F2FS: A New File System Designed for Flash Storage in Mobile

Presented at ELC Europe 2012

Nov 05, 2012

Joo-Young Hwang
S/W Development Team
Memory Business, Samsung Electronics Co., Ltd.

SAMSUNG

TURN ON TOMORROW

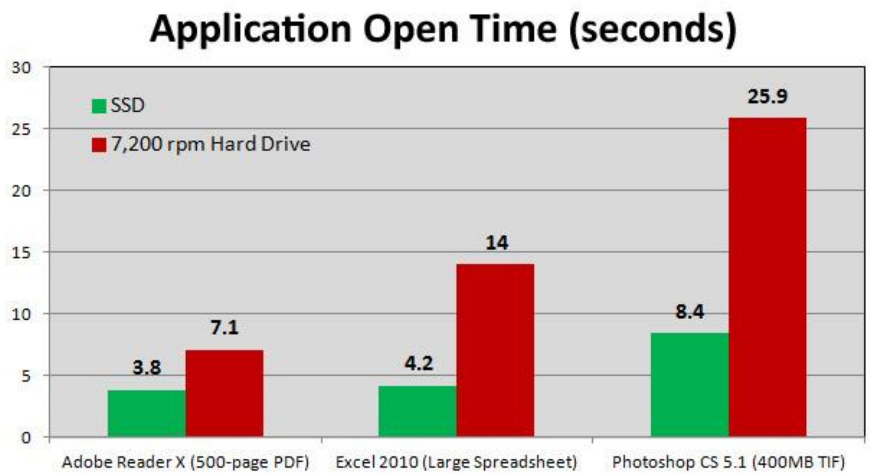
Agenda

- Introduction
- Design Overview
- Performance Evaluation Results
- Summary

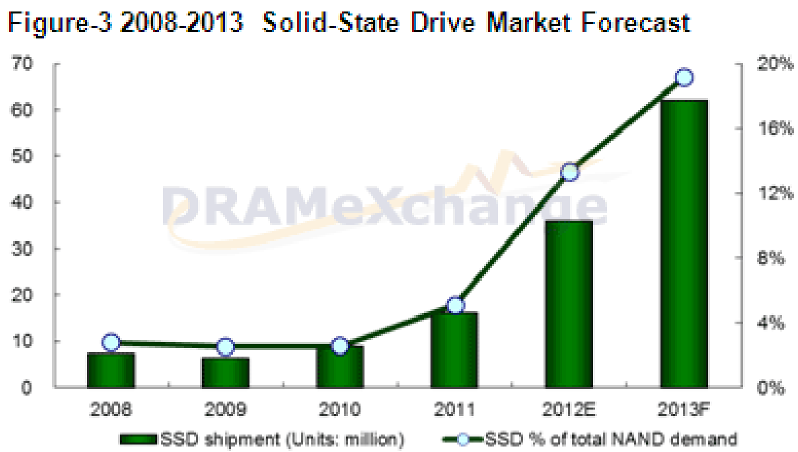
Introduction

- NAND Flash-based Storage Devices
  - SSD for PC and server systems
  - eMMC for mobile systems
  - SD card for consumer electronics

- The Rise of SSDs
  - Much faster than HDDs
  - Low power consumption



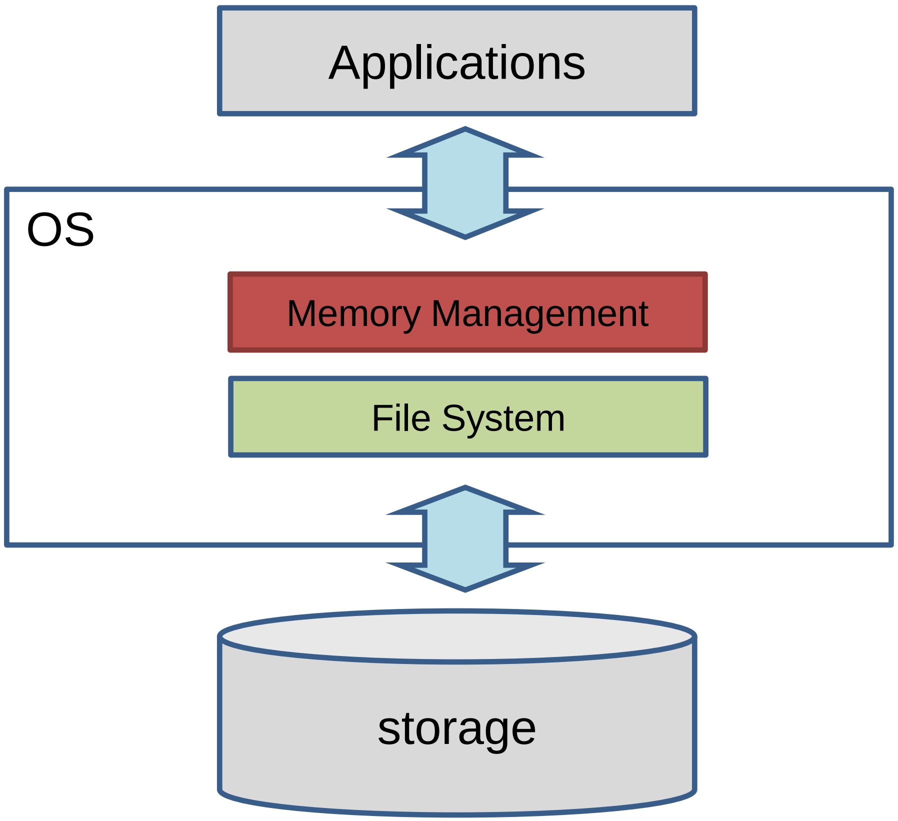
Source: March 30th, 2012 by Avram Piltch, LAPTOP Online Editorial Director

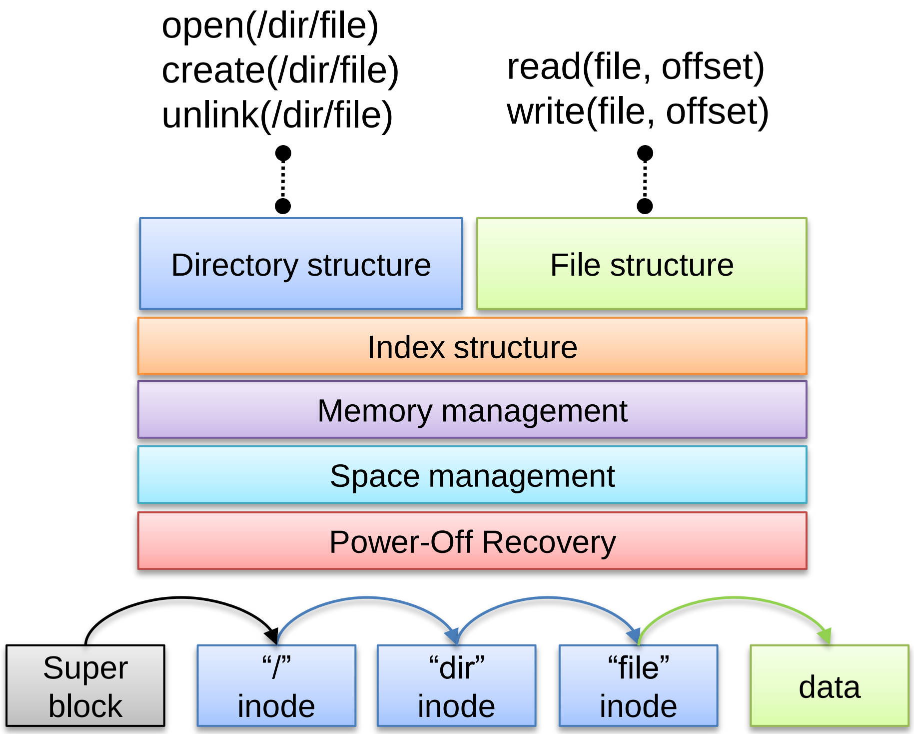


Source: DRAMeXchange, Jan., 2012

Introduction (cont'd)

- File System
  - Serve directory and file operations to users
  - Manage the whole storage space



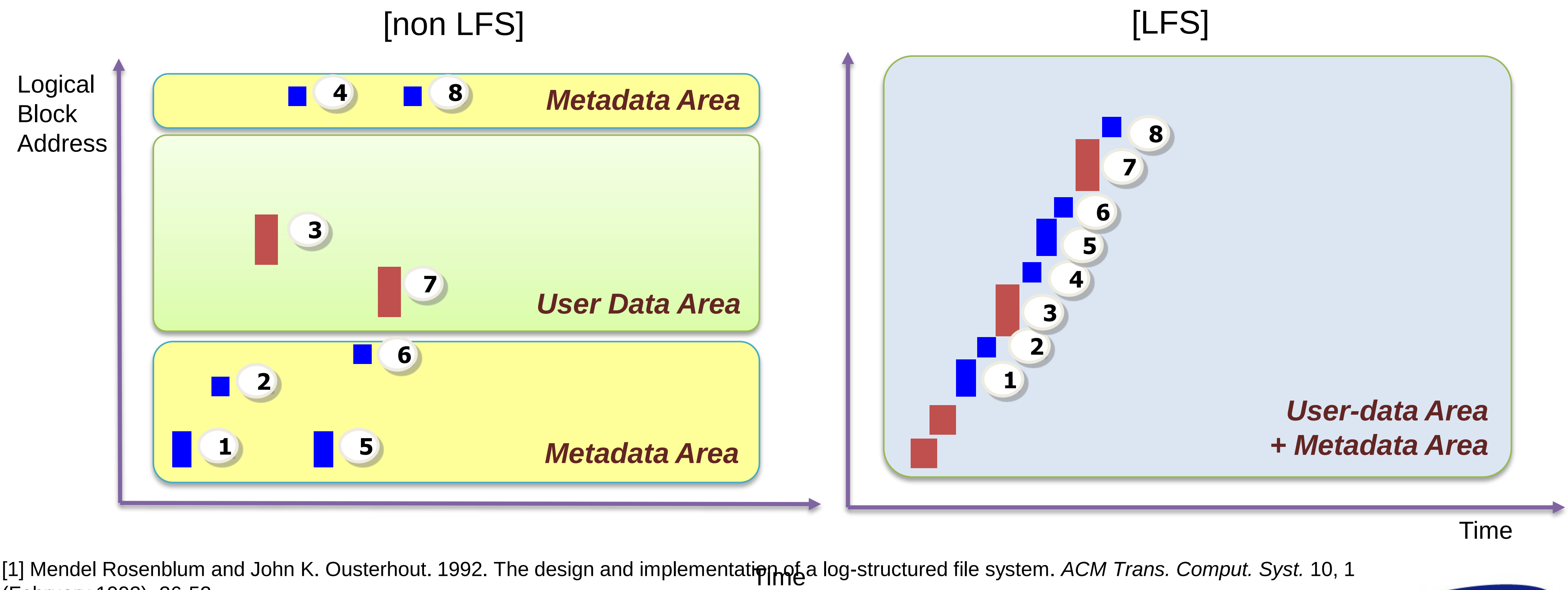


Introduction (cont'd)

- NAND Flash Memory
  - Erase-before-write
  - Sequential writes inside the erase unit
  - Limited program/erase (P/E) cycle
- Flash Translation Layer (FTL)
  - Conventional block device interface: no concern about **erase-before-write**
  - Garbage collection
  - Wear-leveling
  - Bad block management
- Issues in cheap FTL devices
  - Random write performance
  - Life span and reliability
- Conventional file systems for FTL devices?
  - Optimization for HDD performance characteristics may not be good for FTL.
  - No consideration for FTL device characteristics

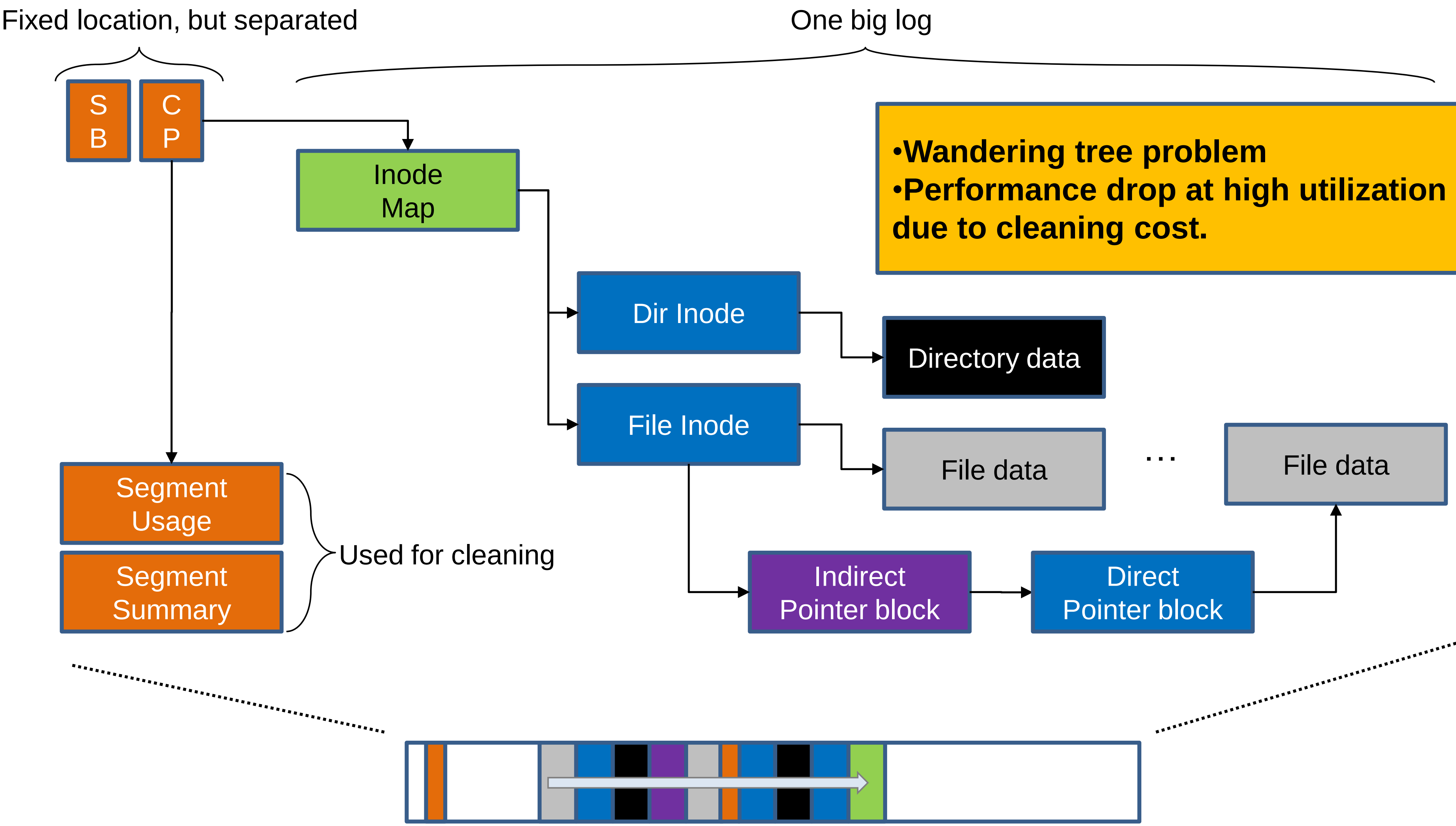
LFS Approach

- Sequential write is preferred by FTL devices.
- Log-structured File System (LFS)[1] fits well to FTL devices.
  - Assume the whole disk space as a big log, write data and metadata sequentially
  - Copy-on-write: recovery support is made easy.



[1] Mendel Rosenblum and John K. Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.* 10, 1 (February 1992), 26-52.

Conventional Log-structured File System (Index Structure)



- Wandering tree problem
- Performance drop at high utilization due to cleaning cost.

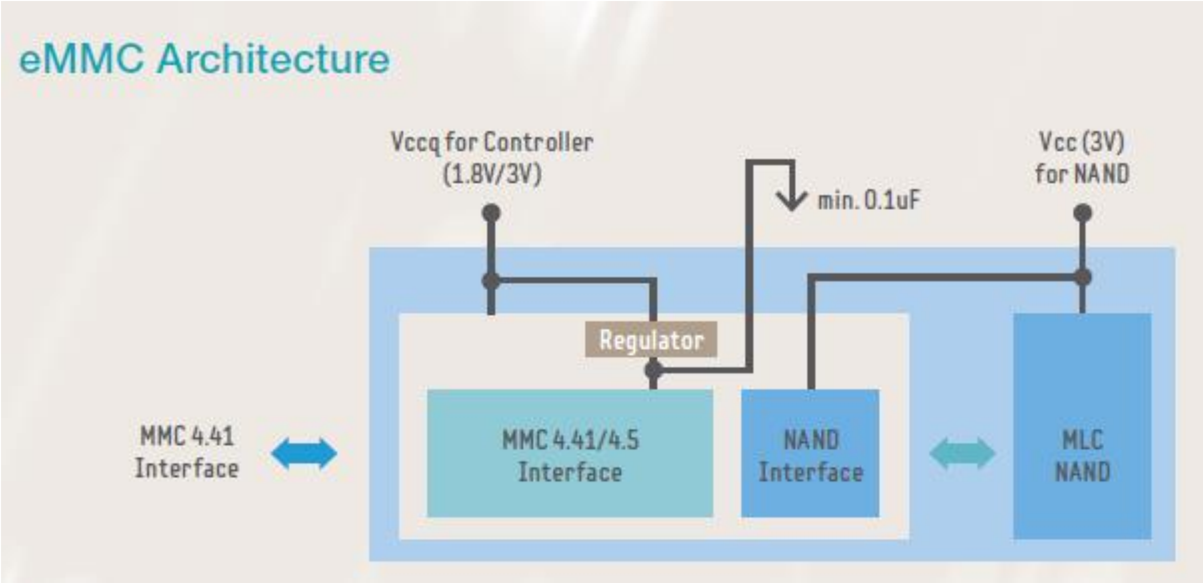
F2FS Design Overview

- Alignment with FTL operation unit
  - Align FS data structures to the FTL operation units.
- Avoiding Metadata Update Propagation
  - Indirection for inode and pointer blocks
- Efficient Cleaning using Multi-head Logs and Hot/Cold Data Separation
  - Write-time data separation → more chances to get binomial distribution
  - Two different victim selection policies for foreground and background cleaning
  - Automatic background cleaning
- Adaptive Write Policy for High Utilization
  - Switches write policy to threaded logging at right time (logging to FTL overprovision space)
  - Graceful performance degradation at high utilization

FTL Device Characteristics

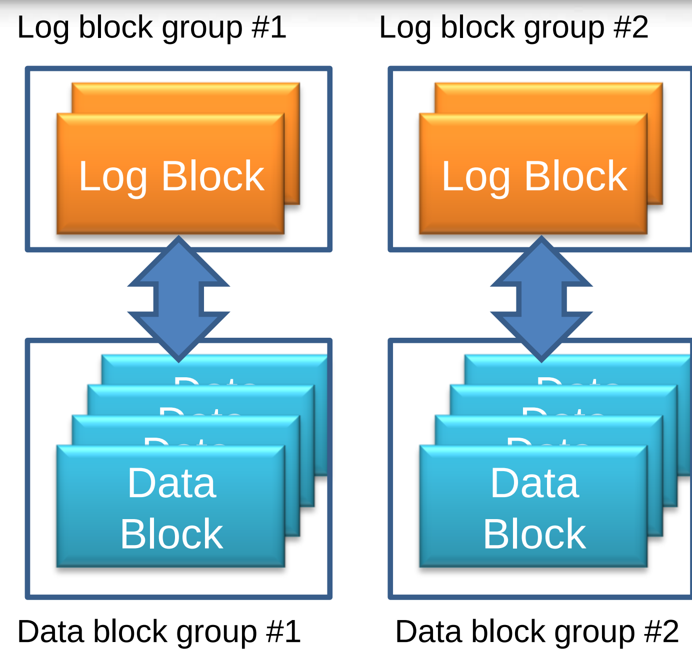
- FTL Functions
  - Address Mapping
  - Garbage Collection

- Address Mapping Methods
  - Block Mapping
  - Page Mapping
  - Hybrid Mapping



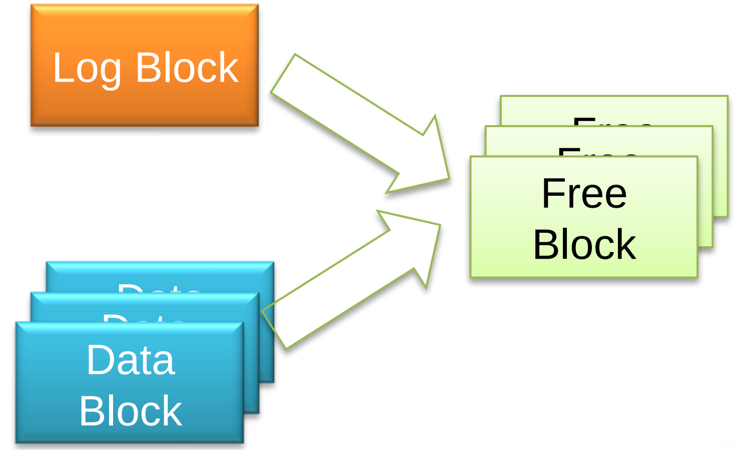
Hybrid Mapping

- Hybrid Mappings
  - BAST
  - FAST
  - **SAST (N: N+K mapping)**



[Example - 4:4+2 mapping]

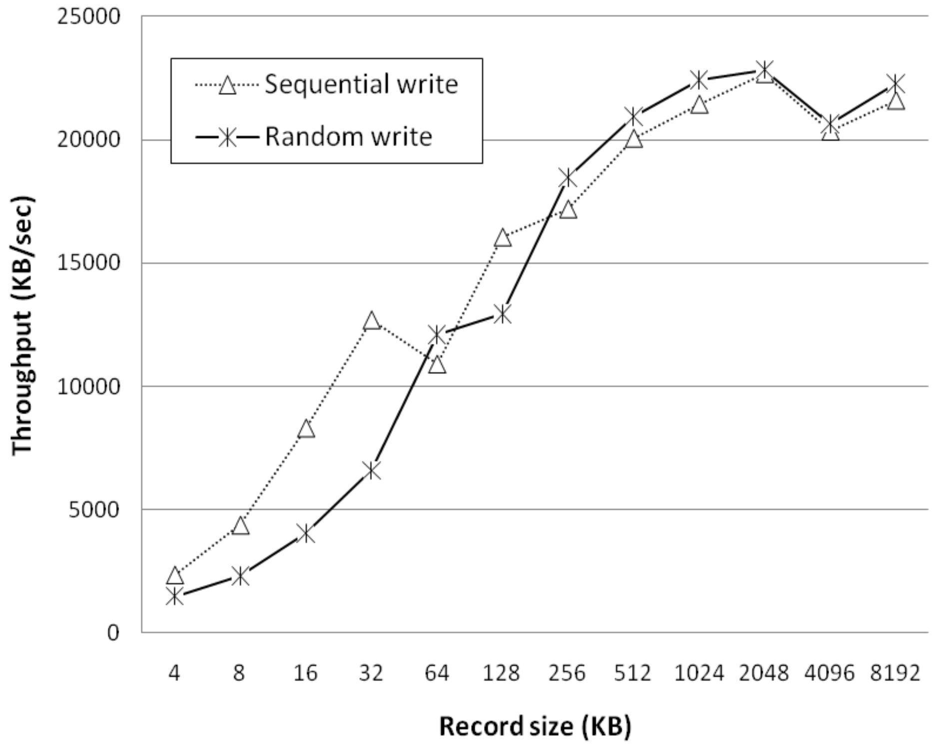
- Merge in Hybrid Mapping
  - Performed to get a new log block
  - Merge types
    - Full merge
    - Partial merge (aka copy merge)
    - **Switch merge**



FTL Device Characteristics (cont'd)

- FTL operation unit
  - Superblock – simultaneously erasable unit
  - Superpage - simultaneously programmable unit

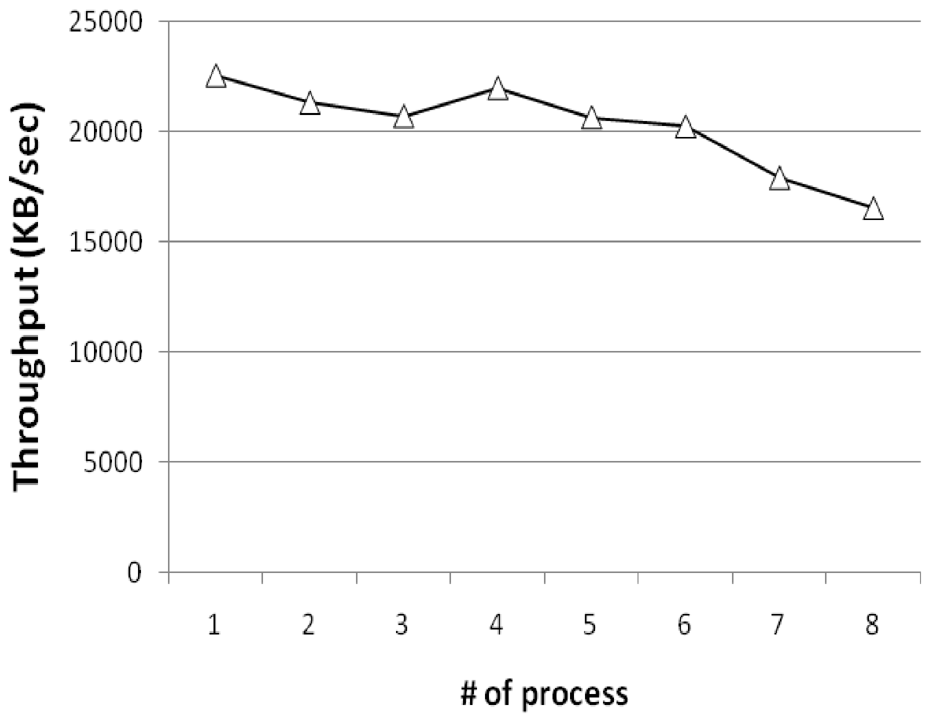
- Implications for segment size



FTL Device Characteristics (cont'd)

- FTL device may have multiple logging streams without performance degradation.
  - How many streams? How to identify?
  - Data block group geometry?

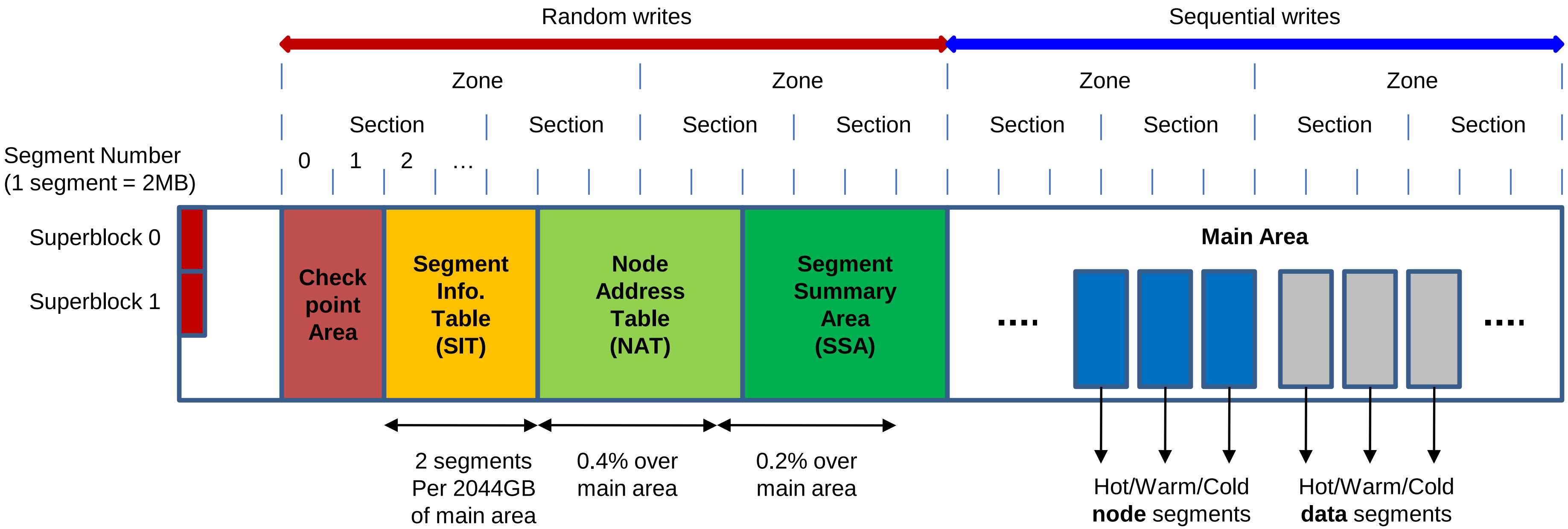
- Implications for multi-headed logging



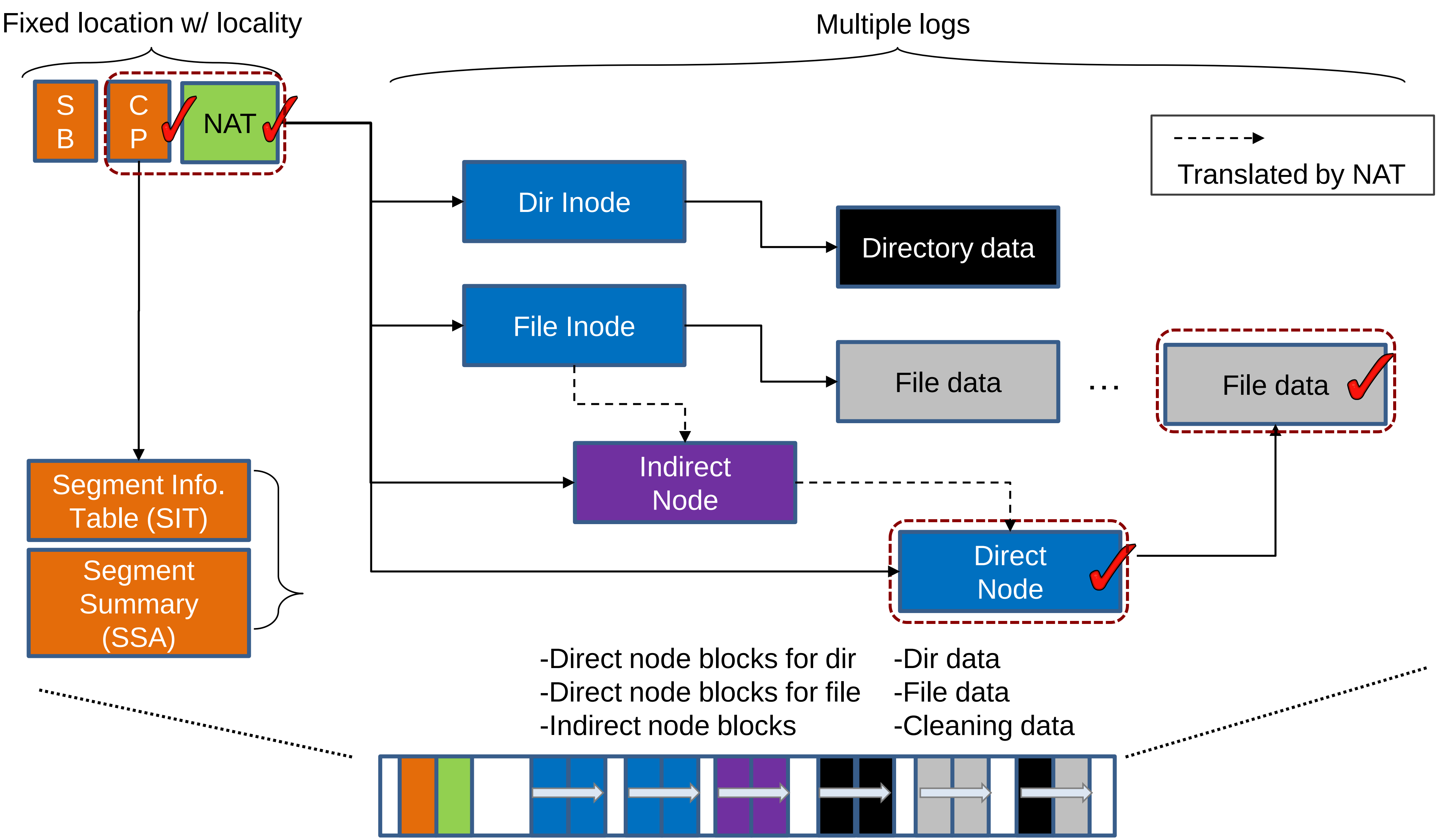
FTL Awareness

- 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*

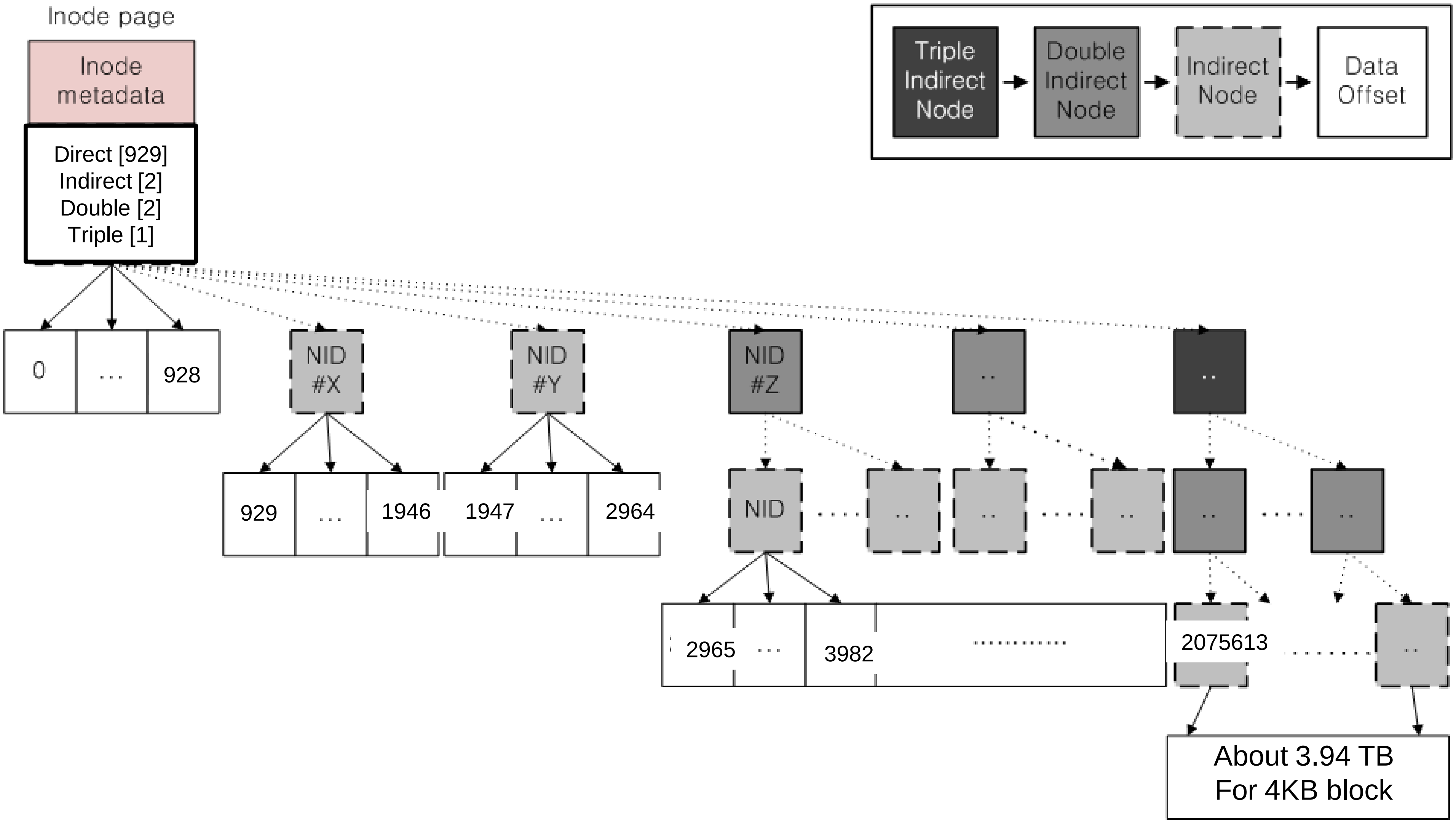
*zone: data block group
*section: FTL GC unit



Avoiding Metadata Update Propagation



File Indexing



Cleaning

- Cleaning Process
  - Reclaim obsolete data scattered across the whole storage for new empty log space
  - Get victim segments through referencing segment usage table
  - Load parent index structures of there-in data identified from segment summary blocks
  - Move valid data by checking their cross-reference
- Goal
  - Hide cleaning latencies to users
  - Reduce the amount of valid data to be moved
  - Move data quickly
- Issues
  - Hot and cold data separation
  - Victim selection policy

Cleaning (cont'd)

- Efficient hot/cold separation is possible by exploiting the FTL's multiple logs.
- Hot/cold separation at data writing time based on object types
  - Cf) hot/cold separation at cleaning time requires per-block update frequency information.

Type	Update frequency	Contained Objects
Node	Hot	Directory's inode block or direct node block
	Warm	Regular file's inode block or direct node block
	Cold	Indirect node block
Data	Hot	Directory's data block
	Warm	Updated data of regular files
	Cold	Appended data of regular files, moved data by cleaning, multimedia file's data

Cleaning (cont'd)

- Automatic Background Cleaning
  - Kicked in when I/O is idle.
  - Lazy write: cleaning daemon marks page dirty, then flusher issued I/O later.
  - Do not intervene foreground jobs.

- Victim Selection Policies
  - Greedy algorithm for foreground cleaning
  - Cost-benefit algorithm for background cleaning

Adaptive Write Policy

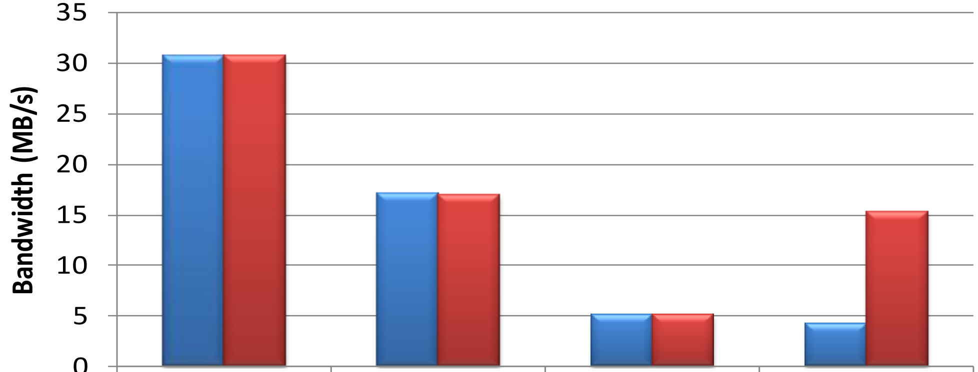
- Normal write policy is logging to a clean segment
  - Need cleaning operations if there is no clean segment.
  - Cleaning causes mostly random read and sequential writes.

- Change policy to 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*)

Performance (Panda board + eMMC)

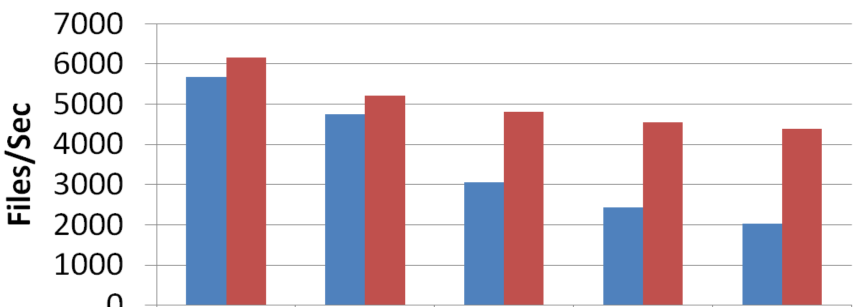
[System Specification]

CPU	ARM Cortex-A9 1.2GHz
DRAM	1GB
Storage	Samsung eMMC 64GB
Kernel	***Linux 3.3***
Partition Size	***12 GB***



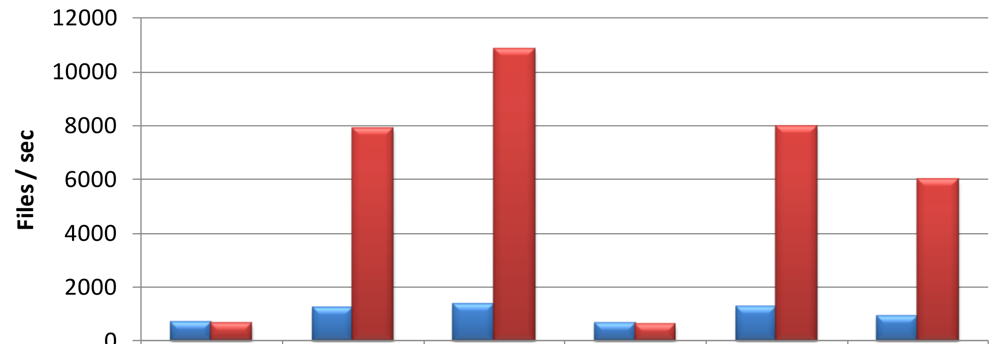
	seq. Read	seq. Write	rand. Read	rand. Write
EXT4	30.753	17.066	5.06	4.15
F2FS	30.71	16.906	5.073	15.204

[iozone]



	5000	10000	15000	20000	25000
ext4	5675.9	4741.6	3065	2439.5	2038.6
f2fs	6167.3	5207.9	4815.9	4544.4	4382.9

[fs_mark]



	seq.create	seq.stat	seq.delete	rand.create	rand.stat	rand.delete
EXT4	692	1238	1370	663	1250	915
F2FS	631	7871	10832	620	7962	5992

[bonnie++]

Performance on Galaxy Nexus

CPU	ARM Coretex-A9 1.2GHz
DRAM	1GB
Storage	Samsung eMMC (VFX) 16GB
Kernel	***3.0.8***
Android ver.	***Ice Cream Sandwich***

< Clean >

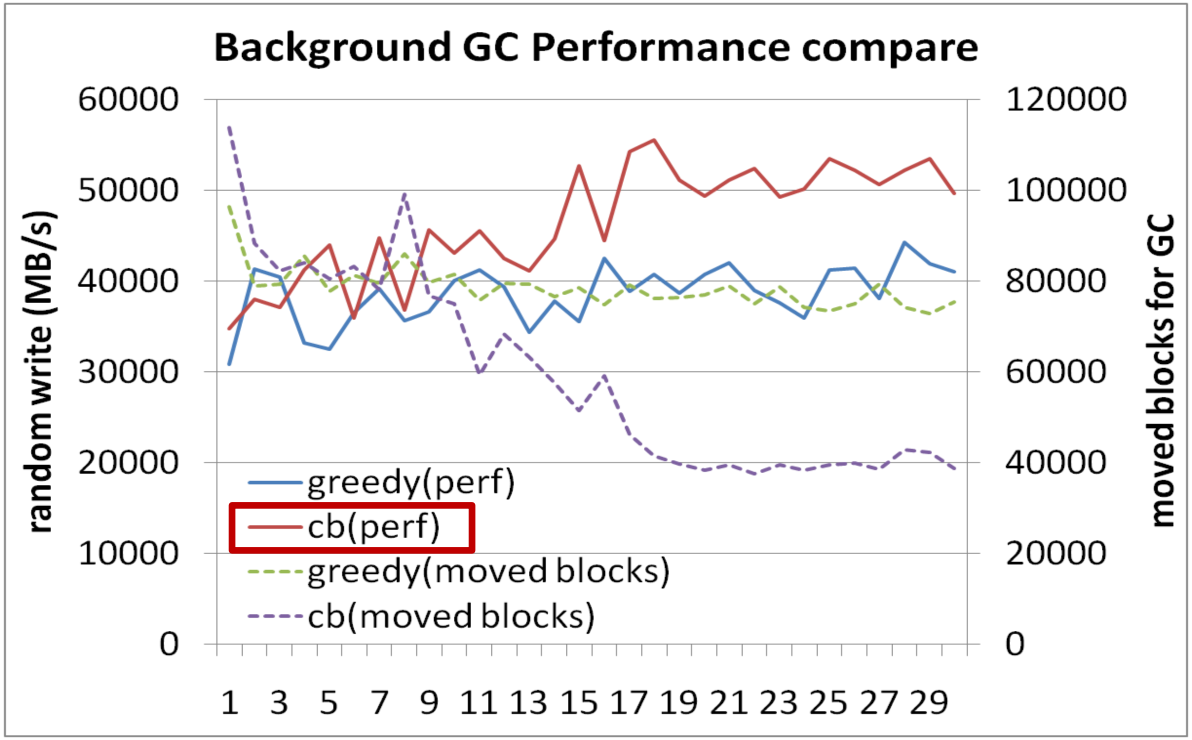
Items		Ext4	F2FS	Improv.
Contact sync time (seconds)		431	358	20%
App install time (seconds)		459	457	0%
RLBench (seconds)		92.6	78.9	17%
IOZoneWith AppInstall (MB/s)	Write	8.9	9.9	11%
	Read	18.1	18.4	2%

< Aged >

Items		Ext4	F2FS	Improv.
Contact sync time (seconds)		437	375	17%
App install time (seconds)		362	370	-2%
RLBench (seconds)		99.4	85.1	17%
IOZone With AppInstall (MB/s)	Write	7.3	7.8	7%
	Read	16.2	18.1	12%

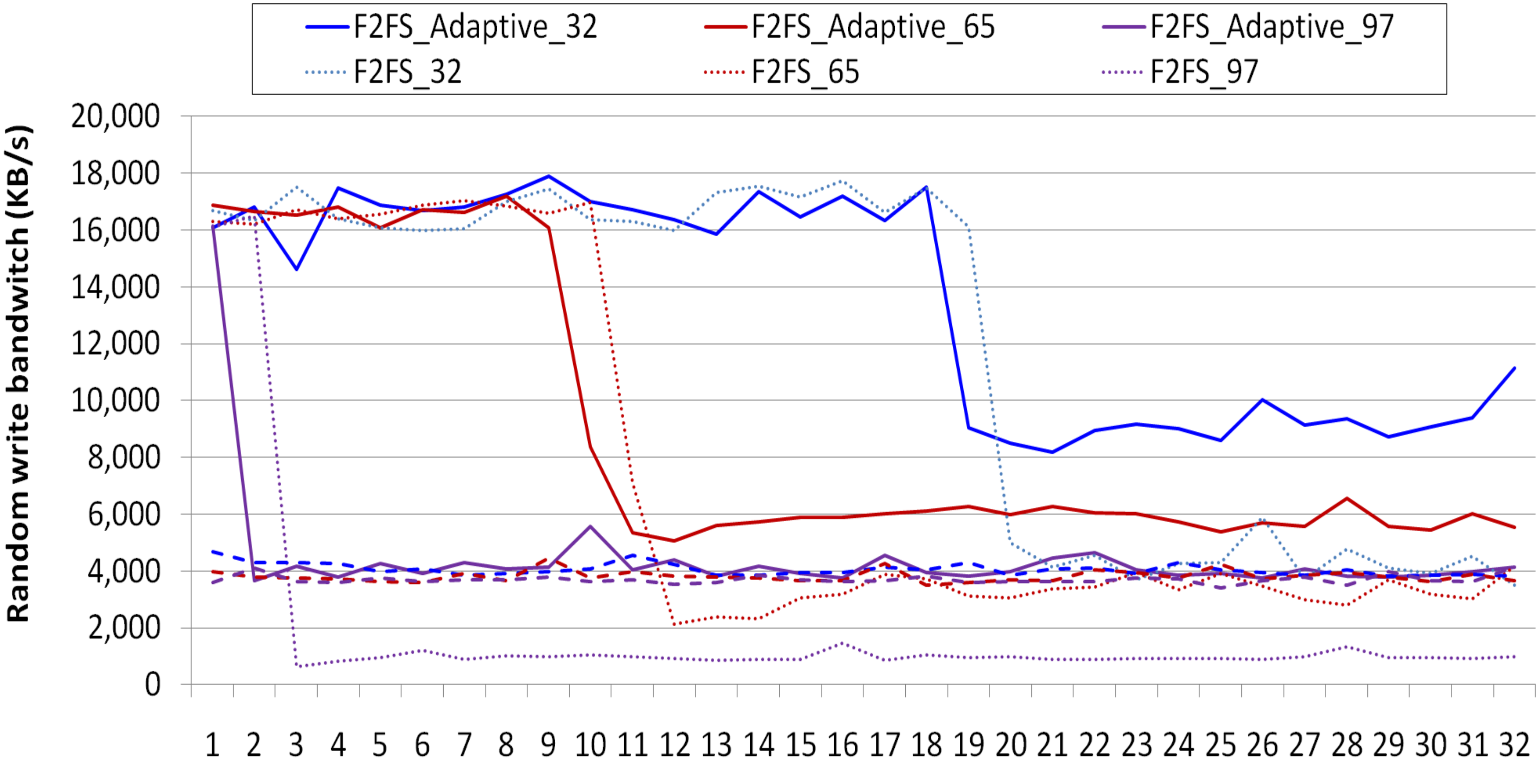
Evaluation: Cleaning Victim Selection Policies

- Setup
  - In x86, set 3.7 GB partition
  - Create three files having 1GB data
  - Write 256MB data randomly to the three files
  - Write 256MB data randomly to one of them, 30 times



Evaluation: Adaptive Write Policy

- Experimental setup
  - Embedded system with eMMC 12GB partition
  - Iozone random write tests on several 1GB files
- Results
  - Sustained performance is improved by adaptive write.
  - Ext4 shows about 4MB/s sustained performance.



Summary

- Flash-Friendly File System
  - File system for FTL block devices
  - Optimized for mobile flash solutions

- Performance evaluation on Android Phones
  - /data is F2FS volume
  - Factory reset & run android apps

- Current Status
  - Patch review in progress at LKML: v3 patch series are released.
  - Code and performance review on various devices are welcome.

Thank you!