Buffer Management Techniques

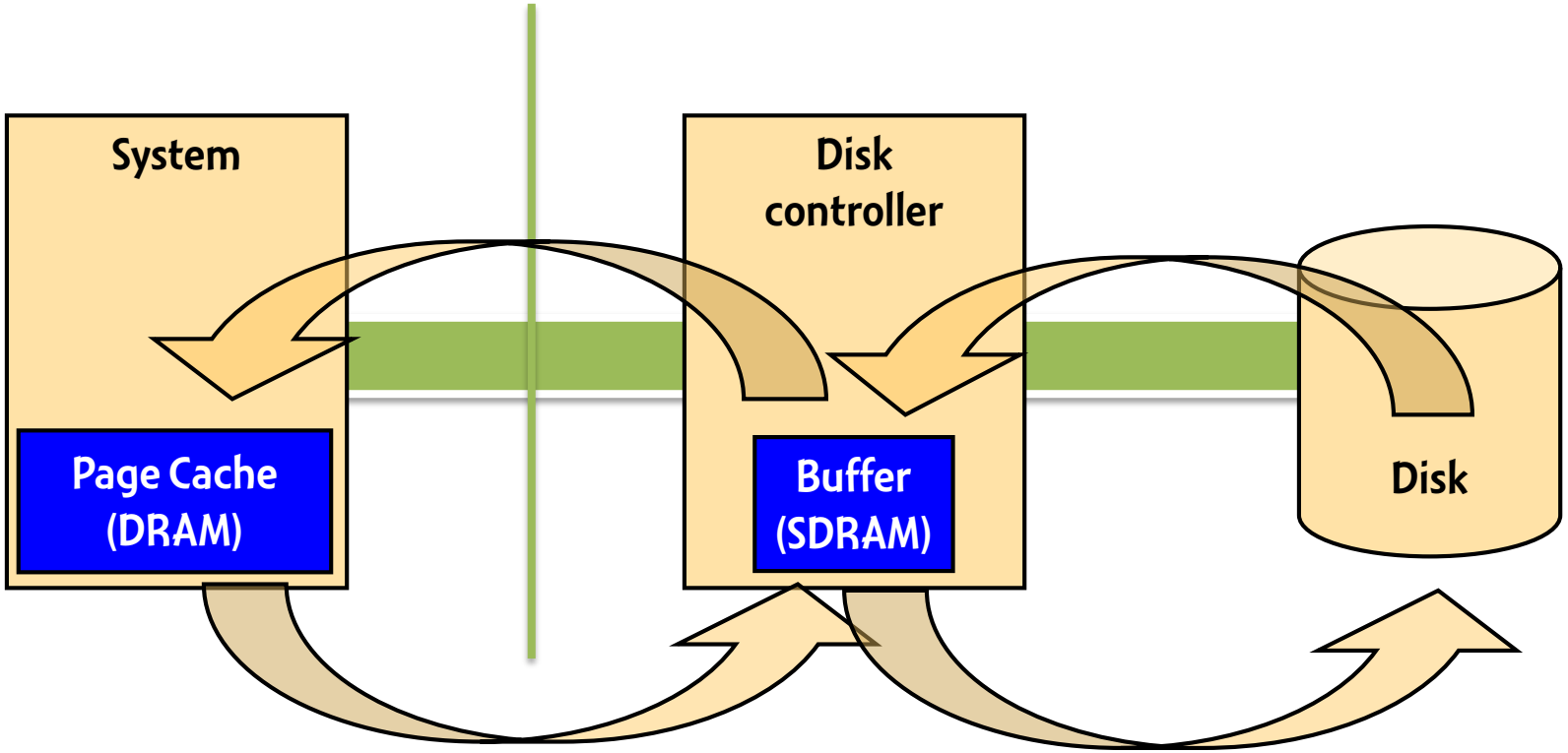
Jihong Kim
Dept. of CSE, SNU

Outline

- Overview of Buffer/Cache for Storage Devices
  - Page Cache & Disk Cache
- Flash-Aware Buffer Management Schemes
  - CFLRU
  - FAB
  - BPLRU

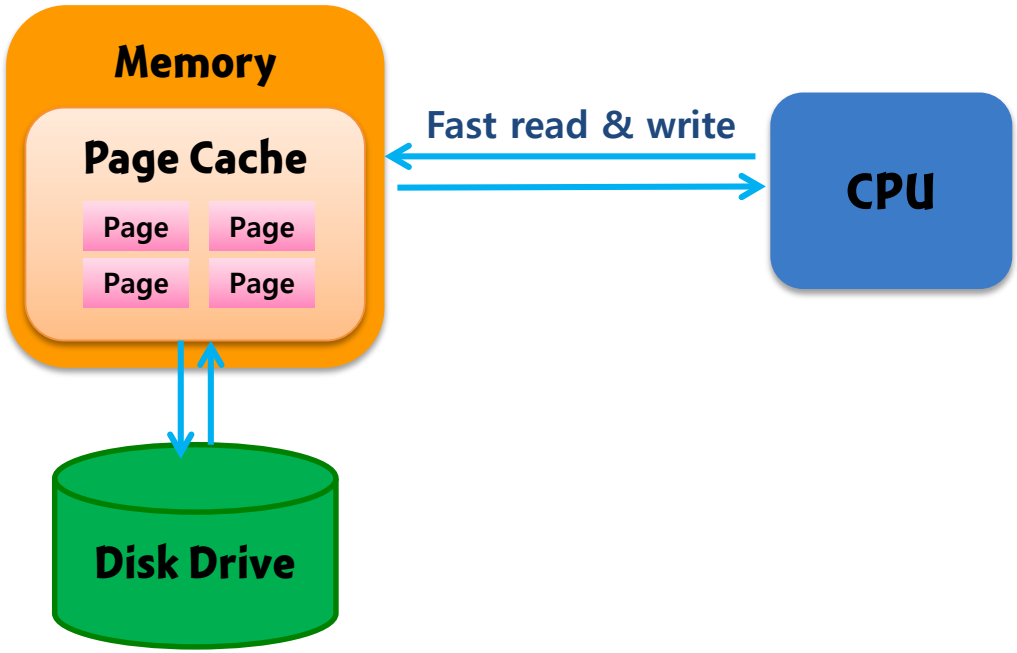
Goal of I/O Buffering

- Reduce the number of expensive accesses to secondary storages



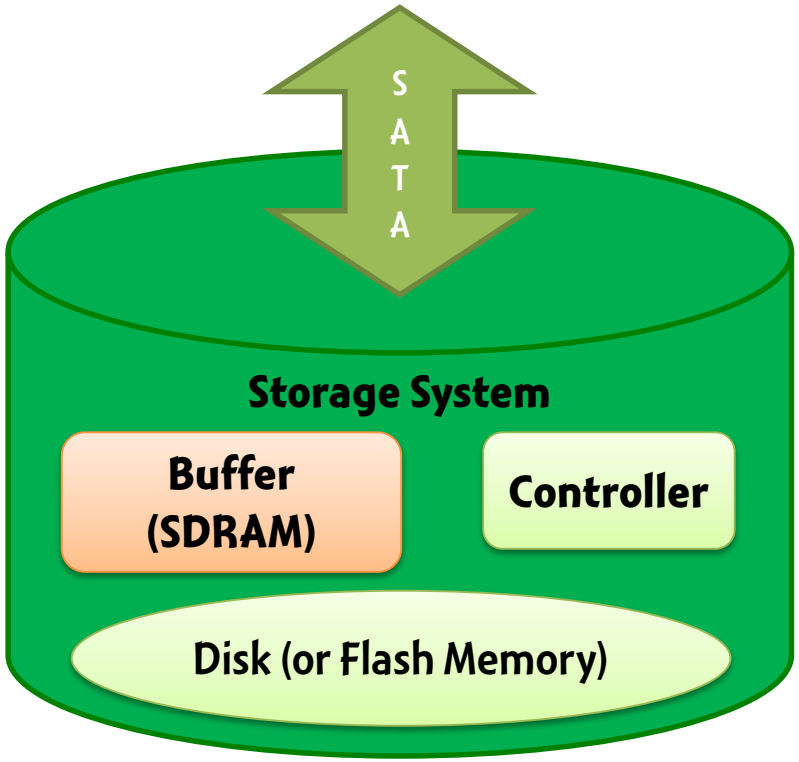
Page Cache

- A disk cache used by OS (e.g., Linux Kernel)
- All regular file I/Os go through the page cache
  - The performance gap between HDD and DRAM is huge.
    - E.g., 700,000 times for latency
      - Response time of DRAM: 12ns
      - Response time of HDD: 8.5ms



Buffer in Storage System

- A cache controlled by firmware in a storage system
- Reduces the number of disk (or flash) accesses

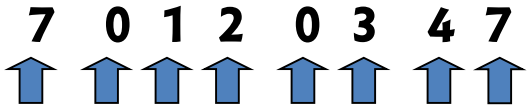


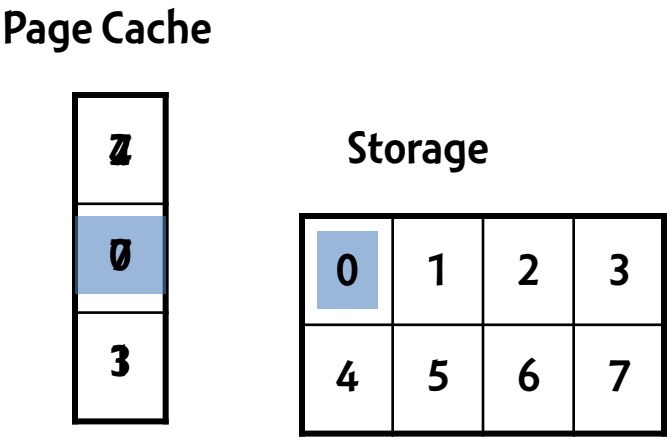
Roles of Buffer within Storage Devices

- Read Caching
  - Improves the read latency
  - Prefetching may be important (e.g., HDD)
- Write Caching
  - Reduces the write service time
    - Dirty data: not yet written to disk
  - Destage operations are necessary
    - Flush command
    - Force Unit Access option
  - Read hits in write Cache
  - Read priority over Destage
- Write caching is more important in flash-based storage systems

A representative page replacement algorithm

- Least Recently Used (LRU) Algorithm
- Select the least recently used page as a victim page





Different Metric for Flash Memory

- Legacy page replacement schemes consider only 'hit rate'
- In flash memory,
  - Each page has a different eviction cost
    - Dirty => expensive write! => increases Garbage Collection
    - Clean => free

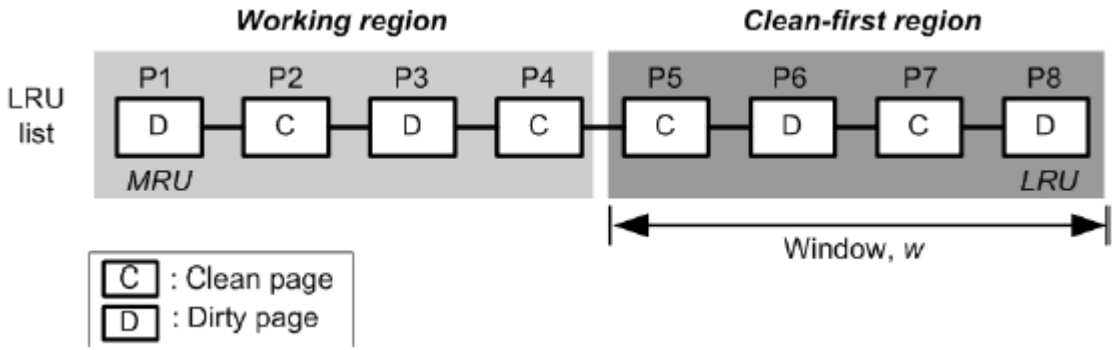
Flash-Aware Buffer Management Techniques

- Clean First Least Recently Used (CFLRU)
  - Page cache level (in host systems)
- Flash Aware Buffer (FAB)
  - Buffer Management Scheme for multimedia embedded systems (e.g. PMP)
- Block Padding Least Recently Used (BPLRU)
  - Buffer cache level (in storage systems)
  - Controlled by firmware in SSD

Clean First LRU (CFLRU)

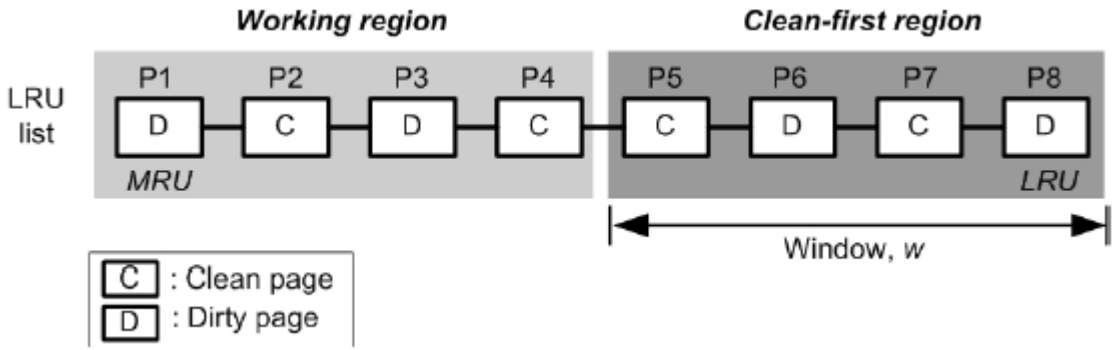
- It is a slight modification of the LRU policy
  - A **dirty page** is a page which was read from the memory and was modified after it was stored in the RAM.
  - A **clean page** is one which remains in the same state as when it was read from the memory.
  - The cost of moving a dirty page from the RAM to the flash is higher than the cost of moving a clean page
  - Moving only the clean pages result in too many misses
    - increasing the time to read the pages
- CLFRU maintains **a threshold region** in which it removes the clean page before moving on to the dirty pages.

Architecture of CFLRU



- Working region
  - Recently used pages
- Clean-first region
  - Candidates for eviction
- Window size
  - The size of Clean-first region
  - Hit ratio depends on the size of this region

Example of CFLRU

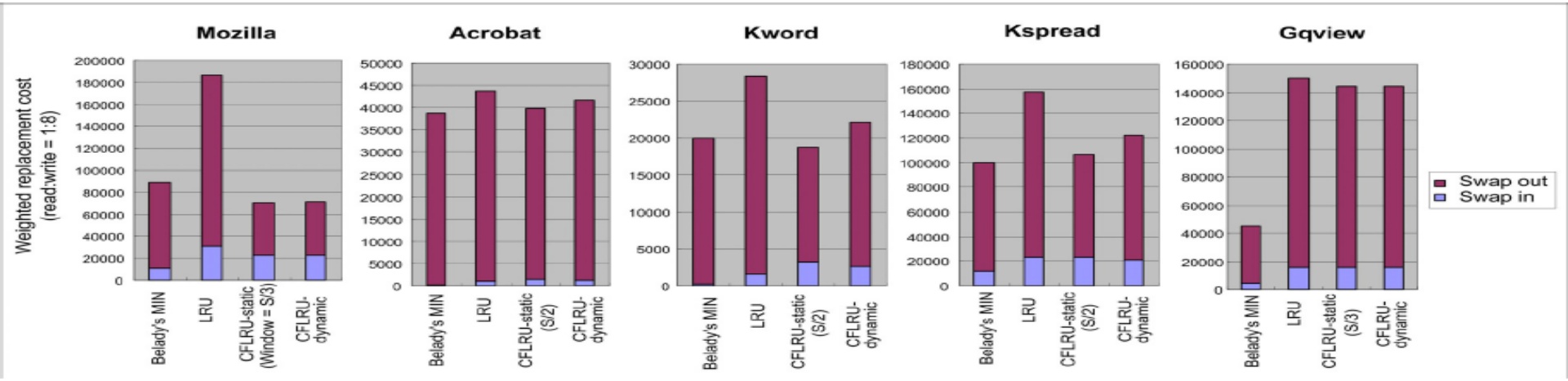


- When a replacement happens
  - Select the LRU clean page in Clean-first region (i.e. Window)
  - If there is no clean page in Clean-first region
    - Select the dirty page
- P7 → P5 → P8 → P6

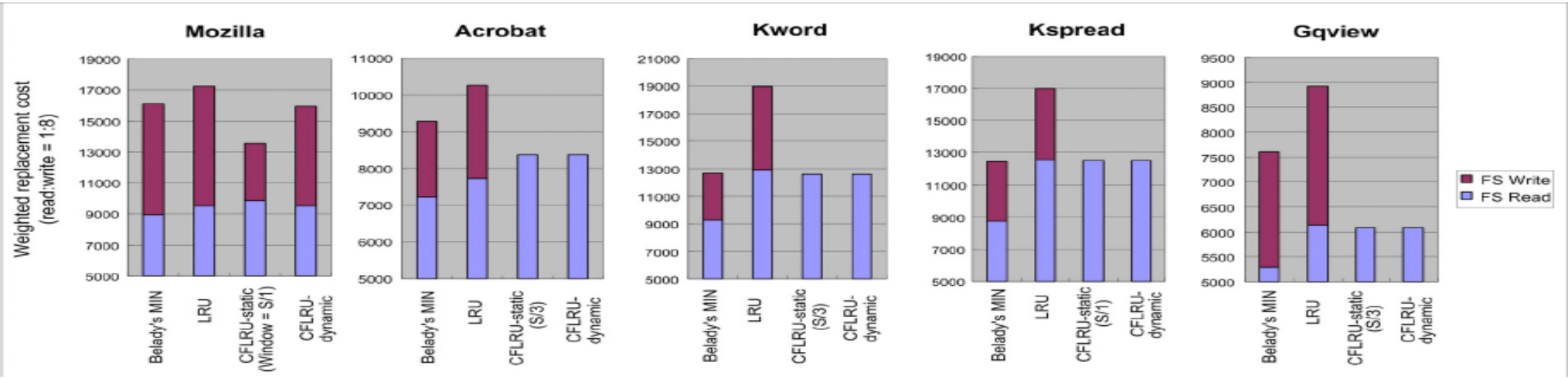
CFLRU algorithm

- Window size (w)
  - Increase w -> decrease hit rate
  - Decrease w-> increase dirty page
- Static method
  - Using static w
- Dynamic method
  - Dynamically change w based on the variance of ratio of read and write

Replacement Cost



(a) Replacement cost for swap system

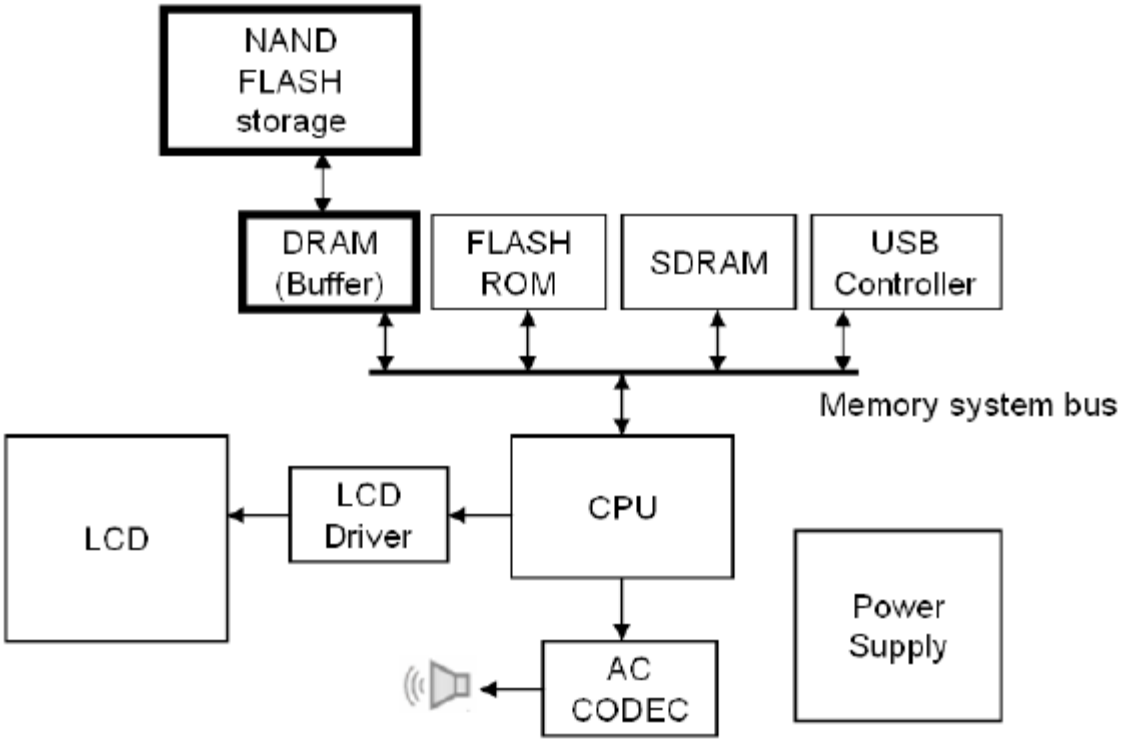


(b) Replacement cost for file system buffer cache

Flash-Aware Buffer(FAB)

- Target System
  - Portable Media Player(PMP)
    - Stores or plays multimedia contents
    - e.g.) MP3 players, portable DVD players, digital cameras, and PDAs
  - Buffer is used for PMPs
    - DRAM buffer is a lot faster than flash memory in write operation

Architecture of Target System



Problem of LRU

- LRU is not adequate for NAND flash memory
- The characteristics of NAND flash memory prohibits LRU from being the best solution
  - E.g) Typical file access pattern of PMP
    - Long sequential access for media data
    - Several short accesses for metadata of the file
    - LRU cannot hold the metadata in the buffer because the long sequential access pushes them away from the buffer

    =>The evicted metadata causes frequent garbage collection

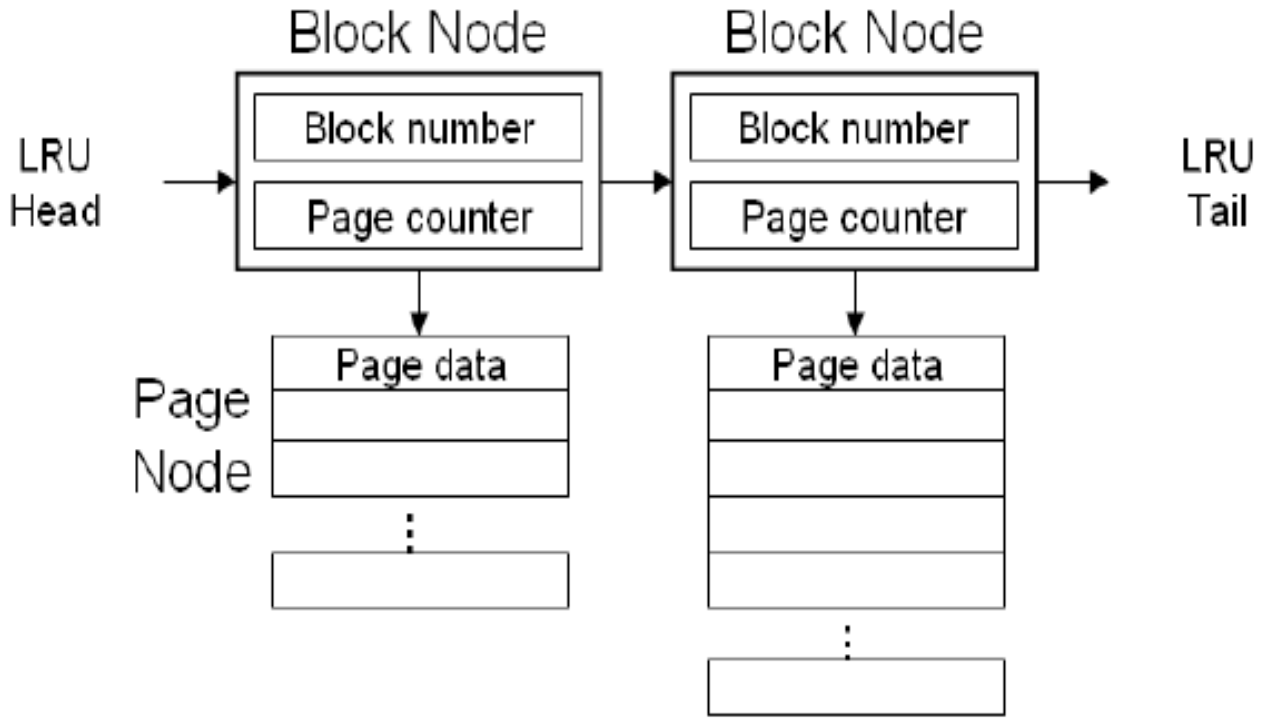
    =>Long sequential pattern is broken by the short metadata

Design Goal of FAB

- Effectively minimizes the number of write and erase operations to the flash memory
  - Helps the storage controller to utilize the switch merge
    - Reduce the G.C overhead
  - Maximize overwrite of hot pages in buffer
    - Reduce the number of triggered G.C.

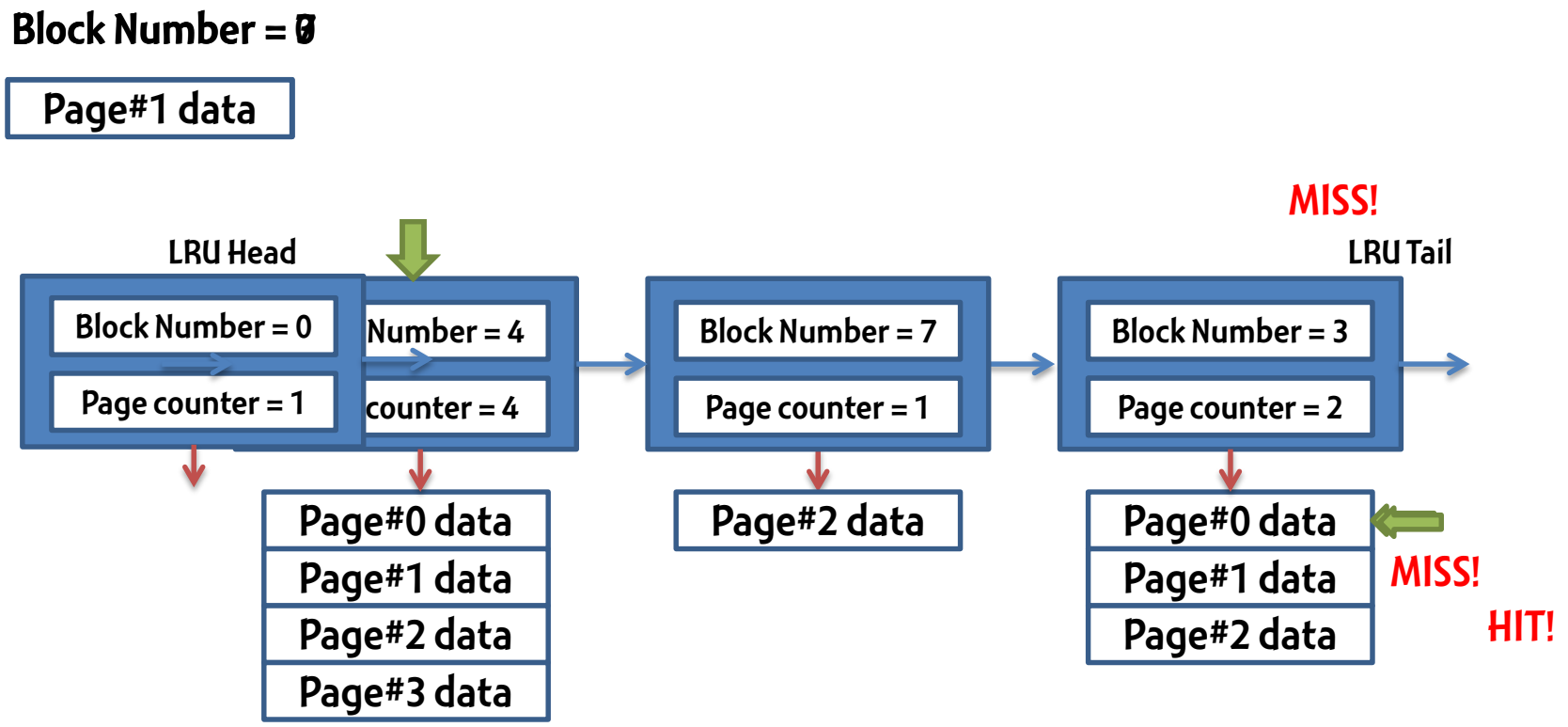
=> FAB selects a victim block based on the ratio of valid pages in the block rather than based on its recency

Data Structures for FAB



Horizontally, the block nodes are linked as a linked list
Vertically, the pages nodes are listed for each block node

Handling a Write Request



Handling a Write Request

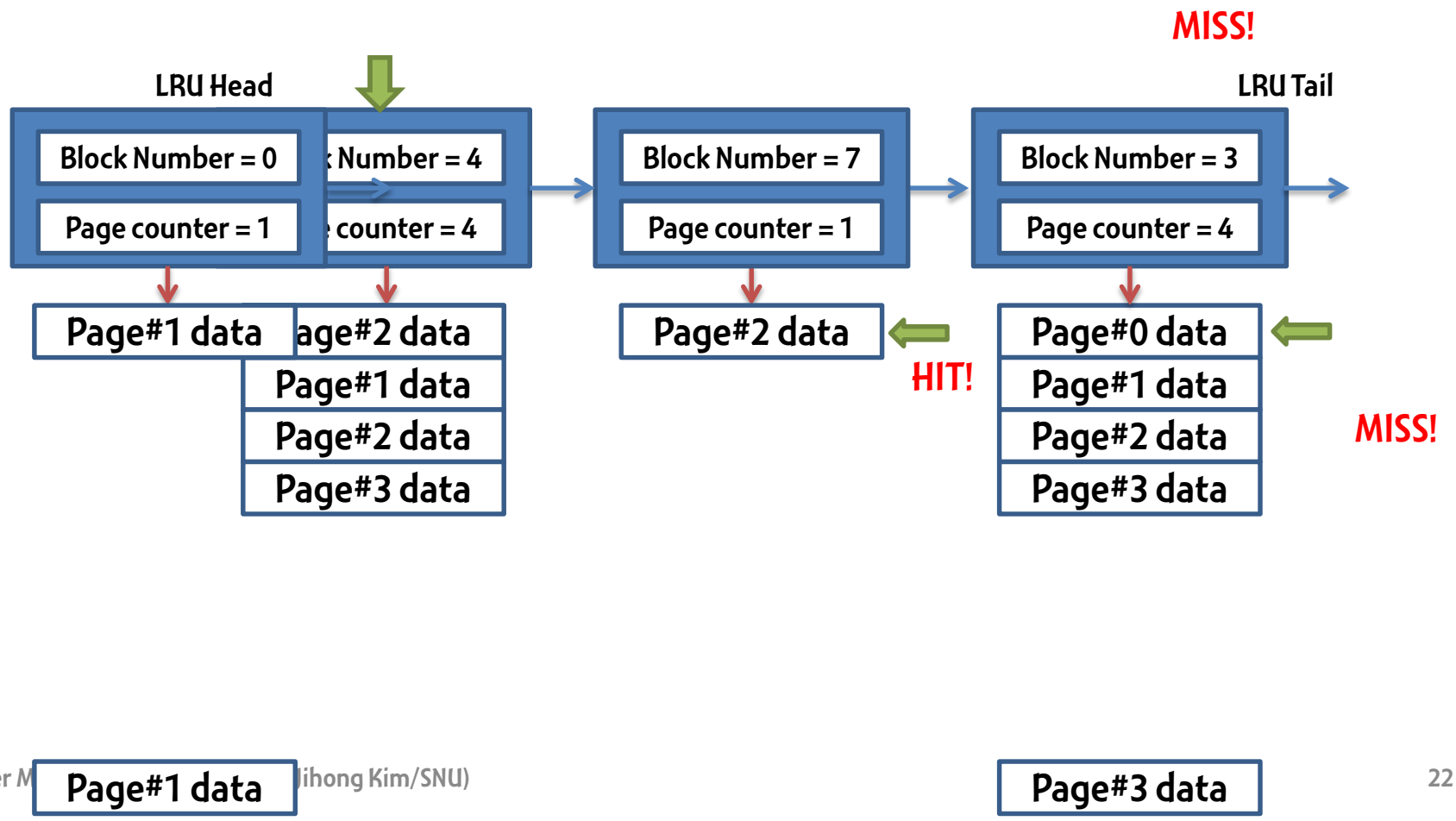
```
FAB_Write (block, page, data)
{
    if ((bufloc = Search_Buffer (block, page)) != null) {
        Write_Page (bufloc, data);
    }
    else {
        if (Buffer_Full ()) {
            victim  = Select_Victim_Block ();
            Flush_Victim_Block (victim);
        }
        bufloc = Allocate_New_Page ();
        Write_Page (bufloc, data);
    }
    Rearrange_Blocklist_For_LRU (block);

}
```

Fig. 2. Handling a write request in FAB

Handling a Read Request

Block Number = 0, Page Number = 1



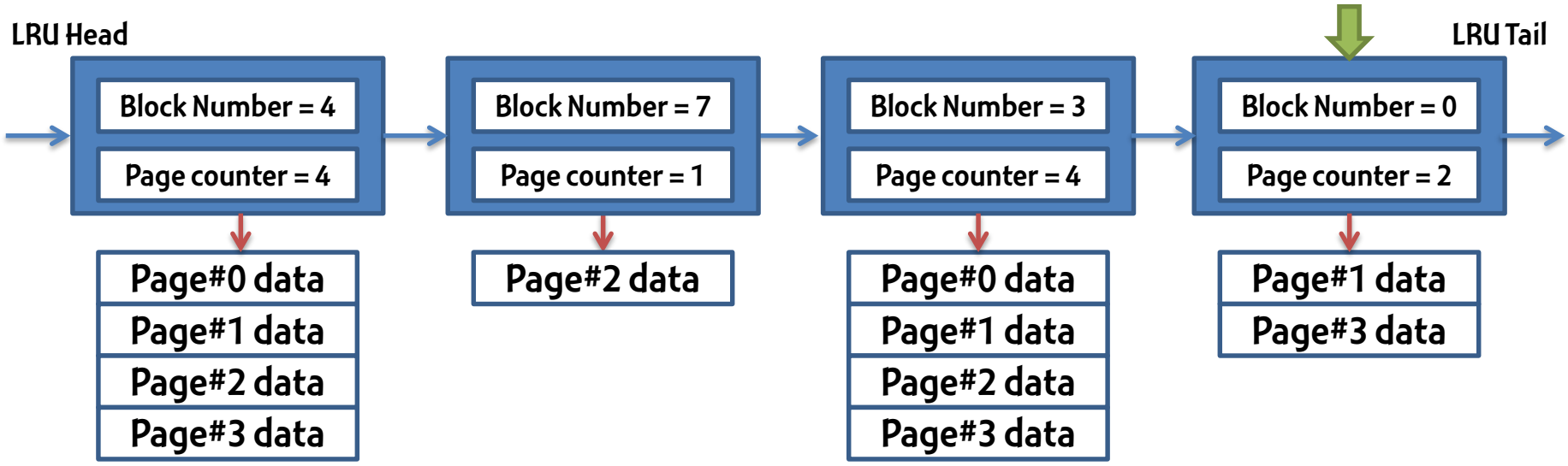
Handling a Read Request

```
FAB_Read (block, page, data)
{
    if ((bufloc = Search_Buffer (block, page)) != null) {
        Rearrange_Blocklist_For_LRU (block);
        Read_Page (bufloc, data);
    }
    else {
        if (Buffer_Full ()) {
            victim  = Select_Victim_Block ();
            Flush_Victim_Block (victim);
        }
        bufloc = Allocate_New_Page ();
        Read_From_Flash (bufloc, block, page);
        Read_Page (bufloc, data);
    }
    Rearrange_Blocklist_For_LRU (block);
}
```

Fig. 3. Handling a read request in FAB

Selecting a Victim Block

1. The block has the largest number of pages
2. If multiple blocks have the largest number of pages, FAB select the tail of the LRU list



Selecting a Victim Block

```
#define BlockPerPageNum 64
Select_Victim_Block ()
{
    VictimBlock = -1;
    MaxPageNum = 0;
    For each BlockNode from BlockNodeListTail to
    BlockNodeListHead
    {
        if (BlockNode.PageCount == BlockPerPageNum)
            return BlockNode.BlockNum;
        if (BlockNode.PageCount > MaxPageNum)
        {
            MaxPageNum = BlockNode.PageCount;
            VictimBlock = BlockNode.BlockNum;
        }
    }
    return VictimBlock;
}
```

Fig. 5. Selecting a victim block in FAB

Advantages of FAB

- Features
  - Because most PMP data are contiguous, most flash memory blocks are either full of valid pages or empty
  - Only several pages are updated frequently and some valid and invalid pages are mixed together
  - By choosing a full block as a victim, FAB chiefly evict data of the contents
- Advantages
  1. Because most victim blocks are full of valid pages, switch merges are needed.
  2. Hot pages are rarely evicted.

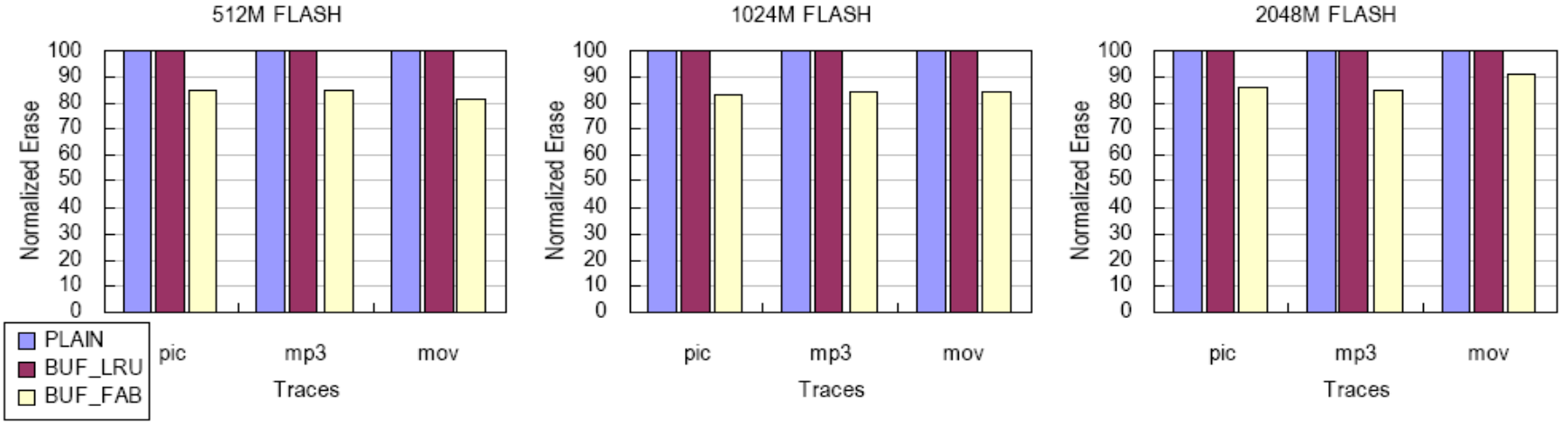
Evaluation Methodology

- Simulator is used
  - Determine parameters related to current technologies as exactly as possible
- FAST is adopted as an FTL scheme of flash memory
- The traces are extracted from disk access logs of real user activities on FAT32 file system

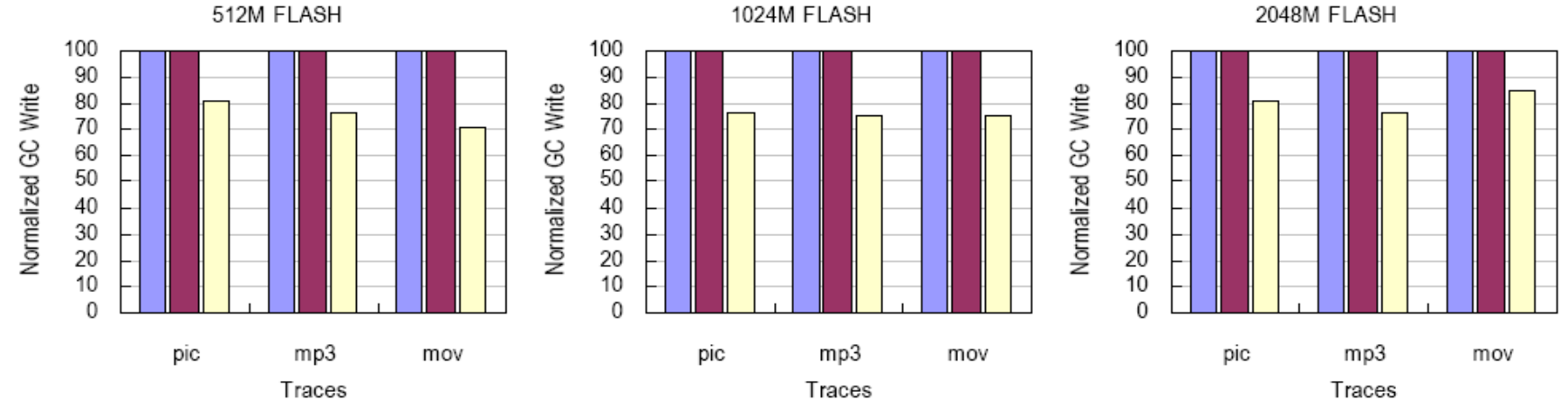
TABLE I
TRACES USED FOR SIMULATION

Trace	Description	The size of flash storage	The number of sectors written
Pic	The traces of digital camera. Picture files are about 1Mbytes.	512 MB	868,154
		1024 MB	1,803,561
		2048 MB	4,335,023
MP3	The traces of MP3 player. MP3 files are about 4-5Mbytes.	512 MB	1,146,234
		1024 MB	1,935,572
		2048 MB	5,602,743
Mov	The traces of movie player. Movie files are about 15-30Mbytes.	512 MB	1,067,905
		1024 MB	2,296,341
		2048 MB	7,196,759

Performance Evaluation



(a) 512 Mbytes (c) 1024 Mbytes (d) 2048 Mbytes



(b) 512 Mbytes (d) 1024 Mbytes (f) 2048 Mbytes

Fig. 6. The normalized number of write and erase operations of FAST, FAST+LRU buffer scheme, and FAST+FAB buffer scheme

Performance Evaluation(cont.)

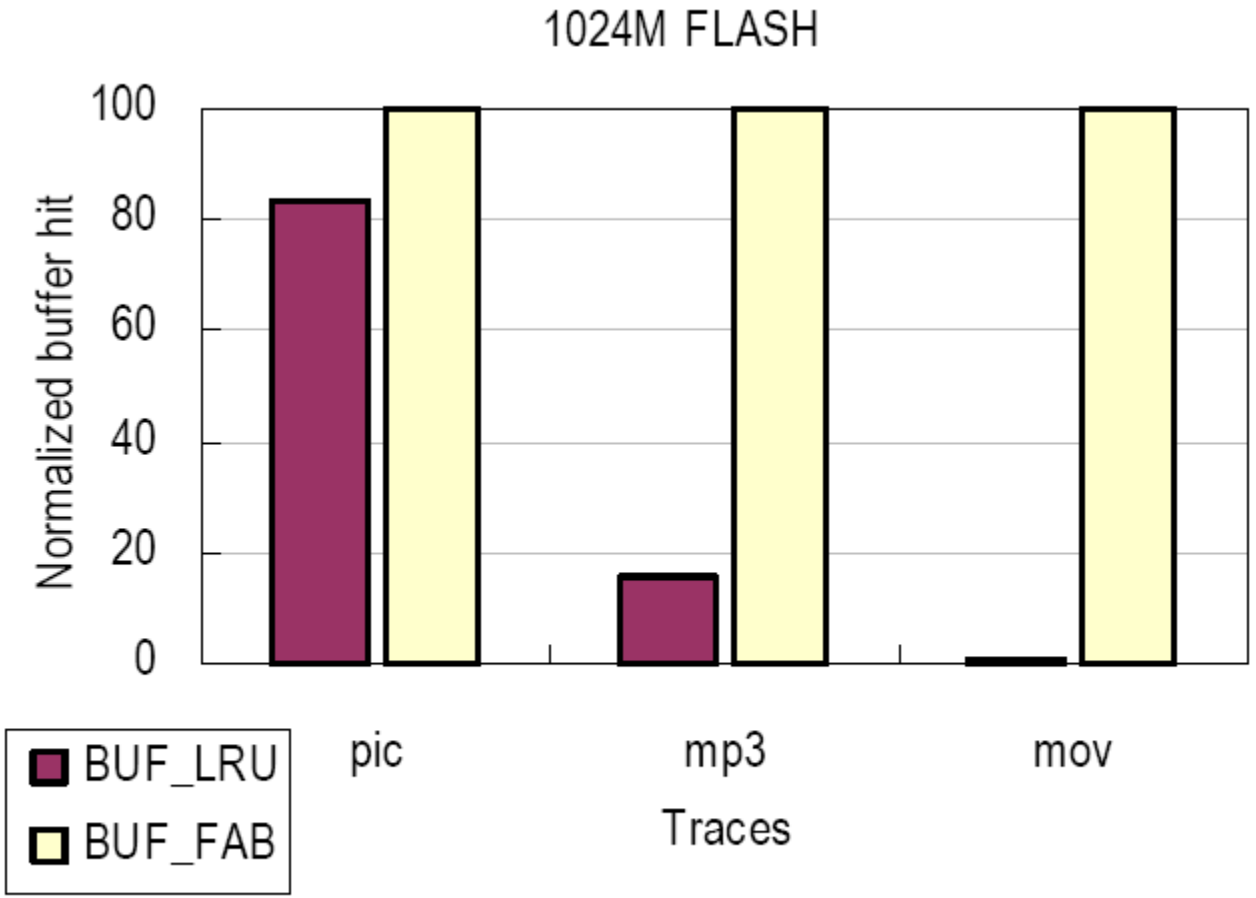


Fig. 8. The normalized number of buffer hits.

Problem of CFLRU & FAB

- All these algorithms perform well for sequential write but fail when it comes to random writes
- To use the flash memories as SSDs, we need a good performance for random writes
  - Two approaches are possible
    - Use a good **flexible mapping algorithm** in the FTL.
      - However, this might require more RAM, CPU power, mount time, etc.
    - Develop a **RAM buffer** in the SSD

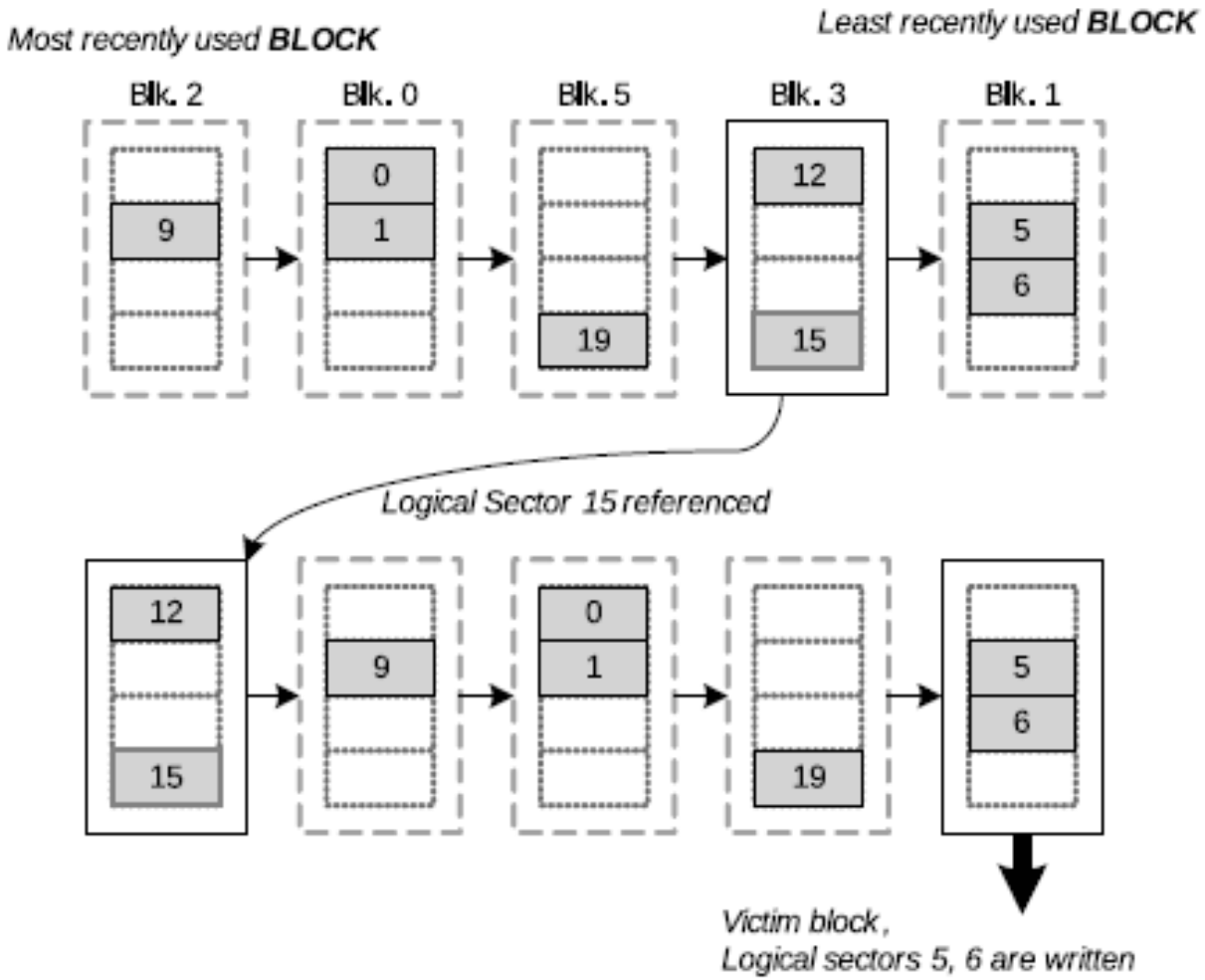
Buffer Padding LRU (BPLRU)

- Target System
  - flash memory-based SSDs for **random writes**
  - Uses the entire RAM available inside the SSD for write requests.
- BPLRU uses three techniques to improve the efficiency:
  - **Block-level LRU**
  - **Page padding**
  - **LRU Compensation**

Block-level LRU

- The LRU list is managed in **units of blocks**
- This minimizes the number of merges and hence the cost if the writes are random.
- This is because if the writes are random, cache hit ratio is very low and LRU just acts similar to FIFO.

Block-level LRU



Improvement in Performance

Sector Writes	LRU		Block-level LRU	
	Cache status (8 sectors)	Log block (2 blocks)	Cache status (8 sectors)	Log block (2 blocks)
0, 4, 8, 12, 16	**16, 12, 8, 4, 0**		**[16], [12], [8], [4], [0]**	
1, 5, 9	**9, 5, 1**, 16, 12, 8, 4, 0		**[8, 9], [4, 5], [0, 1]**, [12], [16]	
13	**13**, 9, 5, 1, 16, 12, 8, 4 → 0	[0]	**[12, 13]**, [8, 9], [4, 5], [0, 1] → 16	[16]
17	**17**, 13, 9, 5, 1, 16, 12, 8 → 4	[4], [0]	**[17]**, [12, 13], [8, 9], [4, 5], → 0, 1	[0, 1], [16]
2	**2**, 17, 13, 9, 5, 1, 16, 12 → 8	[8], [4] → M[0]	**[2]**, [17], [12, 13], [8, 9], [4, 5]	
6	**6**, 2, 17, 13, 9, 5, 1, 16 → 12	[12], [8] → M[4]	**[6]**, [2], [17], [12, 13], [8, 9] → 4, 5	[4,5], [0, 1] → M[16]
10	**10**, 6, 2, 17, 13, 9, 5, 1 → 16	[16], [12] → M[8]	[8, 9, **10]**, [6], [2], [17], [12, 13]	
14	**14**, 10, 6, 2, 17, 13, 9, 5 → 1	[1] , [16]→ M[12]	**[14]**, [8, 9, 10], [6], [2], [17] → 12, 13	[12, 13], [4, 5]→ M[0, 1]
Buffer flushing by LRU order	14, 10, 6, 2, 17, 13, 9 → 5	[5] , [1]→ M[16]	[14], [8, 9, 10], [6], [2] → 17	[17], [12, 13]→ M[4, 5]
	14, 10, 6, 2, 17, 13 → 9	[9], [5] → M[1]	[14], [8, 9, 10], [6] → 2	[2], [17]→ M[12, 13]
	14, 10, 6, 2, 17 → 13	[13], [9]→ M[5]	[14], [8, 9, 10] → 6	[6], [2]→ M[17]
	14, 10, 6, 2 → 17	[17], [13]→ M[9]	[14] → 8, 9, 10	[8, 9, 10], [6]→ M[2]
	14, 10, 6 → 2	[2], [17]→ M[13]	→ 14	[14], [8, 9, 10]→ M[6]
	14, 10 → 6	[6], [2]→ M[17]		
	14 → 10	[10], [6]→ M[2]		
	→ 14	[14], [10]→ M[6]		
	Total merge count	**12**		**7**

Note: [], →, M mean a block boundary, flushing, merge operation in FTL, respectively

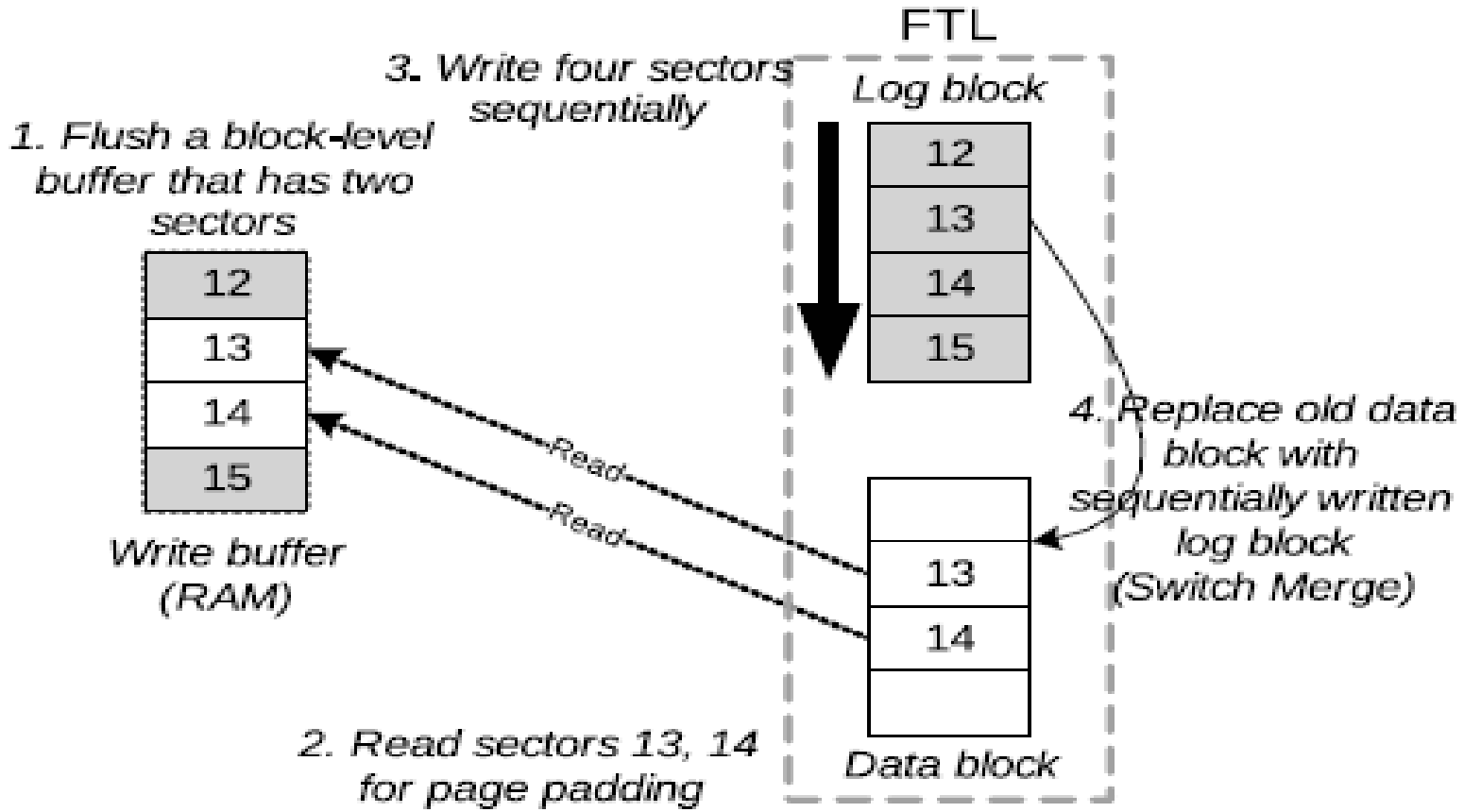
Page Padding

- The key idea is to **reduce the number of full merges.**
- A slight modification of log-block FTL
  - **Reads some pages that are not in a victim block, and writes all sectors in the block range sequentially.**

  => Full Merge -> Switch merge! + extra copies

    - Page padding may seem to perform unnecessary reads and writes, but it is more effective because it can change an expensive full merge to an efficient switch merge.

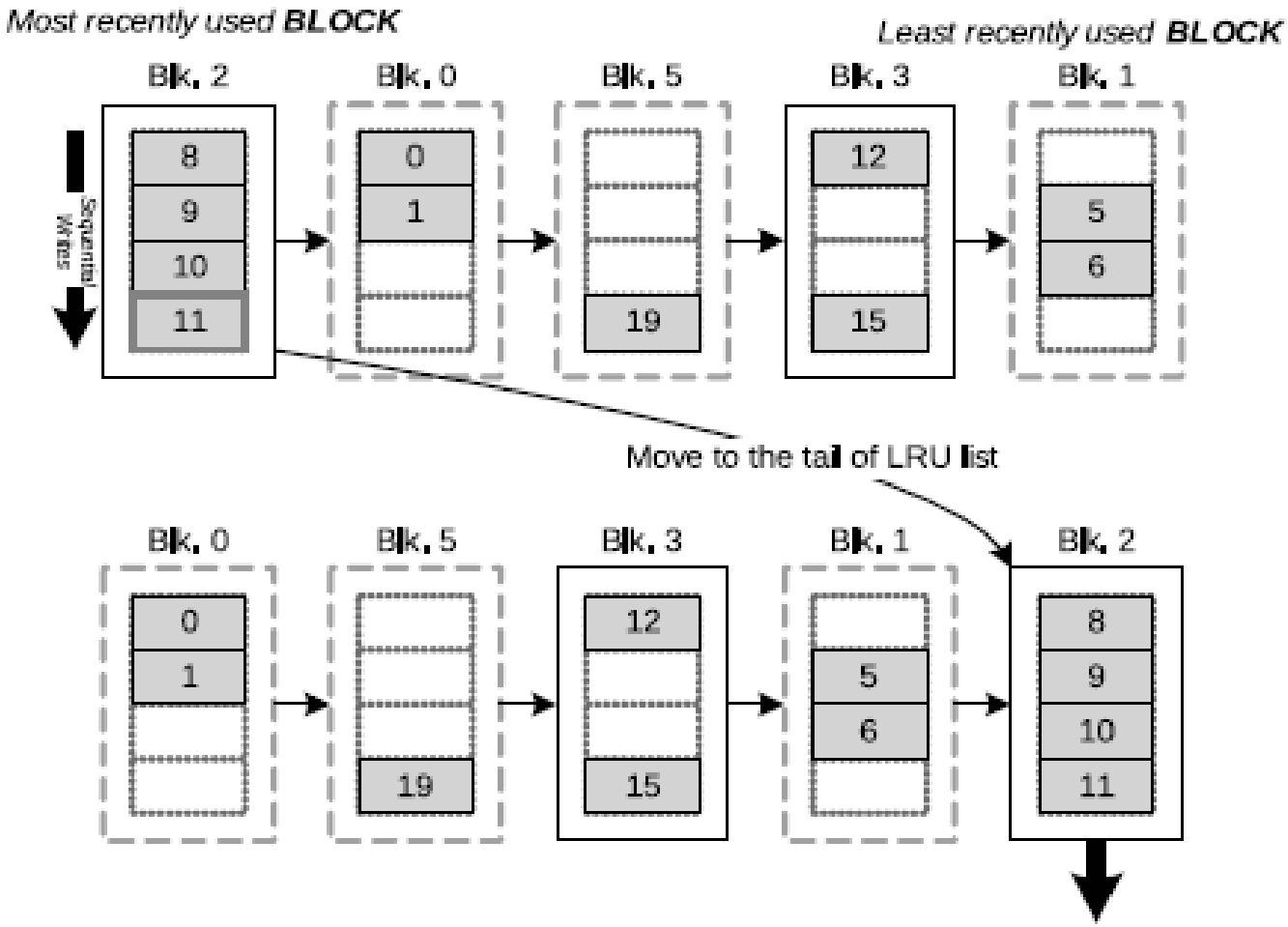
Example of Page Padding



LRU Compensation

- The BPLRU checks the most recently used block consistently
- If the data in it are written sequentially, then it implies a **sequential write**
- This implies that this block **will not be rewritten in the near future**.
  - Hence it is **moved to become the LRU block**
  - Once the cache becomes full, this block becomes the first victim

LRU Compensation

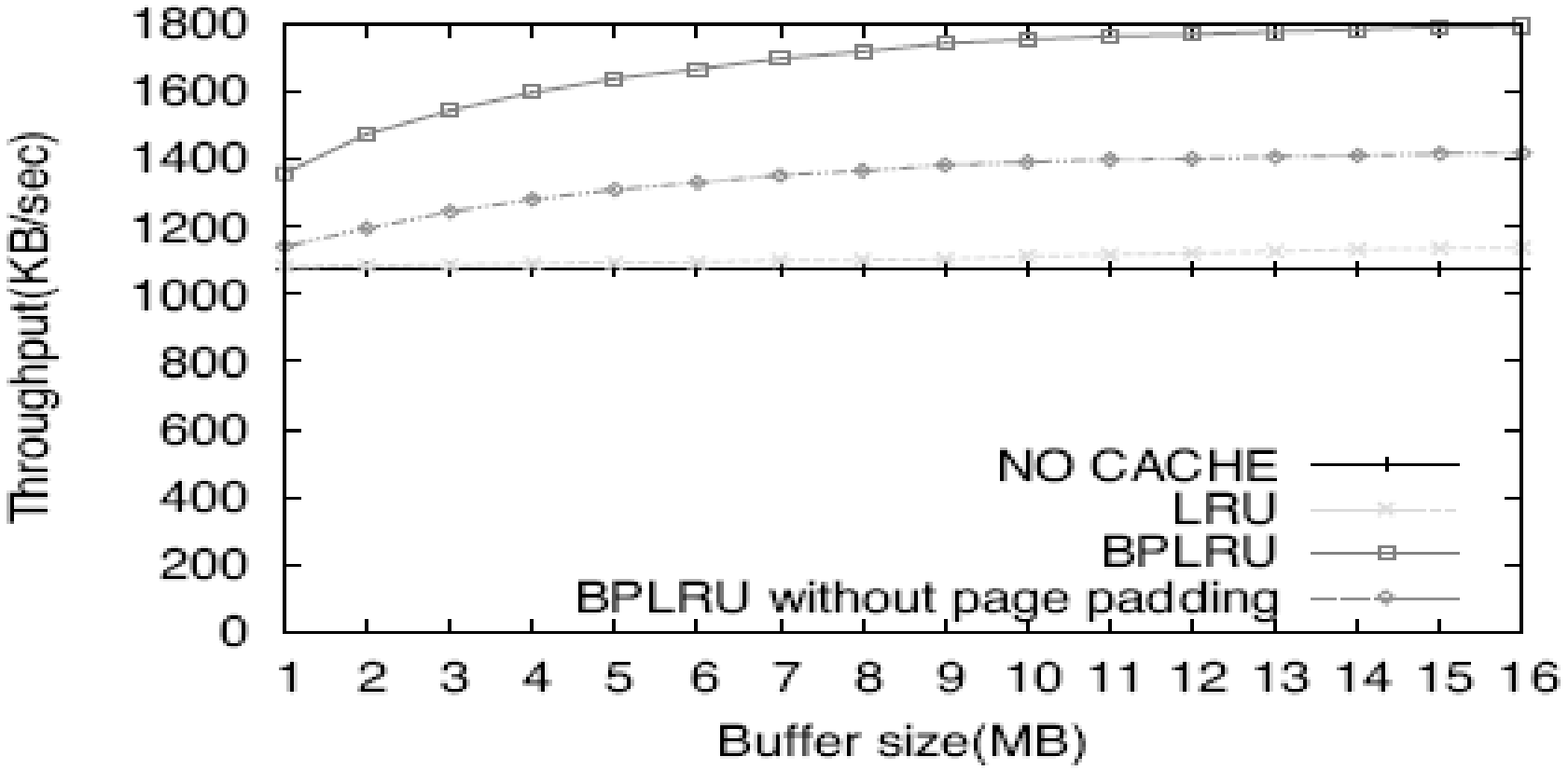


Evaluation

- Used both simulation and experiments on a real hardware prototype to compare four cases:
  - no RAM buffer, LRU, FAB, and the BPLRU
- The graphs are given for the three types of file systems namely **NTFS**,**FAT16** and **EXT3**.

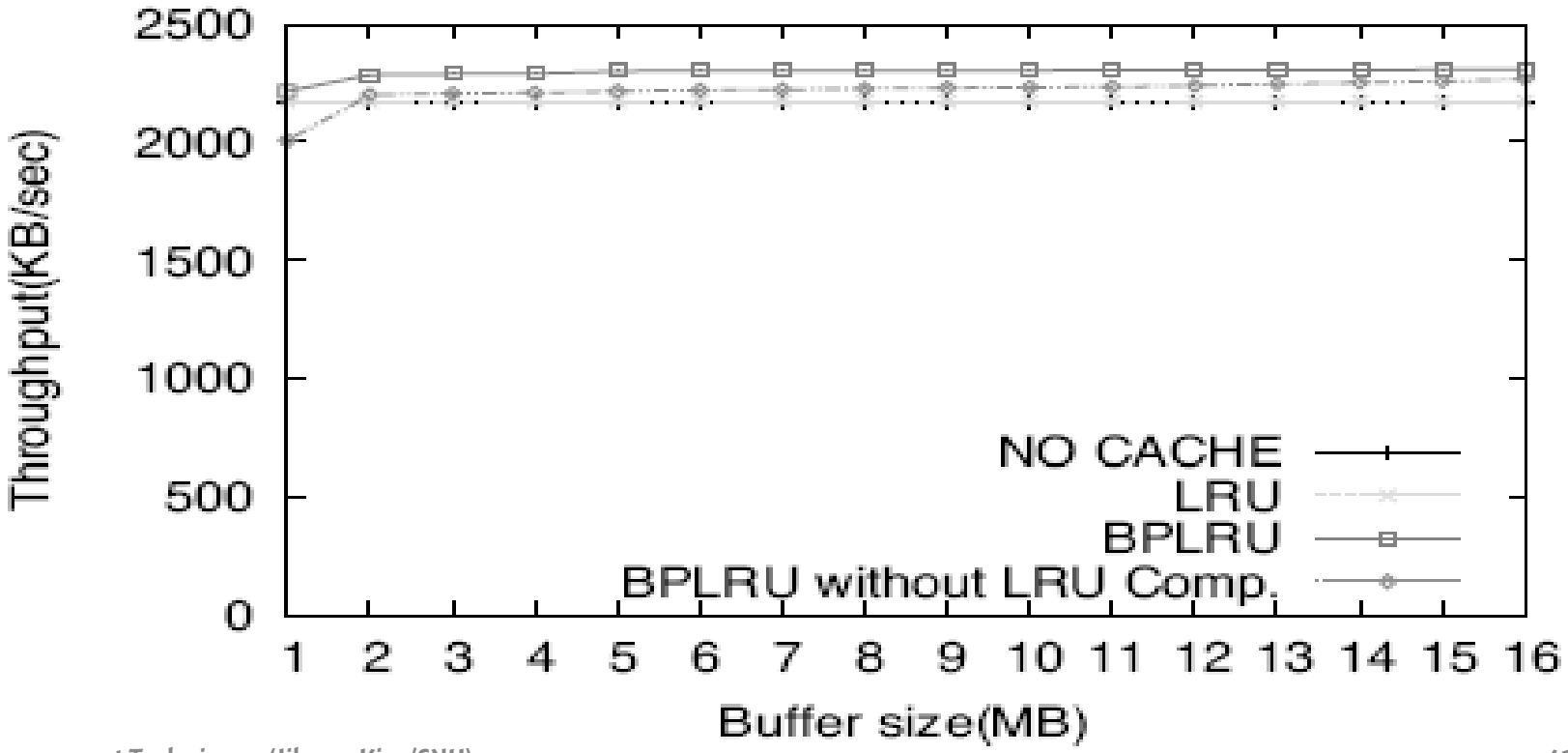
Page Padding Effect

- **Benchmark: MS office 2003 installation on NTFS**



LRU Compensation effect

- This simulation was done for copying MP3 files on FAT16



References

- H. Jo et. al., "FAB: flash-aware buffer management policy for portable media players," IEEE Trans. on Consumer Electronics, vol. 52, no. 2, pp. 485-493, 2006.
- H. Kim et. al., "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," Proceedings of the 6th USENIX Conference on File and Storage Technologies, 2008.
- S. Park et. al., "CFLRU: A Replacement Algorithm for Flash Memory," Proceedings of the international conference on Compilers, architecture and synthesis for embedded systems, 2006.