

공학 석사 학위 논문

거대 규모 데이터 캐시 관리 기법

Large scale data cache management scheme

2024년 2월

서울시립대학교 대학원
컴퓨터과학과
유수원

거대 규모 데이터 캐시 관리 기법

Large scale data cache management scheme

지도교수 이 동 희

이 논문을 석사학위 논문으로 제출함

2023년 12월

서울시립대학교 대학원

컴퓨터과학과

유수원

유수원의 공학 석사학위 논문을 인준함.

심 사 위 원 장 황 혜 수 인

심 사 위 원 김 민 호 인

심 사 위 원 이 동 희 인

2023년 12월

서울시립대학교 대학원

국 문 초 록

컴퓨터 시스템에서 각 장치의 상대적인 속도는 매우 다르다. 예를 들면 DRAM과 같은 메인 메모리에 비해 하드디스크와 같은 저장장치는 데이터 접근 시간이 매우 크다. 또한, 일반적으로 데이터를 네트워크로부터 가져오는 속도 역시 DRAM의 접근 시간보다 크다. 따라서 메인 메모리의 일부 영역을 데이터 캐시 공간으로 사용하고, 한 번 읽어온 데이터를 캐시에 저장하며, 해당 데이터가 다시 접근될 때 캐시에서 가져오면 데이터 접근 시간을 감소시킬 수 있다. 이러한 목적으로 데이터 캐시 기법이 시스템의 많은 부분에서 널리 사용되고 있다.

새로운 데이터를 캐시에 저장하기 위해서는 캐시에 존재하는 기존 데이터를 제거하여 새로운 데이터를 위한 공간을 마련해야 한다. 따라서 어떤 데이터를 제거할 것인지를 결정하는 교체 기법이 존재한다. 전통적으로 사용되는 기법으로 LRU, 2Q, 그리고 ARC 기법들이 있다. 그리고 현재 많은 기법은 이러한 전통적인 기법을 응용하거나 조합하여 만들어진 것들이다. 캐시 교체 기법의 목표는 미래에 사용가능성이 가장 작은 데이터를 골라 제거하여 캐시 히트율을 높이는 것이다. 따라서 캐시 교체 기법의 성능은 주로 히트율을 사용하여 비교한다.

현재 많이 사용되고 있는 캐시 관리 기법은 LRU, 2Q 그리고 ARC 캐시 알고리즘과 같은 고전적인 캐시 관리 기법을 응용하여 사용하고 있다. 이러한 고전적인 캐시 관리 기법은 데이터를 관리하기 위해서 더블 링크 리스트를 사용하는데, 이러한 리스트는 포인터를 이용하여 데이터들을 연

결한다. 또한, 데이터가 캐시 내에 존재하는지, 그리고 존재하면 어느 위치에 있는지를 빠르게 찾기 위해 해시 테이블을 사용해서 데이터를 검색한다. 하지만 이러한 리스트와 해시 테이블 방식은 많은 공간을 차지한다. 아울러 해시 테이블을 검색하고 리스트에 데이터를 삽입하고 삭제하는 관리 작업은 전체 데이터 접근 시간을 증가시켜 성능을 떨어뜨리는 요소로 작용한다.

컴퓨팅 환경은 계속 변화하고 있다. 특히 최근의 변화를 살펴보면, 데이터 소스로부터 데이터를 가져오는 데이터 접근 시간의 감소와 컴퓨터 시스템 메모리의 증가를 들 수 있다. 고전적인 데이터 소스 중 하나인 저장장치의 경우 하드디스크를 대신하여 SSD와 같은 플래시 메모리 저장장치가 널리 사용되고 있다. 그 결과 데이터를 가져오는데 소요되는 입출력 시간이 급격히 감소하였다. 또 다른 변화는 메모리의 증가이며, 그 결과 가져온 데이터를 저장하는 캐시의 크기 역시 매우 크다.

디바이스로부터 데이터를 읽는 속도가 빨라지고, 캐시의 크기가 커질수록 캐시 교체 기법의 효율은 그 중요성이 감소한다. 특히 캐시의 크기가 커질수록 캐시 교체 기법 간 히트율 차이는 감소한다. 반면 거대한 캐시에 존재하는 많은 수의 데이터들을 관리하기 위한 관리 오버헤드가 증가한다. 아울러 디바이스로부터 데이터를 읽어오는 시간이 감소할수록 히트율의 증가로 얻을 수 있는 이득은 감소한다. 본 논문은 이러한 컴퓨팅 환경의 변화에 따라 새로운 기준으로 캐시 교체 기법을 비교하고, 또한 거대 규모 캐시를 위한 교체기 법으로 AP(Associative Pocket) 기법을 제안한다. AP 기법은 세트 연관 사상을 응용하여 만들어진 교체 기법이다.

본 논문에서는 LRU, 2Q, ARC, 그리고 AP 기법의 오버헤드를 측정한

다. 특히 각 기법이 데이터를 관리하기 위해 감당해야 하는 메모리 오버헤드를 측정한다. 다음으로 각 캐시 교체 기법의 실행시간을 측정한다. 이러한 비교는 캐시 히트율만 비교하는 전통적인 비교 방법과는 다르며, 캐시 교체 기법의 오버헤드와 히트율 간의 트레이드 오프를 보여준다.

비교 결과에 따르면 AP 기법은 거대 규모 캐시에서 히트율도 전통적인 기법과 유사하면서도 오버헤드가 매우 낮음을 보여준다. 구체적으로 AP 기법은 데이터를 연결하기 위해서 포인터가 사용하지 않으며, 데이터를 검색하기 위해 사용되는 해시 테이블 방식을 사용하지 않기 때문에 연산 오버헤드가 작다. 그리고 AP 기법은 리스트나 해시 테이블을 사용하지 않아서, 다른 기법보다 메모리 공간을 더 적게 사용하며, 그 결과 더 많은 데이터를 캐싱할 수 있다. 그리고 이러한 캐싱 가능한 용량의 차이는 캐시의 크기가 커질수록 더 커진다. AP 기법은 대표적인 캐시 교체 기법인 LRU 보다 약 60% 정도 빠른 데이터 접근 시간을 보여주며, 히트율 면에서 캐시 크기가 커질수록 교체 기법 간의 성능은 비슷하다.

키워드: 세트 연관 사상, LRU, 2Q, ARC, 캐시 알고리즘, large scale cache

공학석사 학위논문

거대 규모 데이터 캐시 관리 기법

Large scale data cache management scheme

2024년 2월

서울시립대학교 대학원
컴퓨터과학과
유수원

목 차

1. 서론	1
2. 배경 지식 및 관련 연구	5
2.1 LRU	5
2.2 2Q	7
2.3 ARC	10
2.4 기타 교체 정책	12
3. 세트 연관 사상 블록 캐싱 설계	14
3.1 헤더 구조체 설계	14
3.2 AP 기법의 동작	16
4. 분석 및 비교 실험 결과	21
4.1 데이터 블록 메모리 사용량 비교	21
4.2 참조 스텝 비교 분석	23
4.3 실험 환경	27
4.4 세트 수에 따른 참조 시간 비교 및 분석	29
4.5 각 캐시 교체 기법들의 성능 비교	31
4.5 AP 캐싱 세트에 따른 성능	39
5. 결론	42
참고 문헌	43
Abstract	47
감사의 글	51

그 림 목 차

그림1 LRU 알고리즘	6
그림2 2Q 알고리즘	9
그림3 ARC 알고리즘	11
그림4 AP 기법	19
그림5 AP HIT인 경우	20
그림6 AP 모두 채워져 있지 않은 경우	20
그림7 AP 모두 채워져 있는 경우	20

표 목차

표 1 알고리즘 데이터 노드 개수 비교표	21
표 2 AP 저장 가능 노드 비교 비율	22
표 3 실험에 사용된 워크로드 0 특성	28
표 4 실험에 사용된 워크로드 1 특성	28
표 5 실험에 사용된 워크로드 2 특성	28
표 6 실험용 컴퓨터 사양	28
표 7 세트별 할당공간	29
표 8 워크로드0의 캐시 히트율과 메모리 오버헤드 실험 결과 비교	31
표 9 워크로드1의 캐시 히트율과 메모리 오버헤드 실험 결과 비교	33
표 10 워크로드2의 캐시 히트율과 메모리 오버헤드 실험 결과 비교	35
표 11 AP 기법에서 세트 수 변경 시 캐시 히트율 메모리 오버헤드 실험 결과	39

차트 목차

차트 1 알고리즘 데이터 노드 개수 비교 차트	22
차트 2 세트별 히트율	30
차트 3 세트별 컴퓨팅 오버헤드	30
차트 4 워크로드0 알고리즘 캐시 크기별 캐시 히트율 비교	32
차트 5 워크로드0 알고리즘 캐시 크기별 메모리 오버헤드 비교	32
차트 6 워크로드1 알고리즘 캐시 크기별 캐시 히트율 비교	34
차트 7 워크로드1 알고리즘 캐시 크기별 메모리 오버헤드 비교	33
차트 8 워크로드2 알고리즘 캐시 크기별 캐시 히트율 비교	36
차트 9 워크로드2 알고리즘 캐시 크기별 메모리 오버헤드 비교	36
차트 10 알고리즘 캐시 크기별 캐시 히트율 비교	40
차트 11 AP 기법 세트 수당 컴퓨팅 오버헤드	40

알고리즘 목차

알고리즘 1 AP 알고리즘	18
알고리즘 2 LRU 알고리즘	23
알고리즘 3 2Q 알고리즘	24
알고리즘 4 ARC 알고리즘	25

1. 서론

컴퓨터 시스템에서 각 장치의 상대적인 속도는 매우 다르다. 예를 들면 DRAM과 같은 메인 메모리에 비해 하드디스크와 같은 저장장치는 데이터 접근 시간이 매우 크다[15]. 또한, 일반적으로 데이터를 네트워크로부터 가져오는 속도 역시 DRAM의 접근 시간보다 크다. 따라서 메인 메모리의 일부 영역을 데이터 캐시 공간으로 사용하고, 한 번 읽어온 데이터를 캐시에 저장하며, 해당 데이터가 다시 접근될 때 캐시에서 가져오면 데이터 접근 시간을 감소시킬 수 있다. 이러한 목적으로 데이터 캐시 기법이 시스템의 많은 부분에서 널리 사용되고 있다.

현재 캐싱은 하드웨어와 소프트웨어의 발전이 함께 이루어지고 있다 [12]. 하드웨어 기반은 캐시는 자동화 측면이 우수하지만, 사용자가 직접 정의할 수 있는 부분이 떨어진다. 이에 반해 소프트웨어 기반 캐시는 사용자가 직접 정의할 수 있는 부분이 우수하지만, 자동화 측면이 떨어지는 점을 보인다. 대규모 캐시의 경우 워크로드의 다양성 때문에 워크로드의 특성에 맞게 캐싱하기 위해서 소프트웨어적인 측면의 발전이 중요할 것이다.

새로운 데이터를 캐시에 저장하기 위해서는 캐시에 존재하는 기존 데이터를 제거하여 새로운 데이터를 위한 공간을 마련해야 한다. 따라서 어떤 데이터를 제거할 것인지를 결정하는 교체 기법이 존재한다. 전통적으로 사용되는 기법으로 LRU, 2Q, 그리고 ARC 기법들이 있다. 그리고 현재 많은 기법은 이러한 전통적인 기법을 응용하거나 조합하여 만들어진 것

들이다. 전통적인 캐시 교체 기법의 목표는 미래에 사용 가능성이 가장 작은 데이터를 골라 제거하여 캐시 히트율을 높이는 것이다[16]. 따라서 캐시 교체 기법의 성능은 주로 히트율을 사용하여 비교한다.

LRU, 2Q 그리고 ARC 캐시 알고리즘과 같은 캐시 관리 기법은 데이터를 관리하기 위해서 더블 링크드 리스트를 사용하는데, 이러한 리스트는 포인터를 이용하여 데이터들을 연결한다. 또한, 데이터가 캐시 내에 존재하는지, 그리고 존재하면 어느 위치에 있는지를 빠르게 찾기 위해 해시 테이블을 사용하여 데이터를 검색한다[9]. 하지만 이러한 리스트와 해시 테이블 방식은 많은 공간을 차지한다. 아울러 캐시의 크기가 커지면 더욱 많은 해시 테이블이 필요하게 된다. 이에 따라서 검색하고 리스트에 데이터를 삽입하고 삭제하는 관리 작업은 전체 데이터 증가에 따라 접근 시간을 증가시켜 성능을 떨어뜨리는 요소로 작용한다.

컴퓨팅 환경은 계속 변화하고 있다. 특히 최근의 변화를 살펴보면, 데이터 소스로부터 데이터를 가져오는 데이터 접근 시간의 감소와 컴퓨터 시스템 메모리의 증가를 들 수 있다. 고전적인 데이터 소스 중 하나인 저장 장치의 경우 하드디스크를 대신하여 SSD와 같은 플래시 메모리 저장장치가 널리 사용되고 있다. 그 결과 데이터를 가져오는데 소요되는 입출력 시간이 급격히 감소하였다. 또 다른 변화는 워크로드의 다양성 및 크기 증가이며 이에 따른 메모리의 크기 증가이며, 그 결과 가져온 데이터를 저장하는 캐시의 크기 역시 매우 크다.

현재 모든 디스크의 크기가 증가하고 있다. 더 정확히는 지금까지 18개월마다 디스크의 크기가 약 두 배씩 증가하고 있다. 이에 따라서

캐시의 크기가 같이 꾸준히 증가하고 있으므로 캐시 교체 기법은 프록시 캐싱 발전을 제한하는 요소로 보이지는 않는다[21]. 따라서 컴퓨터 하드웨어의 발전으로 인한 캐시의 크기 증가로 인하여서 LRU 캐시 알고리즘으로도 캐시의 성능을 발휘하는 데 충분하다[20].

워크로드의 다양성 증가로 인해 오버헤드의 증가로 기존의 전통적인 캐시 방법보다 새로운 방식이 필요하다[13, 14, 16]. 전통적인 캐시 방법에서는 캐시의 크기가 커질수록 캐시 교체 기법 간 히트율 차이는 감소한다[6, 13, 16]. 아울러 디바이스로부터 데이터를 읽어오는 시간이 감소할수록 히트율의 증가로 얻을 수 있는 이득은 감소한다. 본 논문은 이러한 컴퓨팅 환경의 변화에 따라 새로운 기준으로 캐시 교체 기법을 비교하고, 또한 거대 규모 캐시를 위한 교체 기법으로 AP(Associative Pocket) 기법을 제안한다.

LRU, 2Q, ARC, 그리고 AP 기법의 오버헤드를 측정한다. 특히 각 기법이 데이터를 관리하기 위해 감당해야 하는 메모리 오버헤드를 측정한다. 다음으로 각 캐시 교체 기법의 실행시간을 측정한다. 이러한 비교는 캐시 히트율만 비교하는 전통적인 비교 방법과는 다르며, 캐시 교체 기법의 오버헤드와 히트율 간의 트레이드 오프를 보여준다. 비교 결과에 따르면 AP 기법은 거대 규모 캐시에서 히트율도 전통적인 기법과 유사하면서도 오버헤드가 매우 낮음을 보여준다. 캐시 관리 기법의 오버헤드가 줄어들면 전력 소비량도 감소하며, 이러한 이유로 효율적인 캐시 관리 기법은 배터리 소비량이 중요한 임베디드 디바이스에 유용하게 사용될 수 있다 [11].

기준 기법들은 데이터를 리스트로 관리하는데, 데이터들을 리스트로 연결하기 위해 이중 포인터를 사용하며, 또한 캐시 내에 해당 데이터가 존재하는지 알기 위해 해시 테이블을 사용한다. 아울러 캐시 교체 효율을 높이기 위해 캐시에서 제거된 데이터 ID를 일정 시간 유지하는 고스트 버퍼(ghost buffer)를 사용하기도 한다. 이러한 복잡한 방식은 캐시된 데이터를 검색하고 접근하는 시간을 줄이고 캐시 히트율을 항상 시키는 데 도움이 되지만 메모리 오버헤드와 해싱에 필요한 시간과 같은 컴퓨팅 오버헤드도 유발한다. 그러나 AP 기법은 리스트나 해시 키이블과 같은 복잡한 알고리즘을 이용하지 않는다. 즉 AP 기법은 메모리 공간을 더 효과적으로 사용하면서 매우 빠르게 데이터를 검색하고 접근할 수 있다. AP 기법은 세트 연관 사상 기법을 응용하여 만들어진 기법으로 세트 개수에 따라서 접근 시간 및 히트율이 변할 수 있다. 본 논문에서는 실험을 통해 이러한 히트율의 변화를 관찰하였다.

본 논문은 다음과 같이 구성되어 있다. 2장에서는 캐싱과 관련된 배경 지식 및 다양한 연구를 살펴보며, 3장에서 세트 연관 사상을 이용한 AP 캐싱 기법에 대해서 설명한다. 4장에서는 LRU, 2Q 그리고 ARC와 제안된 AP 캐싱 기법을 비교하고, 실험 결과를 제시한다. 마지막으로 5장에서 결론을 맺는다.

2. 배경 지식 및 다양한 연구

2.1 LRU(Least Recently Used)

LRU(Least Recently Used) 기법은 캐시에 데이터 블록을 저장하는데 사용되는 가장 널리 사용되는 기법이다[1]. LRU 기법은 LRU 리스트라 불리는 리스트를 유지한다. LRU 리스트의 양 끝단은 LRU 위치와 MRU(Most Recently Used) 위치라고 불린다. 새로운 데이터는 LRU 리스트에서 MRU (Most Recently Used) 위치에 삽입된다. 새로운 데이터를 위한 공간이 필요하면, LRU 위치에 있는 데이터를 제거하여 새로운 데이터를 위한 공간을 마련한다. 이처럼 LRU 위치에 있는 데이터를 제거하기 때문에 LRU 교체 기법이라 불린다. LRU 기법은 또한 데이터가 참조될 때마다 저장된 데이터를 LRU 리스트에서 제거한 후 MRU 위치로 다시 삽입한다. 따라서 데이터들은 LRU 리스트에서 참조 시간순으로 정렬되어 있다.

LRU 리스트는 특정 데이터가 캐시 내에 존재하는지, 그리고 LRU 리스트의 어디에 존재하는지 빠르게 판별하기 위해서 해시 테이블이 필요하다 [9]. 만약 해시 테이블이 존재하지 않으면 특정 블록 ID가 캐시 내에 존재하는지 판별하기 위해 LRU 리스트를 순차 검색해야 하는데 이 경우에 캐시의 크기가 조금만 커져도 엄청난 컴퓨팅 오버헤드를 가져올 수 있다. 따라서 이러한 순차 검색을 피하려면 해시 테이블과 같은 별도의 자료 구조 사용은 피하기 어렵다. 이러한 해시 테이블의 크기는 인자로 조절할 수 있다. 해시 테이블의 크기가 클수록 검색시간은 빠르지만, 메모리 오버

헤드는 커진다. 예를 들어 해시 테이블이 N개의 리스트로 구성되면 해시 테이블의 가지는 공간 복잡도는 $O(n)$ 이다. 그리고 N값이 커질수록 해시 테이블을 통해 검색하는 시간이 감소한다. 예를 들어 캐시 크기가 N이고 해시 테이블이 N개의 리스트로 구성되면, 평균적으로 한 번의 해시 리스트 접근으로 해당 데이터를 검색할 수 있으며, 이 경우 평균 시간 복잡도는 $O(1)$ 이다. 그리고 해시 테이블이 $N/2$ 개의 리스트로 구성되면, 하나의 해시 리스트에 평균 2개의 블록이 연결되어 있으므로 평균 시간 복잡도는 $O(2)$ 가 된다. 데이터가 들어오게 되면 데이터 ID를 인자로 해시 함수를 실행하여 해시 리스트 하나를 선택한다. 그 후 해당 인자에서 순차적으로 검색하여 해당 데이터를 찾아가는 방식이다. 데이터의 특성에 따라서 다양한 해시 함수를 사용할 수 있다. 정해진 해시 테이블의 크기만큼 나누는 간단한 방법부터 GF, adHash, Xhash 그리고 MuHash함수와 같이 다양한 해시 함수를 사용할 수 있다[10, 17, 18].

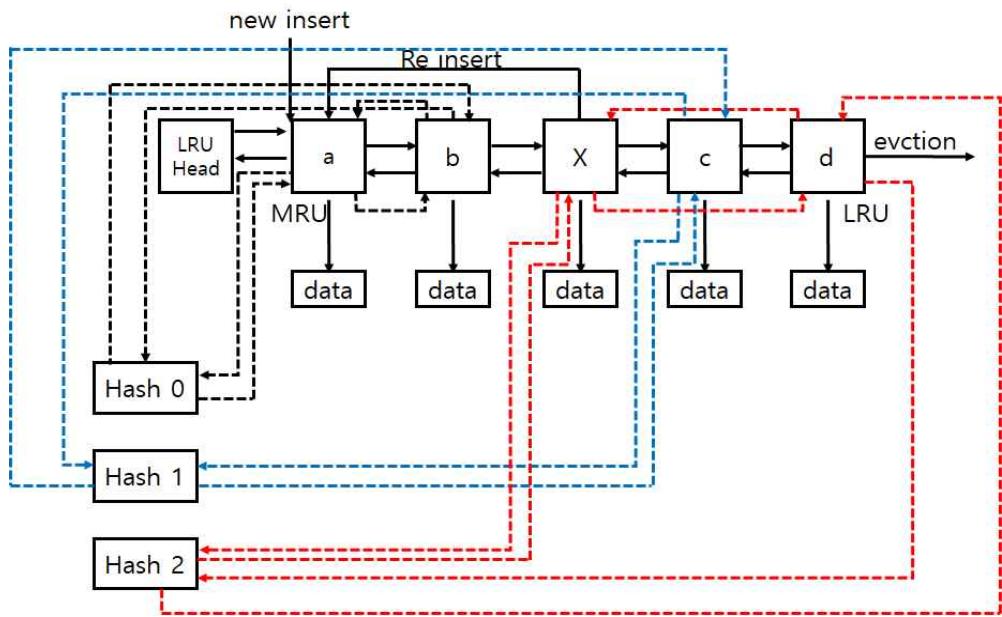


그림 1 LRU 알고리즘

2.2 2Q

2Q 교체 기법은 그 이름이 의미하는 것처럼 기본적으로 2개의 리스트를 사용한다[2]. 2개의 리스트 중 하나는 A1 in 리스트이며, 다른 하나는 Am 리스트이다. 데이터 블록이 처음 참조되면 해당 블록은 A1in 리스트의 MRU 위치에 삽입된다. 그러나 뒤에서 설명하는 바와 같이 해당 블록의 정보가 A1out에 존재하여 여러 번 참조된 블록으로 판명되면 해당 블록은 Am 리스트에 삽입된다. Ain 리스트의 MRU 위치에 데이터를 삽입할 때 캐시에 여유 공간이 없으면 A1in 리스트의 LRU 위치에 있는 데이터를 제거한다. 데이터 블록이 Am 리스트의 MRU 위치에 삽입될 때 캐시에 여유 공간이 없다면 Am 리스트의 LRU 위치에 있는 데이터를 캐시에서 제거한다.

이와 같은 동작은 2Q 기법이 스캔 저항성(scan resistant)을 가지게 한다. 영화 파일과 같이 데이터 블록을 한 번만 참조하는 시퀀셜 액세스가 발생하면, 다른 유용한 데이터들을 모두 캐시에서 내보내고 영화 파일의 데이터 블록이 캐시를 차지하게 된다. 그런데 이러한 시퀀셜 참조 블록은 다시 참조될 가능성이 매우 작으며, 다른 유용한 데이터를 캐시에서 내보냈기 때문에 캐시의 히트율이 매우 낮아진다. 이러한 현상을 방지하기 위해 캐시 교체 기법을 설계할 때 스캔 저항성이 매우 중요하다. 2Q 기법은 한 번만 참조된 데이터 블록은 A1 in 리스트에 유지하기 때문에 시퀀셜 액세스가 발생하더라도 Am 리스트에 유지되는 데이터들을 보호할 수 있으며, 이러한 이유로 스캔 저항성을 가진다.

2Q 교체 정책은 한 단계 더 나아가 A1out이라는 또 하나의 리스트를 가

진다. A1out 리스트에는 고스트 버퍼가 연결되어 있다. A1in에 있던 데이터 블록이 제거되면 해당 데이터의 메타 데이터만 고스트 버퍼에 저장하고 이 고스트 버퍼를 A1out 리스트에 연결한다. 즉 해당 데이터의 실제 내용인 데이터 블록 자체는 존재하지 않으며 데이터의 블록 ID와 같은 메타 정보만 고스트 버퍼에 유지하기 때문에 고스트 버퍼가 차지하는 메모리 공간은 A1in이나 Am 리스트에 있는 데이터 블록보다 매우 작다.

2Q 기법은 A1in에 존재하던 데이터 블록이 제거되면, 해당 블록의 메타 정보만 고스트 버퍼로 만들어서 A1 out 리스트의 MRU 위치에 삽입한다. 이때 A1out 리스트의 길이가 최대 크기에 도달하여 여유 고스트 버퍼가 없다면 A1out의 LRU 위치에 있는 고스트 버퍼를 삭제하고 이를 재활용 한다. 이제 2Q 알고리즘의 정확한 동작을 기술하면 다음과 같다. 어떤 블록이 참조되었다고 가정하자. 만약 해당 블록이 Am 리스트에 존재하면 이 블록은 Am 리스트에서 MRU 위치로 이동한다. 그러나 해당 블록이 A1in 리스트에 존재하면 이 블록을 A1in 리스트의 MRU 위치로 이동시키지 않는다. 2Q 기법은 A1in 리스트의 경우 FIFO 방식으로 교체한다. 해당 블록이 Am과 A1in 리스트에 존재하지 않으며, 해당 블록 ID가 A1out 리스트에 존재하면 해당 블록을 저장장치로부터 읽고 Am 리스트의 MRU 위치에 삽입한다. 이와 동시에 A1out 리스트에서 해당 블록의 고스트 버퍼를 삭제한다. 만약 위 3개의 리스트에서 해당 블록을 참지 못하면 해당 블록을 저장장치로부터 읽고 A1in 리스트의 MRU 위치에 삽입한다.

2Q 기법의 구현을 보면 Am, A1in, 그리고 A1out와 같은 세 개의 리스트 이외에도 3개의 해시 테이블을 유지해야 한다. 이 각각의 해시 테이블

을 해당 리스트에 특정 블록 ID가 존재하는지 빠르게 검색하기 위해 사용된다. 그리고 새로운 블록이 리스트에 삽입되거나 삭제될 때 해시 테이블도 변경되어야 한다. 결국, 2Q 기법은 세 개의 리스트와 세 개의 해시 테이블을 유지해야 하므로 하나의 리스트와 하나의 해시 테이블을 유지하는 LRU 기법과 비교하여 메모리 오버헤드가 크다. 또 다수의 리스트를 관리하고 해시 테이블을 검색해야 하므로 컴퓨팅 오버헤드도 크다.

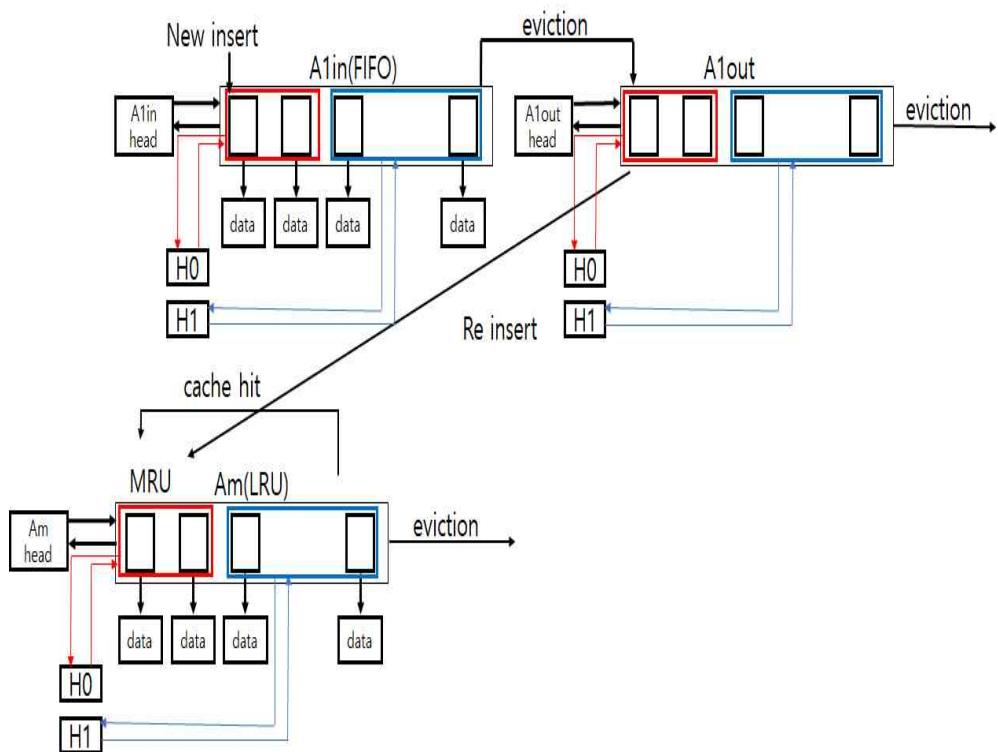


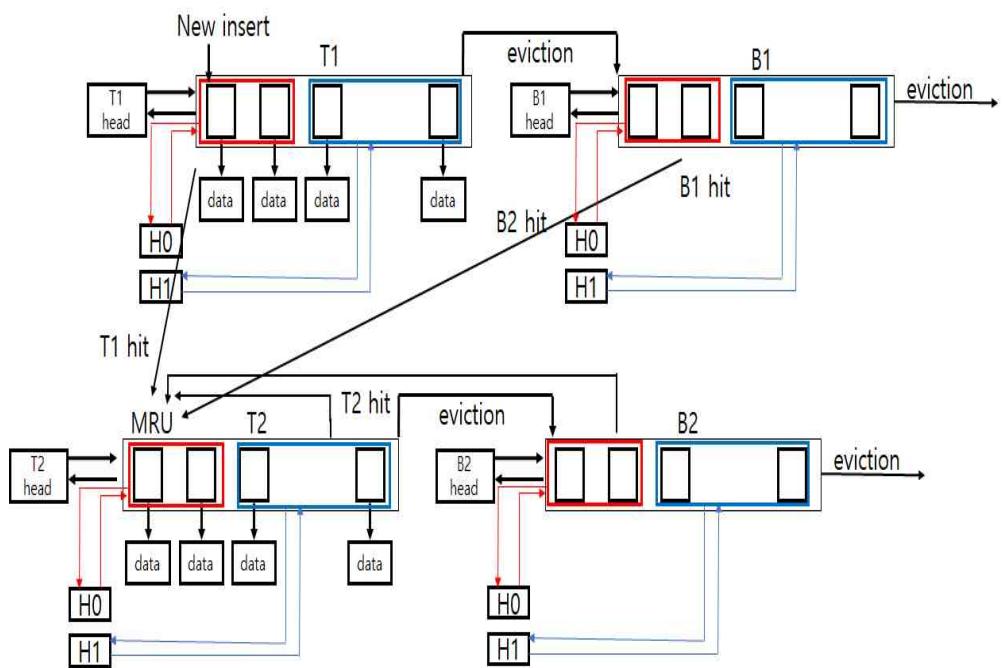
그림 2 2Q 알고리즘

2.3 ARC(Adaptive Replacement Cache)

ARC(Adaptive Replacement Cache) 기법은 기본적으로 T1과 T2라는 두 개의 LRU 리스트를 사용한다[3]. 처음 참조된 블록은 T1 리스트에 삽입되며, T1 리스트에 유지될 때 한 번 더 참조되면 해당 블록은 T2 리스트로 이동한다. ARC는 또한 B1과 B2라는 두 개의 리스트를 더 가지고 있으며, 이 리스트들에는 고스트 버퍼가 연결되어 있다. T1 리스트에 있는 데이터 블록이 제거되면 해당 블록의 메타 데이터를 고스트 버퍼에 넣고 이를 B1 리스트에 유지한다. T2 리스트에 있는 데이터 블록이 제거되면 해당 블록의 고스트 버퍼가 B2 리스트에 삽입된다. ARC의 가장 큰 특징은 4개의 리스트 크기를 자동으로 조절한다는 것이다. 리스트 크기를 동적으로 조절하는 ARC의 동작은 다음과 같다.

1. B1에서 히트가 발생할 경우, T2의 한 공간을 T1으로 이동(T2의 이동된 공간에 있는 데이터를 B2로 이동)
2. B2에서 히트가 발생할 경우, T1의 한 공간을 T2로 이동(T1의 이동된 공간에 있는 데이터를 B1으로 이동)

ARC 기법은 4개의 리스트 이외에도 각 리스트에 대응되는 해시 테이블을 유지한다. 이러한 해시 테이블에 어떤 블록이 해당 리스트에 존재하는지 빠르게 검색하기 위해 사용된다. 따라서 ARC 기법은 4개의 리스트와 4개의 해시 테이블을 유지해야 하므로 하나의 리스트와 하나의 해시 테이블을 유지하는 LRU 기법과 비교하여 메모리 오버헤드가 크다. 또 다수의 리스트를 관리하고 해시 테이블을 검색해야 하므로 컴퓨팅 오버헤드도 크다.



Hit in b1 → increase size of t1, drop entry from t2 to b2
 Hit in b2 → increase size of t2, drop entry from t1 to b1

그림 3 ARC 알고리즘

2.4 기타 교체 정책

Cachelib은 Facebook에서 개발된 고성능 캐시이며 대규모 웹서비스에서 데이터 캐싱을 위해서 설계되었다[4]. Cachelib은 모듈화 아키텍처를 가지고 있으며 Facebook의 서비스를 제공하기 위해 일관성과 안정성에 초점을 두고 개발되었다. 클라이언트가 요구한 데이터는 저장장치로부터 메모리로 읽어 들여 제공한다. DRAM에 저장된 데이터의 접근 횟수가 일정 횟수를 만족하지 못하면 이 데이터를 플래시 메모리 저장장치에 캐싱한다. 그 후 플래시에서는 큰 크기의 데이터는 LOC에서 B+트리로 작은 크기의 데이터는 SOC에서 블룸 필터로 관리를 한다. 이러한 고차원적인 캐시 정책은 그 내부를 보면 전통적인 교체 기법 LRU, 2Q, FIFO 그리고 ARC를 응용하고 있다.

슬랩, DRAM에서의 작동원리인 슬랩은 비슷한 크기로 나누어서 데이터를 저장한다. 따라서 크기에 따라서 데이터를 클래스로 나누고 각 슬랩 클래스마다 접근 횟수에 따른 방출 정책으로 관리를 하게 된다.

Kangaroo는 참조된 데이터를 로그 형태로 유지하는 로그 구조 캐시인 Klog와 세트 연관 캐시 구조인 Kset을 사용한다[5]. Klog 구조에서는 파티션으로 분리를 해서 관리하며 그 파티션 안에서도 해당 데이터를 해싱을 통해서 분리해 테이블로 관리한다. 이 점을 통해서 로그의 단점인 DRAM을 많이 사용한다는 점을 파티션을 통해서 해결할 수 있다. 그리고 해당 데이터가 교체되면 다시 이 데이터를 플래시 메모리 캐시로 보내는데, 해당 플래시 메모리 캐시는 세트 연관 방식으로 데이터 블록을 관리한다. 그러나 Kangaroo는 세트 연관 방식의 정확한 설명이 없으며, 플래

시 메모리에 데이터를 관리하는 방식으로 세트 연관 방식과 블룸 필터를 사용한다고 언급하고 있다. 이는 메모리 캐시 교체 기법으로 세트 연관 사상 방식을 사용하는 AP 기법과 다르며, 특히 그 정확한 메모리 및 컴퓨팅 오버헤드를 측정하고 비교하지 않았다는 점에서 본 논문의 연구 방향과 다르다.

3. 세트 연관 사상 캐싱 기법 설계

3.1 헤더 구조체 설계

기존 LRU, 2Q 그리고 ARC와 같은 알고리즘은 리스트에 데이터 블록을 연결하며, 또한 리스트에 대응되는 해시 테이블에도 데이터 블록을 연결해야 한다. 이러한 동작을 원활하기 위하여 다음과 같은 헤더 구조체를 사용한다

```
struct HEADER_T {  
    uint64    blkno;  
    struct HEADER_T *prev, *next;  
    struct HEADER_T *hash_prev, *hash_next;  
    char     *data_ptr;  
}
```

위 HEADER_T 자료 구조에서 blkno는 데이터 블록의 ID를 가지며, prev와 next는 HEADER_T 자료 구조를 리스트에 연결하기 위해 사용된다. 그리고 hash_prev와 hash_next는 해시 리스트에 연결하기 위해 사용되며, data_ptr은 실제 데이터가 저장된 블록을 가리킨다. 캐시에 존재하는 데이터 블록의 경우 data_ptr이 실제 데이터 블록을 가리키지만, 고스트 버퍼의 경우 data_ptr은 NULL을 가리키도록 설정하고 HEADER_T 자료구조를 그대로 사용한다. 64비트 시스템의 경우 위 자료구조는 48바이트를 차지한다.

AP 기법은 리스트와 해시 테이블을 사용하지 않는다. 따라서 AP 기법은 다음과 같은 헤더 구조체를 사용한다.

```
struct HEADER_T {  
    uint64    blkno;  
    uint64    ref_time;  
    char     *data_ptr;  
}
```

AP 기법은 리스트에 연결하기 위한 포인터들을 사용하지 않지만, 참조 시간을 저장하기 위한 ref_time 필드를 가지고 있다. 이 ref_time 필드는 같은 세트 내에서 가장 예전에 참조된 블록을 찾기 위한 용도, 즉 LRU 순서로 교체하기 위해 사용된다. 위 자료 구조는 64비트 시스템의 경우 24바이트를 차지한다.

3.2 AP 기법의 동작

AP 기법은 헤더 구조체들을 2차원 배열 형태로 관리한다. 즉 헤더 구조체들이 $H[0][0]$, $H[[0][1]$, .. $H[0][S-1]$, $H[1][0]$, $H[1][1]$... $H[1]$, $[S-1]$ 과 같이 2차원 배열 형태를 가지며, 2차원 배열에서 세로축은 세트 번호 그리고 가로축은 세트 내에서 위치를 가리킨다. 그리고 X 축으로 X개, 그리고 Y 축으로 Y개가 존재한다면 이 캐시에는 $X \times Y$ 의 데이터 블록이 존재하며, 이 개수가 캐시의 크기라고 할 수 있다.

$$i = \text{hashing}(\text{blkno}) (0 < i < n - 1)$$

해싱 함수는 다양한 함수가 존재하지만, AP 기법에서는 해당 X의 수를 이용하여 나머지 계산을 하여서 해시 계산을 한다. 데이터가 참조될 경우 Y개의 세트가 존재한다고 가정할 때 해당 블록 번호가 몇 번째 배열에 참조될 것인지는 해싱을 통해서 결정하게 된다.

AP 기법에서 데이터 블록은 해싱에 따라 특정 X array 위치에 존재할 수 있다. 이 점은 세트 연관 사상을 응용하고 있다. 작동원리를 설명하면 다음과 같다. 블록 번호가 참조되면 먼저 X 축에서의 위치를 해싱을 통해서 n이라는 값을 계산한다. 그럼 해당 블록 번호는 $H[n][0]$, $H[n][1]$ ~ $H[n][S-1]$ 에서만 존재할 수 있다. 그 후 순차적으로 S개의 세트의 n번째 X축 위치를 검색하여 블록 번호를 찾는다. 만약 해당 세트 중에 해당 블록 번호가 존재하면 캐시 히트가 발생한다. 히트가 발생하게 되면 해당 블록 번호의 ref_time을 현재 시작으로 설정 후 해당 블록을 반환한다. 히트가 발생하지 않을 경우는 두 가지의 경우가 존재하는 세트의 n번째 X축에 다른 블록 번호로 채워져 있는 경우와 그렇지 않은 경우이다. 모든

세트의 n번째 X축의 저장 공간들이 다른 값으로 가득 채워지면 세트의 n번째 X축 저장 공간에 존재하는 블록 중에서 참조 시간이 가장 오래된 블록을 새로 참조된 블록과 교체한다. 즉 해당 세트의 n번째 X축에서 블록 번호를 새로운 블록 번호로 바꾸고 새로 참조된 데이터 블록을 저장장치로 읽어온 후 data_PTA도 새로 참조된 블록의 데이터를 가르치도록 설정한다. 마지막으로 ref_time 변수로 현재 시각으로 설정한다. 그리고 세트의 n번째 X축에 모두 채워져 있지 않은 경우는 비어있는 세트의 n번째 X축을 찾아내어서 블록 번호를 새로 참조된 블록 번호로 바꾸고 data_PTA의 새로 참조된 블록 데이터를 가르치도록 설정하여 준다. 위 설명을 실제 데이터가 참조된 경우로 예를 들어서 아래에서 설명한다.

AP 기법에 대한 설명을 그림과 함께 예시를 들어보겠다. 먼저 그림4를 보면 X는 5, 세트 S의 Y는 4이며, 총 데이터를 저장할 수 있는 크기는 20개가 된다. 블록 5번이 참조된다고 가정하자. 블록 번호 5를 X의 총 개수인 5로 나머지 계산하면, n은 0이 된다. 여기서 총 3개의 경우의 수가 가능하다. 첫 번째는 그림5와 같이 4개의 데이터 블록을 캐싱하고 있는 X축 0에 블록 5가 존재하는 경우이다. 이 경우 블록 5의 ref_time을 global_time으로 갱신한 후 블록을 클라이언트에게 전달한다. X축 0에 블록 5가 존재하지 않으면 두 번째 또는 세 번째 경우가 된다. 두 번째 경우는 그림6과 같이 X축 0에 비어있는 공간이 있는 경우이며, 이 경우 블록 5를 저장장치로부터 읽어 비어있는 공간에 적재한다. 블록 5의 ref_time을 global_time으로 설정하고 해당 블록을 클라이언트에게 전달한다. 이 때 만약 세트가 무작위로 배정될 때 해당 세트에 다른 값이 존재하면 차례로 순환하면서 비어있는 세트를 찾아서 블록을 클라이언트에게 전달한다. 세 번째는 X축 0에 비어있는 공간이 없고 모두 다른 블록을 캐싱

하기 위해 사용 중인 경우이다. 그림7과 같이 이 경우 X축 0에 있는 4개의 블록 중 ref_time이 가장 오래된 블록을 검색한다. 그 결과 ref_time이 가장 오래된 50의 값을 가진 블록을 찾아내어서 캐시에서 제거하고 그 위치에 블록 5를 읽어 저장한다. 다음으로 블록 5의 ref_time을 global_time으로 설정하고 이 블록을 클라이언트에게 전달한다.

```
//Accessing blkno
global++ // 현재 시작
X=blkno % size of X
Y=find_blkno_in_array
If(Y == 0,1,2,3) { // blkno is hit
    array[X][Y]. ref_time=global
}
else { // blkno is miss
    if (no empty block in X) {
        Y=find blkno with smallest reference time in X array
        put blkno into array[X][Y]
    }
    else {
        Y = find empty in X array
        put blkno into array[X][Y]
    }
}
```

알고리즘 1 AP 알고리즘

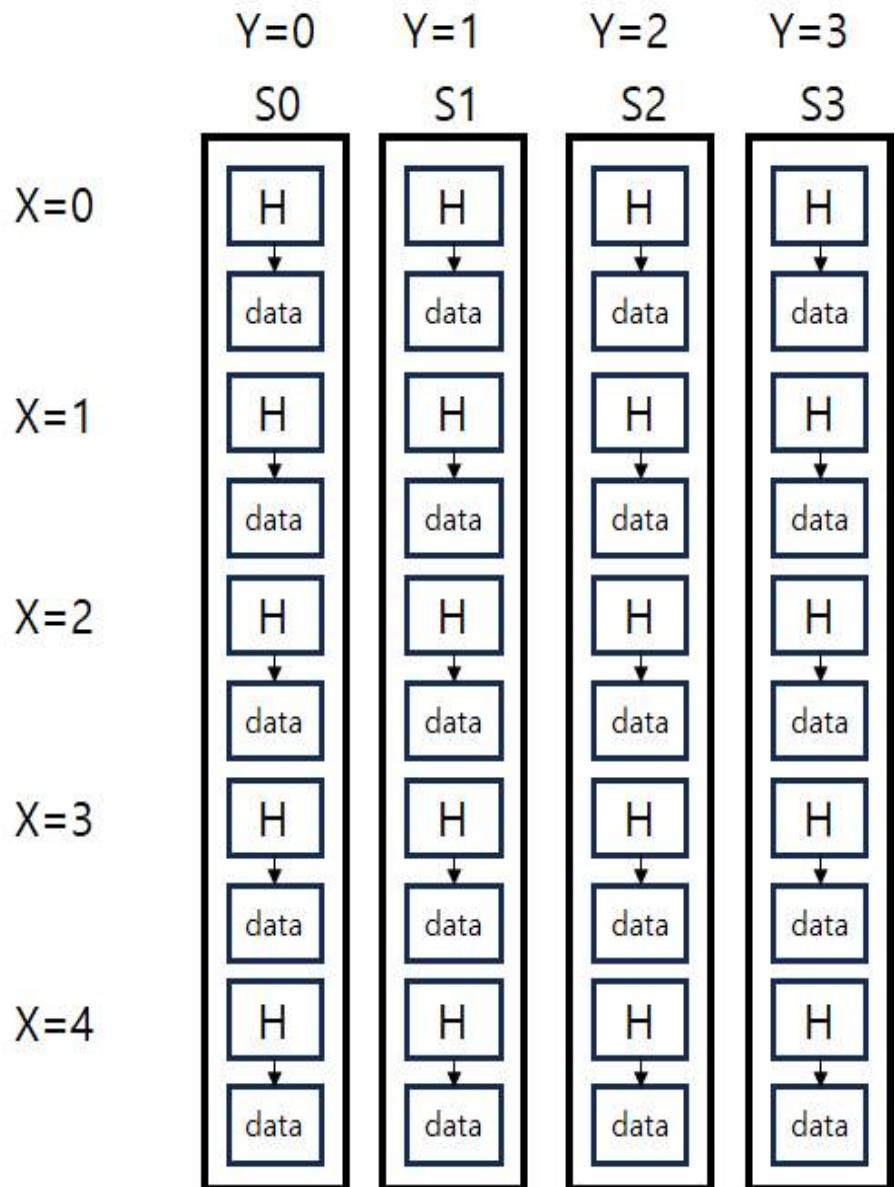


그림 4 AP 기법

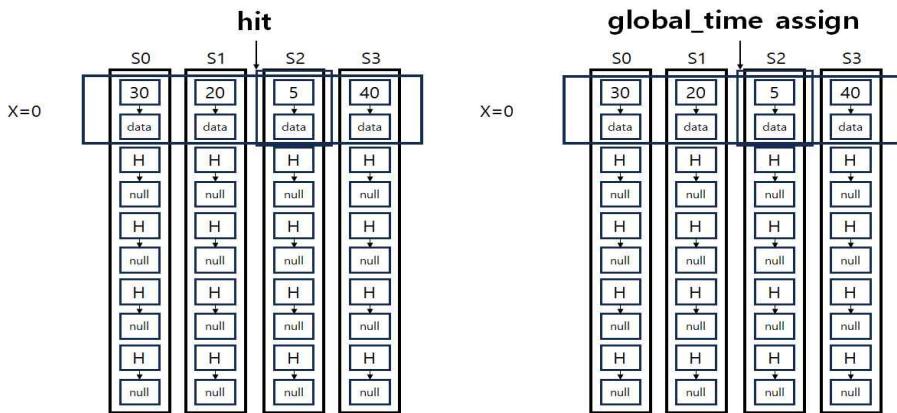


그림 5 AP HIT인 경우

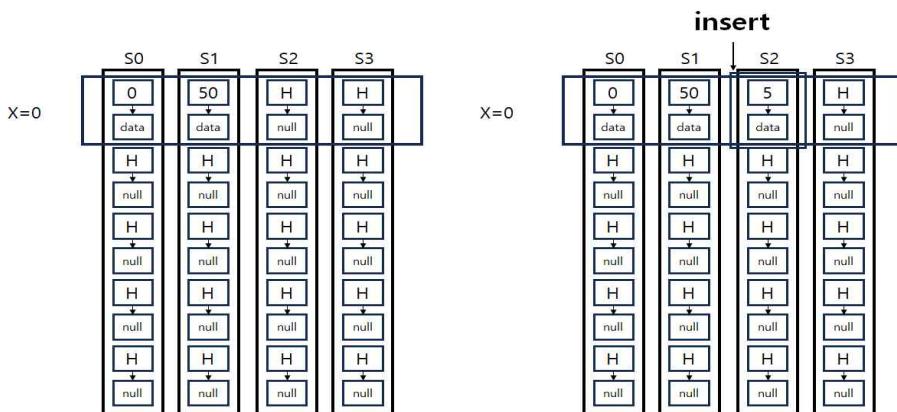


그림 6 AP 모두 채워져 있지 않은 경우

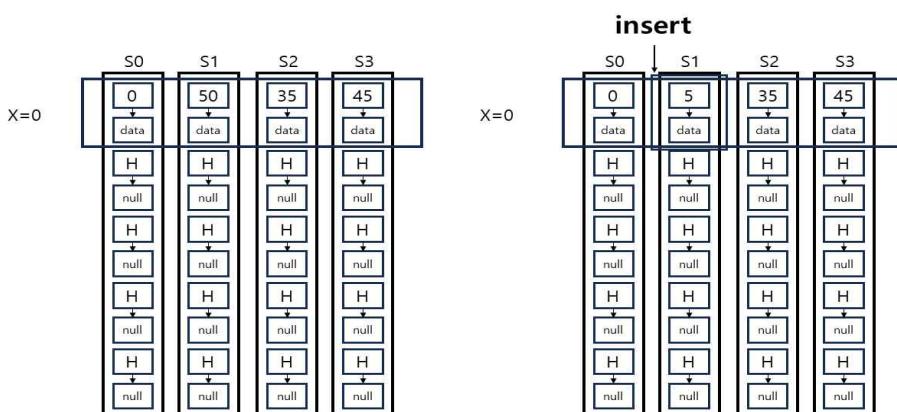


그림 7 AP 모두 채워져 있는 경우

4 분석 및 비교 실험 결과

4.1 데이터 블록 메모리 사용량 비교

LRU, 2Q 그리고 ARC 같은 경우 한 노드를 유지하기 위해서는 3장에서 설명한 것과 같은 구조체 형식이 필요하다. 데이터 블록 크기가 4kb인 경우 64비트 시스템에서 헤더와 데이터 블록을 모두 더한 경우 크기는 총 4144byte이며, 고스트 버퍼의 경우에는 48byte를 사용한다. 이에 반해 AP 캐싱 기법은 하나의 데이터를 저장하기 위해서는 4116byte를 사용하게 된다. 해시 테이블 크기를 캐싱 가능한 데이터 블록의 개수와 같이 설정하였을 때 기법별로 메모리 사용량을 계산하여 표 1에 표시하였다. 또한, 각 기법은 고스트 버퍼를 제외하고 실제 캐싱 가능한 데이터 블록의 개수를 계산하여 차트 1에 표시하였다.

	LRU	2Q AM	2Q A1 in	2Q A1 out	ARC T	ARC B	AP
1GB	256106	179298	56350	894784	244922	244922	260870
2GB	512212	358596	108902	1789569	513781	513781	521740
4GB	1024425	717193	225403	3579139	979691	979691	1043480
8GB	2048851	1434387	450807	7158278	1959383	1959383	2086961
16GB	4098201	2868775	901615	14316557	3918766	3918766	4173923
32GB	8196403	5737551	1803231	28633115	7837532	7837532	8347847

표 1 알고리즘 데이터 노드 개수 비교표

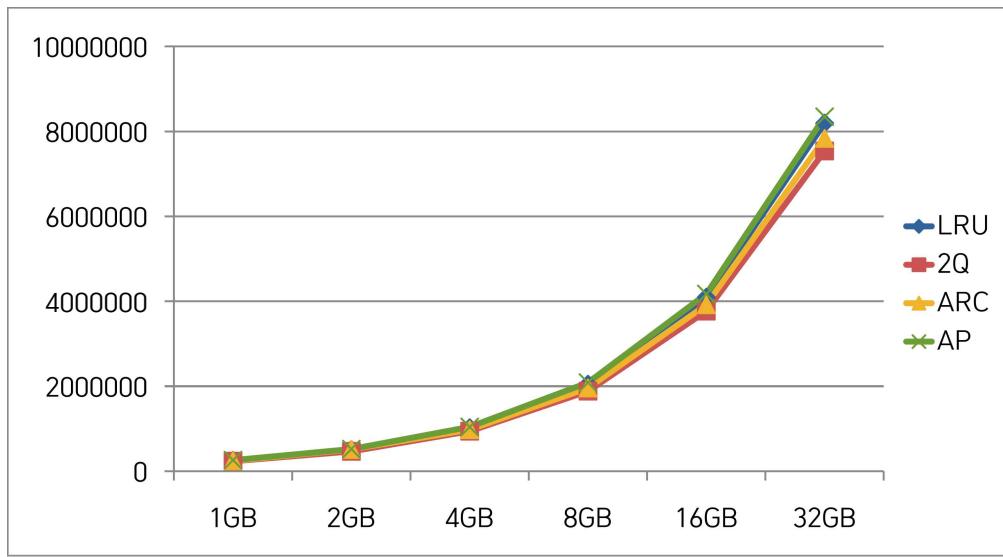


차트 1 알고리즘 데이터 노드 개수 비교 차트

LRU	-1.8%
2Q	-10.7%
ARC	-6.5%

표 2 AP 저장 가능 노드 비교 비율

캐시 크기가 1GB일 때 LRU와 AP 기법에서 캐싱 가능한 블록의 개수는 약 천 개 정도 차이가 발생한다. 그러나 캐시의 크기가 32GB가 되면 그 차이는 약 15만 개로 늘어나게 된다. 이를 비율로 계산을 하게 되면 표와 같이 LRU는 -1.8%, 2Q는 -10.7% 그리고 ARC는 -6.5%가 나오게 된다. 따라서 저장 용량이 늘어나게 될수록 해당 비율만큼 저장 공간의 차이가 발생하게 될 것이다.

4.2 참조 스텝 비교 분석

LRU, 2Q ARC 그리고 제안한 AP 기법의 실행 단계를 비교해보다.

```
//Accessing data X
if X in LRU list {
    move X mru position
}
else//miss {
    if LRU list is full {
        remove LRU block from LRU list
    }
    add X to LRU list
}
```

알고리즘 2 LRU 알고리즘

```
//Accessing a data X
if X in Am {
    move X to MRU position in Am list
}
else if X in A1out list {
    if Am is full {
        remove LRU block from Am list
    }
    insert X into MRU position of Am list
}
else if X in A1in      // do nothing
else //first access{
    if A1in is full {
        move LRU block of A1in to A1out's MRU position
    }
    add X to MRU position of A1in list
}
```

알고리즘 3 2Q 알고리즘

```

//Accessing a data X
if X in T1 or T2 {
    move X to MRU position in T2
}
else if X in B1 {
    Adaptation: p=min(p+ $\alpha$ ,c) where  $\alpha = \begin{cases} 1 & \text{if } |B1| \geq |B2| \\ |B2|/|B1| & \text{otherwise} \end{cases}$ 
    Replace(x,p) move X from B1 to MRU in T2
}
else if X in B2 {
    Adaptation: p=max(p- $\alpha$ ,0) where  $\alpha = \begin{cases} 1 & \text{if } |B1| \geq |B2| \\ |B2|/|B1| & \text{otherwise} \end{cases}$ 
    Replace(x,p) move X from B2 to MRU in T2
}
else //X is first access {
    Case A T1  $\cup$  B1{
        if T1's size < c {
            remove LRU block in B1
            REPLACE(x,p)
        }
        else {
            B1 is empty. remove LRU block in T1
        }
    }
    Case B T1  $\cup$  B1 < c {
        if (T1's size+T2's size+B2's size+B1's size)  $\geq$  c {
            remove LRU block in B2
        }
        if(T1's size+T2's size+B1's size+B2's size=2c) {
            Replace(x,p)
        }
    }
    insert X into MRU position in T1
}
Subroutine Replace(x,p)
    if (T1 is not empty)&(T1 >p)^((X in B2)&T1=p) {
        remove LRU in T1, and move LRU(in T1) to MRU in B1
    }
    else {
        remove LRU in T2, and move LRU(in T1) to MRU in B2
    }
}

```

위 알고리즘을 비교해보면 LRU보다 2Q 그리고 ARC는 더 많은 리스트를 사용하는 것을 관찰할 수 있다. 2Q에서는 처음 참조된 블록은 A1in 리스트에, 그리고 빈번히 참조되는 블록은 Am 리스트에 이동하며, 이를 위하여 여러 단계를 거친다. 또한, 앞에서 설명한 것처럼 각 Am, A1in 그리고 A1out의 리스트가 변경될 때 해시 테이블도 함께 수정하기 때문에 컴퓨팅 오버헤드가 심하게 발생한다. ARC도 여러 리스트가 존재하고 2Q와 비슷한 실행 단계를 거치며 추가로 캐시의 치수를 동적으로 조절하는 기능도 가지고 있다. 그에 비해 LRU는 하나의 LRU 리스트만 유지하며, 블록이 참조될 때 LRU 리스트에서 MRU 위치로 옮기는 기능만 가지고 있다. 그렇지만 LRU 리스트에 데이터를 삽입하거나 삭제할 때 해시 테이블도 함께 변경하는 컴퓨팅 오버헤드는 여전히 존재한다. 각 기법은 메모리 사용량이 다를 뿐만 아니라 위에서 설명한 이유로 컴퓨팅 오버헤드도 다르며 그 실험 결과는 4.5절에서 설명한다.

4.3 실험 환경

본 논문의 실험에서는 zipf 법칙에 따라 블록 참조 패턴을 생성하는 zipf 워크로드를 사용하였다. 현실 세계의 많은 데이터 패턴은 zipf 분포와 유사한 형태를 가지고 있다고 알려져 있다[7]. 특히 웹에서 사용되는 데이터는 zipf 분포와 유사한 패턴을 보이는데 웹 데이터가 점점 커지는 있는 점을 고려하여서 zipf 분포로 워크로드를 만들어서 실험을 진행하게 되었다. zipf 법칙에서는 zipf 분포 법칙 지수를 통해서 데이터의 분포율을 조절할 수 있다[8, 19]. zipf 분포 법칙 지수가 자가 큰 경우 데이터의 참조 지역성(locality)이 커지며 zipf 분포 법칙 지수가 0과 가까울수록 데이터의 참조 지역성(locality)이 작아진다. 실험에서는 zipf 분포 법칙 지수는 0.3, 0.75 그리고 1.0으로 세팅을 다. 실험은 블록 데이터의 크기를 4kb로 가정하고 진행되었다. 표 2, 3, 4, 5 그리고 6은 워크로드의 특성과 실험에 사용된 컴퓨터의 사양을 보여준다.

워크로드 0 특성	
zifp 분포 법칙 지수	0.3
블록 번호 범위	0~ 1.5억
총 워크로드 개수	1억 개

표 3 실험에 사용된 워크로드 0 특성

워크로드 1 특성	
zifp 분포 법칙 지수	0.75
블록 번호 범위	0~ 1.5억
총 워크로드 개수	1억 개

표 4 실험에 사용된 워크로드 1 특성

워크로드 2 특성	
zifp 분포 법칙 지수	1.0
블록 번호 범위	0~ 1.5억
총 워크로드 개수	1억 개

표 5 실험에 사용된 워크로드 2 특성

	제품명	사양
OS	Window 10 Pro	19045.3448
CPU	Intel(R) Core(TM) i5-6600K	3.50GHz
RAM	8GB*2	16GB
Storage	SSD 750 EVO	256GB

표 6 실험용 컴퓨터 사양

4.4 세트 수에 따른 참조 시간 비교 및 분석

AP 기법은 세트의 수에 따라서 데이터가 저장되는 방식이 달라지며 이에 따라 캐시 히트율도 달라지며 참조에 걸리는 시간 또한 달라진다. 따라서 AP 기법에서 세트 수에 따른 성능을 측정하였다. 이 실험에는 워크로드1(zipf 분포 법칙 지수 0.75)을 사용되었다. 참조 시간을 분석할 때는 하나의 블록이 캐시에서 참조되는 시간을 측정하였다. 캐시 크기가 100000이라고 가정할 때 세트 수에 따라 하나의 세트에 할당되는 데이터 블록의 개수는 아래 표와 같다.

세트 개수	세트에 할당되는 데이터 블록
1	100000
2	50000
4	25000
5	20000
10	10000
20	5000
50	2000

표 7 세트별 할당공간

세트 수가 많아지게 되면 제안된 알고리즘은 지정된 세트를 다 검사해야 하므로 컴퓨팅 오버헤드가 증가하게 된다. 하지만 세트 개수가 작으면 같은 세트에 저장되어어야 하는 블록들끼리 경쟁이 심해져 캐시 히트율이 떨어질 수 있다. 이처럼 세트 수에 따라 컴퓨팅 오버헤드와 캐시 히트율 사이에 트레이드 오프가 발생하게 된다.

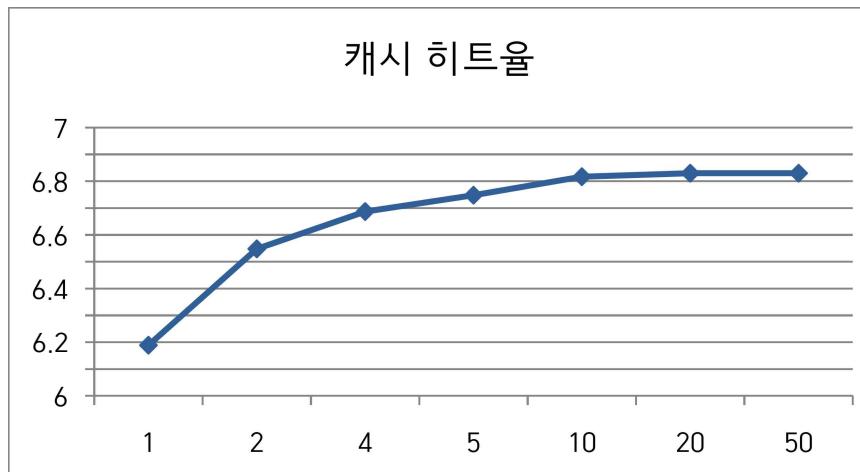


차트 2 세트별 히트율

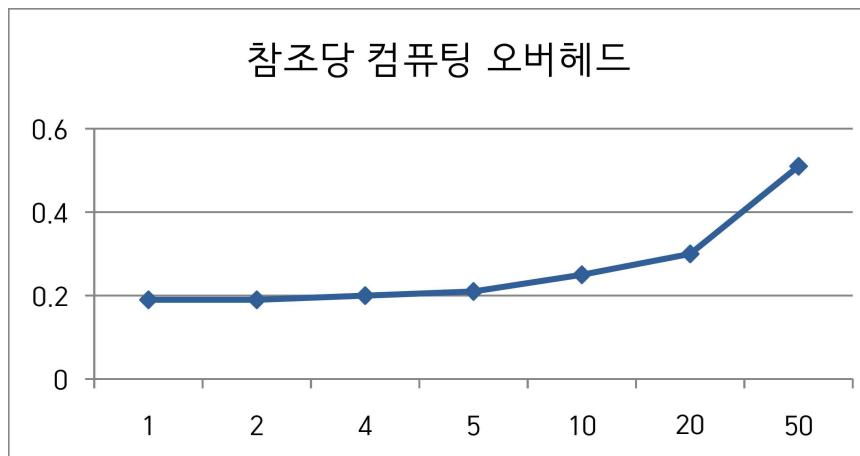


차트 3 세트별 컴퓨팅 오버헤드

위의 실험 결과와 같이 세트 개수가 커지면 해시 함수에 의해 우연히 같은 번호로 사상된 블록끼리 경쟁이 작아져 캐시 히트율이 증가하는 것을 관찰할 수 있었다. 하지만 세트 수가 4를 넘어가게 되면 그 증가율이 현저히 감소한다. 그와 동시에 세트 수가 5 이상이면 데이터를 참조하는 데 필요한 컴퓨팅 오버헤드 즉 참조 시간이 급격히 증가하기 시작한다. 이러한 실험 결과를 바탕으로 AP 기법을 다른 기법과 비교할 때 세트 개수를 4로 설정하여 실험하였다.

4.5 각 캐시 교체 기법들의 성능 비교

Zipf 워크로드 트레이스를 이용하여 LRU, 2Q, ARC 기법과 AP 기법의 성능을 비교하였다. AP 기법의 경우 세트 수를 앞에서 설명한 바와 같이 4로 설정하였다. 표 5에서 각 기법의 캐시 히트율과 메모리 오버헤드를 비교하였으며, 차트 4에서 캐시 크기별 히트율을 비교하였다.

		LRU	2Q	ARC	AP
1GB	히트율	0.208%	0.280%	0.255%	0.213%
	컴퓨팅 오버헤드	0.31 μ s	0.66 μ s	0.56 μ s	0.24 μ s
2GB	히트율	0.416%	0.517%	0.519%	0.425%
	컴퓨팅 오버헤드	0.31 μ s	0.7 μ s	0.6 μ s	0.26 μ s
4GB	히트율	0.828%	0.956%	0.961%	0.844%
	컴퓨팅 오버헤드	0.34 μ s	0.75 μ s	0.64 μ s	0.29 μ s
8GB	히트율	1.643%	1.732%	1.843%	1.671%
	컴퓨팅 오버헤드	0.35 μ s	0.81 μ s	0.71 μ s	0.29 μ s
16GB	히트율	3.232%	3.066%	3.499%	3.274%
	컴퓨팅 오버헤드	0.38 μ s	0.86 μ s	0.88 μ s	0.32 μ s
32GB	히트율	6.262%	5.207%	6.520%	6.305%
	컴퓨팅 오버헤드	0.41 μ s	0.95 μ s	0.81 μ s	0.33 μ s

표 8 워크로드0의 캐시 히트율과 메모리 오버헤드 실험 결과 비교

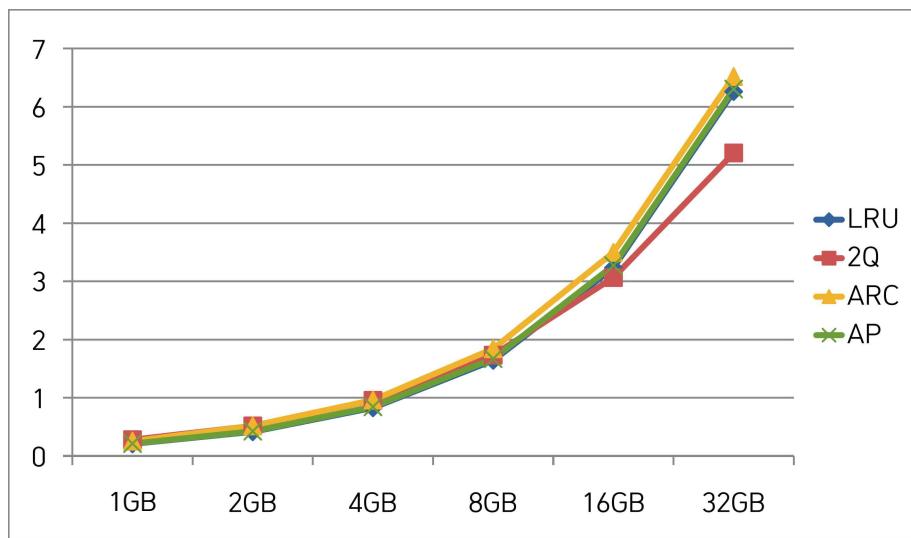


차트 4 워크로드0 알고리즘 캐시 크기별 캐시 히트율 비교

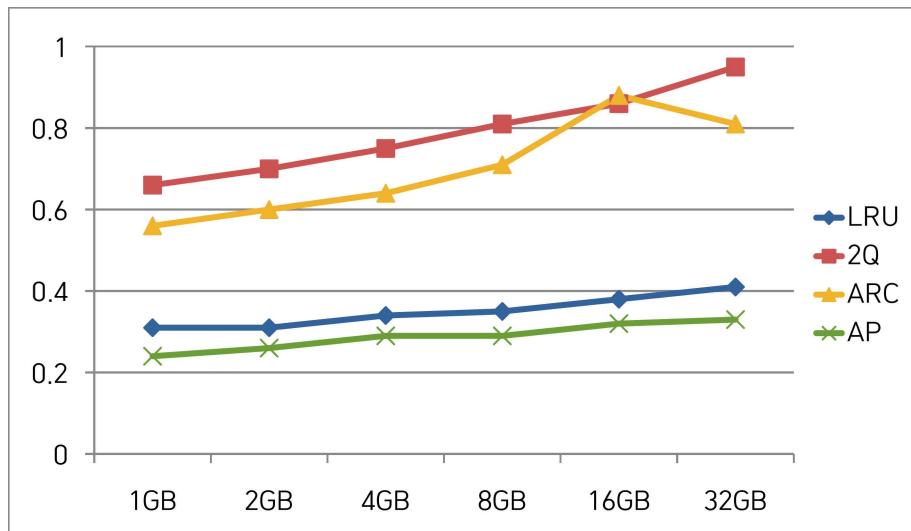


차트 5 워크로드0 알고리즘 캐시 크기별 메모리 오버헤드 비교

		LRU	2Q	ARC	AP
1GB	히트율	9.736%	15.599%	17.264%	9.6064%
	컴퓨팅 오버헤드	0.28 μ s	0.64 μ s	0.52 μ s	0.22 μ s
2GB	히트율	12.547%	18.426%	20.481%	12.497%
	컴퓨팅 오버헤드	0.3 μ s	0.68 μ s	0.58 μ s	0.24 μ s
4GB	히트율	16.128%	21.336%	23.666%	16.078%
	컴퓨팅 오버헤드	0.32 μ s	0.71 μ s	0.59 μ s	0.26 μ s
8GB	히트율	20.666%	25.351%	27.519%	20.632%
	컴퓨팅 오버헤드	0.33 μ s	0.73 μ s	0.65 μ s	0.26 μ s
16GB	히트율	26.389%	29.978%	31.877%	26.372%
	컴퓨팅 오버헤드	0.36 μ s	0.75 μ s	0.73 μ s	0.27 μ s
32GB	히트율	33.464%	33.879%	37.032%	33.401%
	컴퓨팅 오버헤드	0.4 μ s	0.72 μ s	0.72 μ s	0.28 μ s

표 9 워크로드1의 캐시 히트율과 메모리 오버헤드 실험 결과 비교

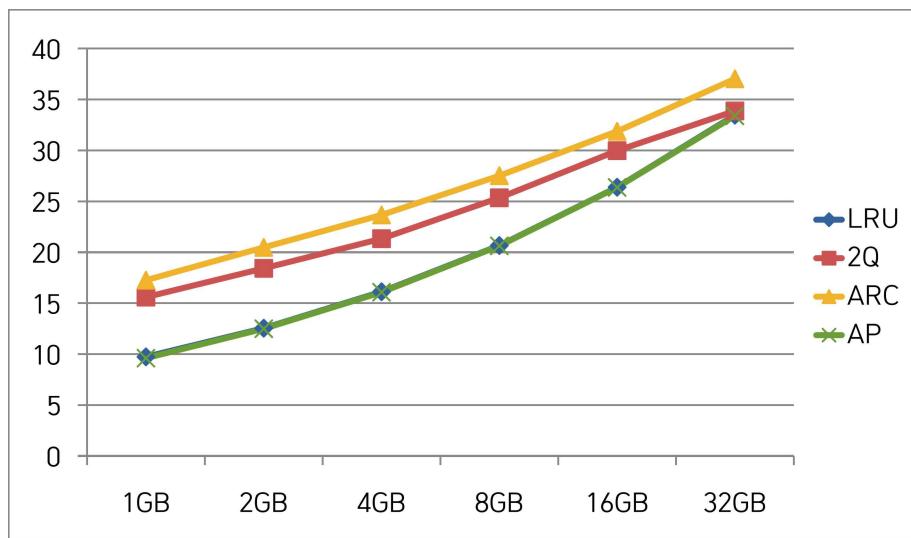


차트 6 워크로드1 알고리즘 캐시 크기별 캐시 히트율 비교

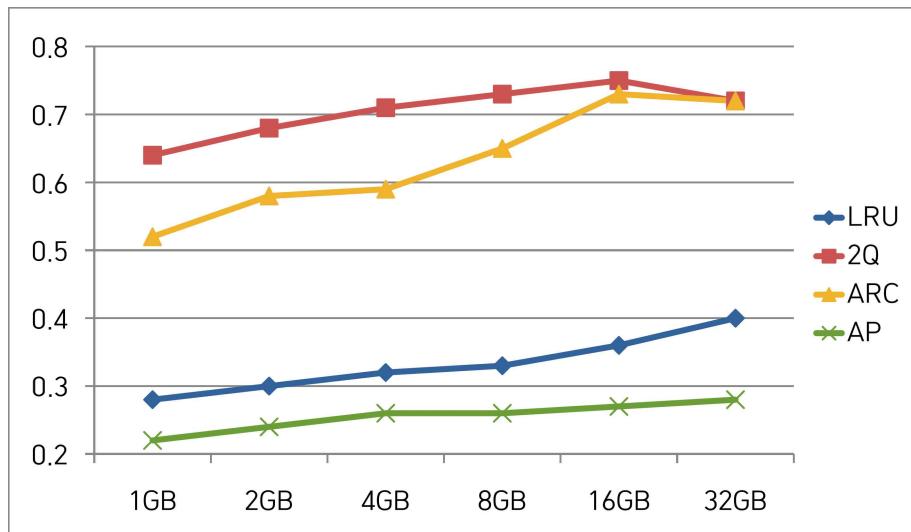


차트 7 워크로드1 알고리즘 캐시 크기별 메모리 오버헤드 비교

		LRU	2Q	ARC	AP
1GB	히트율	58.784%	63.046%	65.304%	58.841%
	컴퓨팅 오버헤드	0.28 μ s	0.49 μ s	0.47 μ s	0.21 μ s
2GB	히트율	62.790%	54.581%	68.718%	62.806%
	컴퓨팅 오버헤드	0.3 μ s	0.53 μ s	0.49 μ s	0.21 μ s
4GB	히트율	66.807%	60.157%	71.471%	66.845%
	컴퓨팅 오버헤드	0.3 μ s	0.57 μ s	0.53 μ s	0.22 μ s
8GB	히트율	70.799%	66.196%	74.067%	70.826%
	컴퓨팅 오버헤드	0.4 μ s	0.62 μ s	0.55 μ s	0.22 μ s
16GB	히트율	74.650%	74.846%	76.323%	74.619%
	컴퓨팅 오버헤드	0.37 μ s	0.49 μ s	0.54 μ s	0.23 μ s
32GB	히트율	78.060%	75.777%	78.473%	77.880%
	컴퓨팅 오버헤드	0.35 μ s	0.46 μ s	0.5 μ s	0.24 μ s

표 10 워크로드2의 캐시 히트율과 메모리 오버헤드 실험 결과 비교

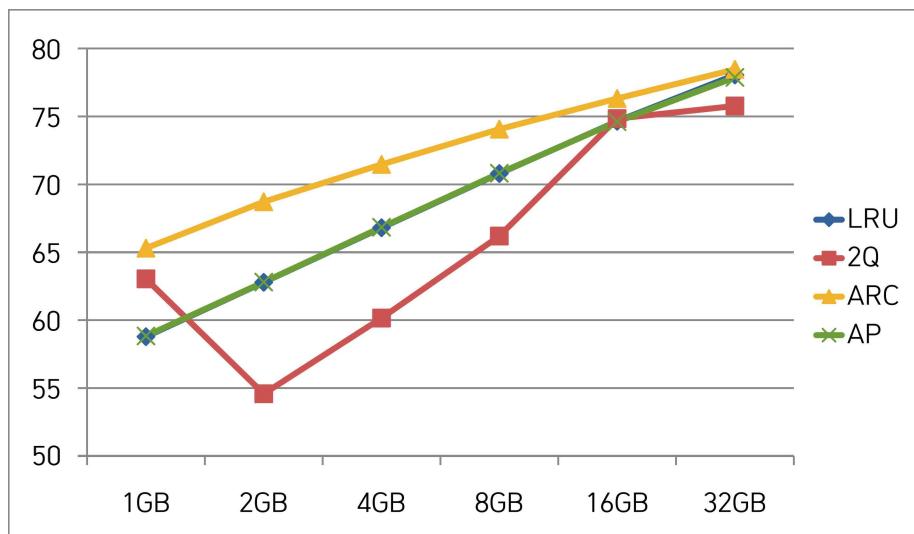


차트 8 워크로드2 알고리즘 캐시 크기별 캐시 히트율 비교

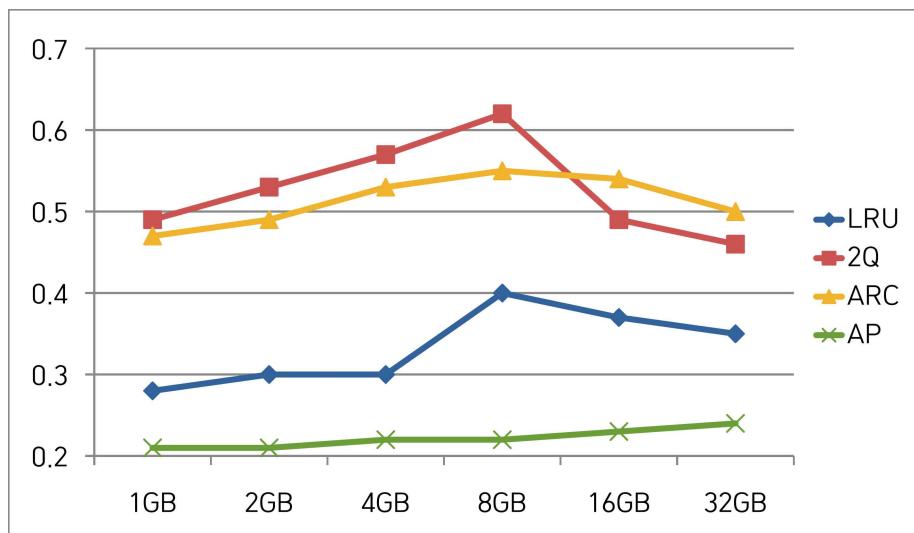


차트 9 워크로드2 알고리즘 캐시 크기별 메모리 오버헤드 비교

위의 차트와 표의 데이터를 보면 모든 워크로드에서 캐시 크기가 1GB일 때는 AP 캐시의 히트율이 2Q와 ARC 보다 떨어지는 것을 관찰할 수 있다. 위의 표처럼 LRU의 경우 1.8%, 2Q는 10.7% 그리고 ARC는 6.5% 저장 가능한 노드의 개수가 더 많다. 따라서 위의 히트율의 차트와 표를 보면 1GB일 경우에는 저장 가능한 노드의 개수가 차이가 적어서 히트율의 차이 떨어지는 것을 볼 수 있지만, 캐시의 크기가 증가할수록 그 차이는 점점 감소하는 것을 볼 수 있다. 즉, 캐시의 크기가 2, 4, 8, 16, 32GB로 증가하면서 성능이 가장 좋게 나오는 ARC와 캐시 히트율 격차가 좁혀지는 것을 볼 수 있다. 그리고 캐시의 크기가 더 커지면 그 격차는 더 줄어들 것으로 예상한다. 추가로 워크로드 2 실험결과 중 2Q의 히트율의 경우 2GB일 때 갑자기 감소하는 것을 관찰할수있는데 이는 아마 캐시의 크기가 증가하면서 히트율이 높은 데이터가 먼저 방출되는 상황이 발생하였고 2Q 캐시 알고리즘이 해당 워크로드에 약한 특성을 보이기 때문에 이러한 결과가 나온 것으로 추측된다. 워크로드0과 1의 실험결과에서 ARC의 캐시 히트율이 더 높은 이유는 Zipf워크로드가 각 블록에 고정된 참조 확률을 부여하는 정적인 워크로드이며, 두 워크로드의 분포 법칙 지수 분포도가 낮은 편에 속하여서 ARC는 높은 참조 확률을 보이는 블록을 T2에, 그렇지 않은 블록은 T1에 저장하는 특성이 있어 정적인 워크로드에 잘 동작하는 특성 때문이다.

AP 캐시의 참조 시간과 LRU, 2Q ARC의 참조 시간을 비교하면, 캐시의 크기가 커져도 AP 기법의 참조 시간이 증가하는 폭이 더 작은 것을 볼 수 있다. 이러한 특성은 LRU, 2Q, ARC가 다수의 리스트와 해시 테이블을 사용하고 캐시의 크기가 커질수록 더 큰 리스트와 해시 테이블을 관리해야 하므로 관리 오버헤드가 더욱 증가하지만, AP 기법은 단순히 해싱

을 통해 특정 세트에 바로 접근하기 때문에 오버헤드가 작아서 생긴 결과이다. 그리고 워크로드 2에서 LRU, 2Q 그리고 ARC의 오버헤드가 8GB를 지나게 되면 감소하는 특성을 보이게 되는데 이는 캐시 히트가 많이 발생함에 따라서 히트일 때 미스가 발생하는 경우보다 캐시 알고리즘이 하는 일이 적어서 오버헤드가 감소하는 것이다. 위의 워크로드 0, 1, 2의 오버헤드 실험결과에 따라서 AP기법은 다른 캐시 알고리즘보다 확실히 컴퓨팅 오버헤드는 캐시 크기에 상관없이 적은 것을 볼 수 있다.

4.6 AP 캐싱 세트에 따른 성능

앞에서 AP 기법이 세트 수에 따라서 참조 시간과 캐시 히트율 사이에 트레이드 오프가 존재함을 설명하였다. 이를 더 정확히 분석하기 위해 AP 기법에서 세트 수를 1, 2, 4, 5, 그리고 10으로 설정하여 워크로드 1(zipf 분포 법칙 지수 0.75)로 실험한 결과를 표 6에 제시하였다.

		1	2	4	5	10
1GB	히트율	8.989%	9.481%	9.664%	9.674%	9.751%
	컴퓨팅 오버헤드	0.21 μ s	0.22 μ s	0.22 μ s	0.24 μ s	0.28 μ s
2GB	히트율	11.656%	12.235%	12.497%	12.523%	12.607%
	컴퓨팅 오버헤드	0.23 μ s	0.24 μ s	0.24 μ s	0.25 μ s	0.32 μ s
4GB	히트율	15.066%	15.774%	16.078%	16.125%	16.206%
	컴퓨팅 오버헤드	0.23 μ s	0.25 μ s	0.26 μ s	0.26 μ s	0.34 μ s
8GB	히트율	19.410%	20.277%	20.632%	20.706%	20.784%
	컴퓨팅 오버헤드	0.24 μ s	0.25 μ s	0.26 μ s	0.27 μ s	0.34 μ s
16GB	히트율	24.868%	25.936%	26.372%	26.434%	26.537%
	컴퓨팅 오버헤드	0.24 μ s	0.25 μ s	0.27 μ s	0.29 μ s	0.35 μ s
32GB	히트율	31.481%	32.844%	33.401%	33.483%	33.612%
	컴퓨팅 오버헤드	0.25 μ s	0.26 μ s	0.28 μ s	0.3 μ s	0.36 μ s

표 11 AP 기법에서 세트 수 변경 시 캐시 히트율 메모리 오버헤드 실험 결과

세트 수당 히트율

■ 세트 수 1 ■ 세트 수 2 ■ 세트 수 4 ■ 세트 수 5 ■ 세트 수 10

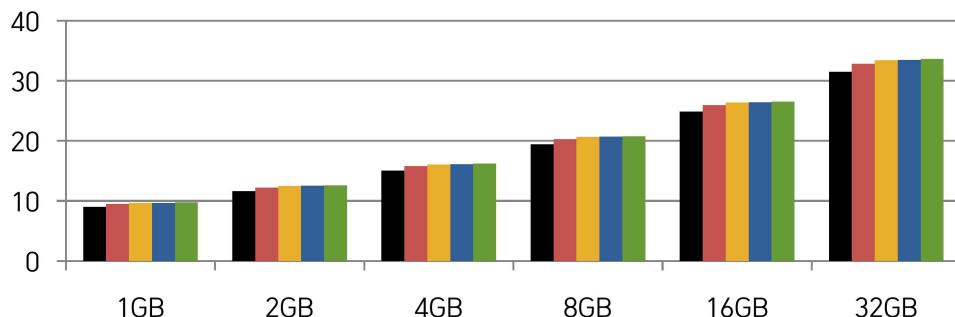


차트 10 AP 기법 세트 수당 히트율

세트 수당 컴퓨팅 오버헤드

■ 세트 수 1 ■ 세트 수 2 ■ 세트 수 4 ■ 세트 수 5 ■ 세트 수 10

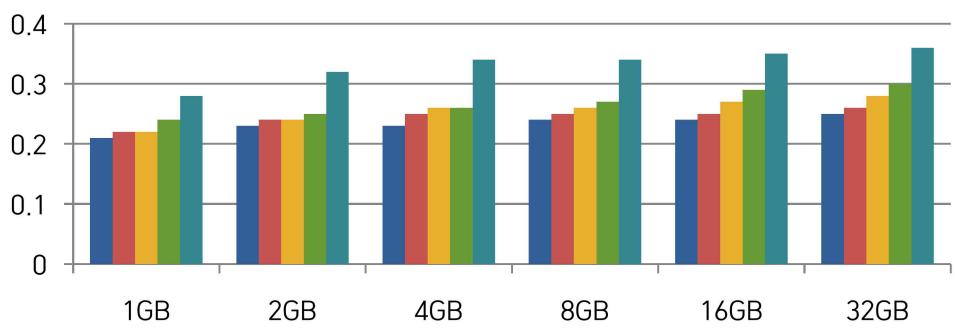


차트 11 AP 기법 세트 수당 컴퓨팅 오버헤드

위 표와 차트에 따르면, 세트 수가 작을수록 AP 기법의 컴퓨팅 오버헤드, 즉 참조 시간이 줄어들며, 세트 수를 1이나 2로 설정하면 캐시 크기가 32GB일 때 다른 세 가지 기법의 참조 시간보다 60% 빠른 참조 시간을 보여준다. 그러나 캐시 히트율이 더 중요하다면 AP 기법의 세트 수를 크게 하여 캐시 히트율을 높일 수 있다. 이런 실험 결과에 따라서 캐시를

히트율을 목적으로 사용할 경우 세트 수를 늘려서 사용할 수 있고 히트율 보다 오버헤드를 줄이는 것이 목적일 경우는 세트를 수를 줄여서 사용할 수 있다. 이처럼 AP 기법은 히트율이나 오버헤드를 조절이 가능한 복합 형 형태의 캐시 알고리즘이다.

5. 결론

본 논문에서는 기가바이트 단위로 캐시 크기가 커질 때 캐싱된 데이터를 효과적이며 빠르게 관리할 수 있는 AP 캐시 관리 기법을 제안하였다. AP 캐시 관리 기법은 세트 연관 사상 기법을 활용하여 데이터를 빠르게 저장하고 다시 읽어오는 방식이다. 본 논문에서는 AP 기법과 기존 LRU, 2Q 그리고 ARC를 비교하였으며, 캐시 크기에 따라 데이터를 참조할 때 걸리는 시간과 캐시 히트율을 비교하였다. 실험 결과를 보면 캐시의 규모가 커지면 AP 기법의 캐시 히트율은 다른 알고리즘과 비슷한 성능을 보였으며 컴퓨팅 오버헤드는 감소하는 결과를 확인할 수 있다. 워크로드의 특성에 따라서 세트 수를 조절하면서 캐시 관리자가 원하는 성능을 낼 수도 있을 것이며 대규모 데이터 캐시 관리에 활용될 수 있을 것이다.

참 고 문 헌

- [1] L. A. Belady, “A study of replacement algorithms for virtual storage computers,” in Proceedings of the IBM Sys. J., vol. 5, no. 2, pp. 78 - 101, 1966.
- [2] Theodore Johnson and Dennis Shasha, “2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm”, in Proceedings of the 20th VLDB Conference Santiago, Chile, 1994
- [3] Nimrod Megiddo and Dharmendra S. Modha IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120 “arc: a self-tuning, low overhead replacement cache”, in Proceedings of the FAST '03:2nd USENIX Conference on File and Storage Technologies, 2003
- [4] Benjamin Berg, Carnegie, Daniel S. Berger, Carnegie Mellon ,Sara McAllister and Isaac Grosof, Carnegie Mellon University; Sathya Gunasekar, Jimmy Lu, Michael Uhlar, and Jim Carrig, Facebook; Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger, “The Cache Lib Caching Engine: Design and Experiences at Scale”, in Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, 2020
- [5] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, , Gregory R. Ganger, “Kangaroo:

Caching Billions of Tiny Objects on Flash”, in Proceedings of the SOSP ’21, October 26 - 29, 2021, Virtual Event, Germany, 2021

- [6] Aoran Wu ,Brian Chang, Yuhao Zhangm, “Analysis of Cache Replacement policies”, in Proceedings of the github, 2020
- [7] Lada A. Adamic Bernardo A. Huberman, “Zipf’s law and the Internet” in Proceedings of the Glottometrics 3, 2002
- [8] Piantadosi ST, “Zipf’s word frequency law in natural language: a critical review and future directions” in Proceedings of the Psychon Bull Rev, 2014
- [9] W. D. MAURER T. G. LEWIS, “HASH TABLE METHODS” in Proceedings of the Computing Surveys, Vol. 7, No. I, 1975
- [10] Mats Nblund, “Universal Hash Functions & Hard Core Bits” in Proceedings of the Spnnger–Verlag Berlin Heidelberg, 1995
- [11] Park Jung–Wook, Kim Cheong–Ghil, Lee Jung–Hoon, and Kim Shin–Dug, “An energy efficient cache memory architecture for embedded systems” in Proceedings of the 2004 ACM symposium on Applied computing (SAC ’04). Association for Computing Machinery, New York, NY, USA, 884 - 890, 2004
- [12] Alberto Scolari, Filippo Sironi, Donatella Sciuto, Marco Domenico Santambrogio Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, “A Survey on Recent Hardware and Software–Level” in Proceedings of the UTC from IEEE Xplore, 2014
- [13] Yang, J., Yue, Y., and Rashmi, K. V, “A large scale analysis

of hundreds of in-memory cache clusters at Twitter” in Proceedings of the 20th ACM SIGOPS Operating Systems Review, 623–638, 2020

- [14] Berk Atikoglu Yuehai Xu Eitan Frachtenberg Song Jiang Mike Paleczny, “Workload Analysis of a Large-Scale Key-Value Store” in Proceedings of the ACM, 2012
- [15] 김아람, 이인환, “임베디드 응용을 위한 플래쉬 메모리와 하드디스크 파일 시스템의 성능 평가” in Proceedings of the 한국정보과학회 가을 학술발표문집, 2007
- [16] Kin Yeung Wong and Macao Polytechnic Institute, “Web Cache Replacement Policies: A Pragmatic Approach” in Proceedings of the IEEE Network, 2006
- [17] Bellare, M. and Micciancio, D, “A New Paradigm for Collision-Free Hashing: Incrementality at Reduced Cost” in Proceedings of the W. Fumy (Ed.): Advances in Cryptology – EUROCRYPT ’97, LNCS 1233, pp. 163–192, 1997
- [18] Damgård, I.B, “Collision Free Hash Functions and Public Key Signature Schemes” in Proceedings of the Chaum, D., Price, W.L. (eds) Advances in Cryptology – EUROCRYPT’ 87. EUROCRYPT, 1987
- [19] Zipf, G, “The Psychobiology of Language” in Proceedings of the London: Routledge, 1936
- [20] STEFAN PODLIPNIG AND LASZLO BOSZ “ORMENYI, “A Survey of Web Cache Replacement Strategies” in Proceedings of the ACM Computing Surveys, Vol. 35, No. 4, December,pp.

374 - 398, 2003

- [21] J. Gray and P. Shenoy, "Rules of thumb in data engineering," Proceedings of 16th International Conference on Data Engineering pp. 3–10, San Diego, CA, USA, 2000,

Abstract

In computer systems, the relative speeds of each device can be very different. For example, data access times for storage devices such as hard drives are much slower than main memory such as DRAM. Additionally, the speed of bringing data from the network is also generally slower than the access time of DRAM. Therefore, by using a portion of the main memory as a data cache space, reading data once and storing it in the cache, and then bringing it from the cache when the data is accessed again, the data access time can be reduced. Data caching techniques are widely used in many parts of the system for this purpose.

In order to store new data in the cache, existing data in the cache must be removed to make room for the new data. Therefore, there is a replacement technique that determines which data to remove. Traditionally used techniques include LRU, 2Q, and ARC techniques. And many techniques currently are those that are applied or combined with these traditional techniques. The goal of the cache replacement technique is to choose the data with the smallest possibility of being used in the future and remove it to increase the cache hit rate. Therefore, the performance of the cache replacement technique is mainly compared using the hit rate.

The cache management techniques that are widely used today are

applied to traditionally cache management techniques such as LRU, 2Q, and ARC cache algorithms. These traditionally cache management techniques use a double linked list to manage data, and these lists connect data using pointers. In addition, to quickly find whether data exists in the cache, and if it exists, where it is located, a hash table is used to search for data. However, these list and hash table methods occupy a lot of space. In addition, the management operations of searching the hash table and inserting and deleting data in the list increase the overall data access time, which acts as a factor that lowers performance.

The computing environment is constantly changing. Especially in recent changes, we can mention the reduction of data access time from data sources and the increase of memory in computer systems. In the case of storage devices, which are one of the classical data sources, flash memory storage devices such as SSD are widely used instead of hard drives. As a result, the input/output time required to bring data has decreased sharply. Another change is the increase of memory, and as a result, the size of the cache that stores the brought data is also very large.

As the speed of reading data from the device increases and the size of the cache increases, the efficiency of the cache replacement technique decreases in importance. In particular, as the size of the cache increases, the hit rate difference between cache replacement techniques

decreases. On the other hand, the management overhead for managing a large number of data that exist in a huge cache increases. In addition, as the time it takes to read data from the device decreases, the benefit that can be obtained by increasing the hit rate decreases. This paper compares cache replacement techniques with new criteria according to these changes in the computing environment, and also proposes the AP (Associative Pocket) technique as a replacement technique for large-scale caches. The AP technique is a replacement technique that is made by applying the set associative concept.

In this paper, we measure the overhead of LRU, 2Q, ARC, and AP techniques. In particular, we measure the memory overhead that each technique has to bear to manage data. Next, we measure the execution time of each cache replacement technique. This comparison is different from the traditional comparison method that only compares cache hit rate, and shows the trade-off between the overhead and hit rate of the cache replacement technique.

According to the comparison results, the AP technique shows that the hit rate is similar to traditional techniques while the overhead is very low in large-scale caches. Specifically, the AP technique does not use pointers to connect data, and does not use the hash table method used to search for data, so the operation overhead is small. And the AP technique does not use lists or hash tables, so it uses less memory space than other techniques, and as a result, it can cache more data.

And this difference in caching capacity is even greater as the size of the cache increases. The AP technique shows data access time about 60% faster than the representative cache replacement technique LRU, and in terms of hit rate, the performance of the replacement techniques is similar as the cache size increases.

Keywords: large scale cache, set associative concept, LRU, 2Q, ARC, cache algorithm

감사의 글

석사 논문이 나오기까지 연구실에서 지도해주시고 연구실에 받아주신 이동희 교수님께 감사의 말씀을 드리고 싶습니다. 제가 많이 부족해서 교수님께 많이 혼나기도 했지만, 석사 생활을 하면서 세미나나 여러 활동을 경험하게 해주셔서 더 깊게 그리고 많이 배울 수 있었던 좋은 기회였습니다. 그리고 석사 생활 중에 연구실에서 함께 시간을 보내주신 현철승 박사님께도 컴퓨터에 대해서 많이 배울 수 있었던 아주 뜻 깊은 시간이었던 것 같습니다. 학기 중에 바쁘신 와중에 시간을 내어서 논문심사위원을 허락해주신 황혜수 교수님 그리고 김민호 교수님께도 감사드립니다. 그리고 가장 소중한 우리 가족 아버지, 어머니 제 동생 오래 시간 저를 믿고 기다려주셔서 항상 고맙고 사랑한다는 말을 전하고 싶습니다. 감사합니다.