# F2FS: A New File System for Flash Storage

**Changman Lee, Dongho Shim, Joo-Young Hwang, Sang-yeun Cho**
**Samsung Electronics.**

# Contents


- **Introduction**
- **Design**
  - Flash Friendly On-disk Layout
  - Index Structure
  - Segment Allocation
  - Cleaning
  - Adaptive Logging
  - Recovery
- **Evaluation**
  - Experimental Setup
  - Multi-head Logging Effect
  - Adaptive Logging Performance (under aged condition)
  - Cleaning Cost Analysis
  - Mobile Benchmark
  - Server Benchmark
- **Conclusions**

# Introduction

- **Random writes is bad to flash storage device.**
  - Sustained random write performance degrades

- **Sequential write oriented file systems**
  - log-structured file systems, copy-on-write file systems

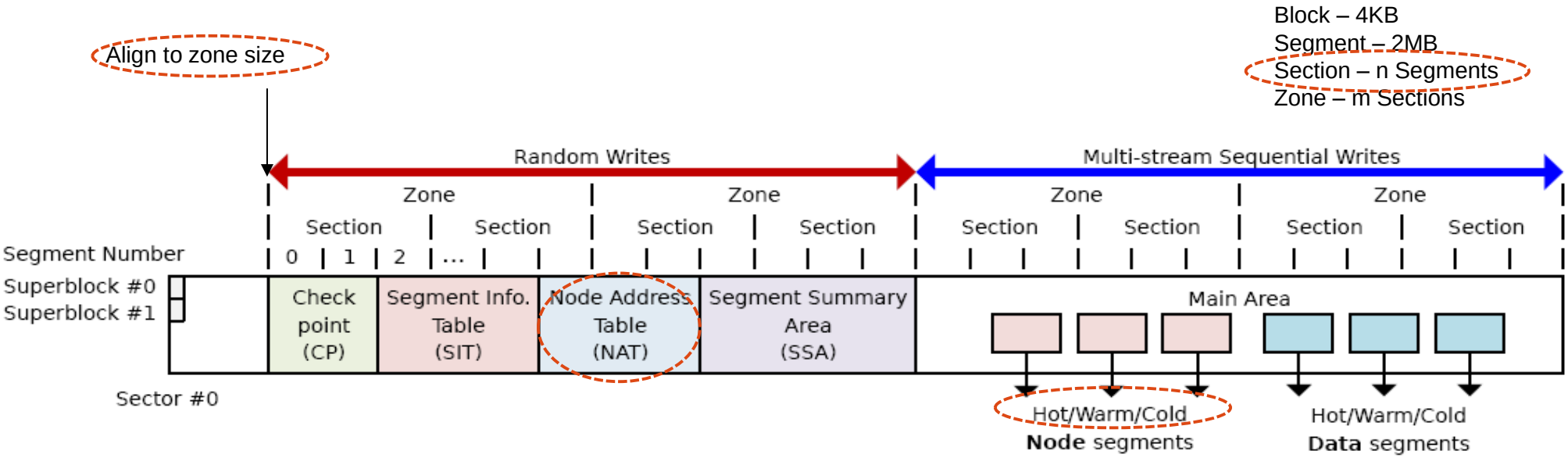
- **Our Contribution**
  - Design and implement a new file system to fully leverage and optimize the usage of NAND flash solutions (with block interface).
  - Performance comparison with Linux file systems (Ext4, Btrfs, Nilfs2).
    - Mobile system and server system

# Key Design Considerations

- Flash-friendly on-disk layout
- Cost-effective index structure
- Multi-head logging
- Adaptive logging
- fsync acceleration with roll-forward recovery

# Flash-friendly On-Disk Layout

- **Flash Awareness**
  - All the FS 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
- **Cleaning overhead**
  - Multiple logs for static hot/cold data separation



# LFS Index Structure

LFS

**Single head logging**
**Wandering tree issue**

Fixed location, but separated

S B

C P

Inode Map

Inode for directory

Directory data

Inode for regular file

File data

…

File data

Segment Usage

Segment Summary

Used for cleaning

Indirect Pointer block

Direct Pointer block

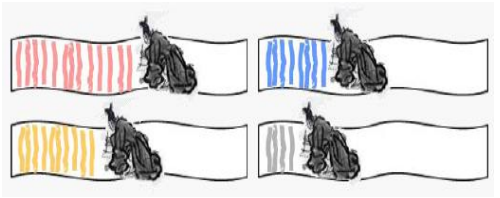
One big log

# F2FS Index Structure

*** NAT: Node Address Table**

F2FS

**Multi-head logging**
**Restrained write propagation**



Fixed location w/ locality

S B

C P

NAT*

Inode for directory

Directory data

Inode for regular file

File data

...

File data

Referenced via NAT lookup

Indirect Node

Referenced via NAT lookup

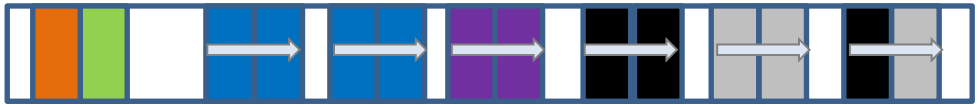
Direct Node

Segment Info. Table (SIT)

Segment Summary (SSA)

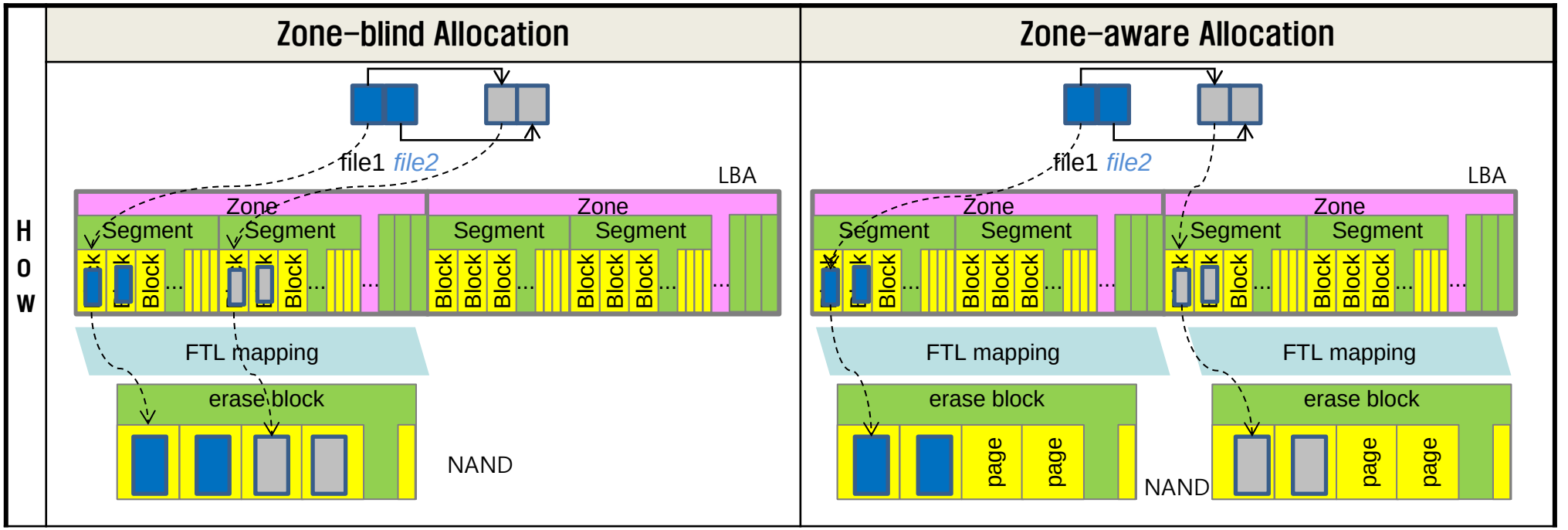
Used for cleaning

Multiple logs



# Segment Allocation for Multi-head Logging

- **To physically separate multi-head logs in NAND flash, each logs are allocated across the separated zones.**
  - This strategy works well with set-associative mapping FTL mapping.



# Cleaning

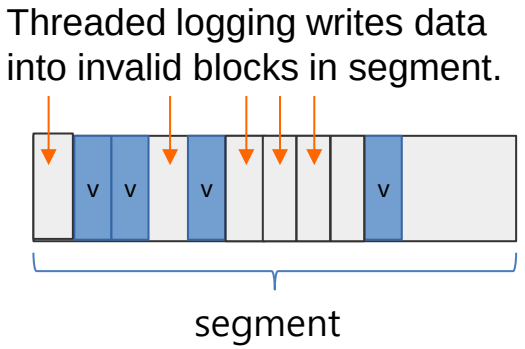
- **Cleaning is done in section unit.**
  - Section to be aligned with FTL's GC unit.

- **Cleaning procedure**
  - Victim selection: get a victim section through referencing Segment Info. Table (SIT).
  - Valid block check: Load parent index structures of there-in data identified from Segment Summary Area (SSA).
  - Migration: move valid blocks by checking their cross-reference
  - The victim section is marked as "pre-free".
    - It will become free after the next checkpoint is made.

- **Victim selection policies**
  - Greedy algorithm for foreground cleaning job
  - Cost-benefit algorithm for background cleaning job

# Adaptive Logging

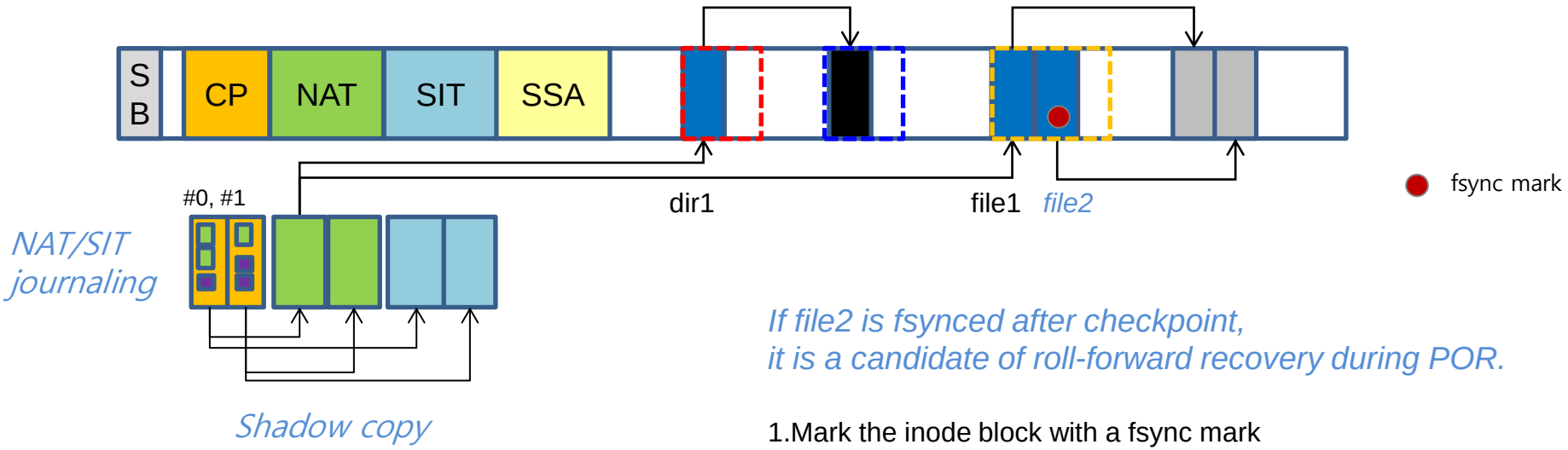
- **To reduce cleaning cost, F2FS changes write policy dynamically.**
  - Write policies: LFS mode, threaded logging
  - LFS mode (logging to clean segment)
    - Need cleaning operations if there is no free segment.
    - Cleaning causes mostly random read and sequential writes.
  - Threaded logging (logging to dirty segment)
    - Reuse obsolete blocks in a dirty segment
    - No need to run cleaning
    - Cause random writes




* Node is always logged in LFS mode.

# Power Off Recovery

- **Shadow copy on Checkpoint, NAT, SIT blocks**
- **NAT/SIT entry journal in checkpoint block**
- **Roll-forward recovery to recover fsync'ed data**




When SPO is occurred after checkpointing #0
1. Roll-back to last checkpoint area and recover dir1, file1
2. Recover fsynced file2 by roll-forward recovery routine

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

1.Mark the inode block with a fsync mark
2.During roll-forward, search marked direct node blocks
3.Search the change of marked node block and its original one checkpointed before
4.Replay data write; update NAT, SIT accordingly
5.Create checkpoint

# Evaluation

- **Experimental Setup**
  - Mobile and server systems
  - 8 benchmarks used
  - Performance comparison with ext4, btrfs, nilfs2

*(Seq-R, Seq-W, Rand-R, Rand-W)*

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

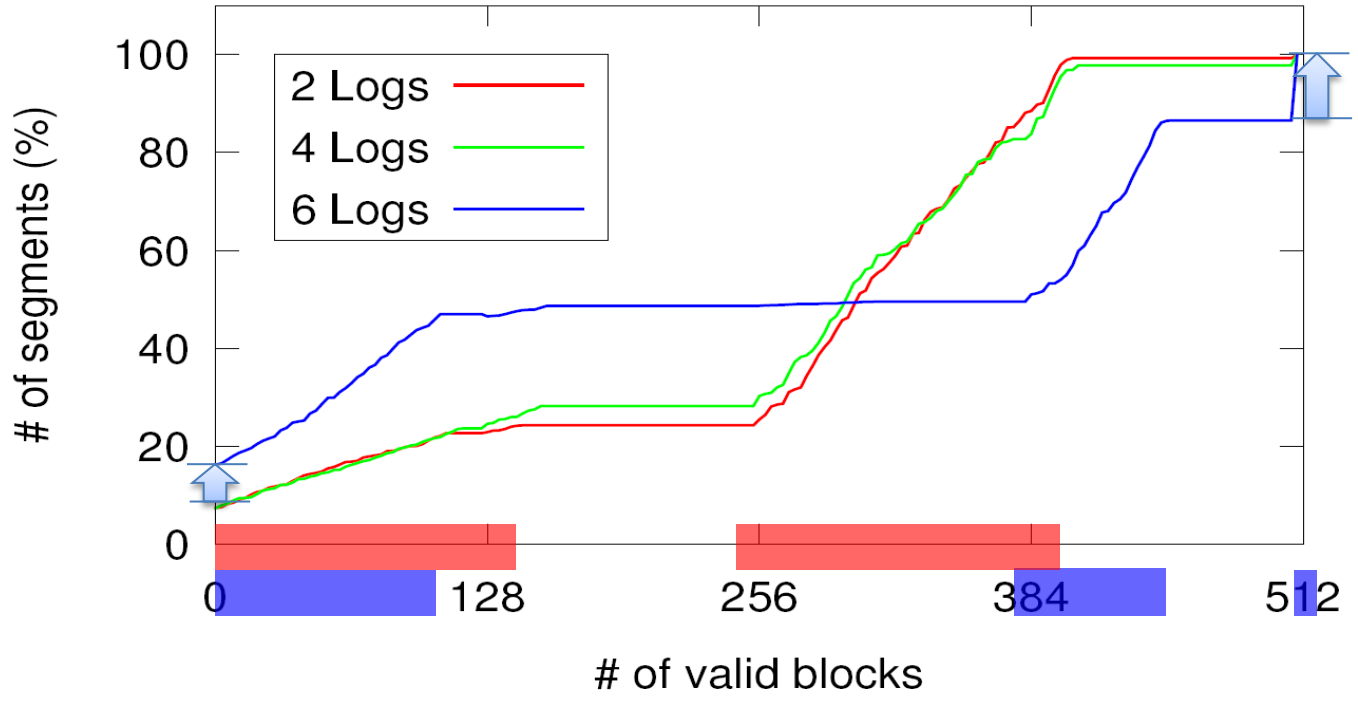
| Target | Name | Workload | Files | File size | Threads | R/W | `fsync` |
|---|---|---|---|---|---|---|---|
| Mobile | iozone | Sequential and random read/write | 1 | 1G | 1 | 50/50 | N |
| | SQLite | Random writes with frequent `fsync` | 2 | 3.3MB | 1 | 0/100 | Y |
| | Facebook-app | Random writes with frequent `fsync` | 579 | 852KB | 1 | 1/99 | Y |
| | Twitter-app | generated by the given system call traces | 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

- **Using more logs gives better hot and cold data separation.**

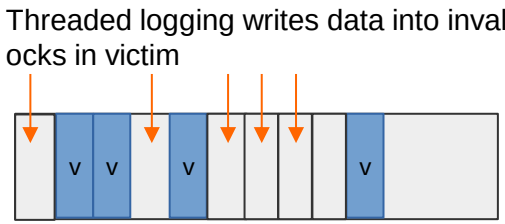
Test condition:
Run two workloads simultaneously:
1. Varmail (10,000 files in 100 dirs, total writes 6.5 GB)
2. Copy jpg files (500KB, 5,000 files, 2.5GB) ➔ classified by F2FS as cold

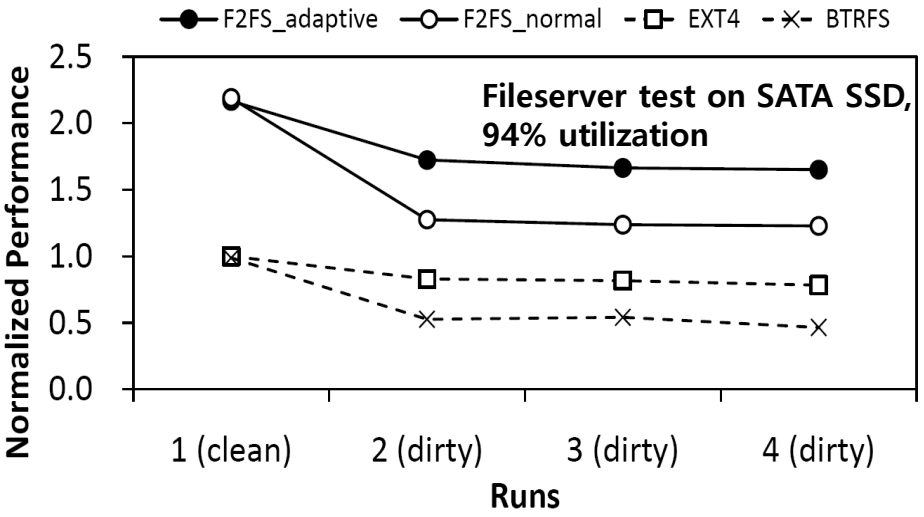
| Type | Temp. | Objects |
|---|---|---|
| Node | Hot | Direct node blocks for directories |
| | Warm | Direct node blocks for regular files |
| | Cold | Indirect node blocks |
| Data | Hot | Directory entry blocks |
| | Warm | Data blocks made by users |
| | Cold | Data blocks moved by cleaning; Cold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

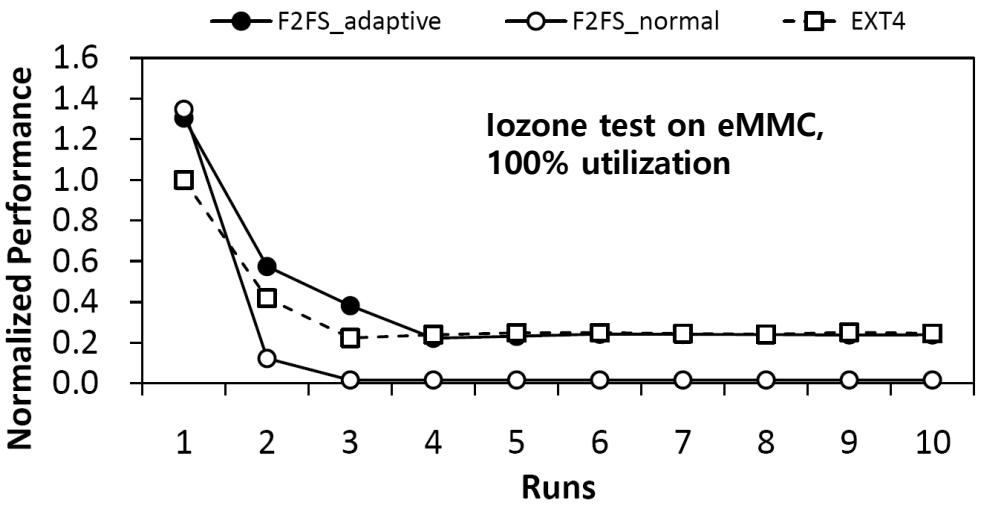


# Adaptive Logging

- **Adaptive logging gives graceful performance degradation under highly aged volume conditions.**
  - Fileserver test on SATA SSD (94% util.)
    - Sustained performance improvement: 2x/3x over ext4/btrfs.
  - Iozone test on eMMC (100% util.)
    - Sustained performance is similar to ext4.



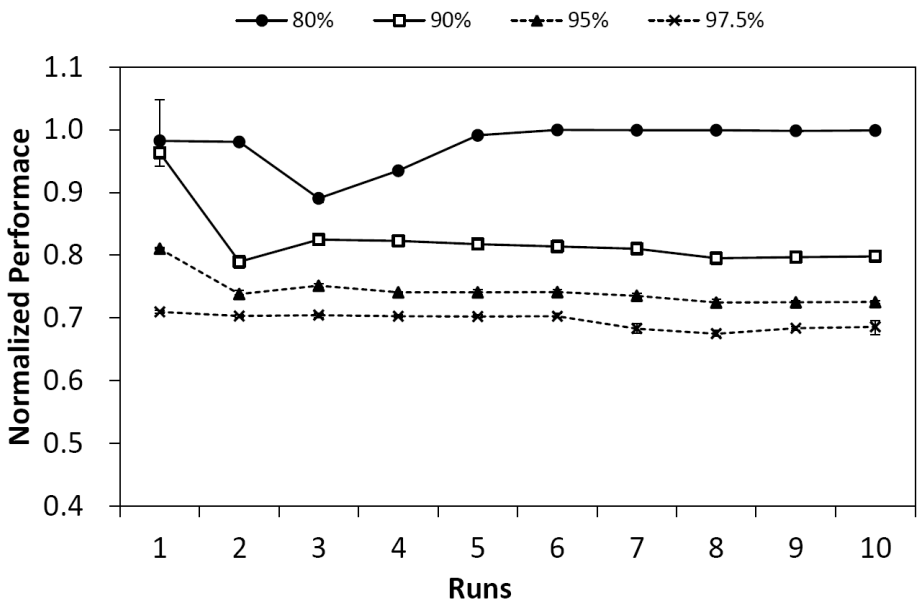



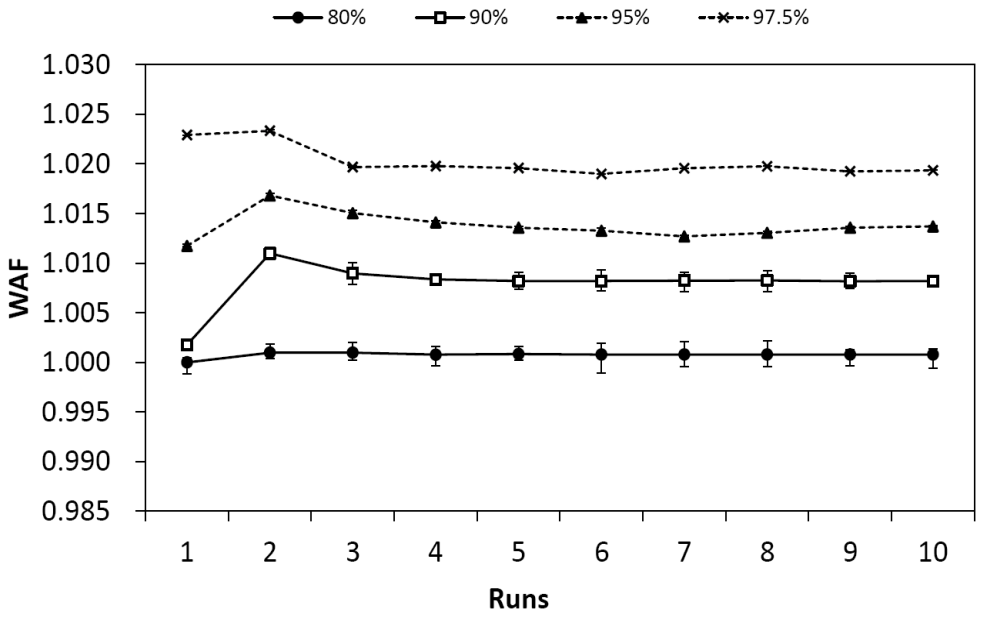



# Cleaning Cost Analysis

- **Under high utilization, F2FS uses adaptive logging.**
  - Only node segment cleaning is done.
  - Even in 97.5% util., WAF is less than 1.025.
- **Without adaptive logging, WAF goes up to more than 3.**

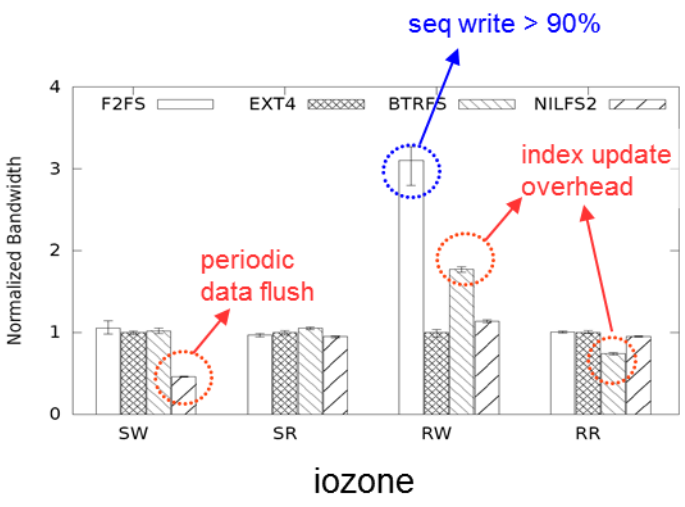
Test condition:
120GB of 250GB (SATA SSD)
Util (Cold : Hot) = 80%(60:20), 90%(60:30), 95%(60: 35), 97.5%(60:37.5)
Workload : 20GB 4KB random writes, 10 iterations



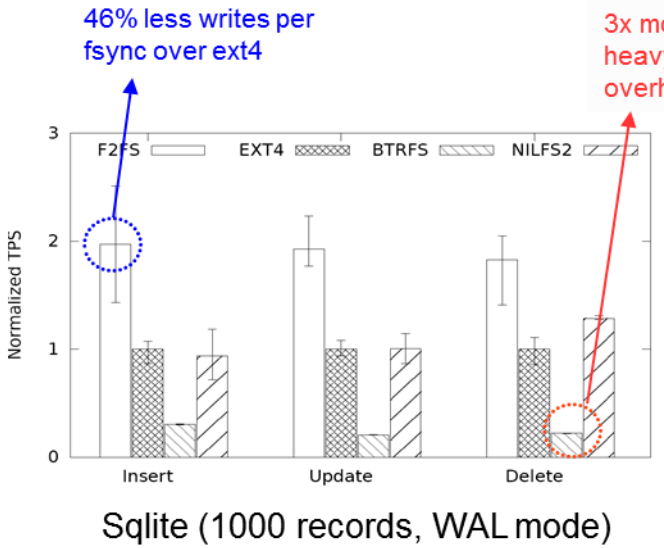



# Mobile Benchmark

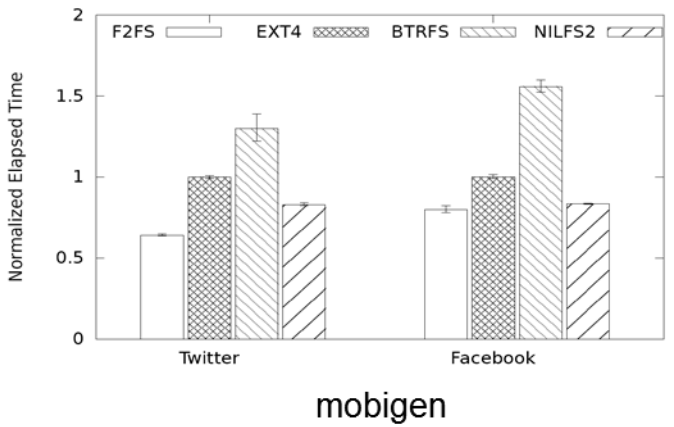
- **In F2FS, more than 90% of writes are sequential.**
- **F2FS reduces write amount per fsync by using roll-forward recovery.**
  - If checkpoint is done per fsync, write amount in SQLite insert test is 37% more than Ext4, and normalized performance is 0.88.
- **Btrfs and nilfs2 performed poor than ext4.**
  - Btrfs: heavy indexing overheads, Nilfs2: periodic data flush



iozone



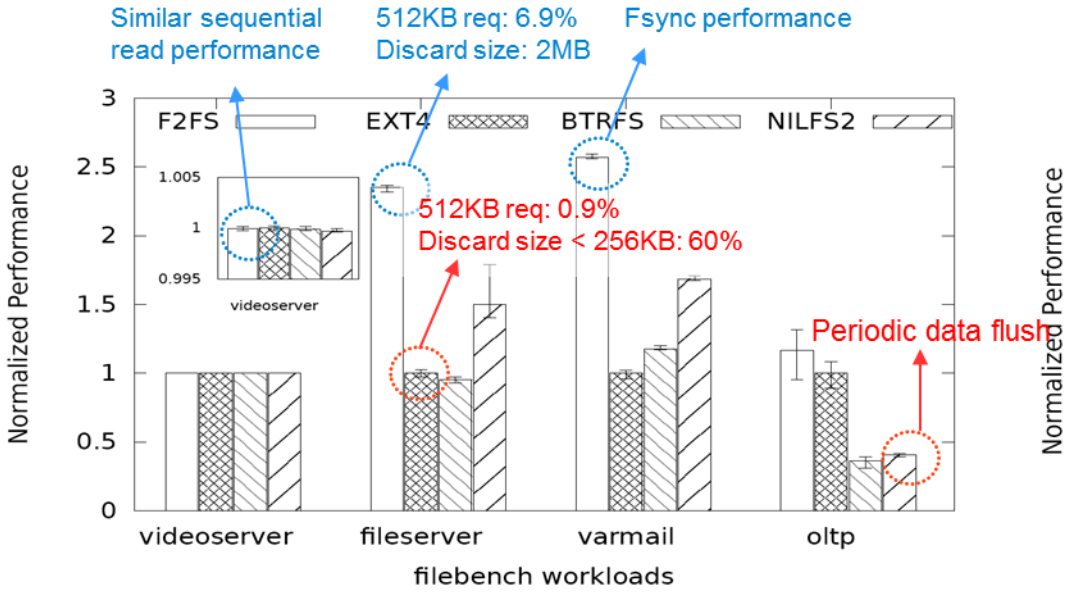
Sqlite (1000 records, WAL mode)

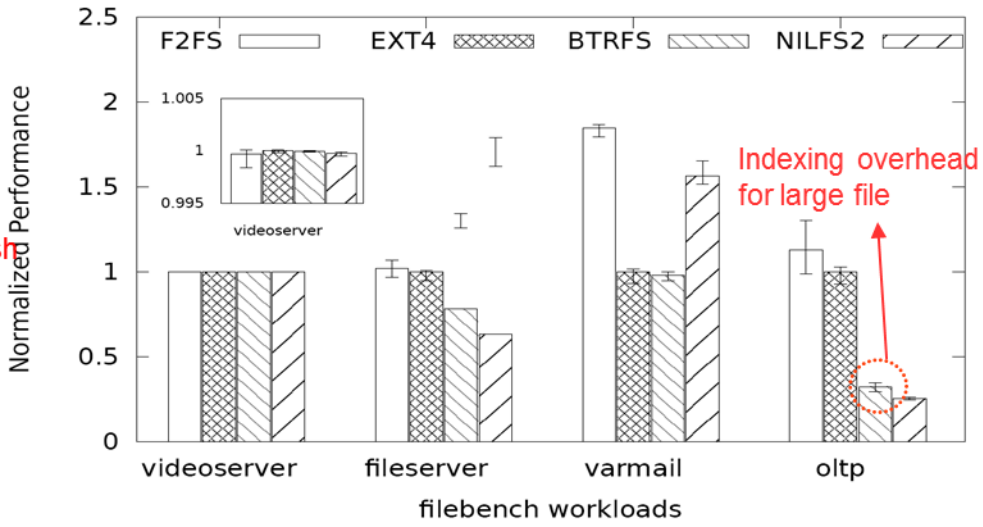


mobigen

# Server Benchmark

- **Fileserver: discard size matters in SATA SSD due to interface overhead.**
- **For a PCIe device that has high random write throughput, reducing write amount is more important than using sequential write pattern.**



# Conclusions

- **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.

- **F2FS shows performance gain over other Linux file systems.**
  - 3.1x (iozone) and 2x (SQLite) speedup over Ext4
  - 2.5x (SATA SSD) and 1.8x (PCIe SSD) speedup over Ext4

- **F2FS is publicly available, included in Linux mainline kernel since Linux 3.8.**

## How to tune F2FS

- ***Through 'mkfs.f2fs'***
  - *heap style allocation*
  - *over provision ratio*
  - *# of segments per section*
  - *# of sections per zone*
  - *file extention*
  - *discard*
- ***Through 'mount option'***
  - *background_gc=on/off*
  - *disable_roll_forward*
  - *discard*
  - *no_heap*
  - *nouser_xattr*
  - *noacl*
  - *active_logs=2/4/6*
  - *diable_ext_identify*
  - *Inline xattr/data/dentry*
  - *flush_merge*
  - *nobarrier*
  - *fastboot*
  - *extent cache*
- ***Through 'sysfs'***
  - *idle_time*
  - *reclaim_segments*
  - *max_small_discards*
  - *ipu_policy*
  - *min_fsync_blocks*
  - *max_victim_search*
  - *dir_level*
  - *ram_thresh*
- ***Through 'ioctl'***
  - *Atomic write*
  - *FITRIM*
  - *Shutdown*

## Deployments

- **Motorola Droid family (2013)**
- **Moto X (2013/2014)**
- **Moto G family (2013/2014)**
- **Google Nexus 9 (2014)**

# Thank You!