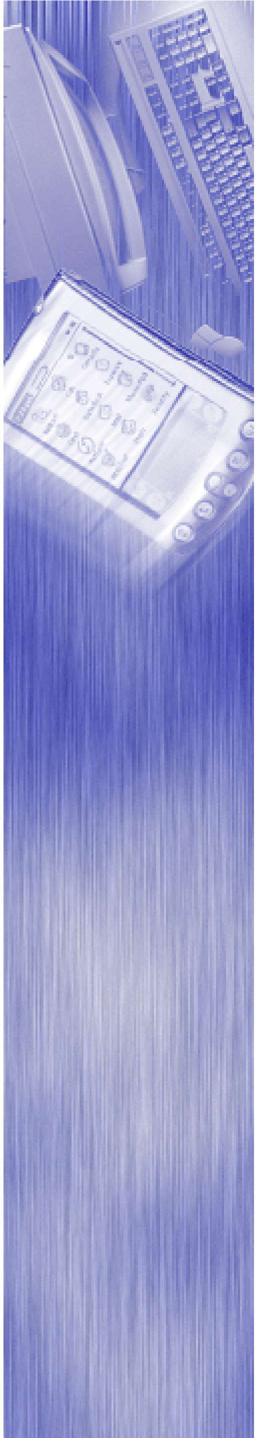


An Efficient Index Structure for NAND Flash Memory and Its Applications

한국과학기술원 전산학과

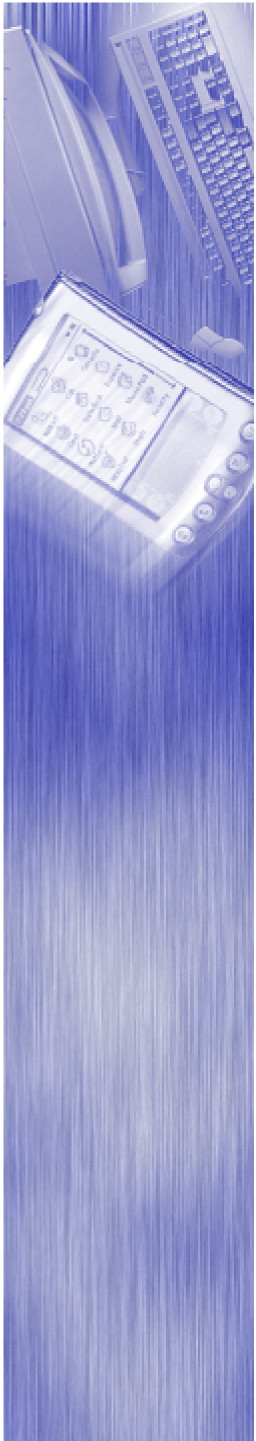
김 진 수

(jinsoo@cs.kaist.ac.kr)

KAIST

Outline

- **μ-Tree**
  - Motivation
  - Design
  - Analysis
  - Evaluations
- **Current Research based on μ-Tree**
  - μFS
  - μ-FTL



μ-Tree

KAIST

Index Structure

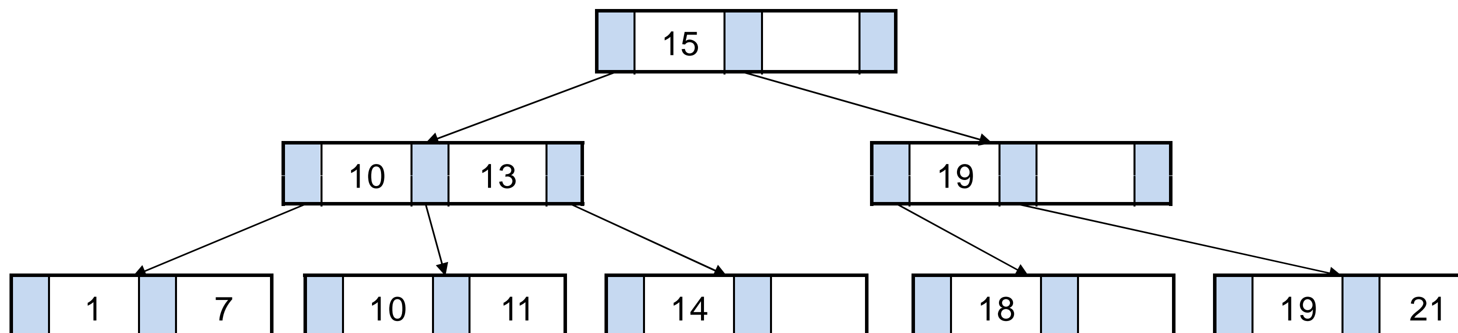
- **Index**
  - A data structure that enables sublinear time lookup. (wikipedia.org)
  - Key → Record

- **Examples**
  - File system:
    - File name → File metadata (directory)
  - DBMS:
    - Primary key → Database record

B+Tree (1)

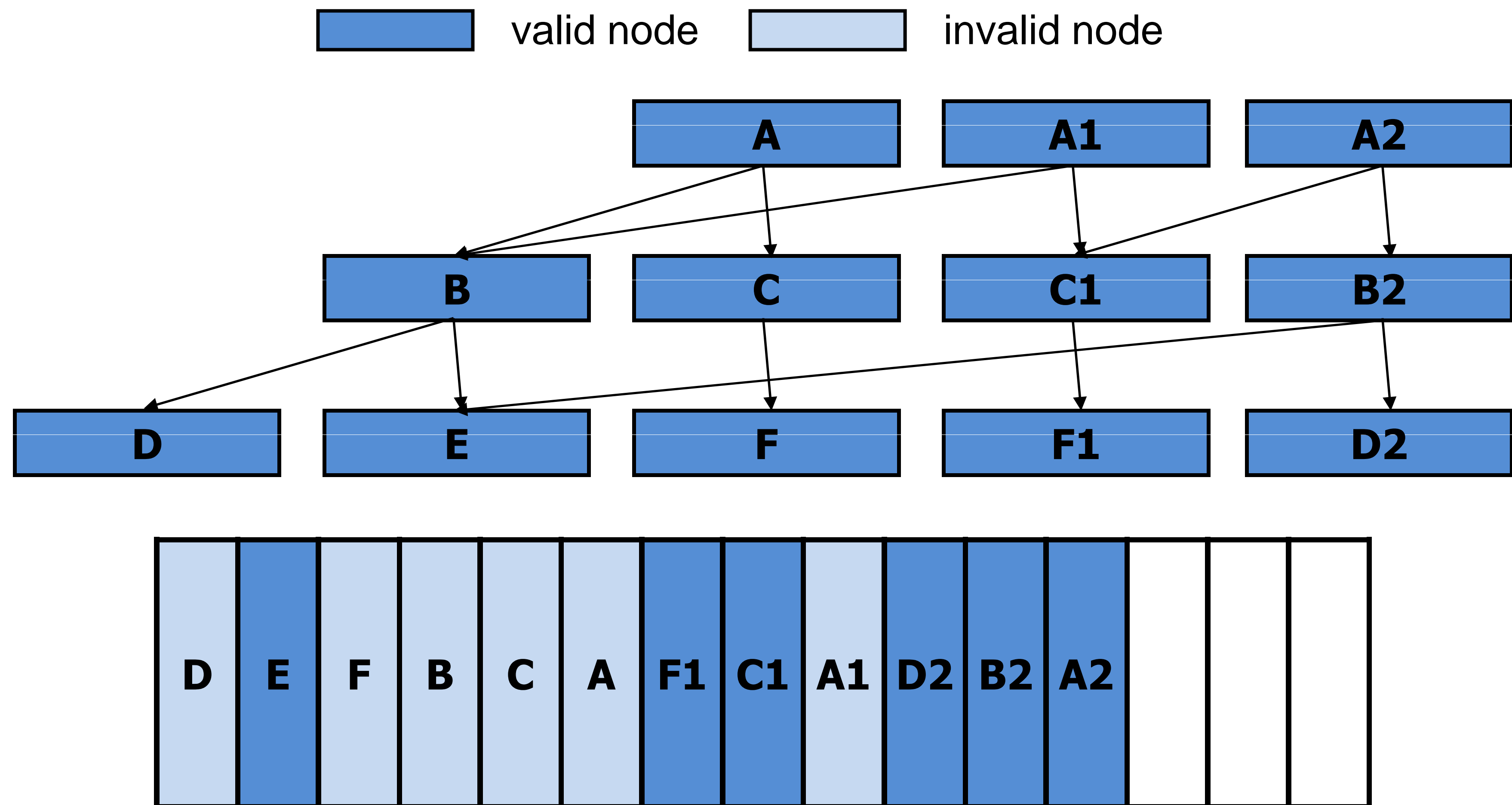
Characteristics

- Disk-based index structure
- Used in most file systems and DBMSs
- *f*-ary balanced search tree
- $O(\log_f N)$ of retrieval, insertion, and deletion
- Records are only in leaf nodes



B+Tree (2)

- **B+Tree on Flash**

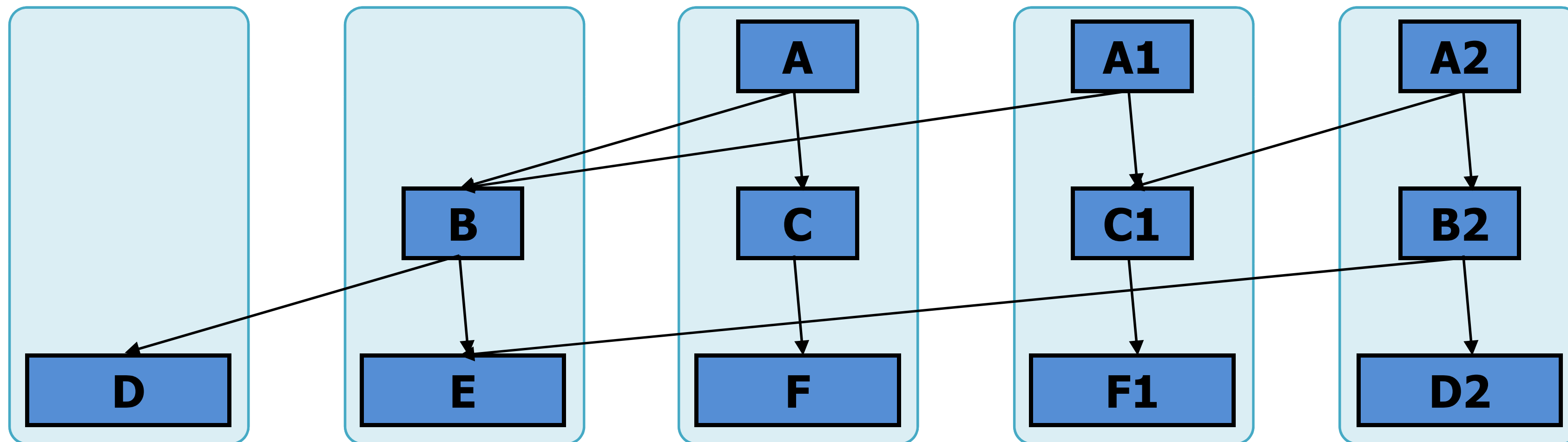


B+Tree (3)

- **B+Tree on Flash (cont'd)**
  - Node size = Page size in NAND flash memory
  - Wandering tree
    - A tree where any update in the tree requires updating parent nodes up to the root
  - Update cost: C_w * H
    - H : The height of B+Tree
    - C_w : The cost of a write operation
  - B+Tree updates can be cached in memory for a while.
    - Journal tree in JFFS3
    - Flush out all the changes in bulk

μ-Tree (1)

valid node | invalid node | page



		A	A1	A2										
	B	C	C1	B2										
D	E	F	F1	D2										

μ-Tree (2)

- **μ-Tree (minimally updated-Tree)**
  - A tree which store all the nodes from leaf to the root into a single page.
- **Pros**
  - Minimal update cost: 1 * C_w
- **Cons**
  - Smaller node size → Higher height
  - Lower space utilization

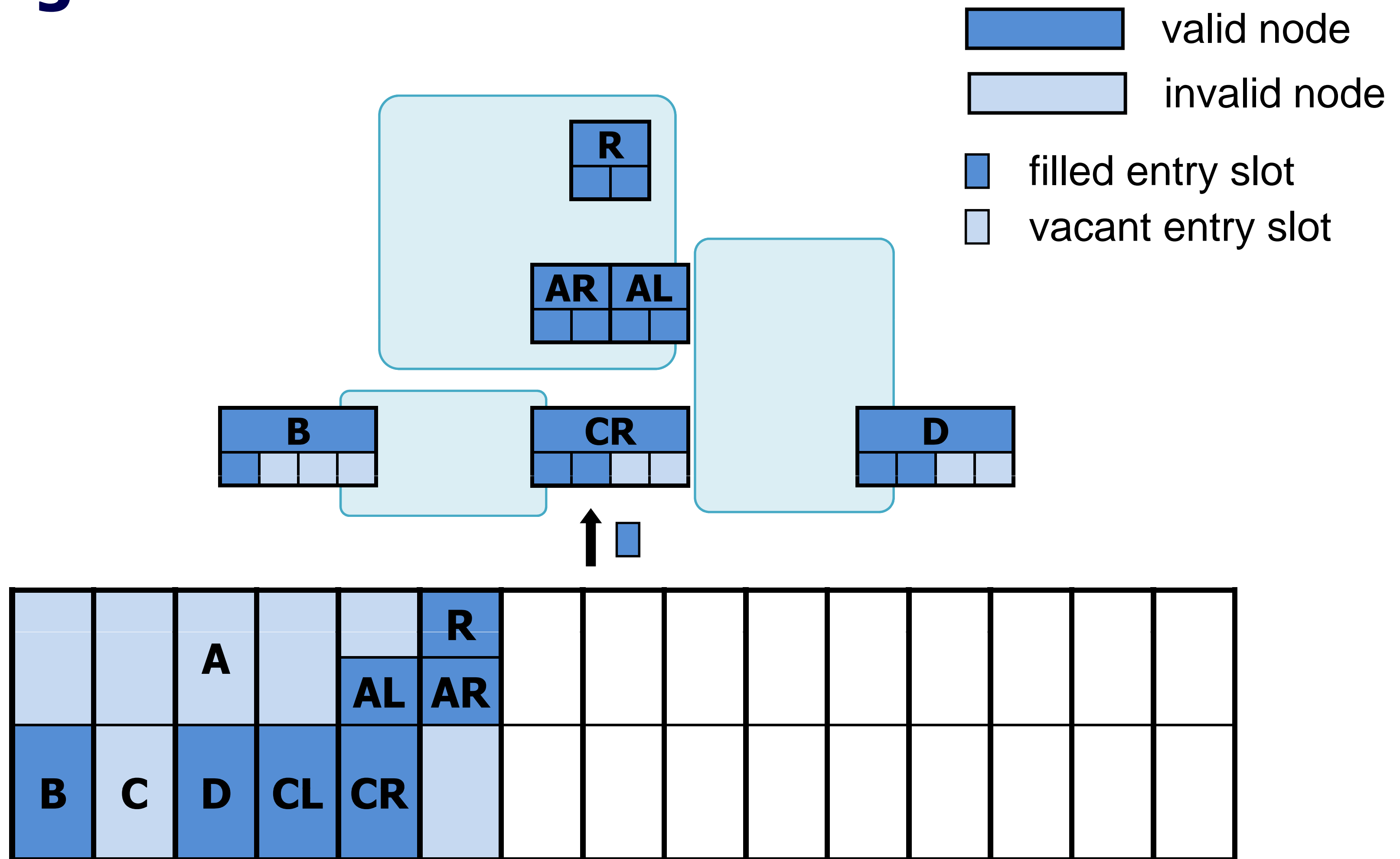
µ-Tree (3)

- **Page Layout**

Height	Layout (top to bottom)
1	Root
2	Root, 1
3	Root, 2, 1
4	Root (512 bytes), 3 (512 bytes), 2 (1024 bytes), 1 (2048 bytes)

μ-Tree (4)

- **Height Increase**



Analysis (1)

- **The Cost of Operations**
  - When there is no garbage collection:

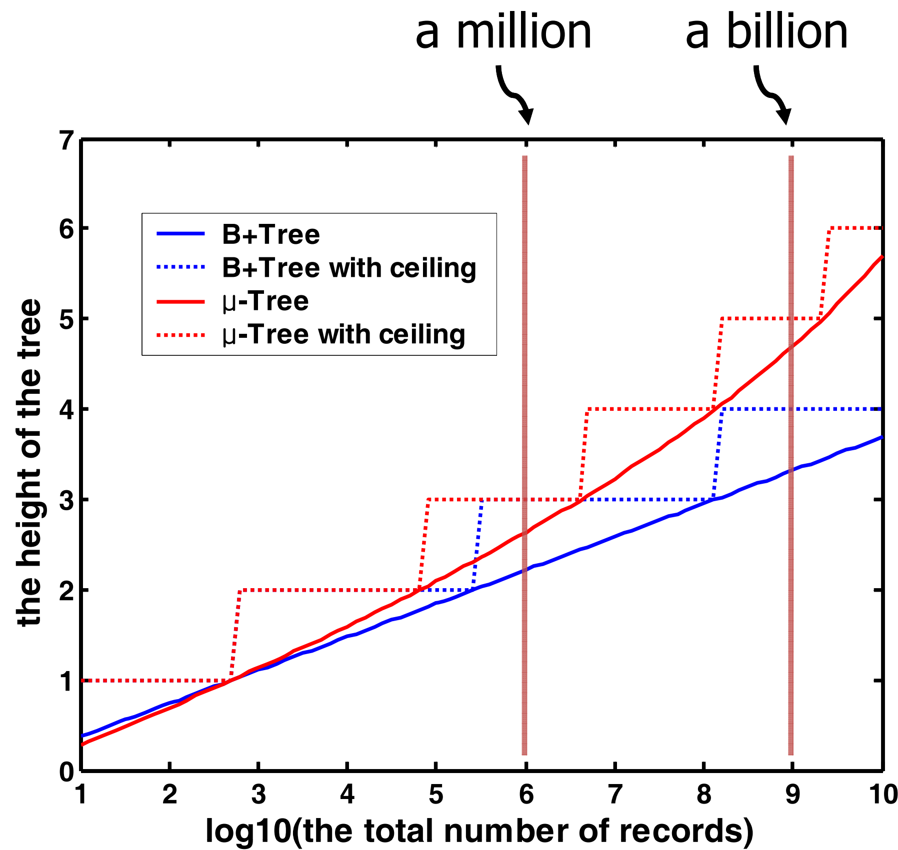
Operations	B^+-Tree	μ-Tree
Retrieval(C_R)	$c_r h_B$	$c_r h_\mu$
Insertion(C_I)	$(c_r + c_w) h_B$	$c_r h_\mu + c_w$
Deletion(C_D)	$(c_r + c_w) h_B$	$c_r h_\mu + c_w$

h_B, h_μ	the height of B^+-Tree (h_B) or μ-Tree (h_μ)
c_r	the cost of read operation on flash memory
c_w	the cost of write operation on flash memory

Analysis (2)

	B+-Tree	μ-Tree
Max # of entries per node at l -th level	$d_l = f, \text{ for all } 1 \leq l \leq h_B$	$d_l = \begin{cases} f/2^{l-1} & \text{if root } (l = h_\mu); \\ f/2^l & \text{otherwise } (l < h_\mu). \end{cases}$
The total # of records indexed with a height h tree	$n_h = \prod_{i=1}^{h} d_i = f^h$	$n_h = \prod_{i=1}^{h} d_i = 2\prod_{i=1}^{h}(f/2^i)$
The height of tree needed for indexing n records	$h_B = \log_f n$	$h_\mu = -\log_2 \frac{\sqrt{2}}{f} - \sqrt{\log_2^2 \frac{\sqrt{2}}{f} - 2\log_2 \frac{n}{2}}$

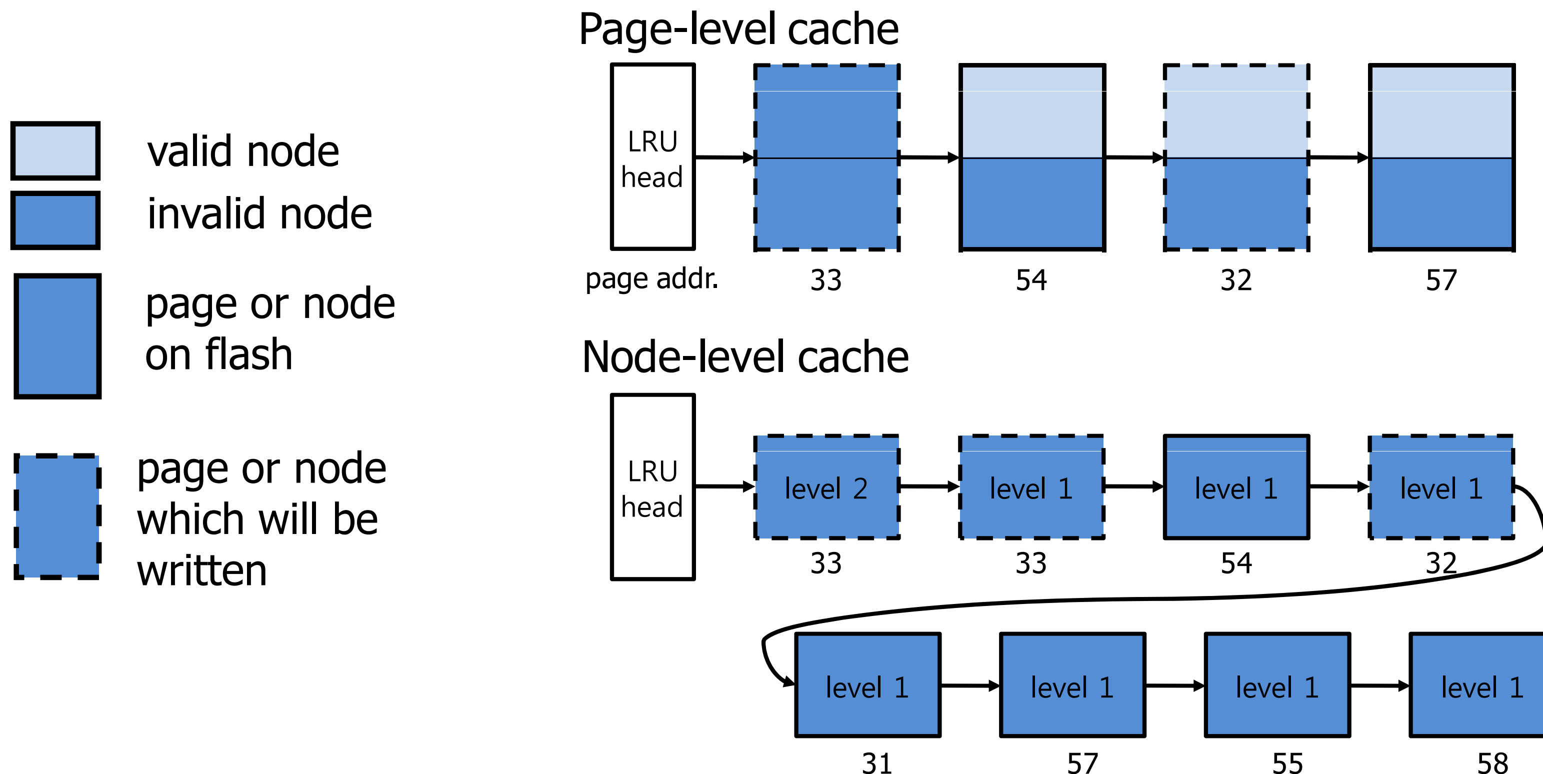
Analysis (3)



Height	B+-Tree	µ-Tree
1	512	512
2	256K	64K
3	128M	4M
4	64G	128M
5	32T	2G
6	16P	16G
7	8E	64G

Caching

- **Page-level cache vs. Node-level cache**

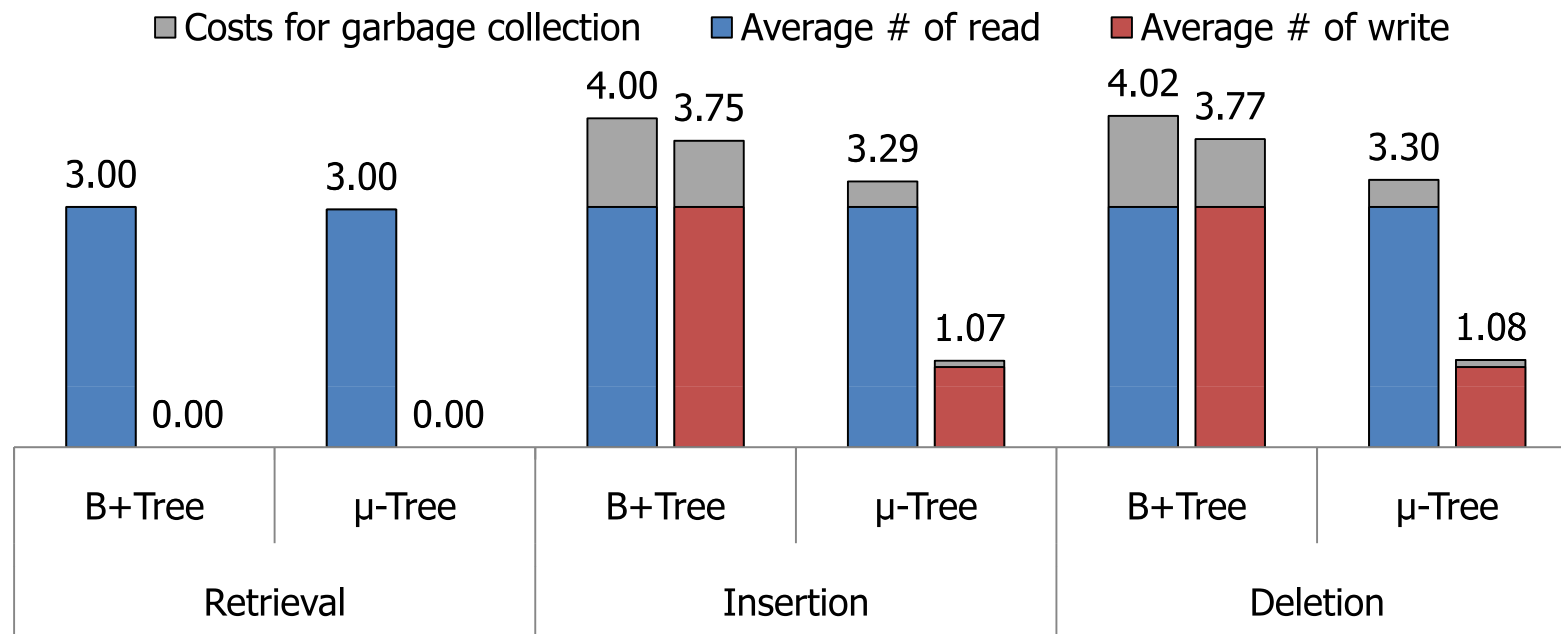


Evaluation (1)

- **Simulator**
  - MLC NAND (Samsung K9GAG08U0M-P)
    - Flash size: 64 – 256MB
    - 4KB page, 512KB block
    - Read latency: 165.6 µs
    - Write latency: 905.8 µs
    - Erase latency: 1500 µs
  - Fanout $f = 512$
    - 4-byte key
    - 4-byte pointer

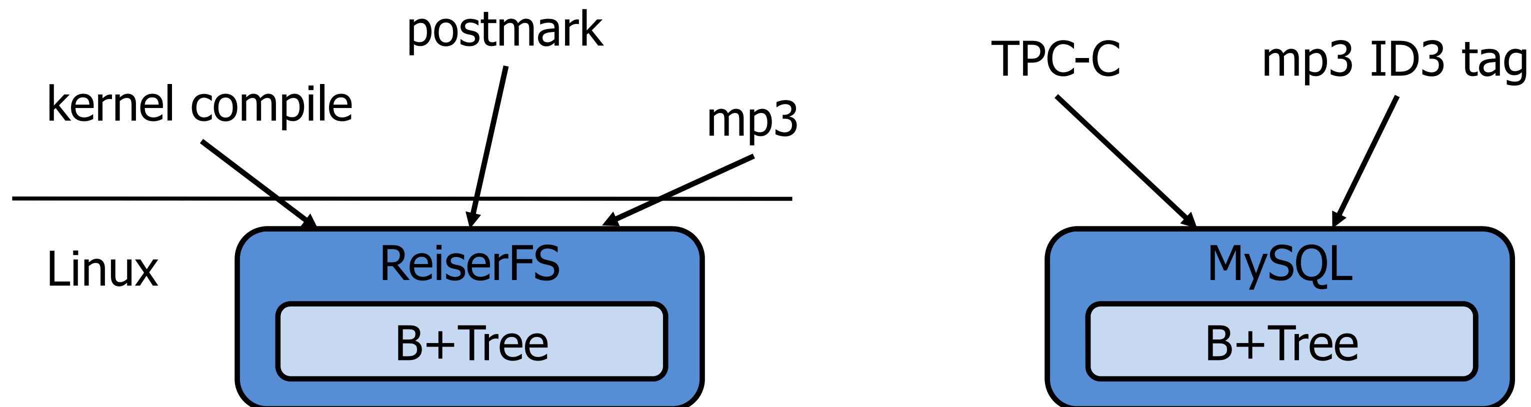
Evaluation (2)

- **Microbenchmarks**
  - Total 1 million records (height = 3)
  - 10,000 retrievals, deletions, and insertions
  - 64MB, No cache



Evaluation (3)

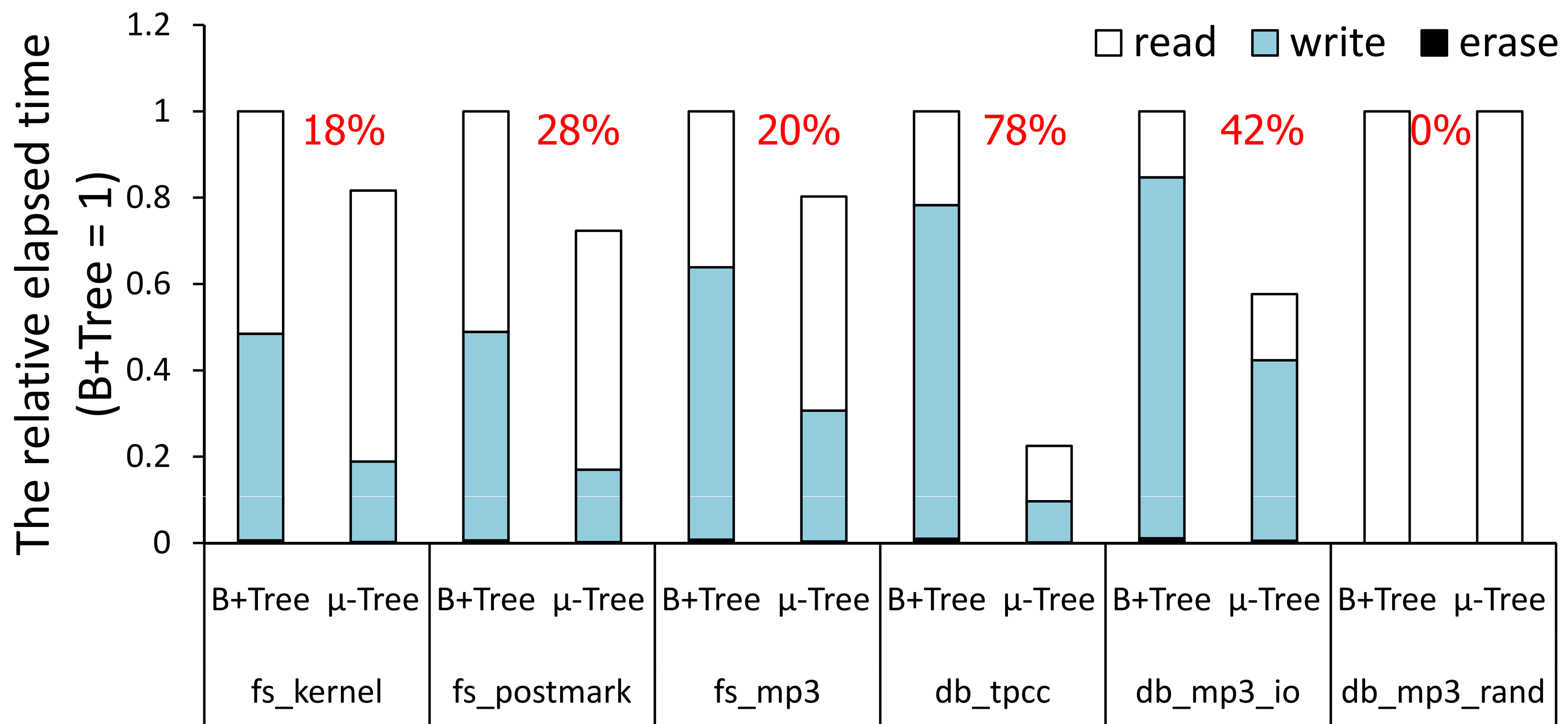
- **Traces from Real Workloads**



Trace		Retrieval	Insertion	Deletion	Initial State
ReiserFS	fs_kernel	2,274,867	260,974	236,027	0
	fs_postmark	4,617,494	574,148	391,847	0
	fs_mp3	512,986	223,188	23,950	0
MySQL	db_tpcc	2,283	1,419	0	4,805,268
	db_mp3_io	0	5,992	4,280	0
	db_mp3_rand	1,712	0	0	1,712

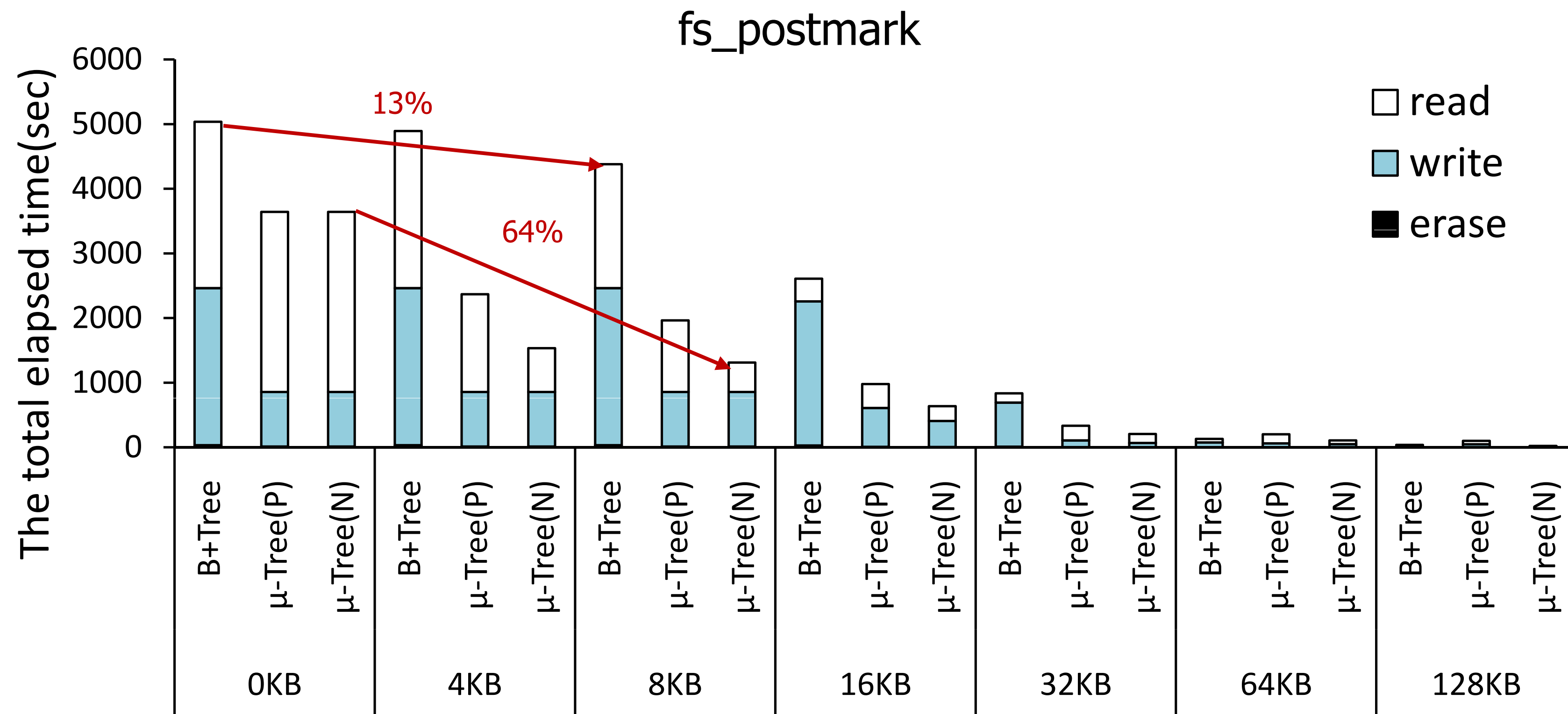
Evaluation (4)

- **Real Workload Results**
  - No cache
  - 64MB (256MB in db_tpcc)



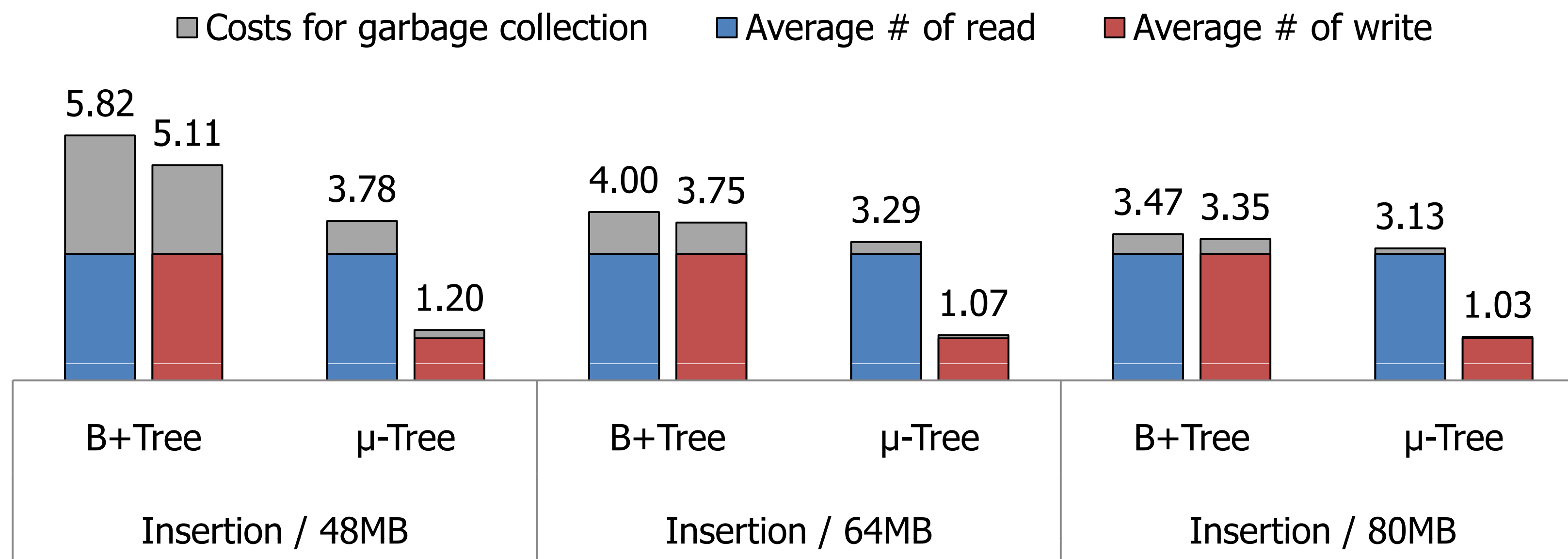
Evaluation (5)

- **Effects of Caching**



Evaluation (6)

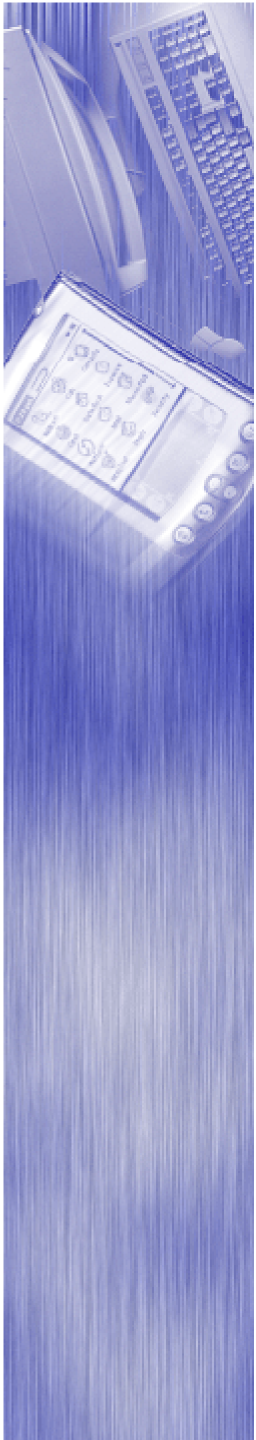
- **Garbage Collection Costs**
  - Total 1 million records (height = 3)
    - Actual tree size: 11MB (B+Tree), 22MB (μ-Tree)
  - 10,000 insertions without caching



Summary

- **μ-Tree**
  - A variant of B+Tree.
  - The sum of the size of nodes in a path from the root to the leaf is constant.
  - The size and the position of a node in each level do not change after height increase or decrease except for the root node and its children.
  - Only the direct ancestors can be stored in the upper levels, if any.

  - Less writes, slightly more reads.



μFS

KAIST

μFS (1)

- **μFS Requirements**
  - A flash-aware file system for portable multimedia devices (mp3 players, digital camcorders, etc.)
  - MLC NAND support
  - High-performance metadata operations
    - Lookup, unlink, unlink all, random seek, truncate, …
  - Real-time support
    - For audio/video recorders (MP3, MP4, SD/HD, etc.)
    - HD: 2MB/s min.
  - Power-off recovery

μFS (2)

- **Index Structures in File Systems**
  - Directory
    - File name → Inode number
    - Linear array, B+Tree (or variants), ...
    - Local vs. Global
  - File structure
    - Inode → File contents
    - FAT, Block pointers, B+Tree (or variants), ...
    - Fixed vs. Variable

μFS Architecture (1)

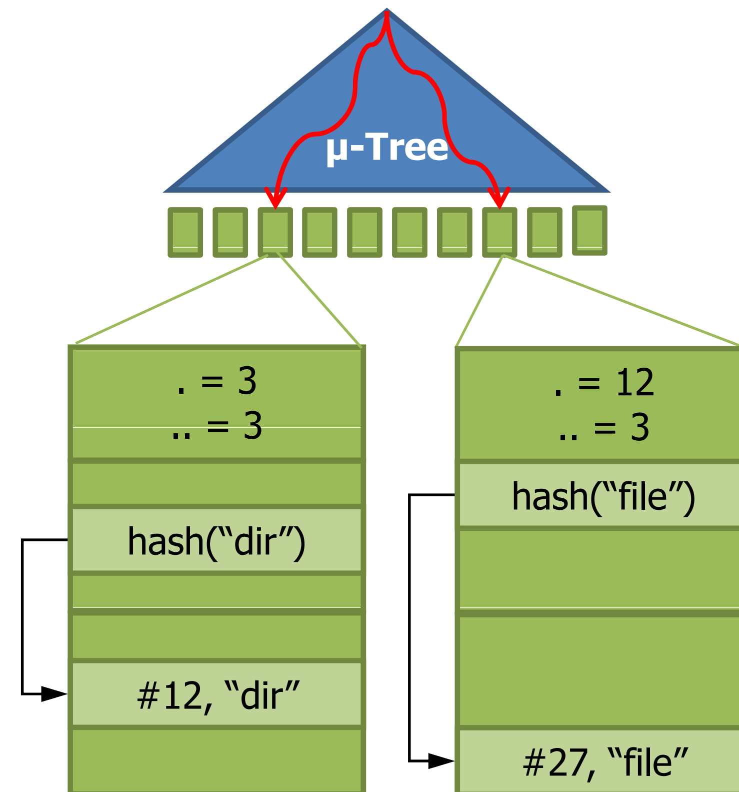
▪ μ-Tree for Directories

- Directory tree key

MSB	1	Parent inode #	Filename hash

- Directory page

Journaling info.		
Current inode # (.)		
Parent inode # (..)		
Hash	Offset	
Hash	Offset	
...	...	
...	...	...
Header	Inode #	Filename
Header	Inode #	Filename



μFS Architecture (2)

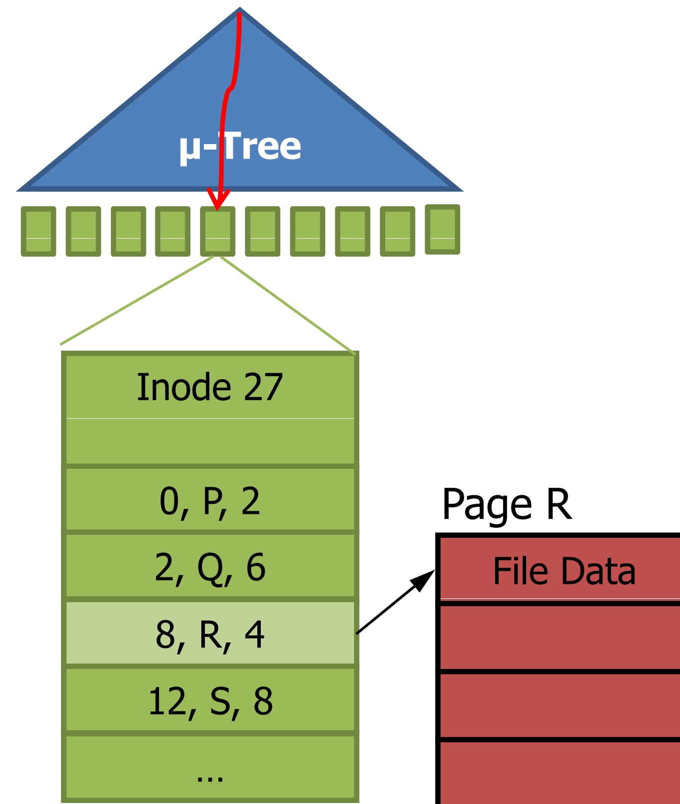
- **μ-Tree for Files**
  - File tree key

  - Directory page

Journaling info.		
Inode Status		
Offset	Page Address	Length
...	...	...

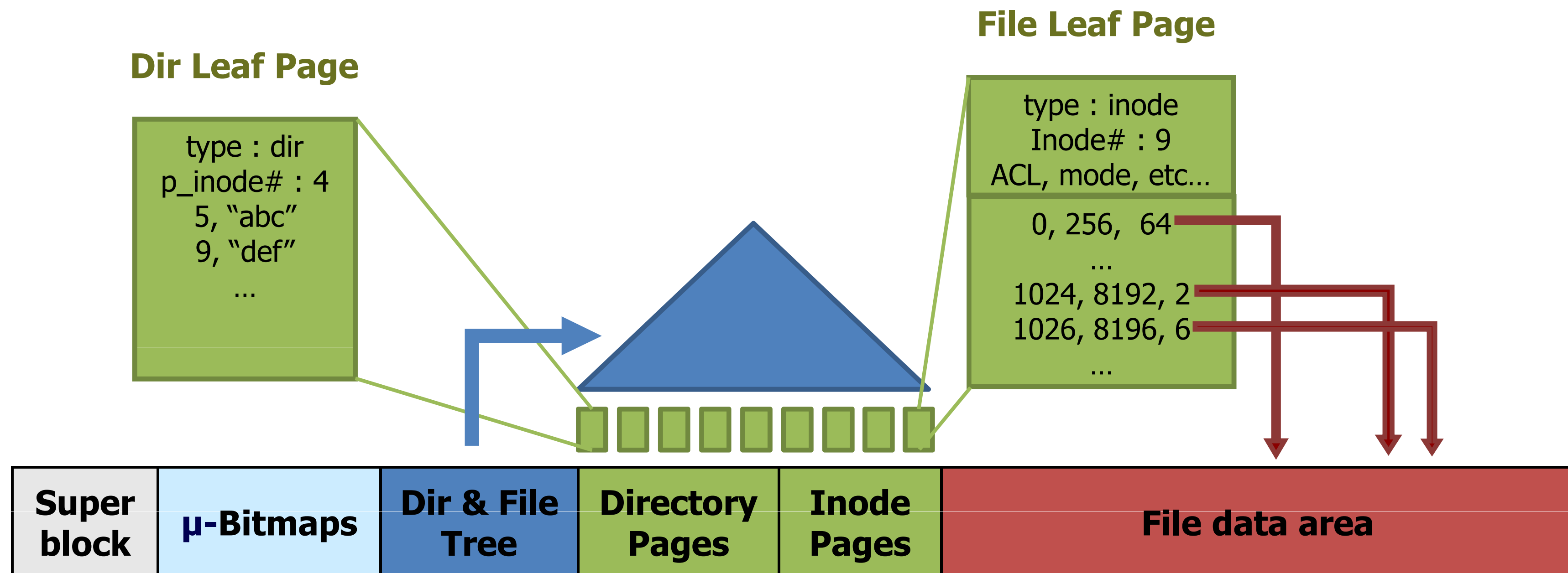
Extents info.

seeking "/dir/file" at offset 32768



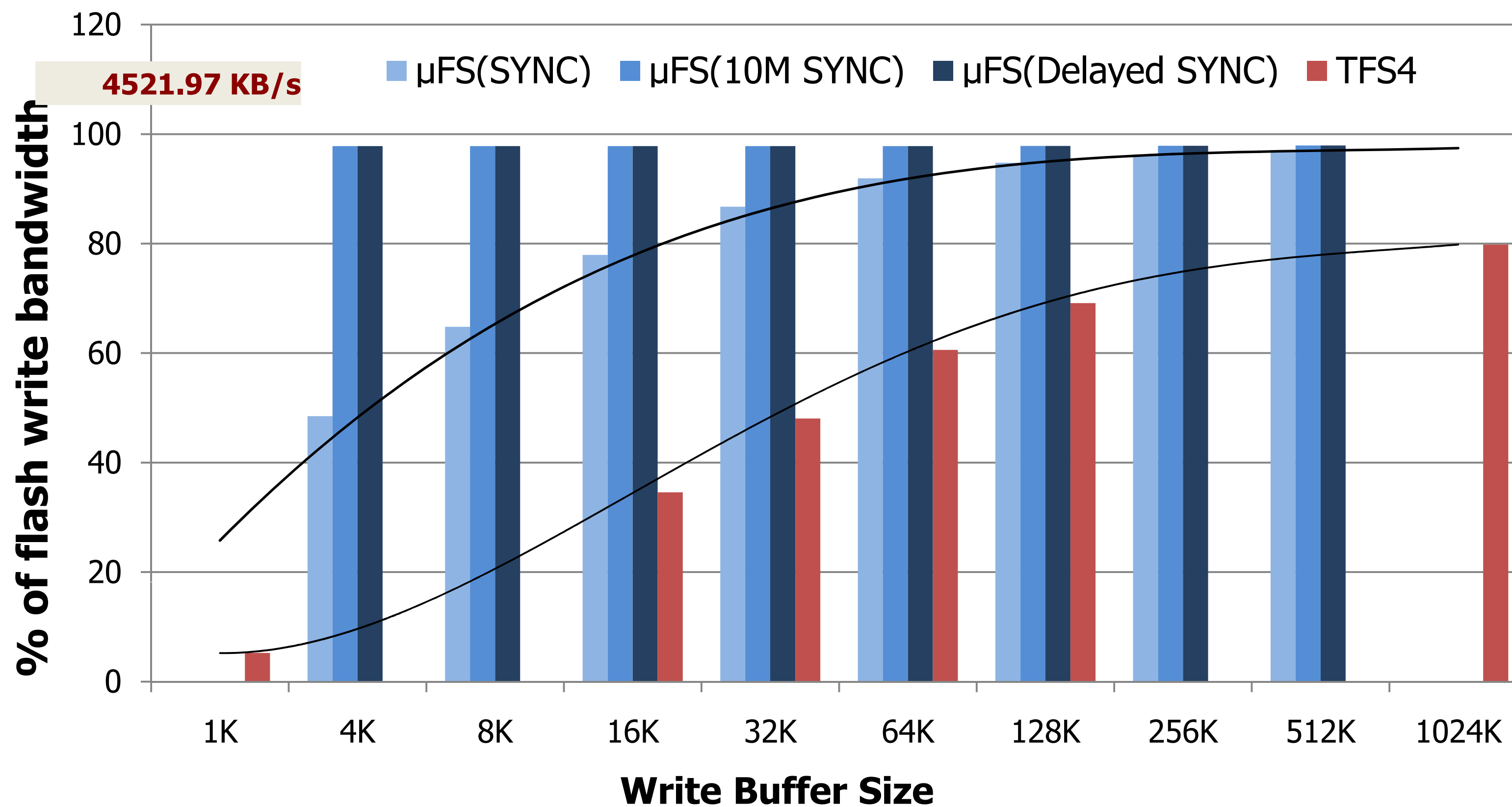
µFS Architecture (3)

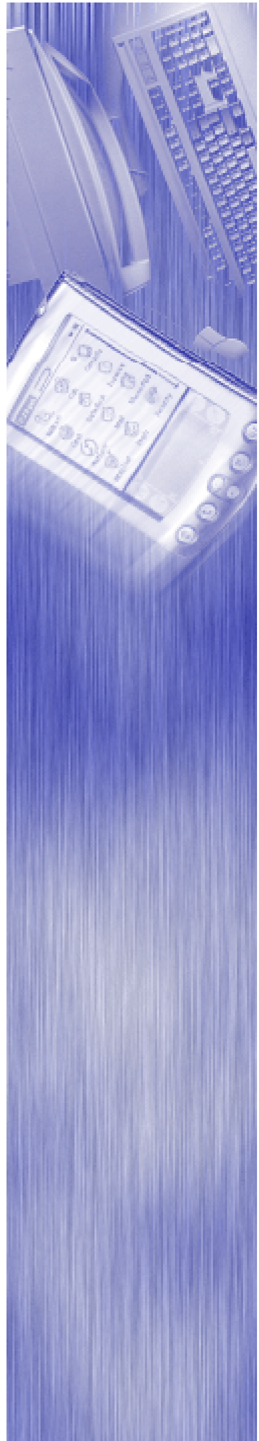
- **Logical Layout**



µFS Performance

- **Write Bandwidth**
  - Write 3GB (average file size: 4.44MB)



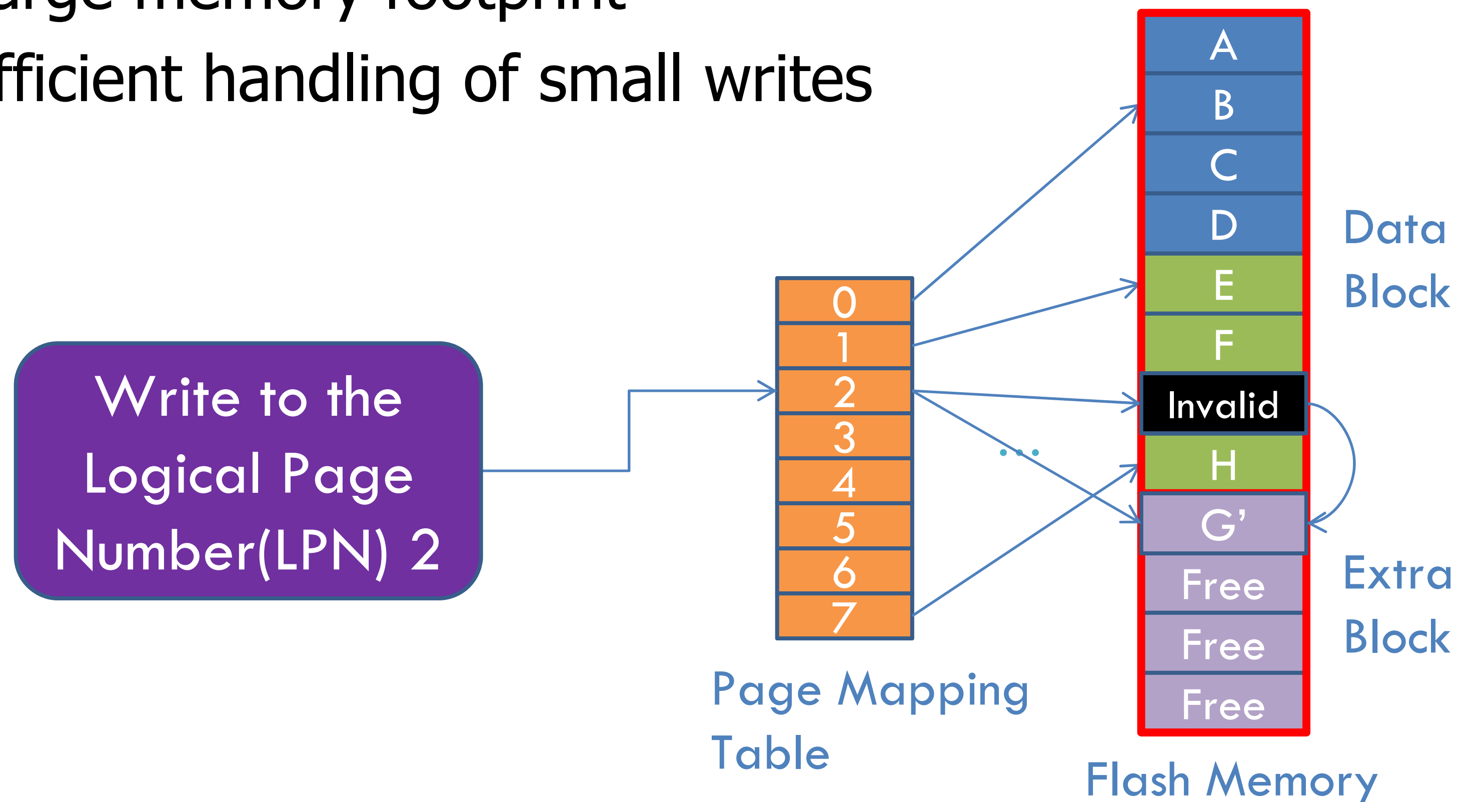


μ-FTL

KAIST

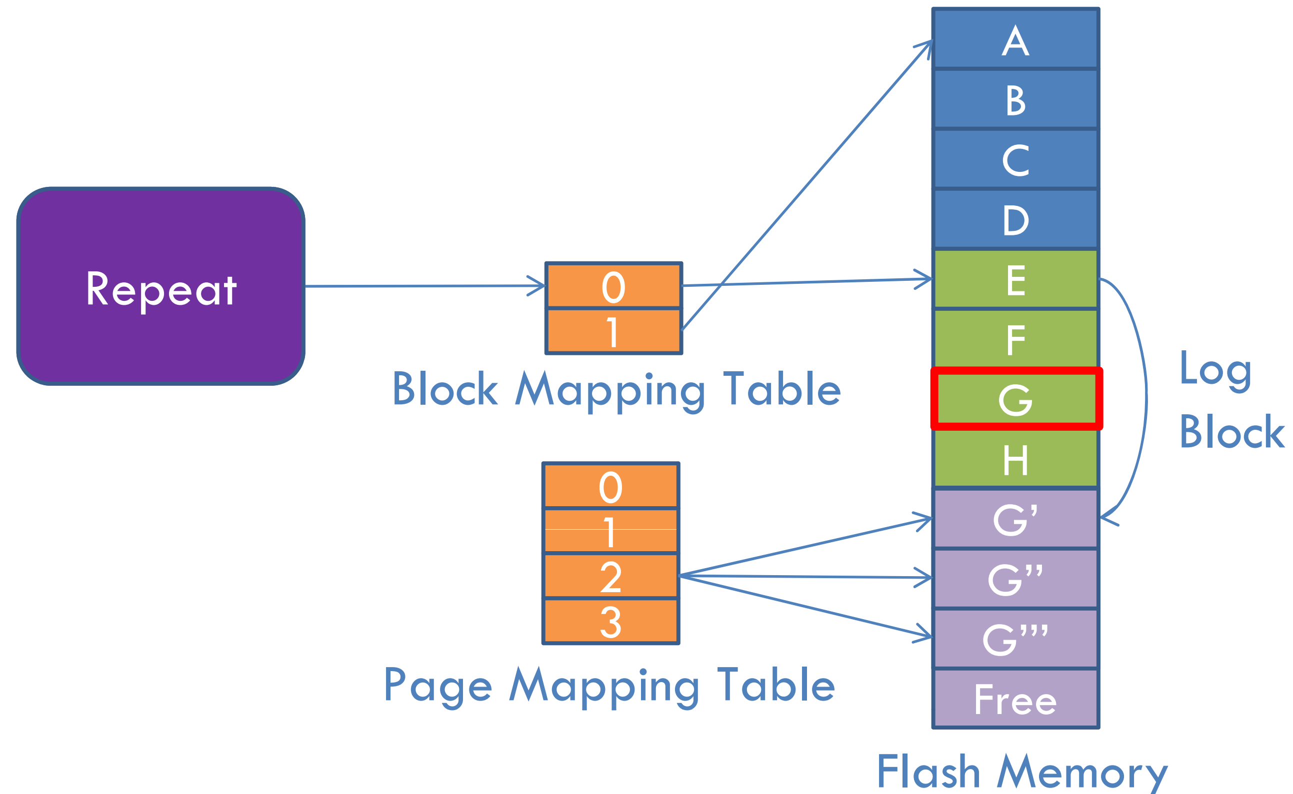
Page Mapping

- **Characteristics**
  - Each table entry maps one page
  - Large memory footprint
  - Efficient handling of small writes



Log Block Scheme

- **Log Block**
  - A temporary storage for small writes
  - Incremental updates from the first page



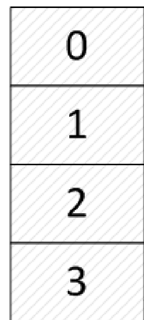
μ-FTL (1)

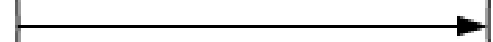
- **Index Structure in FTL**
  - LPN → <PBN, PPN>
    - LPN: Logical Page Number
    - PBN: Physical Block Number
    - PPN: Physical Page Number
  - Table (linear array) is most widely used.
  - Fixed mapping granularity: Page or Block
  - Typical write patterns in real workloads
    - Small and random
    - Large and sequential

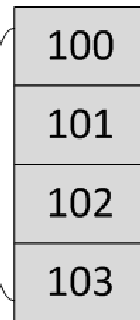
μ-FTL (2)

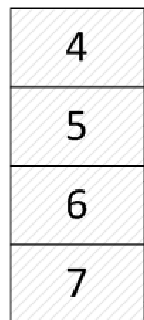
- **Example**

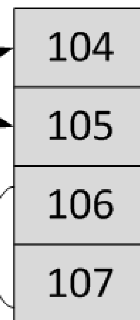
Logical Blocks Physical Blocks

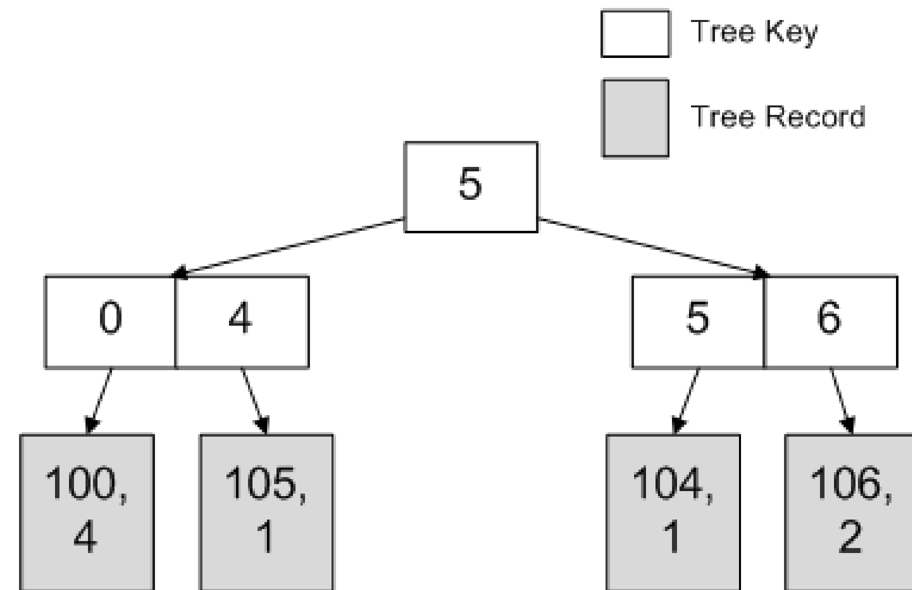






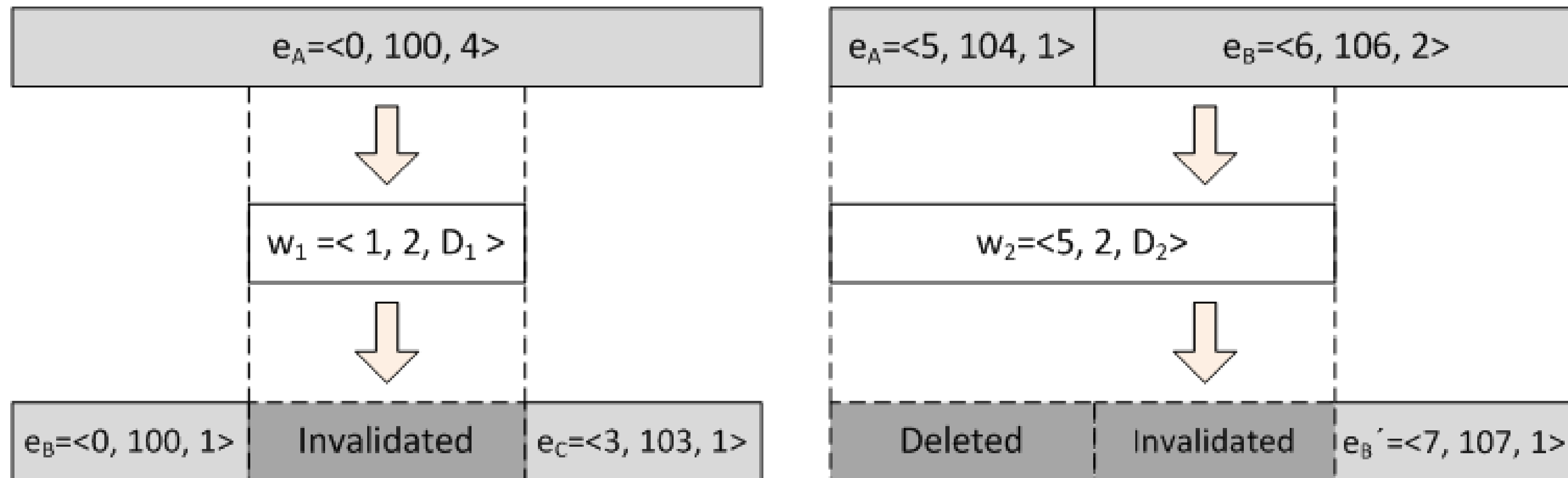






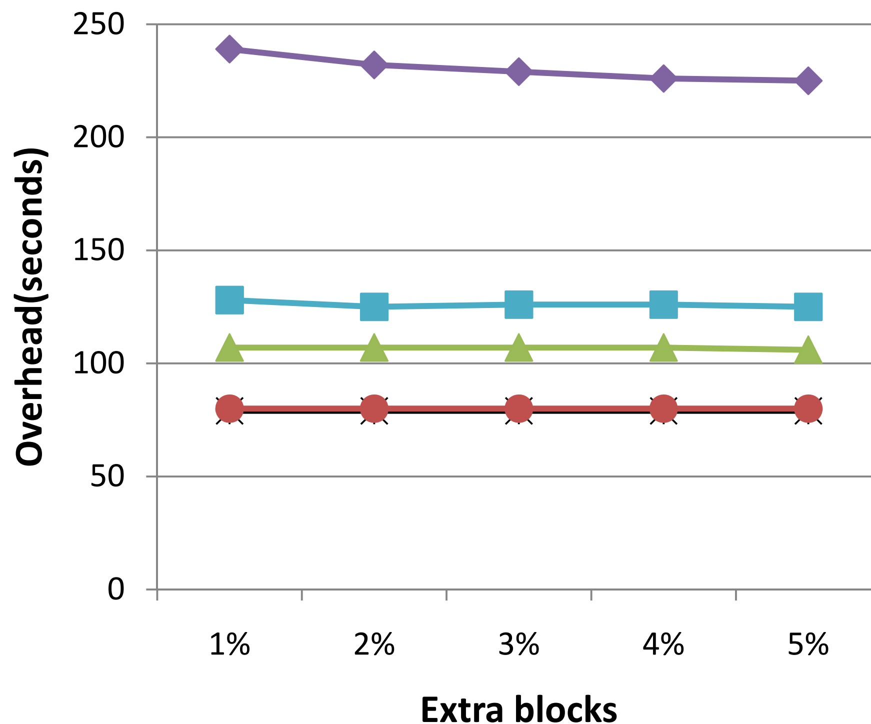
μ-FTL (3)

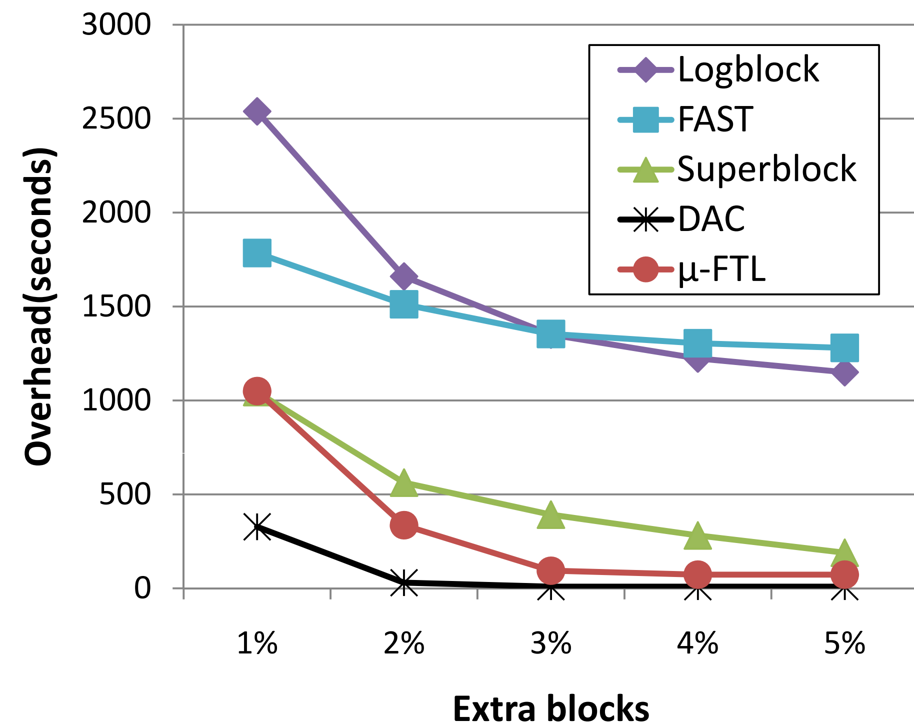
- **Extent Updates**



μ-FTL Performance (1)

- **Garbage Collection Overhead**





MOV_8GB (16KB+44KB+4KB)

WEB_32GB (64KB+160KB+32KB)

μ-FTL Performance (2)

- **Extent Distribution (GENERAL 32GB)**

Installed applications

Downloaded movies

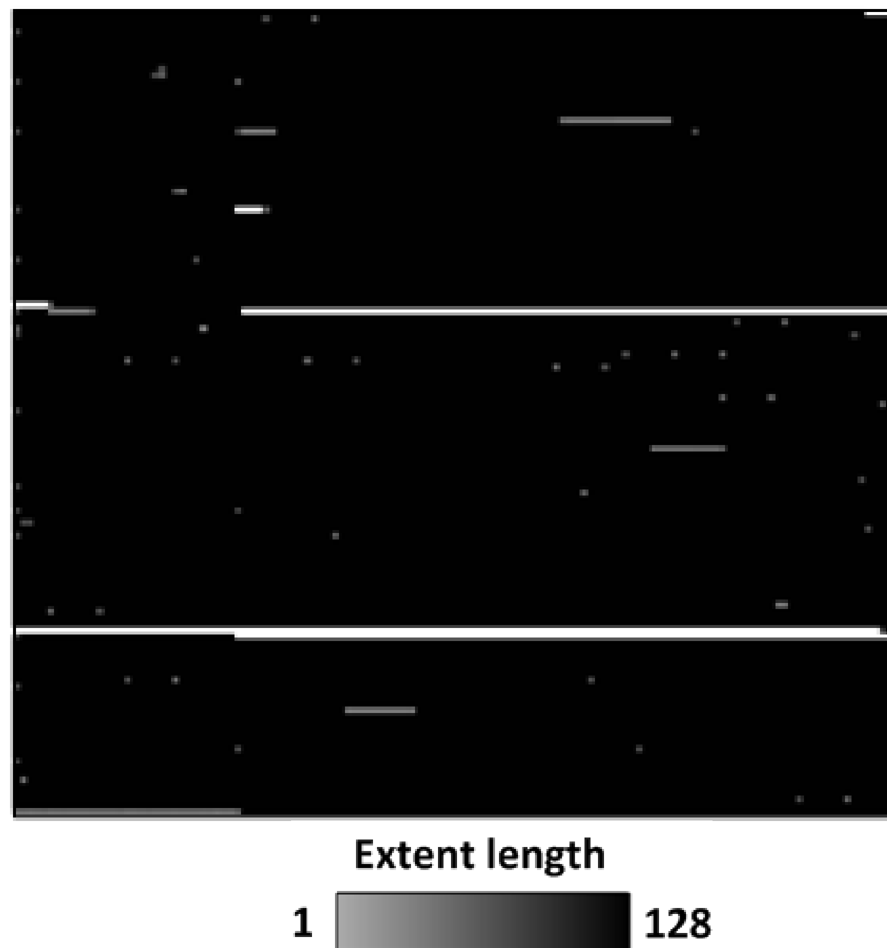
Temporary Internet files

Unused MFT zone

System files

μ-FTL Performance (3)

- **Extent Distribution (MOV 8GB)**



Conclusion

- **μ-Tree**
  - A flash-aware index structure
- **μFS**
  - A flash-aware file system based on μ-Tree
- **μ-FTL**
  - A flash translation layer based on μ-Tree