5. Storage Firmware (Part 2)

Special Topics in Computer Systems:
Modern Storage Systems
(IC820-01)

Instructor:

Prof. Sungjin Lee (sungjin.lee@dgist.ac.kr)

Outline

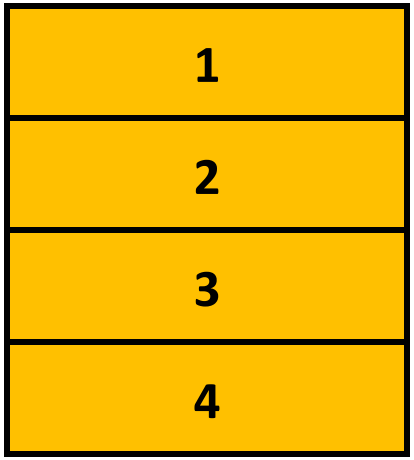
- **Review: Replacement Block Scheme**
- Hybrid FTLs (Log Block Scheme)
- Demand-based FTL (DFTL)

Replacement Block Scheme

- **Idea**
  - A data block has a chain of write buffer blocks called replacement blocks
  - Mapping within a replacement block is managed in block-level

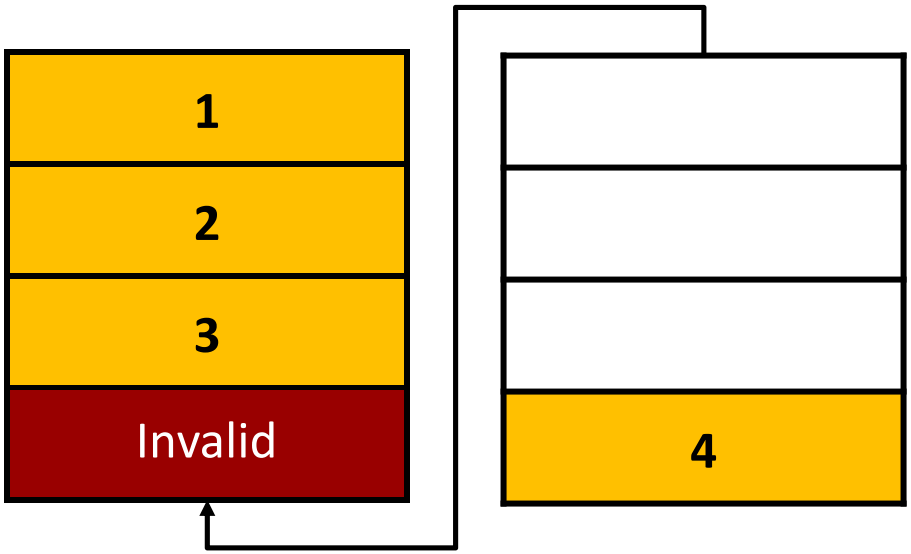
Replacement Block Scheme

- **Replacement-block scheme**
  - Maintain write history between an original block and an updated block
  - E.g., Write trace : 1, 2, 3, 4



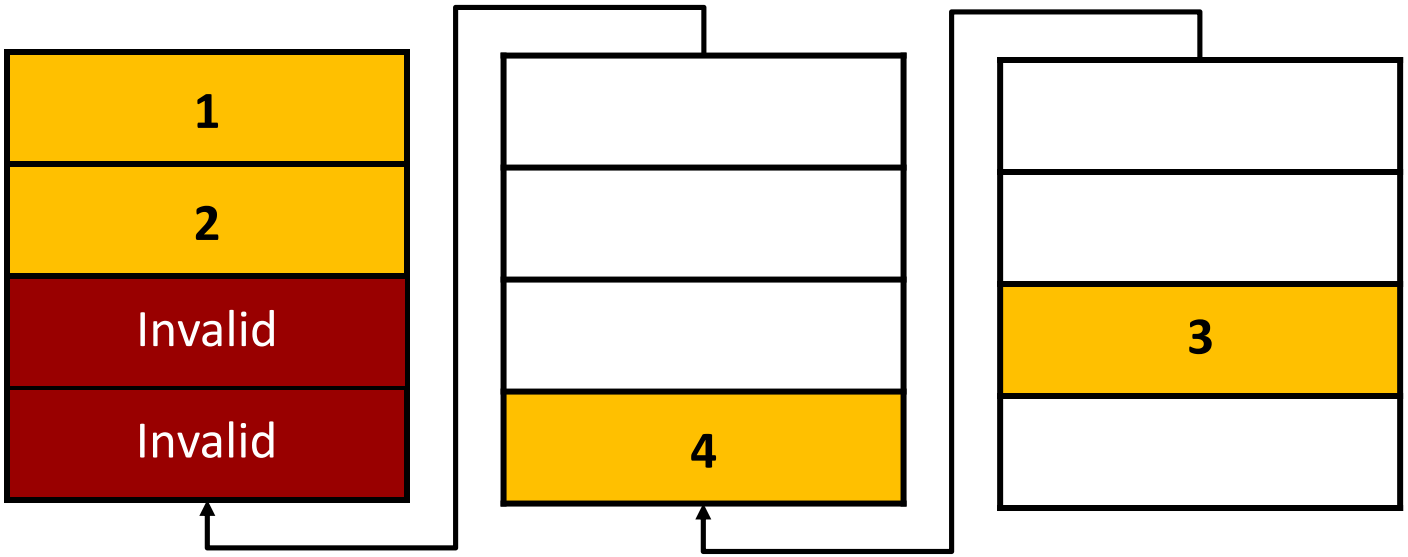
Replacement Block Scheme

- **Replacement-block scheme**
  - Maintain write history between an original block and an updated block
  - E.g., Write trace : 1, 2, 3, 4, 4



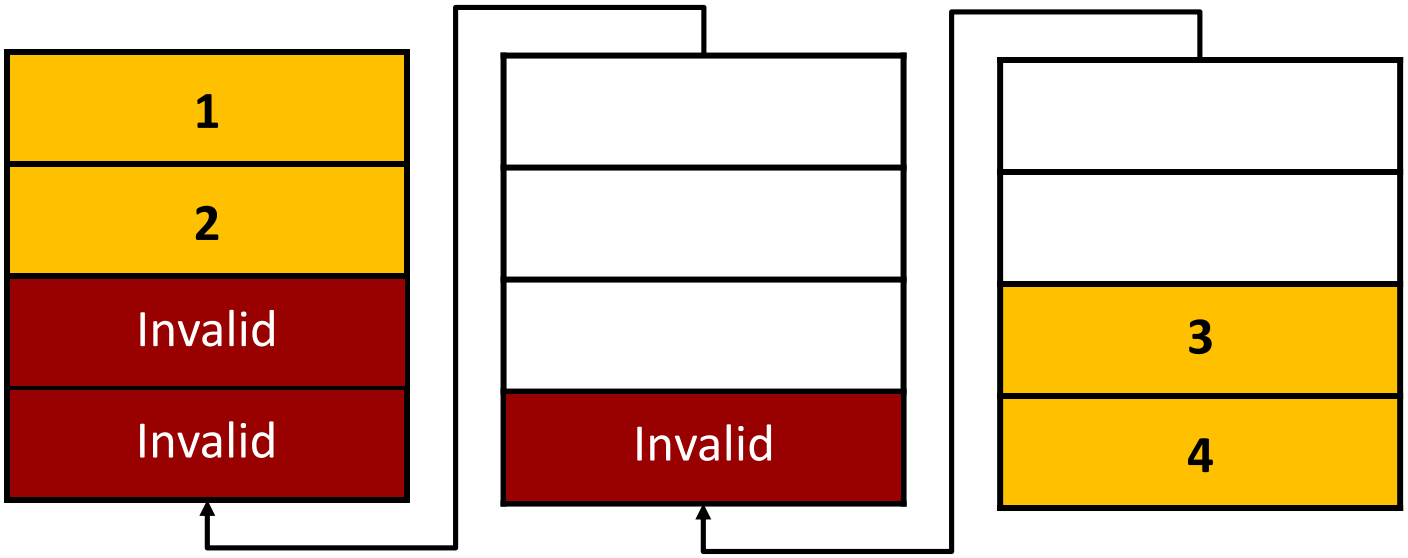
Replacement Block Scheme

- **Replacement-block scheme**
  - Maintain write history between an original block and an updated block
  - E.g., Write trace : 1, 2, 3, 4, 4, 3



Replacement Block Scheme

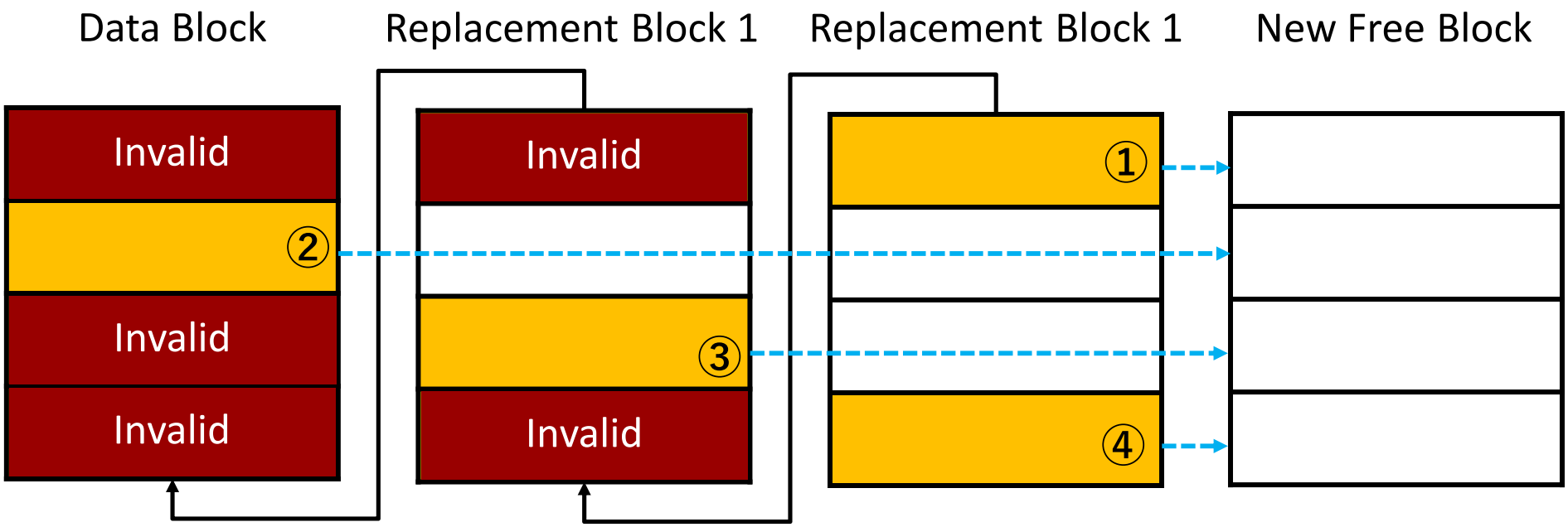
- **Replacement-block scheme**
  - Maintain write history between an original block and an updated block
  - E.g., Write trace : 1, 2, 3, 4, 4, 3, 4



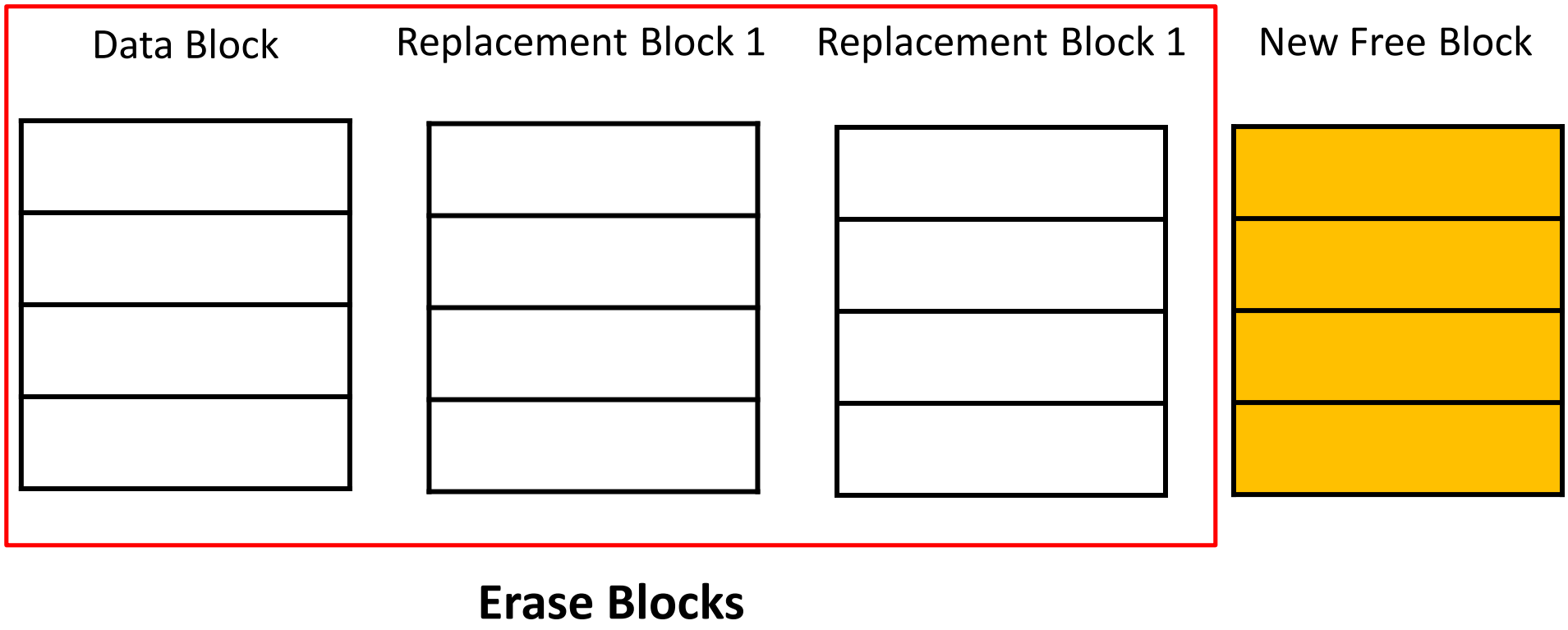
Replacement Block Scheme

- **Merge Operation**
  - Is triggered when there is no free block for a replacement block
  - Gathers valid pages in a data block and write buffer blocks (replacement blocks) to form a single complete data block

Merge Operation

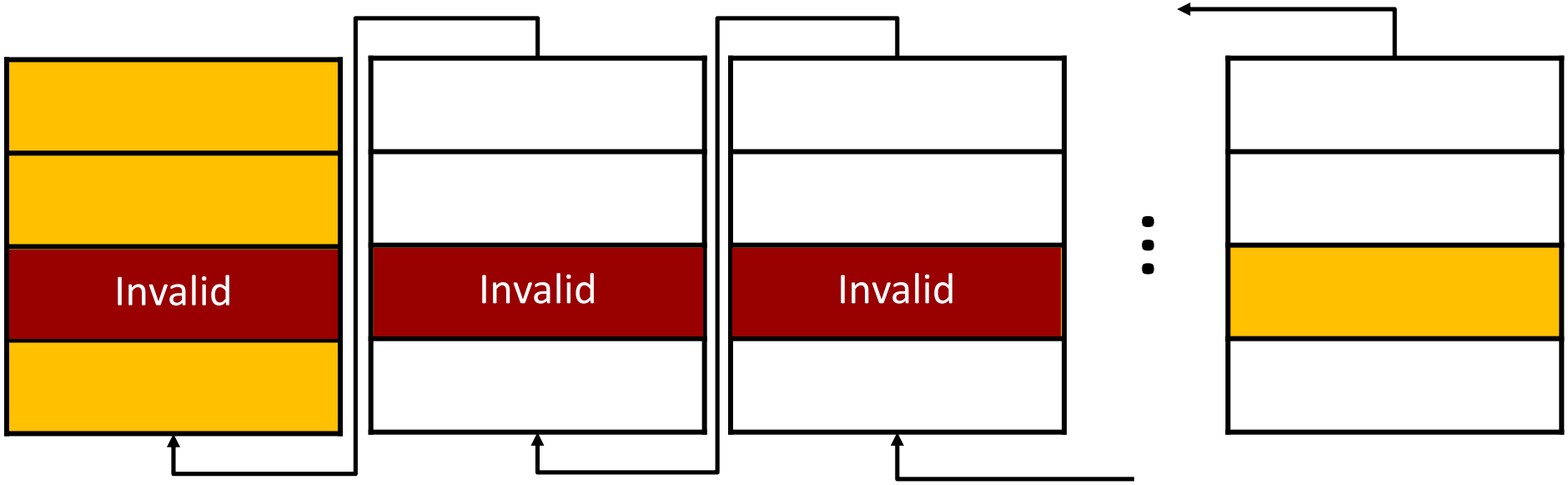


Merge Operation



Replacement Block Scheme

- **Problems**
  - Low utilization of replacement blocks
  - Sequential traverse over replacement blocks during reads and writes
  - No consideration for sequential programming constraint

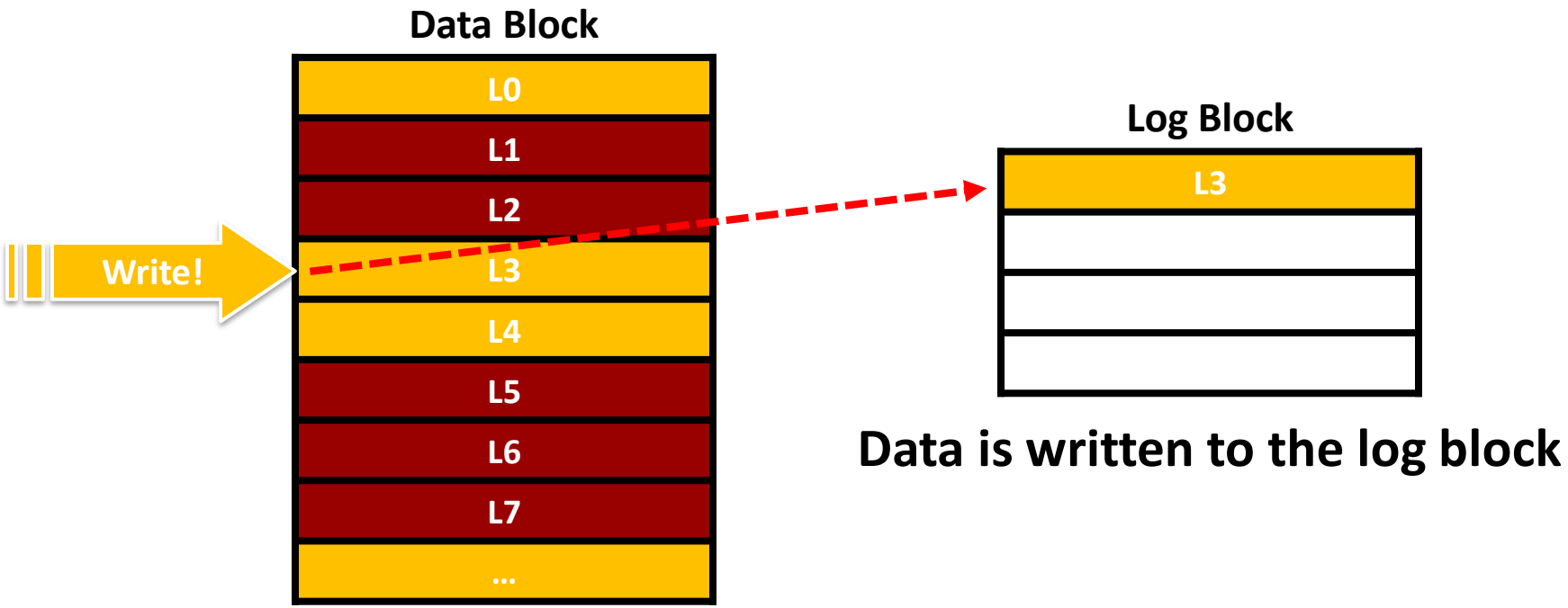


Outline

- Review: Replacement Block Scheme
- **Hybrid FTLs (Log Block Scheme)**
- Demand-based FTL (DFTL)

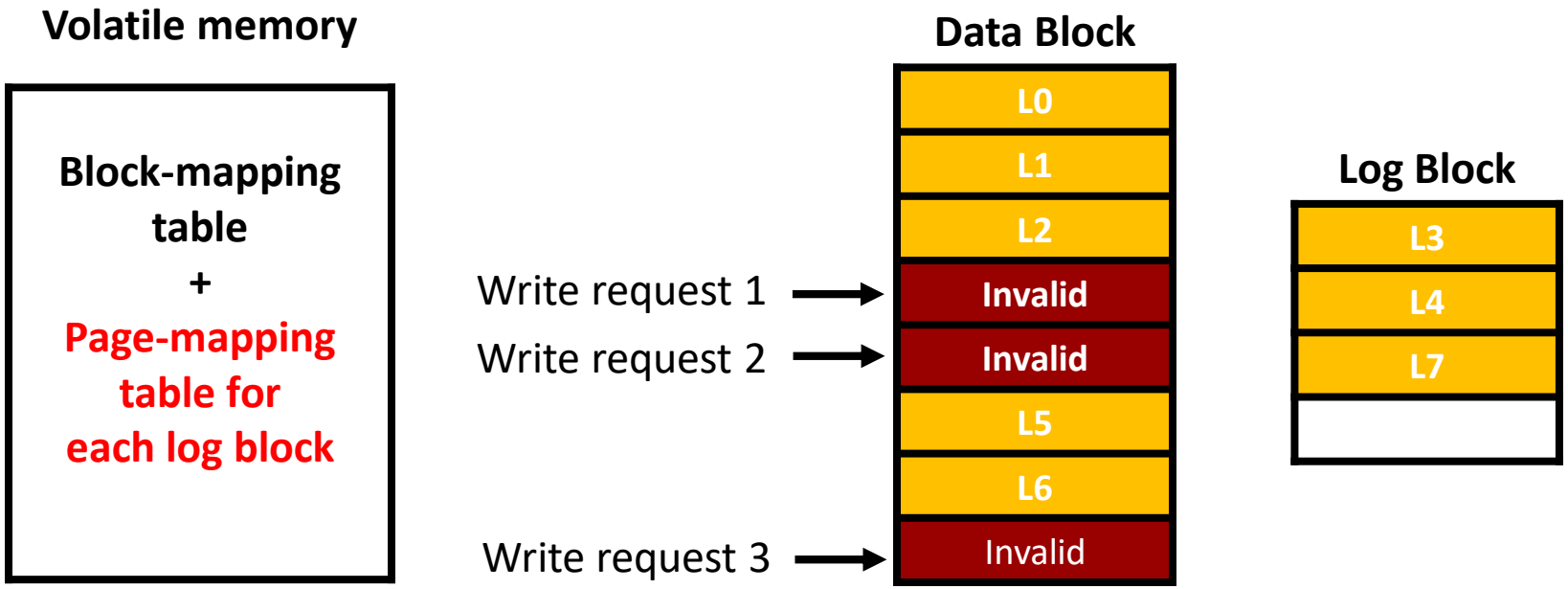
Log Block Scheme

- **Background**
  - 2 kinds of blocks
    - Data block : Block Level managed block (most)
    - Log block : Page Level managed block (a few)
    - Temporary storage for small size writes to data blocks



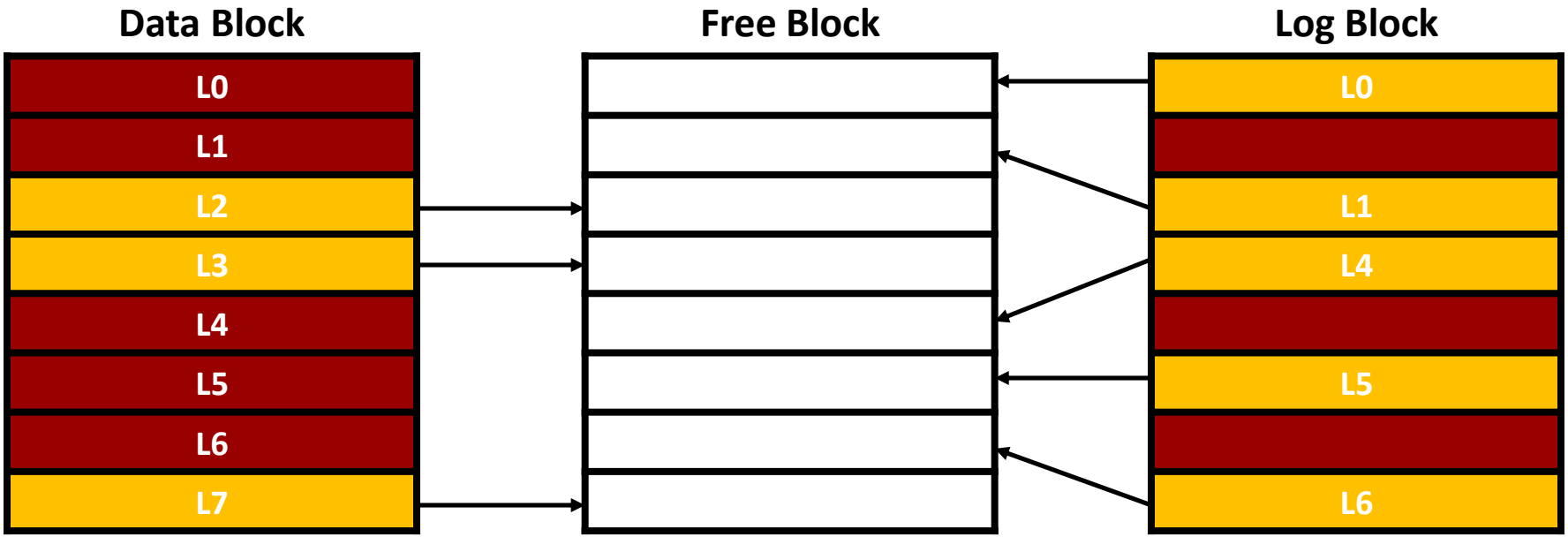
BAST

- **BAST : Block Associative Sector Translation**
- **Key Idea**
  - A certain log block is dedicated to only one data block
  - Mapping within a log block is managed in page level



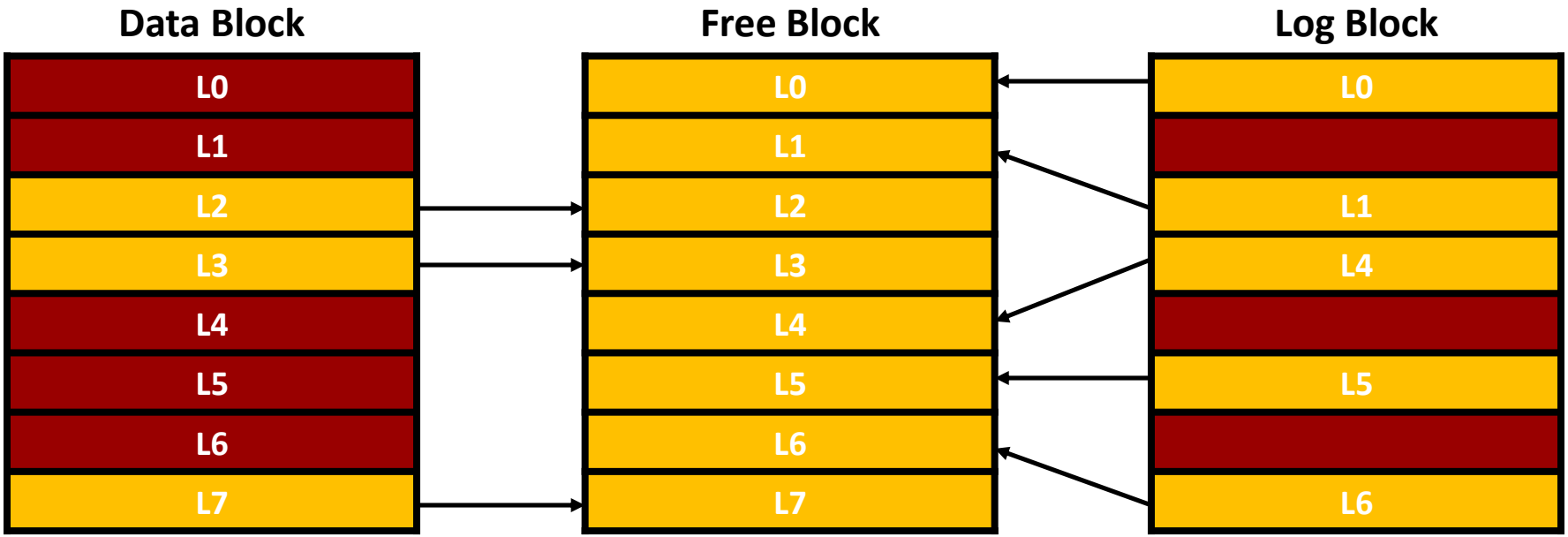
BAST

- **Merge Operation 1**
  - Full merge



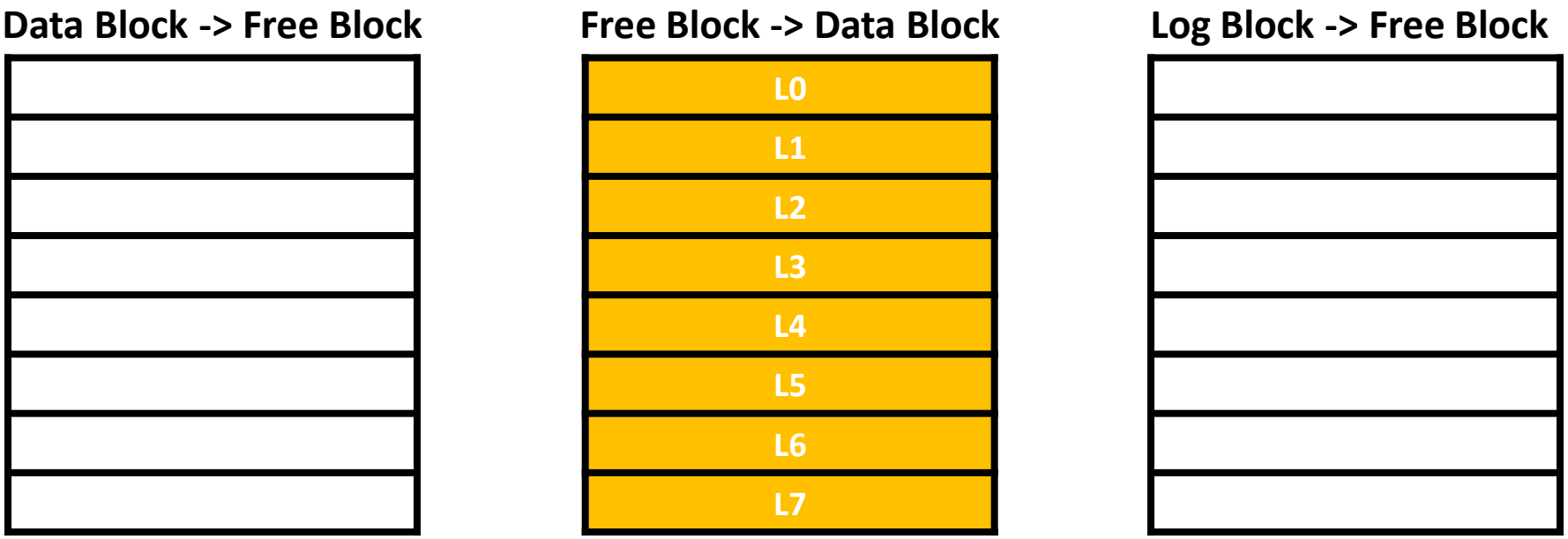
BAST

- **Merge Operation 1**
  - Full merge



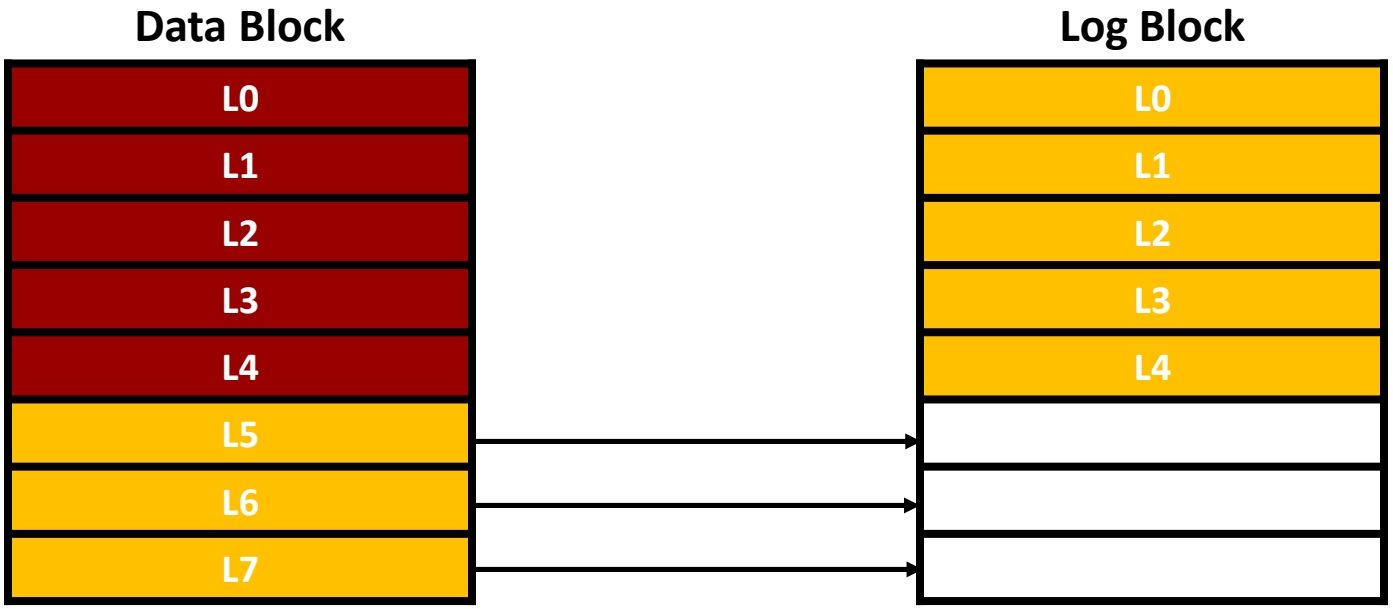
BAST

- **Merge Operation 1**
  - Full merge



BAST

- **Merge Operation 2**
  - Partial merge

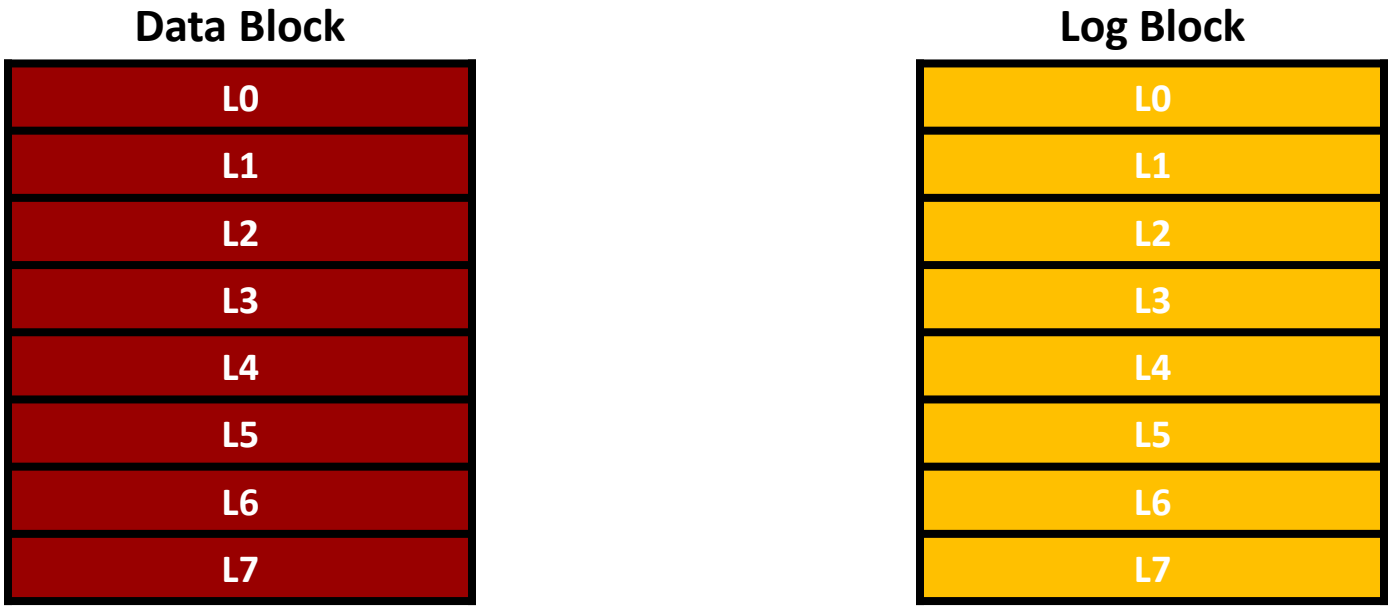


BAST

- **Merge Operation 2**
  - Partial merge

BAST

- **Merge Operation 3**
  - Switch merge



BAST

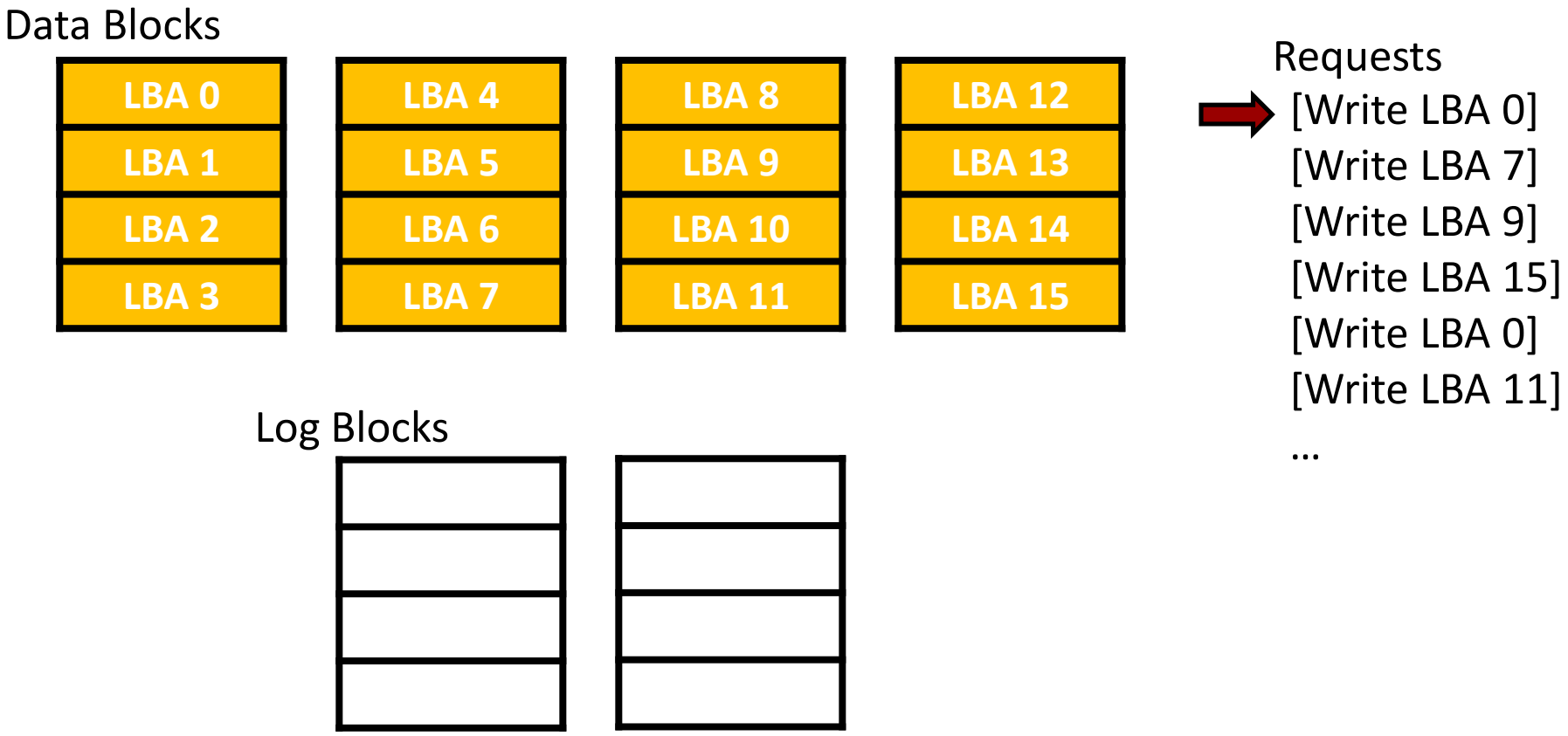
- **Merge Operation 3**
  - Switch merge

BAST

- **Good performance in sequential write**
- **But, frequent merge operation**
  - In random write patterns
  - In complicated application

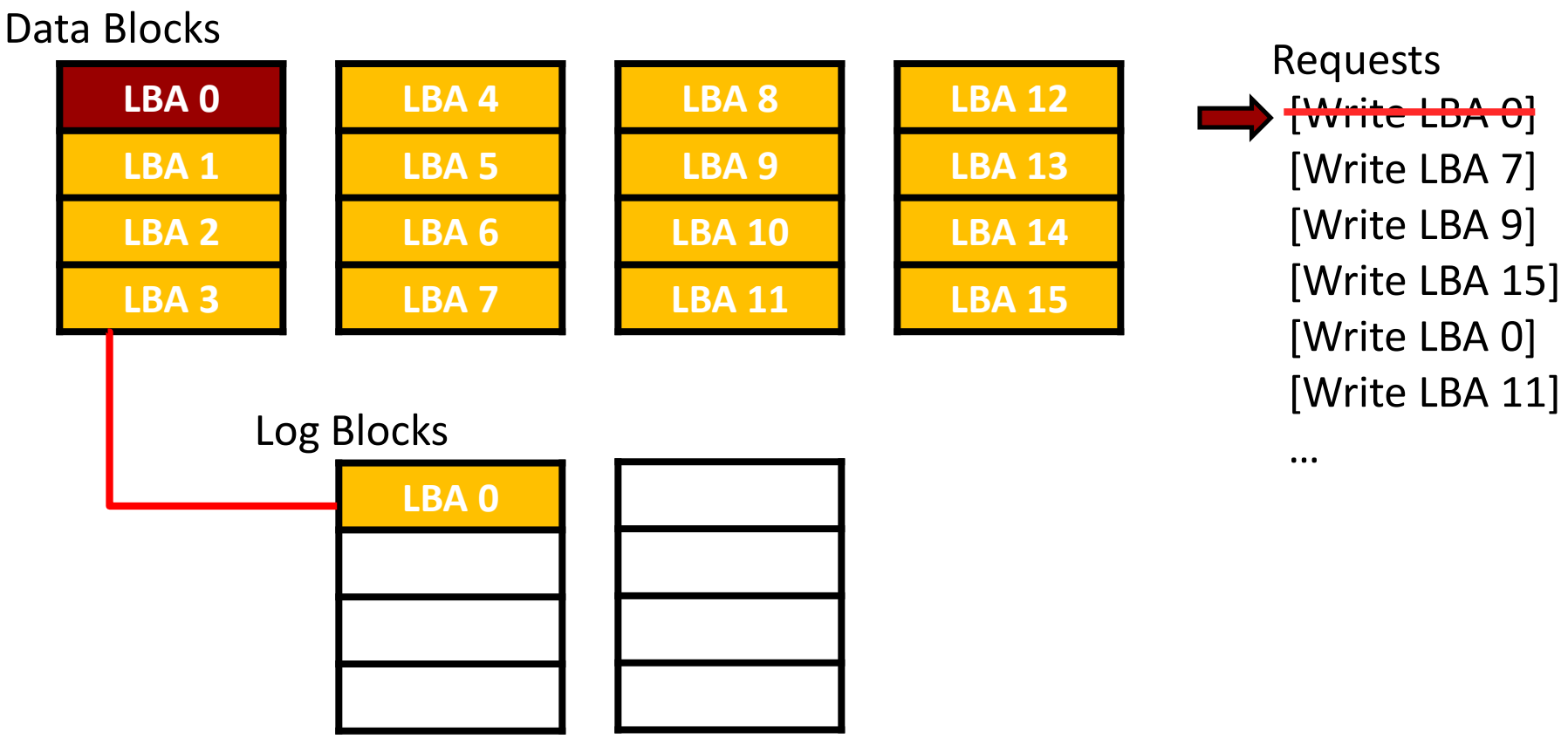
Problems of BAST

- **Log-Block thrashing**
  - Not enough to cover the write requests



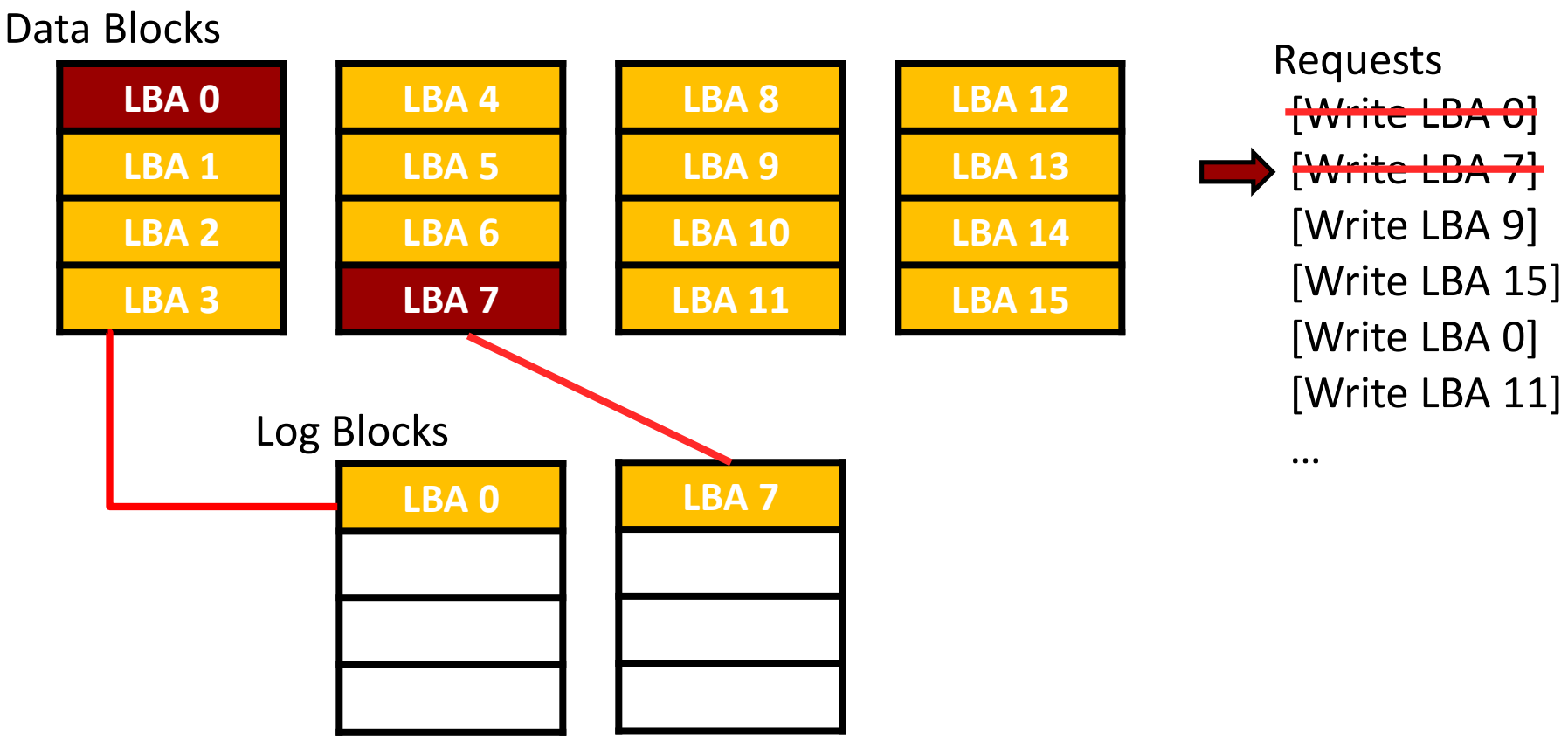
Problems of BAST

- **Log-Block thrashing**
  - Not enough to cover the write requests



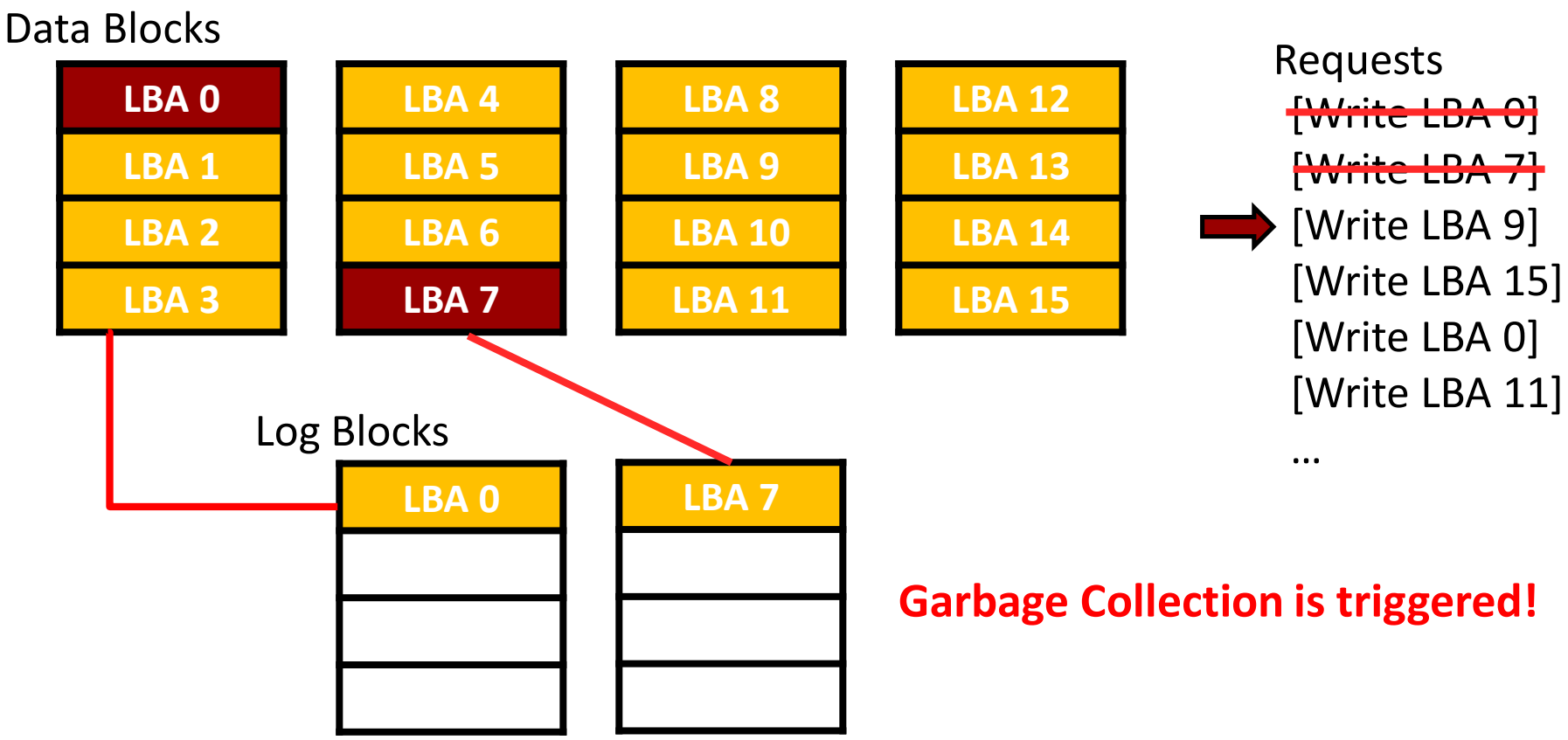
Problems of BAST

- **Log-Block thrashing**
  - Not enough to cover the write requests



Problems of BAST

- **Log-Block thrashing**
  - Not enough to cover the write requests

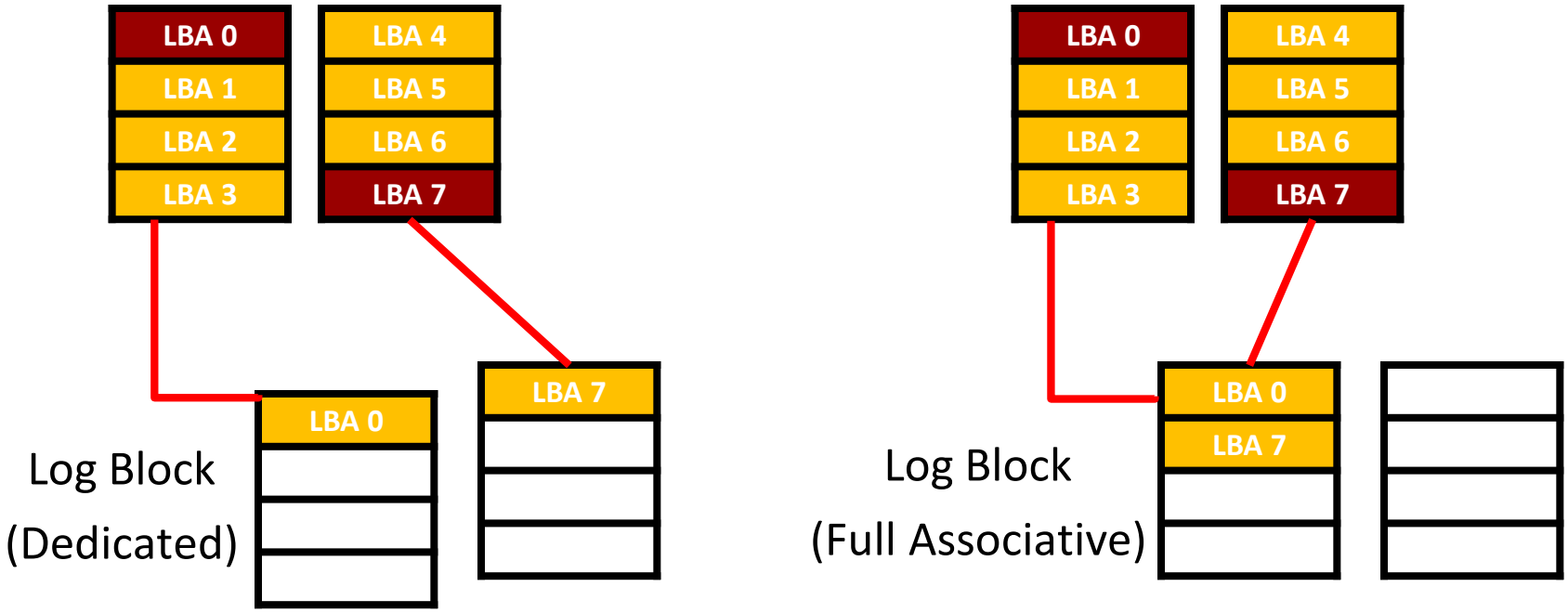


Challenges of BAST

- **Frequent Merge operation**
  - In random write patterns
  - In complicated application

FAST : Fully Associative Sector Translation

- **Key idea**
  - Fully associative mapping between data blocks and log blocks



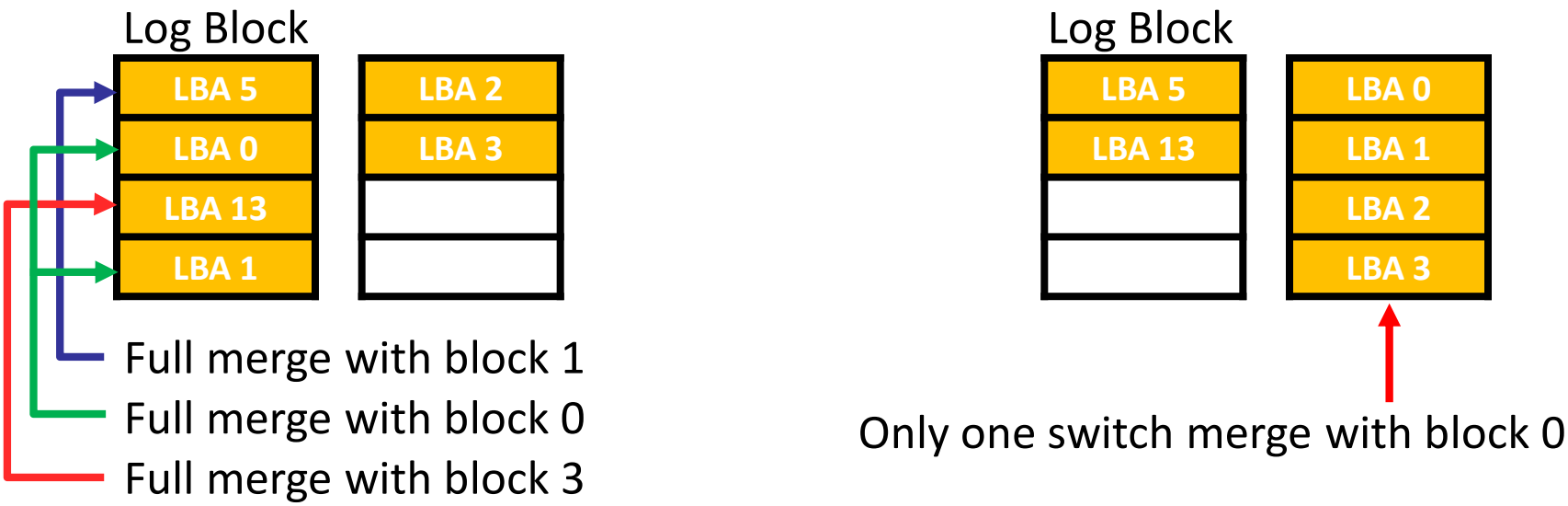
- Mapping within a log block is managed in page-level as in log block scheme

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 involved in a log block

FAST : Sequential Log Block

- **Increase the number of switch operations**
  - Which one is the better option? 5→0→13→1→2→3

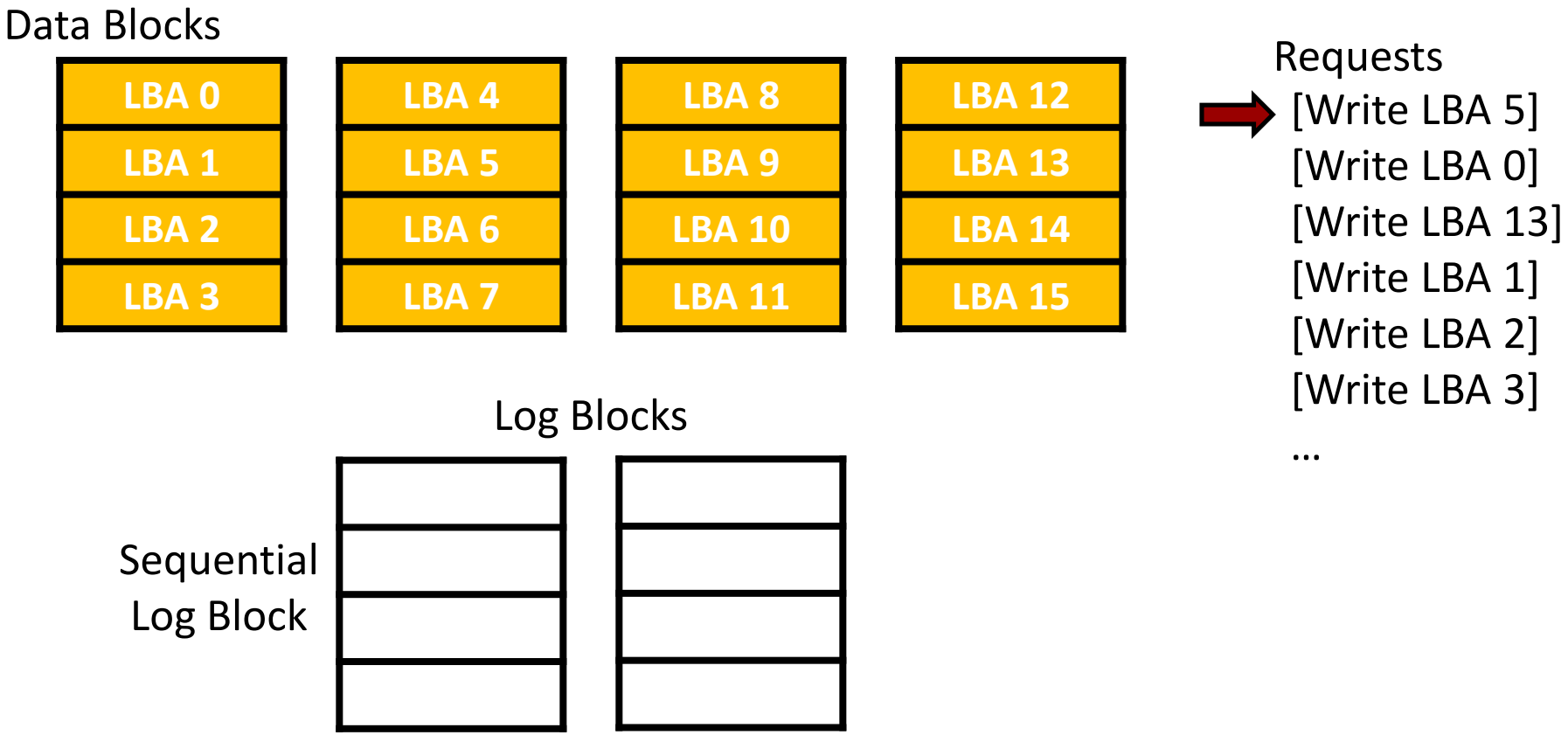


- **Insert a page in the sequential log block if the offset is '0'**
- **Merge sequential log block if there is no empty one or the sequentiality is broken**

FAST : Example

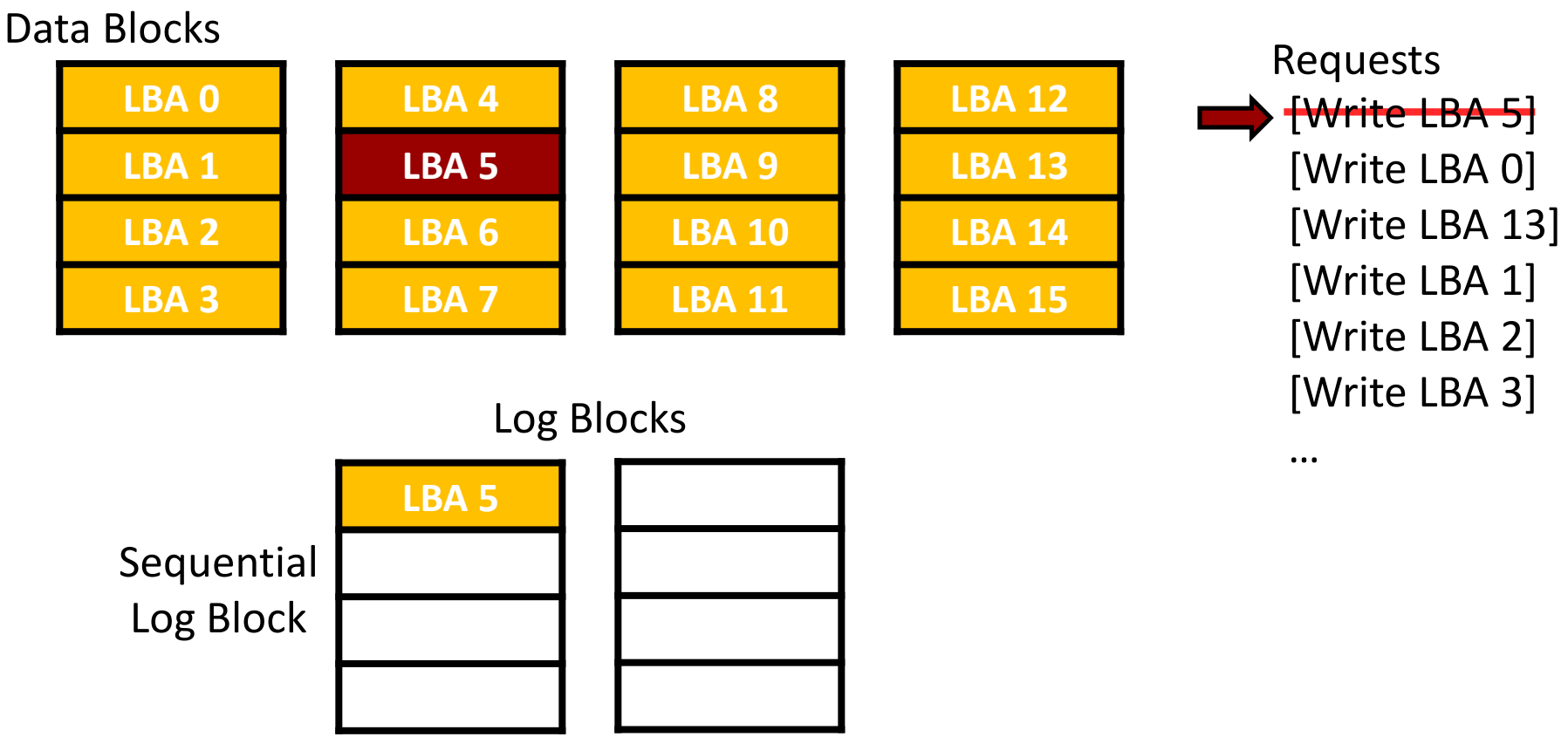
- **Example scenario same as before**

5→0→13→1→2→3



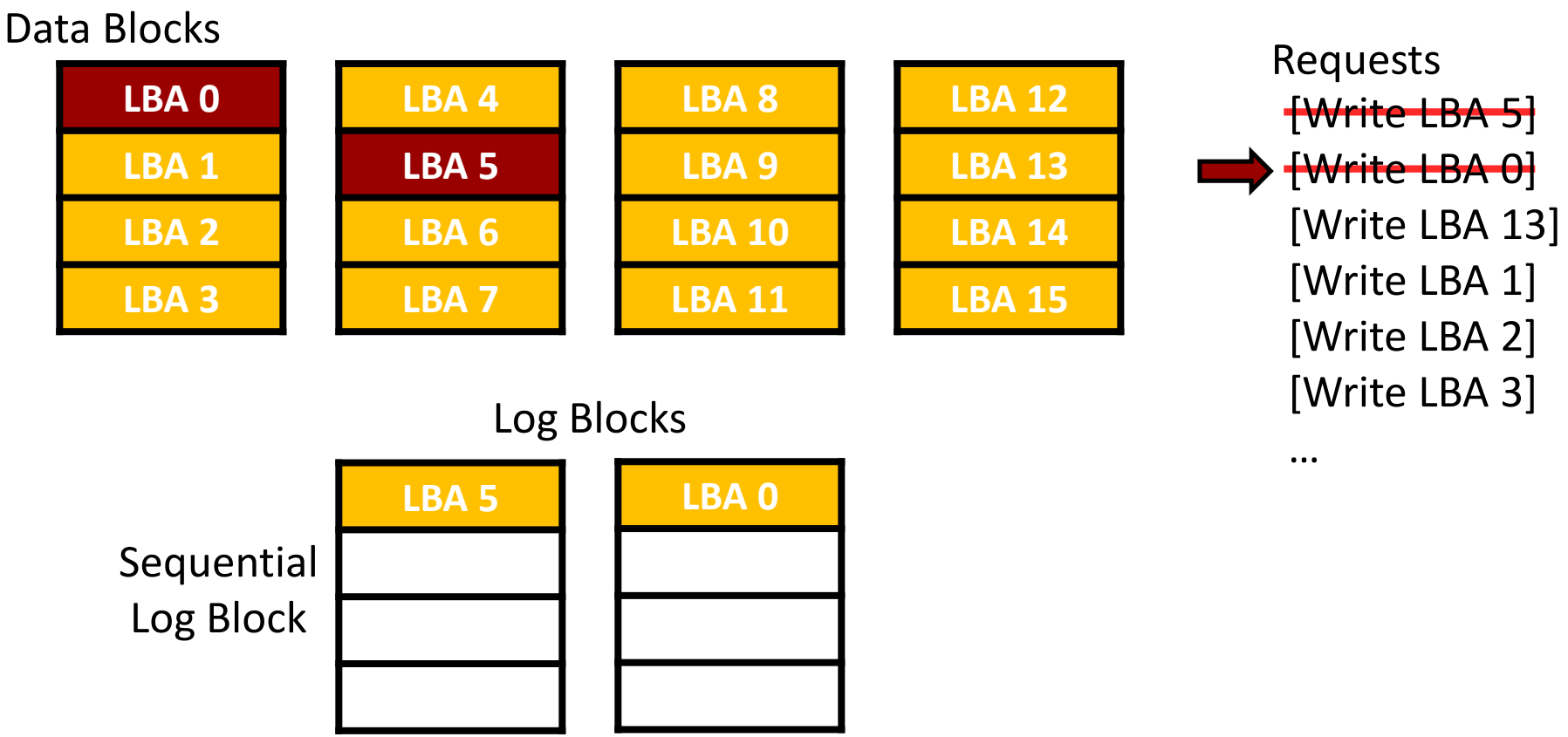
FAST : Example

- **Example scenario same as before**



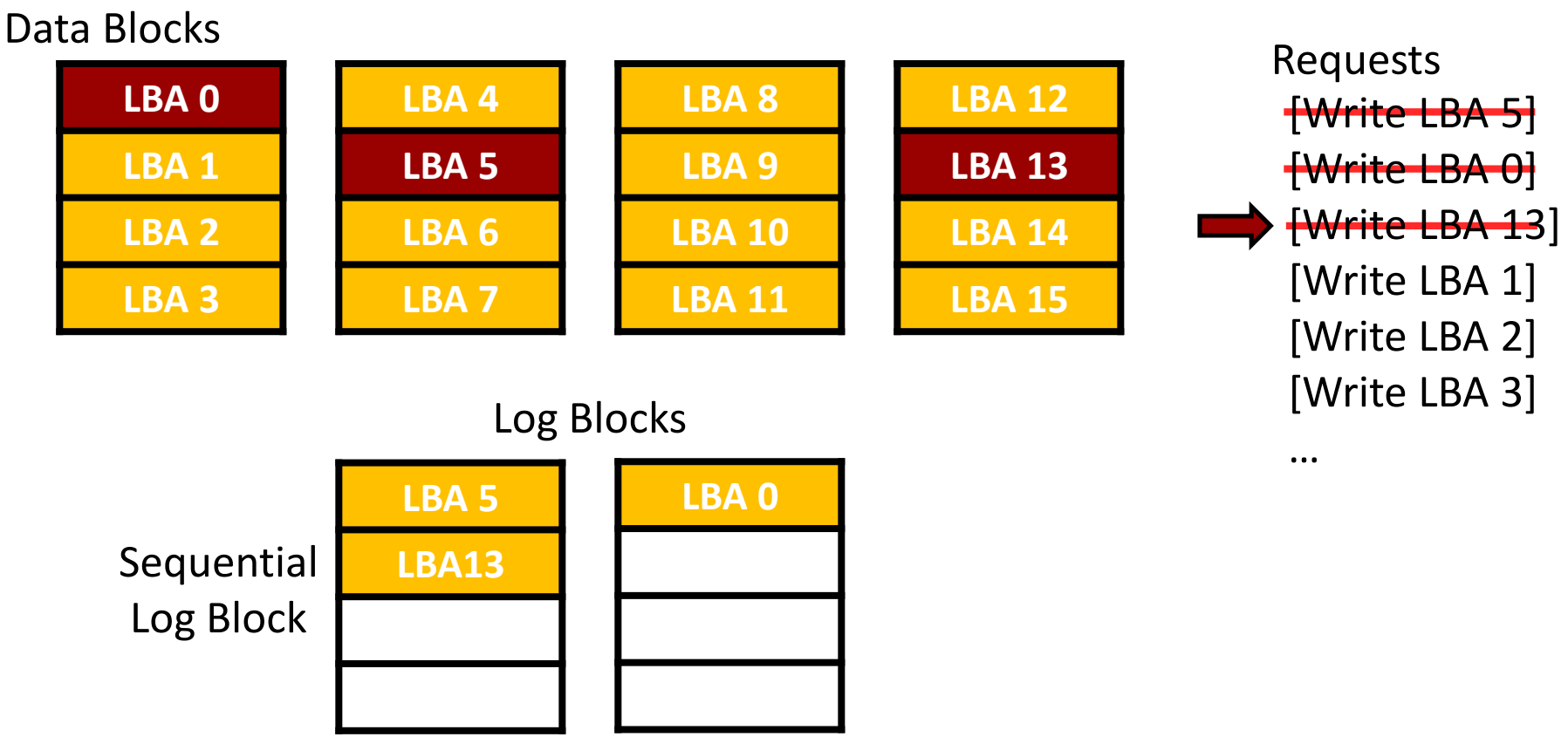
FAST : Example

- **Example scenario same as before**



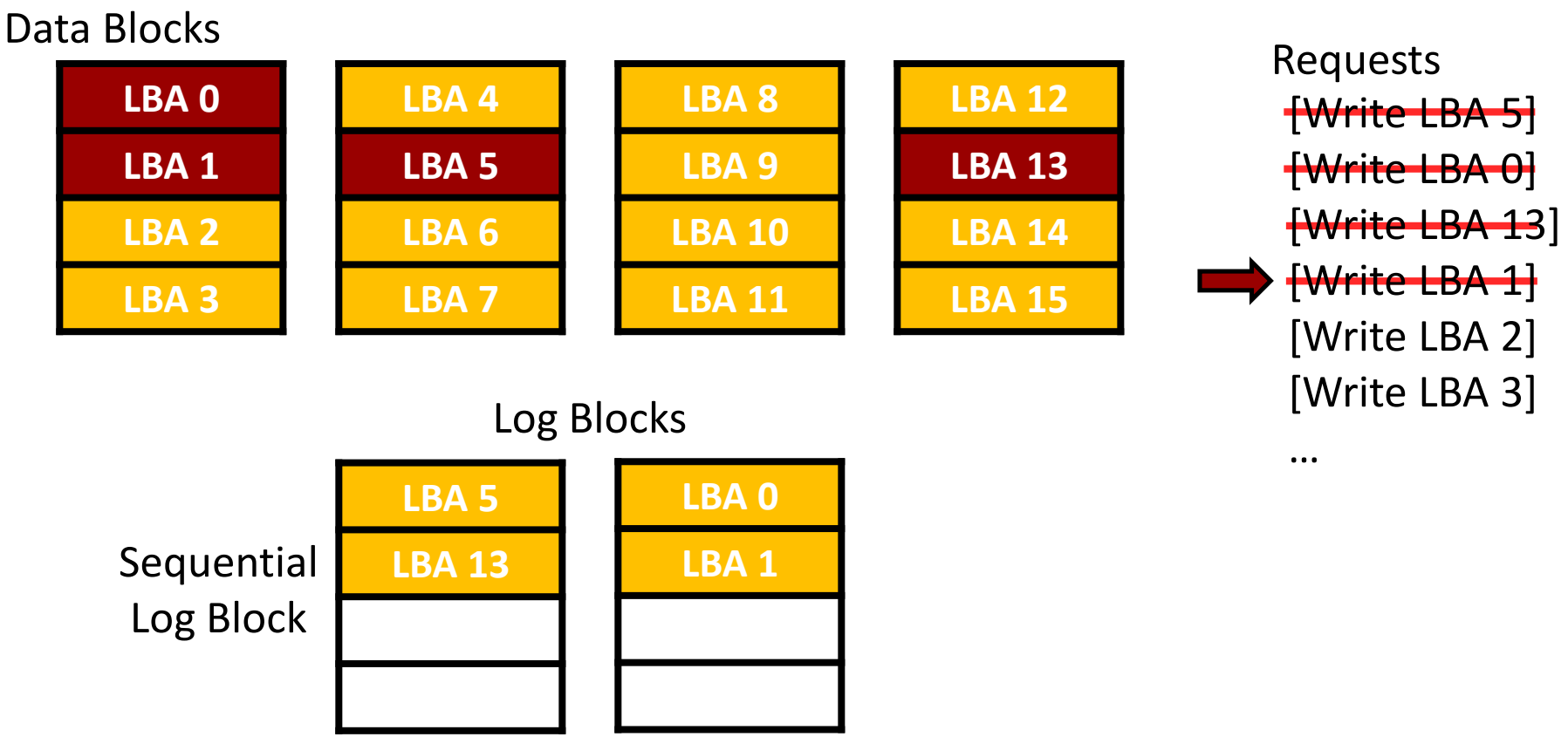
FAST : Example

- **Example scenario same as before**



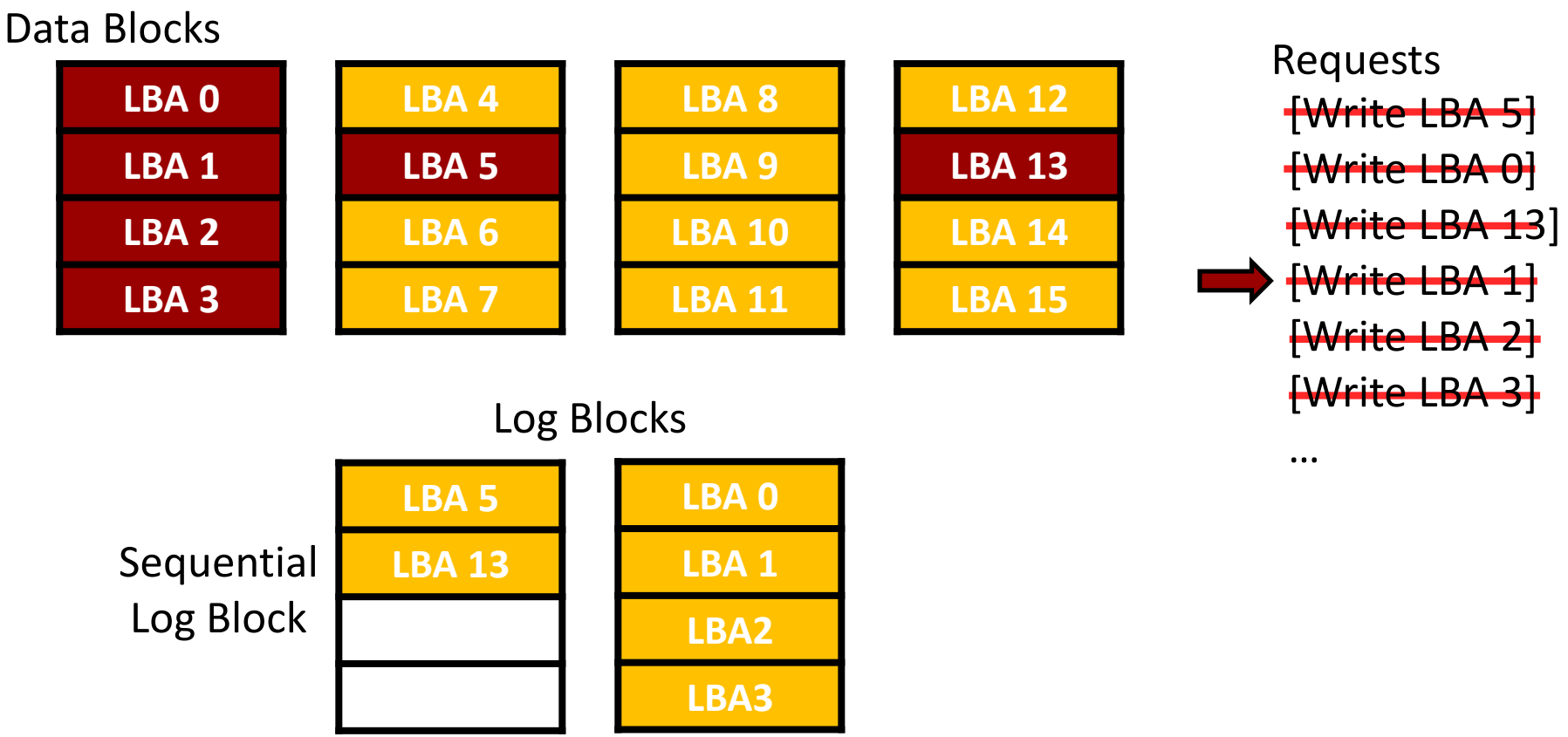
FAST : Example

- **Example scenario same as before**



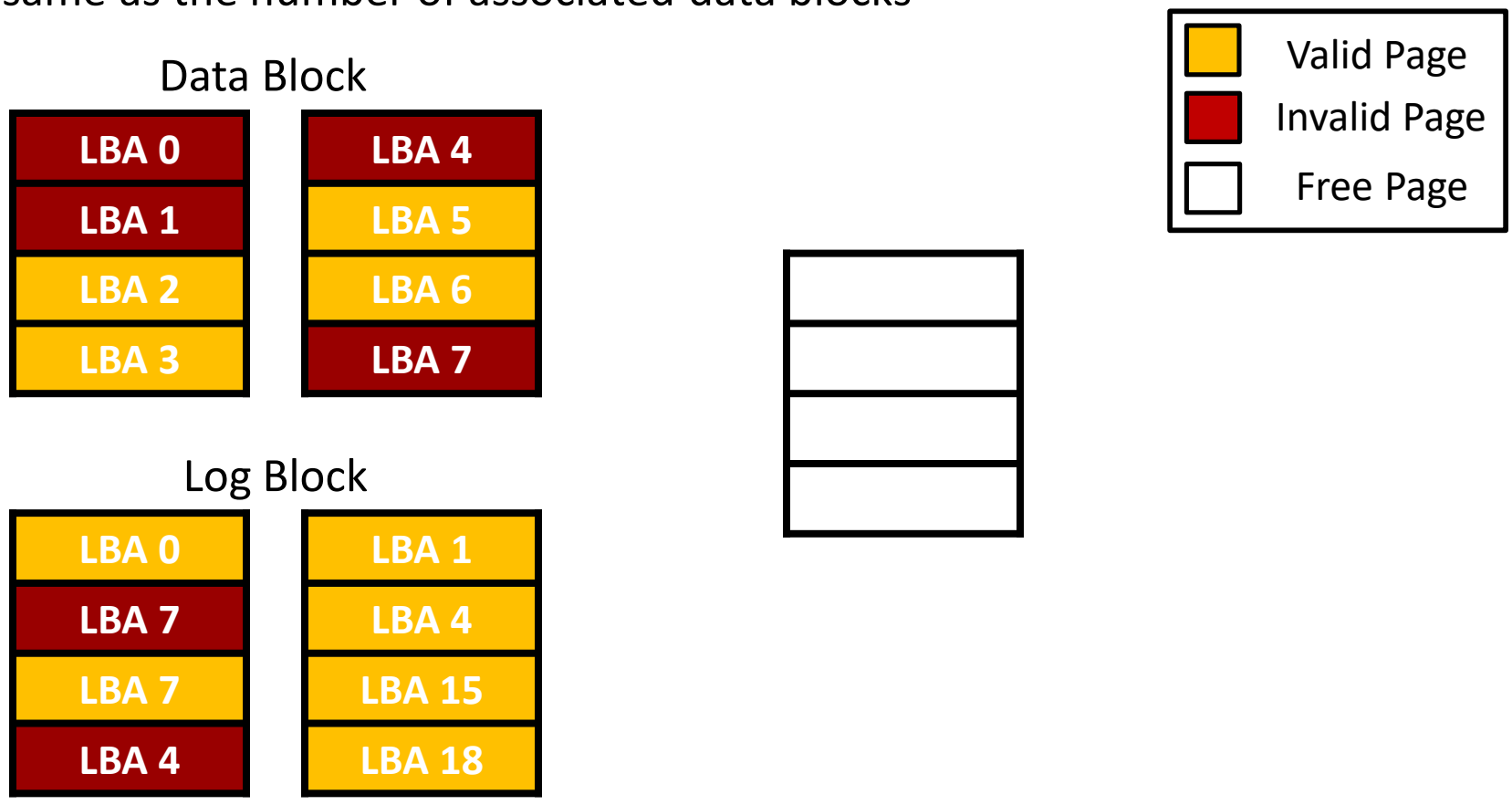
FAST : Example

- **Example scenario same as before**



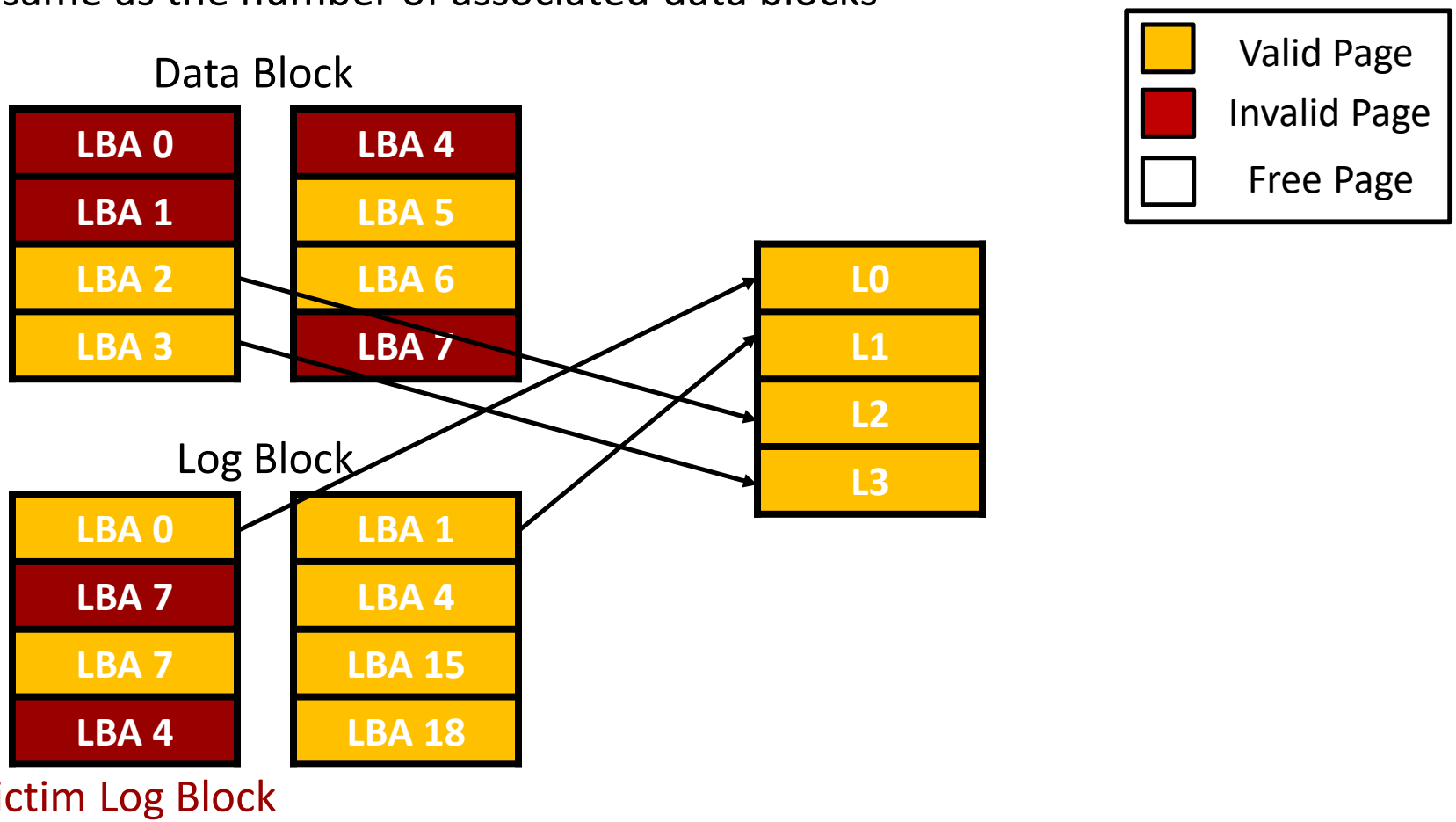
Merge Operation in FAST

- **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



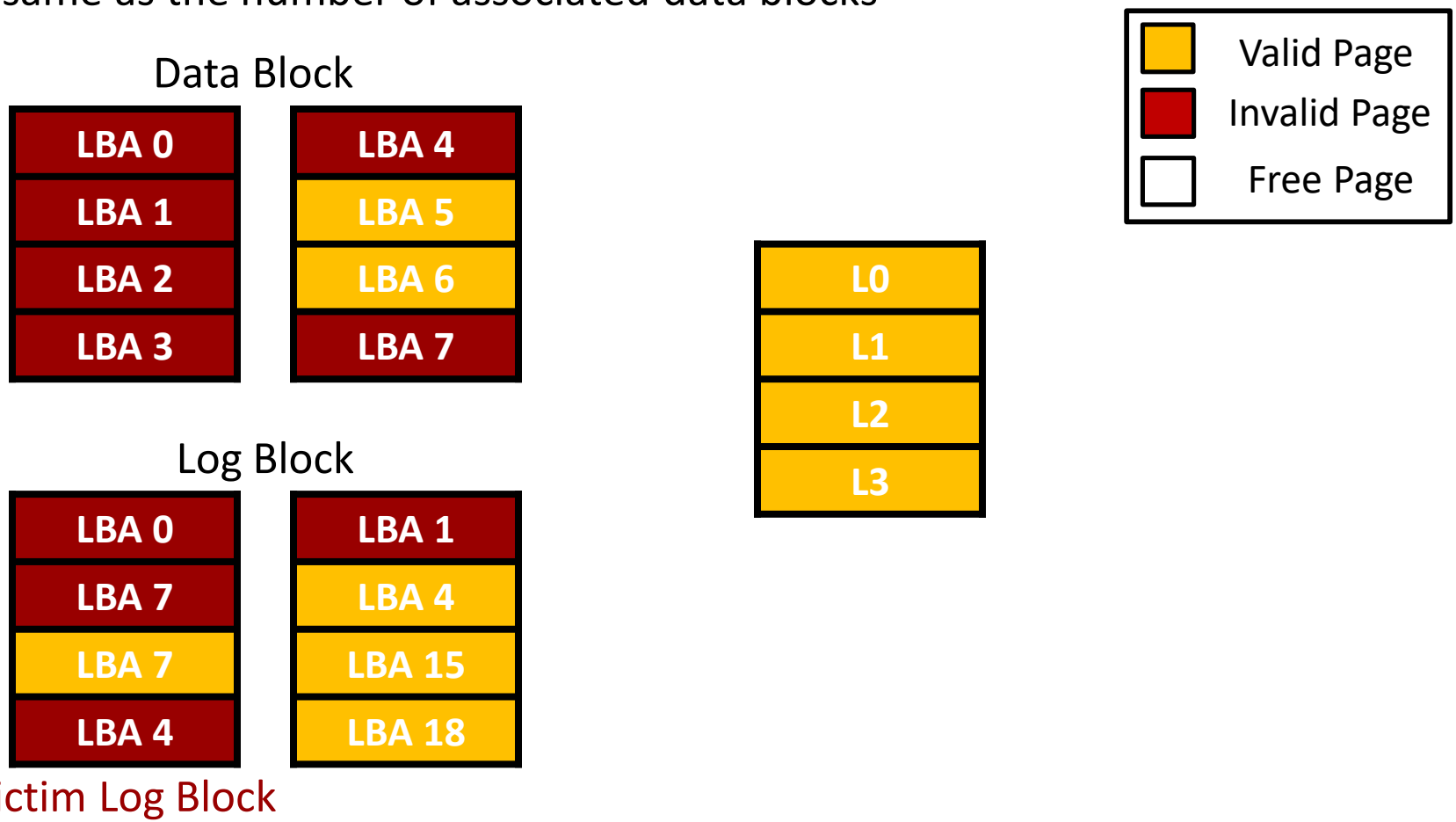
Merge Operation in FAST

- **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



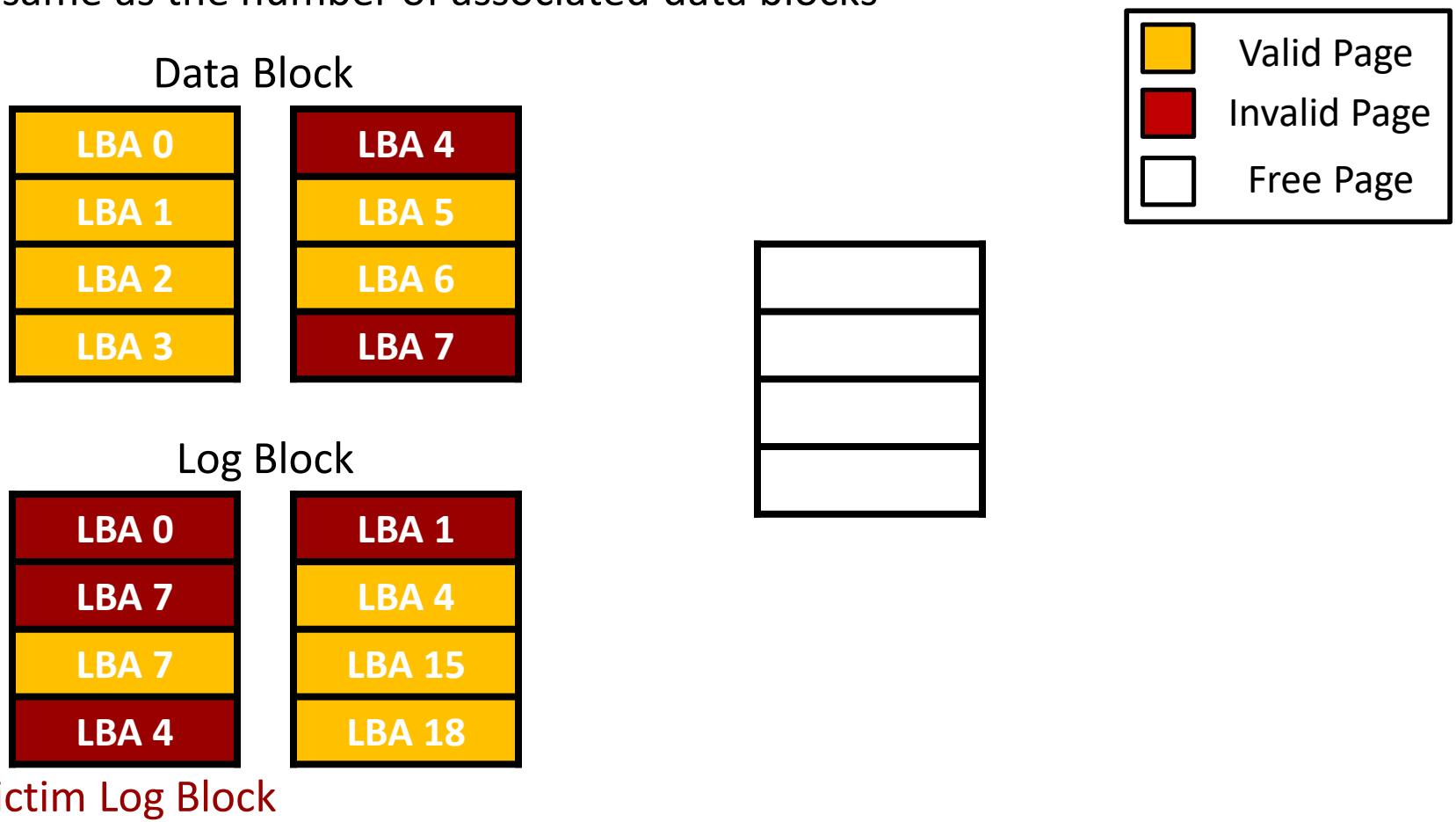
Merge Operation in FAST

- **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



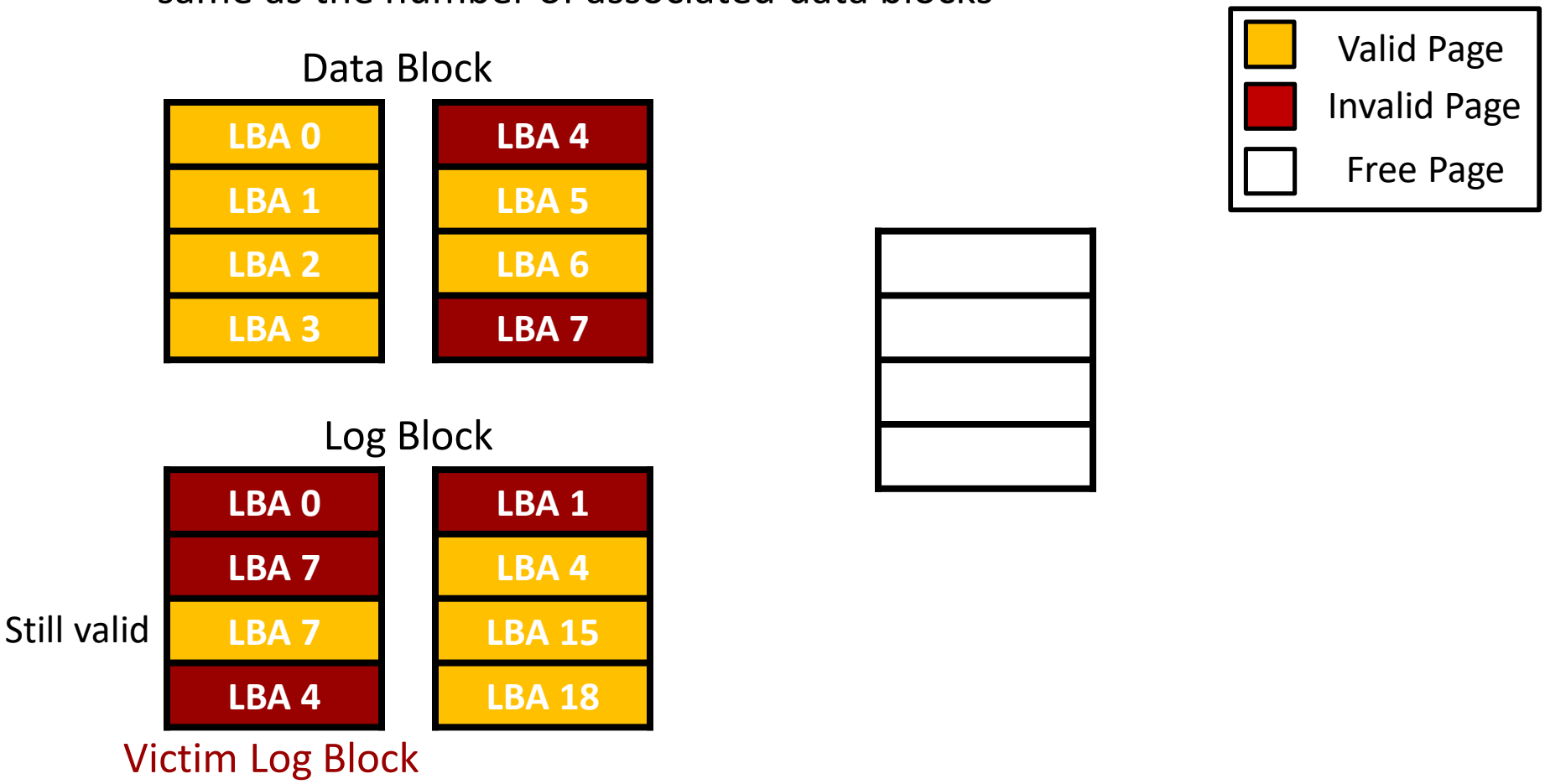
Merge Operation in FAST

- **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



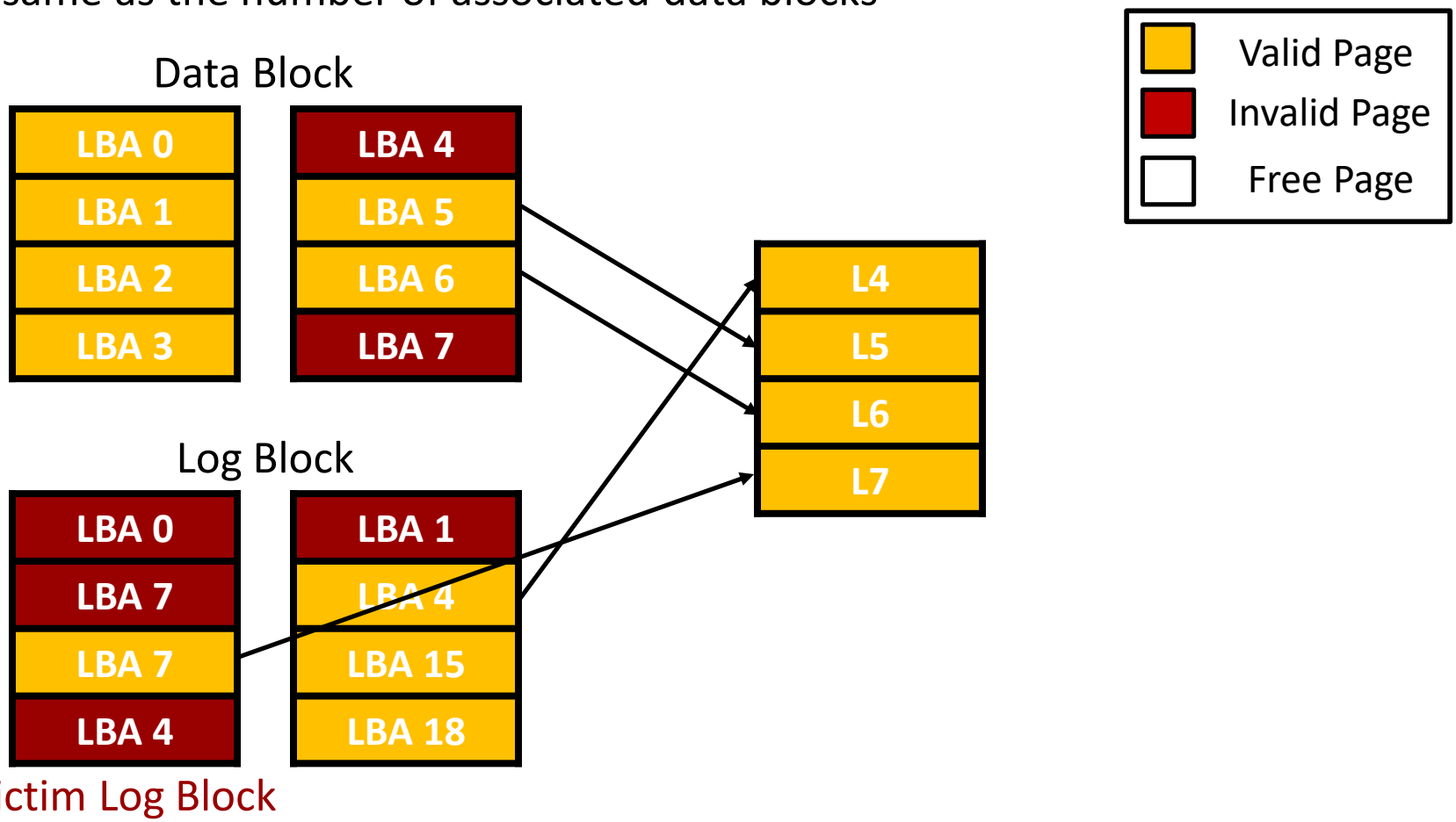
Merge Operation in FAST

- **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



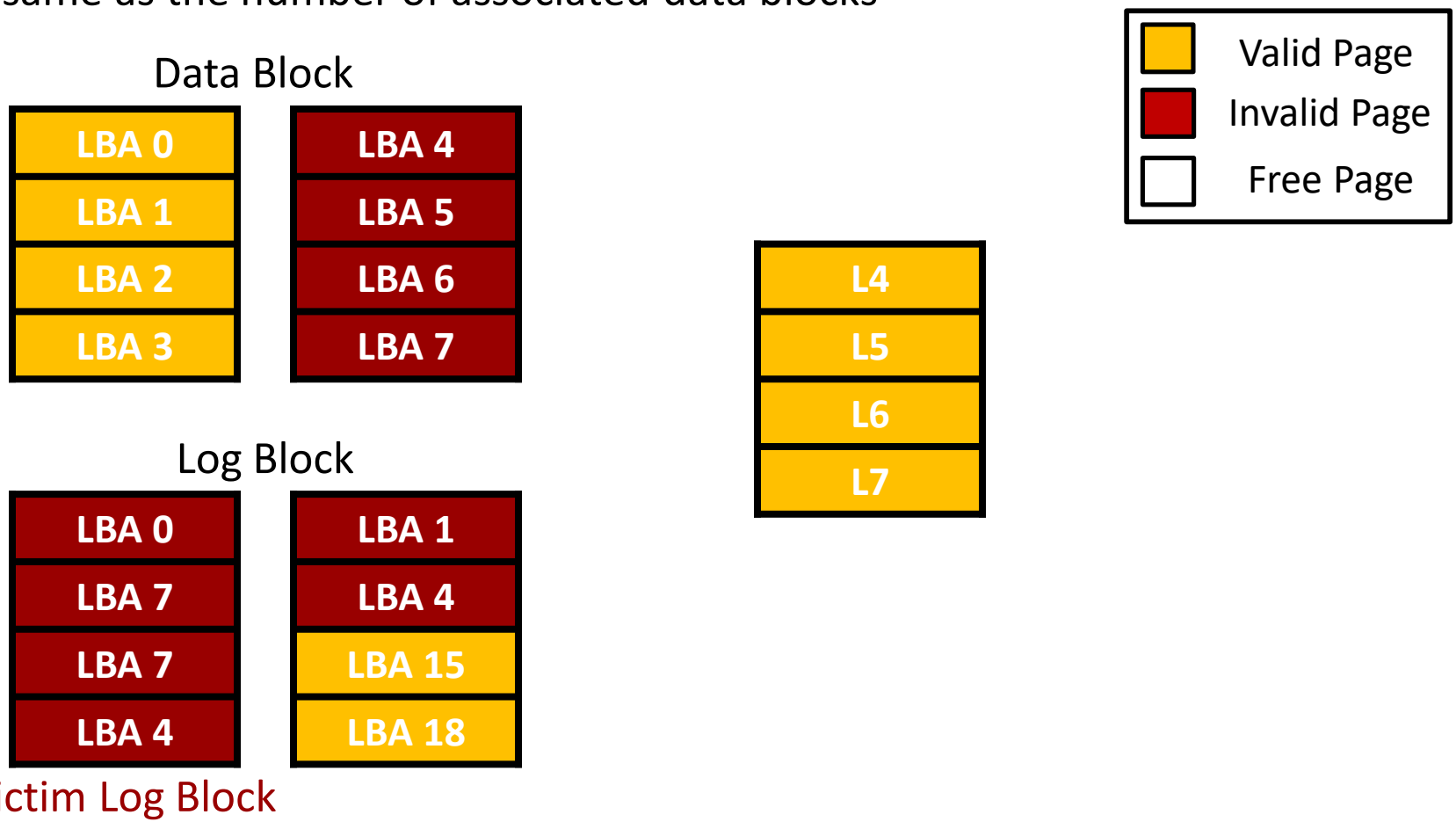
Merge Operation in FAST

- **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



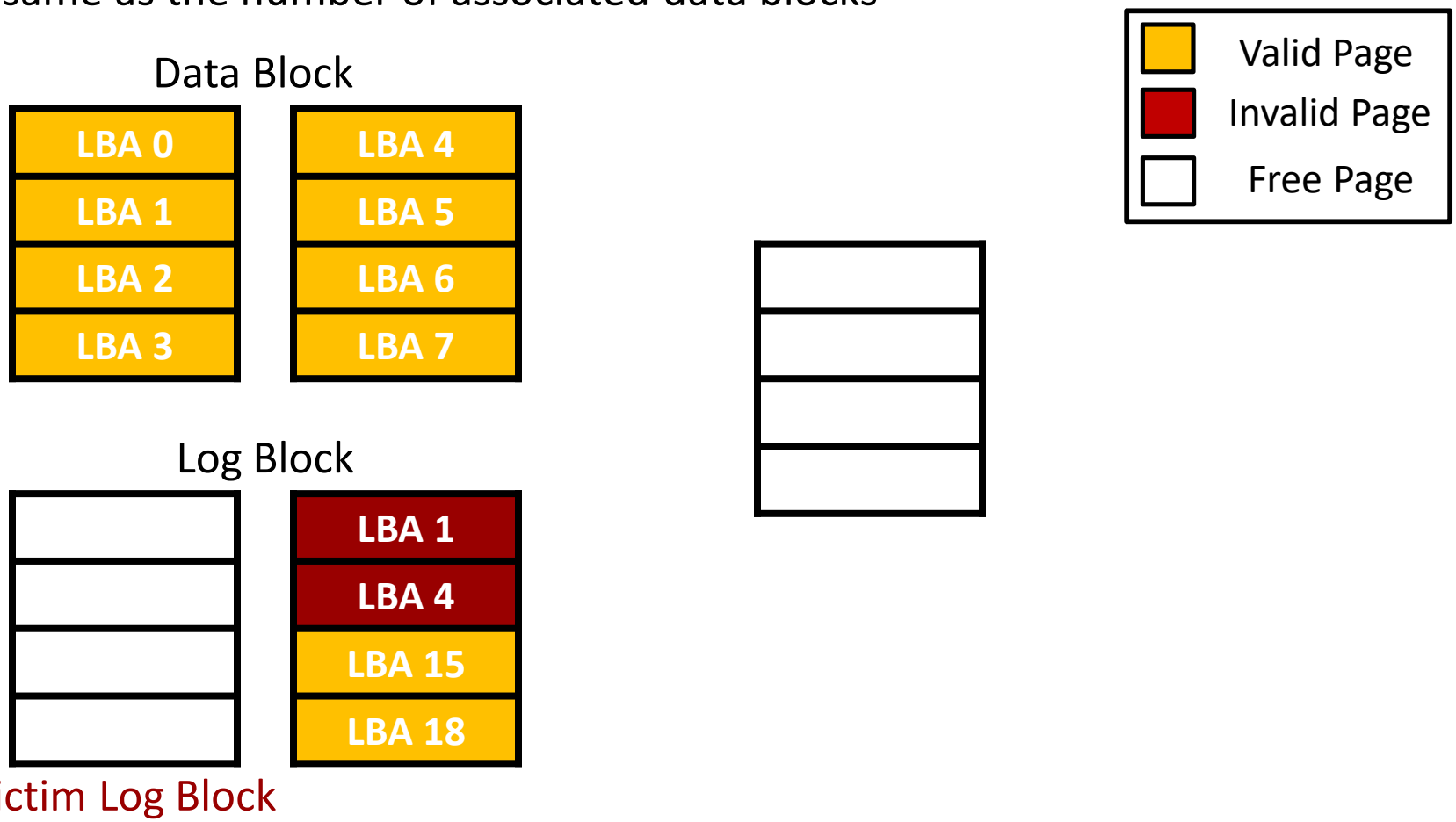
Merge Operation in FAST

- **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



Merge Operation in FAST

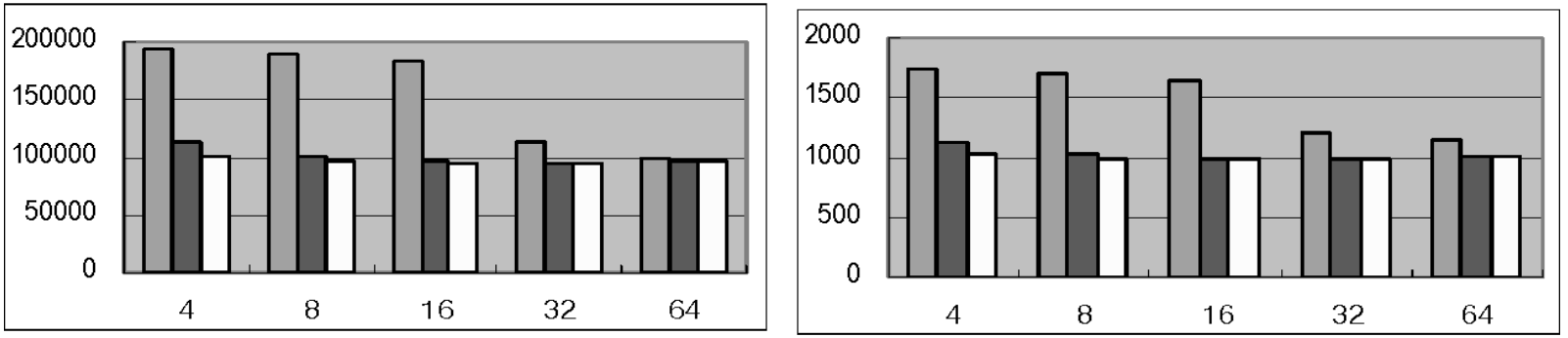
- **In the garbage collection to get a free page**
  - When a log block is the victim block, the number of merge operations is same as the number of associated data blocks



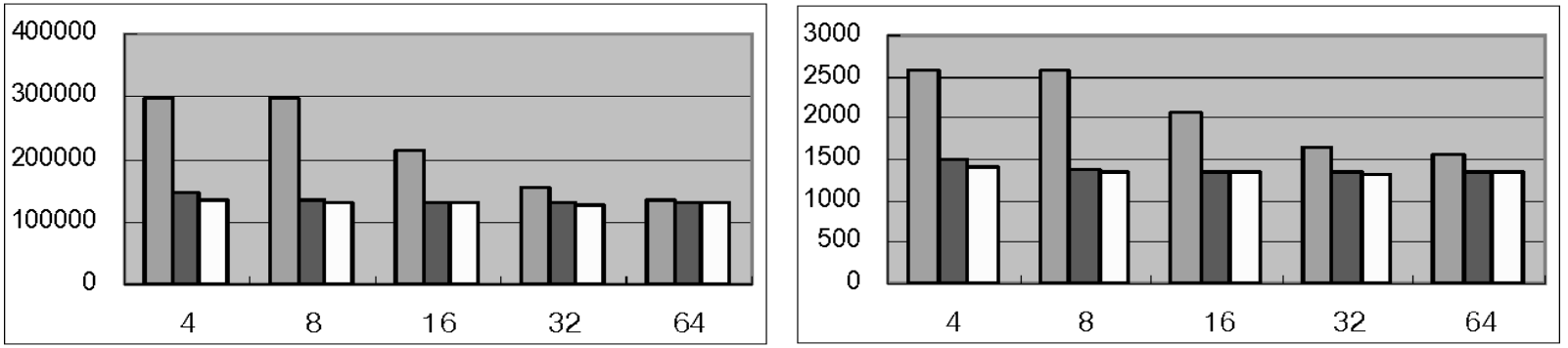
Experimental Result

- **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)

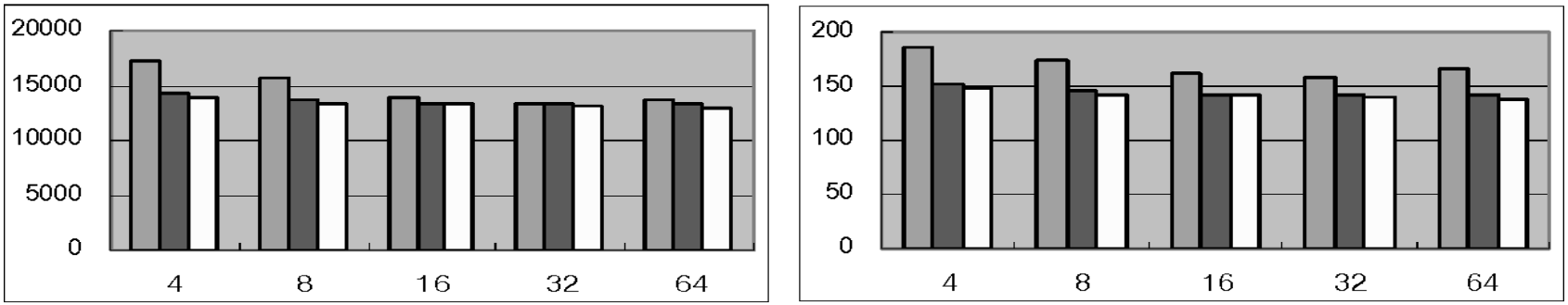


(b) Pattern B: Digital Camera(Company B)

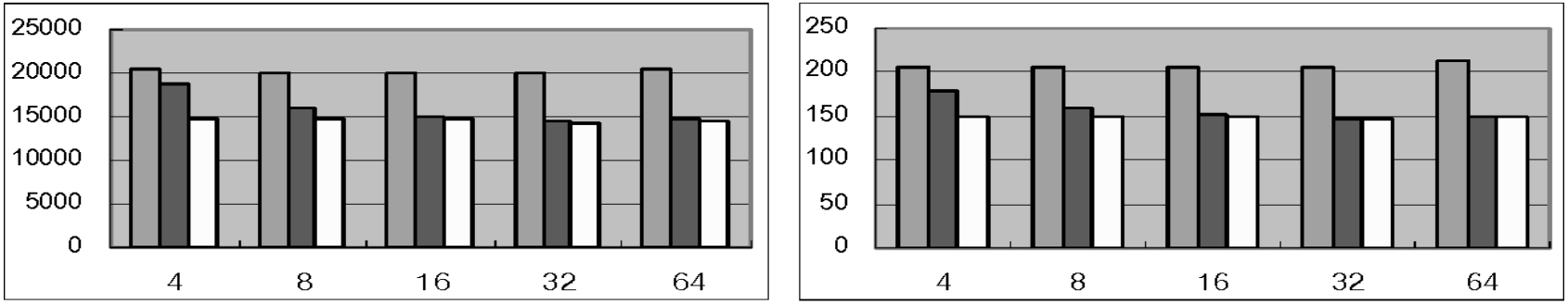
BAST FAST O-FAST

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

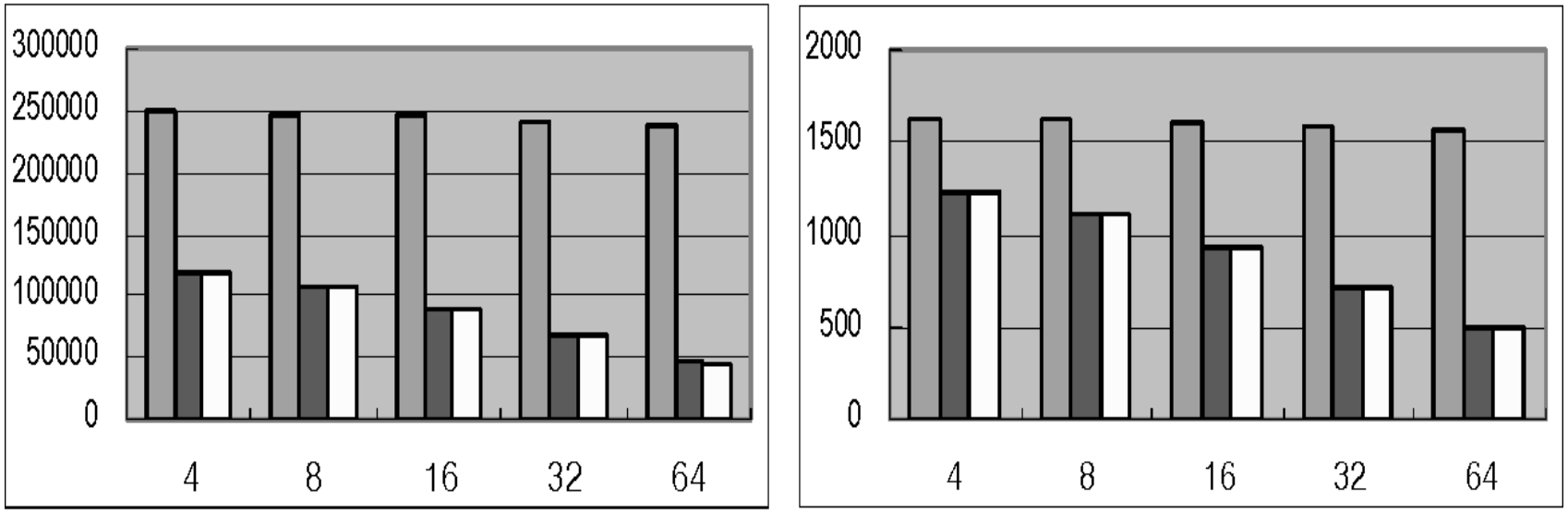


(d) Pattern D: Symbian

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

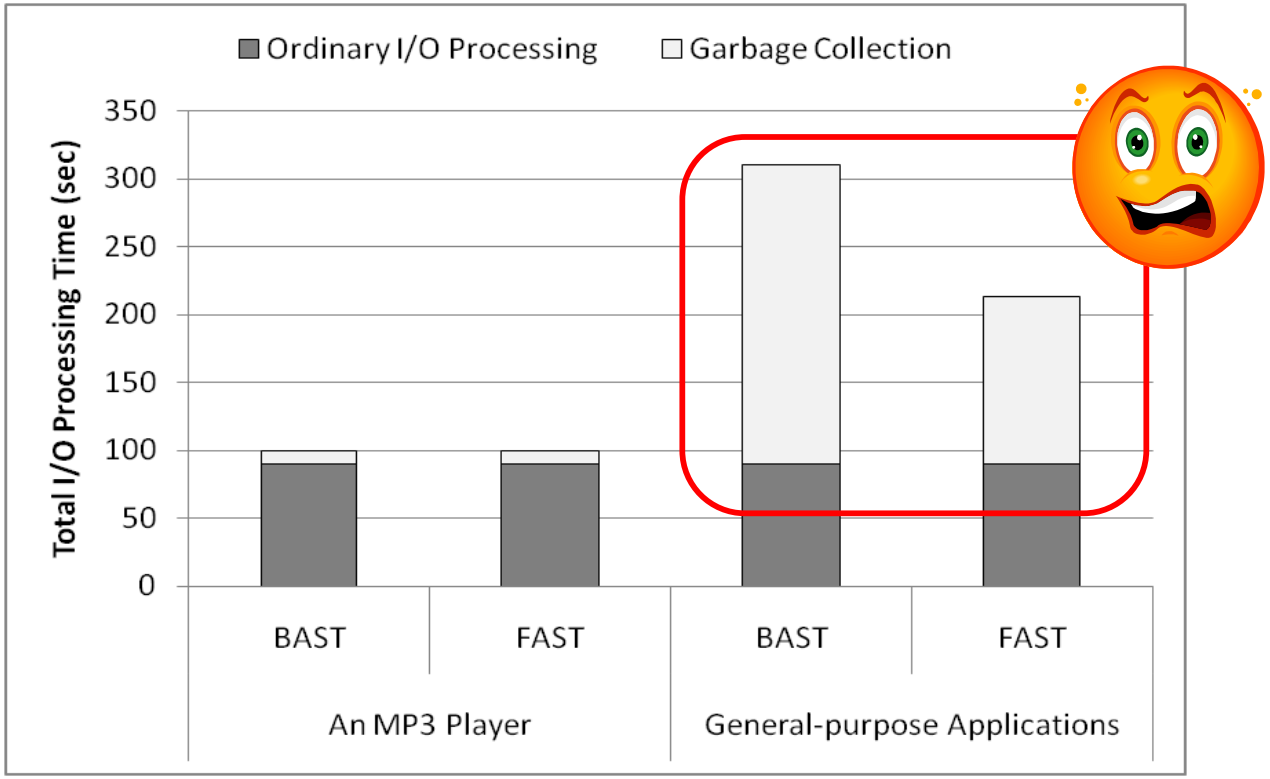
X-axis : # of log blocks, Y-axis in left side : erase count, Y-axis in right side : elapsed time(secs).

Problem of FAST

- **Full merge performed more frequently**
  - The sequential log block for handling sequential writes causes frequent garbage collection
- **Cost of a garbage collection process is high**
  - Associated data blocks of victim log blocks are joined in a garbage collection process
- **Once a log block is allocated, the subsequent write requests to the data block are redirected to the associated log block**

FTL in General-Purpose Computing Systems

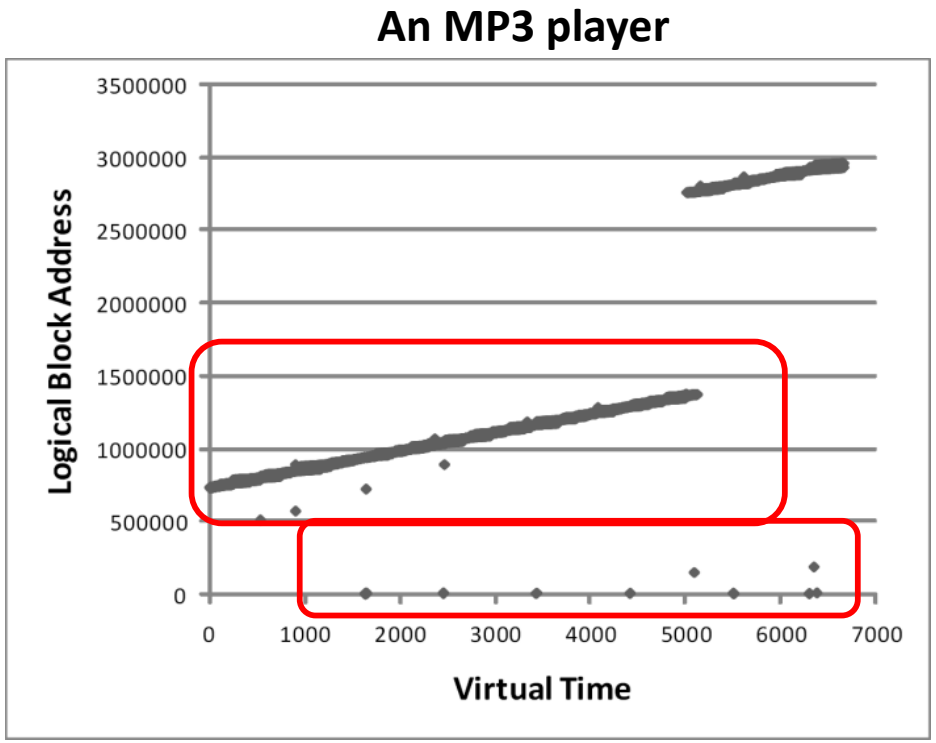
- Existing FTL schemes are ill-suited for general-purpose computing systems



Garbage collection overhead is significantly increased !!!

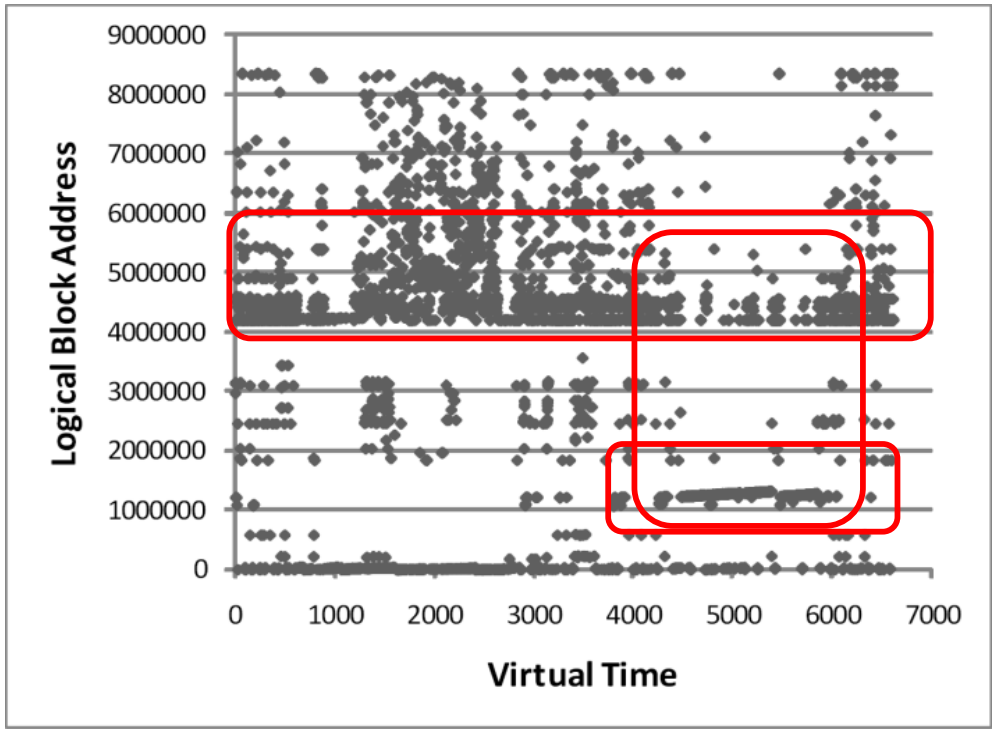
I/O Characteristics of Mobile Embedded Applications

- 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

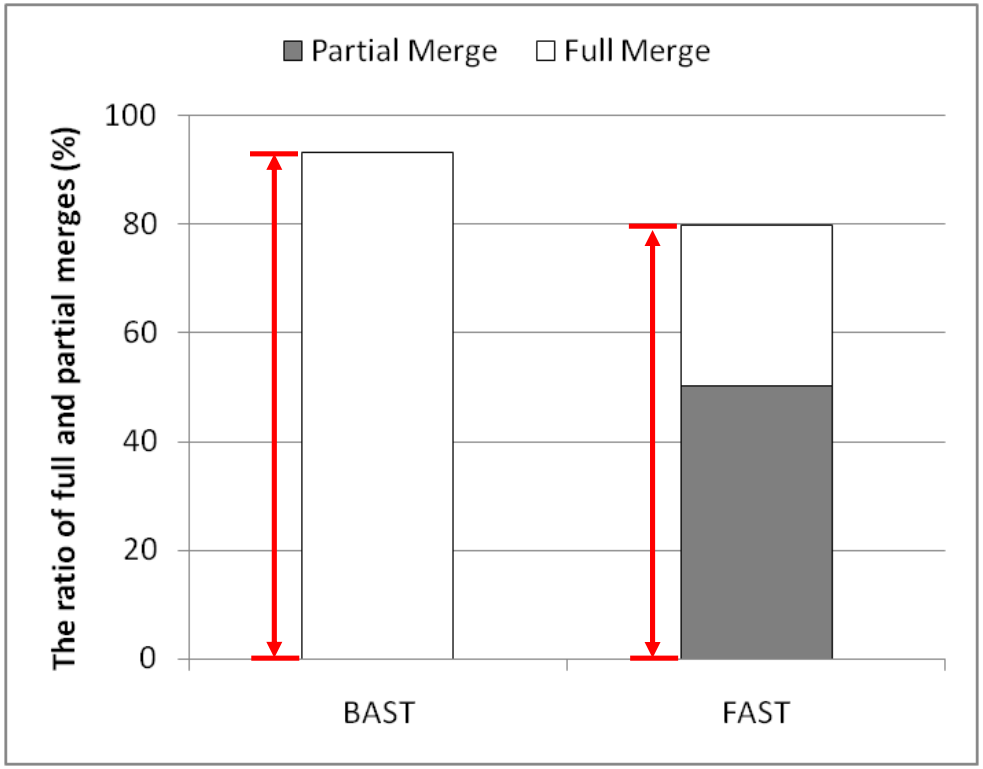
- Many random writes with a high temporal locality
- Many sequential writes with a high sequential locality
- A mixture of random and sequential writes



General-purpose applications

The increased full and partial merge operations

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



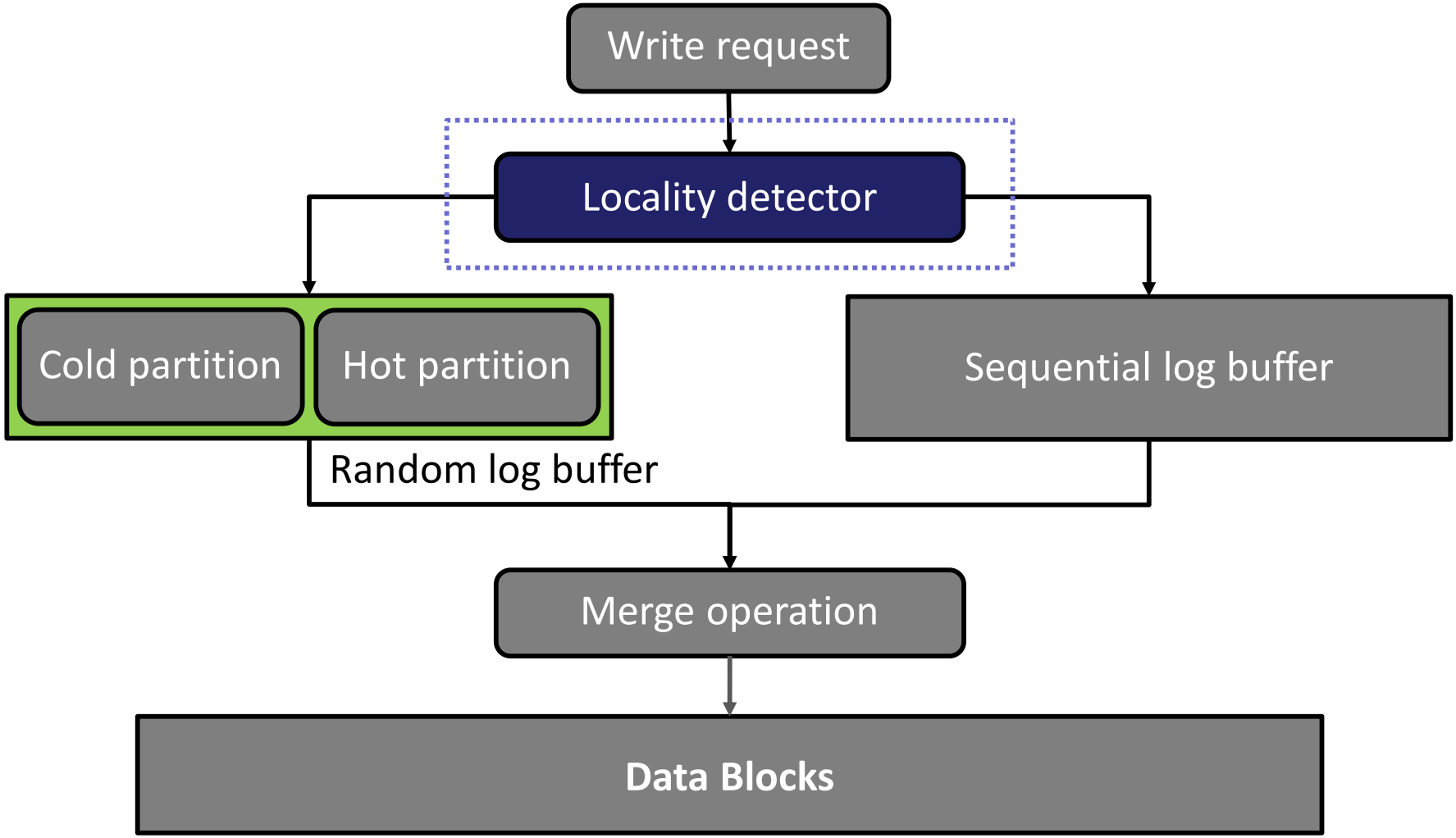
⇒ Need to take advantage of the I/O characteristics of general-purpose applications

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*

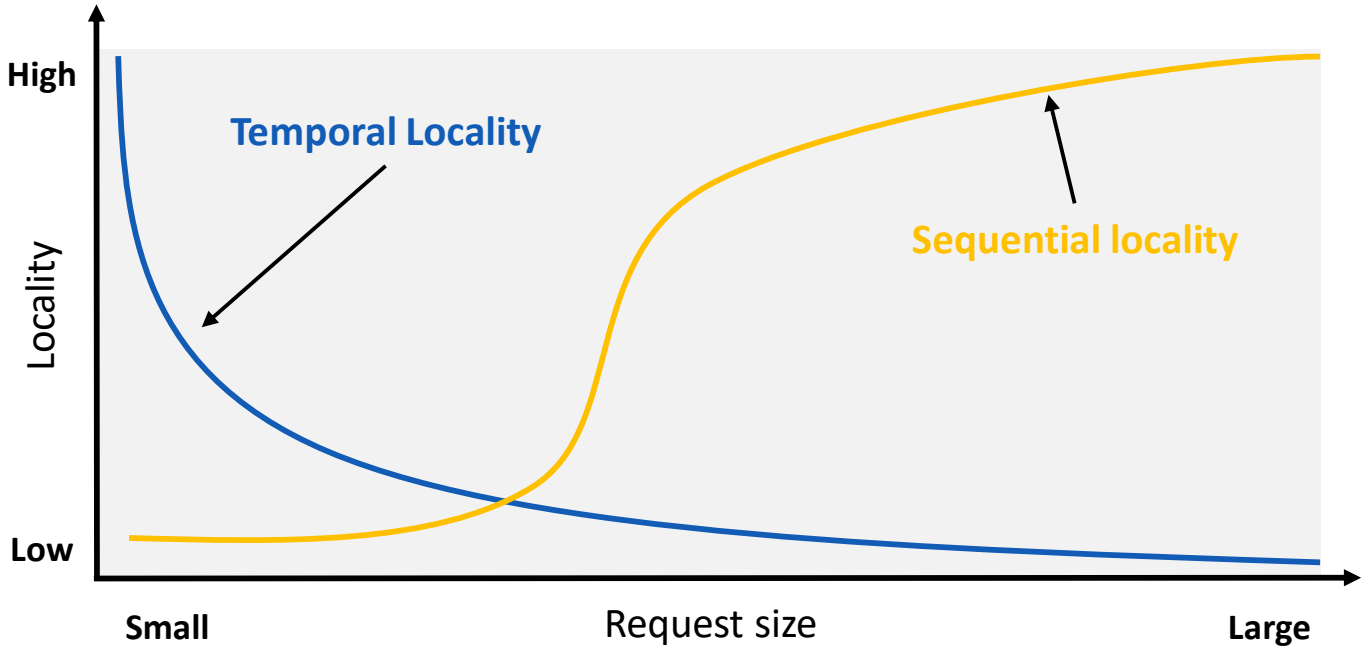
- **Our solutions**
  - Extract a write request having a high sequential locality from the mixed write patterns
    - A locality detector
  - Exploit a high temporal locality of random writes
    - A hot/cold separation policy
    - An intelligent victim selection policy

Overall Architecture of the LAST Scheme



Locality Detector (1)

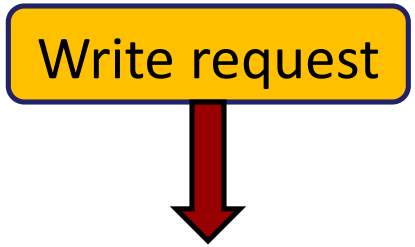
- **How to detect the locality type of a write request**
  - The locality type is highly correlated to the size of write request



- **From the observation of realistic workloads**
  - Small-sized writes have a high temporal locality
  - Large-sized writes have a high sequential locality

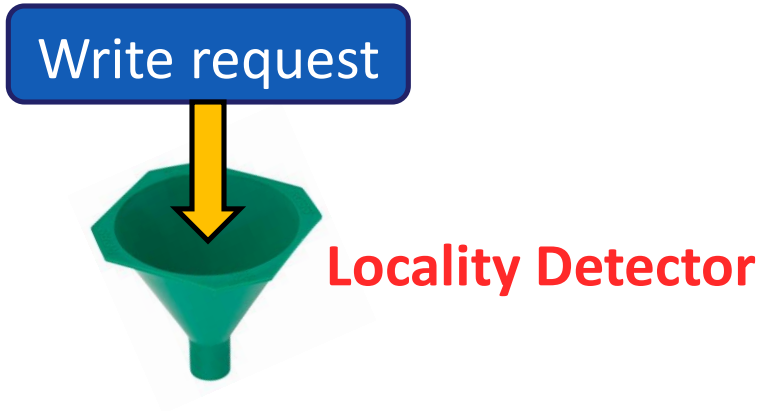
Locality Detector (2)

- A locality-detection policy based on the request size



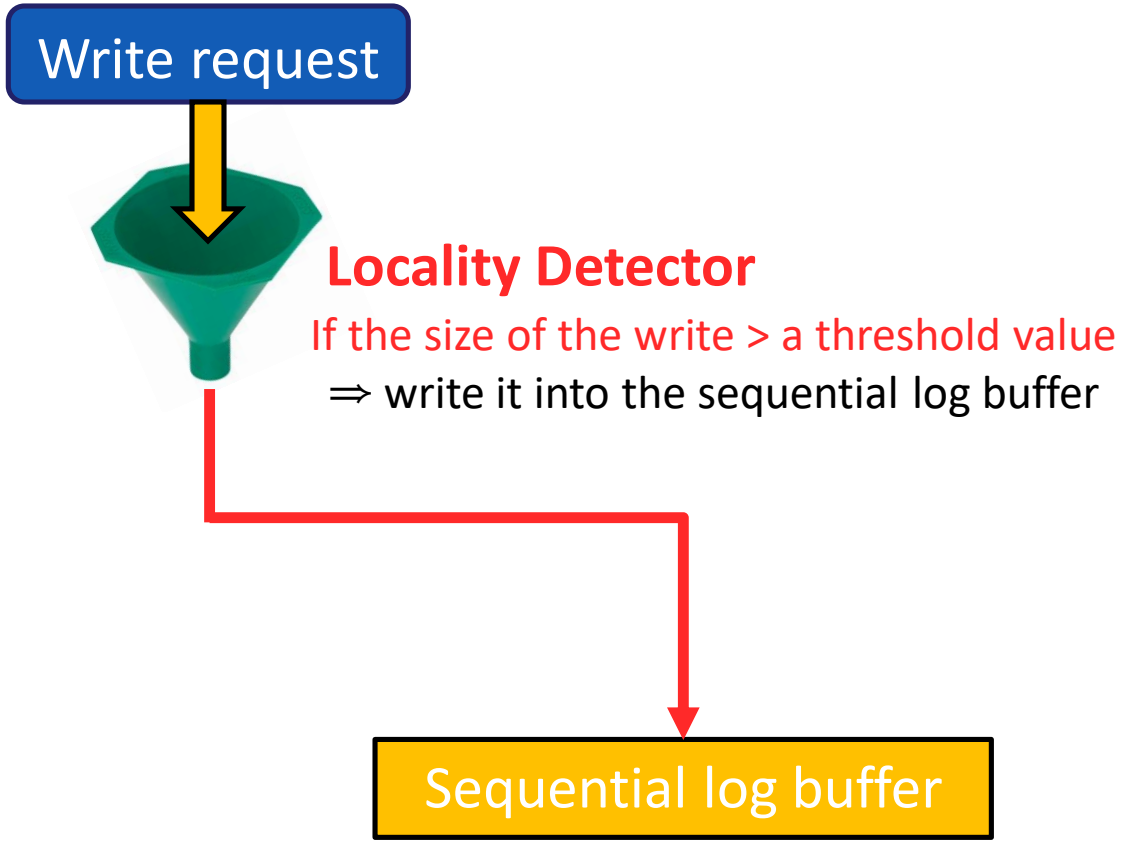
Locality Detector (2)

- A locality-detection policy based on the request size



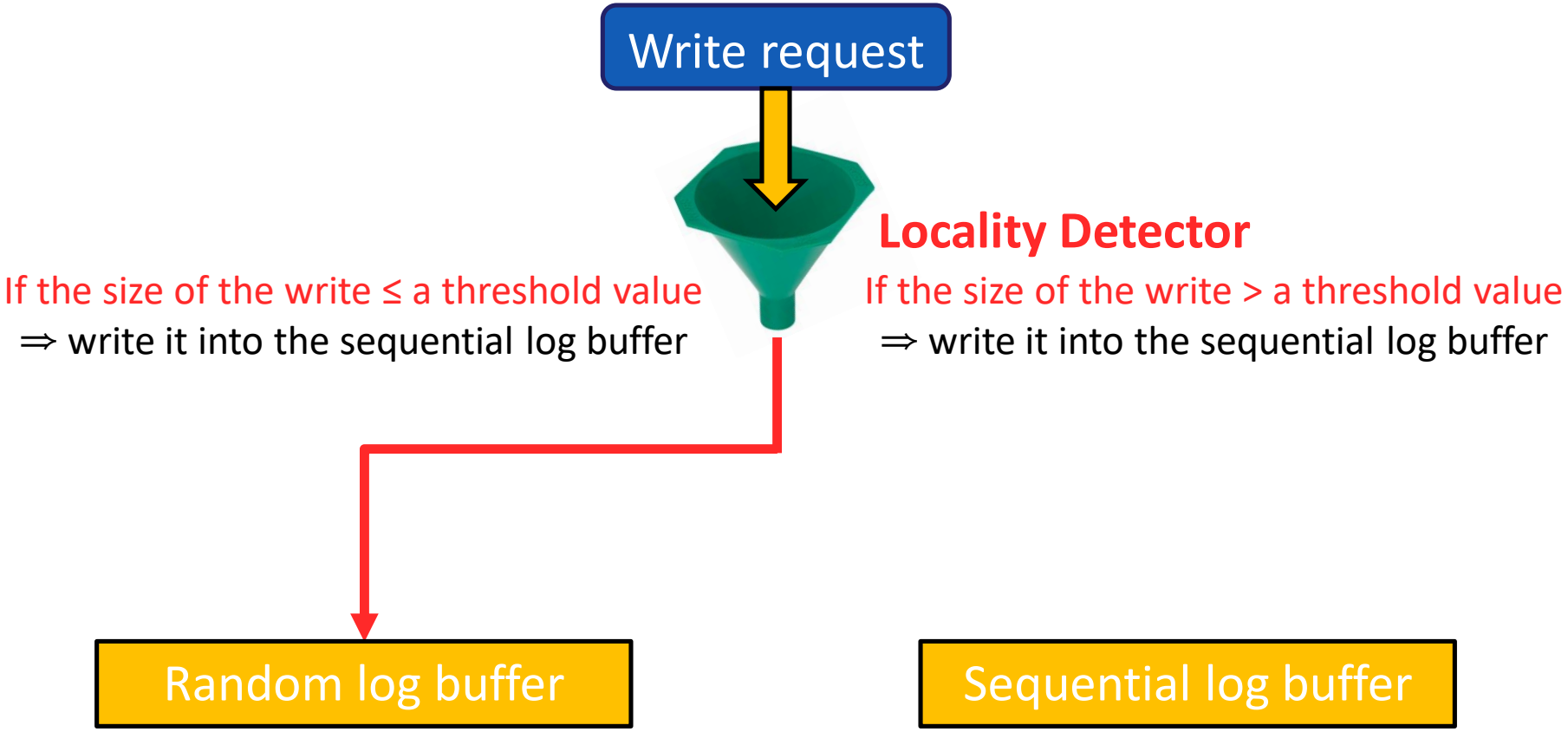
Locality Detector (2)

- **A locality-detection policy based on the request size**



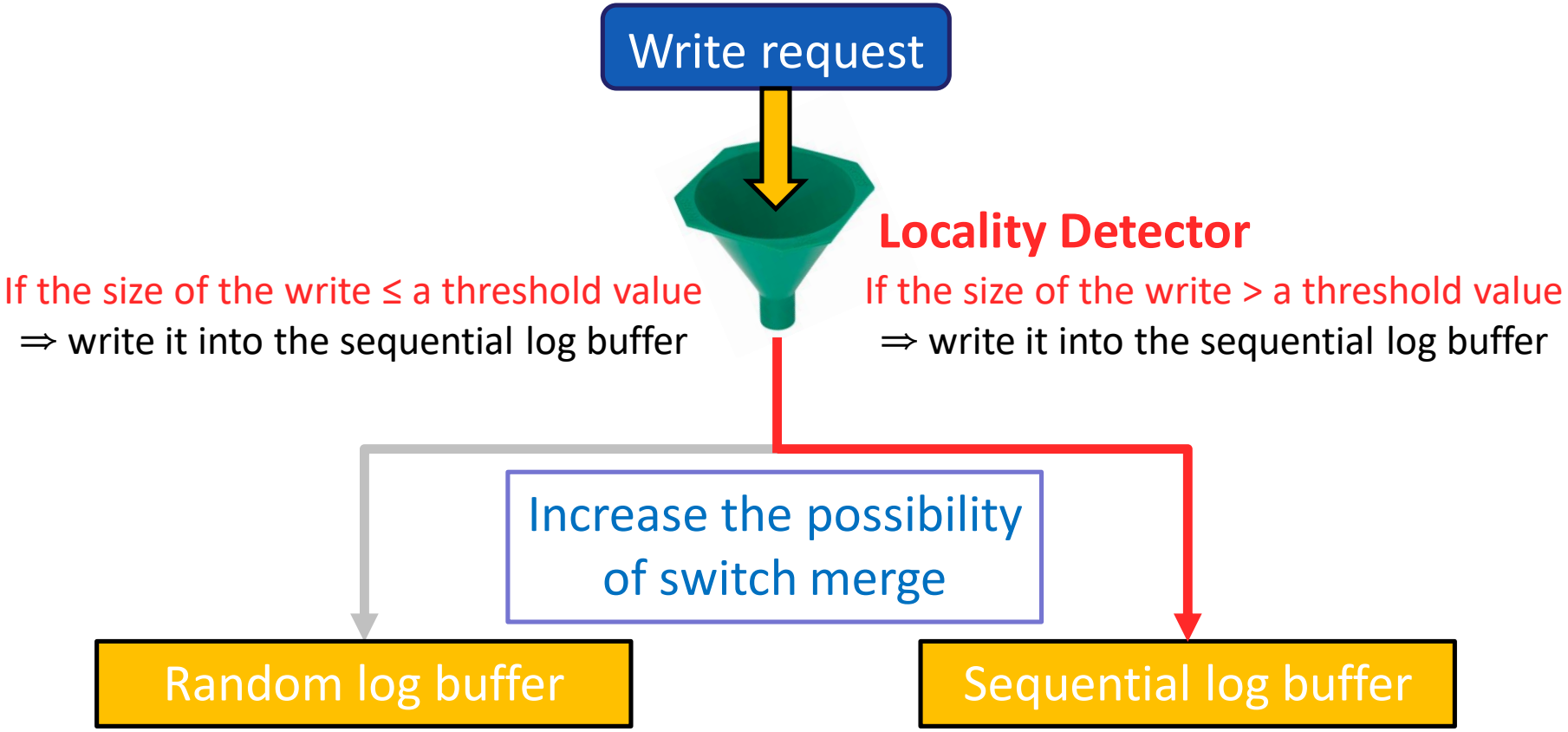
Locality Detector (2)

- **A locality-detection policy based on the request size**

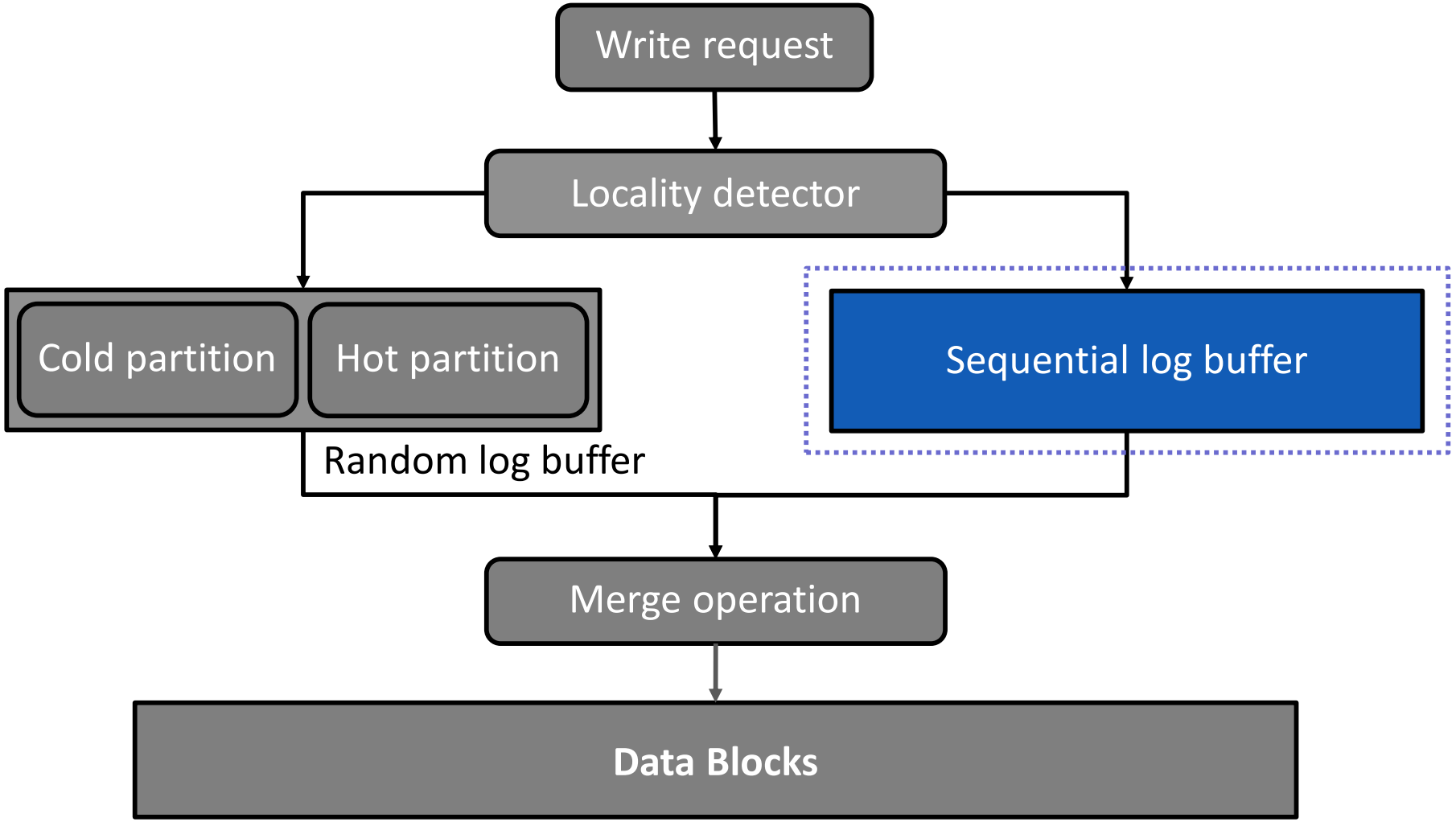


Locality Detector (2)

- **A locality-detection policy based on the request size**

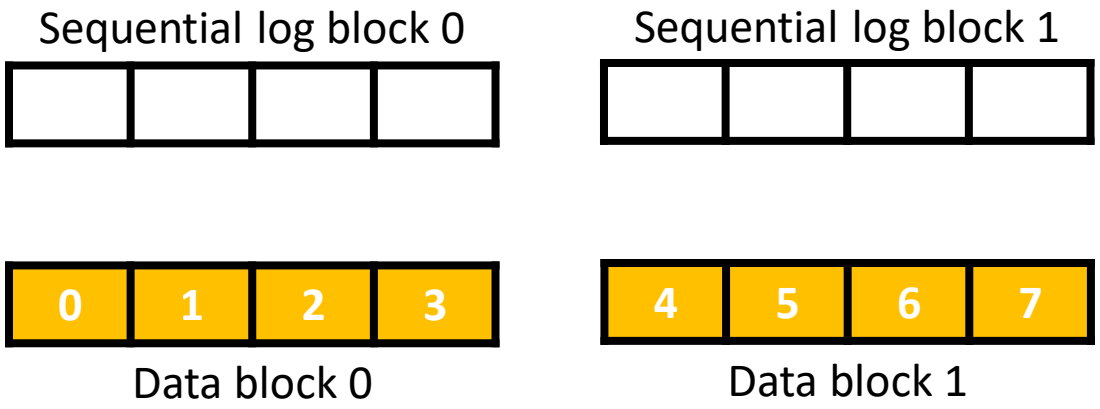


Overall Architecture of the LAST Scheme



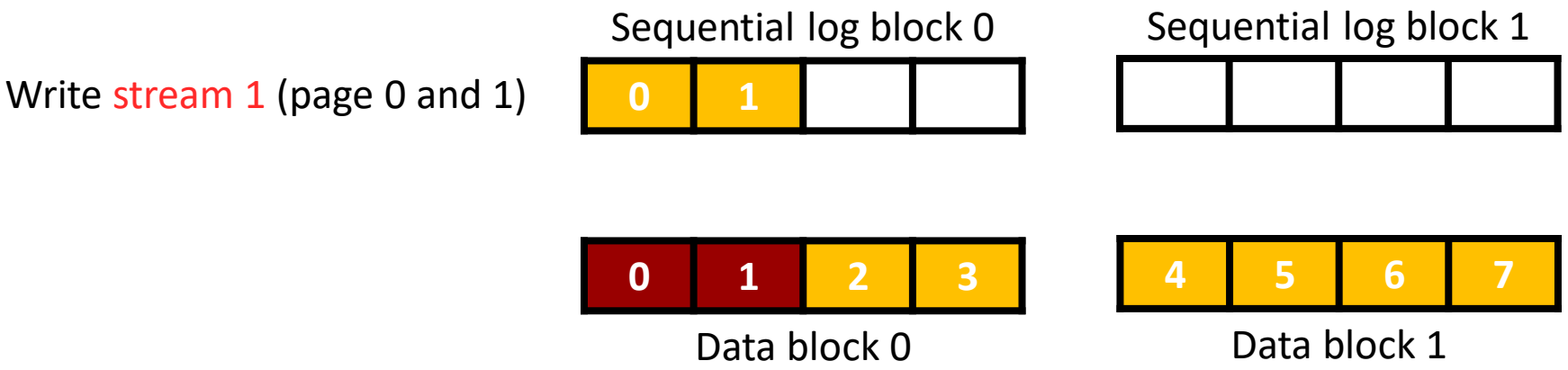
Sequential Log Buffer

- **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



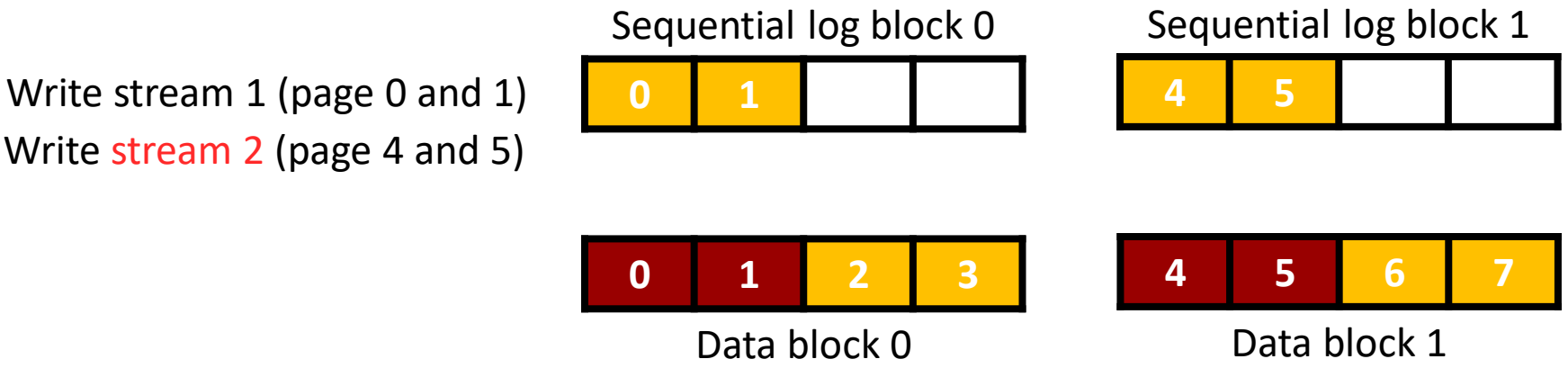
Sequential Log Buffer

- **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



Sequential Log Buffer

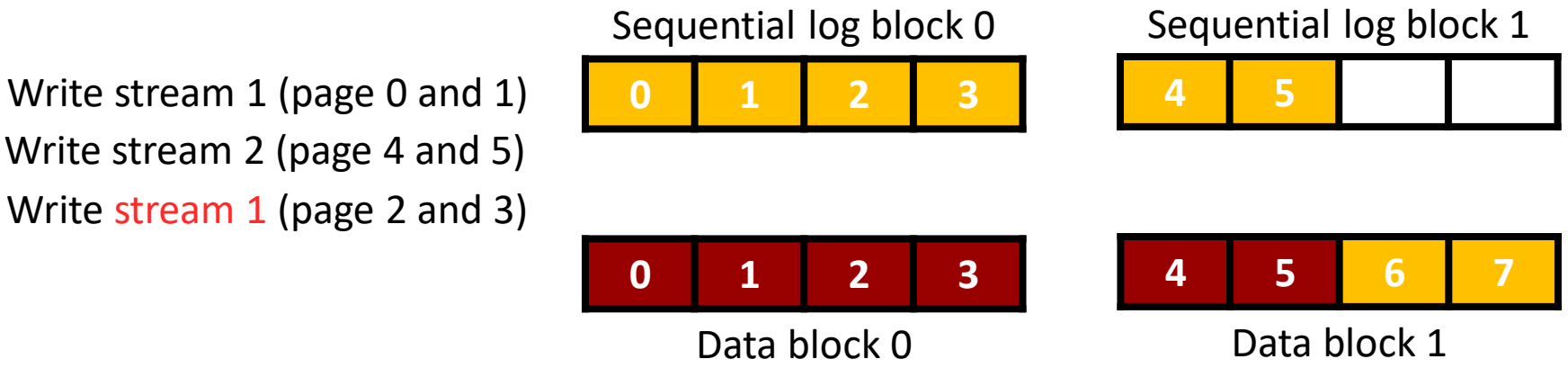
- **Multiple sequential write streams are simultaneously issued from the file system**
  - Accommodate multiple sequential write streams
    - maintain several log blocks in the sequential log buffer
  - Distribute each sequential write into different log block
    - one log block can be associated with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



Sequential Log Buffer

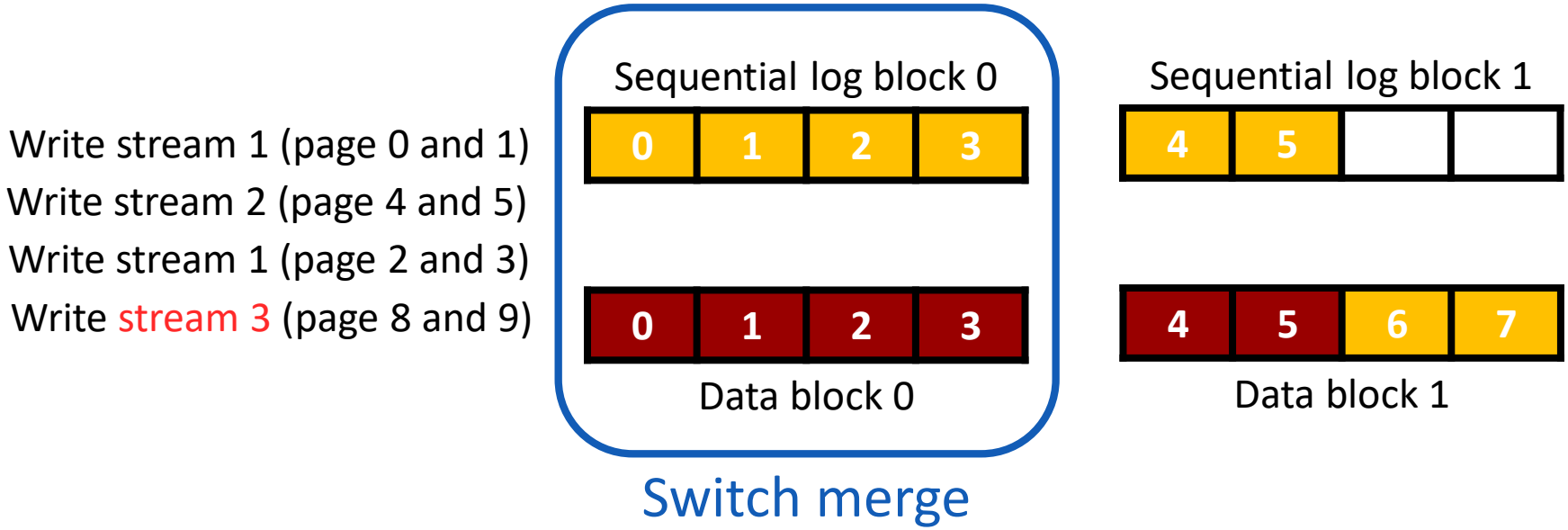
FAST에서는 SW log block이 1개밖에 없던 것 같은데
여기서는 SW log block도 여러개

- **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

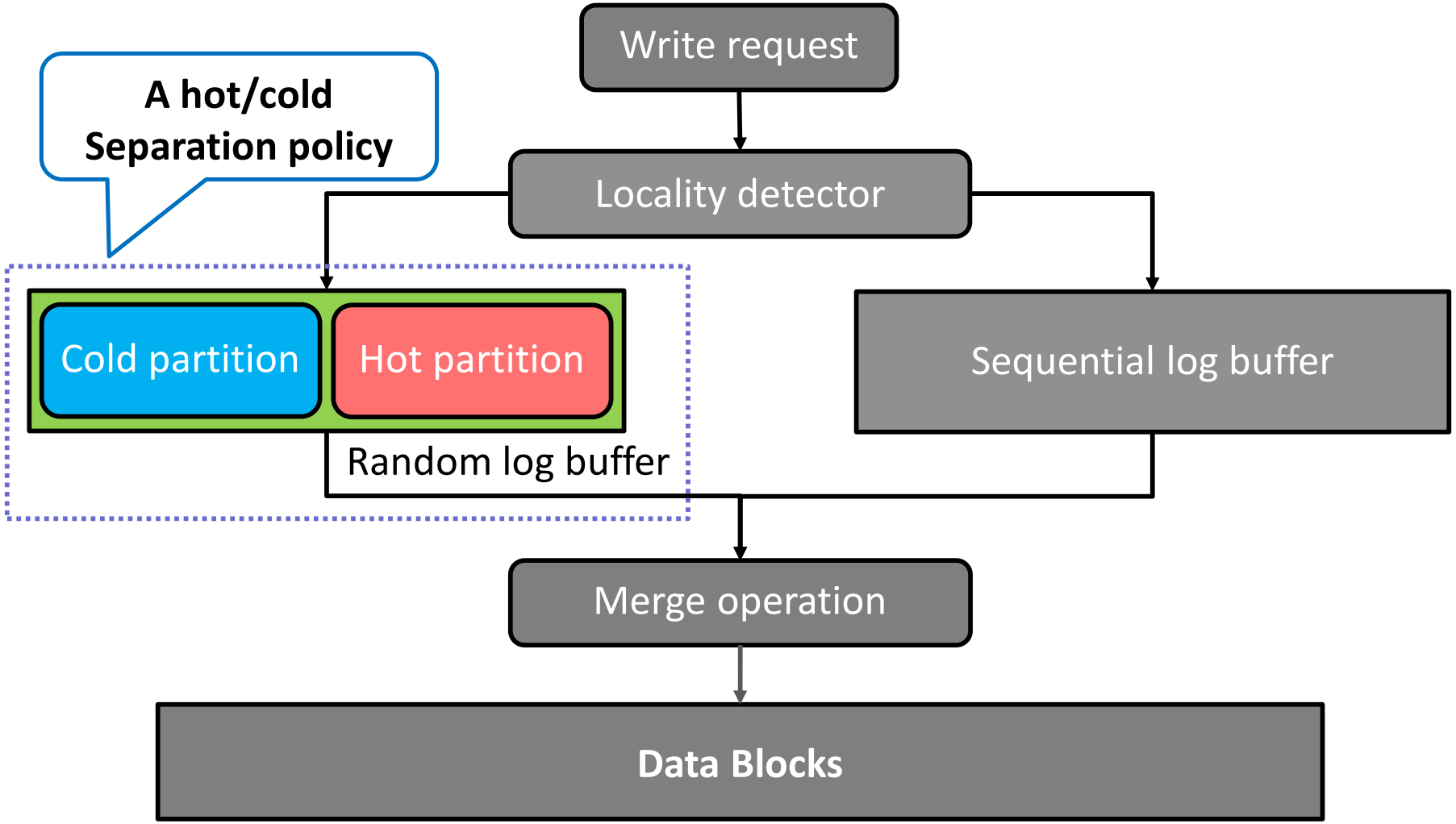


Sequential Log Buffer

- **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

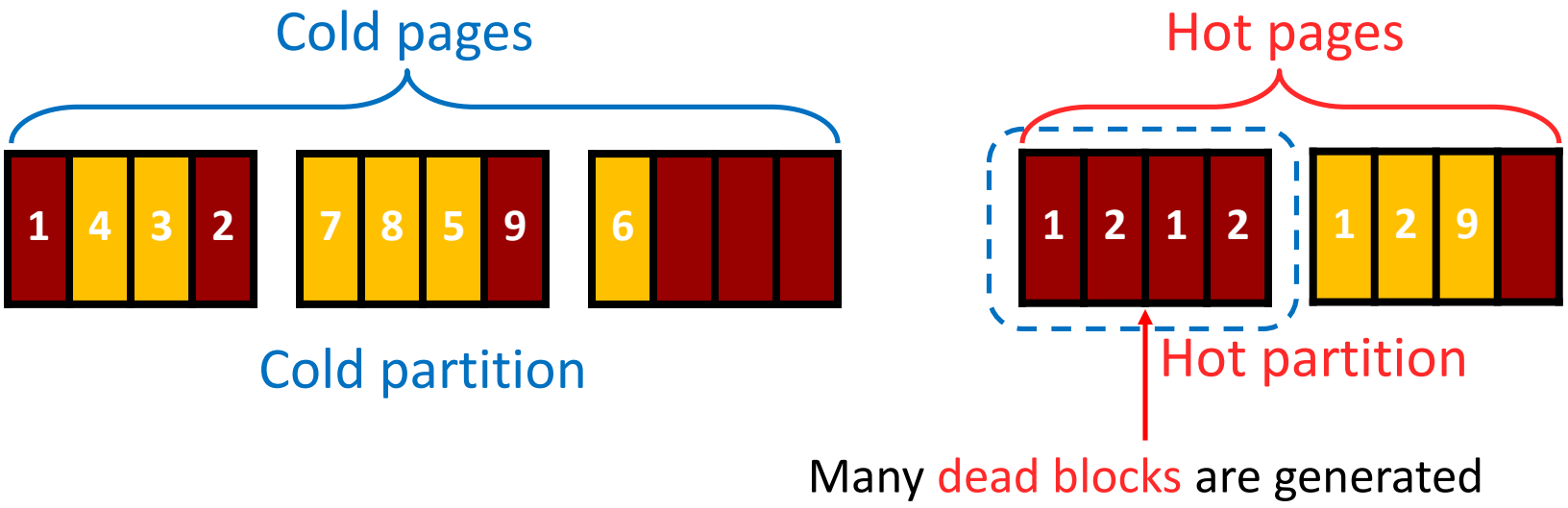


Overall Architecture of the LAST Scheme



Log Buffer Partitioning Policy

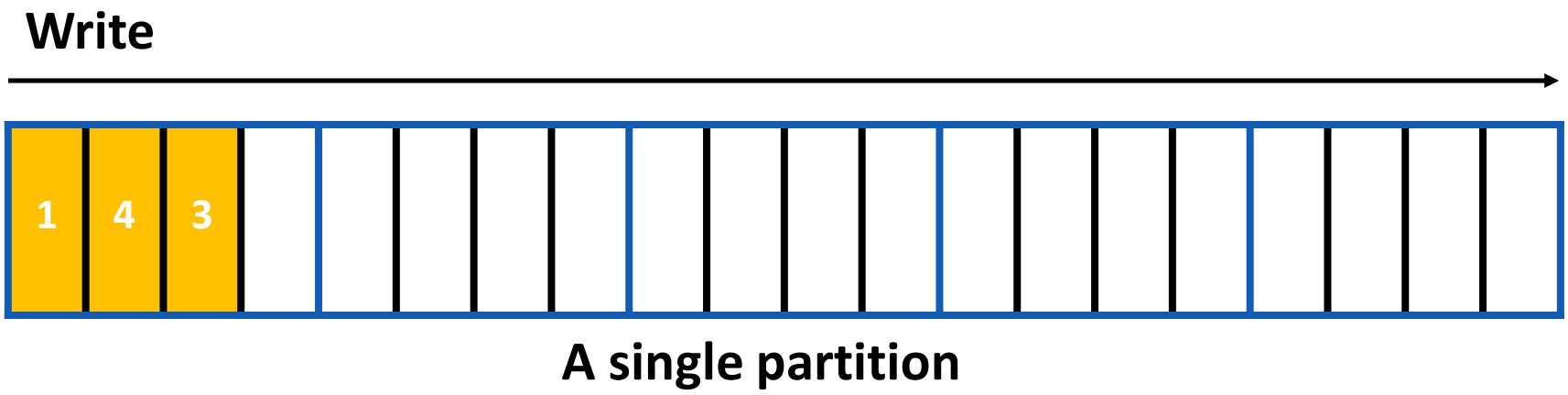
- **Log buffer partitioning policy**
  - Proposed to provide a hot and cold separation policy
  - Separate hot pages from cold pages
  - Invalid pages are likely to be clustered in the same log block
    - All the pages in a log block can be invalidated ⇒ **dead block**
  - Remove dead block with only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



Log Buffer Partitioning Policy

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

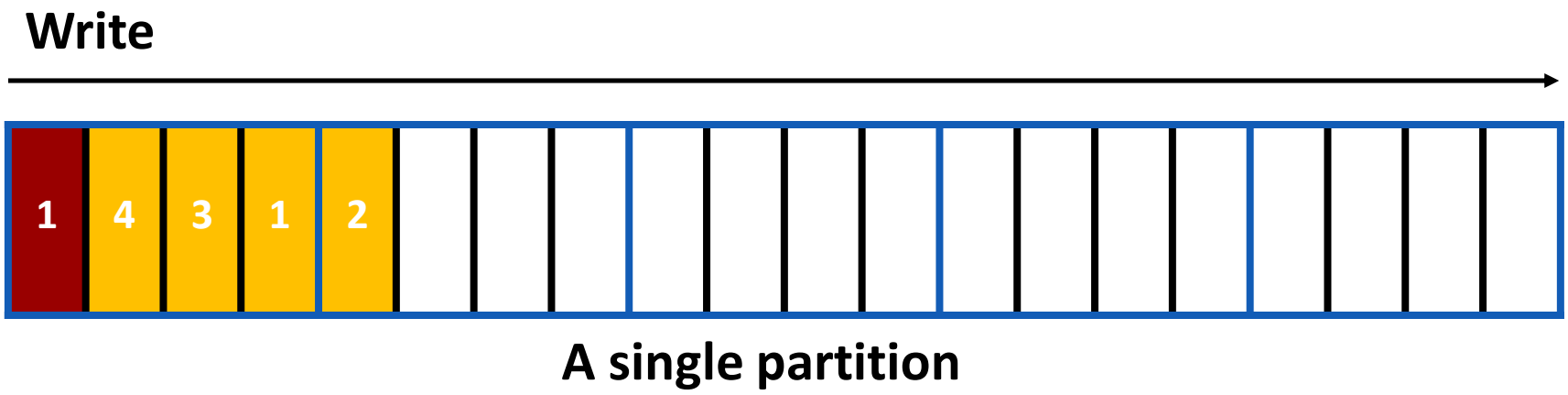
Requested pages :
1 → 4 → 3



Log Buffer Partitioning Policy

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

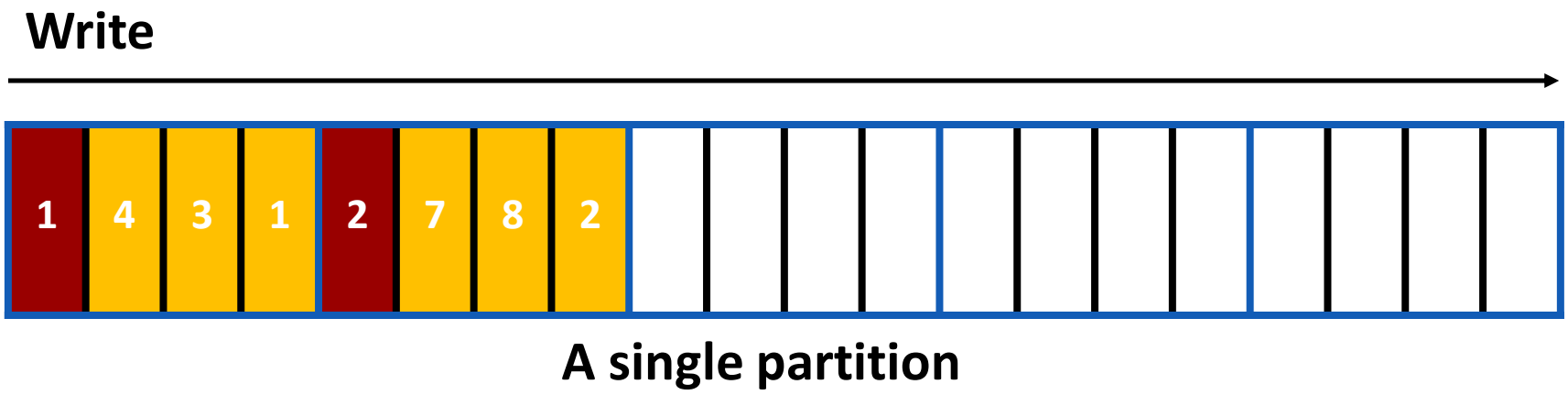
Requested pages :
1 → 4 → 3 →1 → 2



Log Buffer Partitioning Policy

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

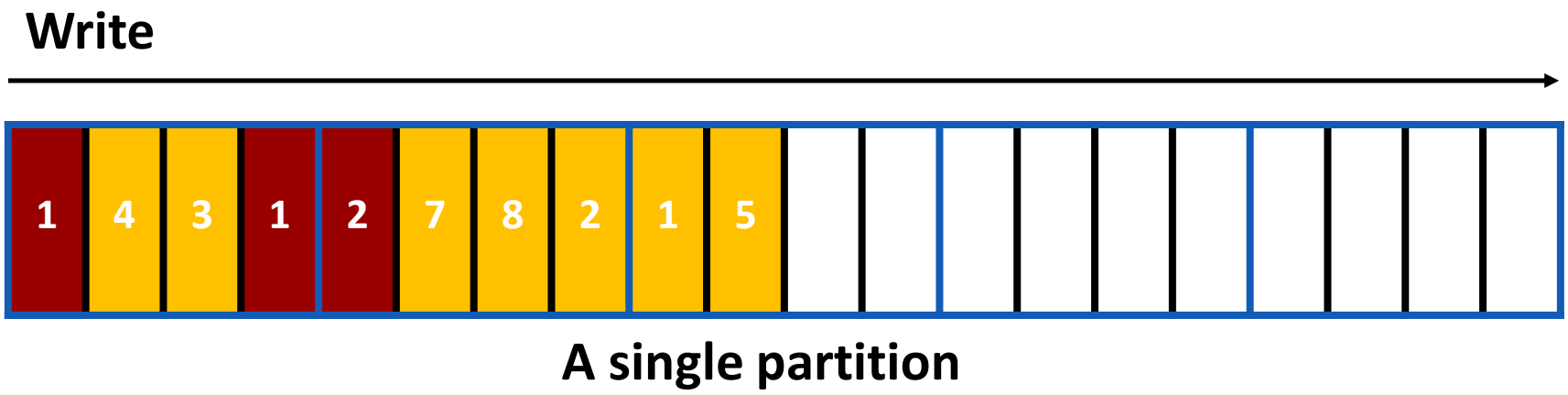
Requested pages :
1 → 4 → 3 →1 → 2 → 7 →8 → 2



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 →

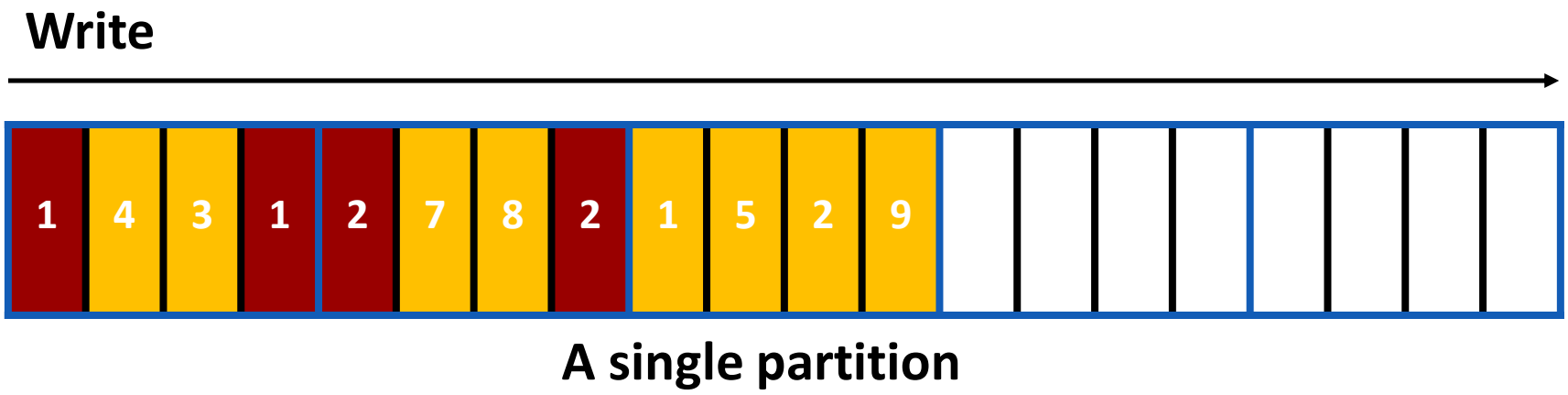


Log Buffer Partitioning Policy

- **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

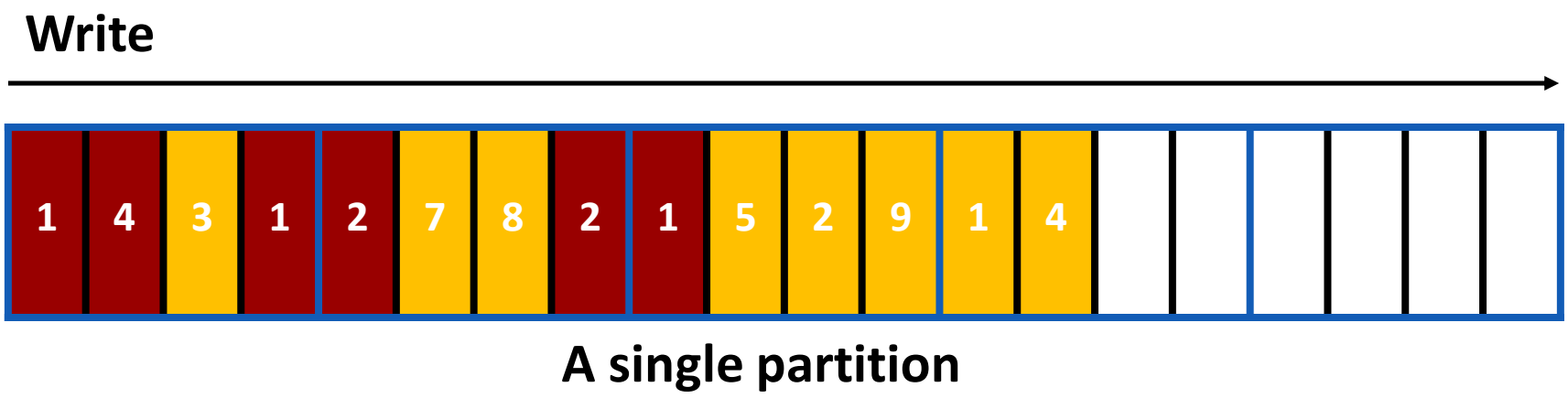


Log Buffer Partitioning Policy

- **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



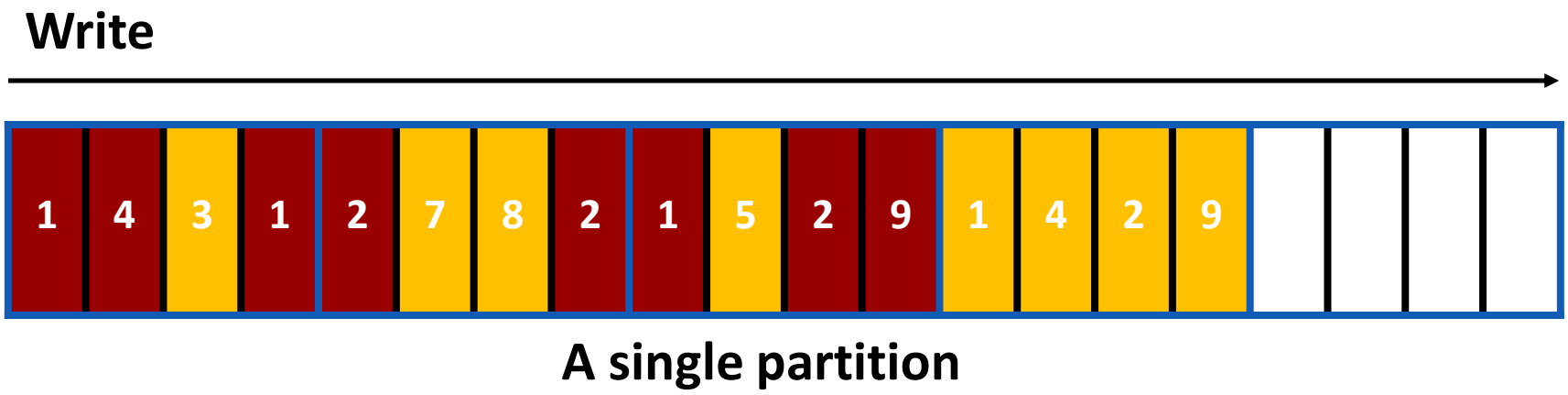
A single partition

Log Buffer Partitioning Policy

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

Requested pages :

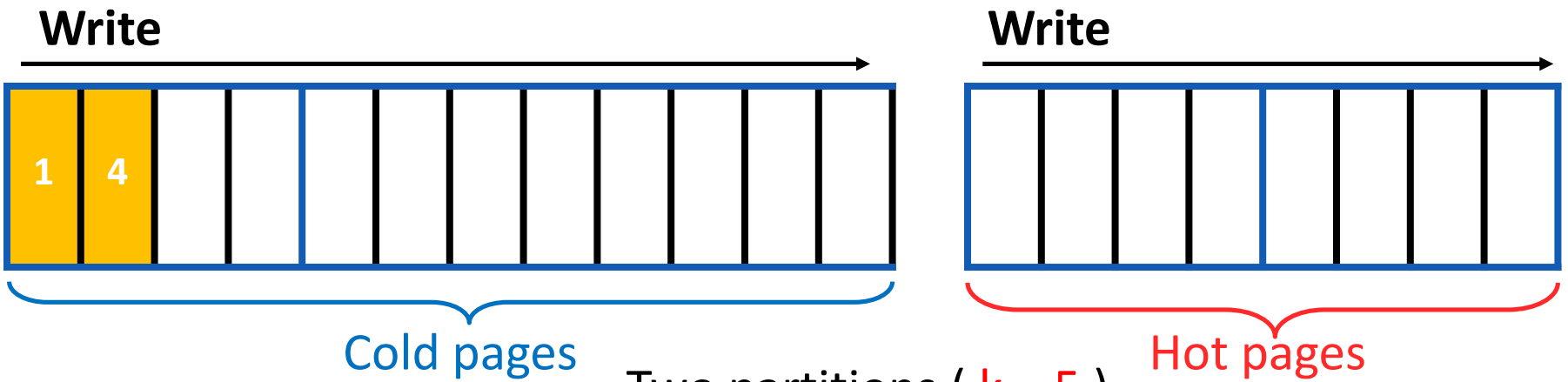
1 → 4 → 3 →1 → 2 → 7 →8 → 2 → 1 → 5 → 2 → 9 →1 → 4 → 2 → 9



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

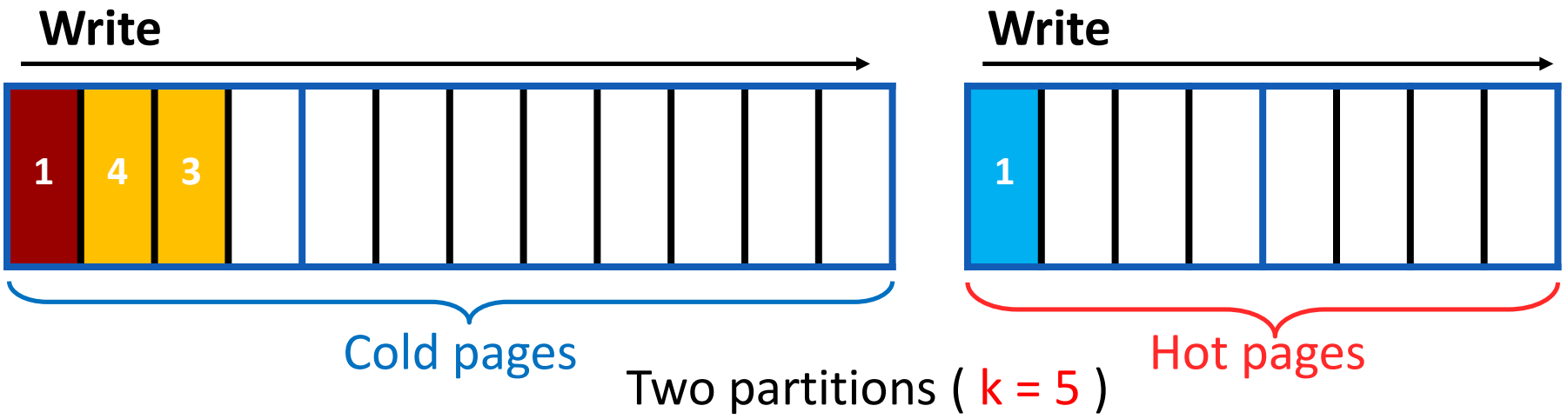
Requested pages :
1 → 4



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

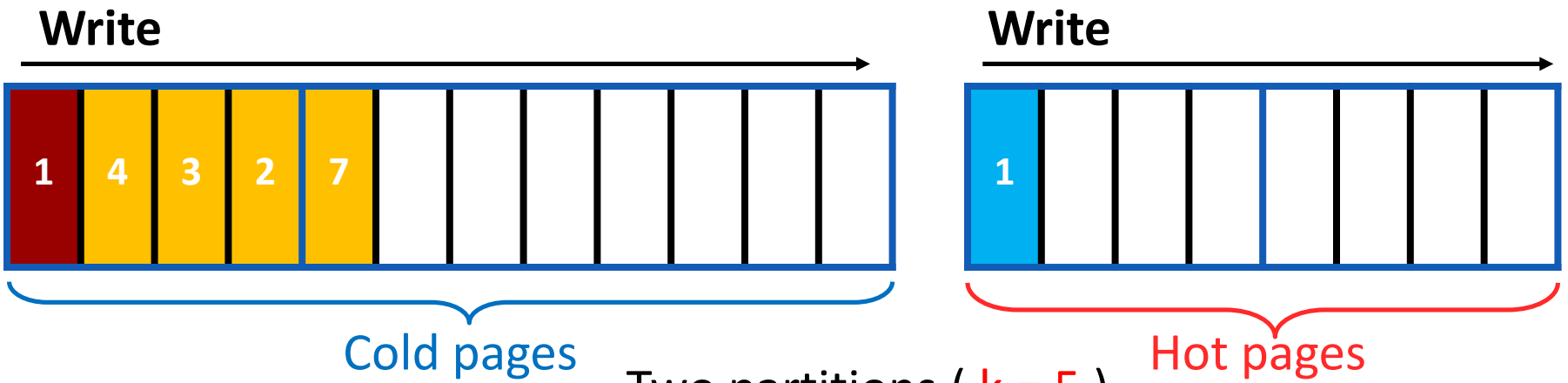
Requested pages :
1 → 4 → 3 →1



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

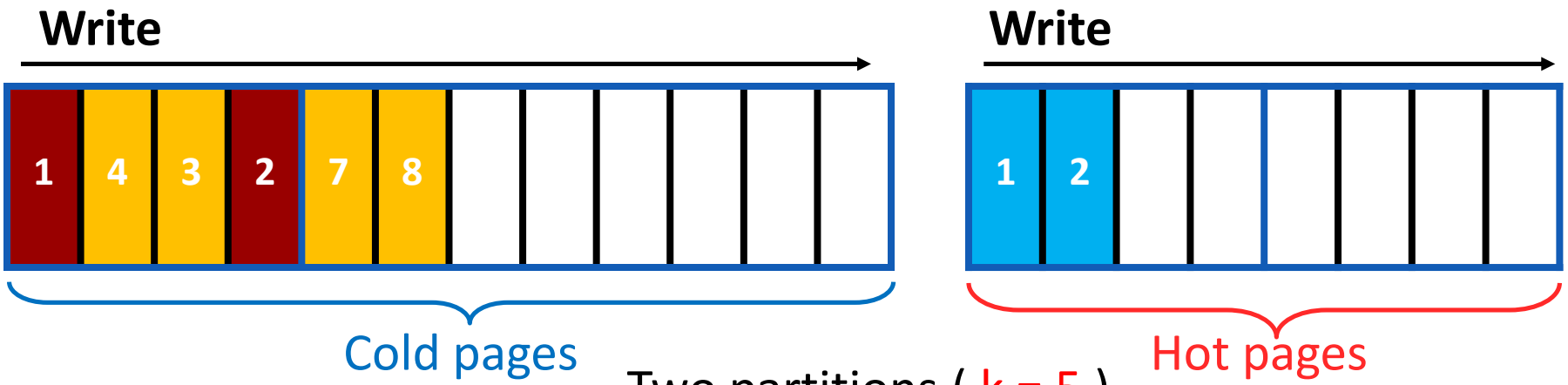
Requested pages :
1 → 4 → 3 →1 → 2 → 7



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

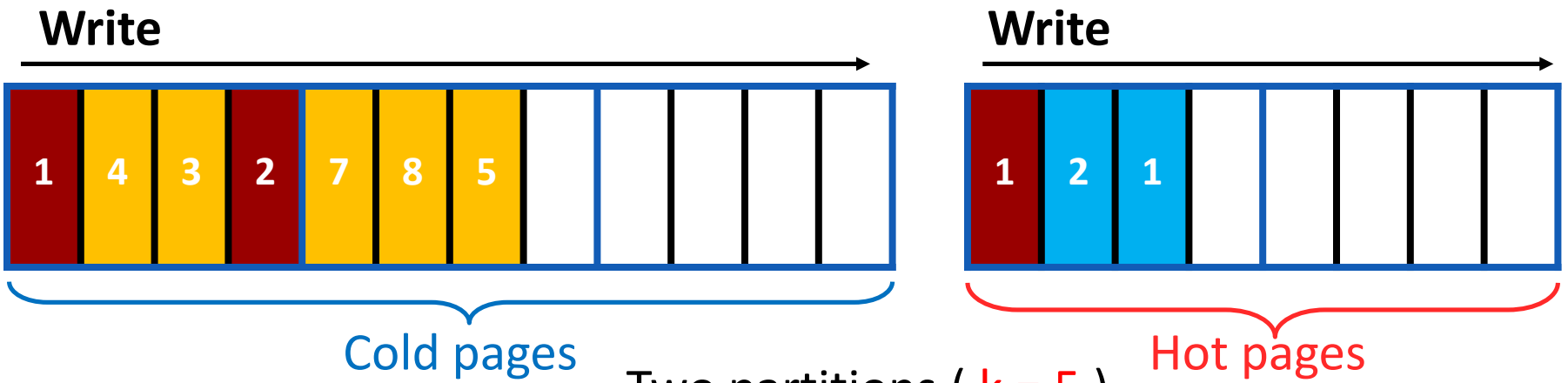
Requested pages :
1 → 4 → 3 →1 → 2 → 7 →8 → 2



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

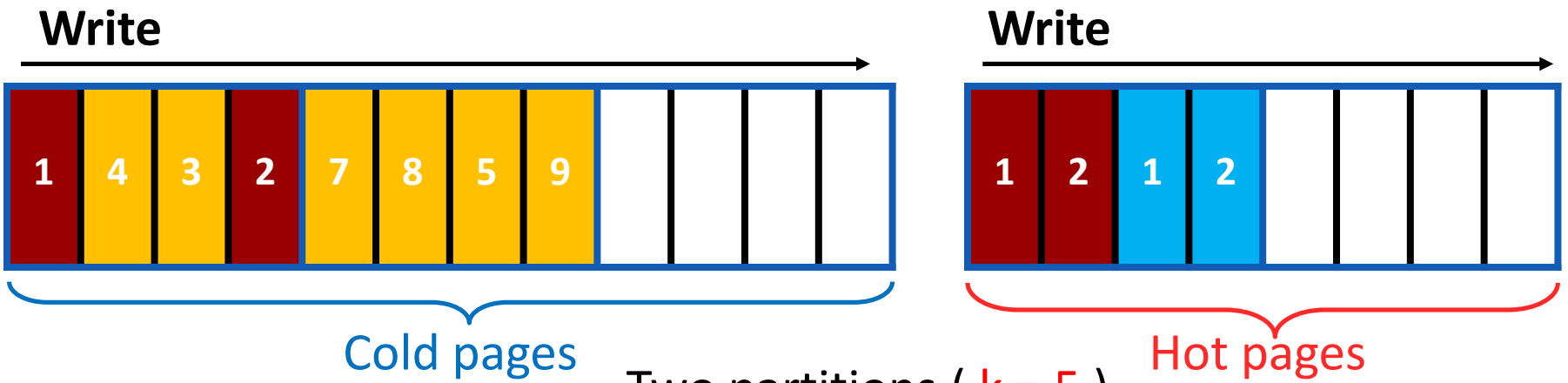


Two partitions (k = 5)

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

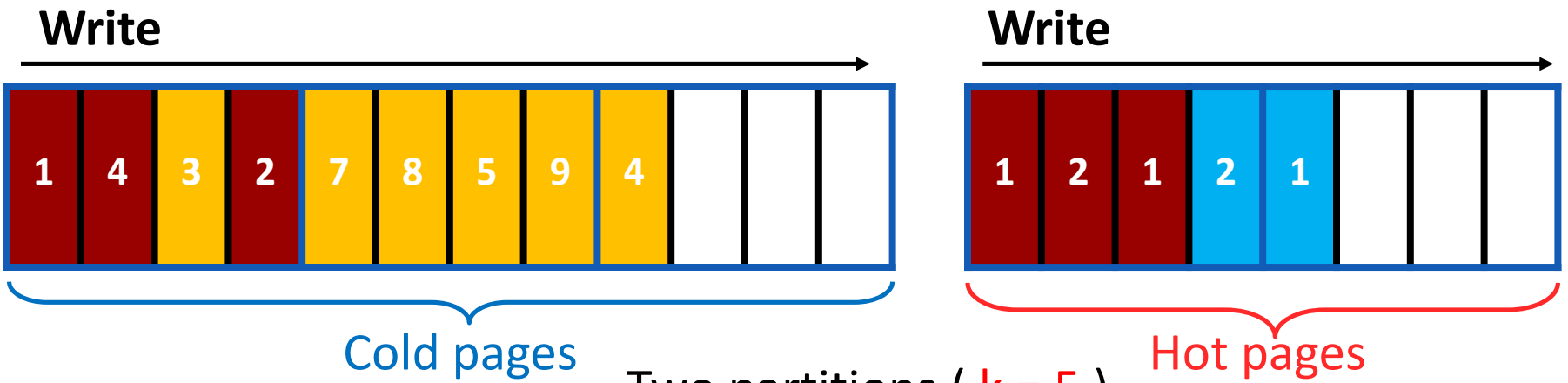
Requested pages :
1 → 4 → 3 →1 → 2 → 7 →8 → 2 → 1 → 5 → 2 → 9



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

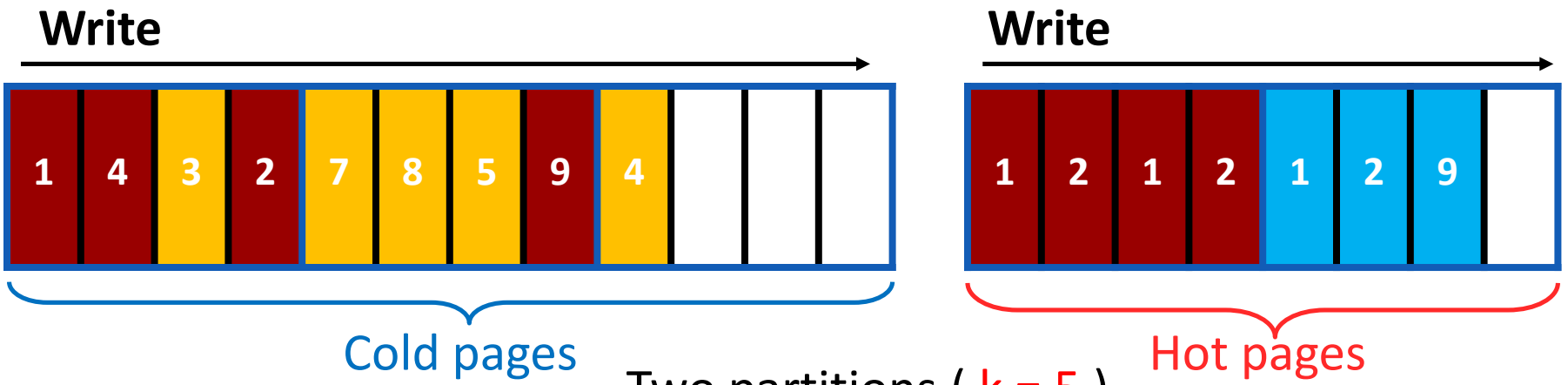


Two partitions (k = 5)

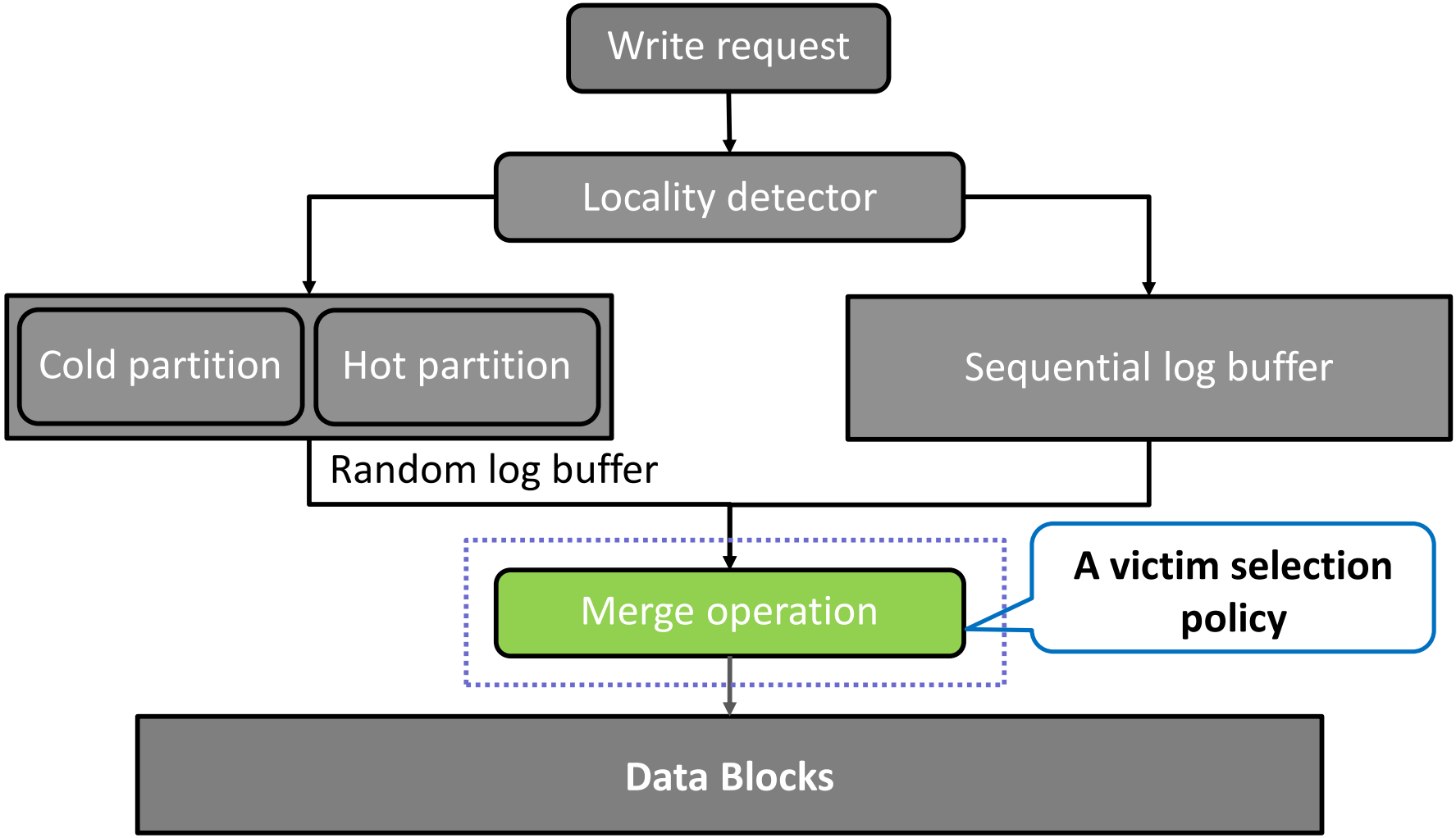
Log Buffer Partitioning Policy

- **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



Overall Architecture of the LAST Scheme

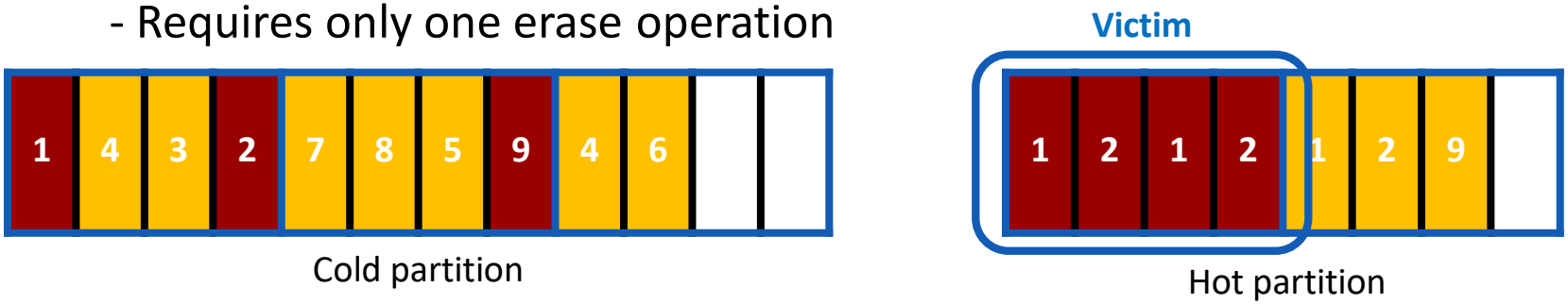


Log Buffer Replacement Policy

- **Log buffer replacement policy**
  - Proposed to provide a more intelligent victim selection
  - Delay an eviction of hot pages as long as possible

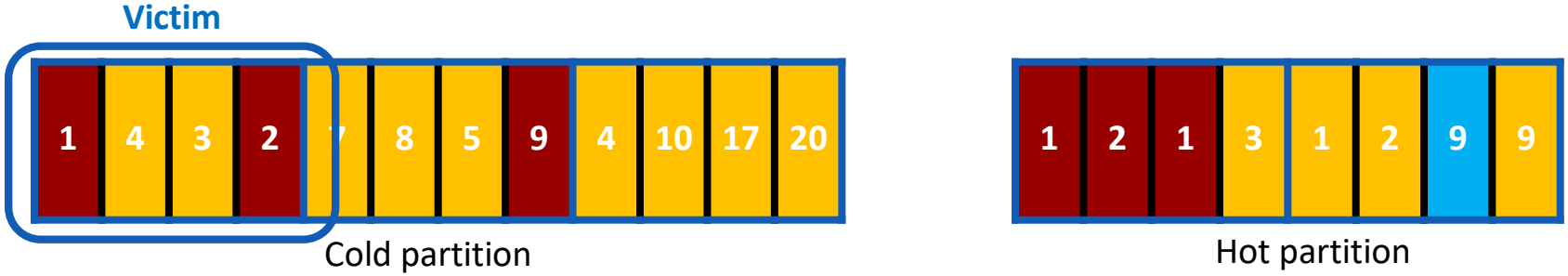
(1) Evict a dead block first from the hot partition

- Requires only one erase operation



(2) Evict a cold block from the cold partition

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



Experimental Results

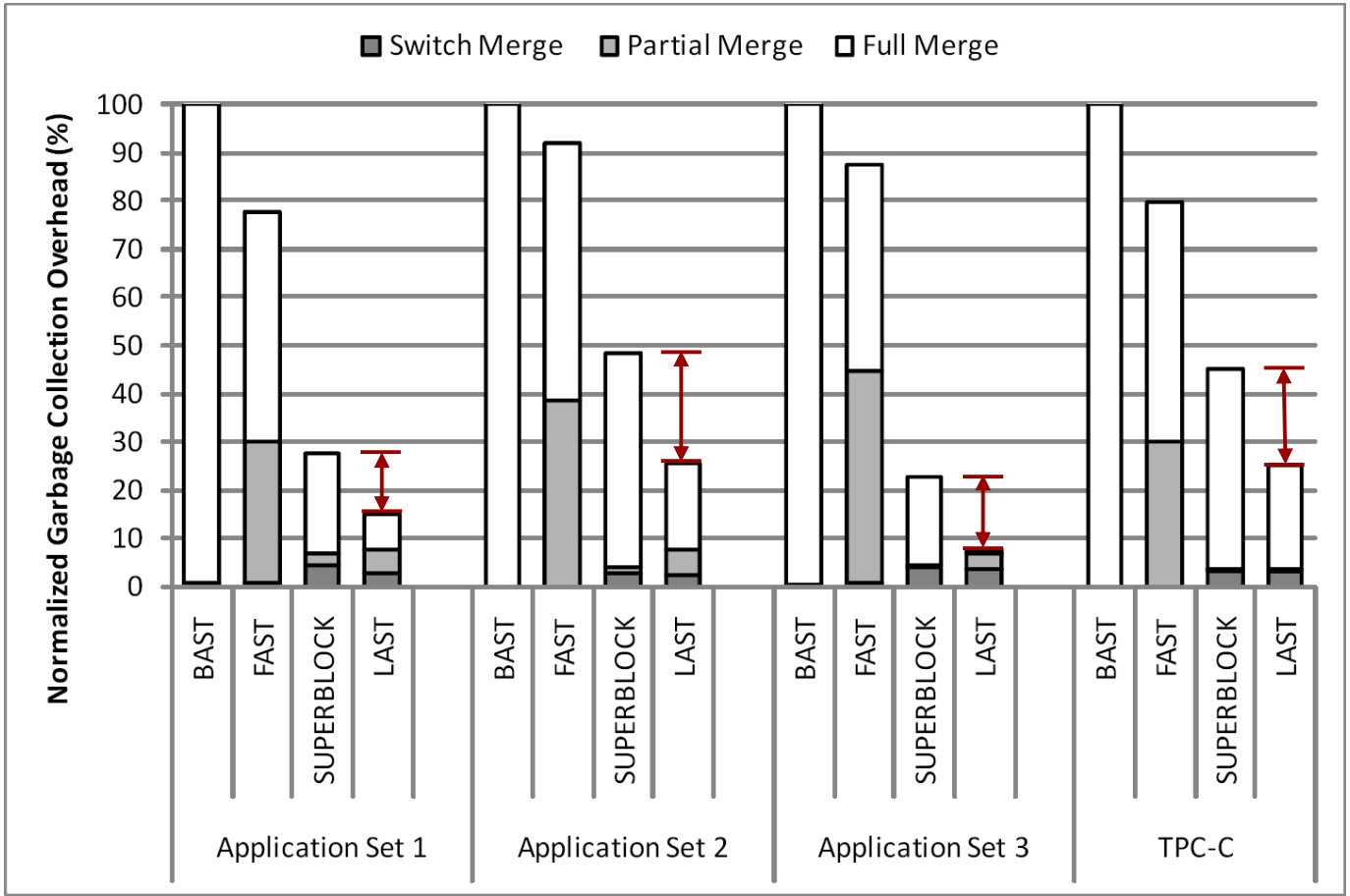
- **Experimental environment**
  - Trace-driven FTL simulator
    - Three existing FTL schemes: BAST, FAST, SUPERBLOCK
    - The propose scheme: LAST
  - Benchmarks
    - Realistic PC workload sets, TPC-C benchmark
  - Flash memory model

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)

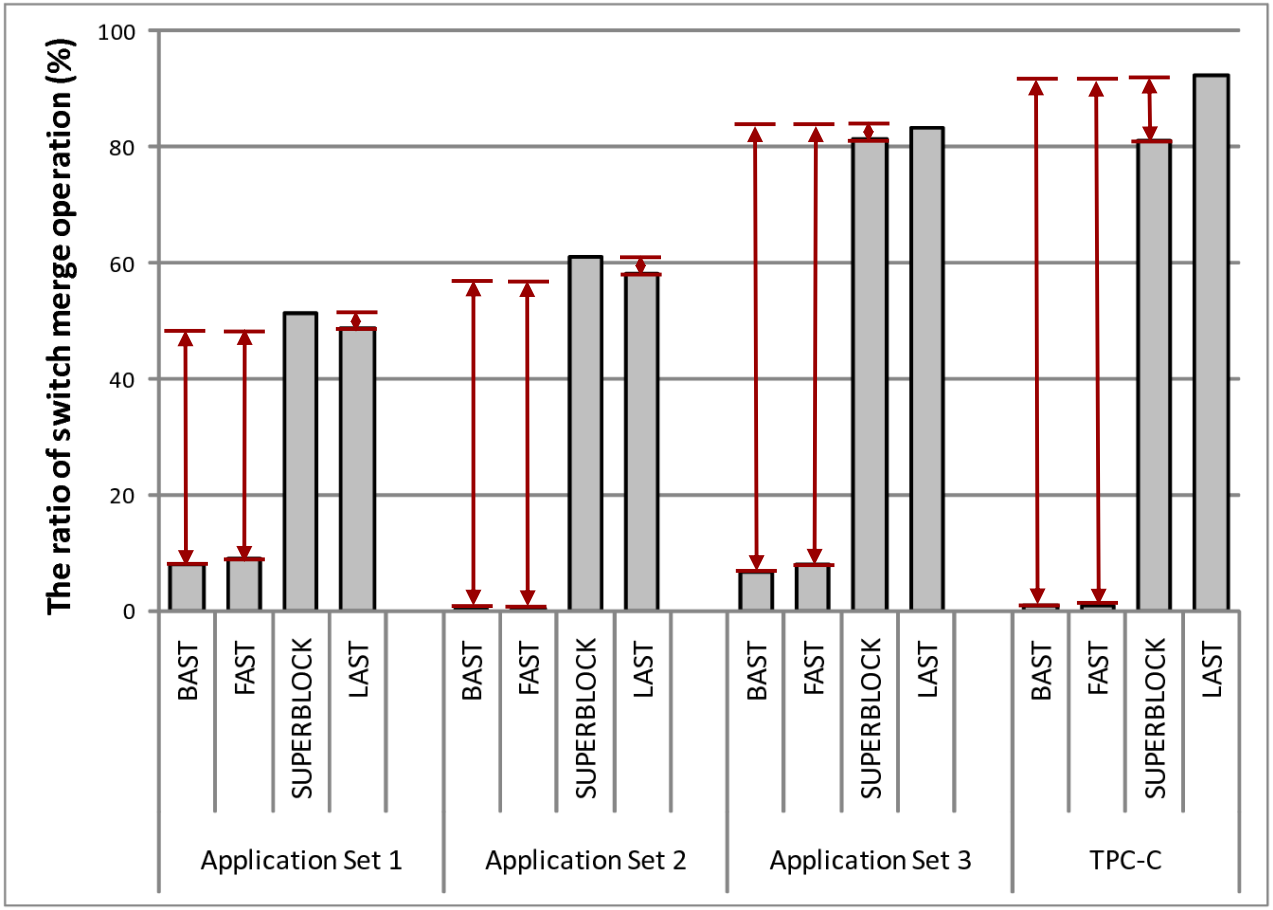
Result 1: Garbage Collection Overhead

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



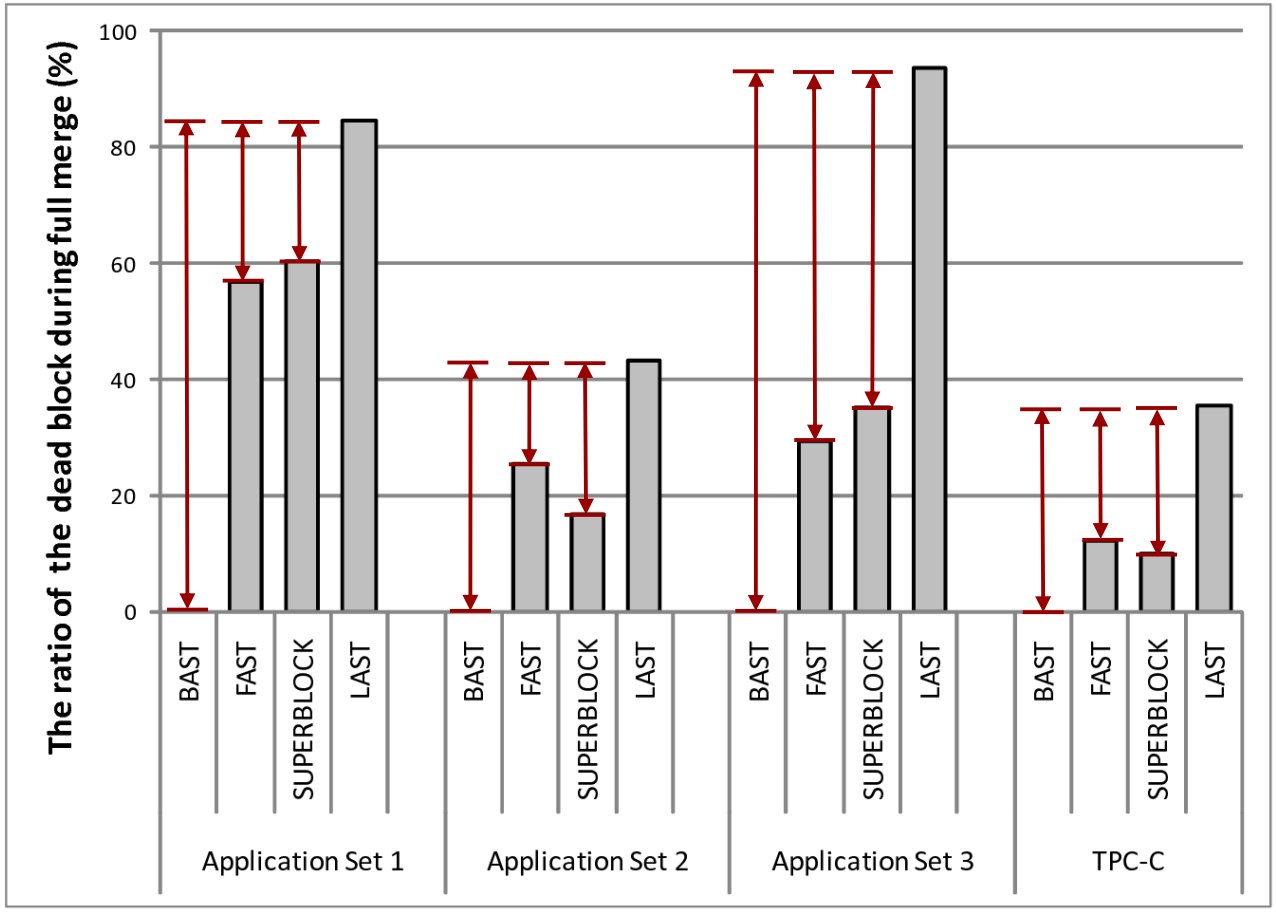
Result 2: Ratio of Switch Merge

- **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



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

Outline

- Review: Replacement Block Scheme
- Hybrid FTLs (Log Block Scheme)
- **Demand-based FTL (DFTL)**

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

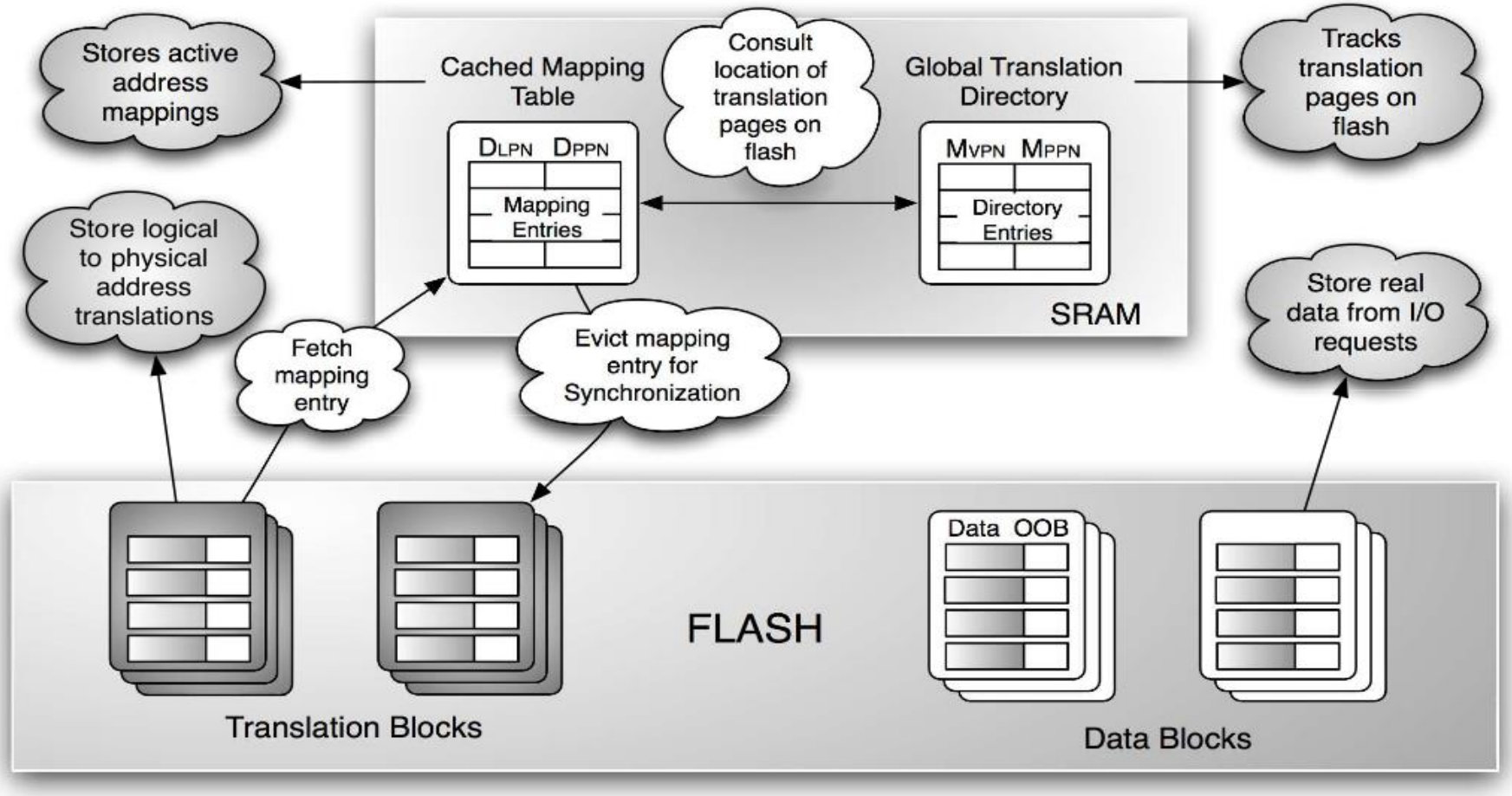
- **DFTL is an enhanced form of the page-level FTL scheme**
  - Allow requests to be serviced from any physical page on flash
  - All blocks can be used for servicing update requests

- **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

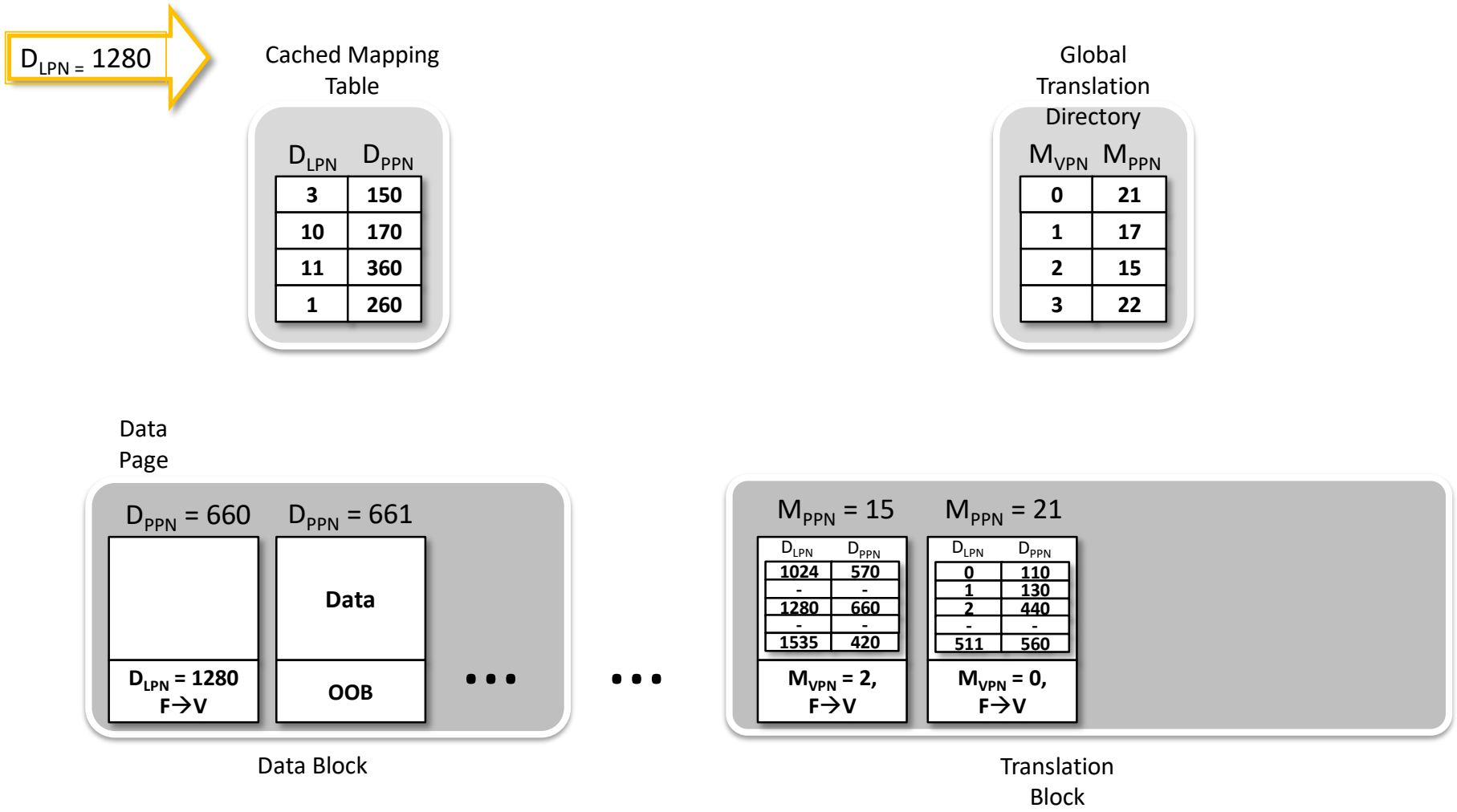
DFTL Architecture



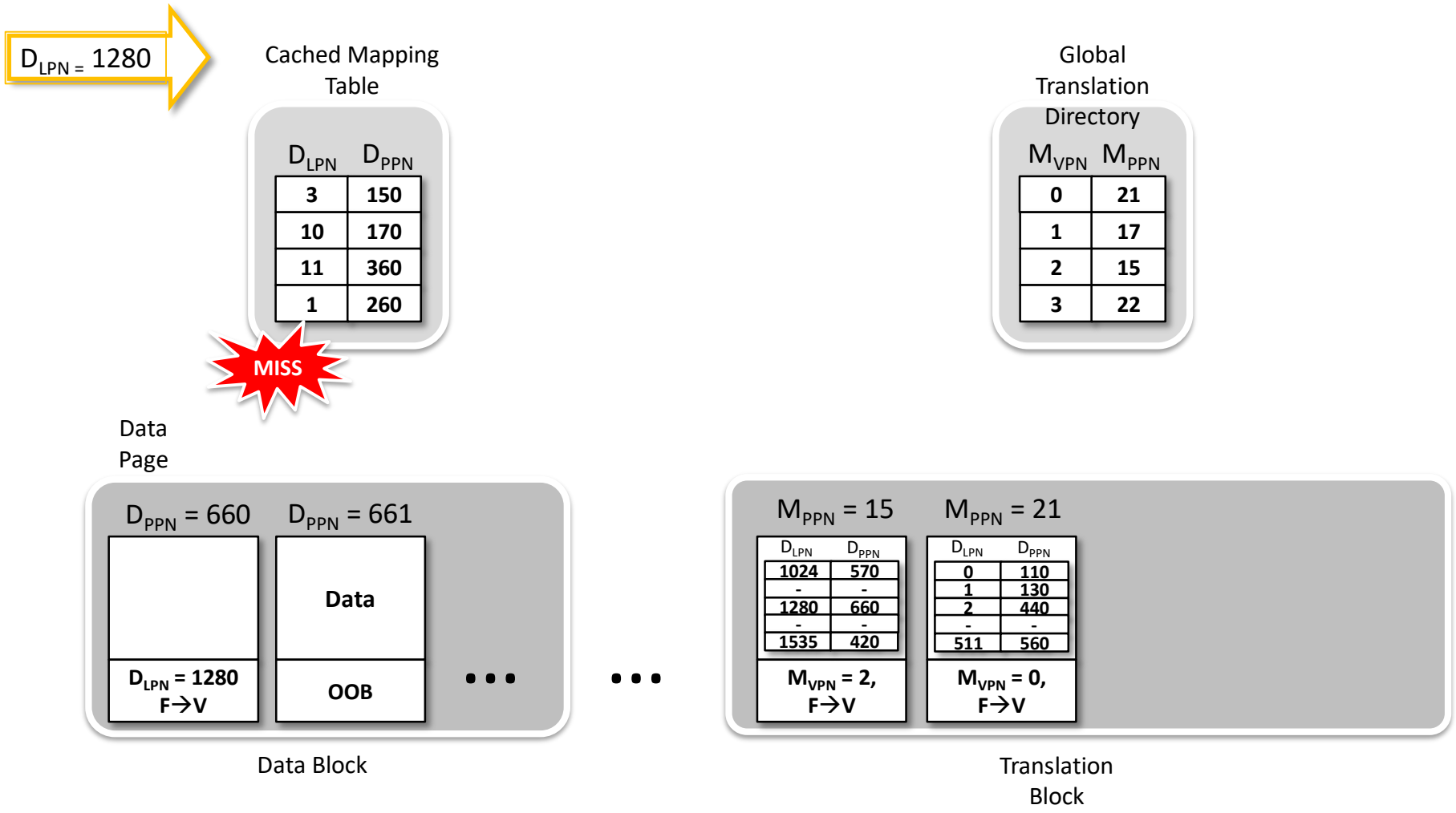
Data Blocks and Translation Blocks

- **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

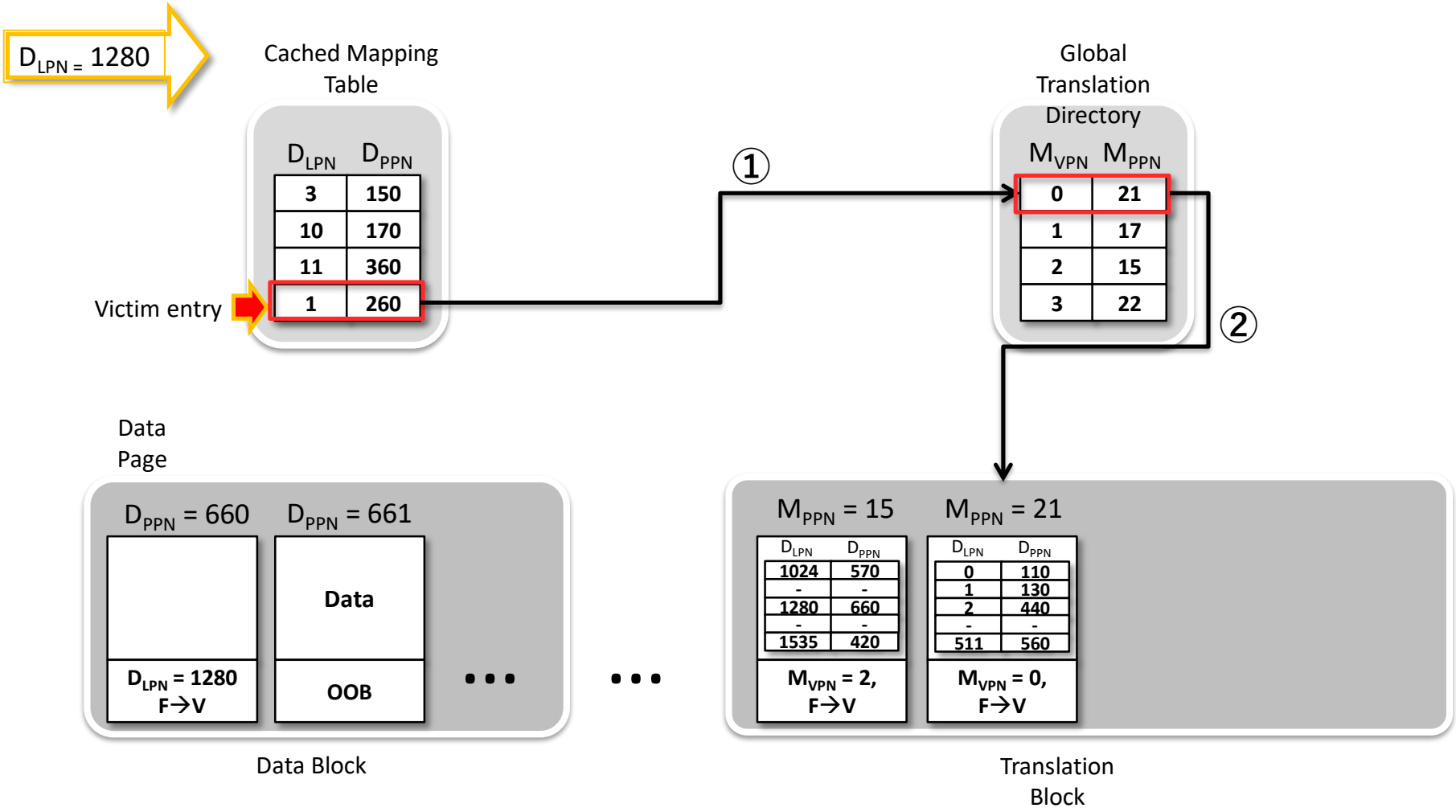


Example:
When a Request Incurs a CMT miss

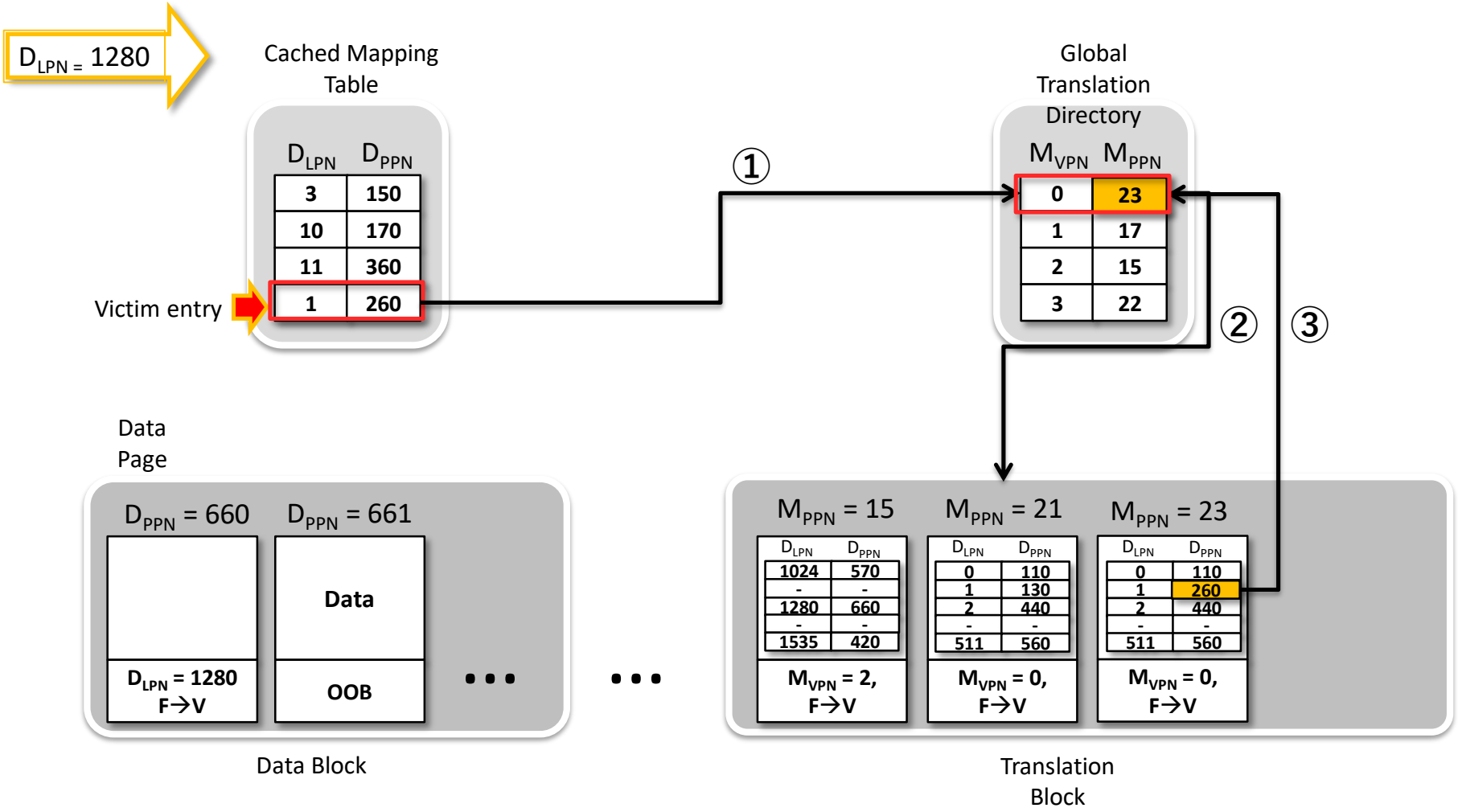


Example:

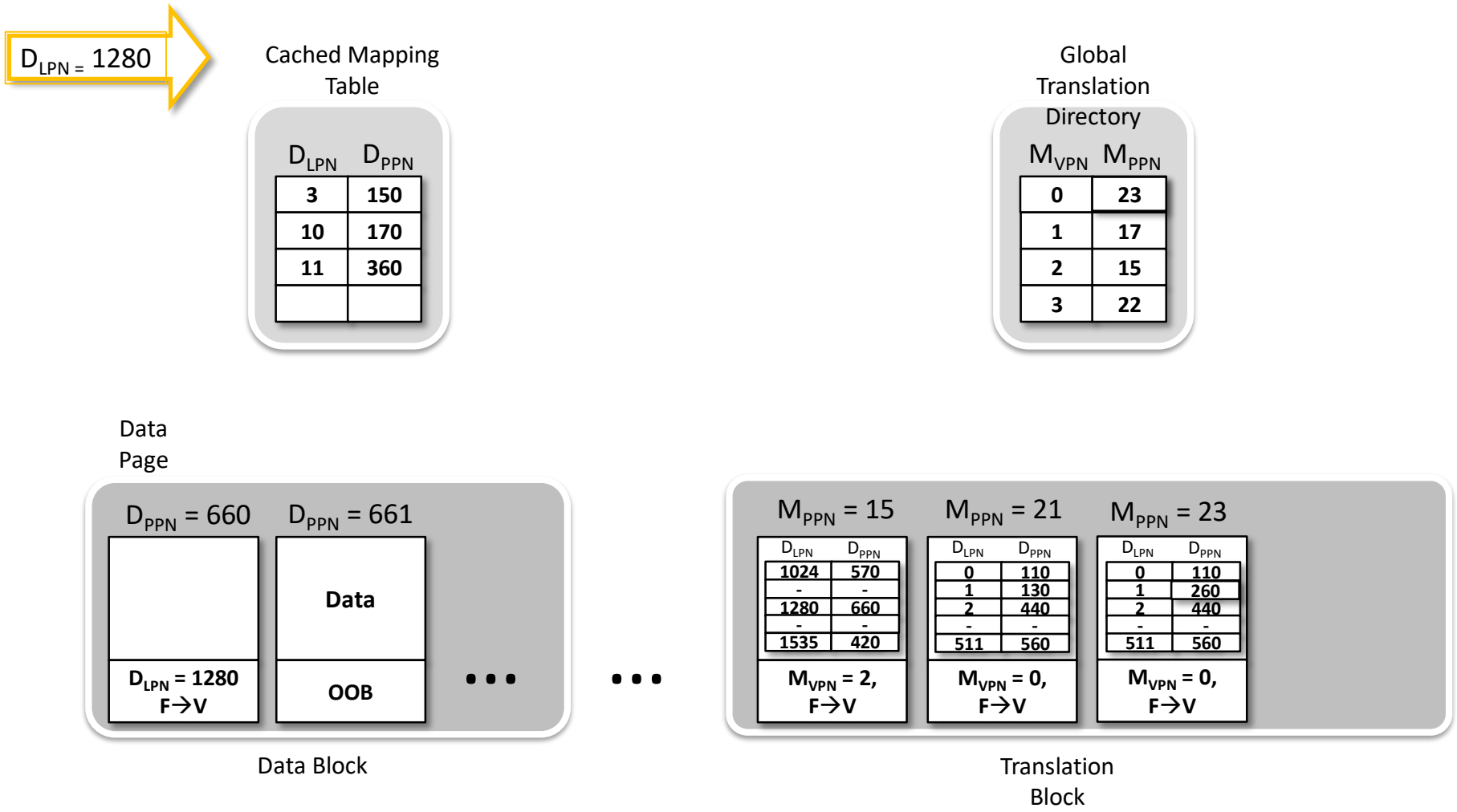
When a Request Incurs a CMT miss



Example:
When a Request Incurs a CMT miss

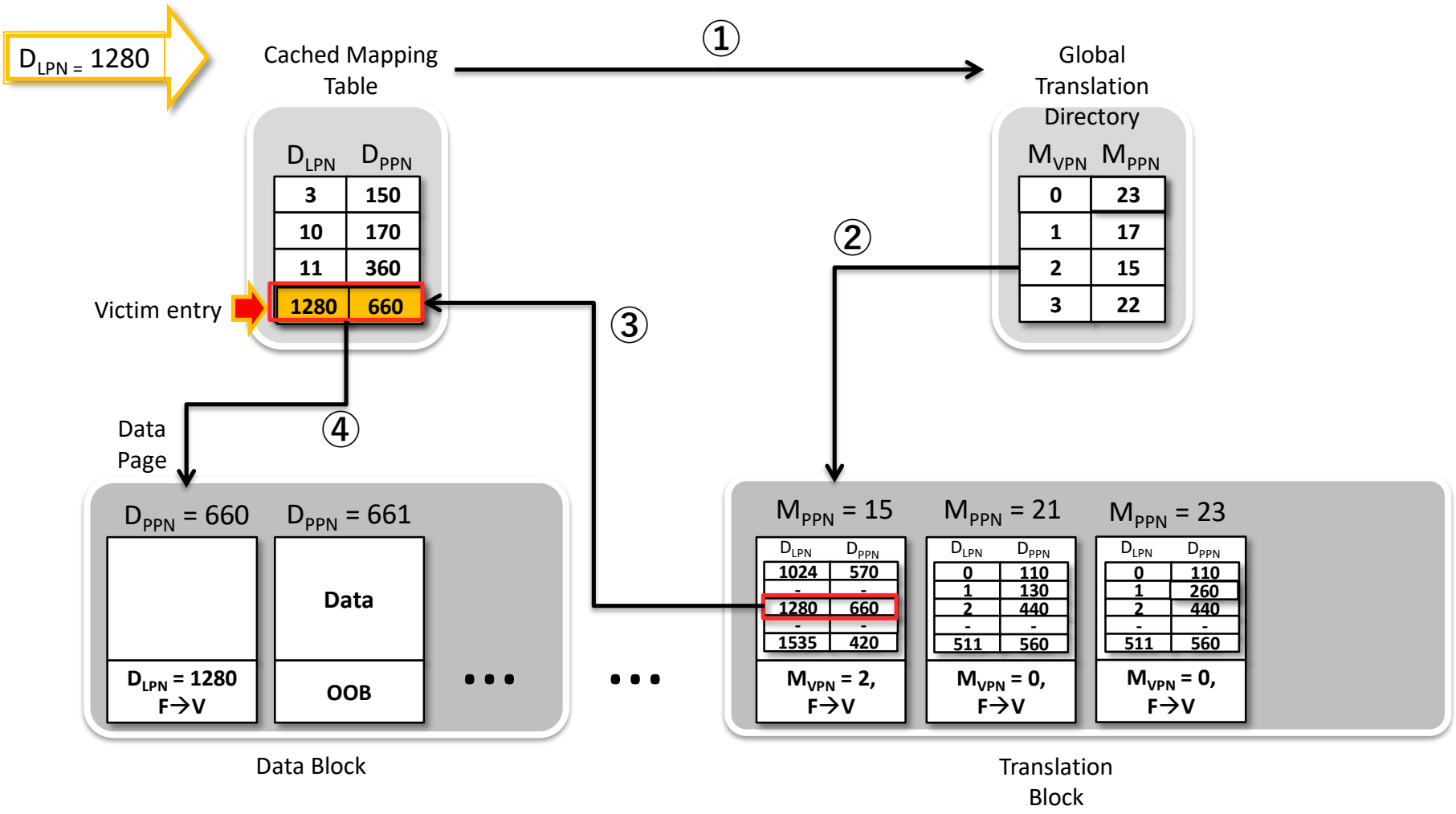


Example:
When a Request Incurs a CMT miss



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

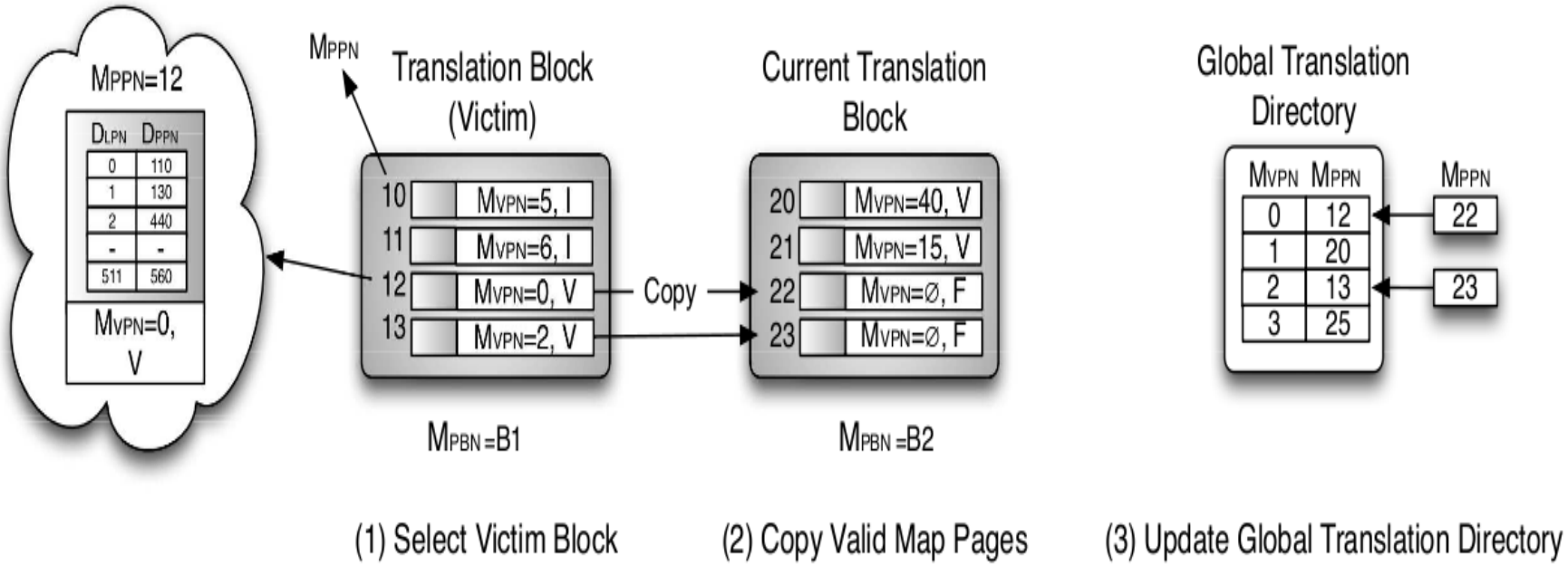
- **For a read operation**
  - Directly serviced through flash page read operation once the address translation is completed
- **For a write operation**
  - Maintain two types of blocks for data block and translation blocks
    - *Current data block* and *current translation block*
  - Sequentially writes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

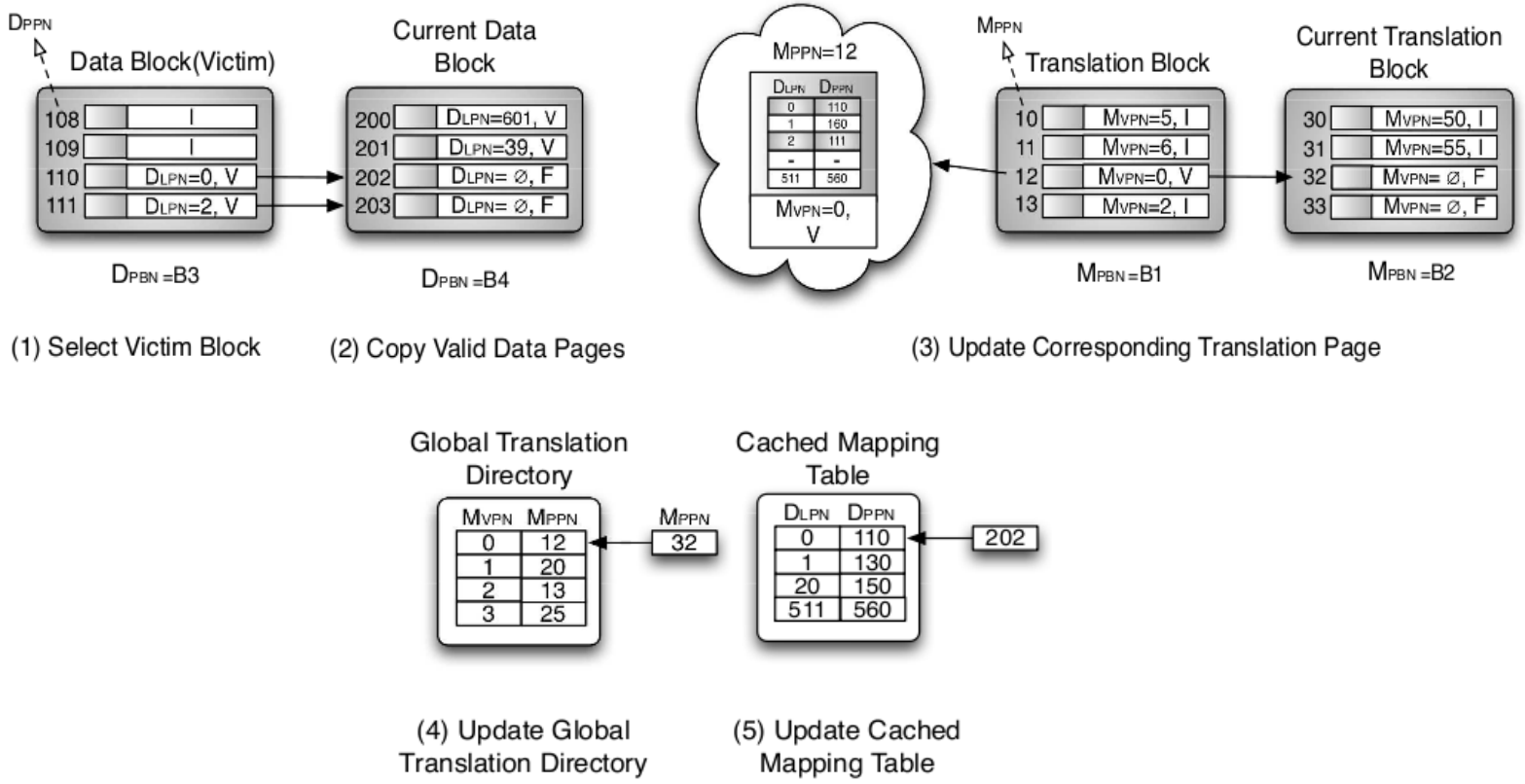
Example : Translation Block

- **Translation block as victim for garbage collection**



Example : Data Block

- **Data block as victim for garbage collection**



Evaluation Setup

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

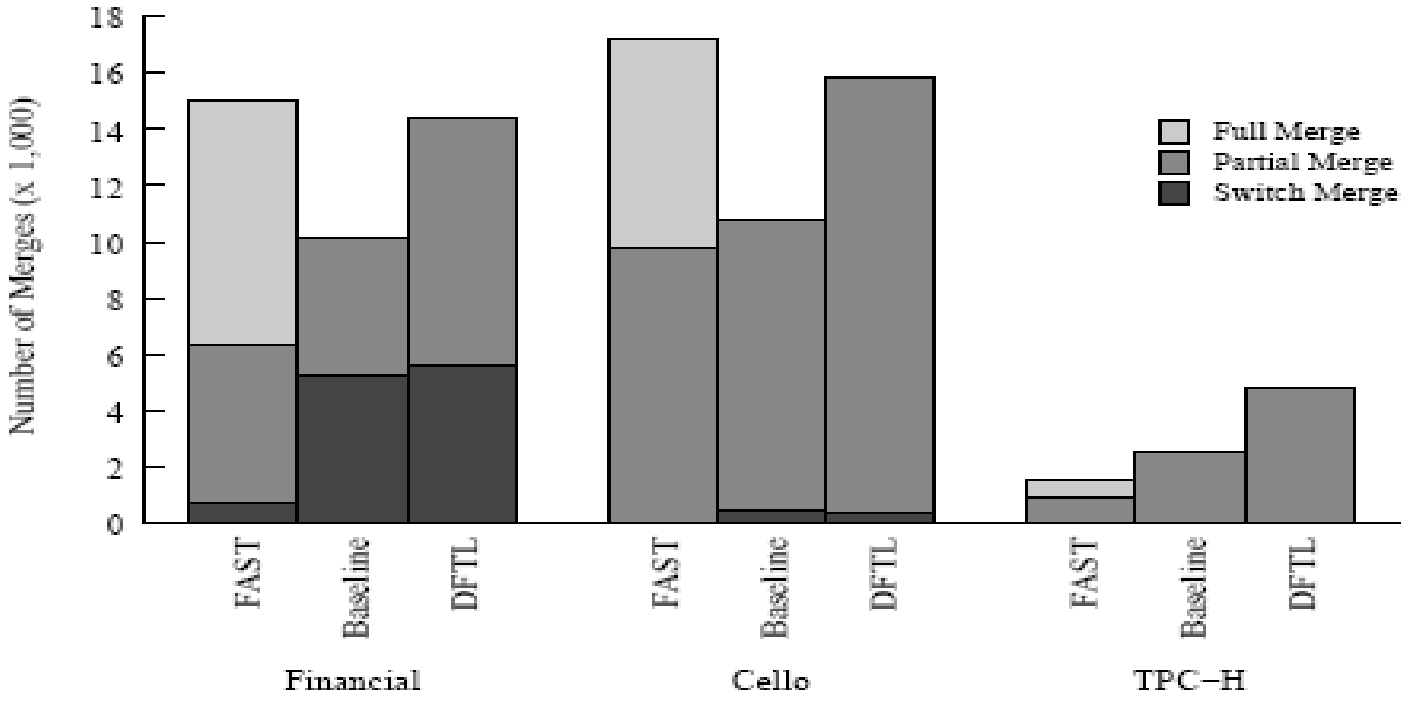
- **Workloads**

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)

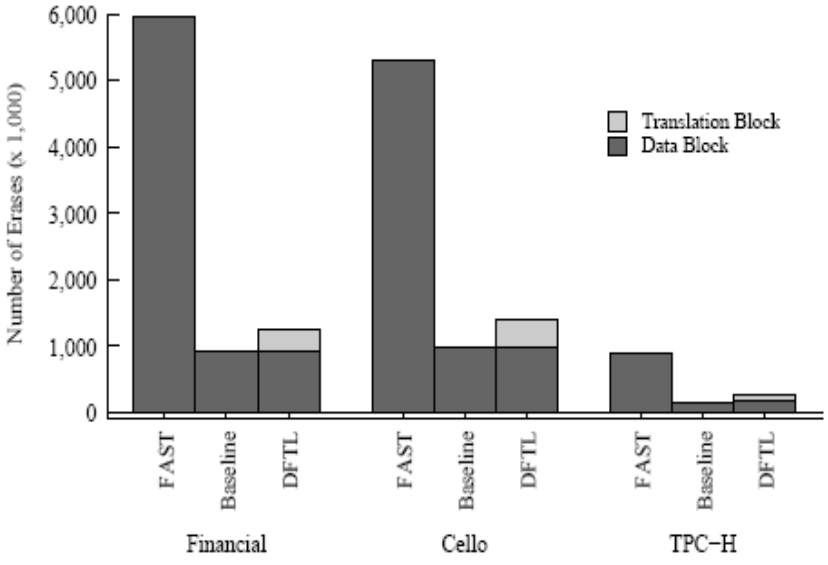
The Number of Block Merges

- **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

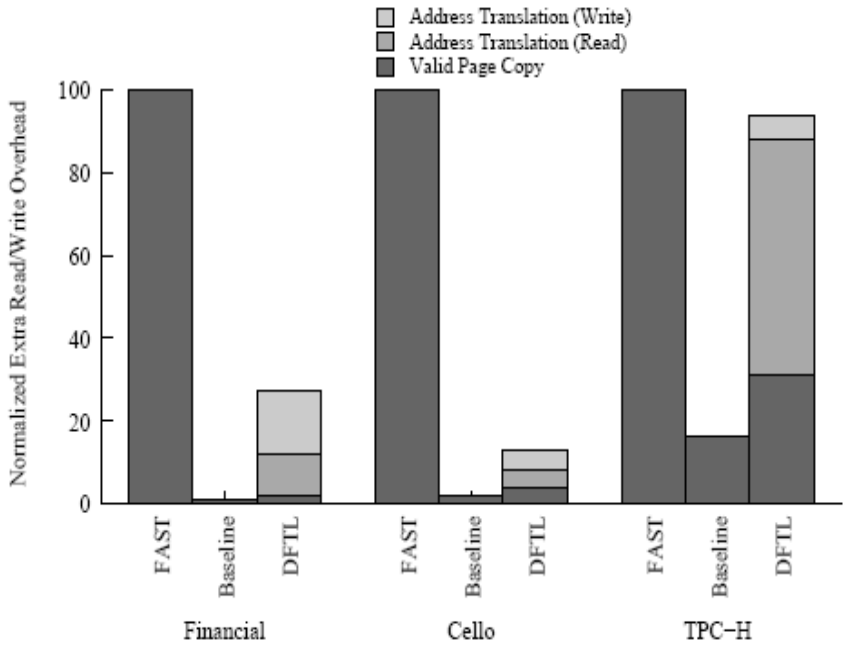


Address Translation Overhead

- **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



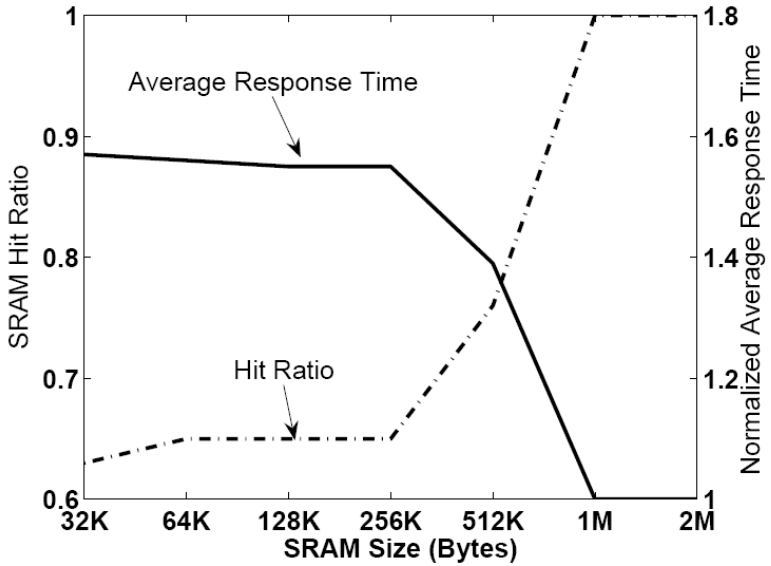
Block Erases



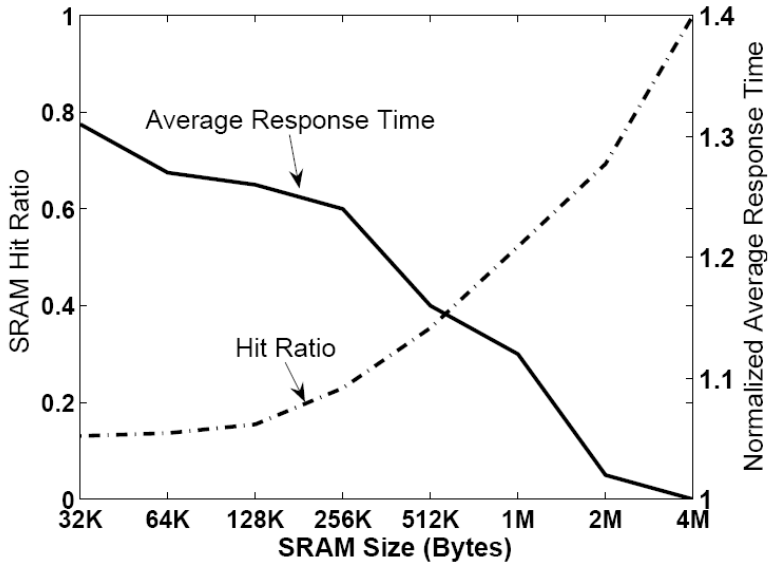
Extra Read/Write Ops.

Impact of SRAM size

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



(a) Financial Trace



(b) TPC-H Benchmark

End of Chapter 5