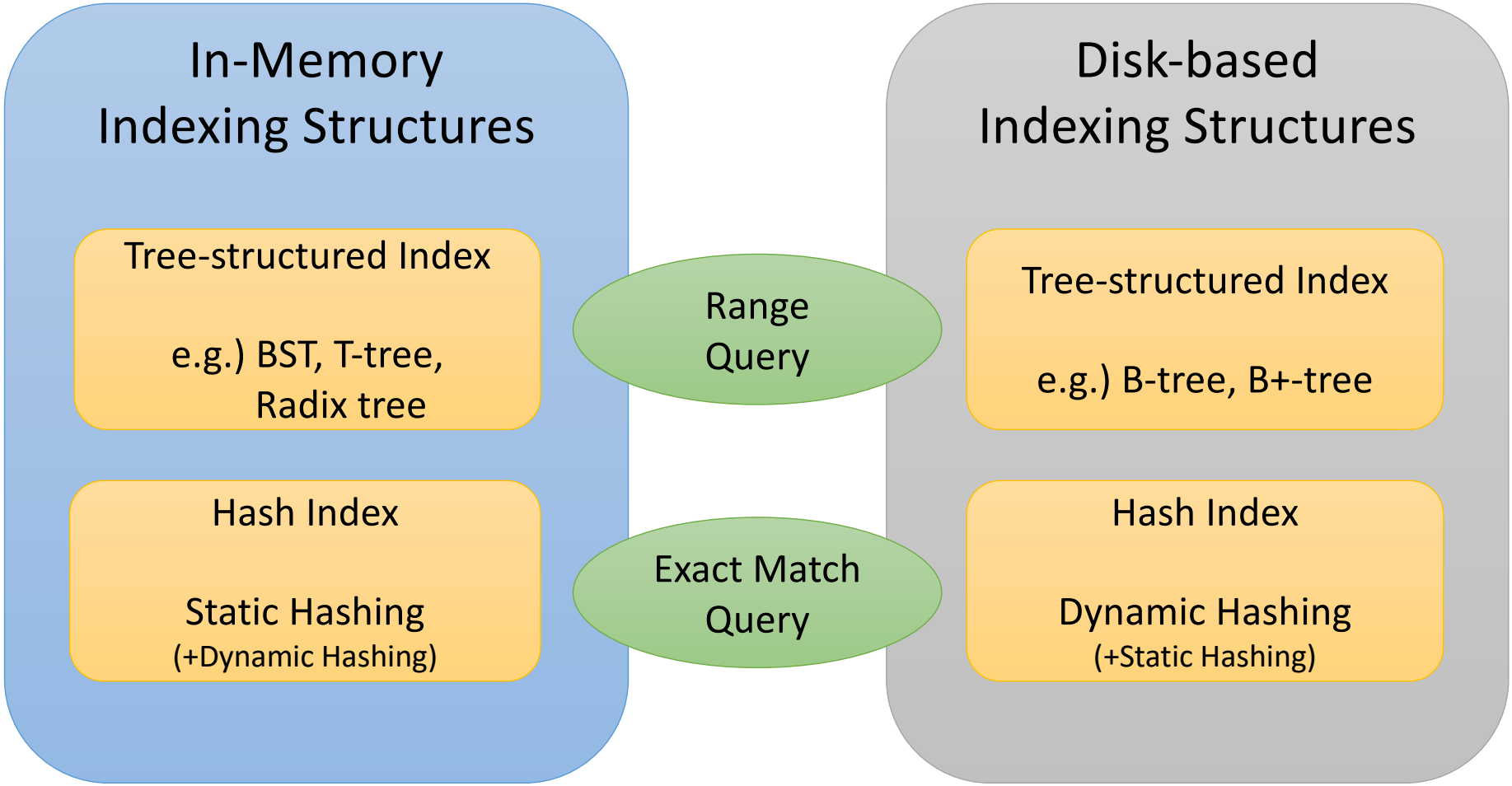
FISS 2019

Byte-Addressable Indexing for Persistent Memory

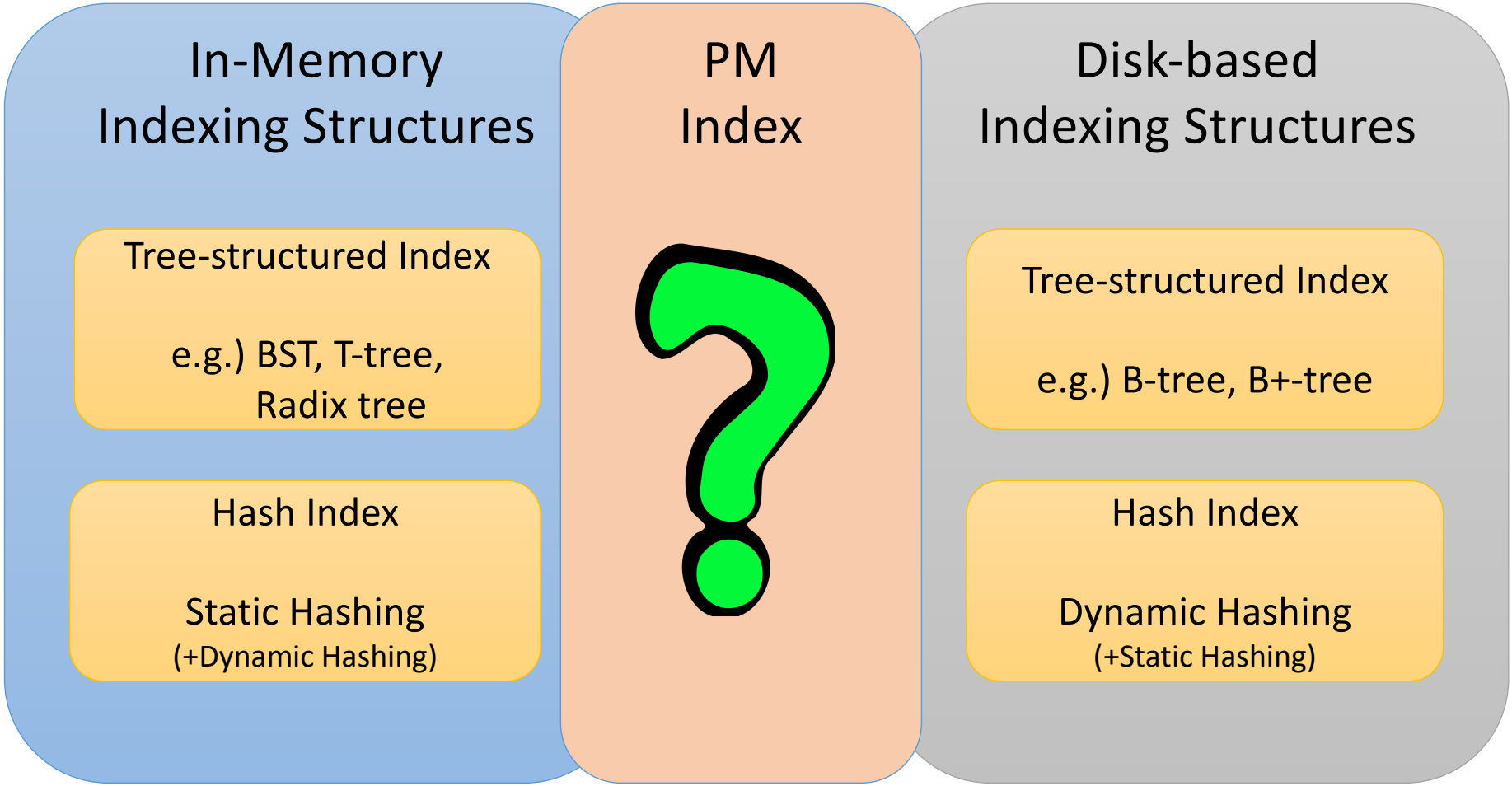
Beomseok Nam

bnam@skku.edu

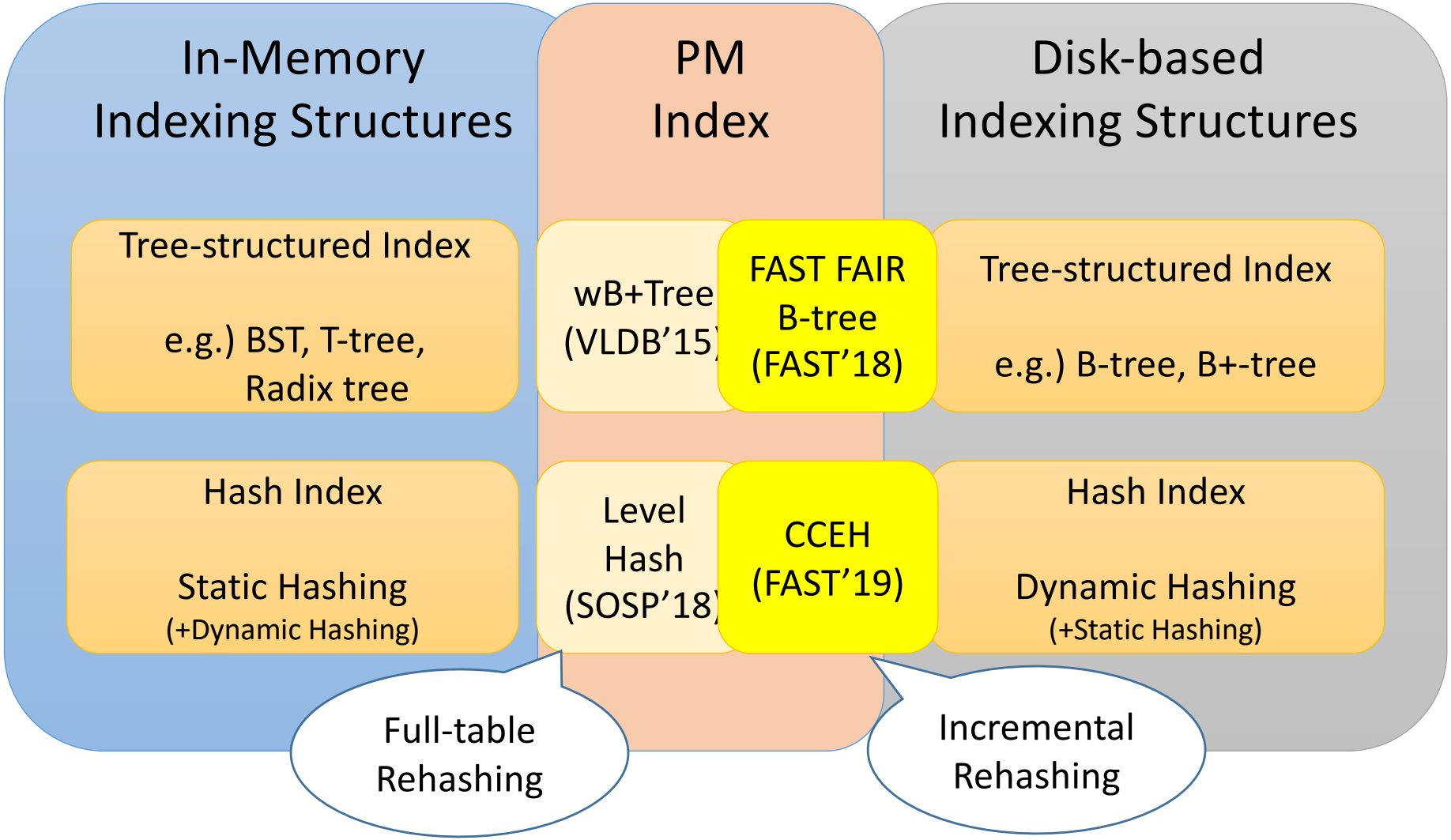
Indexing Structures



Indexing Structures

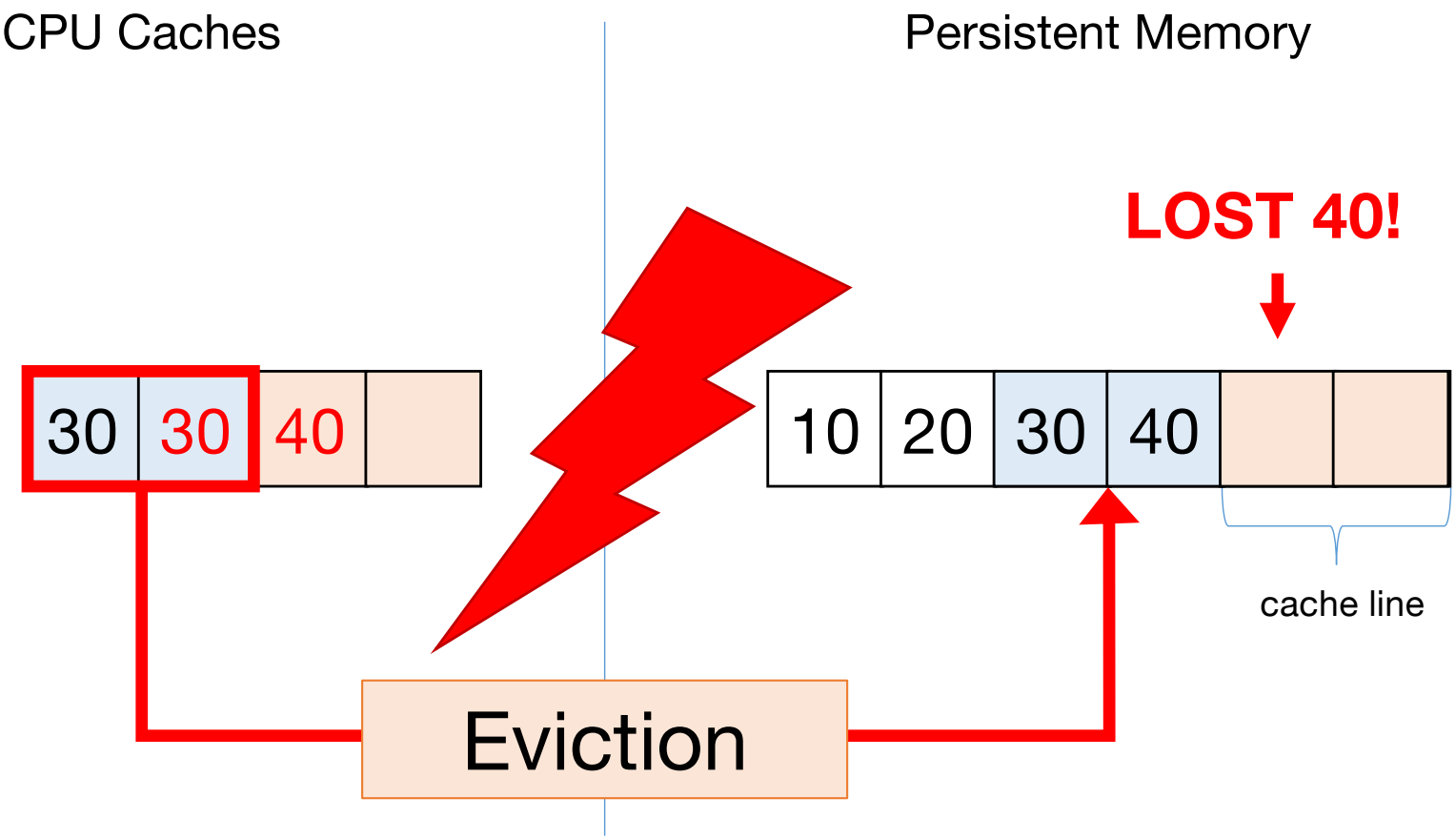


Indexing Structures



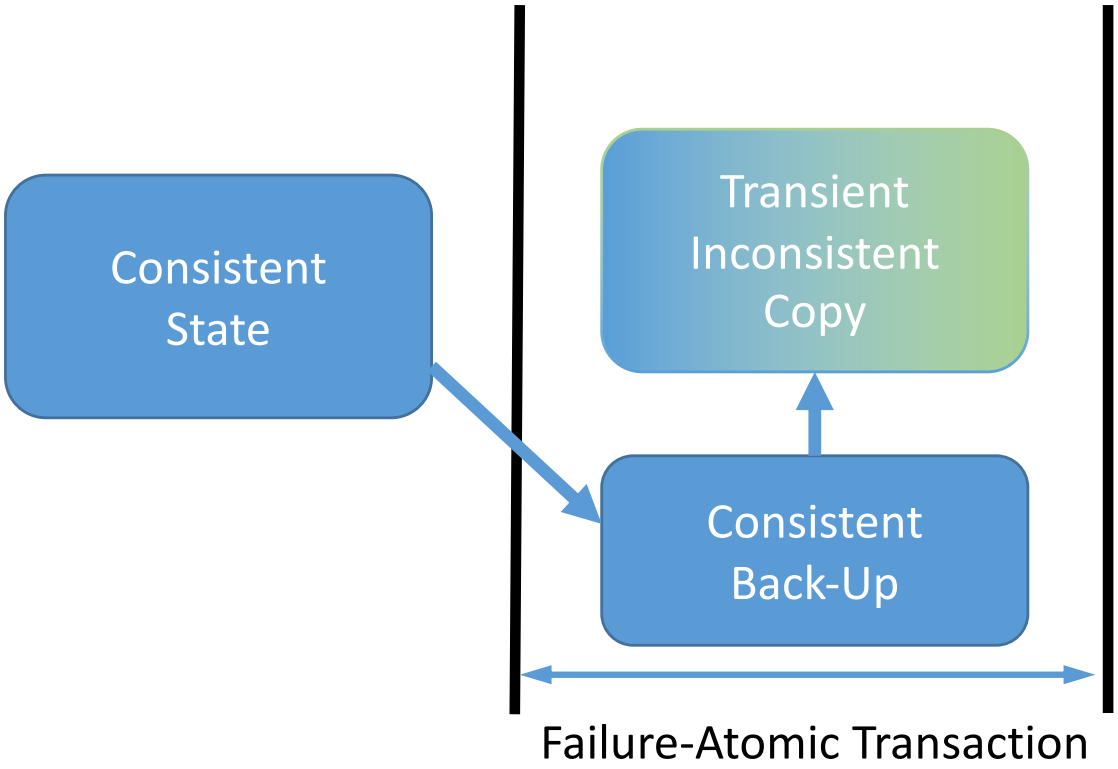
What's the challenge?

- Dirty Cacheline Eviction to PM Results in Persistent Inconsistency
- E.g. Insertion of 25 into a sorted array



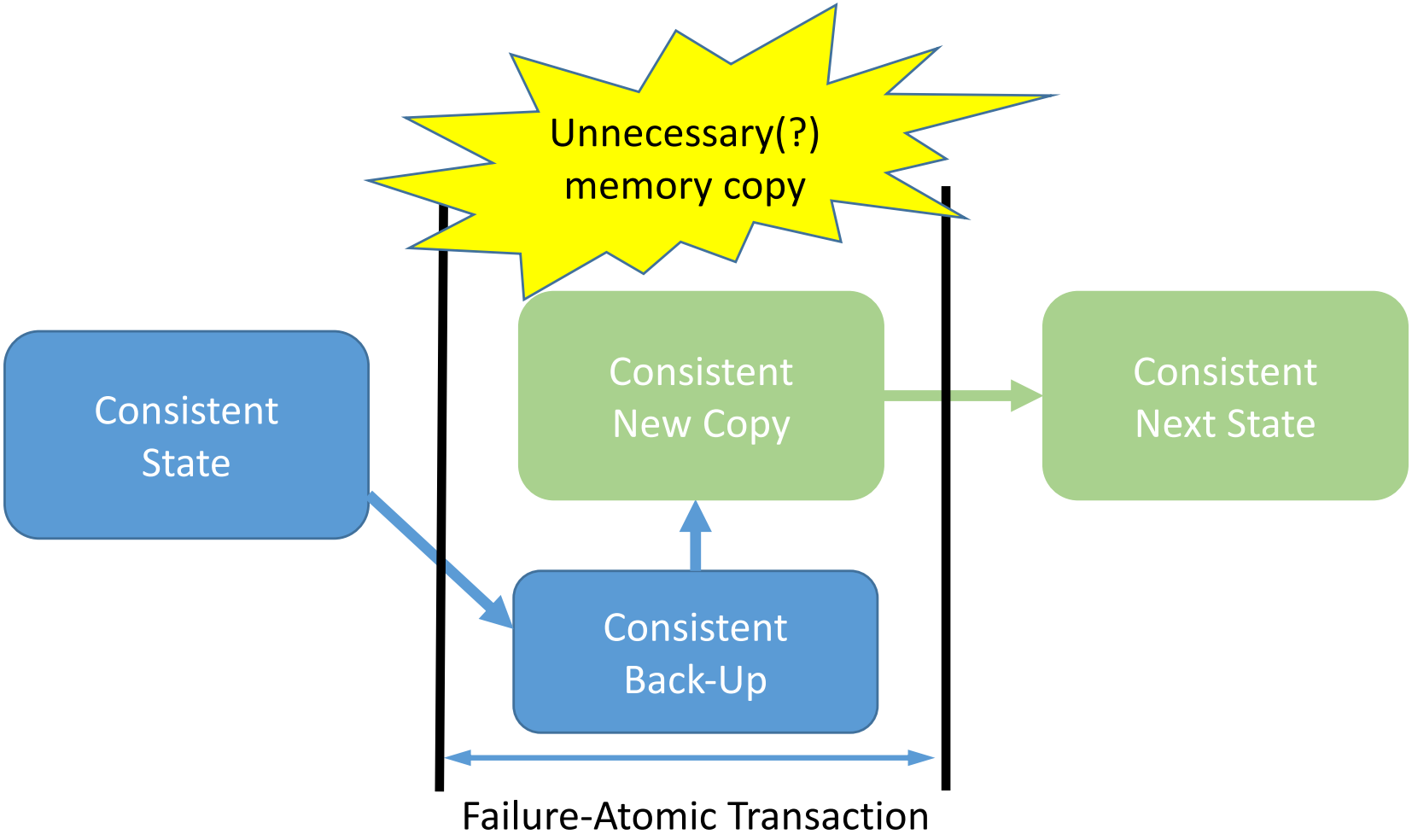
Failure-Atomicity and Transient Inconsistency

- Failure-Atomicity can be guaranteed by having redundant copies in legacy systems.



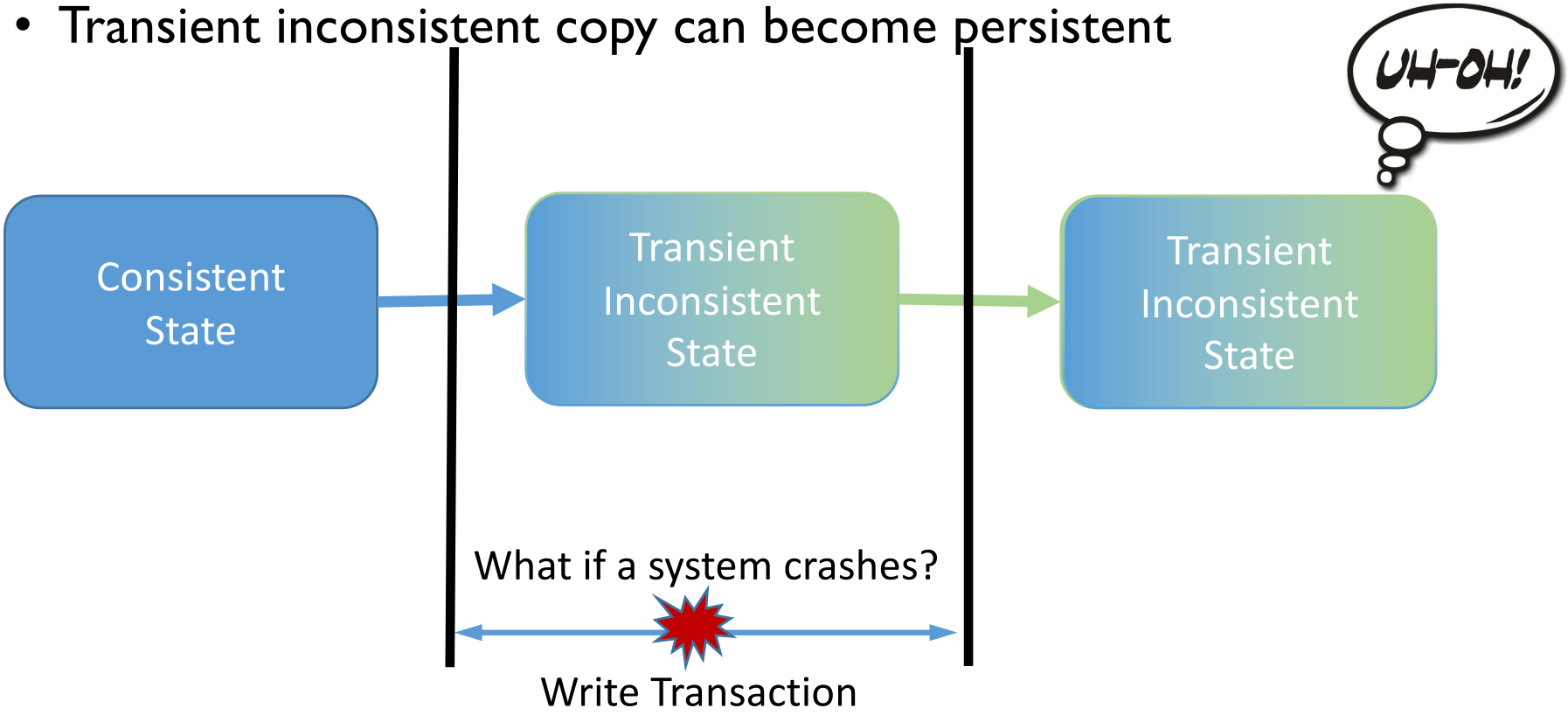
Failure-Atomicity and Transient Inconsistency

- Failure-Atomicity can be guaranteed by having redundant copies in legacy systems.



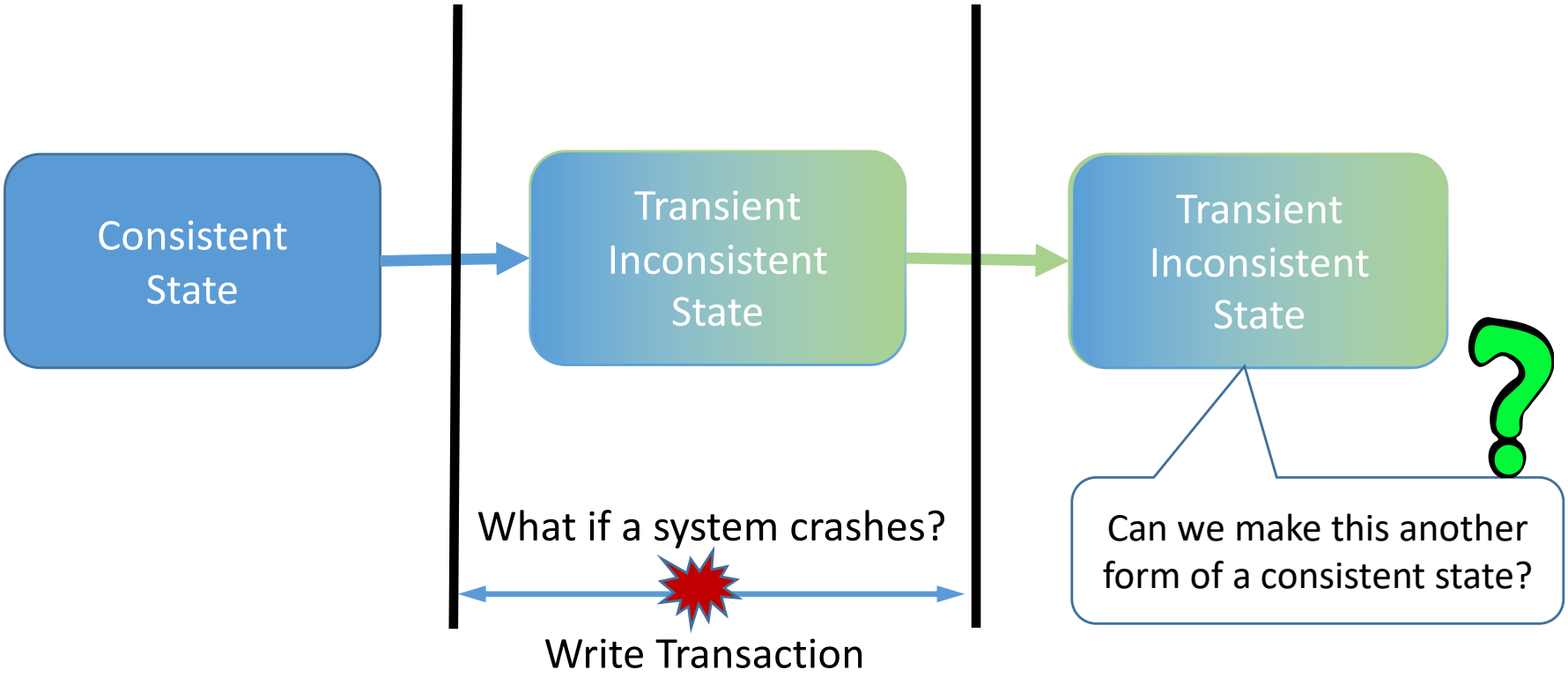
Failure-Atomicity and Transient Inconsistency

- Our motivation
  - In-place updates eliminate redundant copies and leverage byte-addressability of PM
- Challenge:
  - Transient inconsistent copy can become persistent



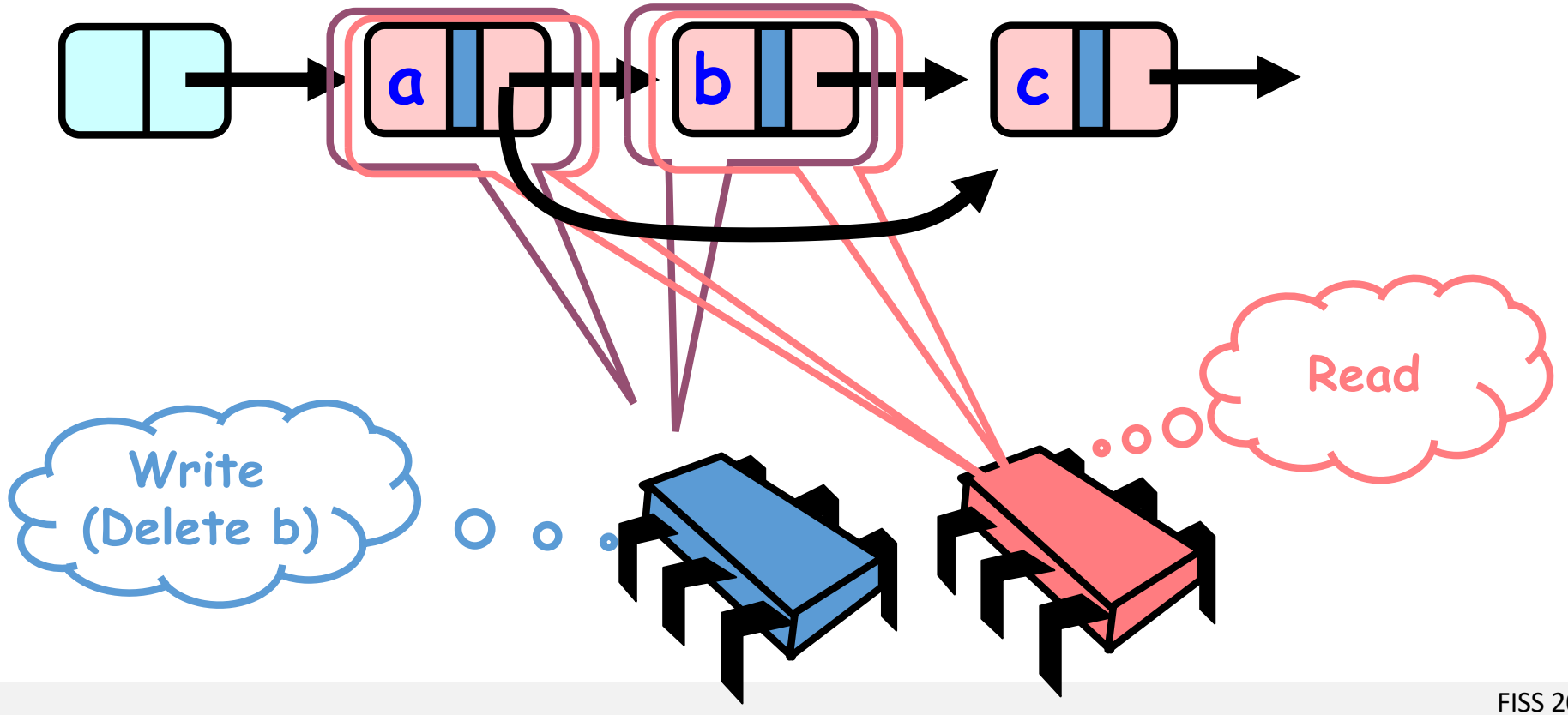
Failure-Atomicity and Transient Inconsistency

- Our approach
  - Carefully order store instructions
  - Make read transactions be aware of the order so that they can detect transient inconsistent states and tolerate inconsistency



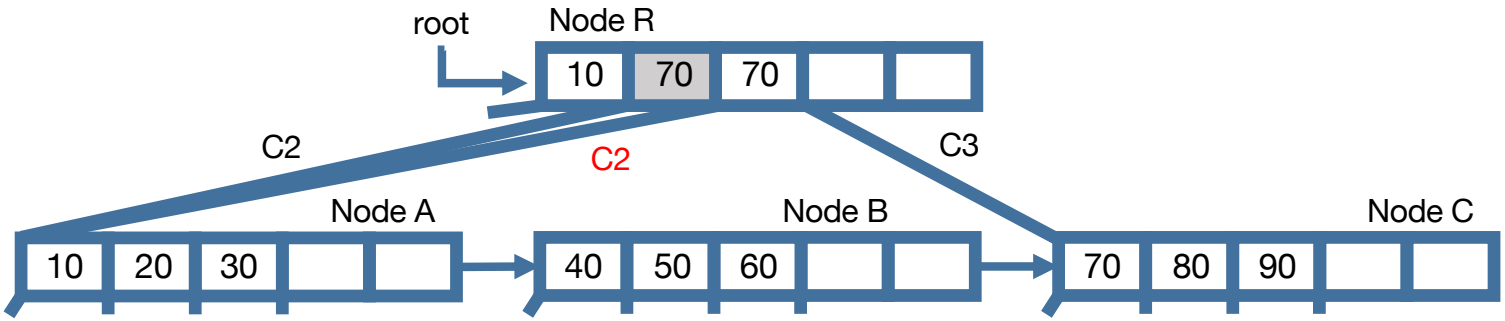
Inspired by Lock-Free Search

- Lock-Free search algorithm performs in-place updates but allows read transactions to access data structures while write transactions are making changes to them.
  - I.e., Read transactions can tolerate partially updated inconsistency

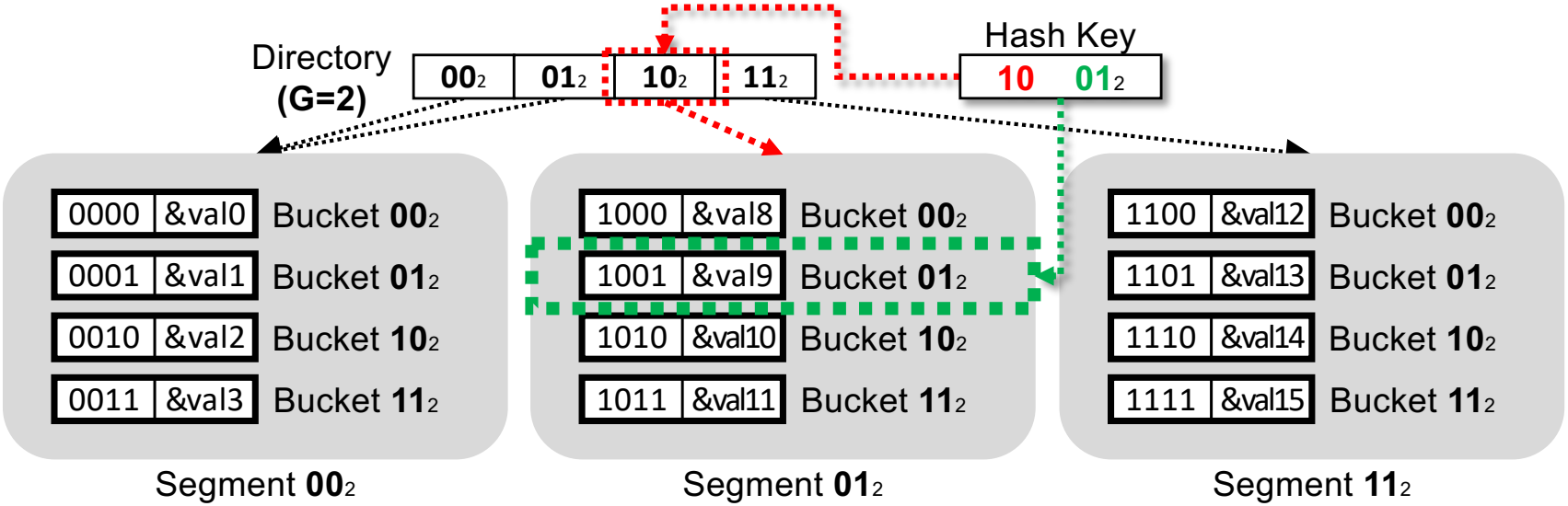


Outcomes

- FAST and FAIR B-tree (FAST '18)



- Cacheline-Conscious Extendible Hashing (FAST '19)



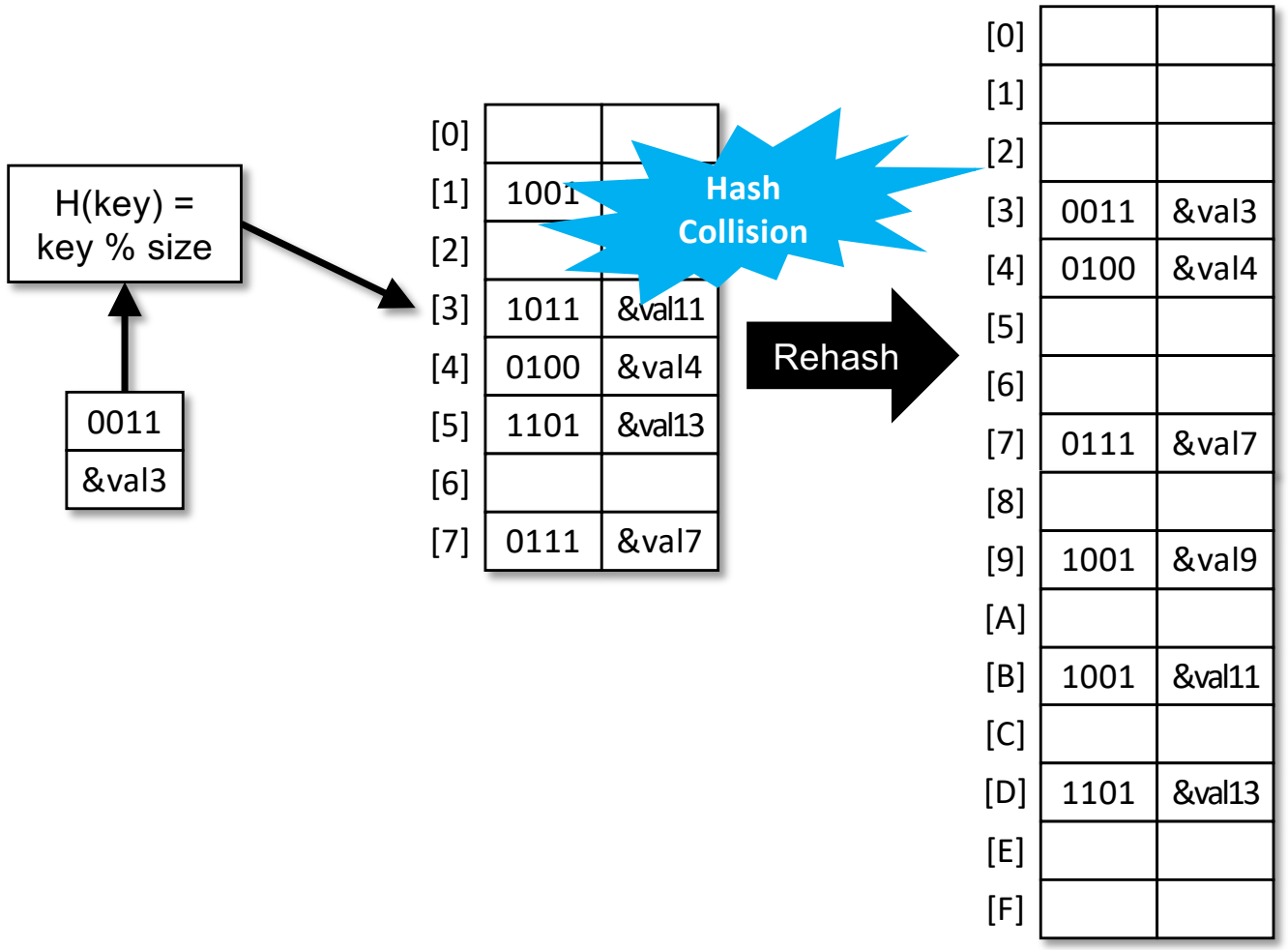
FISS 2019

Write-Optimized Dynamic Hashing for Persistent Memory
(FAST 2019)

Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, Beomseok Nam

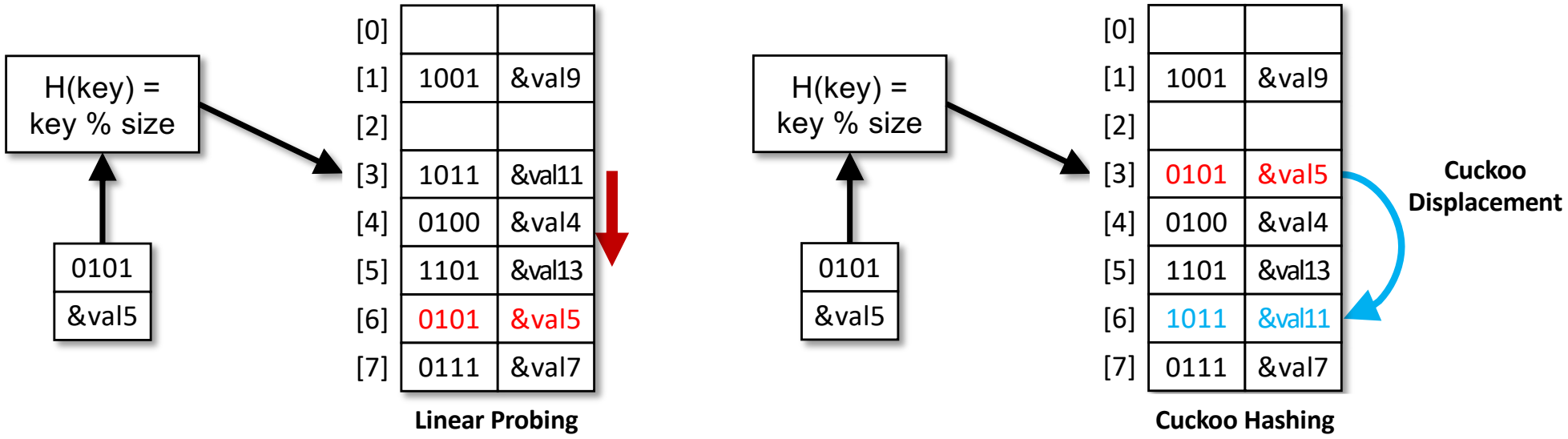
Background: Static Hashing

- Hash key collision → Full table rehashing
  - The most expensive operation in hash table



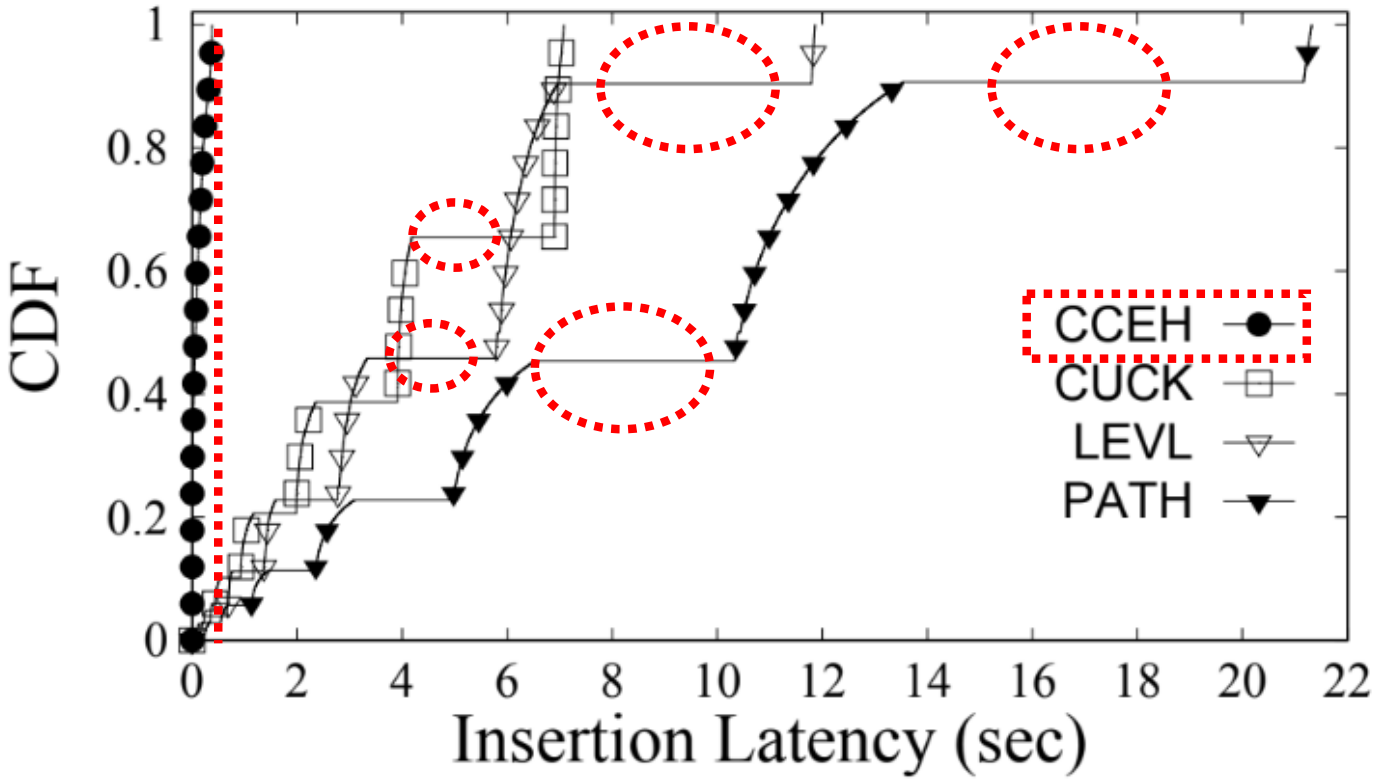
Background: Static Hashing

- To avoid full table rehashing:
  - Linear probing
  - Chaining
  - Double hashing such as Cuckoo hashing



Ad Hoc Optimizations Fail to Resolve Root Cause

- Linear probing, chaining, and cuckooing defer the rehashing problem but still suffer from expensive full-table rehashing

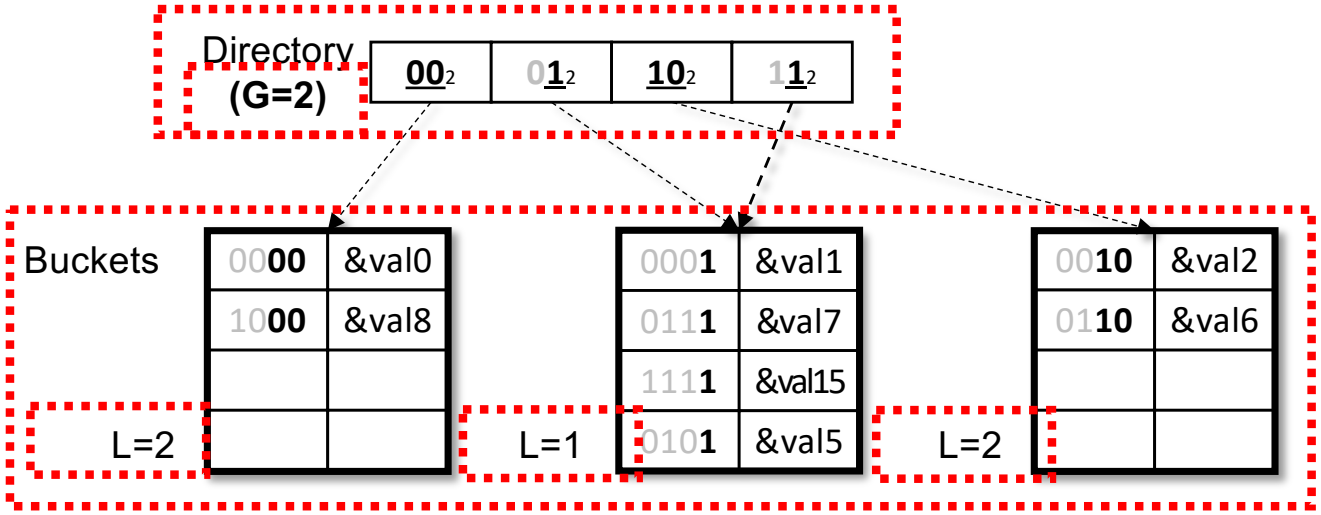


- Flat lines: Overhead of full-table rehashing

Background: Disk-based Extendible Hashing

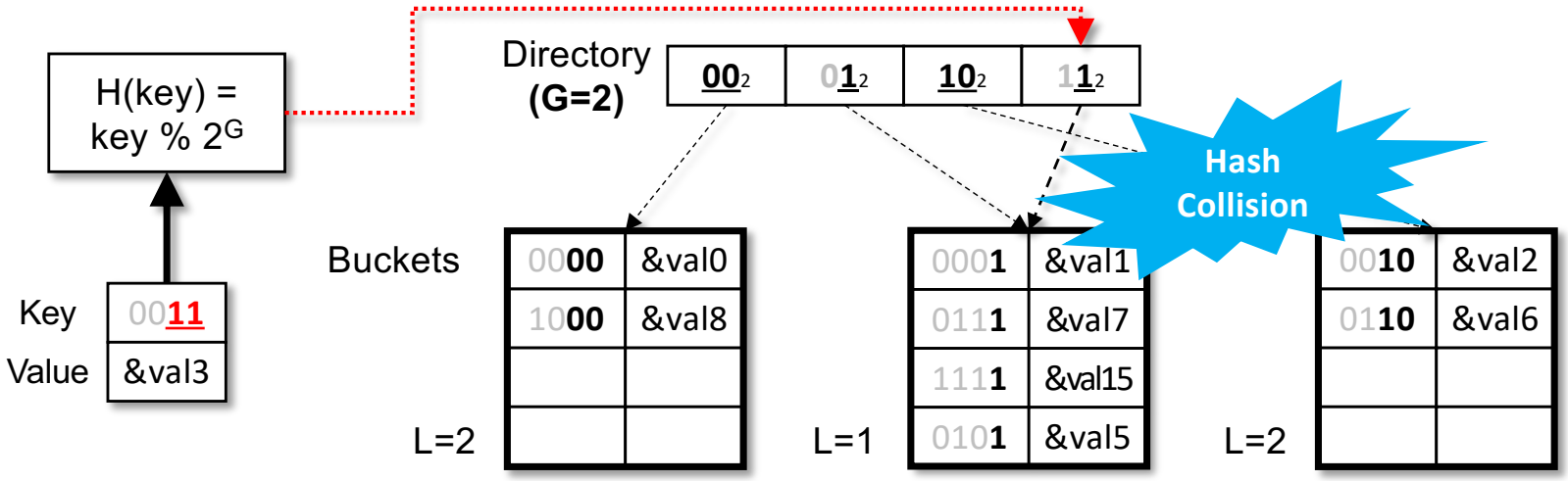
- Oracle ZFS, IBM GPFS, Redhat GFS, and GFS2 file systems
- Dynamically splits one bucket or merges two buckets at a time

Hash Function:
H(key) = key % 2^G



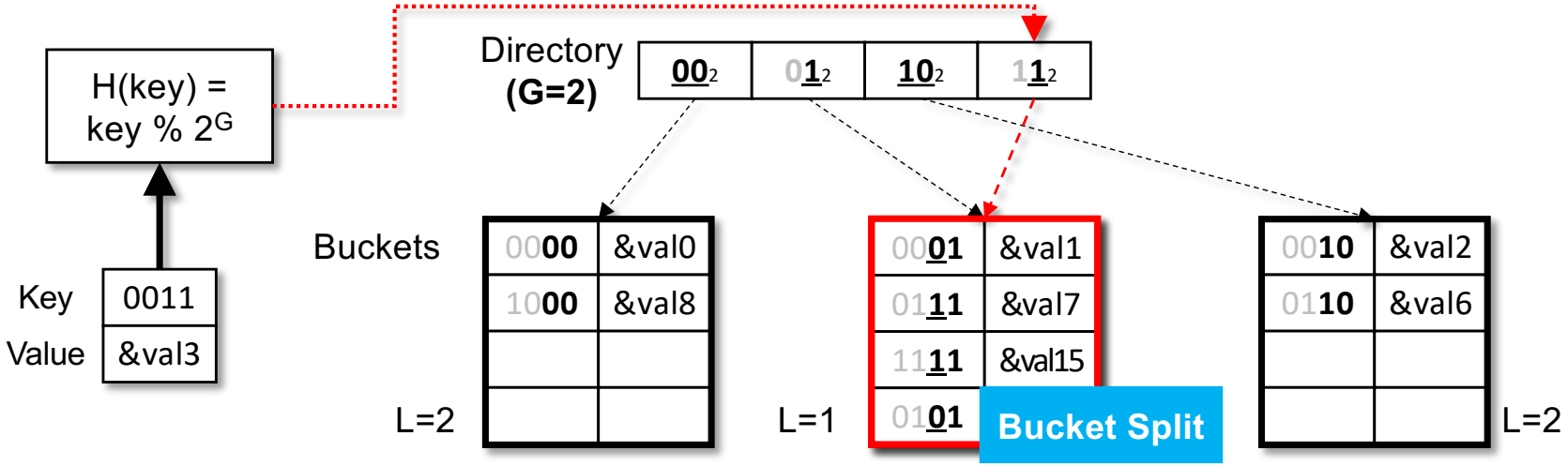
Background: Extendible Hashing – Insertion

- Look-up directory using G bits to choose a bucket



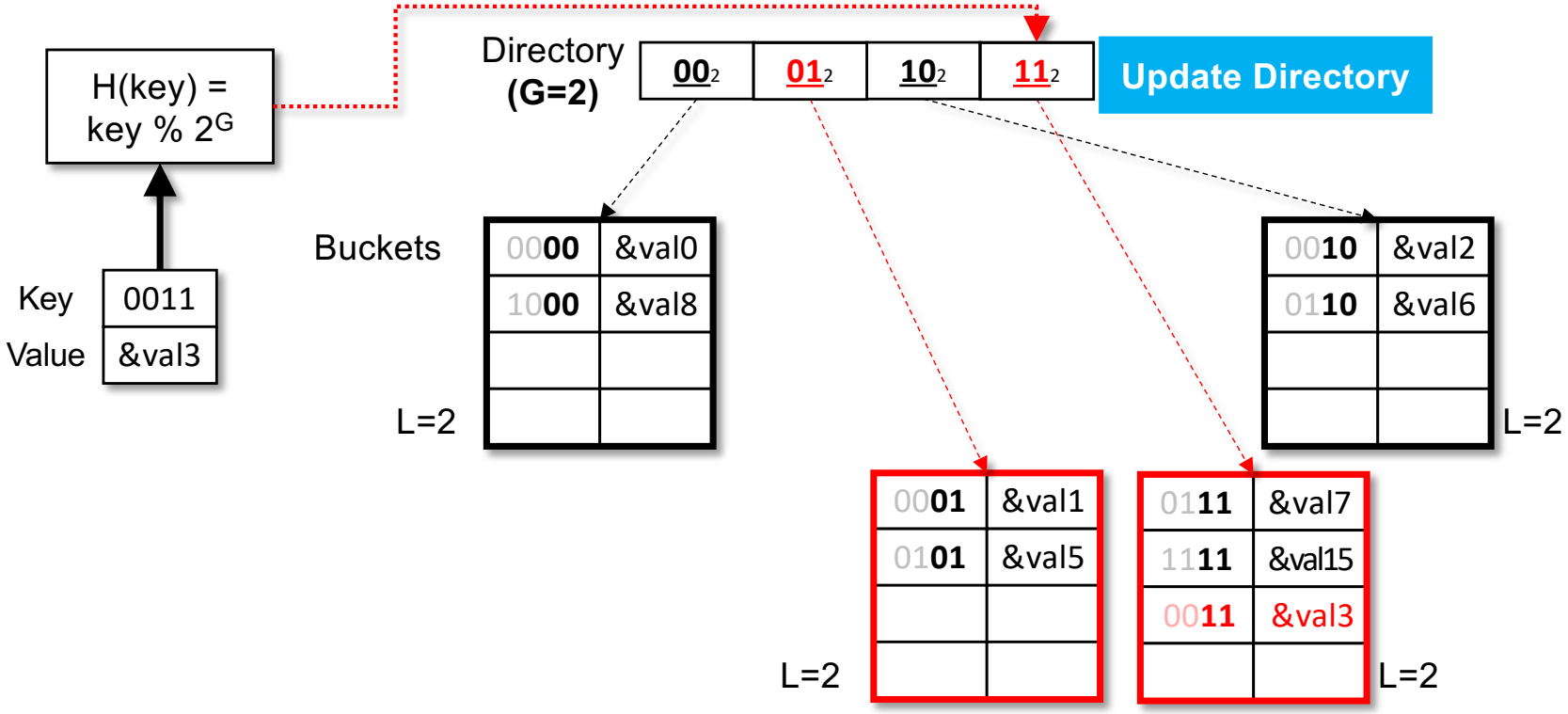
Background: Extendible Hashing – Bucket Split

- Overflow bucket splits



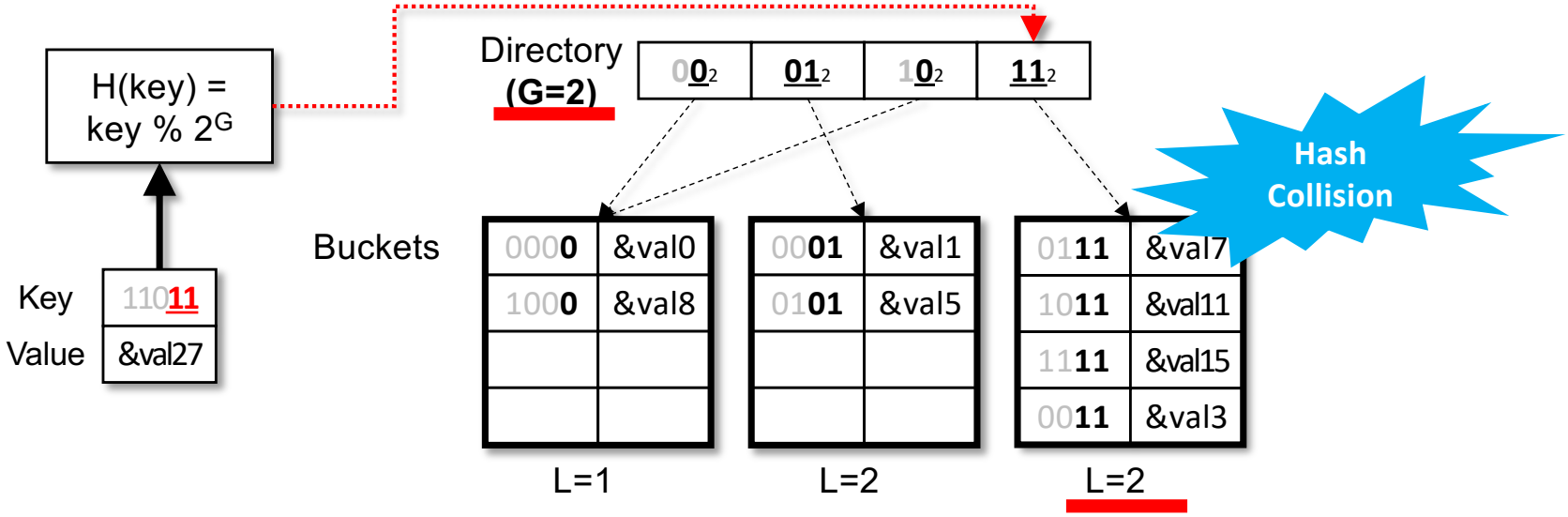
Background: Extendible Hashing – Bucket Split

- When a bucket splits, increase its local depth and update directory
  - At least two pointers need to be updated



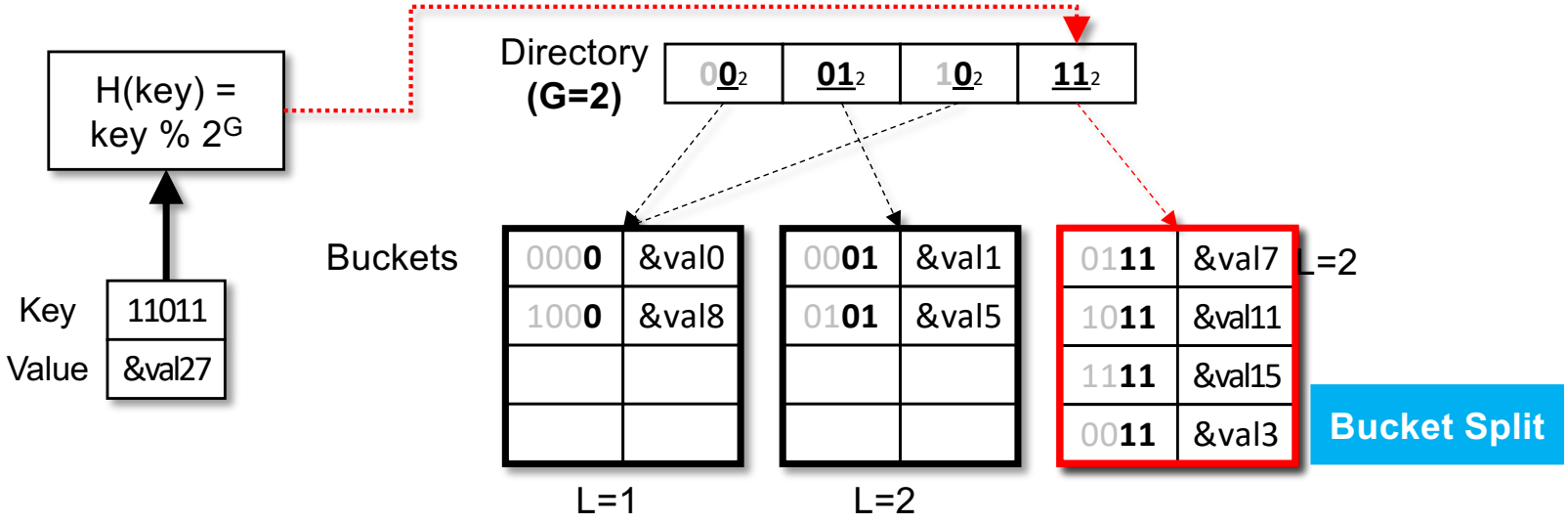
Background: Extendible Hashing – Directory doubling

- If a single pointer was pointing to overflow bucket
  - → Directory Doubling



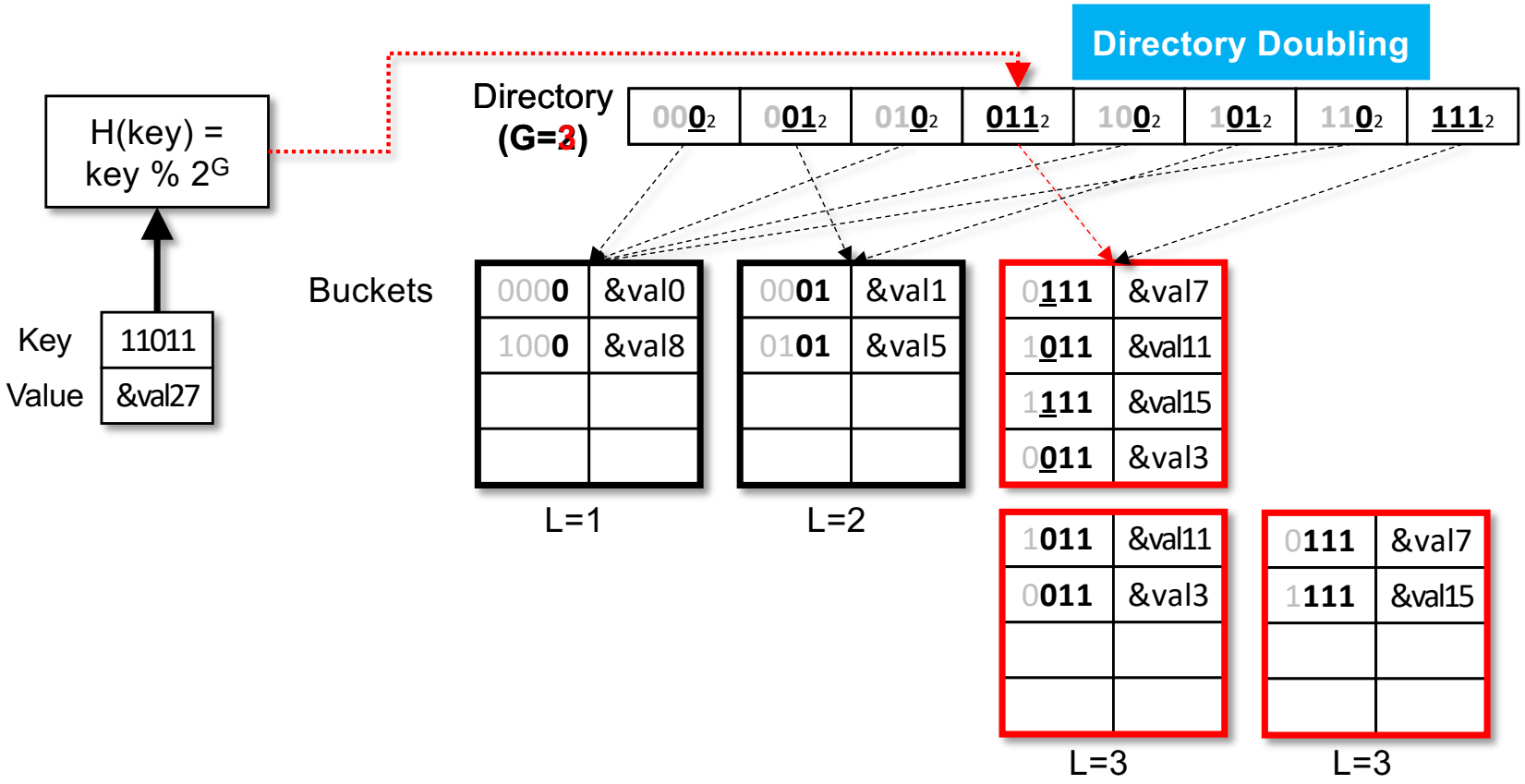
Background: Extendible Hashing – Directory doubling

- If a single pointer points to overflow bucket
  - → Directory Doubling



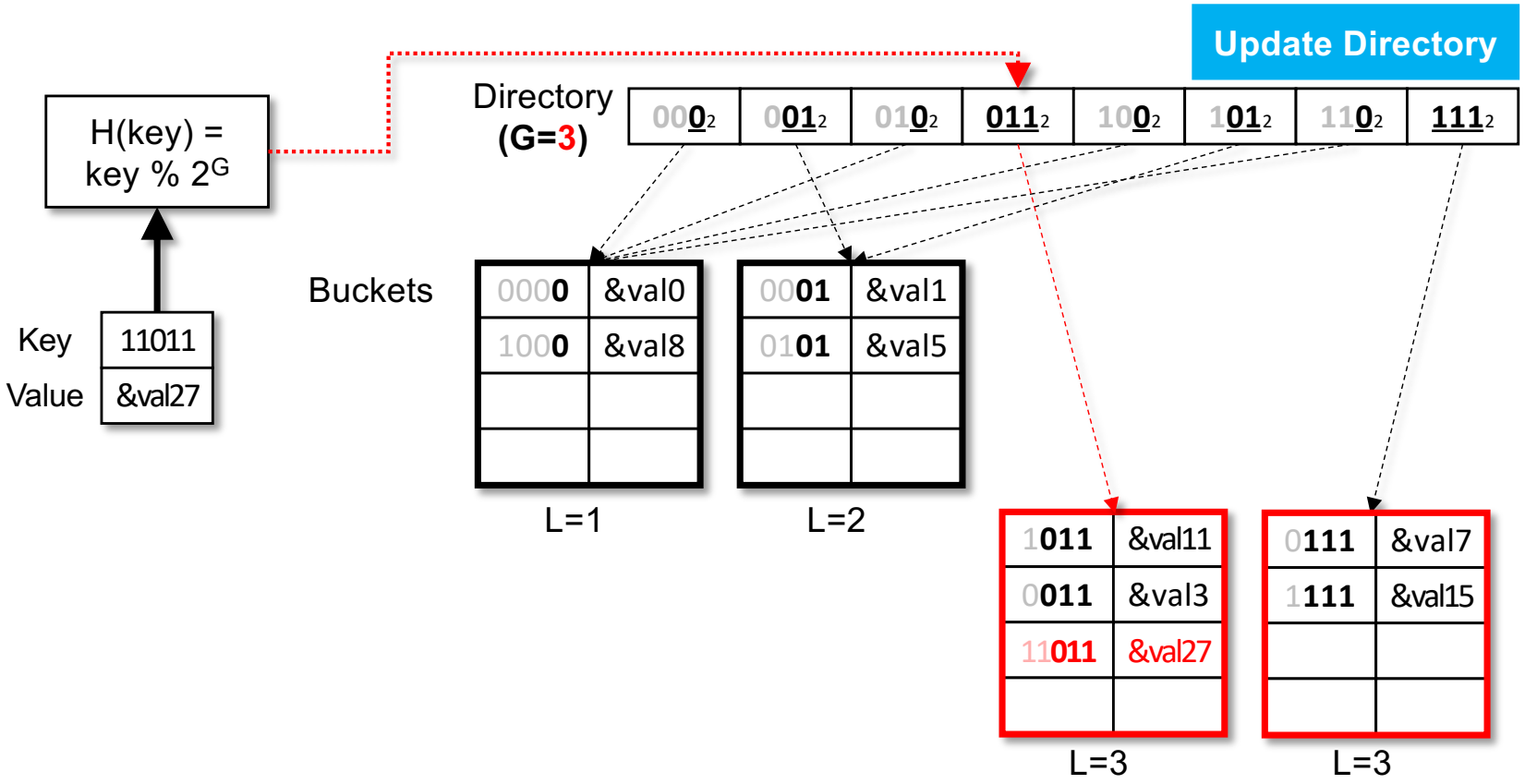
Background: Extendible Hashing – Directory doubling

- If a single pointer points to overflow bucket
  - → Directory Doubling



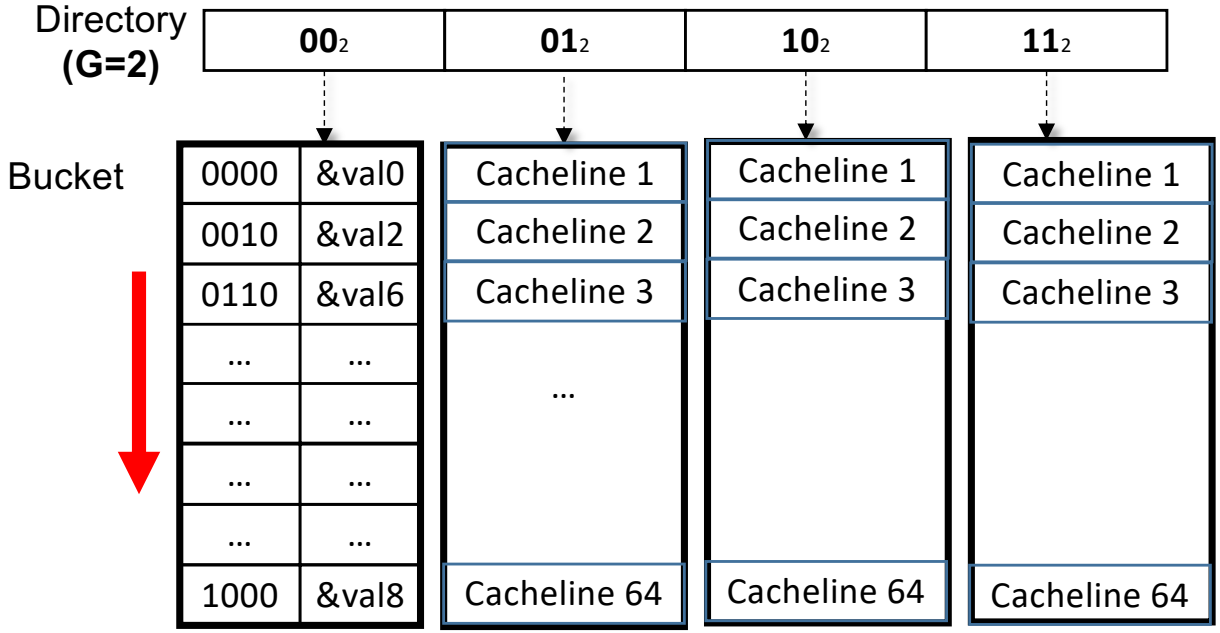
Background: Extendible Hashing – Directory doubling

- If a single pointer points to overflow bucket
  - → Directory Doubling



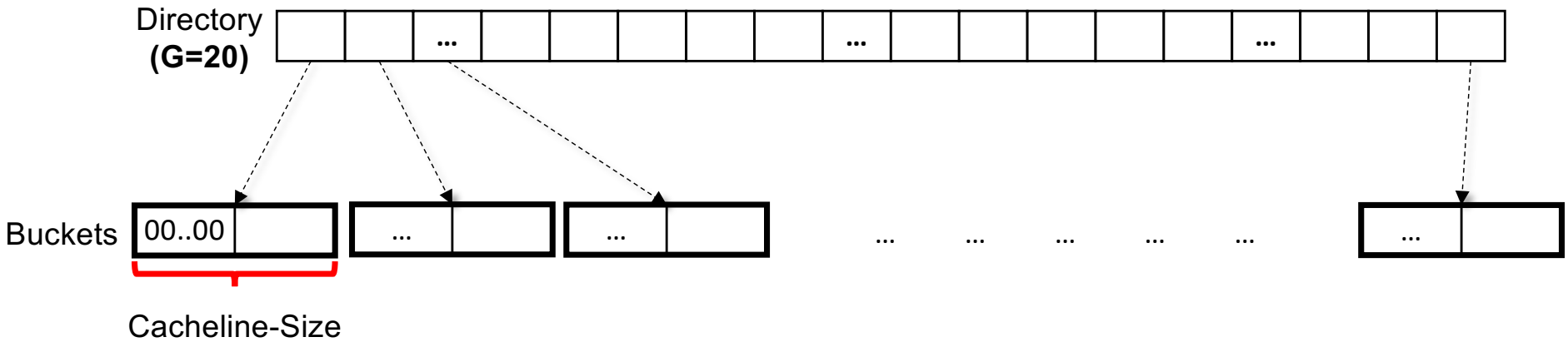
Challenges in In-Memory Extendible Hashing

- Q1: Bucket size?
- If we set the bucket size to 4K
  → 64 cacheline accesses per bucket



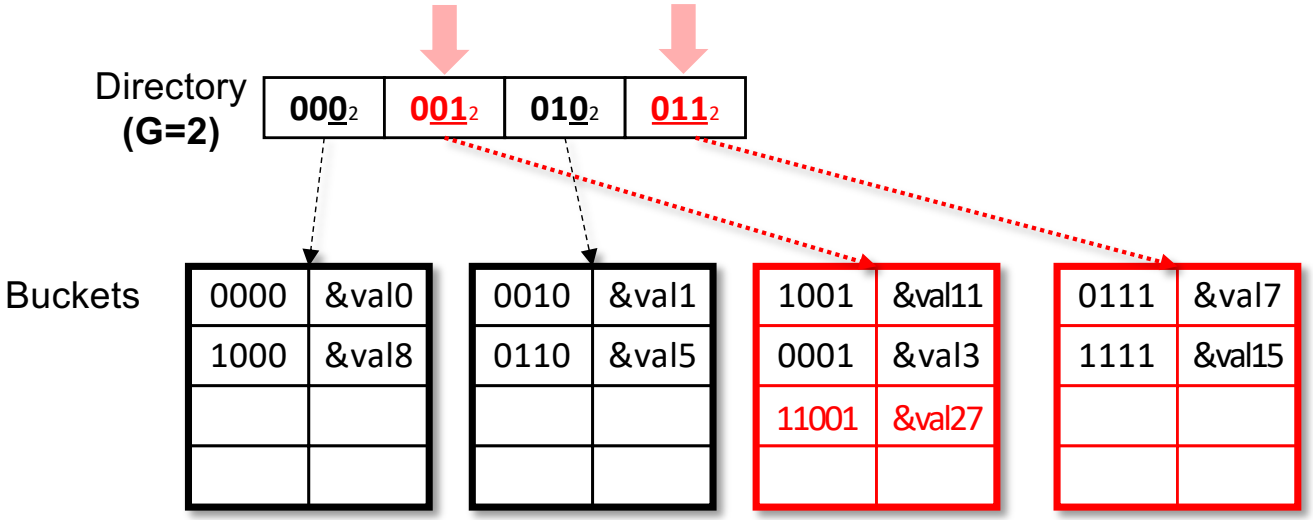
Challenges in In-Memory Extendible Hashing

- Q1: Bucket size?
- If we set the bucket size to a cacheline
  - → Directory size becomes unmanageable (8 byte pointer per cacheline)



Challenges in Extendible Hashing on PM

- Q2: Failure-Atomicity when a bucket splits
  - Split operation updates multiple pointers → Not Failure-Atomic



3-Level Structure

→ Introduces an intermediate level, ***Segment***

→ Lookup via only ***two cacheline accesses***

Failure-atomic Directory Updates

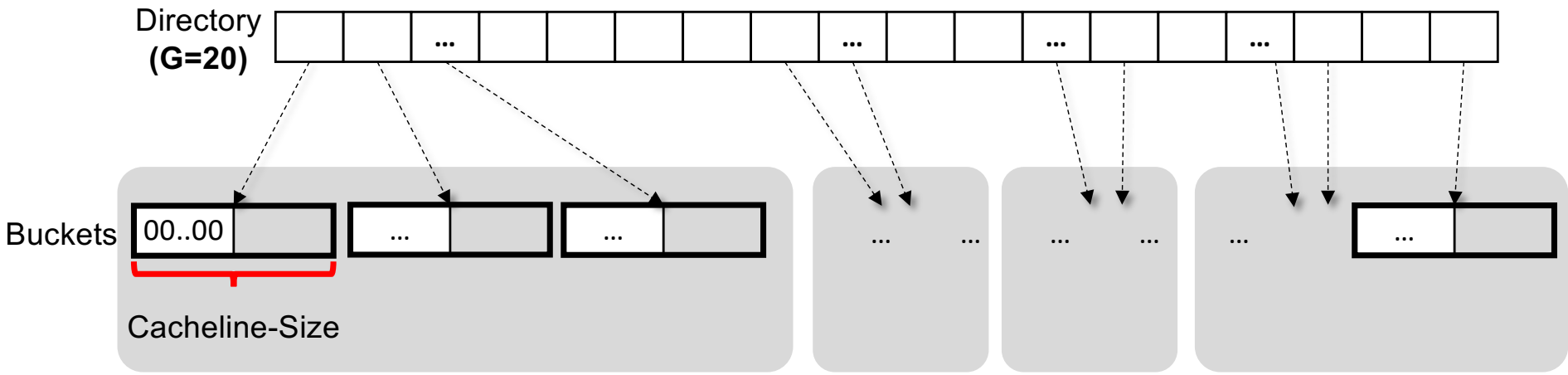
→ Introduces ***buddy tree*** to **manage split history**

Failure-atomic Segment Split

→ ***Lazy deletion*** scheme to **minimize dirty writes**

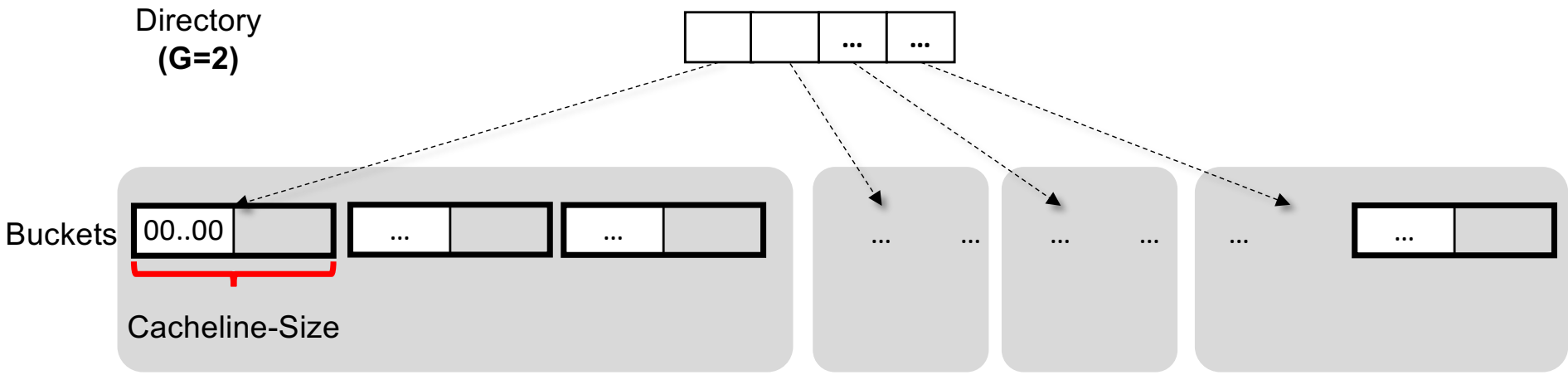
Segment: Intemediate Level

- A group of multiple cacheline-sized buckets = Segment



3-Level Structure

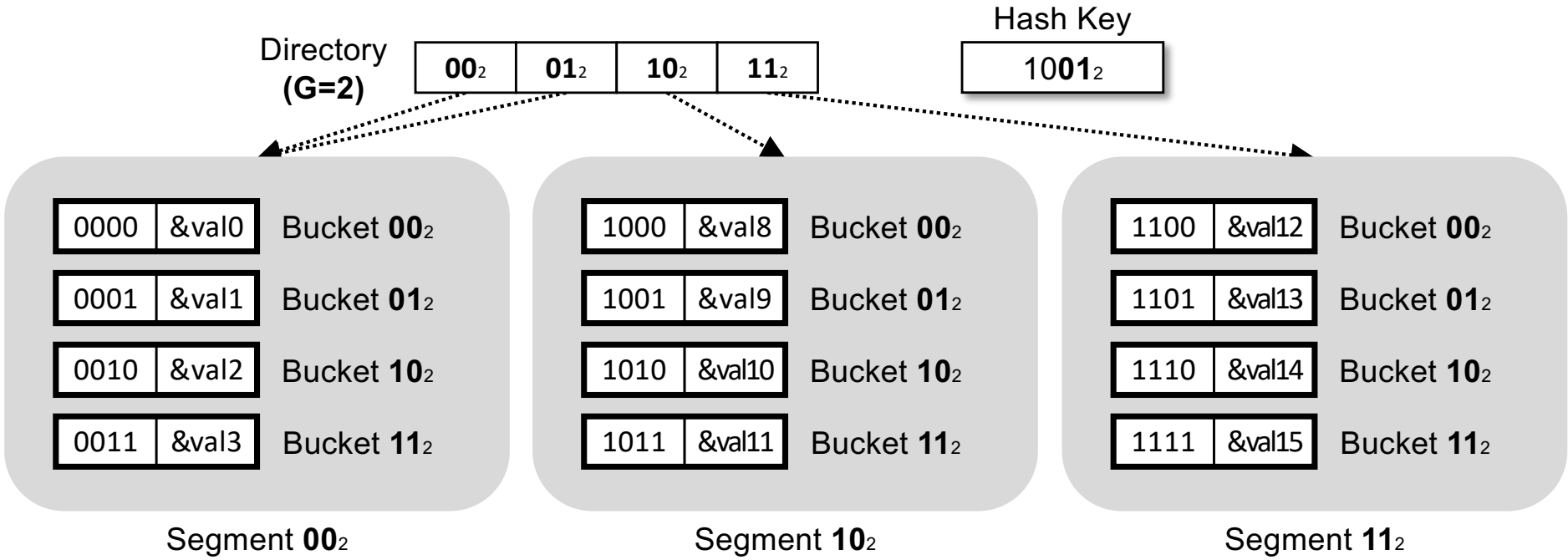
Directory → Segment → Cacheline-sized Bucket



Using intermediate level "**Segment**",
CCEH reduces directory size while keeping bucket size small

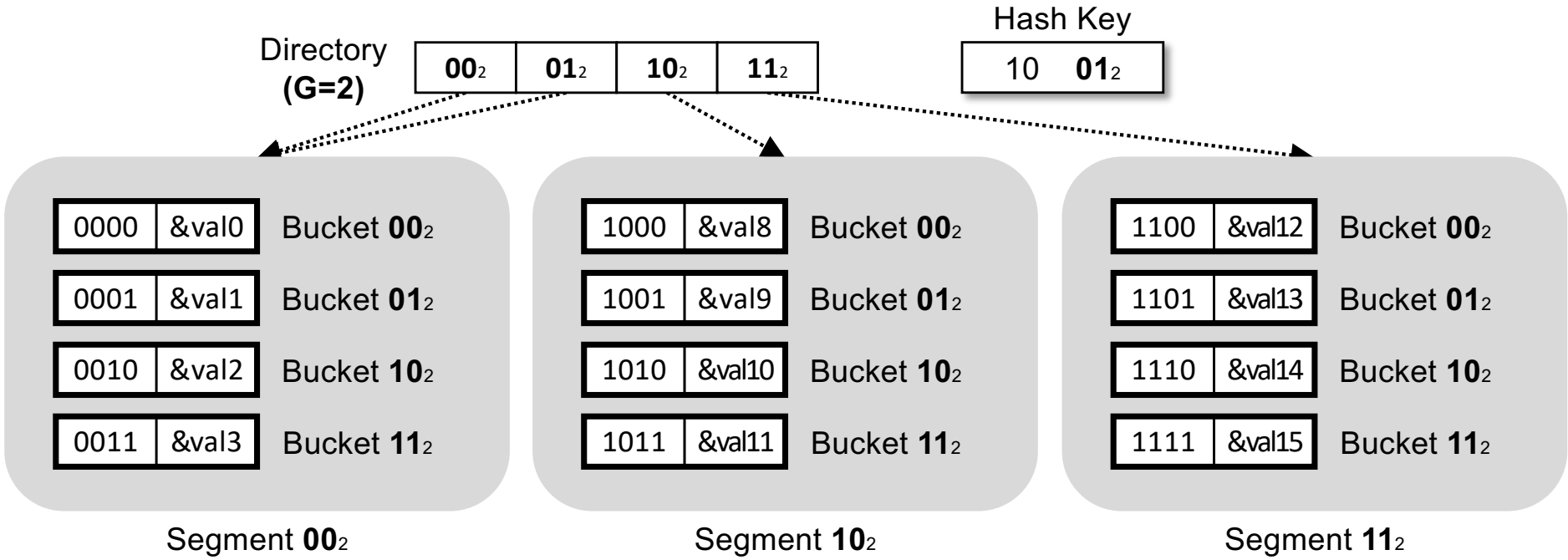
Minimize Cacheline Accesses in Segment

- Q: With large segments, how can we minimize cacheline accesses?



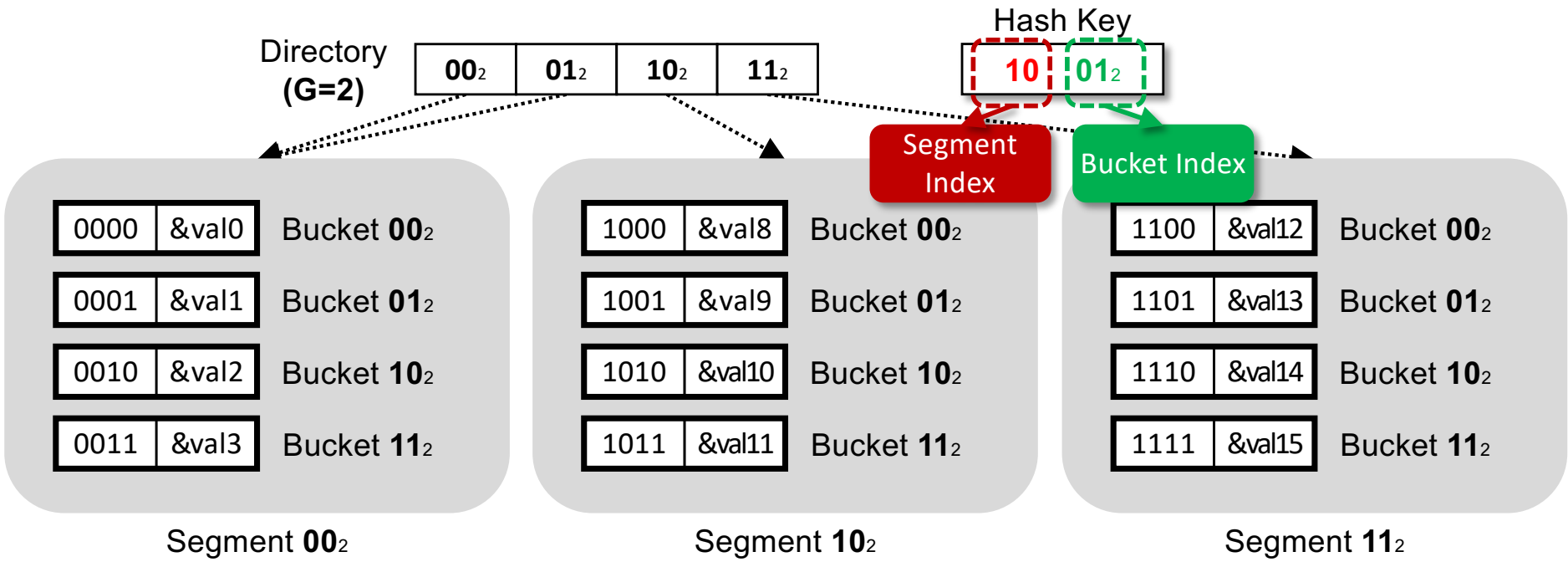
Minimize Cacheline Accesses in Segment

- Q: With large segments, how can we minimize cacheline accesses?



Minimize Cacheline Accesses in Segment

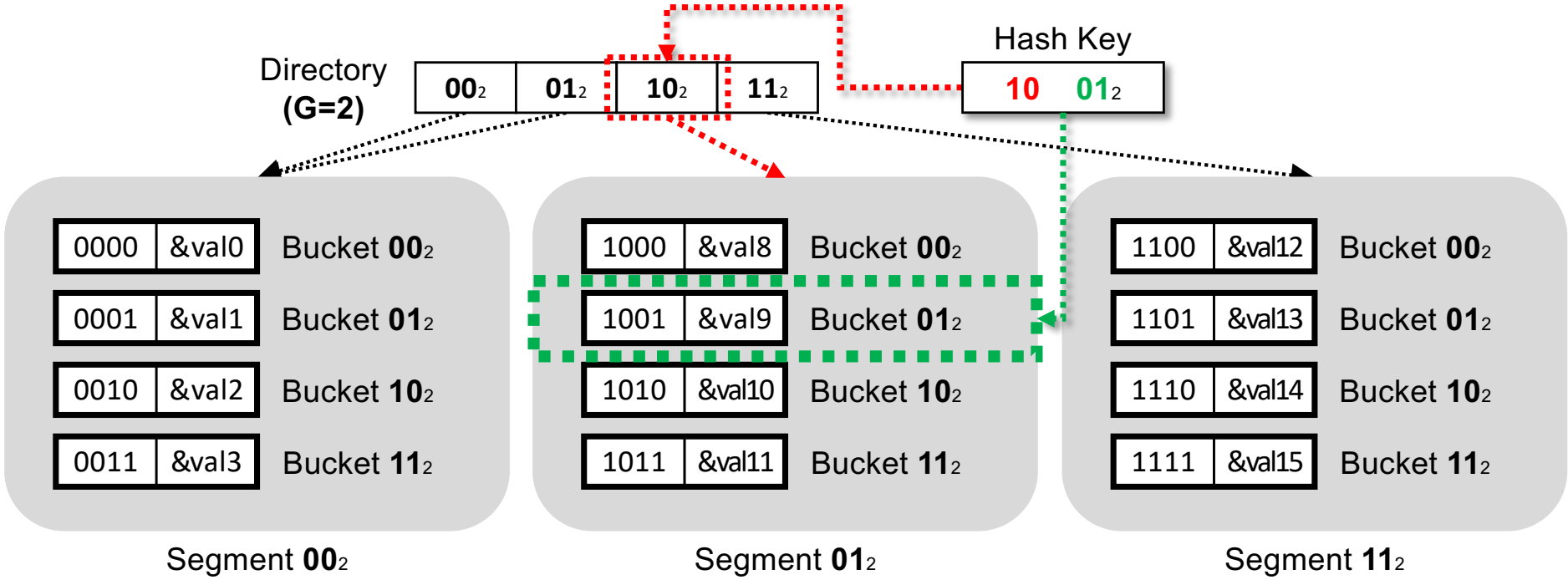
- Q: With large segments, how can we minimize cacheline accesses?



Minimize Cacheline Accesses in Segment

- A: Use hash key as index for both directory and segment

  → No need to access irrelevant buckets



3-Level Structure

→ Introduces an intermediate level, ***Segment***

→ Lookup via only ***two cacheline accesses***

Failure-atomic Directory Updates

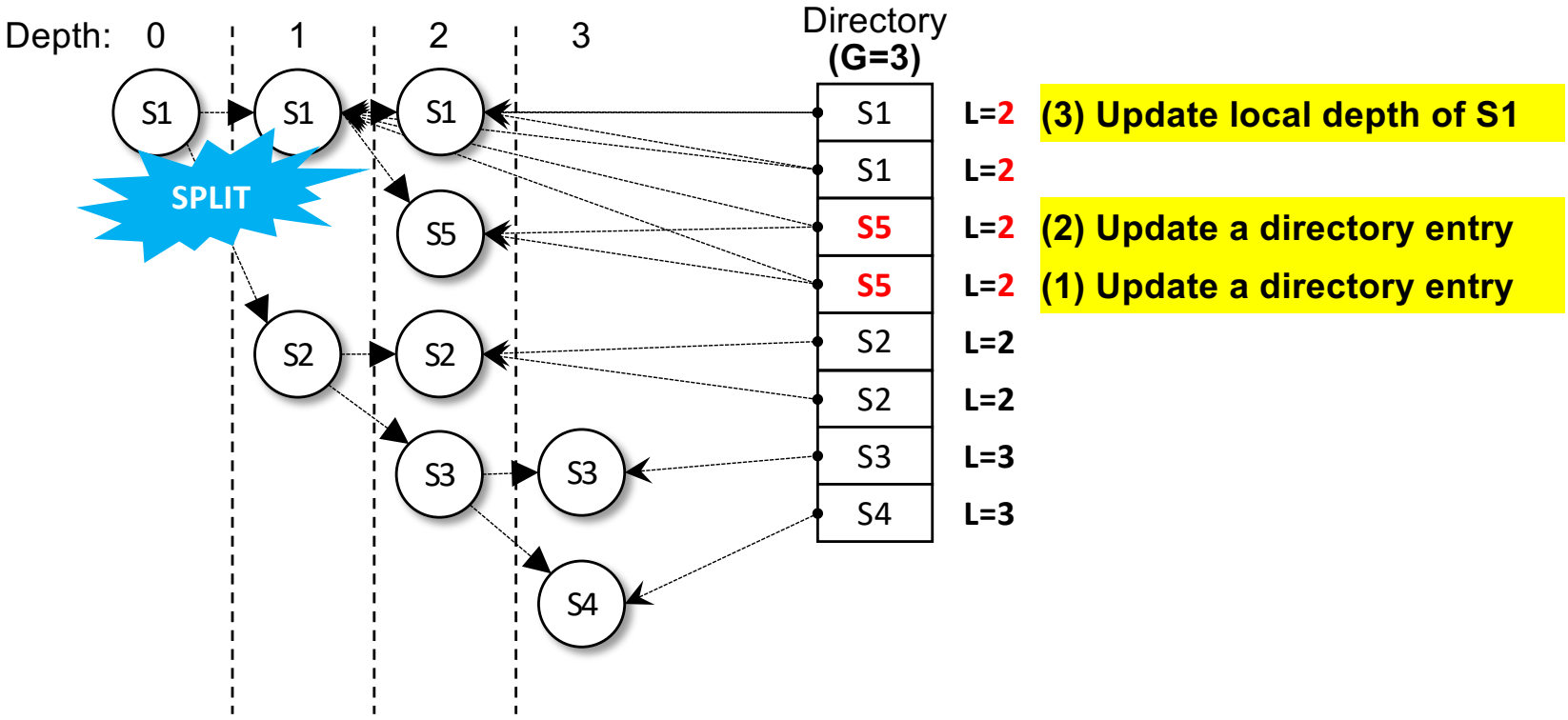
→ Introduces ***buddy tree*** to **manage split history**

Failure-atomic Segment Split

→ ***Lazy deletion*** scheme to **minimize dirty writes**

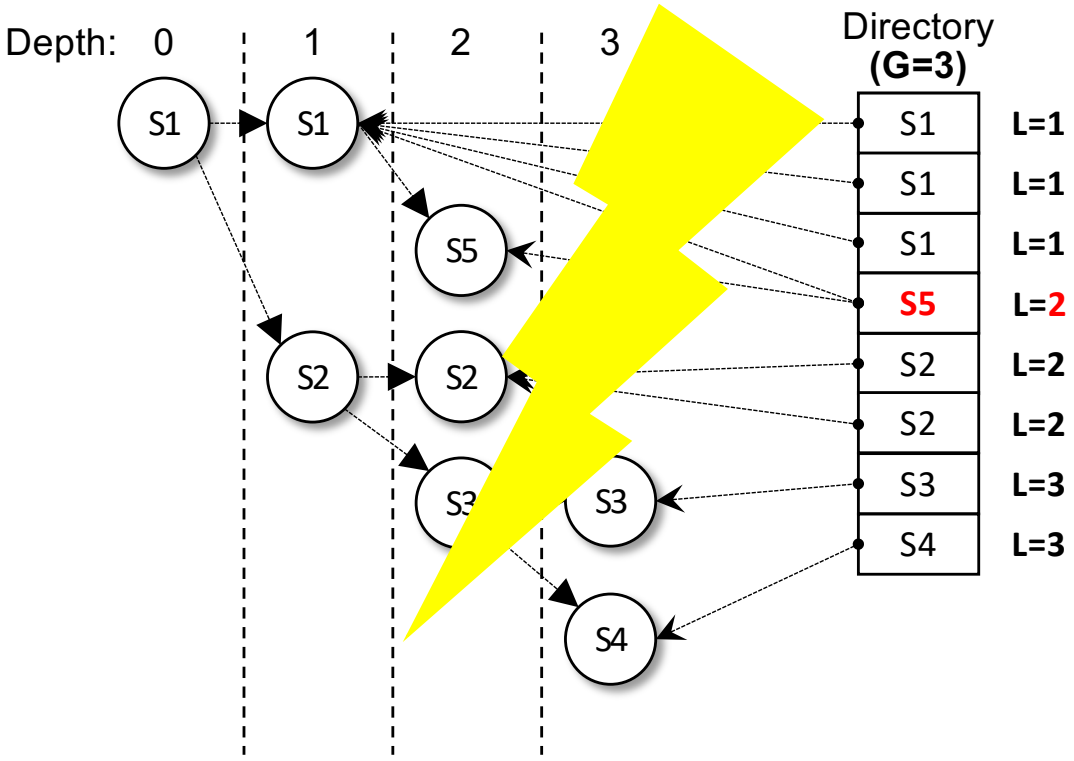
Recovery: Split History Buddy Tree in CCEH

- Using MSB segment index, split segments are pointed by adjacent directory entries



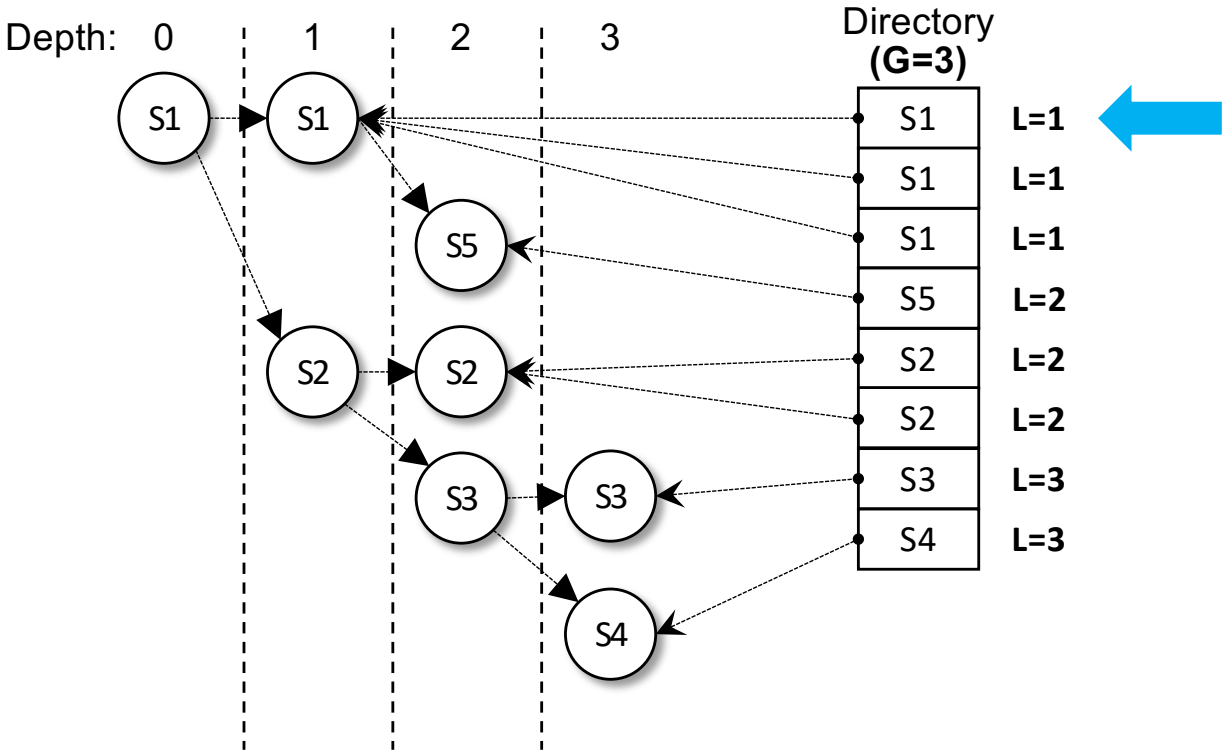
Recovery: Split History Buddy Tree in CCEH

- Suppose a system crashes while S5 splits



Recovery: Split History Buddy Tree in CCEH

- Each **segment** must be pointed by 2^{G-L} directory entries
- If not, rollback the split

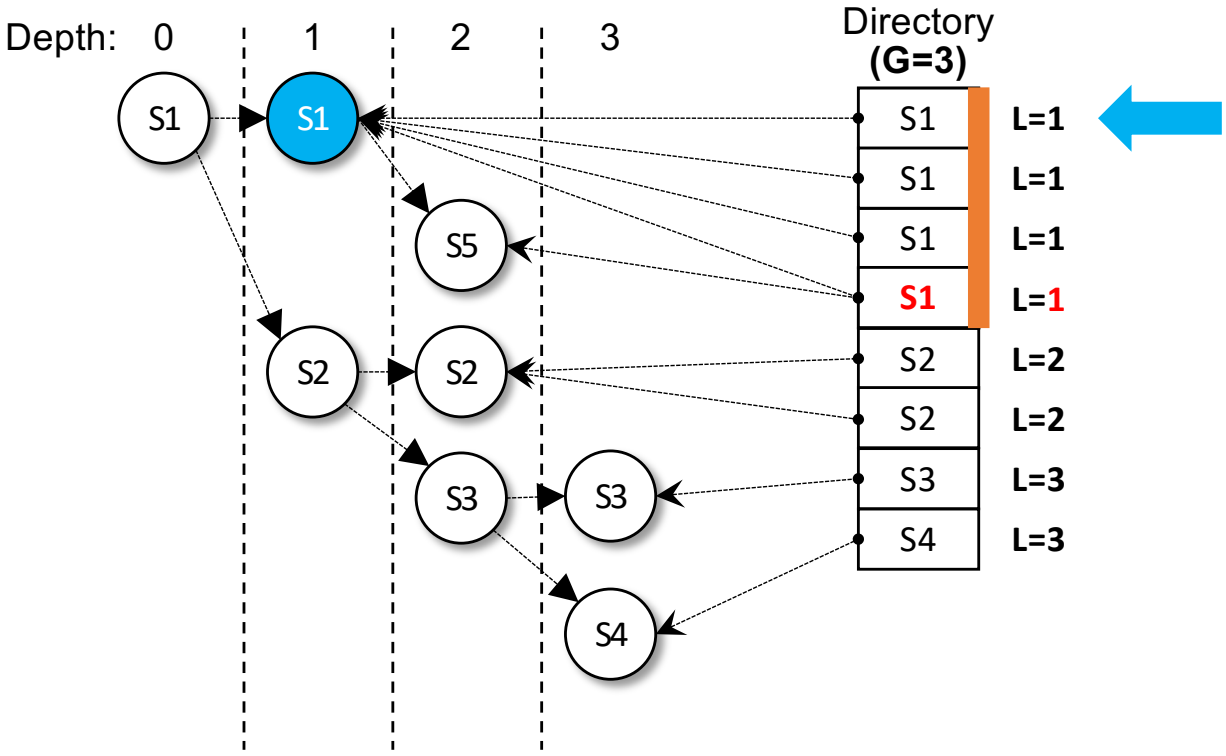


$$Stride = 2^{G-L}$$

Global Depth **G** = 3
Local Depth **L** = 1
Stride = 4

Recovery: Split History Buddy Tree in CCEH

- Each **segment** must be pointed by 2^{G-L} directory entries
- If not, rollback the split



$$Stride = 2^{G-L}$$

Global Depth **G** = 3
Local Depth **L** = 1
Stride = 4

3-Level Structure

→ Introduces an intermediate level, ***Segment***

→ Lookup via only ***two cacheline accesses***

Failure-atomic Directory Updates

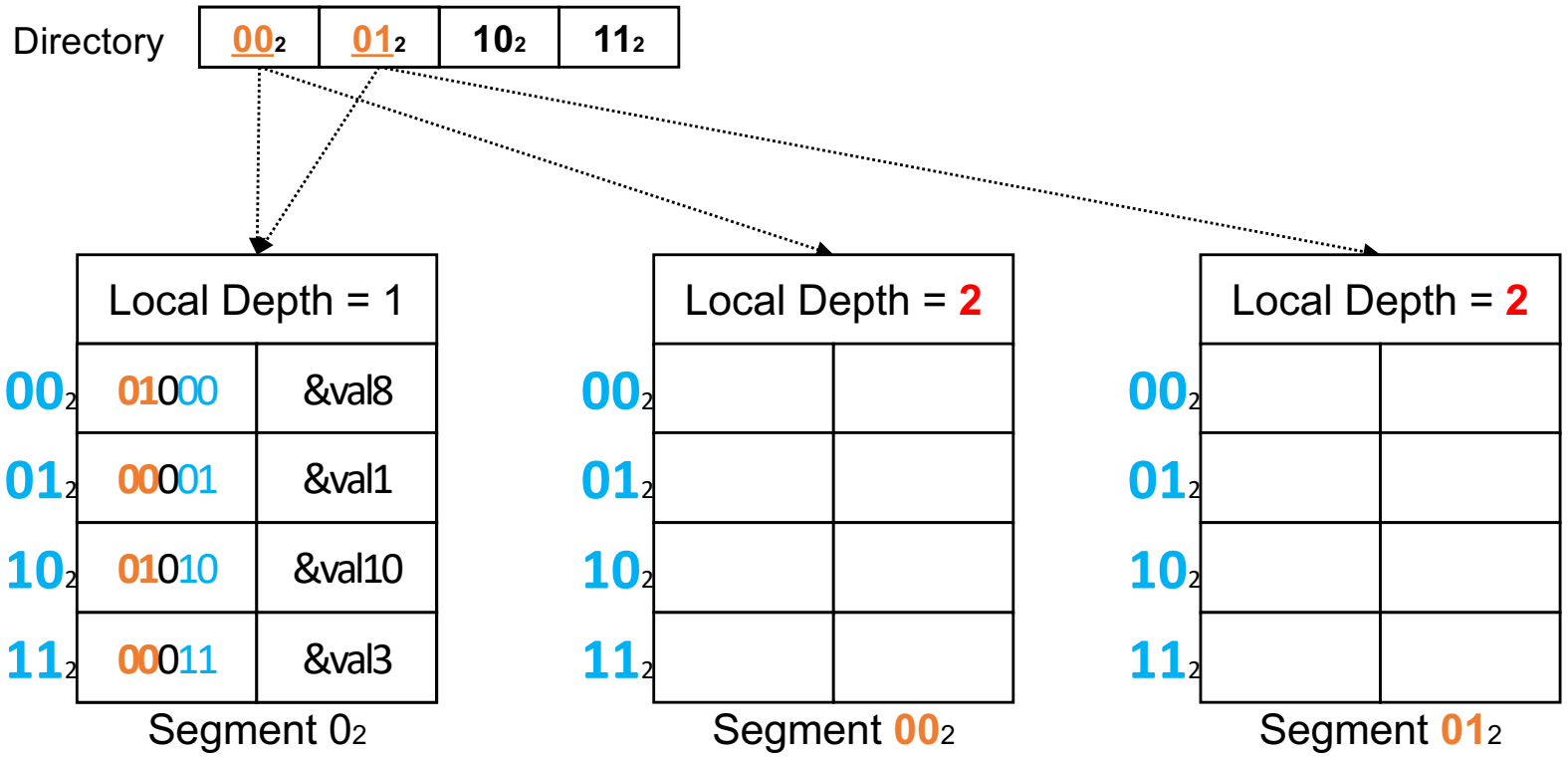
→ Introduces ***buddy tree*** to **manage split history**

Failure-atomic Segment Split

→ ***Lazy deletion*** scheme to **minimize dirty writes**

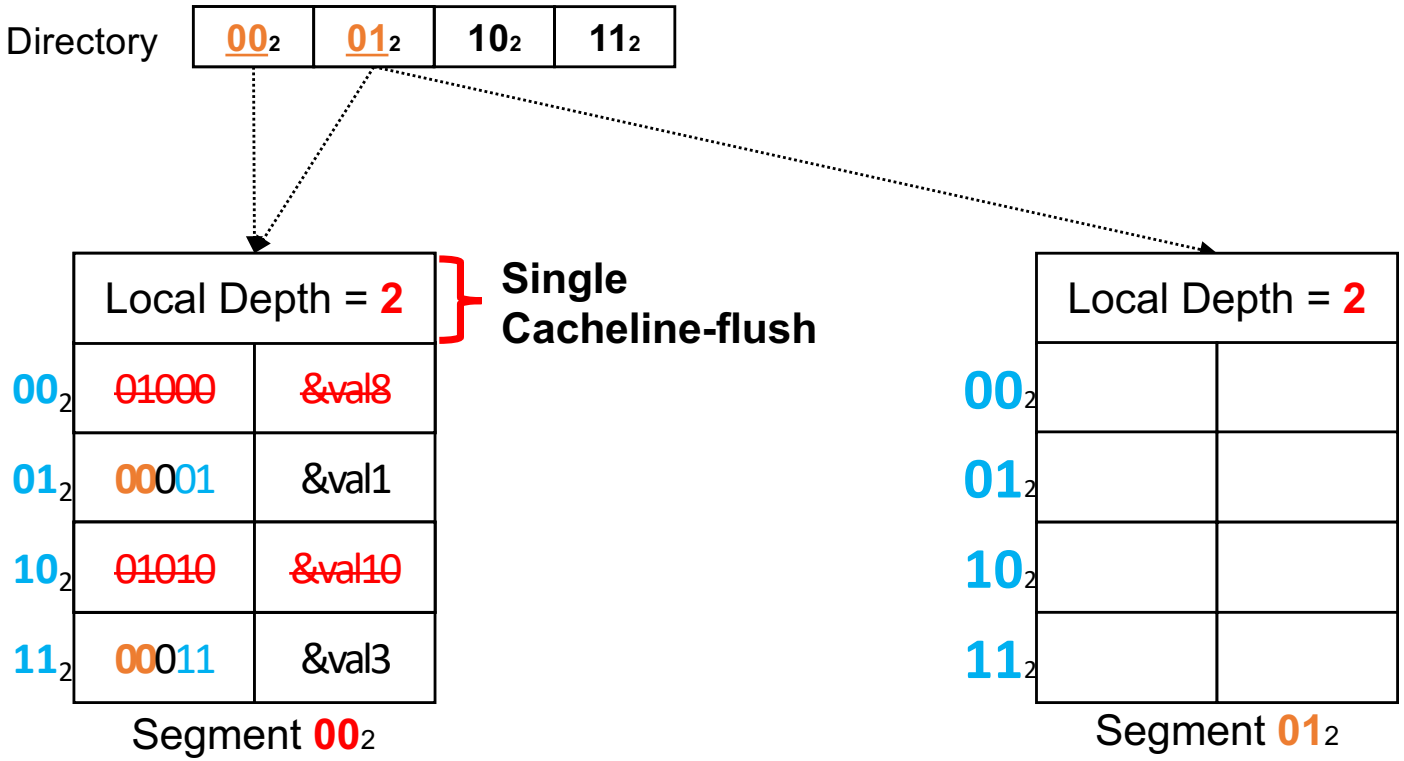
Segment Split: Legacy CoW

- **Copy-on-Write Split**
  - **Pro:** Lock-Free Search
  - **Con**: Move all key-value pairs → a large # of dirty writes



Segment Split: Lazy Deletion

- **Lazy Deletion**
  - **Pro:** Minimize Dirty Writes
  - **Con**: Lock-Free search does not work

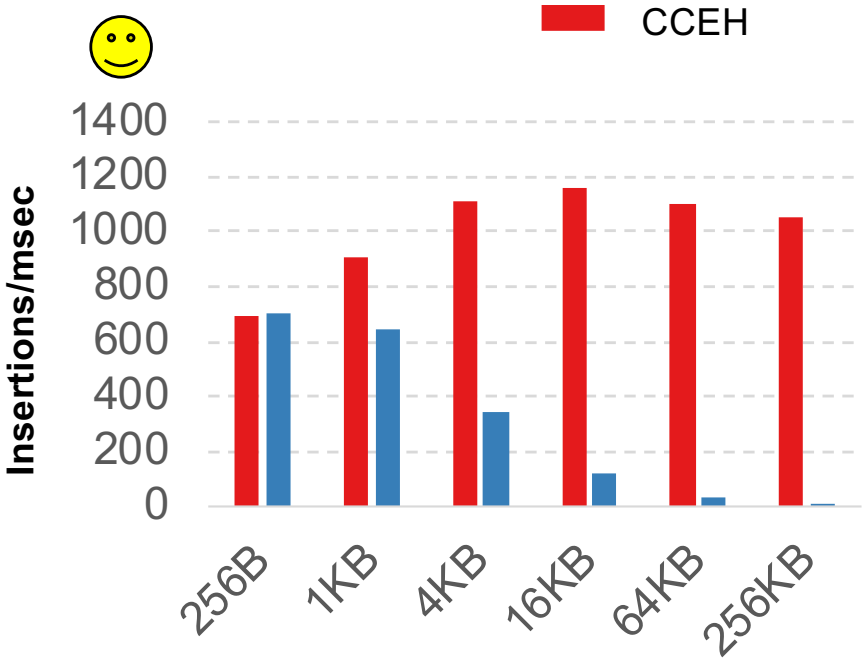


Experimental Setup

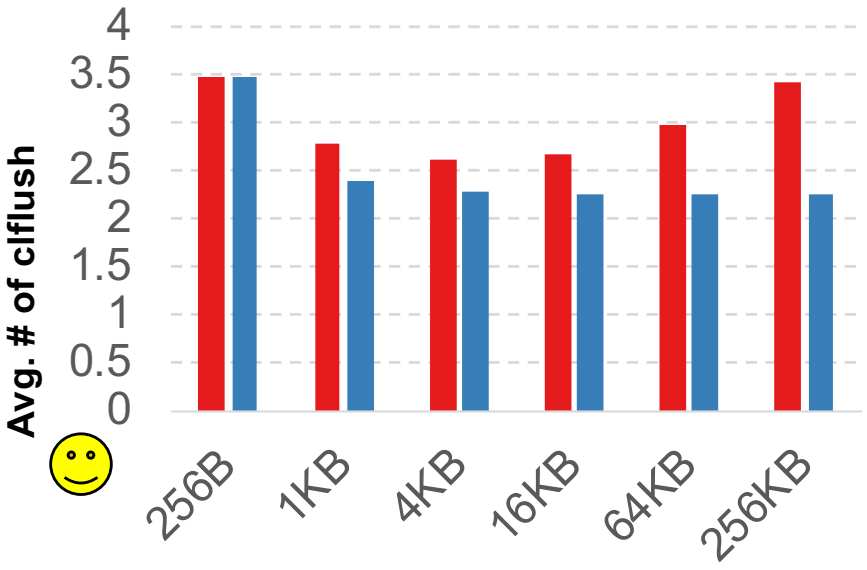
CPU	2x Intel Xeon Haswell-Ex E7-4809 v3 → 8 cores, 2.0 GHz → 20MB L3 cache
Memory	64GB of DDR3 DRAM
PM	**Quartz:** A DRAM-based PM latency emulator * To emulate write latency, we inject stall cycle after each ***clflush***
Workload	160M Random Number Dataset

CCEH VS Legacy Extendible Hash

- Pro: Constant number of cacheline accesses with varying directory size
- Con: Low utilization and more cacheline flushes due to hash collisions



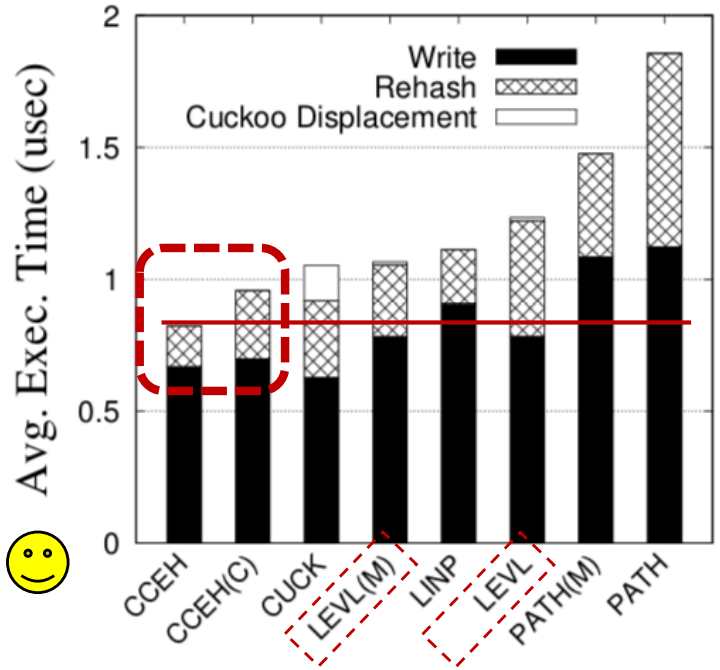
Segment Size for CCEH
Bucket Size for Extendible Hash



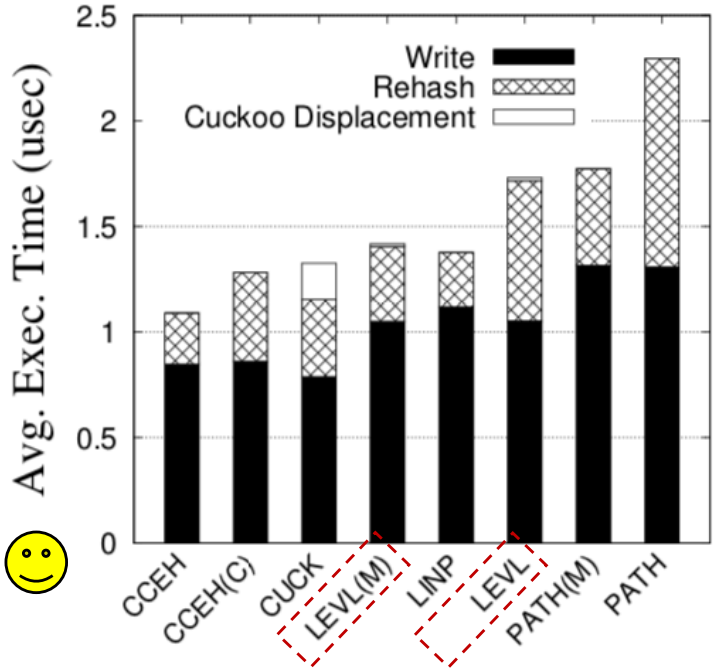
Segment Size for CCEH
Bucket Size for Extendible Hash

Insertion Performance Breakdown

- **CCEH is 70% faster than Level Hashing on PM**
- Fewer # of cacheline accesses
- Lazy deletion → Efficient segment split



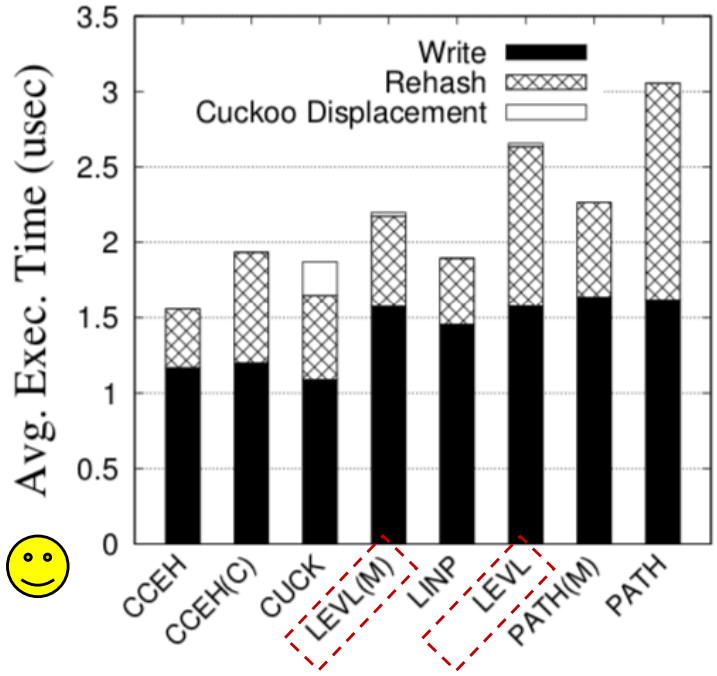
(a) 120/120 (DRAM)



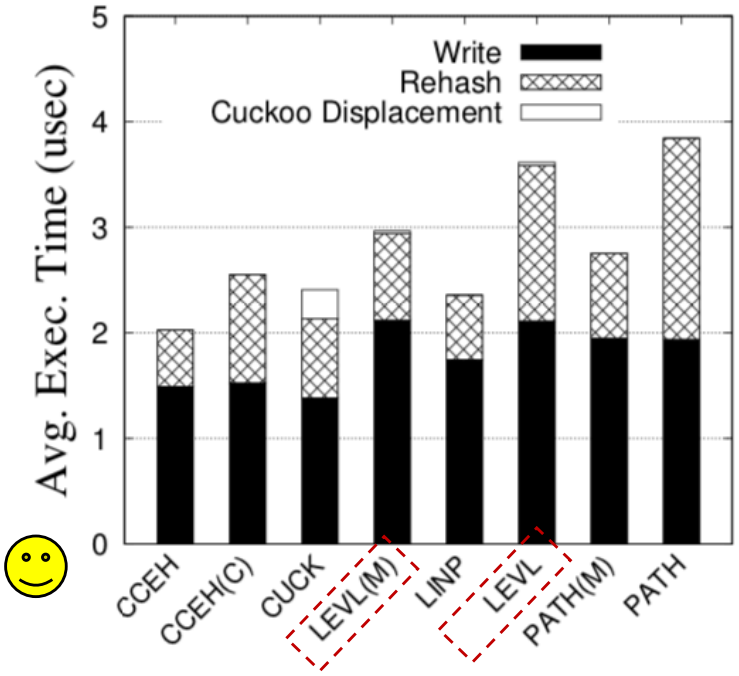
(b) 240/300

Insertion Performance Breakdown

- **CCEH is 70% faster than Level Hashing on PM**
- Fewer # of cacheline accesses
- Lazy deletion → Efficient segment split



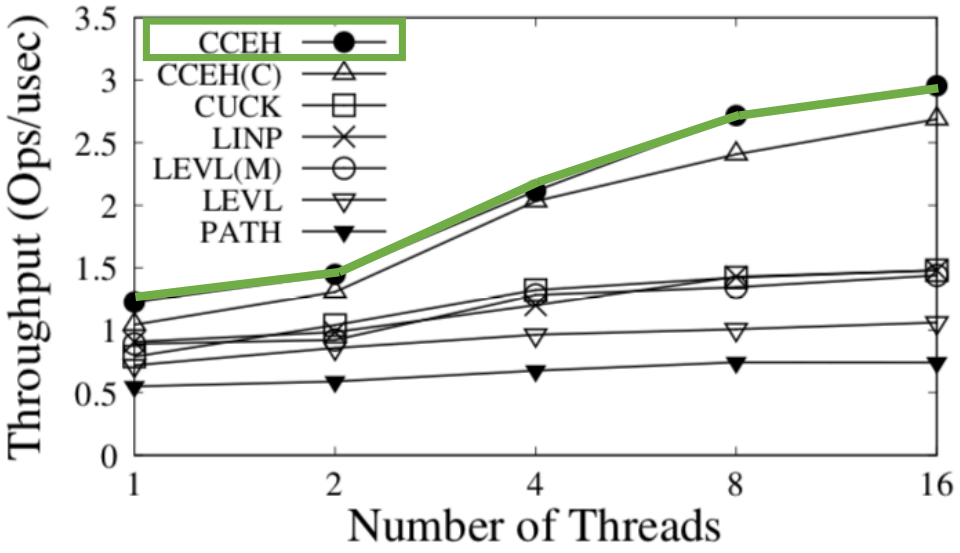
(c) 240/500



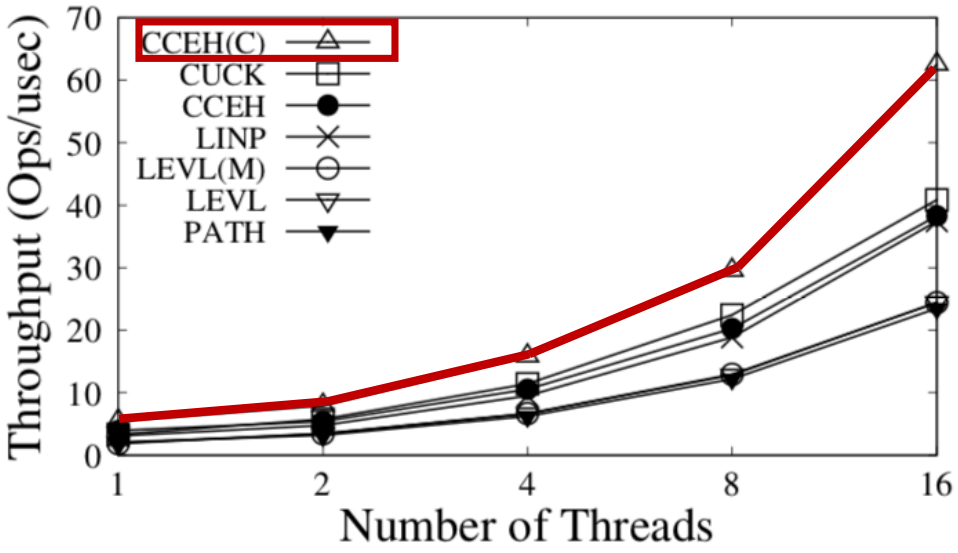
(d) 240/700

Throughput Comparison

- Lock-Free Search
  → CCEH (C) shows the highest search throughput



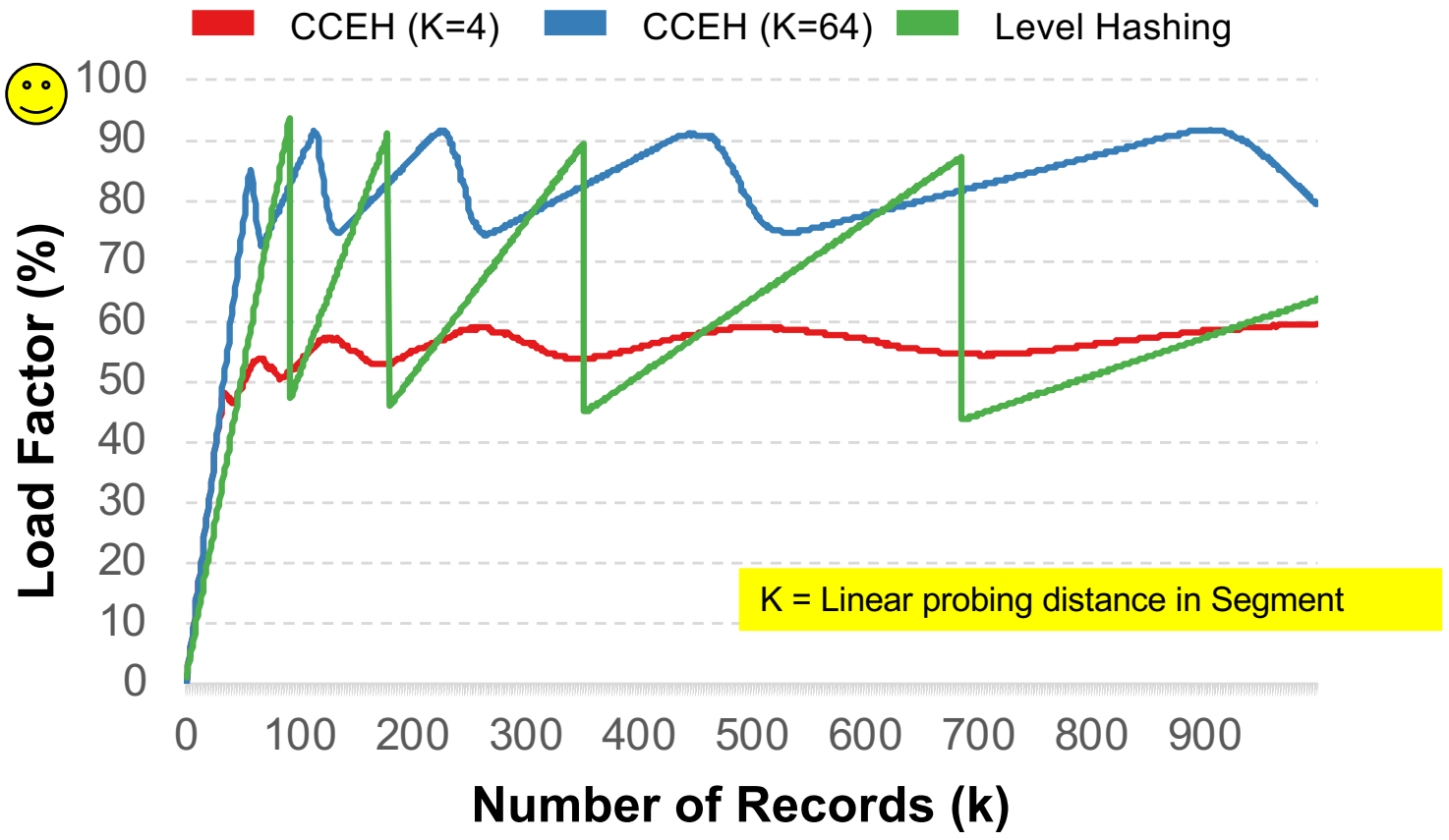
(b) Insertion Throughput



(c) Search Throughput

Load Factor

- **Level Hashing**
  - Suffer from full table rehashing like the other static hashing
- **CCEH**
  - Dynamic allocation of small segments → Smooth curves



Summary

Write-Optimized CCEH

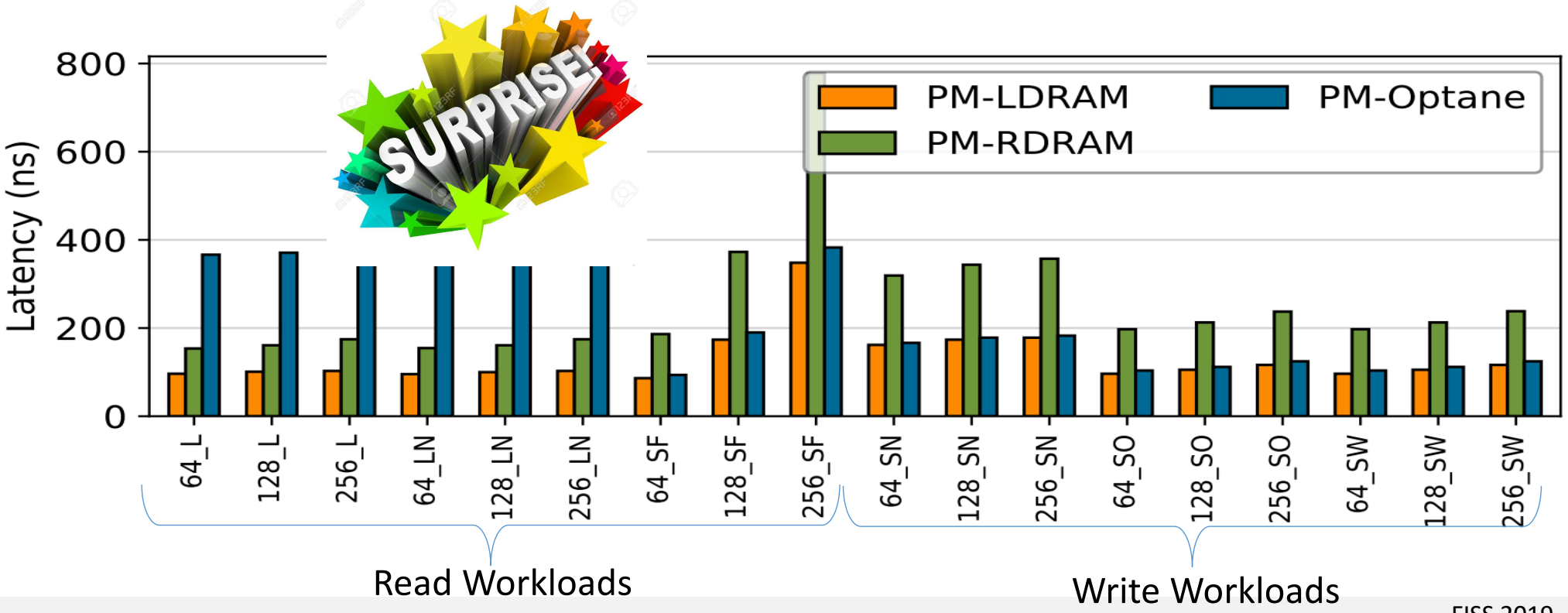
- Optimizations
  - 3-Level Structure
  - Lazy Deletion
  - Logging-less Directory Updates

- Source Codes: http://github.com/DICL/CCEH

Optane PMM Latency

- "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module" - NVML UCSD
  - Load latency of PM-Optane is higher
  - Store latency of PM-Optane is similar to PM-LDRAM

According to this result, we better keep redundant copies, one in volatile DRAM and the other in PM

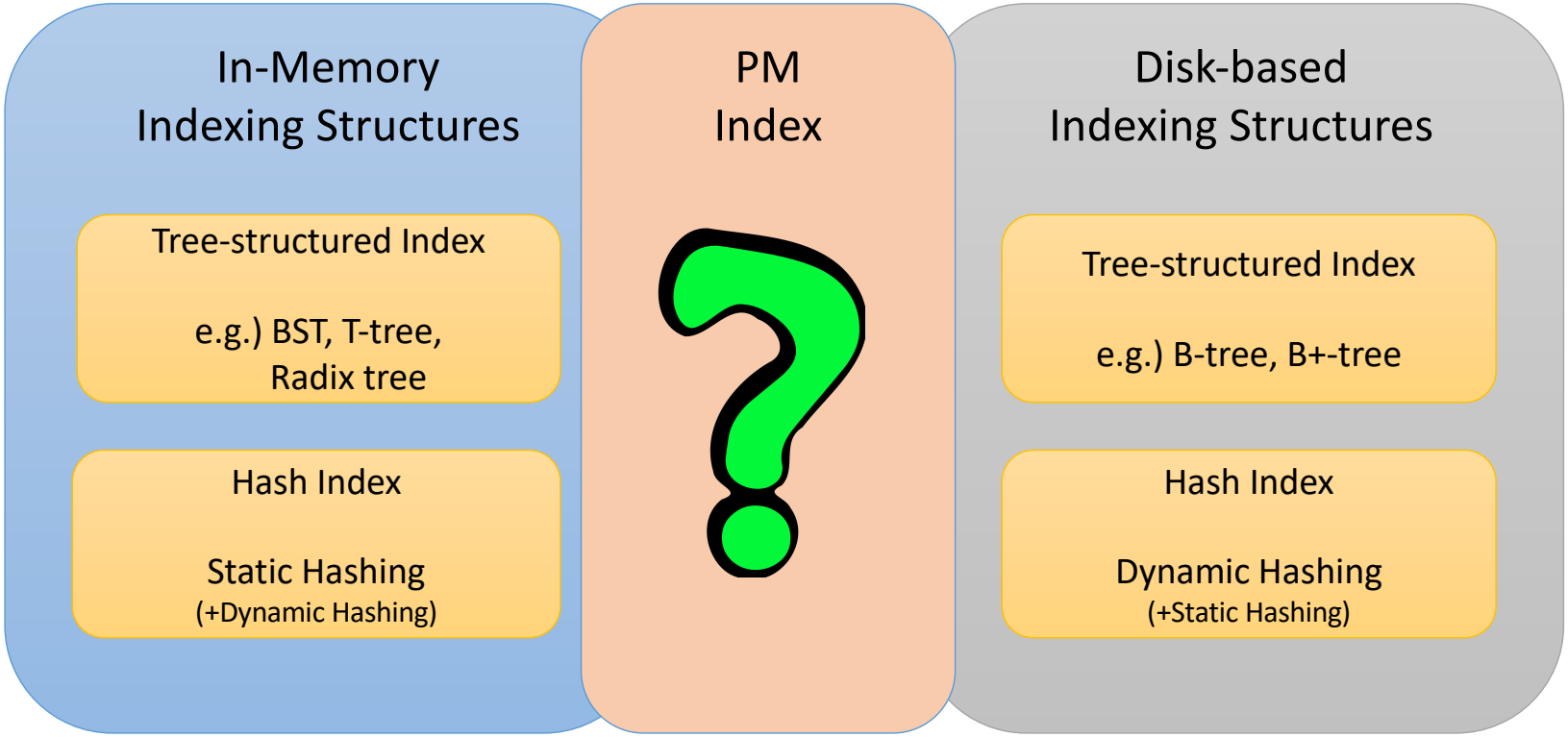


Conclusion

Write-Optimized (uh-oh) CCEH

- Having a single copy of bucket on Optane PMM, CCEH may suffer from its high read latency

- Need to revisit (our) indexing structures for Optane PMM



Thank you.


thank you
danke
рахмат
謝謝
ngiyabonga
teşekkür ederim
Баярлалаа
спасибо
faafetai lava
mersi
kia ora
barka
welalin
tack
spas
dank je
misaotra
matondo
paldies
grazzi
mahalo
tapadh leat
vinaka
спасибі
blagodaram
gracias
хвала
asante
manana
obrigada
tenki
murakoze
kiitos
dankie
dhanyavad
nanni
nandri
mauruuru
köszönöm
hvala
gràcie
bayarlalaa
bedankt
enkosi
dziękuję
akun dankon
aċiū
djiere dieuf
tau
mochchakkeram
дякую
mamnun
chokrane
chnorakaloutioun
gratias ago
gràcies
sulpáy
go raibh maith agat
sobodi
děkuji
obrigado
sagolun
sukriya
kop khun krap
taiku
arigatô
takk
dakujem
trugarez
mèsi
didi madloba
najis tuke
ありがとう
tanemirt
grazie
kam sah hamnida
rahmat
terima kasih
rahmet
ευχαριστώ
diolch
dhanyavadagalu
shukriya
mercê
мерси
তোমাকে ধন্যবাদ
감사합니다
xièxie
merci