서강대학교
SOGANG UNIVERSITY

Tiered Memory 환경에서의 디바이스별 Cache Replacement RL Policy 비교 분석

https://github.com/ctaaone/rlprj_tieredmem

120250337 이승재

Index

프로젝트 배경

Memory-intensive Applications

벡터 검색 엔진 및 벡터 DB(ex. FAISS, Milvus DB)과 in-memory KV 캐시(ex. Redis, Memcached) 같은 서비스는 대용량 데이터를 DRAM에 상주시켜 낮은 지연으로 처리한다.

그러나 로컬 DRAM은 수십-수백 GB 수준에서 **용량/비용 확장에 한계**가 있어, 벡터 검색 엔진 및 벡터 DB는 워크로드가 이를 초과하면 데이터는 **IO device**에 저장하고 **필요 시 DRAM으로 캐싱**하는 방식으로 운영된다.

또한 in-memory KV 캐시는 이러한 DRAM 확장 한계로 인해, 저장 가능한 데이터 용량 측면에 제한을 받는다.

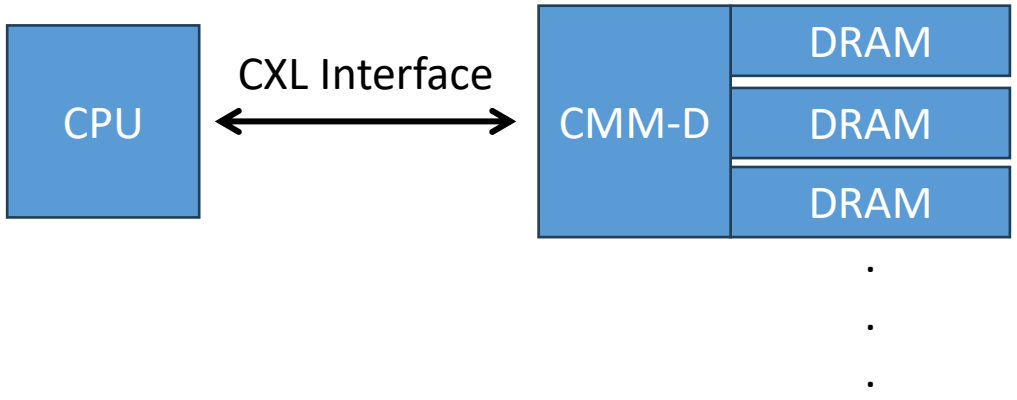
프로젝트 배경

Second Tier Memory

이러한 로컬 DRAM의 확장 한계를 보완하기 위해 **CXL-memory(CMM-D)**가 등장했다.

CMM-D는 byte-addressable한 메모리로 CPU에서 load/store로 접근이 가능하며, CXL-memor의 지연(약 300ns)은 로컬 DRAM(약 100ns)보다 크지만 SSD/NVMe(약 70000ns) 대비 **현저히 낮다**.

따라서 **메모리를 수백 GB~TB 수준으로 확장**하면서도 IO device 기반 캐싱 대비 낮은 지연으로 데이터를 처리할 수 있어, 앞서 언급한 applications의 제약을 상당 부분 완화할 수 있다.



프로젝트 주제

Cache Replacement Policy

Memory-intensive한 application에서 cache replacement는 데이터 페이지의 hotness를 추적하여 충분히 자주 사용하는 페이지를 DRAM으로 promotion 한다. 이 때 워크로드의 패턴에 따라 인접 데이터의 prefetch를 수행하는데, prefetch 페이지 개수 및 stride로 건너뛰는 페이지 개수도 결정할 수 있다.

이때 prefetch 정도를 높이면 추후 접근되는 데이터의 cache hit ratio가 높아질 수 있으나, migration에 소요되는 **device bandwidth 점유**에 따른 **성능 저하의 문제**가 발생할 수 있다.

이러한 환경에서 디바이스별 성능 차이로 인해, 기존 IO device 환경의 cache replacement policy와 CXL-memory 환경의 cache replacement policy가 어떤 차이를 보이는지 **memory intensive workload를 시뮬레이션**하고 DQN을 이용해 **학습된 policy가 어떤 차이**를 보이는지 본 프로젝트에서 분석한다.

환경 및 데이터셋 분석

시뮬레이션 환경 설정

Tiered memory 환경을 시뮬레이션한다.

- **hotness** 추적: 페이지의 접근 빈도를 지수 가중 이동 평균(EWMA) 비율로 저장한 뒤,
  해당 페이지가 전체 페이지에서 상위 N% 이상의 접근 빈도를 가지는지 여부를 확인한다.

- DRAM 페이지 삽입: Promotion 된 페이지를 **DRAM에 저장**하고 DRAM 크기를 넘어가는 페이지는 LRU 방식으로 eviction한다.

- Episode 진행: 하나의 step마다 생성된 메모리 페이지 접근에 대해 **DRAM hit 여부를 확인**하고, miss 발생 시 promotion 및 prefetch 여부를 현재 **policy에 기반**해 결정한다.

- SSD / CXL-memory 디바이스: 디바이스의 접근 latency를 **각각 70000ns, 300ns**로 단순화하여 환경을 구성한다.

환경 및 데이터셋 분석

데이터셋

Phase별 접근 패턴이 다른 임의의 application의 워크로드를 시뮬레이션한다.

- Phase 1: **순차 스캔**
- Phase 2: **Stride**(기본 **4**) 접근
- Phase 3: s=1.1로 하는 Zipf-like한 분포로 **특정 페이지에 집중된 랜덤 접근**

이 세 개의 phase를 일정 주기로 바꿔가면서 다양한 패턴의 워크로드를 시뮬레이션한다.

State, action, reward 설계

State 설계

State는 다음과 같이 5개로 구성된다.

- Fault(cache miss) rate의 ewma 값
  → 현재 워크로드에서 DRAM에 캐시되지 않은 페이지가 얼마나 존재하는지를 나타냄

- 전체 DRAM 용량 중 사용 가능한 용량 비율
  → 남아있는 DRAM 용량을 바탕으로 policy 결정

- Migration 페이지 개수의 ewma 값
  → Migration의 ewma가 높다면 migration pressure가 높다는 뜻이므로 이를 policy 결정에 반영

- 최근 접근된 페이지들 중 중복 페이지 비율
  → 현재 워크로드 페이지의 locality를 나타냄

- 최근 접근된 페이지들 간의 거리 ewma 값
  → 접근 패턴이 striding일 경우 이를 policy 결정에 반영

State, action, reward 설계

Action 설계

Action은 미리 정해둔 15개의 preset을 기반으로,
(promote 여부 / prefetch 개수 / promote hotness 기준값 / prefetch할 페이지의 stride 값)의 튜플 형식으로 설계하였다.

Preset #	Promote	Prefetch	Hot_threshold	Stride
0	0	0	1.00	1
1	1	0	0.90	1
2	1	0	0.50	1
3	1	1	0.90	1
…				
10	1	2	0.50	4
…				

위는 preset의 일부로, 예를 들어 0번 preset은 promote 및 prefetch를 수행하지 않는 **보수적인** preset으로 설정하였다.
1번 프리셋은 promote(데이터를 DRAM으로 복사)를 수행하지만 prefetch를 수행하지 않으며, hotness가 상위 90%인 페이지만 promote를 수행한다. 2번 프리셋은 1번과 동일하지만 상위 50% 페이지를 promote하는 **비교적 공격적**인 preset이다.
6번 프리셋은 2개 페이지를 prefetch하며 4개 페이지를 건너뛰며(stride) prefetch를 진행(ex. 1, 5, 9번 페이지) 한다.

전체 preset

Preset #	Promote	Prefetch	Hot_threshold	Stride
0	0	0	1.00	1
1	1	0	0.90	1
2	1	0	0.50	1
3	1	1	0.90	1
4	1	1	0.50	1
5	1	1	0.50	2
6	1	1	0.50	4
7	1	2	0.90	1
8	1	2	0.50	1
9	1	2	0.50	2
10	1	2	0.50	4
11	1	4	0.90	1
12	1	4	0.50	1
13	1	4	0.50	2
14	1	4	0.50	4

State, action, reward 설계

Reward설계

현재 step에서 걸린 latency(L_t), migration을 결정한 총 byte 수(M_t), 현재 step에서 발생한 write amplification(W_t), migration으로 인한 bandwidth pressure($\overline{m_t} * nvm\_ns$)를 reward로 설정한다.
따라서 reward r_t 는 다음과 같이 나타낼 수 있다.

$$r_t = -L_t - \lambda_{mig}\frac{M_t}{2^{20}} - \lambda_{wa}\frac{W_t}{2^{20}} - \lambda_{mp}\overline{m_t}\frac{nvm\_ns}{4}$$

$$\overline{m_t} = (1-\alpha)\,\overline{m_{t-1}} + \alpha\, m_t, \qquad \overline{m_0} = 0, \quad 0 < \alpha < 1$$

이때, M_t 와 W_t 의 단위는 MiB이므로 일관된 규모를 위해 (1<<20)만큼을 나눠준다. 또한 bandwidth pressure는 디바이스의 latency에 비례한다는 가정하에 nvm_ns 를 곱한 뒤 4를 나눠 다른 변수와 균형을 맞췄다.

Latency를 줄이고, migration(promote) 을 최대한 적게 하며, prefetch로 인한 write amplification 이 낮고, migration 을 bursty하게 수행하여 높아지는 bandwidth pressure 를 최대한 줄이는 방식으로 학습을 진행하도록 설계하였다. → 결과적으로 **무분별한 migration(promote)및 prefetch를 최소화**하고 필요한 promote 만 수행하여 **DRAM hit rate를 높이는 방향**의 학습을 의도하였다.

강화학습 알고리즘 및 hyperparameter 설명

사용 알고리즘 및 hyperparameter

본 프로젝트에서는 DQN을 이용하였다.

Network (MLP)
Body: Linear–ReLU–Linear–ReLU (256,256 hidden layer)
Dueling / Double DQN 사용
Soft target update
Huber loss 및 clip(10.0)으로 outlier 및 학습 안정화

RL exploration: ε-greedy 사용
확률 ε로 랜덤 preset 선택, 그 외에는 $\mathrm{argmax}_a\, Q(s,a)$ 선택
학습 단계: ε 를 1.0→0.05로 decay.
평가 단계: greedy(ε=0)

γ(discount factor)=0.99, learning rate=0.001, τ(soft update rate)=0.005
Replay buffer: 200k
Batch size: 256
매 스텝마다 업데이트

실험 셋업

실험 환경 및 evaluation metric

시스템

CPU: i9-10900K
Memory: 64GB (DDR4)
GPU: RTX 2070 SUPER

시뮬레이션 파라미터 (baseline)

Page size: 4 KB
DRAM access latency: 100 ns
SSD latency(nvm_ns): 70,000 ns
CMM-D latency(nvm_ns): 300 ns
DRAM capacity: 100,000 pages (400MB)
hotness ewma alpha: 0.05, bandwidth pressure ewma alpha: 0.1
Episode 길이: 100,000 steps
Reward 가중치: λ_migration=0.002, λ_writeamp=0.001, λ_mi_pressure=0.002

실험 셋업

실험 환경 및 evaluation metric

Evaluation metric

- 평가 시 preset별 선택된 횟수(action 분포)
- Page fault(DRAM miss) 횟수
- 평균 latency
- Migration 된 페이지 개수
- Migration pressure(bandwidth pressure)

실험 결과

디바이스별 action 분포 차이

SSD 환경에서 학습된 모델을 평가하였을 때 다음과 같은 action 분포를 보인다.

SSD

action	count
12	200750
11	183321
8	90850
7	16987
4	4671
…	

12, 11, 8, 7, 4번 preset 모두 prefetch을 수행하는 preset.
가장 prefetch를 많이 수행하는 preset이 가장 많이 선택됨.

→ DRAM miss 시 발생하는 latency가 높기 때문에 **prefetch를 통한 DRAM hit 확률을 높이는 것이 유리**하도록 학습되었음을 확인할 수 있다.

그러나 stride 패턴으로 prefetch하는 preset을 선택하는 경우가 드문데, 이는 잘못된 stride 패턴으로 prefetch하는 경우 발생하는 migration 비용이 크기 때문이다. Phase1과 3에서 stride 없이 prefetch만을 수행하여 얻을 수 있는 이득이 크기 때문에 이러한 policy 를 학습한 것으로 보인다.

실험 결과

디바이스별 action 분포 차이

CXL-memory에서 각각 학습된 모델을 평가하였을 때 다음과 같은 action 분포를 보인다.

CXL-memory

action	count
1	153751
6	106055
13	41682
10	38929
3	27209
5	24993
0	20566
4	19392
7	16021
…	

SSD와 다르게 특정 preset에 **집중된 경향이 낮음**을 확인할 수 있다. 1번 preset은 hotness가 상위 90% 이상인 페이지만 migration을 수행한다. 6, 13, 10번 preset은 4 단위로 stride 하여 페이지를 prefetch한다. 3, 5, 0, 4, 7의 경우 비슷한 분포를 보이고, 각각은 워크로드 패턴에 맞춘 preset으로 보인다 .

이러한 결과는 CXL-memory 자체의 **낮은 latency** 로 인해 DRAM miss가 발생해도 **비용이 크지 않기** 때문이다. 오히려 **과도한 prefetch 및 지속적인 prefetch**로 인한 migration pressure 로 인한 비용이 상대적으로 크게 학습되어 **낮은 prefetch 개수**를 수행하며 필요에 따라 **migration pressure를 완화**하기 위해 promote를 수행하지 않는(특히 0번 preset) policy로 학습된 것으로 보인다.

→ CXL-memory 환경에서 한 번의 접근에서 **필요한 페이지만 prefetch**하고, 비교적 공격적인 **striding**을 수행하여 성능을 최적화하도록 학습이 진행됨을 확인하였다.

실험 결과

Baseline 지표

기본 설정에서 SSD 및 CXL-memory의 평가 결과는 다음과 같다.

CXL-memory

Episode	Latency avg (ns)	Pages migrated	Faults	Migration pressure
0	177.634	8008	38817	0.08008
1	177.478	8293	38739	0.08293
2	174.892	8641	37446	0.08641
3	178.092	8850	39046	0.0885
4	174.432	8117	37216	0.08117
avg	176.5056	8381.8	38252.8	0.083818

SSD

Episode	Latency avg (ns)	Pages migrated	Faults	Migration pressure
0	27733.57	10928	39533	0.10928
1	27525.27	11207	39235	0.11207
2	25638.66	11605	36536	0.11605
3	27193.94	10915	38761	0.109078
4	25648.45	11536	36550	0.11536
avg	26747.98	11238.2	38123	0.112367

→ Action 분포와 마찬가지로 SSD가 CXL-memory보다 더 많은 page를 migration 하는 것을 확인할 수 있다. Migration pressure 또한 SSD가 더 높다.
Fault(DRAM miss)는 두 경우 비슷한 것을 확인할 수 있다.

실험 결과

Policy 적합성

서로 policy를 바꾼 결과는 다음과 같다.

CXL-memory

Episode	Latency avg (ns)	Pages migrated	Faults	Migration pressure
0	179.066	10928	39533	0.10928
1	178.47	11207	39235	0.11207
2	173.072	11605	36536	0.11605
3	177.522	10915	38761	0.109078
4	173.1	11536	36550	0.11536
avg	176.246	11238.2	38123	0.112367

SSD

Episode	Latency avg (ns)	Pages migrated	Faults	Migration pressure
0	27233.08	8008	38817	0.08008
1	27178.56	8293	38739	0.08293
2	26274.75	8641	37446	0.08641
3	27393.15	8850	39046	0.0885
4	26113.98	8117	37216	0.08117
avg	26838.71	8381.8	38252.8	0.083818

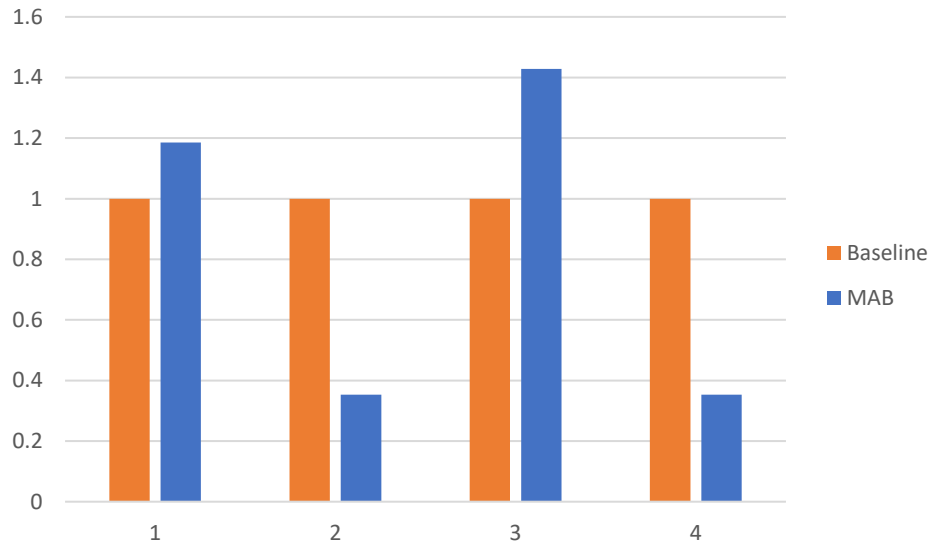
→ SSD의 latency가 100ns가량 하락하였다.
또한 CXL-memory의 pages migrated 지표가 증가함을 확인하였다.

실험 결과

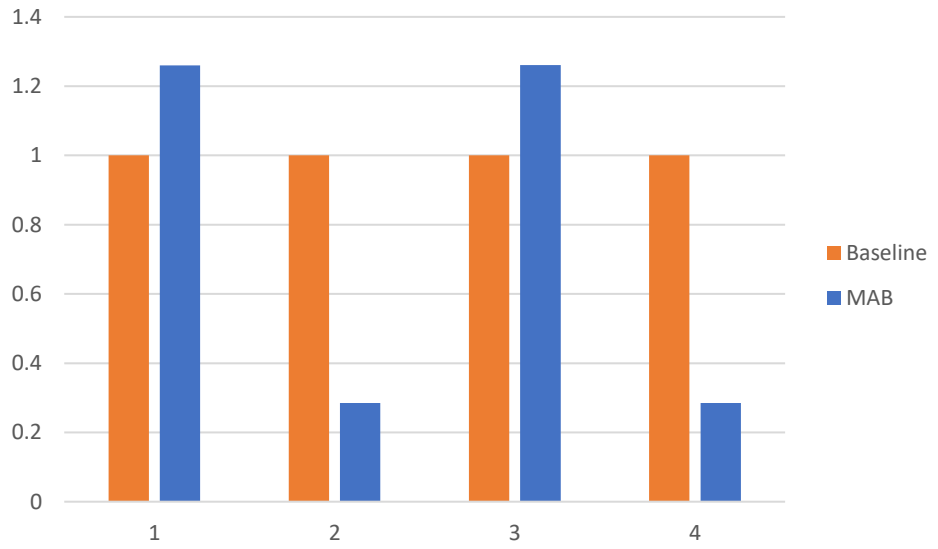
RL algorithm 비교

같은 환경에서 multi-armed bandit(MAB) 알고리즘을 사용한 것과 비교한 결과는 다음과 같다.

그래프 x축의 1, 2, 3, 4 순서대로 **latency(ns), migrated pages, faults, migration pressure**를 나타내고,
각각은 baseline(DQN)에 대한 정규화된 값을 나타낸다.



CXL-memory



SSD

모든 디바이스에 대해서 latency가 **20%가량 증가**하였으며, migration이 **70%가량 줄고** fault가 **30~40% 증가**하였다.
특정 페이지의 promote 시 발생하는 DRAM 점유, cache hit 비율에 끼치는 영향 등의 **지연된 효과**를 MAB 방식은 충분히 **고려하지 못하기 때문**으로 보인다. 따라서 시스템 측면에서 예측이 어려운 워크로드를 **state를 통해 간접적으로 표현**하여 장기적인 action의 가치를 추론 가능한 **DQN이 본 프로젝트에 적합**하다.

실험 결과

Learning rate에 따른 학습 추이

LR 값을 기존 0.001에서 0.002로 바꾸어 수행한 결과는 다음과 같다.

	LR 0.001	LR 0.002
SSD		
CXL-memory		

Scale차이

SSD 환경에서 loss값은 수렴하나 전 구간에서 증가하였다.

CXL-memory 환경에서는 LR=0.002인 경우 loss 값이 발산하여 기존에 비해 loss 값이 약 30배 이상 차이나는 것을 확인하였다.

평균 **latency** 또한 CXL-memory는 약 **5%**, SSD는 약 **20% 증가**되었음을 확인하였다.

실험 결과

Random seed에 따른 차이 및 신뢰구간

5개의 random number generator seed를 설정하여 실험을 수행한 뒤,
각각의 지표(latency, migrated pages, faults, migration pressure)를 수집하였다.

수집된 데이터를 바탕 episode에 대해 평균을 낸 뒤, T-분포를 이용해 계산한
CXL-memory의 95% 신뢰구간은 다음과 같다.

Latency avg (ns): 177.329 ± 6.711
Migrated pages: 8,194.9 ± 2,758
Faults: 38,664.6 ± 3,359
Migration pressure: 0.08194 ± 0.0276

동일 방식으로 계산한 SSD의 95% 신뢰구간은 다음과 같다.

Latency avg (ns): 30,776.06 ± 2,861.0
Migrated pages: 6,310.08 ± 3,541.0
Faults: 43,885.64 ± 4,094.0
Migration pressure: 0.06309 ± 0.0354

토의 및 결론

실험 결과 정리 및 개선안

더 낮은 latency를 가진 CXL-memory 디바이스는 특정 seed에서 SSD보다 promote 시 더 적은 개수의 페이지를 prefetch하여 bursty한 접근을 줄이는 방식으로 학습되었다.

그러나 신뢰 구간에서 확인할 수 있듯, 다양한 seed에서는 오히려 CXL-memory가 SSD보다 **더 많은 page를 공격적으로 migration** 하는 경우도 많은 것을 확인할 수 있다. 이는 CXL-memory 환경의 reward에서 **nvm_ns가 상대적으로 작기 때문에** DRAM miss penalty가 작아, **비슷한 성능**(평균 latency)을 보이는 **더 다양한 policy**를 학습할 수 있다는 것을 의미한다. 따라서 CXL-memory 환경에서는 **더 다양하고 공격적인 cache replacement policy를 적용**할 수 있다는 것을 시사한다.

본 프로젝트에서는 시뮬레이션 환경의 한계로 migration pressure가 실제 디바이스의 latency 에 미치는 영향을 정량적으로 고려하지 않았다. 이는 **실제 디바이스 환경**에서 실험을 진행하는 것으로 개선할 수 있다.
디바이스 bandwidth 점유 및 queueing overhead로 인해 **colocate된 다른 application 에 미치는 성능 영향**을 분석하는 방향으로 본 프로젝트를 확장 가능하다.

추가로 학습 규모로 인해 수행하지 못하였으나, 실제 **application 의 메모리 접근을 직접 trace**하여 확보된 데이터셋을 바탕으로 학습을 진행하면 보다 실제적인 모델을 학습하도록 개선할 수 있다.
마지막으로, 메모리 접근마다 DQN 추론을 수행하지 않고, 백그라운드에서 **일정 주기마다 promotion policy를 갱신**하는 방식으로 실제 서비스에서의 성능 영향을 최소화 할 수 있다.