F2FS: A New File System for Flash Storage

Changman Lee, Dongho Sim, Joo-Young Hwang,

and Sangyeun Cho

*S/W Development Team, Memory Business

Samsung Electronics Co., Ltd.

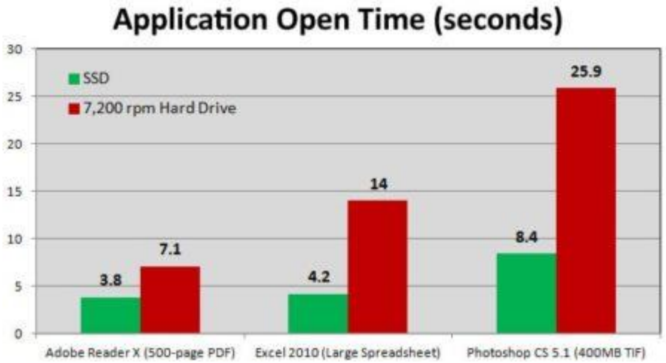
USENIX FAST'15

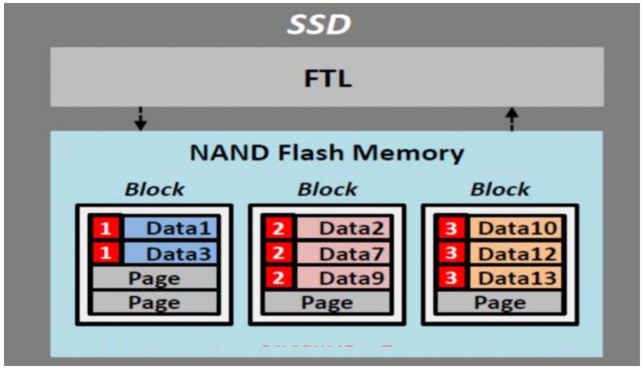
Contents

- Introduction
- Design and Implementation of F2FS
- Evaluation
- Conclusion

Introduction

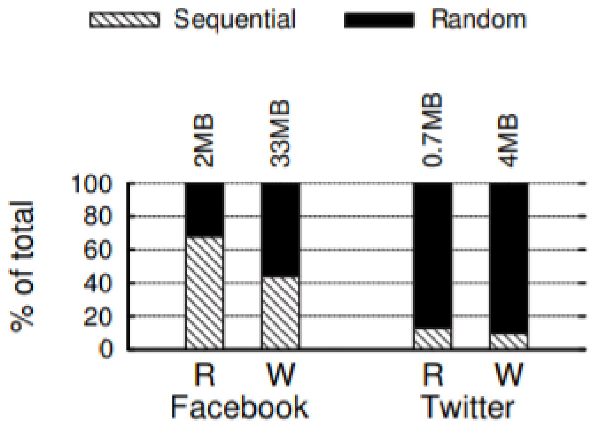
- The Rise of SSDs
  - Much faster than HDDs
- NAND Flash Memory
  - Erase-before-write
  - Sequential writes inside the erase unit
  - Limited Program/Erase (P/E) cycle
- Flash Translation Layer
  - Garbage collection
  - Wear-leveling
  - Bad block management
- Issues
  - Poor random write performance
  - Life span and reliability

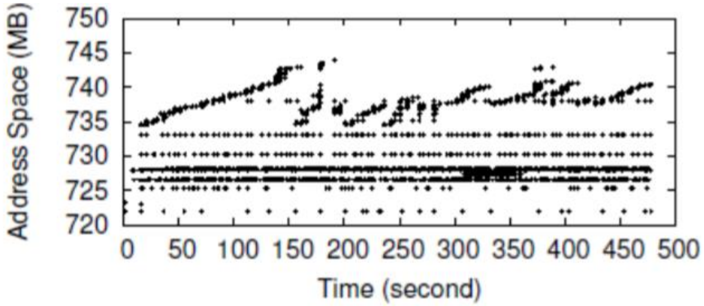




Introduction (cont.)

- I/O Patterns
  - Random writes to flash storage devices are bad
    - Free space fragmentation
    - Lifetime reduction
    - Performance degradation
  - Sequential writes are preferred by flash storage devices
    - Log-structured file systems
    - Copy-on-write file systems

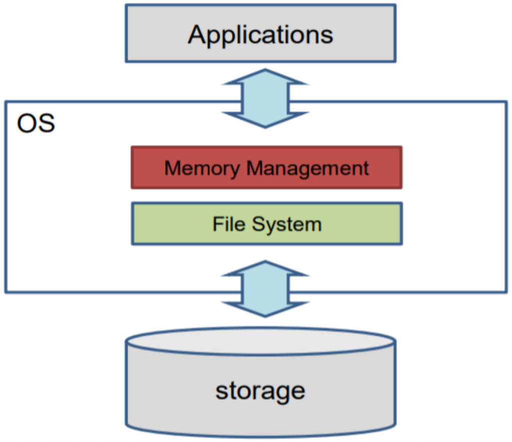


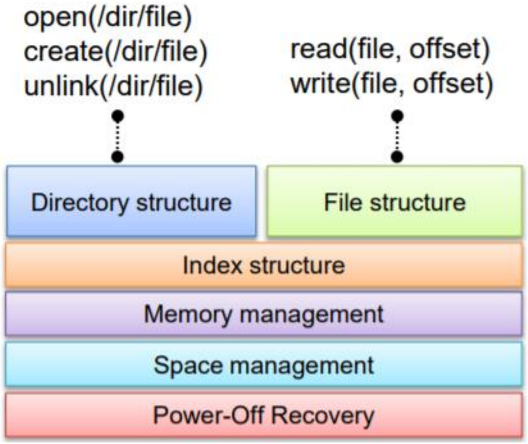


Reference: Revisiting Storage for Smartphones, Kim et al., USENIX FAST 2012

Introduction (cont.)

- File System
  - Serves directory and file operations to users
  - Manages the whole storage space



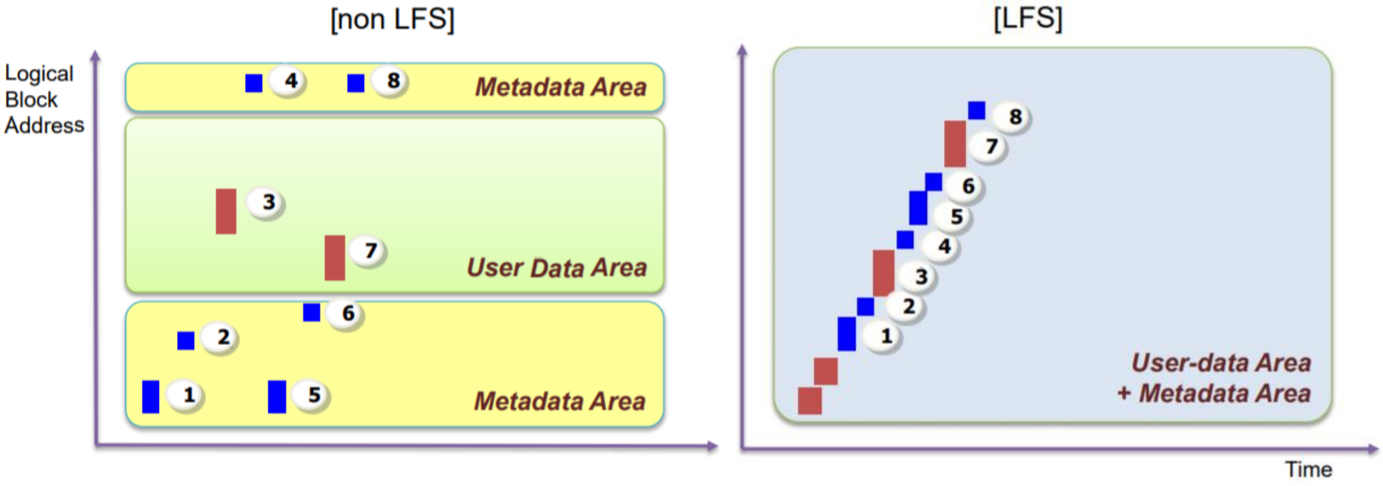


- Conventional file systems
  - Optimized for HDDs
  - No consideration of SSDs' characteristics

Background

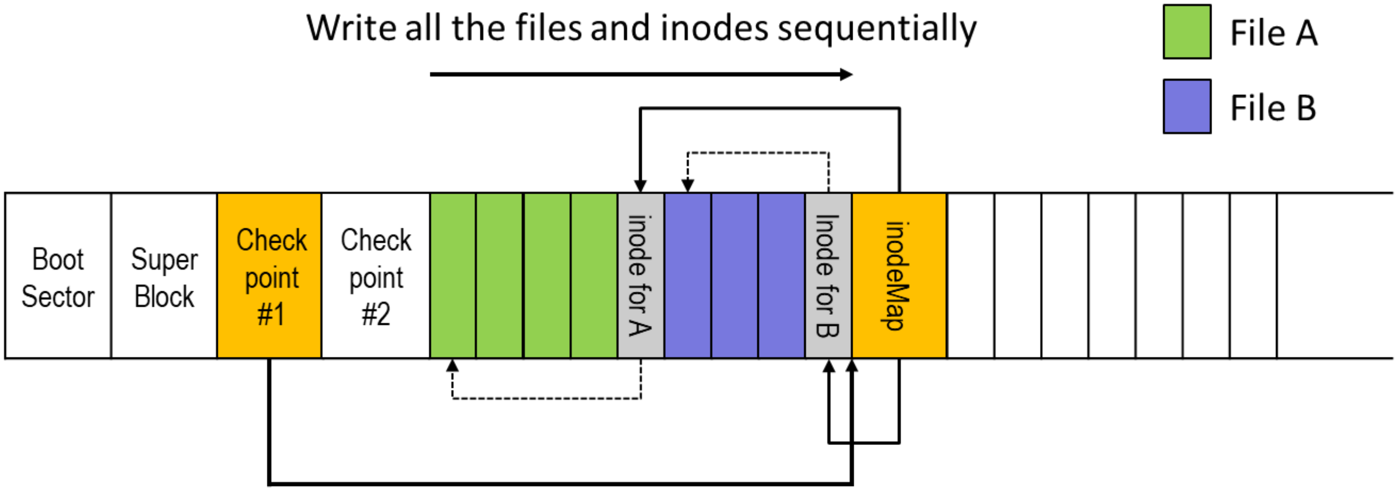
Log-structured **F**ile **S**ystem(LFS)

- Sequential write is preferred by SSDs
- Fits well to SSDs
- Assuming the *whole disk space* as a big log
  Write data and metadata *sequentially*



Log-structured File System

- Log-structured file systems (LFS)
  - Treats a storage space as a huge log
  - Appends all files and directories sequentially
  - The state-of-the-art file systems are based on LFS or CoW
    - e.g., Sprite LFS, F2FS, NetApp's WAFL, Btrfs, ZFS, …



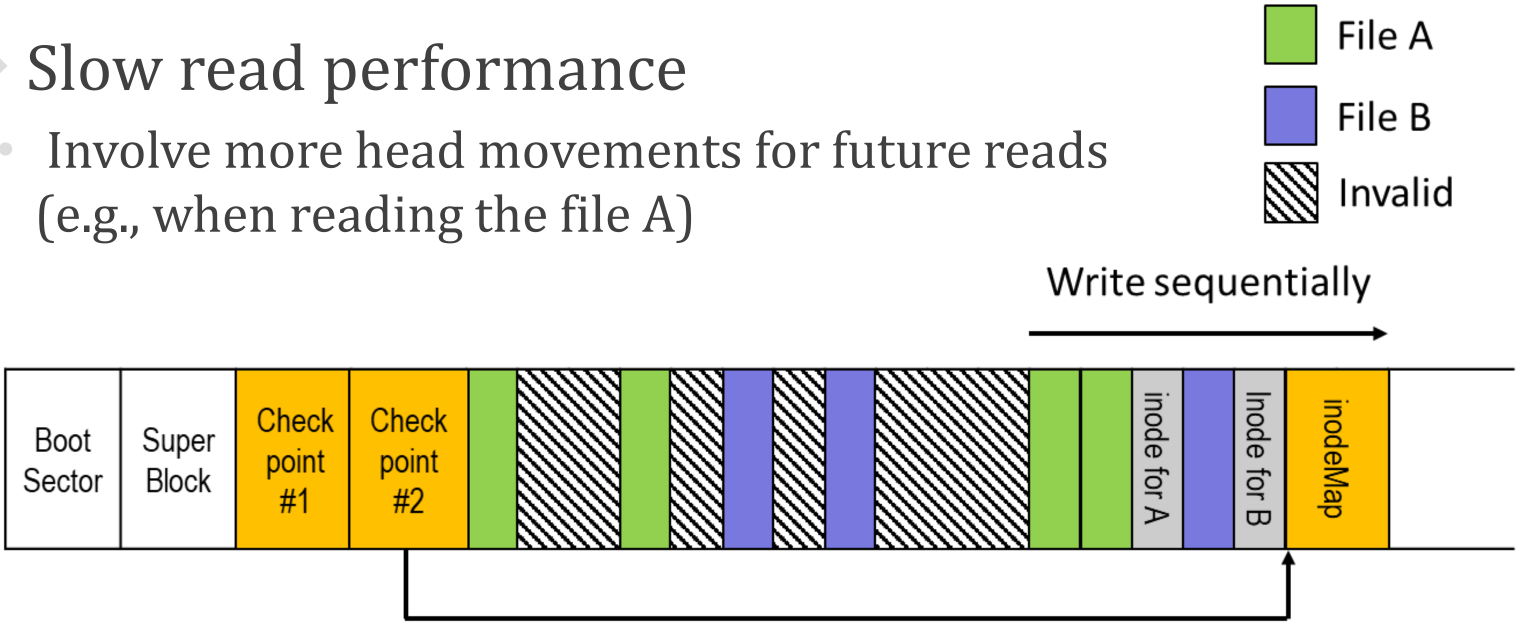
Log-structured File System (Cont.)

- Advantages
  - **(+)** No consistent update problem
  - **(+)** No double writes – an LFS itself is a log!
  - **(+)** Provide excellent write performance – disks are optimized for sequential I/O operations
  - **(+)** Reduce the movements of disk headers further (e.g., inode update and file updates)

- Disadvantages
  - **(–)** Expensive garbage collection cost
  - **(–)** Slow read performance

Disadvantages of LFS

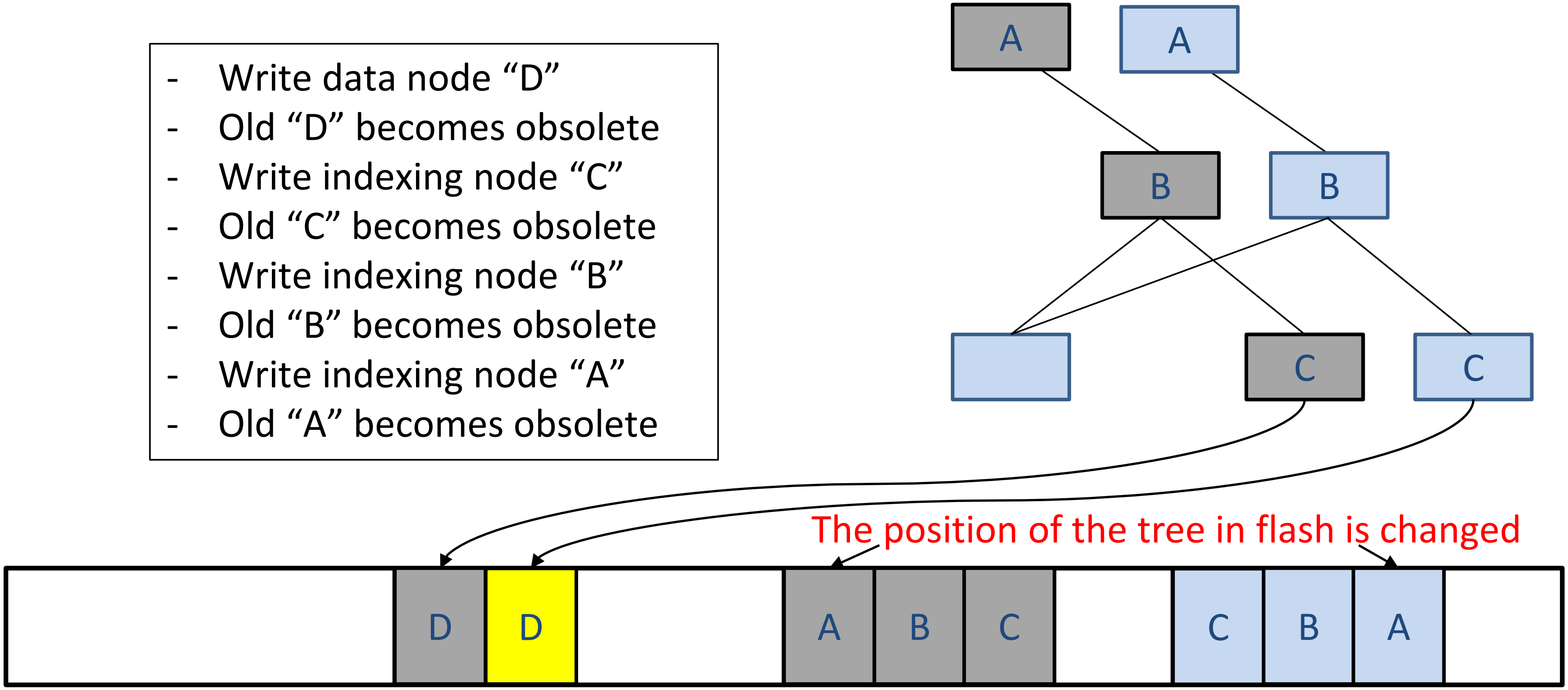
- Expensive garbage collection cost
  - Invalid blocks must be reclaimed for future writes; otherwise, free disk space will be exhausted
- Slow read performance
  - Involve more head movements for future reads (e.g., when reading the file A)



LFS: Wandering Tree Problem

- How to find the root of the tree?

- Write data node “D”
- Old “D” becomes obsolete
- Write indexing node “C”
- Old “C” becomes obsolete
- Write indexing node “B”
- Old “B” becomes obsolete
- Write indexing node “A”
- Old “A” becomes obsolete



The position of the tree in flash is changed

Contribution

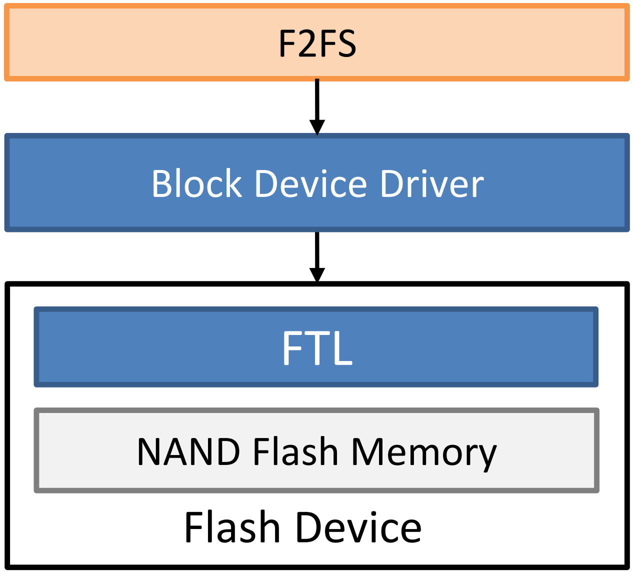
- Design of F2FS
  - Flash Awareness
    - *Alignment* of file system data structures with operational units in FTL
  - Wandering Tree Problem
    - Use a term, "*node*", that represents inodes as well as various pointer blocks
    - Introduce **NAT**(Node Address Table) containing the locations of all node blocks
  - Cleaning Overhead
    - Support a *background cleaning*
    - Support two different *victim selection* policies
    - Support *multi-head logs* for static hot and cold *data separation*
    - Introduce *adaptive logging* for efficient block allocation

Contents

- Introduction
- Design and Implementation of F2FS
  - On-Disk Layout
  - Index Structure
  - Multi-head Logging
  - Cleaning
  - Adaptive Logging
  - Checkpointing and Recovery
- Evaluation
- Conclusion

F2FS: Flash-friendly File System

- Log-structured file system for FTL devices
  - It runs atop FTL-based flash storage and is optimized for it
  - Exploit *system-level information* for better performance and reliability (e.g., better hot-cold separation, background GC, …)

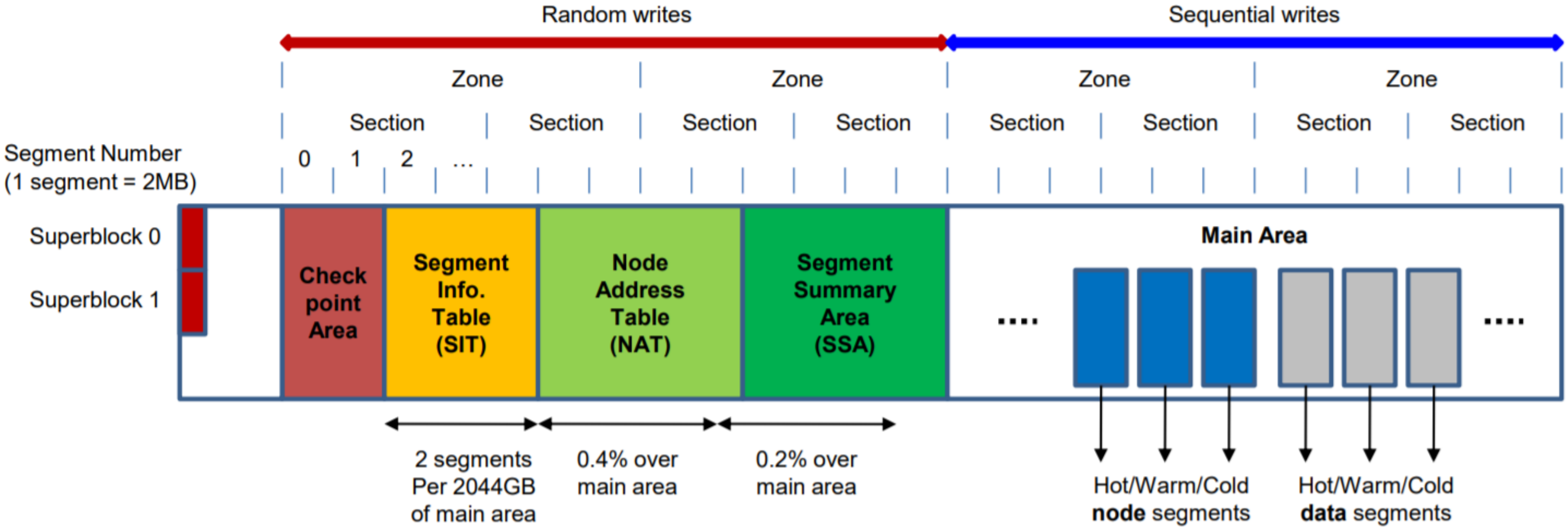


F2FS

On-Disk Layout

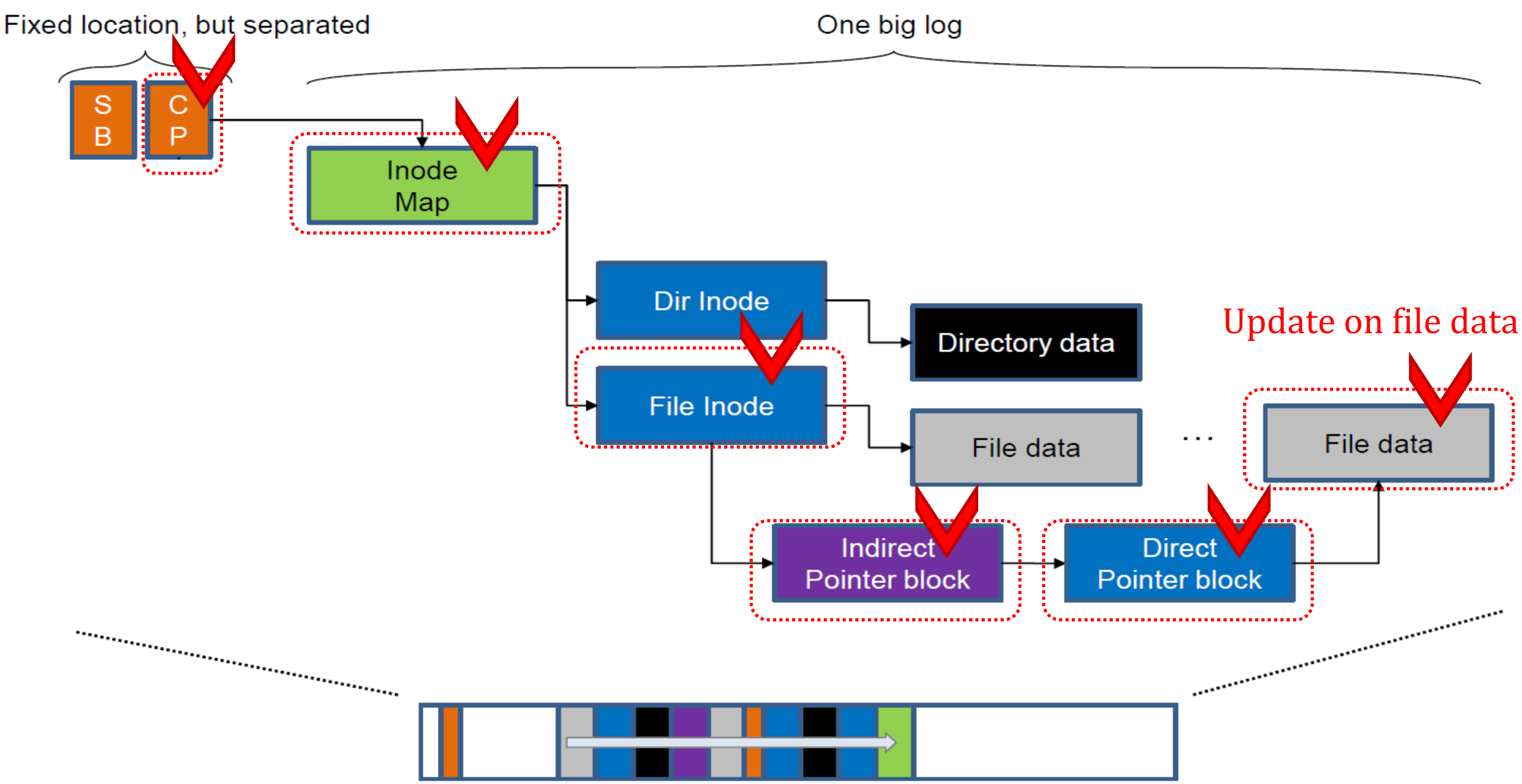
- Flash Awareness
  - All the file system metadata are located together for locality
  - Start address of main area is aligned to the *zone** size
  - Cleaning operation is done in a unit of section

Block: 4KB
Segment:2MB
Section: FTL GC unit (n Segments)
Zone*: data block group



LFS Index Structure

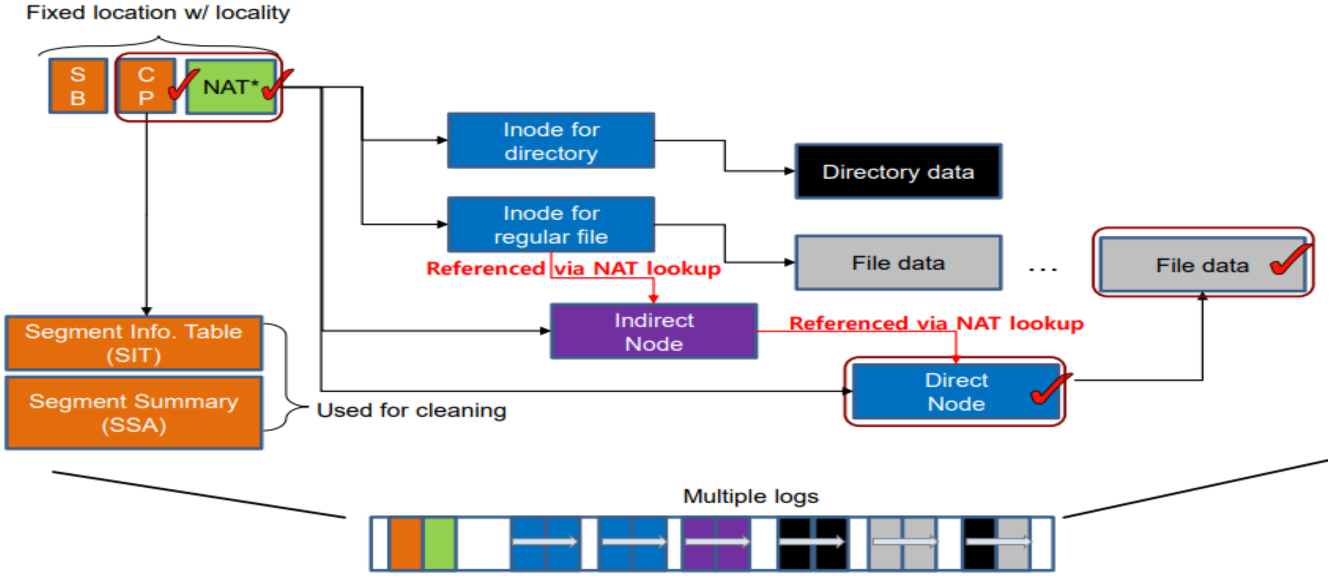
- Suffer from the wandering tree problem



Update on file data

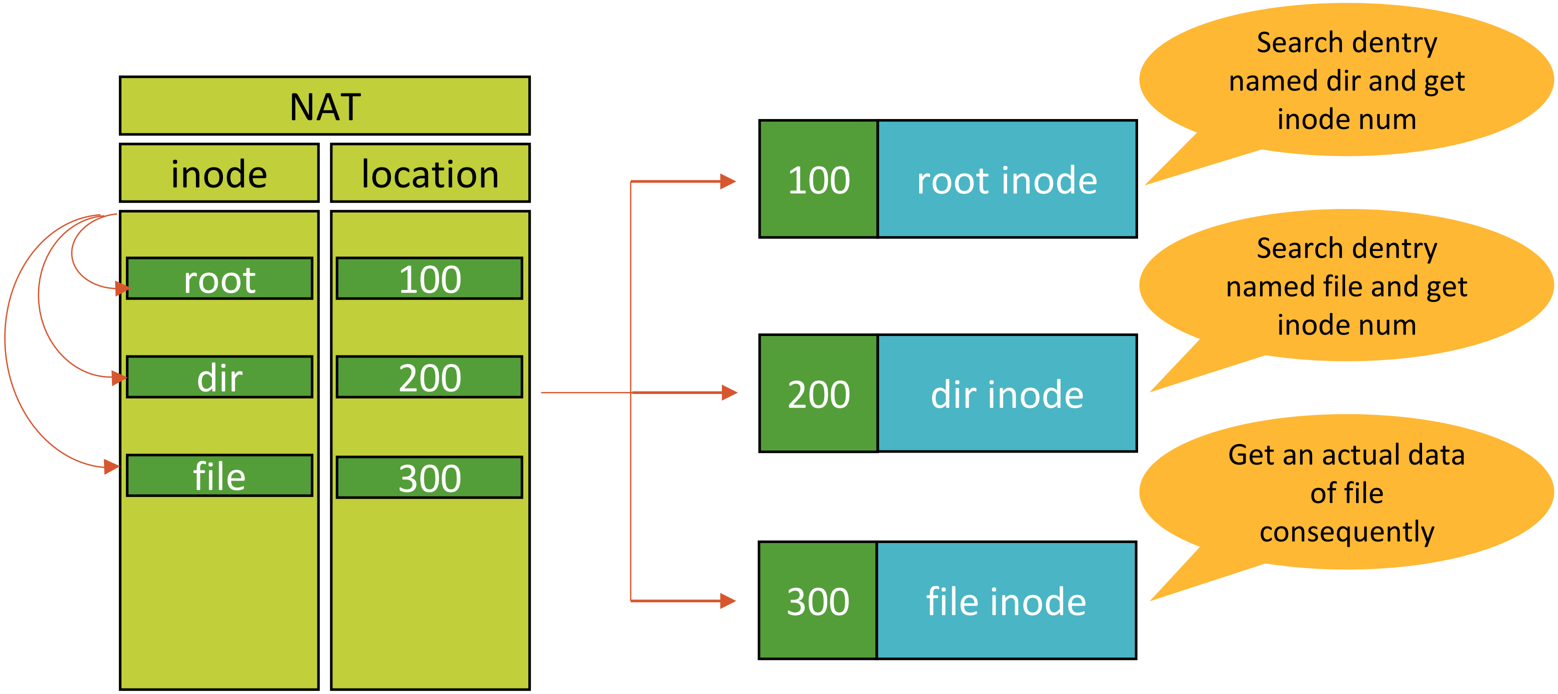
F2FS Index Structure

- **Node Address Translation (NAT)**
  - Containing the locations of all the node blocks
  - Allows us to eliminate the wandering tree problem



Example: File Look-up

- open ("/dir/file")



Search dentry named dir and get inode num

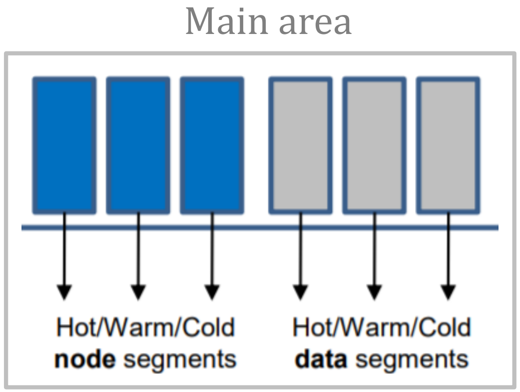
Search dentry named file and get inode num

Get an actual data of file consequently

Multi-head Logging

- Data Temperature Classification

Type	Temperature	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



- Separation of multi-head logs in NAND flash
  - Zone-aware log allocation for set-associative mapping FTL

Cleaning

Cleaning Process

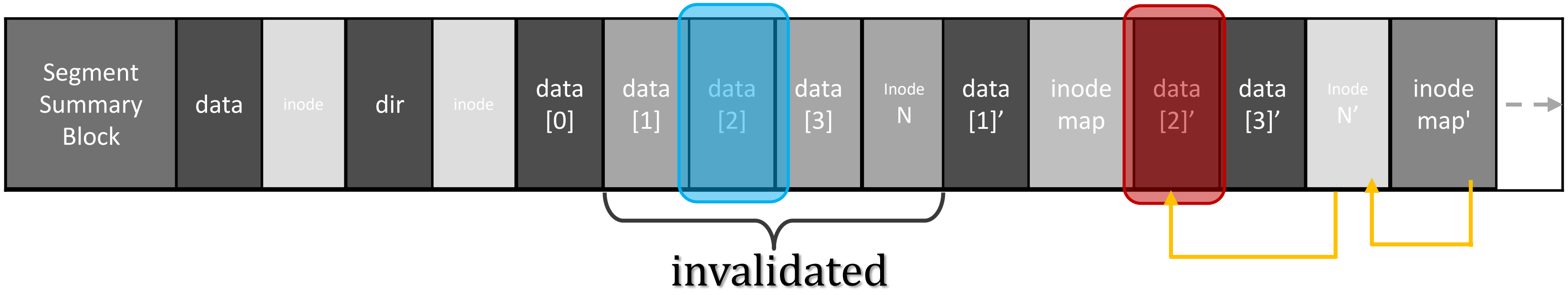
- Reclaim *obsolete data scattered* across the whole storage for a new empty log space
- Get victim segments
- Load parent index structures from segment summary blocks
- Move valid data by checking their cross-reference

Foreground Cleaning

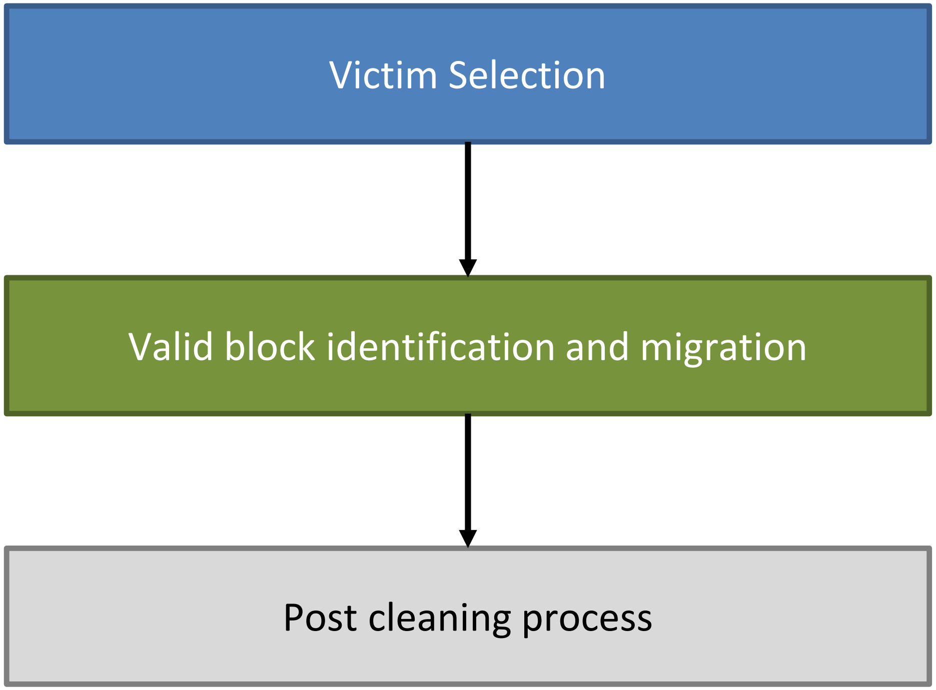
- Triggered when there are not enough free sections

Background Cleaning

- A kernel thread doing the cleaning job periodically at idle time



Cleaning process in F2FS



Victim Selection

Valid block identification and migration

Post cleaning process

Victim Selection

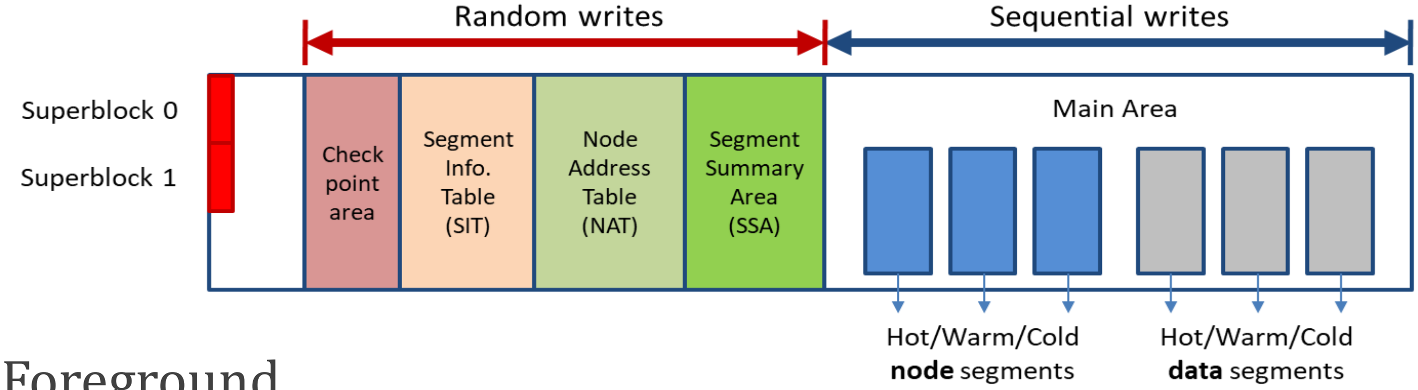
- Greedy Policy
  - The cleaner chooses the smallest number of valid blocks
  - Used in foreground cleaning

- Cost-Benefit Policy
  - The cleaner chooses a victim not only based on its utilization but also its age
  - Age of a section: average of the age of segments in the section
  - Another chance to separate hot and cold data
  - Used in background cleaning

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

Valid Block Identification and Migration

- Identification
  - Scan through a validity bitmap per segment in SIT and identify valid blocks
  - Retrieve parent node blocks containing their indices from a SSA information



- Foreground
  - Migrate valid blocks to other free logs
- Background
  - load the blocks into page cache and mark them as dirty

Post Cleaning Process

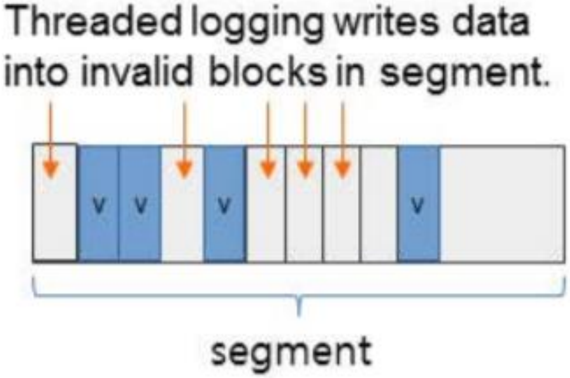
- Pre-free section
  - Victim section is marked as a *“pre-free”* section
  - Pre-free sections are freed after the next checkpoint is made
  - To prevent losing data referenced by a previous checkpoint when unexpected power outage occurs

Adaptive Logging

Adaptive Write Policy

- Normal write policy
  - Logging to a clean segment
  - Need cleaning operations if there is no clean segment
  - Cleaning causes mostly random read and *sequential writes*

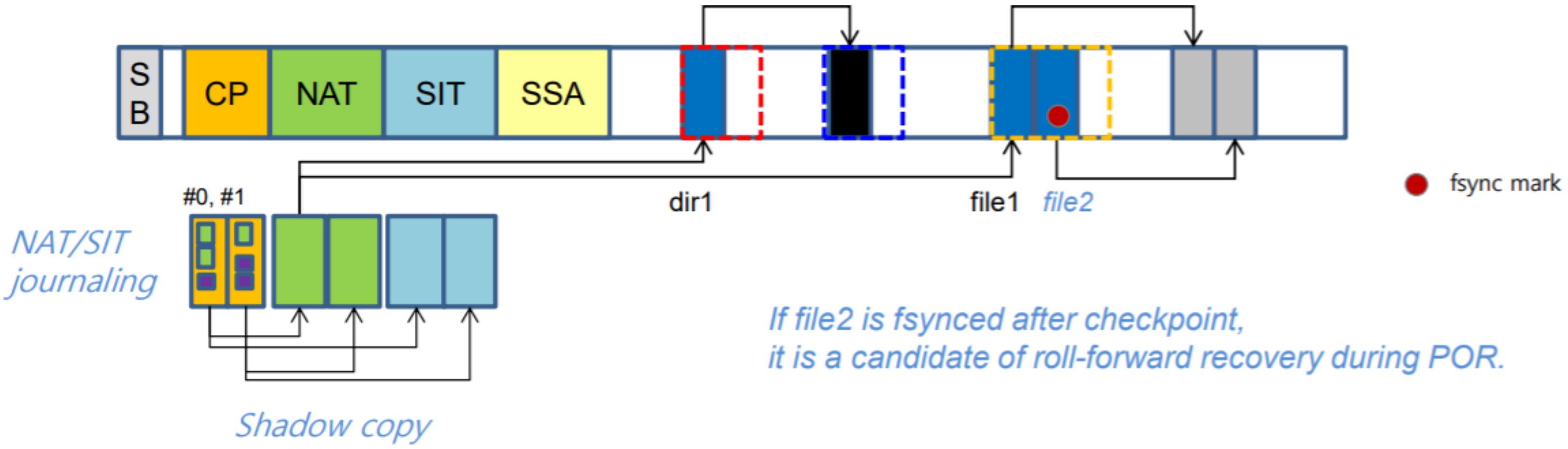
- Threaded logging
  - If there are not enough *clean segments*
  - Reuse obsolete blocks in a dirty segment
  - No need to run cleaning
  - May cause random writes (in a small range)



Threaded logging writes data into invalid blocks in segment.

Sudden Power Off Recovery

- Checkpoint and rollback
  - Maintain shadow copy of checkpoint, NAT, SIT blocks
  - Recover the latest checkpoint
  - Keep NAT/SIT journal in checkpoint to *avoid NAT, SIT writes*
  - Roll-forward recovery to recover fsync'ed data



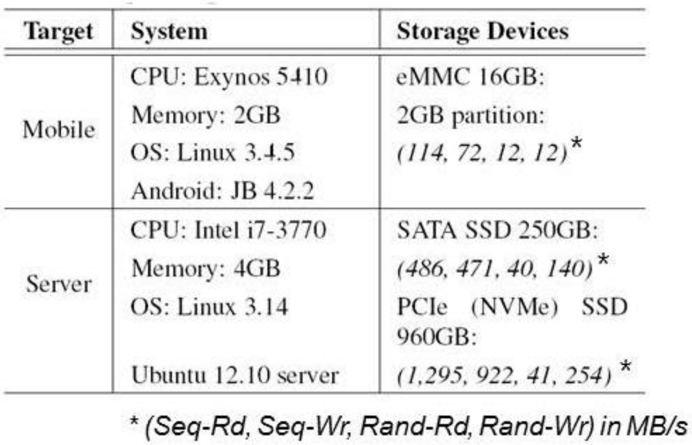
If file2 is fsynced after checkpoint, it is a candidate of roll-forward recovery during POR.

Contents

- Introduction
- Design and Implementation of F2FS
- Evaluation
  - Experimental Setup
  - Performance on the Mobile System
  - Performance on the Server System
  - Multi-head Logging Effect
  - Cleaning Cost
  - Adaptive Logging Performance
- Conclusion

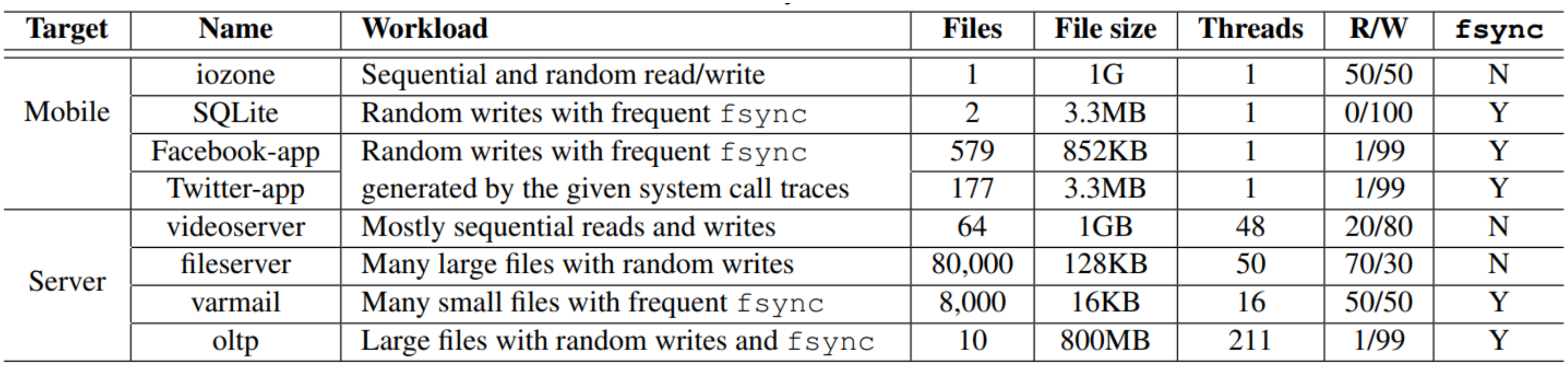
Evaluation

- Experimental Setup
  - *Mobile* and *server* systems
  - Performance comparison between ext4, btrfs, nilfs2 and f2fs



Target	System	Storage Devices
Mobile	CPU: Exynos 5410 Memory: 2GB OS: Linux 3.4.5 Android: JB 4.2.2	eMMC 16GB: 2GB partition: *(114, 72, 12, 12)**
Server	CPU: Intel i7-3770 Memory: 4GB OS: Linux 3.14 Ubuntu 12.10 server	SATA SSD 250GB: *(486, 471, 40, 140)** PCIe (NVMe) SSD 960GB: *(1,295, 922, 41, 254)**

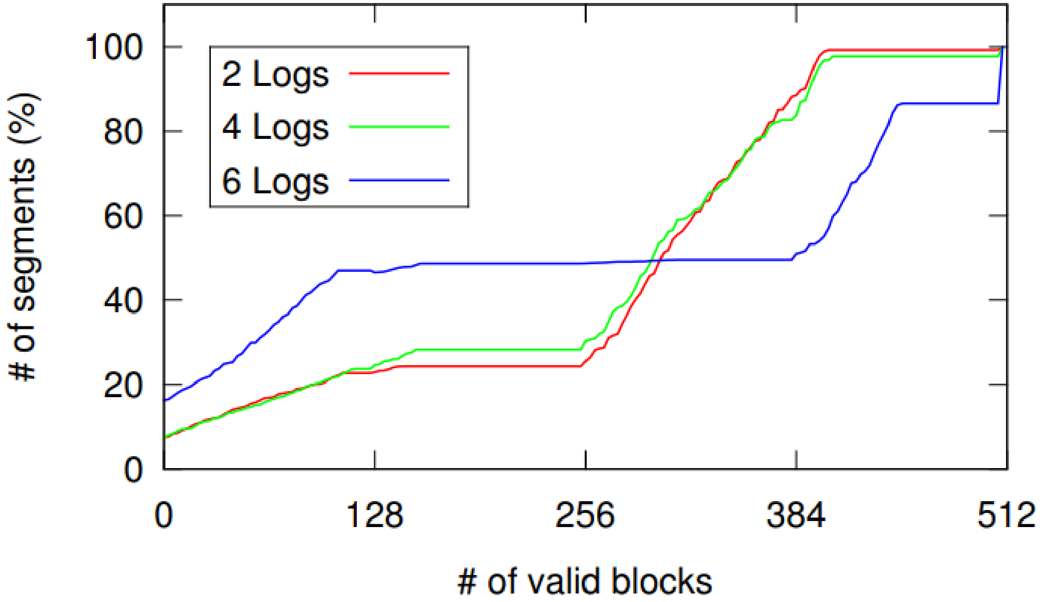
* *(Seq-Rd, Seq-Wr, Rand-Rd, Rand-Wr) in MB/s*



Target	Name	Workload	Files	File size	Threads	R/W	`fsync`
Mobile	iozone	Sequential and random read/write	1	1G	1	50/50	N
	SQLite	Random writes with frequent `fsync`	2	3.3MB	1	0/100	Y
	Facebook-app	Random writes with frequent `fsync` generated by the given system call traces	579	852KB	1	1/99	Y
	Twitter-app		177	3.3MB	1	1/99	Y
Server	videoserver	Mostly sequential reads and writes	64	1GB	48	20/80	N
	fileserver	Many large files with random writes	80,000	128KB	50	70/30	N
	varmail	Many small files with frequent `fsync`	8,000	16KB	16	50/50	Y
	oltp	Large files with random writes and `fsync`	10	800MB	211	1/99	Y

Multi-head Logging Effect

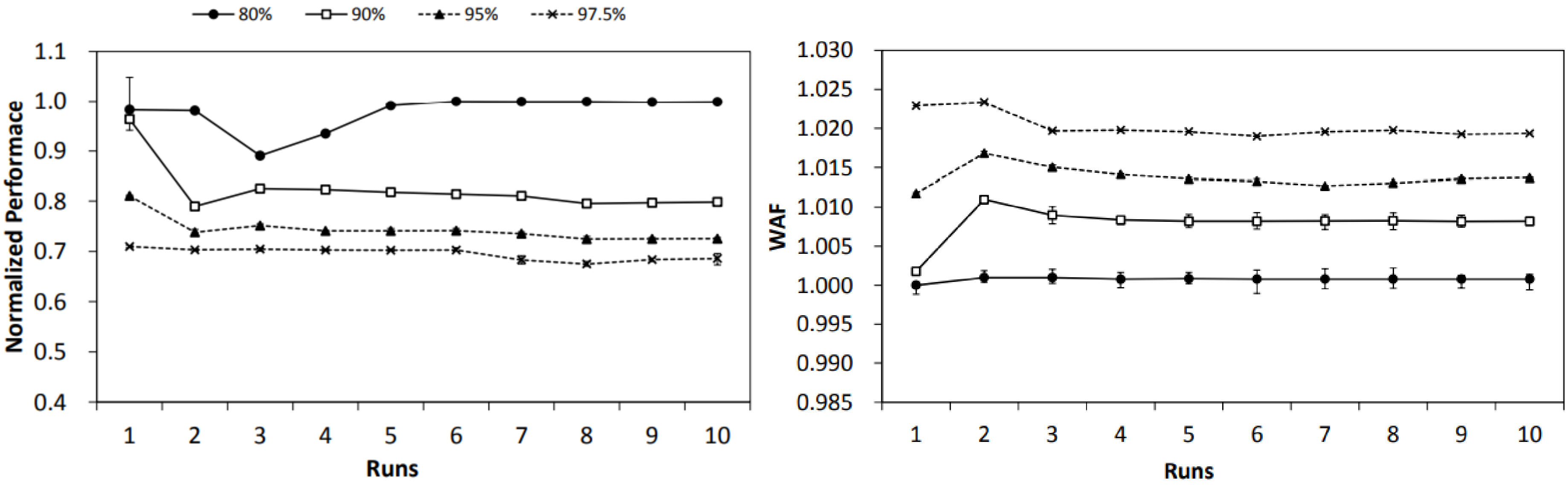
- Multi-head Logging
  - Using *more logs* gives better *hot* and *cold* data separation



Dirty segment distribution according to the number of valid blocks in segments.

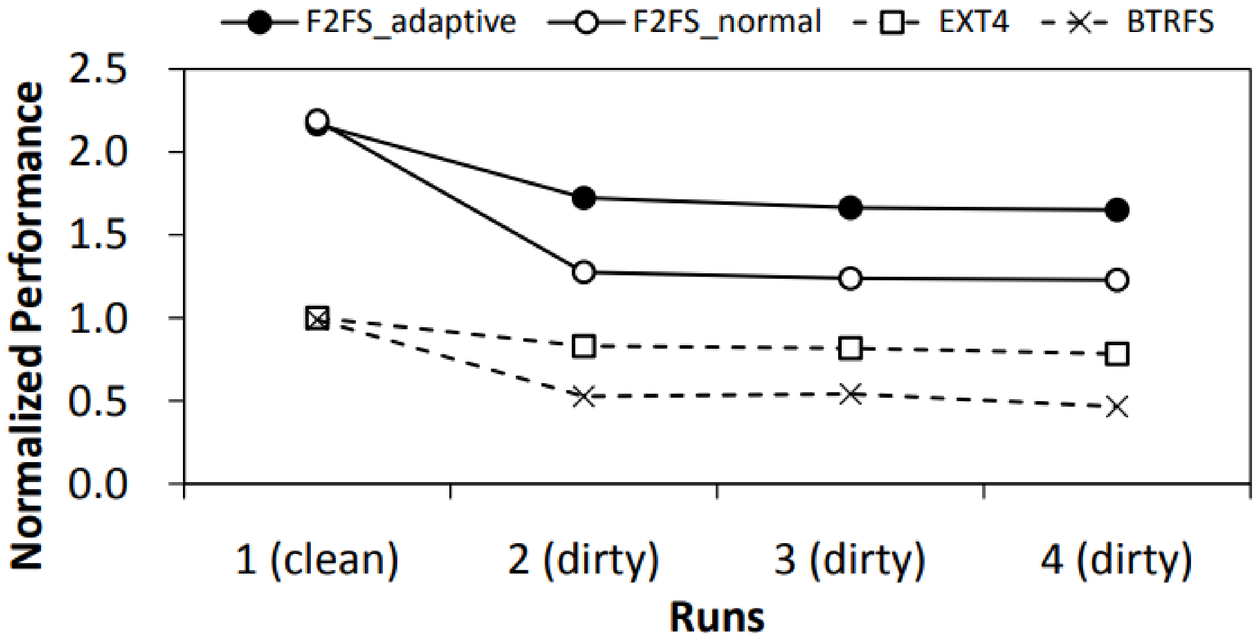
Cleaning Cost Analysis

- Under high utilization
  - Even in 97.5% utilization, WAF is less than 1.025
  - W/O adaptive logging, WAF goes up to more than 3



Adaptive Logging Performance

- Adaptive logging
  - Gives graceful performance degradation with highly aged volume



fileserver (random operations) on a device filled up 94%.

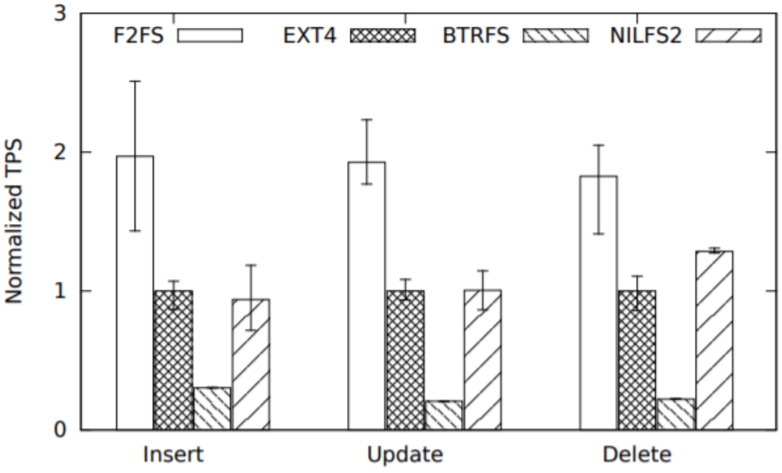
Evaluation Results

Mobile System

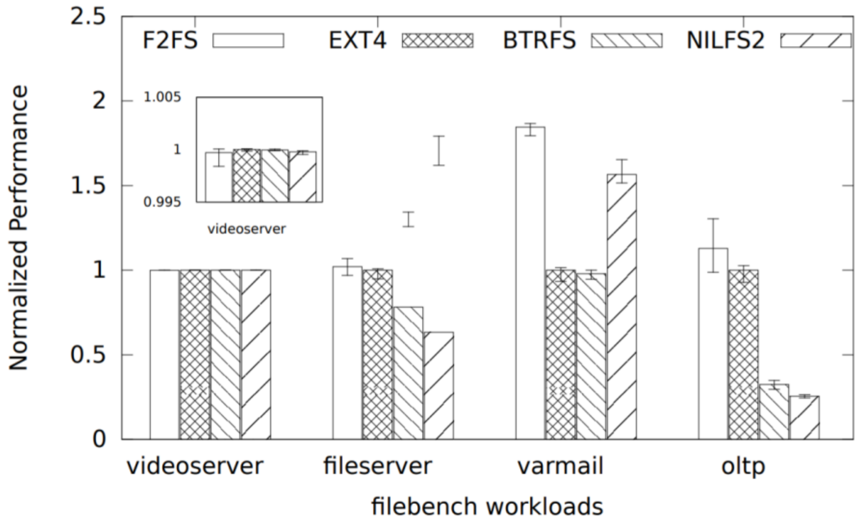
- Reduces write amount per fsync by using roll-forward recovery

Server System

- Varmail



SQLite
(1000 records, WAL mode)



PCIe SSD

Contents

- Introduction
- Design and Implementation of F2FS
- Evaluation
- Conclusion

Conclusion

- F2FS Features
  - Flash friendly on-disk **layout**
    - Align FS GC unit with FTL GC unit
  - Cost effective **index structure**
    - Restrain write propagation
  - **Multi-head** logging
    - Cleaning cost reduction
  - **Adaptive** logging
    - Graceful performance degradation in aged condition
  - **Roll-forward** recovery
    - fsync acceleration

- Performance gain over other file systems
  - About 3.1x speedup over Ext4