LDH의 논믈리에
S3-FIFO와 SIEVE 캐시 자료구조 소개

BBConf 2023 Winter

오늘은 아래 2개의 논문을 다룹니다

- SIEVE is Simpler than LRU: an Efficient Turn-Key Eviction Algorithm for Web Caches (NSDI'24)
- FIFO queues are all you need for cache eviction (SOSP'23)

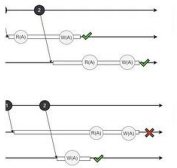
넌 누구냐

취미로 논문 읽는 그룹 운영중

Published in 취미로 논문 읽는 그룹 · Sep 4

DynamoDB의 시스템 디자인과 분산 트랜잭션 구현 원리

DynamoDB는 AWS 관리형 분산 키-밸류 저장소입니다. 만약 본인이 AWS 클라우드를 이용하고 있고 대규모 트래픽을 견뎌야 하는 키-밸류 NoSQL 데이터베이스가 필요하다면 DynamoDB를 검토하게 되실 겁니다. AWS...

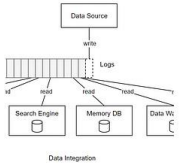


Dynamodb 35 min read

Published in 취미로 논문 읽는 그룹 · Aug 14

소프트웨어 엔지니어가 알아야 할 로그에 대한 모든 것

Apache Kafka의 탄생배경을 알아보자 — 많은 시스템이 Apache Kafka를 메시지 큐와 실시간 데이터 처리에 이용하고 있습니다. Linkedin 에서 Apache Kafka를 개발한 Jay Kreps는 이 글에서 Kafka를 개발하게 된 이...



Distributed Systems 26 min read

Published in 취미로 논문 읽는 그룹 · Jun 26

벡터 데이터베이스 소개 | 음악 검색 기능은 어떻게 만드는 걸까?

벡터 임베딩(Vector Embedding)은 머신 러닝의 핵심 요소로 이미지, 음성, 단백질 분자 구조 등의 비정형 데이터를 벡터로 표현하는 방법이다. 비정형 데이터를 일단, 벡터로 표현하고 나면 이를 수로 취급할 수 있어...

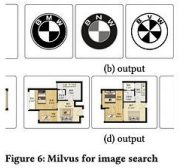


Figure 6: Milvus for image search

scalalang2

388 Followers

평범한 프로그래머입니다. 취미 논문 찾아보기, 코딩 컨테스트 등 / Twitter @scalalang2

Edit profile

Following

- Prof Bill Buchanan OBE
- NAVER CLOUD PLATFORM
- Interchain
- Sigrid Jin
- Mijeong (Rachel)

See all (110)

넌 누구냐

- 닉네임 : LDH
- 신촌 슈퍼스타 raararaara를 만나 Competitive Programming 입문
- raararaara의 모교, R관에 있는 한 연구실에서 생활, 대학원 신분세탁
- 라인 인턴 하다 중간에 넥슨으로 이동해 월급도둑으로 활약중
- 취미로 논문 읽는 모임 운영중

넌 누구냐

CP 인생 최고점 : ABC 329회차 컨테스트 8848명 중 1096th 한마디로 그냥 잘 못해요..

내년엔 앳코더 민트가 목표 …

5 Kyu

scalalang ★

[Change Photo]

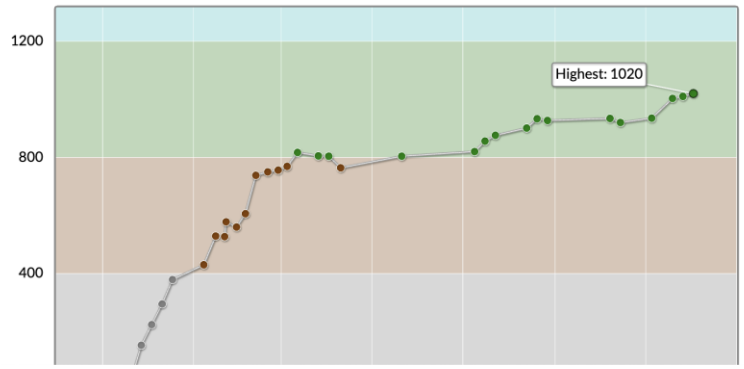
Country/Region	Republic of Korea
Birth Year	1993
Affiliation	NEXON

Contest Status

Algorithm | Heuristic

Rank	11351st
Rating	1020
Highest Rating	1020 — 5 Kyu (+180 to promote)
Rated Matches	33
Last Competed	2023/12/02

1020 2287th +10 2023. 12. 2. Daiwa Securities Co. Ltd. Programming Contest 2023 (AtCoder Beginner Contest 331)



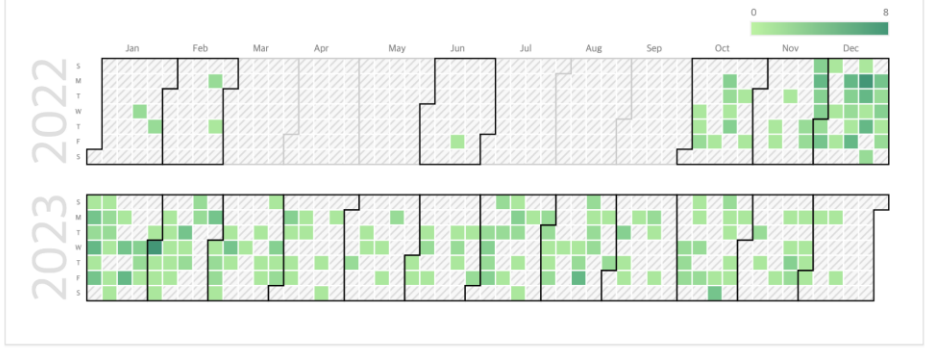
scalalang

good luck, have fun :)

정보 언어

등수	1961
맞은 문제	781
시도했지만 맞지 못한 문제	10
제출	2041
맞았습니다	853
출력 형식	4
틀렸습니다	691
시간 초과	168
메모리 초과	77

2022-2023년



이제 시작합니다.

소프트웨어 캐시

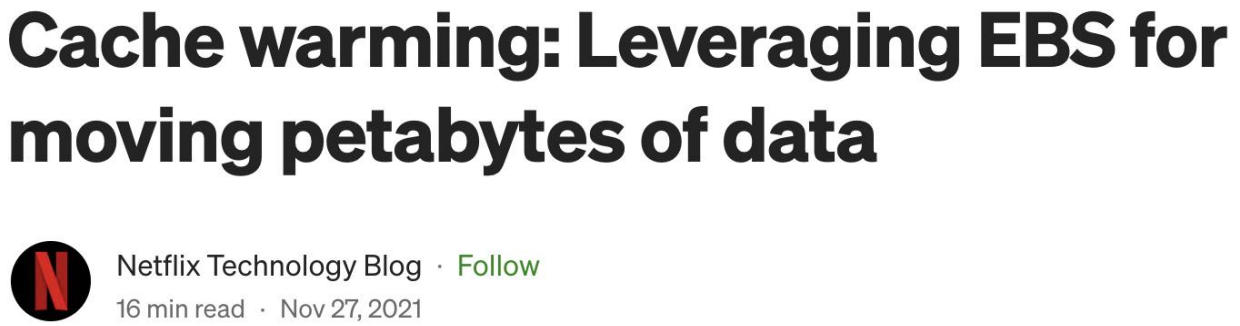
- 성능을 높이기 위해 쓰임
  - 키/밸류 DB, 웹 캐시, CDN 등 메모리나 지역에 미리 데이터를 올려둠
  - 반복적인, 중복되는 계산을 미리 해두기 (e.g. 대통령 선거 실시간 투표 집계)
- 캐시 히트율이 높다?
  - 사용자의 평균 레이턴시가 감소

소프트웨어 캐시

넷플릭스는 18,000여대의 캐시 서버가 14 페타바이트 이상의 데이터를 서빙중

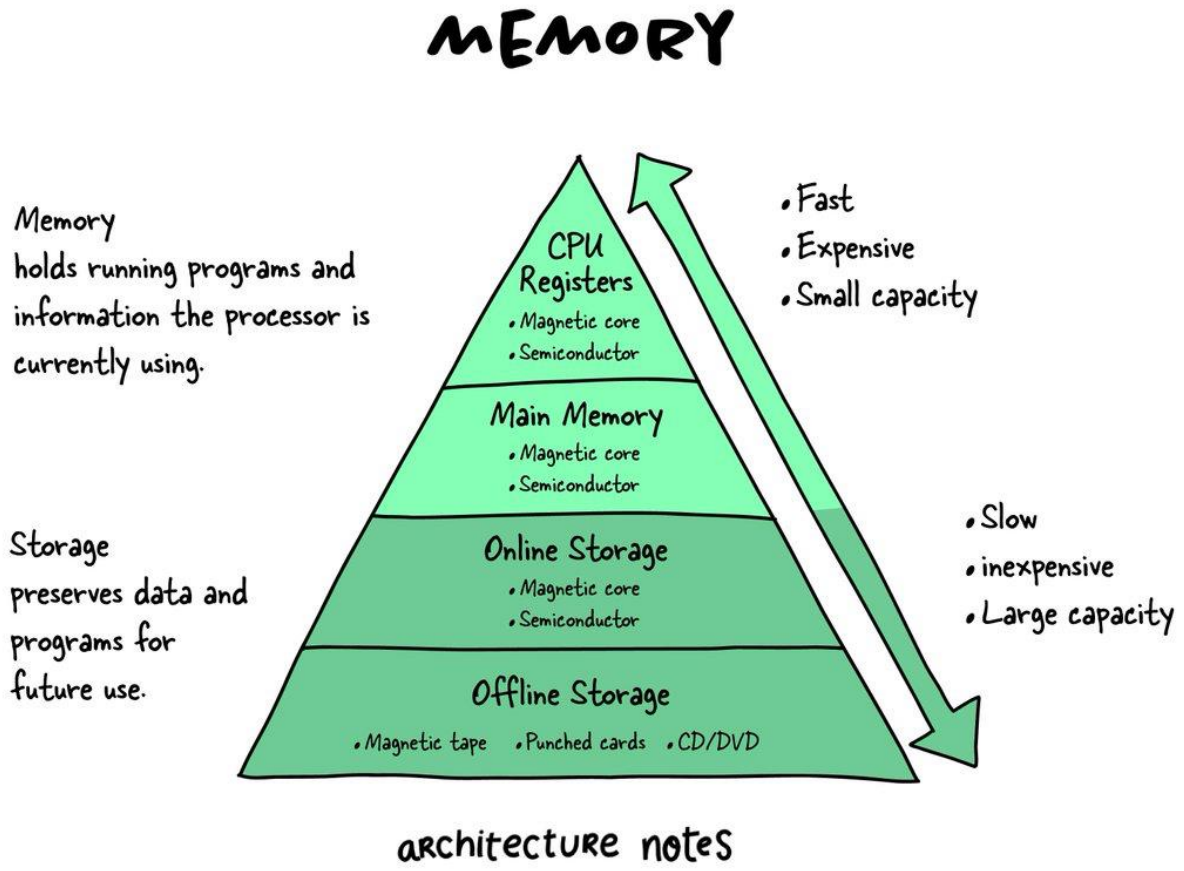
트위터의 캐시 서버를 다 합치면 10만 CPU, 100TB 이상 DRAM 메모리에 맞먹음

캐시 성능 1%를 높인다는게 얼마나 큰 비용을 절약해주는지 쉽게 상상 할 수 있음



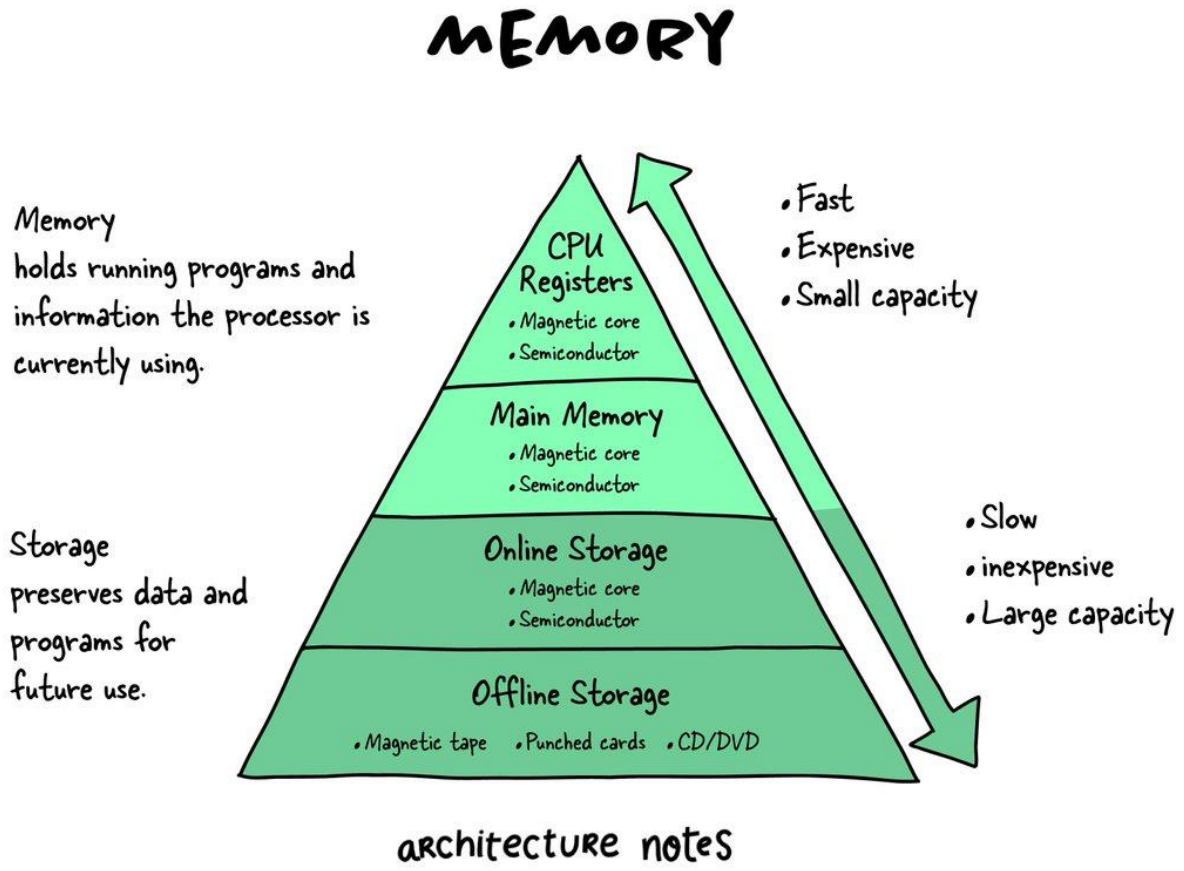
소프트웨어 캐시

후려쳐서 말하면 메모리는 CPU와 가까울 수록 비싸고 용량이 적지만 레이턴시가 빠르고 멀어질수록 반대의 성향을 가진다. (거의 모든 계층에 캐시 알고리즘이 존재 L1, L2, ⋯, LLC 등)



소프트웨어 캐시

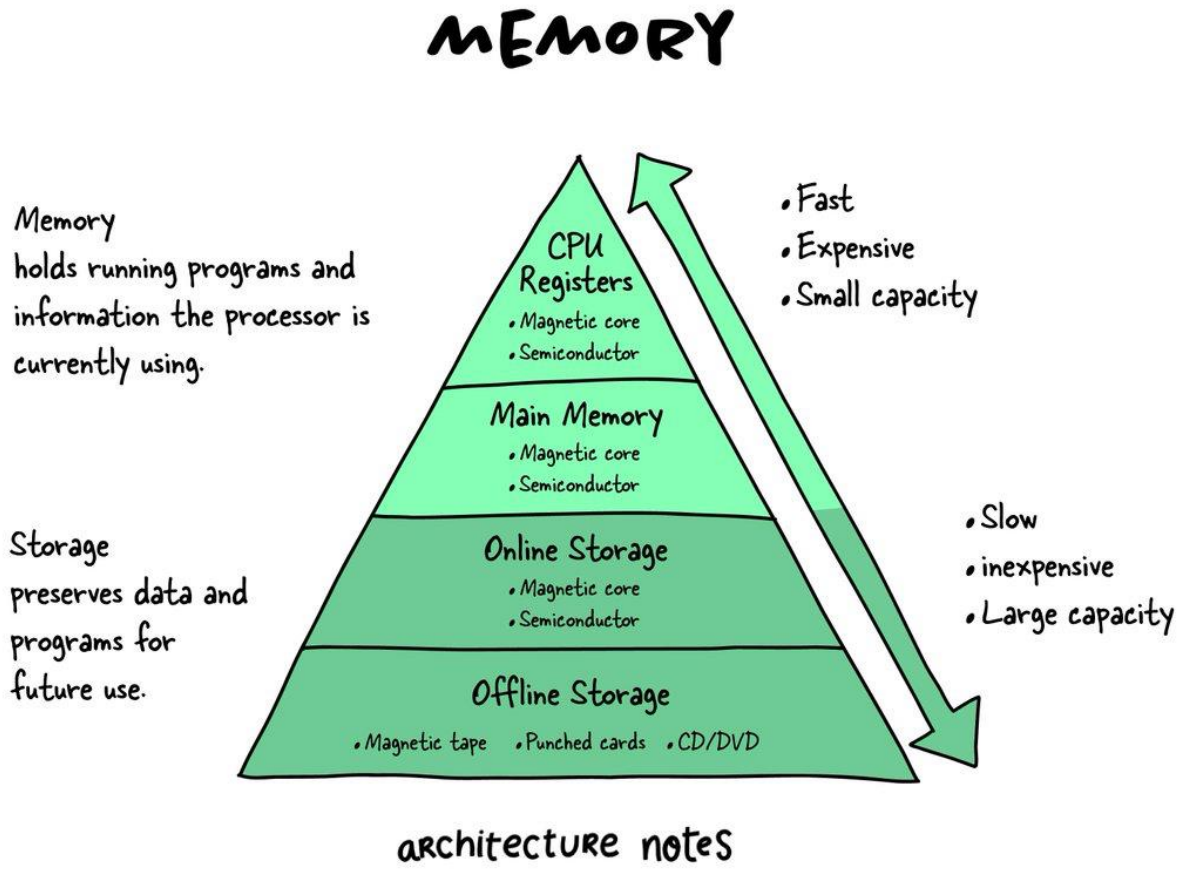
캐시의 핵심 알고리즘은 부족한 용량안에서 hot data를 최대한 남기고 cold data를 밀어내는 정책이다. 이를 eviction 이라고 말함



소프트웨어 캐시

사실 hot data라는 평가를 정확히 내릴 수는 없다. 미래를 알 수 없기 때문,

그래서 최신성과 빈도 이 두가지 정보를 가지고 미래를 예측하는 캐시를 만듬 (e.g. LRU/LFU)



캐시를 평가하는 메트릭들

효율성

처리율

확장성

플래시 친화성

심플함

캐시를 평가하는 메트릭들

효율성 — 사용자 요청이 들어왔을 때 캐시에 얼마나 자주 데이터가 남아있는가?

처리율 — 캐시 히트가 높다 = 효율성이 높다

확장성

플래시 친화성

심플함

캐시를 평가하는 메트릭들

효율성	캐시가 초당 몇 건의 쿼리를 처리하는가
처리율	QPS (Queries Per Seconds)
확장성	
플래시 친화성	
심플함	

캐시를 평가하는 메트릭들

효율성

처리율

확장성

플래시 친화성

심플함

동시 접근하는 쓰레드/CPU를 늘리면 QPS가 증가하는가?

캐시를 평가하는 메트릭들

효율성

처리율

확장성

플래시 친화성

심플함

캐시가 얼마나 플래시 저장 장치에 친화적으로 설계되었는가?

일반적인 DRAM = 휘발성

플래시 메모리 = 비휘발성

캐시를 평가하는 메트릭들

효율성

처리율

확장성

플래시 친화성

심플함

플래시 메모리 P/E 사이클

플래시 메모리의 한 셀은 지정된 쓰기/삭제 사이클을 넘기면 더이상 쓰지 못한다

사용할 수 있는 수명이 정해져 있다.

SK hynix NEWSROOM

[반도체 특강] 메모리 반도체 신뢰성(Reliability)上- 보존성 (Retention)과 내구성(Endurance) 편

진종문 반도체특강

캐시를 평가하는 메트릭들

효율성

처리율

확장성

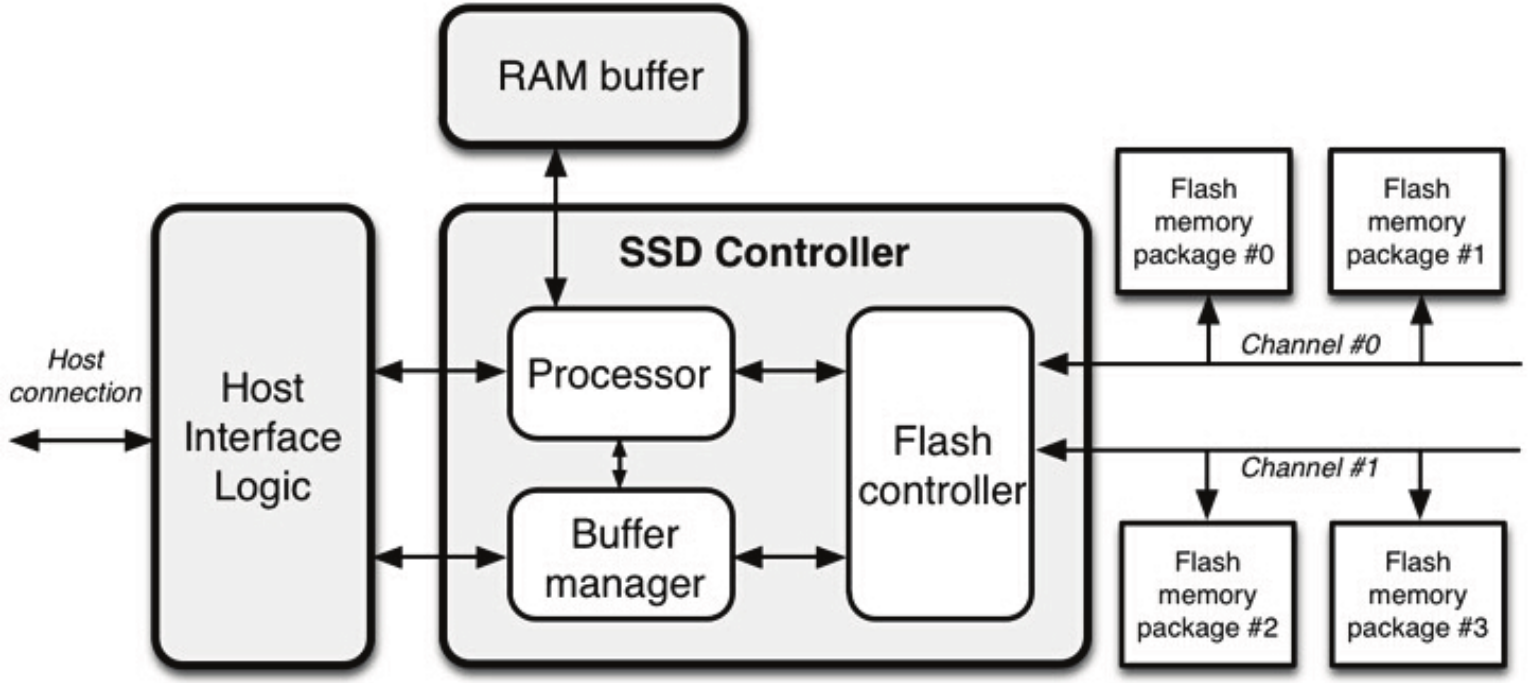
플래시 친화성

심플함

갑자기 플래시 친화성이라니?

SSD를 얼마나 오래 쓸 수 있는가를 의미한다 CDN 서버들은 주로 SSD를 캐시로 사용

SSD = NAND 플래시 메모리의 집합체



캐시를 평가하는 메트릭들

효율성

처리율

확장성

플래시 친화성

심플함

논문에서 말하는 오래 쓰는 방법

되도록이면 SSD말고 DRAM에 중요한 정보 올려두기

Random Access를 피하기

캐시를 평가하는 메트릭들

효율성 알고리즘이 얼마나 간편한가를 의미한다

처리율 TinyLFU, SLRU, ARC 등 많은 알고리즘이 있지만, 거의 잘 안씀

확장성 복잡한 알고리즘 =〉 버그의 출현 가능성 증가

플래시 친화성

심플함

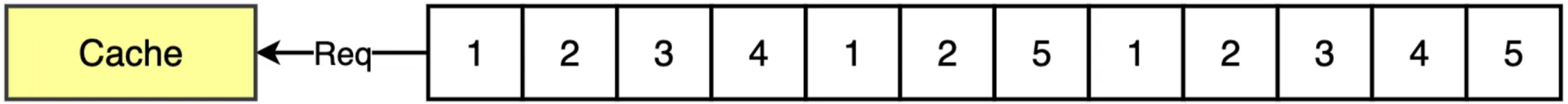
과거 캐시 연구 동향

- FIFO 큐를 이용하면 Bélády's Anomaly (1960s)가 발생한다고 알려져 있다
- 그 이유로 한동안 LRU 캐시가 대세를 이룬다
  - 2000s : LRU 캐시 2~3개 섞어 쓰는 복잡한 캐시 구조가 탄생
  - 2010s ~ now : 복잡한 평가 메트릭이나 머신러닝 기법을 사용함

Bélády's Anomaly (1960s)

아래 워크로드가 FIFO 캐시로 주어진다고 상상해보자

SIZE = 3, X = 캐시 히트, PF = 페이지 폴트

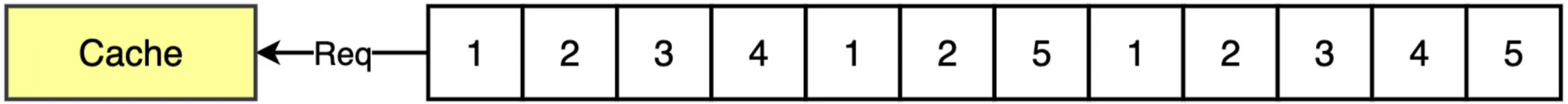


1	1	1	2	3	4	1	1	1	2	5	5
기	2	2	3	4	1	2	2	2	5	3	3
		3	4	1	2	5	5	5	3	4	4
PF	PF	PF	PF	PF	PF	PF	PF	X	X	PF	PF	X

Bélády's Anomaly (1960s)

아니?? 캐시 크기를 늘렸는데 왜 캐시 히트가 오히려 감소해?

명확한 수학적 증명이 있겠지만, 캐시에서 제거할 객체의 우선순위가 없기 때문이라고 이해하자



1	1	1	1	1	1	2	3	4	5	1	2
	2	2	2	2	2	3	4	5	1	2	3
		3	3	3	3	4	5	1	2	3	4
			4	4	4	5	1	2	3	4	5
PF	PF	PF	PF	X	X	PF	PF	PF	PF	PF	PF

Bélády's Anomaly (1960s)

요청 워크로드는 Temporal Locality 성질을 가지고 있다

Belady's Anomaly 현상과 Temporal locality 에 근거하여 캐시 연구는 LRU를 중심으로 많이 발전함

Bélády's Anomaly (1960s)

그러나 LRU는 3가지 문제점이 있었으니

1. 객체당 2개의 포인터를 요구해 객체 크기가 작을수록 오버헤드가 커짐
2. 캐시 히트가 발생하면 락을 잡고 객체를 head로 땡겨와야함
   - 최소 6번의 랜덤 엑세스 발생
   - Meta(구 페이스북) 개발자들이 RocksDB의 주요 병목으로 LRU 캐시의 락 컨텐션 문제를 지적함
3. 플래시 친화적으로 설계되지 않음

Reducing Lock Contention in RocksDB

Posted May 14, 2014

원-히트-원더 (One-hit-wonder)

요청 시퀀스에서 단 한 번만 등장하는 비율을 의미함

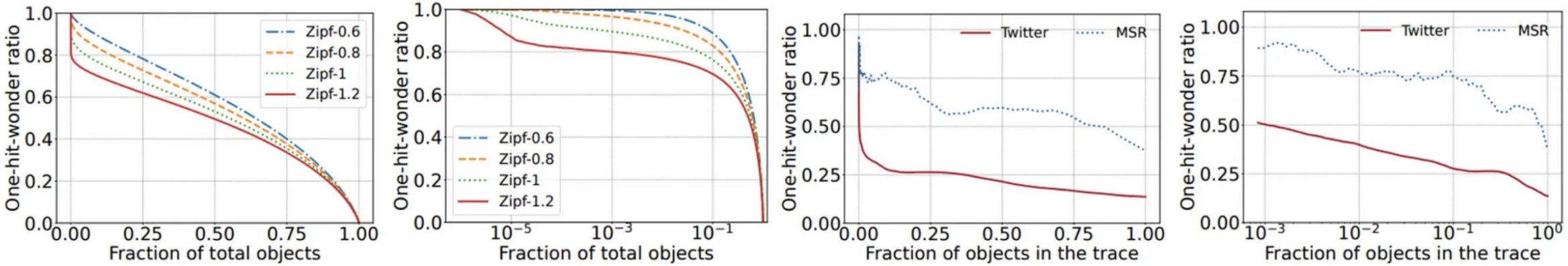
start time	end time	sequence length (# objects)	# one-hit wonder	one-hit-wonder ratio
1	17	5	1 (E)	20%
1	7	4	2 (C, D)	50%
1	4	3	2 (B, C)	67%

Figure 1. A shorter sequence has a higher one-hit-wonder ratio.

원-히트-원더 (One-hit-wonder)

모던 애플리케이션 워크로드는 power law distribution을 따른다고 알려져 있음

아래 그림은 Zipf's distribution과 Twitter / MSR 에서 공개한 데이터의 원-히트-원더를 보여줌



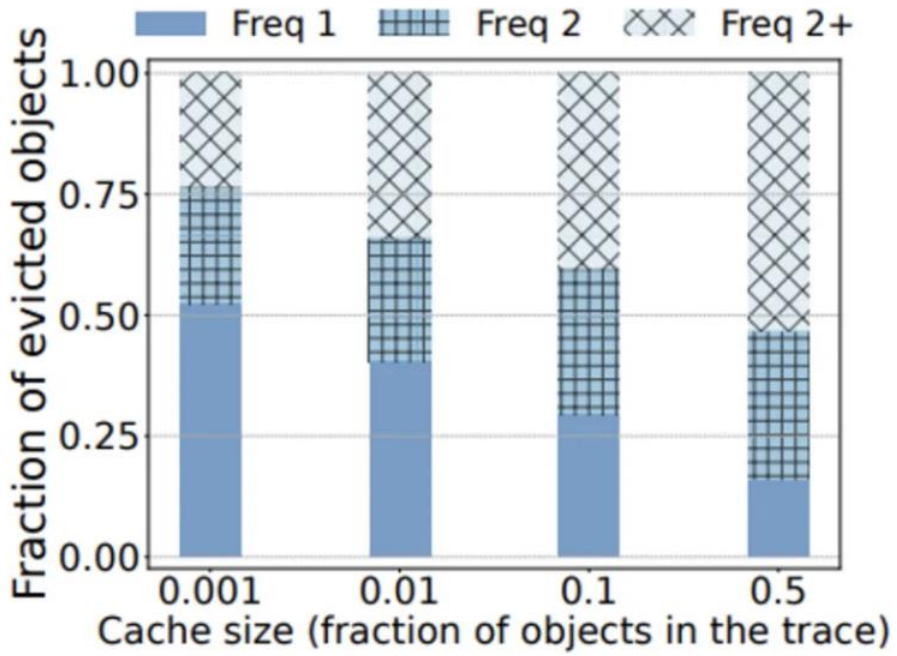
(a) Zipf trace, linear-scale (b) Zipf trace, log-scale (c) Production traces, linear-scale (d) Production traces, log-scale

Figure 2. Left two: the one-hit-wonder ratio decreases with sequence length (as a fraction of the unique objects in the full sequence) for synthetic Zipf traces. Different curves show different skewness α. We plot both linear and log-scale X-axis for ease of reading. Right two: production traces show similar observations. Note that the X-axis shows the fraction of objects in the trace, much smaller than the number of possible objects in the backend. Therefore, the production curves capture the left region of the Zipf curves.

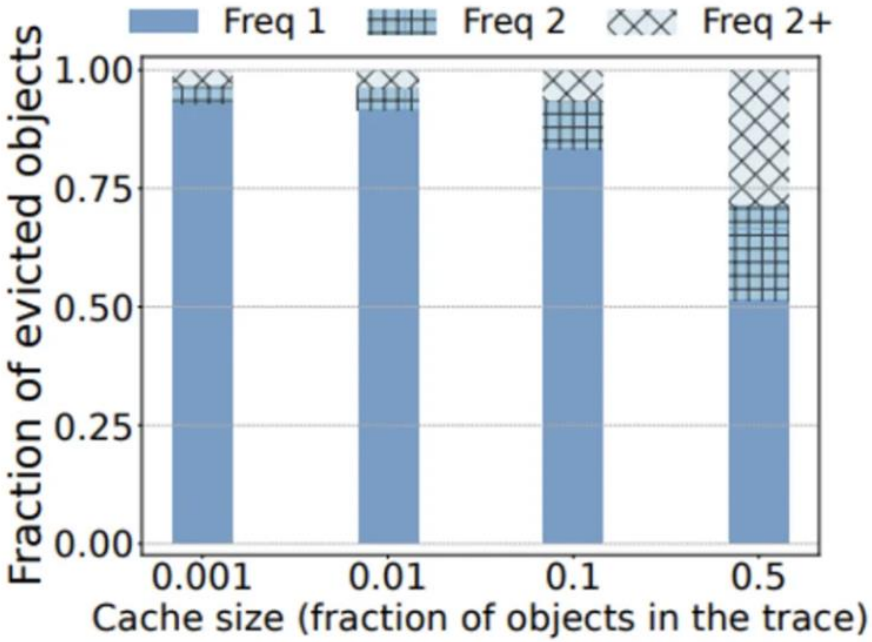
원-히트-원더 (One-hit-wonder)

이를 LRU관점에서 Eviction 되는 시점의 객체들의 빈도수를 가지고 이야기 해보자

- 트위터 데이터는 일반적으로 skewed 되어 있다고 알려짐 (OSDI'20)
- MSR의 경우 캐시 사이즈가 작으면 대 부분의 데이터의 freq 값이 1 이다.



(a) Twitter trace, LRU



(c) MSR trace, LRU

Quick Demotion

빠르게 강등시키기 / LRU 캐시에서 head로 올리는 일을 승진(Promotion) 이라고 함

자리만 차지하는 쓸모 없는 객체를 빠르게 제거 할 필요가 있음

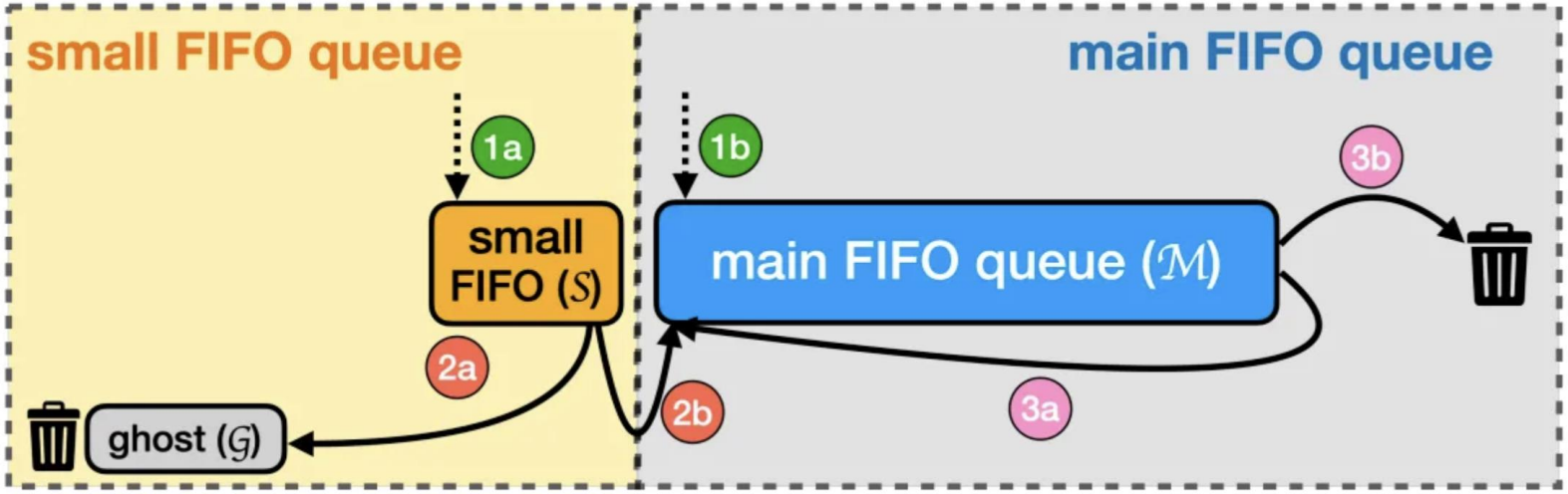
- CDN은 bloom filter를 사용하고 있지만, 너무 빠르게 제거해서 정확도가 떨어진다
- ARC, LIRS에도 비슷한 역할을 하는게 있지만, 캐시 객체의 생존 기간을 표현하지 못함
  - 너무 빨리 제거되거나, 느리게 제거 되거나

S3-FIFO 큐 구현

Quick Demotion을 만족하는 설계를 만들어보자

main queue, small queue 와 ghost로 구성됨

main queue = 90% of cache size / small queue = 10% of cache size / ghost = bucket hash table



Insert: if not in ghost, insert to small 1a, else insert to main 1b

Evict (small): if not visited, insert to ghost 2a, else insert to main 2b

Evict (main): if visited, insert back 3a, else evict 3b

S3-FIFO 큐 구현

읽기 연산

캐시 히트가 발생하면 freq 값을 최대 3까지 갱신함

즉 객체마다 2 비트만 요구한다. LRU는 객체마다 2개의 포인터를 요구함

```
1: function READ(x)
2:     if x in S or x in M then                          ▷ Cache Hit
3:         x.freq ← min(x.freq + 1, 3)                   ▷ Frequency is capped to 3
4:     else                                              ▷ Cache Miss
5:         insert(x)
6:         x.freq ← 0
```

S3-FIFO 큐 구현

쓰기 연산

일반적인 캐시 구현체들은 삽입 연산에 캐시가 꽉 차있으면 eviction 을 수행한다.

evict() 로 캐시 공간을 확보 한 후, G에 등록된 키가 있으면 M으로 없다면 S로 삽입한다.

즉, G의 역할은 한 번 발견했던 객체에 2번째 기회를 주는 것, $\text{CLOCK}_{1969년}$ 캐시와 유사

```
7:  function INSERT(x)
8:      while cache is full do
9:          evict()
10:     if x in G then
11:         insert x to head of M
12:     else
13:         insert x to head of S
```

S3-FIFO 큐 구현

캐시 축출

small cache가 전체 크기의 10%를 넘었다면 small의 공간을 확보함

그게 아니라면 main 에서 공간을 확보함

```
14: function EVICT
15:     if S.size ≥ 0.1 · cache size then
16:         evictS()
17:     else
18:         evictM()
```

S3-FIFO 큐 구현

EVICT_S

freq값이 2이상이면 Main으로 이동시킨다.

아니라면, 삭제하고 G에 키만 저장함

단기기억에서 장기기억으로

보내는 것으로 비유해도 좋다.

```
19: function EVICTS
20:     evicted ← false
21:     while not evicted and S.size > 0 do
22:         t ← tail of S
23:         if t.freq > 1 then
24:             insert t to M
25:             if M is full then
26:                 evictM()
27:         else
28:             insert t to G
29:             evicted ← true
30:         remove t from S
```

S3-FIFO 큐 구현

EVICT_M

객체의 freq 값이 1 이상이면

큐의 헤드로 옮긴다.

마침내 0을 만나면 M에서 제거한다.

```
31: function EVICTM
32:     evicted ← false
33:     while not evicted and M.size > 0 do
34:         t ← tail of M
35:         if t.freq > 0 then
36:             Insert t to head of M
37:             t.freq ← t.freq-1
38:         else
39:             remove t from M
40:             evicted ← true
```

S3-FIFO 큐 구현

연산 비용을 생각해보자

메인 큐에서 캐시를 프로모팅 할 때 re-insertion 과정이 있다.

worst-case의 경우 O(kN)의 비용이 발생함 =〉 모든 객체가 캐시 히트가 되는 경우

다만 저자는 캐시 히트에서 얻는 이득이 더 크니 이정도 오버헤드는 무시할 만 하다고 표현한다.

캐시 히트 =〉 reduced mean latency

S3-FIFO 큐 구현

연산 비용을 생각해보자

G는 해시 기반 버킷 테이블로 구성하는데

오브젝트 크기가 4KB이고 키 값이 4 byte (integer) 라면 G는 전체에서 0.09%의 용량만 차지함

S3-FIFO 성능 평가

아래 세 가지 질문의 답을 찾을 것이다.

1. S3-FIFO의 효율성이 정말 SOTA 인가?
2. S3-FIFO가 정말로 확장성이 높은가?
3. S3-FIFO가 진짜로 flash cache design에 도움이 되는가?

S3-FIFO 성능 평가

평가 데이터 셋

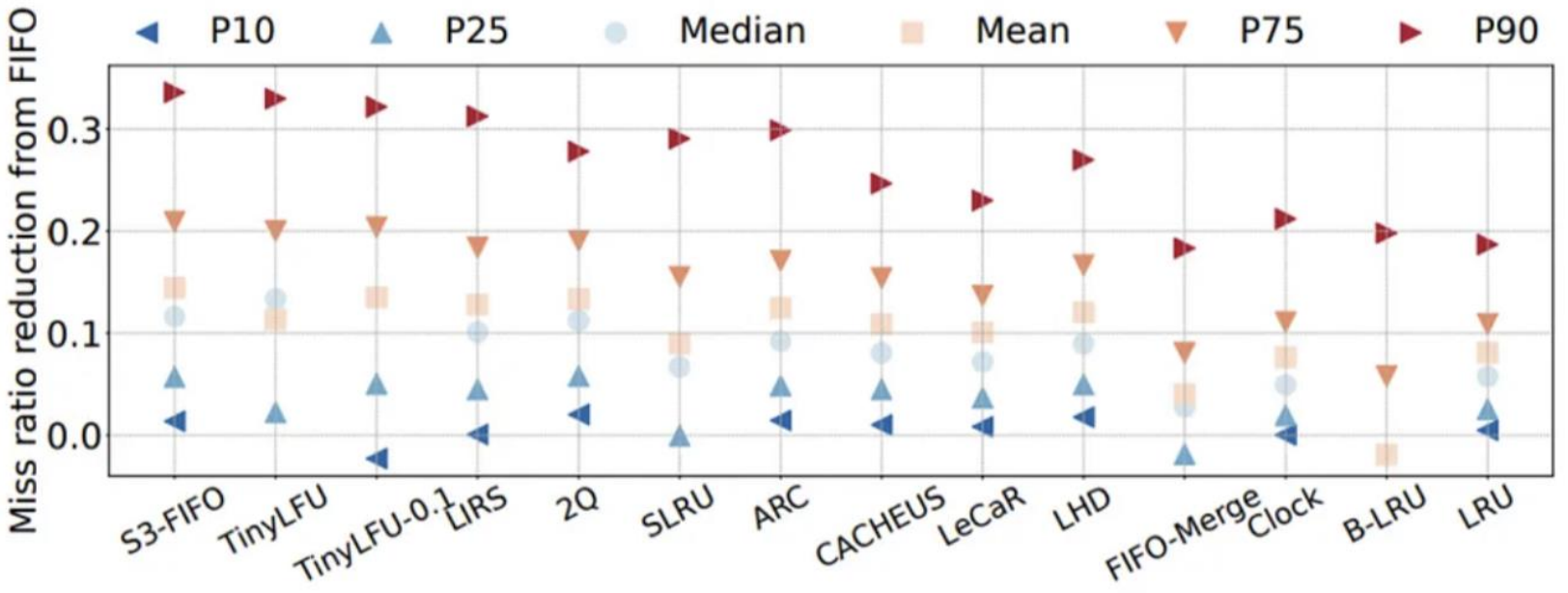
854조 개의 트레이스, 61조개의 오브젝트를 포함해 21,088TB 규모의 트래픽으로 실험 진행

Trace collections	Approx time	Cache type	time span (days)	# Traces	# Request (million)	Request (TB)	# Object (million)	Object (TB)	One-hit-wonder ratio full trace	10%	1%
MSR [104, 105]	2007	Block	7	13	410	10	74	3	0.56	0.74	0.86
FIU [83]	2008-11	Block	9-28	9	514	1.7	20	0.057	0.28	0.91	0.91
Cloudphysics [136]	2015	Block	7	106	2,114	82	492	22	0.40	0.71	0.80
CDN 1	2018	Object	7	219	3,728	3640	298	258	0.42	0.58	0.70
Tencent Photo [167, 168]	2018	Object	8	2	5,650	141	1,038	24	0.55	0.66	0.74
WikiMedia CDN [140]	2019	Object	7	3	2,863	200	56	13	0.46	0.60	0.80
Systor [84, 85]	2017	Block	26	6	3,694	88	421	15	0.37	0.80	0.94
Tencent CBS [163, 164]	2020	Block	8	4030	33,690	1091	551	66	0.25	0.73	0.77
Alibaba [2, 89, 139]	2020	Block	30	652	19,676	664	1702	117	0.36	0.68	0.81
Twitter [157]	2020	KV	7	54	195,441	106	10,650	6	0.19	0.32	0.42
Social Network 1	2020	KV	7	219	549,784	392	42,898	9	0.17	0.28	0.37
CDN 2	2021	Object	7	1273	37,460	4,925	2,652	1,581	0.49	0.58	0.64
Meta KV [11]	2022	KV	1	5	1,644	958	82	76	0.51	0.53	0.61
Meta CDN [11]	2023	Object	7	3	231	8,800	76	1,563	0.61	0.76	0.81

S3-FIFO 성능 평가

효율성 평가

Y축 = FIFO큐를 기준으로 얼마나 Miss Ratio을 줄였는가? 즉 값이 클 수록 좋음

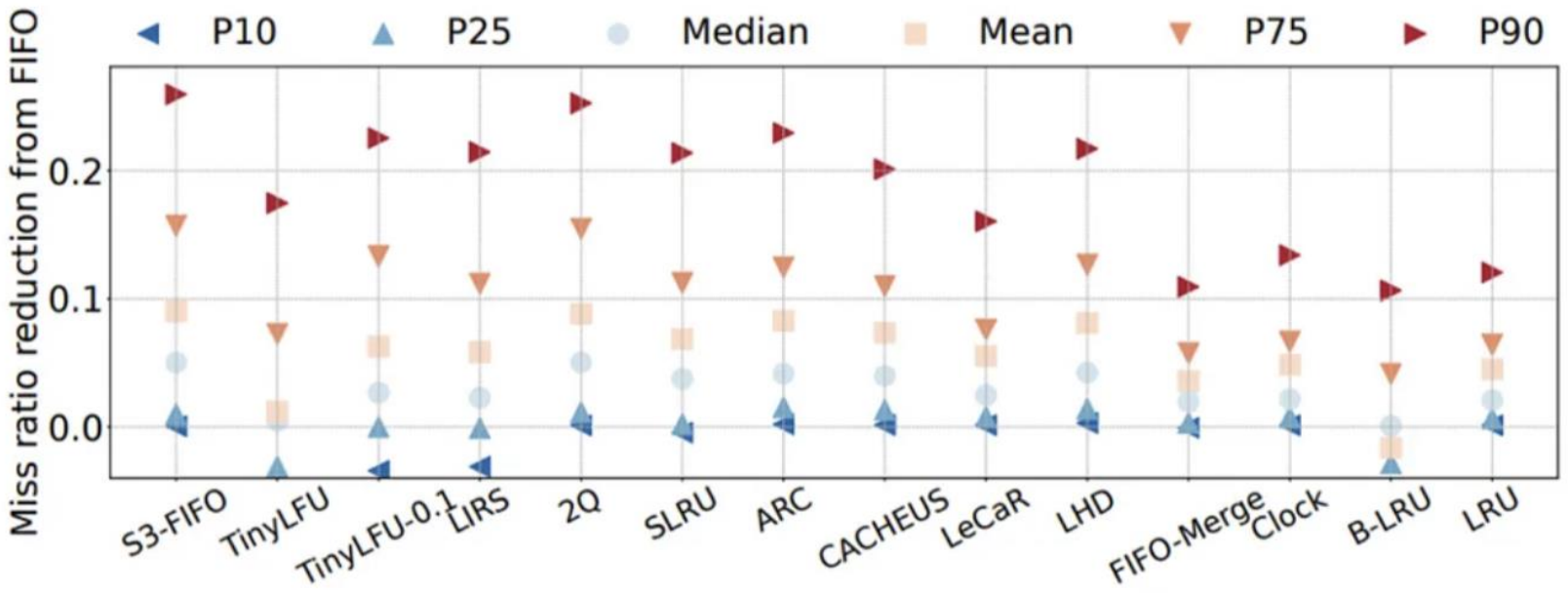


(a) Large cache size, 10% trace footprint

S3-FIFO 성능 평가

효율성 평가

TinyLFU와 효율성 면에서 견줄만 한데, TinyLFU는 캐시 크기가 작으면 불안정하다고 보고있음



(b) Small cache size, 0.1% trace footprint

S3-FIFO 성능 평가

논문에서는 각 캐시 종류마다

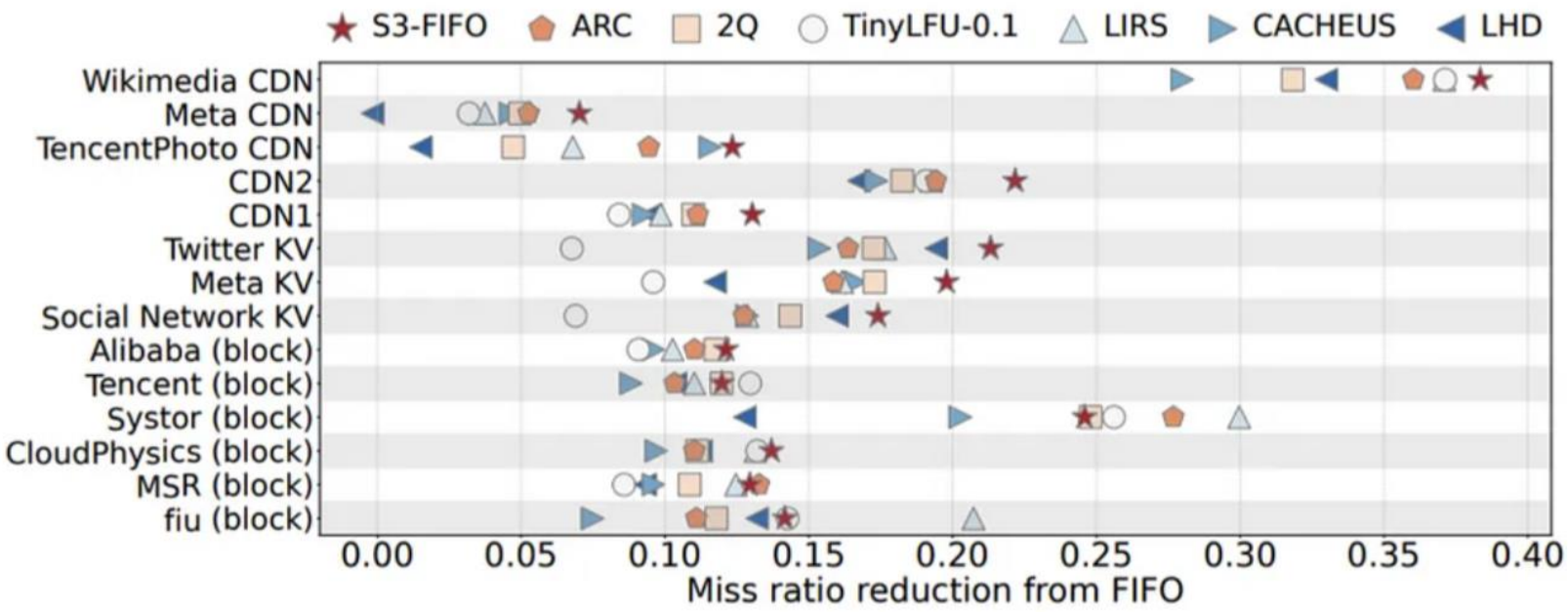
결과가 S3-FIFO 큐보다 떨어진 이유를 저자의 관점에서 하나씩 나열하고 있다.

- 이 발표에서는 생략 -

S3-FIFO 성능 평가

각 데이터 셋 마다 S3-FIFO 큐의 효율성을 나타낸 표이다.

13개 중 10개 데이터 셋에 대해 SOTA의 성능을 보여준다



(a) Large cache size

S3-FIFO 성능 평가

그렇다면 왜 효과적인가?

- Normalized quick demotion speed
  - 객체가 얼마나 빨리 small queue S 에서 제거되었는가
- Quick demotion precision
  - 제거된 오브젝트가 다시 캐시에 오기 까지 걸리는 시간
  - 캐시 크기/캐시 미스율 보다 크면 “아 빨리 지우길 잘했다” 라고 판단한다

S3-FIFO 성능 평가

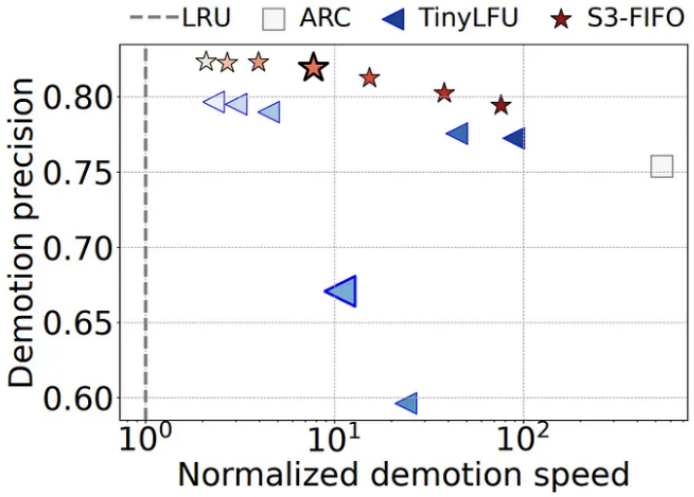
그렇다면 왜 효과적인가?

small queue S의 크기에 따른 분포를 나타낸다

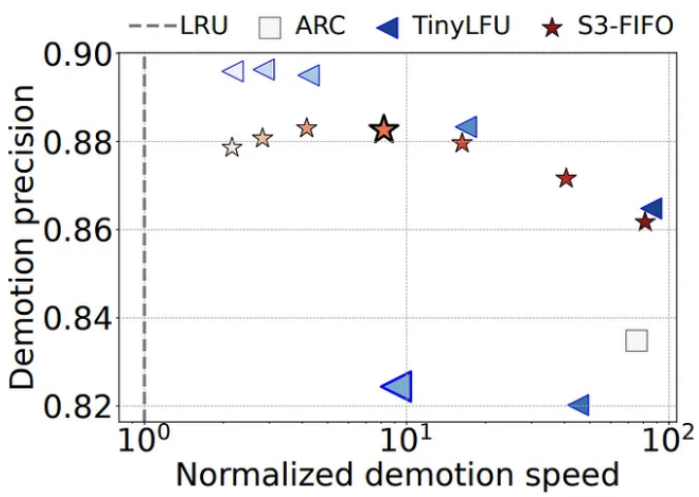
TinyLFU는 다소 불안정한 모습을 보여줌

경향성만 보면, 객체를 빠르게 지울수록

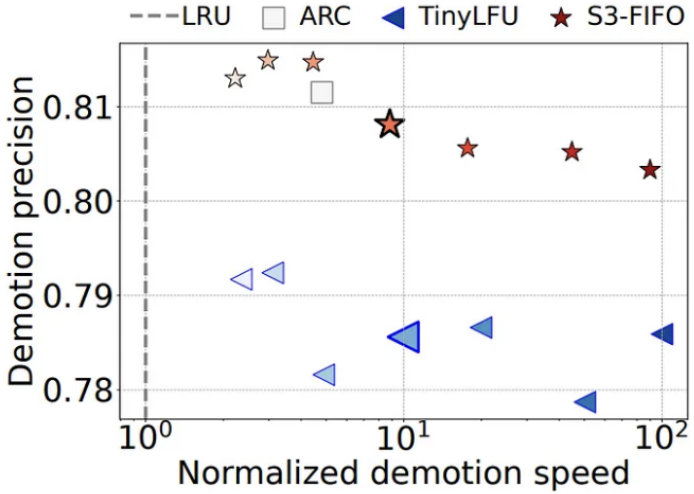
Demotion precision이 증가함



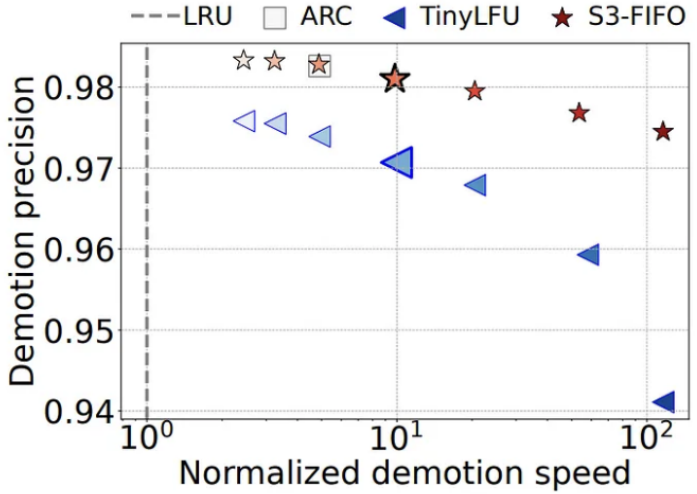
(a) Twitter trace, large cache



(b) Twitter trace, small cache



(c) MSR trace, large cache



(d) MSR trace, small cache

S3-FIFO 성능 평가

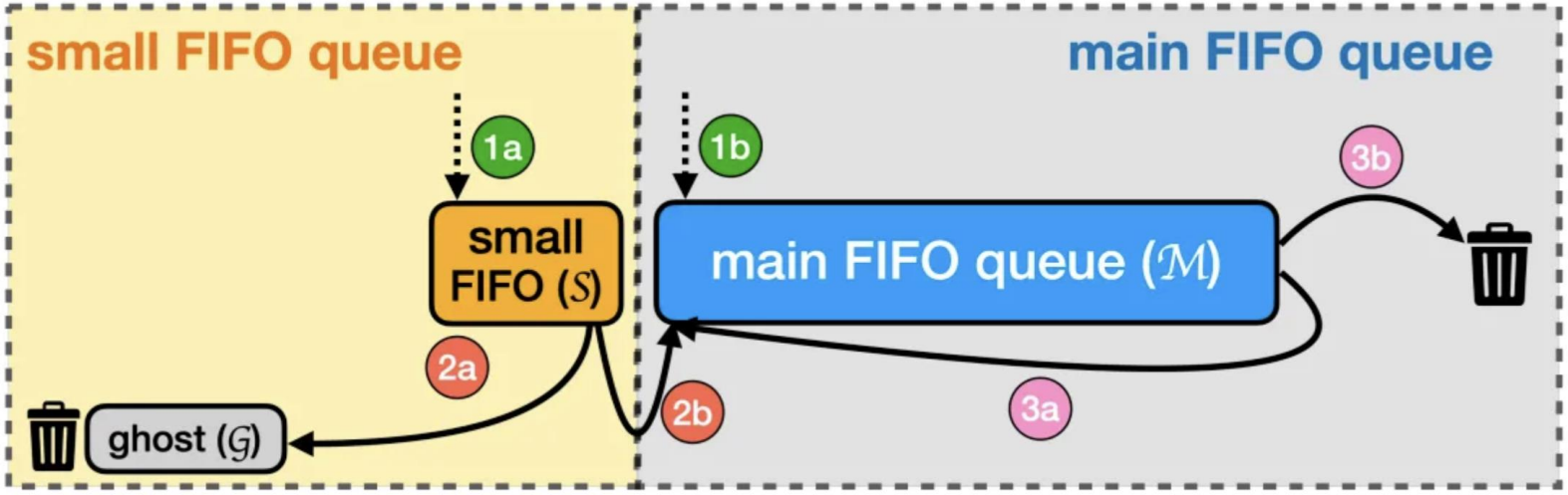
동일한 구조로 LRU 3개 붙여놓고 하면 안되는건가?

논문 저자는 논문에 실을 가치도 없이 제대로 동작하지 않았다고 함

S3-FIFO 성능 평가

플래시 친화적인가? 이런 상상을 해볼 수 있다. 단기기억에 해당하는 S는 DRAM에 배치하고 메인 큐 M은 SSD에 배치한다. 대 부분의 객체가 demotion될 때 S에서 발생하니

SSD를 최대한 '덜' 쓰도록 설계 해본다.



Insert: if not in ghost, insert to small 1a, else insert to main 1b

Evict (small): if not visited, insert to ghost 2a, else insert to main 2b

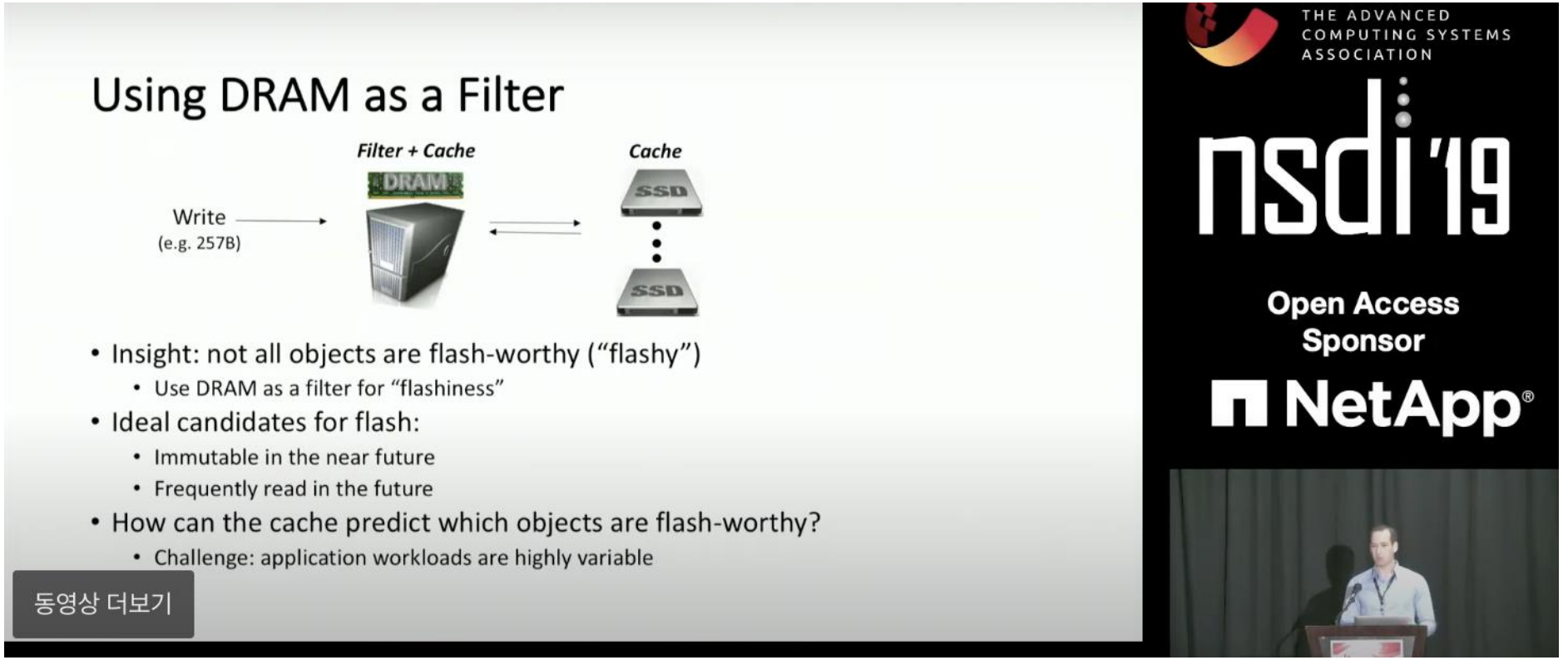
Evict (main): if visited, insert back 3a, else evict 3b

S3-FIFO 성능 평가

비교 대상은 스탠포드 대학 연구진이 발표한 Flashield 이다.

달성하고자 하는 목표는 동일하다. DRAM을 총알 받이로 쓰자는 것

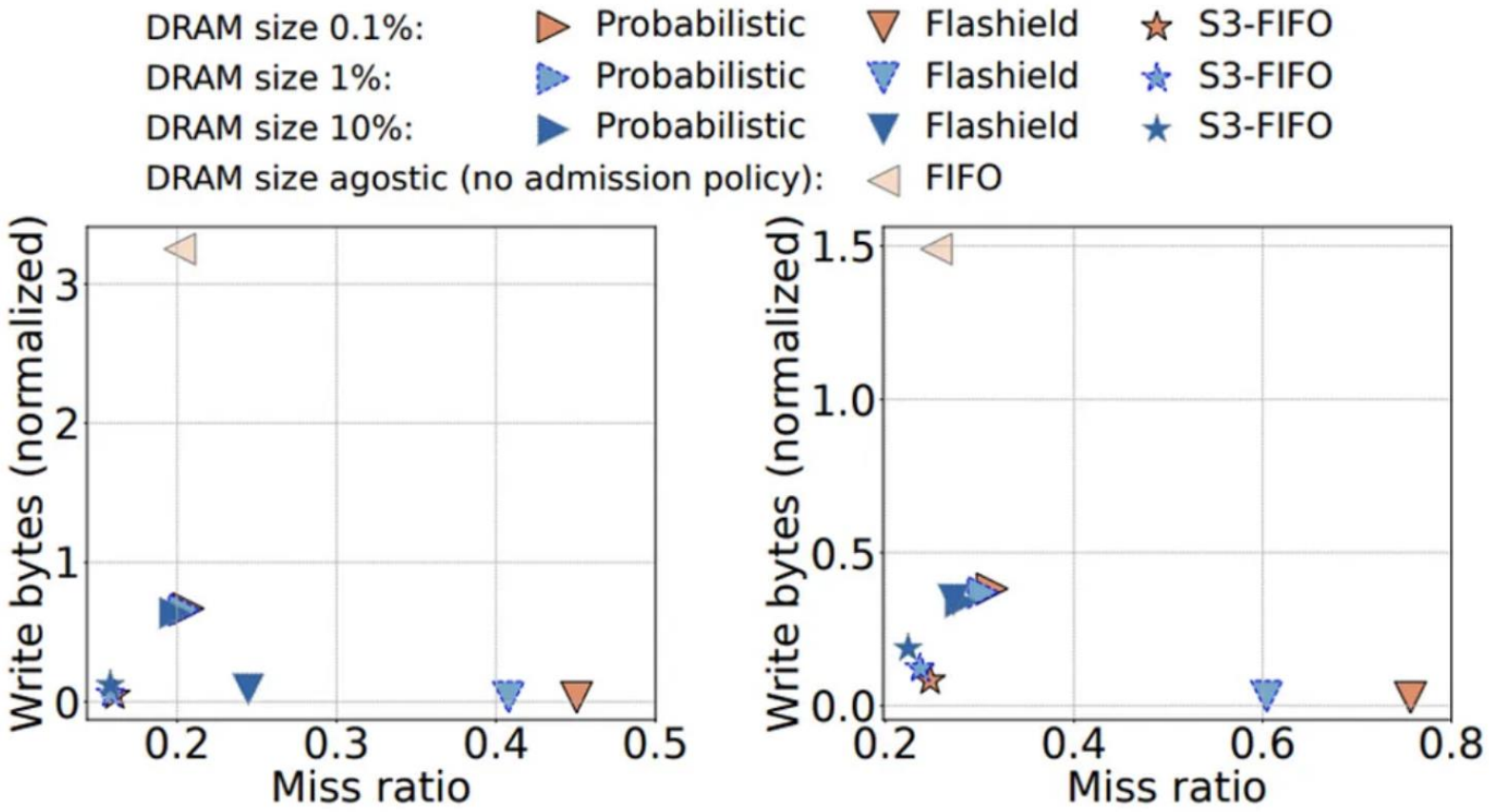
다만 이 연구는 어떤 객체를 SSD로 밀어넣을 지 결정하는 걸 머신러닝에 맡겼다.



S3-FIFO 성능 평가

플래시 친화성 평가 | 원점에 가까울수록 좋은 캐시이다.

CDN 업체들은 캐시를 평가할 때 write bytes를 얼마나 아꼈는가를 평가하기도 한다.

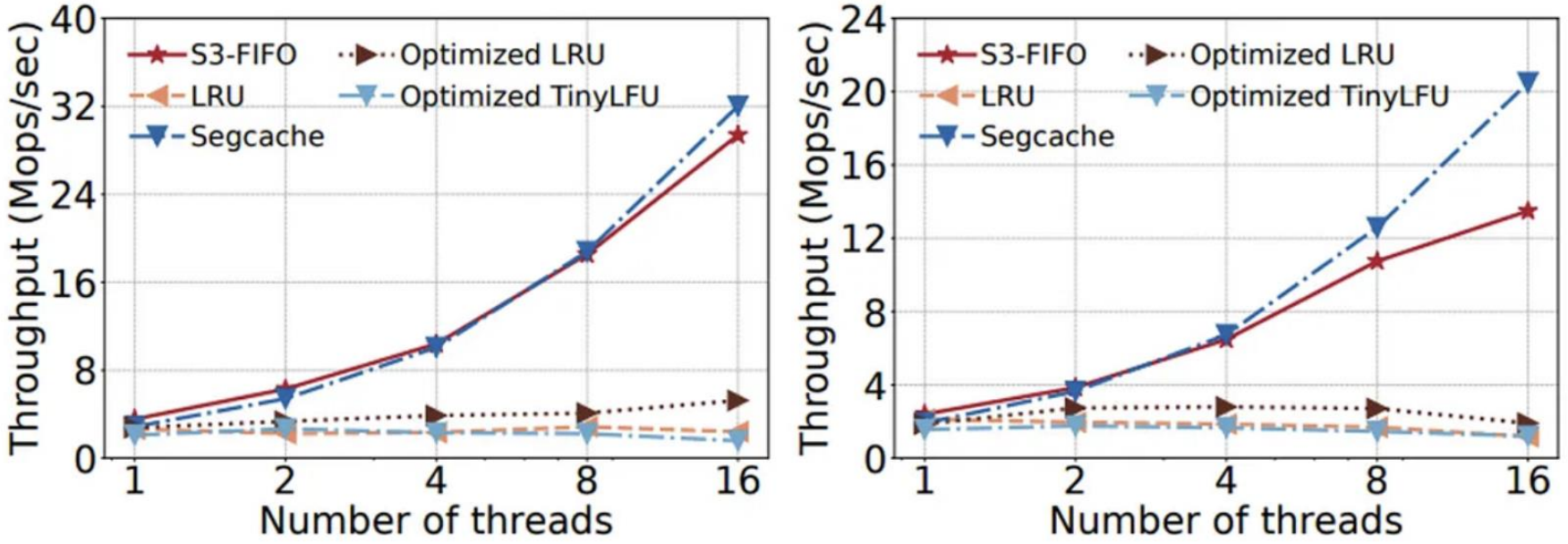


S3-FIFO 성능 평가

성능평가

LRU는 읽기/쓰기 모두 락을 잡기 때문에 당연히 확장성이 없다.

S3-FIFO는 lock-free로 구현할 수 있어서 확장성이 높다고는 하는데.. 논쟁의 여지가 있다



(a) Large cache, LRU miss ratio 0.02 **(b)** Small cache, LRU miss ratio 0.21

S3-FIFO 성능 평가

Lock-free queue 예시

loop를 돌면서 CAS연산이 통과할 때 까지 무한 시도한다.

Optimistic Concurrency와 동일, 일단 투기의 마음으로 부딪쳐본다. Spin lock 이랑은 다르다.

```
ENQUEUE(x)
  q ← new record
  q^.value ← x
  q^.next ← NULL
  repeat
      p ← tail
      succ ← COMPARE&SWAP(p^.next, NULL, q)
      if succ ≠ TRUE
          COMPARE&SWAP(tail, p, p^.next)
  until succ = TRUE
  COMPARE&SWAP(tail, p, q)
end
```

SIEVE

SIEVE 설계 및 구현

SIEVE는 2024년 NSDI에서 소개될 논문으로 S3-FIFO의 1저자가 교신저자로 참여한 논문이다.

여기서는 한술 더 떠서 큐 하나만 써도 되던데? 라고 발표한다.

SIEVE 설계 및 구현

SIEVE가 얼마나 심플한지 보여주기 위해, 각 알고리즘을 구현하는데 필요한 코드 라인을 보여준다

Algorithm	cache hit	eviction	insertion	metadata size
FIFO	1	4	3	16B
LRU	5	4	3	16B
ARC	64	108	20	17B
LIRS	96	120	64	17B
LHD	192	81	64	13B
LeCaR	72	76	20	40B
CACHEUS	168	140	150	54B
TwoQ	28	16	8	17B
Hyberbolic	4	20	4	16B
CLOCK	4	9	3	17B
SIEVE	4	9	3	17B

SIEVE 설계 및 구현

오픈소스에 구현된 캐시에 SIEVE도입을 위해 필요한 코드 라인 수

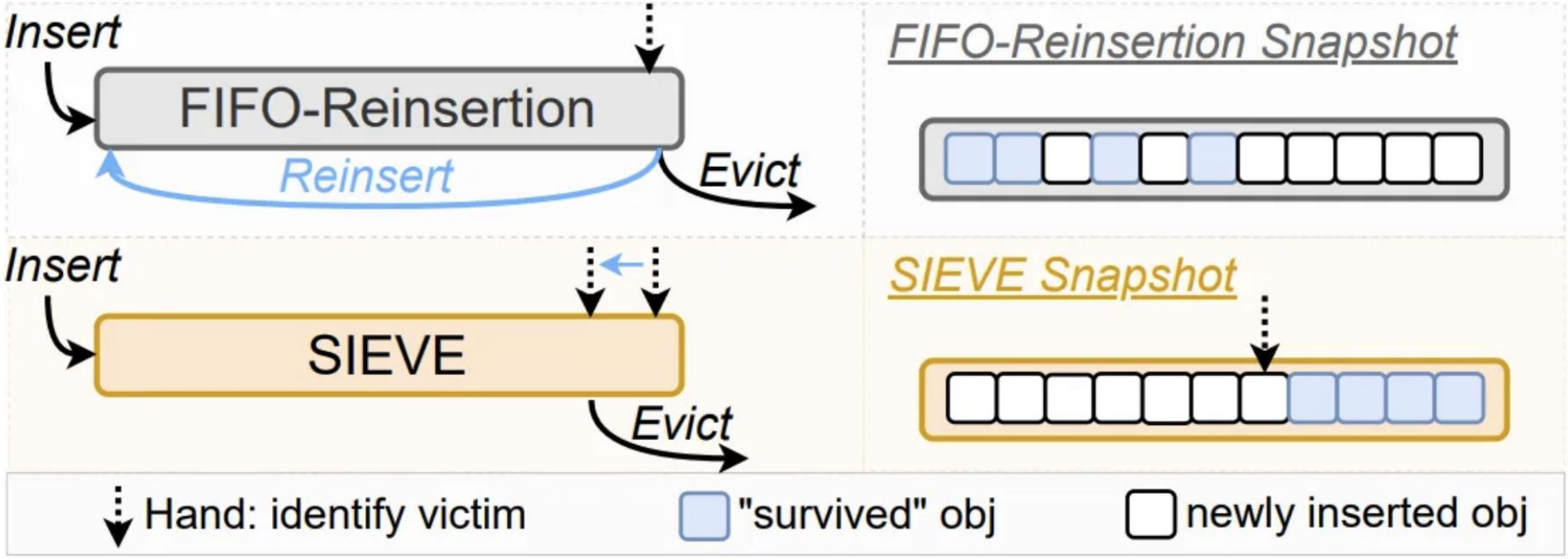
20줄만 고치면 아주 좋은 캐시를 사용할 수 있다?

Cache library	Language	Lines
groupcache [25]	Golang	21
mnemonist [6]	Javascript	12
lru-rs [4]	Rust	16
lru-dict [3]	Python + C	21

SIEVE 설계 및 구현

구조 자체는 간단하다. 하나의 큐와 hand 라는 포인터로 이루어져 있다.

SIEVE는 $CLOCK_{1969년}$ 의 확장판이기 때문에 이 둘을 같이 비교한다.



SIEVE 설계 및 구현

전체 구현 의사 코드

CLOCK은 vis가 1인 경우

큐의 헤드로 프로모팅하는 반면

SIEVE는 그냥 무시하고 hand가

이동하는 것이 특징

Algorithm 1 SIEVE

Input: The request x, doubly-linked queue T, cache size C, hand p

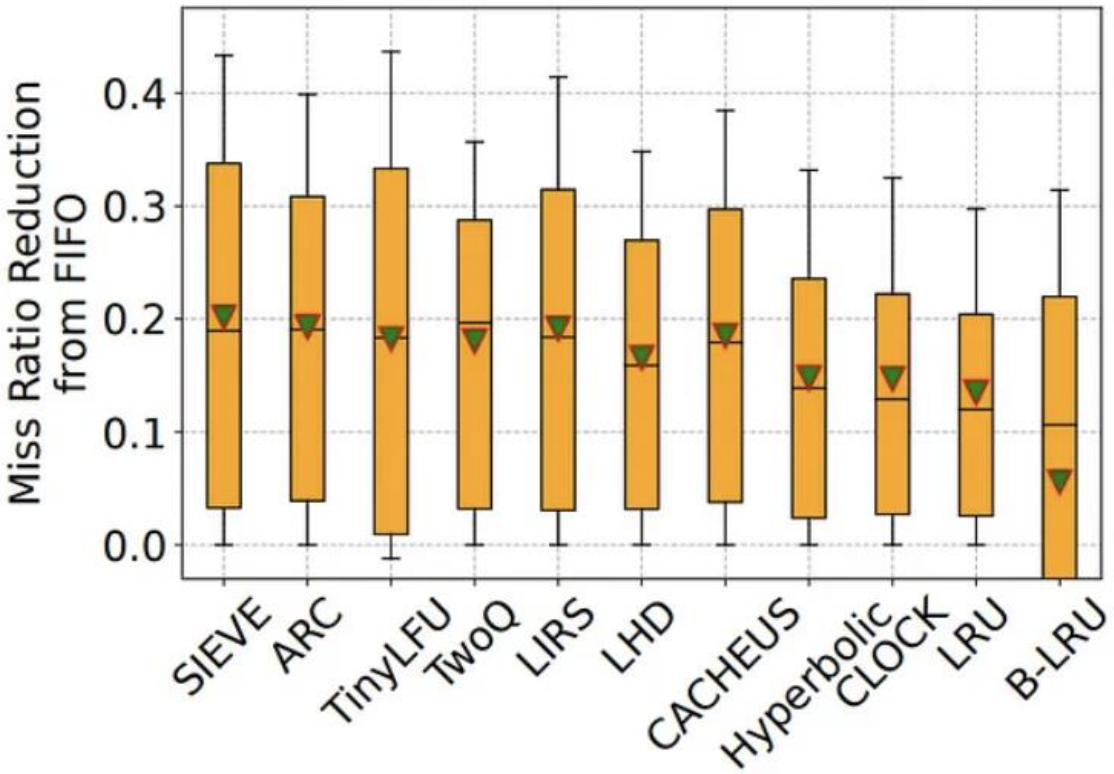
```
1:  if x is in T then                                   ▷ Cache Hit
2:      x.visited ← 1
3:  else                                                ▷ Cache Miss
4:      if |T| = C then                                 ▷ Cache Full
5:          o ← p
6:          if o is NULL then
7:              o ← tail of T
8:          while o.visited = 1 do
9:              o.visited ← 0
10:             o ← o.prev
11:             if o is NULL then
12:                 o ← tail of T
13:         p ← o.prev
14:         Discard o in T                              ▷ Eviction
15:     Insert x in the head of T.
16:     x.visited ← 0                                   ▷ Insertion
```

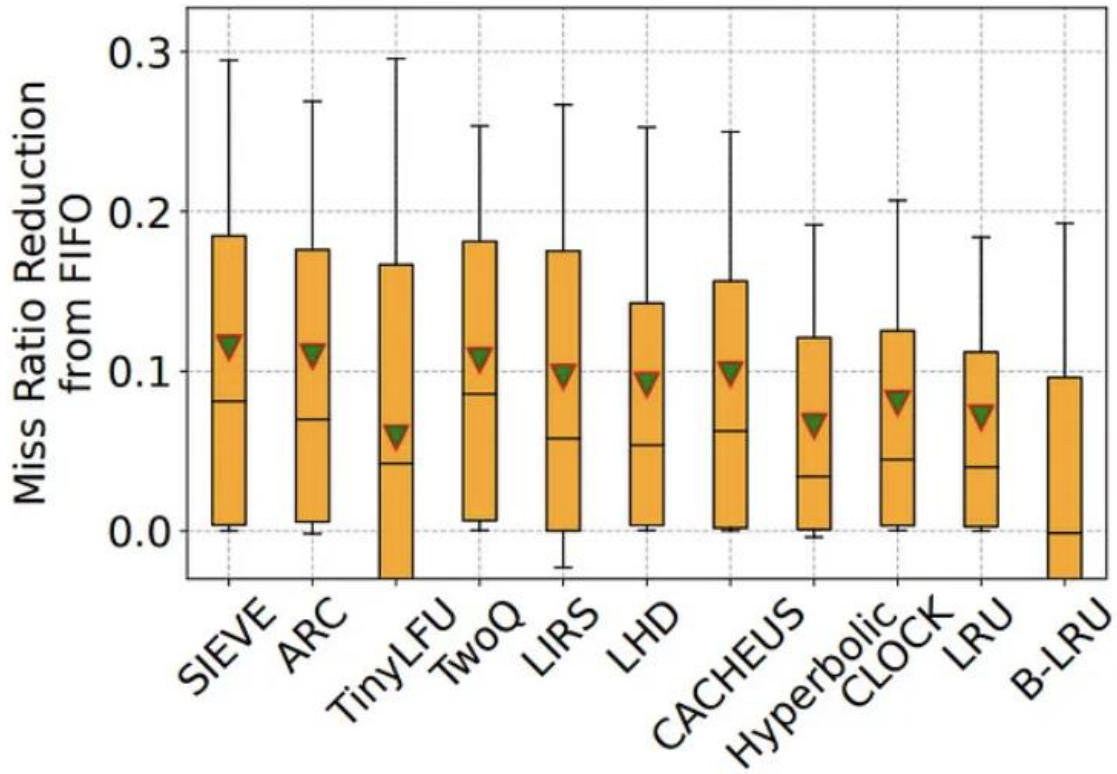
SIEVE 설계 및 구현

ARC보다 평균 1.5%/최대 60% 이상 효율성 개선이 있었다

CLOCK과 비교하여 성능이 엄청 높다는 것을 주목할 필요가 있다

논문에는 더 많은 평가 표가 있는데 지면이 부족해서 2개만 가져왔다



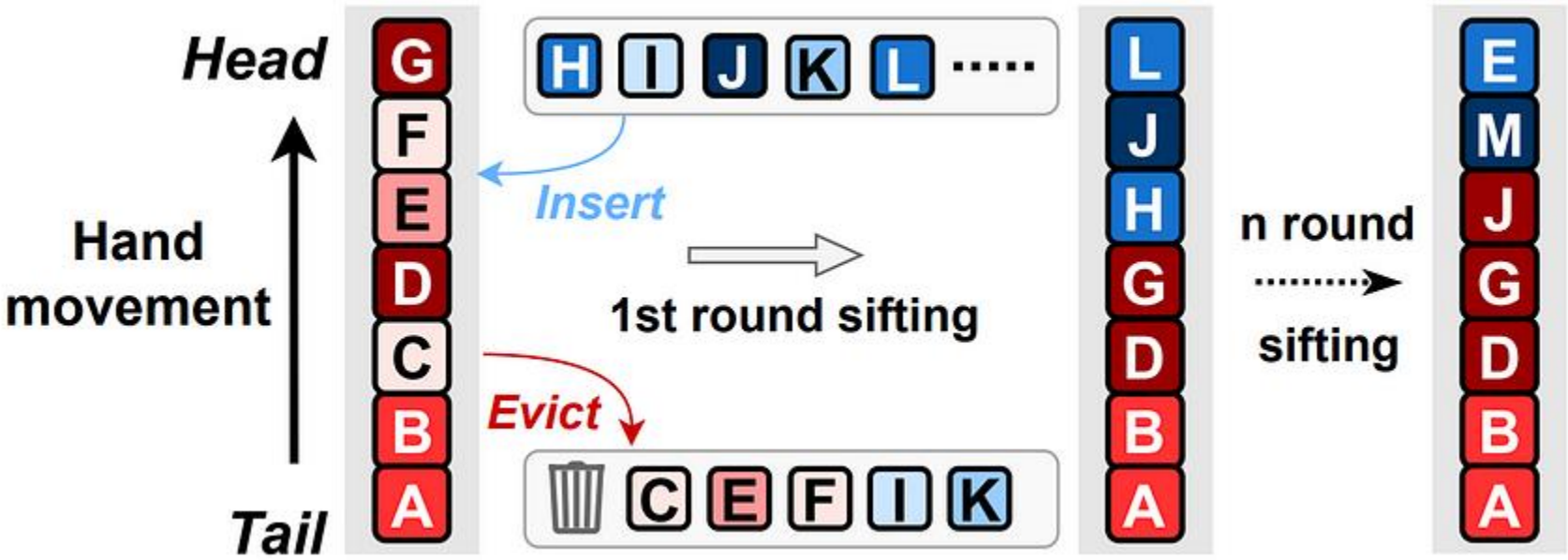


SIEVE 설계 및 구현

그렇다면 왜, SIVE가 CLOCK보다 성능이 좋을까??

앞서 Zipf's workload의 원-히트-원더 비율을 설명하면서 Quick Demotion의 필요성을 설명했다.

SIEVE는 hand 포인터를 이동시킴으로써 큐의 앞 부분에서 신규 객체들을 빠르게 제거하는 역할을 한다.

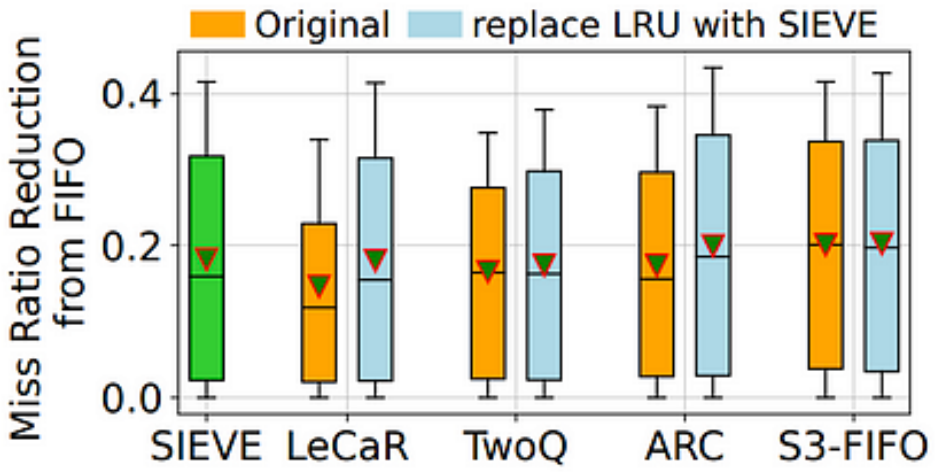


SIEVE 설계 및 구현

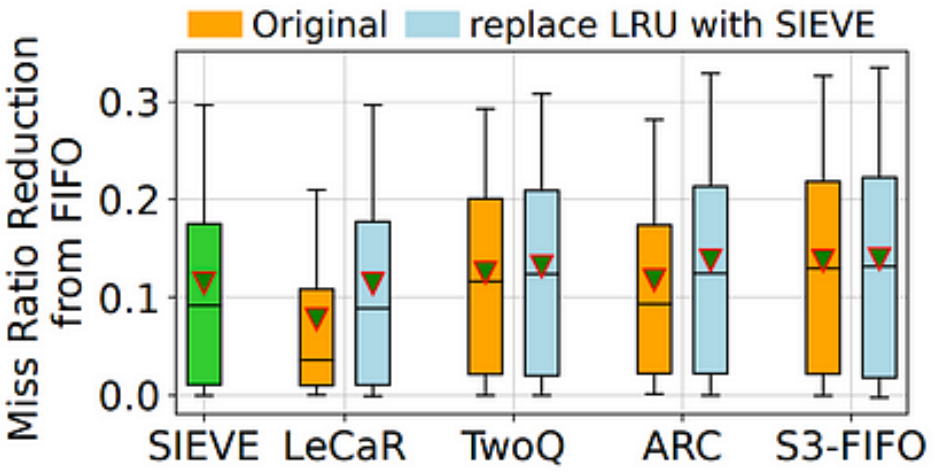
Cache Primitive 로서의 가능성

논문에서 하나 더 강조하는 점이 SIEVE가 너무 간단하기 때문에 다른 캐시와 결합해서 사용할 수 있다는 점이다.

예를 들어, ARC에서 LRU를 SIEVE로 교체하면 최대 62.5%, 평균 3.7%의 개선이 있었다고 한다



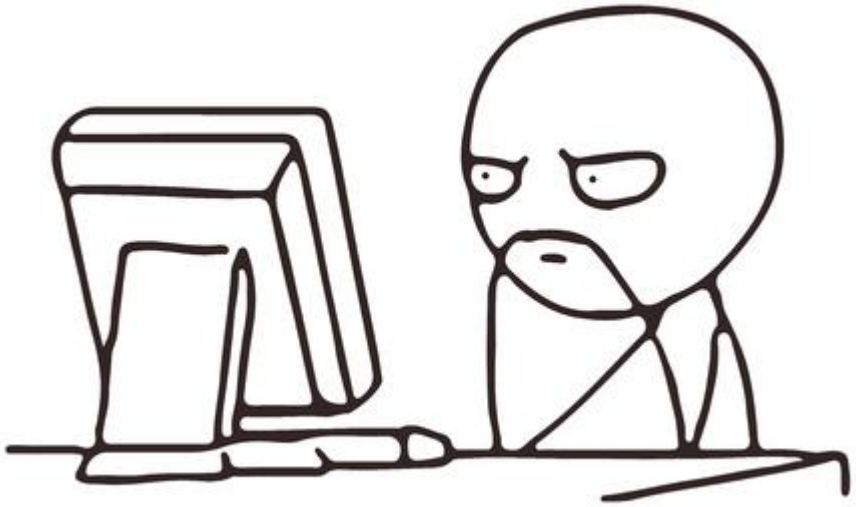
(a) Large cache



(b) Small cache

SIEVE 설계 및 구현

아니 근데 1969년에 “A Paging Experiment with the Multics System” 논문에서 CLOCK이 소개되고 지금 시간이 얼마나 지났는데.. 그렇게 어려운 거 같지도 않은데 정말 겹치는 연구가 없었다고?



SIEVE 설계 및 구현

FIFO 캐시는 스캔 저항성이 없기 때문에 그 동안의 연구에서 주목받지 못한 것 같다고 회고한다

We conjecture that not being scan-resistant is probably the reason

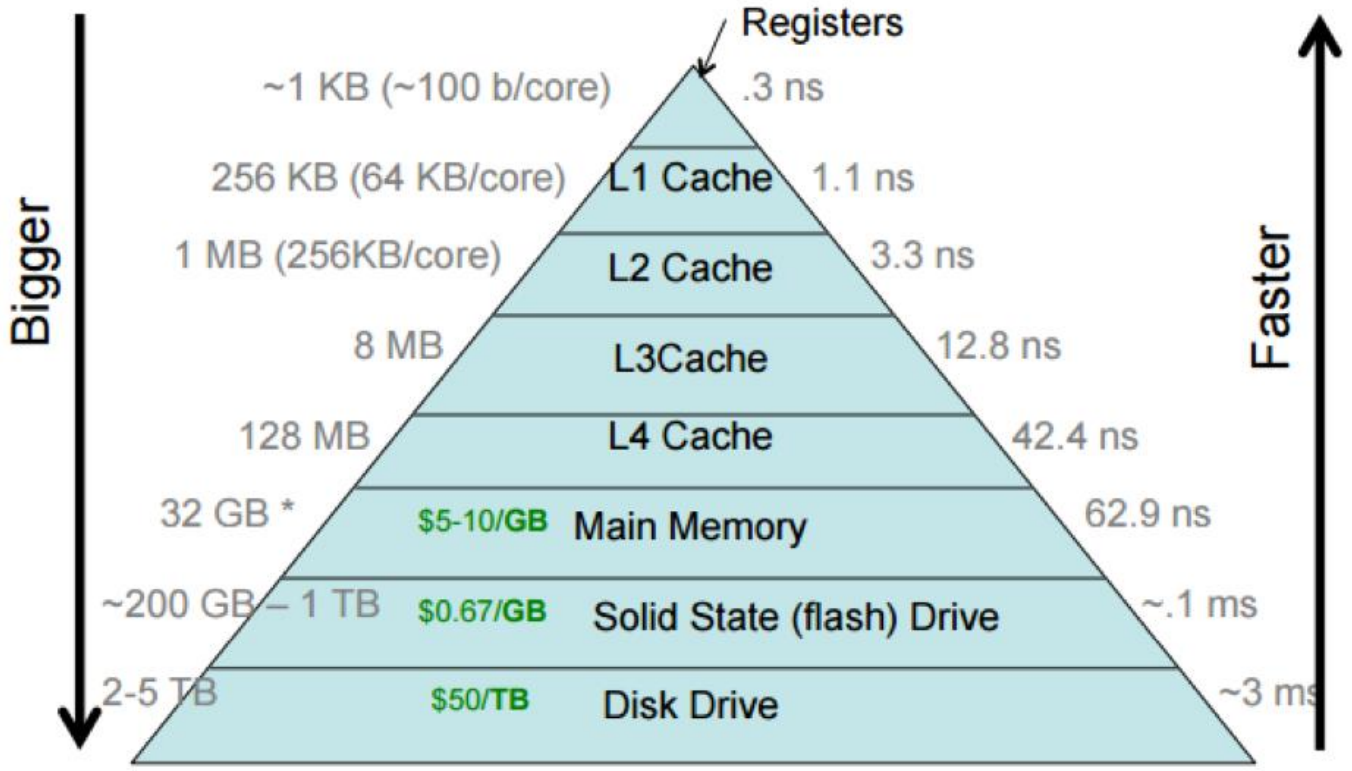
why SIEVE remained undiscovered over the decades of caching research

SIEVE 설계 및 구현

스캔 저항성

캐시는 하위 계층 저장소의 낮은 레이턴시을 개선하기 위해 핫 데이터를 상위 저장소로 올리는 작업이다.

거의 모든 계층에 캐시가 존재하는데 L1, L2, LLC 등 블록 I/O 같은 경우에는 범위 탐색 쿼리가 빈번하다.

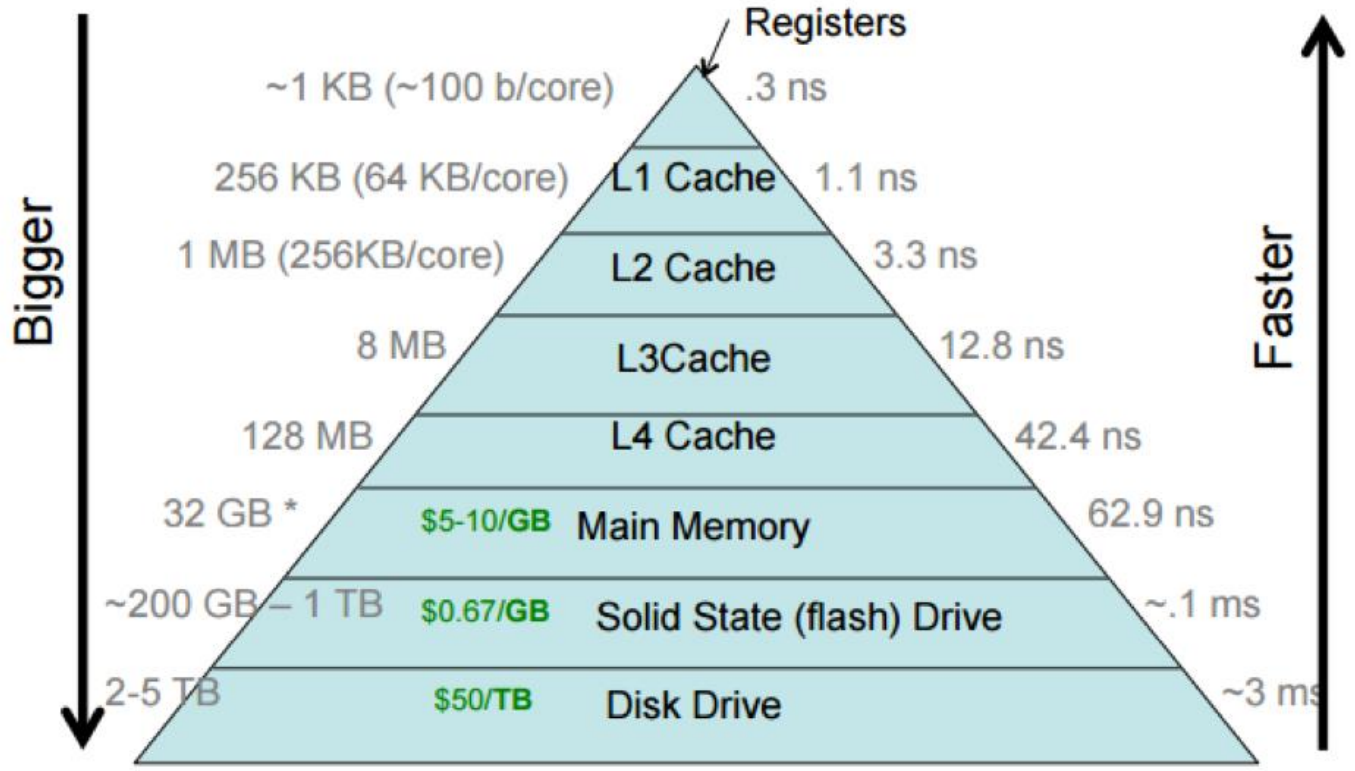


SIEVE 설계 및 구현

스캔 저항성

키/밸류로 조회하는 웹 서버와는 달리 범위로 조회하면 다량의 최신 키가 갑자기 밀려 들어오니까

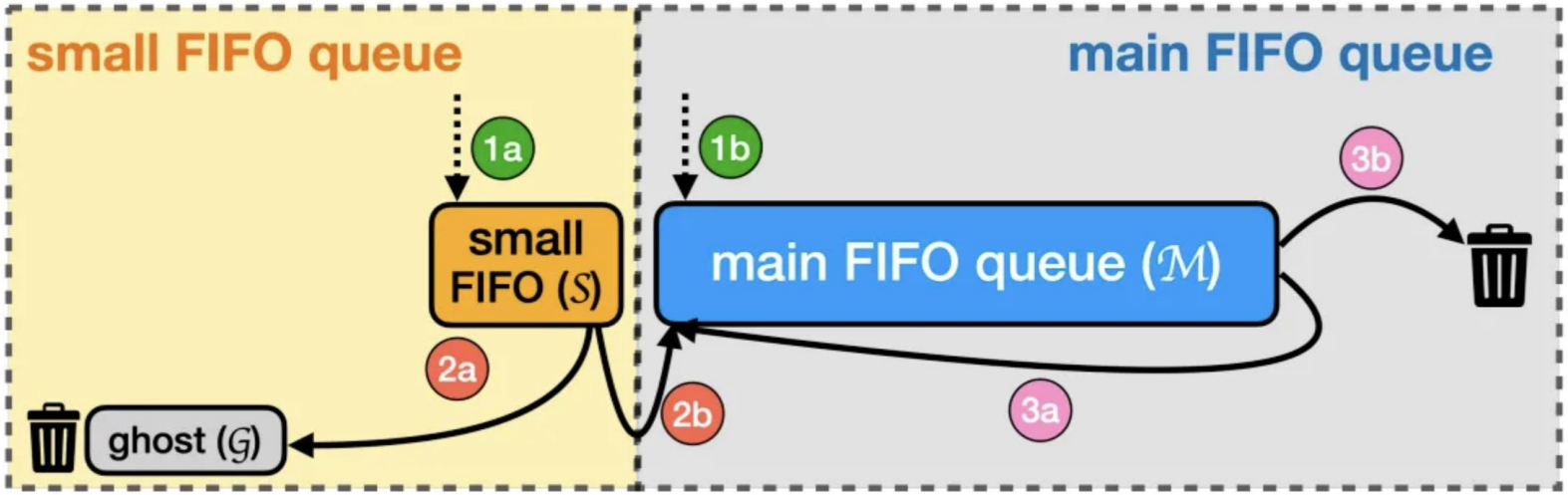
핫 데이터 평가할 새도 없이 캐시를 Eviction 하는 상황이 발생할 수 있다.



SIEVE 설계 및 구현

스캔 저항성

S3-FIFO는 small queue S가 이를 막아주는 버퍼 역할을 하기 때문에 스캔 저항성이 있다.



Insert: if not in ghost, insert to small 1a, else insert to main 1b

Evict (small): if not visited, insert to ghost 2a, else insert to main 2b

Evict (main): if visited, insert back 3a, else evict 3b

거의 끝

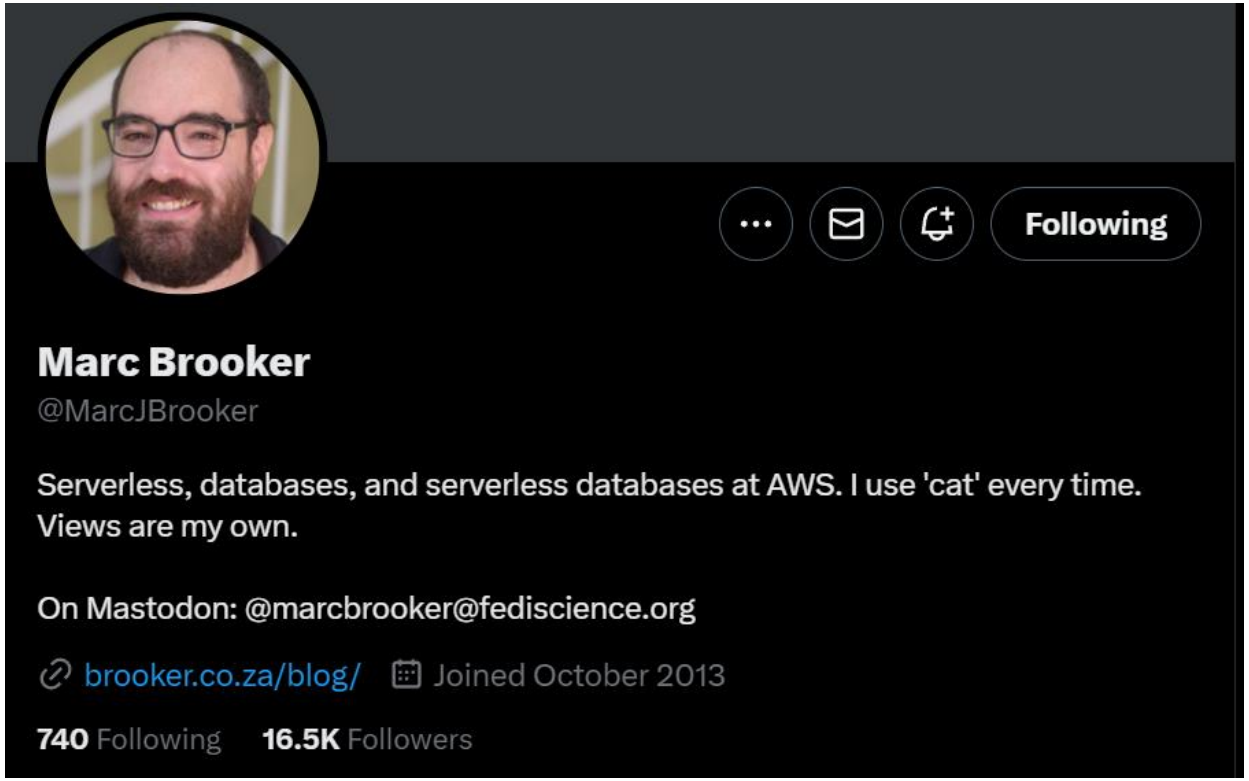
APPENDIX 1. AWS 엔지니어의 평가

AWS 엔지니어 Marc Brooker는 자신의 블로그에서 “SIEVE의 스캔저항성이 문제야?”

SIEVE는 빈도 값을 2이상 설정하는 SIEVE-k 를 제안함

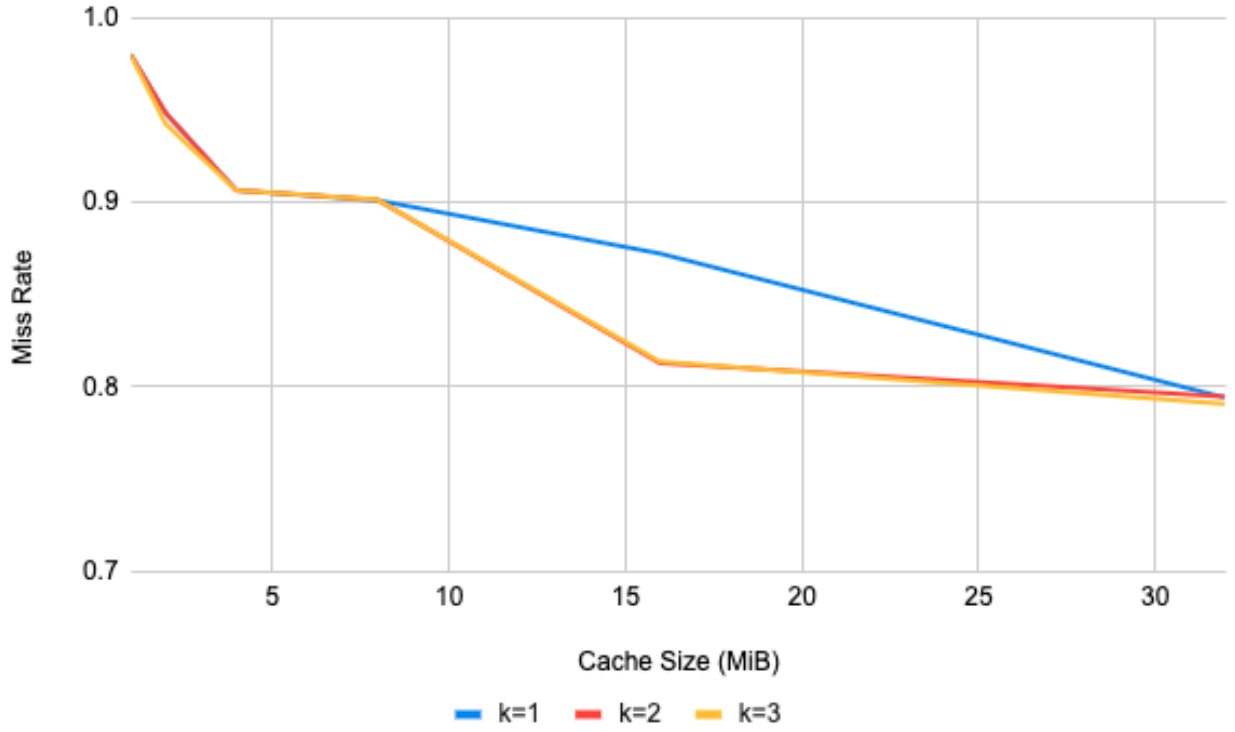
Why Aren’t We SIEVE-ing?

Captain, we are being scanned!



APPENDIX 1. AWS 엔지니어의 평가

캐시 사이즈가 16MiB 에서는 SIEVE-2 / SIEVE-3 이 블록 I/O 워크로드에서 개선이 있음을 밝힘



APPENDIX 2. 발표자의 평가

그리고, 오늘의 발표자는 Go언어로 구현해보고 각 오픈소스 캐시 라이브러리와 성능을 비교했습니다.

오늘 발표 마음에 드셨다면 별표 하나만..

APPENDIX 2. 발표자의 평가

평가는 Zipf’s distributio을 따라 데이터를 생성시키고 16개의 고루틴에서 성능을 측정함

전체 오브젝트 개수는 50만 / 요청은 750만 번 발생시킴

```
itemSize=500000, workloads=7500000, cacheSize=0.10%, zipf's alpha=0.99, concurrency=16

      CACHE      | HITRATE | MEMORY  |   QPS   |  HITS   | MISSES
-----------------+---------+---------+---------+---------+----------
  sieve          | 47.66%  | 0.09MiB | 2508361 | 3574212 | 3925788
  tinylfu        | 47.37%  | 0.11MiB | 2269542 | 3552921 | 3947079
  s3-fifo        | 47.17%  | 0.18MiB | 1651619 | 3538121 | 3961879
  slru           | 46.49%  | 0.11MiB | 2201350 | 3486476 | 4013524
  s4lru          | 46.09%  | 0.12MiB | 2484266 | 3456682 | 4043318
  two-queue      | 45.49%  | 0.17MiB | 1713502 | 3411800 | 4088200
  clock          | 37.34%  | 0.10MiB | 2370417 | 2800750 | 4699250
  lru-groupcache | 36.59%  | 0.11MiB | 2206841 | 2743894 | 4756106
  lru-hashicorp  | 36.57%  | 0.08MiB | 2055358 | 2743000 | 4757000
```

APPENDIX 2. 발표자의 평가

SIEVE가 히트 레이트가 가장 높을 뿐 아니라 QPS도 높게 측정됨

LRU는 캐시 히트 될때 객체를 프로모팅 해야 해서 lock을 잡아야 하는데

SIEVE는 캐시 히트 될 때 RWMutex로 reader lock만 잡아도 되기 때문에 성능이 더 높게 나올 수 밖에 없다.

```
itemSize=500000, workloads=7500000, cacheSize=0.10%, zipf's alpha=0.99, concurrency=16

      CACHE       | HITRATE | MEMORY   |   QPS    |  HITS    | MISSES
------------------+---------+----------+----------+----------+----------
  sieve           | 47.66%  | 0.09MiB | 2508361 | 3574212 | 3925788
  tinylfu         | 47.37%  | 0.11MiB | 2269542 | 3552921 | 3947079
  s3-fifo         | 47.17%  | 0.18MiB | 1651619 | 3538121 | 3961879
  slru            | 46.49%  | 0.11MiB | 2201350 | 3486476 | 4013524
  s4lru           | 46.09%  | 0.12MiB | 2484266 | 3456682 | 4043318
  two-queue       | 45.49%  | 0.17MiB | 1713502 | 3411800 | 4088200
  clock           | 37.34%  | 0.10MiB | 2370417 | 2800750 | 4699250
  lru-groupcache  | 36.59%  | 0.11MiB | 2206841 | 2743894 | 4756106
  lru-hashicorp   | 36.57%  | 0.08MiB | 2055358 | 2743000 | 4757000
```

APPENDIX 2. 발표자의 평가

논문 저자가 찾아와 댓글도 달아줬다..

Feedback on a statement #6

Closed **1a1a11a** opened this issue last week · 6 comments

1a1a11a commented last week • edited

This is awesome work!

One suggestion on the result. It mentions that "SIEVE is about 10% more efficient than a simple LRU cache", which I think is not very accurate.
When comparing the miss ratio, we usually state relative numbers. For example, reducing the mass ratio from 2% to 1% is a (2-1)/2=50% improvement because the backend load is reduced by 50%. Moreover, the mean latency is about proportional to the mass ratio and has a similar reduction. :)

BTW, do you have results for S3-FIFO as well?

scalalang2 commented last week • edited Owner

Thank you visiting here 👍

I understand what you are saying. Increasing efficiency means not only reducing cache misses, but also reducing the demand for heavy operations such as backend database access, which lowers the mean latency.

And also as you said, It seems that I expressed incorrectly the evaluation of miss ratio. it would be correct to express that the SIEVE has improved efficiency by roughly 30% in comparison with simple LRU cache.

👍 1

APPENDIX 2. 발표자의 평가

공식 홈페이지에 Go언어 구현체로 소개됨

s3fifo.com

Adoption

Google, VMware, Redpanda,

mulitple open-source libraries, including surrealkv, otter, spica,

and several other libraries, such as Golang, Python, Rust 1, Rust 2, C++ 1, C++ 2.

Discussed in Aleksey's Online Reading Group.

Covered in blog [1], [2], [3] in Korean, [4] in Japanese, newsletters [1], [2]