일반 플래시 SSD를 이용한 지속형 읽기/쓰기 캐시

A Persistent Read/Write Cache using General Flash-based SSDs

백승훈, 박기웅[1)]
Sung Hoon Baek, Ki-Woong Park

(367-805) 충북 괴산군 괴산읍 문무로 85 중원대학교
(300-716) 대전광역시 동구 대학로 62 대전대학교
shbaek@jwu.ac.kr, woongbak@dju.ac.kr

요 약

플래시 기반 SSD는 램과 하드디스크 사이의 2차 캐시로서 적은 비용으로 스토리지 시스템의 성능을 크게 향상 시킬 수 있다. 하지만 기존 기술들은 SSD의 비휘발성 특성을 활용하지 못하고 읽기 캐시만을 제공함으로써 쓰기에 대한 성능을 향상시키지 못하였다. 기존의 캐시 정책들은 읽기 캐시만 지원하기 때문에 쓰기캐시를 위해서는 완전히 새로운 형태의 캐시 정책이 연구되어야 한다. 쓰기 캐시가 가능하려면 캐시의 메타데이터를 정전에도 일관성이 유지될 수 있도록 설계되어야 한다. 본 논문은 일반 플래시 SSD를 이용한 일관성 있는 지속형 읽기/쓰기 캐시를 제시한다. 이 방법은 읽기뿐만 아니라 쓰기 요청도 SSD에 캐싱될 수 있게 하고; 정전 및 시스템기능정지 대해서 데이터 일관성과 무결성을 보장하고; 낮은 오버헤드를 가지며; 시장에서 구할 수 있는 일반 플래시 기반 SSD를 사용하고; 데이터의 시근성(recency)과 사용빈도(frequency)를 고려하고; 정전 후에도 캐싱된 데이터가 지속 가능하게 한다. 우리는 제안한 기술을 리눅스에서 구현하였으며 실제 여러 사용자로부터 장시간의 워크로드를 수집하여 성능을 측정하였다. 읽기 요청만 캐싱하는 기존 기술에 비하여 제안하는 지속형 읽기/쓰기 캐시는 약 두 배의 성능향상을 보인다.

Abstract

The flash-based SSD can improve the performance of storage systems at a low cost as a second cache between RAM and hard disk drives. However, traditional schemes do not utilize the non-volatility of SSD thus support only read cache and cannot improve the write performance. We need to investigate new cache architecture for a write cache that is totally different from the traditional read-only cache. To make the write cache possible, the cache must consistently manage its meta-data even in the event of crash or sudden power-off. This paper presents a consistent and persistent read/write cache using a general flash-based SSD. The scheme allows write requests to be cached in a SSD; ensures consistency and data integrity for crashes and power failures; considers both recency and frequency of data; and persistently retains cached data and meta data even after power failures. We implemented the proposed scheme in a Linux kernel. We aggregated a long-term workload from multiple users and evaluated the system using the real workloads. Our

1) 교신저자

논문투고일자 : 2014. 07. 04 심사일자 : 2014. 07. 11 게재확정일자 : 2014. 08. 20

read/write cache shows two times better performance than a traditional cache scheme that cannot cache write requests.

키워드: 스토리지, 캐시, 솔리드 스테이트 디스크
Keyword: Storage, Cache, SSD

1. 서론

플래시 메모리 기반의 SSD 등장으로 새로운 시스템 구조 및 기술들이 연구되어 왔다. 플래시 메모리는 크기가 작고 충격에 강하기 때문에 스마트폰과 같은 모바일 장치에 많이 사용되고 있으며, 플래시 기반 SSD는 하드 디스크를 대체하여 시스템의 성능을 높이는데 활용되고 있다.

SSD는 용량 측면에서 DRAM보다 크고 하드디스크보다 작고, 성능 측면에서 DRAM보다 느리고 하드 디스크보다 빠르며, 가격 측면에서 DRAM보다 싸고 하드 디스크보다 비싸다. 특히 하드디스크의 기계적 구동 특성으로, 하드디스크는 비순차 요청에 대해 긴 지연 시간을 발생시키나, 반도체로 구성된 SSD는 입출력 요청의 유형에 관계없이 하드 디스크에 비해 짧은 지연 시간을 제공한다.

이런 SSD의 용량, 성능, 가격 특성으로, SSD를 HDD와 DRAM사이의 새로운 계층으로 활용하기 위한 연구가 진행되어 왔다. 즉, DRAM은 1차 캐시로서 사용되고 SSD는 2차 캐시로 사용되고, HDD는 최종 데이터 저장 공간으로 사용된다.

기존 플래시 기반 SSD를 사용하는 2차 캐시 기술은 RAM 기반 캐시 기술[1]에 기반을 두고 있어 SSD의 비휘발성 특성을 활용하고 있지 못하고 있으며, 대용량의 SSD에 적용하기에 주메모리 자원을 많이 소모한다는 단점을 가지고 있다. 예를 들어, 휘발성 RAM에 사용하던 캐시 기술을 비휘발성 저장매체인 SSD와 RAM을 함께 쓰는 캐싱 시스템에 그대로 적용하게 되면, 캐시를 관리하는 메타 데이터는 빠른 데이터 처리를 위해 RAM에 저장하기 때문에 시스템이 재 시작된 후에는 SSD에 캐싱된 데이터를 활용할 수 없게 된다.

2차 캐시로서 SSD의 용량은 DRAM에 비교할 수 없을 만큼 크다. 그래서 SSD에 캐시 데이터를 채우는데 수 시간에서 하루가 걸릴 수 있다. 즉 빈 SSD로 시작하면 2차 캐시가 제 역할을 할 때까지 너무 오랜 시간이 소요된다.

이와 같은 문제를 해결하기 위하여, SSD 캐시, 즉 스토리지 급 2차 캐시의 재시작 시간을 단축하기 위해서 재시작 전에 학습하였던 캐싱정보로 2차 캐시를 채우고 시작하는 warm-start도 제안되었다[2]. 하지만 warm-start도 거대한 2차 캐시를 채우는 데 많은 시간이 소요된다.

많은 2차 캐시관련 기술들은 SSD의 비휘발성 기능을 활용하지 못하고 DRAM처럼 휘발성 캐시 장치처럼 이용하였다. SSD 저장매체의 비휘발성 장점을 캐시 시스템에서 십분 활용하기 위해서는 캐시가 갱신될 때마다 캐시 메타데이터도 일관성 있게 비휘발성 장치에 갱신해야 한다. 또한 정전 시에도 캐시 데이터와 메타데이터사이의 무결성도 보장되어야한다. 따라서 지속형 캐시는 큰 실시간 오버헤드를 가질 수 있다.

대부분의 SSD 캐시 기술들은 더티 데이터를 SSD 캐시에 두지 않고 Write through 정책을 사용한다. 즉, SSD에 캐싱된 데이터에 쓰기 요청이 오면 그 데이터는 SSD에서 퇴출되고 HDD에 최신 데이터가 유지되게 한다. 그러므로 HDD에 항상 최신의 데이터가 유지되어 캐시데이터의 무결성이 보장된다[3-7]. 이러한 정책은 전체 SSD 데이터를 잃어도 모든 데이터가 안전하게 보존되도록 한다.

SSD는 비휘발성 저장매체이지만 더티 데이터를 SSD에 유지하는 것은 어려운 문제이다. 더티 데이터를 SSD에 유지하려면 캐시 정보가 변경될 때마다 캐시 메타데이터를 비휘발성 저장장치에 일관성

있게 보존해야 하며, 정전 시에도 메타데이터와 데이터의 일관성을 잃지 않도록 해야 하기 때문이다.

또한, 읽기만을 위한 SSD 캐시 정책은 읽기 및 쓰기가 공존하는 대부분의 워크로드에서 SSD를 읽기 성능만을 개선시킬 수 있는 반쪽짜리 캐시에 머무르게 한다. 쓰기 가능한 SSD 캐시는 읽기뿐만 아니라 쓰기 성능도 향상시킬 수 있다. 그래서 쓰기 가능한 2차 캐시 기술이 기존 기술보다 두 배 높은 성능을 제공할 수 있다. 하지만 쓰기 캐시는 정전 및 크래시(시스템기능정지)에 대해서 데이터의 일관성과 무결성을 보장해줄 수 있는 기술을 요구한다.

Saxena[8]는 작은 성능 오버헤드로 쓰기 가능한 캐시를 제안하였다. Saxena는 Flash의 Out of Band(OOB)를 이용하는 원자쓰기(atomic write)[9]로써 메타데이터를 작은 부하로 일관성 있게 관리한다. 하지만 일반 SSD로는 호스트 컴퓨터가 Flash의 OOB를 접근할 수 없기 때문에 Saxena의 기법은 일반적으로 사용되기 어렵다.

우리는 시장에서 구할 수 있는 일반 플래시 SSD로써 정전 및 크래시에도 캐싱된 데이터가 휘발하지 않으며, 적은 오버헤드로 데이터 및 메타데이터 일관성과 무결성을 보장하고 쓰기 캐싱이 가능한 지속형 2차 캐시를 제안한다.

2. 관련 기술

2.1. 독립 캐시 계층

독립 캐시 계층 구조의 2차 캐시는 1차 캐시 정책과 독립적으로 동작한다. 그래서 1차 캐시에 있는 모든 데이터는 2차 캐시에도 존재하는 포함관계(inclusion property)가 있다. 기존 1차 캐시 계층을 수정하지 않아도 되며, 기존 캐시 기술을 2차 캐시에 적용할 수도 있다.

램에 캐싱된 데이터가 SSD에도 캐싱되어 있으므로 SSD의 공간 효율 측면에서 비효율적일 수도 있으나 SSD의 용량이 램 용량에 비해서 상당히 크면 공간 효율성은 무시할 수 있다.

2.2. 통합 캐시

1차 캐시와 2차 캐시(SSD)가 통합적으로 관리된다. 그래서 1차 캐시에 있는 데이터가 2차 캐시에 중복으로 저장되지 않는다. 1차 캐시에서 퇴출된 데이터는 2차 캐시로 캐싱이 되고 1차 캐시로 캐싱된 데이터는 2차 캐시에서 퇴출되어야 한다.

1차 캐시와 2차 캐시의 공간을 하나의 공간으로 통합하여 기존의 단일 캐싱 알고리즘인 LRU, LFU, FIFO, ARC [10] 등을 적용할 수도 있다. 하지만 L2ARC처럼 2차 캐시만을 위한 알고리즘도 있다[1].

2.3. 쓰기 캐시

쓰기 가능한 캐시 정책에 있어서, 2차 캐시에 캐싱된 데이터에 대해서 쓰기 요청이 오면 2차 캐시에만 쓰기가 발생하고 최종 저장공간(하드디스크)의 해당 데이터는 무효화된다. 2차 캐시에 최신의 데이터가 있기 때문에 정전 및 크래시에도 2차 캐시의 메타데이터를 안전하게 보존해야 한다.

크래시에도 안전하게 캐시 메타데이터를 일관성 있게 보존해야하며, 이를 위한 오버헤드를 줄이는 것이 쓰기 캐시에서 중요한 문제이다.

기존기술은 플래시의 OOB영역을 캐시 계층이 직접 제어하여 데이터와 메타데이터의 원자 쓰기[9]를 통한 쓰기 캐시를 구현하였다[8]. 하지만 일반 SSD의 OOB를 사용자가 사용할 수 없는 문제가 있다.

Koller는 저널링 기법을 이용한 SSD의 쓰기 버퍼를 제안하였다[11]. 쓰기 요청된 데이터를 로그 방식으로 저장하고 로깅한 여러 데이터들 뒤에 체크포인트를 기록한다. 이런 방식은 체크포인트 사이에 있는 데이터를 정전 시에 잃어버리는 단점이 있다.

2.4. 기타 기술

Albrecht는 워크로드에 따라 2차 캐시에 대하여 LRU와 FIFO 캐싱 정책 중에 최적의 정책을 동적

으로 결정하는 방법을 제안하였다[12].

Aguilera는 파일 시스템 수준에서 자주 접근되는 파일은 고속의 저장장치에 배치하고 그렇지 않은 파일은 저속의 저장장치에 배치하는 방법을 제안하였다[13].

2차 캐시로 사용되는 SSD의 Over-provision의 크기를 조절하여 최고의 성능이 나오는 크기를 찾는 방식도 제안되었다[14]. Over-provision이란 사용자에게 보이지 않고 SSD 안의 숨어 있는 저장공간이다. Over-provision 공간을 키우면 쓰기 성능이 좋아지나 캐시 공간이 줄어들어 히트율이 감소한다. 그러므로 적절한 Over-provision 크기에 대해서 최적의 성능이 나올 수 있다.

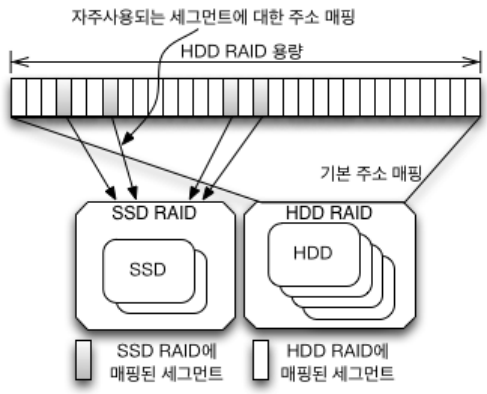
3. SSD Friendly 쓰기 캐시 기법 제안

본 논문은 정전 및 크래시에 대해서 데이터 일관성과 무결성을 보장하고, 시장에서 구할 수 있는 일반 플래시 기반 SSD를 사용하고, 시근성(recency)과 사용빈도(frequency)를 고려하는 지속형 쓰기 캐시를 제안한다.

쓰기 캐시란 크래시나 정전 후에도 하이브리드 스토리지 데이터의 일관성 유지하면서 SSD에 더티 데이터를 보유할 수 있는 캐시이다. 쓰기 캐시는 기존 읽기 전용 캐시에 비해서 두 배에 가까운 성능 향상을 제공할 수 있다. 하지만 성능 저하 및 데이터 일관성 유지 문제 때문에 일반 SSD로 만든 쓰기 캐시는 아직 나오지 못했다.

쓰기 캐시는 캐싱된 데이터와 더불어 캐시 메타데이터를 비휘발성 저장장치에 일관성 있게 저장해야 한다. 캐시 정보가 변경될 때마다 캐시 메타데이터가 변경되어야하기 때문에 성능저하가 크다. 우리는 쓰기 캐시의 오버헤드를 무시할 수 있는 쓰기 캐시를 제시한다.

정전 또는 크래시(시스템기능정지)에도 캐싱된 데이터와 캐시 메타데이터의 일관성을 유지하여 시작 후 바로 캐싱된 데이터를 사용할 수 있는 기능을 제공한다. 본 논문은 캐시 메타데이터를 크래시에도 안전한 관리 방법을 제시한다.

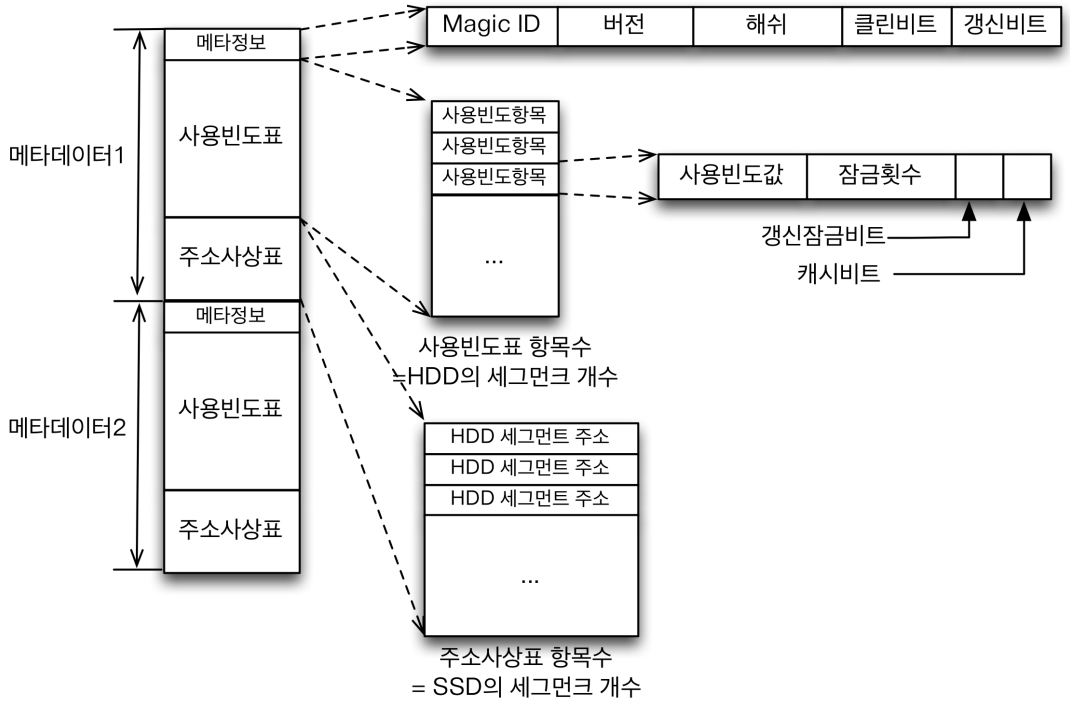


(그림 1) 하이브리드 스토리지의 주소 매핑 구조

본 연구는 지속형 쓰기 캐시의 일관성 문제를 해결하는데 있어서 특수한 장치를 사용하지 않고 시장에서 쉽게 구할 수 있는 일반 SSD를 사용한다.

제안한 캐시의 메타데이터 저장 오버헤드는 매우 적다. 요청받은 데이터들의 사용률 학습을 장시간 수행한 후에 주기적으로 캐시 데이터를 변경하는 방식을 사용하기 때문이다. 하지만, 데스크톱 워크로드에서는, 실시간에 캐시 데이터를 변경하는 기존 캐시 정책 못지않게 본 논문이 제안하는 주기적 캐시 갱신 방식이 높은 캐시 히트율을 보임을 실험에서 확인할 수 있다. 그러므로 본 연구는 데스크톱이나 가상 데스크톱 기반 서버에 사용하기에 적합하다.

또한, 시근성(recency)과 사용빈도(frequency)를 동시에 고려한다. 최근에 사용된 데이터들이나 높은 사용빈도의 데이터들 캐싱될 수 있다. 사용빈도가 높더라도 최근에 사용되지 않았다면 캐싱되지 않는다. LRFU [15], ARC [16], DULO [17] 등은 시근성(recency)과 사용빈도(frequency)를 고려한 캐싱정책이다, 하지만 이것들은 휘발성 램을 위해서 설계되었기 때문에 지속가능한 쓰기 캐시에는 사용할 수가 없다. 그래서 본 연구는 기존 캐시 정책들과는 구조적으로 다른 접근을 할 수 밖에 없었다.



(그림 2) HDD에 저장되는 메타데이터 구조

3.1 스토리지 구조

(그림 1)은 SSD를 2차 캐시로 사용하고 있는 하이브리드 스토리지의 구조를 보여준다. 단일 HDD 또는 HDD RAID[18]가 주 저장장치가 되고 자주 사용되는 저장공간은 단일 SSD 또는 SSD RAID에 매핑된다.

저장공간은 여러 섹터로 구성된 세그먼트 단위로 관리된다. 세그먼트 크기가 클수록 공간 지역성의 이득을 얻을 수 있다. 실험결과에서 큰 크기의 세그먼트가 좋은 성능을 보여준다. 실험에서는 128KiB에서 1MiB까지 세그먼트 크기를 사용하였다.

HDD의 모든 세그먼트에 대해서 사용빈도를 기록하고, 자주 사용되는 세그먼트(핫 세그먼트)를 SSD에 캐싱한다. 각 HDD 세그먼트마다 어느 SSD 세그먼트로 매핑되었는지 알 수 있는 주소 사상 수단(Radix Tree)을 제공한다. 거꾸로 각 SSD 세그먼트가 어느 HDD 세그먼트를 캐싱하고 있는지 알 수 있는 주소사상표를 제공한다.

입출력 요청이 오면 Radix Tree를 통해서 요청된 데이터가 속한 세그먼트가 SSD에 매핑되었는지 검사한다. SSD에 매핑되었으면 요청한 주소를 SSD로 매핑된 주소로 변경하여 서비스한다. 요청한 블록이 SSD에 매핑하고 있지 않다면, 주소 변경 없이 HDD에서 요청을 처리한다.

3.2 캐시 메타데이터

기존의 캐시 정책은 휘발성 메모리를 위한 정책이다. 즉, 전원이 중단되면 캐시의 내용을 잃게 되는 비지속형이며 정전에 따른 일관성 문제를 고민하지 않아도 된다. 하지만 정전 또는 크래시 후에도 캐시의 데이터를 보존하는 지속형 캐시에서는 일관성 문제가 발생한다.

지속형 캐시에서는 캐시 메타데이터를 변경될 때마다 비휘발성 저장장치에 저장해야 한다. 스토리지급 캐시의 용량이 큰 만큼 캐시 메타데이터가 매우 크다. 더구나, 최악의 경우에는 매 요청마다 캐시 메타데이터를 저장해야하기 때문에 성능 저하가 심

해져서 기존 캐싱 정책의 구조와는 전혀 다른 방식이 필요하다.

캐시 메타데이터의 갱신 오버헤드를 줄여야 하기 때문에 본 기술에서는 일정시간동안 세그먼트들의 사용빈도를 학습한 뒤에 캐시를 갱신하는 방식을 택한다. 캐시 갱신 시간 간격은 몇 시간에서 하루가 될 수 있다. 또는 캐시 갱신은 사용자가 사용하지 않는 유휴 시간동안 발생할 수도 있다. 실험에서는 하루에 한번 갱신하는 방식을 택하였지만 기존 방식보다 좋은 성능을 보였다. 그 이유는 워크로드가 대형 캐시(SSD)의 용량을 넘어서는데 긴 시간이 걸리기 때문이다.

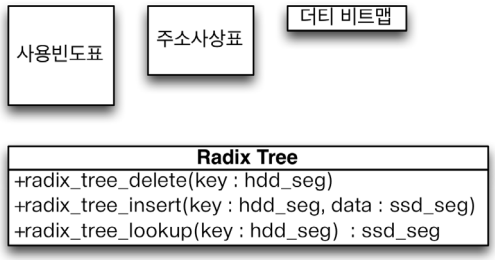
(그림 2)는 본 캐시 정책에 따라 안전한 메타데이터를 관리하기 위한 메타데이터의 구조이다. 이 메타데이터는 하드디스크 또는 SSD에 저장된다. 메타데이터의 각 필드의 기능은 다음과 같다.

메타데이터의 무결성을 위해서 두 개의 메타 영역이 존재한다. 메타데이터1과 메타데이터2의 위치를 번갈아 메타데이터가 저장된다. 메타데이터 저장을 미완료한 상태에서 정전이 발생하면 이전 영역의 메타데이터를 사용할 수 있다. 메타데이터의 메타정보의 버전 필드로써 가장 최신의 메타데이터가 어느 영역에 있는지 구분한다. 메타데이터를 저장할 때마다 버전 값은 1씩 증가한다.

메타정보의 해쉬는 메타데이터를 저장하는 중에 정전이 발생하였을 때에 메타데이터가 무결하지 않은지를 검사하기 위하여 사용된다. 메타데이터가 부분적으로 저장되다가 정전이 되면 해쉬 필드로써 메타데이터 저장이 실패하였다는 것을 가리키게 된다. 그러면 다른 메타영역의 메타데이터를 사용한다.

클린비트가 1이면 정상종료가 있었는지를 가리키고 클린비트가 0이면 정전이나 크래시가 있었음을 나타낸다. 갱신비트는 캐싱 및 퇴출 프로시저가 동작 중에 정전이 있었는지를 가리킨다.

사용 빈도표를 구성하는 사용빈도항목은 각 HDD세그먼트마다 할당된다. 각 사용빈도항목에서 사용빈도값은 해당 HDD세그먼트가 최근에 자주 사용되었는지를 가리킨다. 잠금 횟수 및 갱신잠금



(그림 3) 주메모리에서 관리되는 메타데이터

비트는 메타갱신 프로시저와 입출력처리 프로시저 사이에 세그먼트의 배타적 접근 및 동기화를 위하여 사용된다. 캐시비트는 이 HDD세그먼트가 SSD로 캐싱되었는지를 나타낸다.

주소사상표는 각 SSD 세그먼트가 어떤 HDD세그먼트를 캐싱하고 있는지를 가리킨다. 주소사상표의 항목 수는 SSD의 세그먼트 개수와 같고, 각 항목은 HDD세그먼트 주소를 보유하고 있다.

실시간에 사용되는 사용빈도표와 주소사상표 및 Radix Tree는 주메모리에 위치한다. 사용빈도표와 주소사상표는 대용량 데이터이기 때문에 이것들이 변경될 때마다 HDD에 저장하는 것은 성능에 큰 영향을 미친다. 그래서 메타데이터를 저장하는 회수를 최소화해야 한다.

주메모리에 존재하는 메타데이터는 (그림 3)에 나타나 있다. 주메모리의 사용빈도표 및 주소사상표는 HDD에 저장되는 사용빈도표 및 주소사상표와 같은 구조를 갖고 있다.

주메모리의 더티비트맵과 Radix Tree는 저장장치에 존재하지 않는다. 더티 비트맵은 각 SSD 세그먼트가 더티 상태인지 클린 상태인지를 가리킨다. 정상적인 종료일 경우에는 모든 SSD 세그먼트가 클린 상태이고, 비정상적인 종료일 때에는 모든 SSD 세그먼트를 더티 상태로 간주하기 때문에, 더티 비트맵을 저장장치에 저장할 필요가 없다.

Radix Tree는 주어진 HDD세그먼트가 어느 SSD세그먼트에 캐싱이 되었는지를 찾는 데에 사용된다. Radix Tree의 데이터는 HDD에 저장되지 않

는다. 왜냐하면 주소사상표를 통하여 Radix Tree를 구성할 수 있기 때문이다.

3.3 시근성과 사용빈도

캐시에서 어떤 것을 퇴출하고 어떤 것은 삽입할 것인가를 결정하는 것은 매우 중요한 부분이다. 우리는 최근에 사용되었으며 자주 사용된 데이터의 순서를 부여하여 캐싱할 대상을 결정해야 한다.

하지만 기존의 캐싱 정책(LRU, LRFU, ARC, L2ARC, DULO)들은 이중 연결 리스트를 사용하고 있다. 이중 연결 리스트는 구현상 효과적이지만 많은 메모리를 사용한다. SSD를 캐시 장치로 사용하면 이중 연결 리스트로 사용되는 메모리의 양이 매우 크다. 만약 1TB의 SSD를 캐시로 사용하는 서버가 있고 4KiB 단위로 캐시를 관리한다면, 리스트의 포인터에만 약 4GB의 메모리가 리스트 포인터에 필요하게 된다.

우리는 연결 리스트를 사용하지 않고, 메모리를 적게 사용하면서 시근성과 사용빈도를 고려할 수 있는 방식을 제안한다.

세그먼트의 시근성(recency)과 사용빈도(frequency)를 고려할 수 있도록 모든 HDD 영역의 각 세그먼트는 2바이트의 사용빈도값 변수를 가진다. 사용빈도값이 높을수록 최근에 자주 사용되었음을 나타낸다.

세그먼트에 히트가 발생할 때마다 그 세그먼트의 사용빈도값을 기 설정된 상수 값만큼(실험에서는 5를 사용하였다) 증가시킨다. 사용빈도값의 오버플로우를 막기 위해서 $2^{16}-1$까지만 증가한다. 사용빈도값의 과도한 증가를 막기 위해서 연속으로 히트된 경우에는 히트가 한번인 것으로 간주한다. 사용빈도값이 기설정한 문지방값(실험에서는 20으로 설정하였다)을 넘으면 그 HDD세그먼트를 핫 세그먼트로 간주한다.

이와 같은 방법을 사용하면 자주 사용되는 세그먼트일수록 높은 사용빈도값을 가진다. 하지만 시근성을 반영하지는 못한다. 즉 아주 오래 자주 사용되었지만 최근에는 사용되지 않는 세그먼트를 구별해내지 못한다. 그래서 다음과 같이 사용빈도값에 시근성을 반영하도록 한다.

시근성(recency)을 위해서 핫 세그먼트(HDD 중에 자주 접근되는 세그먼트)의 개수가 캐싱 가능한 세그먼트 개수(SSD 세그먼트 수)를 넘을 때마다. 사용빈도값에 대한 감쇠 절차가 수행된다. 감쇠 절차에서는 모든 사용빈도값에 감쇠상수(실험에서는 4/5를 사용하였다)를 곱하여 모든 사용빈도값을 줄인다. 이렇게 하면 사용빈도가 많더라도 사용한지 오래되었다면 낮은 사용빈도값을 가진다. 그러므로 이 캐시 정책은 시근성과 사용빈도를 고려하게 된다.

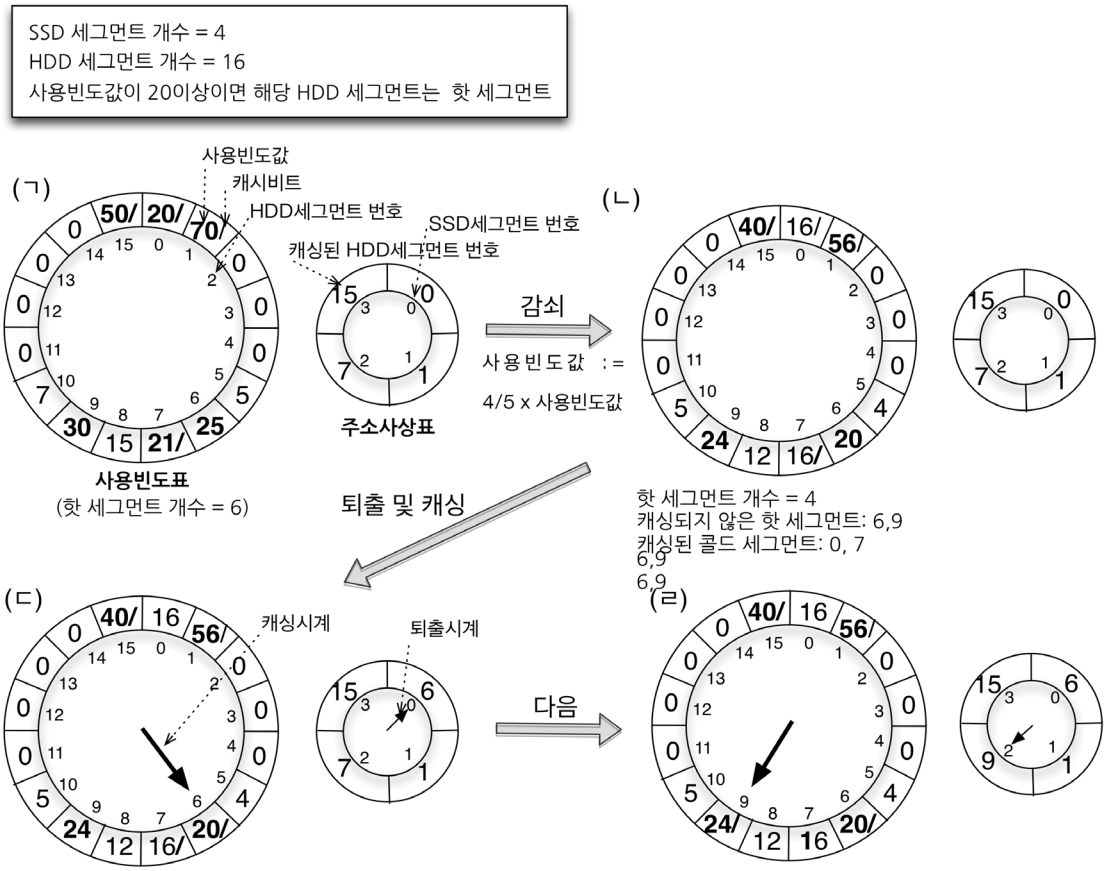
이러한 감쇠절차는 O(N)의 연산량을 필요로 하지만, 매우 드물게 발생되는 연산이기 때문에 이 감쇠절차의 오버헤드가 전체 성능에 영향을 거의 주지 못한다. 또한 감쇠절차는 일반 입출력 요청 서비스와 별개의 쓰레드에서 처리되기 때문에 입출력 요청 서비스의 응답시간에도 영향을 주지 않는다.

사용빈도값이 기설정한 문지방값을 넘을 때까지 자주 사용되어야 핫 세그먼트가 되어 2차 캐시로 캐싱이 된다. 하지만. 초기 상태부터 충분한 개수의 핫 세그먼트를 갖추는 데에 수일이 필요하다. 초기 상태에서 SSD의 활용도를 높이기 위해서, 만약 한번이라도 접근한 HDD세그먼트의 수가 SSD세그먼트 수보다 적다면 한번이라도 접근한 HDD세그먼트를 핫 세그먼트로 간주한다.

핫 세그먼트인데 2차 캐시로 캐싱되지 않은 세그먼트를 캐싱하고, 캐싱되어 있지만 핫 세그먼트가 아닌 것을 퇴출해야 한다. 이것은 캐싱 및 퇴출 절차 과정에서 처리된다.

3.4 캐싱 및 퇴출

캐싱 및 퇴출은 스케줄링에 의해 사용자가 설정한 시점에 실행되거나 주기적으로 실행되고 가장 낮은 우선순위로 실행된다. 이 주기는 몇 시간 또는 하루로 정할 수 있다. 캐싱 및 퇴출 절차가 실행되는 시점을 컴퓨터를 사용하지 않는 시간에 설정하게 되면 캐싱 및 퇴출 오버헤드를 사용자는 경험하



(그림 4) 캐싱은 유휴상태 또는 예약된 시간에만 수행되며, 사용빈도값의 감쇠를 수행한 후에 퇴출 및 캐싱을 수행한다. 이 예제에서 SSD의 세그먼트 개수는 4이며 HDD의 세그먼트 개수는 16이고, 사용빈도값이 20이상인 세그먼트는 핫세그먼트로 간주된다. (ㄱ) 워크로드의 변화로 캐싱되지 못한 핫 세그먼트가 발생한다. (ㄴ) 캐싱을 수행하기 전에 핫 세그먼트의 개수가 SSD 세그먼트의 수를 넘지 않을 때까지 모든 '사용빈도값'에 대하여 감쇠를 수행한다. (ㄷ) 캐시시계는 0부터 시계방향으로 돌아 다음 '핫 세그먼트이지만 캐싱되지 않은 HDD세그먼트'에서 멈춘다. 퇴출시계는 지난 위치 다음부터 시계방향으로 돌아 '핫 세그먼트가 아닌 HDD세그먼트를 캐싱하고 있는 SSD세그먼트'에 멈춘다. 퇴출시계가 가리키는 SSD세그먼트 0번에 캐싱된 HDD세그먼트 0번을 퇴출시키고, SSD세그먼트 0번(퇴출시계)에 캐시시계가 가리키는 HDD세그먼트 6번을 캐싱시킨다. (ㄹ) 캐시시계는 계속 돌아 다음 '핫 세그먼트이지만 캐싱되지 않은 HDD세그먼트'에서 멈춘다. 퇴출시계는 계속 돌아 다음 '핫 세그먼트가 아닌 HDD세그먼트를 캐싱하고 있는 SSD세그먼트'에 멈춘다. 캐시 시계는 2에서 멈추고 퇴출 시계는 7에서 멈춘다. 퇴출시계가 가리키는 SSD세그먼트 2번에서 HDD 세그먼트 7번을 퇴출시키고, SSD세그먼트 2번에 캐시시계가 가리키는 HDD세그먼트 9번을 캐싱시킨다. 캐싱시계가 한 바퀴를 돌 때까지 '퇴출 및 캐싱'을 수행한다. 퇴출시계의 마지막 위치는 다음을 위해서 보관된다.

지 못하게 된다.

기존의 캐싱 정책은 핫 세그먼트가 바뀔 때마다 캐싱 및 퇴출이 발생하는 반면에, 제안하는 방법에서는 주기적으로 캐싱 및 퇴출을 수행하며 대부분의 서비스 시간에는 이것을 수행하지 않는다. 후자의 방법은 히트율을 낮게 하지만, 캐싱 및 퇴출에 따라 실시간에 2차 캐시 장치(SSD)에 대역폭을 잠식하지 않고(캐싱할 때 SSD로 쓰기발생, 퇴출할 때에 더티 세그먼트에 대해서는 SSD에서 HDD로 복사하는 과정 발생), 메타데이터 갱신 오버헤드를 줄여서 쓰기 캐시가 가능하게 하는 장점이 있다.

실제 사용자 워크로드를 통하여 측정한 성능평가

에서는, 실시간 캐싱 및 퇴출에 따른 2차 캐시 장치의 대역폭 잠식이 감소된 히트율에 의한 성능에 상응하게 된다는 것을 보여준다. 하지만 메타데이터 갱신 오버헤드의 감소에 따른 쓰기 캐시의 효과는 기존 기술의 성능을 훨씬 뛰어 넘는다.

캐싱 및 퇴출 절차가 시작하면 '핫 세그먼트인데 캐싱되지 않은 세그먼트'를 SSD로 캐싱하고 '핫 세그먼트가 아닌데 SSD에 캐싱된 세그먼트'를 퇴출시킨다. 캐싱 및 퇴출 절차가 시작할 때에는 핫 세그먼트 개수가 SSD 세그먼트의 개수를 넘기지 않도록 감쇠 절차를 수행한다. 각 세그먼트의 사용빈도값이 기설정한 문지방값 이상이면 그 세그먼트를 핫 세그먼트로 간주한다.

(그림 4)는 SSD세그먼트가 4개이고 HDD세그먼트가 16개인 경우에 감쇠 절차와 캐싱 및 퇴출 절차가 수행되는 예를 보여준다.

사용빈도표와 주소사상표는 시작과 끝이 연결된 원형으로 구성된다. 사용빈도표는 각 HDD세그먼트의 사용빈도값을 항목으로 하고 항목의 개수는 HDD세그먼트의 개수이다. 주소사상표는 각 SSD 세그먼트가 캐싱하고 있는 HDD세그먼트의 주소를 항목으로 하고, 항목의 개수는 SSD세그먼트의 개수이다.

사용빈도표에서 동작하는 캐시시계는 캐싱 및 퇴출 절차를 시작할 때에 0부터 시작하고, 시계방향으로 돌아 '핫 세그먼트이지만 캐싱되지 않은 HDD세그먼트'에서 멈춘다. 퇴출시계는 지난 위치 다음에서 시계방향으로 돌아 '핫 세그먼트가 아닌 HDD세그먼트를 캐싱하고 있는 SSD세그먼트'에서 멈춘다. 퇴출 시계가 가리키는 곳(SSD세그먼트)에 캐싱된 HDD세그먼트를 퇴출하고, 그곳에 캐시시계가 가리키는 HDD세그먼트를 캐싱한다. 캐시시계가 1회전 할 때까지 캐시시계와 퇴출시계는 지난 위치 다음부터 시계방향으로 돌아 이 절차를 반복한다.

(그림 4 (ㄷ))에서, 퇴출시계가 가리키는 SSD세그먼트 0번에 캐싱된 HDD세그먼트 0번을 퇴출시키고, SSD세그먼트 0번(퇴출시계)에 캐싱시계가 가리키는 HDD세그먼트 6번을 캐싱시킨다. 캐시시계는 계속 돌아 다음 '핫 세그먼트이지만 캐싱되지 않은 HDD세그먼트'에서 멈춘다.

(그림 4(ㄹ))에서, 캐싱시계는 9에서 멈추고 퇴출 시계는 2에서 멈춘다. 퇴출시계가 가리키는 SSD세그먼트 2번에서 HDD 세그먼트 7번을 퇴출하고, SSD세그먼트 2번에 캐싱시계가 가리키는 HDD세그먼트 9번을 캐싱한다. 캐싱시계가 1회전하면 캐싱 및 퇴출 절차는 종료된다.

3.5 복원

메타데이터의 클린비트를 통하여 정상적 종료 후의 재시작인지 비정상 종료 후 재시작인지를 확인할 수 있다.

정상적으로 종료가 되었다면 다음 시작에서는 HDD 또는 SSD에 저장된 최신 메인 메타데이터의 주소사상표와 사용빈도표를 주메모리로 읽고 주소사상표를 기반으로 HDD세그먼트에서 캐싱된 SSD세그먼트 주소를 찾아주는 RadixTree를 생성한다.

캐싱 및 퇴출 절차 시작 전에 메타데이터의 갱신비트를 1로 변경하고 완료 후에 이 갱신 비트를 0으로 설정한다.그래서 갱신비트로써 캐싱 및 퇴출 과정에서 정전이나 크래시가 발생하였는지 알아낼 수 있다.

캐싱 및 퇴출 과정이 아닌 경우에 정전이나 크래시가 발생하였었다면 SSD에 저장된 데이터를 HDD로 복사하는 과정이 추가된다. 왜냐하면 SSD에 캐싱된 데이터중에 어느 것이 더티 (최신) 데이터인지 모르기 때문이다. 정상 종료과정에서는 더티 데이터들을 HDD로 저장하였기 때문에 재시작 후에 SSD에는 더티 데이터가 남아 있지 않는다.

HDD에서 로딩한 최신 메타데이터의 갱신비트가 1이다면 캐싱 및 퇴출 과정에서 정전이 발생하였다는 것을 의미한다. 그래서 메인메타와 SSD 캐시가 보유한 데이터가 다르므로 메인메타 및 SSD캐시의 데이터를 무시한다. 캐싱 및 퇴출 과정에서는 SSD에 더티 데이터가 없기 때문에 SSD의 데이터를 무시해도 일관성에는 문제가 없다.

캐싱 및 퇴출 프로시저는 가끔 수행되므로 정전

으로 인하여 SSD 캐시를 리셋하는 확률은 낮다. 하지만 SSD 캐시를 리셋하는 경우가 발생한다면 마지막 메인메타의 정보를 기반으로 캐시를 다시 채워서 웜 스타트(warm start)로 시작한다.

3.6 일관성 유지

쓰기가능한 SSD 캐시에서는 데이터의 일관성을 유지하는 방법이 매우 중요하다. 기존 기술들은 일관성 유지를 위하여 간단히 읽기 캐시만 가능한 SSD 캐시를 제시하였다. 본 논문이 제시하는 기술은 메타데이터의 일관성을 유지할 수 있기 때문에 쓰기 캐시가 가능한 방법이다.

캐싱 및 퇴출 절차가 실행되지 않는 구간에서는 HDD에 저장된 메타데이터와 SSD 캐시의 상태가 동일하므로, 이 구간 안에서의 정전이 발성하여도 메타데이터 및 데이터의 일관성이 유지된다.

캐싱 및 퇴출 절차가 실행되는 동안에 데이터의 일관성 유지를 위하여 더티 데이터를 보유하지 않는다. 즉, 캐싱된 곳에 쓰기 요청이 오면 SSD와 HDD로 동시에 이중 쓰기가 수행된다. 또한 캐싱 및 퇴출 절차가 시작하기전에 모든 더티 데이터는 HDD로 복사된다. 그러므로 이 절차를 수행하는 중에 크래시가 발생하여도 원본 데이터를 잃지 않는다. 캐싱 및 퇴출 절차는 아주 드물게 (하루에 한번) 실행되므로 대부분의 경우에는 쓰기 캐싱을 수행할 수 있다.

캐싱 및 퇴출 절차가 실행되기 전에 SSD의 캐싱된 세그먼트 중 더티 세그먼트(HDD에 저장된 데이터보다 SSD에 캐싱된 데이터가 최신인 세그먼트)를 HDD의 원래 위치로 복사한다. 그렇게 해서 모든 더티 데이터를 클린 상태로 만든다. 또한 메타데이터의 갱신비트를 1으로 설정한다.

캐싱 및 퇴출 프로시저가 수행 중이지 않을 때에 정전이 발생하면 HDD에 저장된 메타데이터는 무결한 SSD의 캐싱상태를 가리키고 있다. 그래서 이런 재시작에서는 HDD에 저장된 주소사상표와 사용빈도표를 그대로 주메모리의 주소사상표와 사용빈도표로 로딩하여 사용한다.

정상종료를 수행할 때에는 모든 SSD의 세그먼트를 클린 상태로 만든다. 즉 모든 더티 세그먼트를 HDD의 원본 위치에 복사한다. 그러므로 정상 종료 후의 재시작시에는 모든 SSD의 세그먼트는 클린 상태이다.

재시작후에 로딩한 메타정보의 클린비트가 0이면 비정상종료 즉 정전 또는 크래시가 발생하였음을 나타낸다. 이 경우에는 일부 SSD 세그먼트가 더티 상태이나 어느 것인 더티인지 모르기 때문에 모든 SSD 세그먼트를 HDD의 원본 위치로 복사한다. 이렇게 함으로써 데이터의 일관성을 유지할 수 있다.

3.7 의사코드

(그림 5)의 의사코드로는 본 기술의 전체 동작을 볼 수 있다. 캐싱및퇴출 프로시저(1~28행)는 스케쥴러에 의해서 호출된다.

캐싱및퇴출프로시저 안에서는 핫 세그먼트들의 대상이 변경되는 것을 막기 위해서 모든 사용빈도값은 변경되지 않는다(3행).

SSD의 더티 세그먼트들을 HDD로 복사한다(4행), 또한 캐싱및퇴출프로시저가 수행중에는 더티 세그먼트를 생성하지 않지 않기 때문에 (캐싱영역과 HDD에 동시 쓰기를 수행하기 때문에) 캐싱및퇴출프로시저 수행 중에 정전이 발생해도 HDD만으로 데이터를 구할 수 있다.

그 다음 갱신비트를 1로 설정한다(5행). 이 프로시저가 끝나면 갱신비트를 0으로 설정하므로써(26행) 이 프로시저 수행 도중에 정전이 발생하였는지를 감지할 수 있다.

핫 세그먼트들의 수가 SSD 세그먼트의 개수를 넘으면 안되므로 핫 세그먼트의 개수가 SSD의 세그먼트 개수와 같거나 작을 때까지 사용빈도표의 모든 사용빈도값에 대해서 감쇠 절차를 실행한다(6~7행).

캐시시계는 0에서부터 한 바퀴를 돌때까지 (8행) 캐싱 및 퇴출이 진행된다. 캐시시계가 가리키는 HDD세그먼트가 핫 세그먼트이고 캐싱이 되지 않

```
1  PROCEDURE 캐싱및퇴출프로시저
2  BEGIN
3    모든 사용빈도값 변경 금지
4    모든 SSD의 더티 세그먼트를 HDD로 복사
5    HDD의 최신 메타데이터의 갱신비트에 1을 세팅
6    WHILE  num_of_hot_segments > num_of_ssd_segments
7      사용빈도표에 대한 감쇠 절차 수행
8    FOR 캐시시계 :=0 to HDD의 세그먼트 개수-1
9    BEGIN
10     IF 캐시시계가 가리키는 hdd세그먼트가 핫 세그먼트이고 캐싱
         이 되지 않았다면
11     BEGIN
12        퇴출시계 := 다음_퇴출시계(퇴출시계);
13        퇴출HDD세그먼트 := 주소사상표[퇴출시계]
14        victim_lock := lock_segment(퇴출HDD세그먼트)
15        퇴출HDD세그먼트의 캐시비트 = 0
16        radix_tree_delete(tree, 퇴출HDD세그먼트)
17        cache_lock := lock_segment(캐시시계)
18        주소사상표[퇴출시계] := 캐시시계
19        캐시시계가 가리키는 HDD세그먼트의 캐시비트 := 1
20        캐시시계가 가리키는 HDD세그먼트의 데이터를 퇴출시계
            가 가리키는 SSD세그먼트로 복사한다.
21        radix_tree_insert(tree, 캐시시계, 퇴출시계)
22        unlock_segment(cache_lock)
23        unlock_segment(victim_lock)
24     END IF
25   END FOR
26   갱신비트에 0을 세팅하고 메타데이터 저장
27   모든 사용빈도값 변경 금지 해제
28 END
29
30 PROCEDURE 시작프로시저
31 BEGIN
32   HDD에 저장된 최신 버전의 메타데이터를 주메모리로 로딩한다.
33   IF 갱신비트가 0이면
34   BEGIN
35      IF 클린비트가 0이면
36         SSD에 캐싱된 모든 데이터를 HDD로 복사한다.
37      FOR EACH 주소사상표의 각 주소사상항목 <SSD세그먼트,
                HDD세그먼트> 쌍에 대하여
38        radix_tree_insert(tree, HDD세그먼트, SSD세그먼트)
39   ELSE
40      FOR EACH 주소사상표의 각 주소사상항목 <SSD세그먼트,
   HDD세그먼트> 쌍에 대하여
41      BEGIN
42        HDD세그먼트의 데이터를 SSD세그먼트로 복사한다.
43      radix_tree_insert(tree, HDD 세그먼트, SSD세그먼트)
44      END FOR
45   END IF
46   클린비트를 0으로 설정하고 메타데이터를 HDD에 저장한다.
47   더티 비트맵을 초기화 한다.
48 END
49
50 PROCEDURE 종료프로시저
51 BEGIN
52   SSD에 캐싱된 세그먼트 중 더티 세그먼트들을 HDD로 복사한다
53   클린비트를 1로 설정하고 메타데이터를 HDD에 저장한다.
54 END
```

(그림 5) 의사 코드

았다면 캐시시계는 거기서 멈추고 (10행) 퇴출시계는 핫 세그먼트가 아닌 HDD세그먼트를 캐싱하고 있는 SSD세그먼트에서 멈춘다 (12행).

퇴출될 HDD세그먼트 주소(퇴출HDD세그먼트)는 퇴출시계값과 주소사상표에서 얻을 수 있다 (13행). 그리고 퇴출시계가 가리키는 SSD세그먼트에 캐싱된 HDD세그먼트를 퇴출한다 (14~16행).

퇴출HDD세그먼트가 퇴출 절차를 수행하는 동안 IO 요청을 막기 위해서 이 세그먼트를 잠근다 (14행). 만약 이 세그먼트에 대해서 IO 요청이 진행 중이라면 이 IO 요청이 완료될 때까지 멈춘다. 반대로 세그먼트가 잠겨지면 이 세그먼트에 대한 IO요청이 블록된다. 퇴출HDD세그먼트는 캐싱되지 않은 것으로 설정하고 (15행) Radix Tree에서 퇴출HDD 세그먼트 정보를 제거한다(16행).

이제는 퇴출시계가 가리키는 SSD세그먼트로 캐시시계가 가리키는 HDD세그먼트를 캐싱하는 절차를 수행한다 (18~21행). 캐싱될 세그먼트가 위치할 곳(퇴출시계)에 대한 주소사상표에 캐싱할 HDD세그먼트(캐시시계) 주소를 대입한다(18행). 그다음 캐싱시계가 가리키는 HDD세그먼트가 캐싱되었음을 나타내는 캐시비트를 셋한다(19행). 그리고 캐싱할 데이터(캐시시계가 가리키는 HDD세그먼트)를 퇴출시계가 가리키는 SSD 세그먼트로 복사한다(20행).

모든 핫 세그먼트의 캐싱이 완료된 후에 주메모리의 주소사상표 및 사용빈도표를 HDD에 위치한 다음 메타데이터 영역에 저장한다.(26행)

재시작을 하면 시작프로시저가 실행된다 (31~36행). 가장 먼저 수행하는 것은 메타데이터1과 메타데이터2 중에서 어느 것이 최신 버전인지를 찾고 해시 오류가 없는 최신 메타데이터를 주메모리의 주소사상표 및 사용빈도표로 로딩한다(32행).

갱신비트가 0이면 캐싱 및 퇴출프로시저 수행이 아닌 상태에서 정전이 발생하였음을 의미한다(33행). 이때에 만약 클린비트가 0이면 정상종료를 수행하지 못하였음을 의미한다. 그래서 SSD에는 더티 데이터들이 있고 어느 것이 더티 인지 클린인지 모르기 때문에 SSD의 모든 세그먼트를 해당 HDD 세그먼트로 복사한다 (35~36행). 그리고 로딩한 주소사상표로써 Radix Tree를 구성한다(37~38행).

갱신비트가 1이면 캐싱및퇴출프로시저 수행 상태에서 정전이 발생하였음을 의미하므로 로딩한 메타데이터와 실제로 SSD에 캐싱된 데이터는 불일치하므로 SSD에 캐싱된 데이터를 사용할 수가 없다. 그래서 최신 주소사상표를 바탕으로 SSD에 데이터를 캐싱하는 Warm-start 작업을 수행한다.(40~44행) 정상 종료에서는 종료프로시저가 실행된다. SSD에 캐싱된 세그먼트 중 더티 세그먼트를 HDD로 복사하고(52행) 클린비트를 1로 설정하고 메타데이터를 HDD에 저장한다(53행).

4. 성능 평가

4.1 실험 환경

리눅스 커널 2.6.35에서 블록 장치로서 하이브리드 장치를 구현하였으며, Intel의 SSDSC2CT24(SSD)와 Seagate의 ST2000DM001(HDD)를 각각 한 개씩 사용하였다. SSD와 HDD는 SATA Ⅲ(6Gbps)로 연결하였다. 32GB용량의 SSD를 사용하였으며 HDD의 용량은 2TB로 사용하였다.

윈도우즈 XPerf™를 이용하여 실제 데스크톱 사용자들의 디스크 입출력 트레이스를 10일동안 추출하였다. 페이지 캐시 계층을 사용하지 않는 Direct IO를 사용하여 수집한 트레이스를 재생하였다. Direct IO를 사용하였기 때문에 주메모리 캐싱 및 프리페칭 효과는 없다.

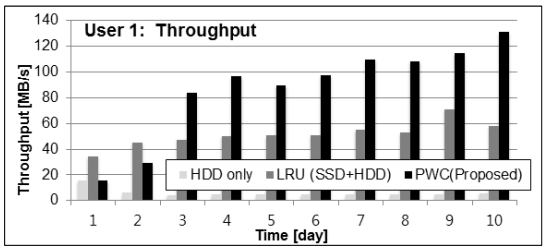
이 실험에서의 비교 대상으로 단일 디스크(HDD only)와 LRU정책으로 SSD캐시를 운영하는 기존 방식-LRU(SSD+HDD)-을 사용하였다. 본 논문에서 제시하는 기술은 'PWC(Proposed)'로 표기하였다.

이 실험에서 PWC는 하루에 한 번씩 유휴시간에 캐싱 및 퇴출 프로시저를 수행한다. LRU정책에서는 컴퓨터를 끄지 않고 캐시정보를 잃어버리지 않는다는 가정으로 성능을 측정하였다.

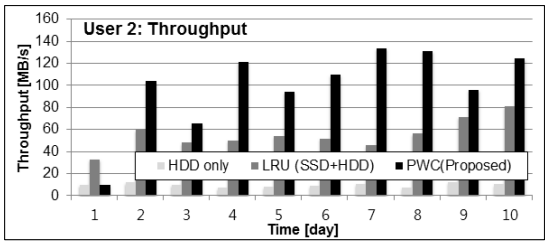
4.2 실험 결과

(그림 6)~(그림 8)은 세 사용자의 처리량을 보여준다. PWC는 첫째 날 캐싱된 데이터가 없어 단일 HDD와 같은 성능을 보이지만 사용자1에서는 셋째 날부터 사용자2에서는 둘째 날부터 LRU방식보다 거의 두 배 빠른 처리량을 보인다.

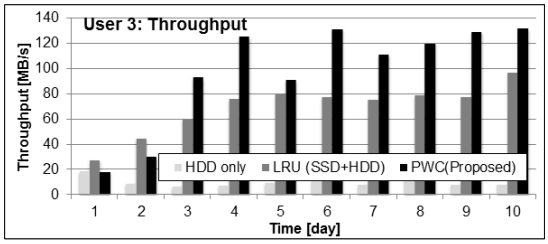
(그림 9)~(그림 11)은 세 사용자의 지연시간을 보여준다. 지연시간은 처리량과 비슷한 현상을 보여준다. 첫째날은 캐싱된 데이터가 없는 PWC는 단일 HDD와 같은 지연시간을 보이지만. 2~3일 이후로 가장 짧은 지연시간을 보여준다.



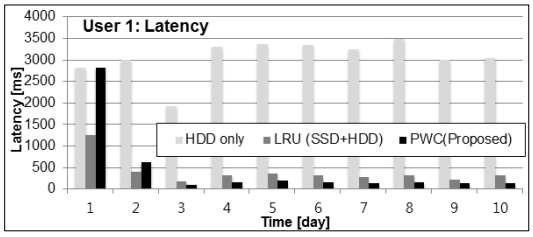
(그림 6) 사용자1의 처리량



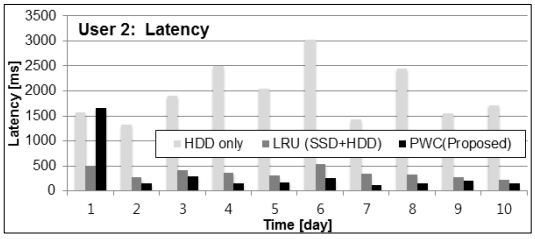
(그림 7) 사용자2의 처리량



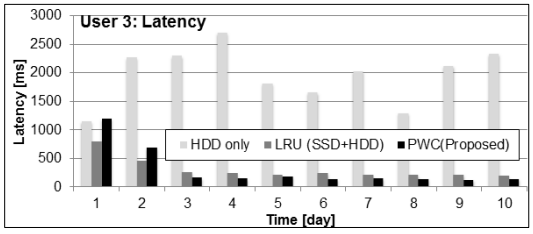
(그림 8) 사용자3의 처리량



(그림 9) 사용자1의 지연시간



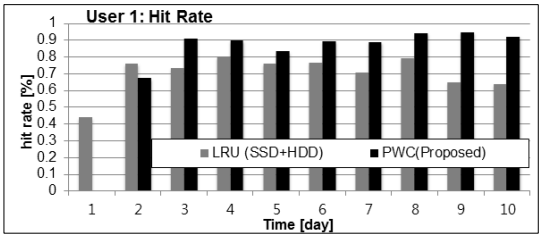
(그림 10) 사용자2의 지연시간



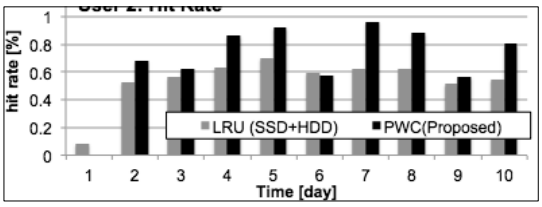
(그림 11) 사용자3의 지연시간

(그림 12)~(그림 14)는 세 사용자의 캐시 히트율을 보여준다. 첫째날에는 PWC의 캐시 히트율은 0이다. 그리고 2~3일 이후에 가장 높은 캐시 히트율을 보여준다. PWC는 쓰기에 대해서는 캐시 히트를 발생시키지만 LRU방식은 쓰기가 발생하면 HDD로 데이터를 저장하기 때문에 Writable Hybrid가 가장 높은 캐시 히트율을 보여준다.

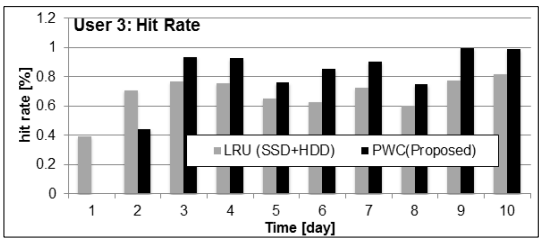
하지만 처리량은 LRU방식보다 약 두배의 빠른 처리량을 보여주였지만 캐시 히트율은 두 배까지 높지는 않다. 왜냐하면 LRU는 실시간에 캐싱과 퇴출 동작 오버헤드가 실시간에 발생한다. 즉, SSD로 데이터를 복사하는 추가 작업이 SSD의 대역폭을 감소하게 한다. 하지만 제안된 방식은 유휴시간에만 캐싱과 퇴출동작이 발생한다. 본 실험에서는 하루에 단 한번만 실행하도록 하였다. 그래서 LRU방식보다 제안하는 방식이 SSD에 대한 실시간 오버헤드가 적다.



(그림 12) 사용자1의 캐시 히트율

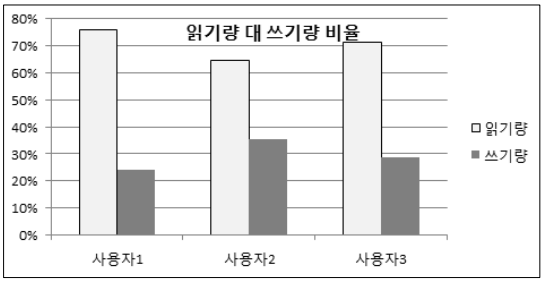


(그림 13) 사용자2의 캐시 히트율



(그림 14) 사용자3의 캐시 히트율

LRU방식은 읽기 캐싱 이득만 있지만, 제안하는 방식은 읽기 뿐만 아니라 쓰기에 대한 캐싱 이득이 있다. 쓰기의 비율이 높을수록 성능의 이득이 더 클 것이다. (그림 15)는 본 실험에서 사용한 세 사용자들의 읽기량 대비 쓰기량 비율을 보여주고 있다. 쓰기가 차지하는 비율은 전체 워크로드에서 약 25%~35%이였다.



(그림 15) 읽기량 대 쓰기량 비율

5. 결론

우리는 시장에서 구할 수 있는 일반 SSD로서 데스크톱 및 가상 데스크톱 인프라스트럭처 서버에 적합한, 쓰기 캐시의 일관성을 지원하는, SSD 캐시를 제안하였다. 우리의 방식은 다음 사항들을 제공한다. 1) 정전 후에도 캐싱된 데이터가 유지된다. 2) SSD는 더티 데이터를 보유할 수 있다. 즉, 읽기뿐만 아니라 쓰기 성능을 향상시킨다. 3) 정전 및 크래시에 대해서 데이터 일관성과 무결성을 보장한다. 3) 적은 오버헤드를 가진다. 4) 데스크톱 워크로드를 지향한다. 5) 시장에서 구할 수 있는 일반 플래시 기반 SSD를 사용한다. 6) 시근성(recency)과 사용빈도(frequency)를 동시에 고려한 정책을 사용한다.

우리는 지속형 쓰기 캐시를 리눅스에서 구현하였으며 실제 여러 사용자로부터 장시간의 데스크톱 워크로드를 수집하여 성능을 측정하였다. 본 논문은 지속형 쓰기 캐시를 제안함으로써 읽기만 캐싱하는 기술에 비하여 거의 두배에 가까운 성능 향상을 보였다.

▌참고문헌

〔1〕 aDam LeVenthaL, B. Y. "Flash storage memory." Communications of the ACM 51.7 (2008).

〔2〕 Zhang, Yiying, et al. "Warming up storage-level caches with bonfire." Presented as part of the 11th USENIX Conference on File and Storage Technologies, 2013.

〔3〕 Byan, Steve, et al. "Mercury: Host-side flash caching for the data center." , 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST), 2012.

〔4〕 Holland, David A., et al. "Flash caching on the storage client." Proceedings of the 11th USENIX conference on Annual Technical Conference. 2013,

〔5〕 Guerra, Jorge, et al. "Cost Effective Storage using Extent Based Dynamic Tiering." Proceedings of the 9th USENIX conference on File and Storage Technologies. 2011.

〔6〕 EMC. VFCache. http://www.emc.com/storage/xtrem/ xtremcache.htm, 2014,

〔7〕 E. V. Hensbergen and M. Zhao. "Dynamic policy disk caching for storage networking.", Technical Report (RC24123), IBM Research, Nov. 2006.

〔8〕 Saxena, Mohit, Michael M. Swift, and Yiying Zhang. "Flashtier: a lightweight, consistent and durable storage cache." Proceedings of the 7th ACM european conference on Computer Systems. ACM, 2012.

〔9〕 Ouyang, Xiangyong, et al. "Beyond block I/O: Rethinking traditional storage primitives." High Performance Computer Architecture(HPCA), 2011 IEEE 17th International Symposium on. IEEE, 2011.

〔10〕 Megiddo, Nimrod, and Dharmendra S. Modha. "ARC: A Self-Tuning, Low Overhead Replacement Cache." FAST. Vol. 3. 2003.

〔11〕 Koller, Ricardo, et al. "Write policies for host-side flash caches." Proc. FAST. Vol. 13. 2013.

〔12〕 Albrecht, Christoph, et al. "Janus: optimal flash provisioning for cloud storage workloads."

Proceedings of the 2013 USENIX conference on Annual Technical Conference. 2013.

〔13〕 Aguilera, Marcos Kawazoe, et al. "Improving Recoverability in Multi-tier Storage Systems." DSN. 2007.

〔14〕 Oh, Yongseok, et al. "Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems." Proceedings of the 10th USENIX conference on File and Storage Technologies, 2012.

〔15〕 Lee, Donghee, et al. "LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies." IEEE transactions on Computers 50.12 (2001): 1352-1361.

〔16〕 Megiddo, Nimrod, and Dharmendra S. Modha. "ARC: A Self-Tuning, Low Overhead Replacement Cache." FAST. Vol. 3. 2003.

〔17〕 Jiang, Song, et al. "DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality." Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies. Vol. 4. San Francisco, CA, USA, 2005.

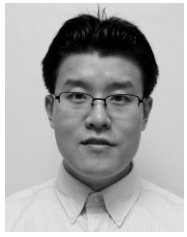
〔18〕 Chen, Peter M., et al. "RAID: High-performance, reliable secondary storage." ACM Computing Surveys (CSUR) 26.2 (1994): 145-185.

저자소개

◆ **백승훈**

- 1997 경북대학교 전자공학과 학사
- 1999 KAIST Electrical Engineering 석사
- 1999~2005 한국전자통신연구원 연구원
- 2008 KAIST Electical Engineering 박사
- 2008~2011 삼성전자 메모리사업부 책임연구원
- 2011~현재 중원대학교 컴퓨터시스템공학과 조교수
- 2013~현재 중원대학교 학술정보원장
- 관심분야: 컴퓨터저장장치, 운영체제, 모바일-클라우드 컴퓨팅

◆ **박기웅**



- 2005년 연세대학교 Computer Science 학사
- 2007년 KAIST Electrical Engineering 석사
- 2012년 KAIST Electrical Engineering 박사
- 2008년 Microsoft Research Asia, Wireless and Networking Gourp, Research Intern
- 2009년 Microsoft Research Redmond, Network Research Group, Research Intern
- 2012년 국가보안기술연구소 연구원
- 2012년 ~ 현재 대전대학교 해킹보안학과 조교수
- 관심분야: 시스템 보안, 모바일-클라우드 컴퓨팅, 보안 프로토콜, 디지털 포렌식 등