

Hybrid Mapping-based Flash Translation Layer

Jihong Kim

Dept. of CSE, SNU

Outline

- Problem of BAST
- Advanced Hybrid-mapping schemes
 - FAST
 - LAST

FAST

Problems of BAST

- Log-block thrashing
 - Not enough to cover the write requests

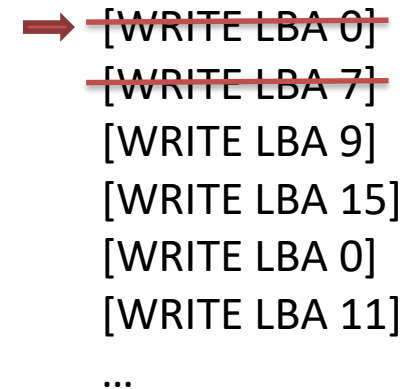
Data Block



Log Block



Requests



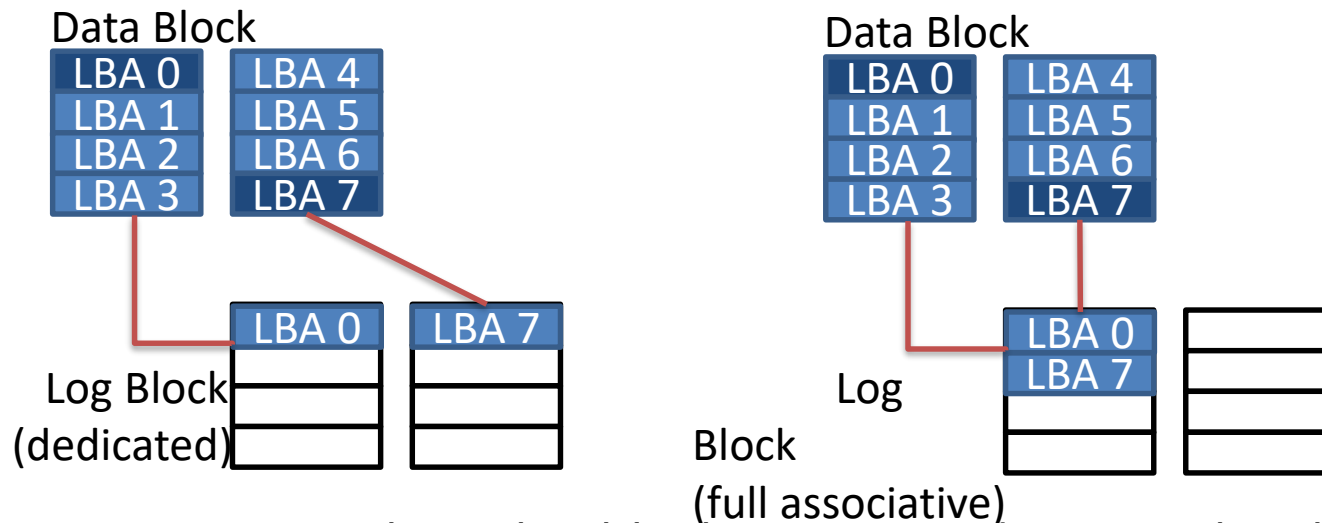
Garbage Collection is triggered!

Challenges of BAST

- Frequent merge operation
 - In random write patterns
 - In complicated application

FAST: Fully Associative S. T.

- FAST : Fully Associative Sector Translation
- Key idea
 - Fully associative mapping between data blocks and log blocks



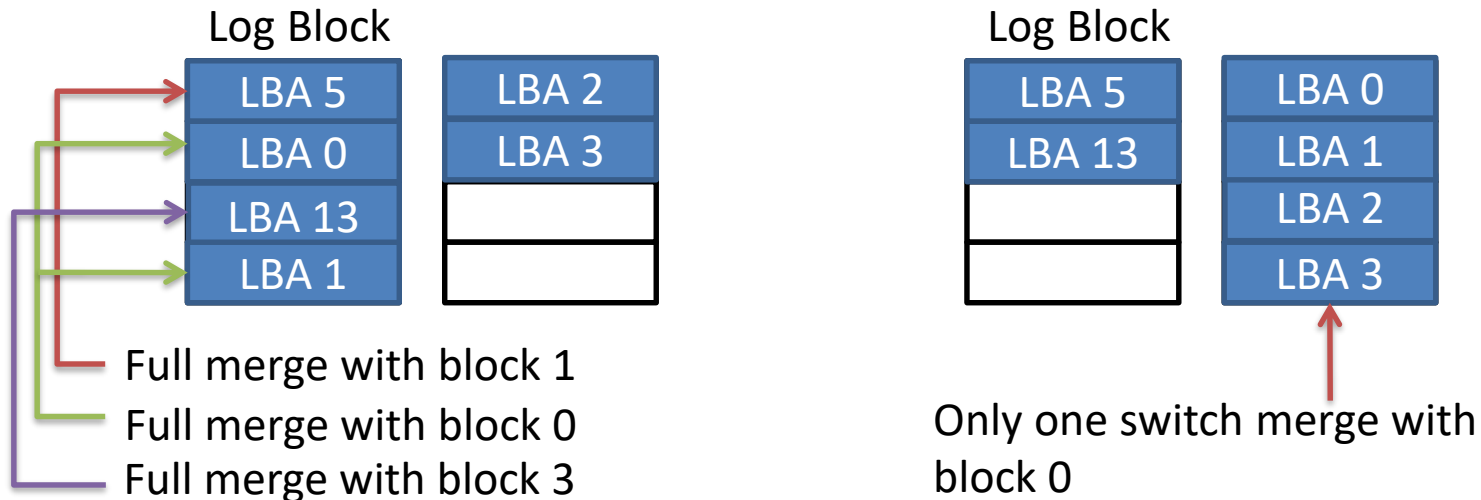
- Mapping within a log block is managed in page-level as in log block scheme

FAST: Pros and Cons

- Pros
 - Higher utilization of log blocks
 - Delayed merge operation
 - increases the probability of page invalidation
- Cons
 - When GC, excessive overhead for a single log block reclamation
 - Severely skewed performance depending on the number of data blocks involved in a log block

FAST: Sequential Log Block

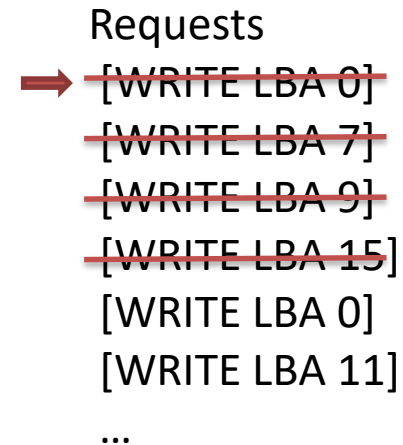
- Increase the number of switch operations
 - Which one is the better option?



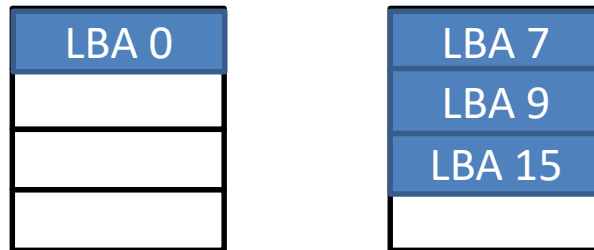
- Insert a page in the sequential log block if the offset is '0'
- Merge sequential log block if there is no empty one or the sequentiality is broken

FAST: Example

- Example scenario same as before



Log Block



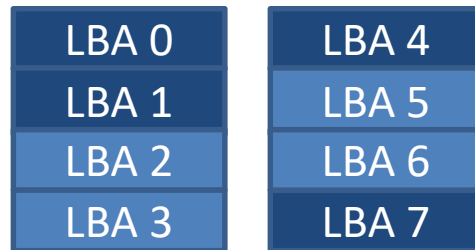
Sequential Log

Block

Merge Operation in FAST

- In the garbage collection to get a free page
 - When a log block is the victim block, the number of merge operations is same as the number of associated data blocks.

Data Block



Log Block

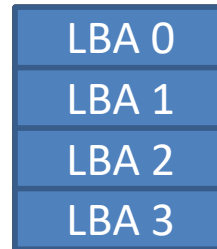
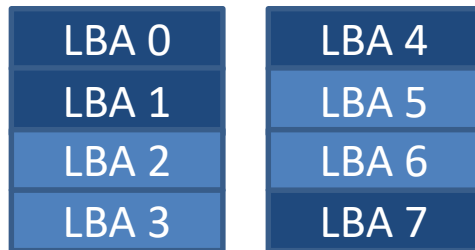


Victim Log Block

O-FAST(Optimized FAST)

- To delay / skip unnecessary merge operations
 - If the data of pages in current victim log block is invalid, skip the merge operations for the pages.

Data Block



Log Block

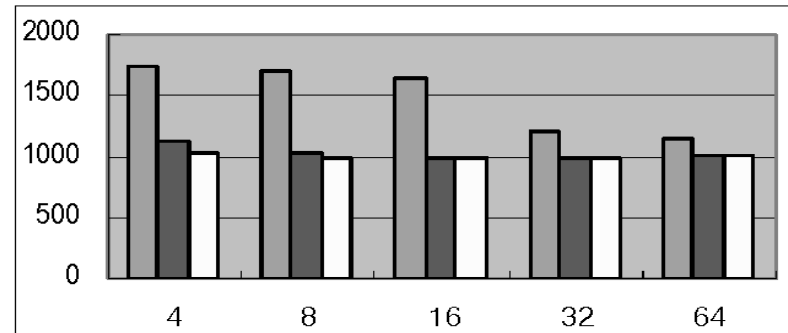
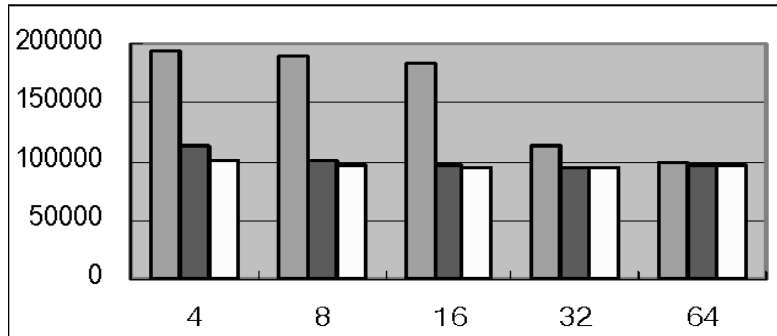


Victim Log Block

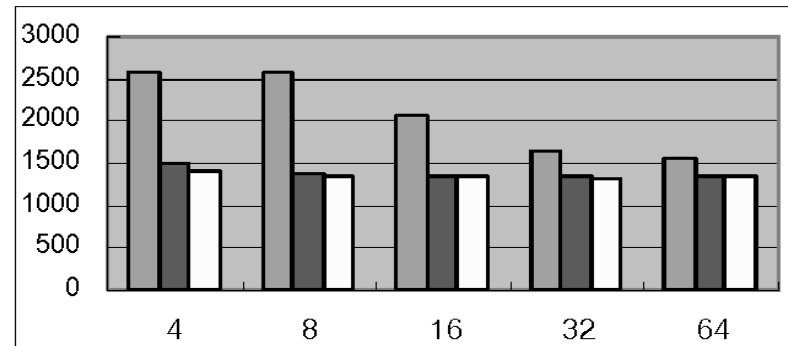
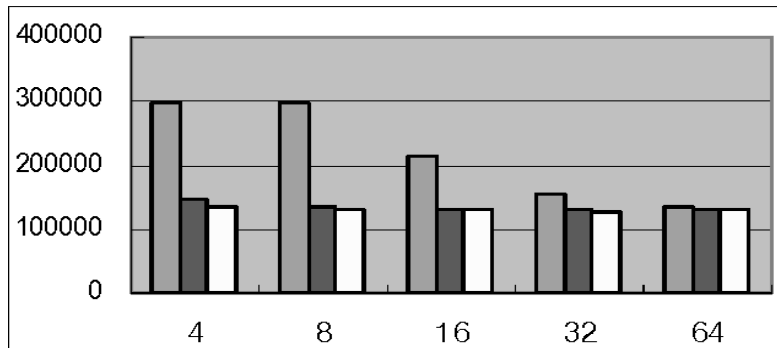
Experimental Result

- Performance metrics
 - Number of total erase count
 - Total elapsed time
- Benchmark characteristic
 - Patterns A and B (Digital Camera)
 - Small random writes and large sequential writes
 - Patterns C and D (Linux and Symbian)
 - Many small random writes and small large sequential write
 - Pattern E (Random)
 - Uniform random writes

Experimental Result



(a) Pattern A: Digital Camera(Company A)

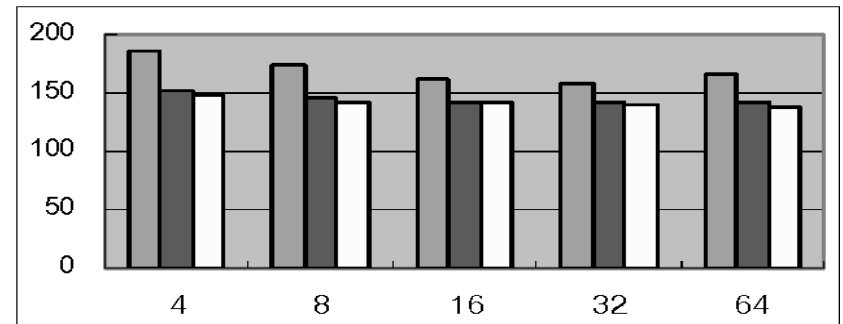
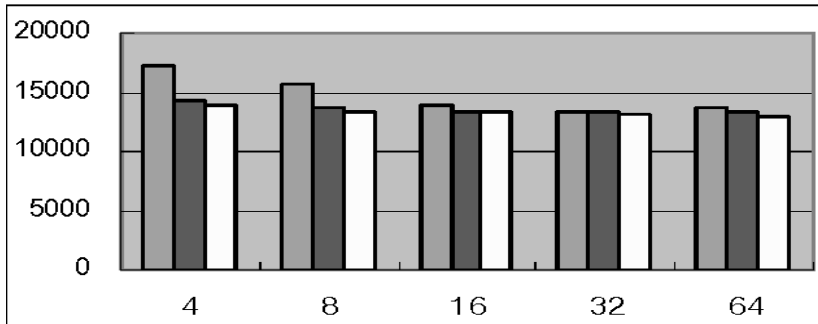


(b) Pattern B: Digital Camera(Company B)

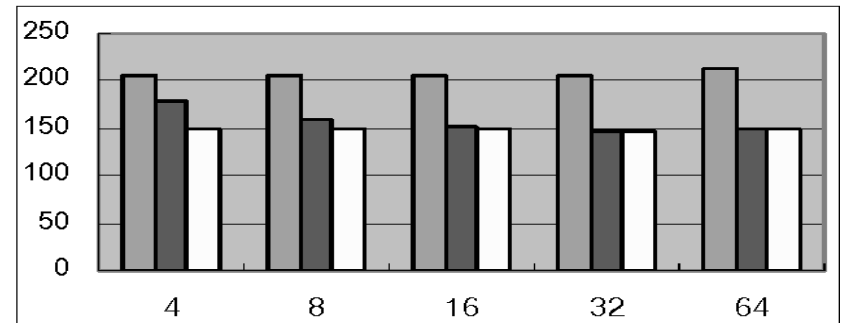
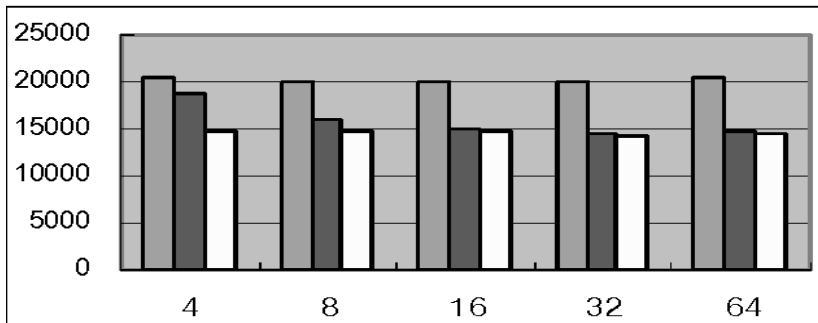
BAST
 FAST
 O-FAST

X-axis : # of log blocks, Y-axis in left side : erase count, Y-axis in right side : elapsed time(secs).

Experimental Result



(c) Pattern C: Linux

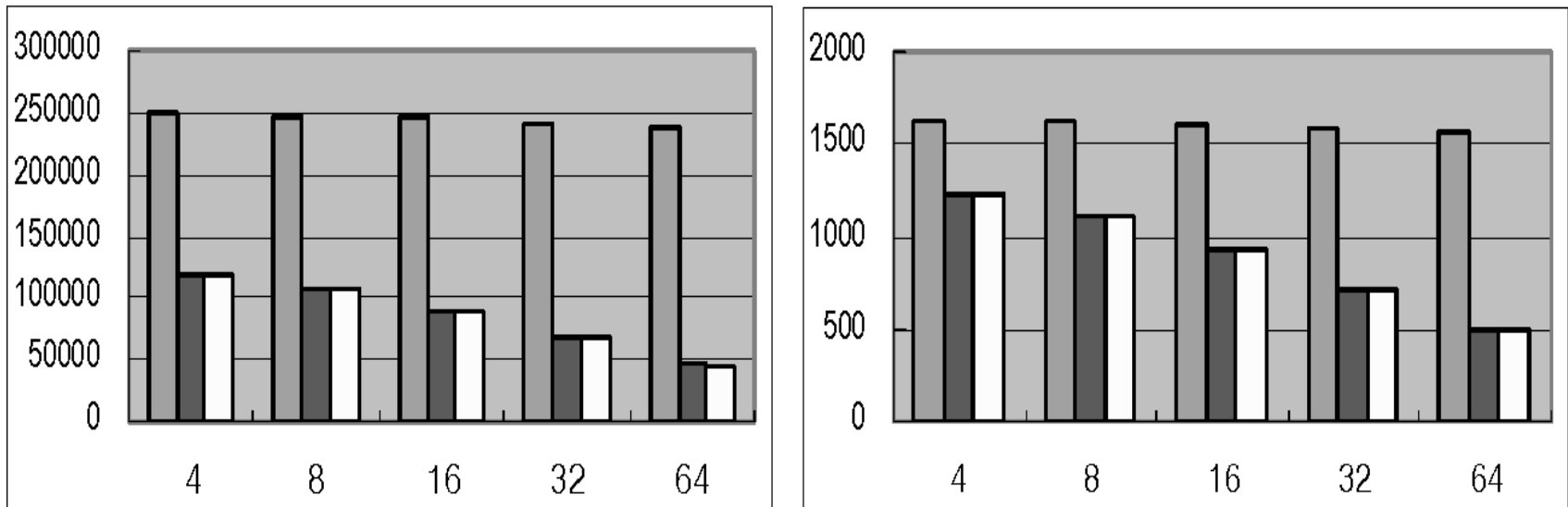


(d) Pattern D: Symbian

BAST
 FAST
 O-FAST

X-axis : # of log blocks, Y-axis in left side : erase count, Y-axis in right side : elapsed time(secs).

Experimental Result



(e) Pattern E: Random

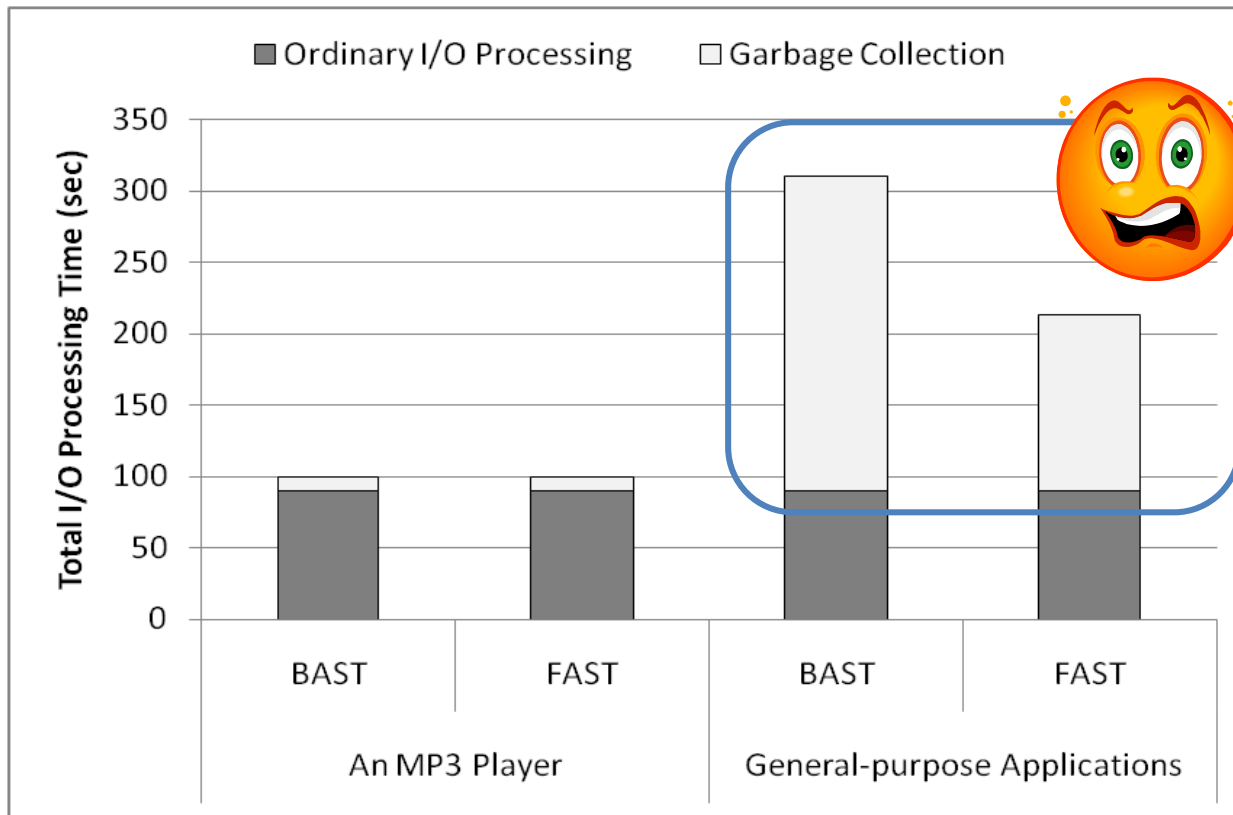
■ BAST ■ FAST □ O-FAST

X-axis : # of log blocks, Y-axis in left side : erase count, Y-axis in right side : elapsed time(secs).

LAST

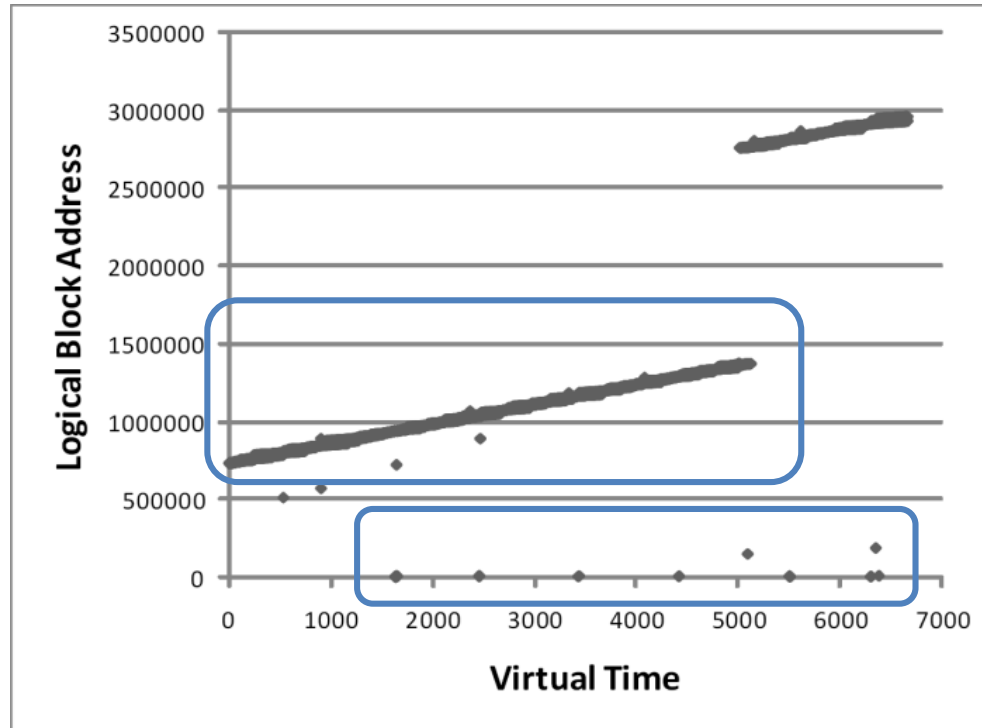
FTL in General-Purpose Computing Systems

- Existing FTL schemes are **ill-suited for general-purpose computing systems**



Garbage collection overhead is significantly increased !!!

I/O Characteristics of Mobile Embedded Applications

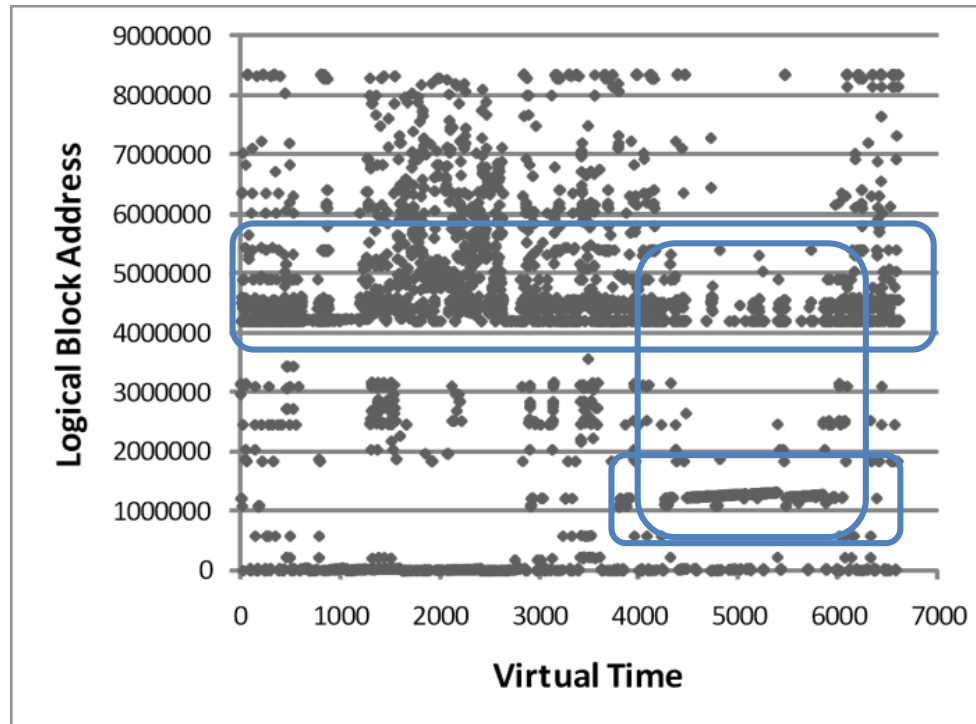


An MP3 player

- Most of write requests are *sequential*
- Many merge operations can be performed *by cheap switch merge*

⇒ A little garbage collection overhead

I/O Characteristics of General-purpose Applications

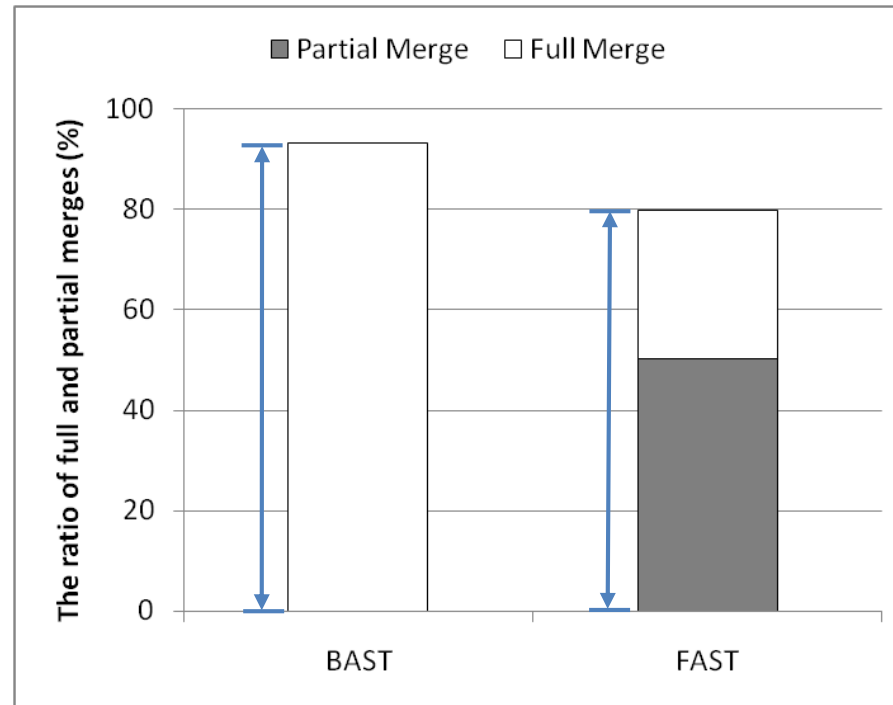


General-purpose applications

- Many random writes with a high temporal locality
- Many sequential writes with a high sequential locality
- A mixture of random and sequential writes

The increased full and partial merge operations

- The ratio of *expensive full* and *partial merges* is significantly increased !!!

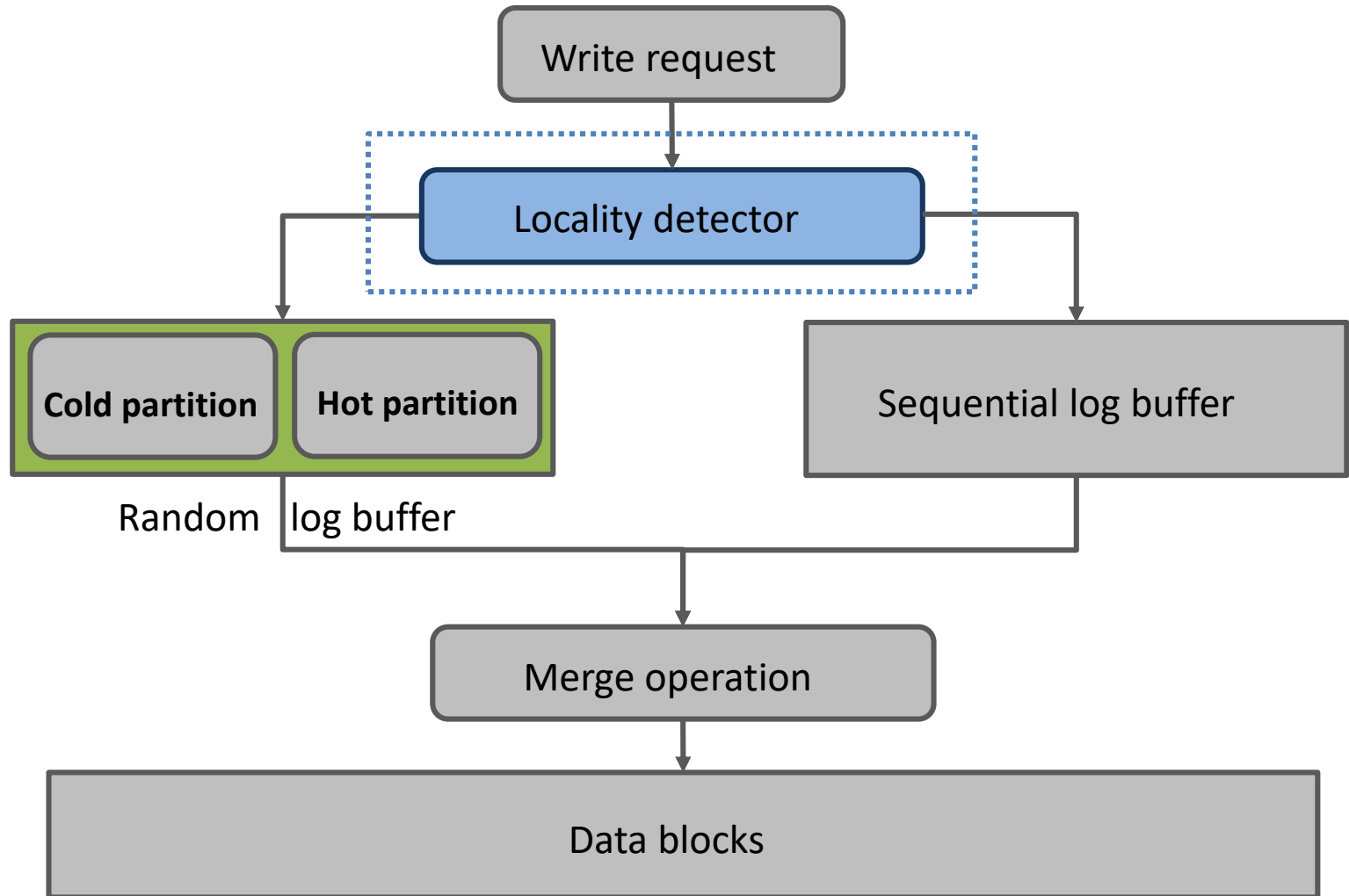


⇒ Need to take advantage of the I/O characteristics of general-purpose applications

Locality-Aware Sector Translation (LAST)

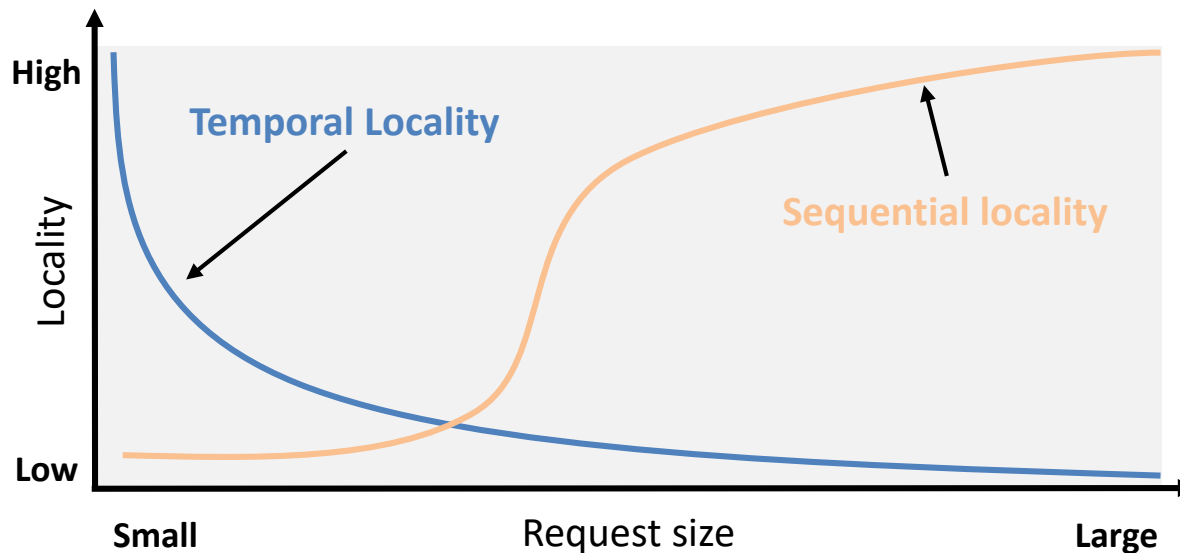
- Design goals of the LAST scheme
 - Replace *expensive full merges* by *cheap switch merges*
 - Reduce the average cost of *full merge*
- Our solutions
 - Extract a write request having **a high sequential locality from the mixed write patterns**
 - a locality detector
 - Exploit **a high temporal locality** of a random write
 - a hot/cold separation policy
 - an intelligent victim selection policy

Overall Architecture of the LAST Scheme



Locality Detector (1)

- How to detect the locality type of a write request
 - The locality type is highly correlated to the size of write request

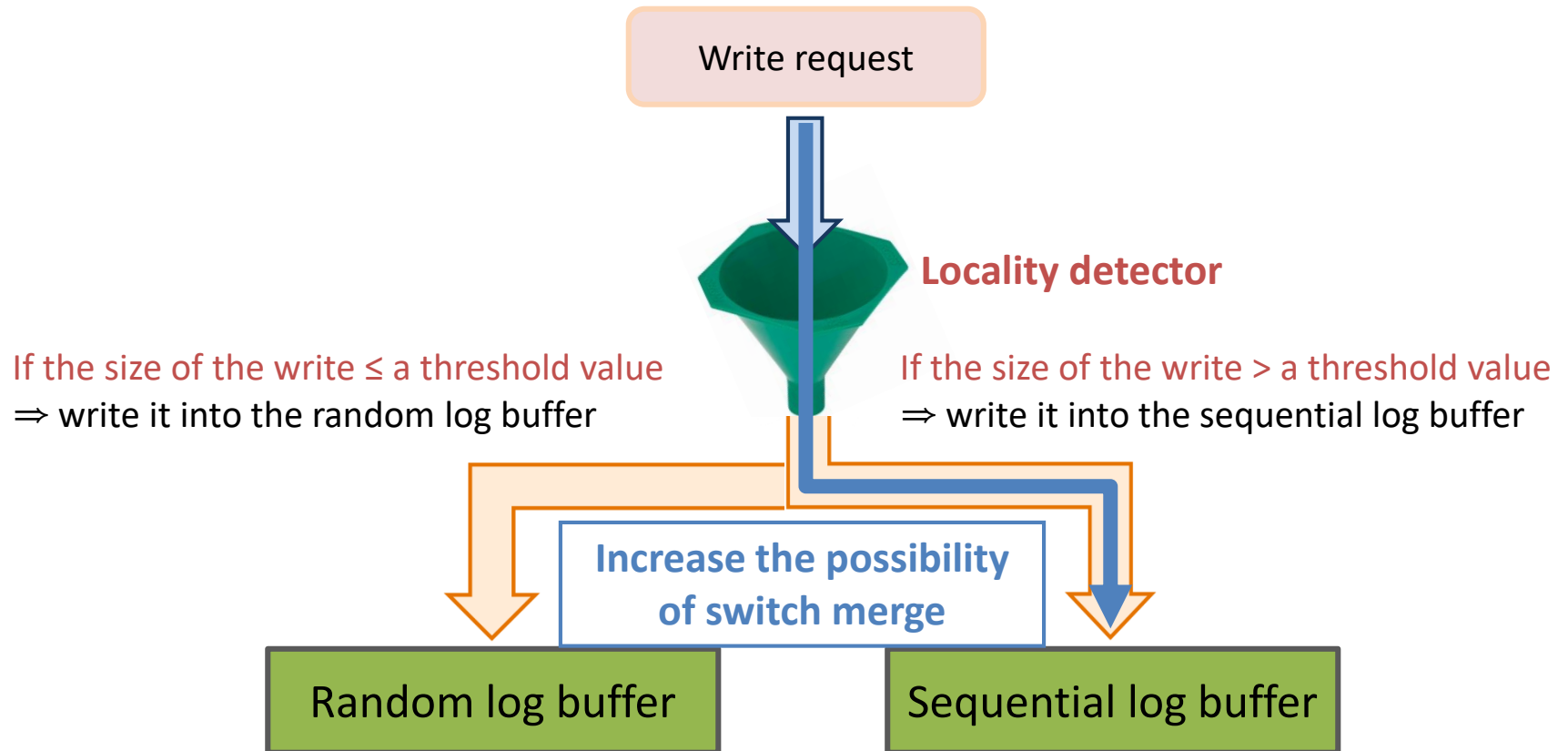


From the observation of realistic workloads

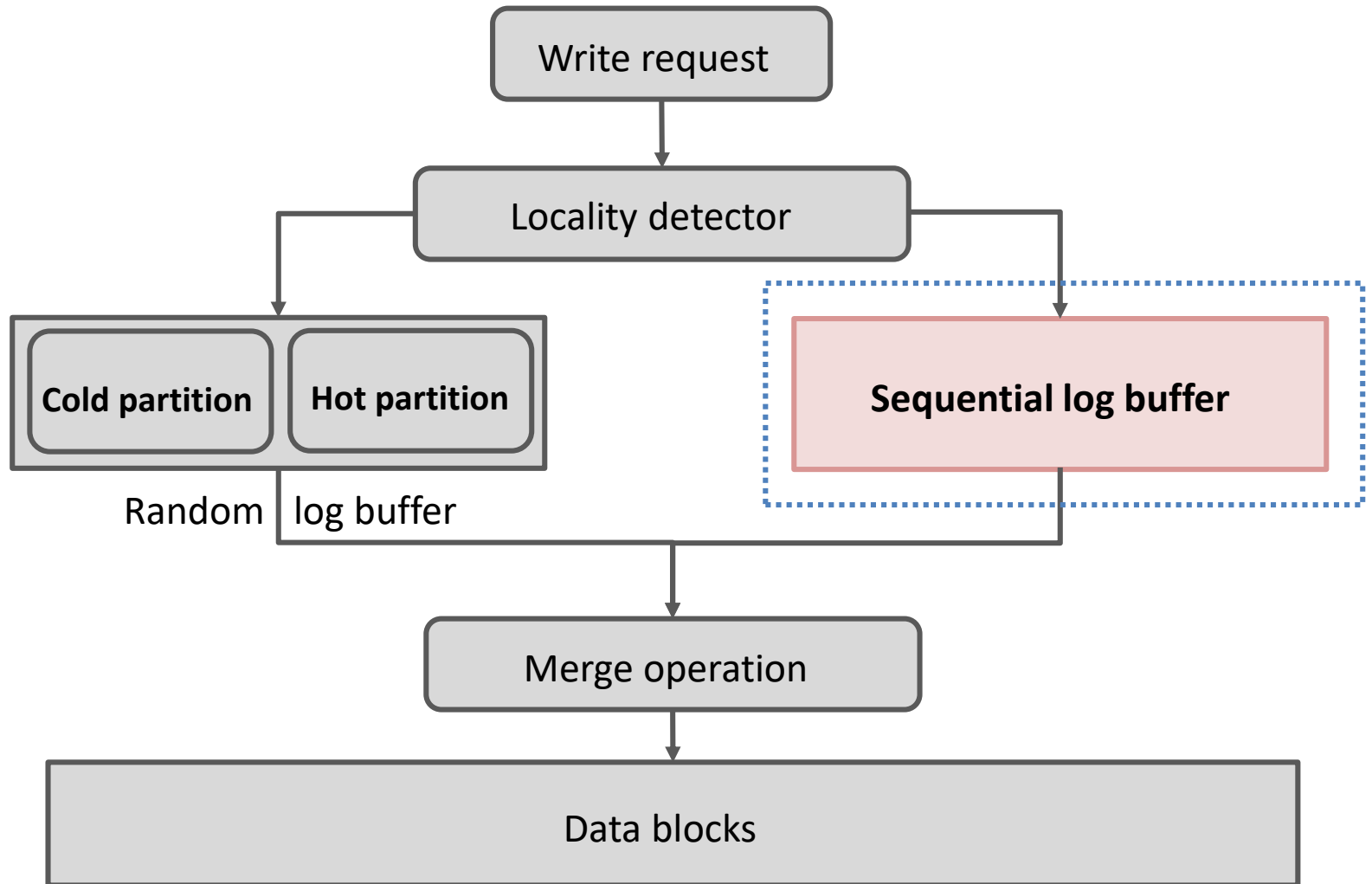
- small-sized writes have a high temporal locality
- large-sized writes have a high sequential locality

Locality Detector (2)

- A locality-detection policy based on the request size



Overall Architecture of the LAST Scheme



Sequential Log Buffer

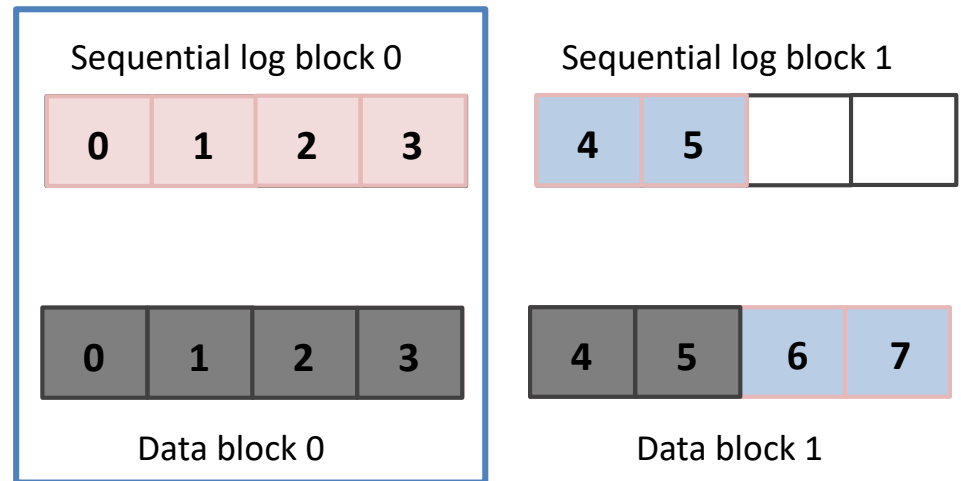
- Multiple sequential write streams are simultaneously issued from the file system
 - Accommodate multiple sequential write streams
 - maintain several log blocks in the sequential log buffer
 - Distribute each sequential write into different log block
 - one log block can be associated with only one data block

Write **stream 1** (page 0 and 1)

Write **stream 2** (page 4 and 5)

Write **stream 1** (page 2 and 3)

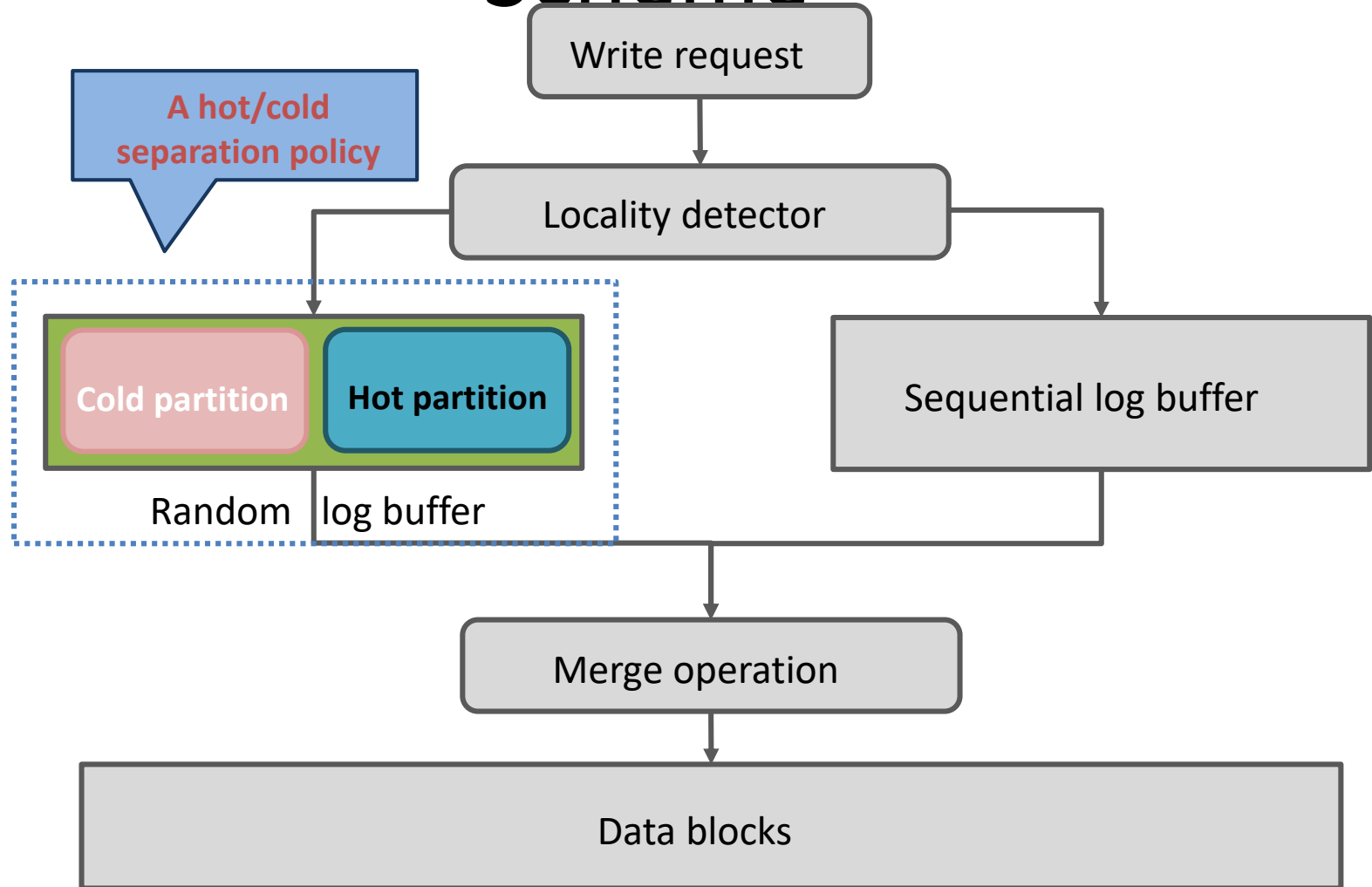
Write **stream 3** (page 8 and 9)



Switch merge

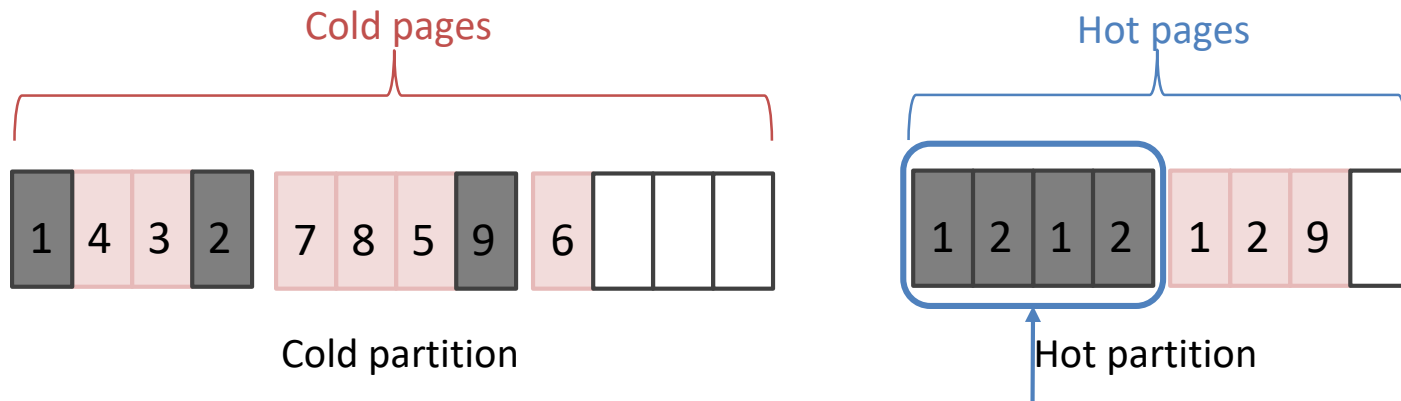
Overall Architecture of the LAST

Scheme



Log Buffer Partitioning Policy

- **Log buffer partitioning policy**
 - Proposed to provide a **hot and cold separation** policy
 - Separate hot pages from cold pages
 - Invalid pages are likely to be clustered in the same log block
 - All the pages in a log block can be invalidated \Rightarrow **dead block**
 - Remove **dead block** with **only one erase operation**



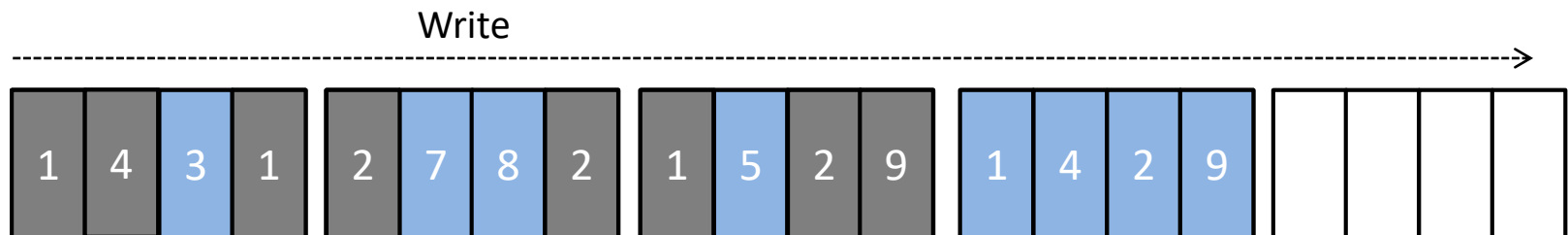
Many **dead blocks** are generated

Log Buffer Partitioning Policy

- A single partition
 - All the requested pages are sequentially written to log blocks

Requested pages:

1 → 4 → 3 → 1 → 2 → 7 → 8 → 2 → 1 → 5 → 2 → 9 → 1 → 4 → 2 → 9



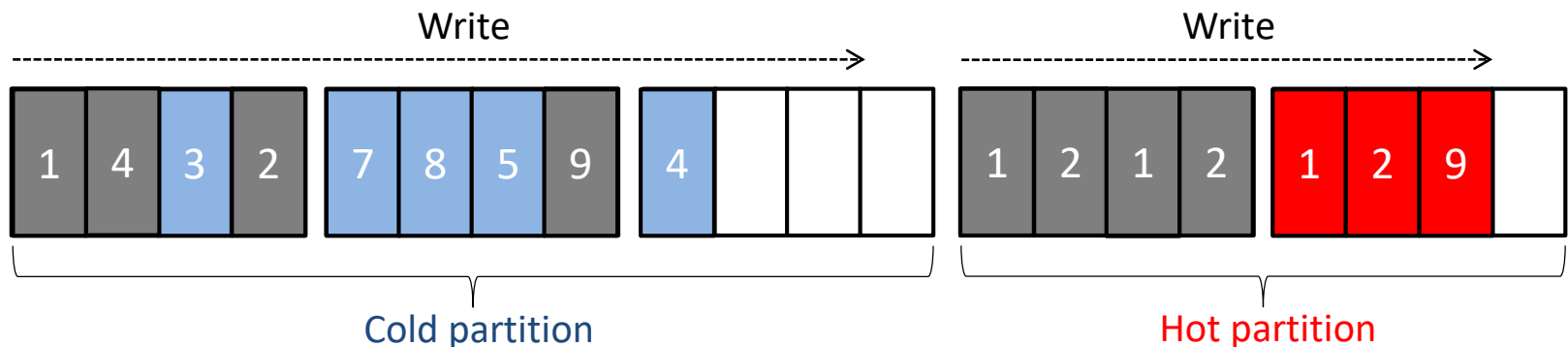
A single partition

Log Buffer Partitioning Policy

- Two partitions
 - The requested page is written to a different partition depending on its locality
 - If the requested page is one of k pages recently written, we regard it as a hot page; otherwise, it is regarded as a cold page

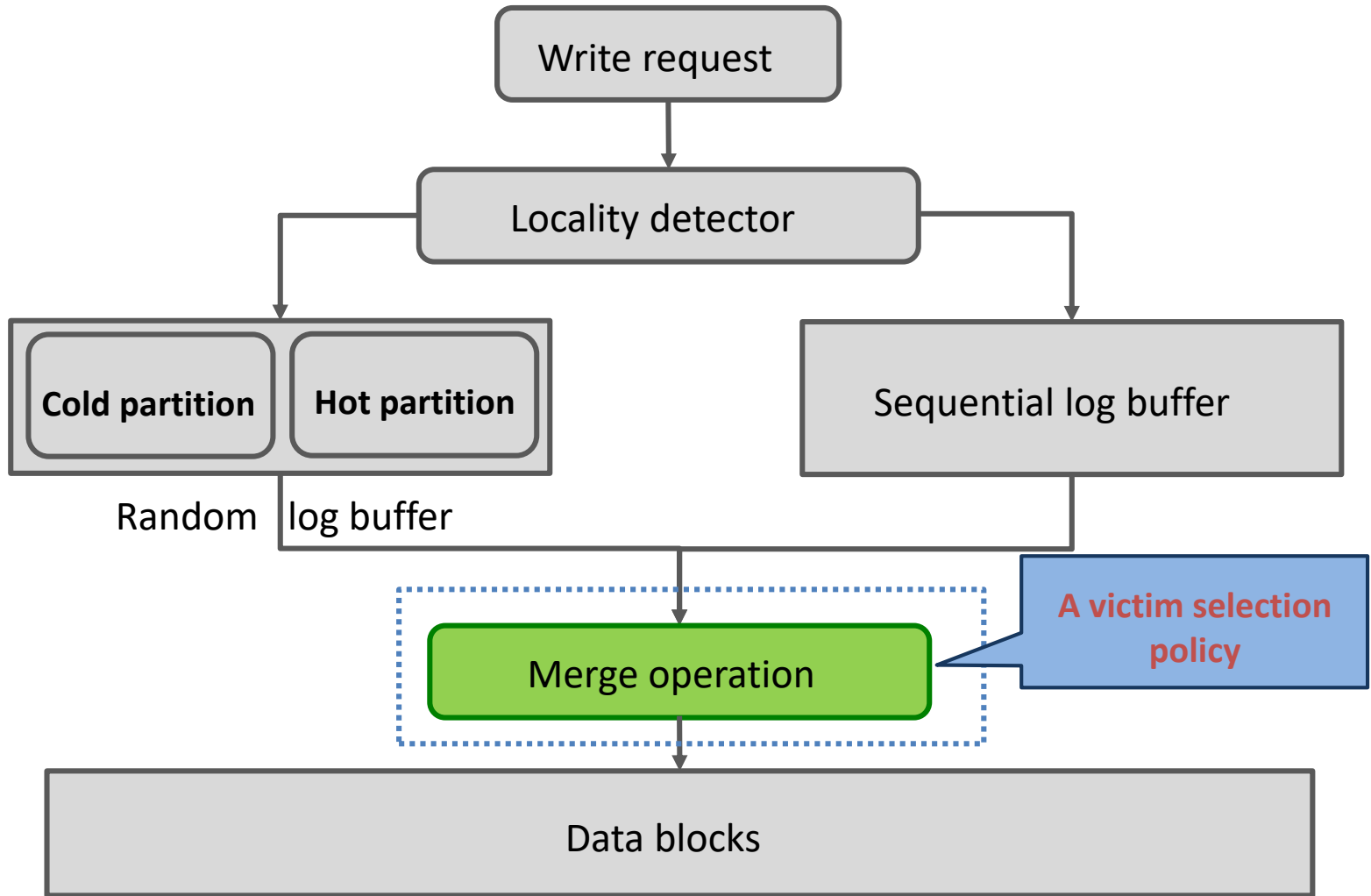
Requested pages:

1 → 4 → 3 → 1 → 2 → 7 → 8 → 2 → 1 → 5 → 2 → 9 → 1 → 4 → 2 → 9



Two partitions ($k = 5$)

Overall Architecture of the LAST Scheme



Log Buffer Replacement Policy

- **Log buffer replacement policy**

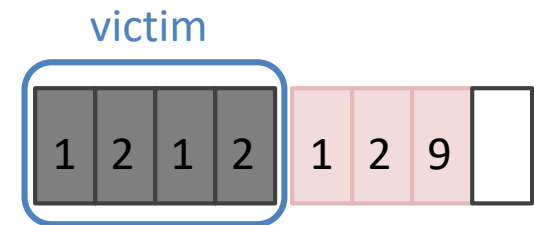
- Proposed to provide a more intelligent victim selection
- Delay an eviction of hot pages as long as possible

(1) evict a dead block first from the hot partition

- requires only one erase operation



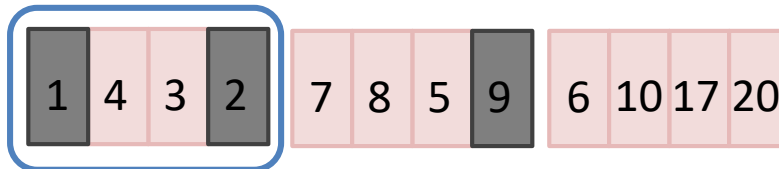
Cold partition



Hot partition

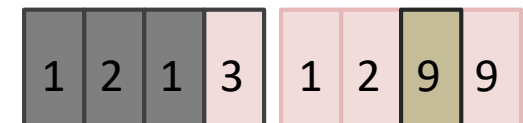
(2) evict a cold block from the cold partition

- select a block associated with a smallest number of data blocks



victim

Cold partition



Hot partition

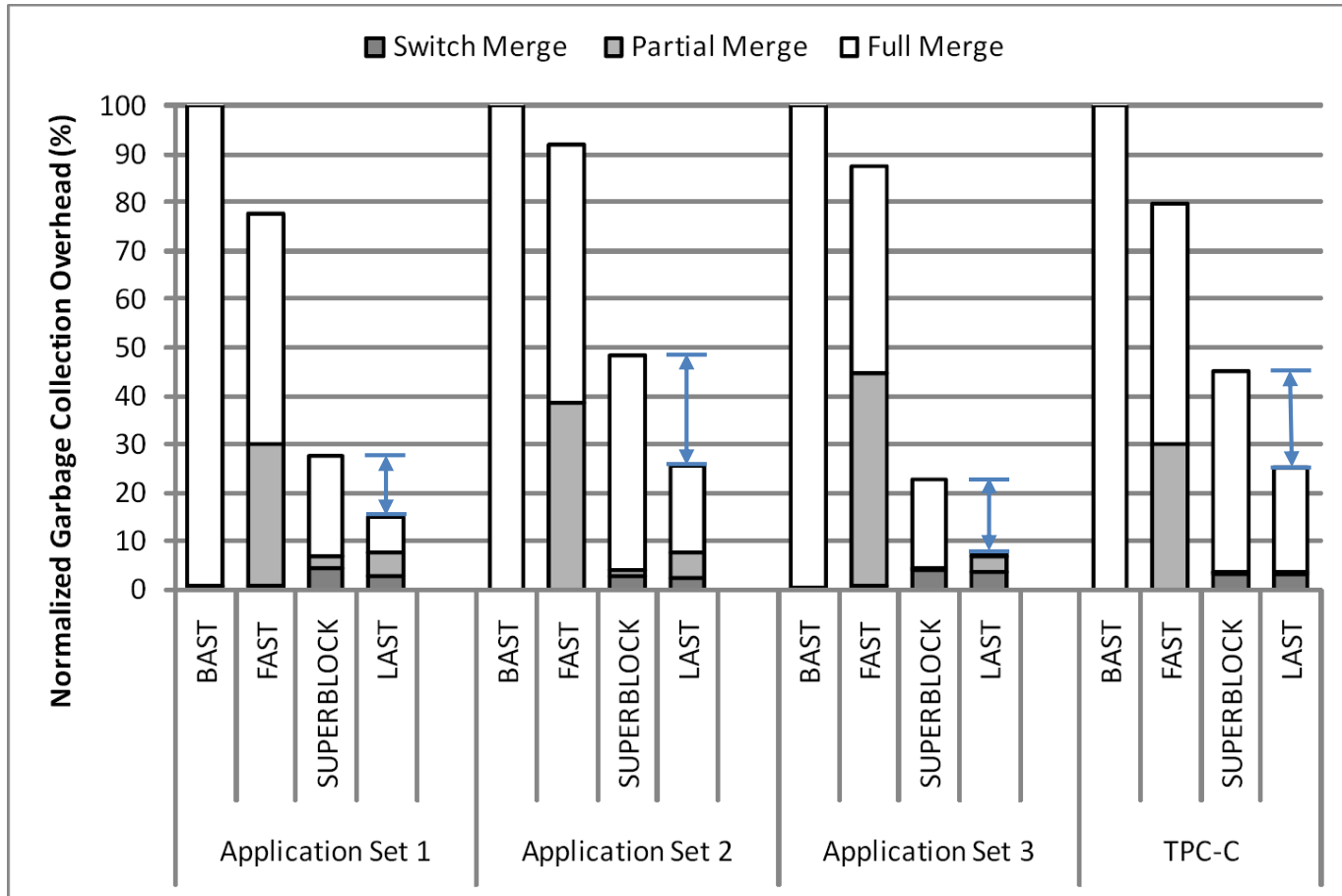
Experimental Results

- Experimental environment
 - Trace-driven FTL simulator
 - Three existing FTL schemes: BAST, FAST, SUPERBLOCK
 - The propose scheme: LAST
 - Benchmarks
 - Realistic PC workload sets, TPC-C benchmark
 - Flash memory model

Flash memory Organization	Block Size	128 KB
	Page size	2 KB
	Num. of pages per block	64
Access time	Read (1 page)	25 usec
	Write (1 page)	200 usec
	Erase (1 block)	2000 usec

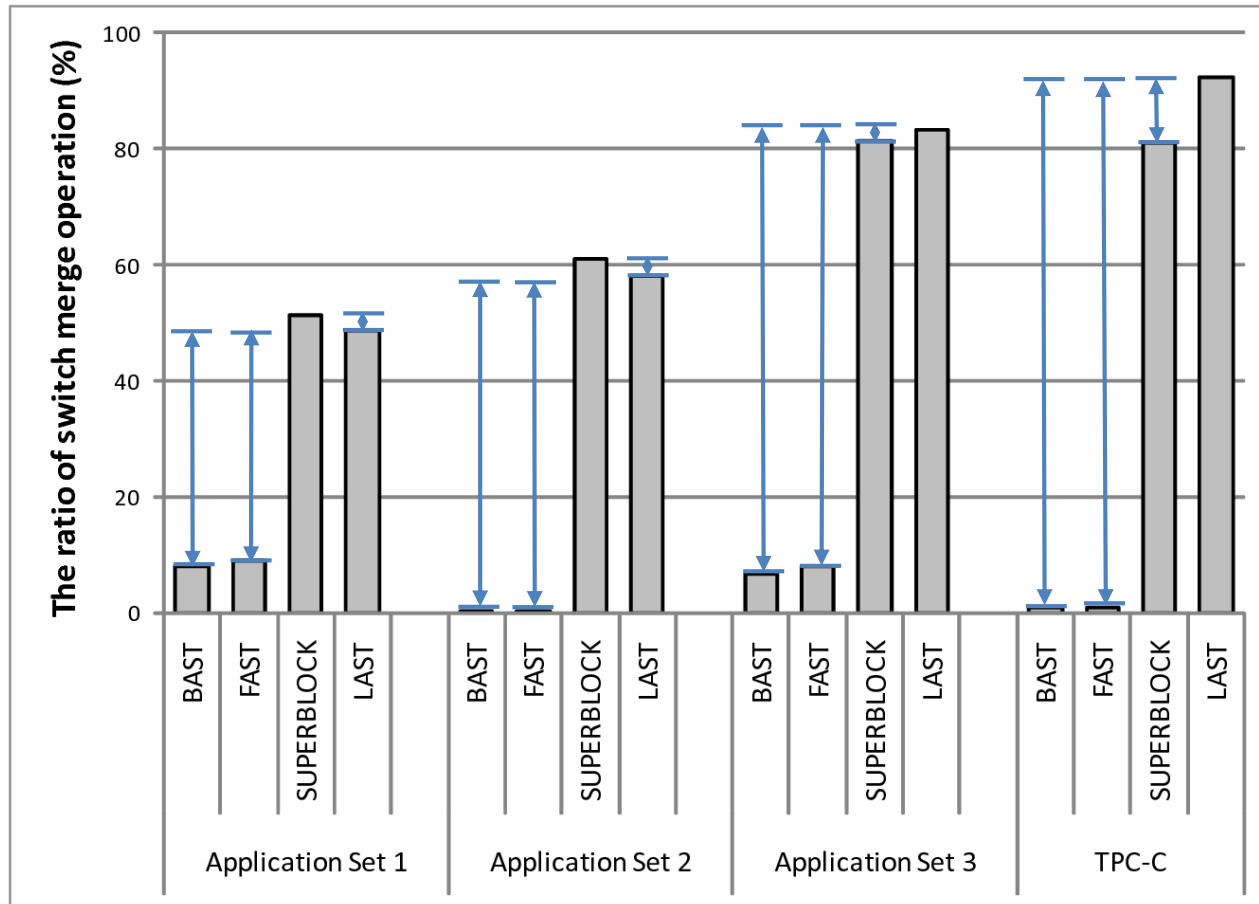
- Important parameters
 - Total log buffer size: **512 MB**
 - Sequential log buffer size: **32 MB**
 - Threshold value: **4 KB (8 sectors)**

Result 1: Garbage Collection Overhead



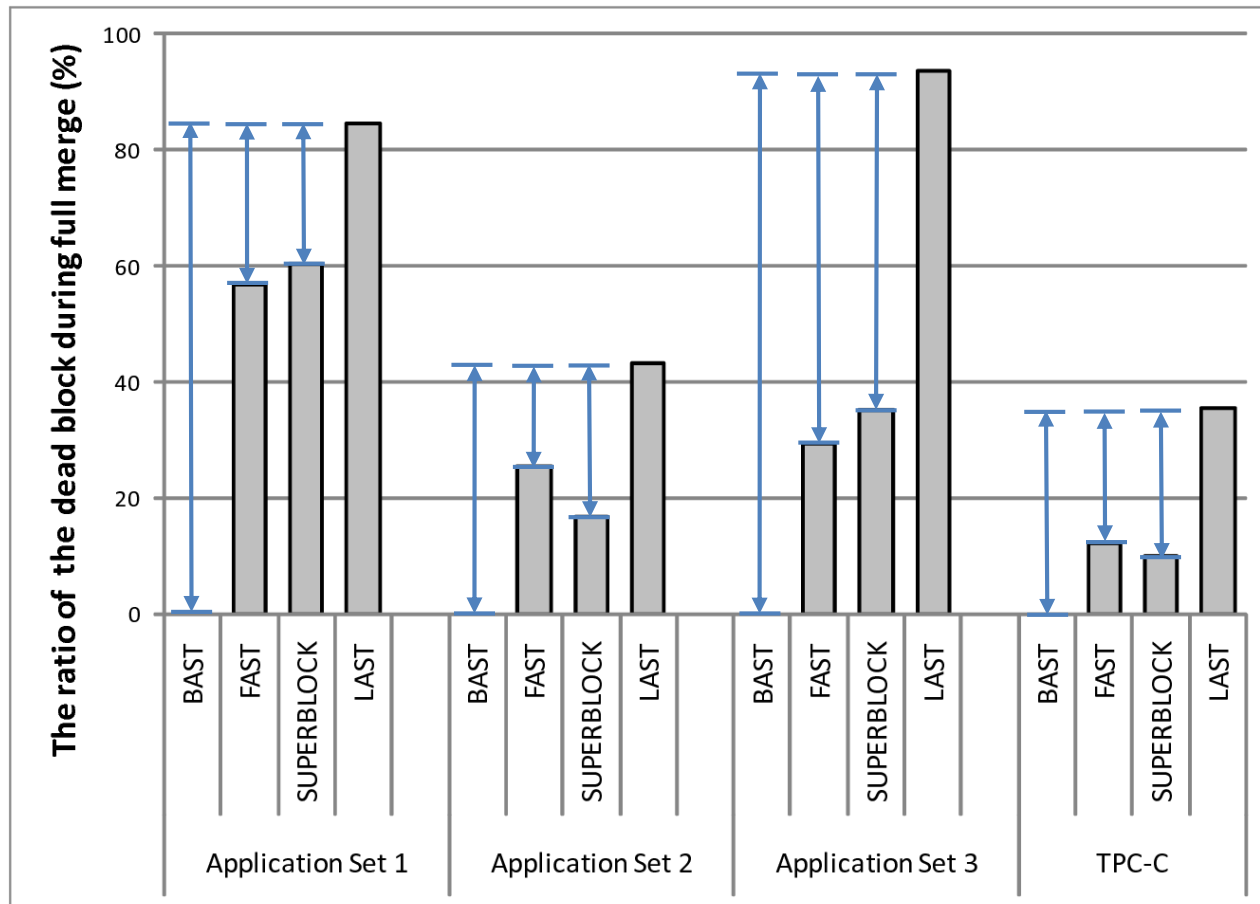
- LAST shows the best garbage collection efficiency
 - Garbage collection overhead is reduced by **46~67%** compared to the SUPERBLOCK scheme

Result 2: Ratio of Switch Merge



- The ratio of switch merges is significantly increased
 - SUPERBLOCK also shows a high switch merge ratio

Result 3: Ratio of Dead Block



- Many dead blocks are generated from the random log buffer

Reference

- J. Kim et al, “A space-efficient flash translation layer for compact flash systems,” IEEE Transactions on Consumer Electronics, vol. 48, no. 2, pp. 366-375, 2002.
- S. W. Lee et al, “A log buffer based flash translation layer using fully associative sector translation,” ACM Transactions on Embedded Computing Systems, vol. 6, no. 3, 2007.
- S. Lee et al, “LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems, ”SPEED 2008.
- J. Kang et al., “A Superblock-based Flash Translation Layer for NAND Flash Memory,” EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software, 2006

Advanced Flash Translation Layer

Jihong Kim

Dept. of CSE, SNU

Outline

- Problems of Hybrid-Mapping-Based FTL
- FTLs for **Memory-Constrained** Storage Systems
 - DFTL

Hybrid FTL Schemes

- The main difficulties the FTL faces in giving high performance is the severely constrained size of SRAM
 - Coarse-grained mapping (block-level mapping)
 - Small SRAM size / Poor garbage collection efficiency
 - Fine-grained mapping (page-level mapping)
 - Efficient garbage collection / Large SRAM size

Problems of Hybrid FTL Schemes

- Fail to offer good performance for enterprise-scale workloads
- Require workload-specific tunable parameters
- Not properly exploit the temporal locality in accesses

Basic Approaches to Memory-Constrained Storage Systems

- **Cached mapping information**
- **On-demand loading** of mapping information
 - DFTL
- **Better data structures** for mapping information
 - **μ -FTL**

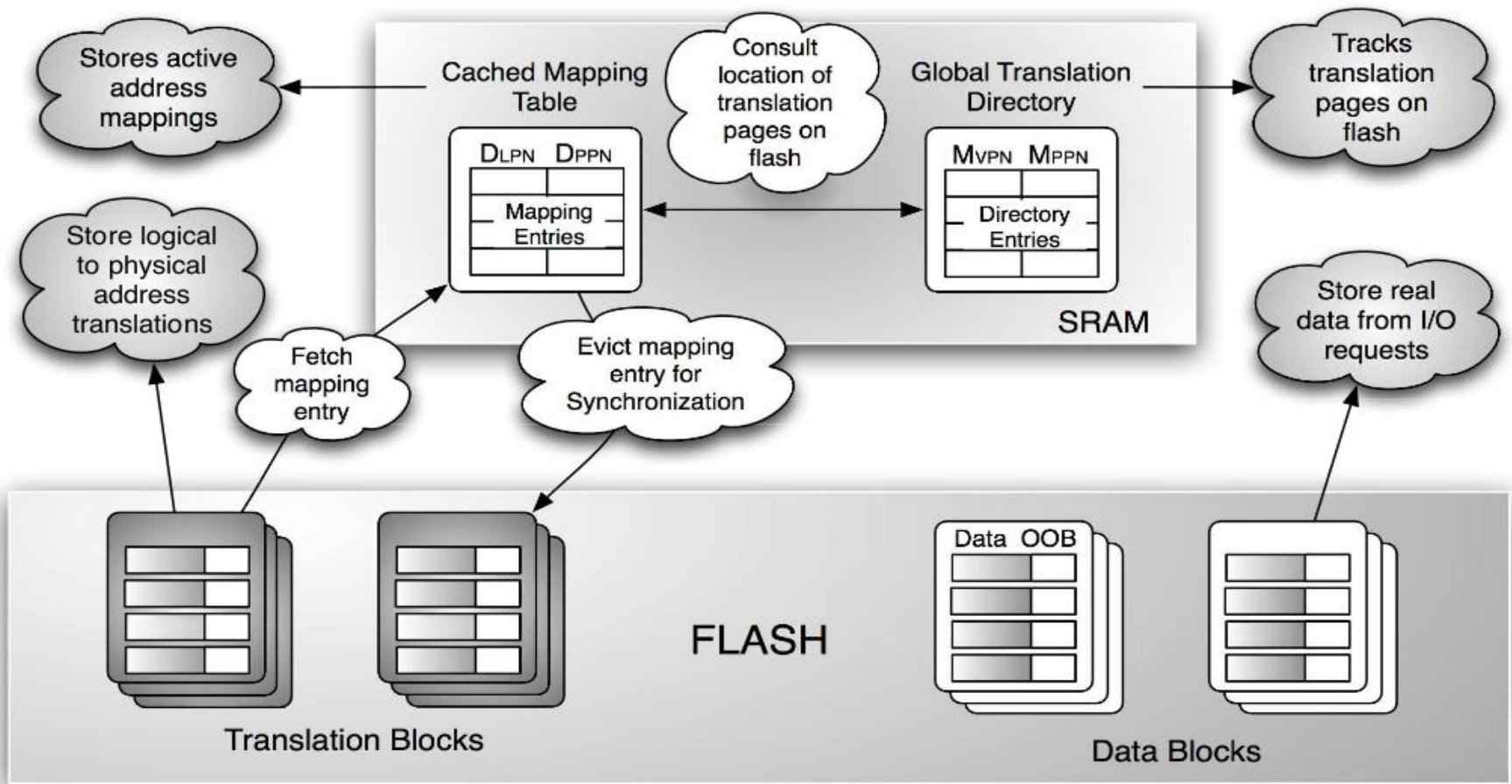
DFTL

- Hybrid FTLs suffer performance degradation due to full merges
 - Caused by the difference in mapping granularity of data and log blocks
 - A high performance FTL must be re-designed without log-blocks
- DFTL is **an enhanced form of the page-level FTL scheme**
 - Allow requests to be serviced from any physical page on flash
 - All blocks can be used for servicing update requests
- How to make the fine-grained mapping scheme feasible with the constrained SRAM size
 - Use an **on-demand address translation mechanism**

Demand-based Selective Caching of Page-level Address Mapping

- Propose a novel FTL scheme (DFTL) : Purely page-mapped FTL
 - Exploit temporal locality of accesses
 - Uses the limited SRAM to store the most popular mappings while the rest are maintained on flash
 - Provide an easier-to-implement solution
 - Devoid of tunable parameters

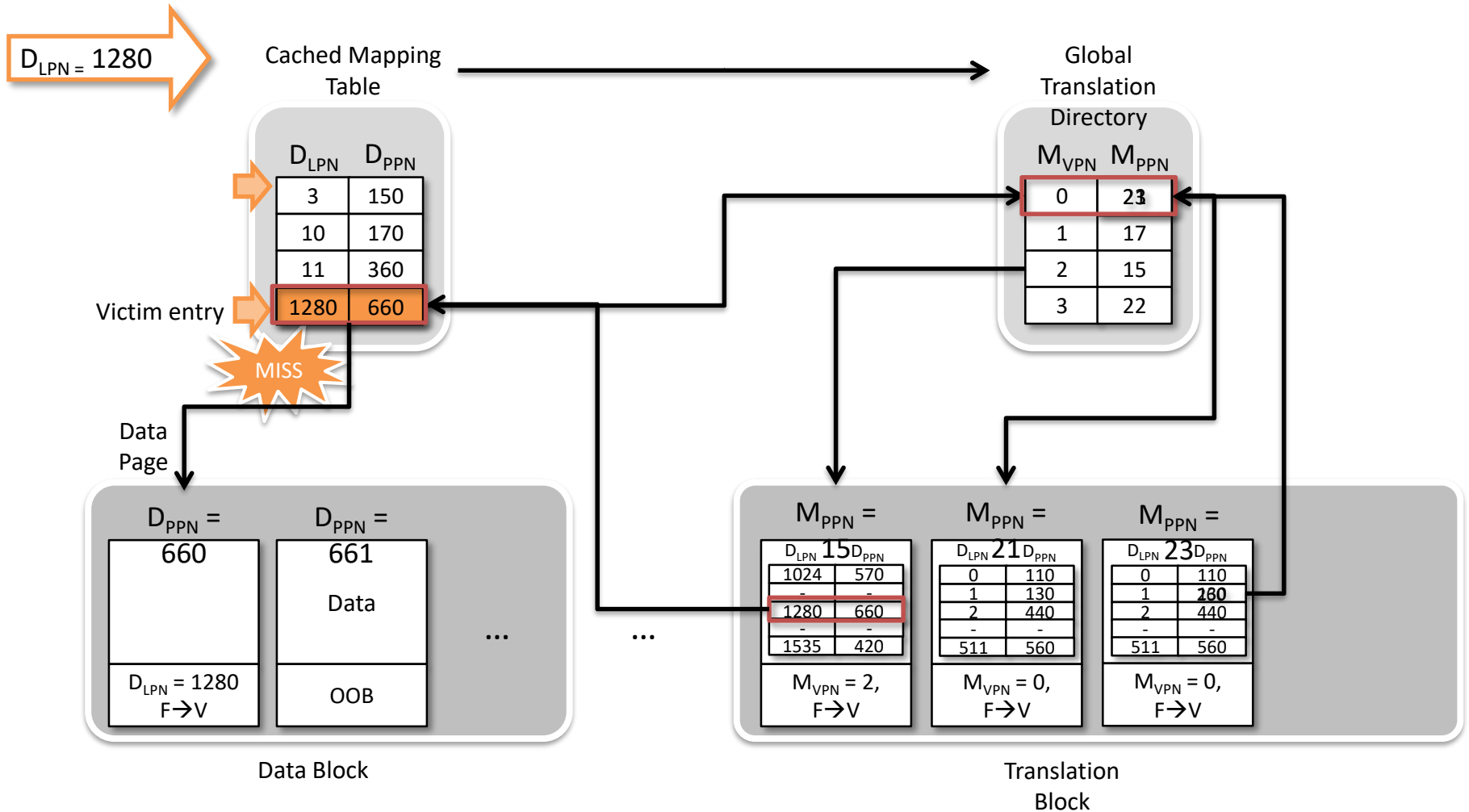
DFTL Architecture



Data Blocks and Translation Blocks

- DFTL partitions all blocks into two groups
 - Data blocks: composed of data pages
 - Each data page contains the real data
 - Translation blocks: consists of translation pages
 - Each translation page stores information about logical-to-physical mappings
 - Logically consecutive mappings information stored on a single page
 - 512 logically consecutive mappings in a single page (page size: 2 KB, addr: 4 Byte)

Example: When a Request Incurs a CMT miss



Overhead in DFTL Address Translation

- The worst-case overhead in DFTL address translation
 - Two translation page reads
 - One for the victim by the replacement policy
 - The other for the original requests
 - One translation page write
 - For the translation page write for the victim
- The address translation overhead can be mitigated
 - The existence of temporal locality helps in reducing the # of evictions
 - Batch updates for the pages co-located in the victim could also reduce the # of evictions

Read/Write Operation

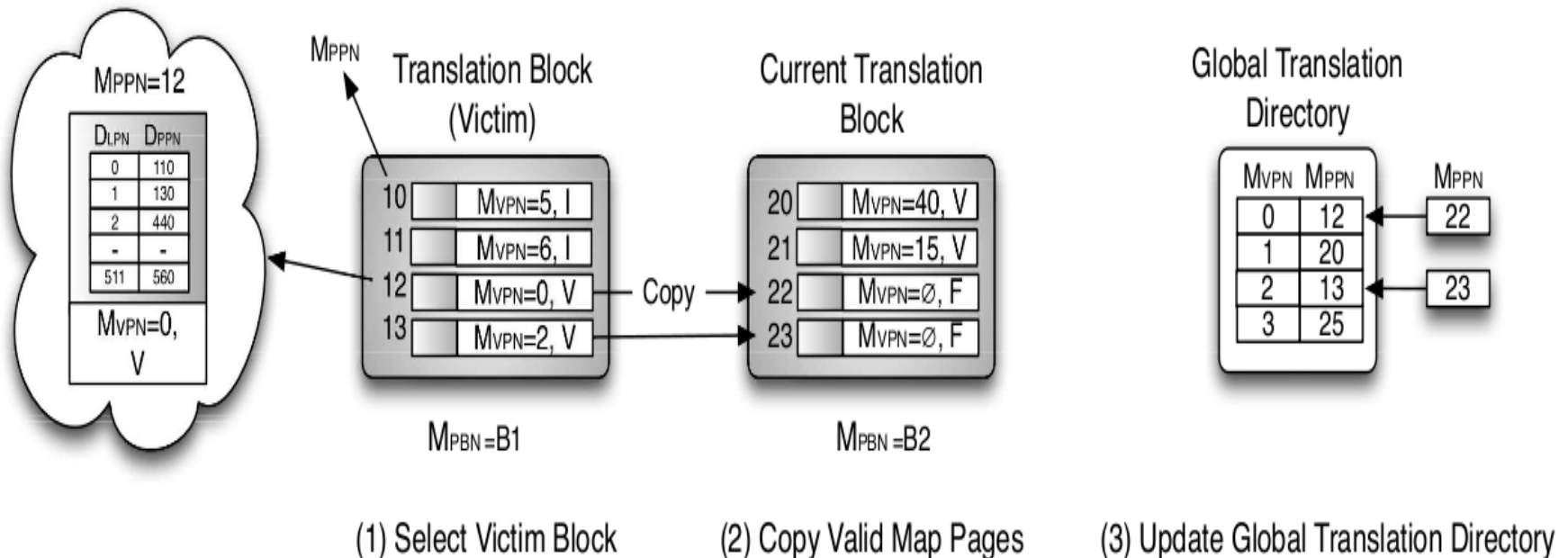
- For a read operation
 - Directly serviced through flash page read operation once the address translation is completed
- For a write operation
 - Maintain two types of blocks for data block and translation blocks
 - *Current data block* and *current translation block*
 - Sequentially writes the given data into these blocks

Garbage Collection

- Different steps are followed depending on the type of a victim block
 - Translation block:
 - Copy the valid pages to the current translation block
 - Update the corresponding GTD
 - Data block:
 - Copy the valid pages to the current data block
 - Update all translation pages and CMT entries associated with these pages

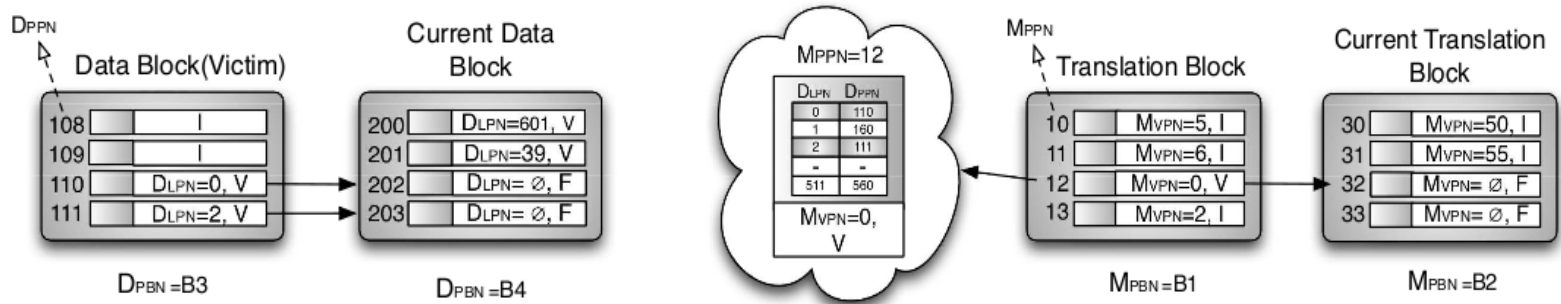
Example: Translation Block

- Translation block as victim for garbage collection



Example: Data Block

- Data block as victim for garbage collection



(1) Select Victim Block

(2) Copy Valid Data Pages

(3) Update Corresponding Translation Page

Global Translation Directory

MVPN	MPPN
0	12
1	20
2	13
3	25

(4) Update Global Translation Directory

Cached Mapping Table

DLPN	DPPN
0	110
1	130
20	150
511	560

(5) Update Cached Mapping Table

Evaluation Setup

● Parameters

- Flash memory size: 32 GB / SRAM size: 2 MB
- Log buffer size: 512 MB (about 3% of the total flash capacity)
- Evaluated schemes: FAST, baseline, DFTL

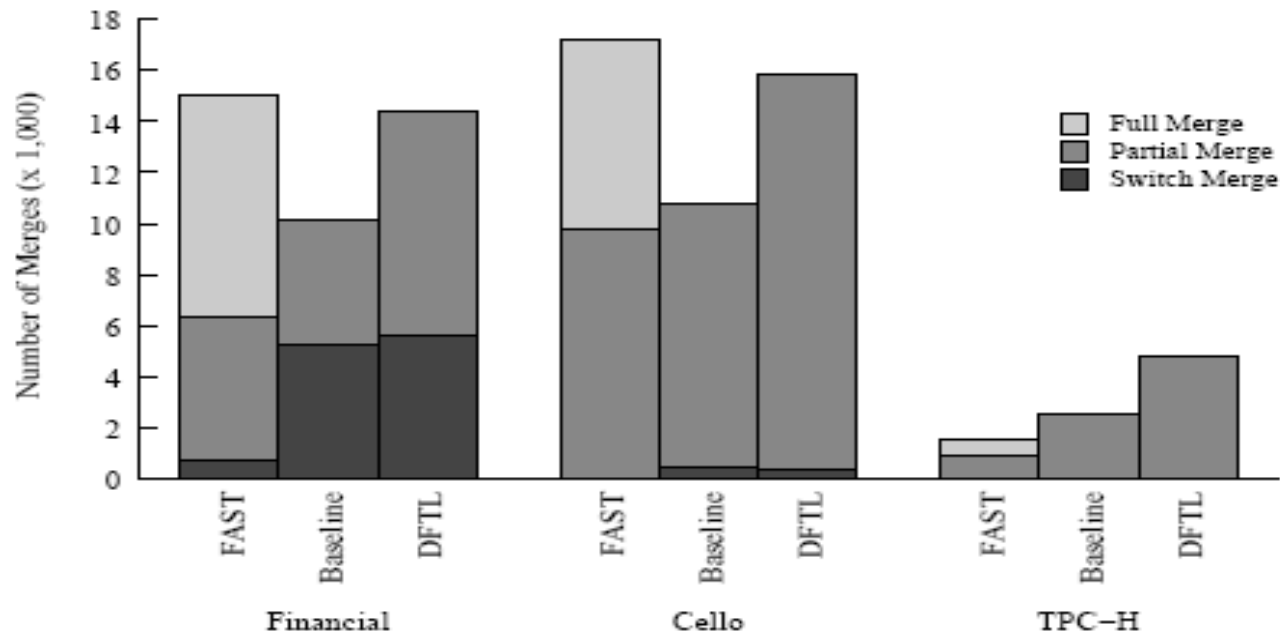
● Workloads

Workloads	Avg. Req. Size (KB)	Read (%)	Seq. (%)	Avg. Req. Inter-arrival Time (ms)
Financial [25]	4.38	9.0	2.0	133.50
Cello99 [10]	5.03	35.0	1.0	41.01
TPC-H [28]	12.82	95.0	18.0	155.56
Web Search [26]	14.86	99.0	14.0	9.97

● Performance metrics

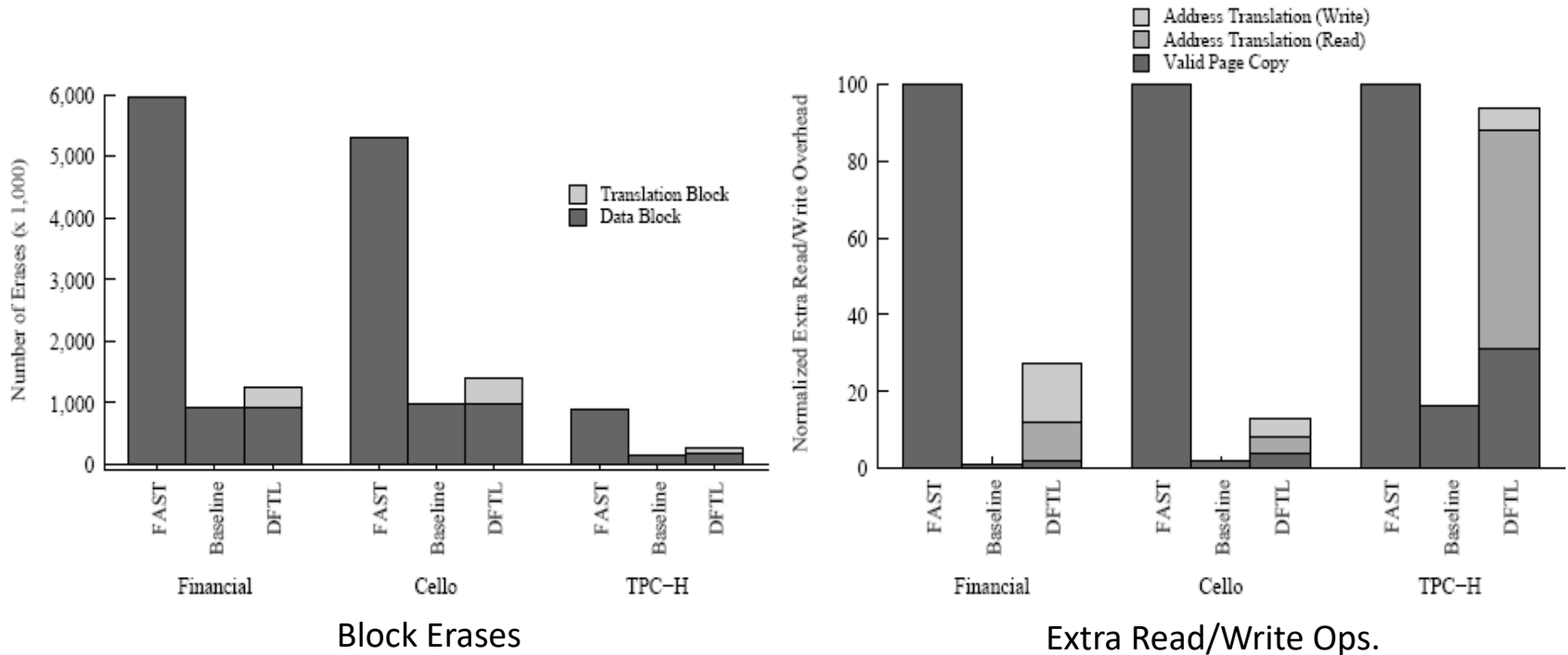
- Garbage collection's efficacy
- Response time (device service time + queuing delay)

The Number of Block Merges



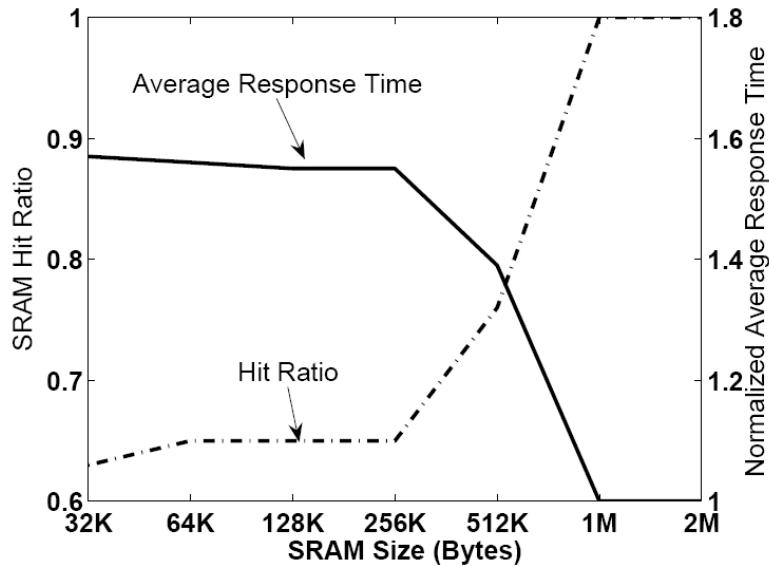
- Baseline and DFTL show a higher number of **switch merges**
- FAST incurs lots of **full merges**
 - 20% and 60% of full merges involve more than 20 data blocks in Financial and TPC-H benchmarks, respectively

Address Translation Overhead

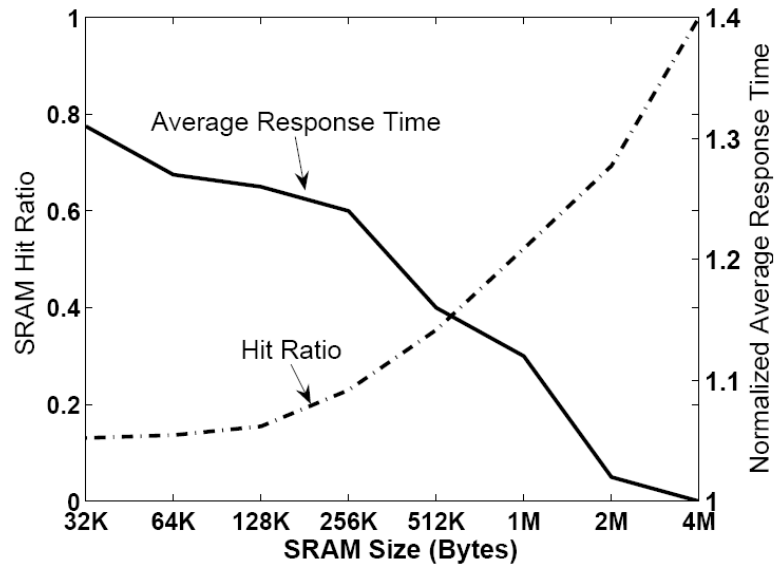


- DFTL incurs extra overheads due to its translation mechanism
 - The address translation accounts for 90% of the extra overhead
- DFTL yields a 3-fold reduction in extra ops. over FAST
 - 63% hits for address translations in SRAM

Impact of SRAM size



(a) Financial Trace



(b) TPC-H Benchmark

- With the SRAM size approaching the working set size
 - DFTL's performance becomes comparable to Baseline (=page level FTL)

Reference

- Aayush Gupta et al., “DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings”, ASPLOS, 2009
- Yong-Goo Lee et al., “ μ -FTL: A Memory-Efficient Flash Translation Layer Supporting Multiple Mapping Granularities,” EMSOFT, 2008
- Dongwon Kang et al, “ μ -Tree : An Ordered Index Structure for NAND Flash Memory,” EMSOFT, 2007

Garbage Collection Technique

Jihong Kim

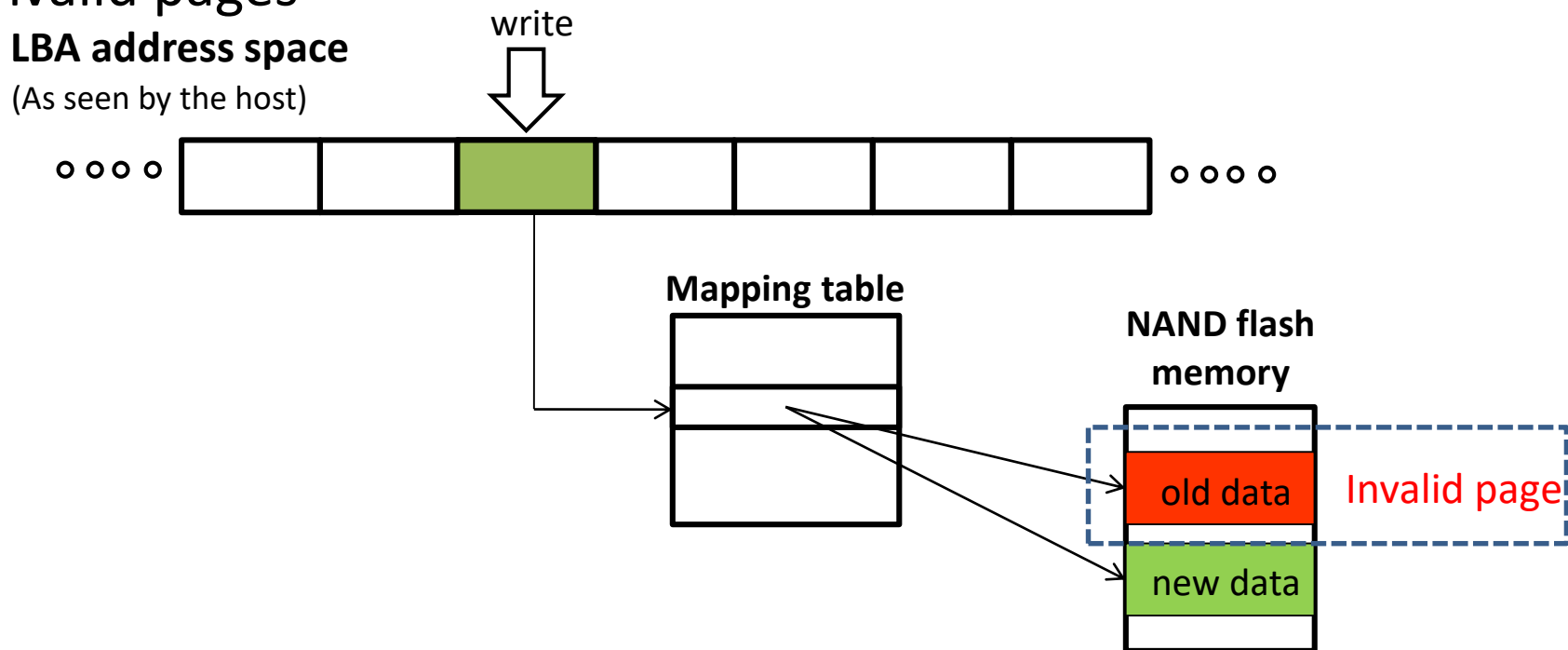
Dept. of CSE, SNU

Outline

- Overview of Garbage Collection
- Technical Issues in Garbage Collection
 - Which block to choose
 - How to organize valid data
 - When to begin
- Conclusion

Out-Place Update

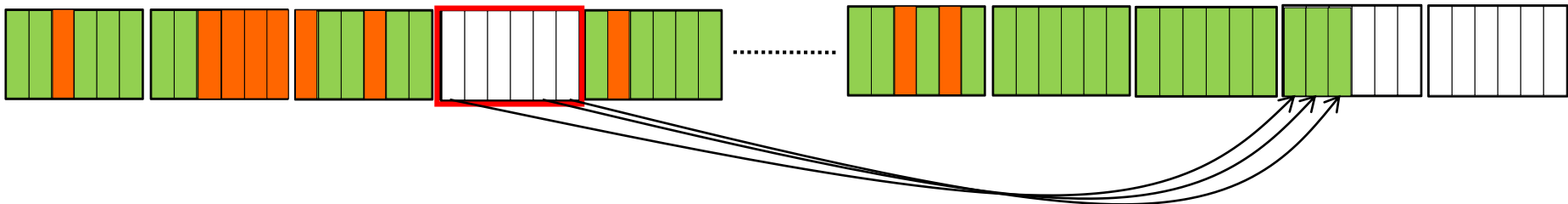
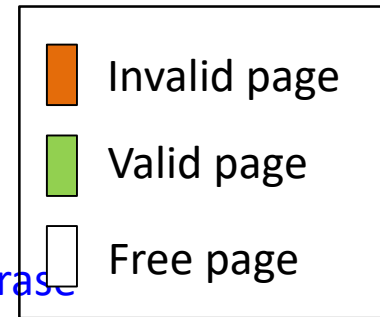
- NAND flash memory does not support an overwrite operation
- FTL uses an out-place update policy, which generates invalid pages



Garbage Collection

- The free space is completely exhausted with invalid pages
- Need to reclaim the space wasted by invalid data
 1. Select the victim block
 2. Copy all valid pages to the free block
 3. Erase the victim block

Garbage collection overhead = valid page copy + block erase



Garbage Collection Overhead

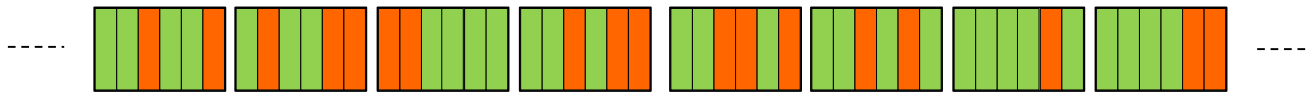
- Garbage collection incurs many valid page copies and block erasures
 - Increase the overall response time of user I/O requests
 - Increase the number of P/E cycles
- Our goal is to reduce the extra operations caused by garbage collection

Technical Issues in Garbage Collection

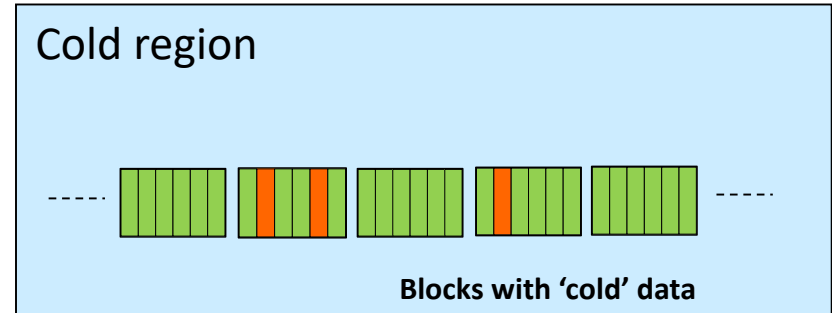
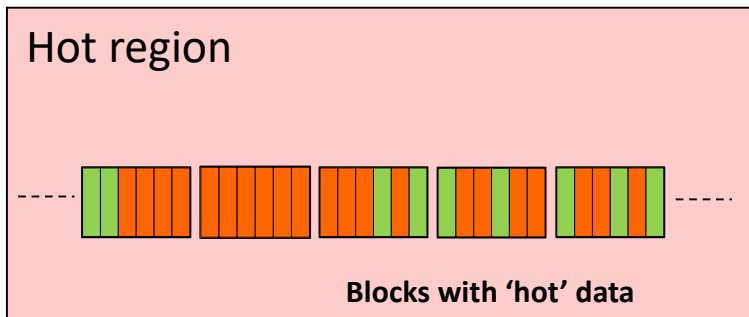
- How to organize valid data
 - Where the user data is written → [Hot and cold separation policy](#)
- Which block to reclaim
 - Which block is preferred for garbage collection → [Victim block selection policy](#)
- When to begin
 - When there are no free blocks → [On-demand garbage collection](#)
 - When there are sufficient idle times → [Background garbage collection](#)

Hot and Cold Separation Policy

- Basic Idea: Age-based Separation
 - Consider the locality of reference
 - Blocks containing 'hot' data tend to be invalidated more rapidly

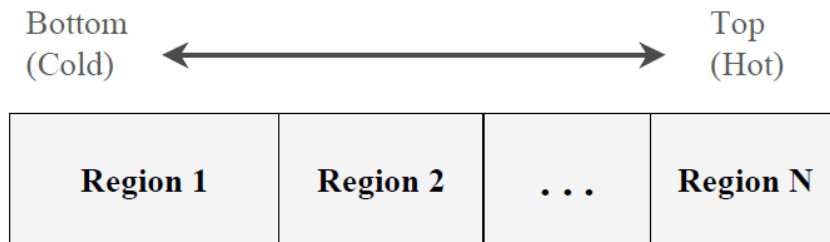


Blocks are classified by its age during garbage collection

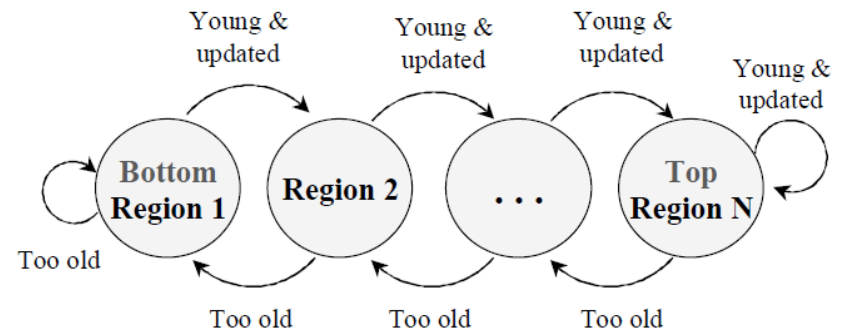


Hot and Cold Separation Policy

- Dynamic dAta Clustering (DAC)
 - Separating Hot/cold data during garbage collection and update



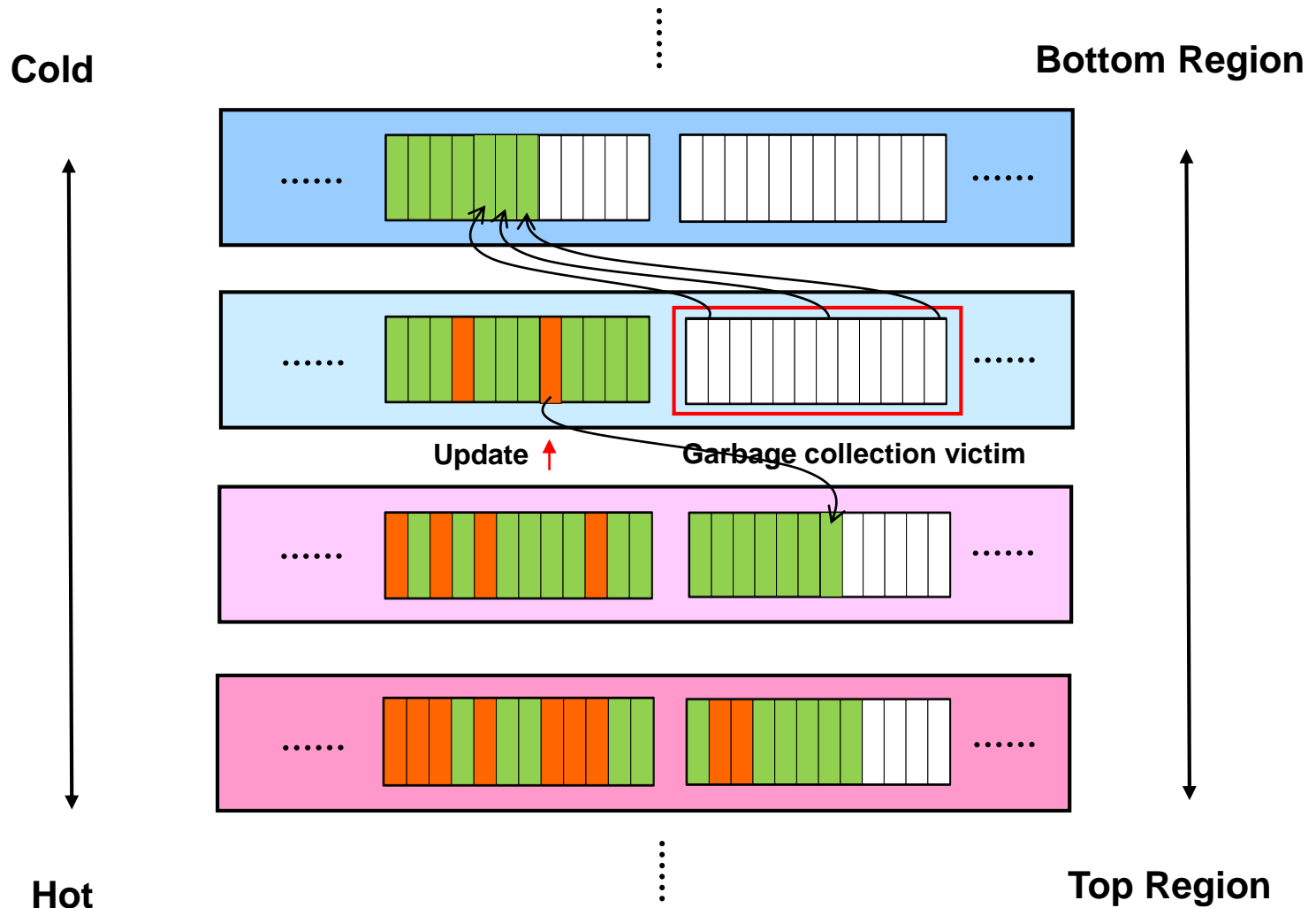
(a) Logically partitioning flash memory into regions.



(b) State transition diagram.

M.-L. Chiang, *et al.*, "Using data clustering to improve cleaning performance for flash memory," *Softw. Pract. Exper.*, 1999.

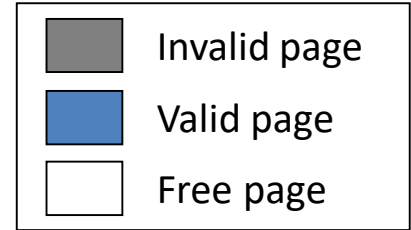
DAC - Example



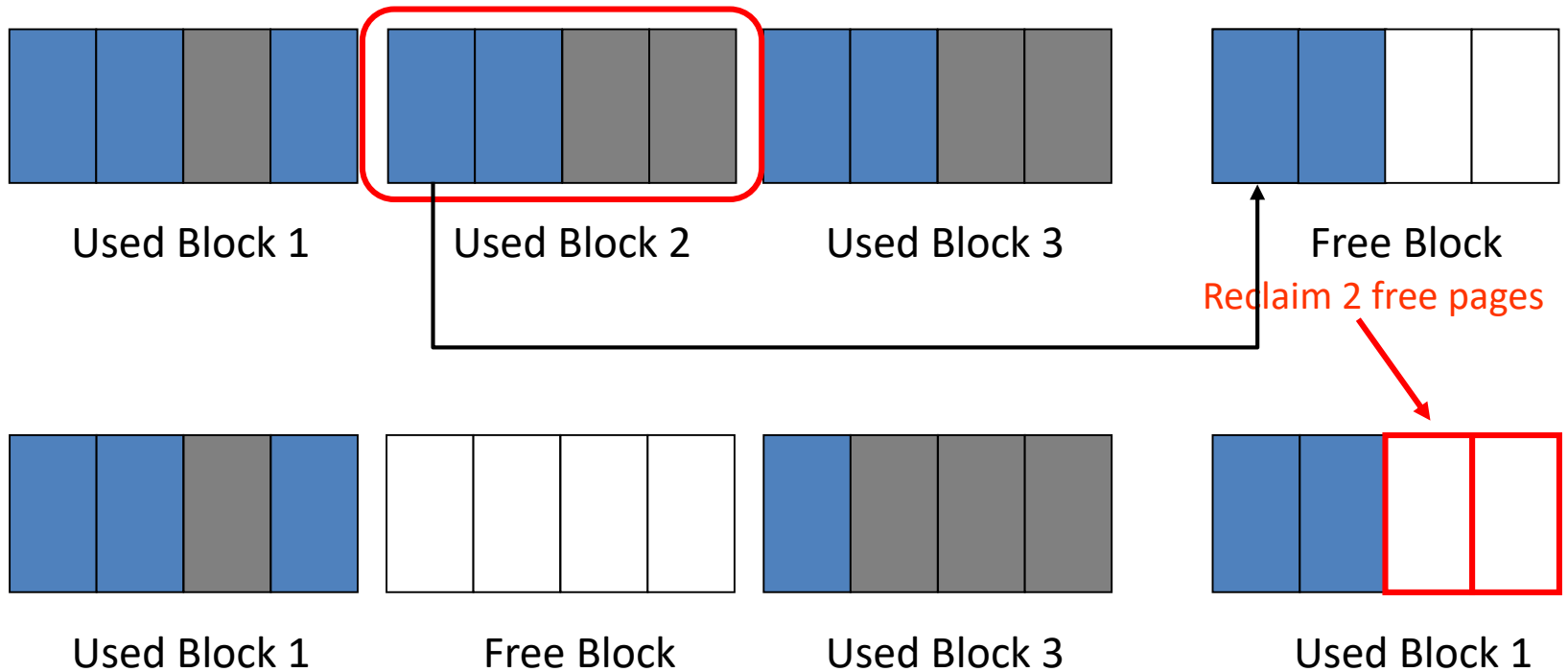
Victim Selection Policy

- Greedy Policy
 - Principle: choose the least utilized block to clean
 - Pros: work well under workloads with uniform access pattern
 - Cons: do not perform well when there's high locality of writes

Greedy Policy - Example



Choose the block with the smallest number of valid pages



Victim Selection Policy

- Cost-Benefit Policy

- Principle: chooses a block that minimizes the equation below

$$\frac{\text{Cost}}{\text{Benefit}} = \frac{u}{(1-u) * \text{Age}}$$

* u : utilization of the block (# of valid pages)

* Age : the most recent modified time of any page in the block

- Pros: perform well with update locality
- Cons: computation/data overhead

Age Transformation Function

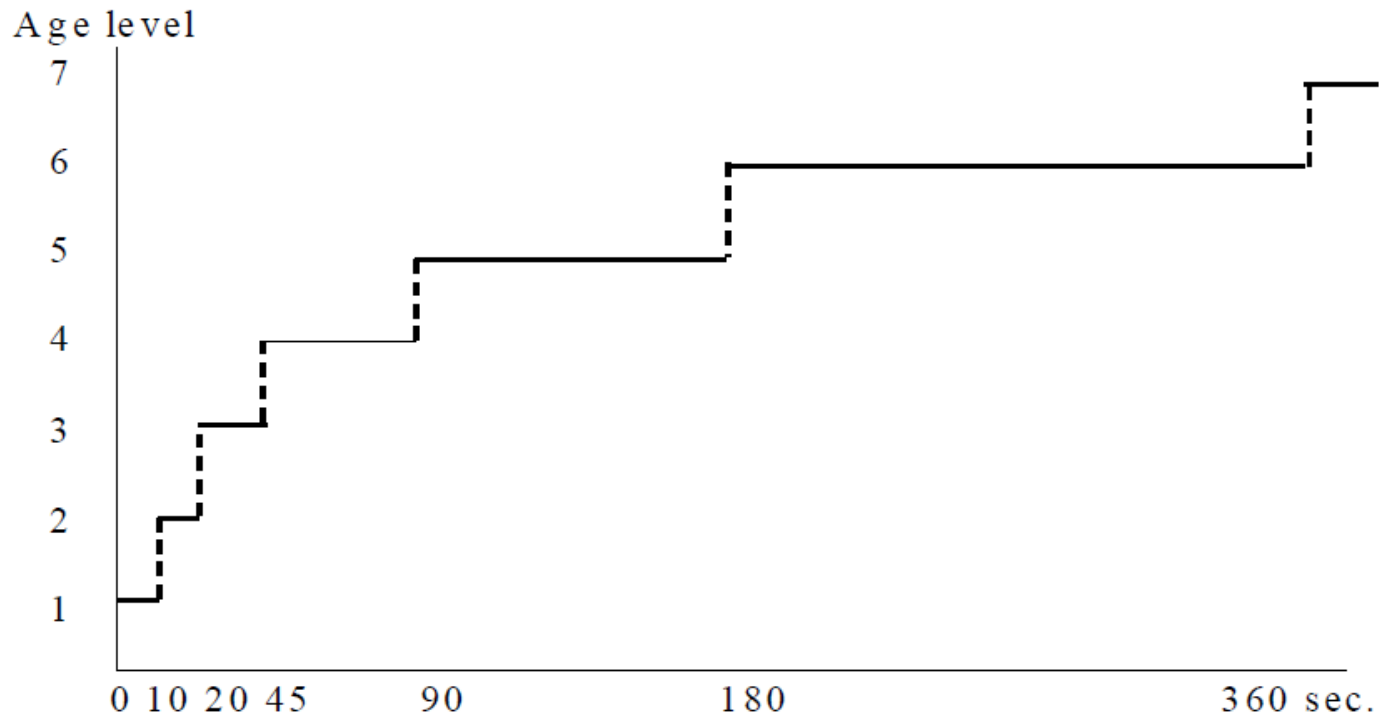
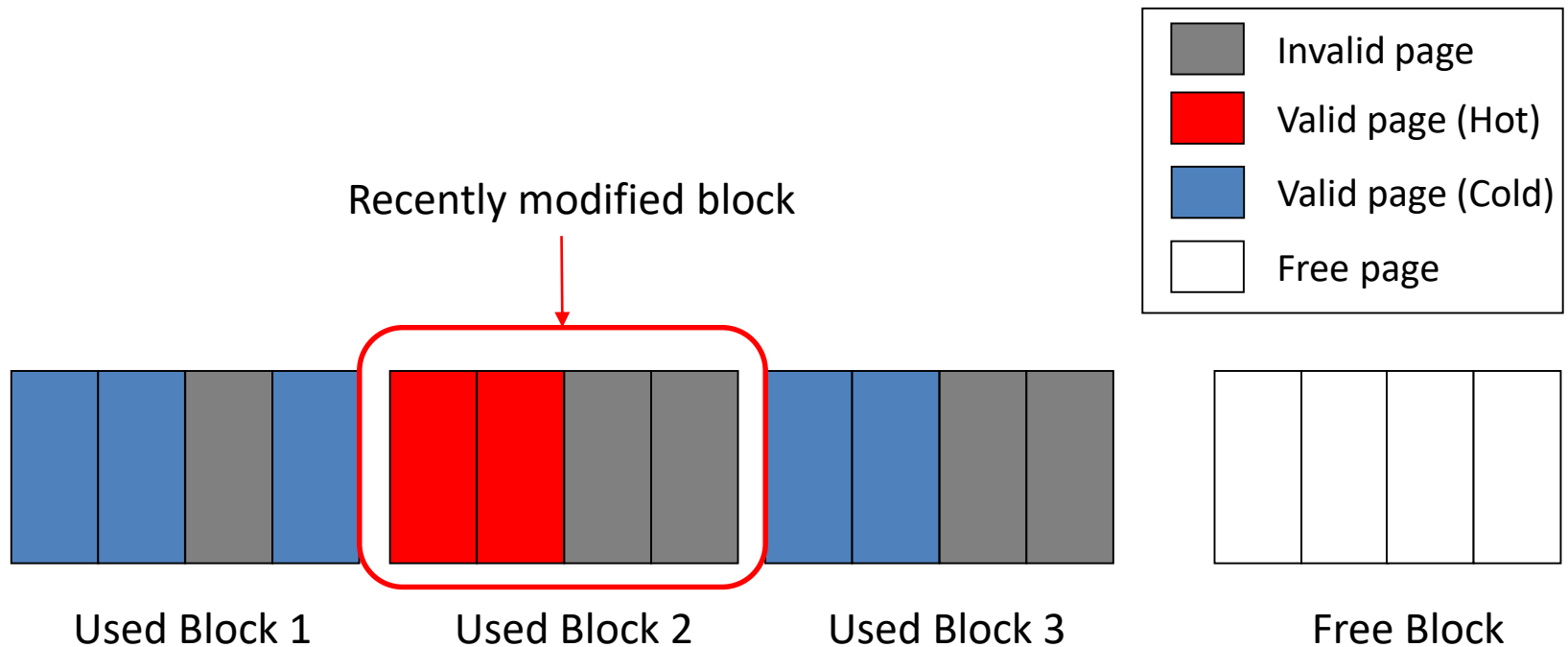


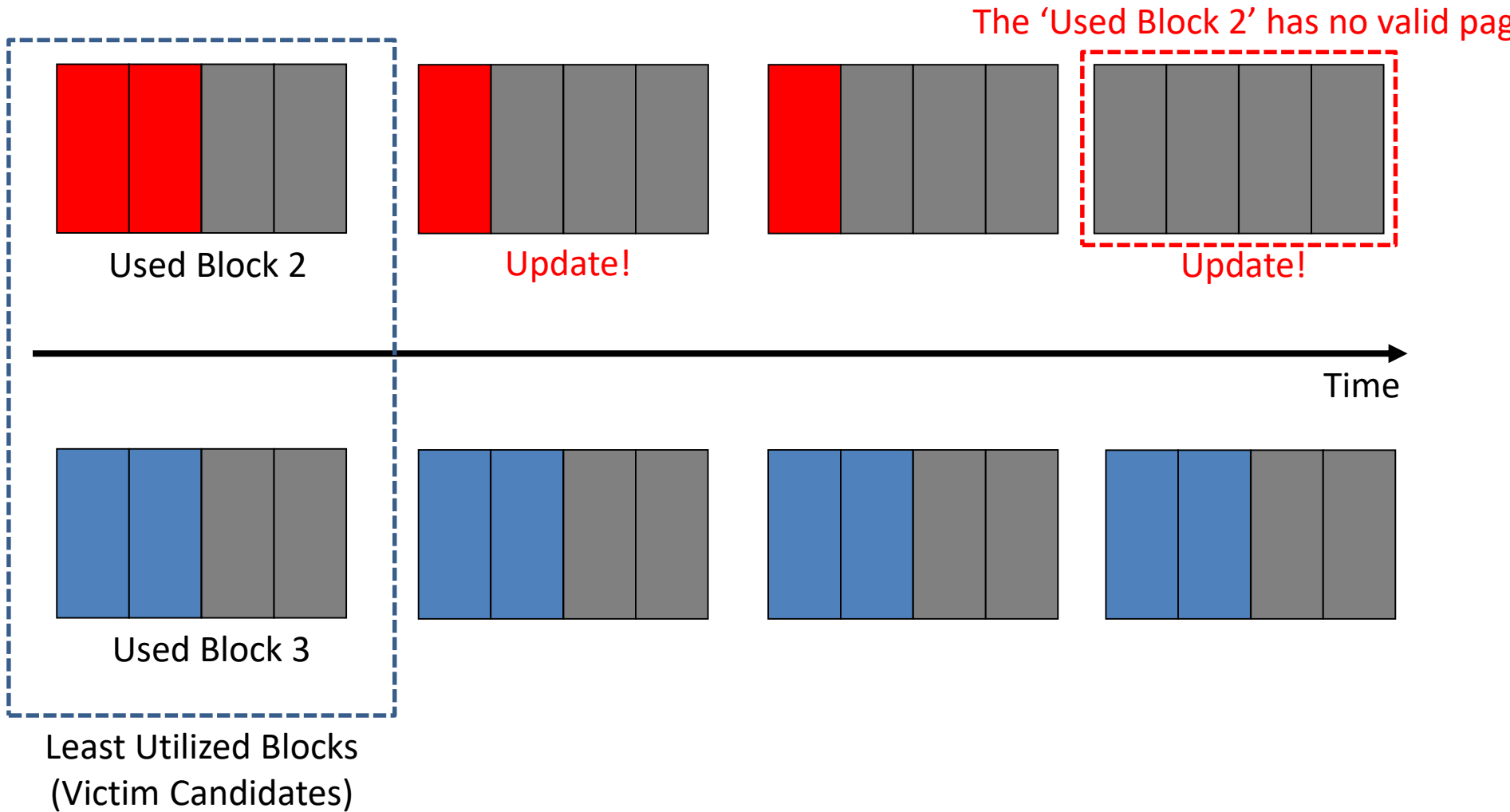
Figure 7: Age transformation function.

Cost-Benefit - Example



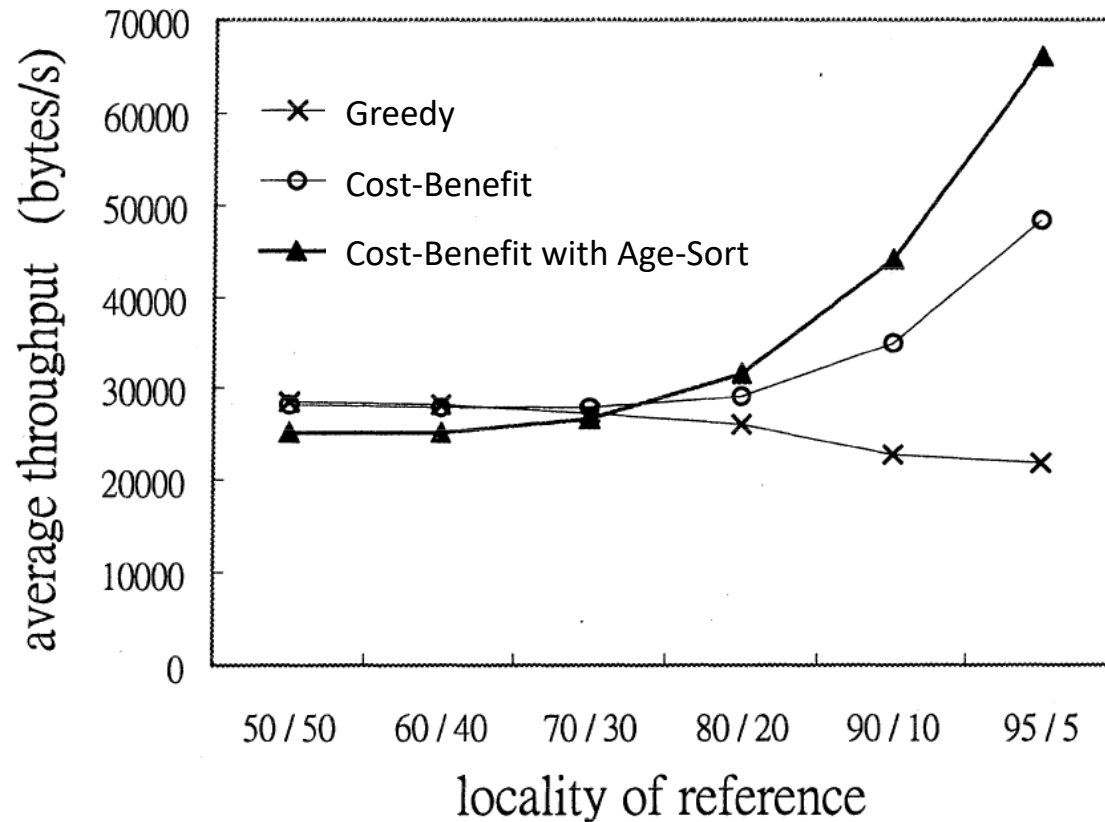
- Used Blocks 2 and 3 have the least block utilization
- Chooses 'Used Block 3' as a victim block because it holds many cold pages

Cost-Benefit - Example



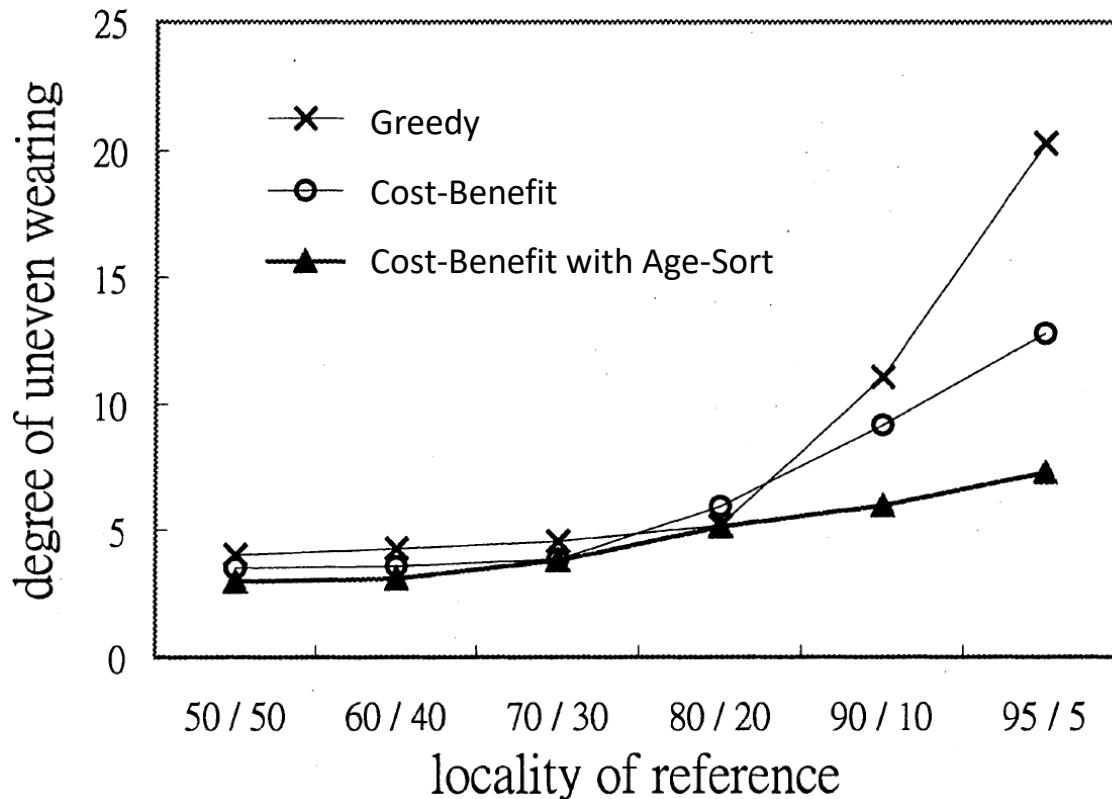
Experimental Results

- Average throughput



Experimental Results

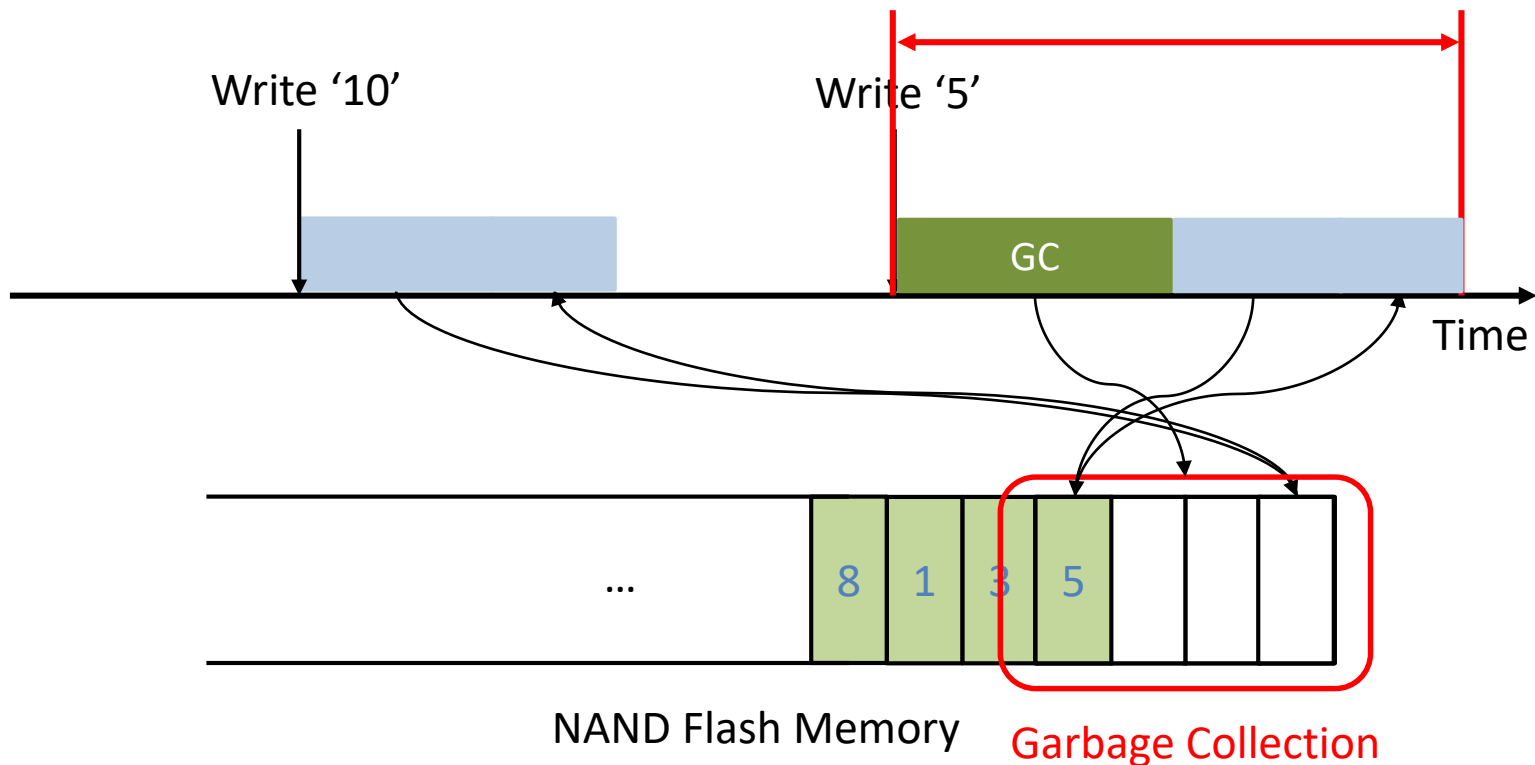
- Degree of uneven wearing



On-Demand Garbage Collection

- Perform garbage collection when there are no free blocks in flash memory

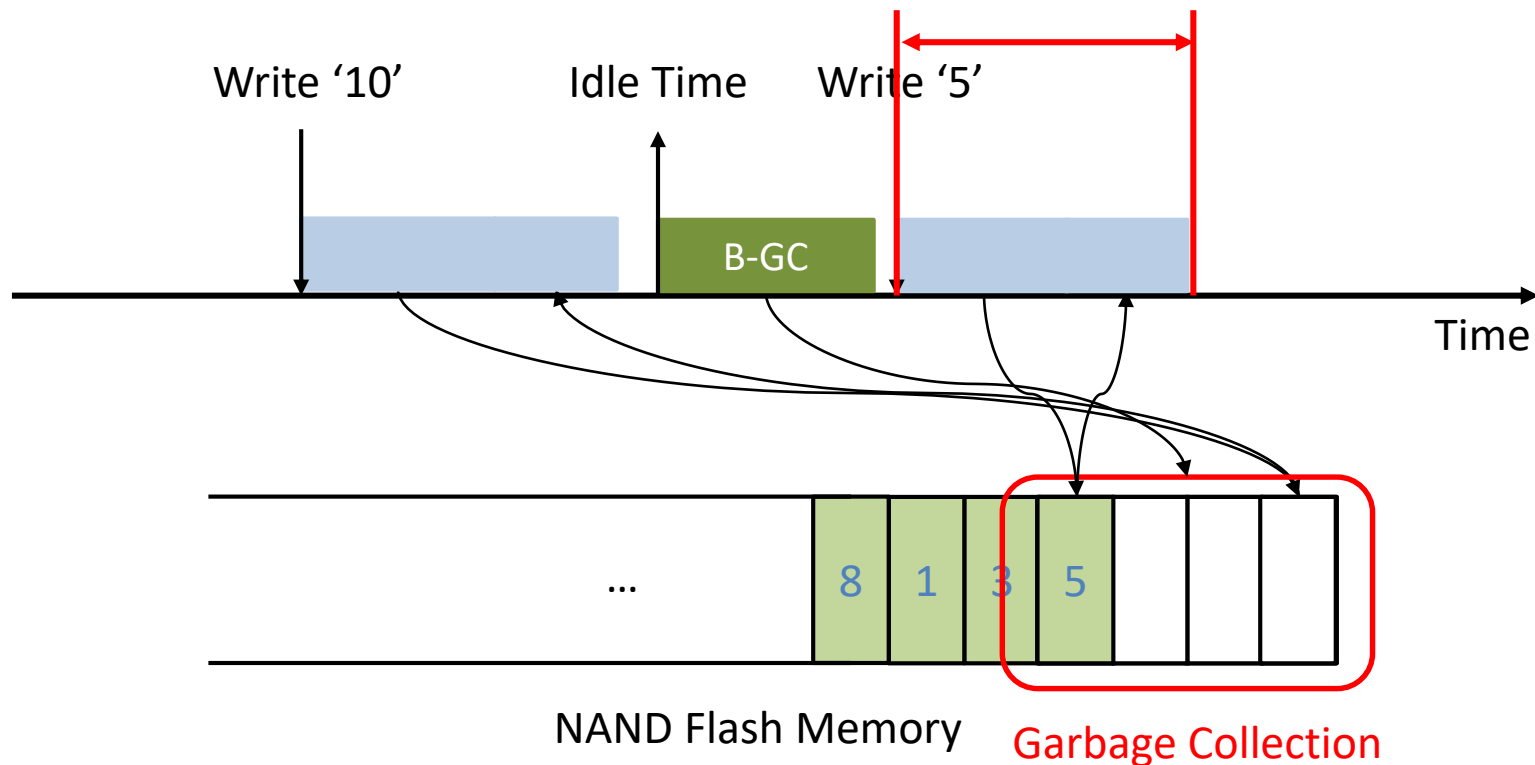
The time taken to write the page '5' is delayed due to GC



Background Garbage Collection (B-GC)

- Perform garbage collection when there are available idle times

There is no performance delay due to GC

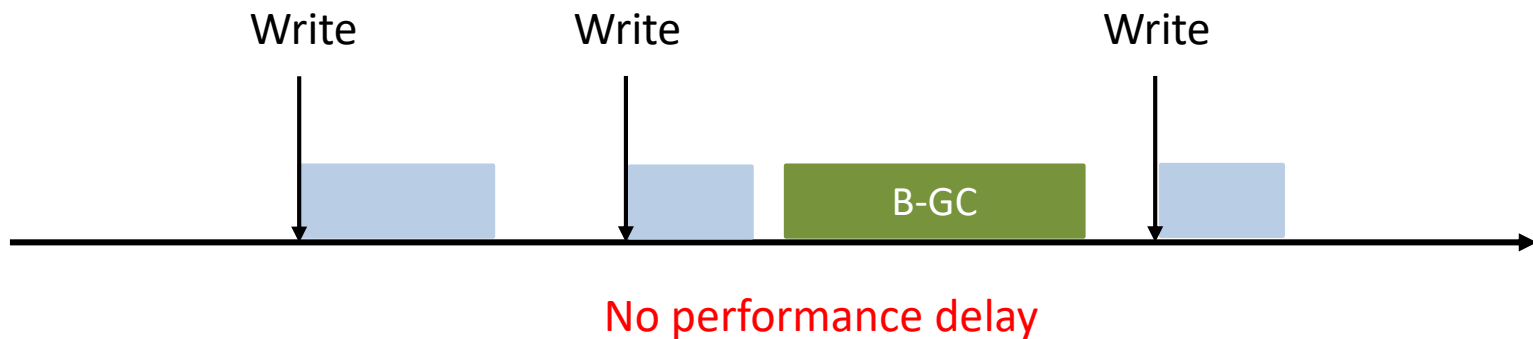
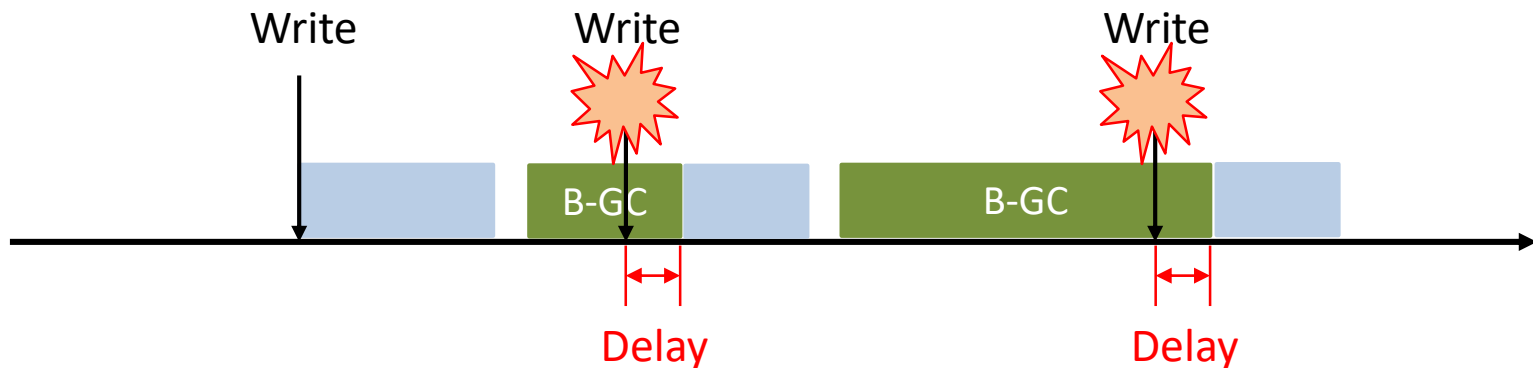


Challenges in B-GC

- When a background garbage collector starts and stops
→ Garbage collection scheduling
- How many over-provisioned pages are maintained
→ Capacity over-provisioning
- ...

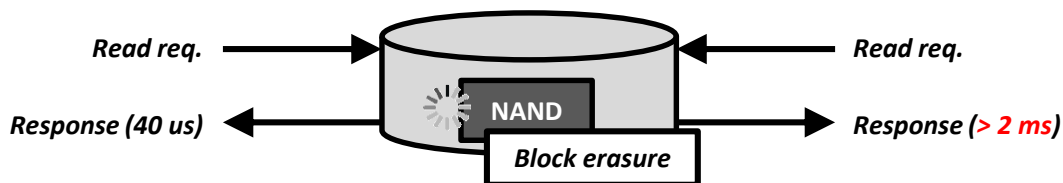
Garbage Collection Scheduling

- Garbage collection must be carefully started and stopped

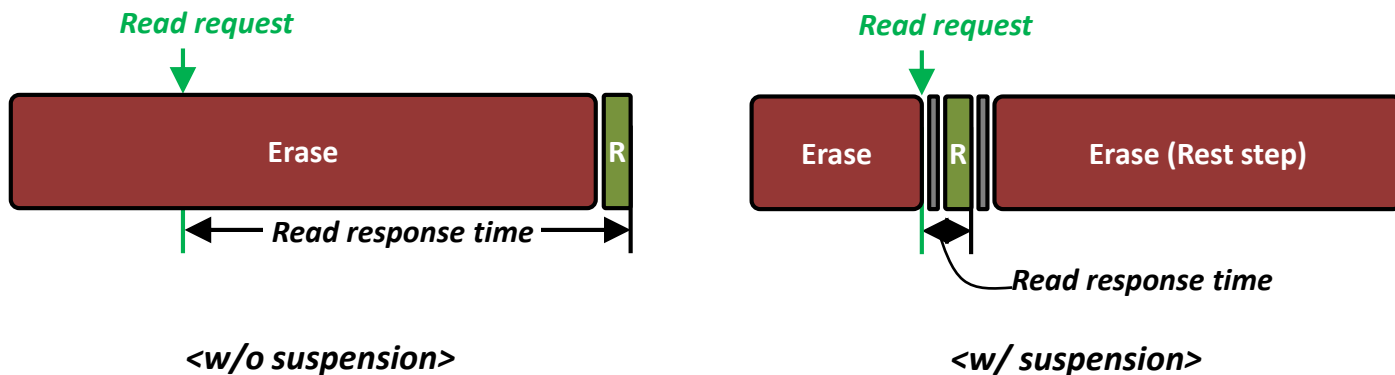


Preemptible Programs and Erases

- Read performance fluctuations
 - Read latency can be increased by one or two orders of magnitudes for waiting the completion of on-going programs and erases.



- Program and erase suspension technique (Wu *et al.* @ FAST'12)
 - Prevents read requests from being blocked by program/erase operation
 - Makes the read latency more deterministic



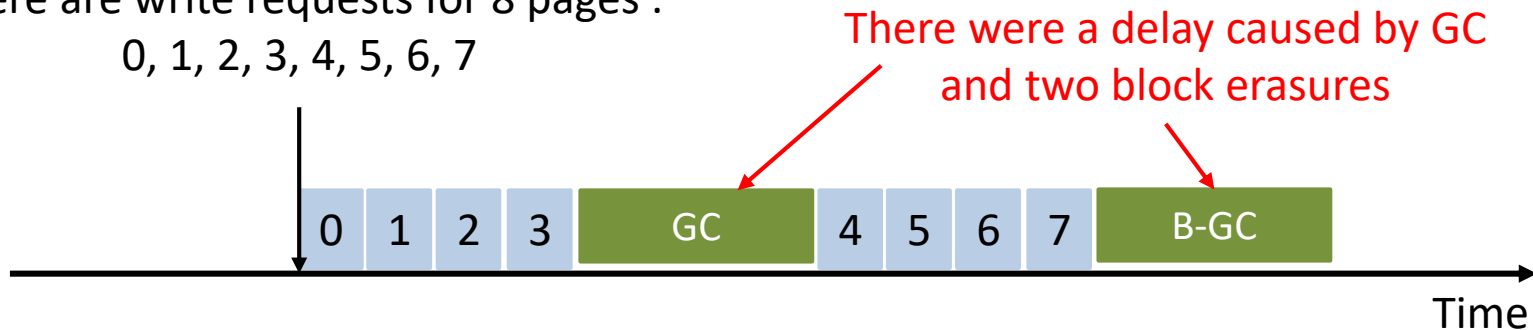
Capacity Over-Provisioning

- A background garbage collector maintains free pages, called over-provisioned capacity
 - To avoid the performance delay caused by on-demand garbage collection
- The over-provisioned capacity must be carefully determined
 - Otherwise, it lowers garbage collection efficiency, reducing the endurance of a flash device

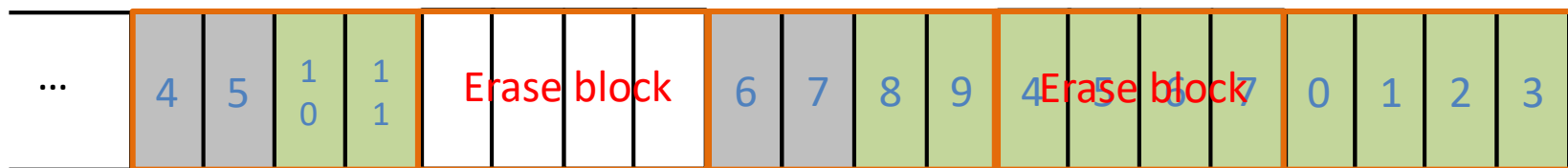
Capacity Over-Provisioning

- Garbage collection occurs when writing incoming pages if the over-provisioned capacity is too small

There are write requests for 8 pages :
0, 1, 2, 3, 4, 5, 6, 7



To maintain the over-provisioned capacity
It is necessary to do GC in background



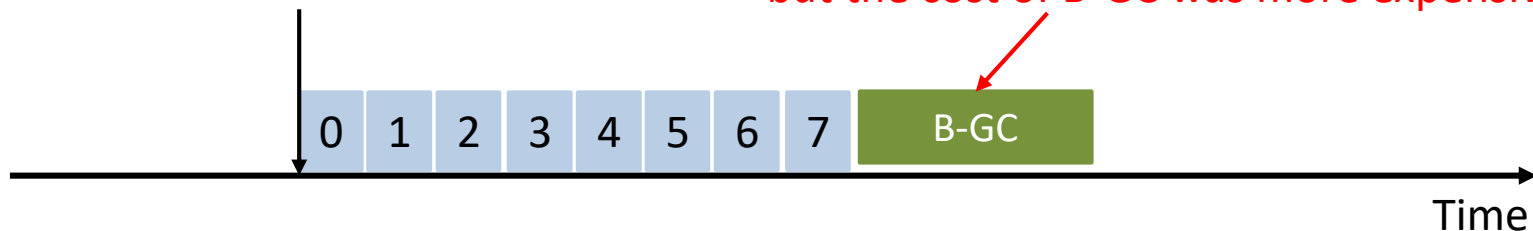
NAND Flash Memory Over-provisioned capacity = 4 pages

Capacity Over-Provisioning

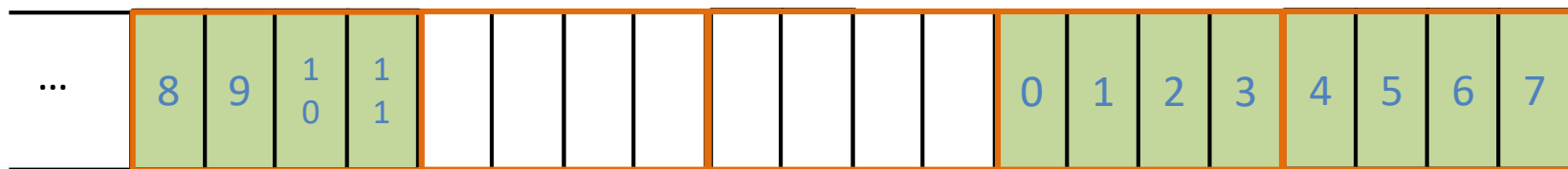
- No performance degradation if there are sufficient over-provisioned pages in flash memory

There are write requests for 8 pages :
0, 1, 2, 3, 4, 5, 6, 7

There was no delay caused by GC,
but the cost of B-GC was more expensive



To maintain the over-provisioned capacity
It is necessary to do GC in background



NAND Flash Memory Over-provisioned capacity = 8 pages

Conclusion

- Reducing the number of copying operations is key to improve garbage collection efficiency
- Combination of hot/cold separation method and victim block selection policy can improve the efficiency of garbage collection
- Background garbage collection can reduce the performance degradation, but the provisioned capacity must be carefully decided

Reference

- Rosenblum, M. and Ousterhout, J. “The design and implementation of a log-structured file system,” ACM Transactions on Computer Systems, vol. 10, pp. 26-52, 1992.
- Chiang, M., Lee, P., and Chang, R. “Using data clustering to improve cleaning performance for flash memory,” Softw. Pract. Exper., vol. 29, pp. 267-290, 1999.
- Kim, H., and Lee, S. “A New Flash Memory Management for Flash Storage System,” 23rd International Computer Software and Applications Conference, 1999.
- Gal, E., and Toledo, S. “Algorithms and data structures for flash memories,” ACM Comput. Surv., vol. 37, pp. 138-163, 2005.
- Chiang, M., Lee, P. and Chang, R. “Cleaning policies in mobile computers using flash memory,” Journal of Systems and Software, vol. 48, no. 3, pp. 213-231, 1999.

Wear-Leveling Techniques

Jihong Kim

Dept. of CSE, SNU

Outline

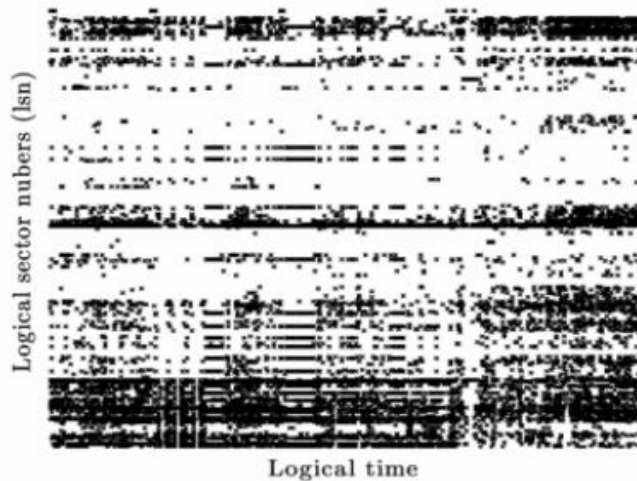
- Overview of Wear-Leveling Technique
- Dynamic Wear-Leveling Technique
 - Cost-Age-Time Policy
- Static Wear-Leveling Technique
 - Hot-Cold Swapping
 - Dual-Pool Algorithm

Wear-Leveling Problem

- Flash memory blocks have a limitation on the number of erase operations (i.e., *erasure cycle*)
 - e.g., MLC: 3~5K, SLC: 100K
- If the same blocks are repeatedly overwritten, there could be the lifetime problem
 - e.g., flash-unaware conventional file systems where the same areas are repeatedly updated.

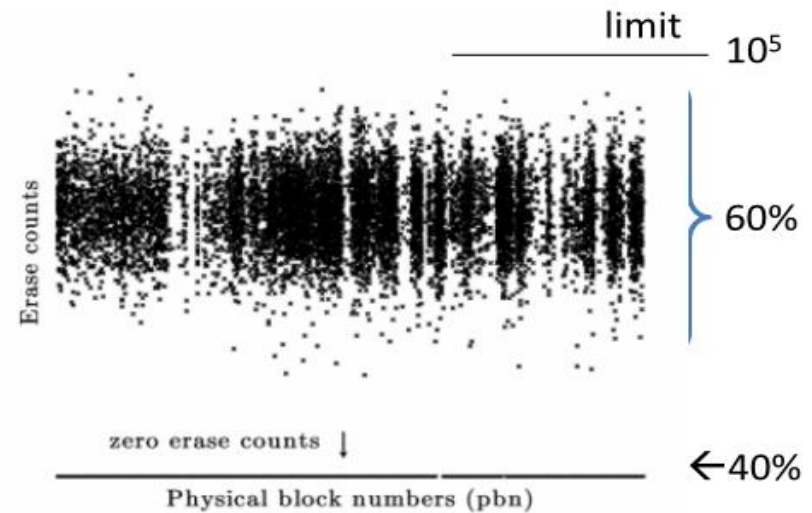
Example: Needs for Wear Leveling

- Write requests toward a **small collection of disk sectors**
- Nearly **40%** of all physical blocks have zero erase



(a)

Disk write pattern



(b)

Flash wear in SSD

Evenly distributing erase operations can double the flash lifespan compared to that without wear leveling

Goal of Wear Leveling

- Wear leveling attempts to work around these limitations by arranging data so that erasures and re-writes are **distributed evenly** across the medium
 - In this way, no single erase block prematurely fails due



Dynamic Wear-Leveling Techniques

- Wear leveling techniques are applied only when data blocks are written or erased
 - E.g., a selection of a new free data block based on the number of program/erase cycles OR a victim block selection based on the PE cycles
- If a data block is not actively written, no wear-leveling is applied to the block under the dynamic approach.
- That is, these techniques are applied to **dynamically changing data blocks only**

Static Wear-Leveling Techniques

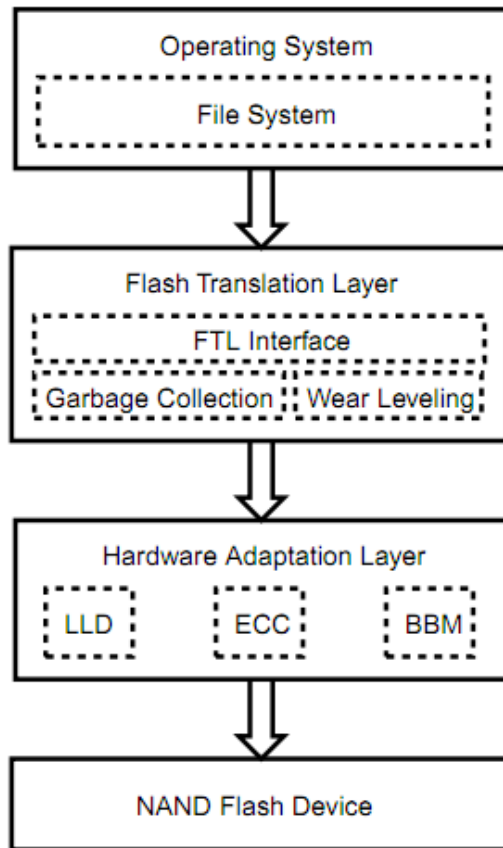
- Wear leveling techniques are applied both static and dynamic data.
 - Cold vs. hot data
- These techniques are applied to data blocks, independently from write/erase operations.

Dynamic vs. Static

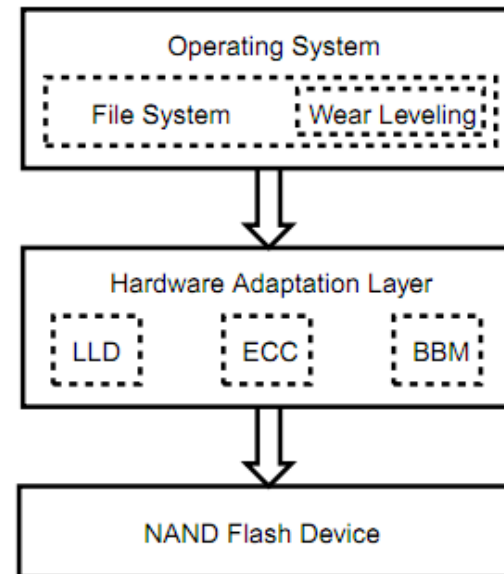
ITEM	Static	Dynamic
Typical Use	SSDs	USB Flash Drives
Performance	Slower	Faster
Endurance	Longer life expectancy	Longer life expectancy
Design Complexity	More complex	Less complex

Wear Leveler in Flash Memory S/W

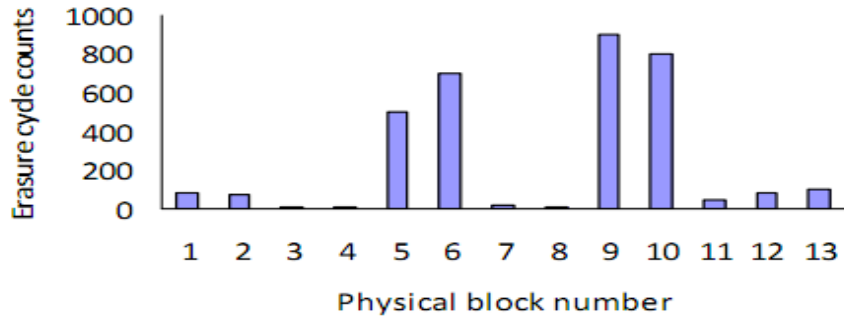
A : Wear Leveling in the FTL



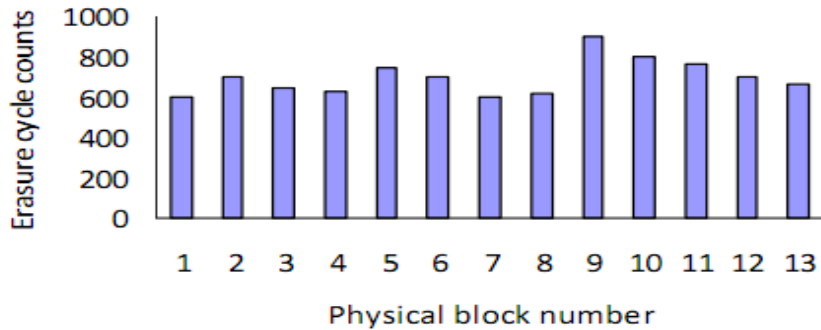
B : Wear Leveling in the File System



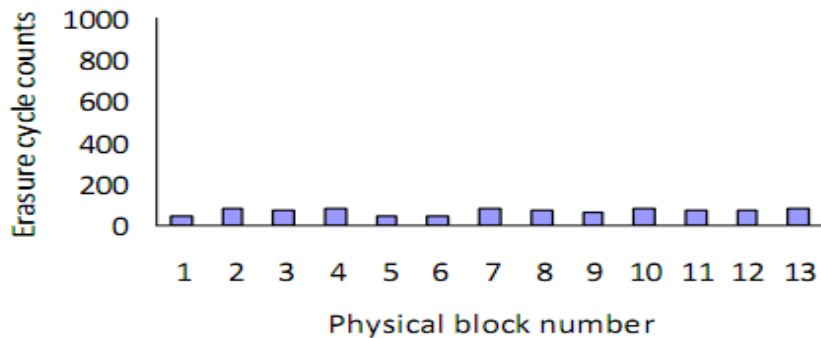
Ideal Wear Leveler



(a) No wear leveling or wear leveling has no effect



(b) Wear-leveling activities introduce too much traffic



(c) The desired result. The overall lifetime is prolonged.

Example of a Dynamic Wear-Leveling Technique

- Wear-level Aware GC
 - When a garbage collection is triggered
 - Choose a victim block
 - Having a small number of erasure counts
 - => **Victim Selection Policy is important**
- A Representative Victim Selection Policy
 - Cost-Age-Time (CAT)

Cost-Age-Time (CAT) Policy

- Principle
 - Chooses a block which minimizes the equation below

$$\frac{\text{Cost}}{\text{Benefit}} * \text{Time} = \frac{u}{1-u * \text{Age}} * \text{EC}$$

* u : utilization of the block

* Time (EC) : total erase count of the block


* Age : the most recent modified time of any page in the block

- Pros
 - Performs well with locality
- Cons
 - Computation/data overhead

Examples of Static Wear-Leveling Techniques

- Operates in a hidden fashion.
- Triggered when a gap between old and young blocks gets big
- Representative Techniques
 - Hot/Cold Swapping
 - Dual Pool Algorithm

Hot-Cold Swapping (1)

 = Block

Old block : Erase count \uparrow

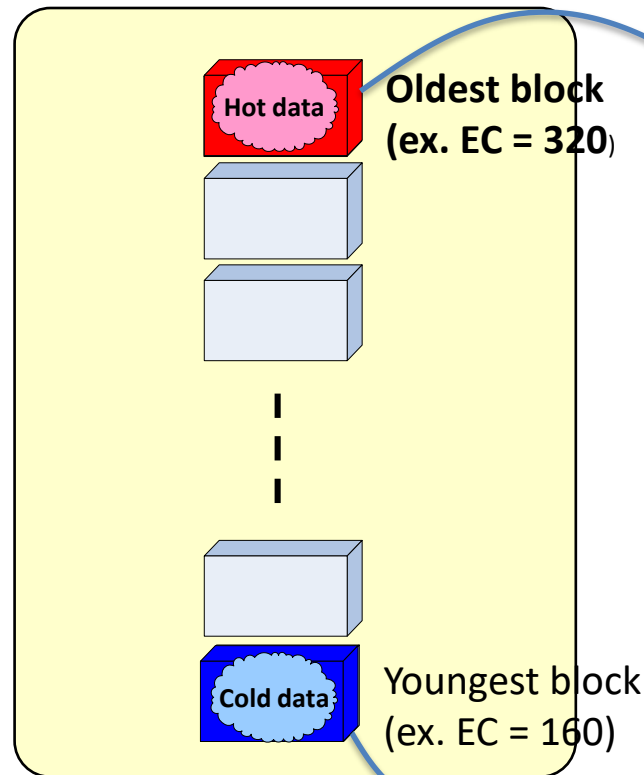
Young block : Erase count \downarrow

EC = Erase Count

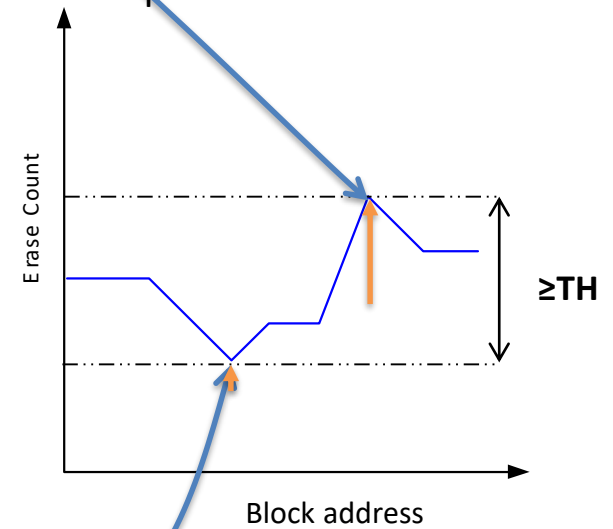
Hot data : frequently accessed data

Cold data : hardly accessed data

Blocks are sorted
by erase count
(descending order)



Assume that difference is made by
'temperature' of data



Triggered when ΔEC exceeds
certain threshold

Hot-Cold Swapping (2)



= Block

Old block : Erase count \uparrow

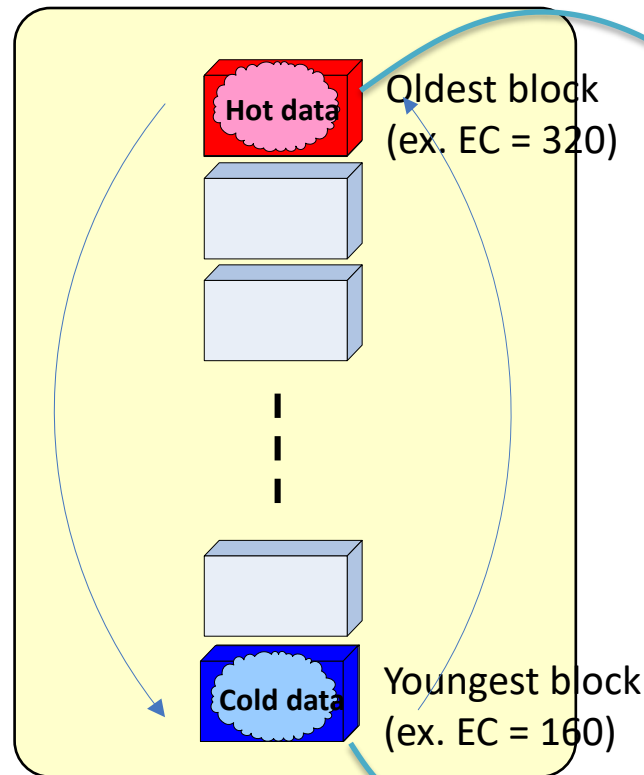
Young block : Erase count \downarrow

EC = Erase Count

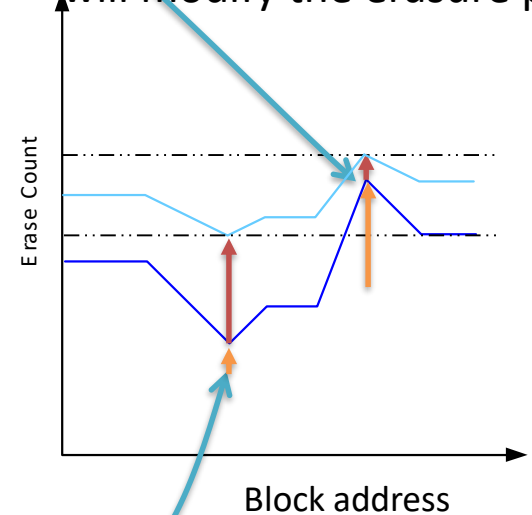
Hot data : frequently accessed data

Cold data : hardly accessed data

Blocks are sorted
by erase count
(descending order)

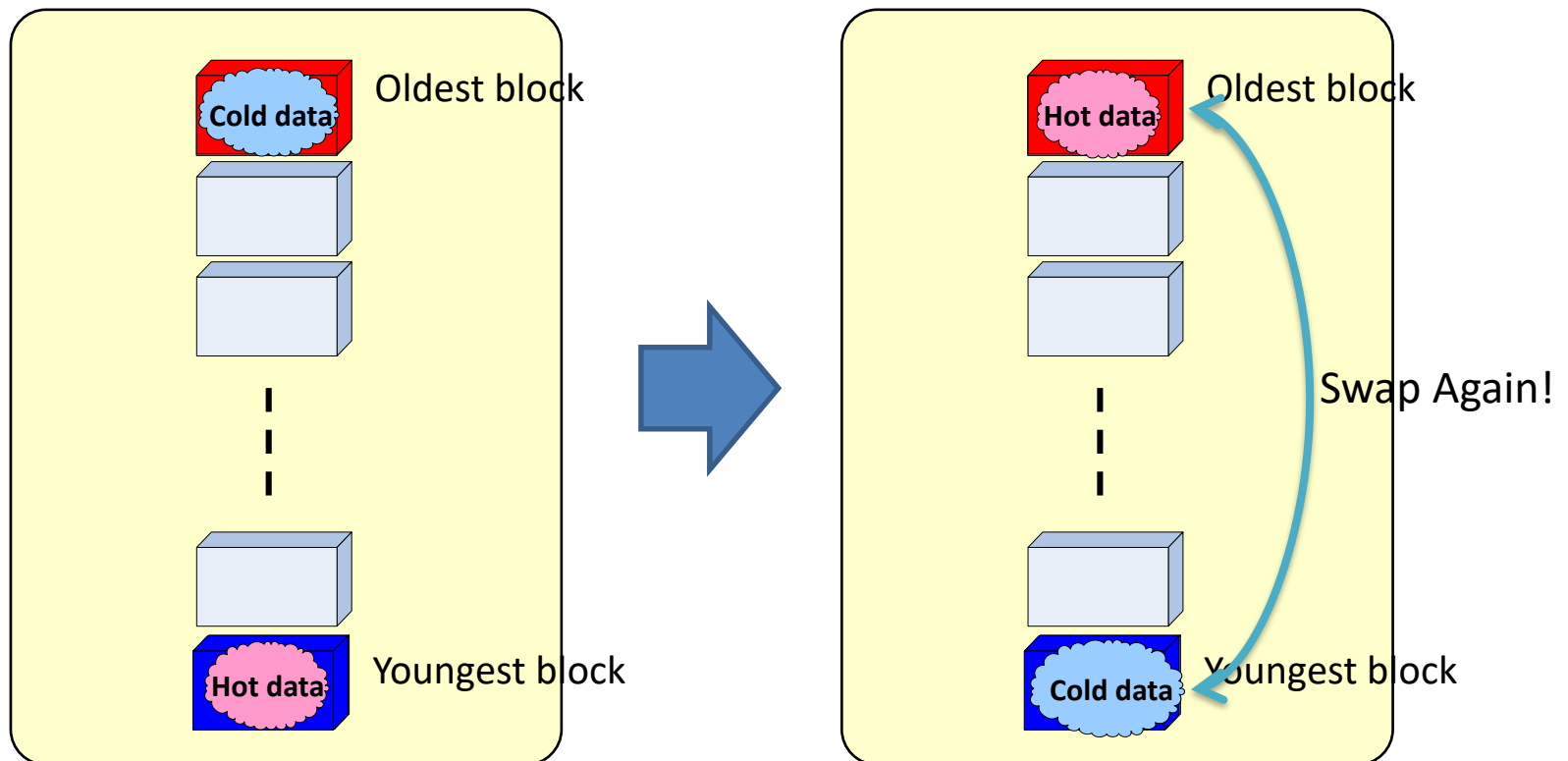


Swapping 'Hot' and 'Cold' data
will modify the erasure pattern



Problem of Hot-Cold Swapping

- The oldest data may be involved repeatedly in the hot-cold swapping
 - e.g. If the hotness of the swapped data are changed

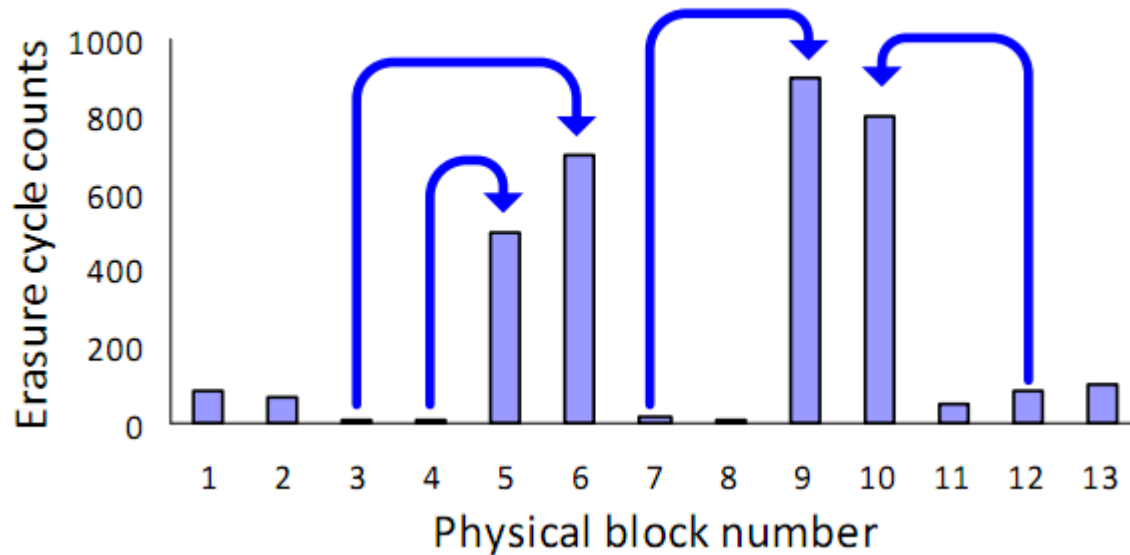


Dual-Pool Algorithm

- Considers the different aspects of wear leveling together
 - Effectiveness (to evenly erase blocks)
 - Cold-data migration
 - Efficiency (to reduce traffic introduced by wear leveling)
 - Dual-pool organization
 - Scalability (to have low resource requirements)
 - Low overhead implementation

Basic Idea: Cold-Data Migration

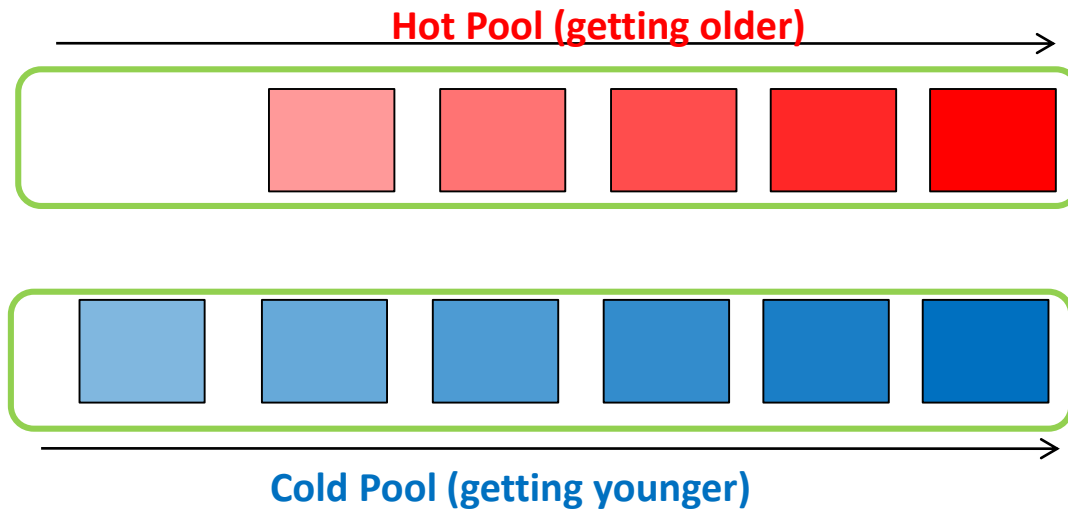
* The majority of data are cold, as the majority of blocks are young!!



- **Migrating data from young blocks to old blocks**
 - To defrost young blocks by moving cold data away
 - To cool down old blocks by moving cold data in

Dual-Pool Organization

- Maintain two pools, Hot Pool and Cold Pool.
 - Hot Pool (sorted by the increasing age)
 - Cold Pool (sorted by the decreasing age)

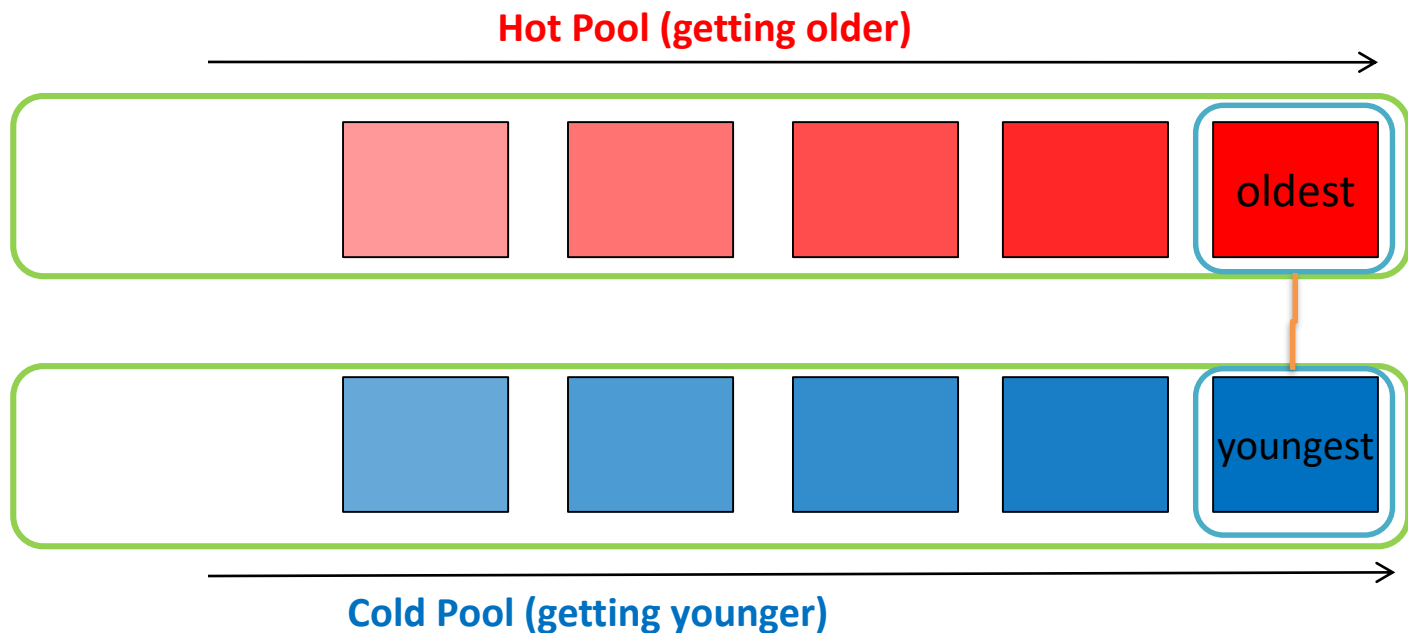


Cold-Data Migration (1)

- Trigger condition
 - On the completion of an erasure request
 - If the difference of erasure cycles (EC) between the oldest block in Hot Pool and the youngest block in Cold Pool is greater than a preset threshold
$$\text{MAX_EC_HOT_POOL} - \text{MIN_EC_COLD_POOL} > \text{TH}$$
- Cold data migration
 - Move valid pages in the oldest block to a block
 - Move valid pages in the youngest block to the oldest block
 - Erase the youngest block

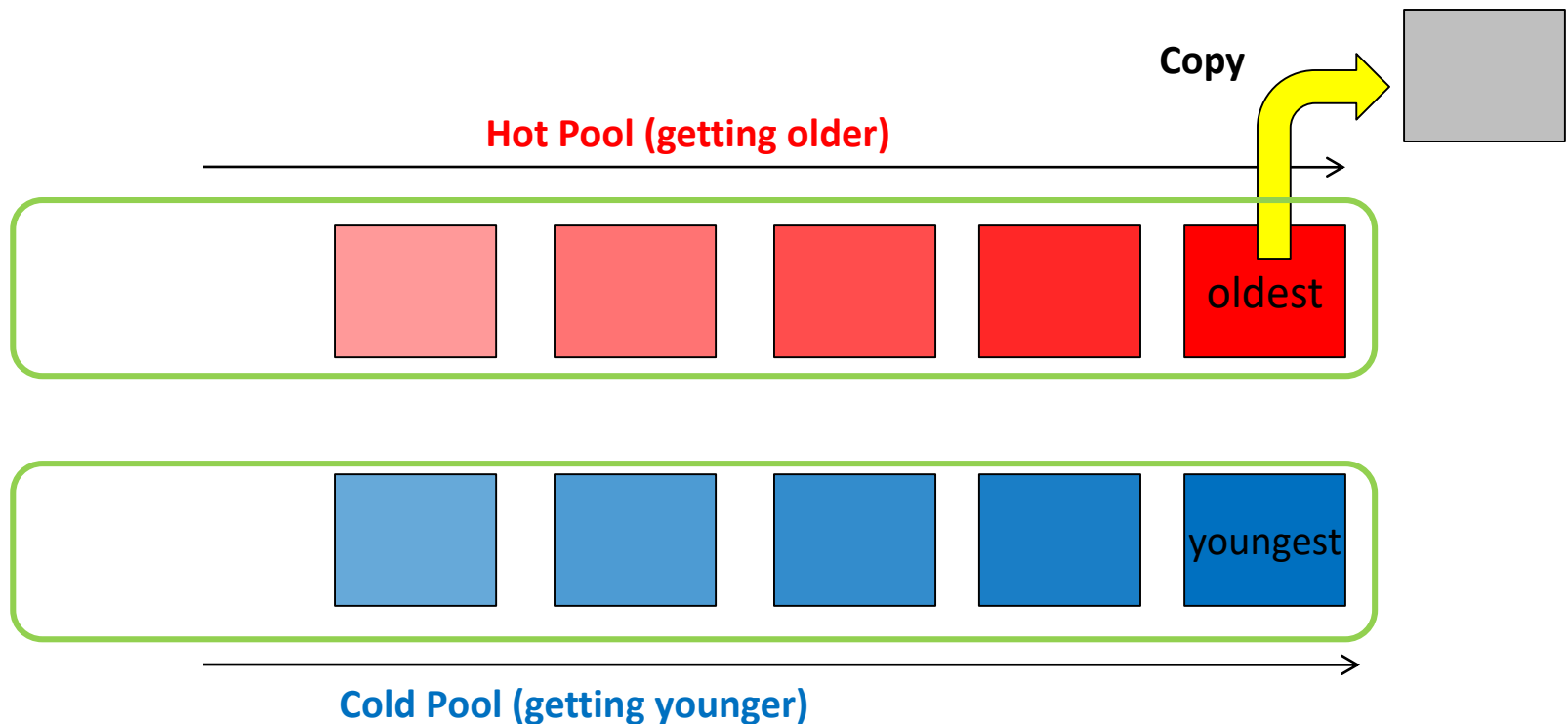
Cold-Data Migration (2)

- Check the erasure cycles of the oldest block and the youngest block



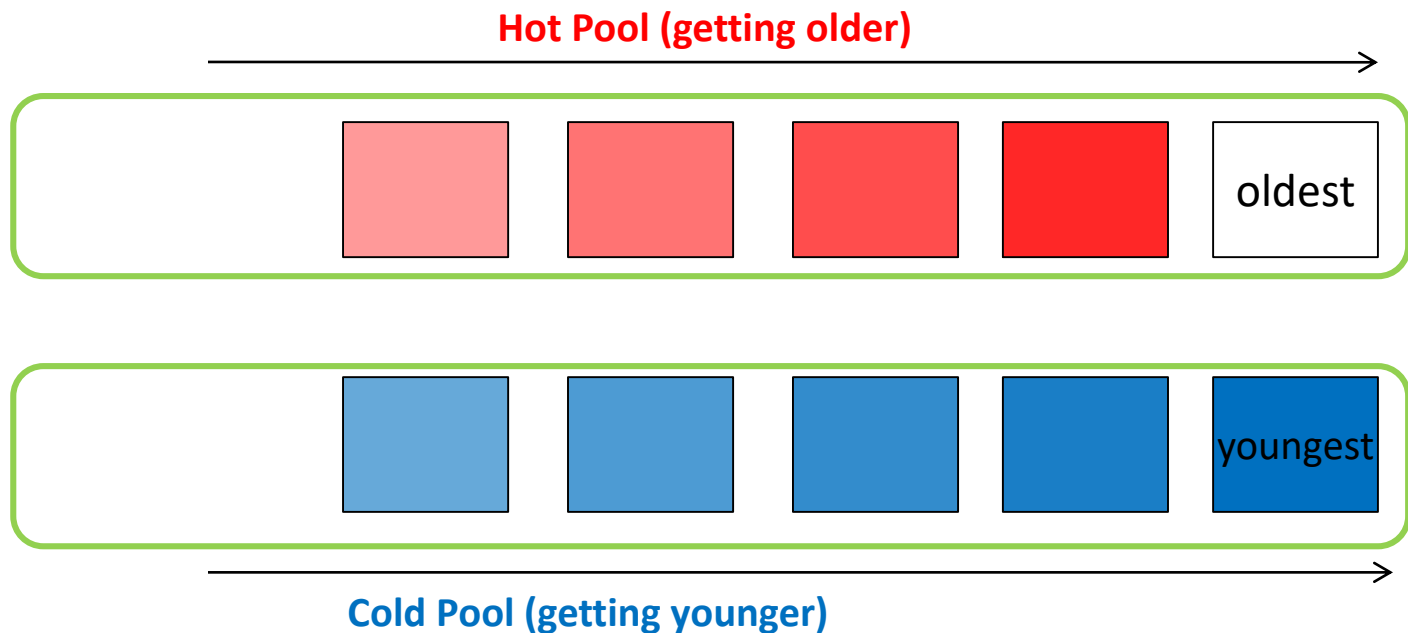
Cold-Data Migration (3)

- Copy valid data in the oldest block in Hot Pool to a different block



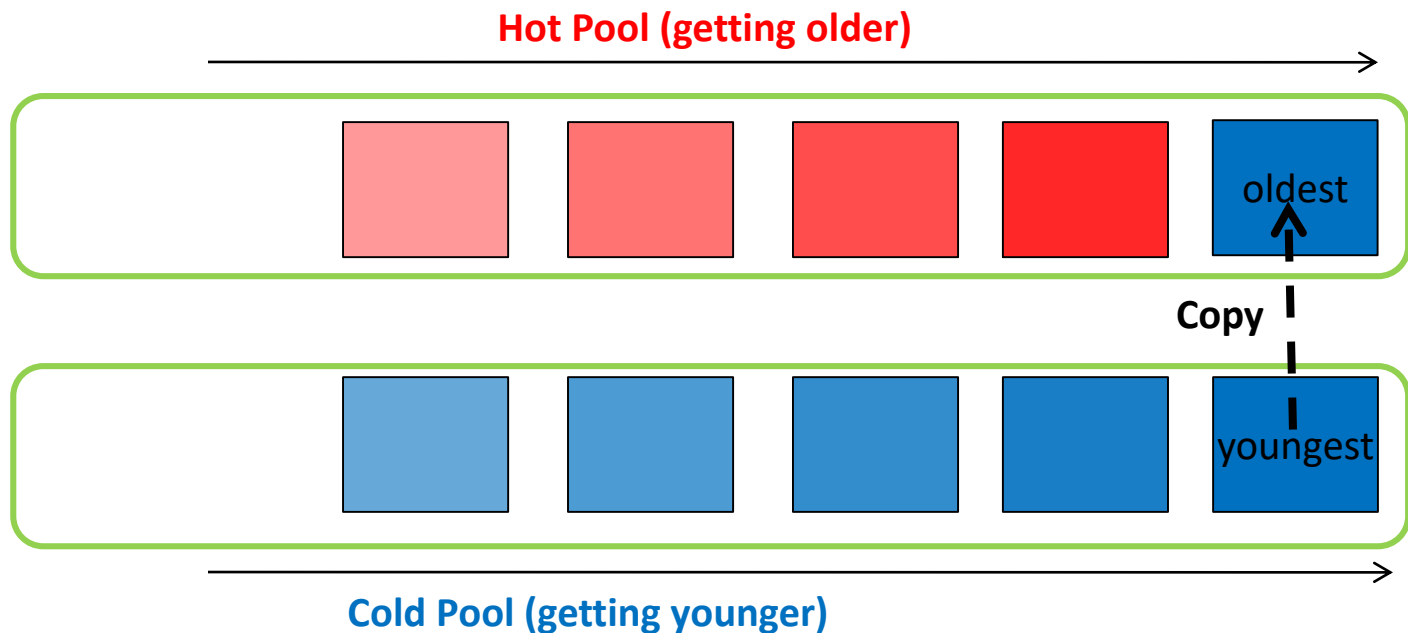
Cold-Data Migration (4)

- Erase the oldest block



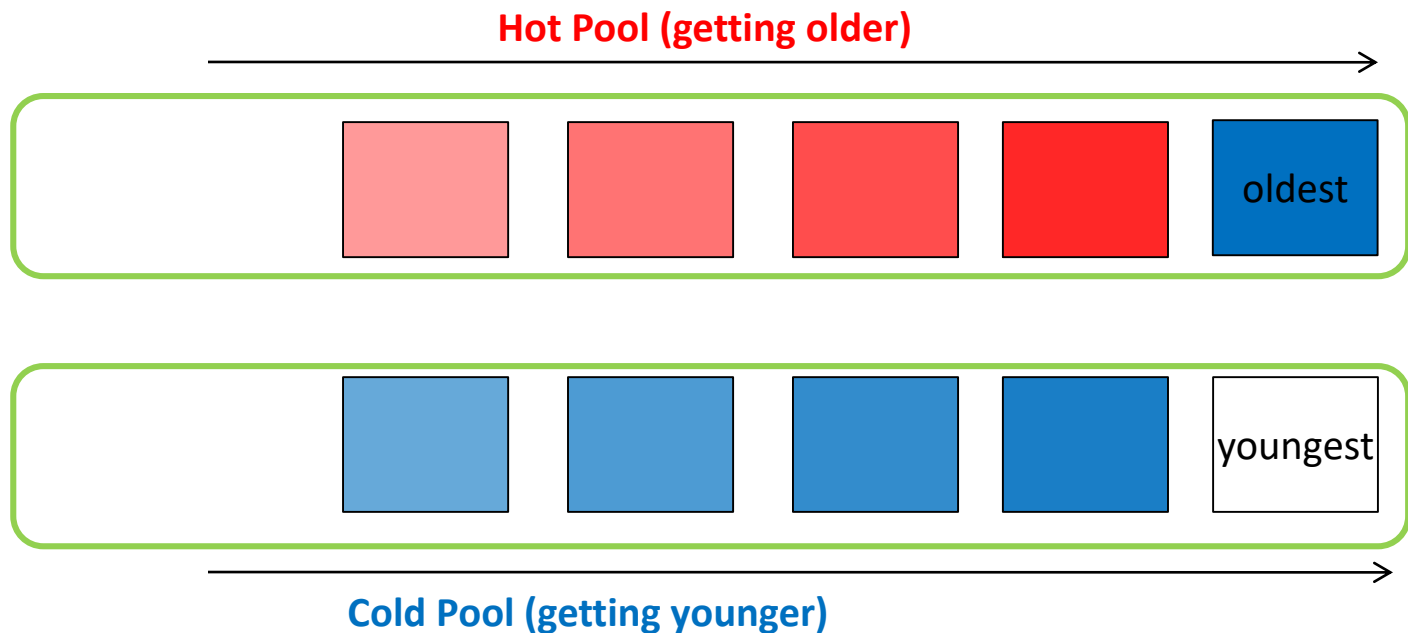
Cold-Data Migration (5)

- Copy valid data in the youngest block in Cold Pool to the oldest block in Hot Pool



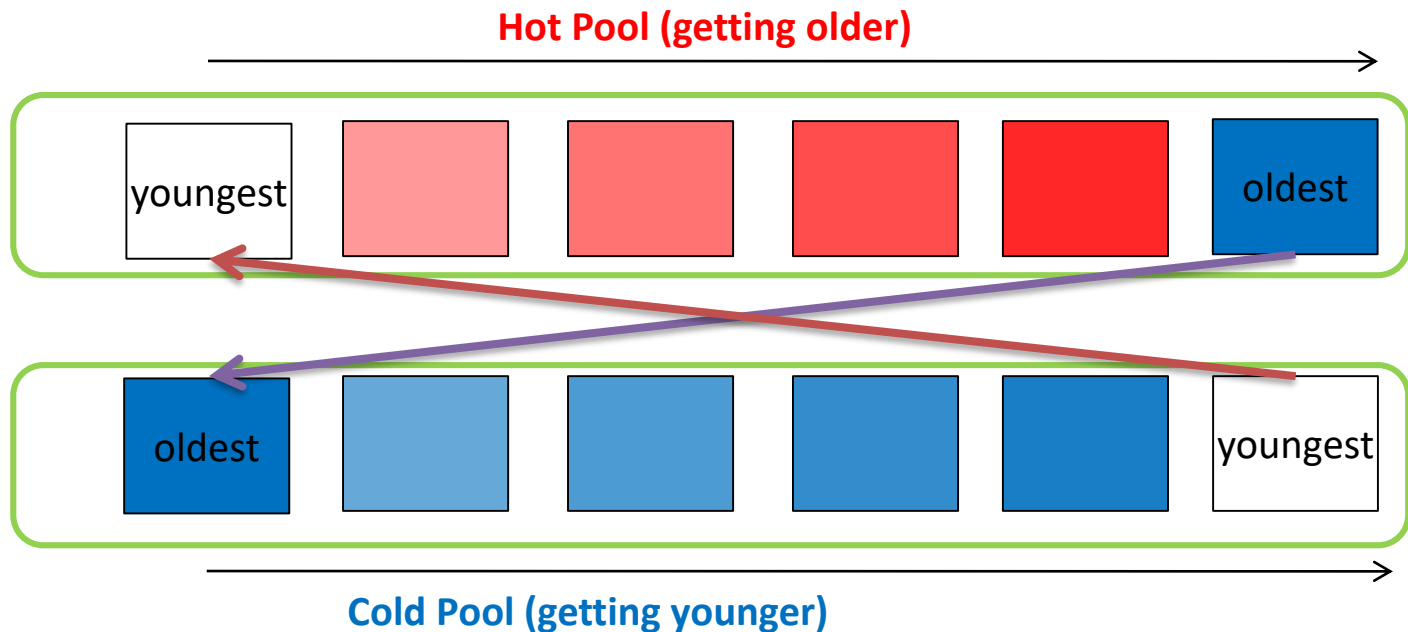
Cold-Data Migration (6)

- Erase the youngest block in Cold Pool



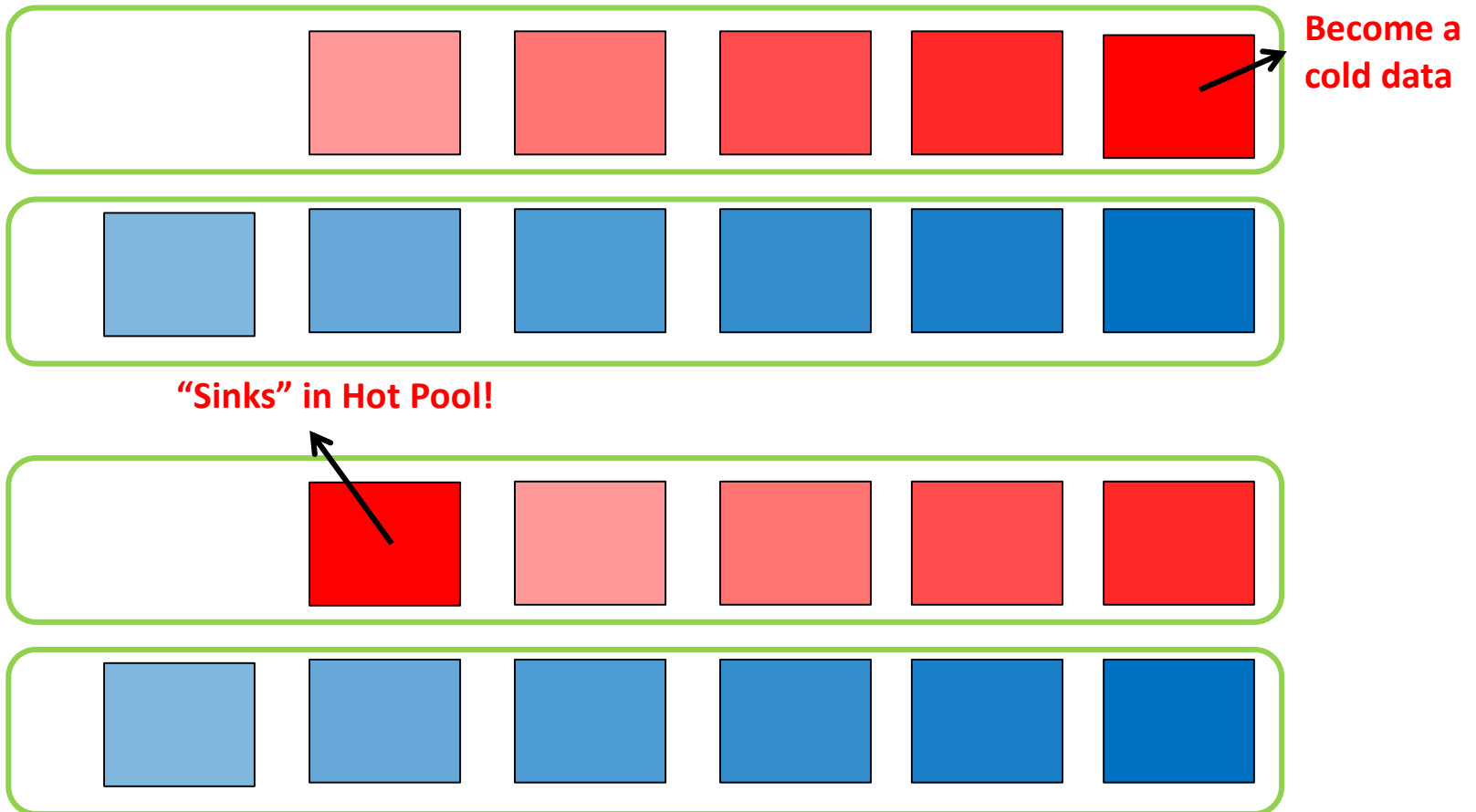
Basic Idea: Hot-Cold Regulation

- Once an old block receives a cold data from a young block, the old block MUST move to Cold Pool.
 - Within Cold Pool, this block becomes the Oldest.
 - Therefore, this block cannot be involved in cold data migrations for a while.
 - In turn, this gives the old block the time to be cooled down.



Adaptive Pool Resizing (1)

- How to deal with dynamic changes in data hotness?



Adaptive Pool Resizing (2)

- For dynamic changes in data hotness, the blocks with opposite type of data to opposite pools.
- Case 1: when **cold data** in **Cold Pool** become **hot**
 - Cold Pool Adjustment
 - Move the block to Hot Pool
- Case 2: when **hot data** in **Hot Pool** become **cold**
 - Hot Pool Adjustment
 - Move the block to Cold Pool

Q: How to Choose a Block to Move?

Q: When **hot data** in **Hot Pool** become
cold ?

- Stuck in Hot Pool
- Should be moved back to Cold Pool for wear leveling

Q: When cold data in Cold Pool become
hot ?

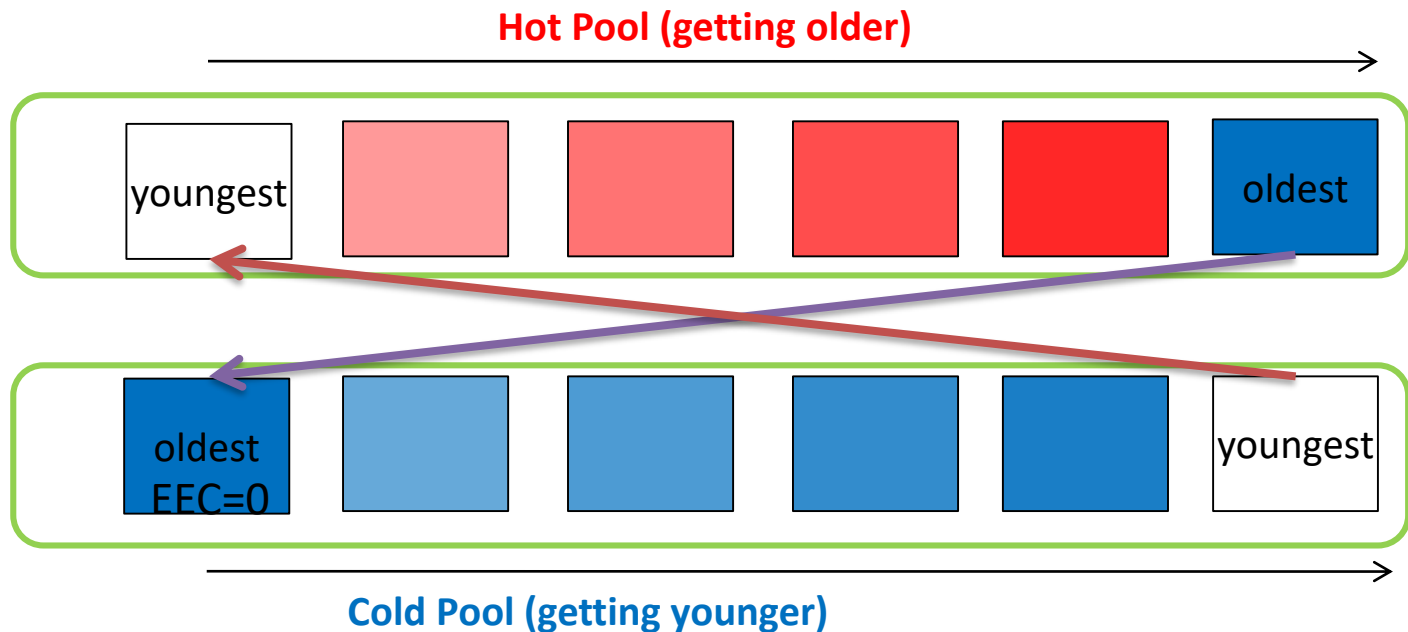
- No chance for wear leveling
- Should be moved back to Hot Pool for wear leveling

Cold Pool Adjustment

- **Idea**
 - Move a block with hot data to Hot Pool
- **Term**
 - **Effective Erasure Cycle (EEC)**: How many times a block is erased since the last time the block is involved in Cold-data migration
- **Trigger condition**
 - On the completion of an erasure request
 - If the difference of the **EECs** between the block with the maximal **EEC** in **Cold Pool** and the block with the minimal **EEC** in **Hot Pool** is greater than a preset threshold
$$\text{MAX_EEC_COLD_POOL} - \text{MIN_EEC_HOT_POOL} > \text{TH}$$
- **Block Migration**
 - Select the block with maximal effective erasure
 - Migrate the block to Hot Pool

Effective Erasure Cycle (EEC)

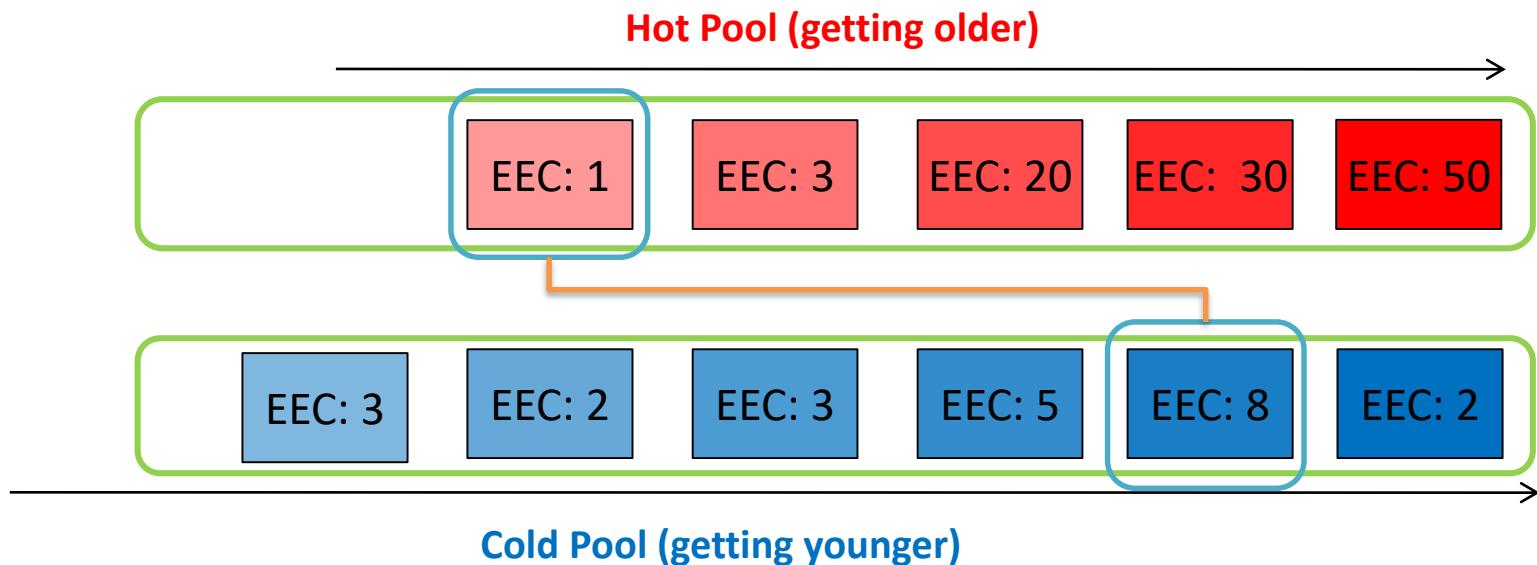
- EEC of a block is reset with 0 when the block is moved to Cold Pool during Cold-Data Migration.



- EEC of a block is increased whenever the block is erased.
- Use ΔEEC to estimate the hotness of a block

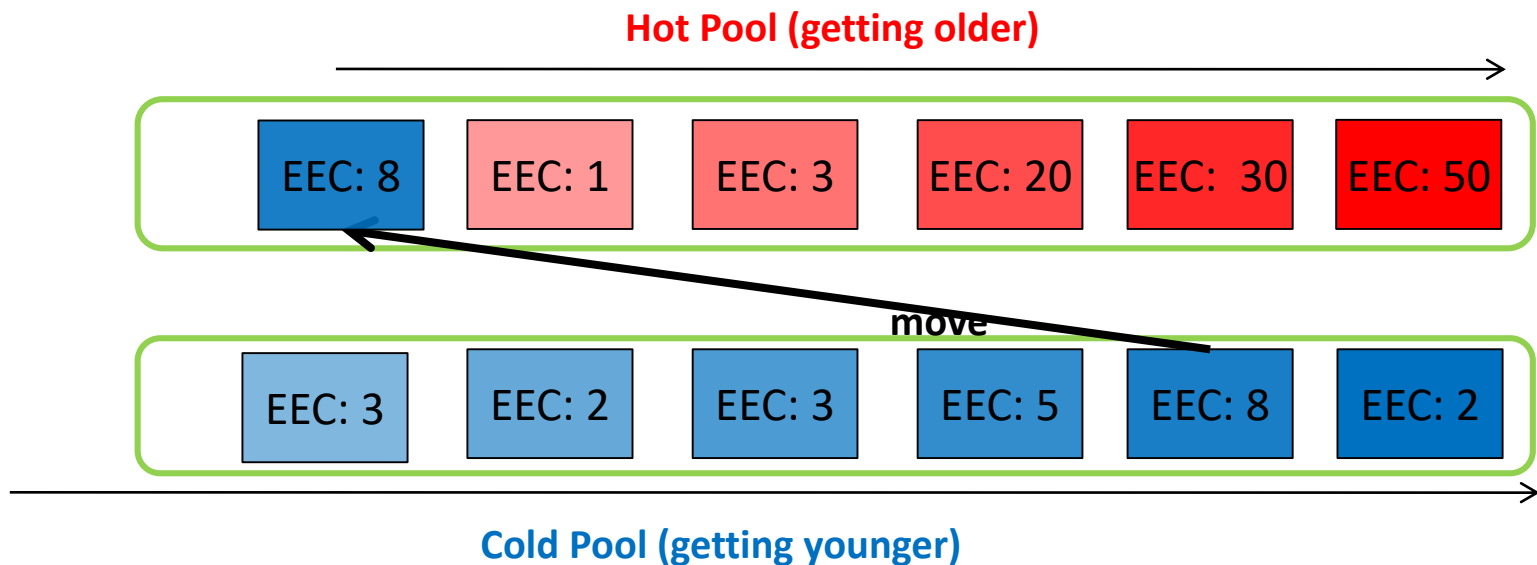
Example of Cold Pool Adjustment (1)

- Threshold: 4
- EEC: Effective Erasure Cycle



Example of Cold Pool Adjustment (2)

- Move the block with the largest EEC to Hot Pool

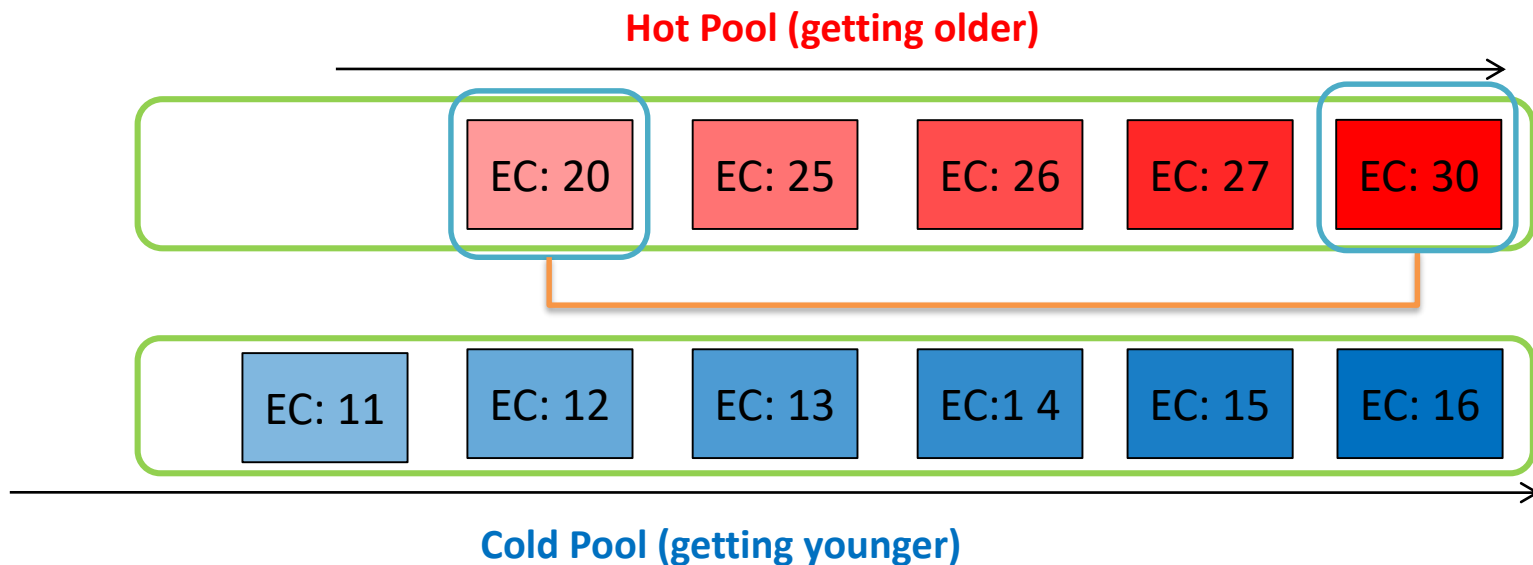


Hot Pool Adjustment

- **Idea**
 - Move a block with cold data to Cold Pool
- **Trigger condition**
 - On the completion of an erasure request
 - If the difference of erasure cycles between the oldest and the youngest blocks in Hot Pool is **over twice** than a preset threshold
$$\text{MAX_EC_HOT_POOL} - \text{MIN_EC_HOT_POOL} > (2 \times \text{TH})$$
- **Block Migration**
 - Select the youngest block
 - Migrate the block to Cold Pool

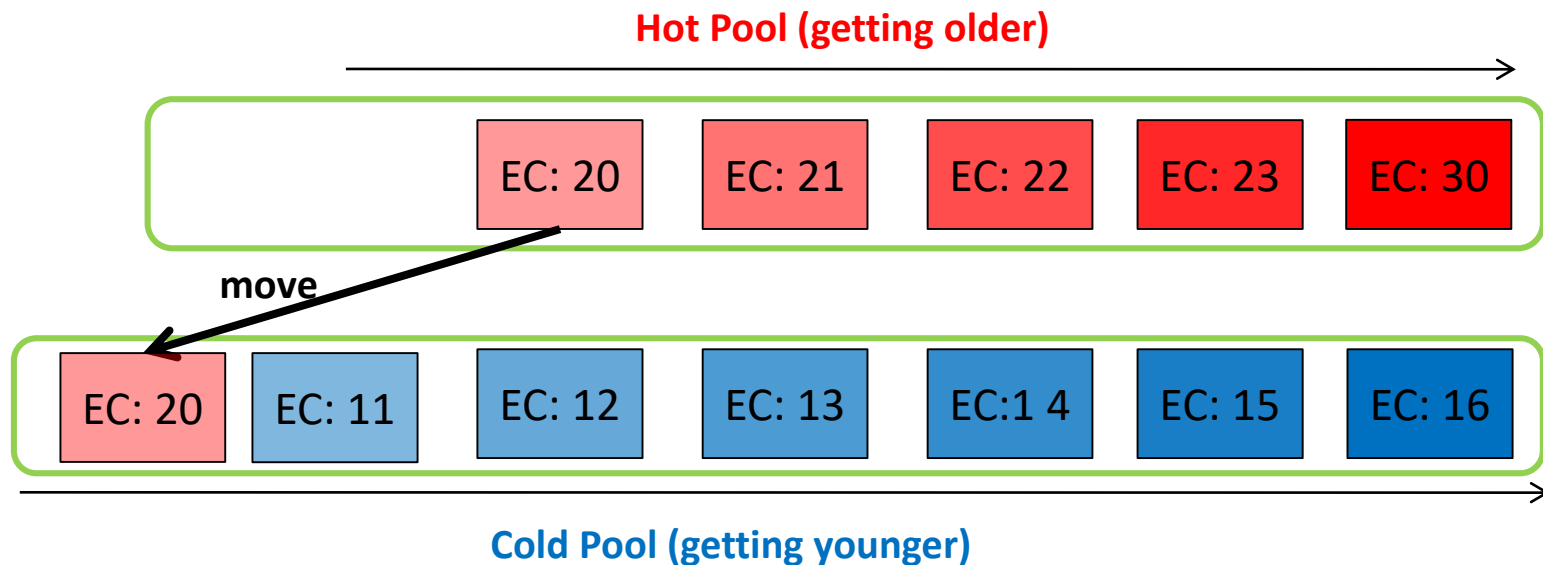
Example of Hot Pool Adjustment (1)

- Example:
 - Threshold: 4
 - $4 \times 2 = 8$
 - EC: Erasure Cycle



Example of Hot Pool Adjustment (2)

- Migrate the youngest block in Hot Pool to Cold Pool

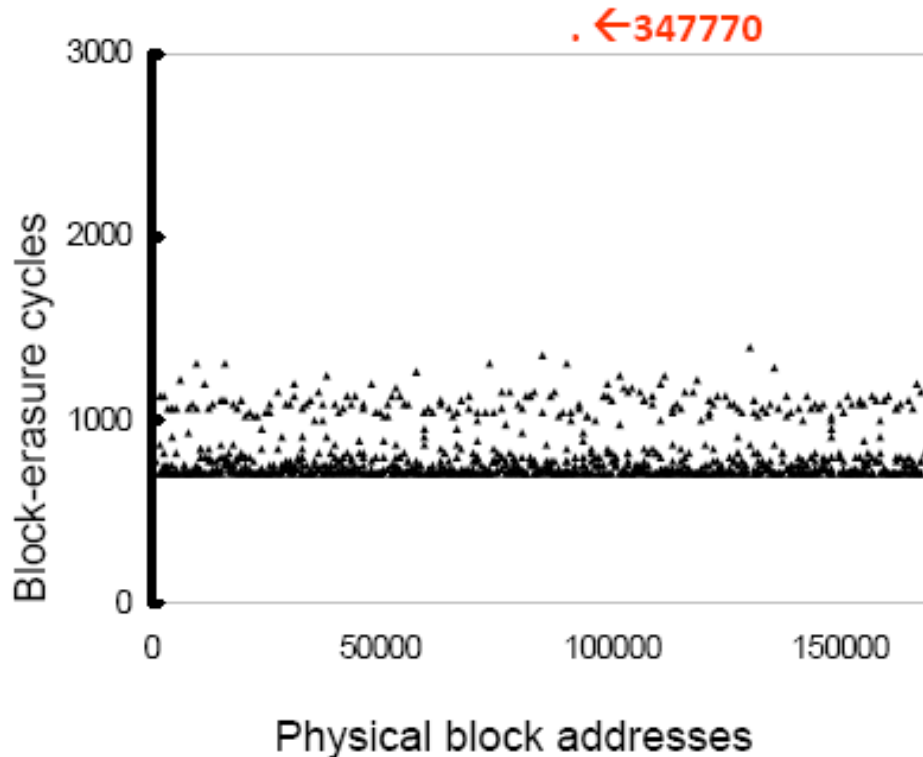


Experimental Setup

- Performance comparison against the previous approaches
- NAND flash characteristics
 - K9NBG08U5M 4GB NAND flash memory, Samsung
- Trace-driven simulation
 - The traces were collected from a real mobile PC for 1 month
 - The traces were replayed 100 times to emulate the use of a couple of years
- Garbage collection algorithm: the greedy policy

Experimental Results (1)

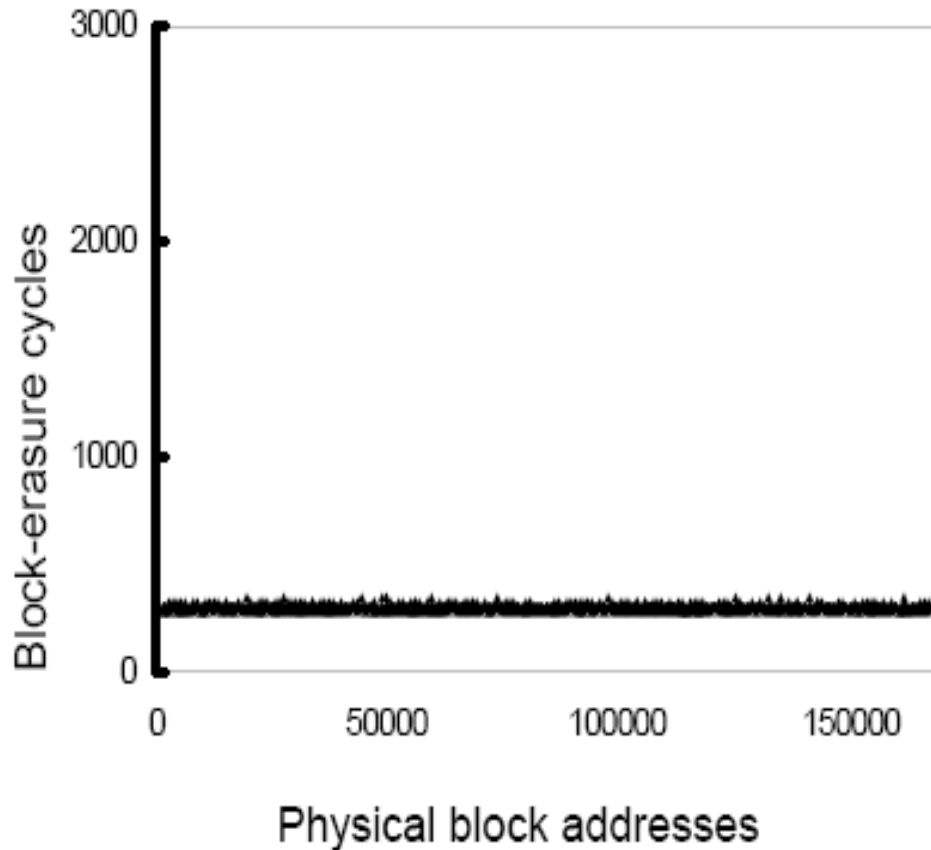
- Hot-Cold Swapping Algorithm
 - Swap data in the oldest block and the youngest block
 - M-System TrueFFS (based on Hot/cold Swapping Technique)



- An old block was constantly involved in the swapping of data
 - The erasure cycle had even achieved **347,770** (beyond the boundary of the figure)

Experimental Results (2)

- Dual Pool Algorithm



- Block-erase cycles were small, and they were close to one another

References

- http://en.wikipedia.org/wiki/Wear_leveling
- Li-Pin Chang, “On Efficient Wear Leveling for Large-Scale Flash Memory Storage Systems, ” SAC, 2007
- Prof. Sang-Lyul Min, “Advanced Computer Architecture.”, Lecture Notes in Seoul National University
- Chiang, M., Lee, P., and Chang, R. “Using data clustering to improve cleaning performance for flash memory,” *Softw. Pract. Exper.*, vol. 29, pp. 267-290, 1999
- Cactus Technologies, Application Note. 19 Sep. 2008
- STMicroelectronics, Application Note. 11 May. 2004

References

- 백승훈, “MLC 스토리지의 신뢰성/수명 이슈,” NVRAMOS, 2010
- Li-Pin Chang et. al. “A Low-Cost Wear-Leveling Algorithm for Block-Mapping Solid-State Disks,” LCTES, 2011