written to the log in a sequential manner, and then a "`inode map`" structure is written to identify their locations

inode maps are written to the log after each (or batch of) updates

LFS's checkpoint region contains all inode maps information

Inode maps are typically cached in the buffer cache for fast lookups

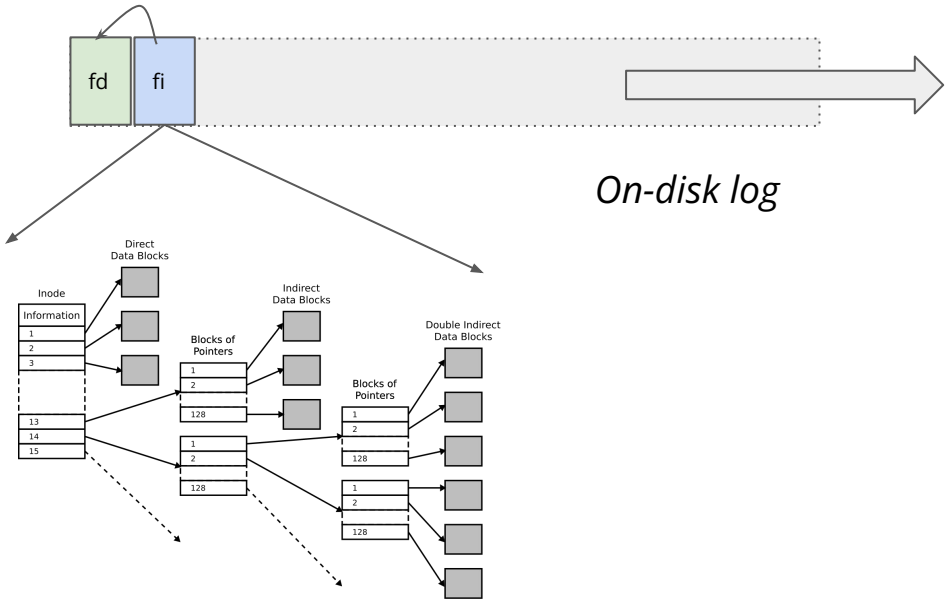
http://www.cs.cornell.edu/courses/cs4410/2014fa/slides/13-lfs.pdf

Simple example

Let’s say we want to create `/dir1/file1`

Data block

File inode block



File inode contains the disk offsets for these pointers

Simple example

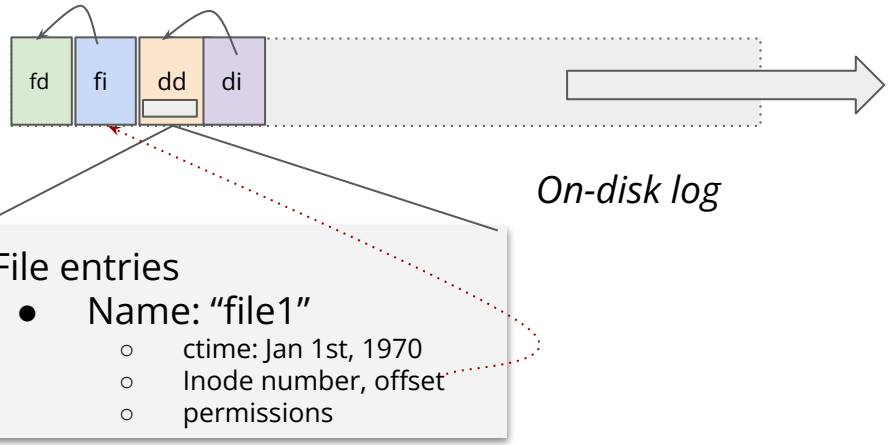
Let's say we want to create `/dir1/file1`

Data block

File inode block

Directory data block

Directory inode block

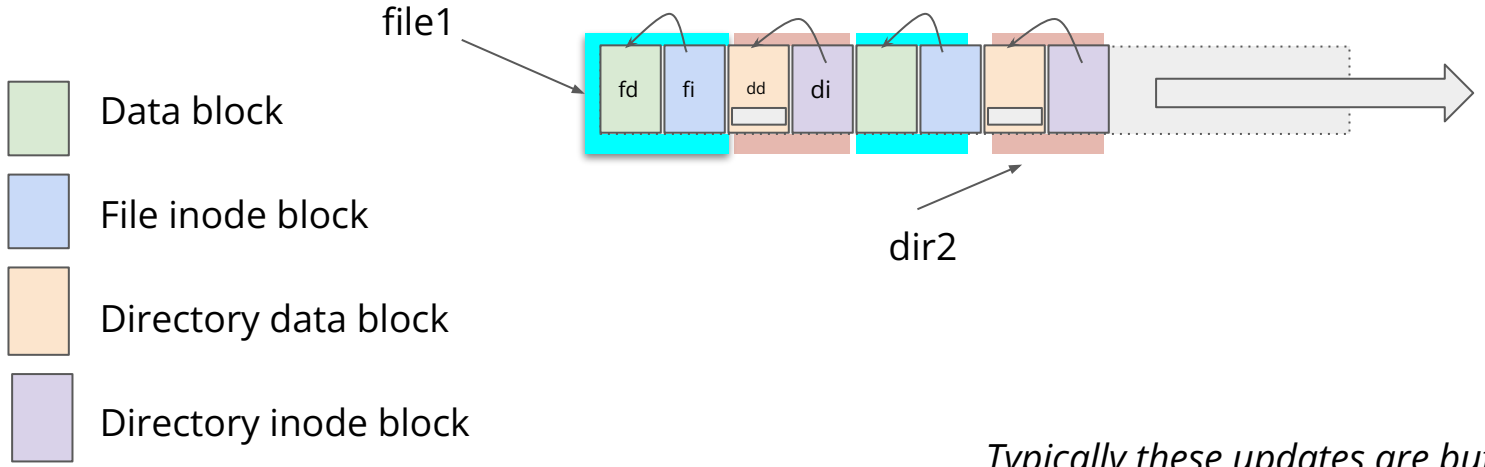


fd = file data
fi = file inode
dd = directory data
di = directory inode

Remember directories are just special files with a special format to keep track of all other files and directories inside it

Simple example

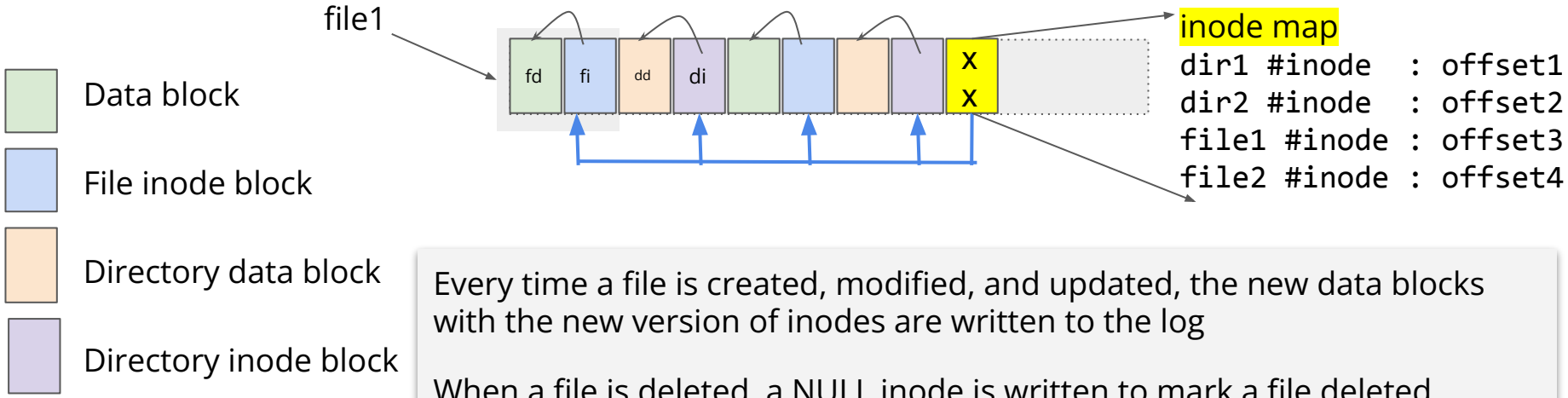
Let's say we want to create another `/dir2/file2`



*Typically these updates are buffered in the DRAM cache and then written out in **a single large sequential segments** to amortize the disk seek cost*

Simple example

Two locations: `/dir1/file1` and `/dir2/file2`



Every time a file is created, modified, and updated, the new data blocks with the new version of inodes are written to the log

When a file is deleted, a NULL inode is written to mark a file deleted

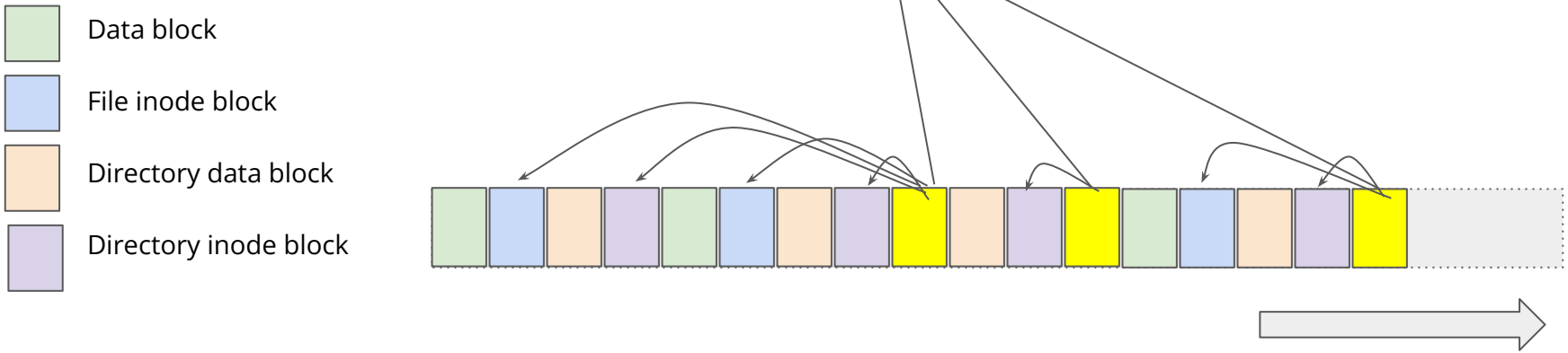
There is **an in-memory big inode map table** that has the latest offset entries for all inode locations written to the log (*optimization, not necessary for the correctness*)

Simple example

In-memory Inode map Table

```
Inode number 100 : offset1
Inode number 200 : offset2
...
```

In case there are concurrent updates
→ The last one wins
→ in case there is a crash, the in-memory table can be build again by scanning the log at the mount time

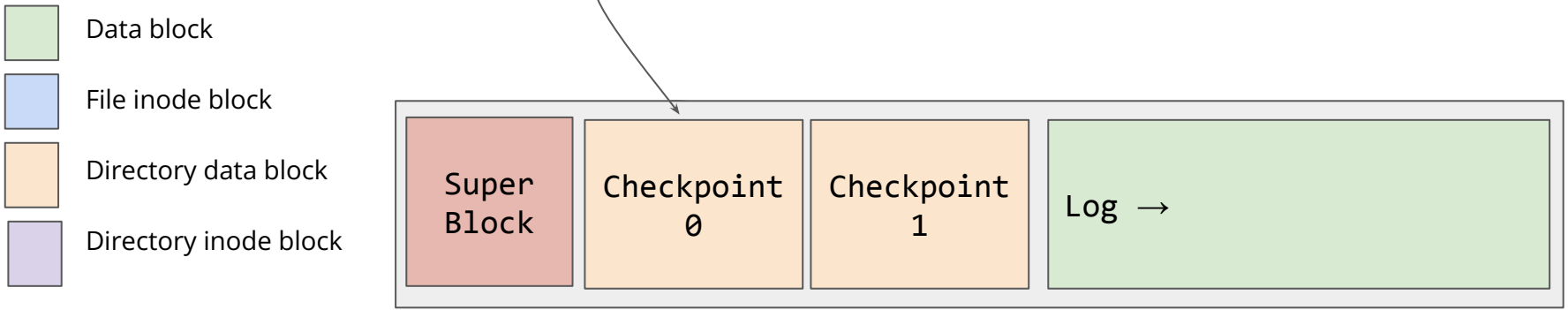


Simple example

In-memory Inode map Table

```
Inode number 100 : offset1
Inode number 200 : offset2
...
```

In case there are concurrent updates
→ The last one wins
→ in case there is a crash, the in-memory table can be build again by scanning the log at the mount time

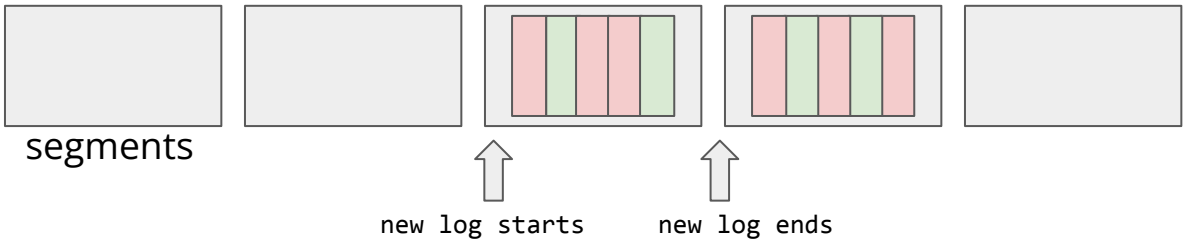


The one thing it has to remember is where is **the root inode location** - that can be stored when the LogFS does checkpointing (like any other file system). The initial SuperBlock location and 2x **checkpoint regions are fixed** (stores inode map table, root inode location etc.)

What happens when a log become full?

The log is divided into large “segments” which are the unit of cleaning (typically 10-100MBs, anyone Zones?)

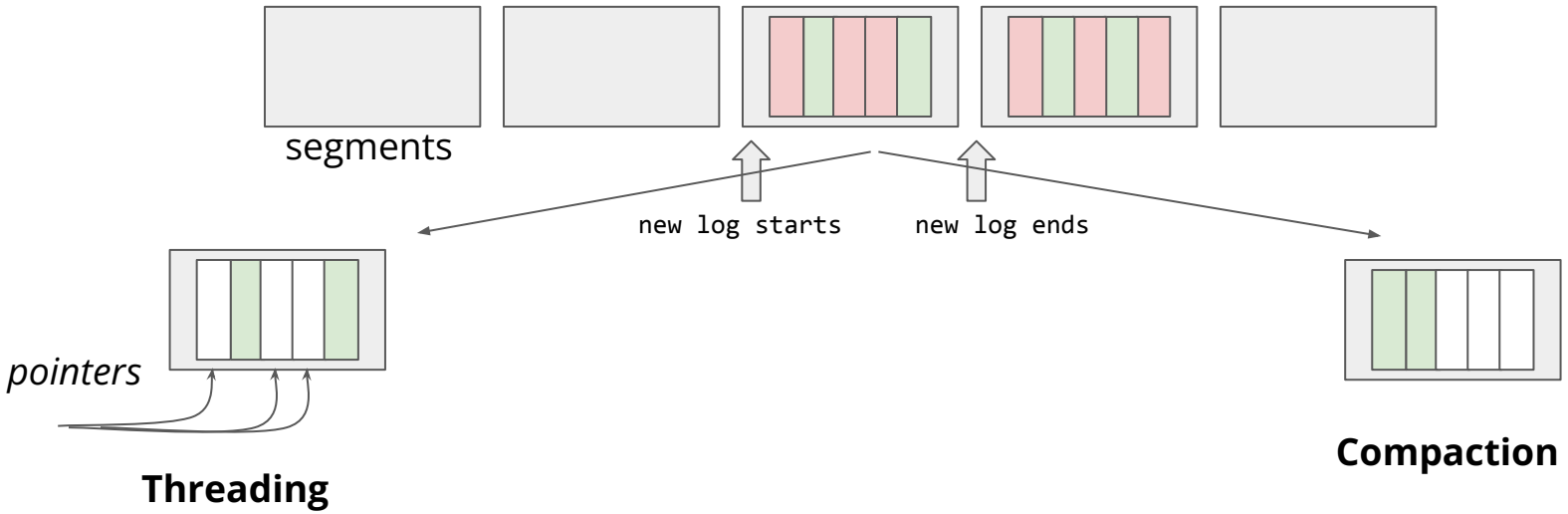
So what can be done with this design? **Two options:**



What happens when a log become full?

The log is divided into large “segments” which are the unit of cleaning (typically 10-100MBs, anyone Zones?)

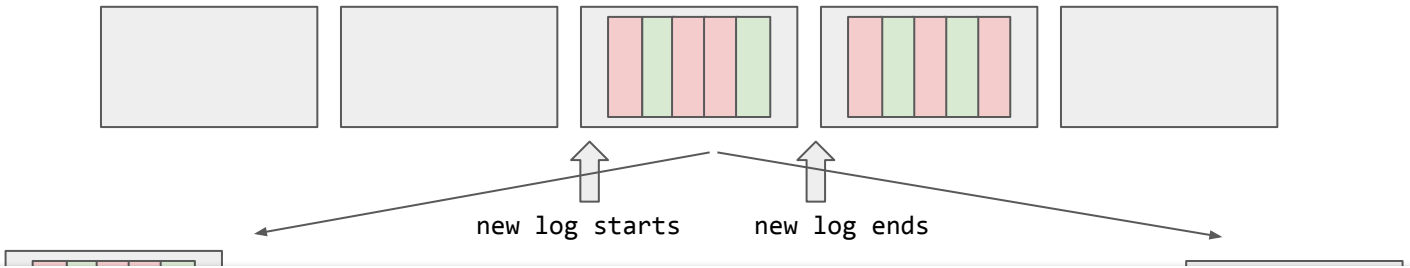
So what can be done with this design? **Two options:**



What happens when a log become full?

The log is divided into large “segments” which are the unit of cleaning (typically 10-100MBs, anyone Zones?)

So what can be done with this design? **Two options:**



Threading : no explicit garbage collection, fast, but metadata to keep track of holes, random accesses

Compaction : explicit garbage collection phase, copy cost, but gives nice clean blocks with less overheads

Sprite LFS used a hybrid: segments is always written sequentially and then copy and compacted
However, the log is threaded segment-by-segment basis

The log cleaning

Segments are the unit of GC cleaning

After each segment there is a segment summary block to keep track of “live” and “dead” blocks

- *How does FTL keep track of this?*

Everytime GC is invoked - it need to select a target/victim segment for cleaning

At the time of cleaning, when data is being re-arranged, the GC has an opportunity to re-arrange blocks in a segment to pack “hot and cold” data separately (**lazy classification**)

Segment cleaning logic: Picking up a victim

Greedy

$$\text{write cost} = \frac{\text{total bytes read and written}}{\text{new data written}} = \frac{\text{read segs} + \text{write live} + \text{write new}}{\text{new data written}} = \frac{N + N*u + N*(1-u)}{N*(1-u)} = \frac{2}{1-u}$$

Cost-Benefit Analysis

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

Goal: the goal is is to create one clean segment for every new segment data written
Greedy picks up the most utilized segment ("u" is utilization between 0 and 1), "N" is the number of pages in a segment

Cost-benefit analysis does include the "hotness" or "age" of data (the last time data was updated) and how much space we will free (1 - u), with total work (1 (read) + u (write))

For FTL: These mechanisms are exactly the same (*now with formulas*) what we discussed in the context of GC, and even actually inspired many "victim" selection policies in FTL/GC implementations

Segment cleaning logic: Picking up a victim

Greedy

Cost-Benefit Analysis

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

What is missing here?

Should I apply these formulas as it is in my FTL?

For FTL: These mechanisms are exactly the same (*now with formulas*) what we discussed in the context of GC, and even actually inspired many “victim” selection policies in FTL/GC implementations

Why Log-Structured FSes were not good enough

1. **Segment cleaning overheads in Log-Structured file systems**
   a. The Achilles heel for LogFS : ***a long and interesting debate***
      i. *"the impact of the cleaner is so severe that BSD-LFS cannot compete with either FFS or EFS. For the tests presented here, the disk was running at 85% utilization, and the cleaner was continually running. Note that FFS and EFS allocate up to 90% of the disk capacity without exhibiting any performance degradation" (https://www.hhhh.org/perseant/lfs/lfs_for_unix.pdf, page 16)*
   b. Not expressive enough for modern file system workloads
      i. Dominated by random, small I/O on files (stresses the primitive segment cleaning)
   c. How would you identify hot/cold data? Is there a FS API?

2. **Ignoring the device characteristics**
   a. Different sector, page, block sizes and layouts
   b. Multiple read/write possible at the same time
   c. Not all random writes are the same
   d. Different read, write, and GC cost and granularities
   e. Performance vs. utilization

The Semantic Gap: File Systems and FTL

Both, FTL designs and Log-Structured file systems advocate to separate cold from hot data

In-place update file systems like FAT32 or ext4

- The FTL can identify hot and cold data by keeping track of #invalidation

But in a log-structured file system, the same page is not written twice. How does the FTL knows now? **Open challenge**

Generally this problem is known as "**Semantic Gap**" between layers, exists in multiple systems fields like virtualization, networking, storage, etc.

⇒ risen from Modularity and Layering principles

SFS: Random Write Considered Harmful in Solid State Drives (2012)

Improving the semantic gap by letting the FS maintain hotness statistics actively

- Hotness is maintained on the **F**ile **B**lock (or offset), not its address (that changes in an LFS)
- Classify Files, FB, and segments into different hotness **groups**
- Use this hotness statistics to do better victim segment selection for GC
- Do an “eager” classification than a “lazy” one as proposed in the original LFS paper

Goal: This makes the life of the device-side FTL easy(-ier)

SFS: Random Write Considered Harmful in Solid State Drives

Changwoo Min[a], Kangnyeon Kim[b], Hyunjin Cho[c], Sang-Won Lee[d], Young Ik Eom[e]
[abde]Sungkyunkwan University, Korea
[ac]Samsung Electronics, Korea
{multics69[a], kangnuni[b],wonlee[d],yieom[e]}@ece.skku.ac.kr, hj1120.cho[c]@samsung.com

Abstract

Over the last decade we have witnessed the relentless technological improvement in flash-based solid-state drives (SSDs) and they have many advantages over hard disk drives (HDDs) as a secondary storage such as performance and power consumption. However, the random write performance in SSDs still remains as a concern. Even in modern SSDs, the disparity between random and sequential write bandwidth is more than ten-fold. Moreover, random writes can shorten the limited lifespan of SSDs because they incur more NAND block erases per write.

In order to overcome these problems due to random writes, in this paper, we propose a new file system for SSDs, SFS. First, SFS exploits the maximum write bandwidth of SSD by taking a log-structured approach. SFS transforms all random writes at file system level to sequential ones at SSD level. Second, SFS takes a new data grouping strategy *on writing*, instead of the existing data separation strategy *on segment cleaning*. It puts the data blocks with similar update likelihood into the same segment. This minimizes the inevitable segment cleaning overhead in any log-structured file system by allowing the segments to form a sharp bimodal distribution of segment utilization.

The limited lifespan of SSDs remains a critical concern in reliability-sensitive environments, such as data centers [5]. Even worse, the ever-increased bit density for higher capacity in NAND flash memory chips has resulted in a sharp drop in the number of program/erase cycles from 10K to 5K for the last two years [4]. Meanwhile, previous work [12, 9] shows that random writes can cause internal fragmentation of SSDs and thus lead to performance degradation by an order of magnitude. In contrast to HDDs, the performance degradation in SSDs caused by the fragmentation lasts for a while after random writes are stopped. The reason for this is that random writes cause the data pages in NAND flash blocks to be copied elsewhere and erased. Therefore, the lifespan of an SSD can be drastically reduced by random writes.

Not surprisingly, researchers have devoted much effort to resolving these problems. Most of work has been focused on a *flash translation layer* (FTL) – an SSD firmware emulating an HDD by hiding the complexity of NAND flash memory. Some studies [24, 14] improved random write performance by providing more efficient logical to physical address mapping. Meanwhile, other studies [22, 14] propose a separation of hot/cold data to improve random write performance. However, such under-the-hood optimizations are purely based on

SFS: Basic workings

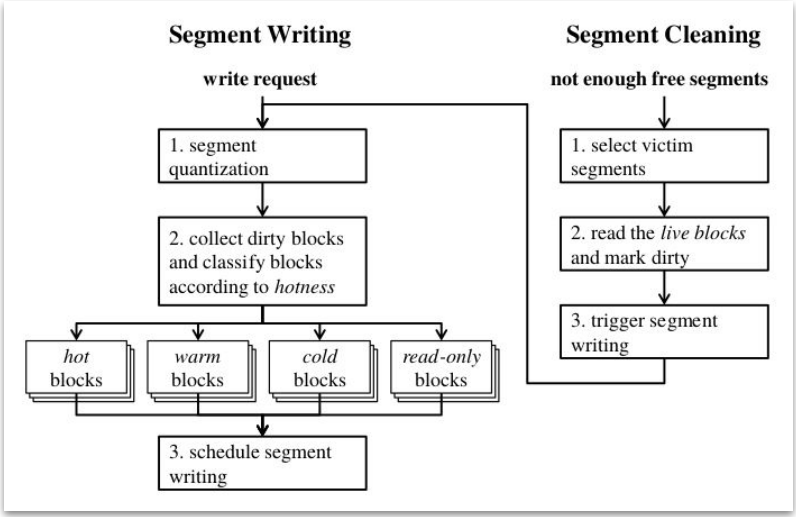
Hotness is maintained on three levels:
(i) File; (ii) File-Block; and (iii) Segment

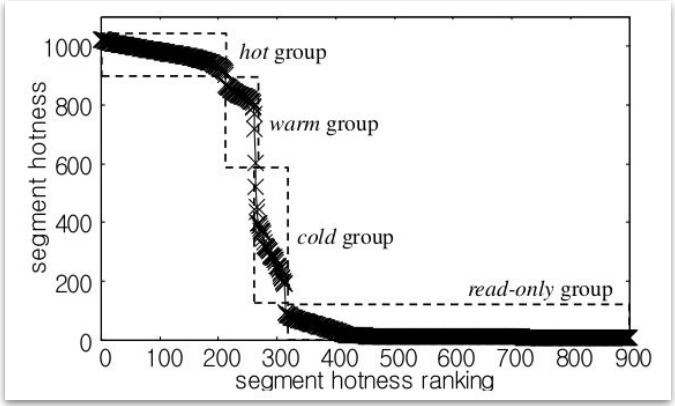
- Segment - the same concept as before (GC unit)
- File-Block - file offset (or its logical address)
- File - can contain multiple file-blocks

Classify writes to [hot, warm, cold, ro]

Segment cleaning is similar to LogFS,
but it also goes through the same path
of writing a segment

Key Difference: Every write is classified (**eagerly**),
unlike LFS which classifies data when doing GC
(this design helps with managing traffic skewness)





Cost-Hotness victim selection policy

Recall: we looked at **Greedy** and **Cost-Benefit** policies before

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

In Sprite FS, they just use the
last modified time as an estimation of hotness

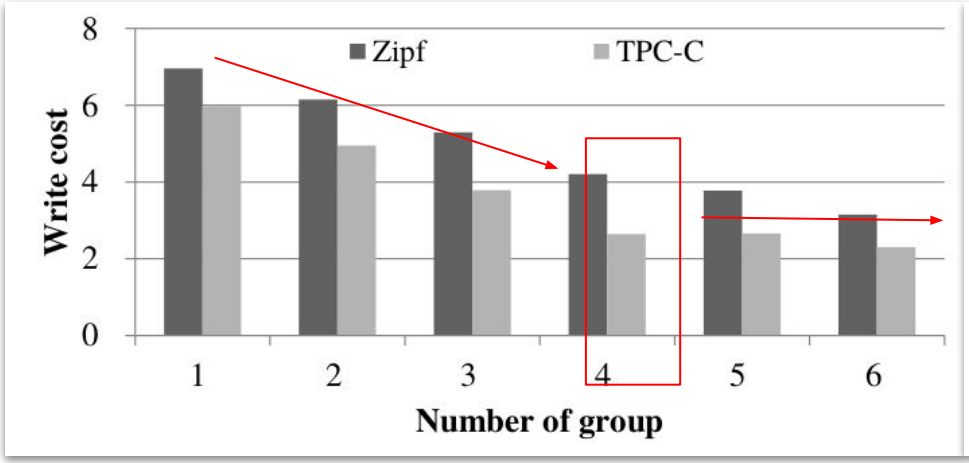
SFS uses proposes **Cost-Hotness policy**

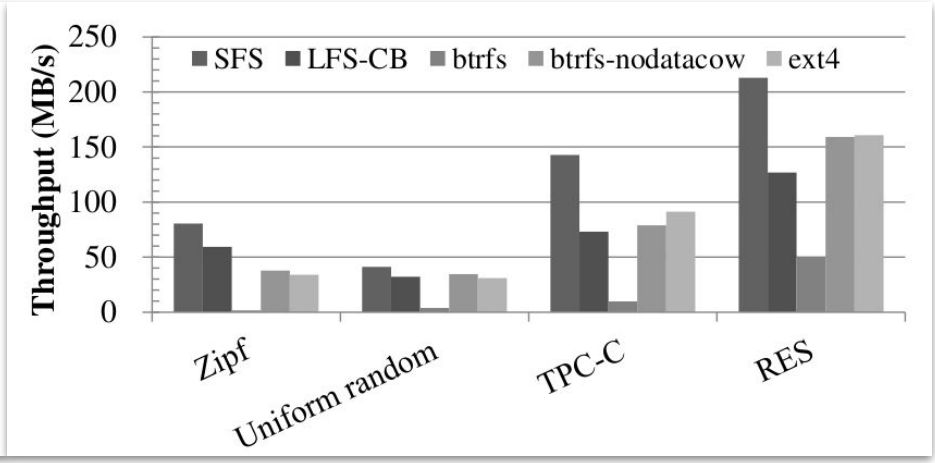
$$\text{cost-hotness} = \frac{\text{free space generated}}{\text{cost} * \text{segment hotness}} = \frac{(1-U_s)}{2U_sH_s}$$

- U_s is segment utilization
- H_s is segment hotness

Similar logic, but now (more) accurately picks up victim segment for cleaning

So, is this effective?





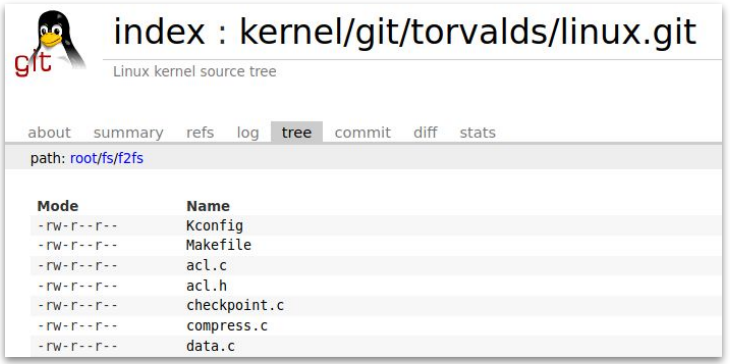
Multiple number of hotness groups help to decrease the **write cost** (WC)

- (New W data + Old R data + Old W data) / New W data
- Same as the write amplification but includes read cost too

SFS beats

(i) LFS-CostBenefit (CB);
(ii) BrtFS (COW-mode);
(iii) BrtFS (no-COW);
(iv) ext4 (which uses logging for data journaling)

Flash-Friendly File System (F2FS) (2015)

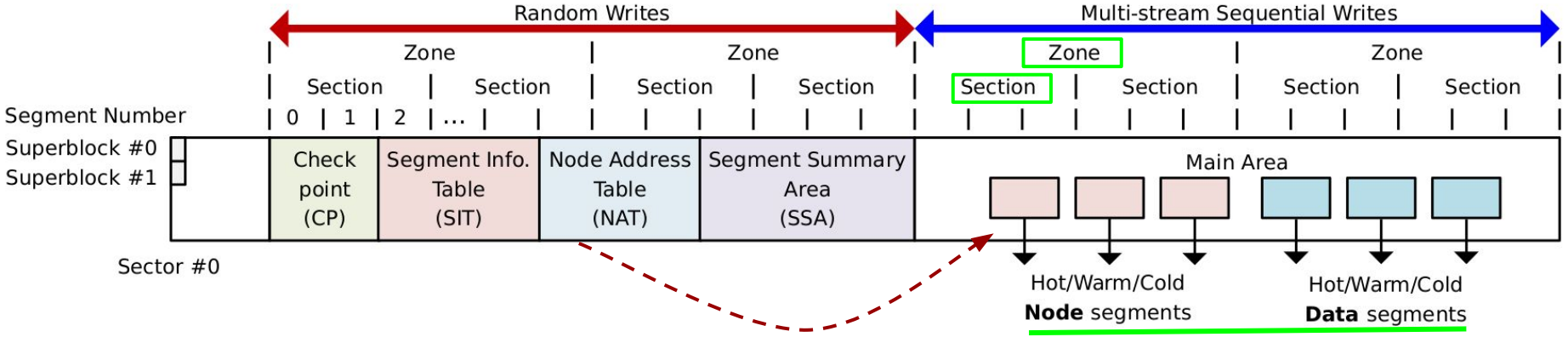


Highly influential work

One of the first file systems designed from scratch for NAND flash and is part of the mainline kernel (**production quality**):
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/fs/f2fs?h=master
https://www.kernel.org/doc/html/latest/filesystems/f2fs.html

Primary concerns: *layout and parallelism inside flash devices* (+previous best ideas)

F2FS: Disk Layout



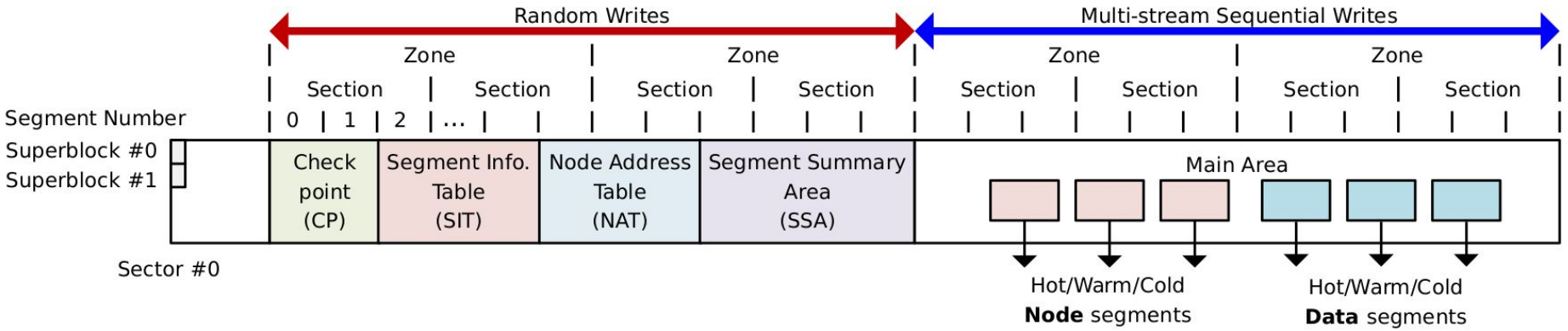
Device is split into:

- Zones : unit of parallelism, can open multiple parallel zones for I/O
  - Sections : unit of cleaning, some multiple of the flash GC units (if known, or large enough)
    - Segments : unit of space allocation (can contain multiple flash pages)

A section stores either (1) **Node** contains inode (with single, double, triple pointer pages) and indices of data pages; or (2) **Data** segments (user data)

Two areas: random writes (F2FS's own metadata) and sequential writes

F2FS: Disk Layout

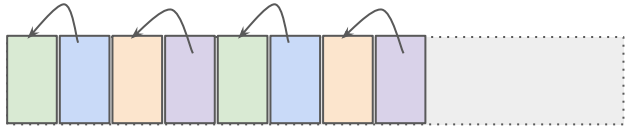


All the file metadata is written in the start Zones (classified as **Random Write Zones**)

- Superblock : read-only information about the file system
- Check point area (CP) : 2x to switch between stable and active
- Segment Information Table (SIT) : per-segment information, live blocks, used in GC
- **Node Address Table (NAT) : address of "nodes" blocks in the Main Area**
- Segment Summary Area (SSA) : to identify parent blocks and fs tree
- Main Area : data (metadata and data) segments are written

F2FS : File Structure

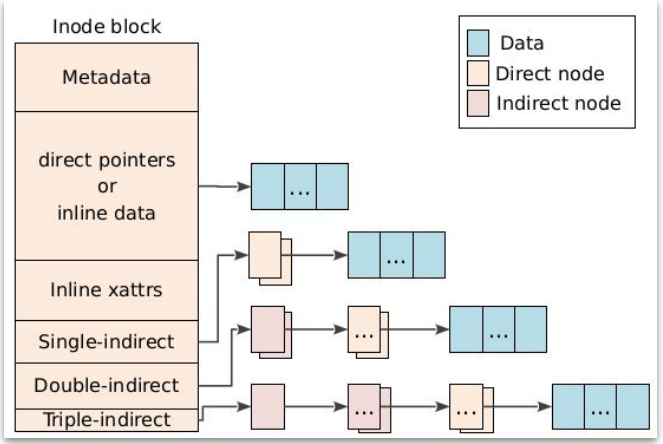
The file structure is not surprising, follows a typical “inode” based tree model



There are direct, single, double, and triple indirect pointers

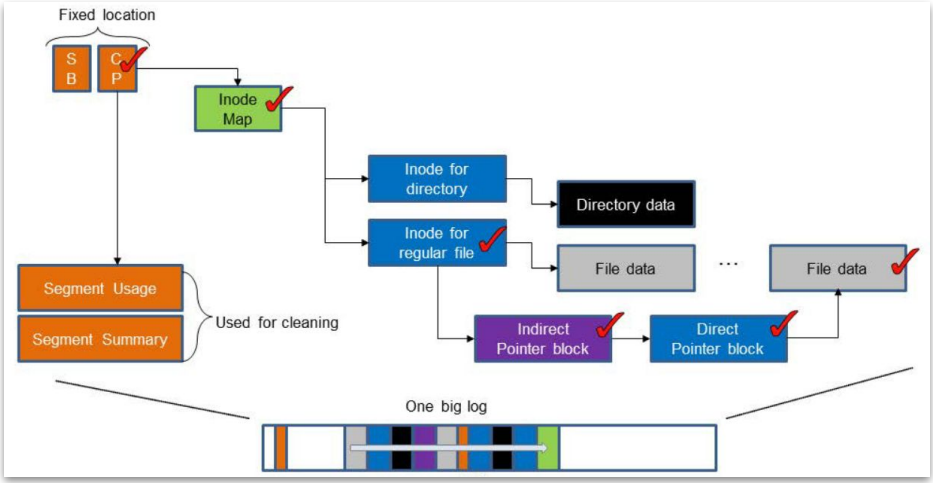
In original LFS: there is an **inode map** to translate an inode number to an on-device location (written at the end of the segment)

F2FS uses **the NAT table** to translate an inode number to its on-device location



This design solves an important problem with log-based file system: **Recursive update problem**

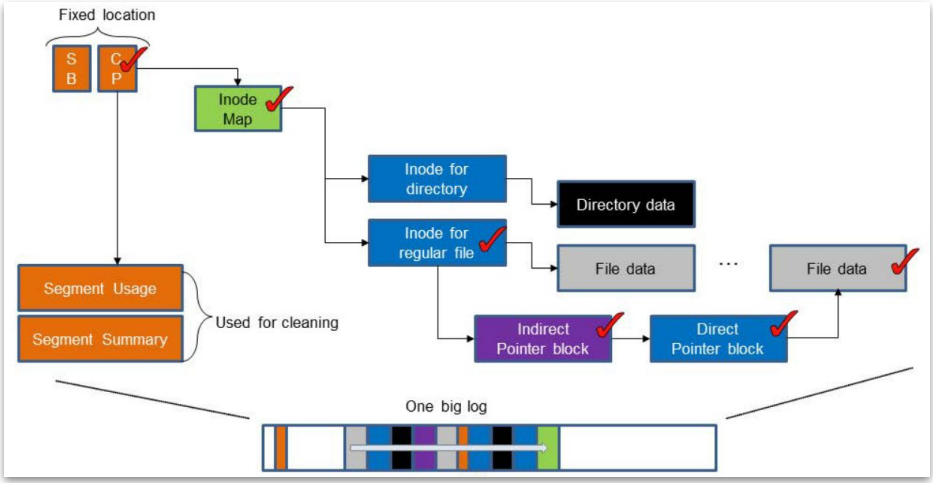
Update Propagation in LFS vs. F2FS

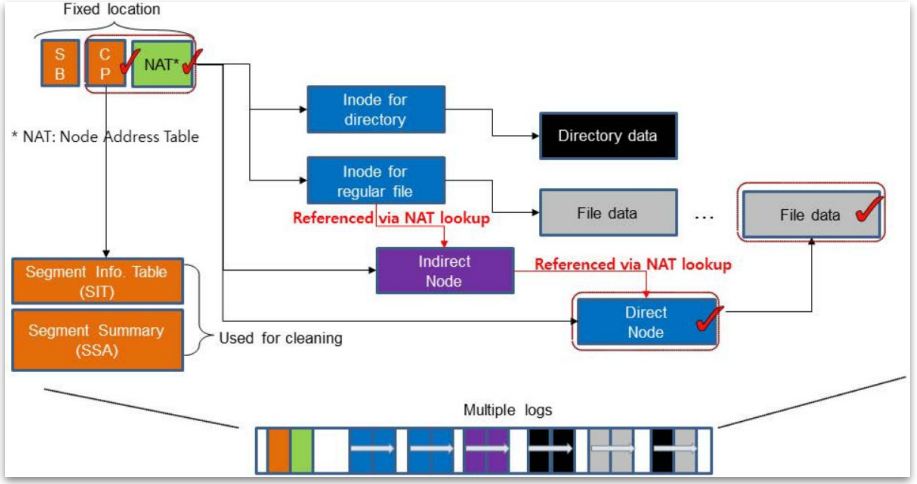


In a Log-Structured file system, updates at the bottom of the tree will be bubbled through the whole tree (updating device addresses) until reached at the top and a new inode map location is written - this is called ***recursive update problem*** *(also known as* ***Wandering Tree problem****)*

https://www.usenix.org/sites/default/files/conference/protected-files/fast15_slides_lee.pdf

Update Propagation in LFS vs. F2FS





In a Log-Structured file system, updates at the bottom of the tree will be bubbled through the whole tree (updating device addresses) until reached at the top and a new inode map location is written - this is called ***recursive update problem*** *(also known as* ***Wandering Tree problem****)*

In contrast, F2FS uses node numbers for indexing and only immediate parent is updated with further updates in-place in NAT (which is at a fixed location, Node Number → Device Address)

https://www.usenix.org/sites/default/files/conference/protected-files/fast15_slides_lee.pdf

Random writes in the NAT?

The idea (*I think*) is that it is a reasonable tradeoff to build a general purpose FS with good performance in most of the cases. The original Log-FS has many “*” for it to operate efficiently

LWN.net
News from the source

Content
Weekly Edition
Archives
Search
Kernel
Security

User: Password: Log in | Subscribe | Register

An f2fs teardown

...[f2fs] leaves a number of tasks to the FTL while focusing primarily on those tasks that it is well positioned to perform. So, for example, f2fs makes no effort to distribute writes evenly across the address space to provide wear-leveling.

...Some metadata, and occasionally even some regular data, is written via random single-block writes. This would be anathema for a regular log-structured file system, but f2fs chooses to avoid a lot of complexity by just doing small updates when necessary and leaving the FTL to make those corner cases work.

An f2fs teardown, https://lwn.net/Articles/518988/, October 2012
https://www.pinterest.com/pin/440226932298031698/

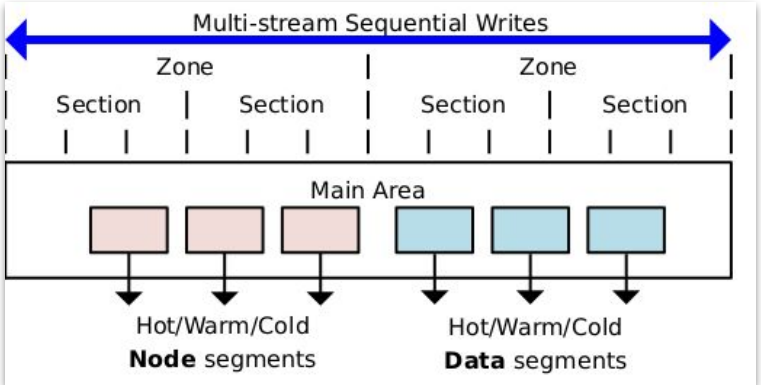
Multi-Headed stream logging

F2FS leverages device parallelism by opening multiple write segment streams

These streams are classified based on their hotness and separated in zones

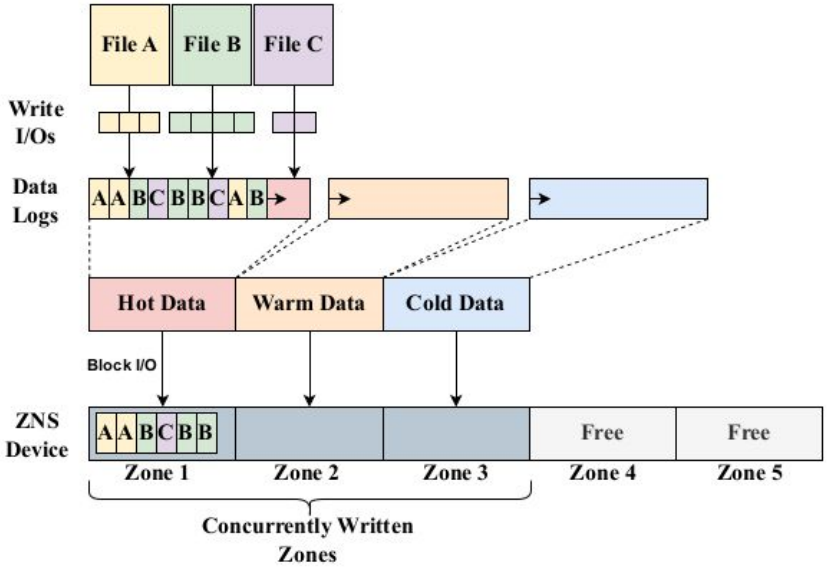
- Uses a simple classification (unlike SFS)
- Different types (table) put in different classes

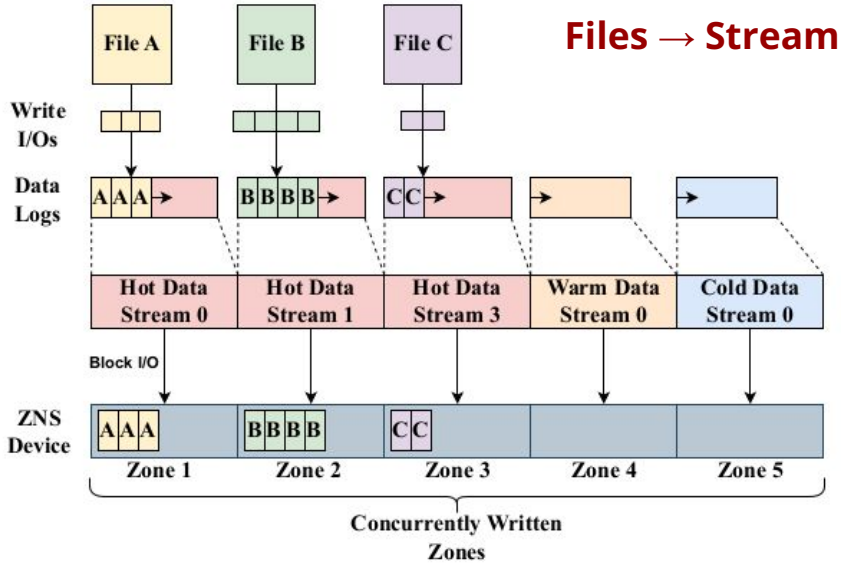
Different zones are mapped to different parallel units inside the flash



Type	Temp.	Objects
Node	Hot	Direct node blocks for directories
	Warm	Direct node blocks for regular files
	Cold	Indirect node blocks
Data	Hot	Directory entry blocks
	Warm	Data blocks made by users
	Cold	Data blocks moved by cleaning; Cold data blocks specified by users; Multimedia file data

On ZNS devices it can get more interesting …





Files → Stream maps

Nick Tehrany (MSc thesis, msF2FS - Design of Multi-Streamed F2FS, TU Delft)
https://repository.tudelft.nl/islandora/object/uuid%3A3c2b3e73-6aff-45f3-af43-31a50314b547

Victim selection and segment cleaning

Recall: Greedy and Cost-Benefit (CB) policies

- Greedy is simple, but perhaps not the most effective
- CB is more effective, but needs more homework

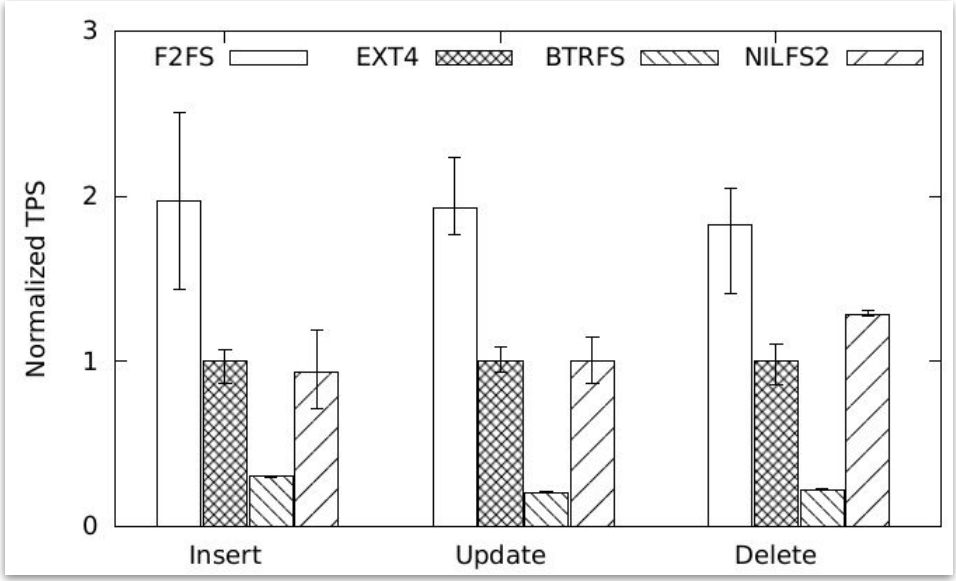
F2FS does two type of cleaning

- **Foreground:** when the free segments drop below a threshold, uses Greedy
- **Background:** routine, takes its time with a CB policy with hotness

The rest of the trick is the same, move the data from the victim segment to the buffer cache, and mark them dirty. They will be written down to the device in the due time. Not to erase the old blocks until the checkpoint-ing is done.

Also: F2FS dynamically switches between threading and cleaning for log management

F2FS performance



Thesis topic: *these numbers ought to be updated for the fastest flash we have*

SQLite workload on 3 different file systems, F2FS outperforms them all

(more detailed performance evaluation in the paper)

F2FS recap

Key choices in the design of F2FS

1. Flash-friendly data layouts : align fs GC unit (segments) with FTL gc unit
2. NAT updates to restrain writes update propagations
    a. Accepts random writes for the FS metadata regions
3. Multi-headed logging for parallelism
4. Adaptive logging (threading vs. cleaning) and GC policies (foreground and background)

It is highly influential work, and one of the few production quality code that we can test and benchmark

So far

You have seen the original **Log-Structured File System design** (Sprite FS)

- Design originally for disks, but fits perfectly with NAND flash too :)
- Typically “GC” is the Achilles Heel of any log-structured file system

SSD File system (SFS) that explores FS-assisted GC policies, but mostly kept the original Log-Structured layout

- File system maintains statistics actively for hotness on file blocks

F2FS, flash-friendly layouts with with *multi-headed* logging capabilities

All these file system assumed a conventional SSDs, can we think of something new to do here?

Thinking outside the (flash) box

All conventional file systems, do these three steps:

1. Determine a location (the on-device address) where to write data
2. Write data
3. Keep track of the location in the file system metadata

We will talk two unique file system designs *(there are more in literature)*:

- Direct File System for virtualized Flash (DFS) (2009)
- Nameless Writes (2012)

DFS: Direct File System (2010)

DFS: A File System for Virtualized Flash Storage

William K. Josephson
wkj@CS.Princeton.EDU

Lars A. Bongo
larsab@Princeton.EDU

David Flynn
dflynn@FusionIO.COM

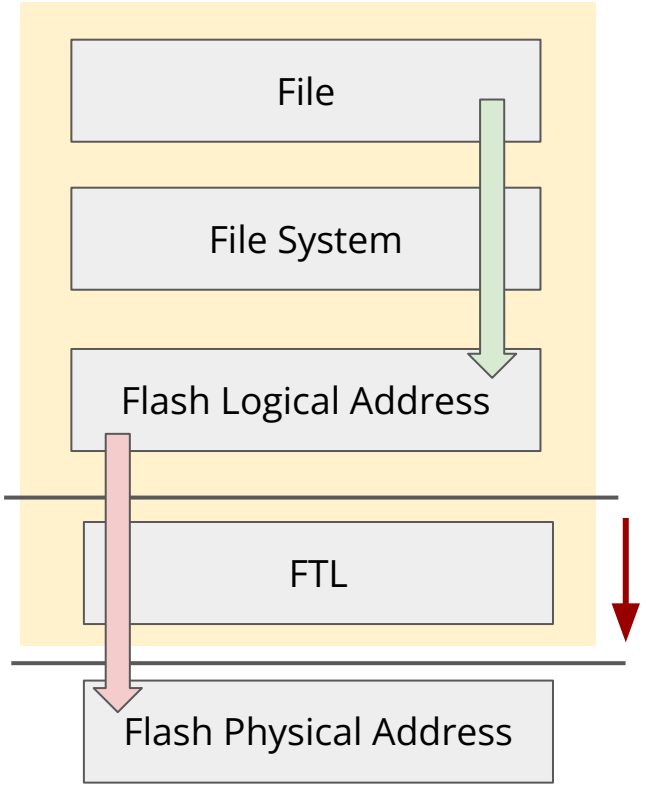
Kai Li
li@CS.Princeton.EDU

Abstract

This paper presents the design, implementation and evaluation of Direct File System (DFS) for virtualized flash storage. Instead of using traditional layers of abstraction, our layers of abstraction are designed for directly accessing flash memory devices. DFS has two main novel features. First, it lays out its files directly in a very large virtual storage address space provided by FusionIO's virtual flash storage layer. Second, it leverages the virtual flash storage layer to perform block allocations and atomic updates. As a result, DFS performs better and it is much simpler than a traditional Unix file system with similar functionalities. Our microbenchmark results show that DFS can deliver 94,000 I/O operations per second (IOPS) for direct reads and 71,000 IOPS for direct writes with the virtualized flash storage layer on FusionIO's ioDrive. For direct access performance, DFS is consistently better than ext3 on the same platform, sometimes by 20%. For buffered access performance, DFS is also consistently better than ext3, and sometimes by over 149%. Our application benchmarks show that DFS outperforms ext3 by 7% to 250% while requiring less CPU power.

timized for magnetic disk drives. Since flash memory is substantially different from magnetic disks, the rationale of our work is to study how to design new abstraction layers including a file system to exploit the potential of NAND flash memory.

This paper presents the design, implementation, and evaluation of the Direct File System (DFS) and describes the virtualized flash memory abstraction layer it uses for FusionIO's ioDrive hardware. The virtualized storage abstraction layer provides a very large, virtualized block addressed space, which can greatly simplify the design of a file system while providing backward compatibility with the traditional block storage interface. Instead of pushing the flash translation layer into disk controllers, this layer combines virtualization with intelligent translation and allocation strategies for hiding bulk erasure latencies and performing wear leveling.



DFS: Context

The year is 2010, flash is this new cool technology that is going to solve all our problems (*allegedly*)

- Host-based FTL designs are being explored
- FS exploration is happening, but not much is understood yet
- SSD device performance is increasing
- PCIe-attached is the way to attach flash storage (NVMe is not there yet)

This work is from Fusion-IO, the company that put flash on PCIe and run the FTL in the device driver on the host-CPU (no-embedded, device-side FTL)

- Attaching to the PCIe bus brings the device within CPU memory management
- Delivered 100K IOPS random read performance (!)

Virtualize Flash Storage: Key ideas

Instead of restrictive “N” block interface (where N is the capacity, like 1TiB) to a flash SSD, present a large 64-bit block address space (like the virtual memory)

Combine the two re-directions:

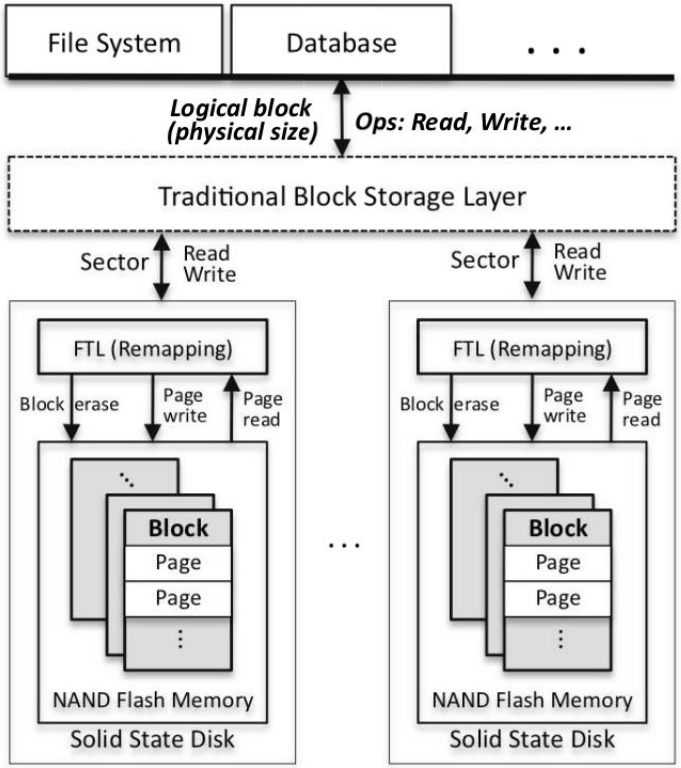
1. **FS-level**: from file to logical flash page
2. **FTL-level**: from logical flash page to physical location

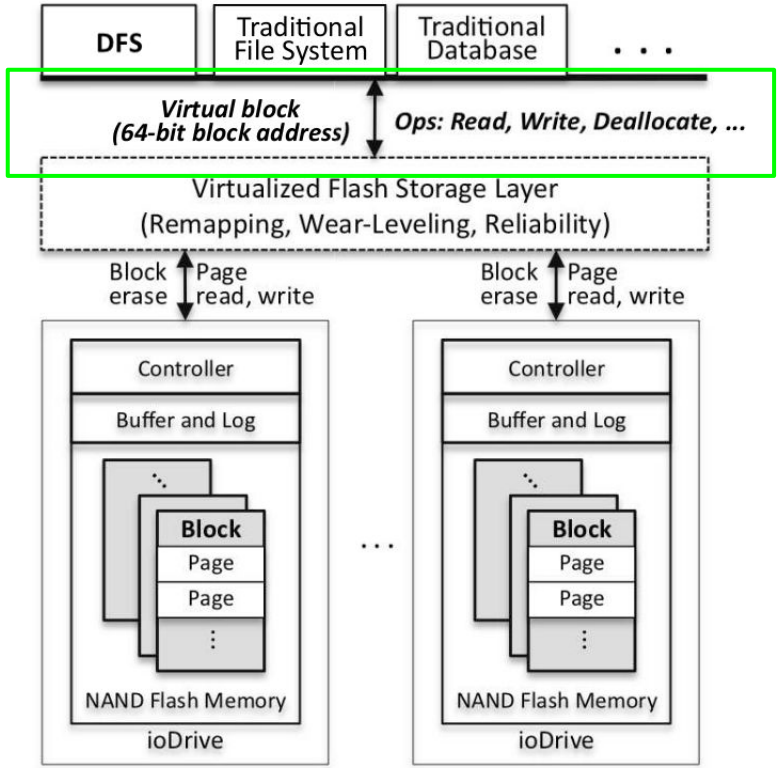
File systems is just responsible for choosing the most easy/lightweight layout for file management

Virtualized FTL: wear-leveling, remapping, and reliability

64-bit page addressing for 512 bytes pages in Fusion-IO flash, 2^{73} bytes space

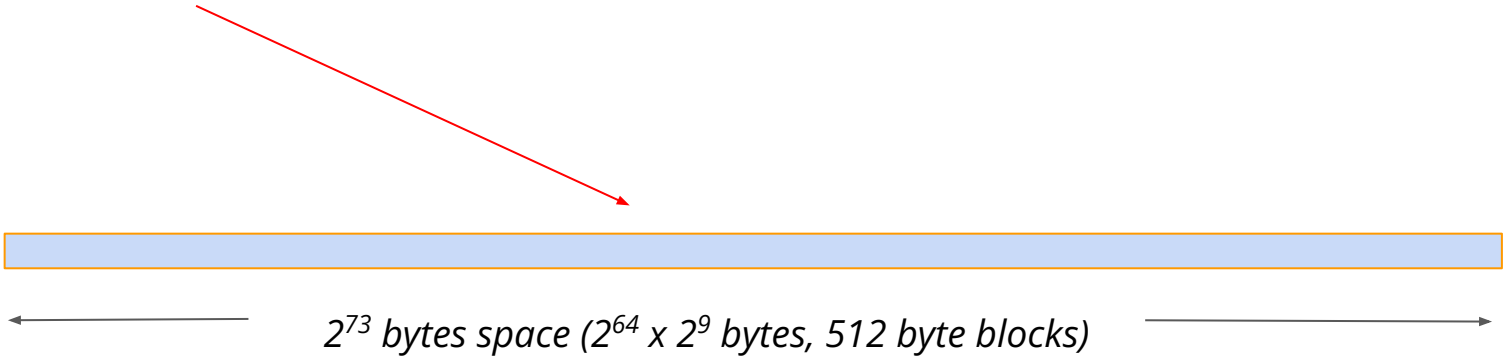
How does it look?



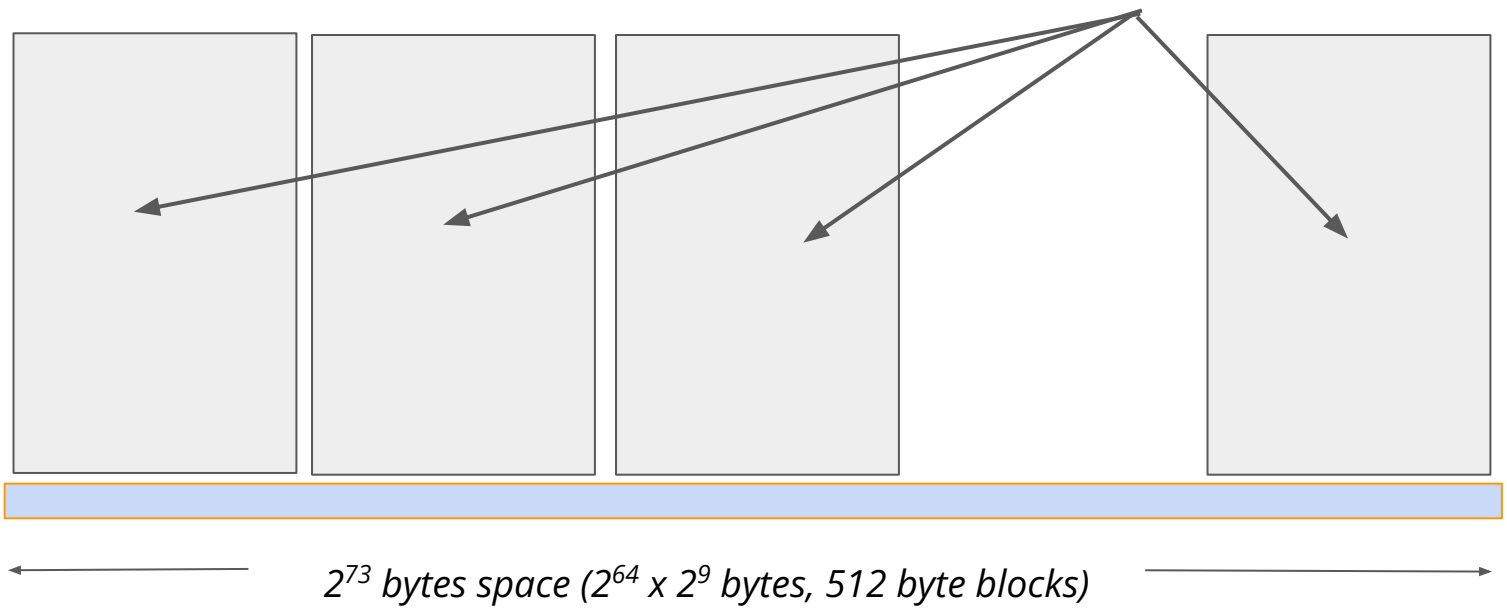


Building a simplified file system

Virtualized address space from flash FTL, 64 bits address identifying the 512 byte block



Building a simplified file system

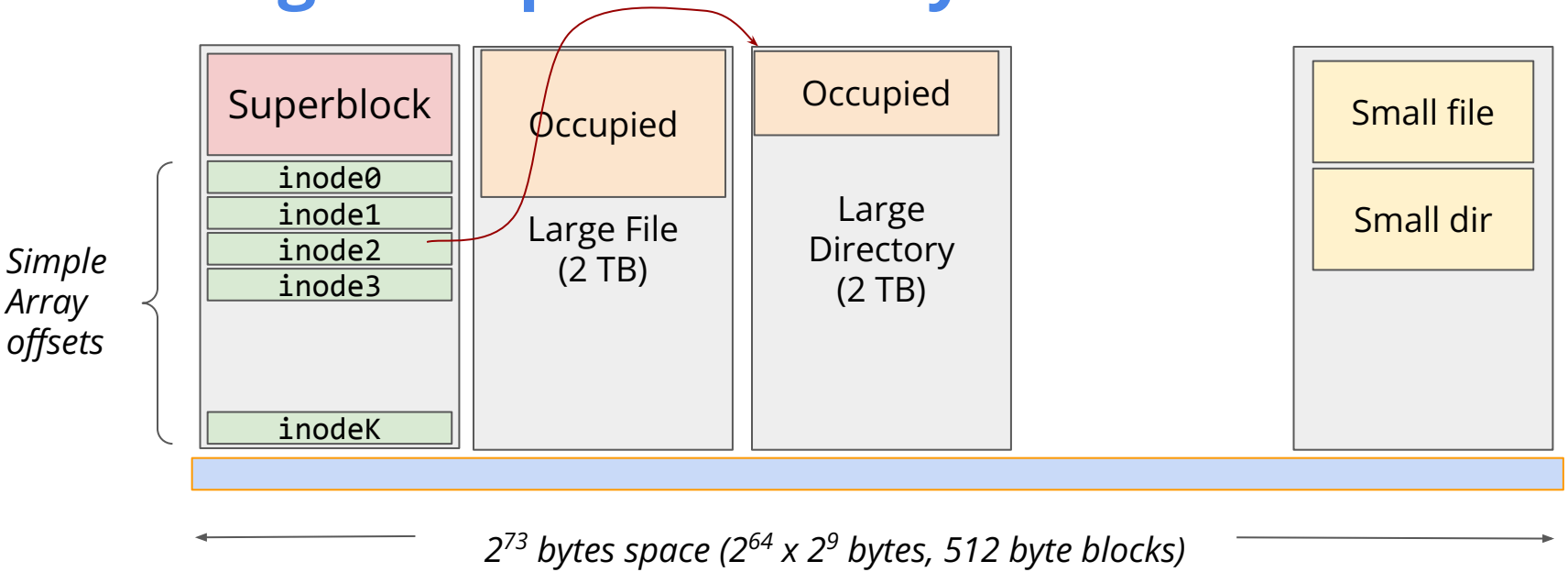


How many 2TB chunks there can be? $2^{73}/2^{41}$ = **2^{32} (hence, a 32 bit chunk addressing is enough)**

The 64 bit storage space is divided as : 2^{32} chunks of size 2^{32} x 512 bytes (2TiB).

Files/directories are divided into large and small. Large files gets full 2TiB (virtual) chunk, multiple smalls files are packed together in a single 2TiB chunk → how to do this classification? User-defined threshold

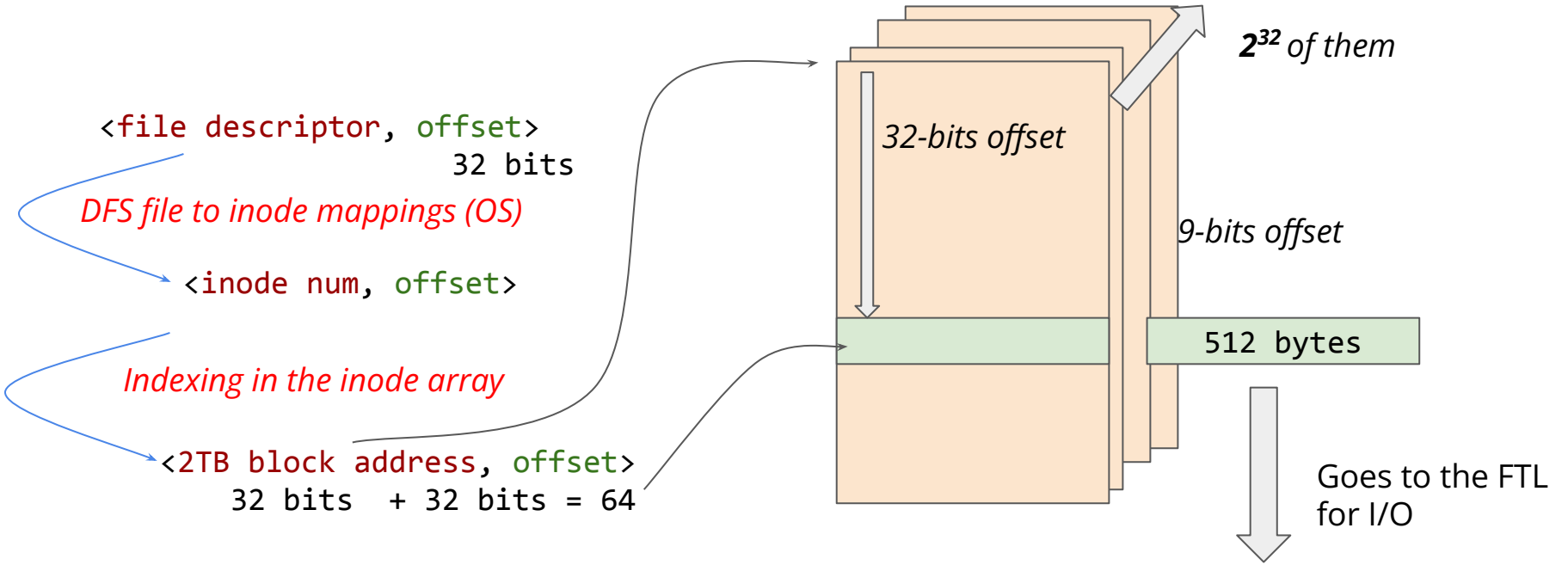
Building a Simplified File System



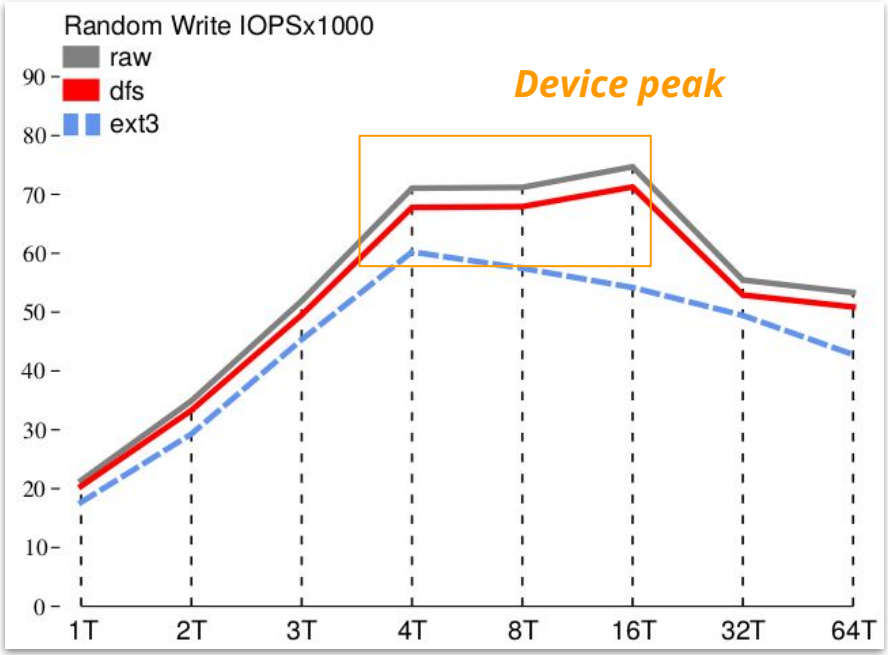
Inodes are 512 bytes, hence there are 2TB / 512 bytes number of inodes → 32 bits address
Inode location look-up is then trivial : inode_number → to location translation (**array offsets**)
Inode entries contain the virtual address of the allocated 2TB block, file type, creation time, access, etc.

File Offset Translation

Let's say I want to read a file at an offset (in the units of 512 bytes)



Does it help with performance?



Application	Wall Time		
	Ext3	DFS	Speedup
Quick Sort	1268	822	1.54
N-Gram (Zipf)	4718	1912	2.47
KNNImpute	303	248	1.22
VM Update	685	640	1.07
TPC-H	5059	4154	1.22

Yes, DFS deliver superior performance in microbenchmarks and in real world workloads

In Summary: DFS (2010)

Module	DFS	Ext3
Headers	392	1583
Kernel Interface (Superblock, *etc.*)	1625	2973
Logging	0	7128
Block Allocator	0	1909
I-nodes	250	6544
Files	286	283
Directories	561	670
ACLs, Extended Attrs.	N/A	2420
Resizing	N/A	1085
Miscellaneous	175	113
Total	3289	24708

Very simple and intuitive implementation
Complexity is avoided in

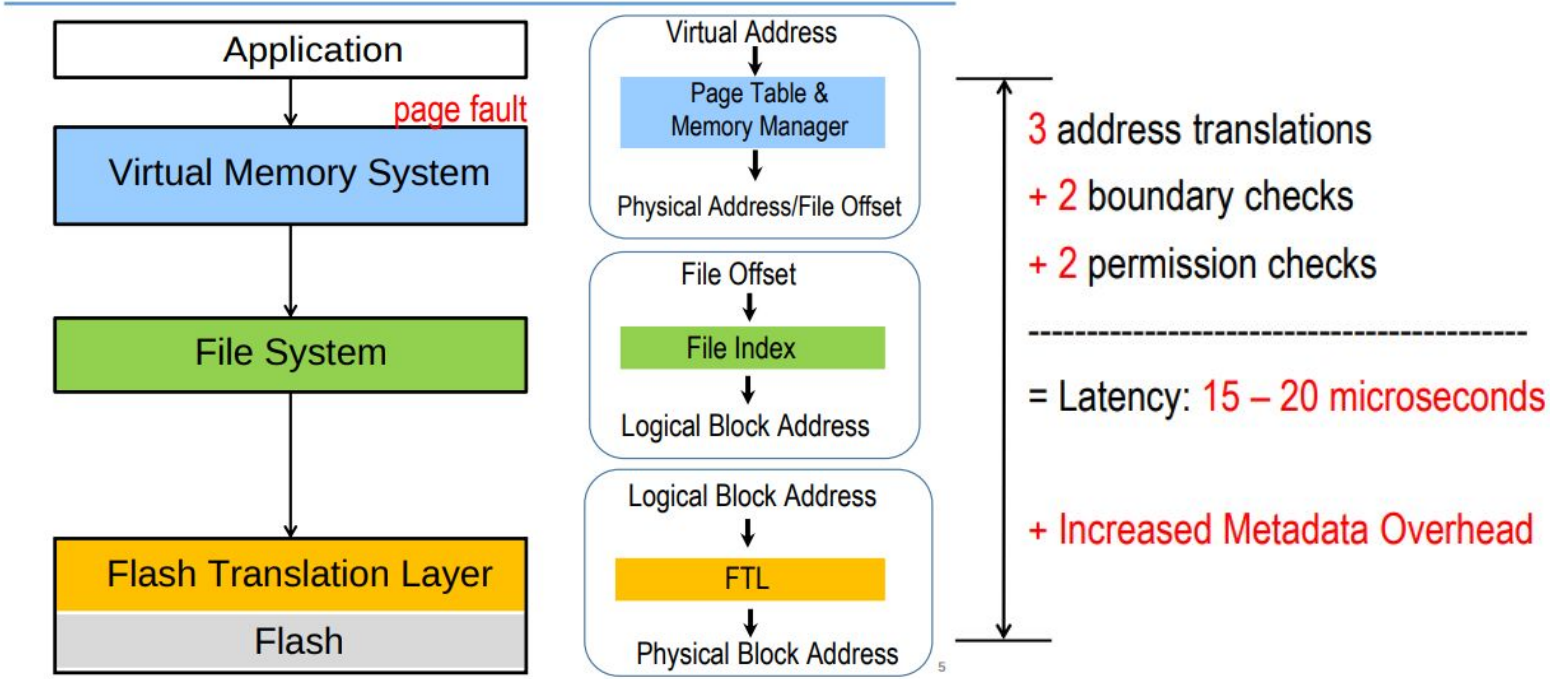
- inode management
- Allocation and logging

DFS has issues, the recovery logic needs support for atomic hardware logging

- Expensive device
- Consumes CPU cycles on the host

But overall, it is a pretty cool work that shows how to revise old abstractions and re-think ideas in presence of new technologies like NAND flash

Another cool project: FlashMap (2015)



Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. 2015. Unified address translation for memory-mapped SSDs with FlashMap. In Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15). https://doi.org/10.1145/2749469.2750420

Nameless Writes (2012)

De-indirection for Flash-based SSDs with Nameless Writes

Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
Computer Sciences Department, University of Wisconsin-Madison

Abstract

We present *Nameless Writes*, a new device interface that removes the need for indirection in modern solid-state storage devices (SSDs). Nameless writes allow the device to choose the location of a write; only then is the client informed of the *name* (i.e., address) where the block now resides. Doing so allows the device to control block-allocation decisions, thus enabling it to execute critical tasks such as garbage collection and wear leveling, while removing the need for large and costly indirection tables. We demonstrate the effectiveness of nameless writes by porting the Linux ext3 file system to use an emulated nameless-writing device and show that doing so both reduces space and time overheads, thus making for simpler, less costly, and higher-performance SSD-based storage.

1 Introduction

Indirection is a core technique in computer systems [28]. Whether in the mapping of file names to blocks, or a virtual address space to an underlying physical one, system

Unfortunately, the indirection such as found in many FTLs comes at a high price, which manifests as performance costs, space overheads, or both. If the FTL can flexibly map each virtual *page* in its address space (assuming a typical page size of 2 KB), an incredibly large indirection table is required. For example, a 1-TB SSD would need 2 GB of table space simply to keep one 32-bit pointer per 2-KB page of the device. Clearly, a completely flexible mapping is too costly; putting vast quantities of memory (usually SRAM) into an SSD is prohibitive.

Because of this high cost, most SSDs do not offer a fully flexible per-page mapping. A simple approach provides only a pointer per *block* of the SSD (a block typically contains 64 or 128 2-KB pages), which reduces overheads by the ratio of block size to page size. The 1-TB drive would now only need 32 MB of table space, which is more reasonable. However, as clearly articulated by Gupta et al. [16], block-level mappings have high performance costs due to excessive garbage collection.

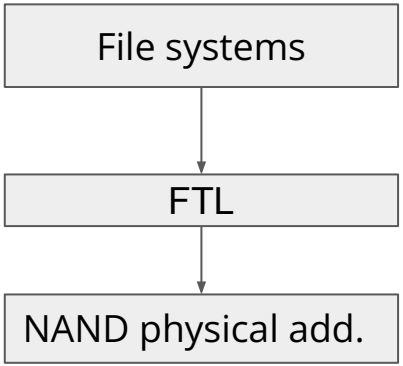
As a result, the majority of FTLs today are built us-

Key Challenge: Excessive Indirection

Redirection adds layer between two abstractions and an API

Very powerful idea in computer science

- Virtual Memory management (hides physical DRAM addresses)
- Virtualization (hides systems resources CPU, memory, devices)
- (here) FTL (hides low-level flash complexity)
- (also) DFS's flash virtualization is an example of indirection

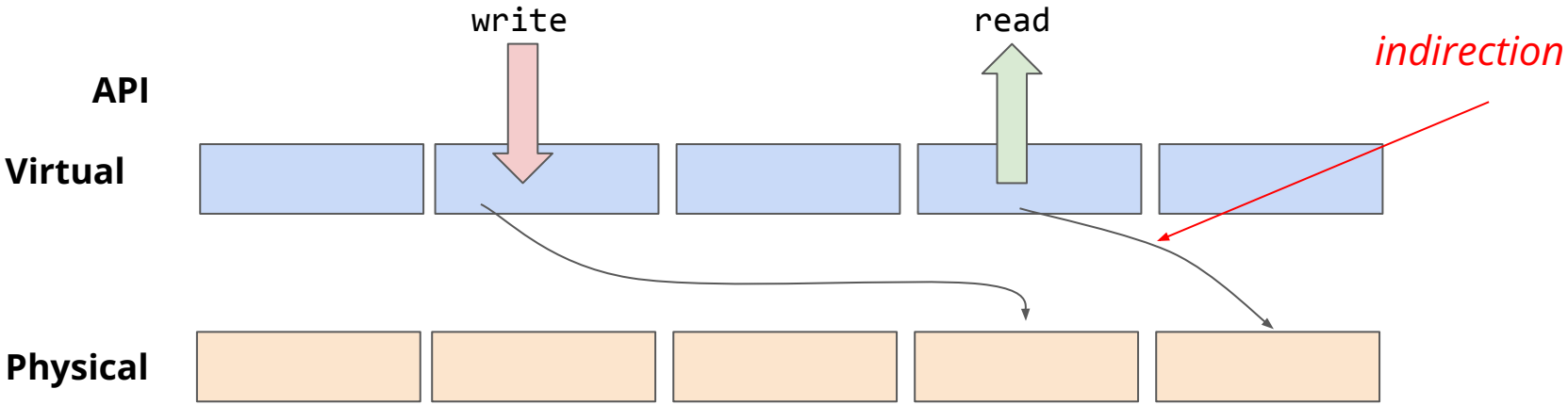


However, they come at a performance or complexity cost. The question here is given that what we know about FTL and the device internals, what can we do?

The Storage Device API and Indirection

How does a storage device API looks like? Where is the indirection?

- `write(sector/page/address, data, length)`
- `read(sector/page/address, data, length)`
- `trim(sector/page/address, length) // only useful for SSDs`



The Storage Device API and Indirection

How does a storage device API looks like? Where is the indirection?

- `write(sector/page/address, data, length)`
- `read(sector/page/address, data, length)`
- `trim(sector/page/address, length) // only useful for SSDs`

The problem comes from the fact that a file system (or any other storage service) tells the device "**where**" to write "**what**"

- The "**name**" or the identifier of the location is already given in the call

These are "named" writes. What if we don't tell the device where to write, only what to write, hence, the nameless writes?

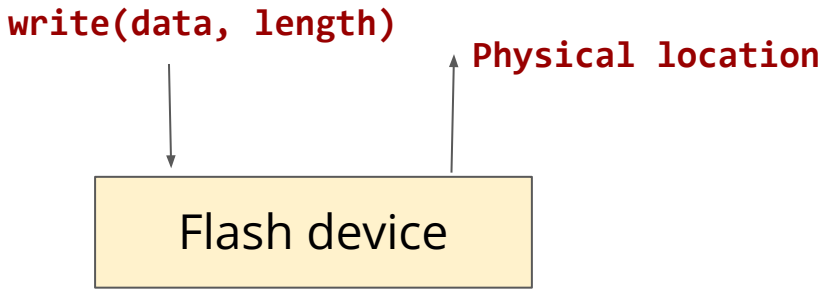
Nameless Writes Idea

Device physical addresses are exposed to the application like file system

Data is written directly directly on physical blocks

Device is free to choose the best location where to write data and notify the application

Maximum flexibility to the device



Challenges (*of course, why else we would do it*)? think of running a Log-Structured File system on it...

- Looking up stuff : everytime FS writes something we get a new address?
- Recursive update problem : inode map changes propagation ?
- GC, wear-leveling, copying -- what if a block is migrated inside the flash device?
- Anything else?

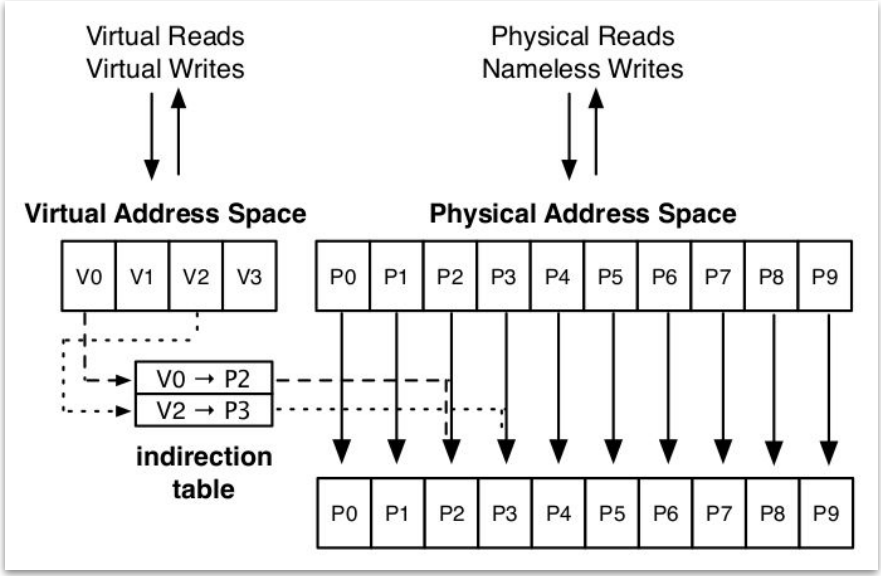
Segmented Address Space

Split the address space into two areas:

- Virtually addressed
- Physically addressed

Virtually addressed is “page-mapped” in the FTL for the best performance

- Small area, hence, low memory requirements for the FTL



This way virtually mapped areas are always addressable in a known location

- Super block, NAT tables, inode maps can be placed here
- Recursive updates terminate here

Nameless API so far

Physical API

- `uint64_t physical_write(data, length) → {paddresses, status}`
- `uint64_t physcial_read(paddress, length) → {data, status}`

Virtual API

- `virtual_write (vaddress, data, length) → {status}`
- `virtual_read (vaddress, data, length) → {status}`

So **what happens if a physical block is moved**, like during GC and wear-leveling? How does the file system know when a block is moved underneath it inside a device?

Callbacks and Metadata

To support free moving of data in physical blocks, Nameless API also introduced callback to file systems (or to any upper layer API)

- `callback → {old paddress, new paddress}`

However, now when a file system get an address "`0xdeadbeef`" is changed, how does it know which file/directory is this belong to?

- Sure, it has this information, but needs the full FS scan (not feasible)
- **Idea**: put a metadata pointer with all read/writes
  - Embed any useful pointers in these metadata, e.g., inode + version
  - Metadata stored in small **OOB flash areas** next to pages and written atomically

Nameless Device and API

Physical API

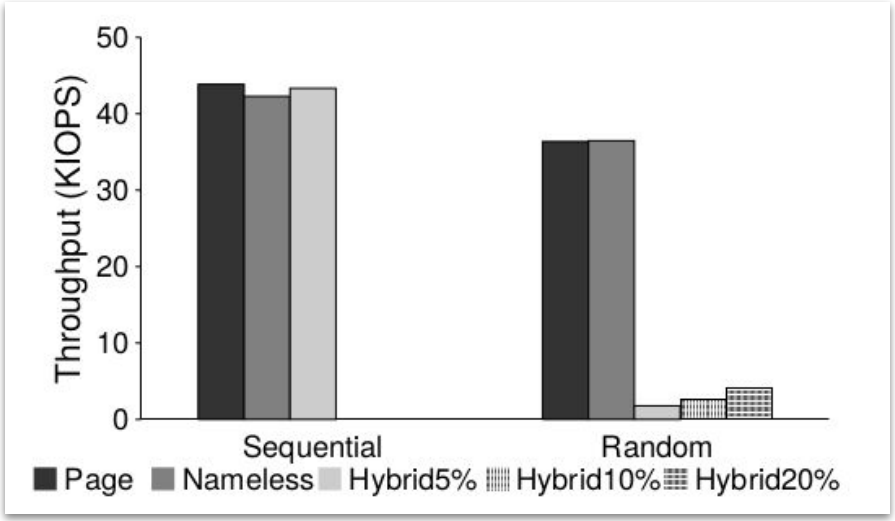
- `physical_write(data, length, mdata) → {paddr, status}`
- `physcial_read(paddr, length, mdata) → {data, status}`
- `physcial_overwrite(old_paddr(es), data, length, mdata} → {new paddr(es), status}`
- `callback → {old paddr(es), new paddr(es), mdata}`
- `free/trim(p/vaddr, length, mdata} → {status}`

Virtual API (not that interesting)

- `virtual_write (vaddress, data, length) → {status}`
- `virtual_read (vaddress, data, length) → {status}`

Evaluation: Nameless

Image Size	Page	Hybrid	Nameless
328 MB	328 KB	38 KB	2.7 KB
2 GB	2 MB	235 KB	12 KB
10 GB	10 MB	1.1 MB	31 KB
100 GB	100 MB	11 MB	251 KB
400 GB	400 MB	46 MB	1 MB
1 TB	1 GB	118 MB	2.2 MB



Nameless device performance closely to a page-mapped FTL without requiring high memory to maintain GBs of FTL mapping tables

Further ideas in the literature

1. Application-Managed Flash (USENIX FAST 2016)
    a. Completely expose flash chips to file systems and no in-place updates
    b. Breakdown recursive updated data structures into small blocks, and build an in-memory data structure at the time of mounting to capture updates
2. Para File system (USENIX 2016)
    a. Also exposes a very simple FTL to the file system exposing all device geometry
    b. Considering page allocation and striping to extract maximum performance
    c. Coordinated I/O scheduling between on-host GC threads and user writes

There is a large body of work out there regarding optimizing file systems for NAND flash storage devices

What we are not covering

Popular file system designs for raw-flash chips in embedded systems (FTL+FS):

- JFFS (The Journalling Flash File System), UBIFS (Unsorted Block Image File System), Yaffs (Yet Another Flash File System), NAFS (NAND flash memory Array File System), CFFS (Core Flash File System), NAMU (NAnd flash Multimedia file system), MNFS (novel mobile multimedia file system), ...

Typically they are build on similar ideas and concepts, but they

- Assume some sort of NOR byte-addressable location
- Focus on wear-leveling for a single class of applications (not server-class diverse workloads)
- Are not scalable to TBs of flash chips capacities

For further reading see…

A Survey on the Integration of NAND Flash Storage in the Design of File Systems and the Host Storage Software Stack

Survey done: July 2022

Nick Tehrany
Delft University of Technology
n.a.tehrany@vu.nl

Krijn Doekemeijer
Vrije Universiteit Amsterdam
k.doekemeijer@vu.nl

Animesh Trivedi
Vrije Universiteit Amsterdam
a.trivedi@vu.nl

Abstract

With the ever-increasing amount of data generate in the world, estimated to reach over 200 Zettabytes by 2025, pressure on efficient data storage systems is intensifying. The shift from HDD to flash-based SSD provides one of the most fundamental shifts in storage technology, increasing performance capabilities significantly. However, flash storage comes with different characteristics than prior HDD storage technology. Therefore, storage software was unsuitable for leveraging the capabilities of flash storage. As a result, a plethora of storage applications have been design to better integrate with flash storage and align with flash characteristics.

In this literature study we evaluate the effect the introduction of flash storage has had on the design of file systems, which providing one of the most essential mechanisms for managing persistent storage. We analyze the mechanisms for effectively managing flash storage, managing overheads of introduced design requirements, and leverage the capabilities of flash storage. Numerous methods have been adopted in file systems, however prominently revolve around similar design decisions, adhering to the flash hardware constrains, and limiting software intervention. Future design of storage software remains prominent with the constant growth in flash-based storage devices and interfaces, providing an increasing possibility to enhance flash integration in the host storage software stack.

1 Introduction

With the increasing amount of data, estimated to reach 200 Zettabytes by the year 2025 [182], efficient storage systems are becoming imperative. A large contribution factor to increased data generation is the gain in popularity for big data [9, 80, 124] and cloud services [6, 215]. While there exist a plethora of different storage technologies, the most prevalent type is *Hard Disk Drive (HDD)* [7, 48], which are now largely being replaced by *Solid State Drive (SSD)* [45]. HDD is one of the cheapest forms of storage, however is limited in performance due to requiring on mechanical movement to access data on the disk. This results in high latency for random access patterns [53, 102] and additionally increases power demand [29, 73]. While SSD is more expensive than HDD, it is becoming more affordable [180] and provides increased performance over HDD [119], resulting in a growing adoption for enterprise businesses [46, 146].

One of the most fundamental mechanisms of storing and organizing data on HDD, SSD, and other storage technologies, is through the use of file systems, enabling the structural organization of data on persistent storage media. Building efficient and performant file systems for the evolving storage media technologies and progressing with future demands of data storage is of paramount importance. With HDD having been the prevailing storage technology for decades, file system and application design revolved around the intrinsic characteristics of these devices. In particular, aiming to limit access patterns to sequential accesses [25, 190], in an effort to minimize mechanical movement on the disk and thus optimize their performance.

The most widely adopted type of SSD is based on *flash storage*, having different characteristics than traditional HDD. Performance of flash storage achieves several GB/s, with millions of *I/O Operations per Second (IOPS)* [91, 220], and access latency as low as single digit μ-second latency. However, flash storage has its own characteristics different from HDD. In particular, flash storage does not support in-place updates, requiring data to be erased at a larger unit in order to be written again. Additionally, the cost of erase operations is substantially higher than read and write operations [91, 235]. In order to hide these constraints from host systems, flash SSD employs firmware, called the *Flash Translation Layer (FTL)*, that exposes a sector-addressable interface. This allows SSD to be addressed in the same way as conventional HDD, requiring no changes in host software for accessing the different storage technologies.

While SSD and HDD utilize the same interfaces to be addressed, in order to exploit the increased performance benefits of flash storage, software must integrate with the characteris-

msF2FS:

Design and Implementation of an NVMe ZNS SSD Optimized F2FS File System

by

Nick Tehrany

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday, March 23rd, 2023, at 12:00 PM.

Student number:	5411726	
Thesis committee:	Prof. dr. ir. Alexandru Iosup,	TU Delft and VU Amsterdam, chair and daily supervisor
	Prof. dr. ir. Fernando A. Kuipers	TU Delft, committee member
	Dr. ir. Animesh Trivedi	VU Amsterdam, daily co-supervisor

An electronic version of this thesis is available at http://repository.tudelft.nl/.

(2023) Nick Tehrany, Krijn Doekemeijer, Animesh Trivedi, **A Survey on the Integration of NAND Flash Storage in the Design of File Systems and the Host Storage Software Stack,** https://arxiv.org/abs/2307.11866.
(2023) Nick Tehrany, msF2FS: Design and Implementation of an NVMe ZNS SSD Optimized F2FS File System, https://repository.tudelft.nl/islandora/object/uuid%3A3c2b3e73-6aff-45f3-af43-31a50314b547.

What you should know from this lecture

1. How did SSD influence the design of file systems
2. What is a Log-Structured File System and why it is the most popular-way to build flash-based file systems
3. What are they key design challenges when building a flash-based file system, choices for
    a. Layouts, GC policies, segmentation management
    b. Ideas presented with Sprite FS, SSD FS, and F2FS
4. New developments with the co-development of FTL and FS semantics
    a. DFS, and Nameless writes

Next week: Flash-based Key-Value Stores

References

- Mendel Rosenblum and John K. Ousterhout. 1991. The design and implementation of a log-structured file system. SIGOPS Oper. Syst. Rev. 25, 5 (Oct. 1991), 1–15.
- Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. 2012. SFS: random write considered harmful in solid state drives. In Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST'12). USENIX Association, USA, 12.
- Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. 2015. F2FS: a new file system for flash storage. In Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15). USENIX Association, USA, 273–286.
- William K. Josephson, Lars A. Bongo, David Flynn, and Kai Li. 2010. DFS: a file system for virtualized flash storage. In Proceedings of the 8th USENIX conference on File and storage technologies (FAST'10). USENIX Association, USA, 7.
- Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. De-indirection for flash-based SSDs with nameless writes. In Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST'12). USENIX Association, USA, 1.
- Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind Arvind. 2016. Application-managed flash. In Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST'16). USENIX Association, USA, 339–353.
- Jiacheng Zhang, Jiwu Shu, and Youyou Lu. 2016. ParaFS: a log-structured file system to exploit the internal parallelism of flash devices. In Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16). USENIX Association, USA, 87–100.