Baleen: ML Admission & Prefetching for Flash Caches

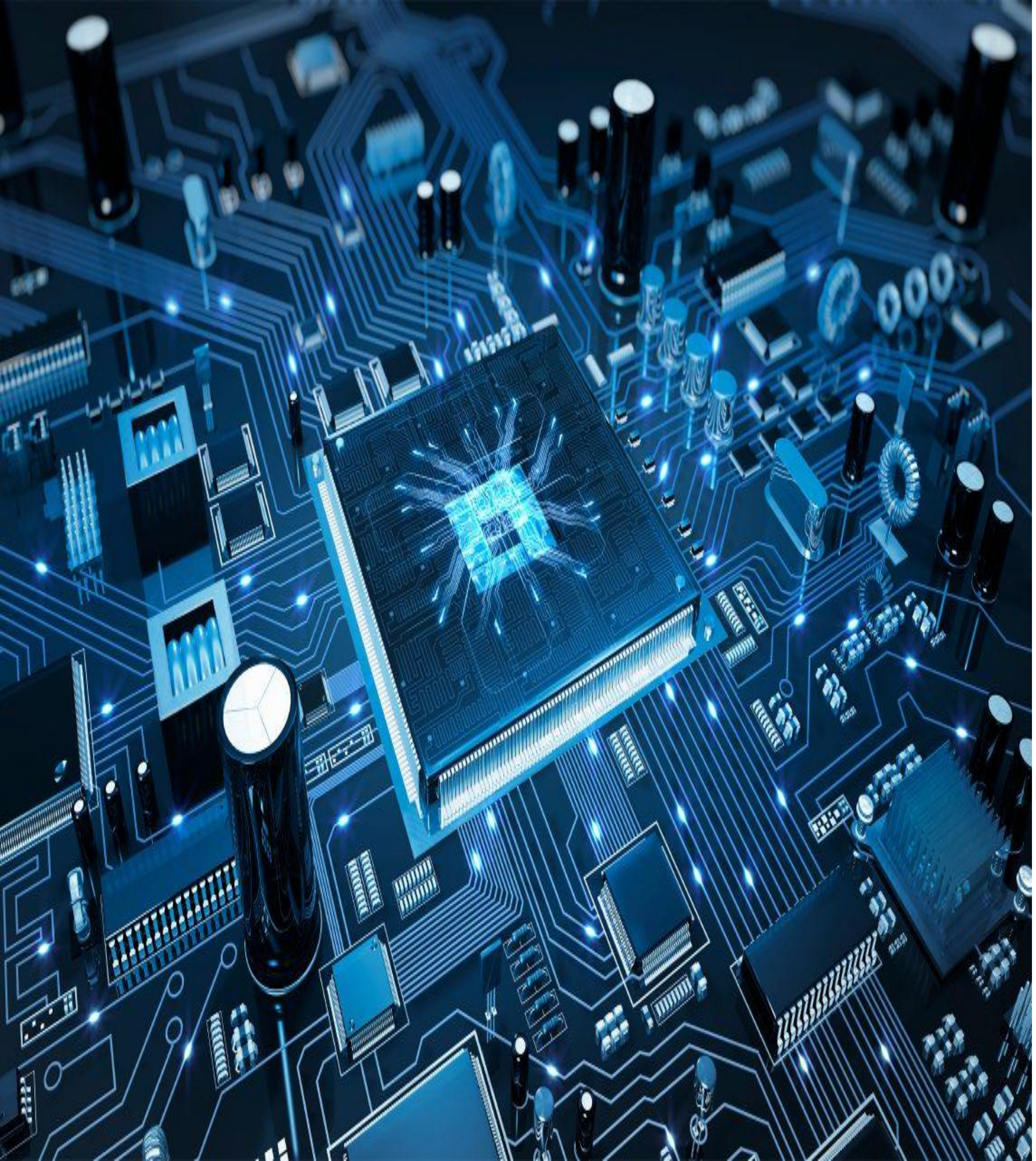
Daniel et al.

FAST'24

2025. 01. 15

Presented by Ramadhan Agung Rahmat

agung@dankook.ac.kr



Content

SSD vs HDD

SSD		HDD
0.1 ms	**Access Times** SSDs exhibit virtually no access time	5.5-8.0 ms
SSDs deliver at least 6000 io/s	**Random I/O Performance** SSDs are at least 15 times faster than HDDs	HDDs reach up to 400 io/s
SSDs have a failure rate of less than 0.5%	**Reliability** This makes SSDs 4-10 times more reliable	HDDs failure rate fluctuates between 2-5%
SSDs consume between 2 and 5 watts	**Energy Savings** This means that on a large server, approximately 100 watts are saved	HDDs consume between 6 and 15 watts
SSDs have an average I/O wait of 1%	**CPU Power** You will have an extra 6% of CPU power for other operations	HDDs average I/O wait is about 7%
The average service time for an I/O request while running a backup remain below 20 ms	**Input/Output Request Times** SSDs allow for much faster data access	The I/O request time with HDDs during backup rises up to 400-500 ms
SSD backups take about 6 hours	**Backup Rates** SSDs allow for 3-5 times faster backup for your data	HDD backups take up to 20-24 hours

On paper the average is **100** IOPS

Low throughput

Source: https://www.enterprisestorageforum.com/hardware/ssd-vs-hdd/

Bulk storage systems depend on flash caches

Bulk Storage

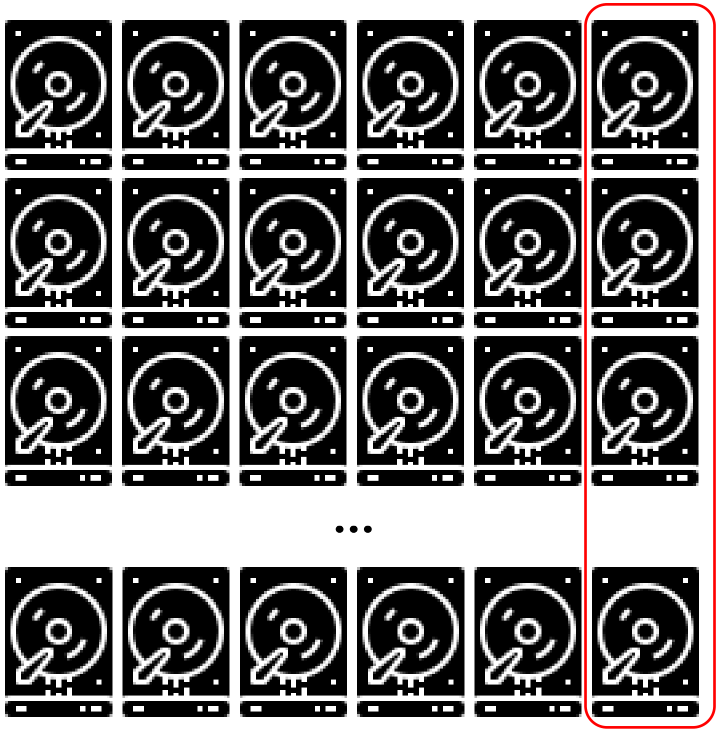
Example: Facebook’s Tectonic Filesystem

Exabytes on Hard Disk

Better flash caches save more HDDs

Bulk Storage

Example: Facebook's Tectonic Filesystem



Exabytes on Hard Disk

Flash caches absorb HDD load

Example: CacheLib

Better flash caches save more HDDs

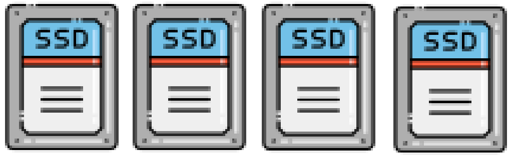
Bulk Storage

Example: Facebook’s Tectonic Filesystem

Exabytes on Hard Disk

Flash caches absorb HDD load

Example: CacheLib

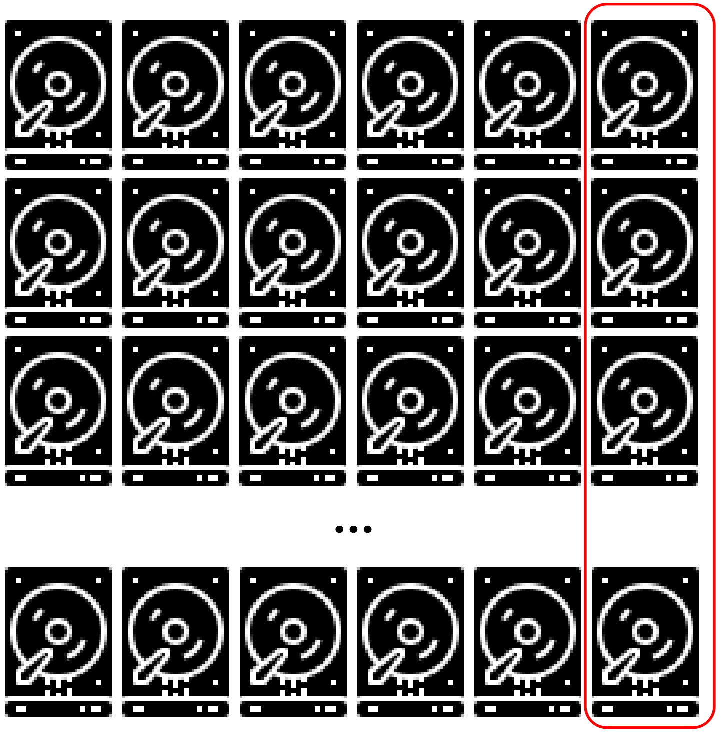


Better cache?

Flash caches are write-heavy

Bulk Storage

Example: Facebook's Tectonic Filesystem



…

Exabytes on Hard Disk

Better cache?

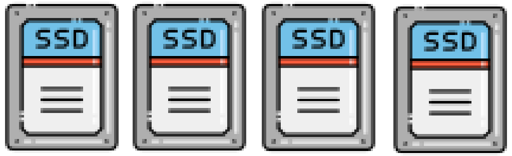
Flash caches absorb HDD load

Example: CacheLib

Problem: Limited write endurance

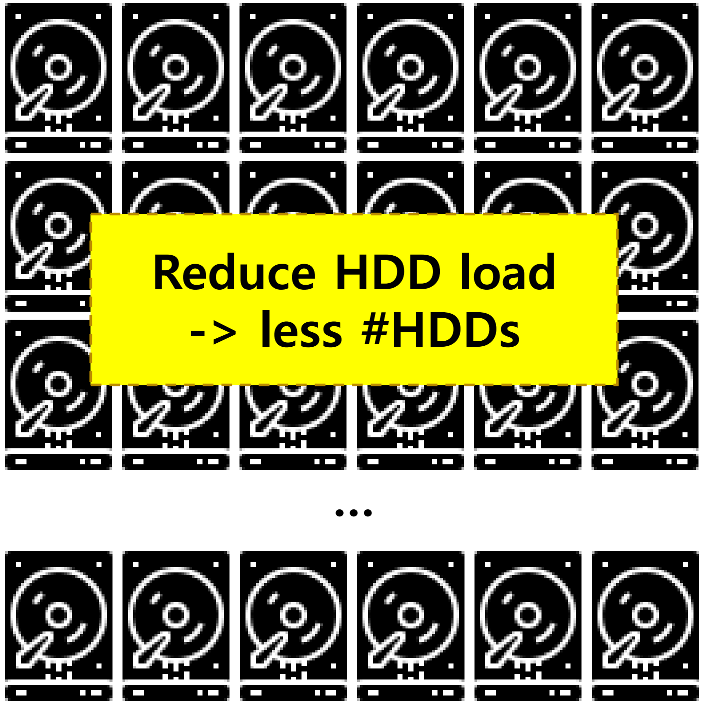
Cost dominated by #HDDs & #SSDs

Exabytes on Hard Disk



Cost dominated by #HDDs & #SSDs

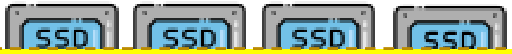
Baleen reduces cost by 17% on 7 traces



Reduce HDD load
-> less #HDDs

…

Exabytes on Hard Disk



Reduce flash writes
-> less #SSDs

How does Baleen reduce costs by 17%

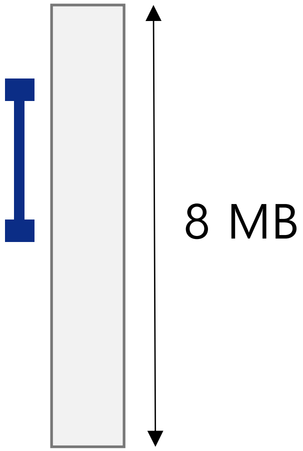
- 3 key ideas
  - Exploit a new cache residency model (**episodes**)
  - Train ML admission & ML prefetching policies
  - Optimize an end-to-end metric (disk-head time)
- Why ML over heuristic?
  - More savings, more adaptive

Bulk storage clients access bytes ranges withih blocks

Access

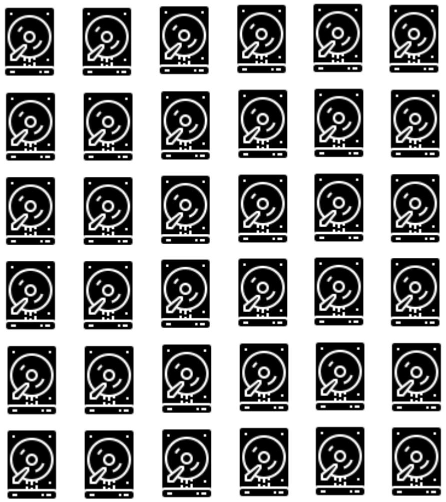
1. Block ID
2. Byte Range

Block



Host

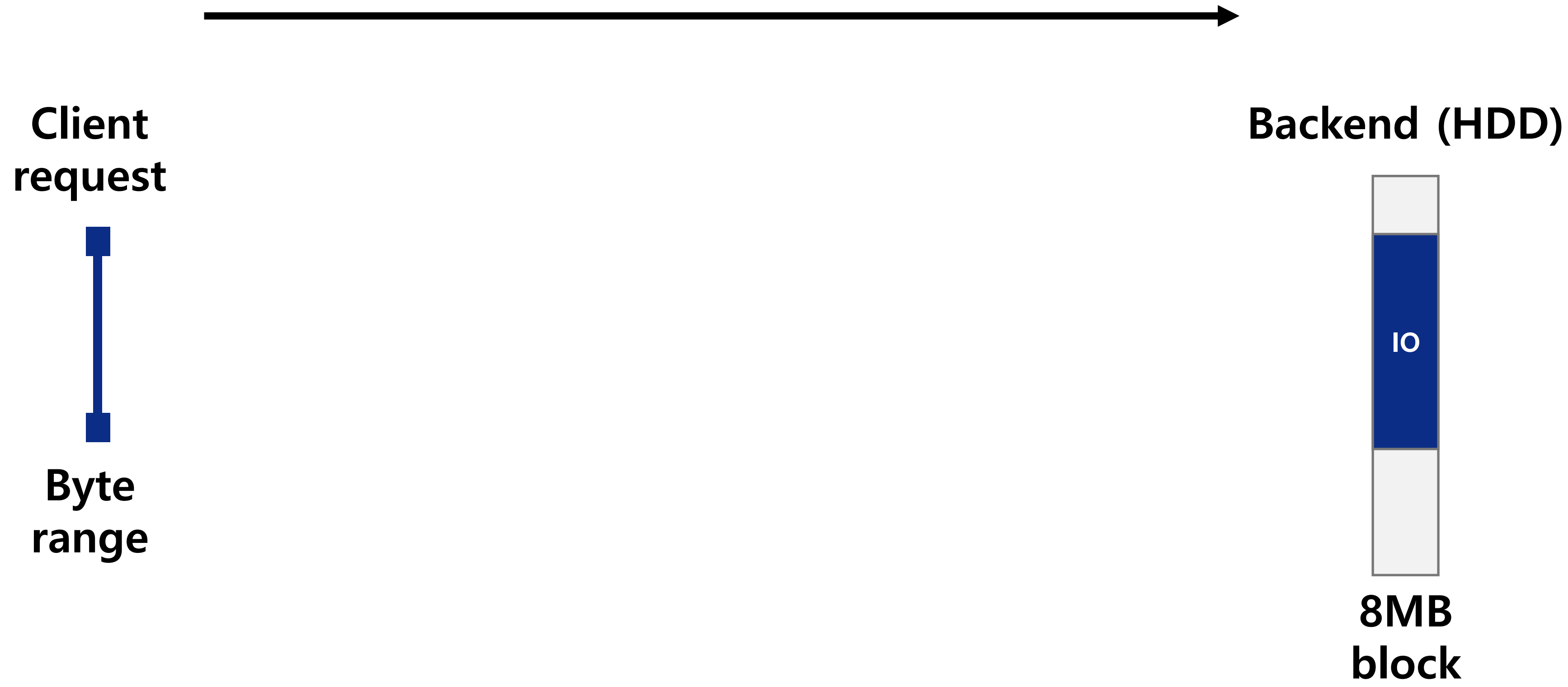
36 hard disks



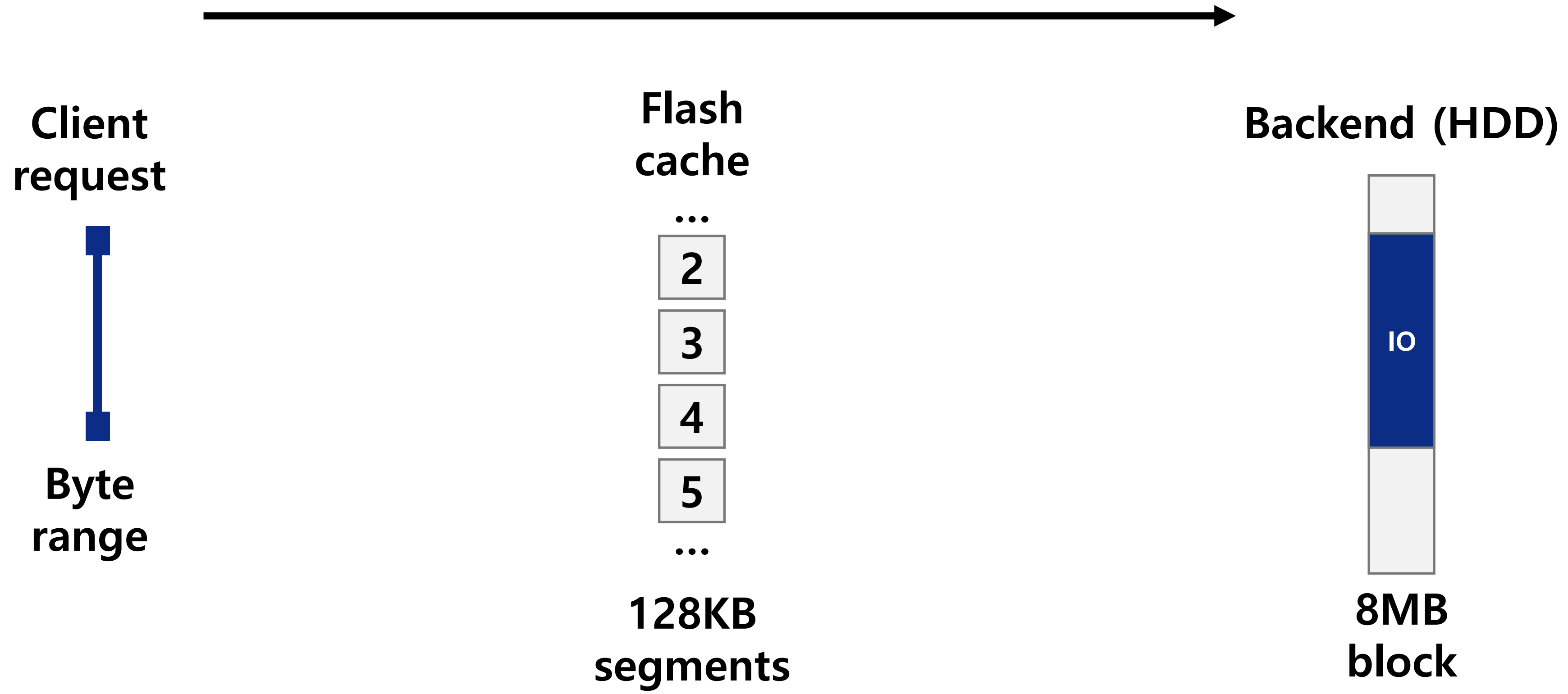
Cluster

Data center
1000s of hosts

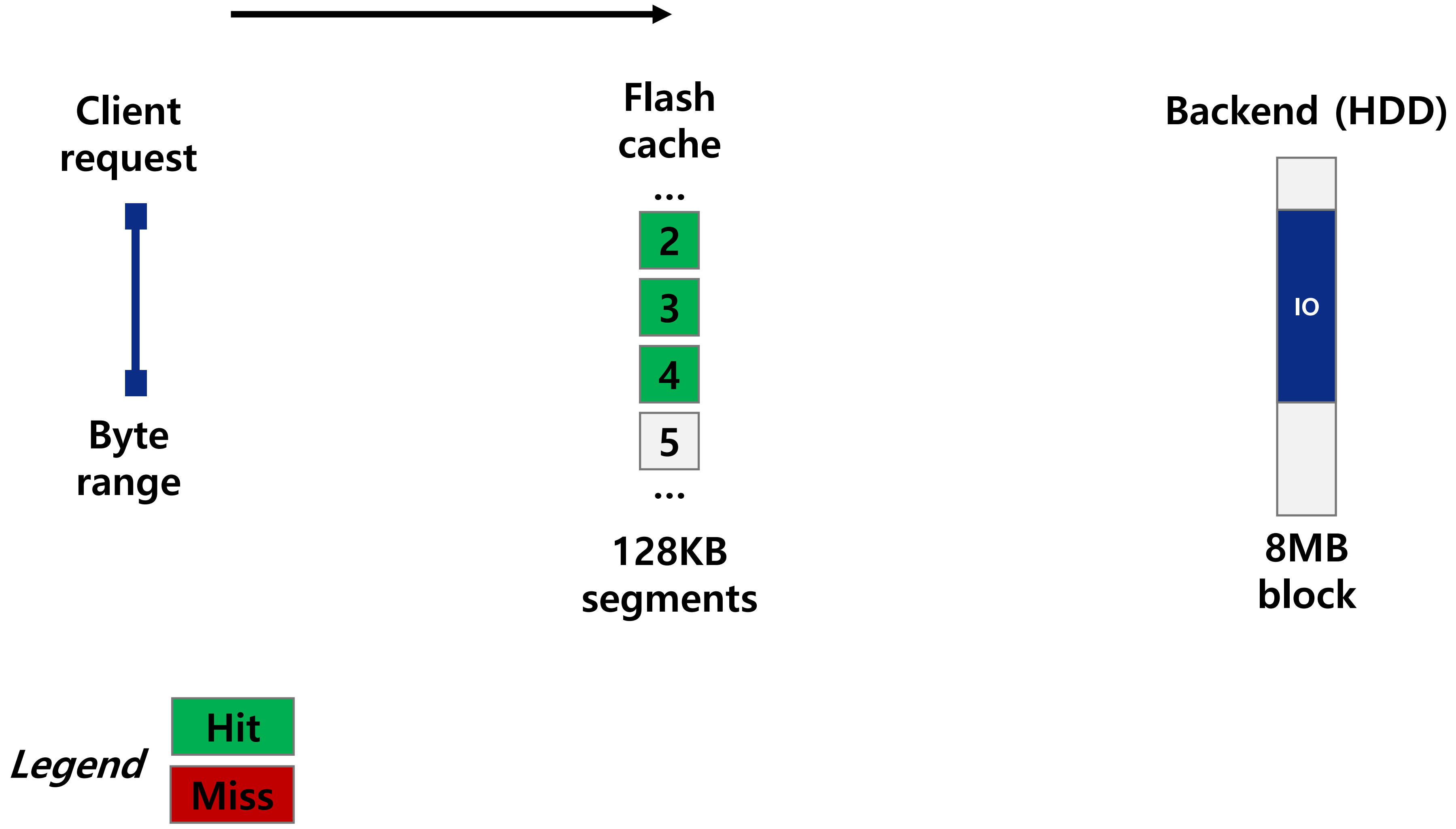
Fetching bytes from backend causes disk IO



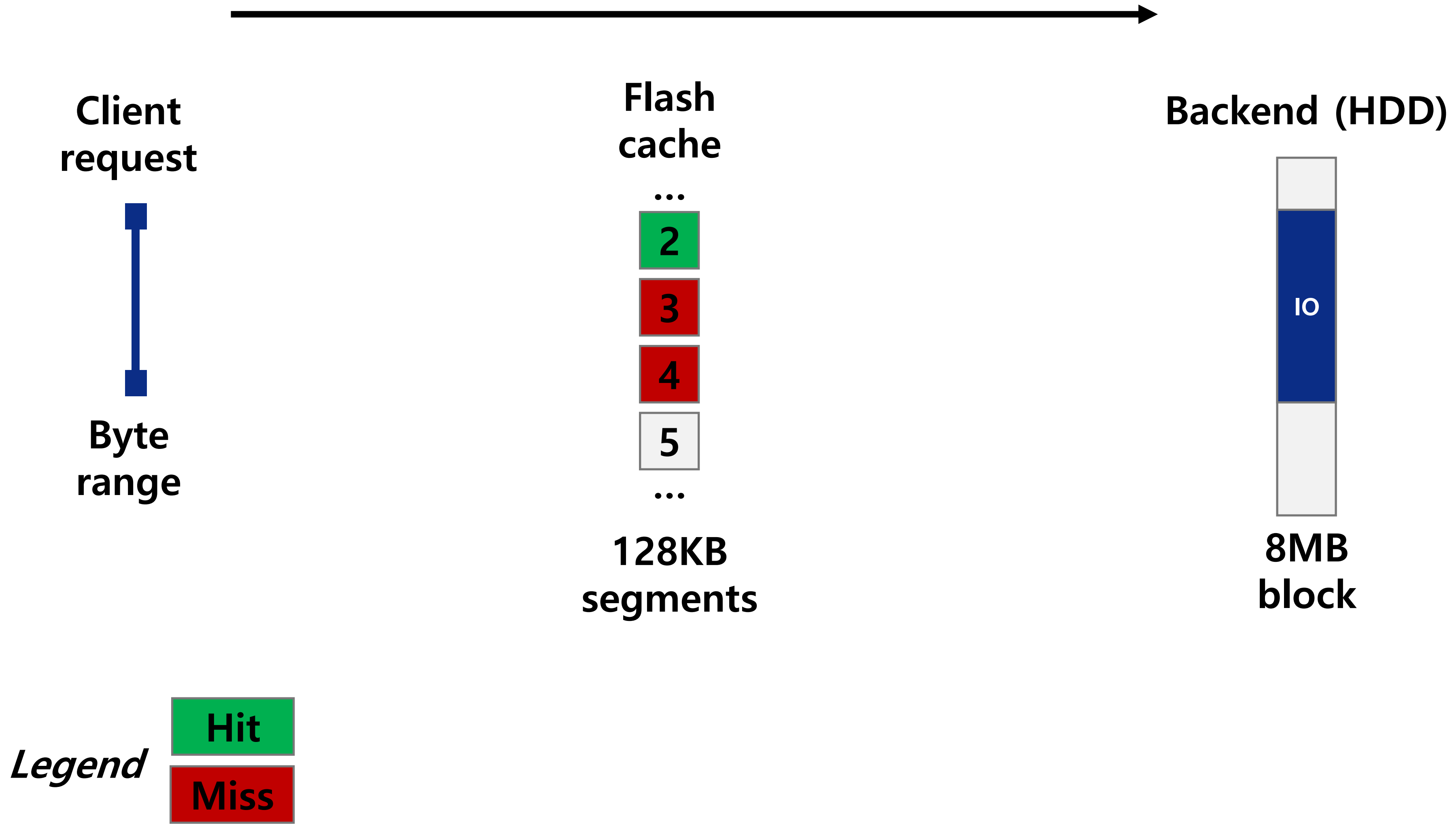
Cache stores segments (subset of block)



Cache hits save disk IO



Cache miss causes disk IO



Decompose flash caching into 3 decisions

Goal: Reduce HDD load without excessive flash writes

Flash cache

…
2
3
4
5
…

Policy Decisions:

(a) Admit misses? 3 4

(b) Prefetch? 5

(c) When to evict?

Decompose flash caching into 3 decisions

Goal: Reduce HDD load without excessive flash writes

Flash cache

...

2

3

4

5

...

Policy Decisions:

(a) Admit misses?	3 4	Baleen
(b) Prefetch?	5	Baleen
(c) When to evict?		LRU

Metric: Disk-head time (DT)

11.5ms Seek, 143MB/s (Dotted Line): This line represents a modeled approximation, combining a constant seek time of 11.5ms and a read bandwidth of 143MB/s.

- **Q: Why DT instead of miss rates?**
  - **A:** Miss rate alone does not indicate how expensive each miss is
  - **A:** DT does directly measure the latency each IO from disk.
- **DT = Seek time + Read time**

$$DT_i = Fetches_{IOs} \cdot t_{seek} + Fetches_{Bytes} \cdot t_{read}$$

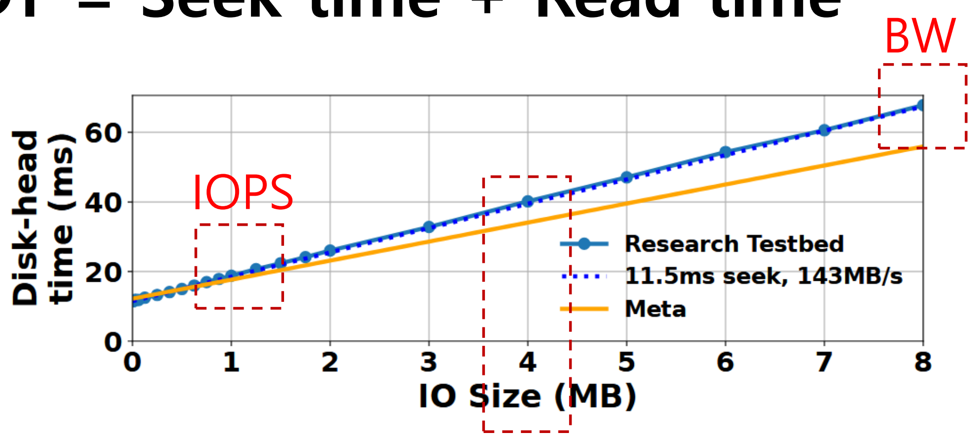


Figure 2: Disk-head Time (DT) for one IO. When a HDD performs an IO, the disk head **seeks** before it **reads** data. For tiny IOs, throughput is limited by *IOPS*; for large IOs, by *bandwidth*. DT encompasses both metrics and generalizes to variable-size IOs.

Example:
FetchesIO = **1** IO
FetchesBytes = **4** MB
Seek time = 11.5 ms = 0.0115 seconds
Read bandwidth = 143 MB/s = 0.006993 s/MB

DT= 1 x 0.0115 + 4 x 0.006993
DT= 0.0115 + 0.027972
DT= 0.039472 seconds = 40ms

DT Validated in production

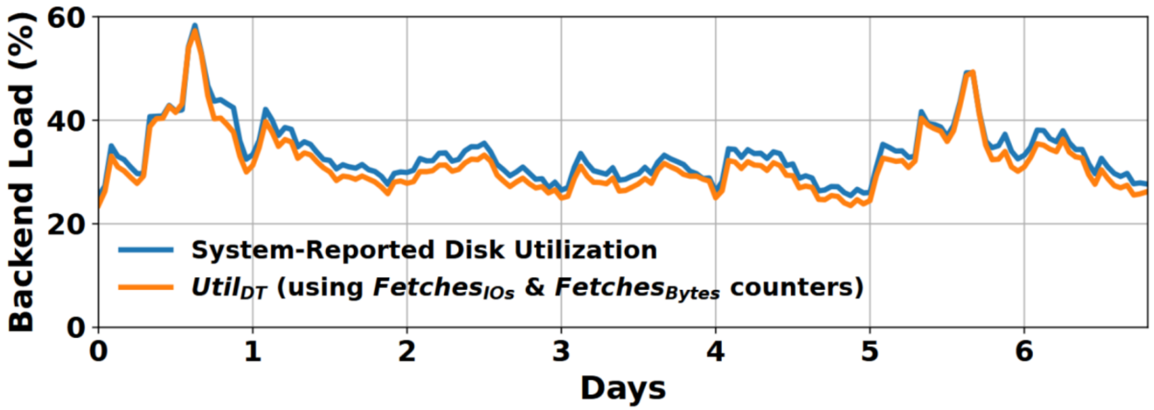


Figure 3: DT validated in production. Our DT formula (plugging counters into Eq 1) matches measured disk utilization (blue) closely. The peak of 58% occurs on Day 0.

Total DT for all misses

$$Util_{DT} = \frac{\sum_i DT_i}{DT_{Provisioned}}$$

1

Design
Episodes model

ML for caching not straightforward

Typical supervised learning

- e.g., “Is this picture a cat?”

ML for Caching

- Data: trace of accesses
- Multiple related decisions: Admit now? Later? Never?
- **Tend to overfit on easy decisions**
- **Underfit on borderline cases that separate different policies.**

ML for caching not straightforward

Typical supervised learning

- e.g., "Is this picture a cat?"

ML for Caching

- Data: trace of accesses
- Multiple r
  - Depend c

Training on accesses non-trivial

- **Tend to overfit on easy decisions**
- **Underfit on examples at margin that distinguish policies**

Architecture

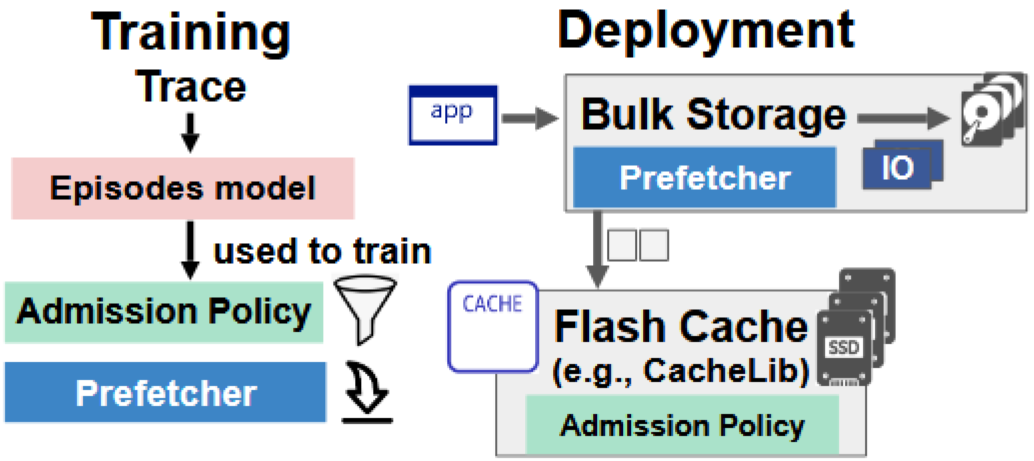


Figure 6: Architecture. An admission policy in CacheLib decides whether to admit items into flash. Prefetching (preloading of data beyond current request) takes place in Tectonic.

(Offline model)

What is an episode?

Episode:

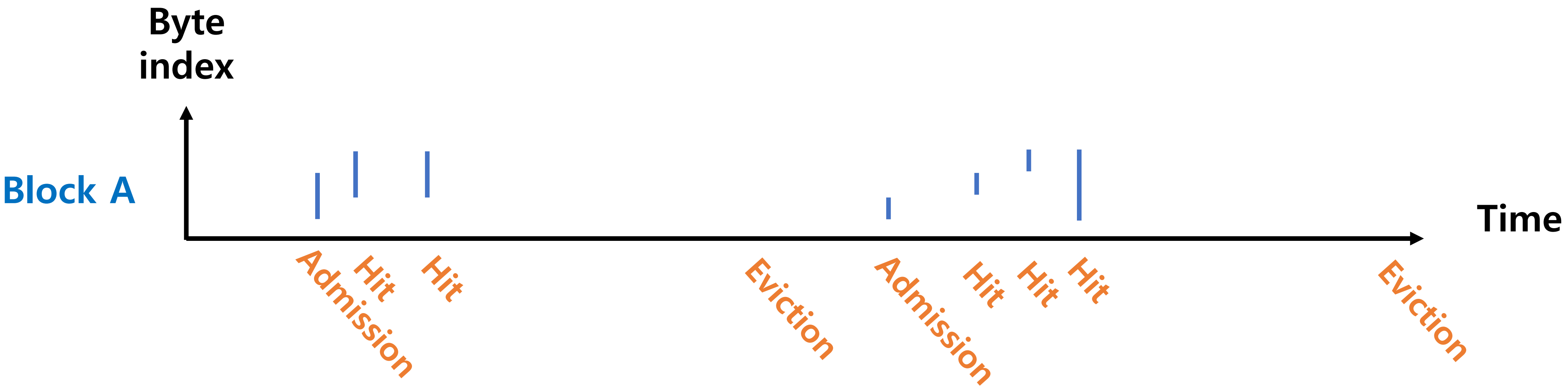
Sequence of accesses that would be hits
if corresponding item was admitted

Why use episodes to train ML?

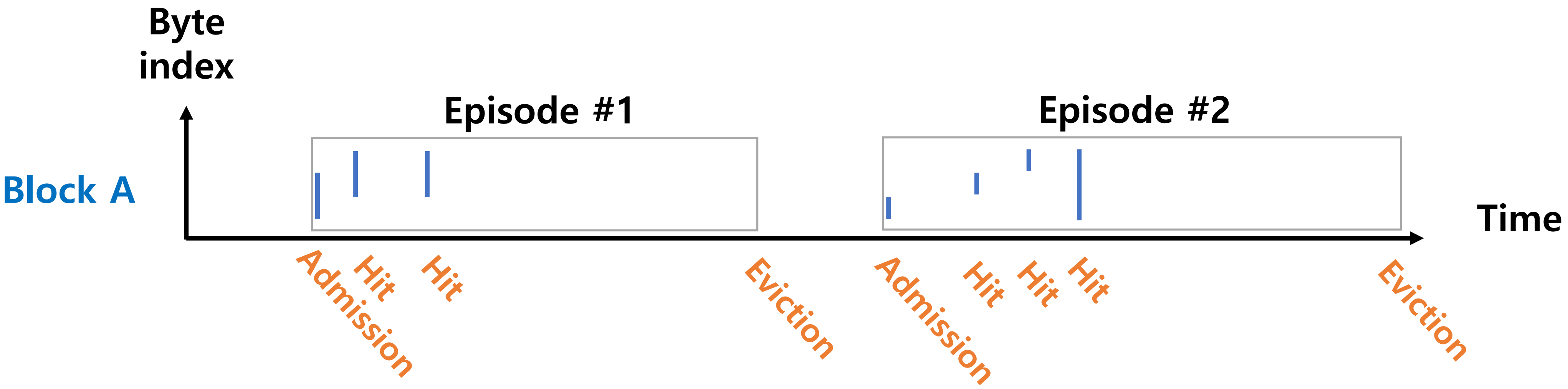
- Right granularity
  - Focus on first access instead of all **accesses**
  - Policies see misses, not accesses
- Right examples
  - Avoid overfitting on popular blocks with many **accesses** but only 1 miss
- Right labels
  - Costs & benefits defined on admission to eviction

Episodes: from admission to eviction

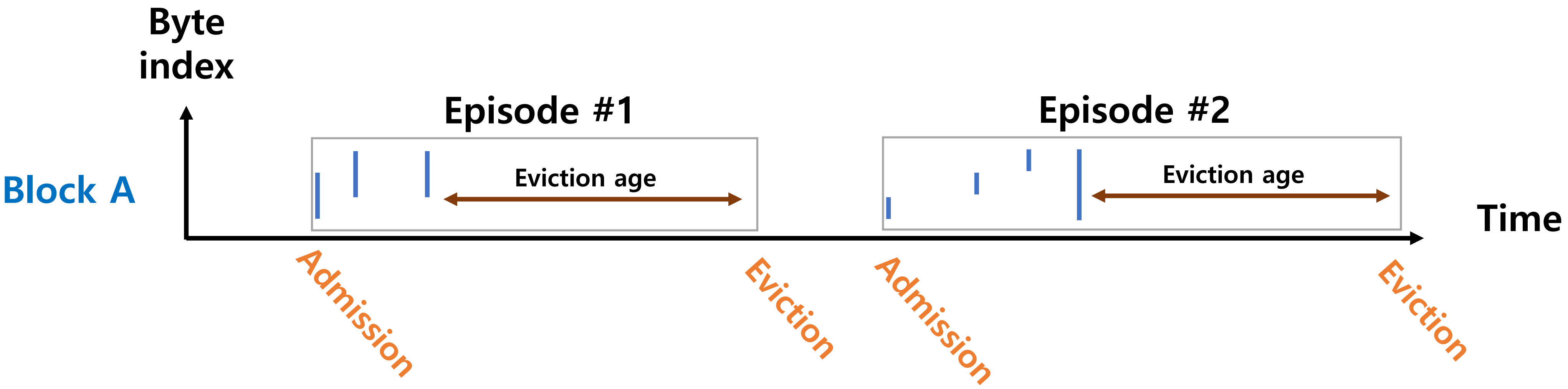
Episodes: from admission to eviction



Episodes: from admission to eviction

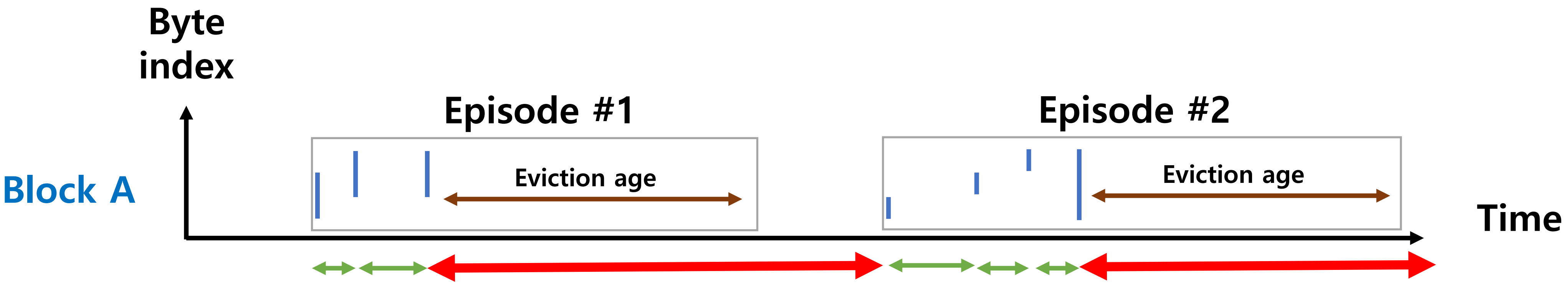


How to know when eviction happens?



How: model LRU cache state with constant eviction age

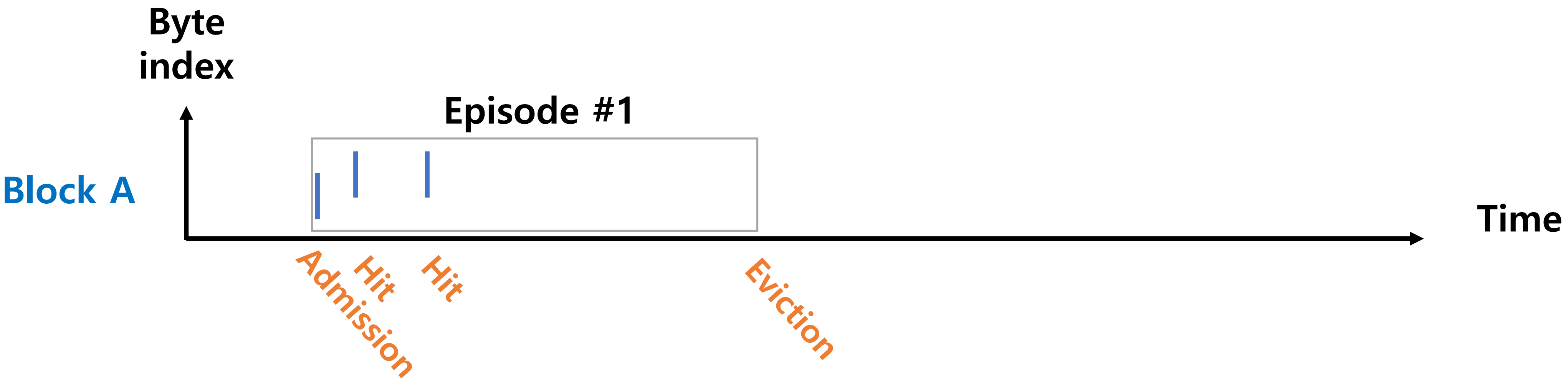
How episodes are generated

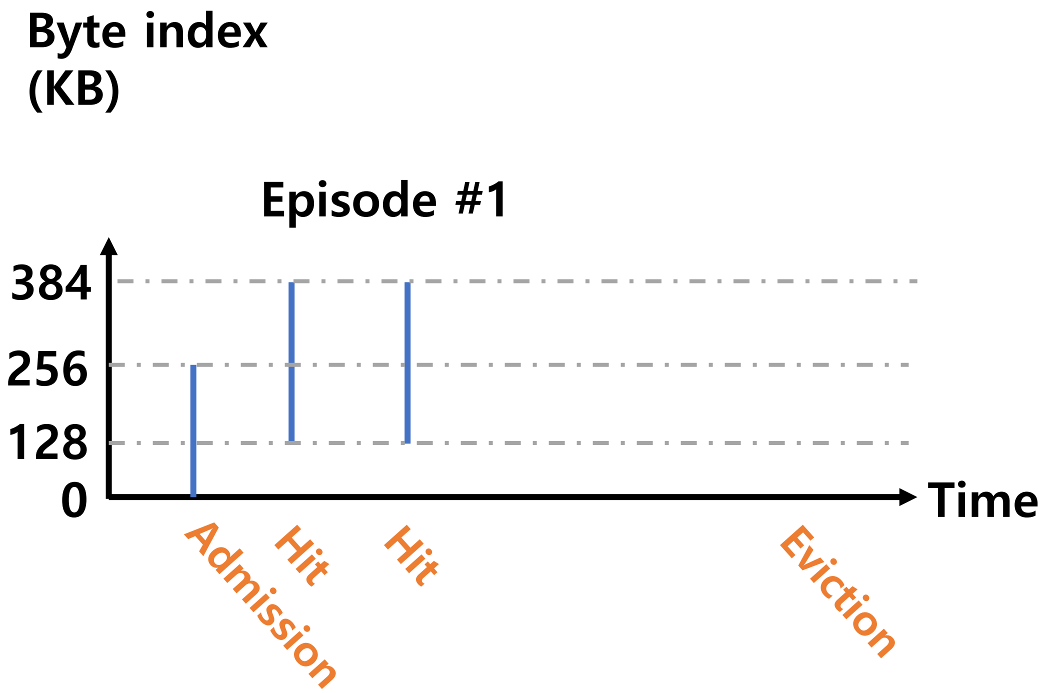


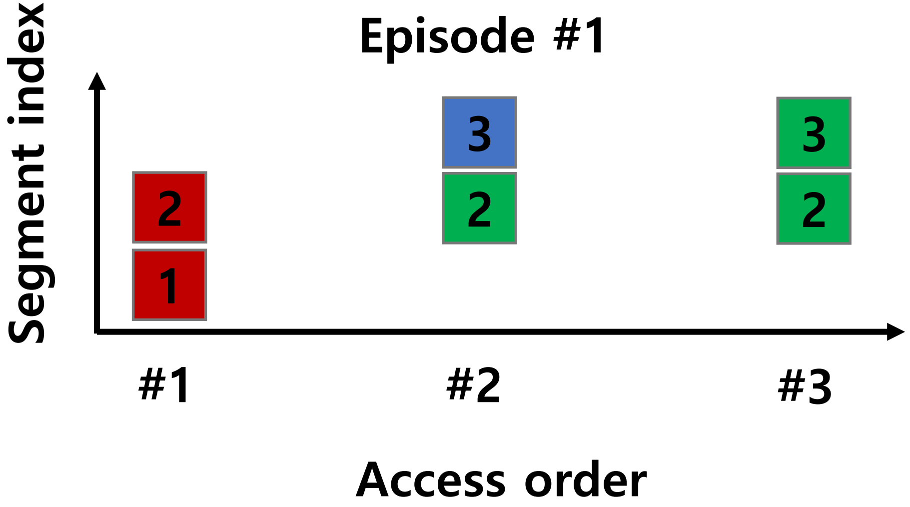
Consider interarrival times of accesses

Split into episodes when interarrival > eviction age

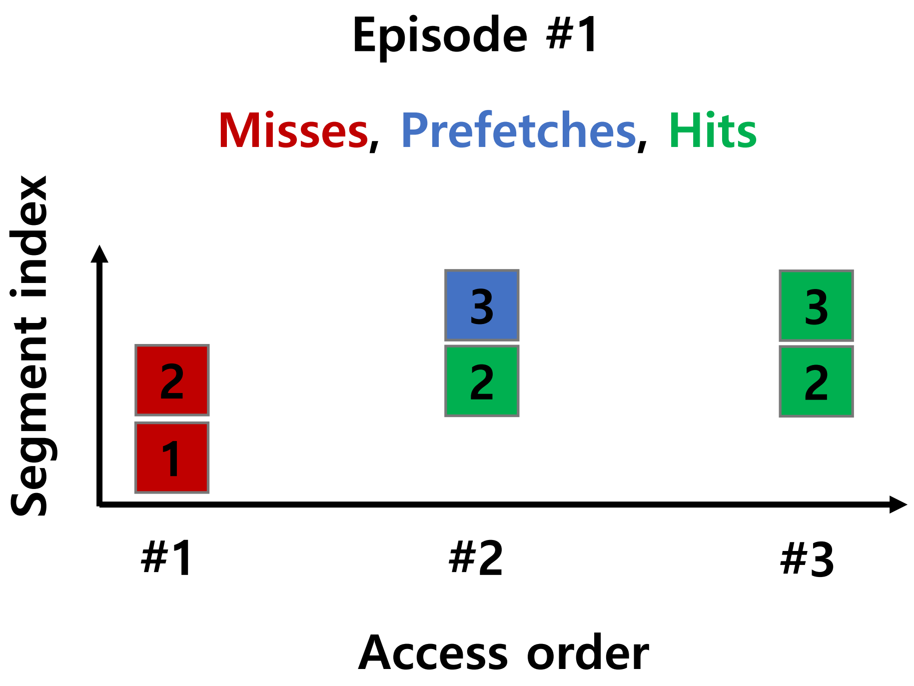
Focusing on Episode 1




Byte index
(KB)
Episode #1
384
256
128
0
Time
Admission
Hit
Hit
Eviction


Episode #1
Segment index
2
1
3
2
3
2
#1
#2
#3
Access order

Benefits & cost defined on episodes



- **Benefit:** 27ms of DT saved
- **Cost: 3** flash writes needed

Admission: Baleen learns from episode-based OPT

OPT (approx. optimal) admits highest scoring episodes

$$\mathrm{Score}(Ep) = \frac{DTSaved(Ep)}{FlashWrites(Ep)} = \frac{27\ ms}{3\ flash\ writes}$$

Episode #1

OPT emits binary labels based on flash write budget

$$Yes\ if\ Score(Ep) > CutoffTa_{rgetFlashWriteRate}$$

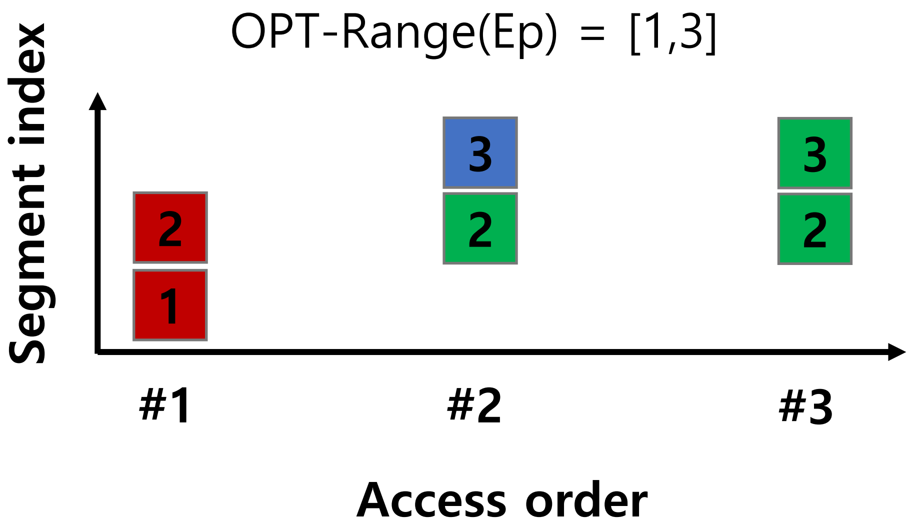
Baleen imitates OPT admission

Baleen's ML-Range learns what to prefetch

- **What** range to prefetch
  - OPT-Range Start: lowest segment
  - OPT-Range End: highest segment

- **ML-Range** is trained on OPT-Range

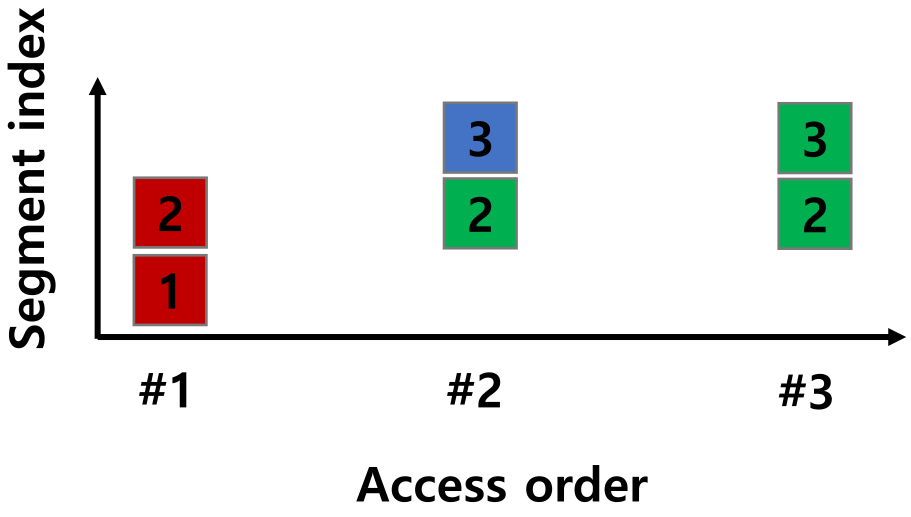
Whole block always fetches [1,64] (8MB)



Baleen's ML-Range learns when to prefetch

- **When** to prefetch
  - Bad prefetching hurts: wasted DT & cache space
  - Prefetch only when confident of benefit
- **ML-When:** Yes if PrefetchBenefit(Ep) > epsilon

epsilon = 5ms



Training Data Comparison

- Trace of accesses

Timestamp	Block ID	Access Frequency	Size (KB)
1	block1	5	32
2	block2	3	16
3	block3	2	64
4	Block1	6	32
5	block2	4	16
6	block4	1	128

- Episodes

block_id	chunk_id	ts_physical	ts_logical	hits	ttl	admission_time	last_access_time	max_interarrival_time
block1	0	1000	0	2	300	900	1000	100
block1	1	1200	200	1	500	1100	1200	200
block2	0	1500	500	3	None	1400	1500	150
block3	0	2000	800	0	400	1900	2000	250

Source: https://github.com/wonglkd/BCacheSim/

Baleen-TCO balances HDD savings against SSD cost

- Q: How to balance #HDD against #SSDs?

HDD cost **+** **SSD cost**

$$\mathrm{TCO}_1 \propto \frac{\mathrm{PeakDT}_1}{PeakDT_0} \cdot \#HDDs_0 + \frac{Cost_{SSD}}{Cost_{HDD}} \cdot \frac{\mathrm{FlashWR}_1}{FlashWR_0} \cdot \#SSDs_0$$

- **Baleen-TCO picks optimal flash write rate**
  - for each workload

*TCO function based on Google's CacheSack [Yang23]

Evaluation

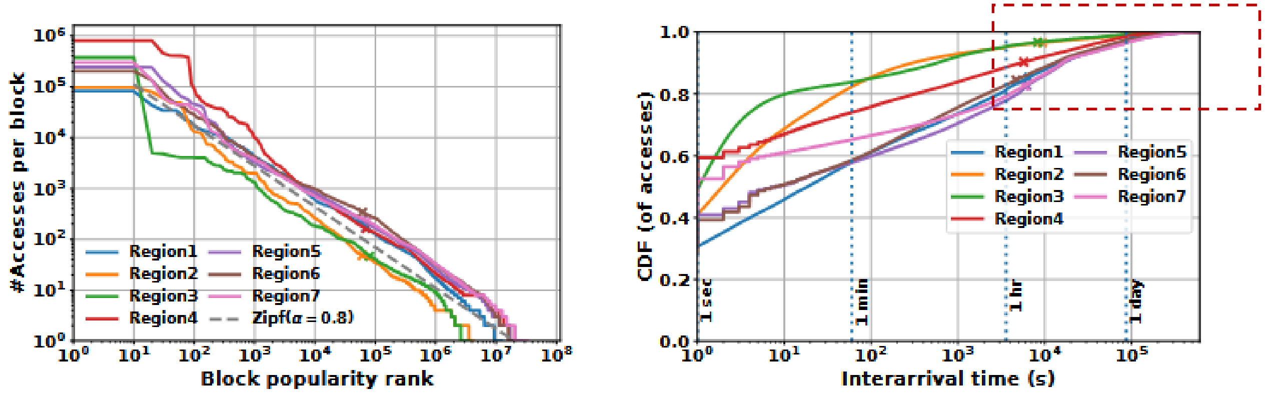
- Production workloads from Meta's Tectonic
  - 7 clusters from 3 years (2019, 2021, 2023)
  - Each serves 1-10 tenants, e.g., data warehouse
  - Each tenant serves 100s of applications
- More details on Tectonic in Pan et al (FAST 2021)
- Traces & simulator code released
- Hardware
  - The Tectonic production setup used to record traces and counter values
    - 400 GB flash cache
    - 10 GB DRAM cache
    - 36 HDDs
  - Their testbed
    - 24-node cluster
      - each node has a
        - 16-core Intel Xeon E5-2698 CPU
        - 64 GB of DRAM
        - Intel P3600 400 GB NVMe SSD
        - Seagate ST4000NM 4 TB HDDs

Less than 20% of interarrival times exceed the eviction age

Table 1: Key statistics of traces.

Dataset	Req Rate (s^{-1})	Access size (MB)	CMR[1]	OHW[2]	Admit-All Writes (MB/s)
Region1	244	3.41	18%	54%	316
Region2	106	2.85	39%	83%	121
Region3	139	2.42	19%	48%	45
Region4	406	2.87	14%	53%	280
Region5	364	2.62	18%	59%	480
Region6	404	2.74	14%	55%	478
Region7	426	2.23	17%	62%	492

[1] CMR (Compulsory miss rate): ratio of blocks to accesses;
[2] OHW (One-hit-wonder): % of blocks with no reuse.



(a) Block popularity (log-log). **(b)** Interarrivals to same block.

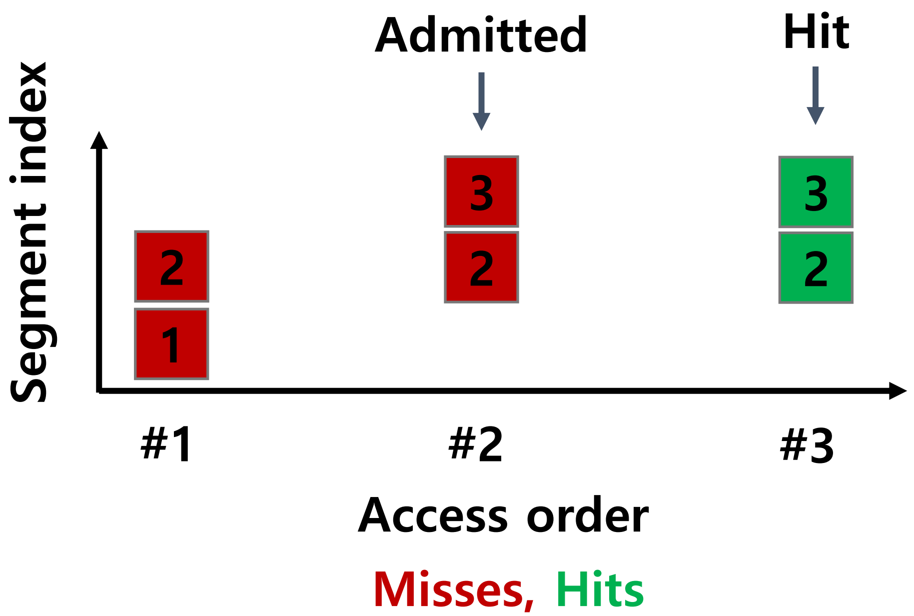
Figure 7: Block popularity and access interarrival. In a, lower values of α indicate it is harder to cache, and $\times$ denotes 400 GB. In b, $\times$ denotes eviction ages for Baleen at 400 GB & 3 DWPD.

The popularity distribution of all trace blocks fit a Zipf (a=0.8) (moderate skew)

- Accesses are more spread out across many blocks.
- This means many blocks need to be cached to achieve a good hit rate.
- Since caches have limited capacity, it becomes harder to predict and store all the required blocks, resulting in more misses.

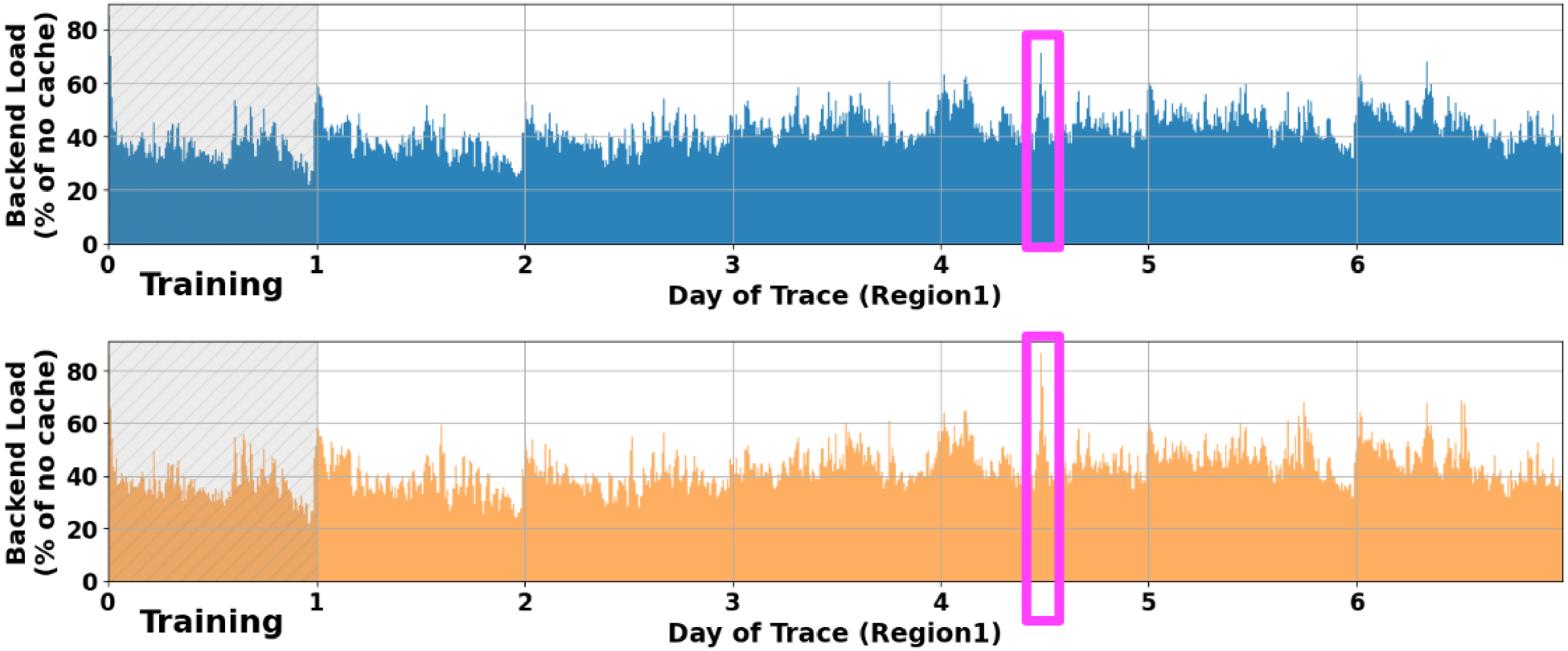
Baseline admission policies

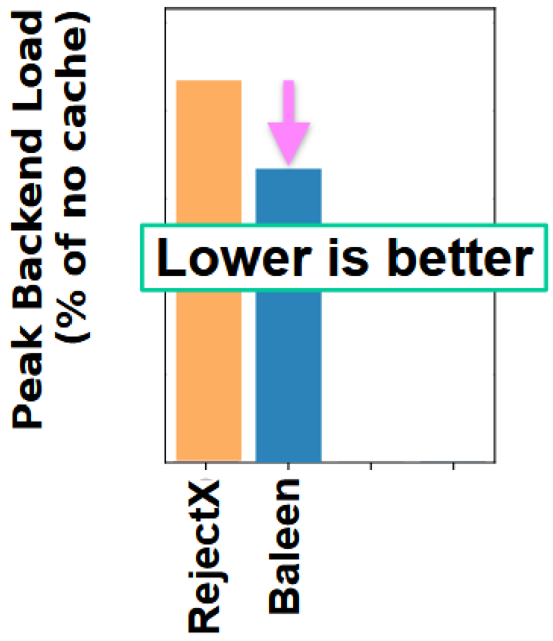
- CoinFlip: flip a coin for each IO
  - Simplest, requires no state
- RejectX (e.g., X=1: accept segment after 1 reject)
  - Used by Meta, Google as baseline
  - 2nd access is always a miss
- CacheLib-ML
  - Used by Meta in production for 3 years
  - Trained on accesses, not episodes



Reduce Peak

- Train (offline) on day 1 and evaluate on day 2-7
- Compare policies Peak DT (as a % of no caching)





Minimize peak backend load to minimize cost

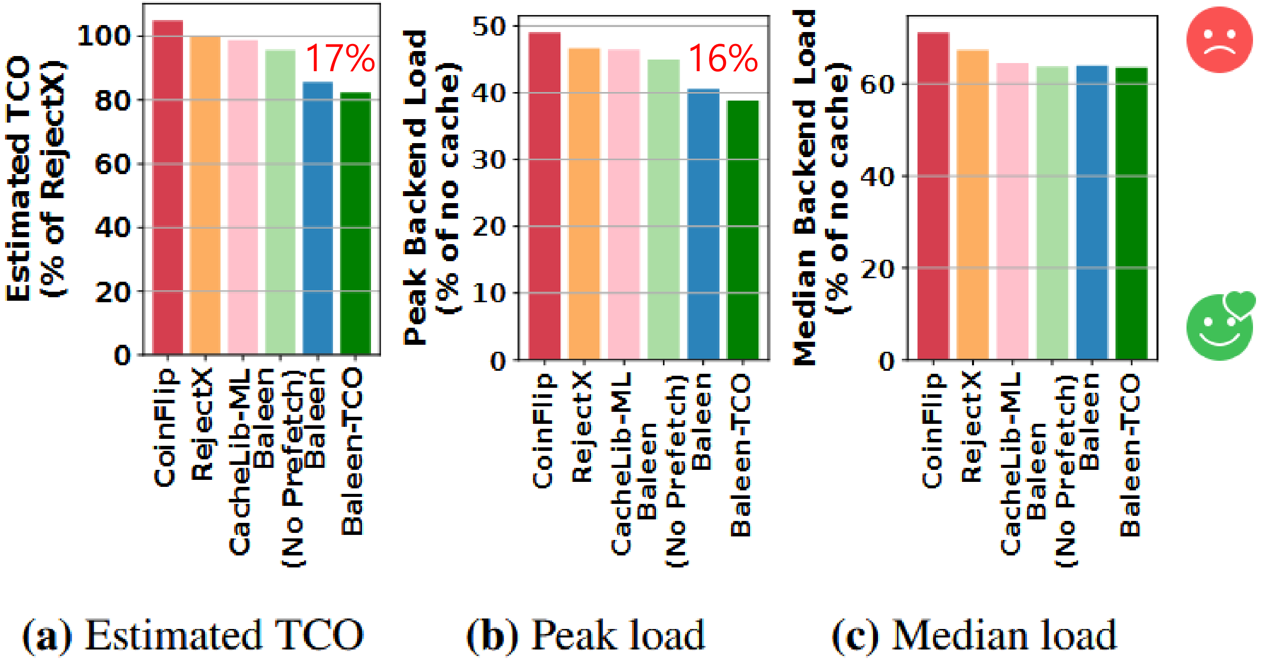


Figure 1: Baleen-TCO reduces (estimated) TCO by 17% and peak load by 16% over the best baseline on 7 Meta traces by choosing the optimal flash write rate. IO and byte miss rates were reduced by 14% and 2% (Suppl A.1). For the default flash write rate, Baleen reduces peak load by 12% over the best baseline.

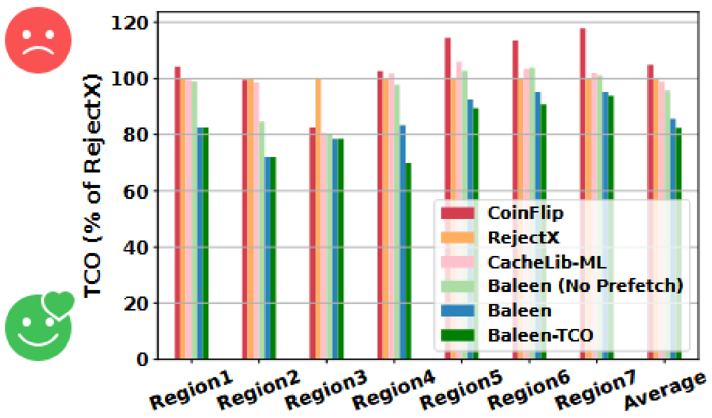


Figure 8: Baleen-TCO reduces TCO.

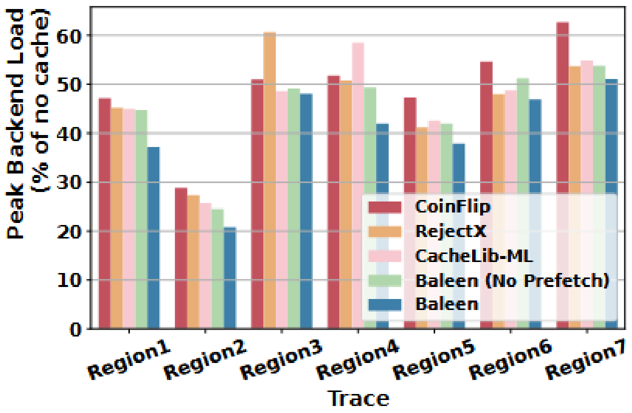


Figure 9: Baleen reduces Peak DT.

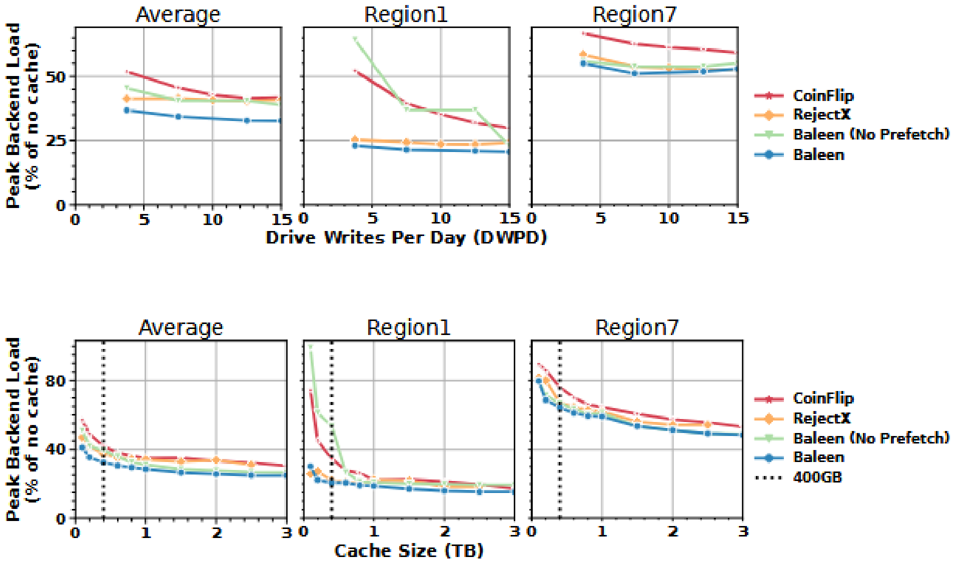


Figure 10: Benefits at higher write rates & cache sizes.

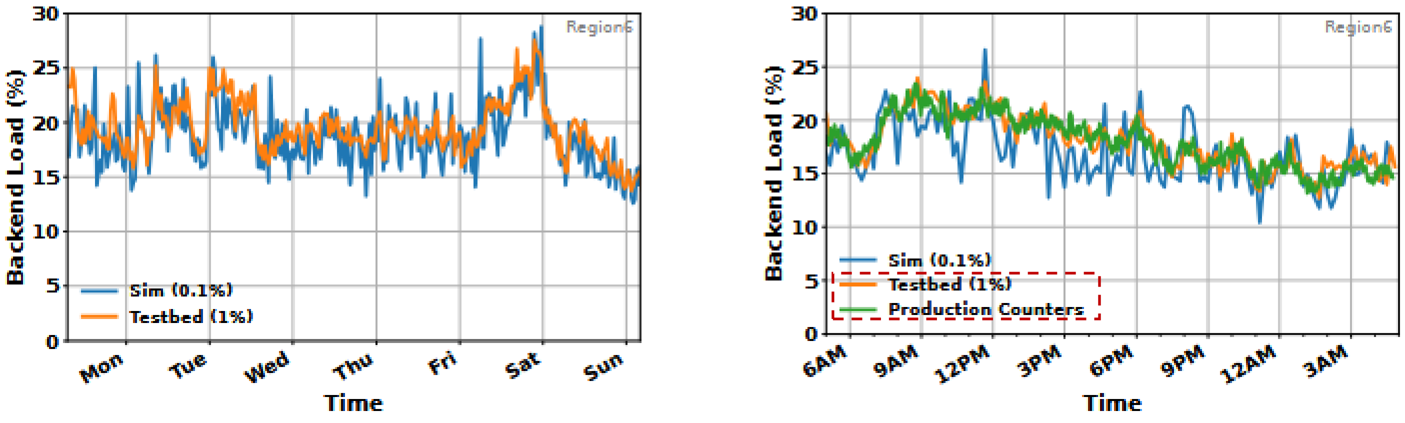
Reduce Total Cost

Reduce Peak IO

Higher Cache Usage

Validation of simulator and testbed

Baleen @ Simulator
Vs
Baleen @ Testbed



(a) Sim vs Testbed, Baleen (b) Testbed vs Prod, RejectX

Figure 11: Validation of simulator and testbed.

Testbed is consistent with production counters

Online flash caching simulator :
a Python simulator to accurately estimate CacheLib performance without doing the actual heavy lifting.

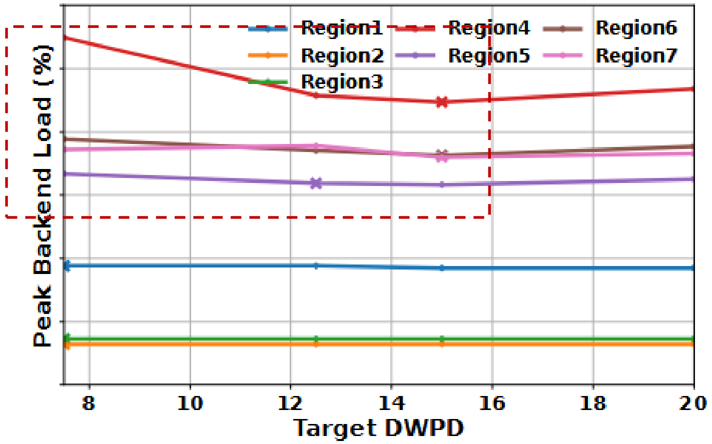


Figure 12: Baleen-TCO chooses higher flash write rates when needed to lower peak backend load (and TCO). × denotes the optimal flash write rate for that workload.

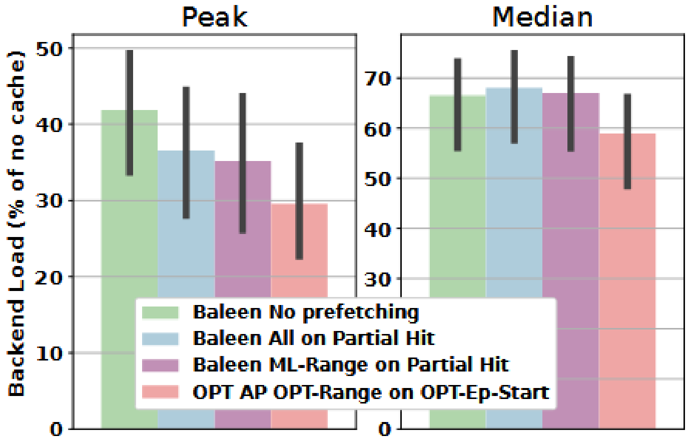


Figure 13: ML-Range saves Peak DT. ML-Range outperforms the baseline (whole block) and No Prefetching at the expense of Median DT.

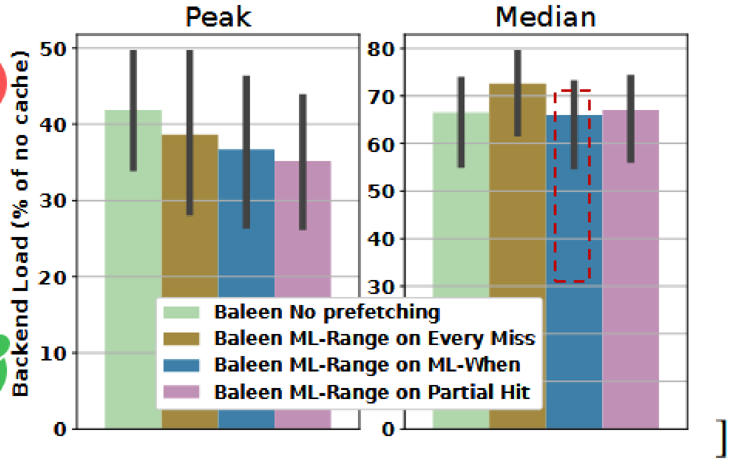


Figure 14: Choose *when* to prefetch. Indiscriminate prefetching (on Every Miss) can hurt. Using ML-When or Partial Hit reduces Peak DT without compromising Median DT.

DWPD : Drive Write Per Day

Simple baseline (All on Partial Hit) Segment [1-64]

Confident of ML-When is better in Median

Lessons from deploying ML in prod

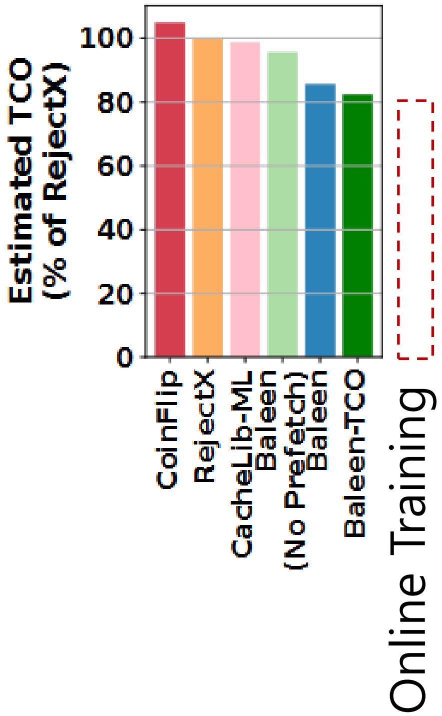
- **Optimize the Right Metric**:
  - Focus on system-level metrics like Disk-head Time (DT), not just hit rate.
- **Production ≠ Development**:
  - ML performance differs between environments; simulate before full deployment.
- **Encapsulate Expertise**:
  - Simplify integration by bridging ML, caching, and storage with shared frameworks.
- **Plan for Model Decay**:
  - Models degrade over time; design for generalization and retraining.
- **Commit to Automation**:
  - Avoid manual dependencies to reduce regressions and improve sustainability.

Conclusion

- **Baleen** leverages ML to optimize prefetching and cache admission, achieving:
  - 16% reduction in peak disk time
  - 17% reduction in Total Cost of Ownership (TCO) on real workloads.
- Its design is built on lessons from early missteps and introduces an **episodes-based** formulation for effective training and ML-guided prefetching.
- Baleen marks a significant advancement in flash caching for disk storage, demonstrating the value of ML in system optimization.

Next Paper Idea

- Implement Online Training with low overhead with simpler model
  - Finding simpler model that can be implemented with episodes.



Q&A

Baleen: ML Admission & Prefetching for Flash Caches

Thank You !

2025. 01. 15

Presented by Ramadhan Agung Rahmat

agung@dankook.ac.kr