Data Separation Techniques

Jihong Kim
Dept. of CSE, SNU

Outline

- Introduction to Data Separation
- Data Separation Techniques for NAND flash
  - 2-Queue Based Approach
  - HASH Based Approach
  - Program Context Approach

Classification of Data

- Key factors in classifying data
  - Frequency
    - More frequently accessed data are likely to be accessed again in near future
  - Recency (i.e., closeness to the present)
    - Many access patterns in workloads exhibit high temporal localities
    - Recently accessed data are more likely to be accessed again in near future

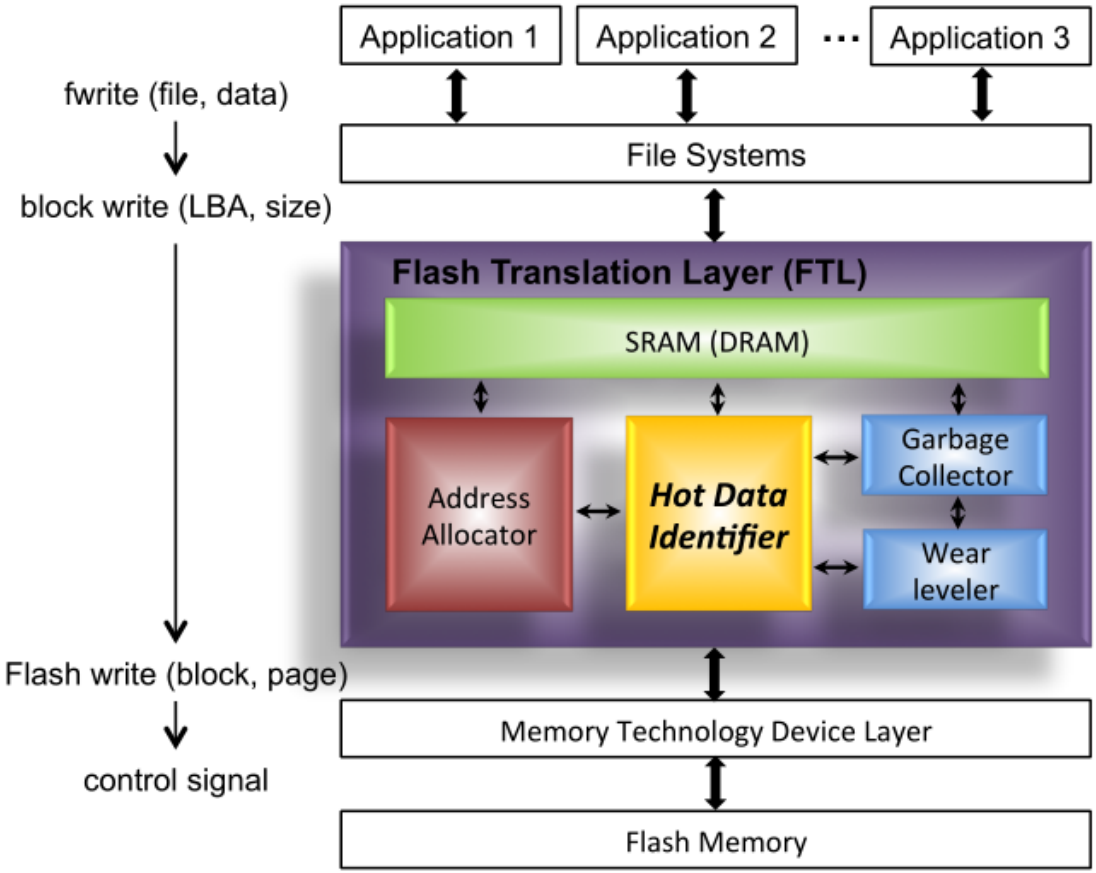
Data Separation in Computer

- Data Cache
  - Caching hot data in the memory space in advance, we can significantly improve system performance
- Sensor Network using FlashDB
  - In FlashDB, the B-tree node can be stored either in read-optimized mode or in write-optimized mode, whose decision can be easily made on the basis of a hot data identification algorithm
- Hard Disk Drive
  - Determine hot blocks and cluster them together so that they can be accessed more efficiently with less physical arm movement
- Hot data identification has a big potential to be exploited by many other applications

Data Separation in NAND

- Garbage collection
  - Reduce garbage collection cost by collecting and storing hot data to the same block
- Wear leveling
  - Improve flash reliability by allocating hot data to the flash blocks with low erase count

Hot Data Identifier in FTL

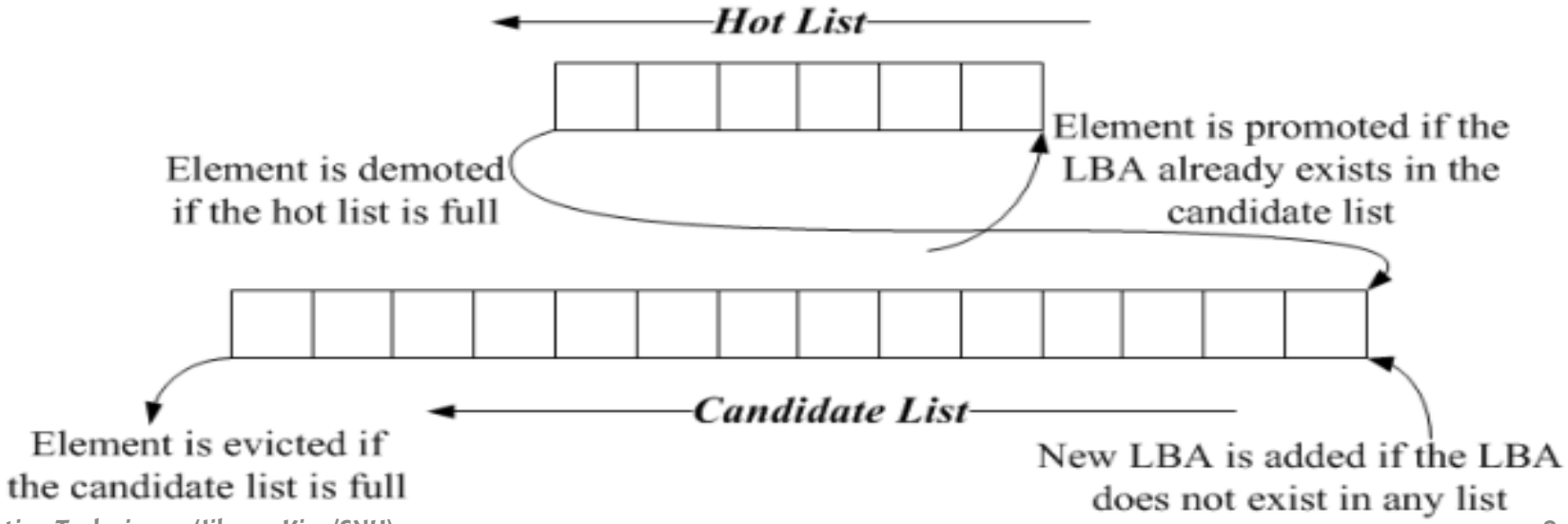


Efficient Hot Data Identification

- Effective capture of recency information as well as frequency information
- Small Memory Consumption
  - Need to store hotness information
  - Limited SRAM size for FTL
- Low Computational Overhead
  - It has to be triggered whenever every write request is issued

2-Level LRU

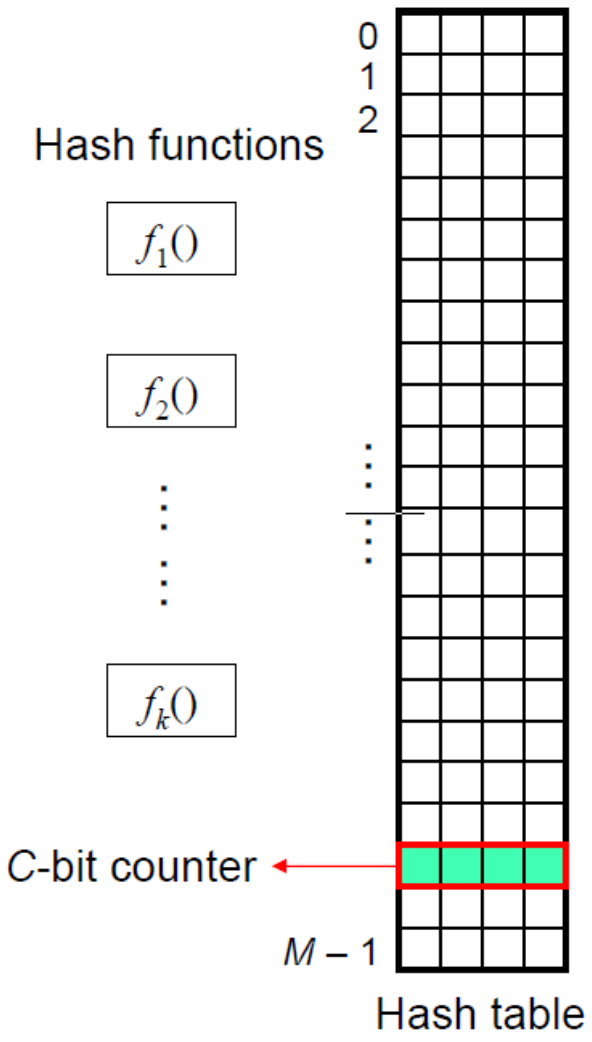
- Maintains hot list and candidate list
  - Operate under LRU
  - Save memory space (i.e. sampling-based approach)
- Performance is sensitive to the sizes of both lists
- High computational overhead



A Multi-Hash-Function Approach

- A Multi-Hash-Function Framework
  - Identify each data request using hash value
- Identify hot data in a constant time
  - Just access hash table without search
- Reduce the required memory space
  - A lot of data requests share a hotness information entry of hash tables

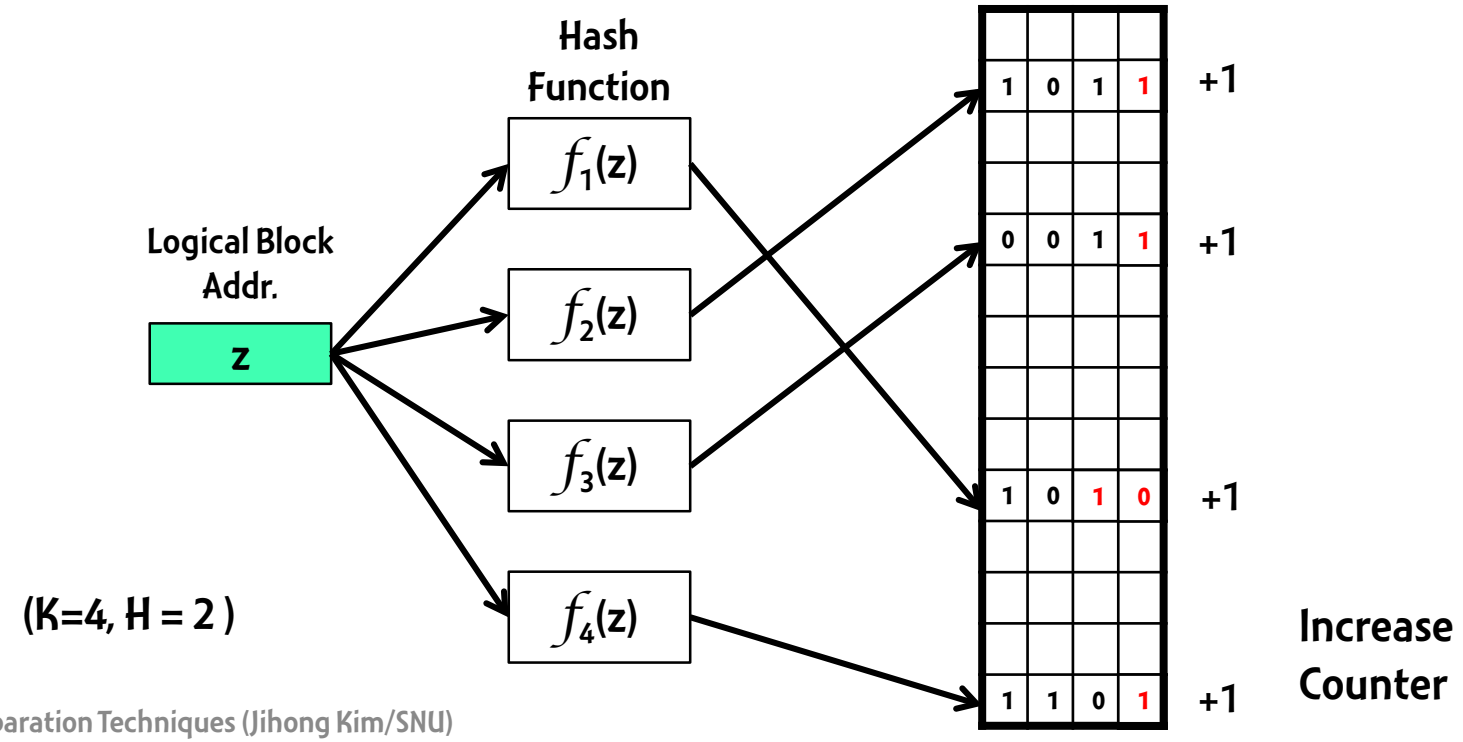
A Multi-Hash-Function Framework



- *Component*
  - *K independent hash functions*
  - *M-entry hash table*
  - *C-bit counters*
- *Operation*
  - **Status Update**
    - Updating of the status of an LBA
    - Storing frequency information
  - **Hotness Checkup**
    - The verification of whether an LBA is for hot data
  - **Decay**
    - Decaying of all counters
    - Storing recency information

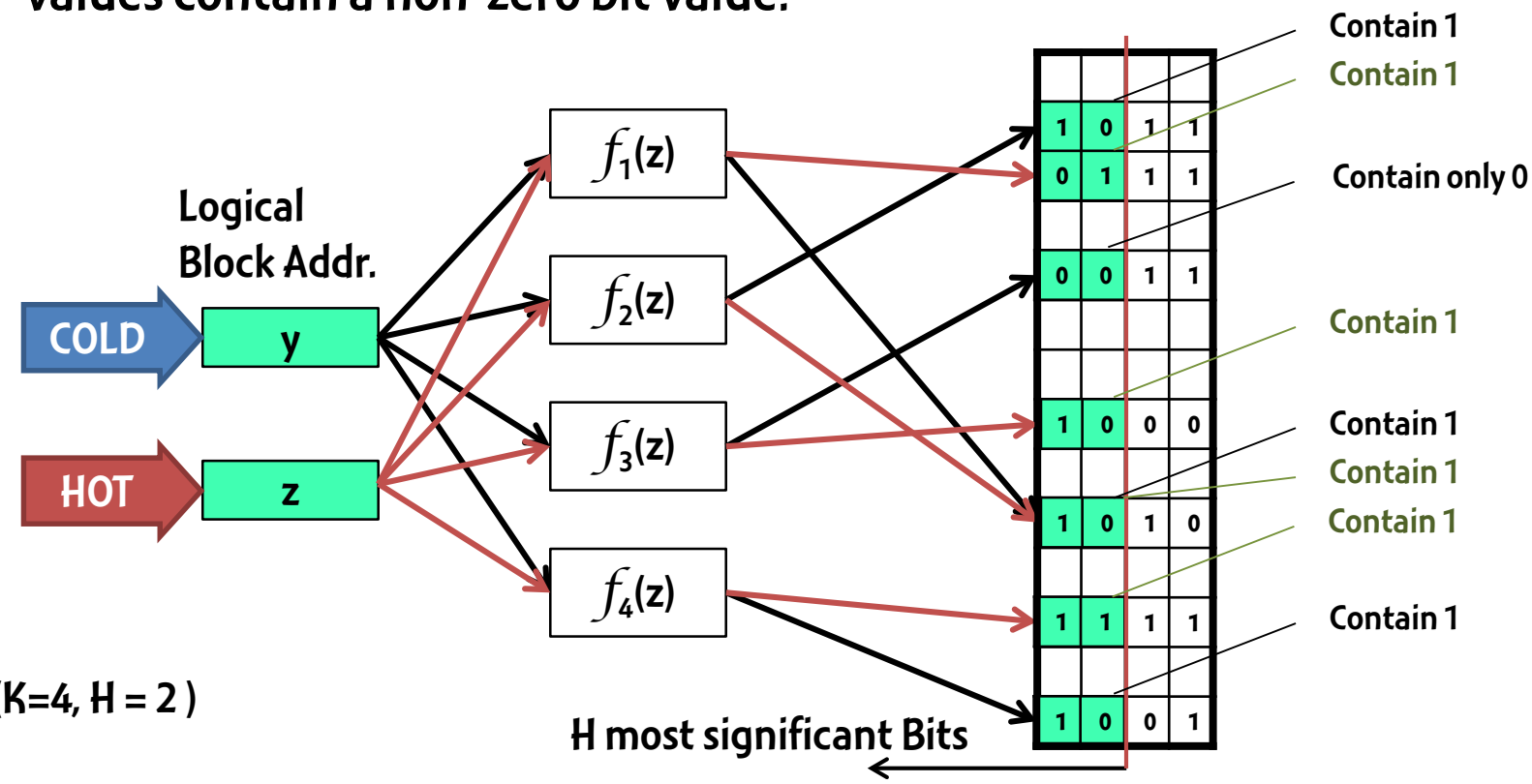
Status Update (Counter Update)

- A write is issued to the FTL
- The corresponding LBA *y is hashed* simultaneously by *K given hash* functions.
- Each counter corresponding to the *K* hashed values (in the hash table) is incremented by one to reflect the fact that the LBA is written again



Hotness Checkup

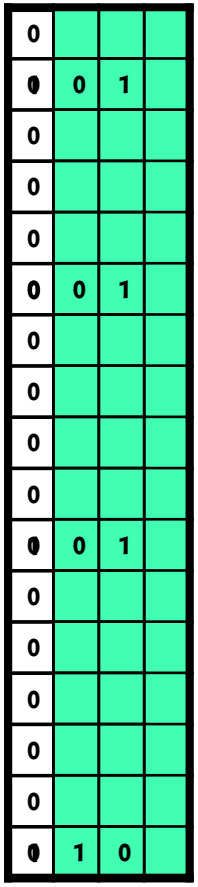
- An LBA is to be verified as a location for hot data.
- Check if the H most significant bits of every counter of the K hashed values contain a non-zero bit value.



Decay

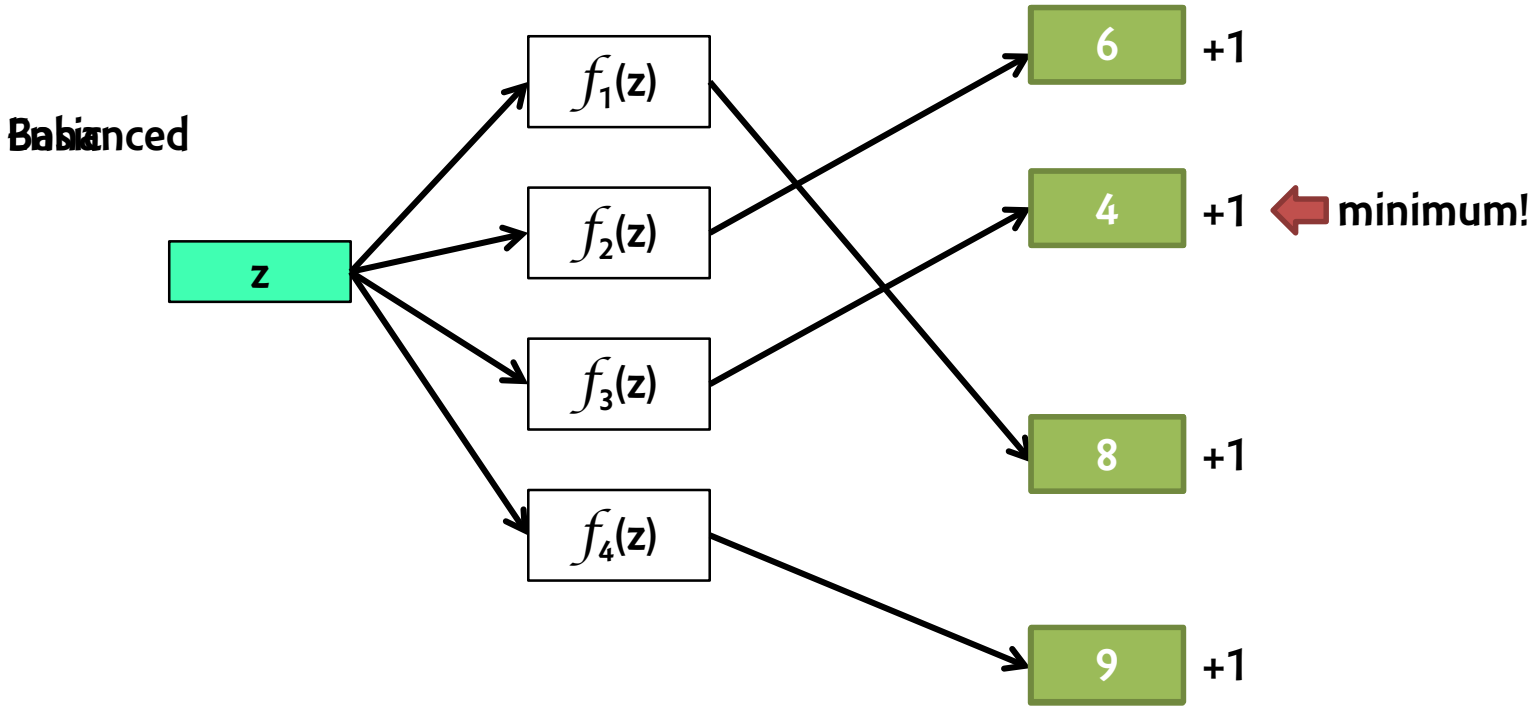
- For every given number of sectors have been written, called the "decay period" of the write numbers, the values of all counters are divided by 2 in terms of a right shifting of their bits.

(K=4, H = 2)



An Implementation Strategy

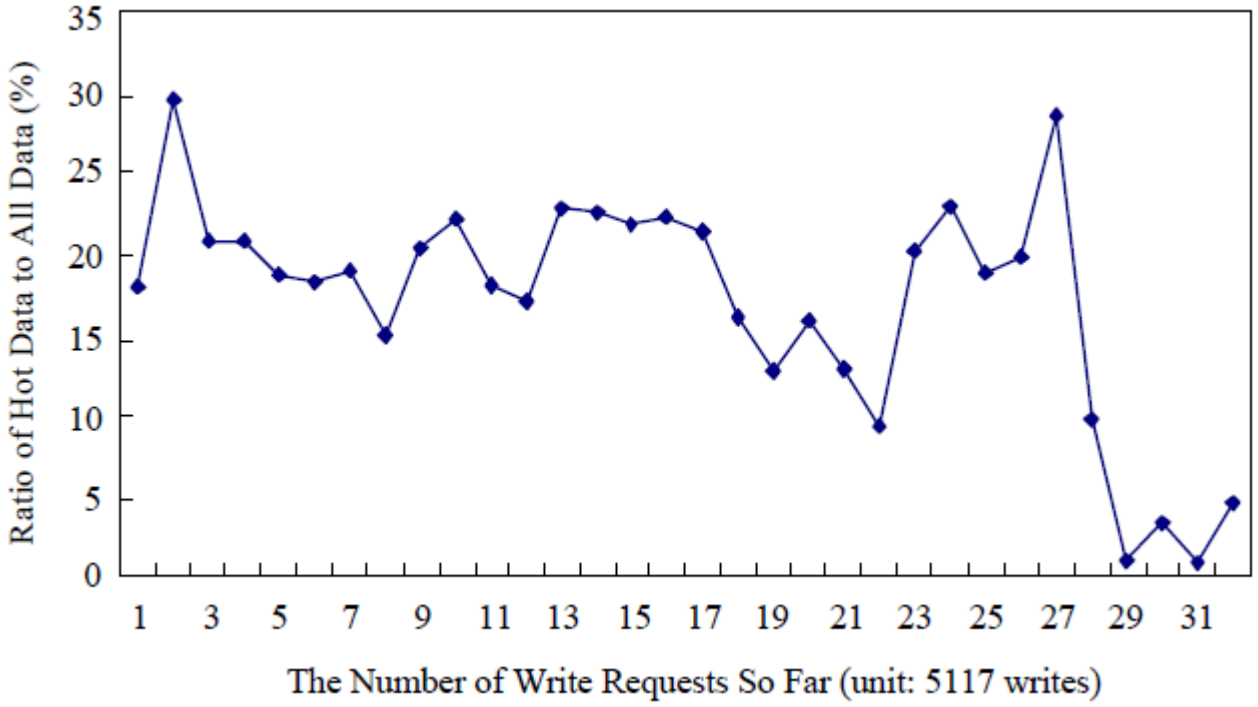
- In order to reduce the chance of false identification, only counters of the *K* hashed values that have the minimum value are increased



Performance Evaluation

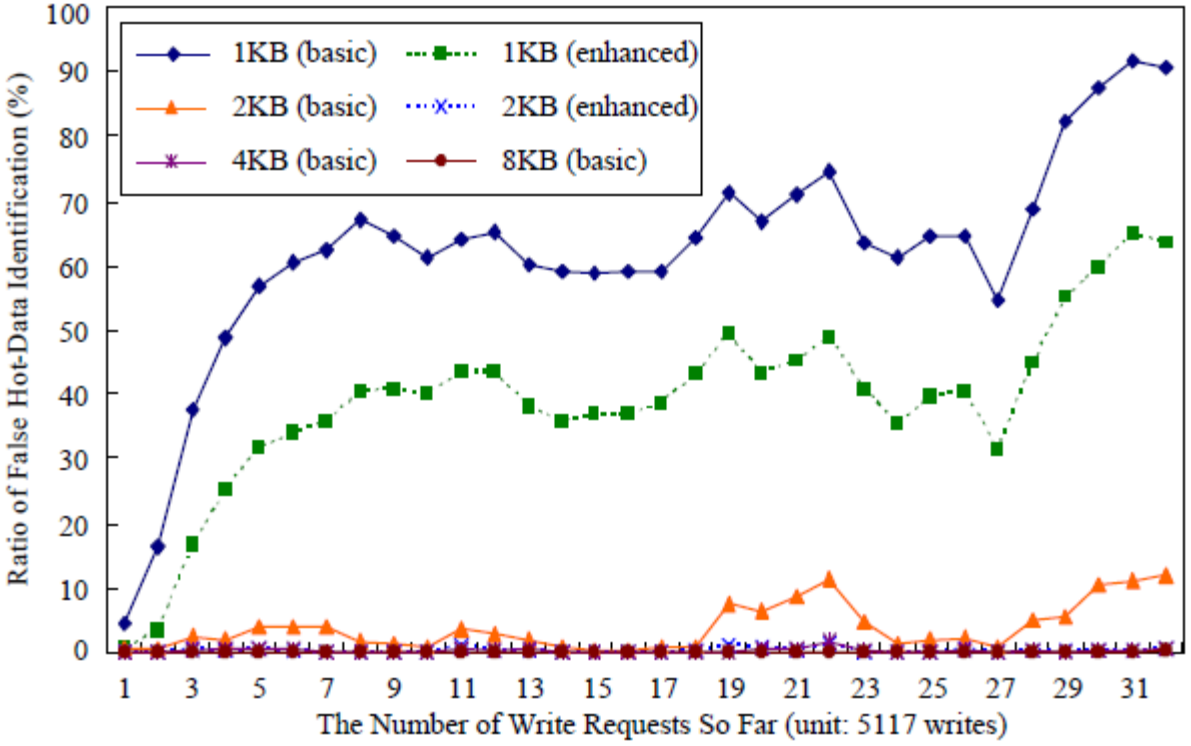
- Metrics
  - Impacts of Hash-Table Sizes
  - Runtime Overheads
- Experiment Setup
  - Number of hash functions: 2
  - Counter size: 4 bits
  - Flash memory size: 512 MB
  - Hot-data threshold: 4

Impacts of Hash-Table Sizes (1)



- The locality of data access (decay period: 5117 writes)

Impacts of Hash-Table Sizes (2)



- Ratio of false hot data identification for various hash table sizes

Runtime Overheads

	Multi-Hash-Function Framework (2KB)		Two-Level LRU List* (512/1024)	
	Average	Deviation Standard	Average	Deviation Standard
Checkup	2431.358	97.98981	4126.353	2328.367
Status Update	1537.848	45.09809	12301.75	11453.72
Decay	3565	90.7671	N/A	N/A

Unit: CPU cycles

Problem of Hash-Based Approach

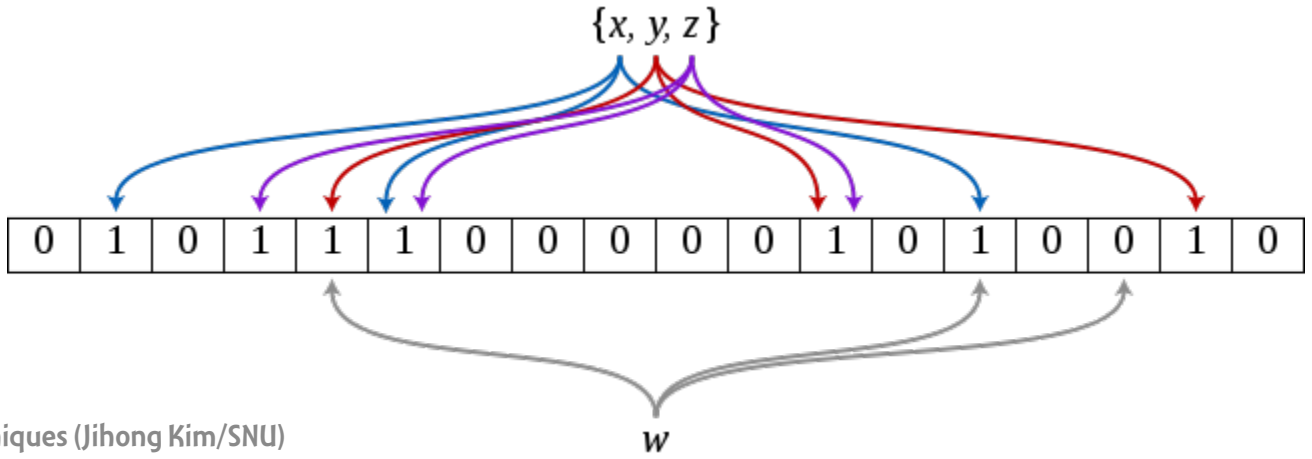
- Accurately captures frequency information
  - By maintaining counters
- Cannot appropriately capture recency information due to its exponential batch decay process (i.e., to decreases all counter values by a half at a time)

Multiple BF-based scheme

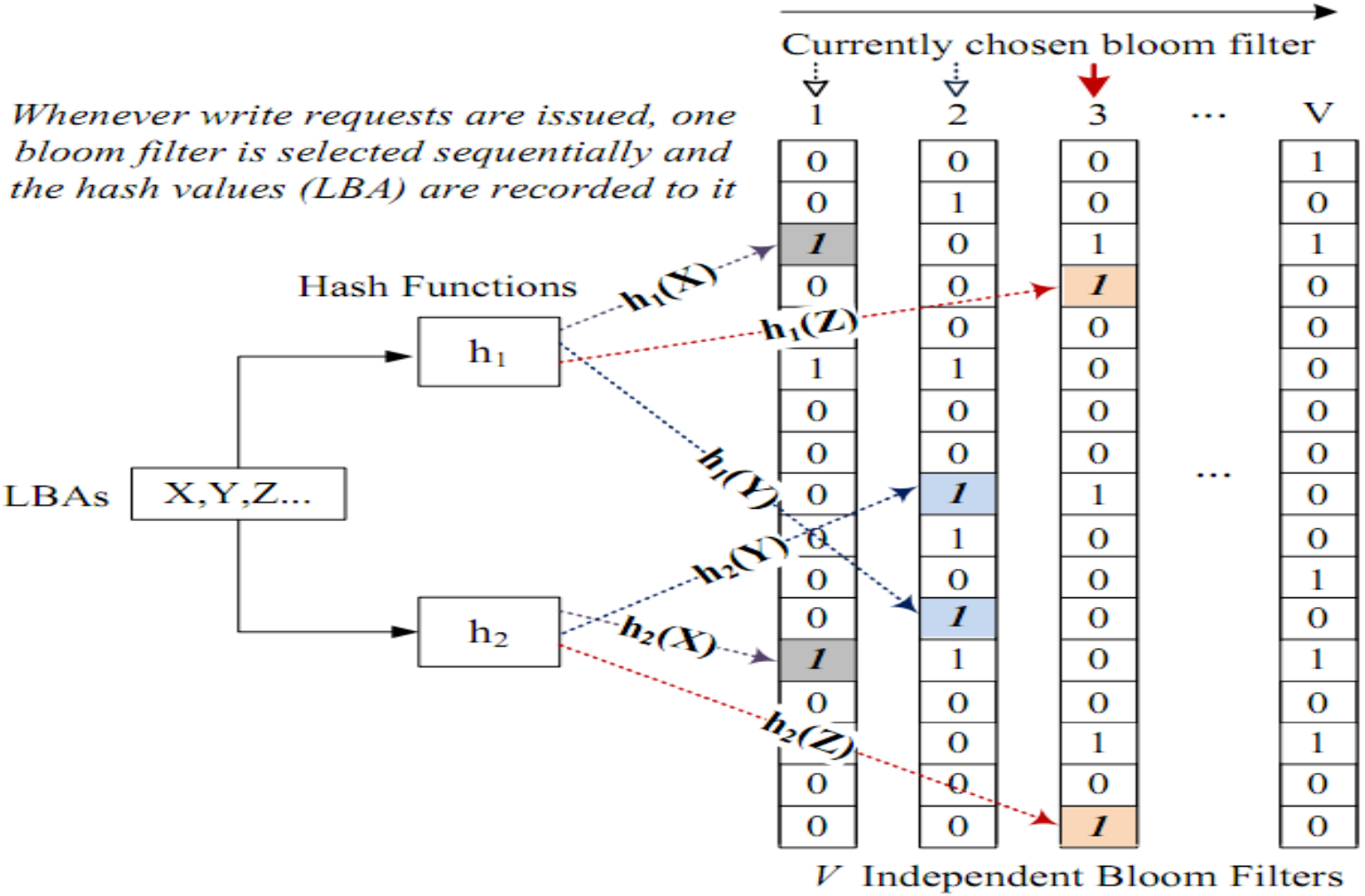
- Overview
  - Multiple bloom filters
    - To capture finer-grained recency
    - To reduce memory space and overheads
  - Multiple hash functions
    - To reduce false identification
- Frequency
  - Does not maintain access counters
- Recency
  - Different recency coverage

Bloom Filter (from Wikipedia)

- A space-efficient probabilistic data structure proposed by Bloom in 1970
- Used to test if $\alpha \in S$
- Allows False Positives, but no False Negatives
  - "possibly in S" or "definitely not in S"



Basic Operations



Capturing Frequency

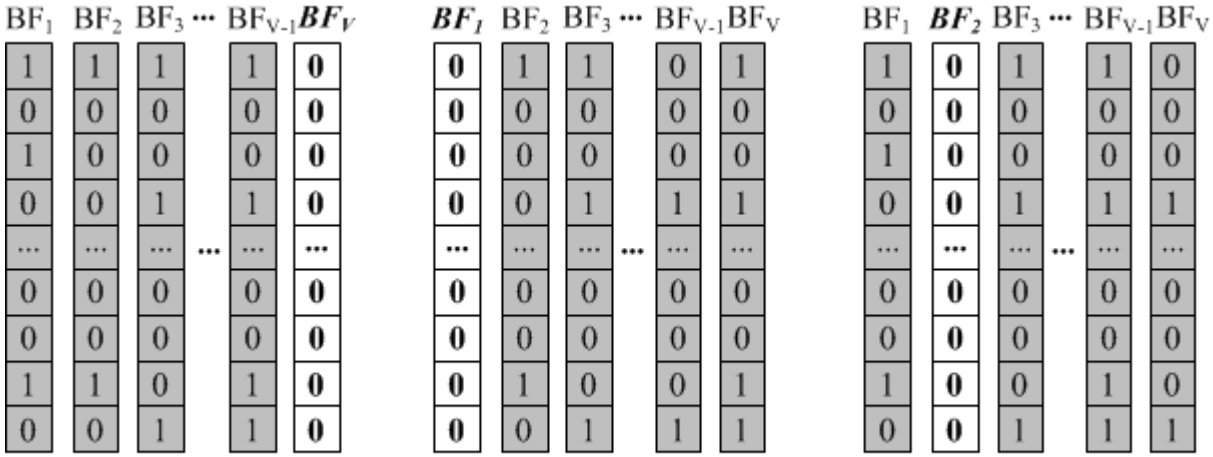
- No access counters
  - Needs a different mechanism
- For frequency capturing
  - Chooses one of BFs in a round-robin manner
  - If the chosen BF has already recorded the LBA
    - Records to another BF available.
  - Shortcut decision
    - If all BFs store the LBA information
      - Simply define the data as hot

➔ *The Number of BFs can provide frequency information*

Capturing Recency

- After a decay period (T)
  - Choose one of *V*-BFs in a round-robin manner
  - Erase all information (i.e., reset all bits to 0)

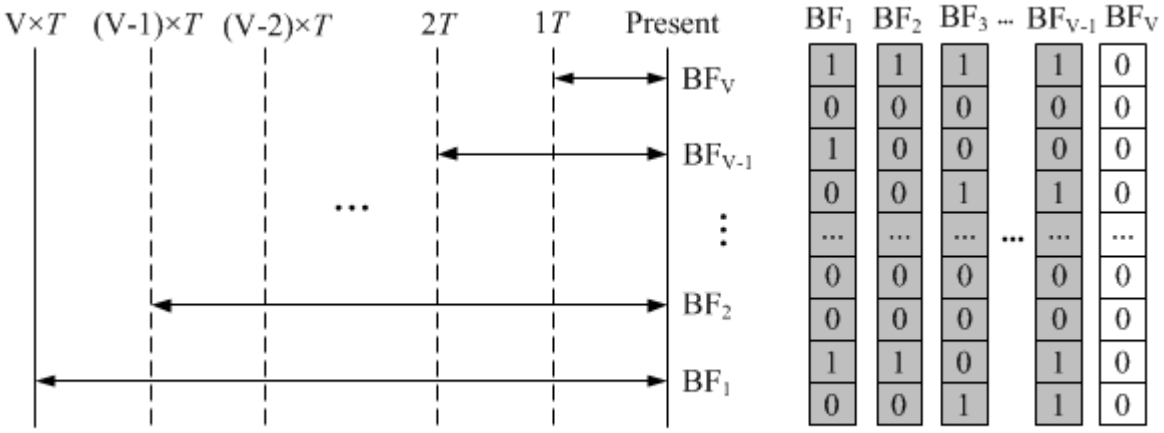
➔ ***Each BF retains a different recency coverage.***



(a) First *T* Interval (b) Second *T* Interval (c) Third *T* Interval

Recency Coverage

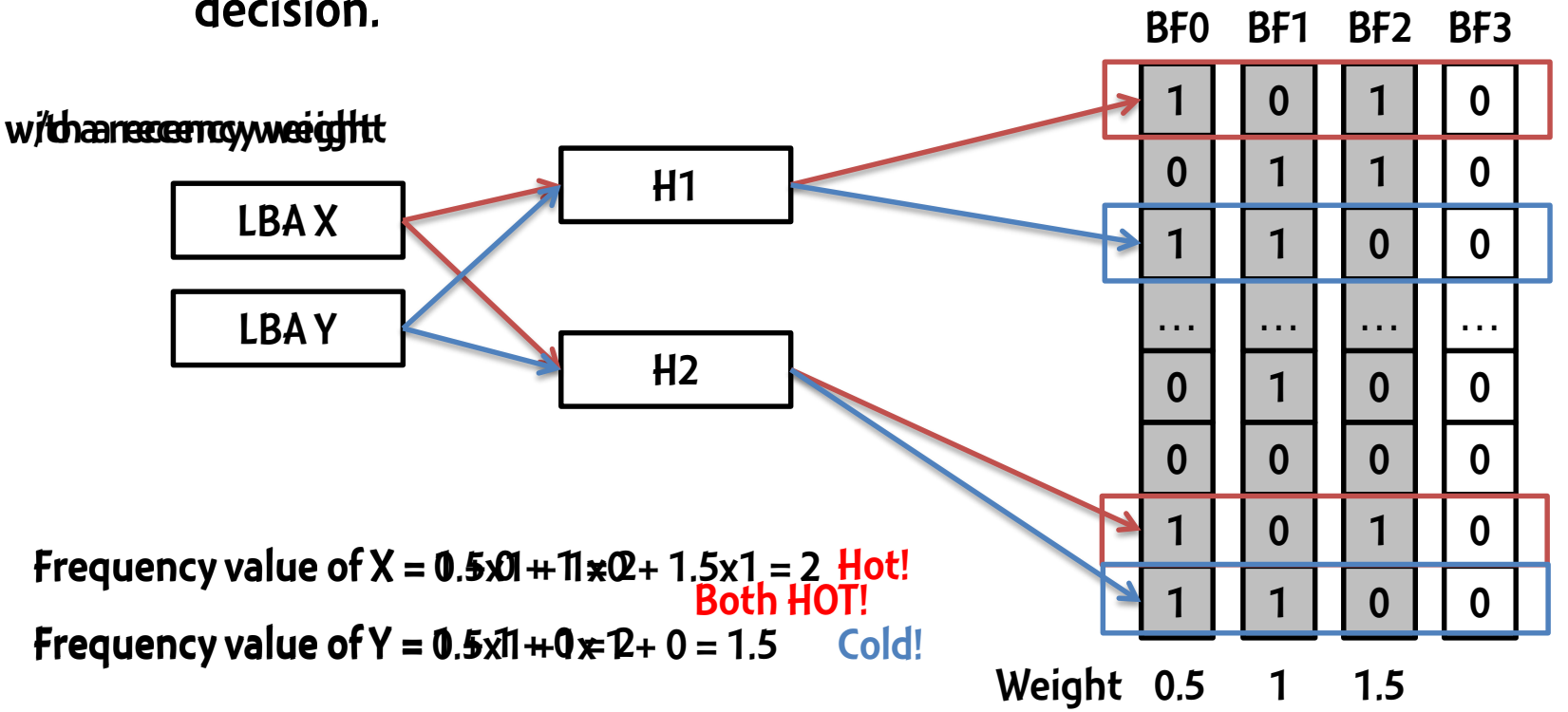
- For finer-grained recency
  - Each BF covers a different recency coverage
    - The reset BF (BF_v): Shortest (latest) coverage
    - The next BF (BF_1): Longest (oldest) coverage
  - Each BF has a different recency value



(a) Recency Coverage (b) Bloom Filter Status

Example: Hot/Cold Checkup Based on Recency Weight

- Assign a different recency weight to each BF
  - Recency value is combined with frequency value for hot data decision.



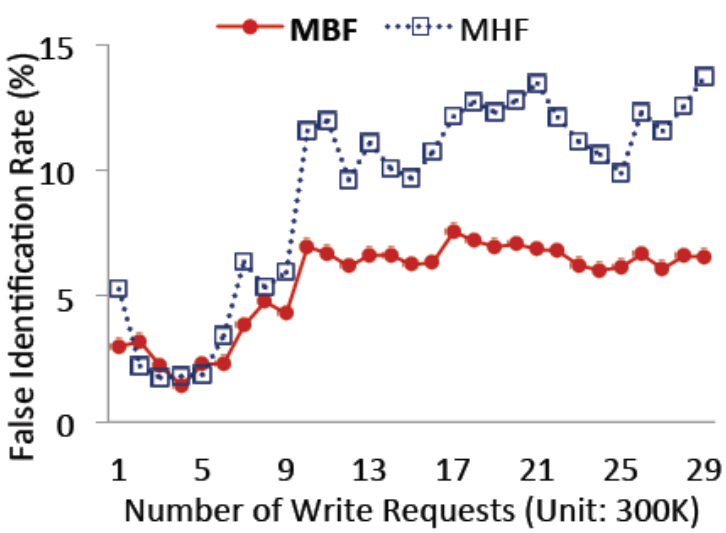
Performance Evaluation

- Evaluation setup
  - Four schemes
    - Multiple bloom filter scheme (refer to as MBF)
    - Multiple hash function scheme (refer to as MHF)
  - Four realistic workloads
    - Financial1, MSR (*prxy volume 0*), Distilled, and Real SSD

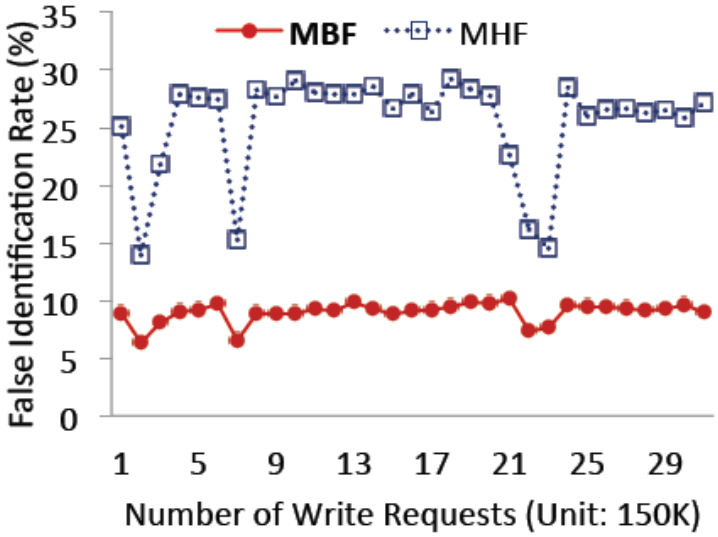
Performance Evaluation

- Performance metrics
  - False identification rate
    - Try to compare each identification result of each scheme whenever a request is issued
  - Memory consumption
  - Runtime overhead
    - Measure CPU clock cycles per operation

False Identification Rate
(MBF vs. MHF)

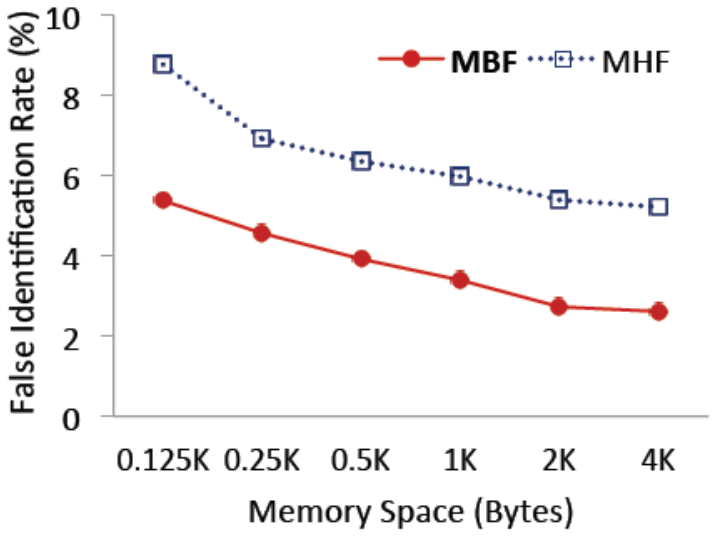


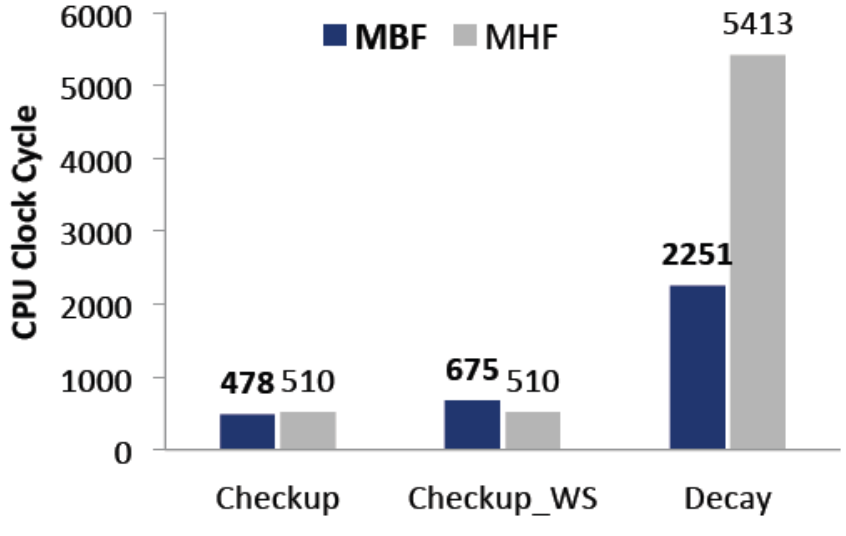
(a) Financial1



(b) MSR

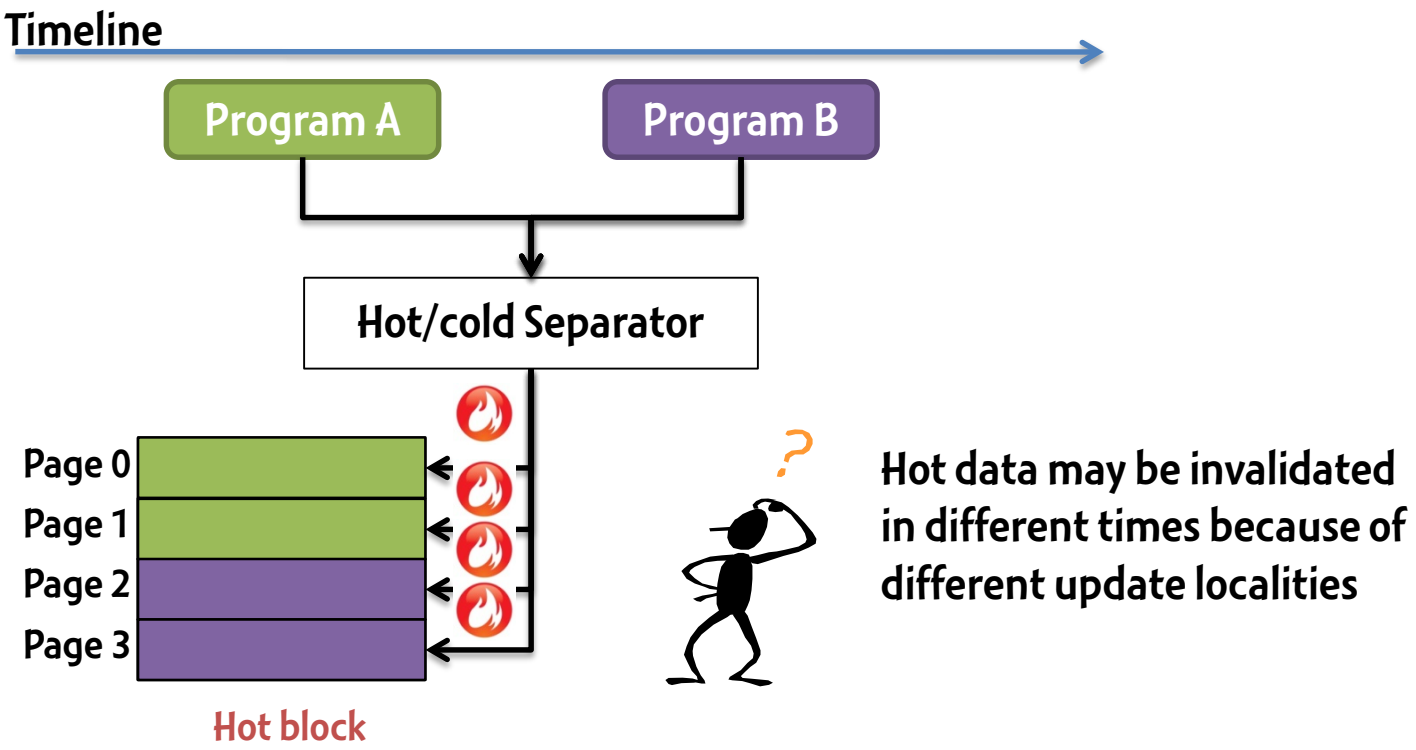
Memory Impact and Computational Overheads





Problems of Hot/cold Separator

- Problem 1: Wide variations on future update times

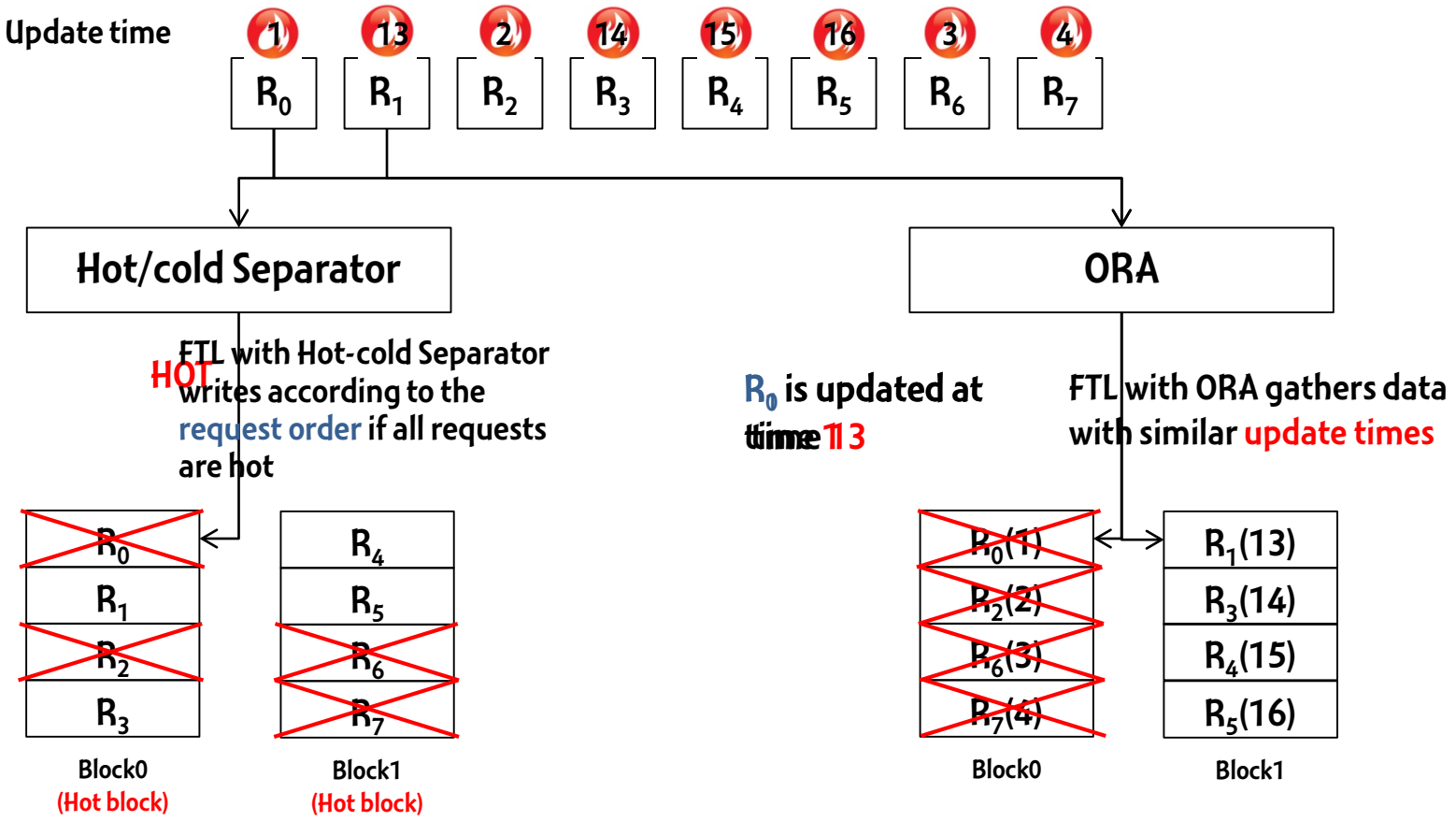


- Problem 2: If there is no clear temporal locality, hot/cold separator does not work

ORA: Oracle Predictor on Future Update Time

- Perfect knowledge on future update times of data
- Can sort data based on the future update times of data
- An FTL with ORA can gather data with similar update times into the same block
- Can be used as lower bound of GC

Motivation Example



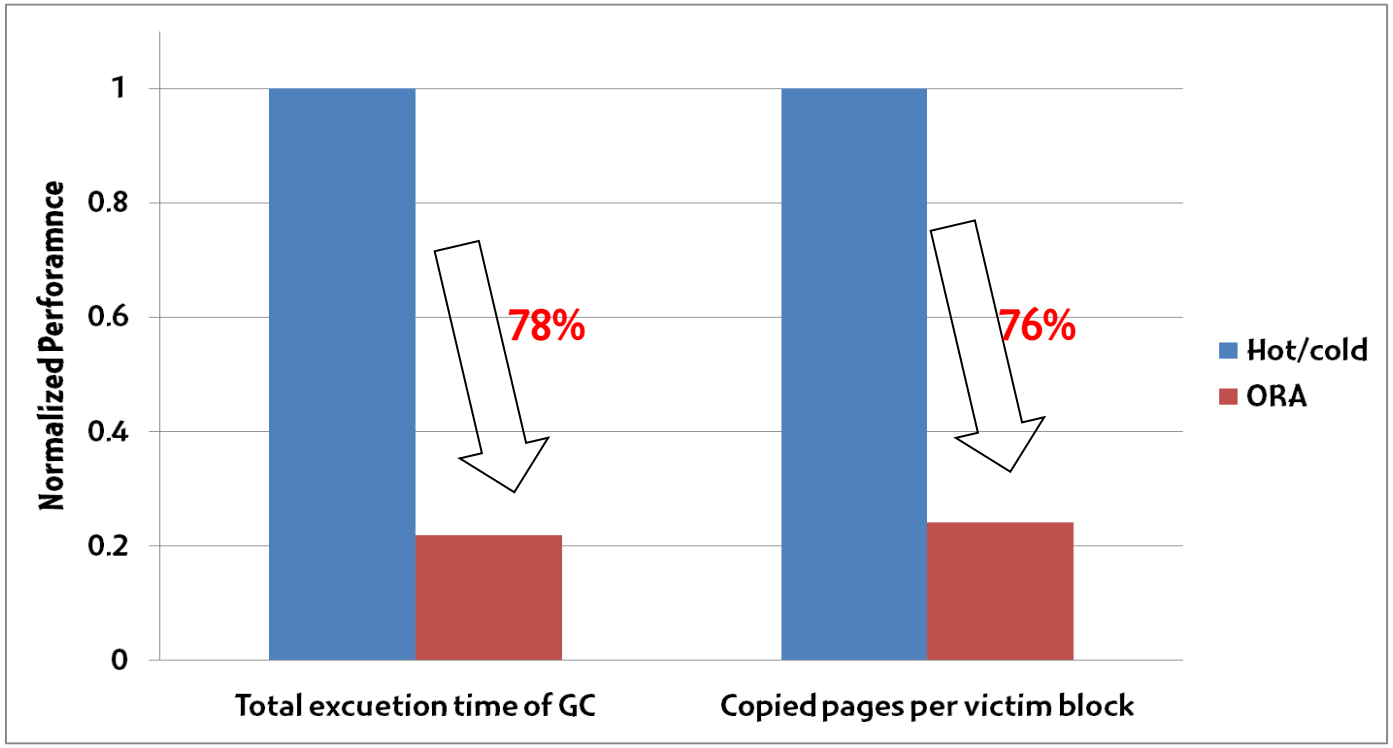
If a GC process was triggered at time 10,

4 copies + 2 erasures

1 erasure

Hot/cold Separator vs. ORA

- ORA can reduce GC overhead significantly



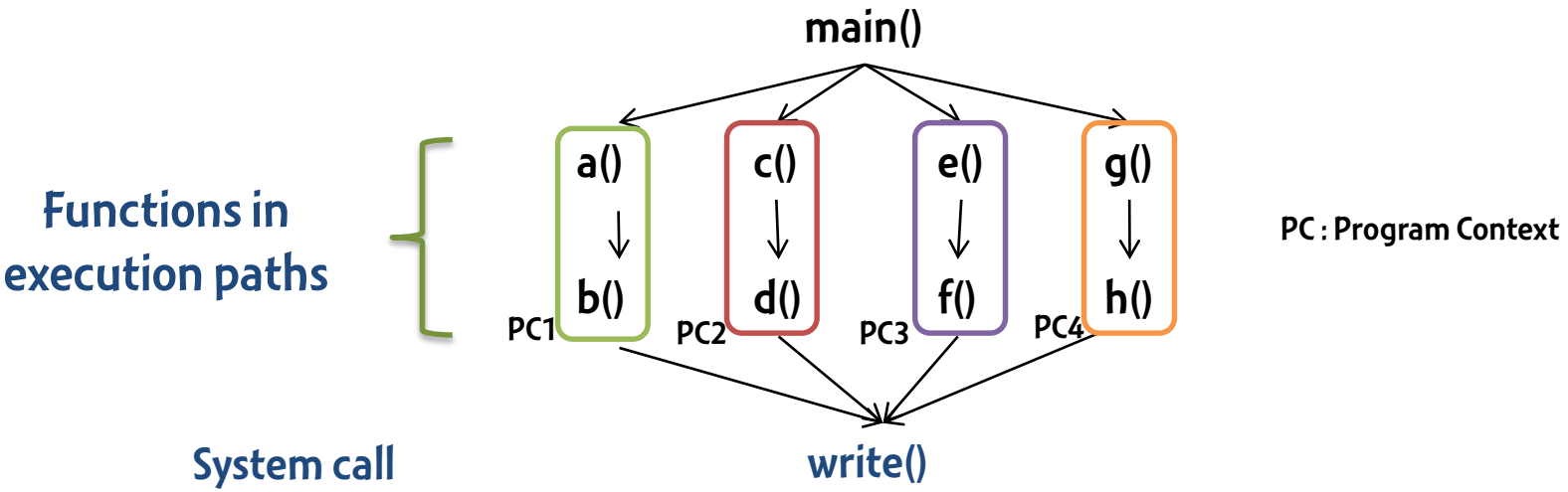
Update time is a more important factor in data separation technique than frequency of updates

Basic Idea

- Program Context-Aware Data Separation Technique
  - Predicts update times of data based on program behavior
  - A program behaves similarly when the same program context is executed
  - Indentifies what program contexts repeatedly generate data with similar update times

Overview of Program Context

- A program context represents an execution path which generates write requests



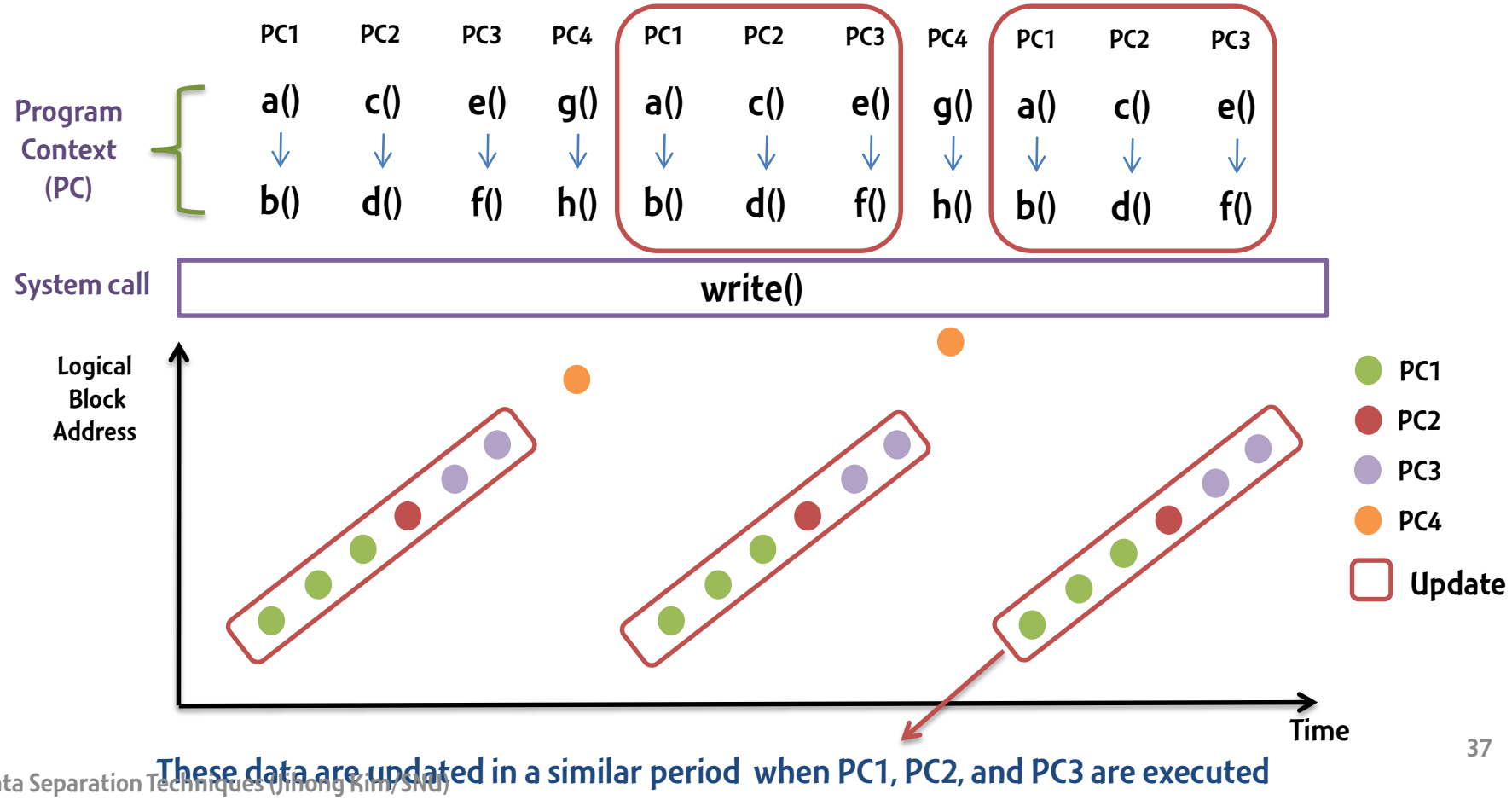
- Identification
  - Each program context is identified by summing program counter values of each execution path of function calls

Reference

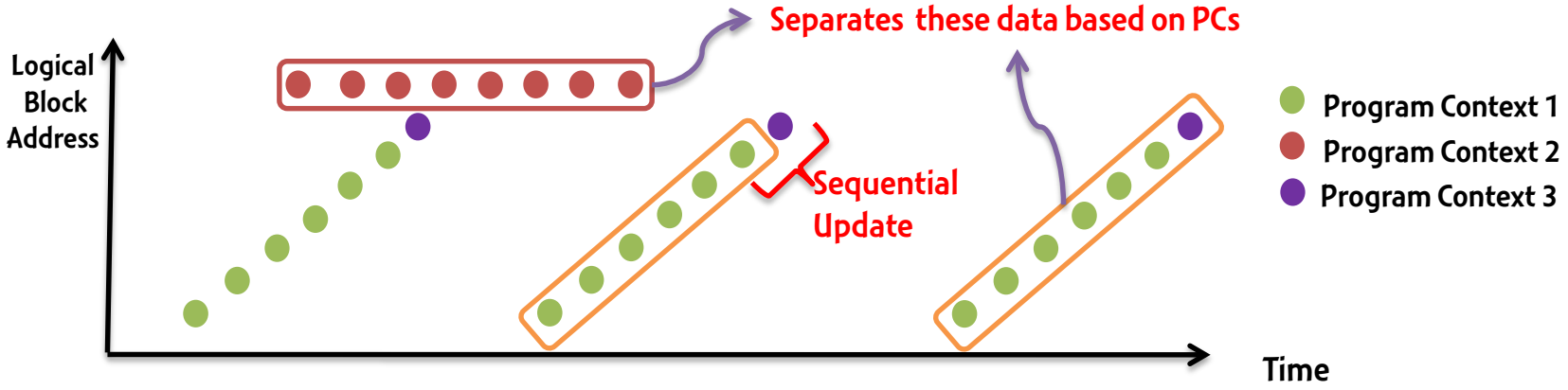
Chris Gniady, and Ali R. Butt, and Y. Charlie Hu, "***Program Counter Based Pattern Classification in Buffer Caching***," **OSDI**, 2004

Program Context-Based Update Time Prediction

- Indirectly predict future update times of data by exploiting program contexts



Separating Data using Program Contexts



Simultaneously updated group 1

Data generated by Program Context 2

Simultaneously updated group 2

Data generated by Program Context 1 and Program Context 3

FTL with this data separator stores data based on simultaneously updated group

Experimental Environments (1)

- Used a trace-driven NAND flash memory simulator
  - Parameters

Flash Translation Layer	Mapping Scheme	Page-level mapping
	GC Triggering	5%
Flash memory	Read Time (1 page)	25usec
	Write Time (1 page)	200usec
	Erase Time (1 block)	1200usec

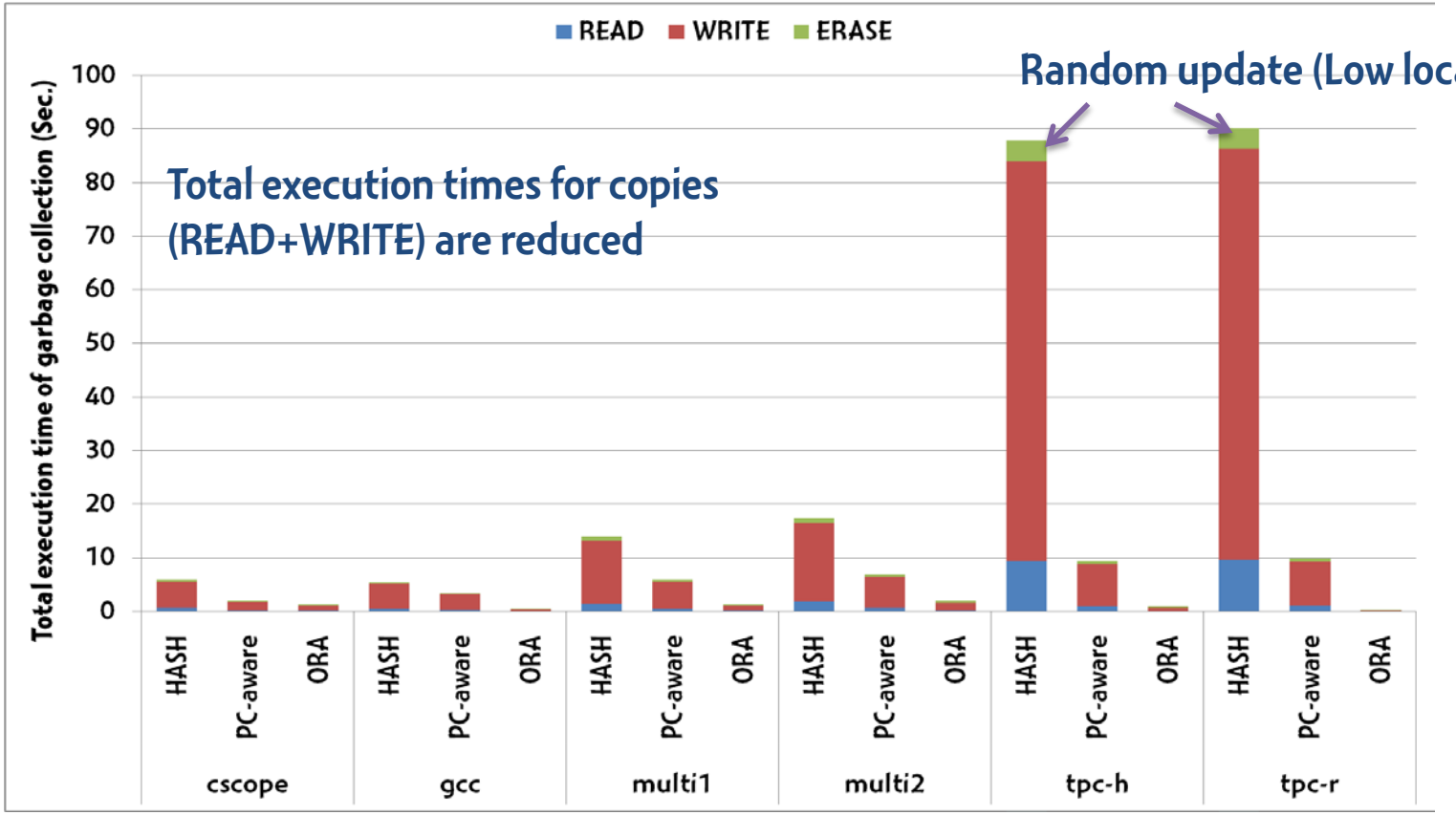
- Techniques for comparison
  - HASH: Hash-based hot/cold separation technique
  - ORA: Oracle predictor on future update times of data

Experimental Environments (2)

- **Benchmarks characteristics**

Benchmarks	Scenario	The number of writes (unit: page)	The number of updates (unit: page)
cscope	Linux source code examination	17575	15398
gcc	Building Linux Kernel	10394	3840
viewperf	Performance measurement	7003	119
tpc-h	Accesses to database	23522	20910
tpc-r	Accesses to database	21897	18803
multi1	cscope + gcc	28400	19428
multi2	cscope + gcc + viewperf	35719	20106

Result: Total Execution Time of GC



Reduces the total execution time of garbage collection on average 58% over HASH.

Reference

- J. Hsieh et al, "Efficient on-line identification of hot data for flash-memory management," SAC 2005.
- D. Park et al., "Hot Data Identification for Flash-based Storage Systems Using Multiple Bloom Filters", MSST 2011
- K. Ha et al., "A Program Context-Aware Data Separation Technique for Reducing Garbage Collection Overhead in NAND Flash Memory," SNAPI 2011