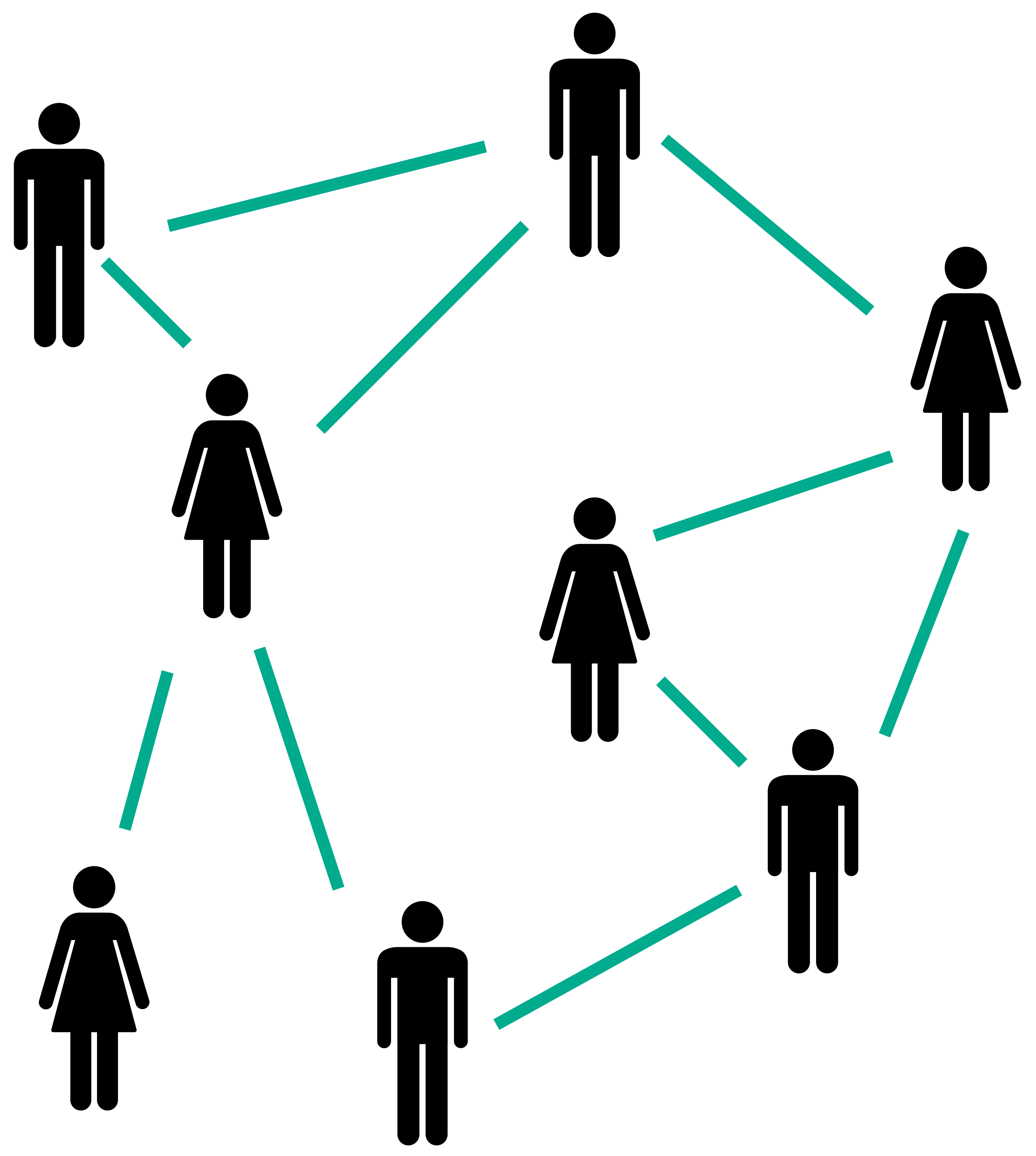
Kangaroo: Caching Billions of Tiny Objects on Flash

Sara McAllister[1]
Benjamin Berg[1], Julian Tutuncu-Macias[1], Juncheng Yang[1], Sathya Gunasekar[2], Jimmy Lu[2], Daniel S. Berger[3], Nathan Beckmann[1], Gregory R. Ganger[1]

SOSP 2021
Wednesday, October 27, 2021

[1] Carnegie Mellon University
[2] Facebook
[3] Microsoft Research / University of Washington

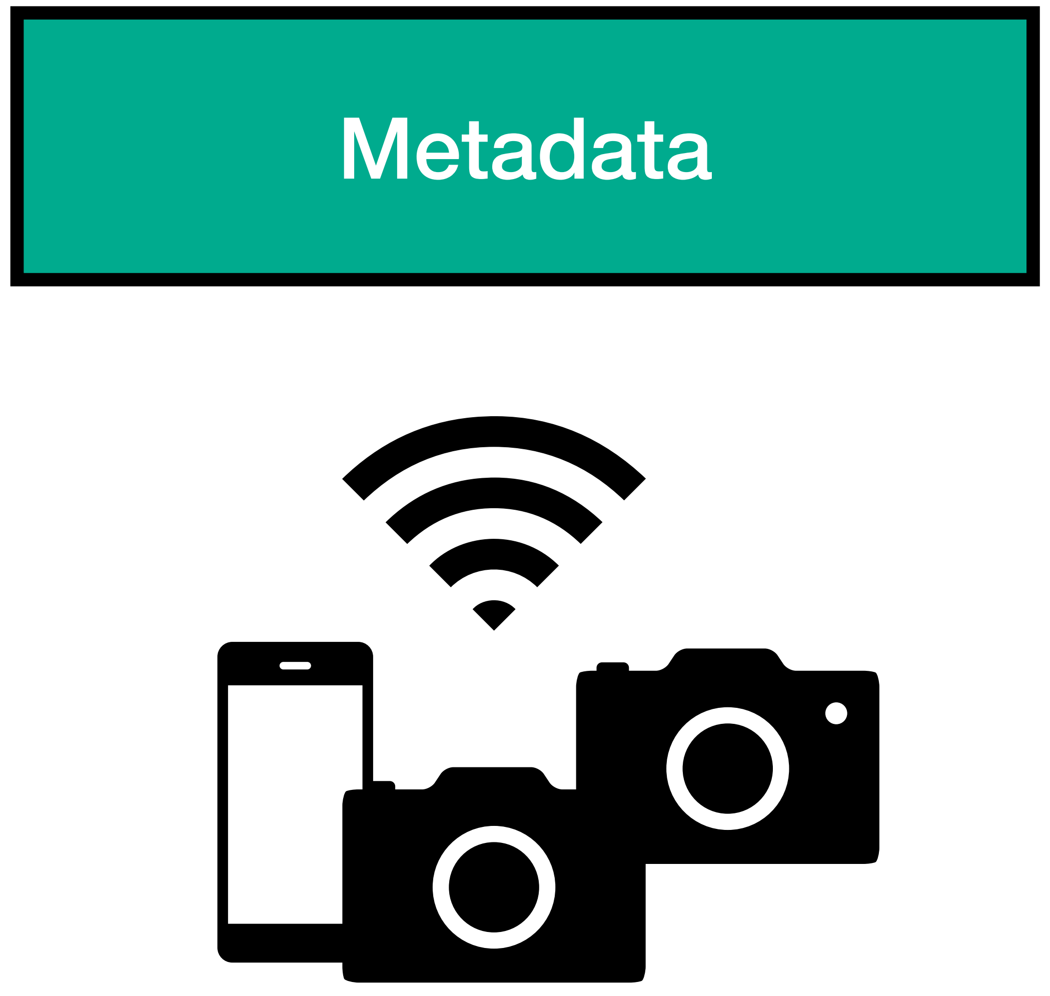
Tiny objects are prevalent



Social Graphs

Facebook social graph edges

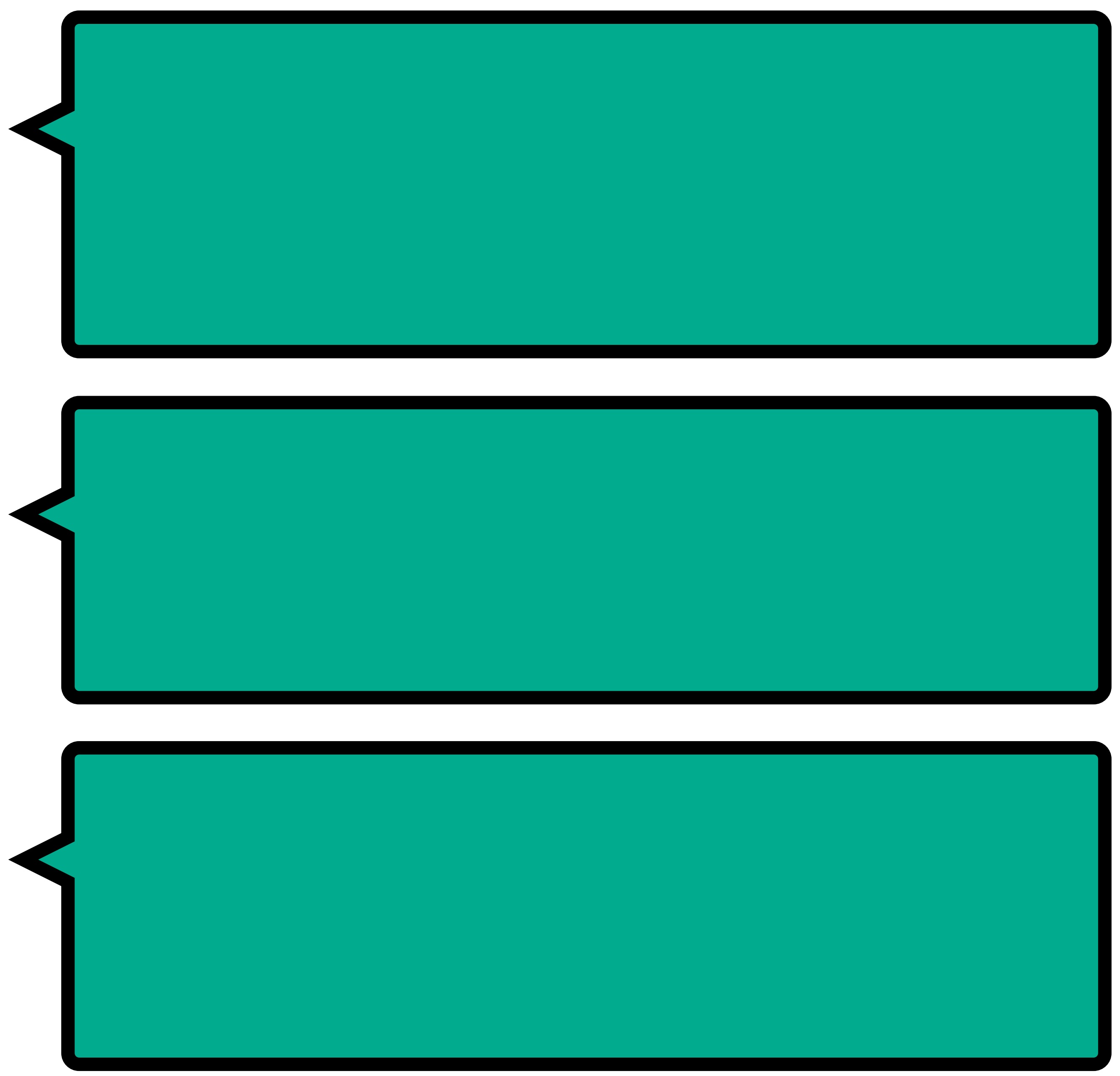
~100 bytes



IoT Metadata

Microsoft Azure sensor metadata

~300 bytes

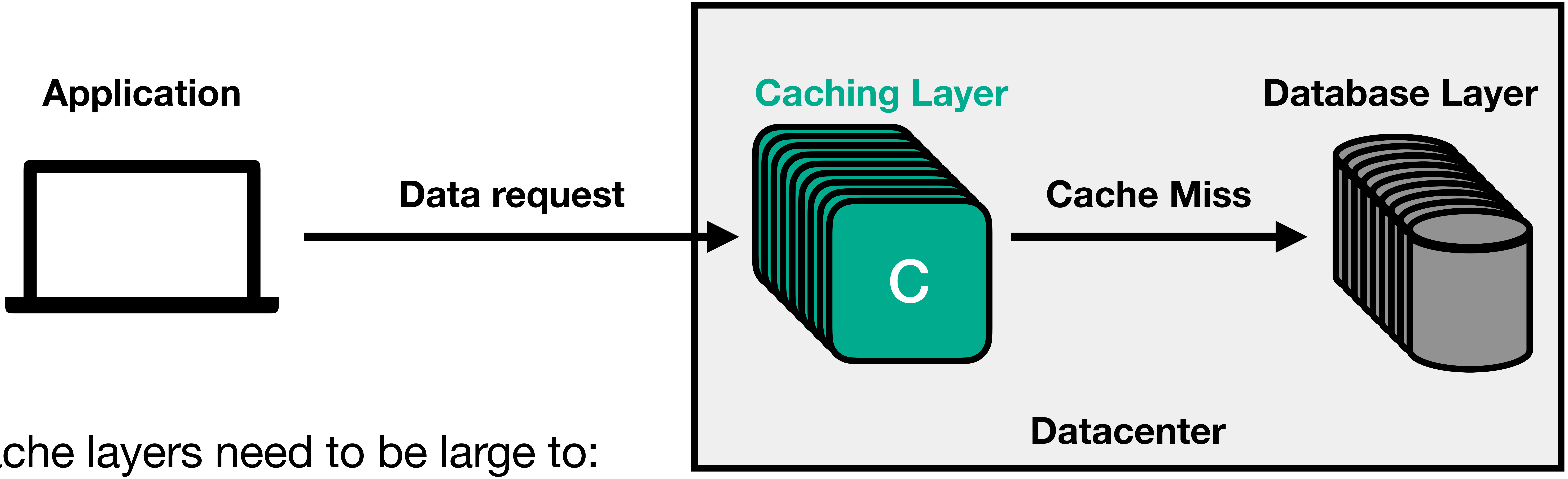


Tweets

Twitter tweets average

<33 characters

Caching at scale



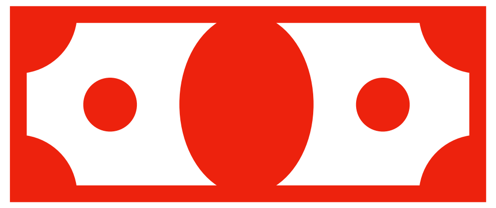
Cache layers need to be large to:

1. lower average latency
2. keep load off of backend services

Flash is 100x cheaper per bit ⟶ Larger caches

Caching **billions of tiny objects** (~100 bytes) on flash

Prior Work → Too many flash writes or Large memory overhead → Wasted Money



Kangaroo reduces misses by 29%

while keeping writes and memory under production constraints

Open source[1] and integrated into CacheLib[2]

[1] github.com/saramcallister/Kangaroo   [2] cachelib.org

Outline

1) Introduction

2) Caching on flash

3) Minimizing DRAM overhead

4) Kangaroo design

5) Results

Caching on flash → Additional challenge

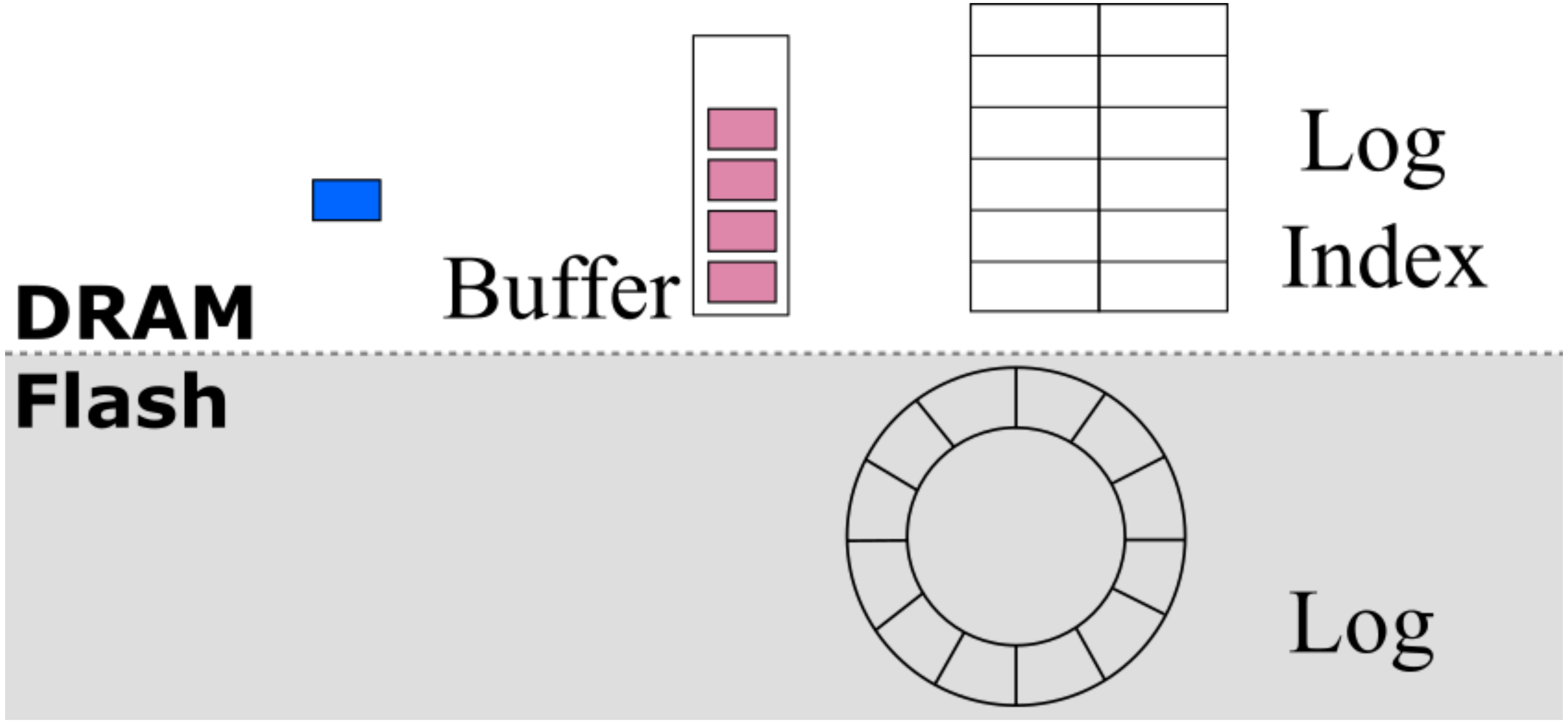
Flash allows cheaper than DRAM, but

- Flash has **limited write endurance**
- Caches have to write in > 4 KB blocks

Most flash caches use a **log-structured cache**

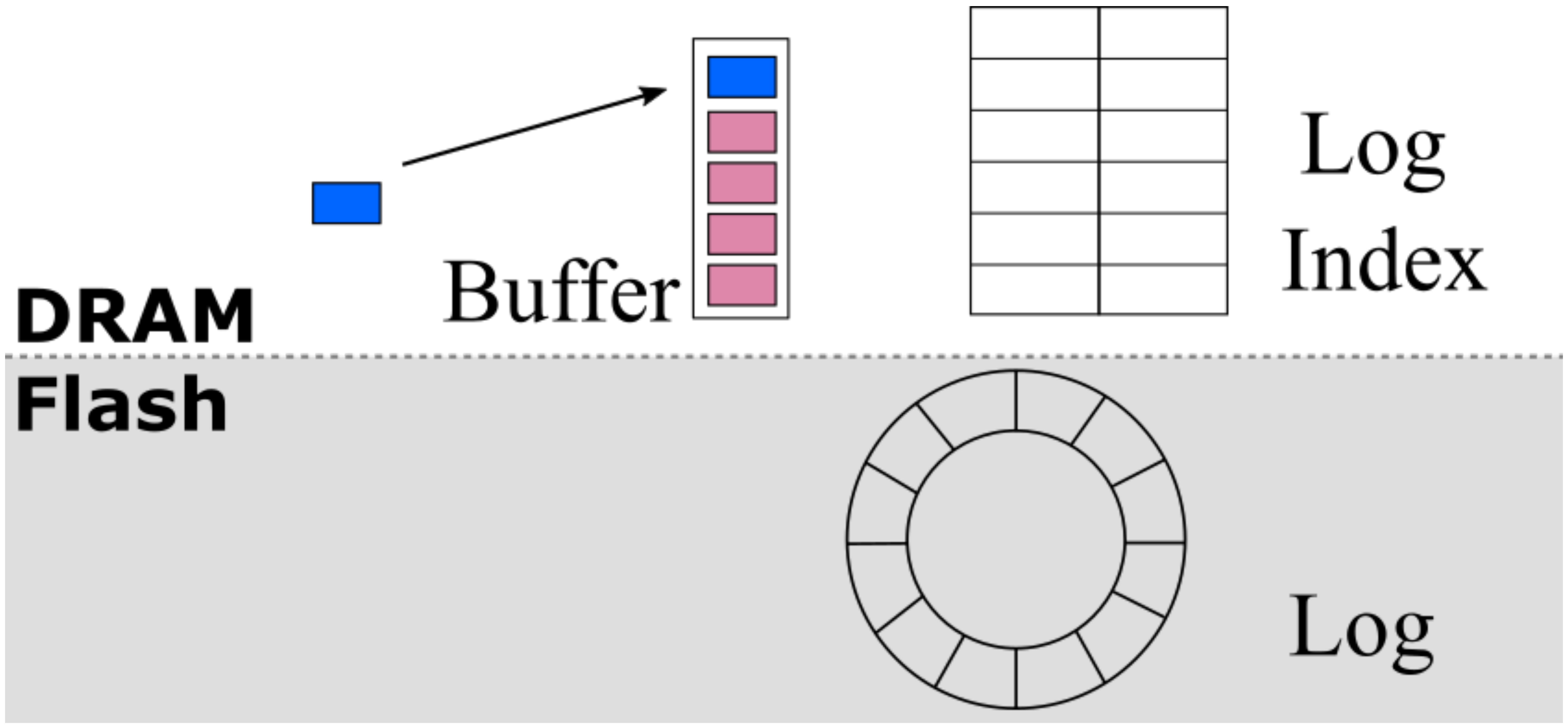
Log-structured Caches

Flashield (Eisenman NSDI '19), FASTER (Chandramouli SIGMOD'18), RIPQ (Tang FAST'15)



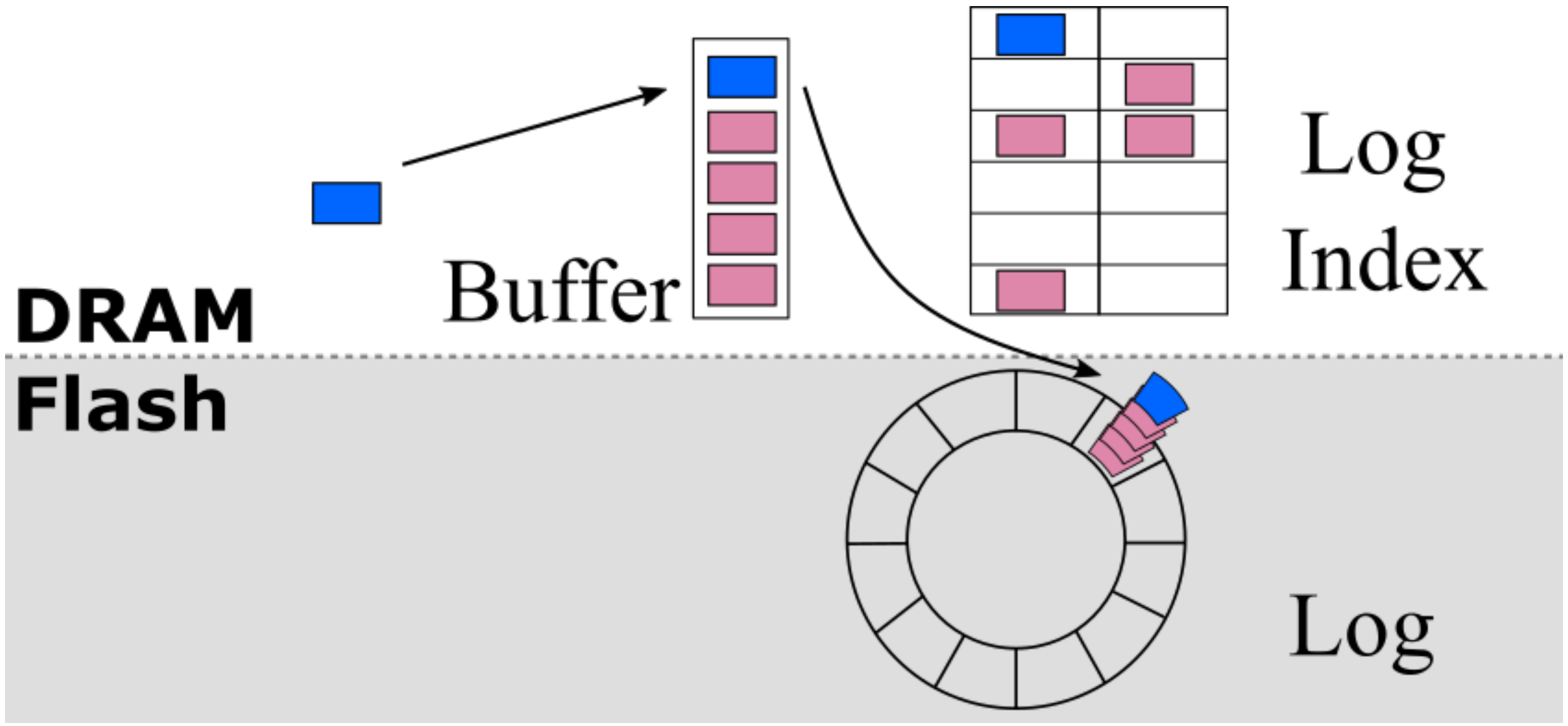
Log-structured Caches

Flashield (Eisenman NSDI ’19), FASTER (Chandramouli SIGMOD’18), RIPQ (Tang FAST’15)



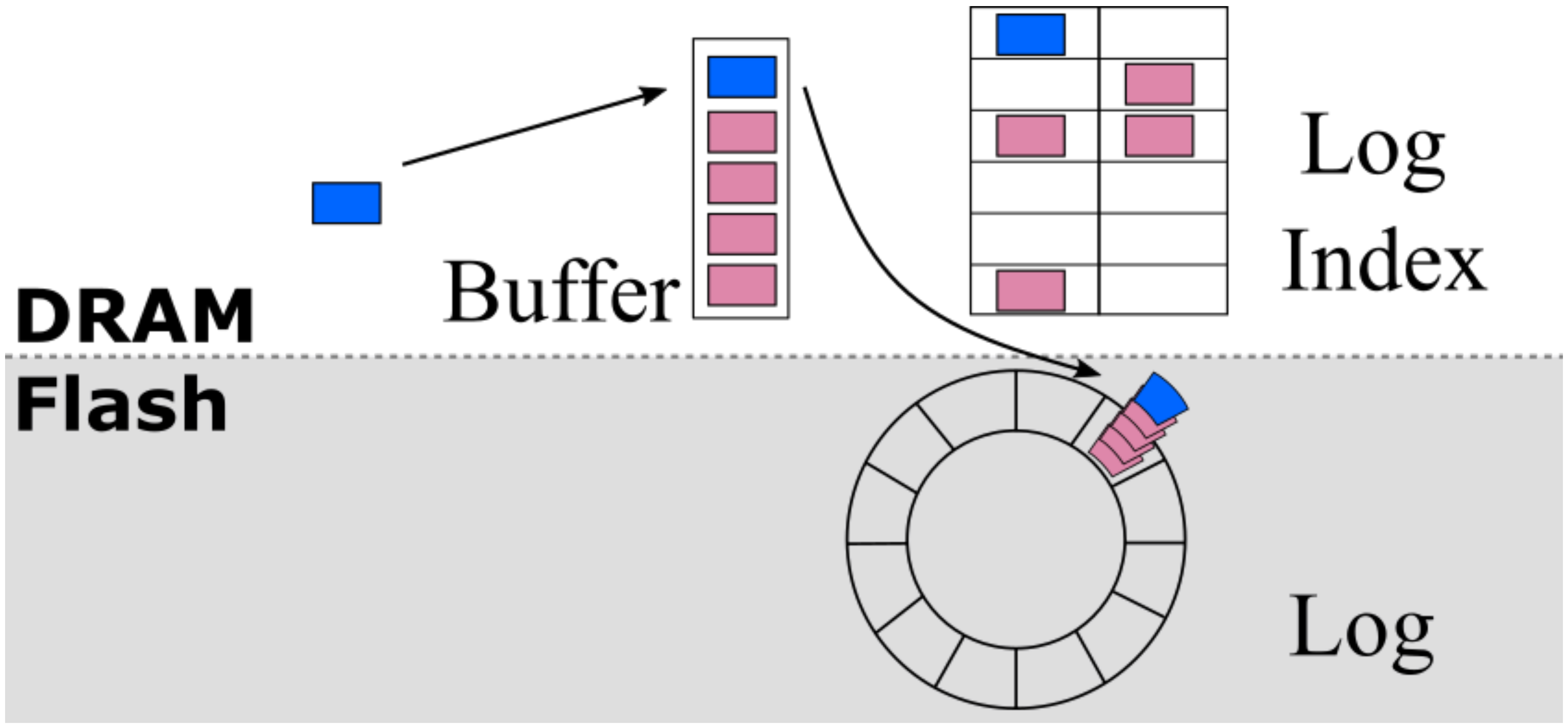
Log-structured Caches

Flashield (Eisenman NSDI '19), FASTER (Chandramouli SIGMOD'18), RIPQ (Tang FAST'15)



Log-structured Caches

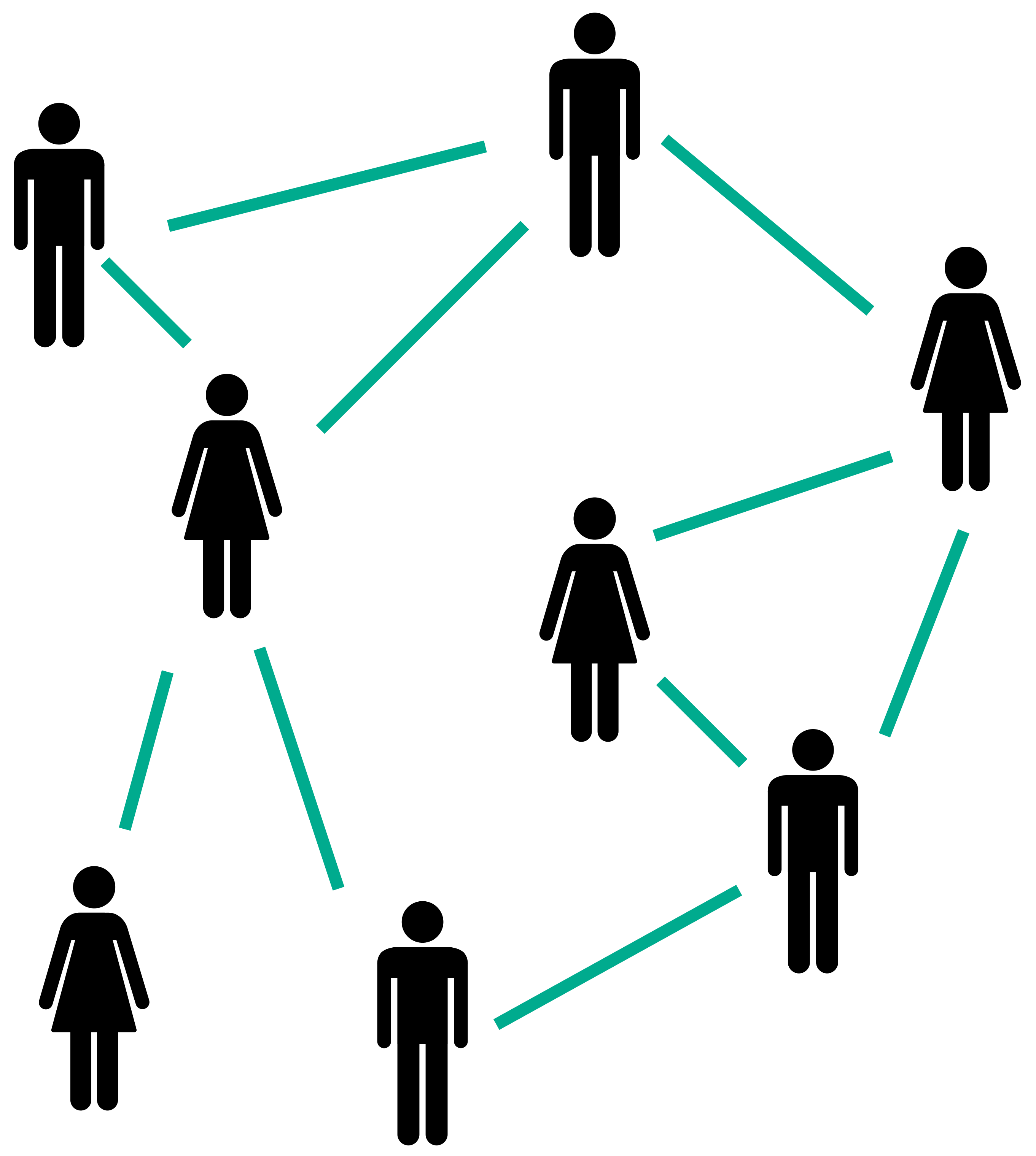
Flashield (Eisenman NSDI ’19), FASTER (Chandramouli SIGMOD’18), RIPQ (Tang FAST’15)



+ **Buffered writes** minimize writes to flash

- **Full in-memory index**

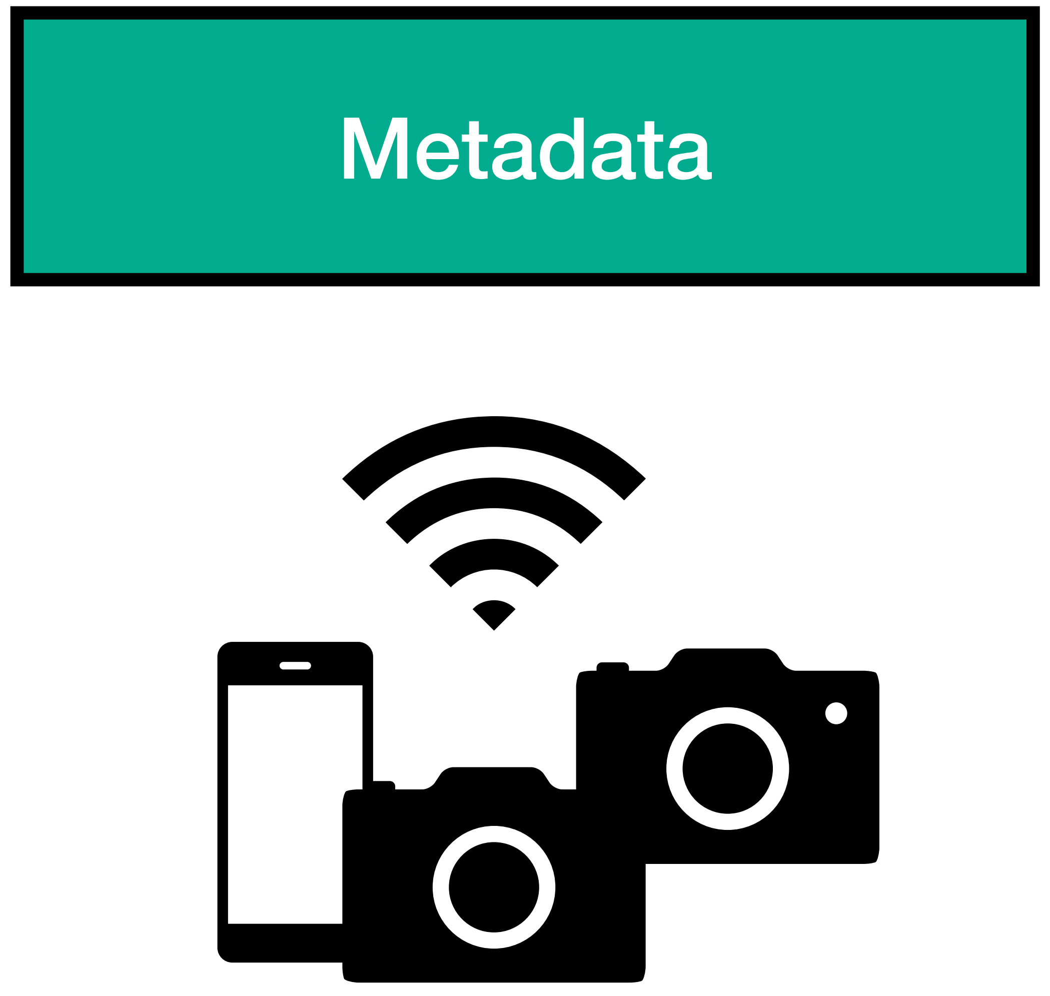
Need to cache tiny objects



Social Graphs

Facebook social graph edges

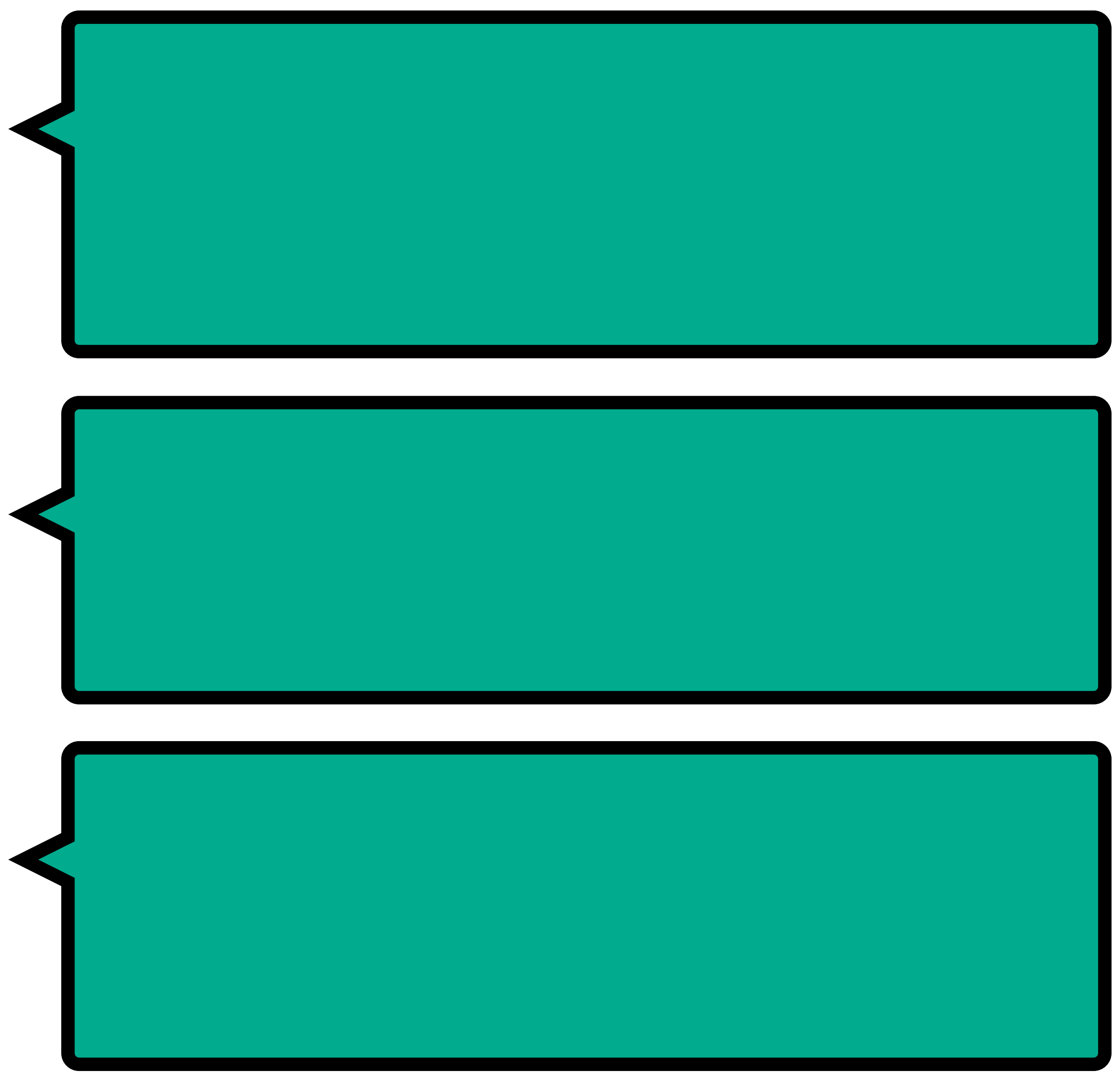
~100 bytes



IoT Metdata

Microsoft Azure sensor metadata

~300 bytes



Tweets

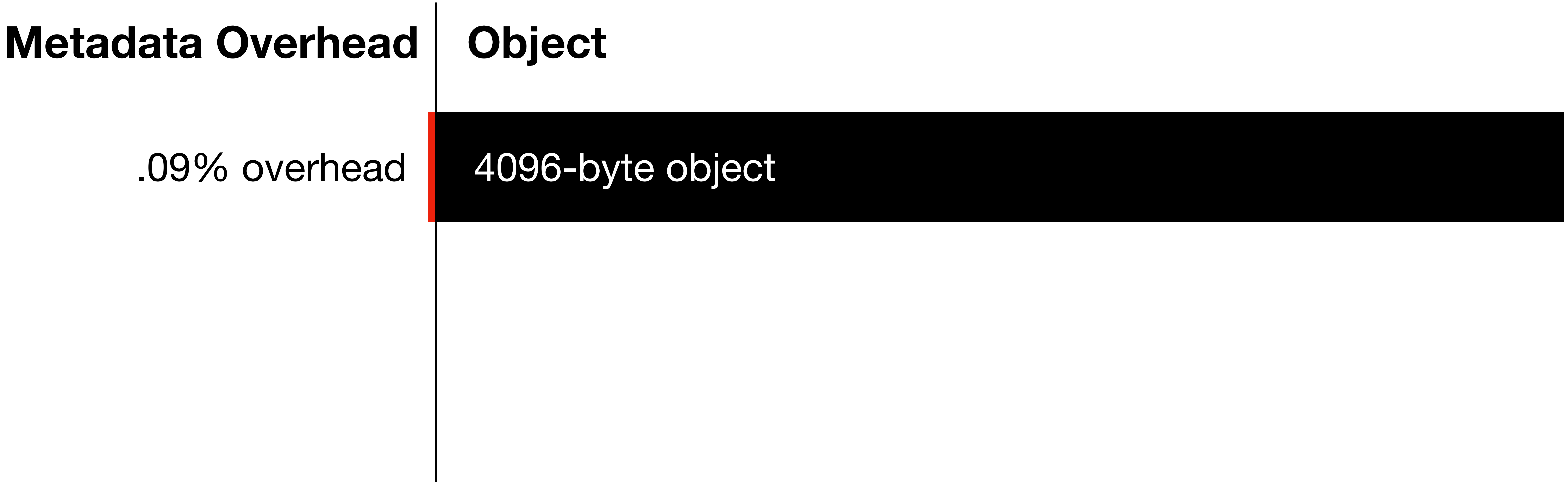
Twitter tweets average

<33 characters

Tiny objects → Large metadata overheads

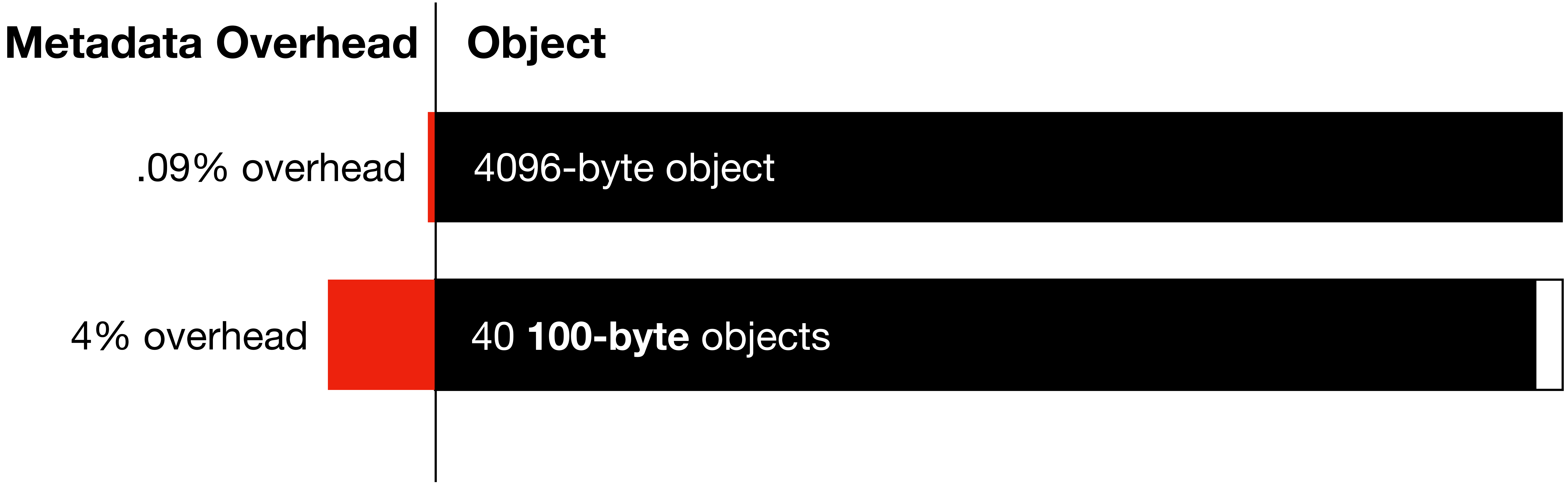
30 bits / object metadata overhead

Flashield (Eisenman NSDI ’19)



Tiny objects → Large metadata overheads

30 bits / object metadata overhead
Flashield (Eisenman NSDI ’19)



2 TB flash cache → 75 GB memory overhead

Outline

1) Introduction

2) Caching on flash

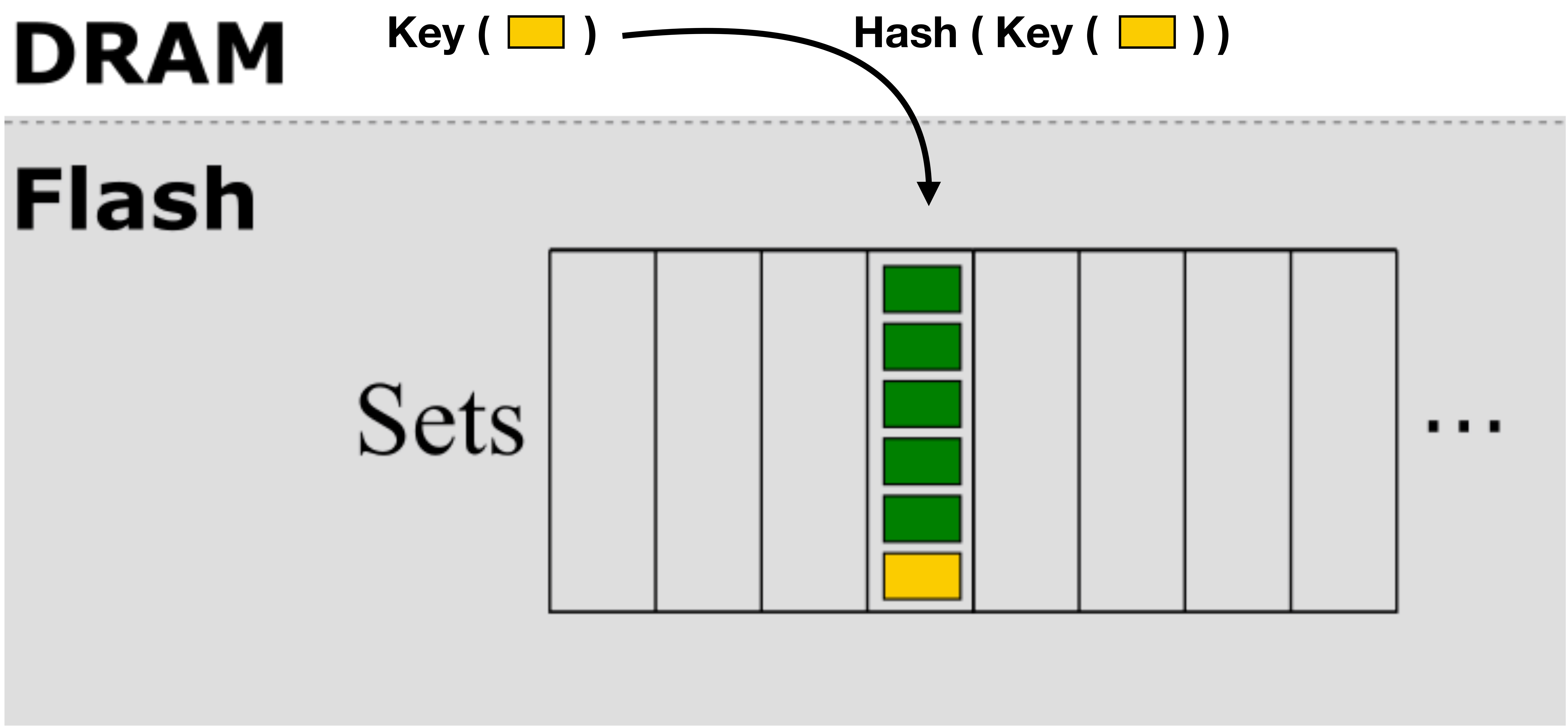
3) Minimizing DRAM overhead

4) Kangaroo design

5) Results

Low memory overhead → Set-associative cache

CacheLib (Berg OSDI '20)

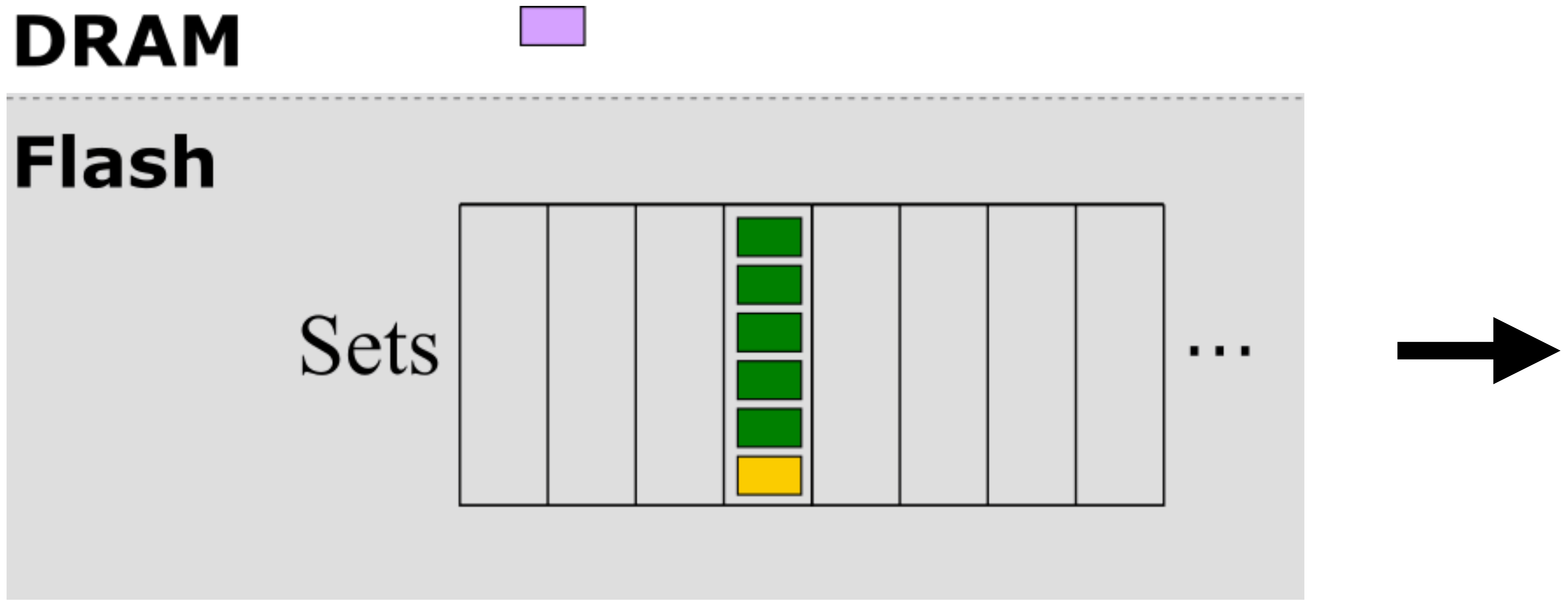


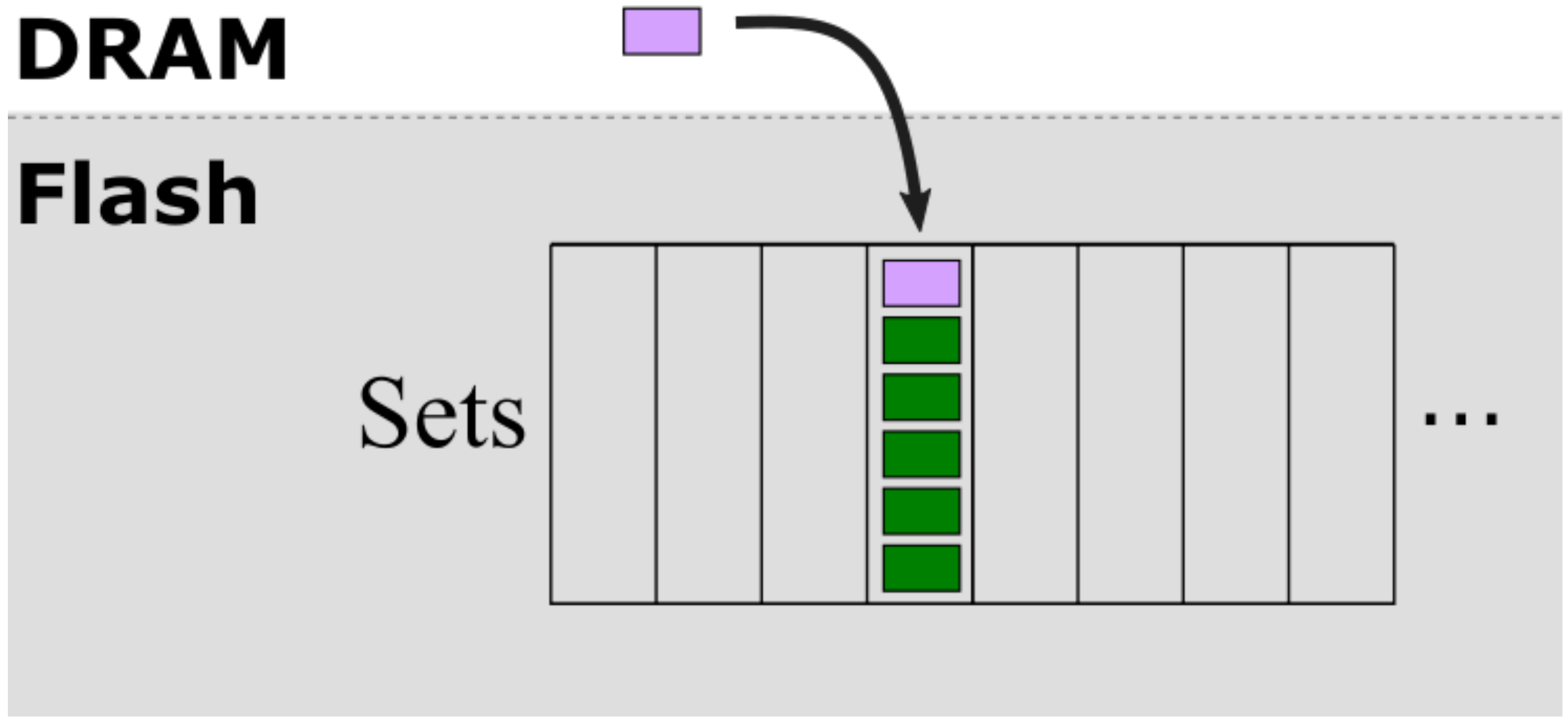
Set-Associative Cache

CacheLib (Berg OSDI '20)

Set-Associative Cache

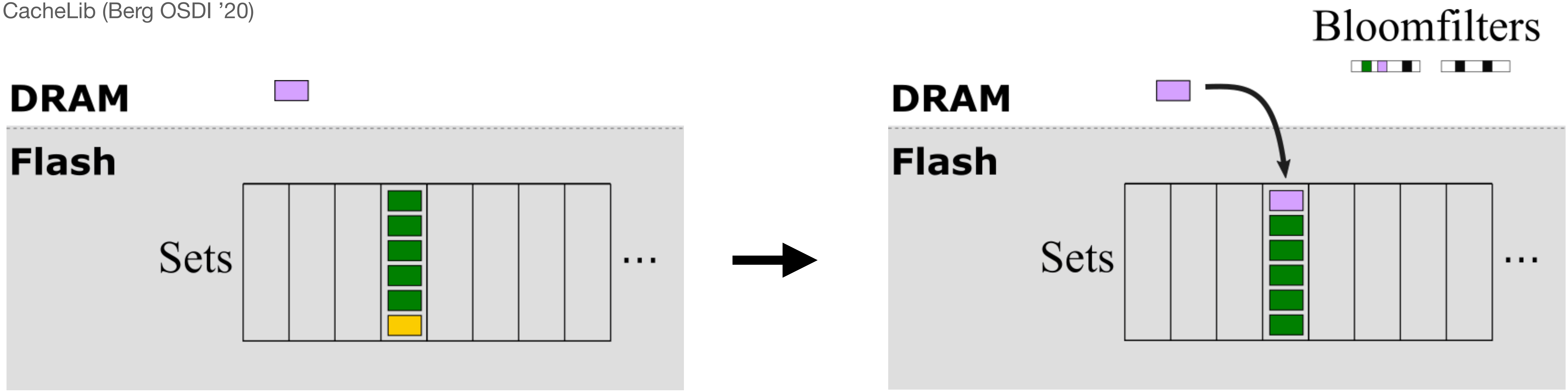
CacheLib (Berg OSDI '20)





Set-Associative Cache

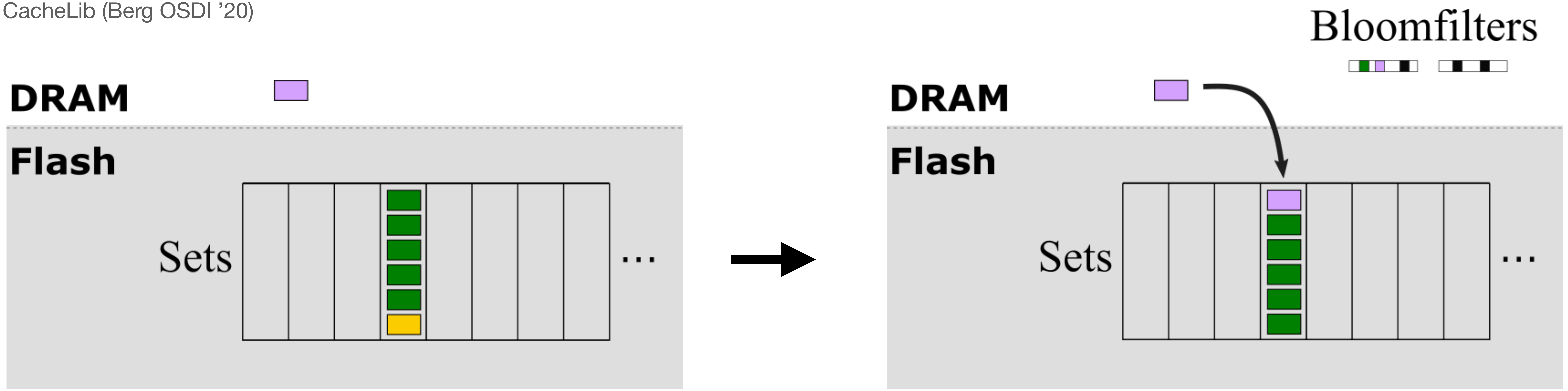
CacheLib (Berg OSDI '20)



+ Low memory overhead

Set-Associative Cache

CacheLib (Berg OSDI '20)

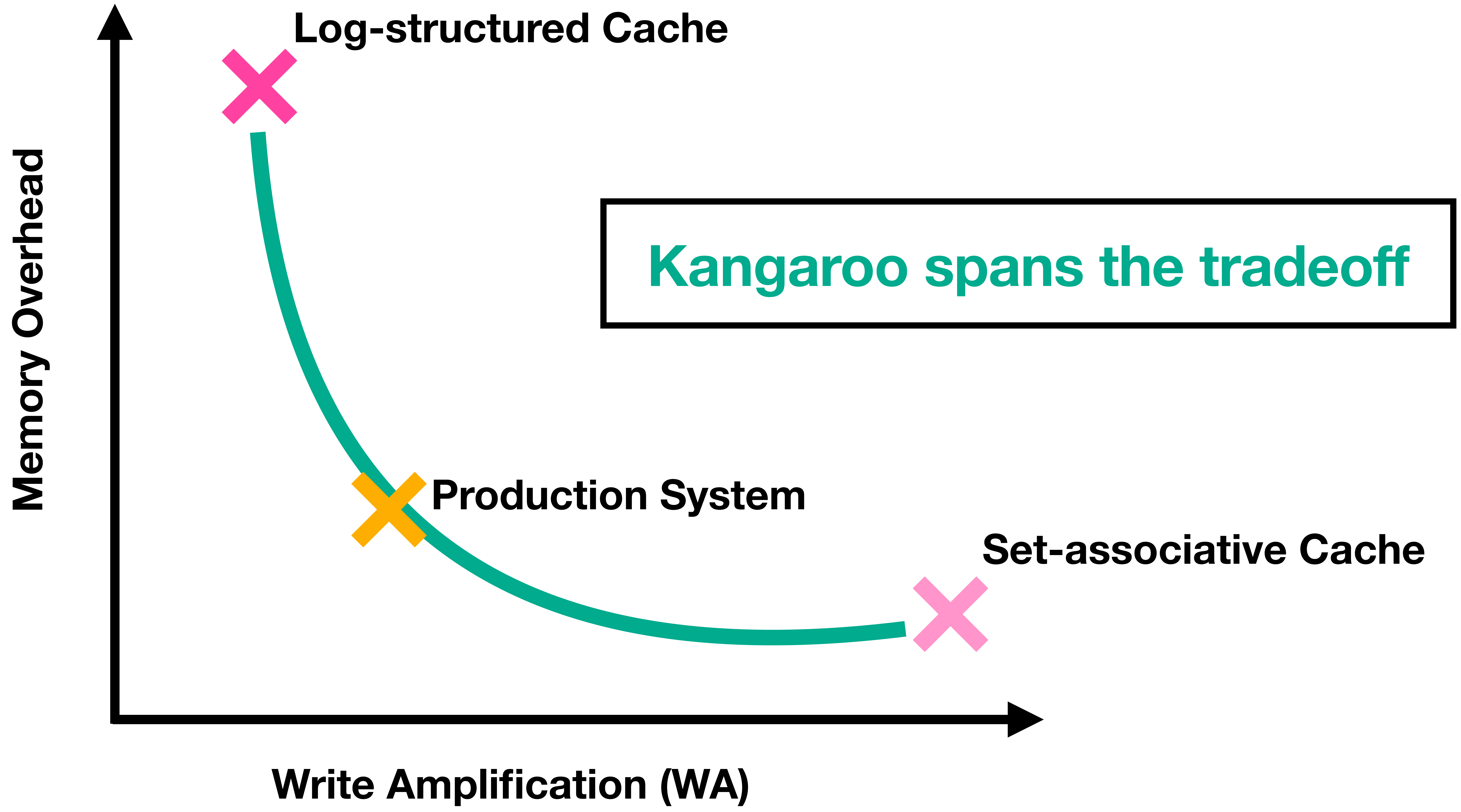


+ **Low memory overhead**

- **Large write amplification** (# bytes written / bytes requested)

$$\text{Write Amplification} = \frac{4096 \text{ bytes}}{100 \text{ bytes}} = \sim 40\text{x}$$

Prior work: Too much DRAM or too many writes



Outline

1) Introduction

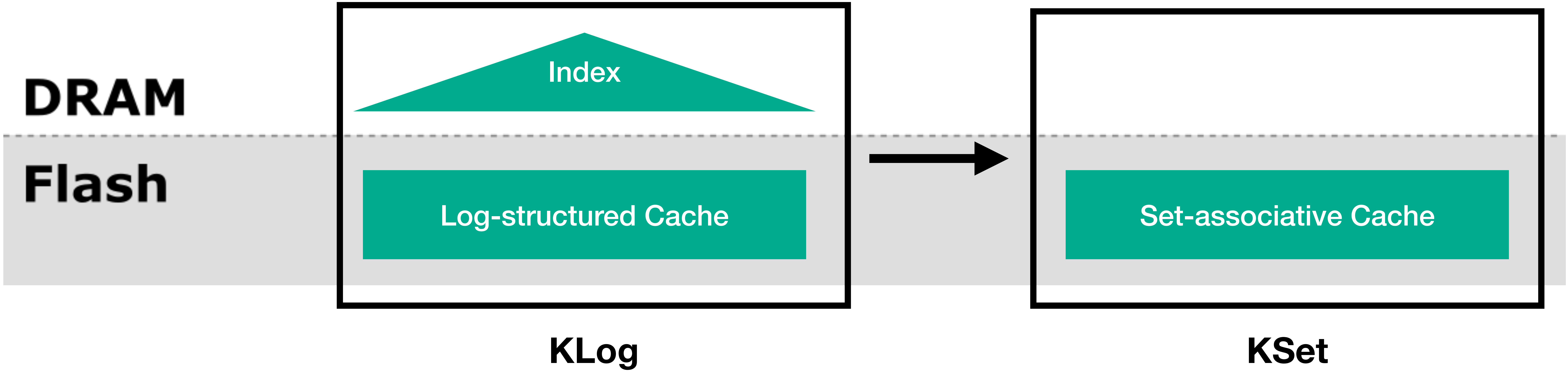
2) Caching on flash

3) Minimizing DRAM overhead

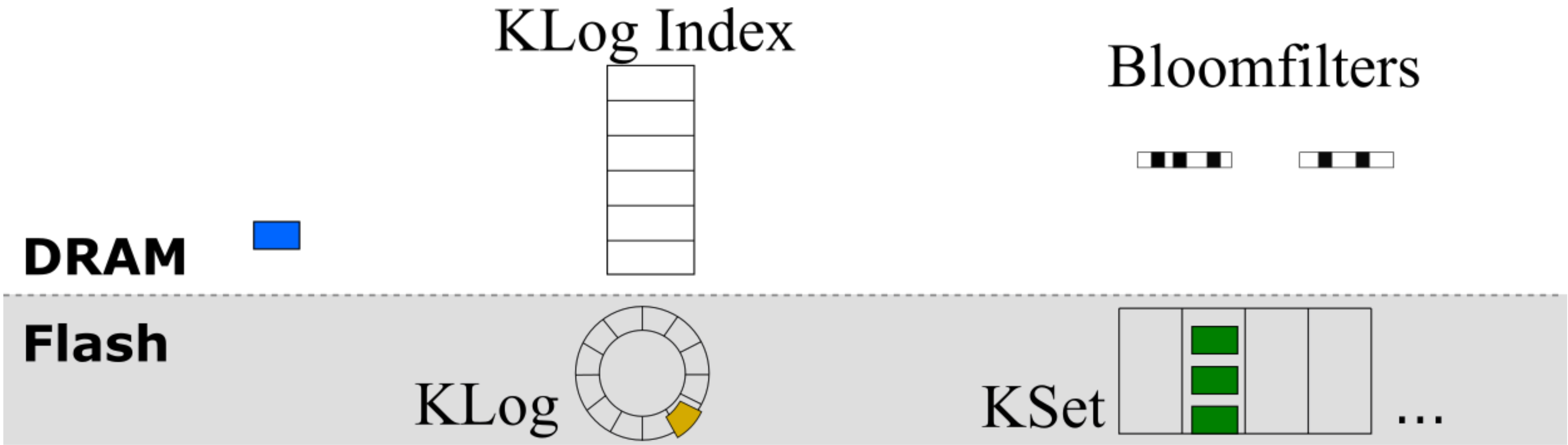
4) Kangaroo design

5) Results

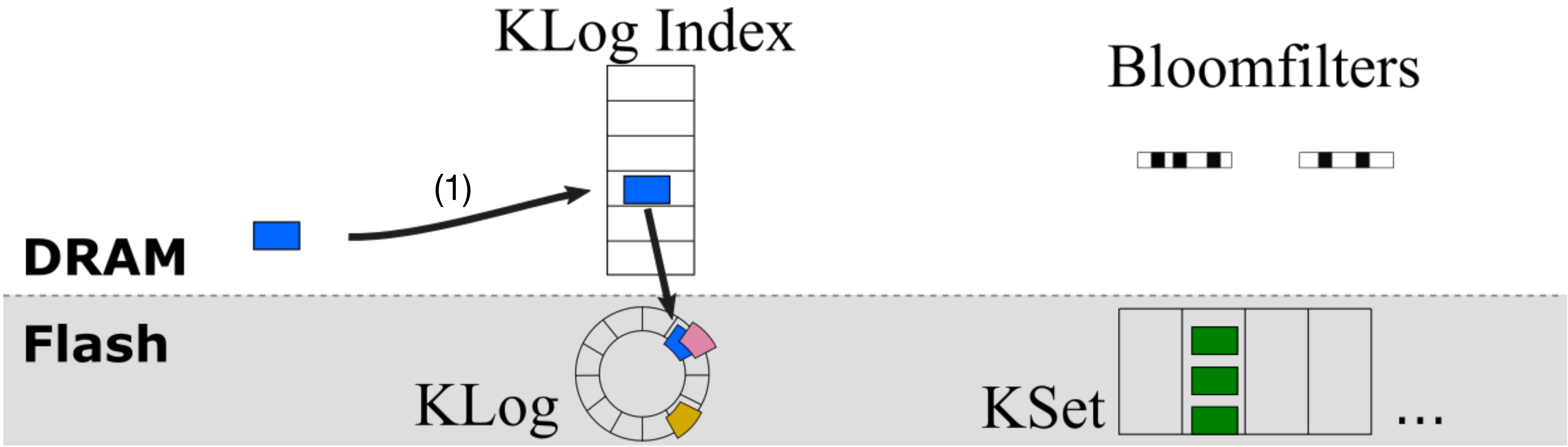
Kangaroo Overview



Inserting Objects in Kangaroo

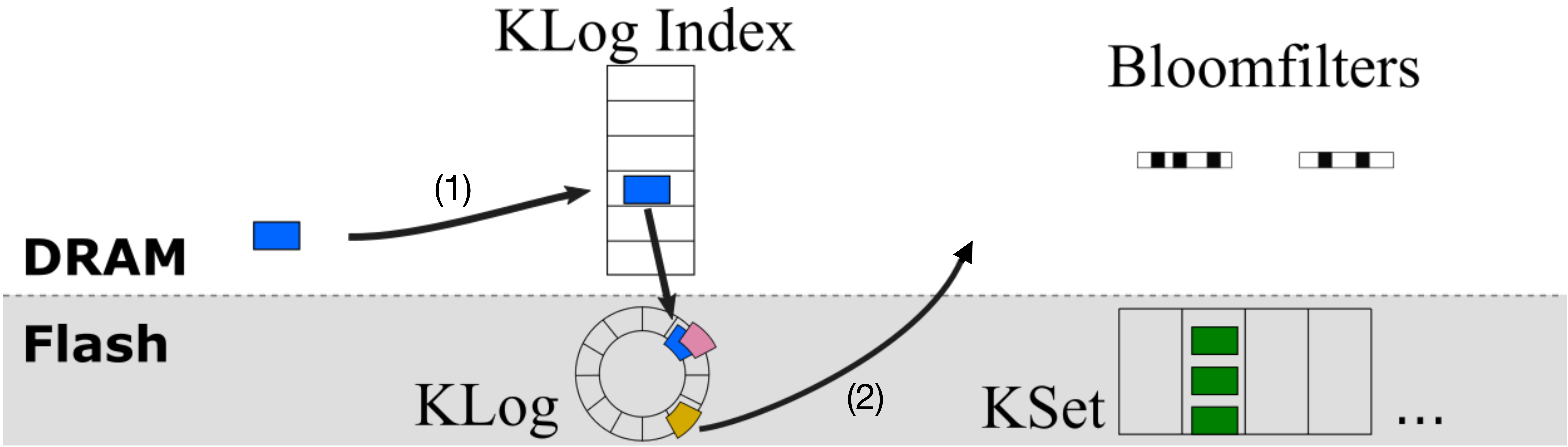


Inserting Objects in Kangaroo



1) Insert to KLog via buffered write

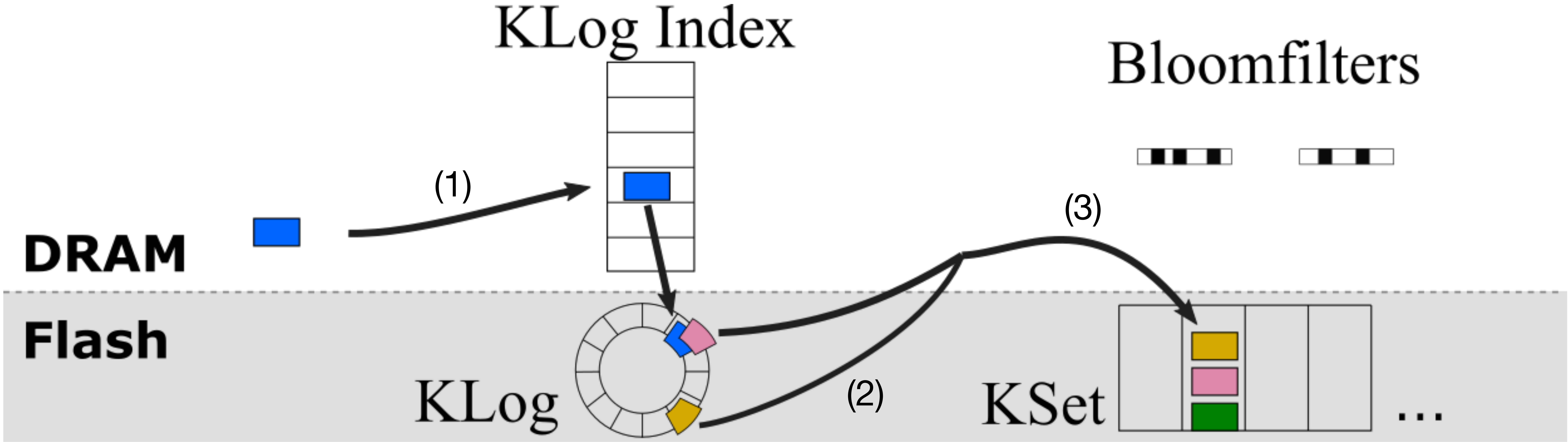
Inserting Objects in Kangaroo



1) Insert to KLog via buffered write

2) Flush object from KLog to KSet

Inserting Objects in Kangaroo



1) Insert to KLog via buffered write

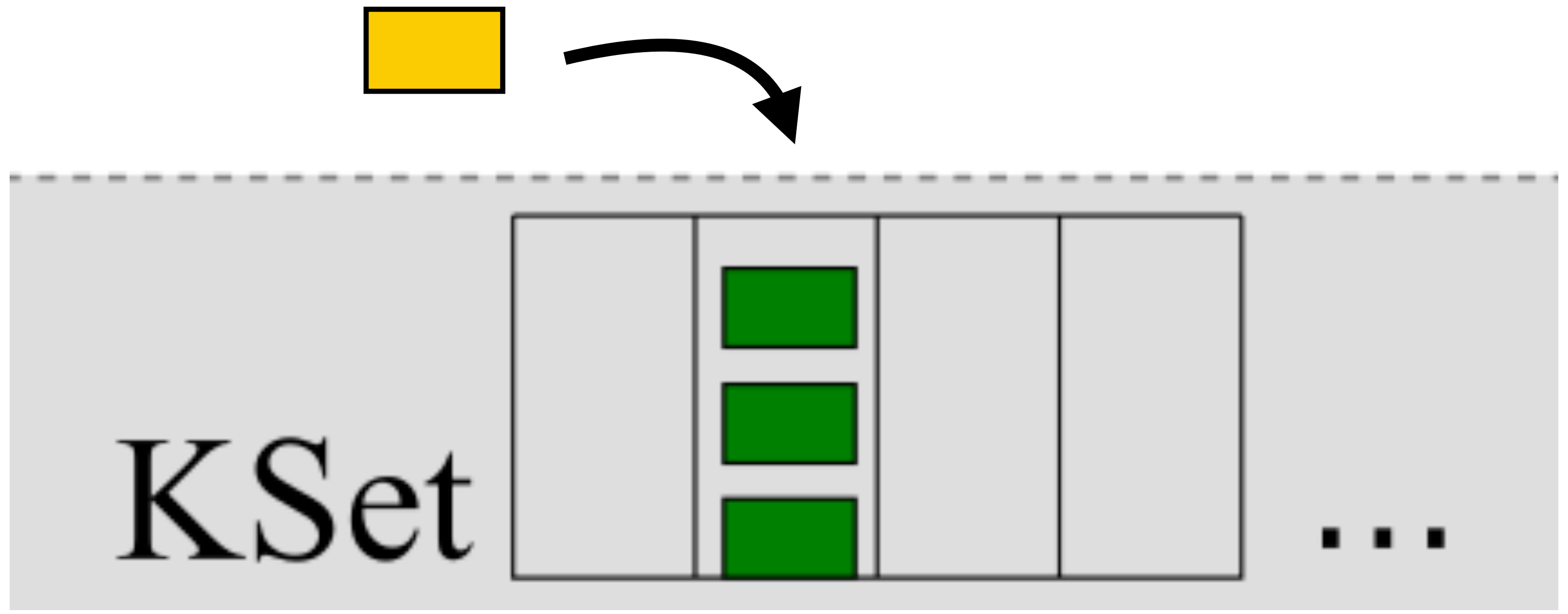
2) Flush object from KLog to KSet

3) Move **all objects** in KLog that map to the same set

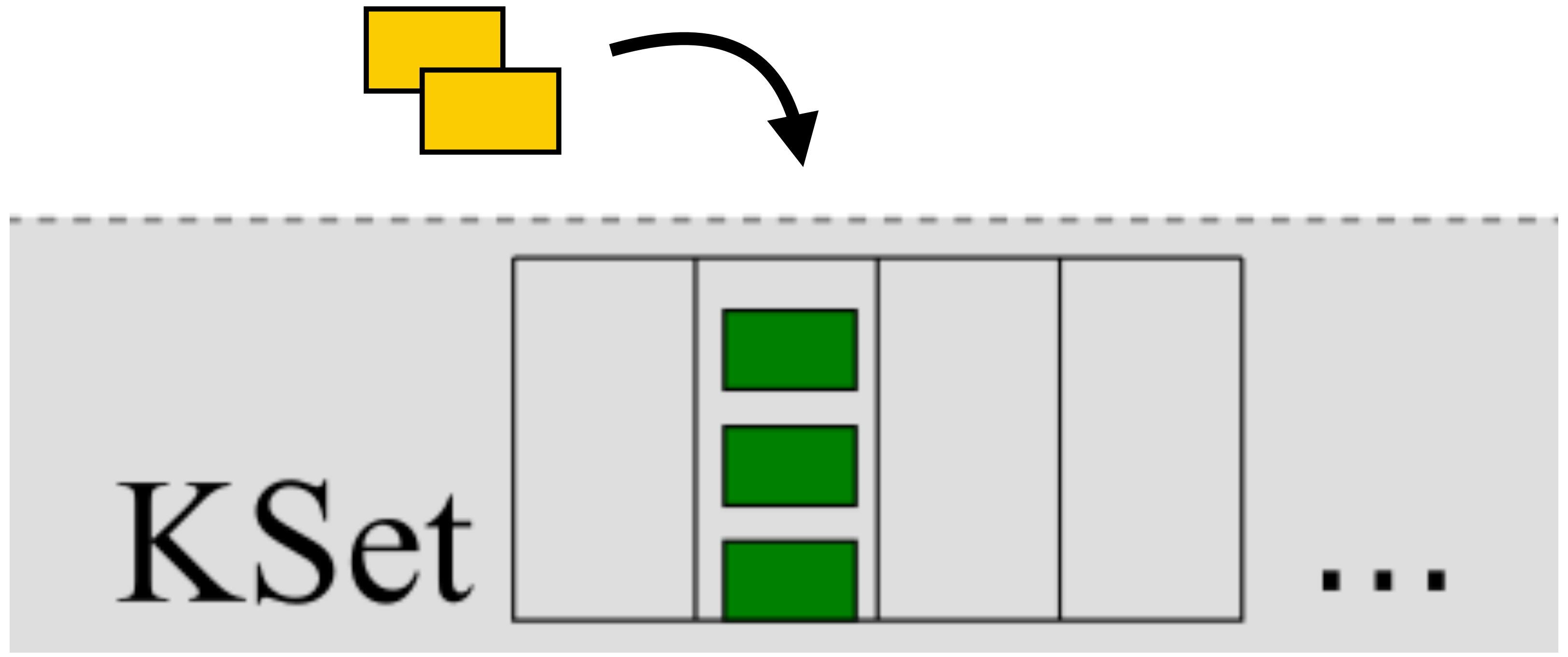
Amortizing KSet flash writes using KLog

Two small objects **halve** write amplification (WA) to KSet

1 new object

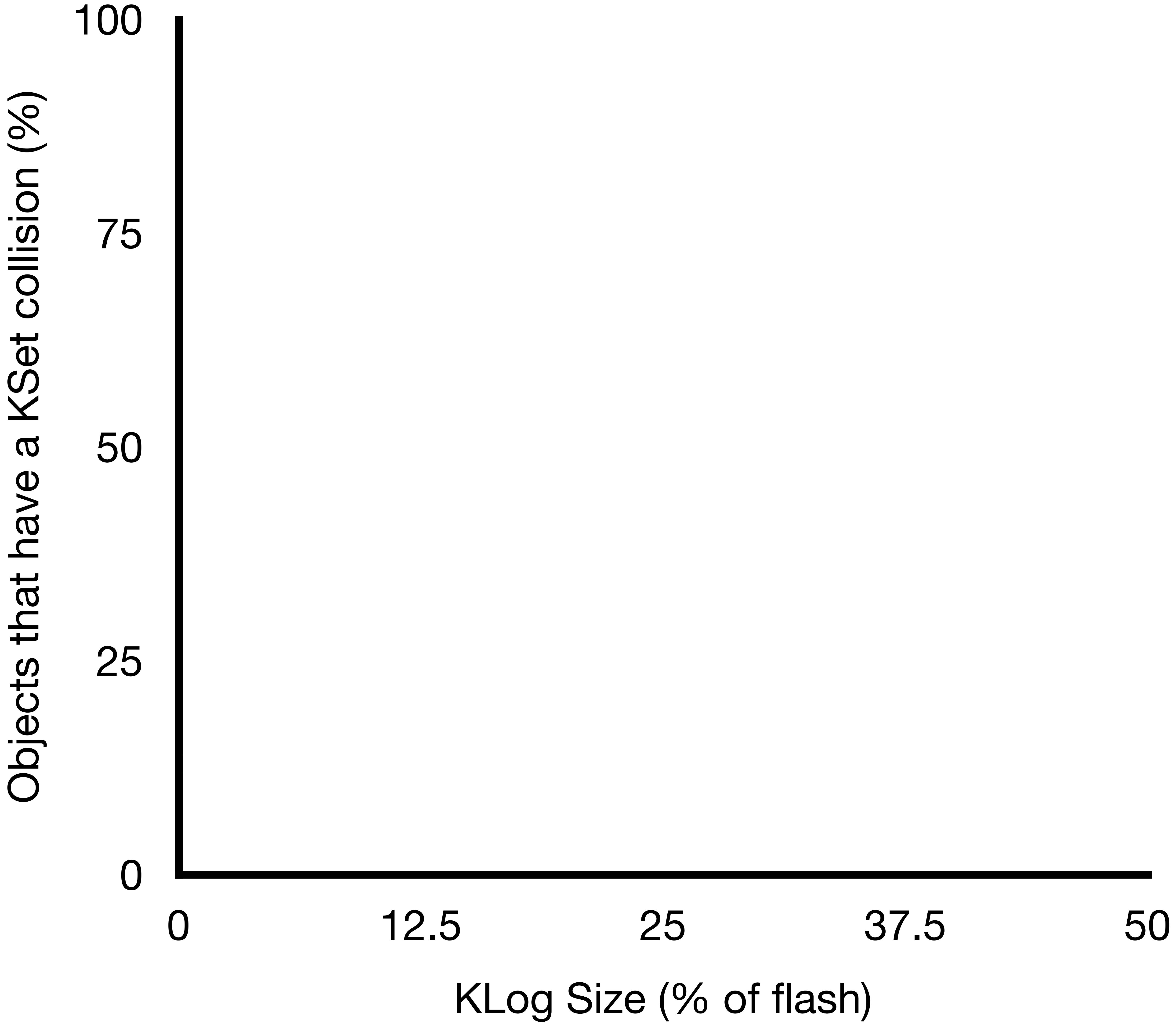


2 new objects

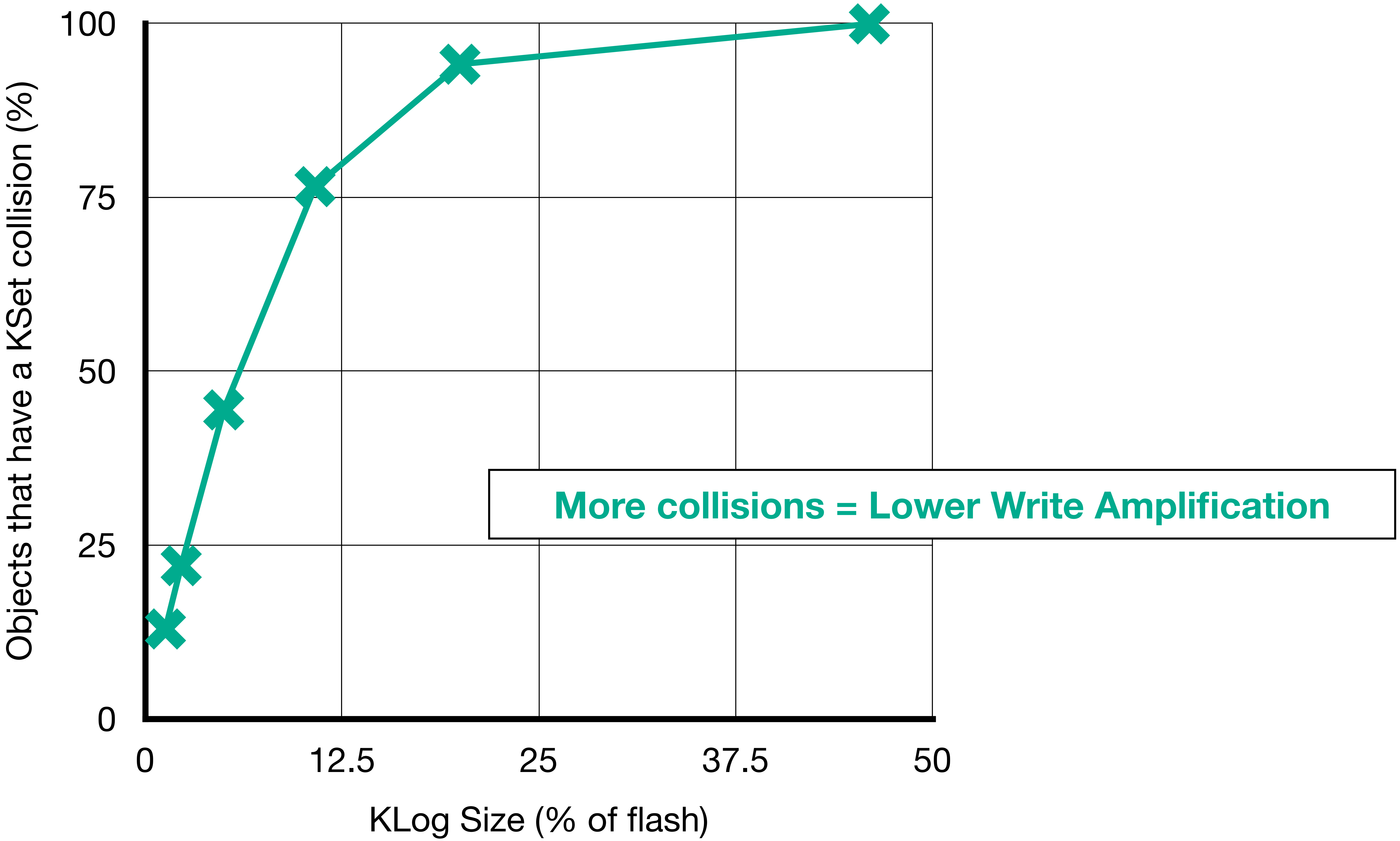


KLog allows more time to find **set collisions** and **amortize WA**

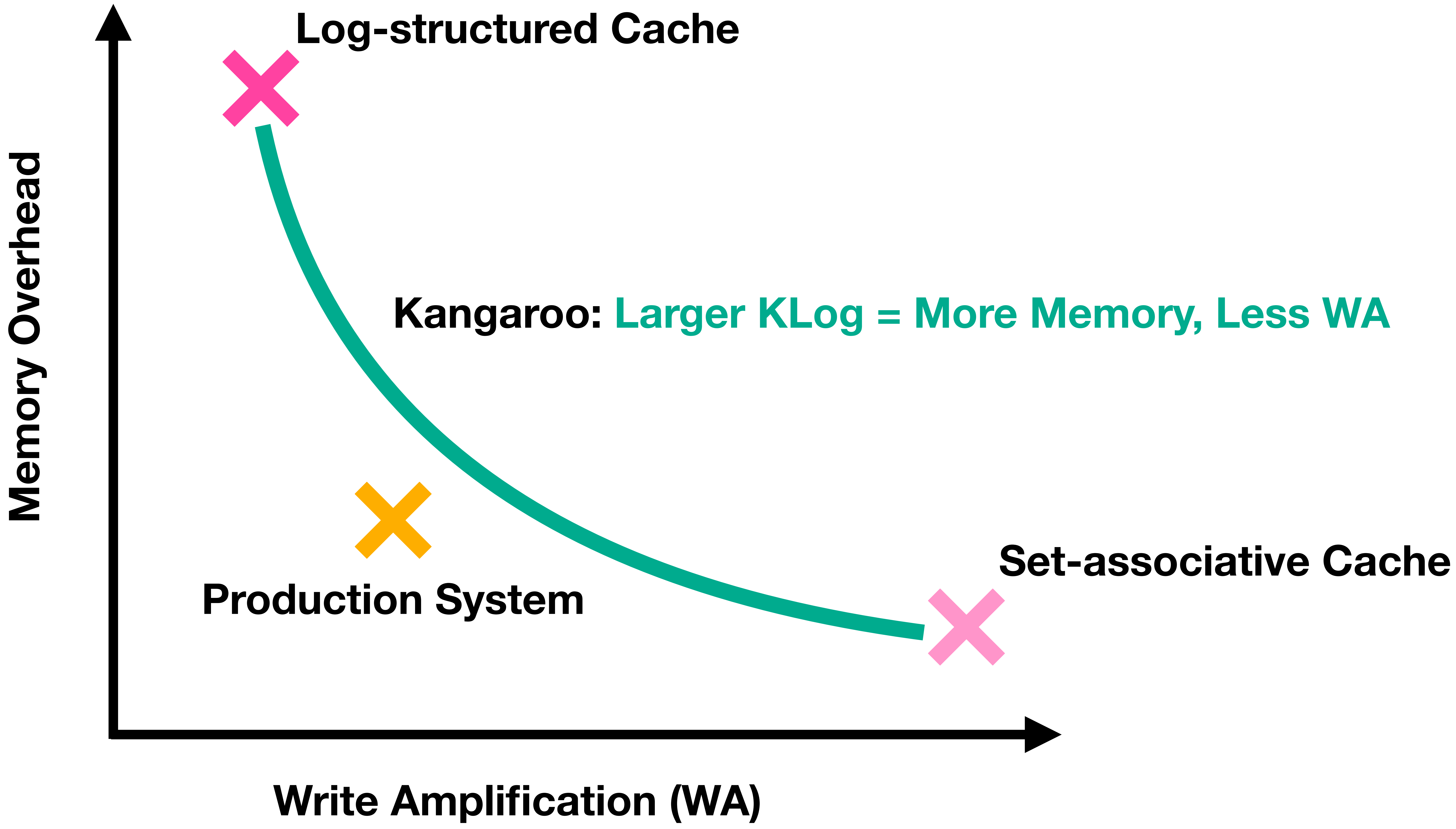
Small KLog → Large Probability of Collision



Small KLog → Large Probability of Collision

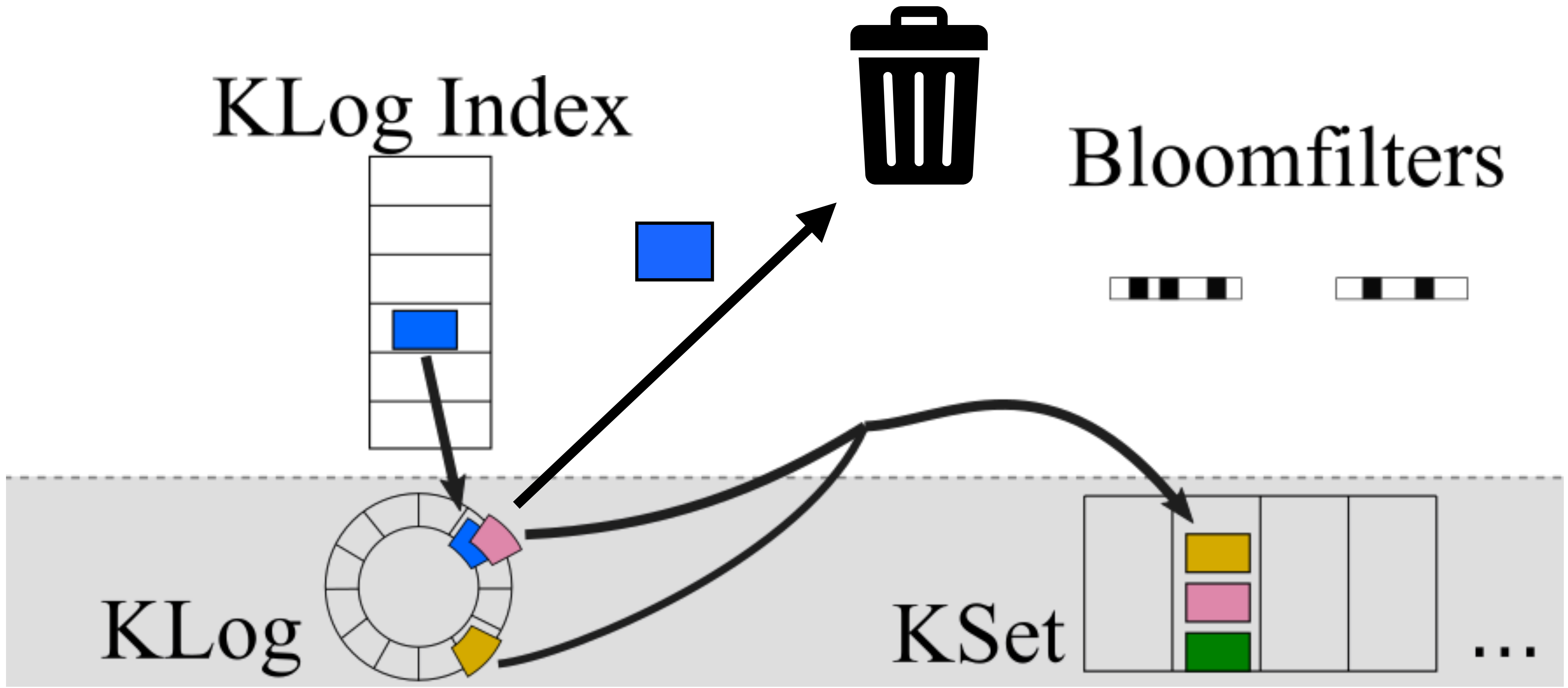


Kangaroo can trade off overheads



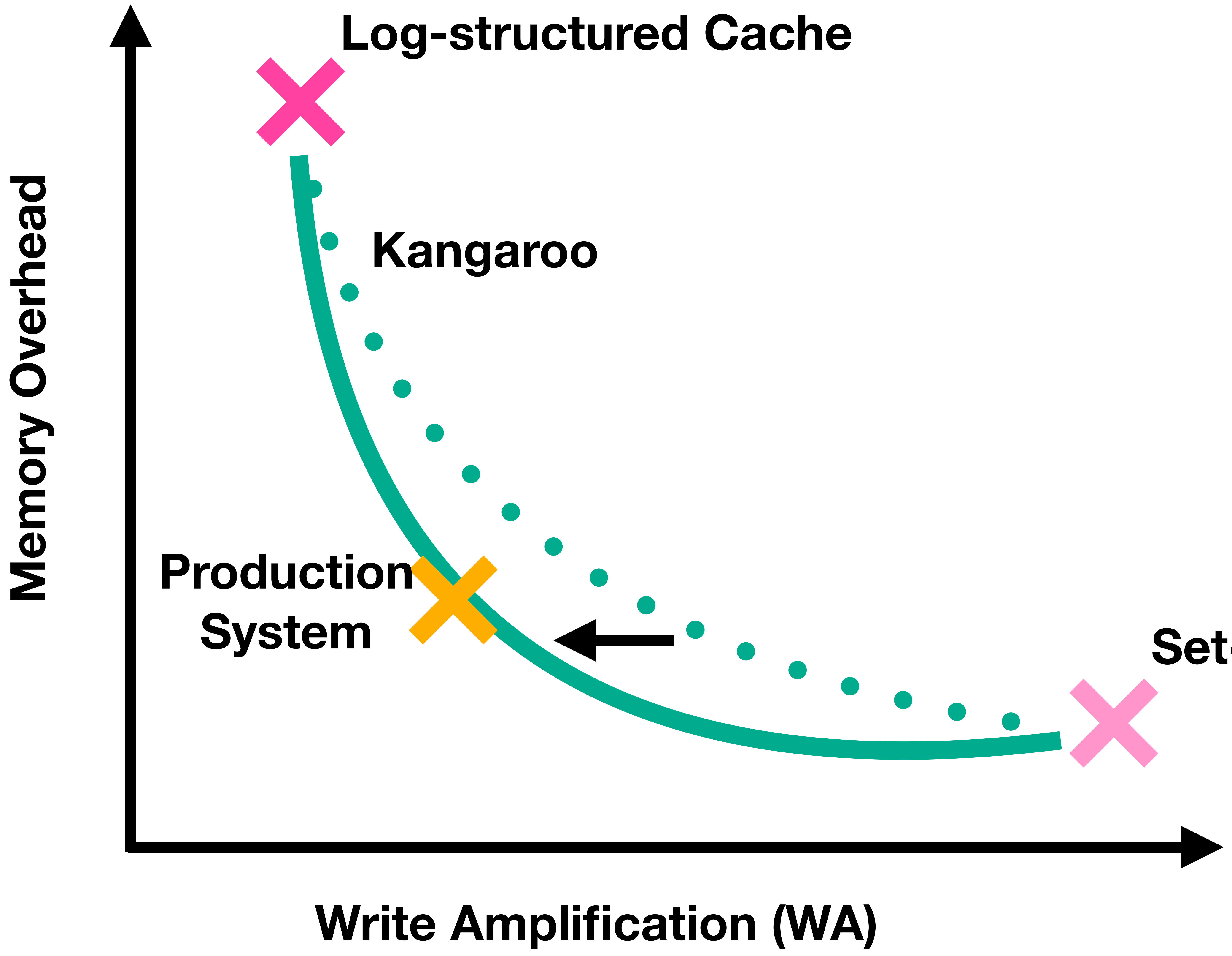
Threshold admission

We can **choose** which objects to discard based on write cost



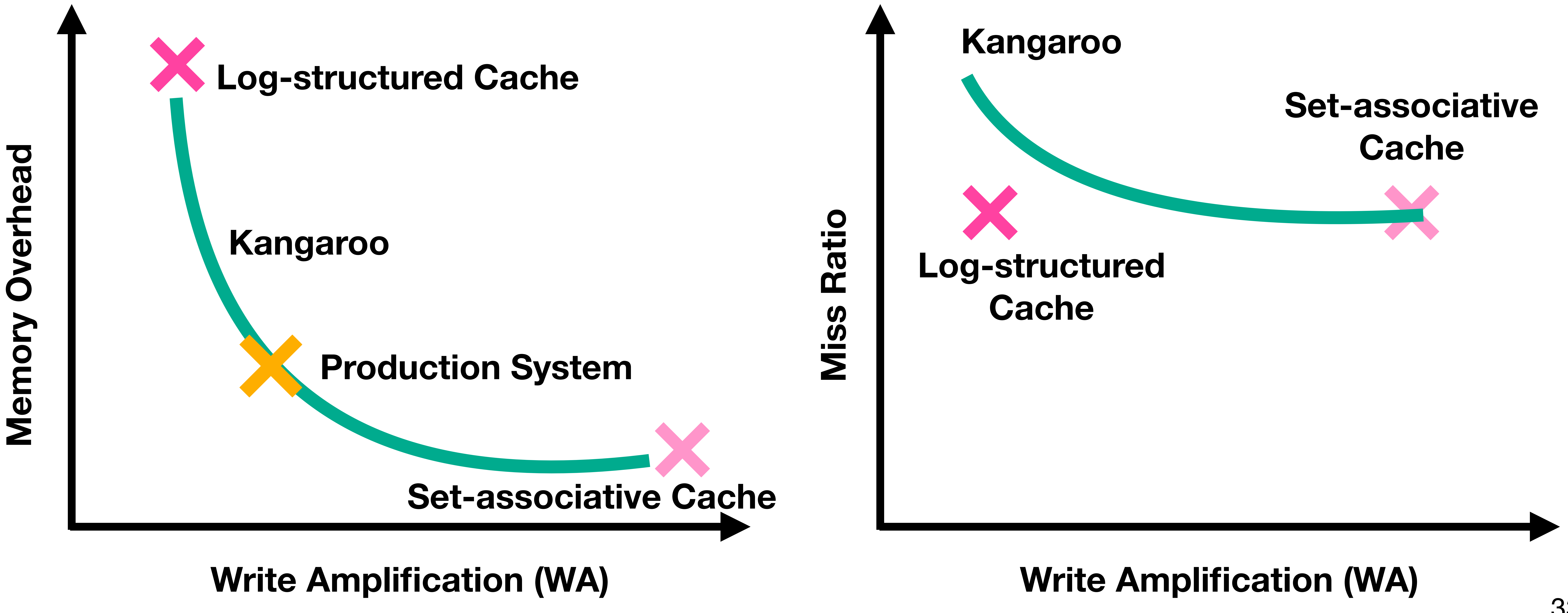
Only rewrite a set in KSet if at least threshold, **n**, number of objects

Threshold admission improves WA



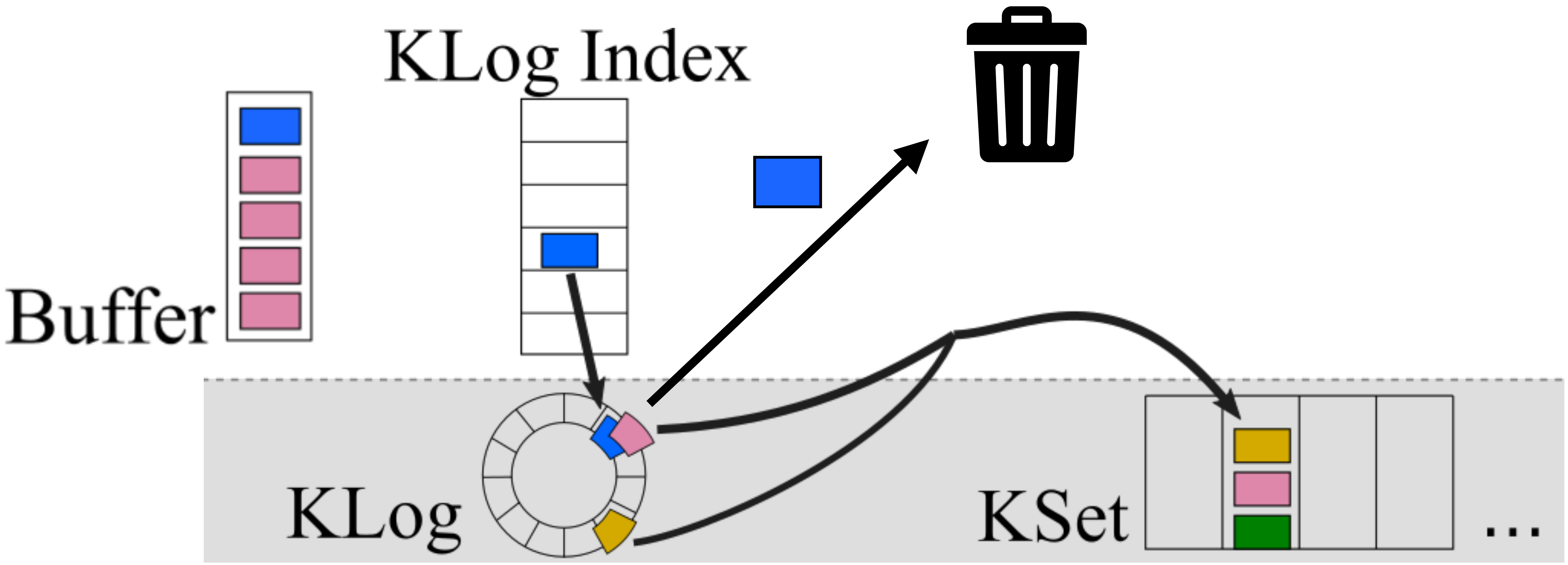
Miss ratio: Another tradeoff

Does discarding objects cause miss ratio losses?



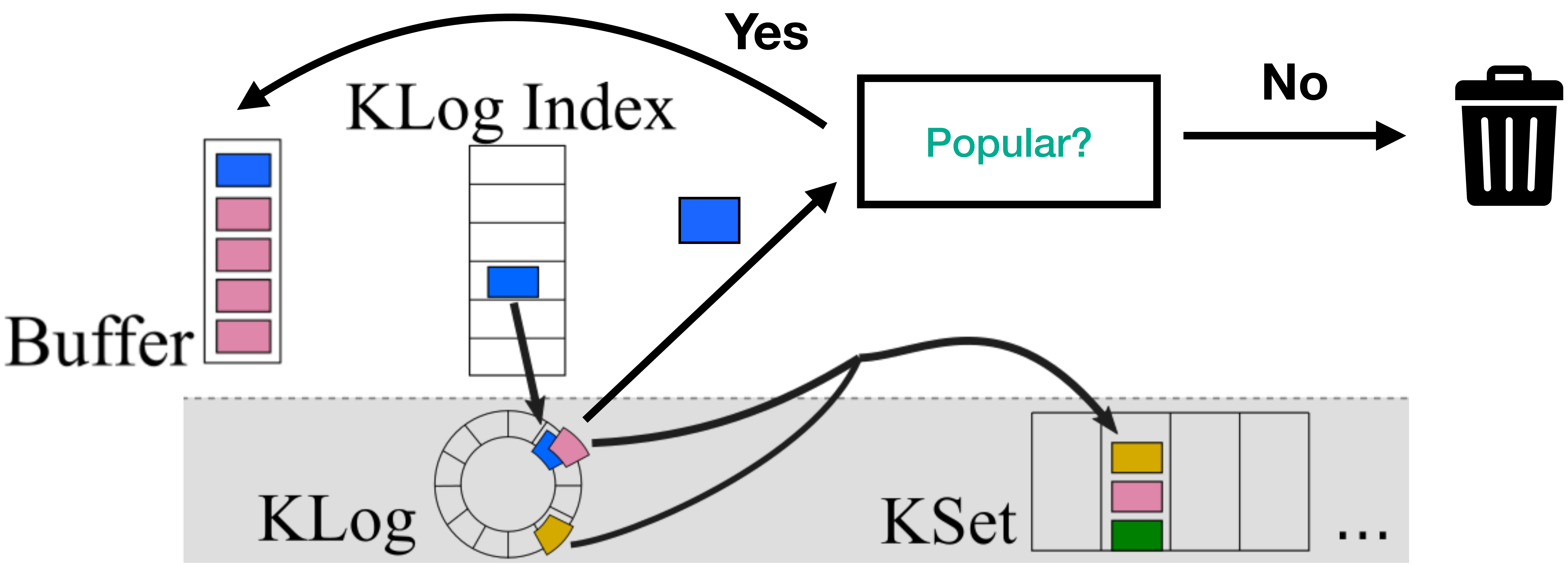
Readmission to KLog

Popular objects rewritten to KLog to minimize write cost

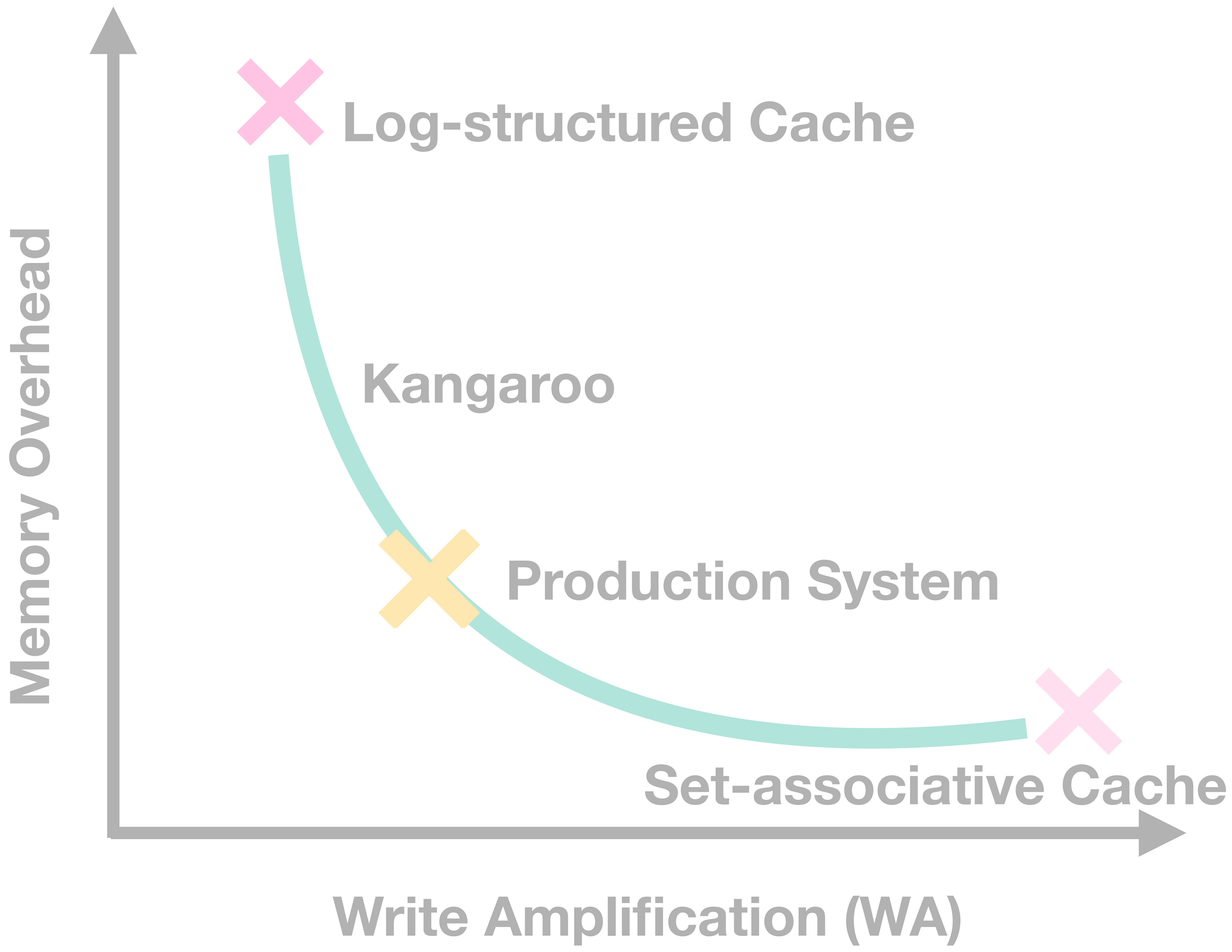


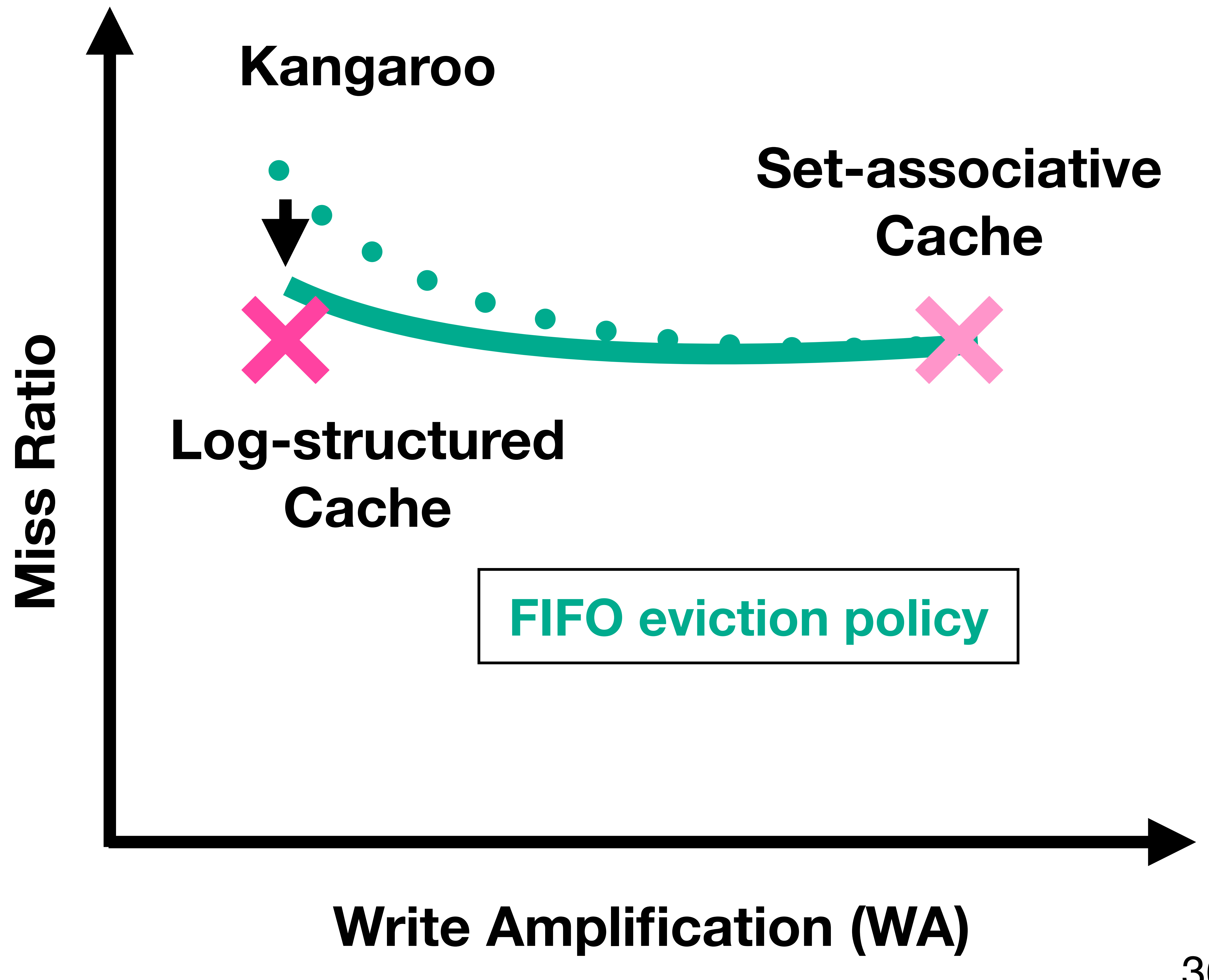
Readmission to KLog

Popular objects rewritten to KLog to minimize write cost

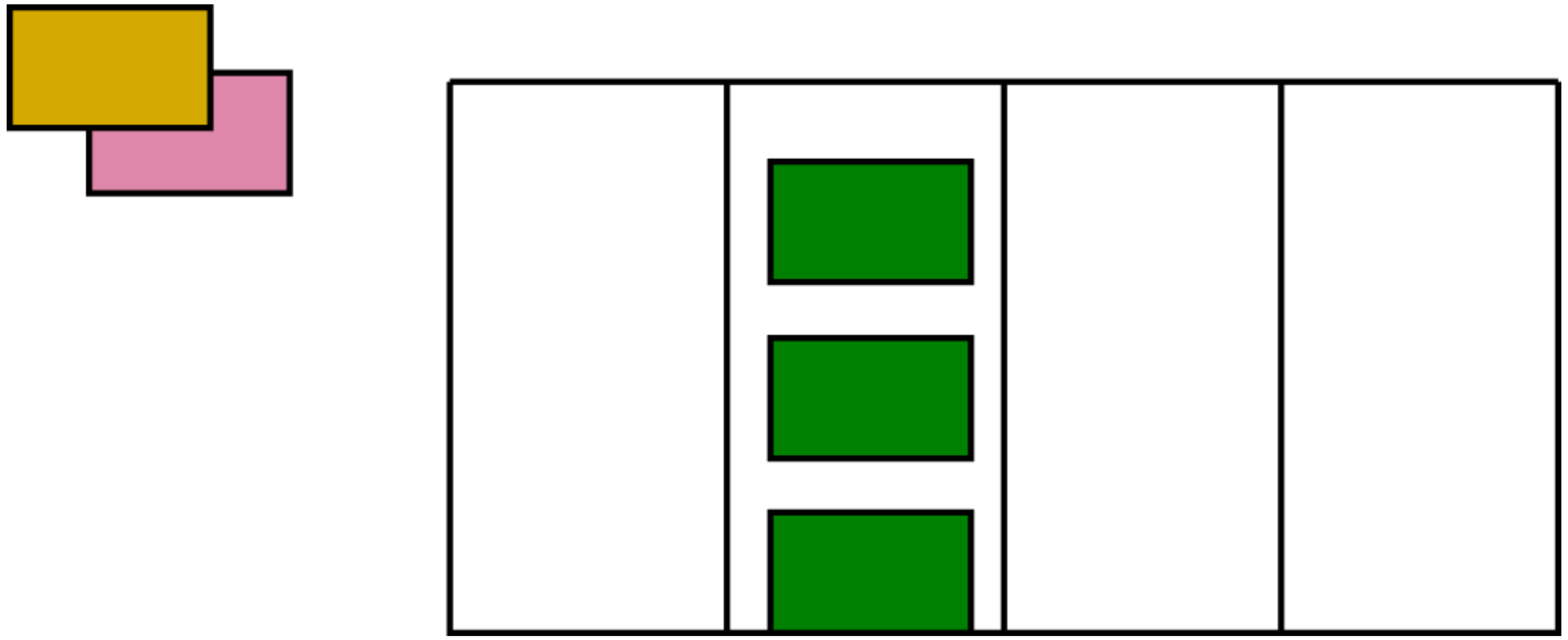


Readmission improves miss ratio





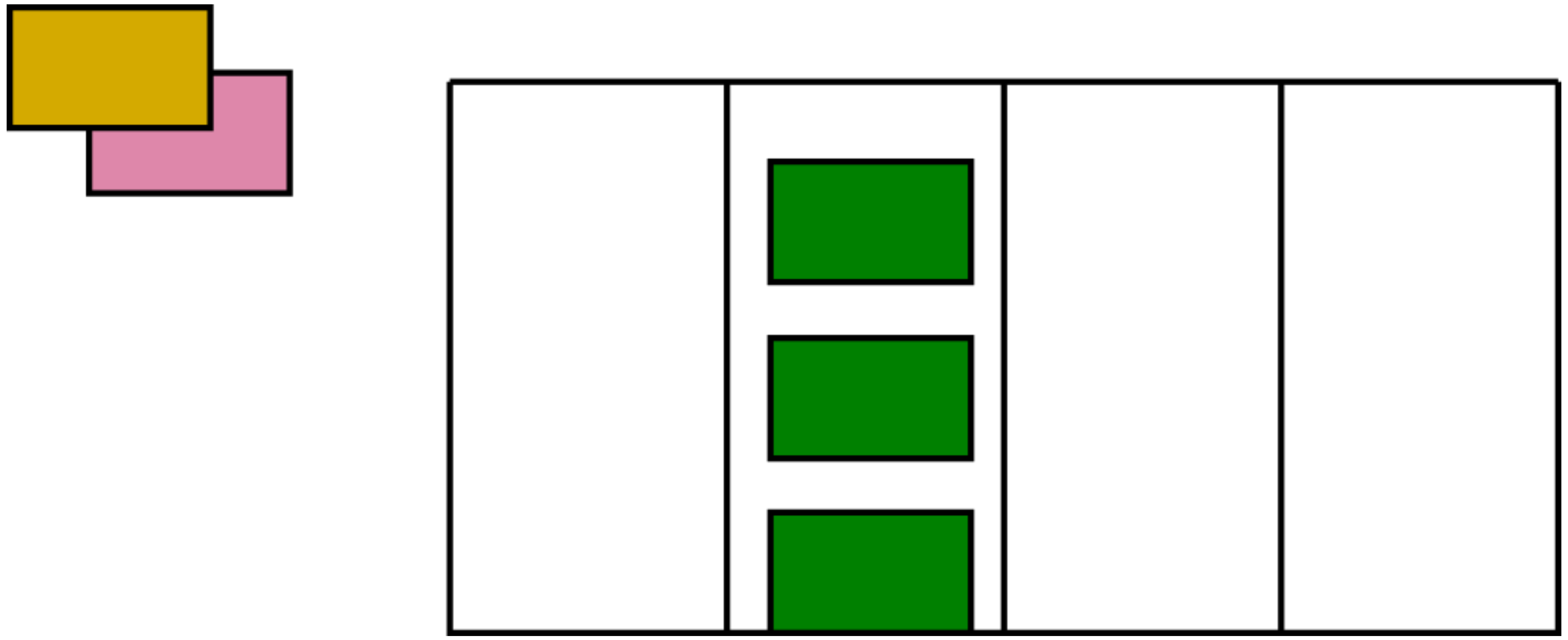
RRIParoo eviction in KSet helps miss ratio



Problem: Evict from set to make room for log objects while:

- Retaining more popular objects
- Maintaining small memory overhead

RRIParoo eviction in KSet helps miss ratio



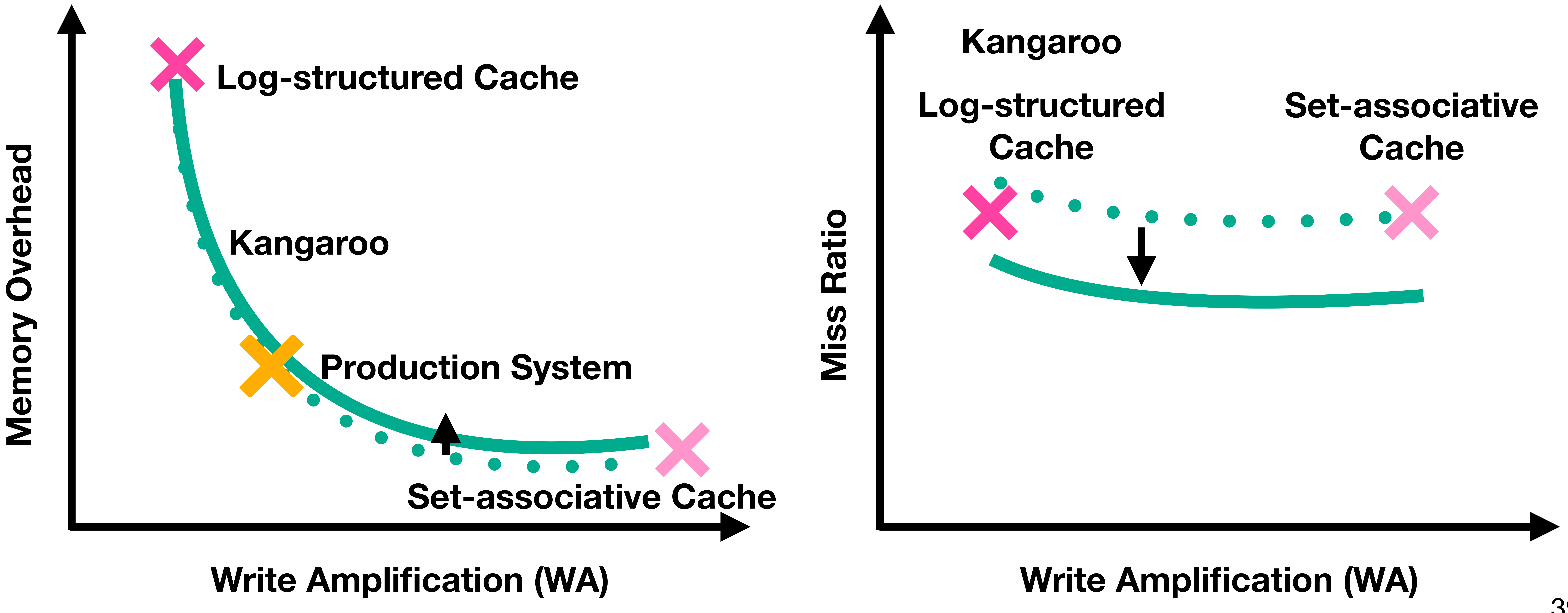
Problem: Evict from set to make room for log objects while:

- Retaining more popular objects
- Maintaining small memory overhead

Solution: RRIParoo, a modified version of RRIP RRIP (Jaleel ISCA'10)

- **1 bit DRAM/object** in KSet with RRIParoo

RRIParoo improves miss ratio

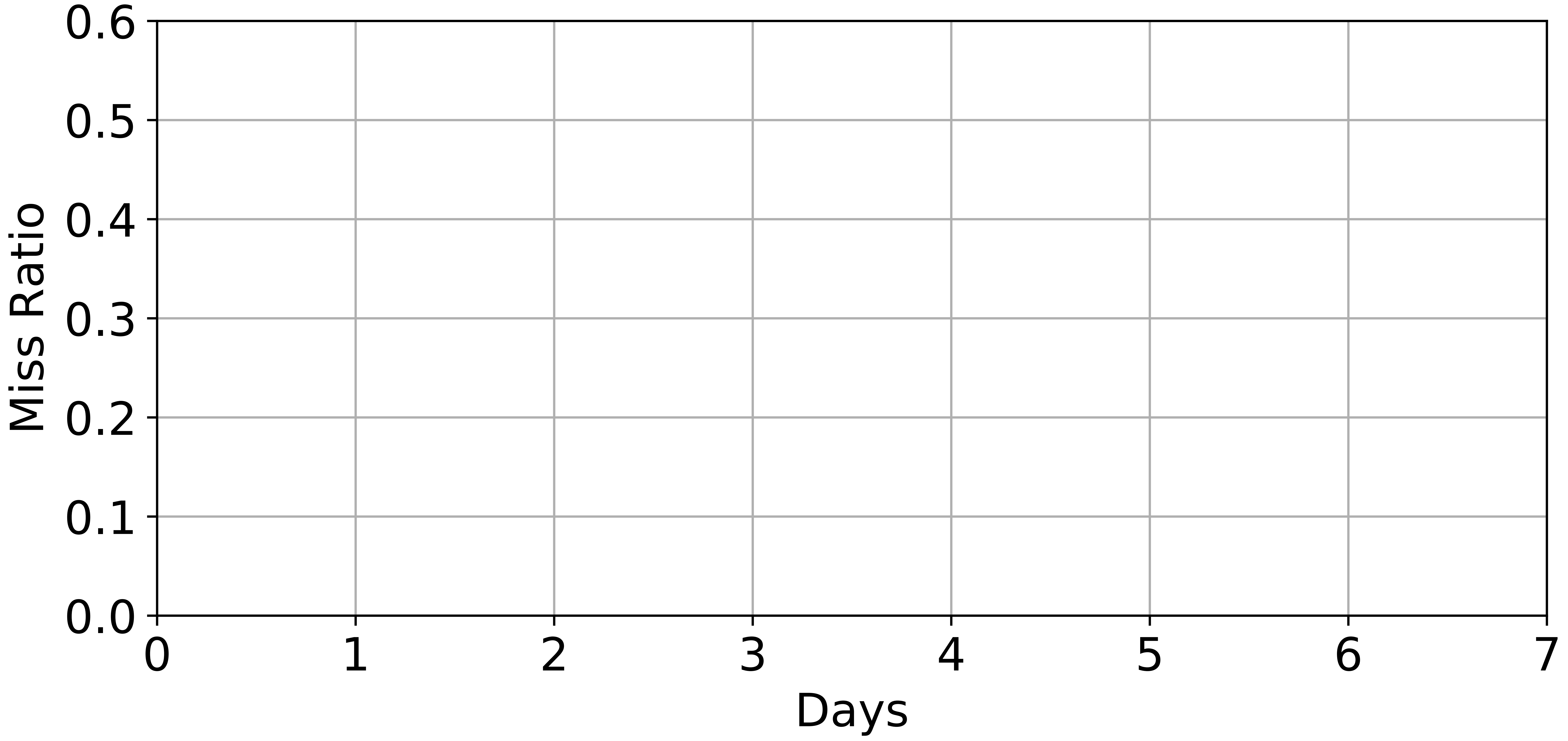


Outline

1) Introduction
2) Caching on flash
3) Minimizing DRAM overhead
4) Kangaroo design
5) Results

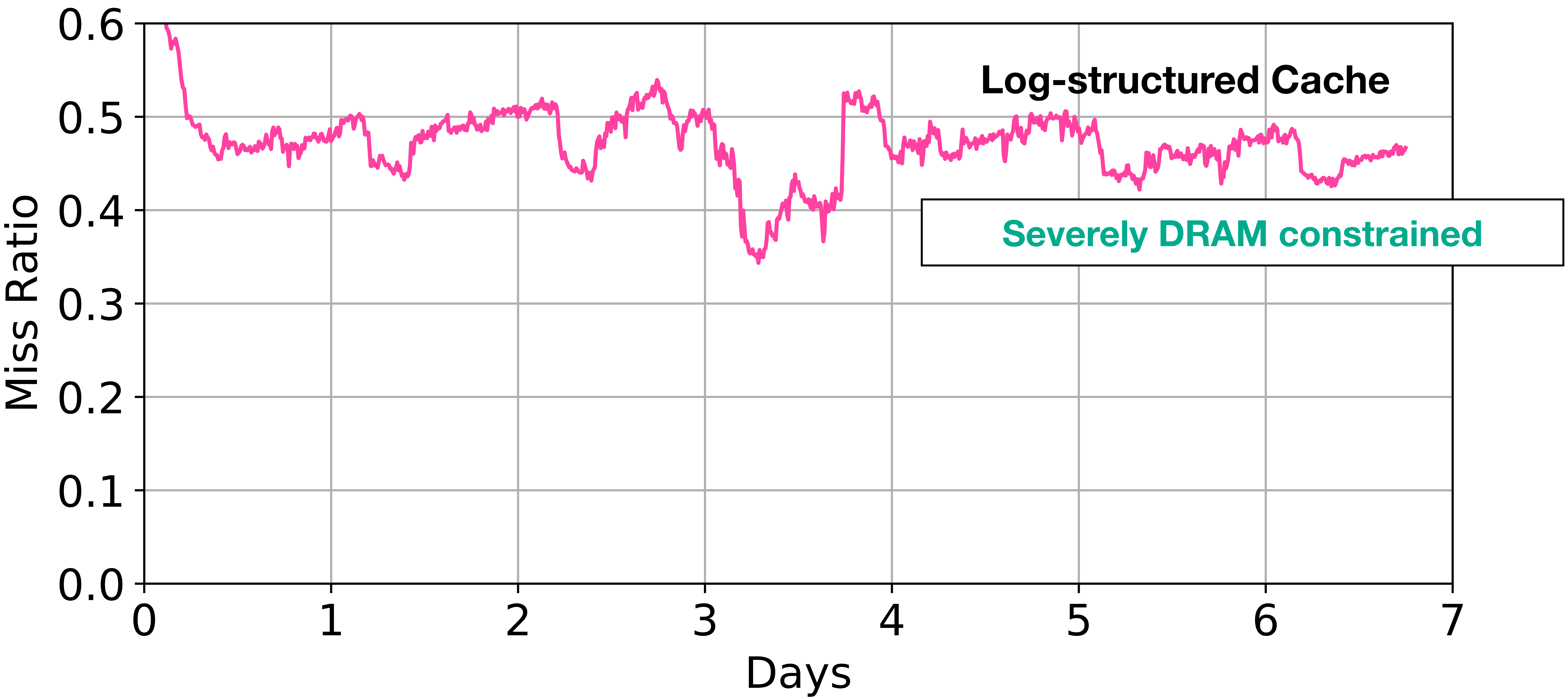
Kangaroo has best miss ratio

Run on 2 TB flash drive with a 7-day Facebook trace with 16 GB DRAM and 3 DWPD



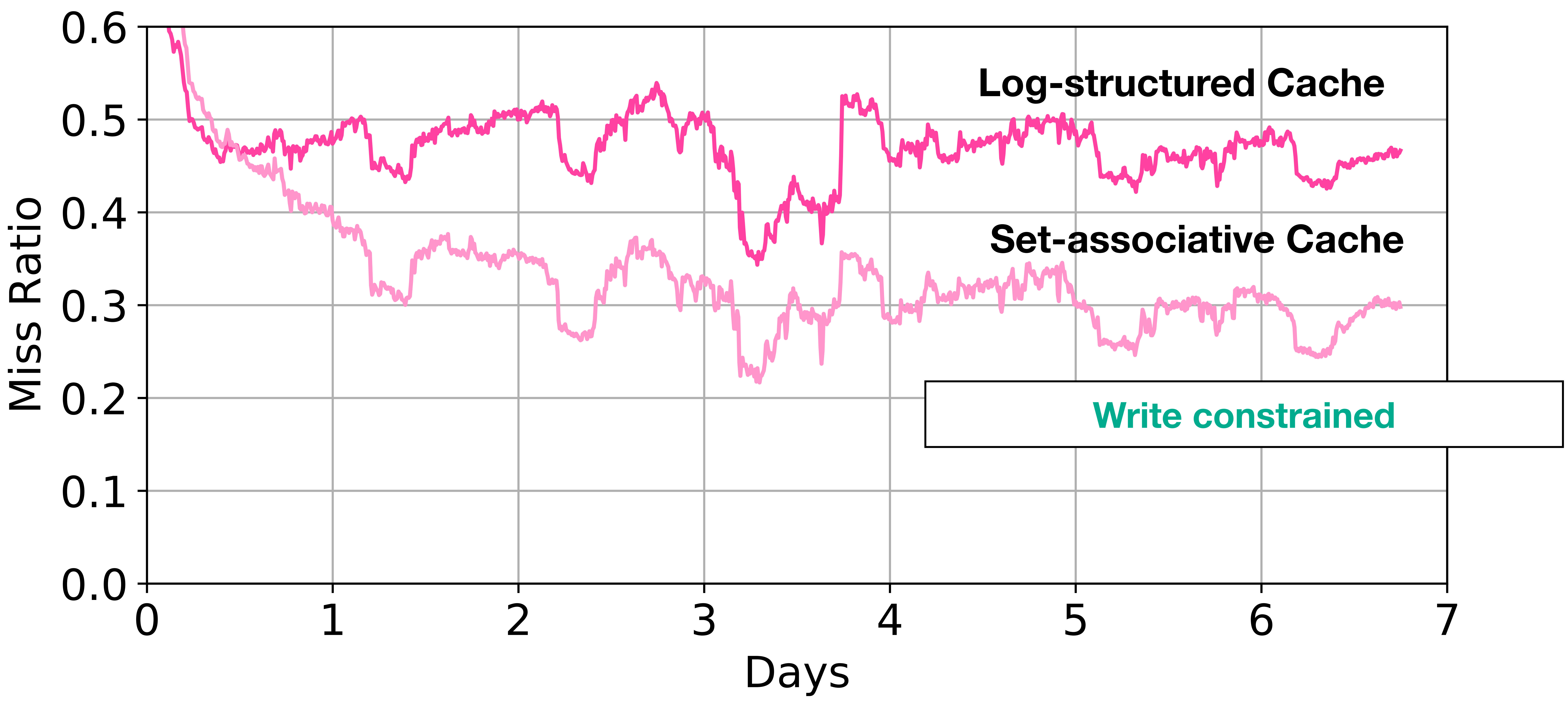
Kangaroo has best miss ratio

Run on 2 TB flash drive with a 7-day Facebook trace with 16 GB DRAM and 3 DWPD



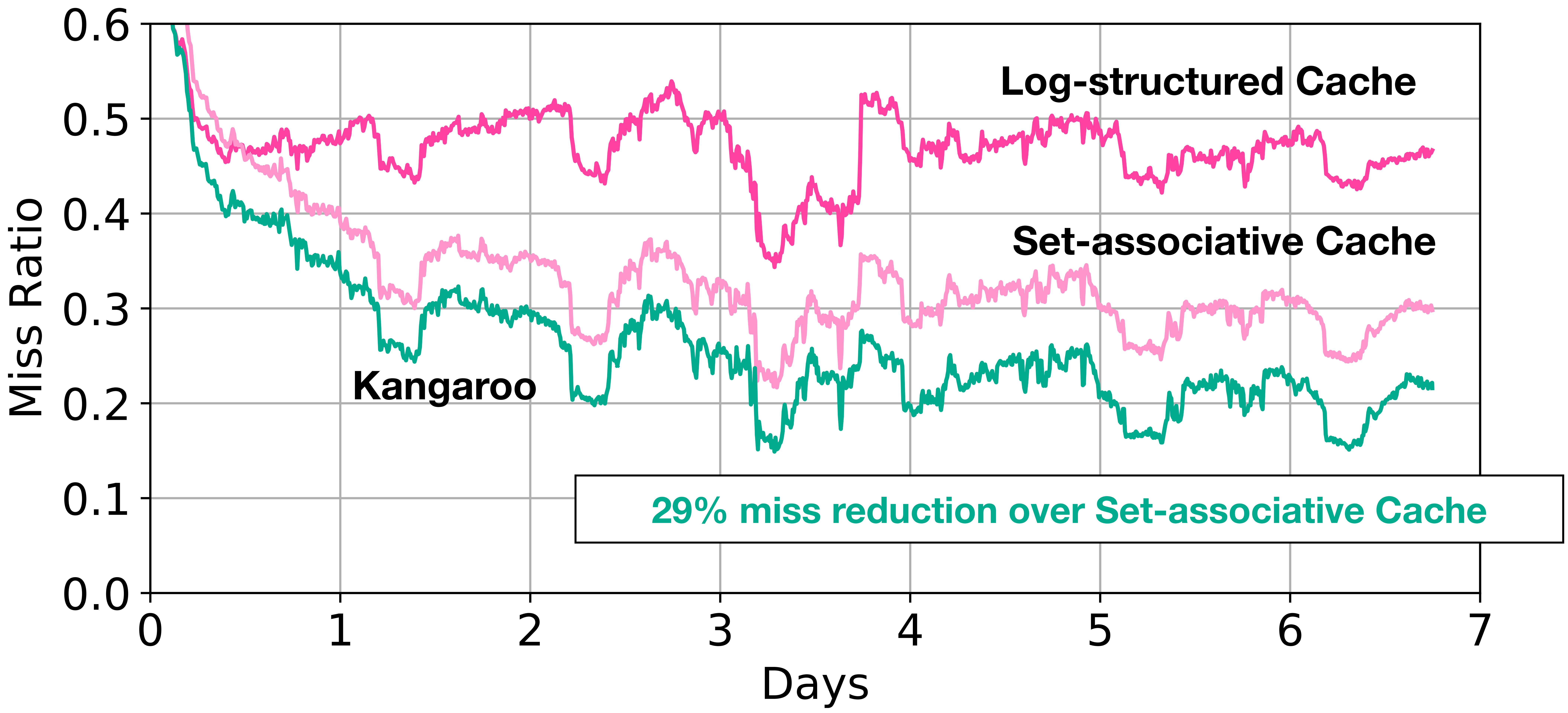
Kangaroo has best miss ratio

Run on 2 TB flash drive with a 7-day Facebook trace with 16 GB DRAM and 3 DWPD



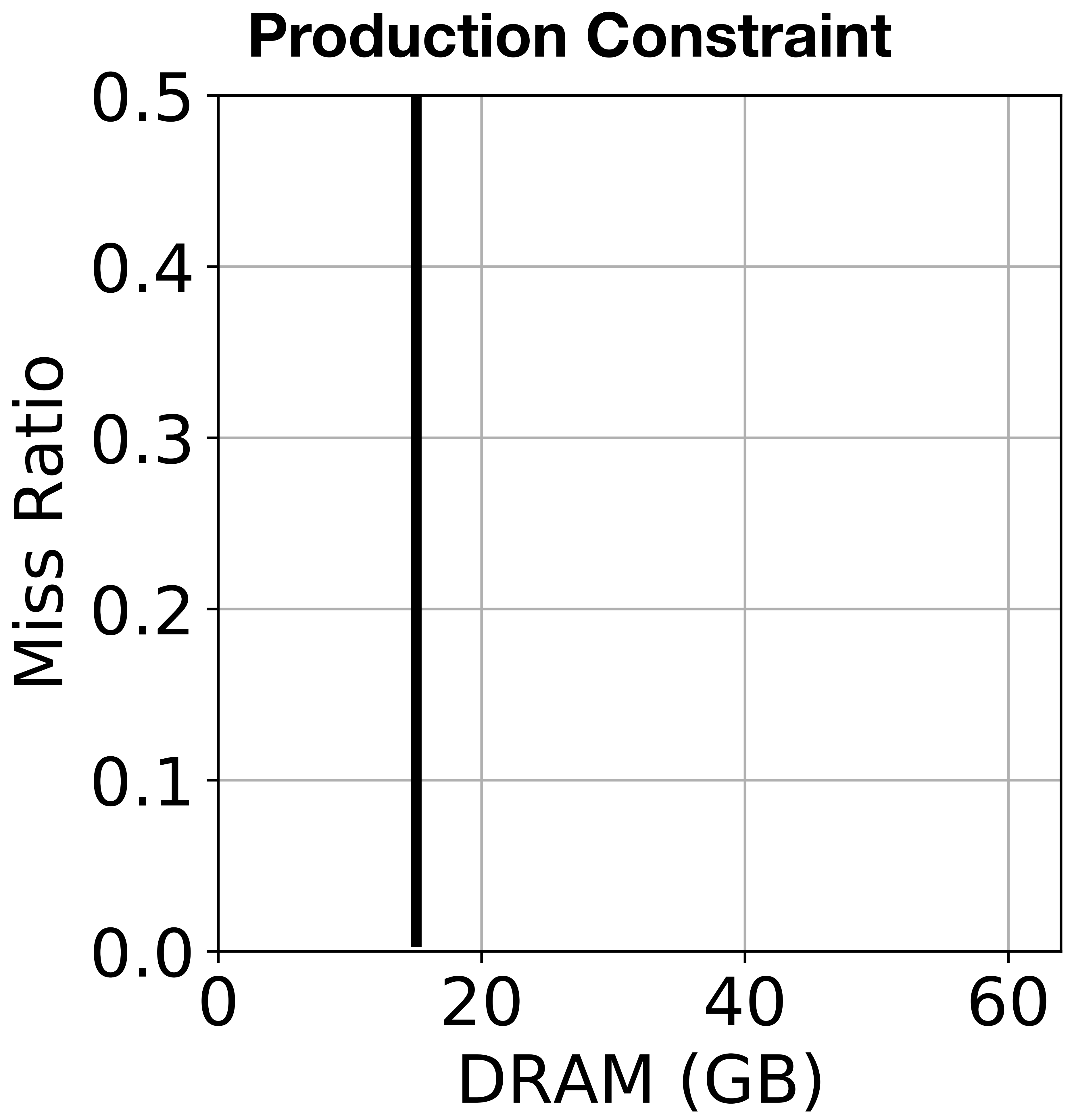
Kangaroo has best miss ratio

Run on 2 TB flash drive with a 7-day Facebook trace with 16 GB DRAM and 3 DWPD



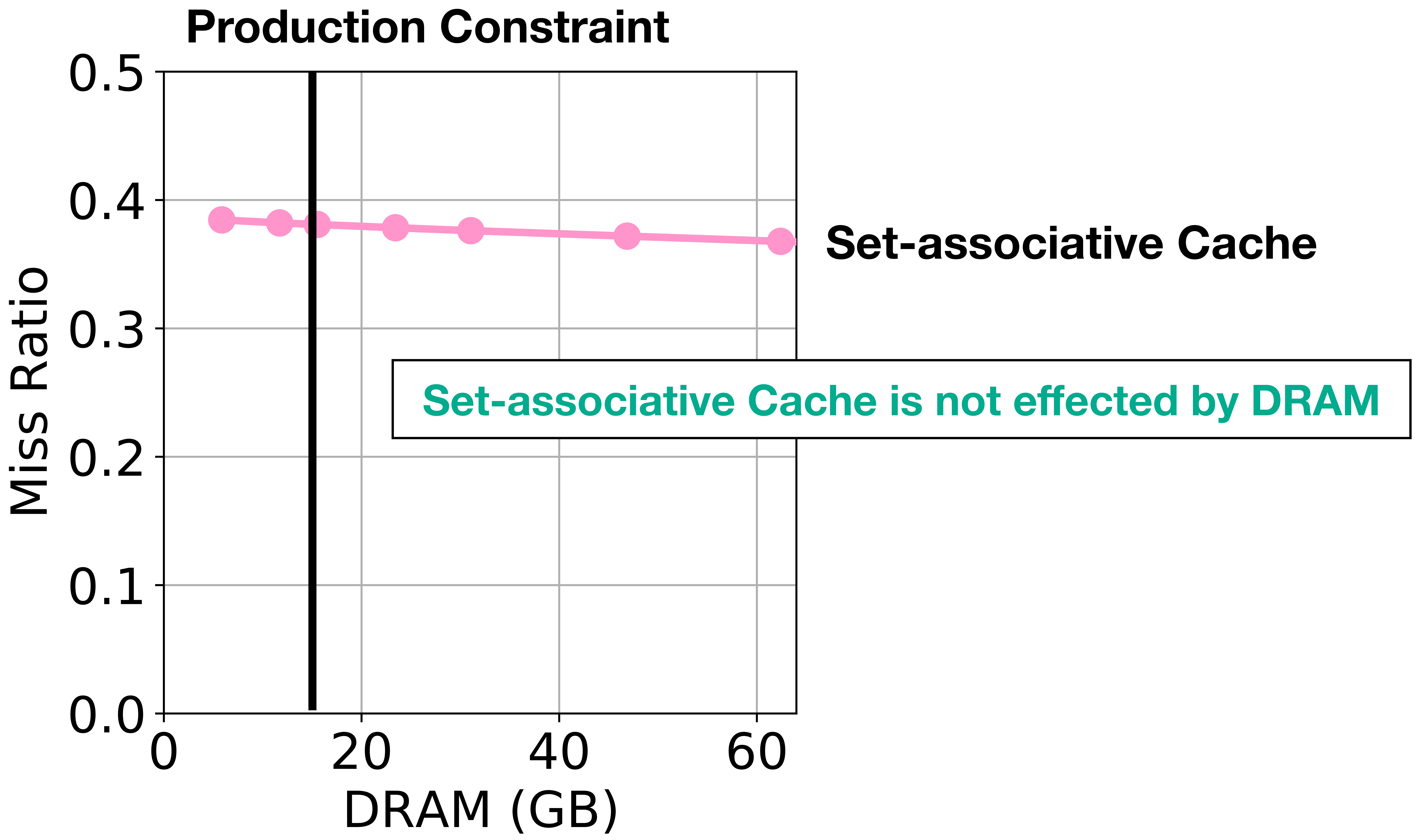
Varying DRAM budget

Simulating caches under different DRAM budgets on a 2 TB flash drive with 3 DWPD



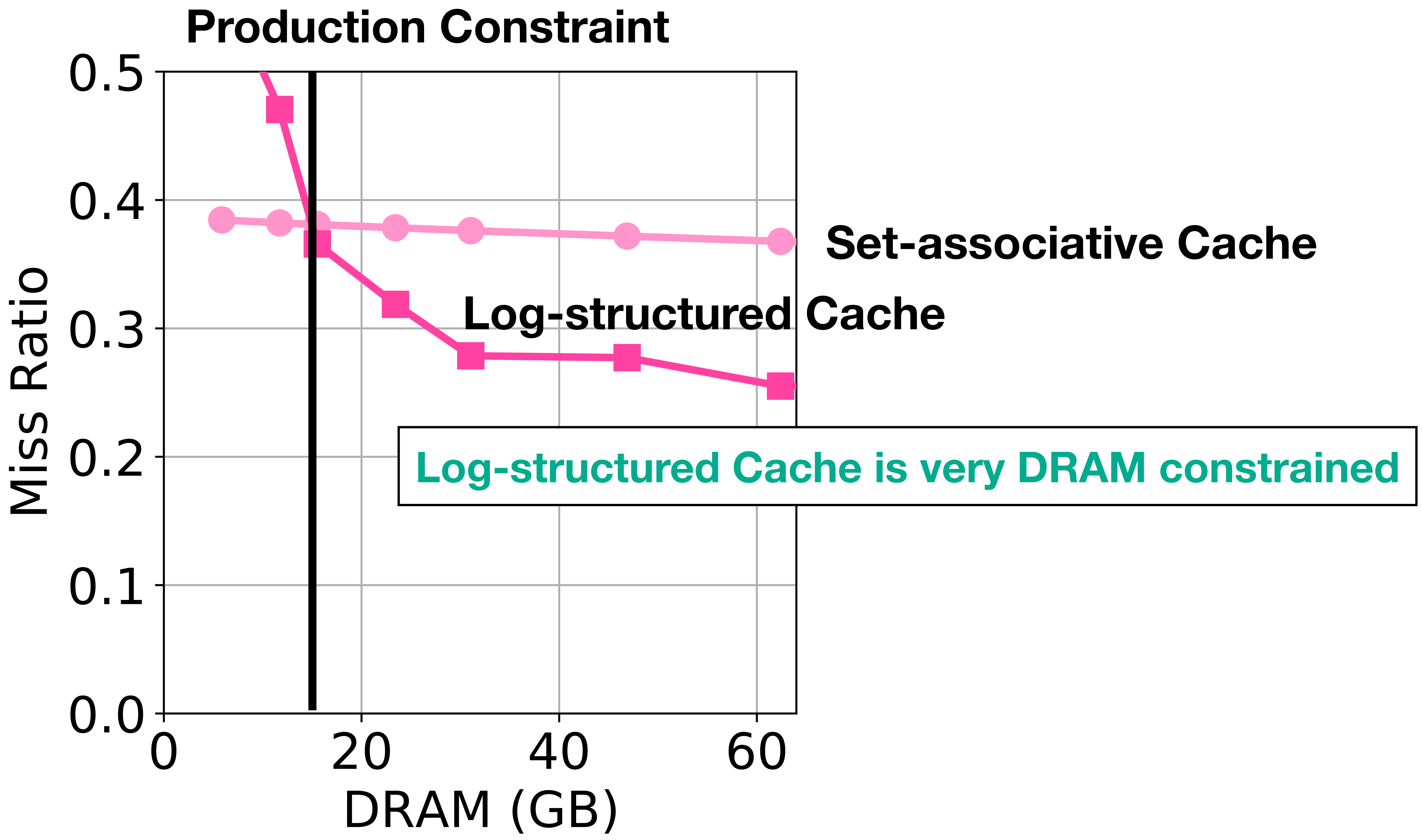
Varying DRAM budget

Simulating caches under different DRAM budgets on a 2 TB flash drive with 3 DWPD



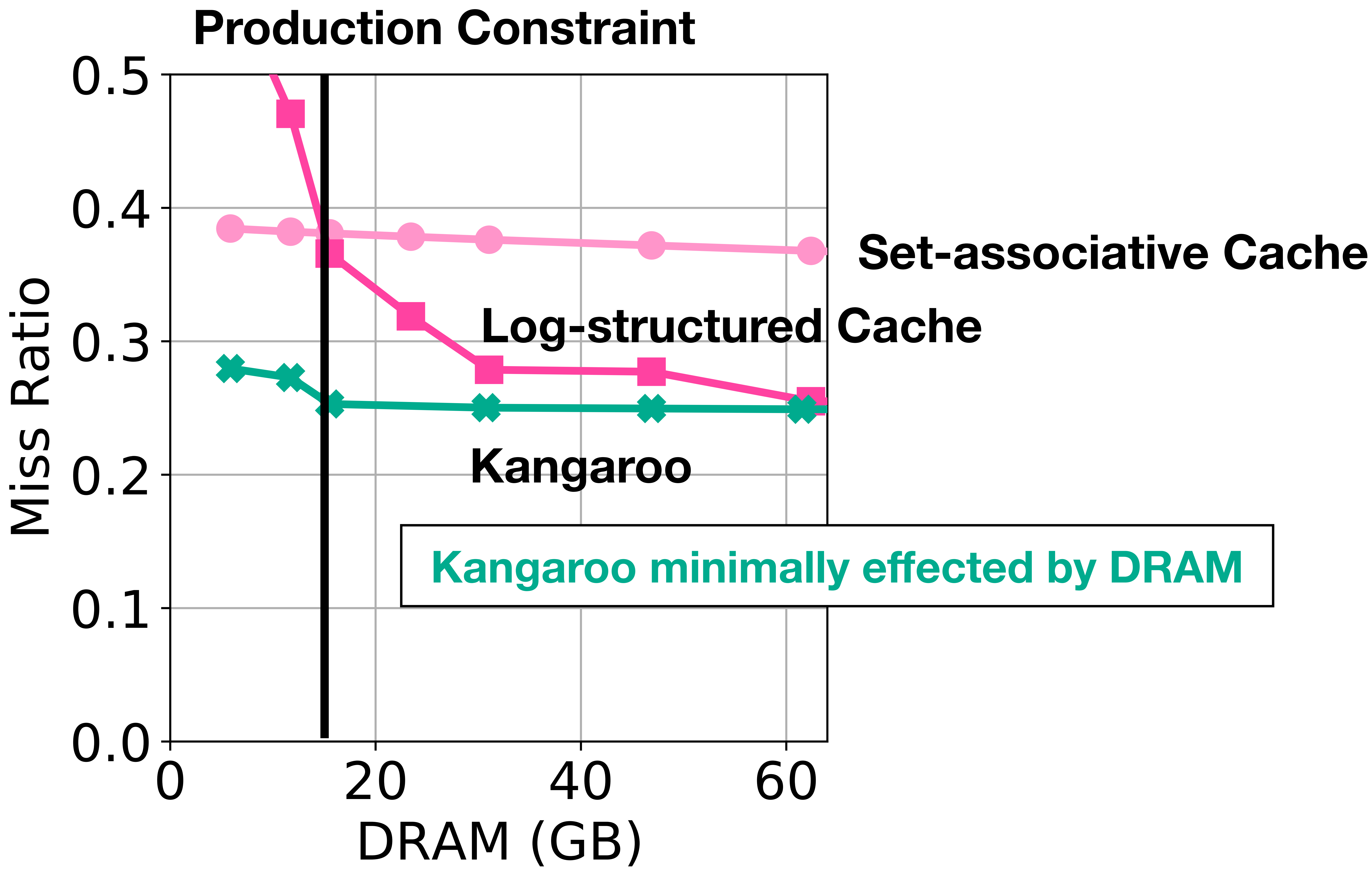
Varying DRAM budget

Simulating caches under different DRAM budgets on a 2 TB flash drive with 3 DWPD



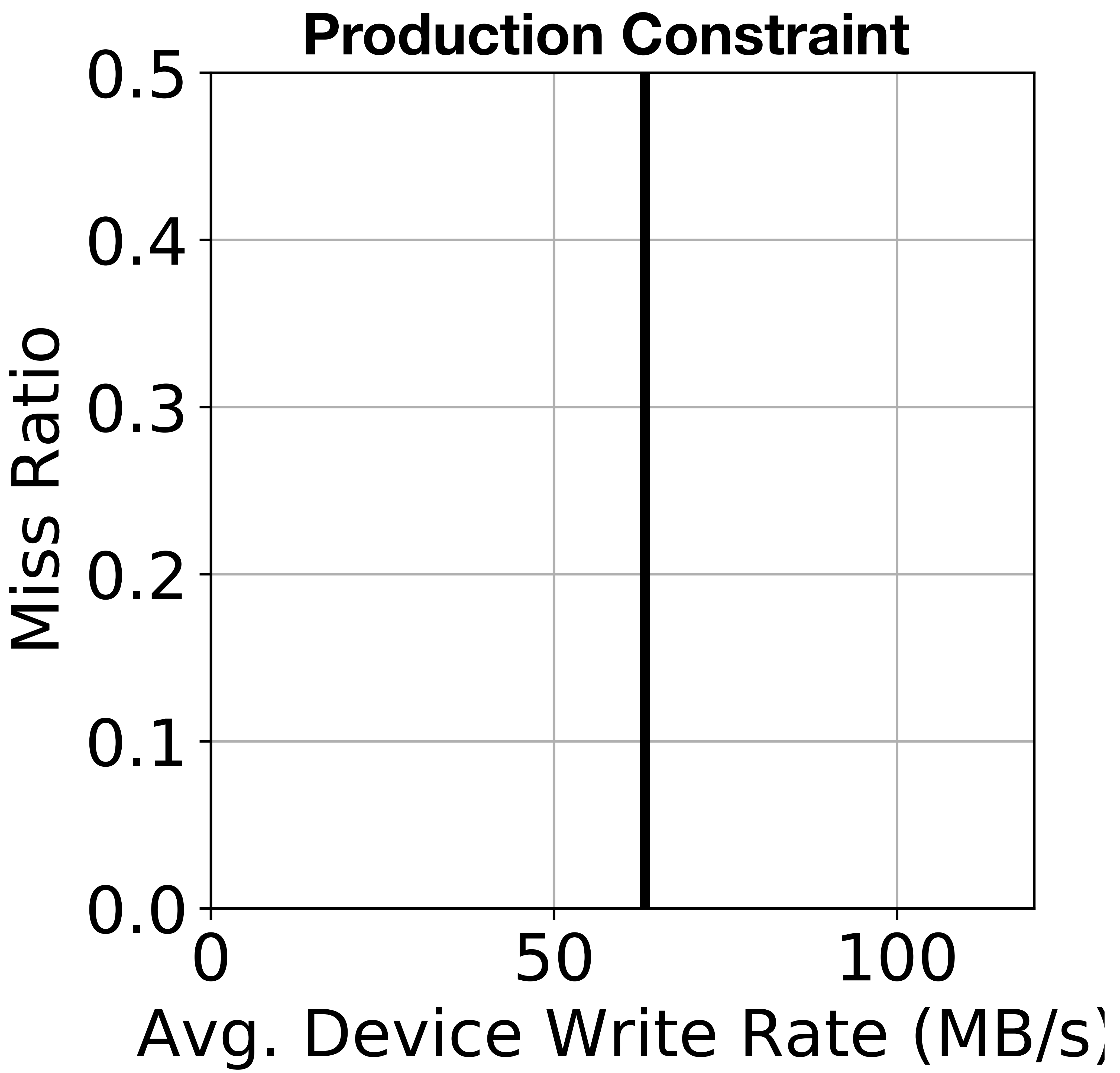
Varying DRAM budget

Simulating caches under different DRAM budgets on a 2 TB flash drive with 3 DWPD



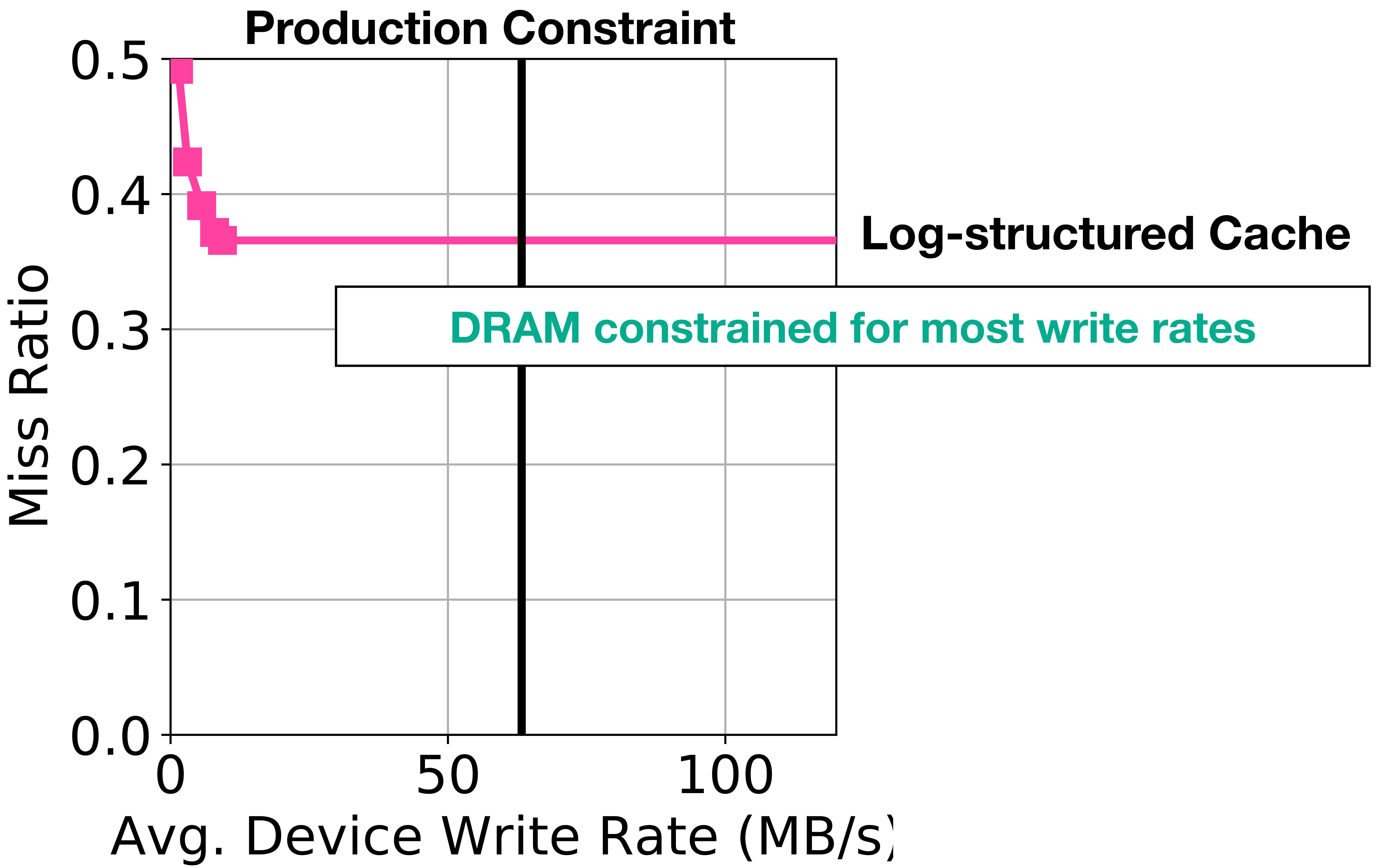
Varying write budget

Simulating caches under different write budgets on a 2 TB flash drive with 16 GB memory



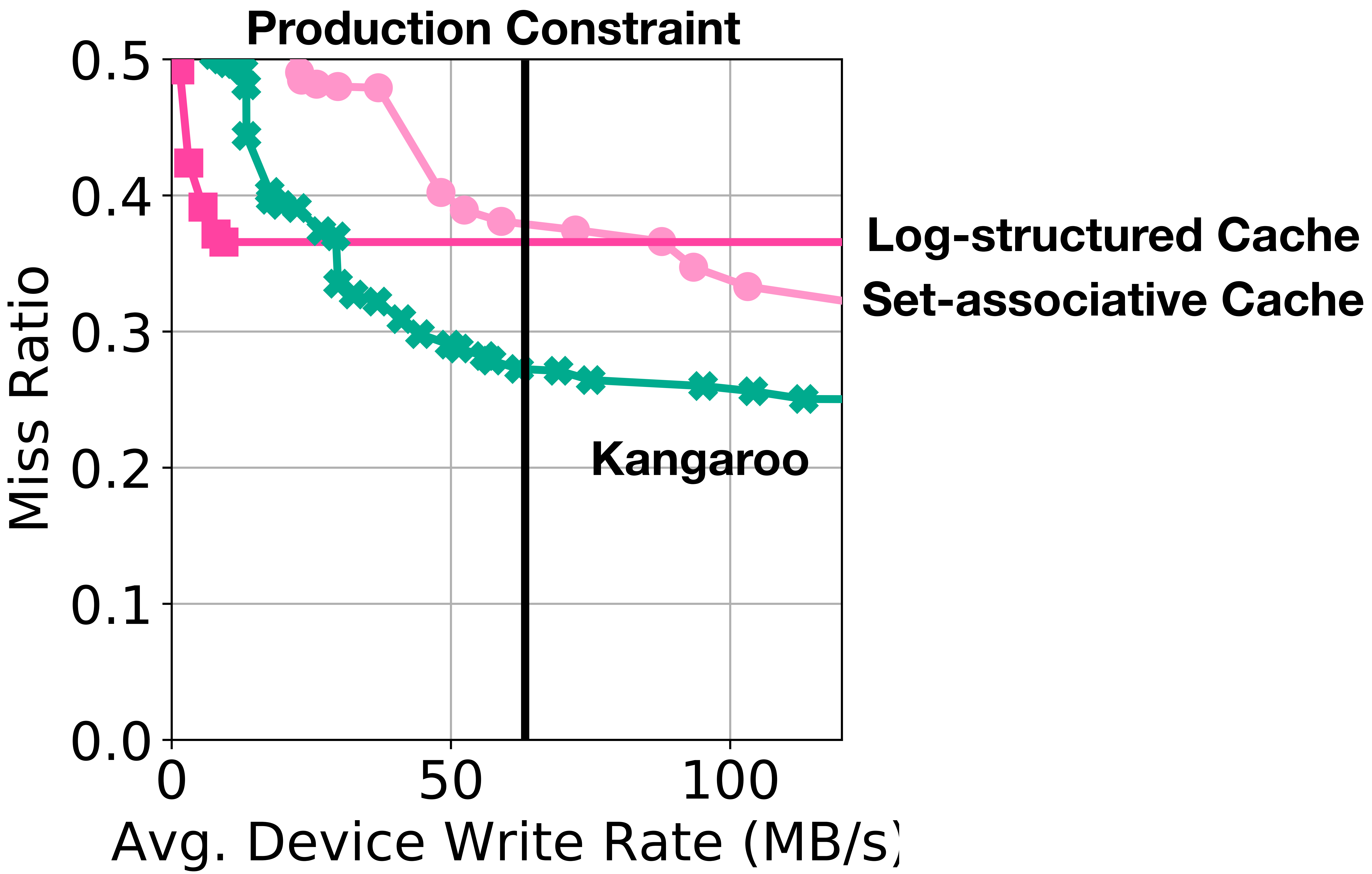
Varying write budget

Simulating caches under different write budgets on a 2 TB flash drive with 16 GB memory



Varying write budget

Simulating caches under different write budgets on a 2 TB flash drive with 16 GB memory



Kangaroo: Caching Billions of Tiny Objects on Flash

A flash cache for tiny objects that has:

1. Write rate within bounds for device lifetime by amortizing write costs
2. Low memory metadata overhead at **7.0 bits/object**
3. **29%** decrease in misses over than competitors

And responds well to changes in system parameters

See paper for more details including:

- KLog's partitioned index providing **>3.9x** DRAM reduction
- Kangaroo's **Pareto-optimality** on Twitter traces
- Kangaroo's test deployment in **production** at Facebook

Kangaroo: Caching Billions of Tiny Objects on Flash

A flash cache for tiny objects that has:

1. Write rate within bounds for device lifetime by amortizing write costs
2. Low memory metadata overhead at **7.0 bits/object**
3. **29%** decrease in misses over than competitors

And responds well to changes in system parameters

Acknowledgements

Thanks to the CacheLib team at Facebook (cachelib.org) and both Facebook and Twitter for sharing traces with us.