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

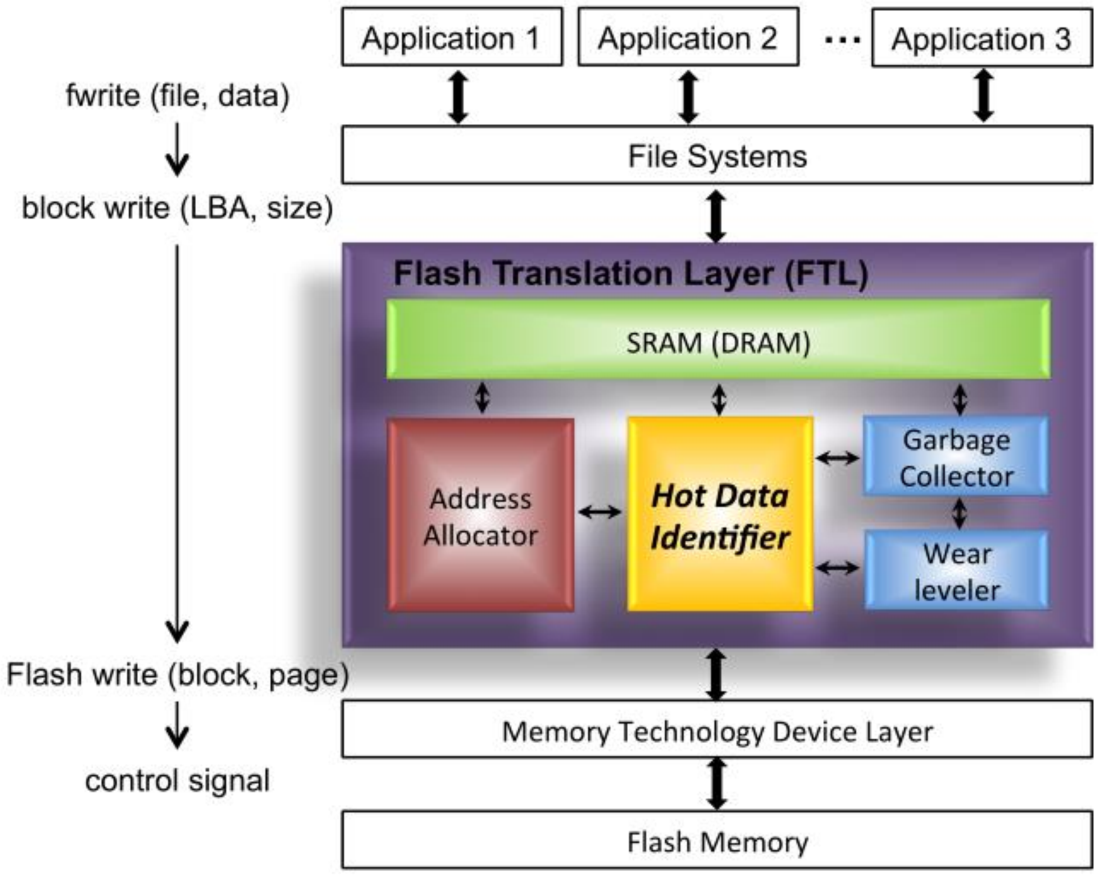
Data Separation in Computer

- 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

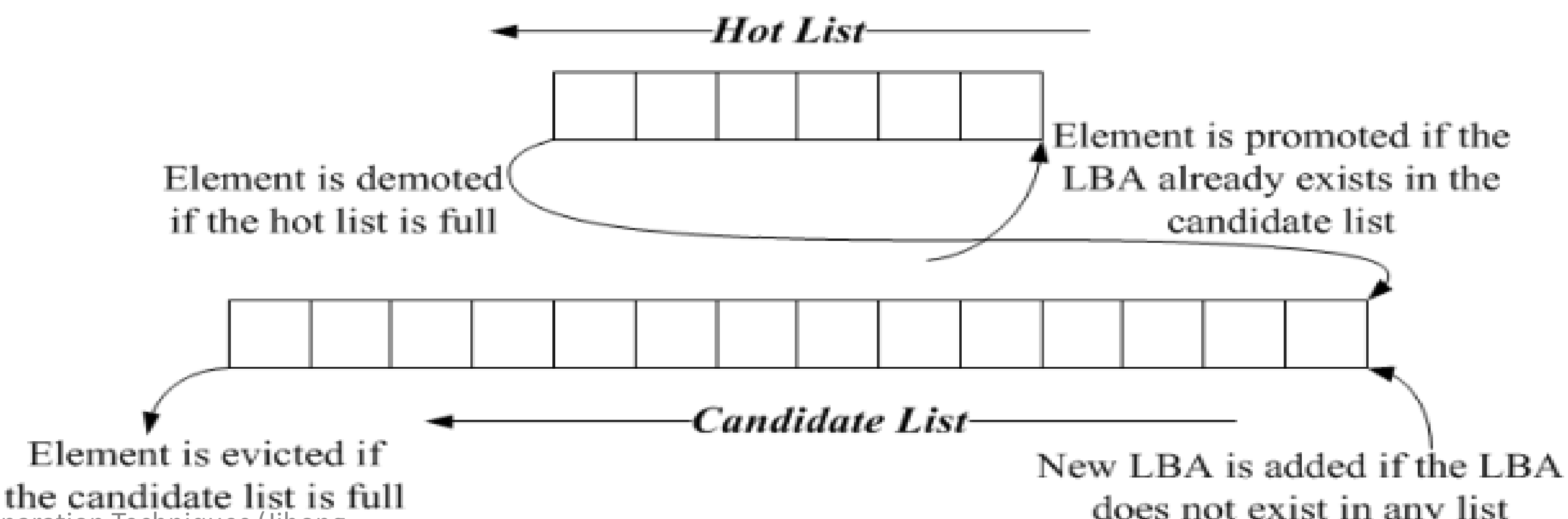


Efficient Hot Data Identification

- 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

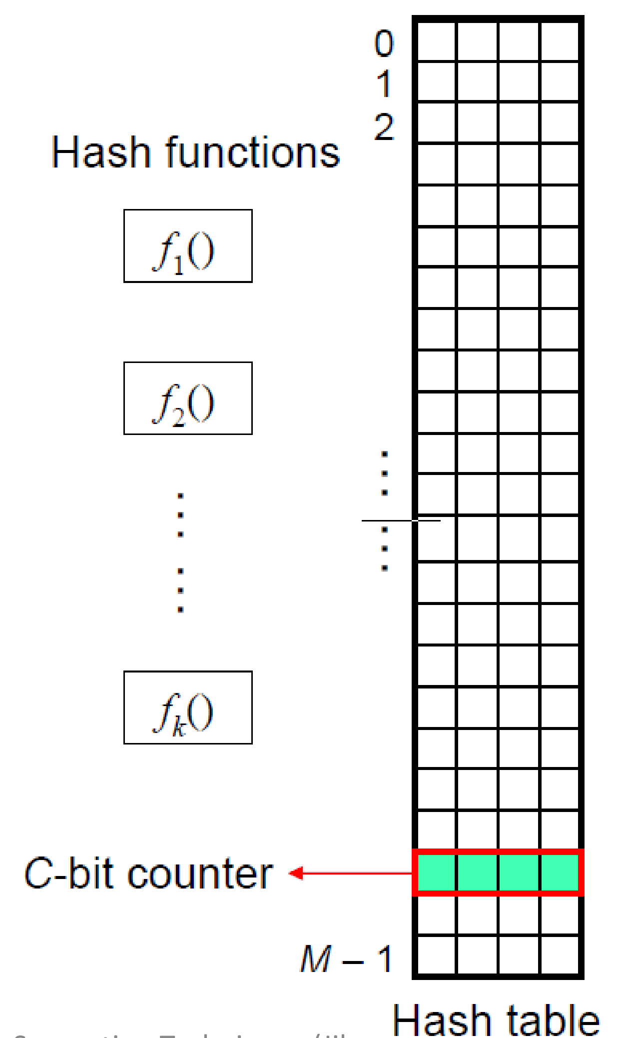
- Maintains hot list and candidate list
  - Operate under LRU
  - Save memory space (i.e. sampling-based approach)
- Performance is sensitive to the sizes of both lists
- High computational overhead



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

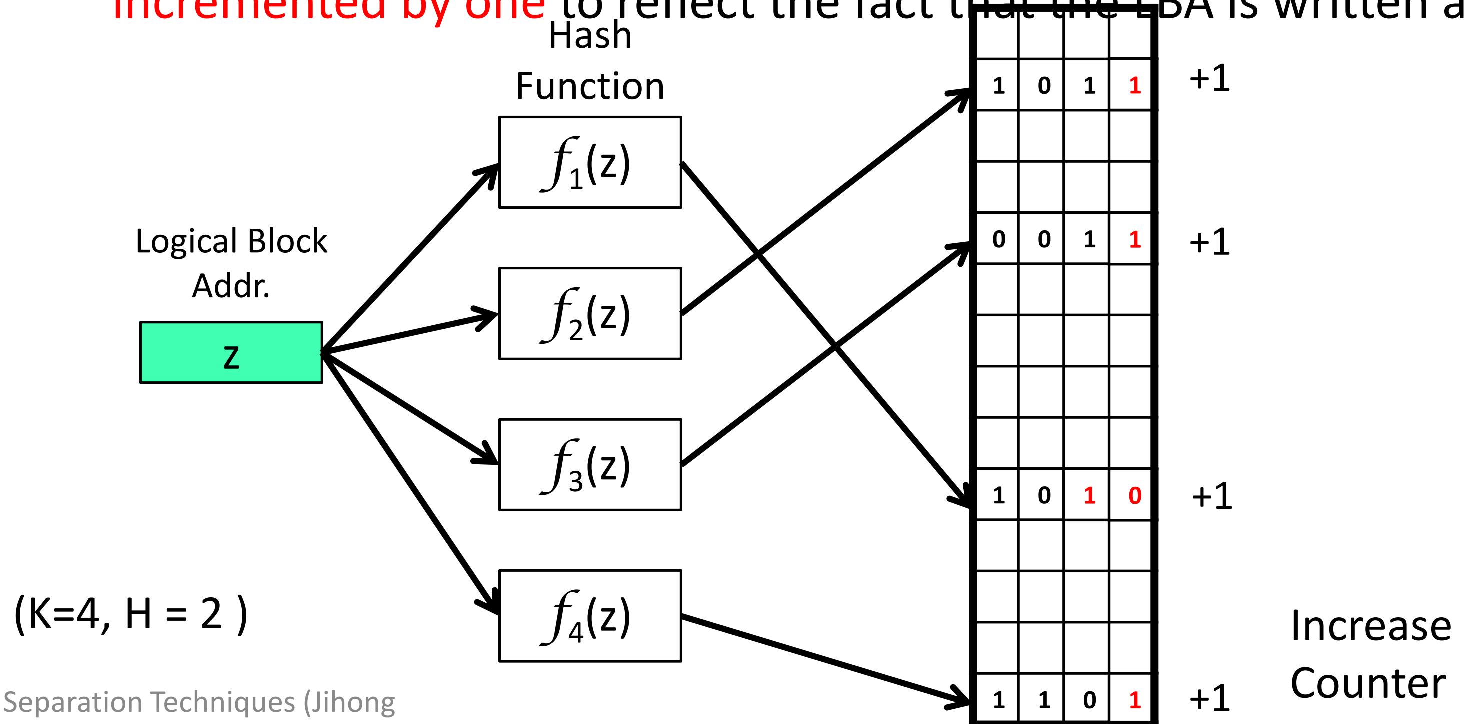
A Multi-Hash-Function Framework



- *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

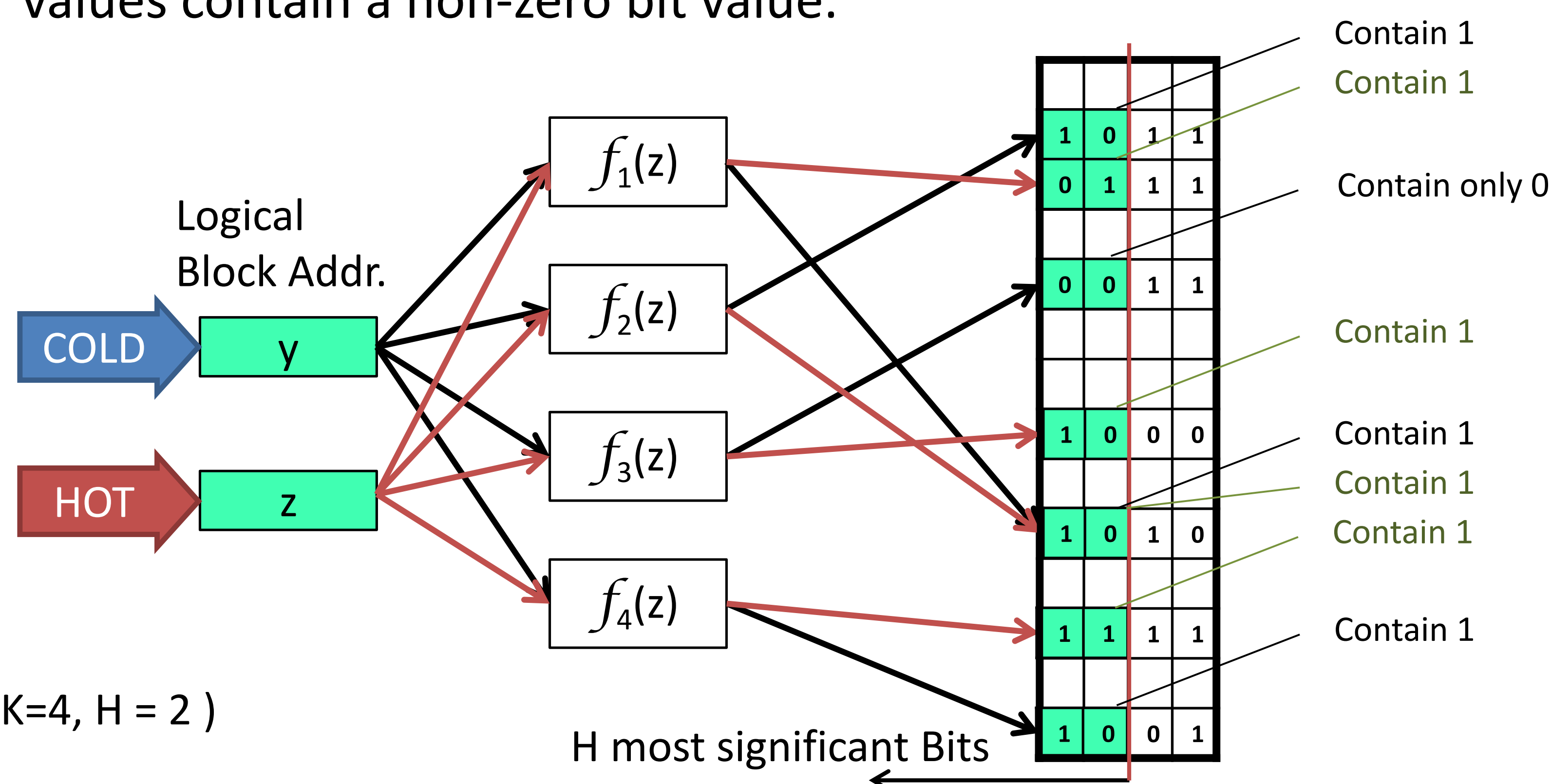
Status Update (Counter Update)

- 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.

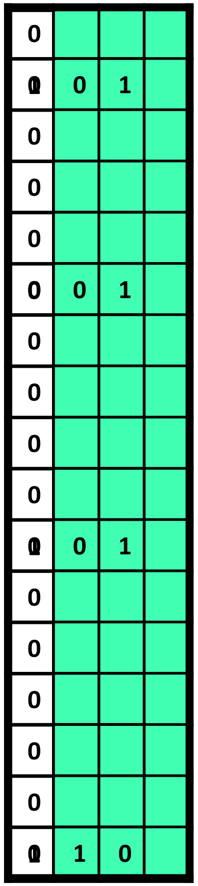


(K=4, H = 2)

Decay

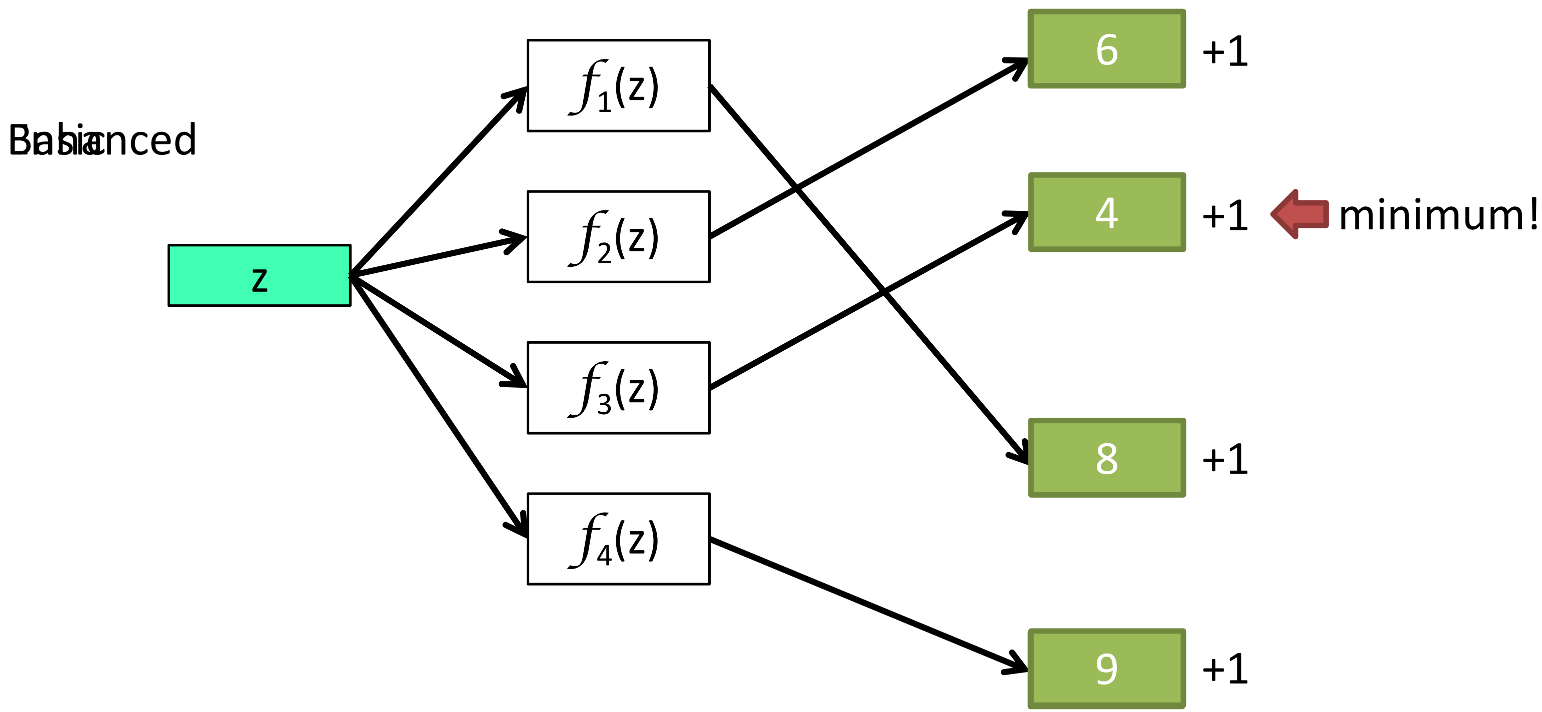
- For every given number of sectors have been written, called the "decay period" of the write numbers, the values of all counters are divided by 2 in terms of a right shifting of their bits.

(K=4, H = 2)



An Implementation Strategy

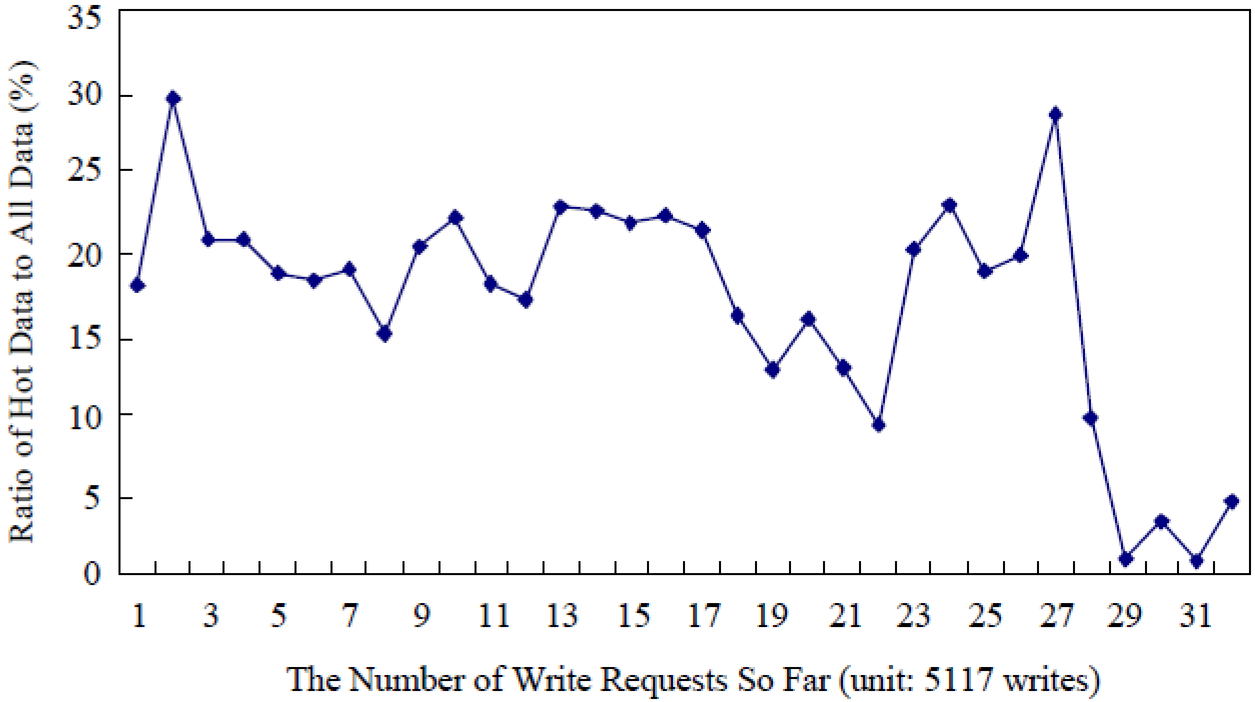
- In order to reduce the chance of false identification, only counters of the *K* hashed values that have the minimum value are increased



Performance Evaluation

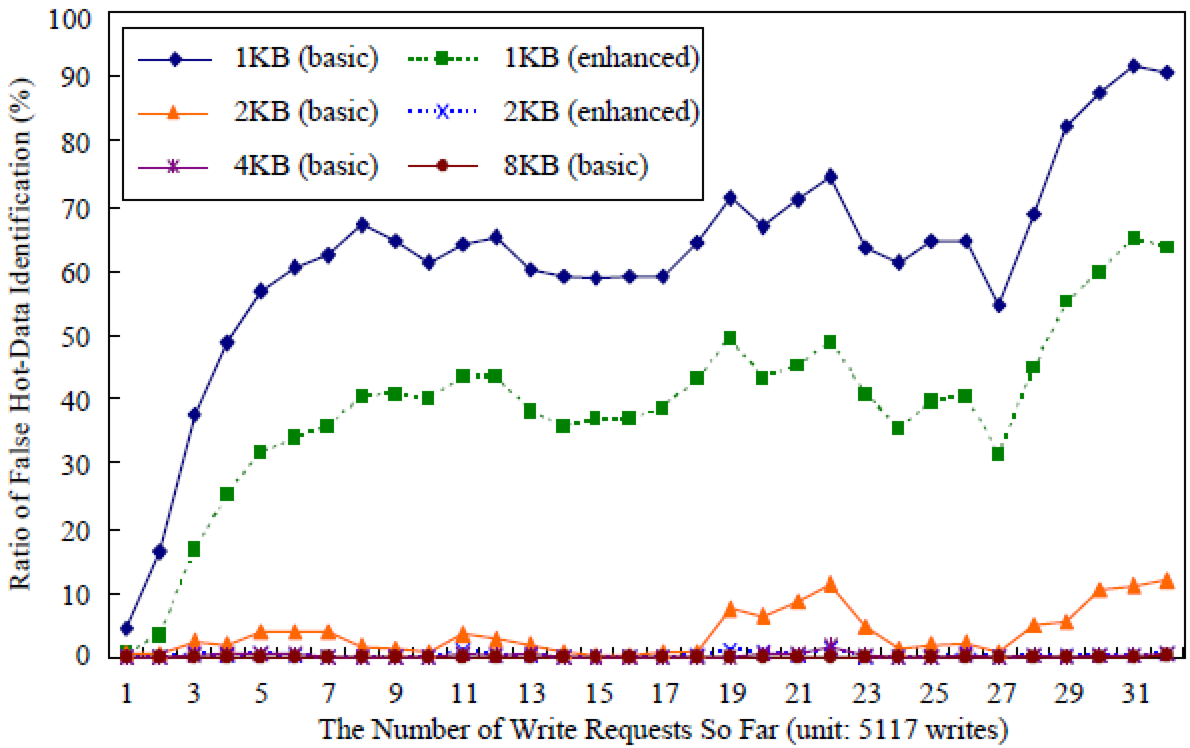
- Metrics
  - Impacts of Hash-Table Sizes
  - Runtime Overheads
- Experiment Setup
  - Number of hash functions: 2
  - Counter size: 4 bits
  - Flash memory size: 512 MB
  - Hot-data threshold: 4

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

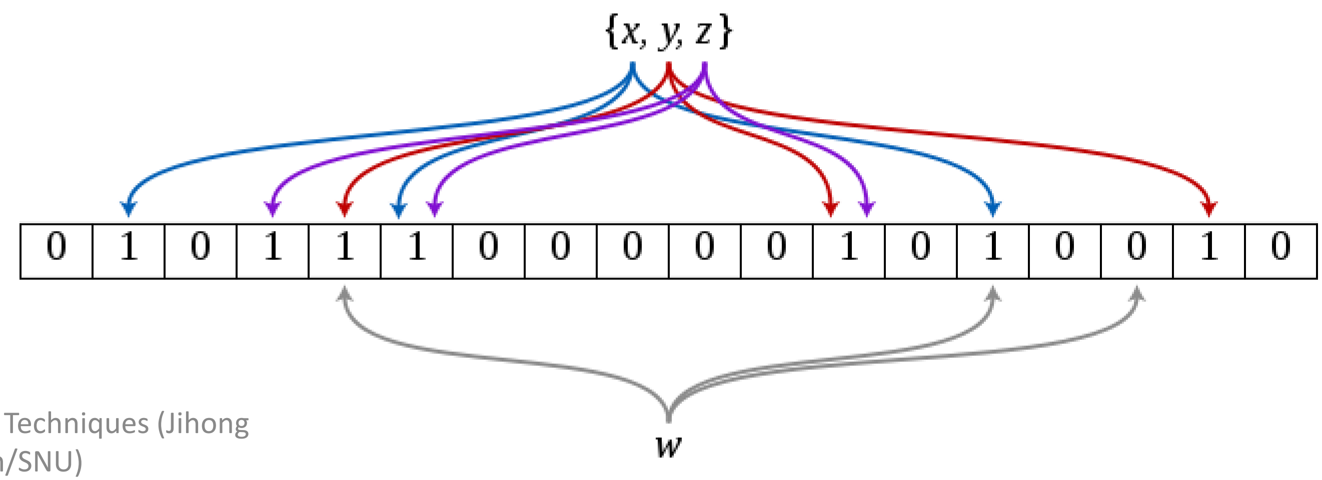
- Accurately captures frequency information
  - By maintaining counters
- Cannot appropriately capture recency information due to its exponential batch decay process (i.e., to decreases all counter values by a half at a time)

Multiple BF-based scheme

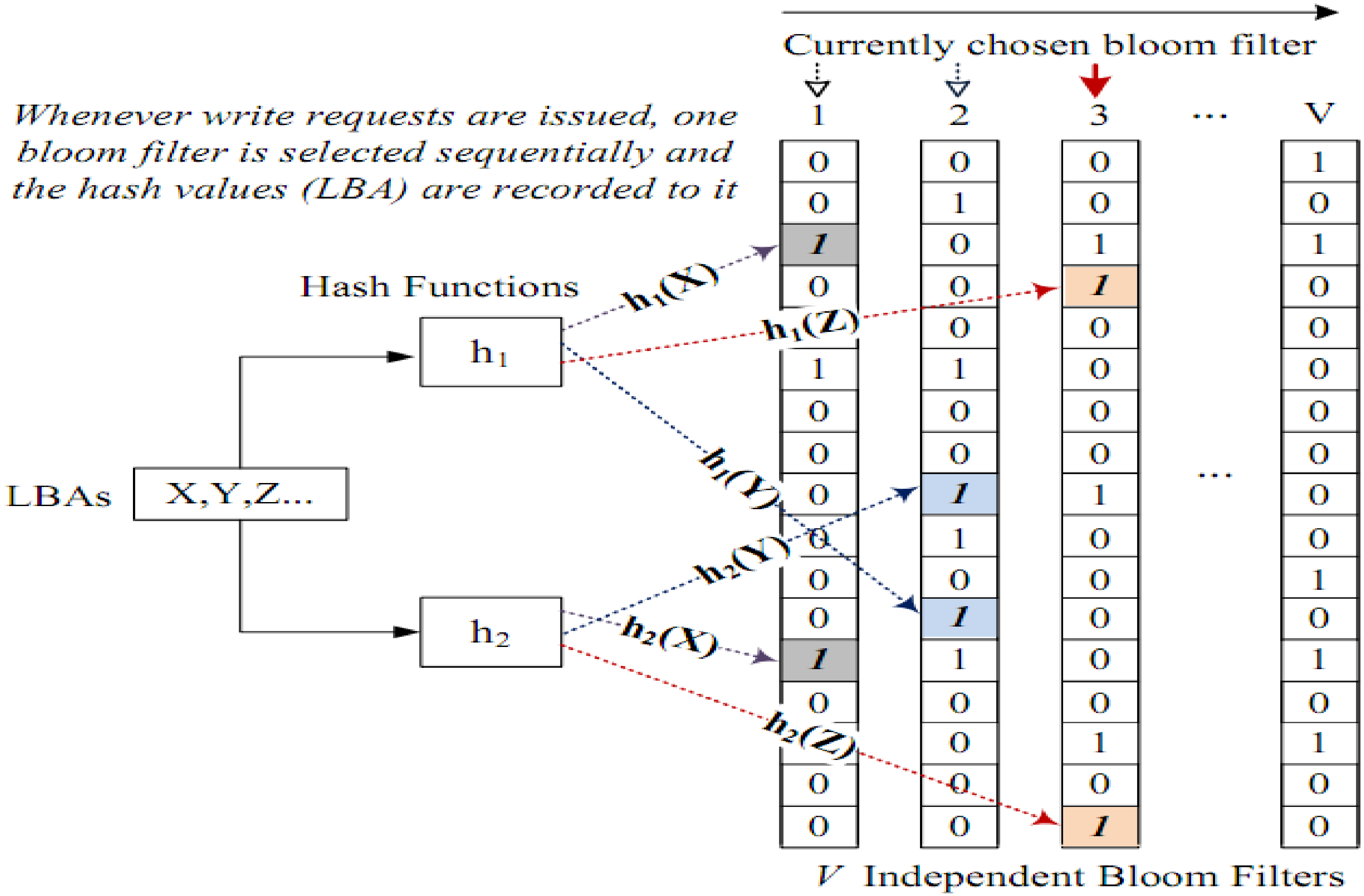
- Overview
  - Multiple bloom filters
    - To capture finer-grained recency
    - To reduce memory space and overheads
  - Multiple hash functions
    - To reduce false identification
- Frequency
  - Does not maintain access counters
- Recency
  - Different recency coverage

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

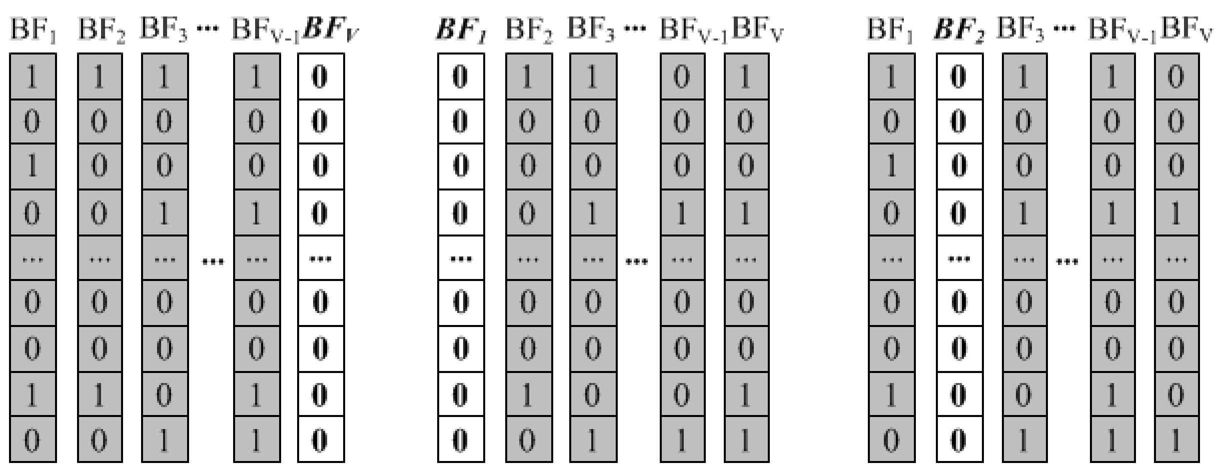
- No access counters
  - Needs a different mechanism
- For frequency capturing
  - Chooses one of BFs in a round-robin manner
  - If the chosen BF has already recorded the LBA
    - Records to another BF available.
  - Shortcut decision
    - If all BFs store the LBA information
      - Simply define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

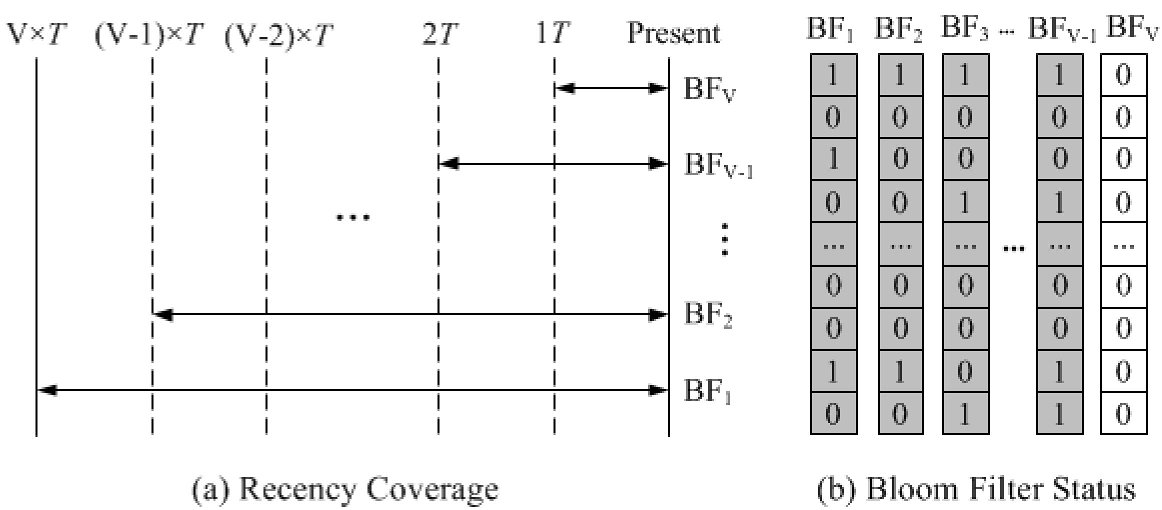
➔ ***Each BF retains a different recency coverage.***



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

Recency Coverage

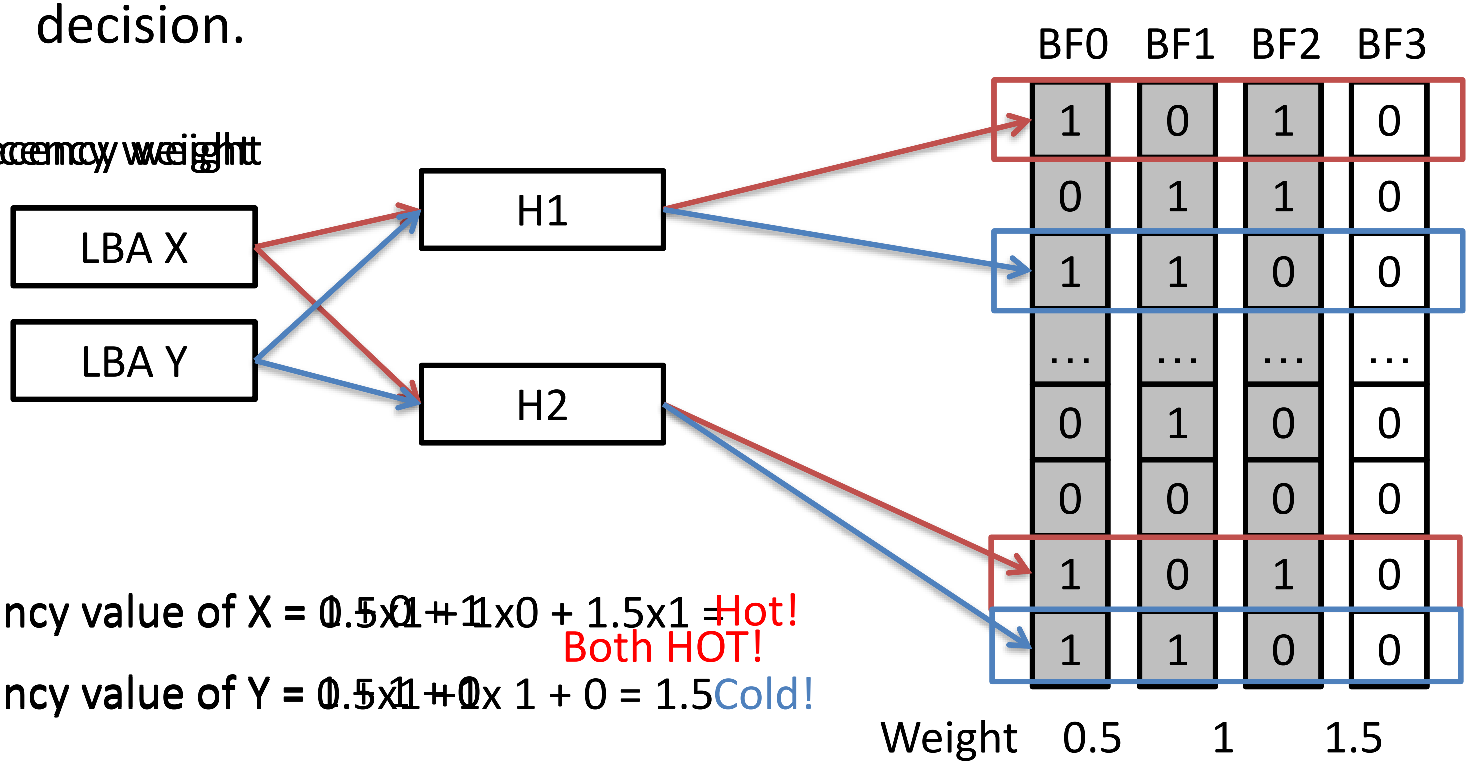
- For finer-grained recency
  - Each BF covers a different recency coverage
    - The reset BF (BF_v): Shortest (latest) coverage
    - The next BF (BF_1): Longest (oldest) coverage
  - Each BF has a different recency value



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.

w/o a recency weight
with a recency weight



Frequency value of X = 1 + 1 = 2
Frequency value of X = 0.5x1 + 1x0 + 1.5x1 = 2 Hot!

Both HOT!

Frequency value of Y = 1 + 1 = 2
Frequency value of Y = 0.5x1 + 1x 1 + 0 = 1.5 Cold!

Threshold value: 1.75

BF with valid data reset BF

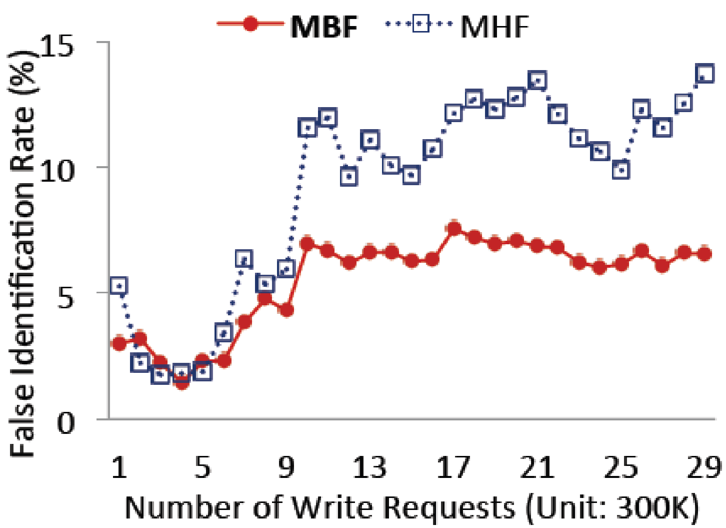
Performance Evaluation

- 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

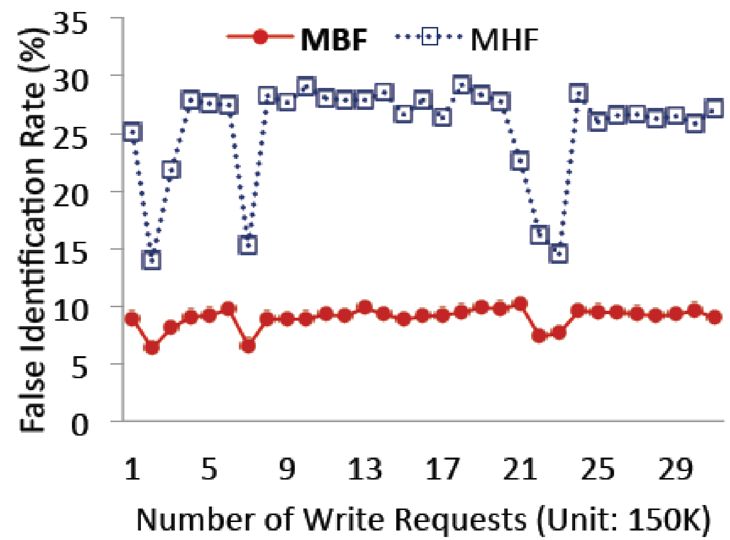
Performance Evaluation

- 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)

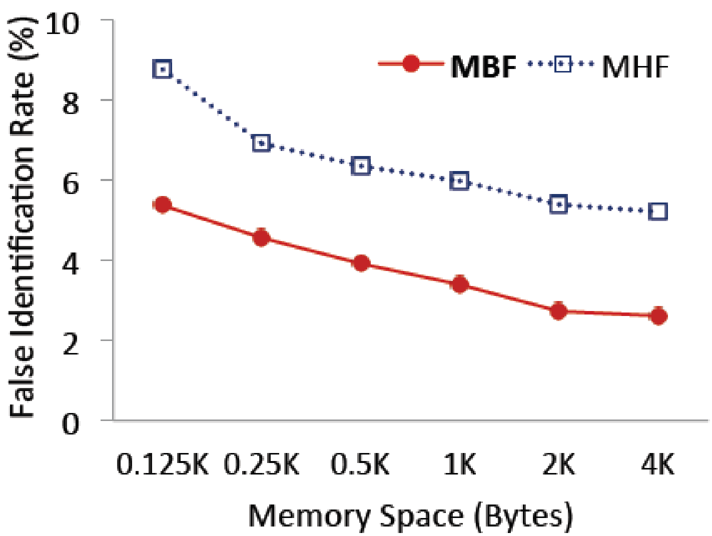


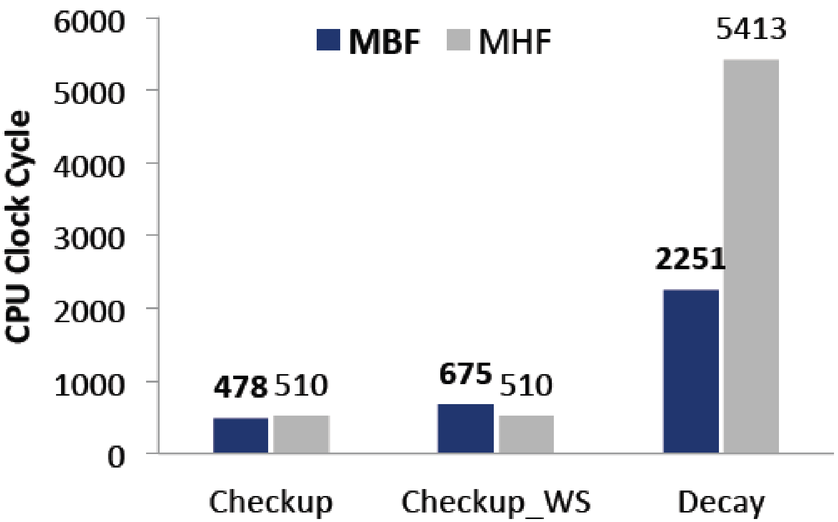
(a) Financial1



(b) MSR

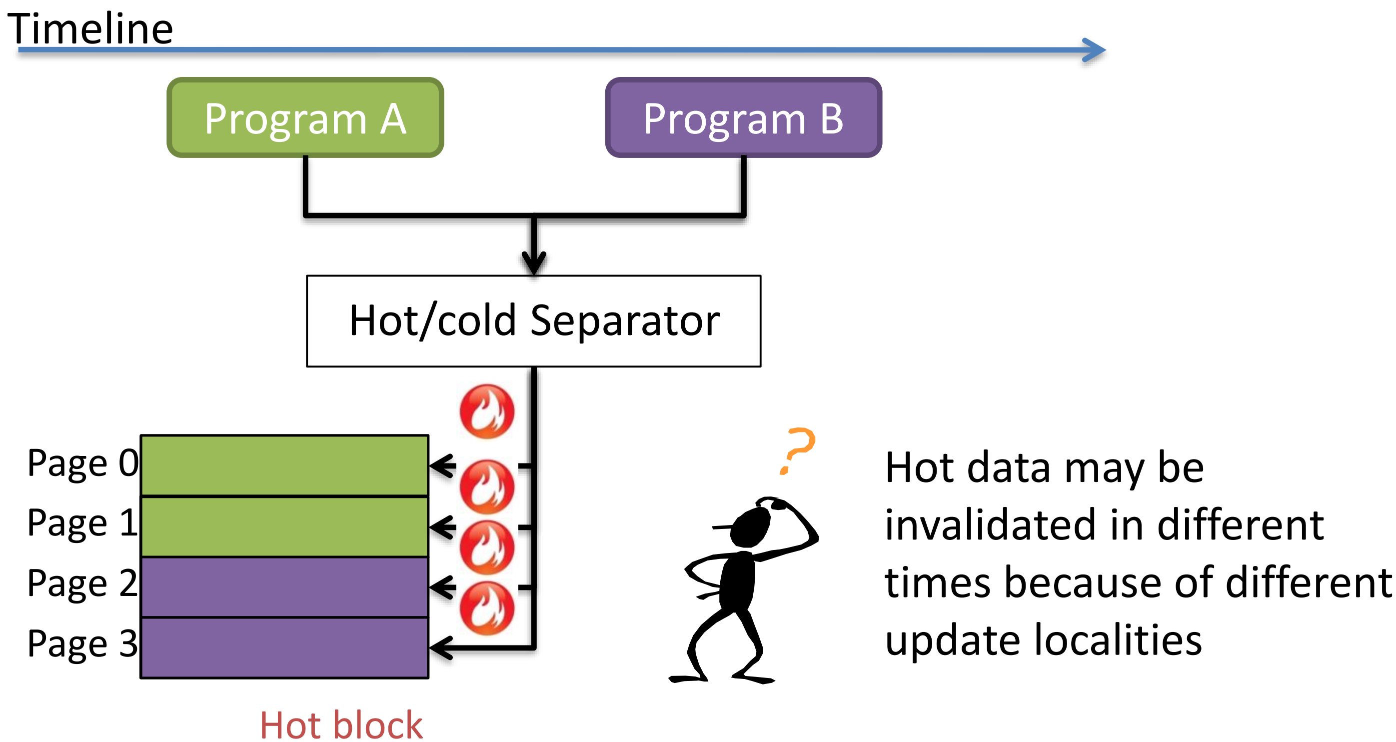
Memory Impact and Computational Overheads





Problems of Hot/cold Separator

– Problem 1: Wide variations on future update times

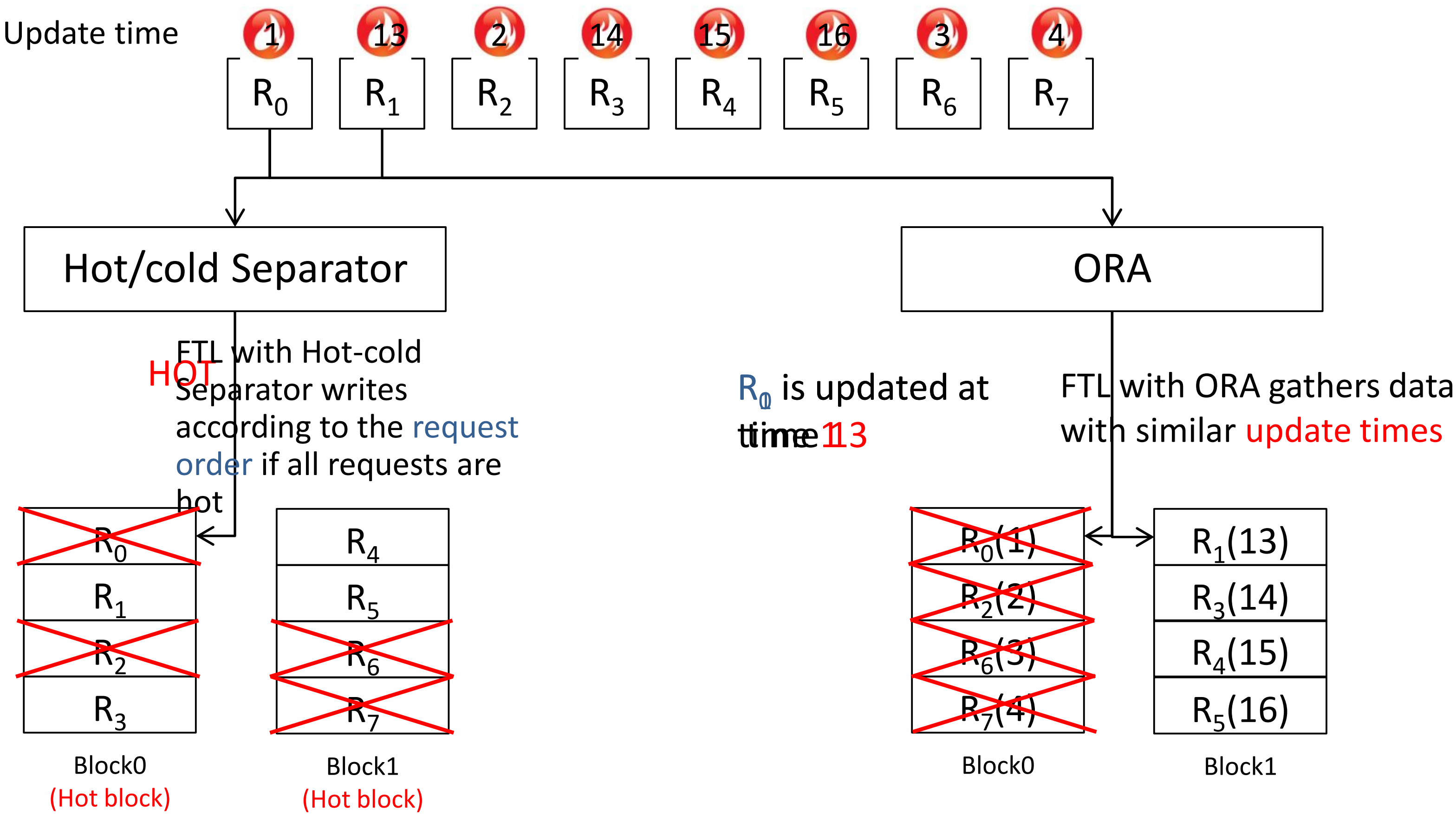


– Problem 2: If there is no clear temporal locality, hot/cold separator does not work

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



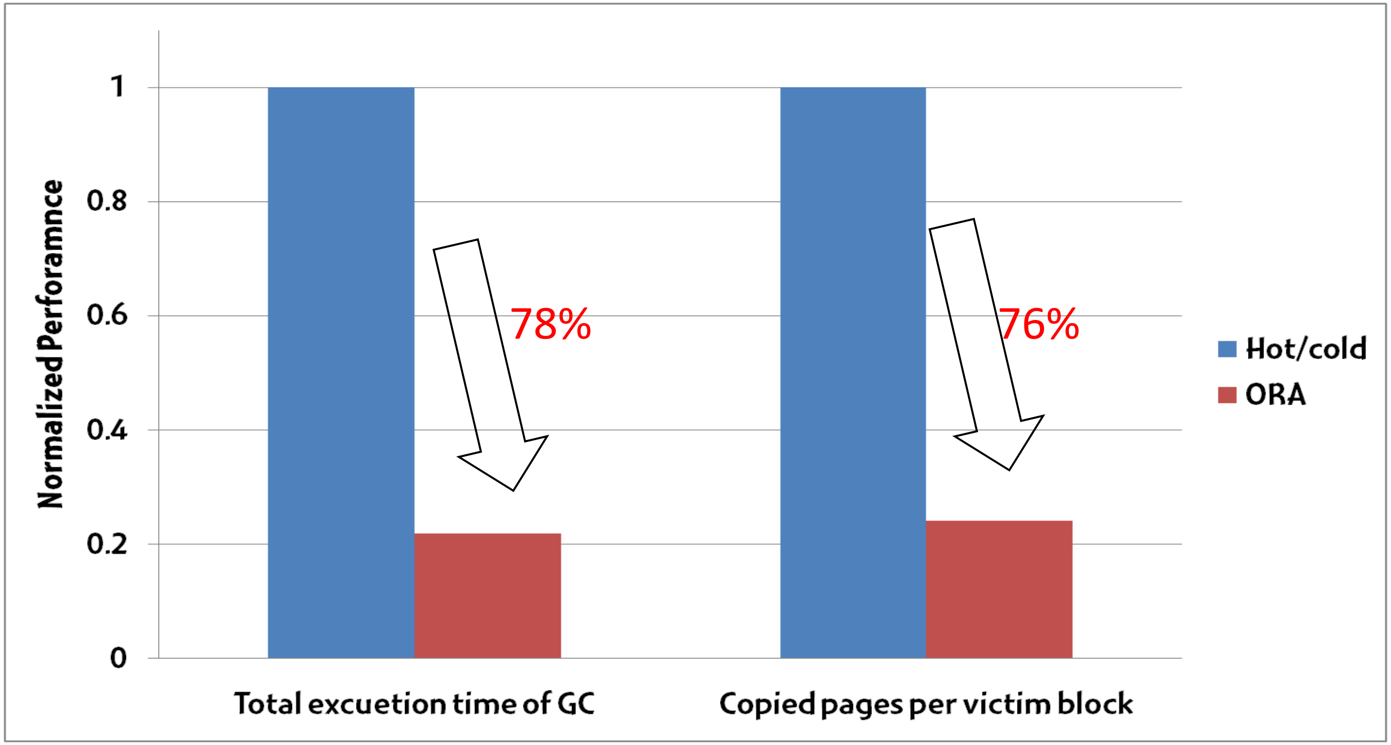
If a GC process was triggered at time 10,

4 copies + 2 erasures

1 erasure

Hot/cold Separator vs. ORA

- ORA can reduce GC overhead significantly



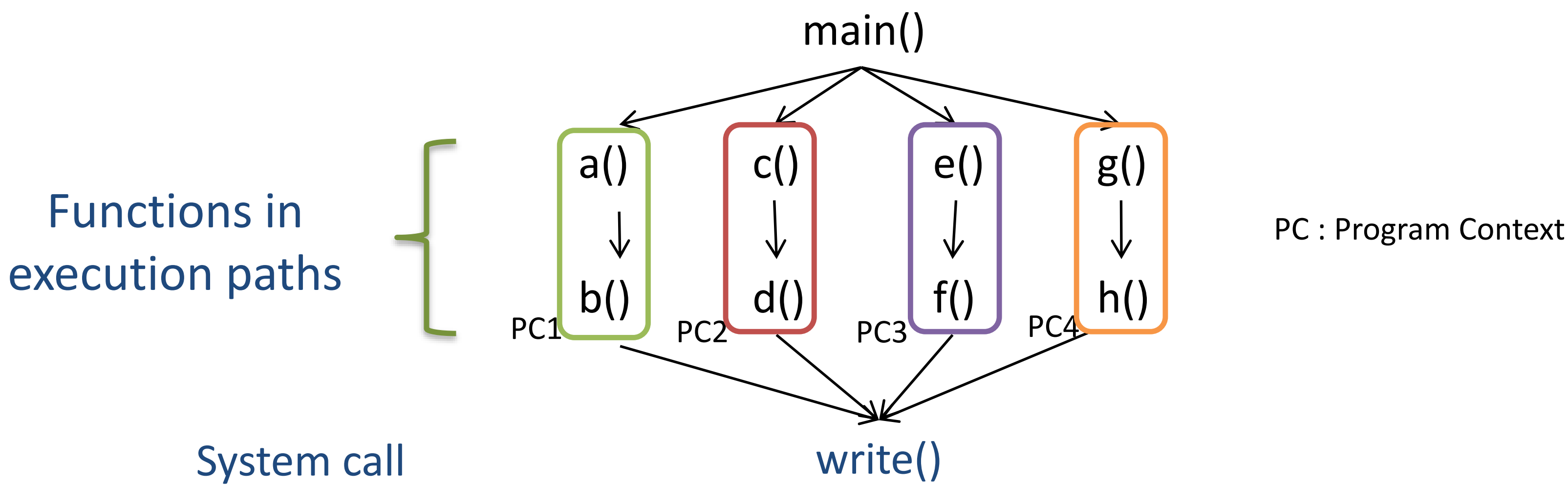
Update time is a more important factor in data separation technique than frequency of updates

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



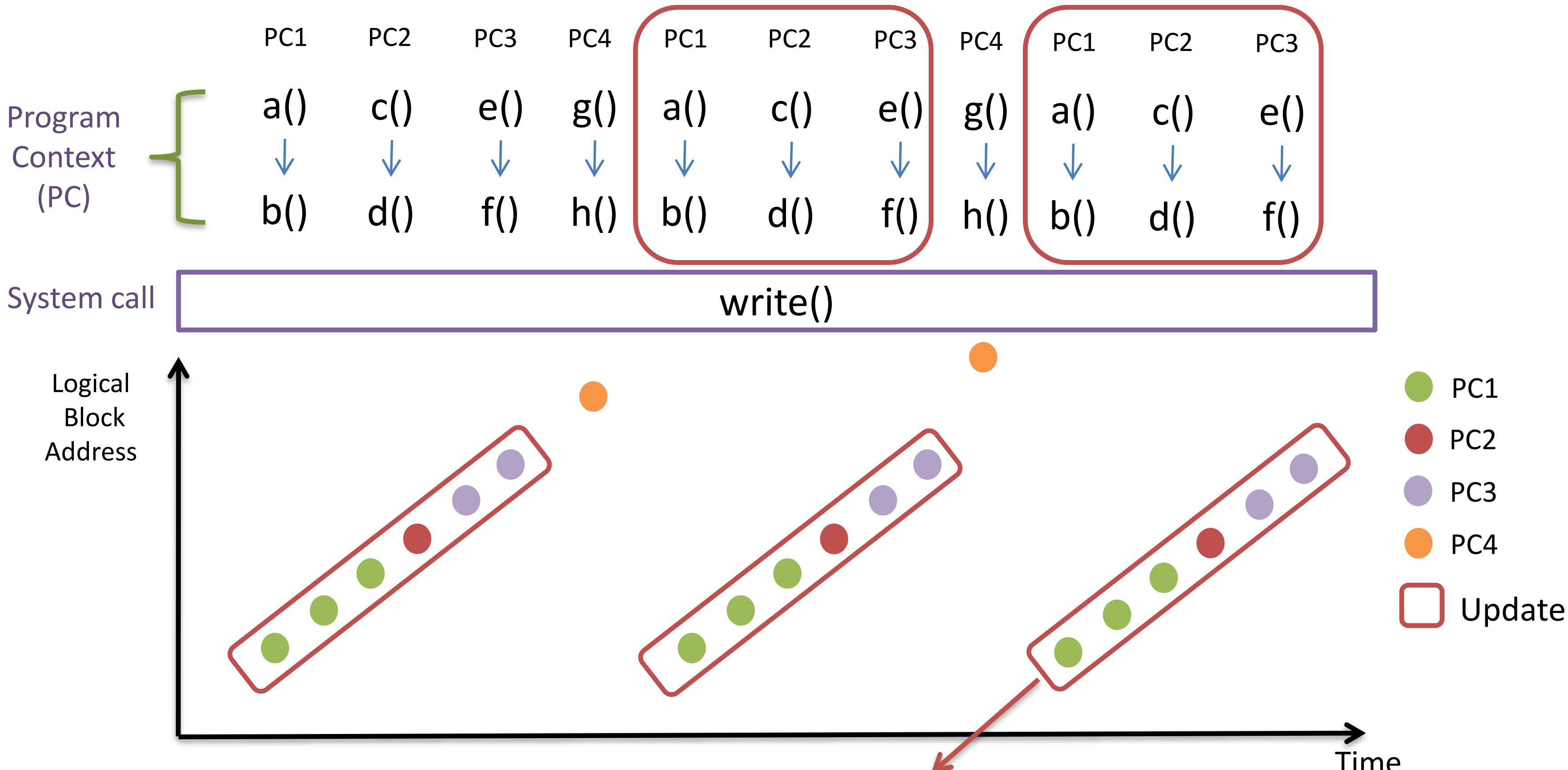
- Identification
  - Each program context is identified by summing program counter values of each execution path of function calls

    Reference

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

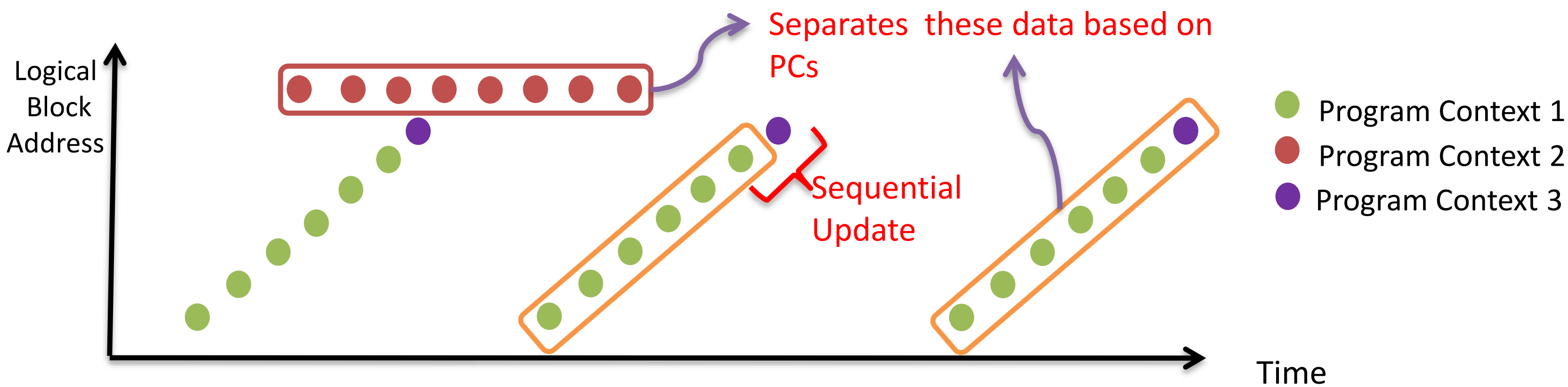
Program Context–Based Update Time Prediction

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



These data are updated in a similar period when PC1, PC2, and PC3 are executed

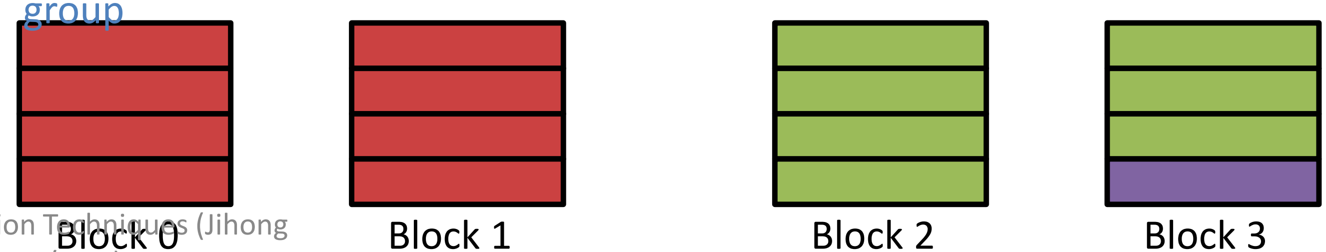
Separating Data using Program Contexts



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

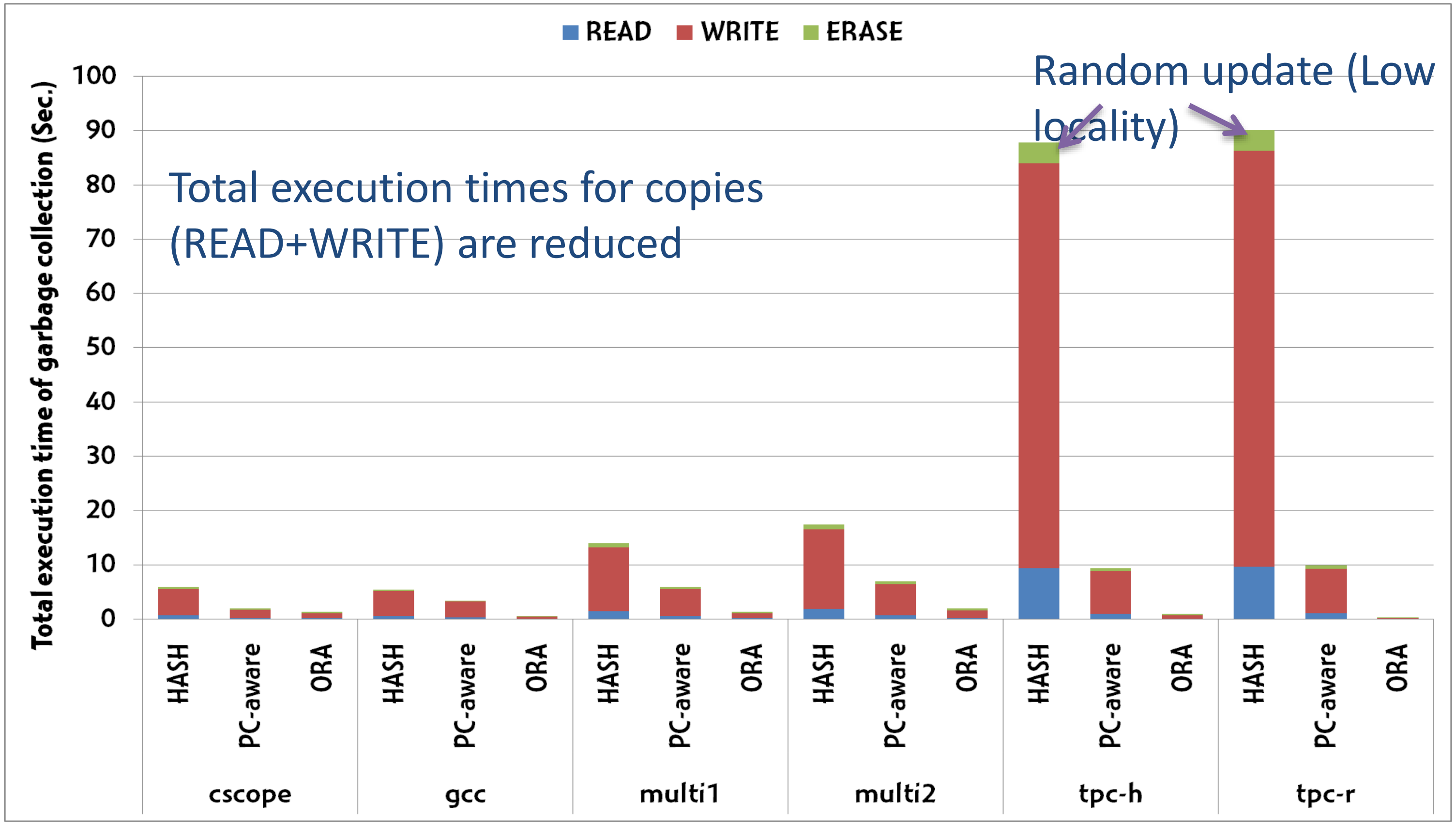
- Techniques for comparison
  - HASH: Hash-based hot/cold separation technique
  - ORA: Oracle predictor on future update times of data

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

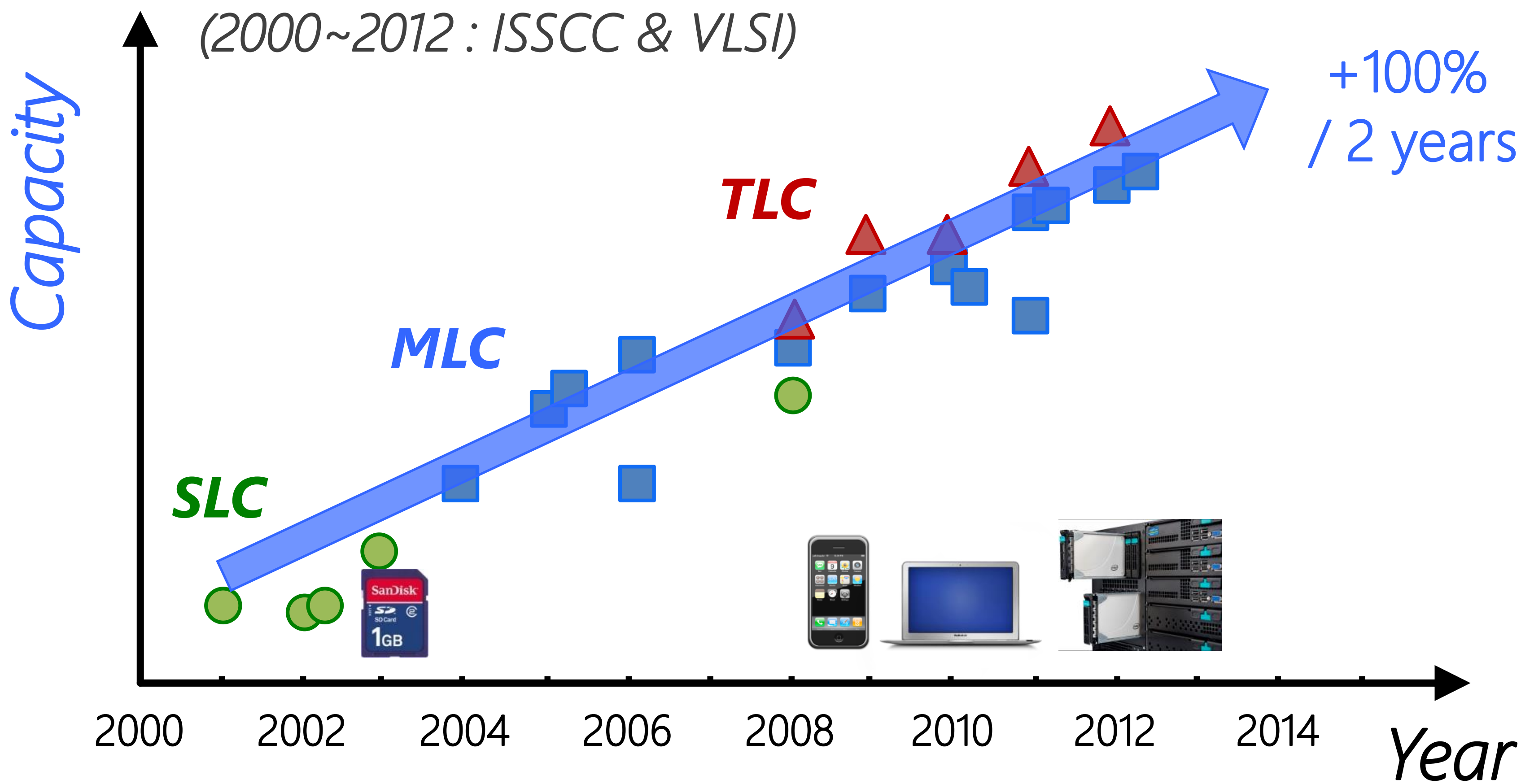
Lifetime Issues & Techniques

Jihong Kim
Dept. of CSE, SNU

Outline

- Introduction to lifetime problem in SSDs
- SSD Lifetime Extension Techniques
  - Compression Technique
  - Deduplication Technique: CAFTL
  - Dynamic Throttling: READY

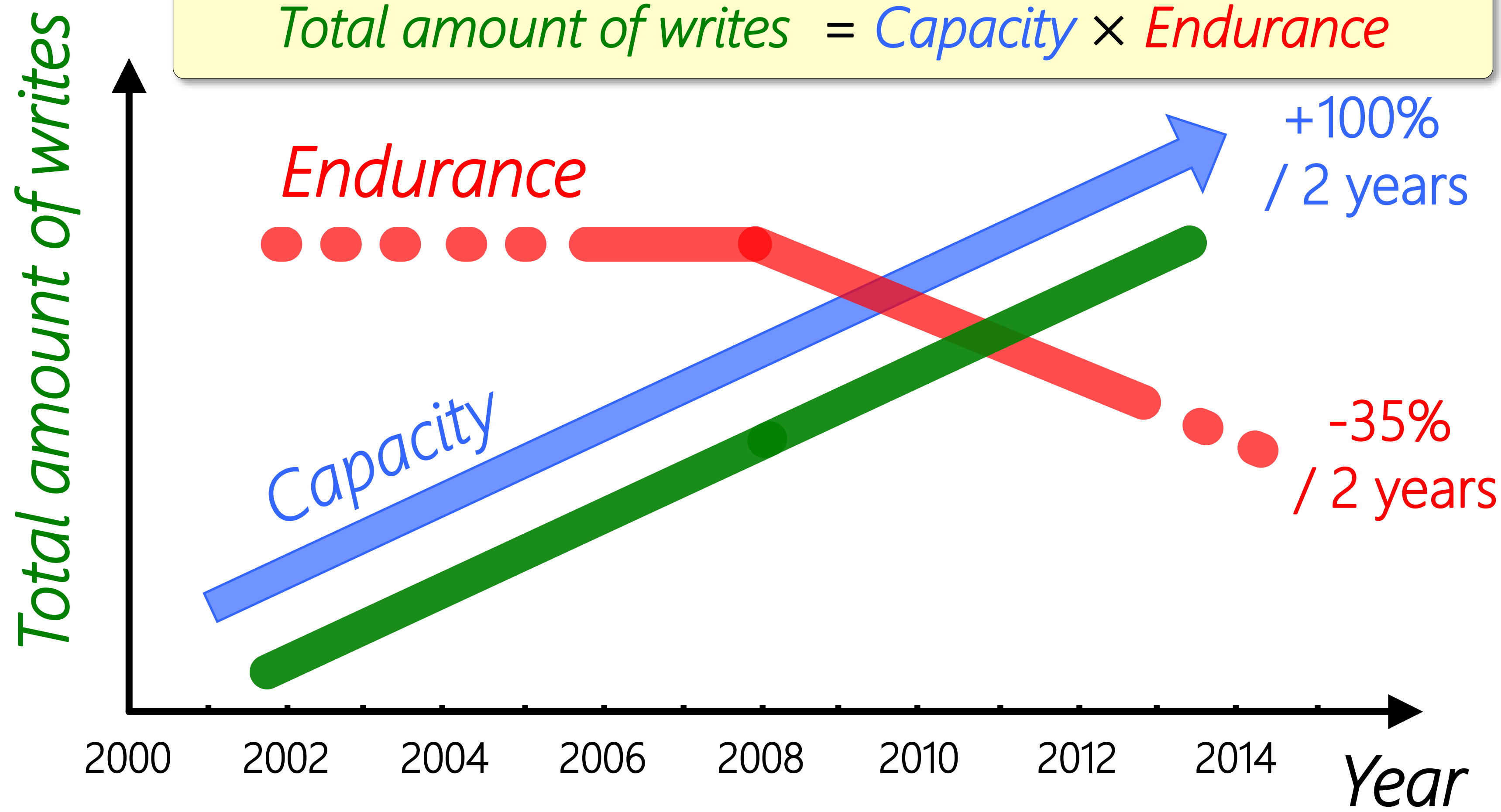
Trend of NAND Device Technologies



NAND capacity is continuously increased, and NAND flash-based storages are widely adopted.

Trend of NAND Device Technologies

Total amount of writes = Capacity × Endurance



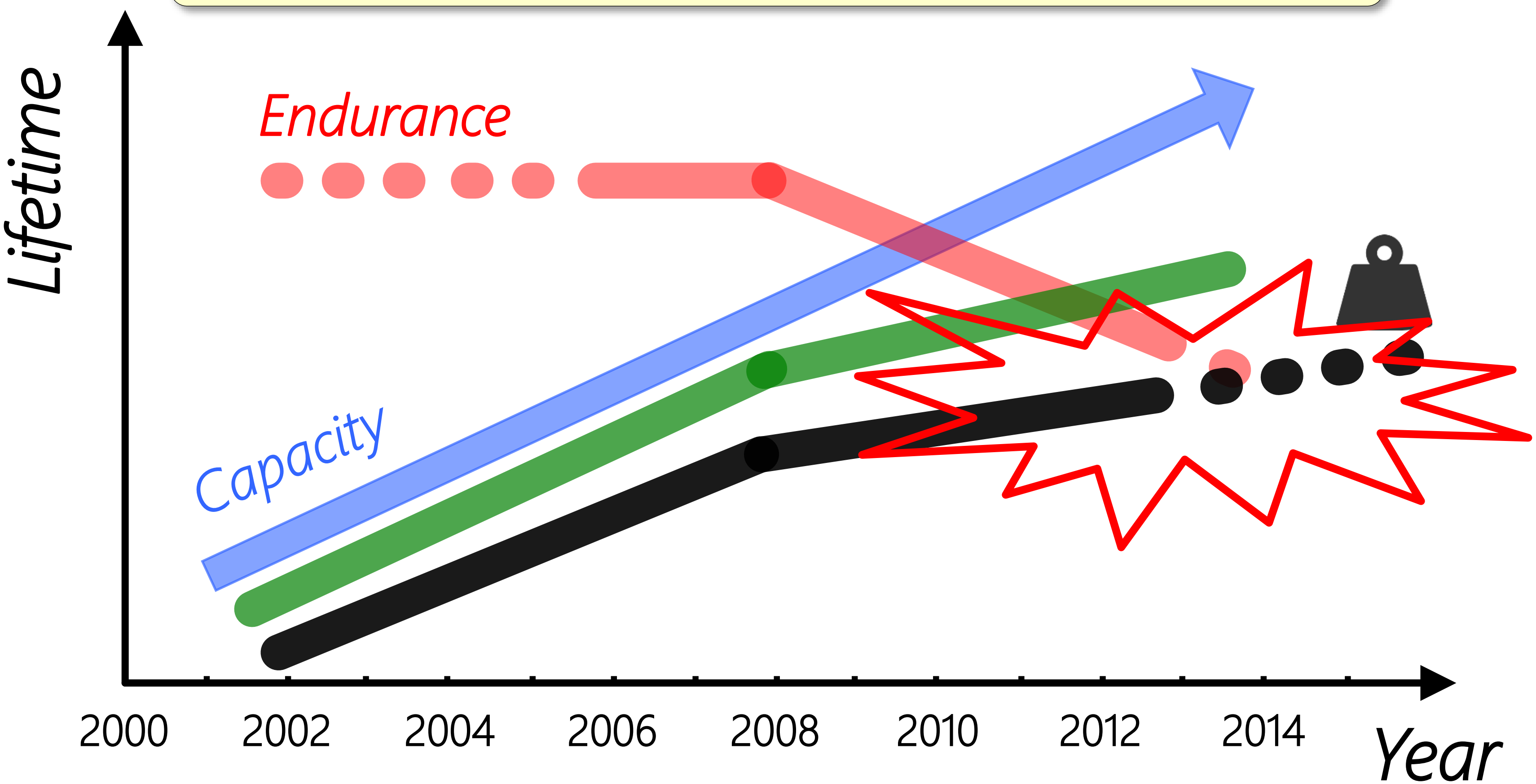
Total amount of writes of NAND flash-based storages

does not increase as much as we expected.

Lifetime Problem of NAND-based Storages

$$\text{Lifetime} = \frac{\text{Total amount of writes}}{\text{Daily amount of data}}$$

$$\text{Lifetime} = \frac{\text{Total amount of writes}}{\text{Daily workload} \times \text{WAF}}$$



Decreasing lifetime is a main barrier for sustainable growth.

Techniques for Improving Lifetime

$$\textbf{Lifetime} \propto \frac{\textbf{Capacity} \times \textbf{Endurance}}{\textbf{Daily workload} \times \textbf{WAF}}$$

Endurance:
- *Self-Healing SSDs*
- *Dynamic erase voltage and time scaling*

Daily workload:
- *Deduplication*
- *Compression*
- *Throttling*

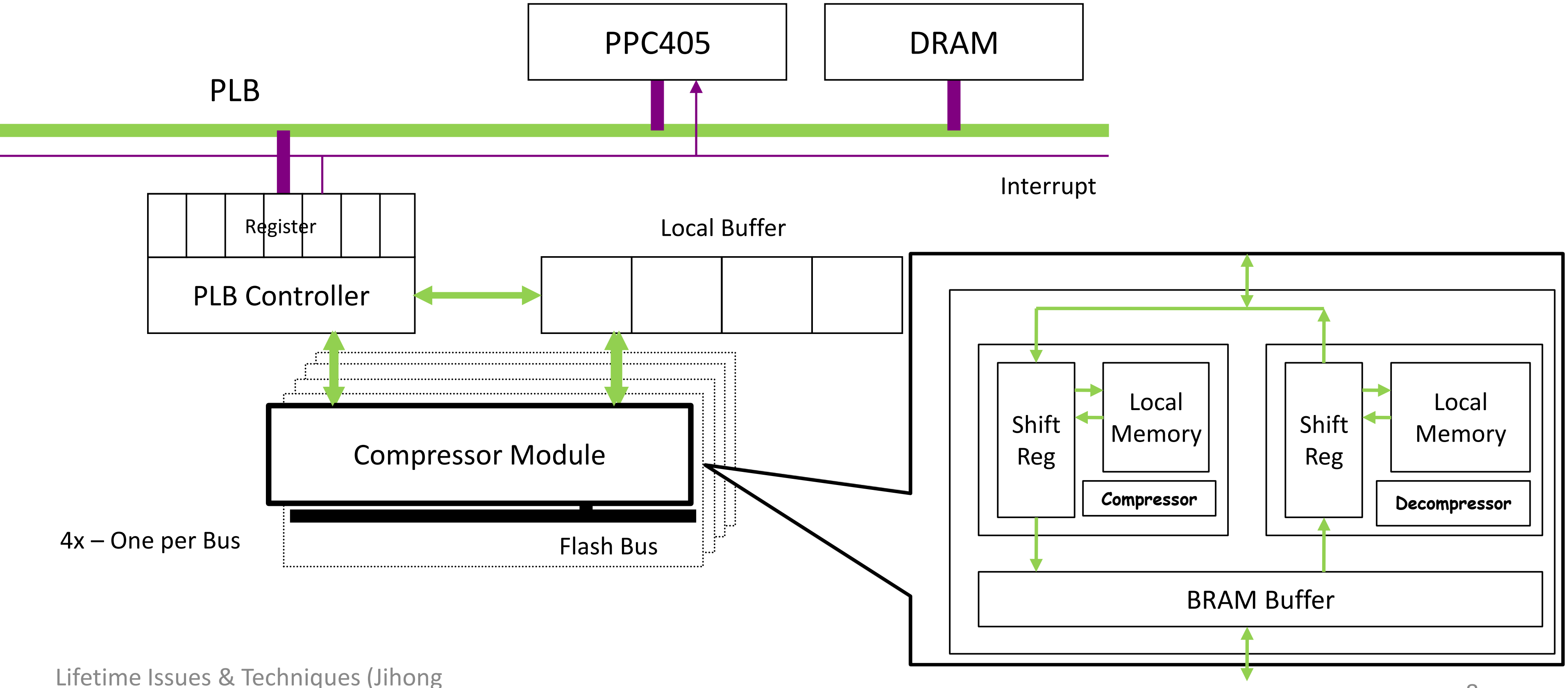
WAF:
- *Optimization of garbage collection & wear leveling*

Workload-Reduction Methods for Extending SSD Lifetime

- Reduce amount of written data
  - Compression technique
    - Compressed data are stored
  - Deduplication technique
    - Prevent redundant data from being stored in SSDs
- Throttling SSD Performance
  - Dynamic Throttling
    - Guarantee the lifetime of SSD by throttling write traffic

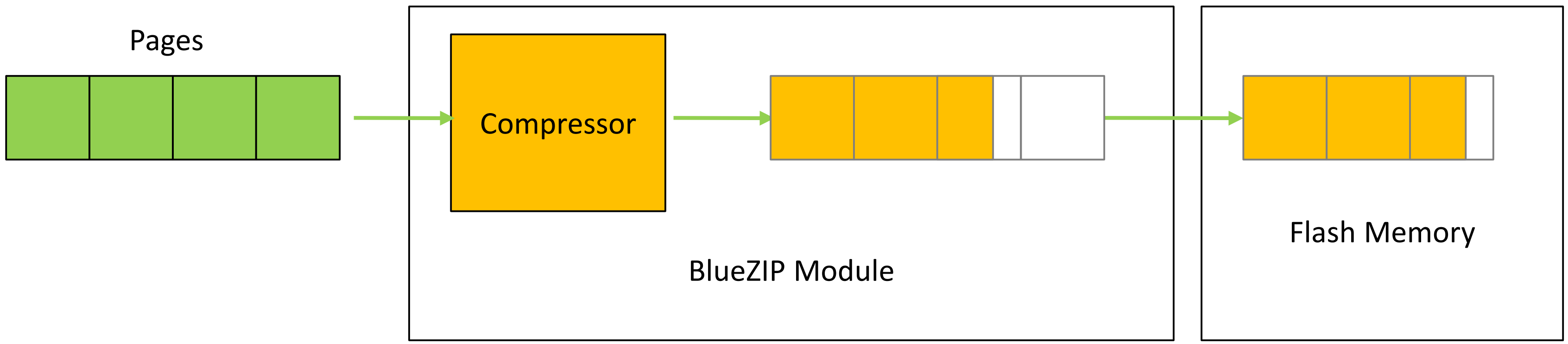
Compression Technique in SSD

- Reduces the amount of data written
- Improve effectively both the write speed and the reliability of a SSD
- Case Study: BlueZip



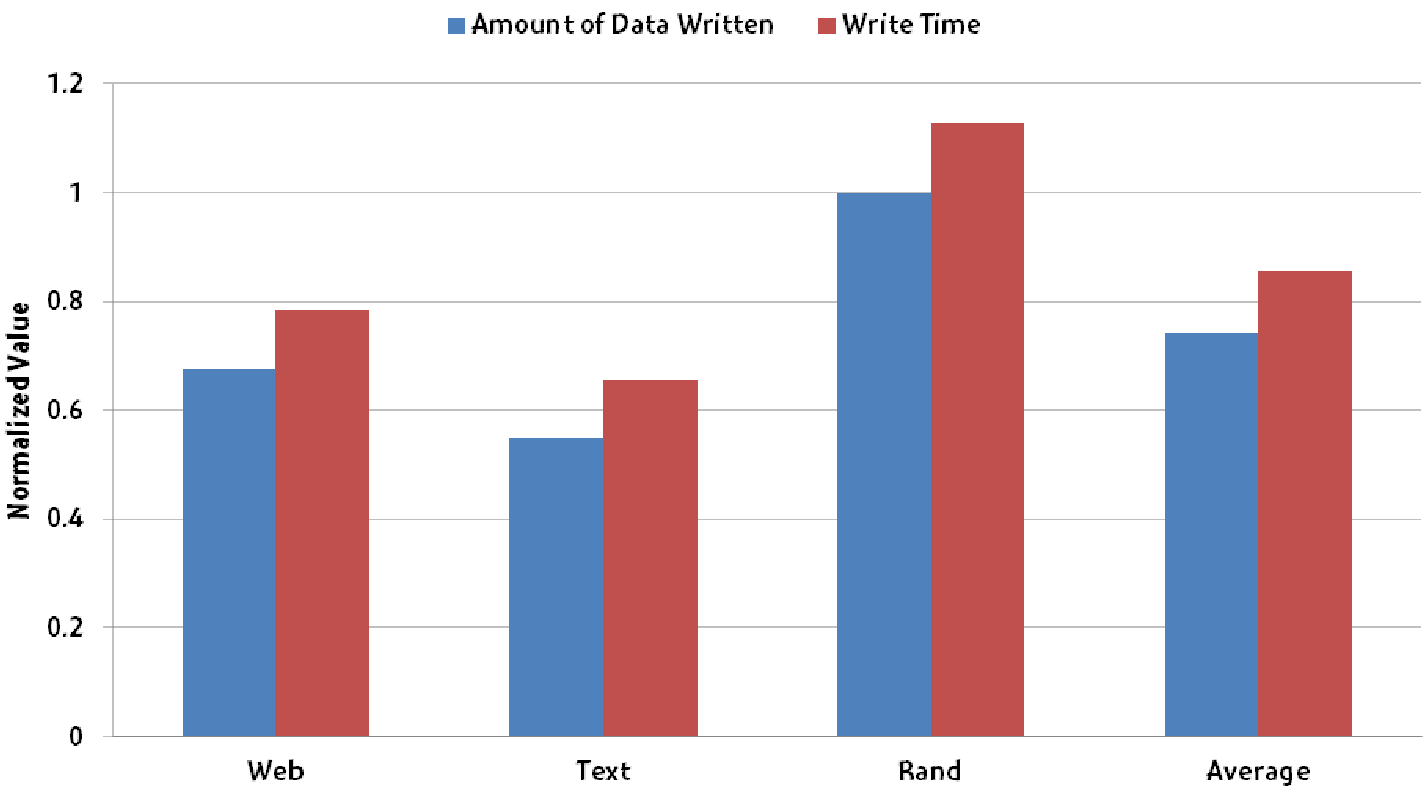
Design of BlueZIP

- BlueZIP
  - Based on the LZRW3 algorithm for compression/decompression
  - Has a local memory which is used as a hash table for compression
  - Compresses data and writes the compressed data into the BRAM buffer
  - The flash controller reads the compressed data from the BRAM buffer and writes them into the flash board
- FTL
  - Gives BlueZIP multiple pages to compress and write them
  - Accepts return value from BlueZip, which is the size of the compressed data



Primary Performance Evaluation

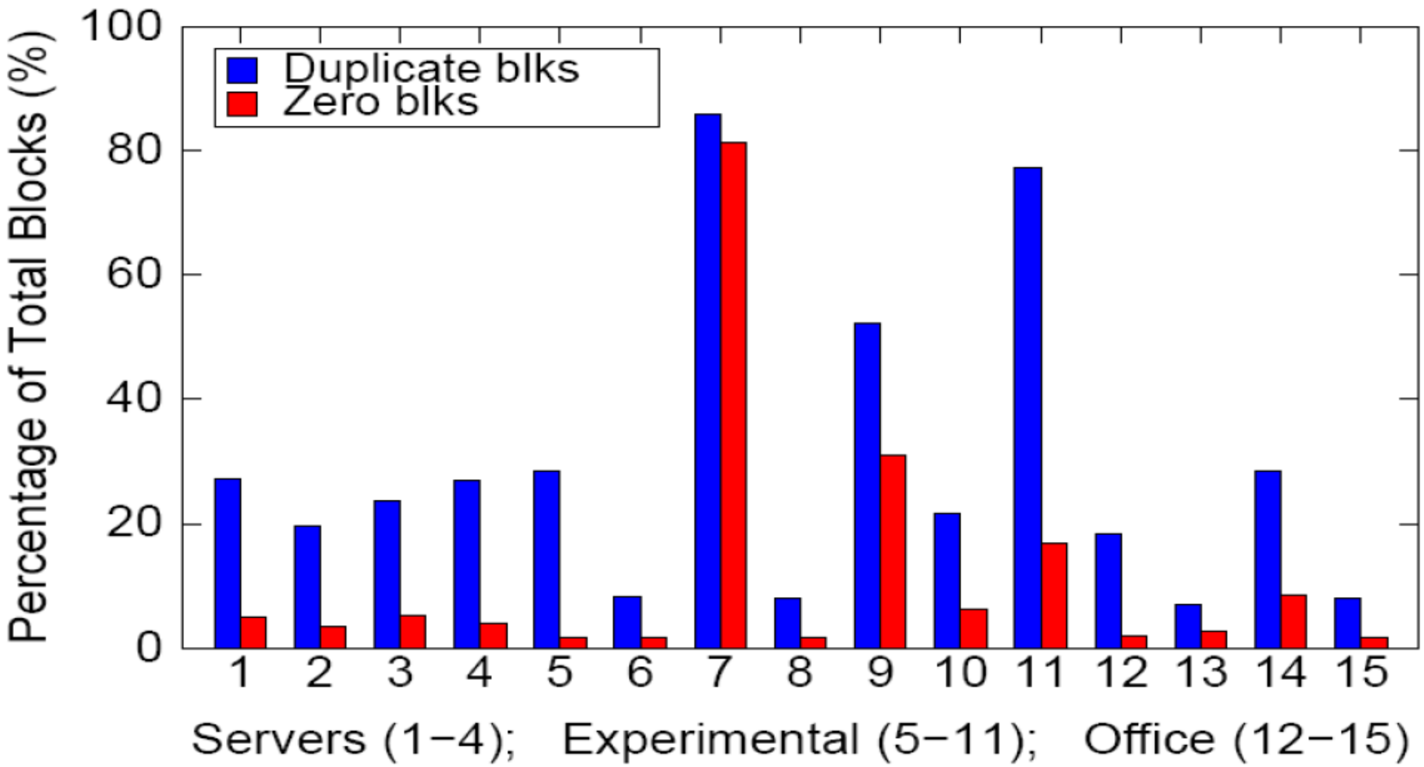
- Reduce the write times by 15% on average
- Reduce the amount of written data by 26% on average



Deduplication Technique - CAFTL

Data Redundancy in Storage

- Duplicate data rate – up to 85.9% over 15 disks in CSE/OSU



Content-Aware Flash Translation Layer (CAFTL)

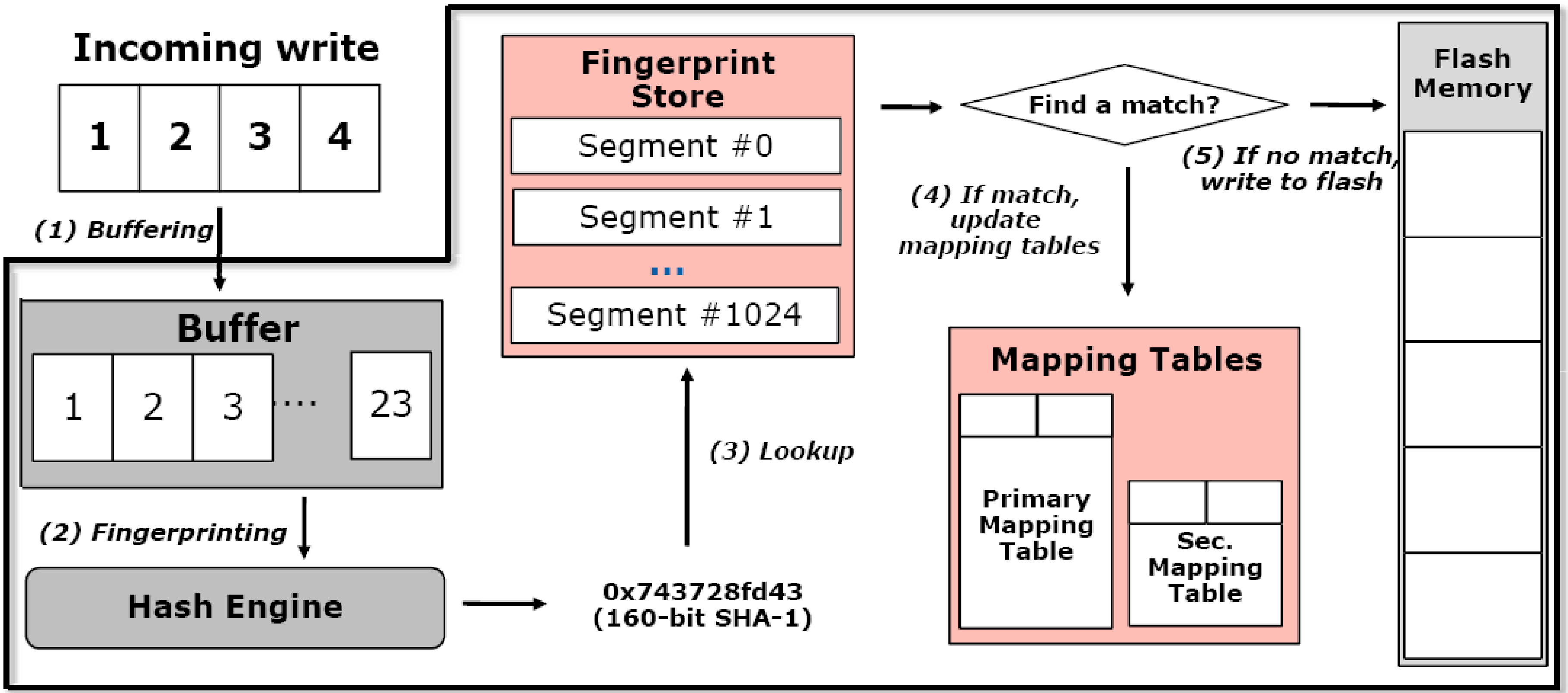
- Key Idea
  - Eliminating duplicate writes
  - Coalescing redundant data
- Potential benefits
  - Removing duplicate writes into flash memory -> reducing "write/day"
  - Extending available flash memory space -> increasing available "flash space"

$$\text{Lifetime} = \frac{\text{Endurance} \times \text{Capacity}}{\text{Write/day} \times \text{Efficiency of FTL}}$$

Overview of CAFTL

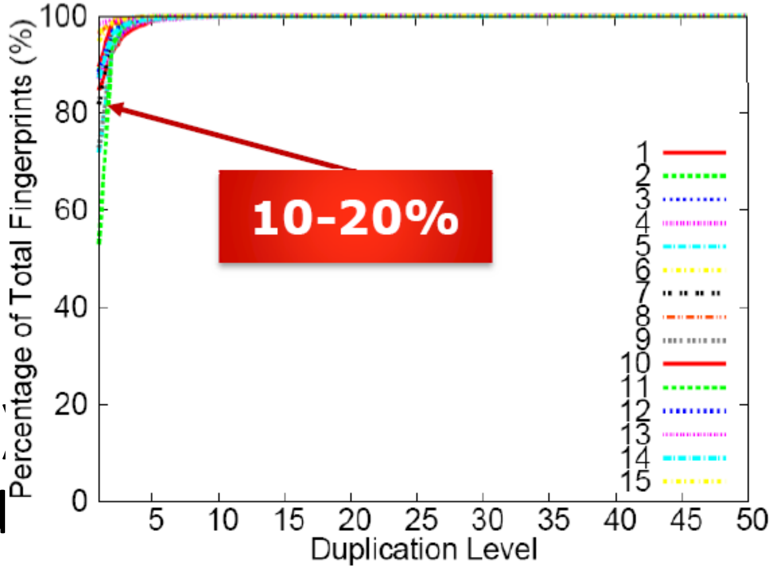
- In-line deduplication
  - Proactively examines incoming data
  - Cancels duplicate writes before committing a request
  - Best-effort solution
- Out-of-line deduplication
  - Periodically scans flash memory
  - Coalesces redundant data out of line

Architecture of CAFTL



Fingerprint Store Challenges

- Fingerprint store
  - Maintains fingerprints in memory
- Challenges
  - Memory overhead (25 bytes each)
  - Fingerprint store lookup overhead

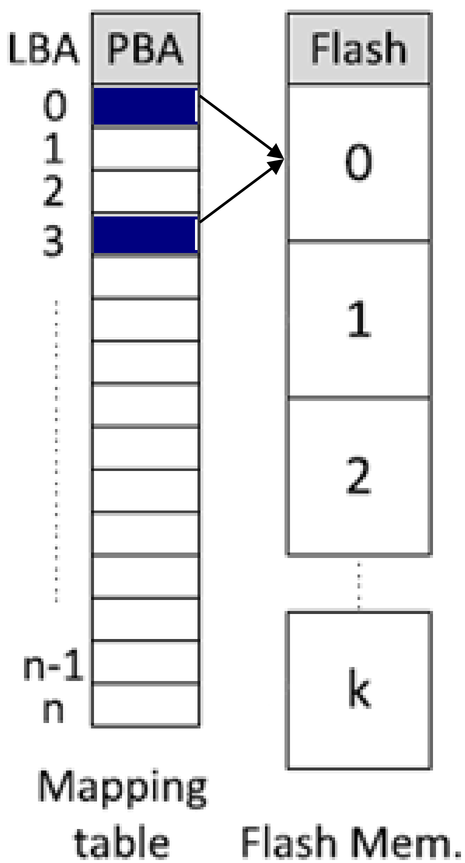


- Observations and indications
  - Skewed duplication fingerprint distribution – only 10~20%
    - Most fingerprints are not duplicate -> waste of memory space

Store only the most likely-to-be-duplicate fingerprints in memory

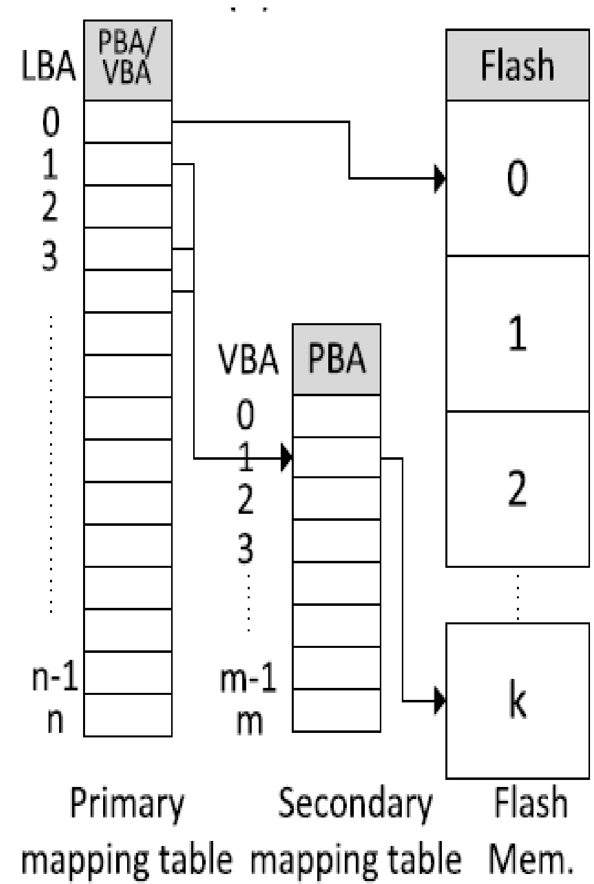
Challenges of Existing Mapping Table

- When a physical page is relocated to another place, all the logical pages mapped to this page should be updated quickly
- For update request, the physical page cannot be invalidated if the page is shared
  - Must track the number of referencing logical pages



Two-Level Indirect Mapping

- Virtual Block Address (VBA) is introduced
  - Additional indirect mapping level
  - Represents a set of LBAs mapped to same PBA
  - Each entry consists of
    {PBA, reference}
- Significantly simplifies reverse updates
- Secondary mapping table can be small
  - Since most logical pages are unique
- Incurs minimal additional lookup overhead

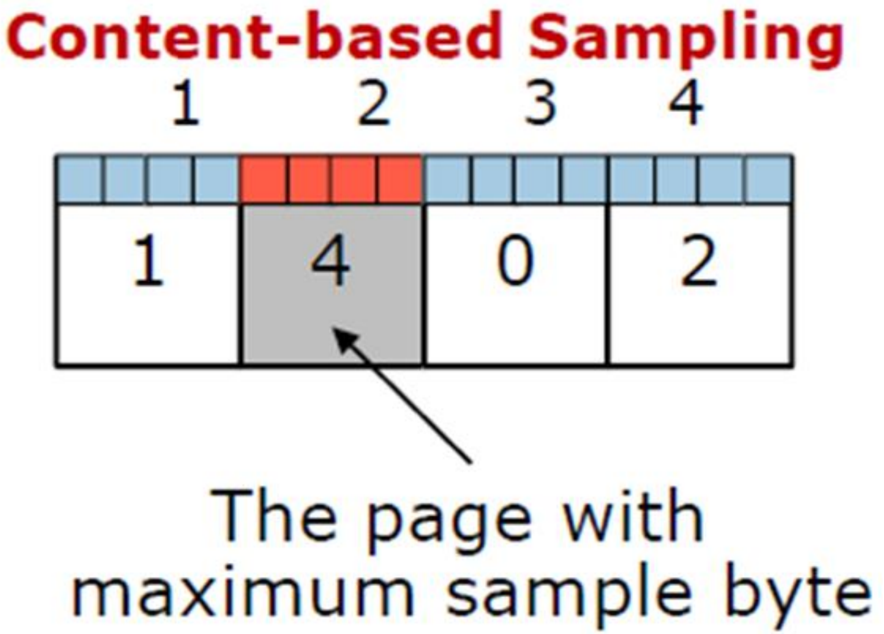


Sampling for Hashing

- Most writes are unique -> most hashing operations turn out useless eventually
- Intuition
  - If a page in a write is a duplicate page, the other pages are likely to be duplicate too
- Sampling
  - Select one page in a write request as a sample
  - If the sample page is duplicate, hash and examine the other pages
  - Otherwise, stop fingerprinting the whole request at earliest time

Selecting Sample Pages

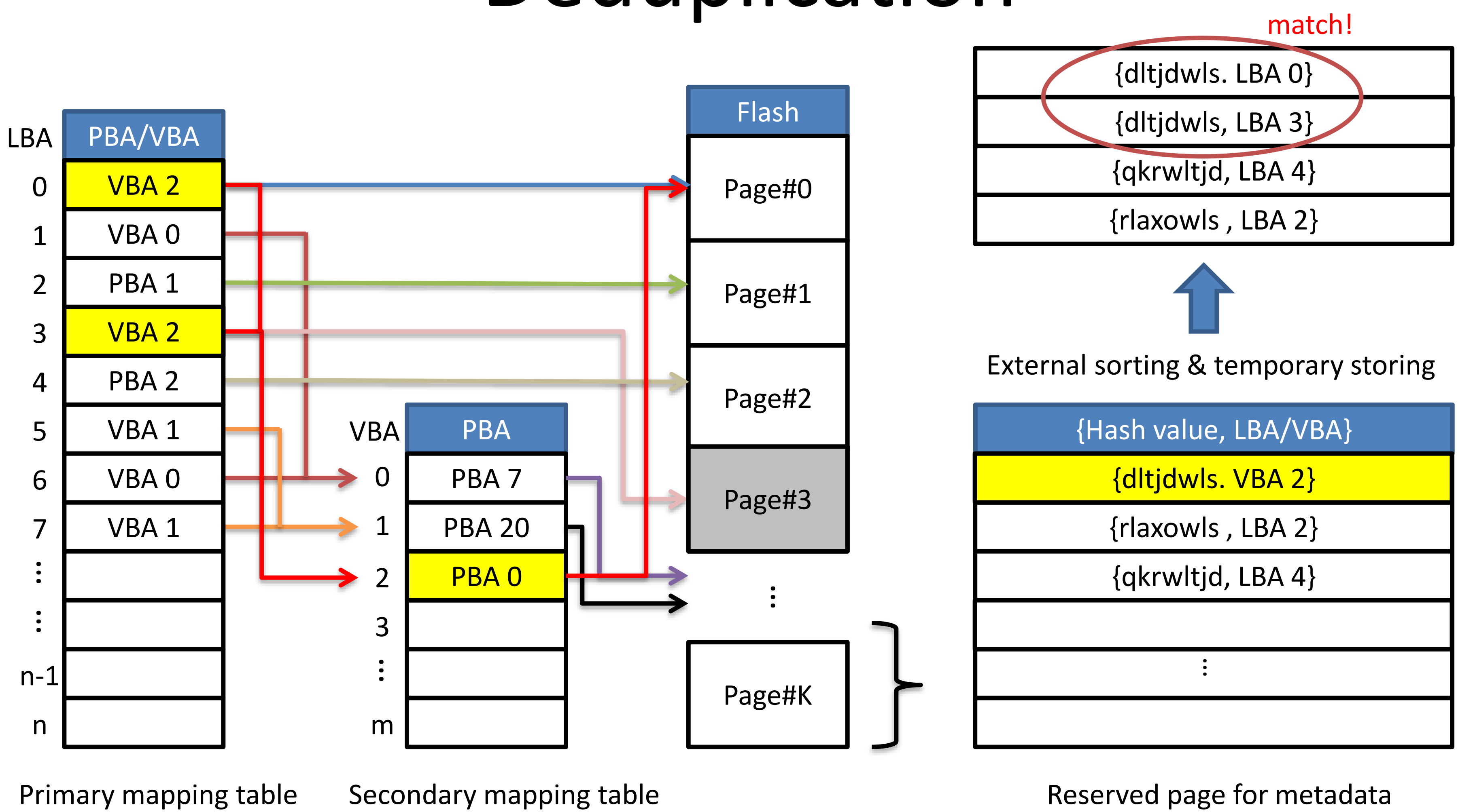
- Content-based sampling
  - Selecting/comparing the first four bytes (i.e. sample bytes) in each page
    - Concatenating the four bytes into a 32-bit numeric value
  - The page with the largest value is the sample page



Out-of-line Deduplication

- Periodically launched during device idle time
- Uses external merge sort to identify duplicate fingerprint
  - Part of the meta data page array is loaded into memory and sorted and temporarily stored in flash
- CAFTL reserves dedicated number of flash pages to store metadata (e.g. LBA and fingerprint)
  - For 32GB SSD with 4KB pages, it needs only 0.6% of flash space

Example of Out-of-line Deduplication



Performance Evaluation

- SSD simulator
  - Microsoft Research SSD extension for Disksim simulator
  - Simulator augmented with CAFTL design and on-device buffer

-

Description	Configurations
Flash page size	4KB
Pages / block	64
Blocks / plane	2048
Num of pkgs	10
Over-provisioning	15%

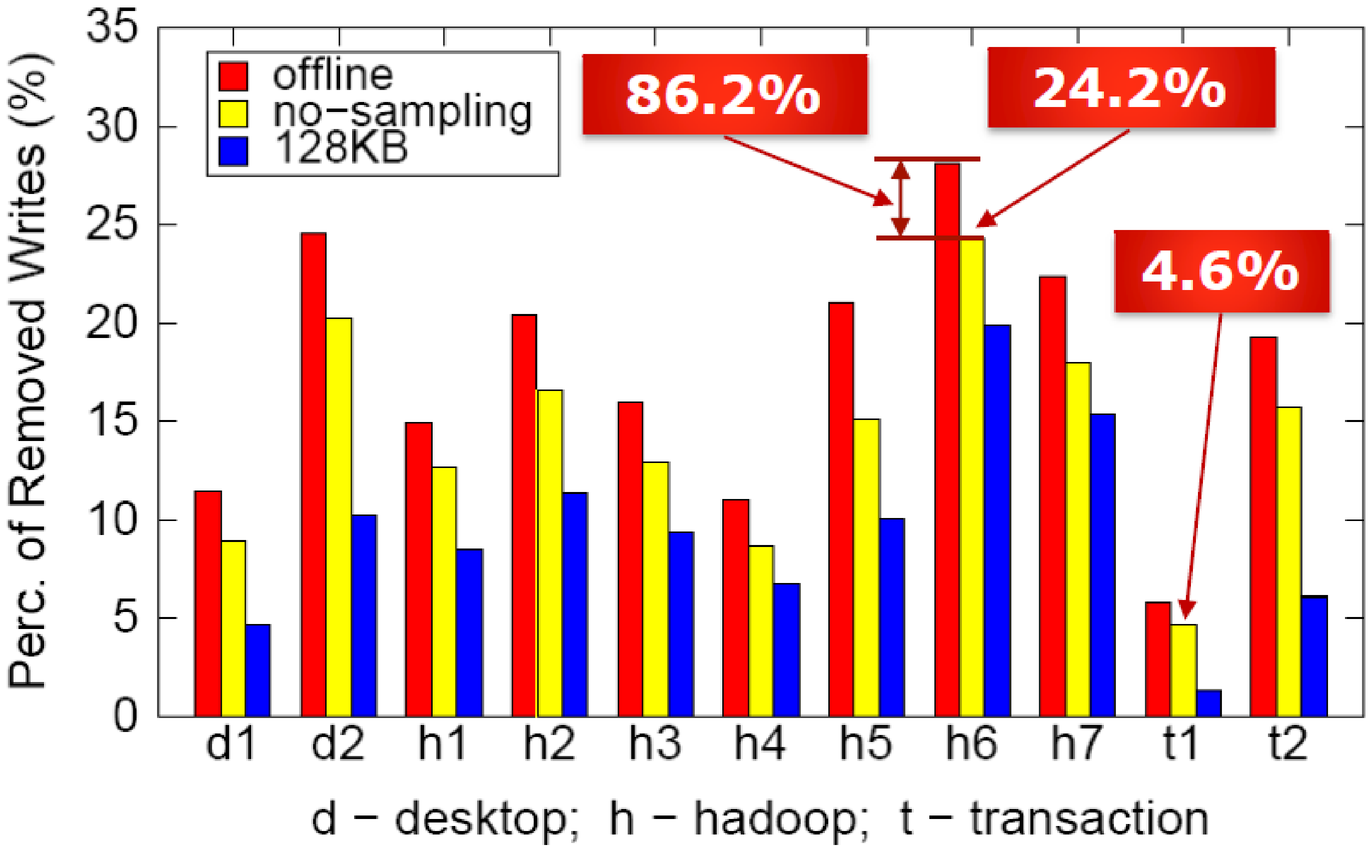
Description	Latency
Flash Read	25µs
Flash write	200µs
Flash Erase	1.5ms
SHA-1 hashing	47,548 cycles
CRC32 hashing	4,120 cycles

Workloads and Trace Collection

- Desktop (d1, d2)
  - Typical office workloads
  - Irregular idle intervals and small reads/writes
- Hadoop (h1-h7)
  - TPC-H data warehouse queries were executed on a Hadoop distributed system platform
  - Intensive large write of temp data
- Transaction (t1, t2)
  - TPC-C workloads were executed for transaction processing
  - Intensive write operations

Effectiveness of Deduplication

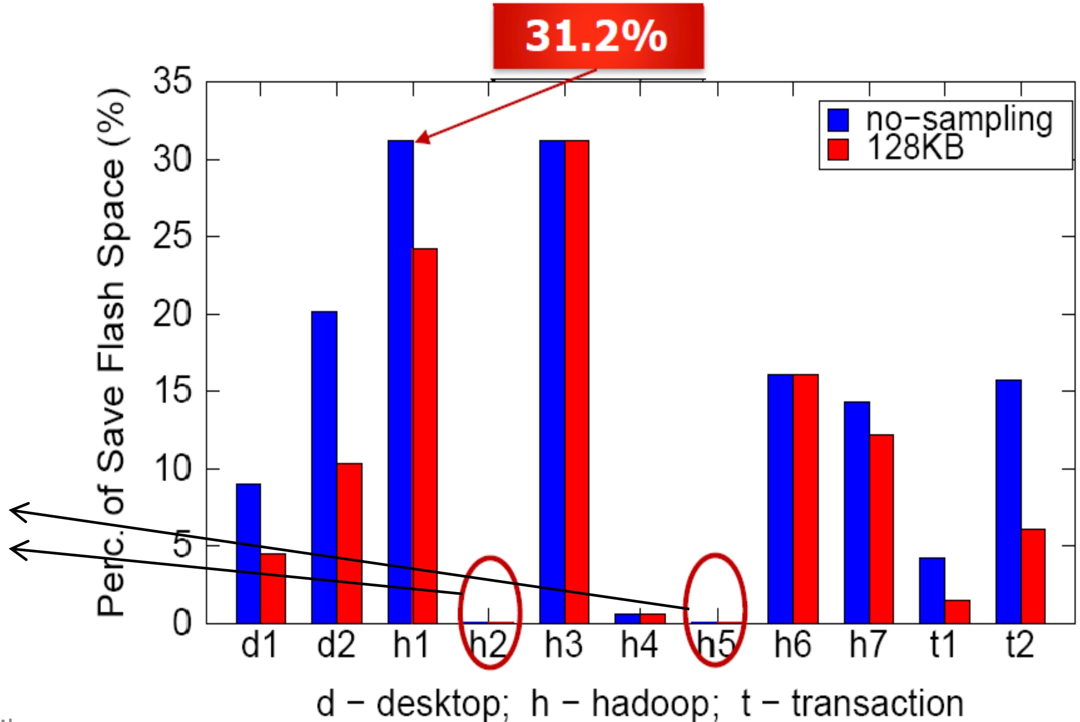
- Removing duplicate writes



Effectiveness of Deduplication

- Extending flash space
  - Space saving rate : (n-m) / n
    - n–total # of occupied blocks of flash memory w/o CAFTL
    - m–total # of occupied blocks of flash memory w/ CAFTL

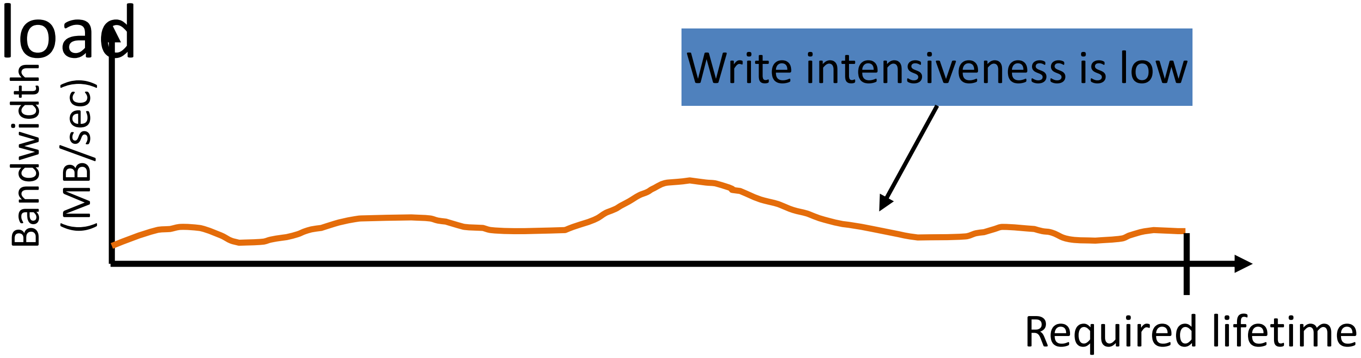
Smaller workloads

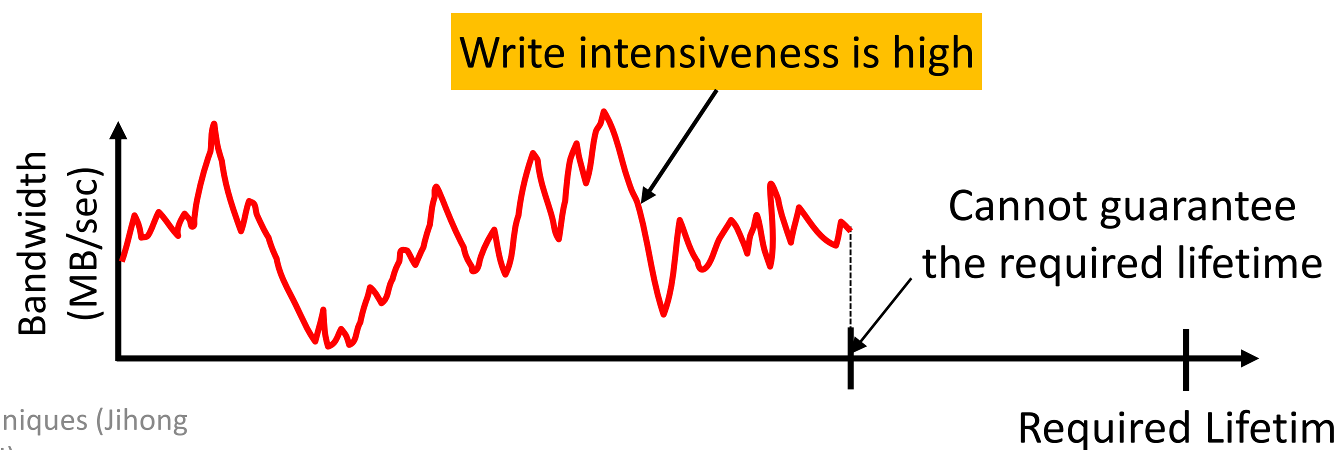


Dynamic Throttling- READY

Unpredictable Lifetime

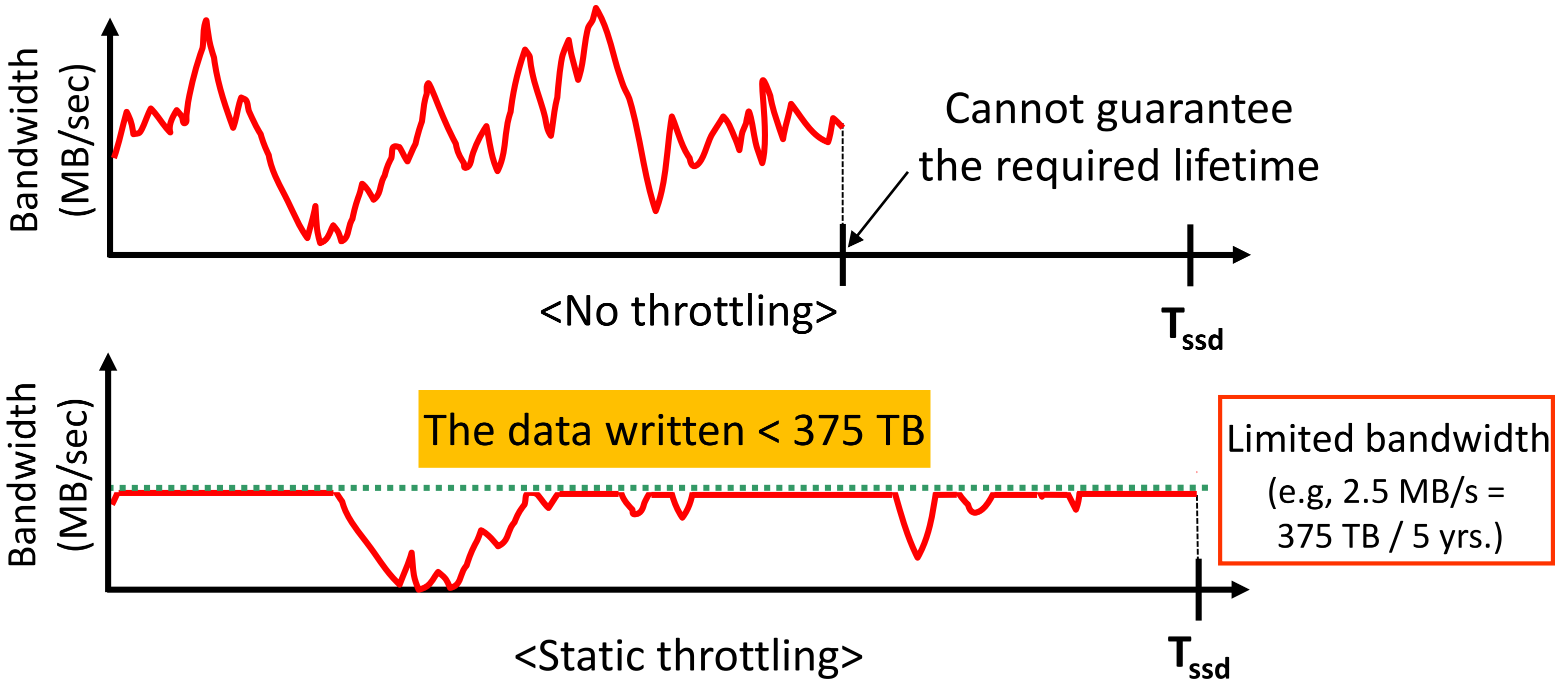
- The lifetime of SSDs strongly fluctuates depending on the write intensiveness of a given workload





Lifetime Guarantee Using Static Throttling

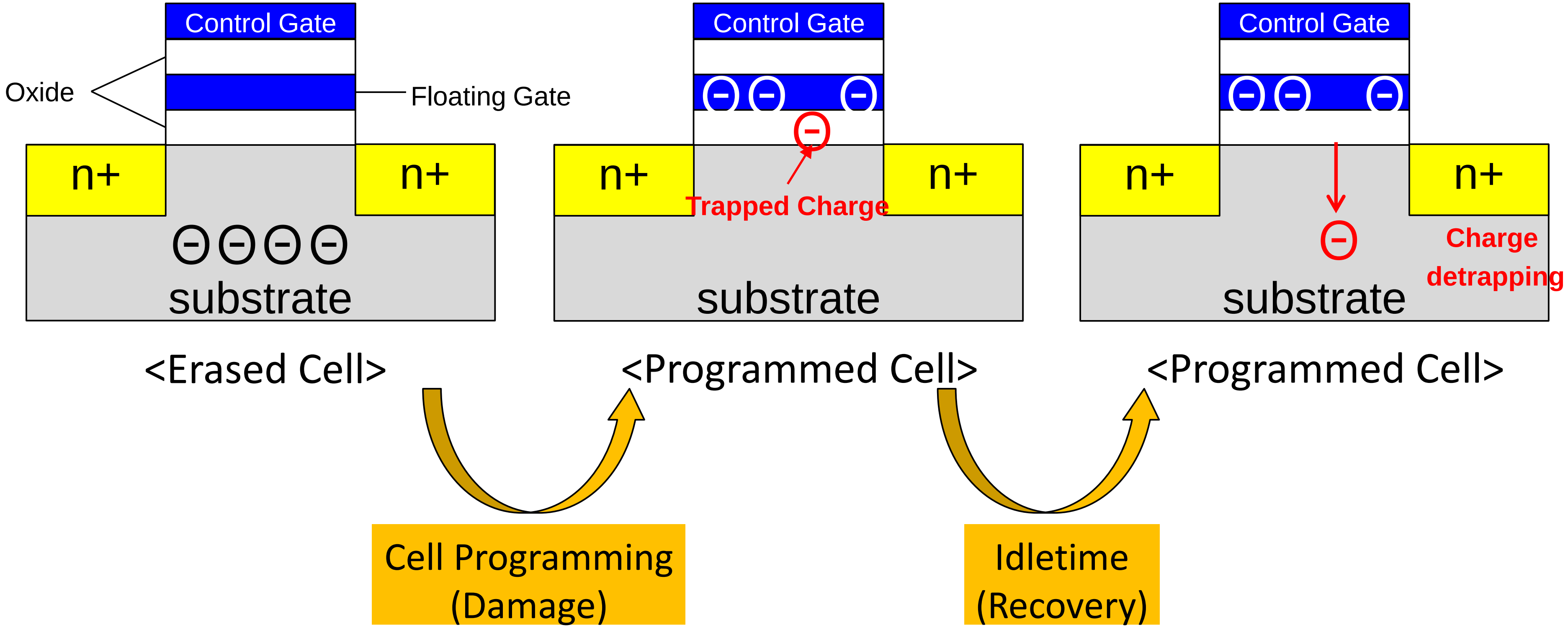
- To guarantee the SSD lifetime, some SSD vendors start to adopt a static throttling technique



- Static throttling is likely to underutilize the endurance of SSDs, incurring performance degradation

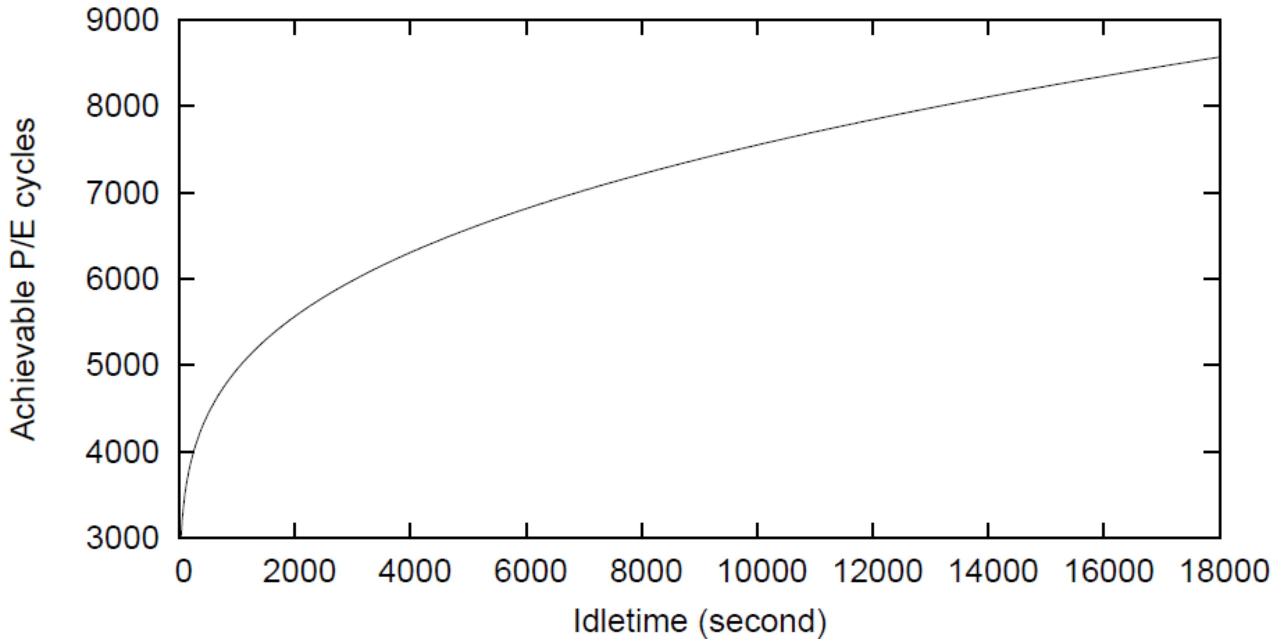
Underutilize the Endurance of SSDs

- Self Recovery Effect of Memory Cell
  - Repetitive P/E cycles cause damage to memory cells
  - The damage of cells can be partially recovered during the idle time between two consecutive P/E cycles



Effective P/E Cycles

- The effective number of P/E cycles is much higher than P/E cycles denoted by datasheets
- Example: 20nm 2-bit MLC flash memory with 3K P/E cycles

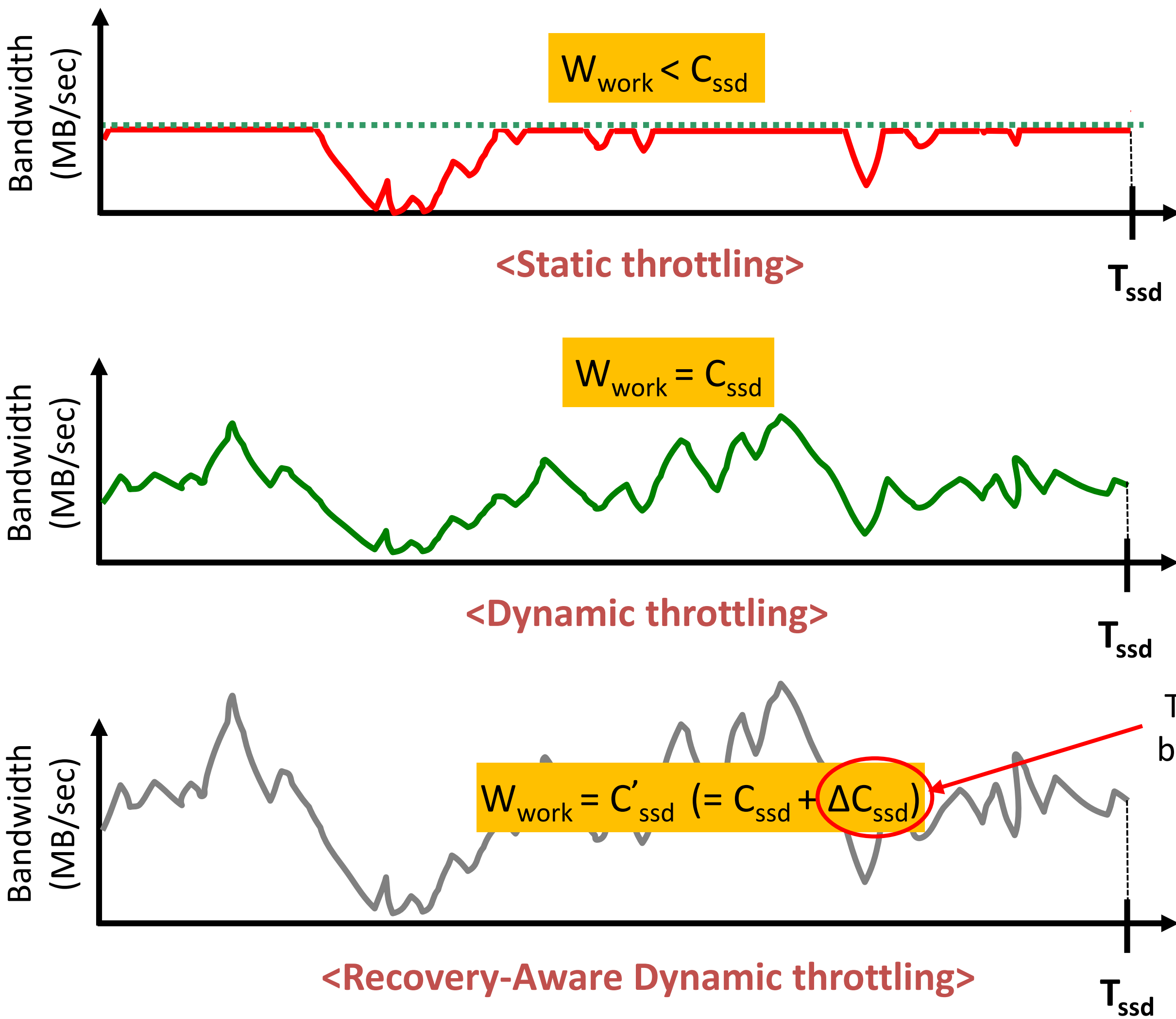


- The endurance can be improved if the self-recovery is exploited in throttling write traffic…

REcovery-Aware DYnamic throttling (READY)

- Guarantee lifetime of SSDs by
  - Throttling SSD performance depending on the write demands of a workload
  - Exploiting the self-recovery effect of memory cells, which improves the effective P/E cycles

Benefit of READY



The improved endurance by the self-recovery effect

C_{ssd}: the number of writable bytes to SSDs (e.g., 375 TB)

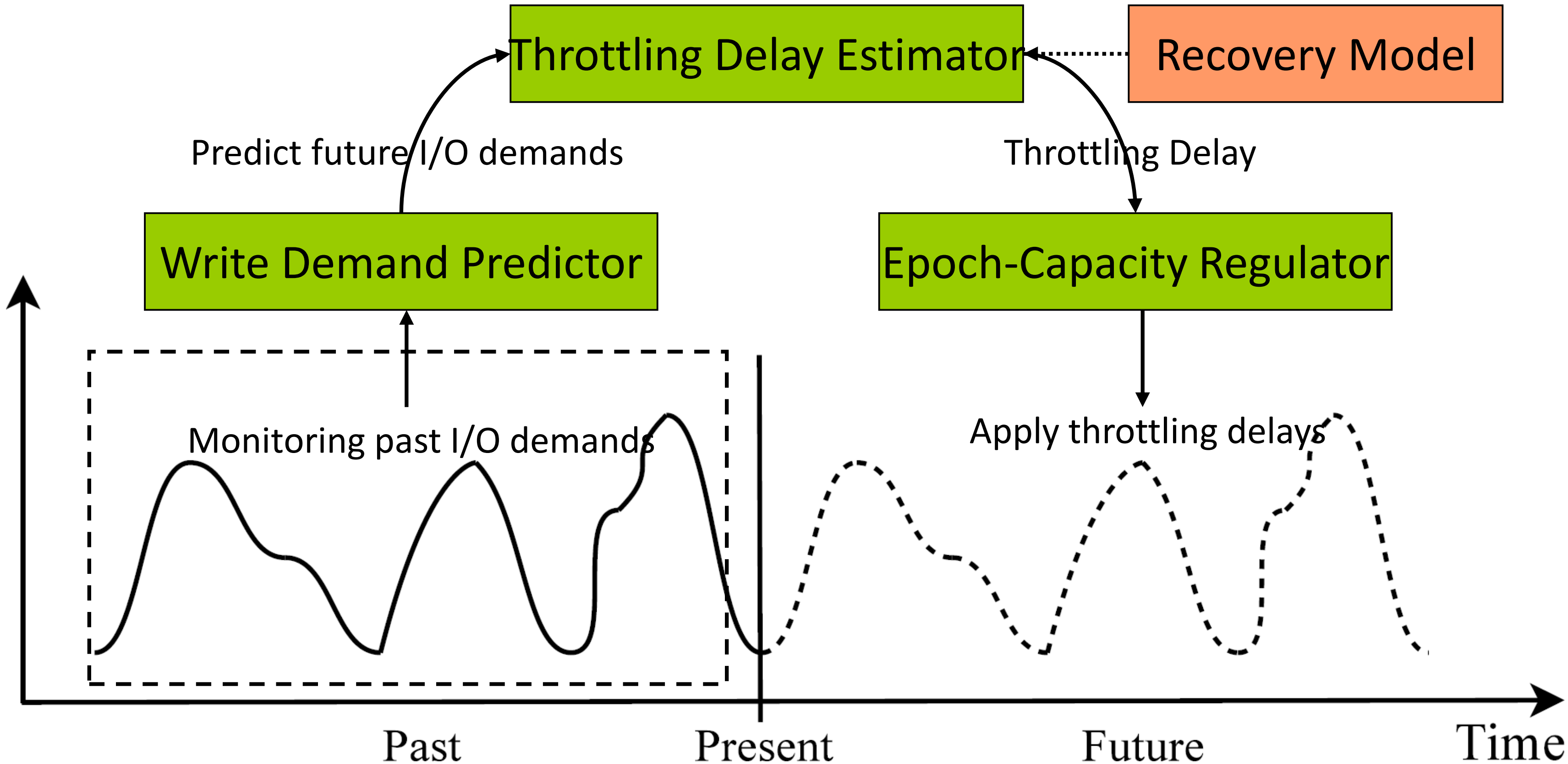
W_{work}: the number of bytes written by a workload

T_{ssd}: the target SSD lifetime (e.g., 5 years)

Design Goals of READY

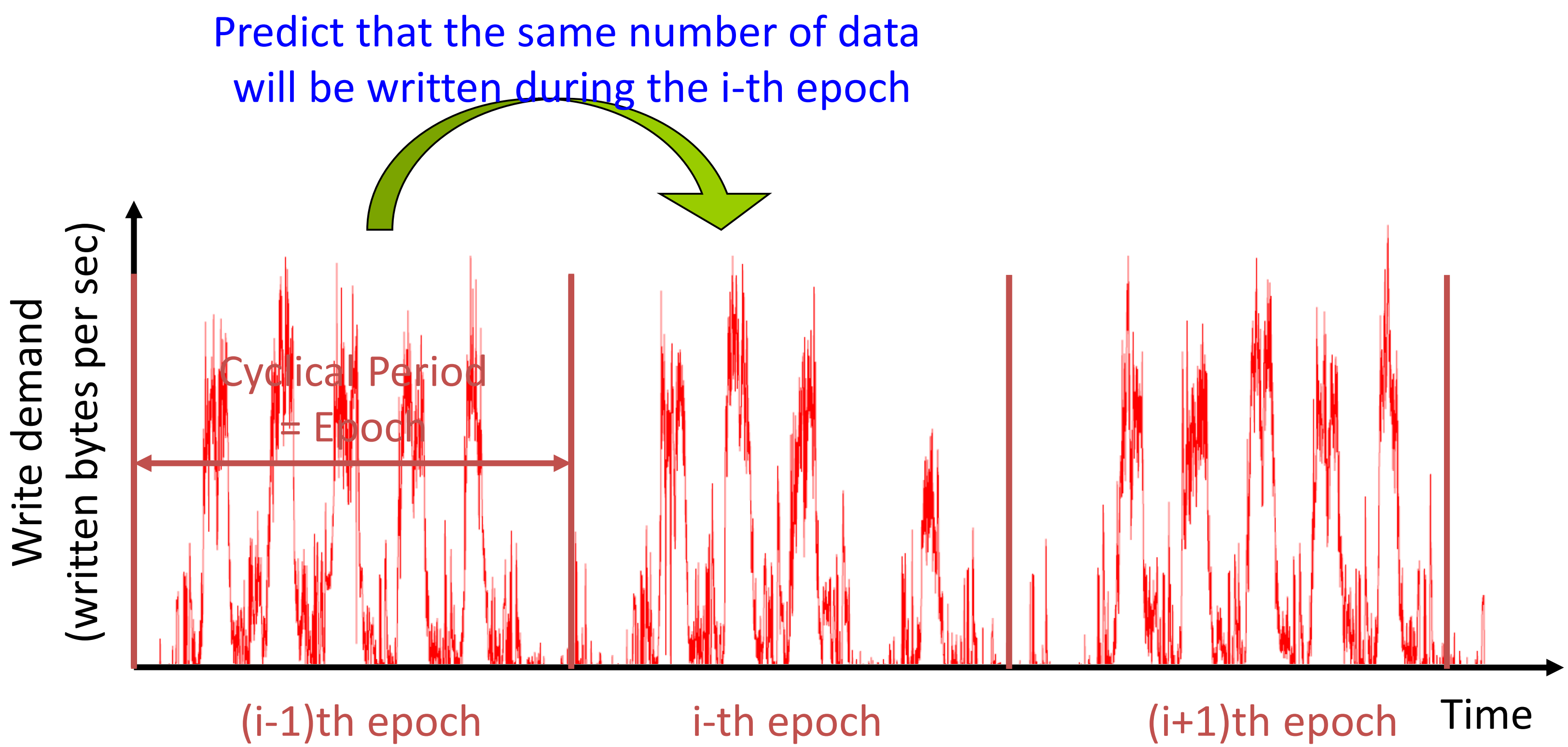
- Design goal 1: minimize average response times
  - Determine a throttling delay as low as possible so that the SSD is completely worn out at the required lifetime
- Design goal 2: minimize response time variations
  - Distribute a throttling delay as evenly as possible over every write request

Overall Architecture of READY



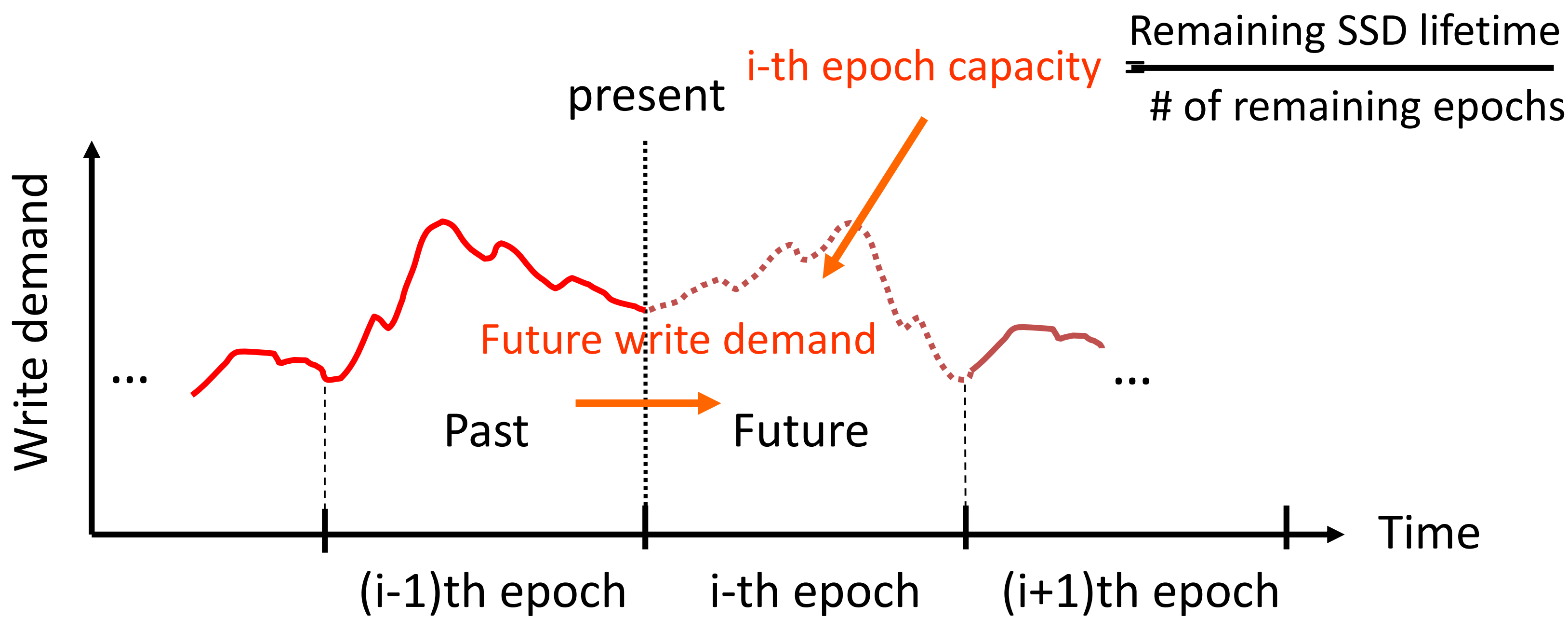
Write Demand Predictor

- Write demand predictor exploits cyclical behaviors of enterprise workloads to predict future write demands



Throttling Delay Estimator

- Decide a throttling delay so that the data written during the next epoch is properly throttled
- Calculate a throttling delay by using the predicted write demand and the remaining lifetime

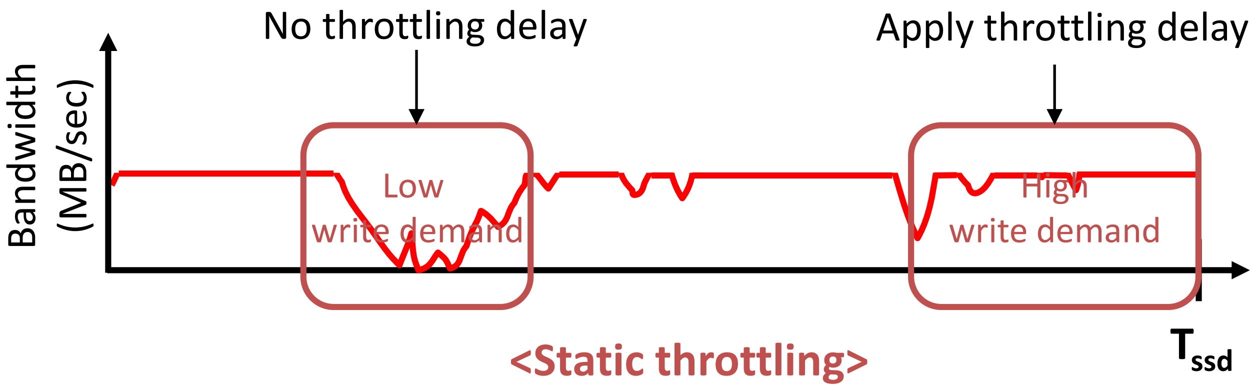


Change Throttling Delay

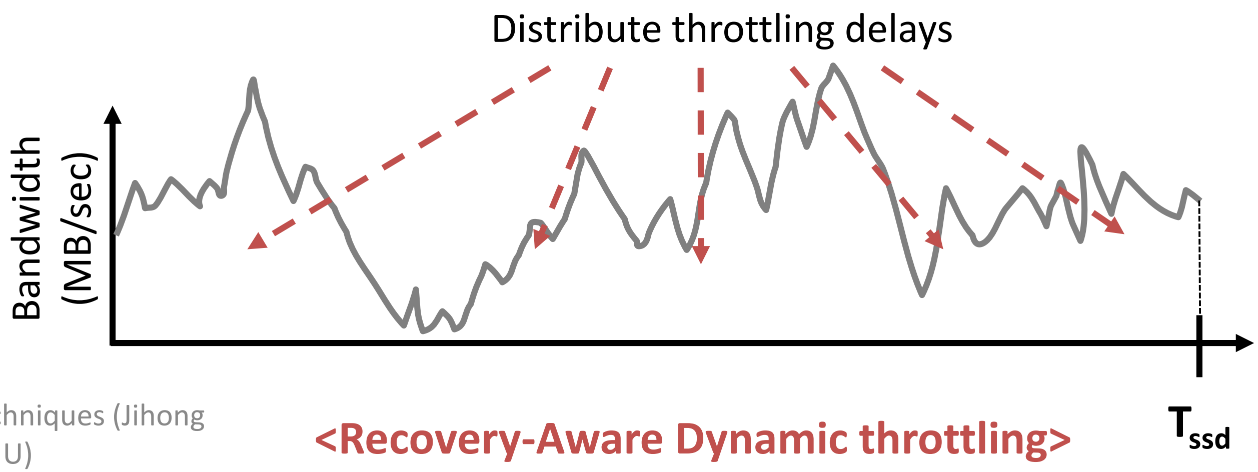
- Case 1: predicted write demand = epoch capacity
  – Don’t change a throttling delay
- Case 2: predicted write demand > epoch capacity
  – Increase a throttling delay to reduce the number of data written
- Case 3: predicted write demand < epoch capacity
  – Decrease a throttling delay to increase the number of data written

Epoch-Capacity Regulator

- Distribute a throttling delay to every page write evenly
  – This is beneficial in minimizing response time variations



<Static throttling>



<Recovery-Aware Dynamic throttling>

Experimental Setting

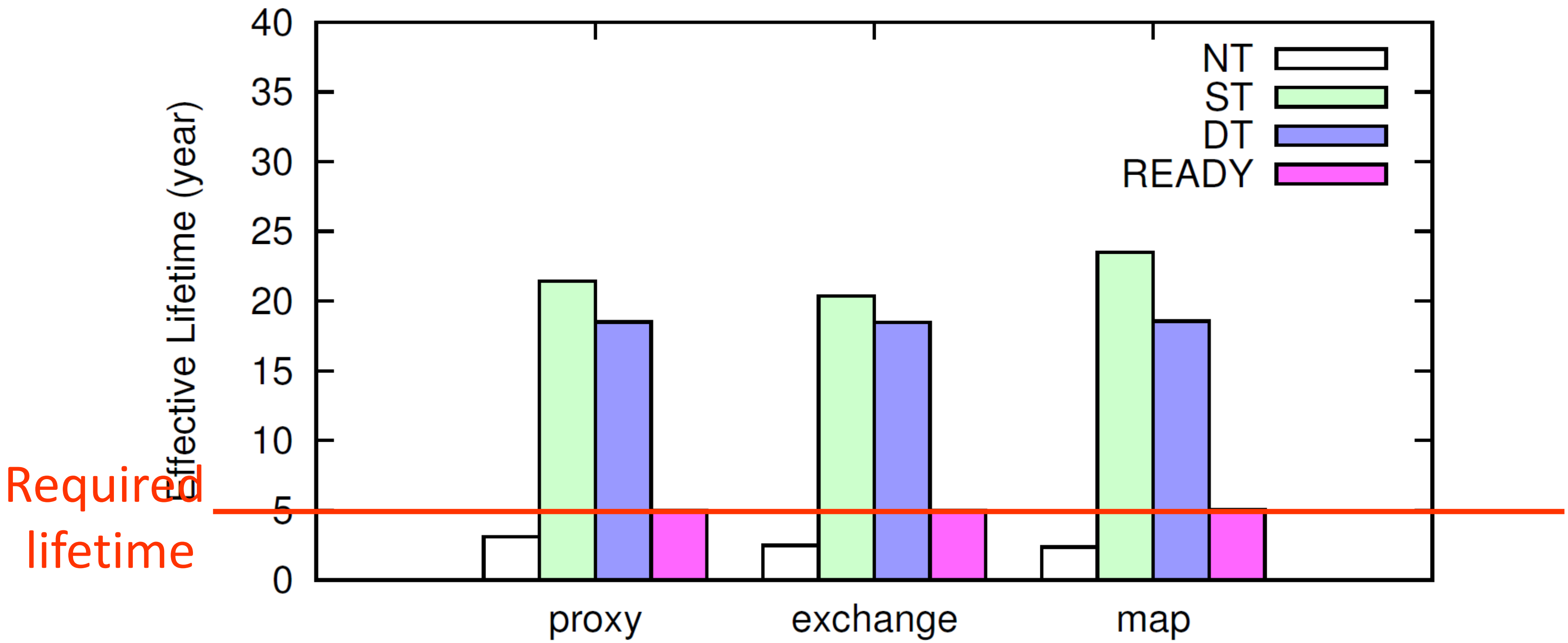
- Use the DiskSim-based SSD simulator for evaluations
  - 20 nm 2-bit MLC NAND flash memory with 3K P/E cycles
  - The target SSD lifetime is set to 5 years
- Evaluated SSD configurations

NT	No Throttling
ST	Static Throttling
DT	Dynamic Throttling
READY	Recovery-Aware Dynamic Throttling

- Benchmarks

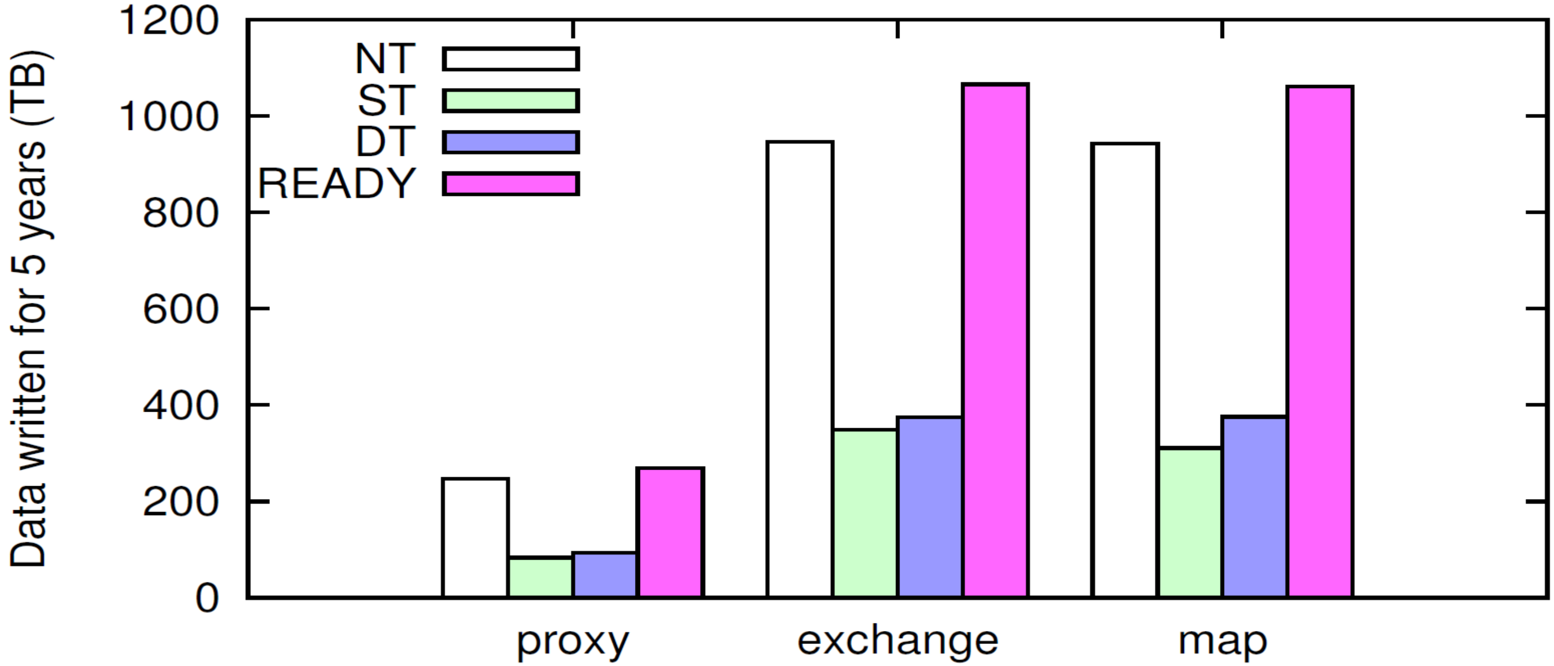
Trace	Duration	Data written per hour (GB)	WAF	SSD capacity (GB)
Proxy	1 week	4.94	1.62	32
Exchange	1 day	20.61	2.24	128
map	1 day	23.82	1.68	128

Lifetime Analysis



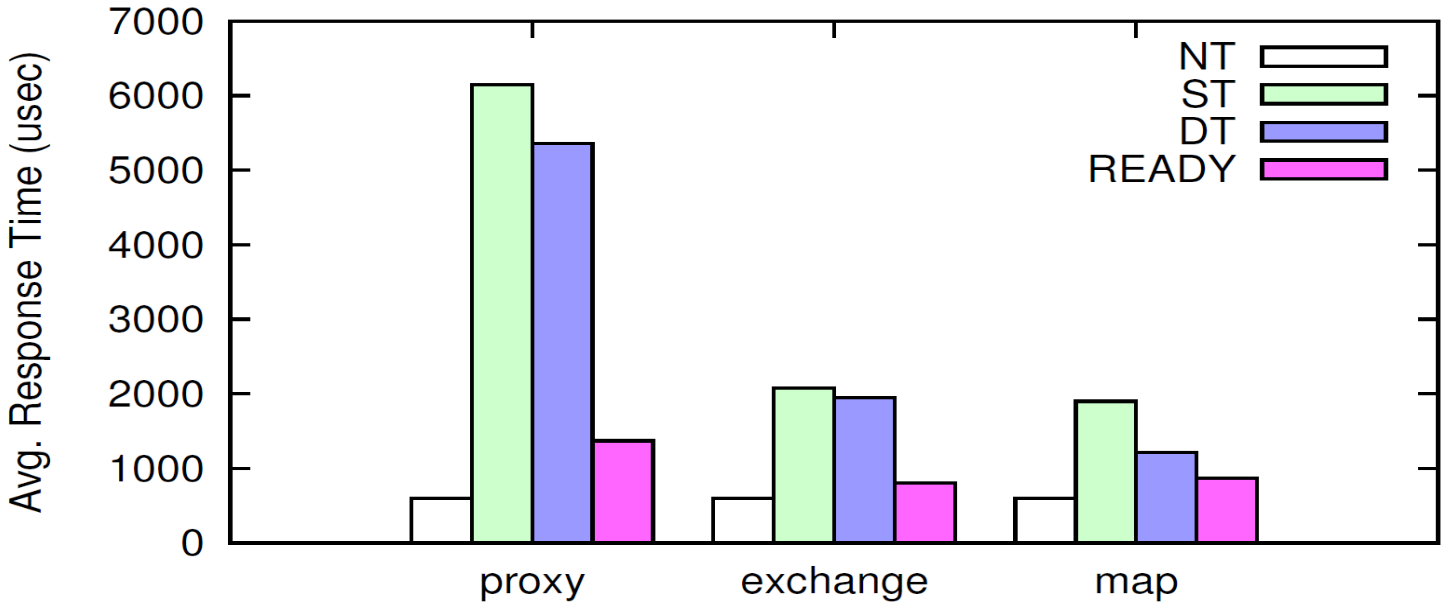
- NT cannot guarantee the required SSD lifetime
- READY achieves the lifetime close to 5 years
- ST and DT exhibit the lifetime much longer than 5 years

Data Written to SSD during 5 years



- ST and DT uselessly throttles write performance even through they can write more data to the SSD
- READY exhibits 10% higher endurance than NT because of the increased recovery time

Performance Analysis



- NT exhibits the best performance among all the configurations
- READY performs better than ST and DT while guaranteeing the required lifetime

References

- **박지훈, 김지홍, "BlueZIP : 고성능 솔리드 스테이트 드라이브를 위한 압축 모듈," 대한임베디드공학회 추계학술대회**, 2010.
- F. Chen et. al., "CAFTL: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," In Proceedings of FAST '11, 2011.
- S. Lee et. al., "Lifetime management of flash-based SSDs using recovery-aware dynamic throttling,"
  In Proceedings of FAST '12, 2012.