리눅스 환경에서Solid-State Disk 성능 최적화를 위한 디스크 입출력요구 변환 계층

Disk I/O Translation Layer for Solid-State Disk Performance Optimization in Linux

김태웅 류준길 박찬익
Taewoong Kim Junkil Ryu Chanik Park

포항공과대학교 컴퓨터공학과
{ehoto, lancer, cipark}@postech.ac.kr

요약

SSD(Solid-State Disk)는 여러 개의 낸드 플래시 메모리들로 구성된 저장 매체로서 뛰어난 읽기 성능을 보인다. SSD는 내장 낸드 플래시 메모리 관리 효율성을 위해 내부적으로 여러 개의 낸드 플래시 페이지들로 구성된 관리 블록을 정의하고 이 단위로 I/O 처리를 한다. 그러나 이러한 구조로 인해 작은 크기의 임의 쓰기는 연속 쓰기에 비해 크게 떨어지는 성능을 보인다. 본 논문에서는 SSD의 작은 크기의 임의 쓰기의 낮은 성능을 개선하기 위해 파일 시스템과 블록 장치 드라이버 사이에 디스크 입출력 요구를 변환시켜 주는 I/O 변환 계층을 추가하였다. 이 계층은 상위에서 내려오는 쓰기 요구를 순차 쓰기 형태가 되도록 변환하여 SSD에 보내준다. 이와 같은 기법을 통해 요구 명령의 크기가 128 KByte 이하인 임의 쓰기 성능이 5~31 배 향상되었다.

Abstract

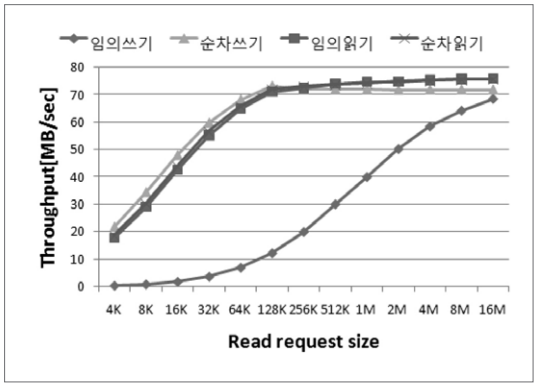
An SSD (Solid-State Disk) is comprised of multiple NAND flash memories for high performance and capacity. SSD management block, which is composed of NAND flash memory pages, is defined as a basic I/O unit in an SSD to manage multiple NAND flash memories efficiently. However, processing in the SSD management block unit instead of small sized NAND flash page unit induces poor performance when dealing small sized write requests. In this paper we propose a method to enhance the poor performance of random small sized write requests in an SSD. The proposed method is implemented as a translation layer, which is between file system and block device driver. Write requests from the upper layer are translated into sequential write requests in the translation layer. As a result, the performance of random write requests smaller than 128 Kbyte is enhanced from 5 to 31 times.

키워드 : SSD, 낸드 플래시 메모리, I/O, 쓰기 성능
Keyword : SSD, NAND flash memory, I/O, Write performance

1. 서론

SSD (Solid-State Disk)는 고성능 고용량을 위해서 여러 개의 낸드 플래시 메모리들로 구성되어 있으며 기존 하드 디스크 드라이브와 동일한 I/O 인터페이스를 제공하는 저장 장치이다. 여러 개의 낸드 플래시 메모리들을 효과적으로 관리하기 위해 SSD는 여러 개의 낸드 플래시 메모리 페이지들로 구성된 SSD 관리 블록을 새로운 내부 I/O단위로 정의하고 사용한다.

낸드 플래시 메모리는 쓰기 작업을 하기 전에 해당 영역의 지움 동작이 선행되어야만 하는 독특한 I/O 특성으로 인해 SSD는 쓰기 요청이 왔을 시 기존의 위치에 데이터를 바로 갱신하지 않고 이미 지움 동작이 행해져 비어있는 SSD 관리 블록에 데이터를 쓴다. 이로 인해 SSD 관리 블록 보다 작은 크기의 쓰기를 처리할 때 기존의 SSD 관리 블록에서 갱신되지 않은 나머지 부분을 새로운 SSD 관리 블록으로 복사하여야 한다. 따라서 작은 크기의 임의 쓰기 성능은 상당히 나쁘다. 반면 순차 쓰기의 경우 SSD 내부 버퍼에 데이터를 모아서 SSD 관리 블록 단위로 한번에 갱신이 일어나므로 데이터 복사 오버헤드가 없어서 우수한 성능을 보인다.



(그림1) SSD의 성능

본 논문에서는 이와 같은 SSD의 특성으로 인해서 나쁜 임의의 작은 크기 쓰기 성능을 향상시킬 수 있는 방법을 제안한다. 이를 위해 파일 시스템과 디바이스 드라이버 사이에 디스크 I/O 요청 변환 계층을 추가하여 임의 쓰기 요청을 순차 쓰기 요청으로 변환 시켜 SSD에 내려주도록 한다. 디스크 I/O 변환이 데이터를 다른 위치에 쓰게 하므로 데이터가 SSD 내에 쓰여진 실제 주소를 관리하여야 한다. 이를 위해 효율적인 주소 맵핑 테이블 관리 기법들을 고안하였다. 이와 같은 I/O 변환을 통해 요구 명령의 크기가 128 KByte 이하인 임의 쓰기 성능이 5~31배 향상되었다.

2. 배경지식

2.1 낸드 플래시 메모리

낸드 플래시 메모리는 페이지 단위 (2KB, 4KB)로 읽기와 쓰기 작업을 하며 지정된 위치에 새로운 데이터를 쓰기 전에 해당 위치에 지움 동작이 이루어져야 한다. 지움 동작 이후에 새로운 데이터를 프로그래밍 동작을 통해 데이터를 쓰게 된다. 지움 동작은 여러 개의 페이지로 구성된 플래시 블록 단위로 이루어지며 읽기나 쓰기에 비해 시간이 오래 걸리는 작업이다. 따라서 낸드 플래시 메모리 전용 파일 시스템 (JFFS2[1], YAFFS[2]) 또는 FTL(Flash Translation Layer)은 기존의 데이터 내용의 변경을 위해 쓰기를 하는 경우 미리 지움 동작을 해둔 빈 페이지들에 내용을 쓰는 out-of-place update 기법을 사용한다.

이와 같은 방법을 사용하기 위해서는 상위에서 요청한 주소의 데이터가 저장된 낸드 플래시 메모리 상에 물리적 위치를 주소 맵핑 테이블에 기록하여야 한다.

차후에 읽기를 수행할 때는 주소 맵핑 테이블을 참조하여 데이터의 낸드 플래시 메모리 상에 물리적 위치를 알아내어 해당 페이지들에서 데이터를 읽어 들이게 된다.

또한 플래시 블록 내의 각 페이지가 유효한 데이터를 담고 있는지 아니면 쓸모없는 과거의 데이터를 갖고 있는 지와 같은 정보를 유지 관리해야 한다. 지움 동작을 효과적으로 수행하기 위해서 여러 플래시 블록에서 유효한 페이지들만 따로 플래시 블록에 모으고 유효하지 않은 페이지들만을 포함한 플래시 블록을 지우는 작업을 garbage-collection이라고 한다. 낸드 플래시 메모리는 플래시 블록마다 지움/프로그래밍 횟수가 제한되어 있어서 사용 수명이 제한되어 있다. 따라서 특정 플래시 블록에만 빈번하게 지움/프로그래밍 작업을 집중하게 되면 그 블록은 수명이 급격히 줄어들게 되고 낸드

플래시 메모리 사용 수명도 줄어들게 된다. 이런 현상을 막기 위해 낸드 플래시 메모리 기반 저장 시스템은 모든 플래시 블록이 골고루 사용되도록 하는 wear-leveling 작업을 한다.

3. 관련 연구

SSD의 자세한 내부 구조는 제조사들이 공개를 꺼리기 때문에 이와 관련된 문서는 공개된 문서는 드물다. 제조사들 마다 고유한 하드웨어 디자인과 FTL알고리즘을 사용하고 있다. 그러나 어느 정도 공통적인 디자인 요소들이 존재하며 Agrawal [3]는 SSD의 성능에 영향을 주는 이런 요소들에 대하여 분석하였다. SSD의 성능은 디스크 인터페이스(SATA, IDE, PCI-Express)나 사용된 플래시 메모리의 성능 이외에도 SSD 내부의 관리 블록의 크기, 플래시 메모리 칩들의 병렬처리 방법, interleaving의 방법 등과 같은 소프트웨어적인 요소가 SSD의 성능에 큰 영향을 준다고 말하고 있다. 여러 가지 요소 중 맵핑의 단위로 사용되는 SSD 관리 블록의 크기는 작은 크기의 쓰기 작업 성능에 큰 영향을 준다. SSD 관리 블록의 크기가 작으면 SSD 내부의 맵핑 테이블의 크기가 커지고 삭제 작업 등 블록 관리가 복잡해지게 된다. 반면 SSD 관리 블록의 크기가 커진다면 내부 관리 작업은 단순화되고 맵핑 테이블 크기가 작아진다. 하지만 SSD관리 블록 보다 작은 크기의 쓰기 요청이 오게 된다면 쓰기 out-of-place update를 하는 과정에서 관리 블록에서 update가 되지 않는 부분을 새로운 관리 블록으로 복사해야 하는 부하가 발생한다. 따라서 관리 블록이 커질수록 작은 크기의 쓰기 요청 성능은 크게 떨어지게 된다.

대용량의 SSD는 약 128KB에서 1MB 정도의 비교적 큰 크기의 SSD 관리 블록을 사용하므로 일반적으로 작은 크기의 쓰기 성능은 크게 떨어지는 현상을 보인다.

그 밖에 SSD 또는 낸드 플래시 메모리 기반의 스토리지 시스템의 성능 개선과 관련된 많은 논문들은 FTL이나 SSD 내부의 구조 변경 알고리즘 변경을 통한 성능 개선 방법을 제안하고 있다. BPLRU[4] 및 CFLRU[5] 등은 SSD내부의 캐시 알고리즘의 개선을 통한 성능 향상 기법을 제안하고 있으며 [6] [7]는 여러 개의 플래시 메모리를 사용한 플래시 스토리지 시스템 상에서의 여러 가지 병렬 처리 기법과 interleaving 기법을 통하여 성능을 향상 방법을 제안하였다.

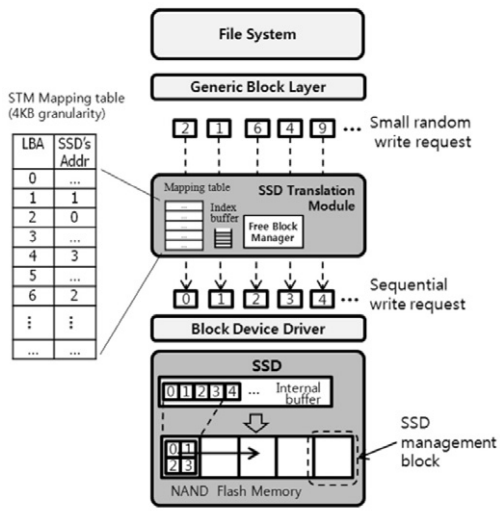
이러한 논문들은 상위 컴포넌트의 최적화 보다는 SSD나 플래시 스토리지 시스템의 디자인 및 사용 알고리즘의 개선에 주로 초점을 맞추고 있다.

4. SSD Translation Module (STM)

4.1 설계

Generic block layer에서 SSD 디스크 드라이버로 내려오는 쓰기 요청들을 순차 쓰기의 형태로 변형하기 위해 그림 2와 같이 두 계층 사이에 SSD Translation Module (STM)을 추가하였다. 순차 쓰기 형태로 쓰기 위치를 변형하고 나면 추후에 다시 이 데이터를 접근하기 위해서 데이터가 실제로 저장된 물리적 위치를 주소 맵핑 테이블 내에 기록해둔다. 읽기 요청이 내려오는 경우 주소 맵핑 테이블을 참조하여 데이터의 위치를 알아낼 수 있게 된다.

효과적인 맵핑 테이블의 업데이트를 위해 STM은 index buffer를 이용한다. Free block manager는 새로 데이터를 쓸 빈 블록을 할당시켜 주는 역할을 한다.



(그림2) SSD Translation Module의 구조

4.2 Free Block Manager

순차 쓰기를 하기 위해서는 SSD 내에 가능한 긴 연속적인 빈 영역이 필요하다. Free block manager은 비어 있는 블록을 관리하고 있으면서 할당을 요청 받을 경우 가능한 긴 연속적인 공간을 할당해준다. Free block manager는 디스크의 공간을 여러 개의 연속적인 디스크 I/O 블록들의 그룹으로 관리한다. 하나의 블록 그룹의 크기는 해당 SSD 제품의 SSD관리 블록의 크기와 같도록 하여서 하나의 SSD 관리 블록에 속한 블록들을 먼저 할당하도록 해준다.

Free block manager는 각 Free 블록 그룹의 descriptor에 그룹 내에 블록들이 유효한 데이터를 소유하고 있는지 또는 빈 블록인지를 bitmap을 이용해 기록하고 그룹 내에 남아 있는 free 블록의 개수를 저장하고 있다.

쓰기 요청이 내려왔을 때 free block manager는 가장 많은 free 블록을 가진 블록 그룹을 current block group으로 선정하고 bitmap을 참조하여 순서대로 요청 받은 만큼의 free 블록을 할당해 준다.

4.3 Address Mapping Table

상위에서 내려오는 입출력 요구는 STM에서 순차 쓰기 형태로 변형된다. 이로 인해 SSD 내부에는 상위 계층이 요구한 위치에 데이터가 배치되지 않는다.

하지만 기록된 데이터를 상위 계층이 다시 읽어 들이기 위해서는 STM은 상위 계층에서 쓰기 요청을 했었던 데이터 배치 상태의 디스크 이미지를 제공하여야 한다. 따라서 상위 계층에서 쓰기를 요청한 주소(LBA: Logical Block Address)와 실제로 SSD 내에 데이터를 쓴 물리적 위치(PBA: Physical Block Address) 간의 맵핑 정보를 주소 맵핑 테이블에 기록한다.

STM이 메인 메모리에 적재 될 때 SSD로부터 super 블록을 읽어 들인다. 여기에는 SSD내에 주소 맵핑 테이블이 저장된 위치와 데이터 블록이 시작되는 위치 등이 기록되어 있다. Super block의 정보를 바탕으로 주소 맵핑 테이블을 메인 메모리로 읽어 들여서 사용하게 된다. STM은 쓰기 요청이 내려올 때 마다 데이터는 항상 새로운 위치에 저장되고 그에 따라 주소 맵핑 테이블은 변경된다. 메모리에 존재하는 주소 맵핑 테이블에서 변경된 부분을 매번 SSD에 저장하는 것은 작은 크기의 임의 쓰기의 형태의 워크로드를 발생시켜 전체적인 성능을 떨어뜨리게 된다. 이 문제를 해결하기 위해 STM에서는 index buffer를 이용한다. 또한 crash recovery를 위해 index log 블록을 이용하게 된다.

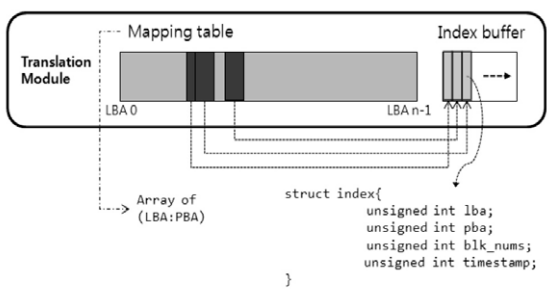
4.3.1 Index Buffer

주소 맵핑 테이블의 변경은 바로 SSD내의 주소 맵핑 테이블에 적용되지 않고 메인 메모리에 존재하는 index buffer에 순차적으로 변경 내용이 기록된다.

그림 3과 같이 업데이트 되어야 할 LBA와 실제로 데이터가 저장된 위치인 PBA, 블록의 개수 정보 그리고 타임 스탬프로 구성된 index의 형태로 index buffer에 기록한다. Index buffer내에 index가 일정 개수 이상 쌓이거나 일정 시간이 지나게 되면 index buffer 내의 index들을 이용하여 메모리내의 주소 매핑 테이블과 SSD 내의 주소 매핑 테이블을 동기화 시킨다.

4.3.2 Address Mapping Table Synchronization

Index buffer가 거의 찼을 경우 또는 일정 시간 주기로 index buffer 내의 index들을 처리해 주어서 index buffer를 비워주어야 한다. Index buffer 내에 index를 차례대로 SSD 내의 주소 맵핑 테이블에 적용한 후 index buffer에서 index를 지우게 된다.



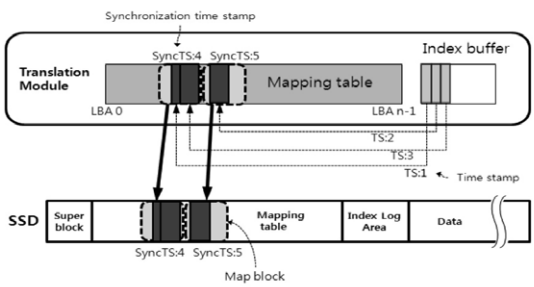
(그림3) Index buffer와 테이블 변경

SSD의 주소 맵핑 테이블 갱신은 효과적인 업데이트를 위해 그림 4처럼 여러 개의 섹터로 구성된 맵 블록의 단위로 업데이트 된다. 각 맵 블록은 갱신된 시간을 기록하는 싱크 타임 스탬프 값을 메모리에 유지하고 있다.

맵 블록을 갱신하기 전에 맵 블록의 싱크 타임 스탬프와 적용할 index의 타임 스탬프 값을 비교하여본다.

Index의 타임 스탬프가 더 최근의 값이라면 index와 관련된 맵 블록의 싱크 타임 스탬프를 현재 시각으로 수정하고 맵 블록을 메모리 내의 주소 맵핑 테이블에서 읽어와 SSD 내의 주소 맵핑 테이블에 쓰게 된다.

반대로 만약 맵 블록의 싱크 타임 스탬프가 index의 타임 스탬프 보다 최근의 시간을 나타낸다면 해당 index는 적용하지 않고 다음 index를 처리한다. 이것은 해당 맵 블록이 다른 index를 처리할 때 이미 처리됐음을 뜻하기 때문이다. 처리가 끝난 index는 index buffer에서 지운다. 그림 4의 예를 보면 index buffer 내에 첫 번째 index를 처리하며 해당 맵 블록은 최신 값으로 갱신하고 맵 블록의 싱크 타임 스탬프를 4로 갱신한다. Index buffer내 두 번째 index를 처리한 후 세 번째 index를 처리할 때 index의 타임 스탬프는 3을 갖고 있고 이 index와 관련된 맵 블록은 4라는 싱크 타임 스탬프를 갖고 있는 상태가 되어 싱크 타임 스탬프의 값이 더 큰 상황이 된다. 싱크 타임 스탬프 값 4는 첫 번째 index를 처리할 때 갖게 된 값이다. 즉 첫 번째 index를 처리할 때 세 번째 index도 함께 처리됐음을 나타내는 것이다. 따라서 세 번째 index와 관련된 맵 블록을 다시 갱신하지 않고 넘어가게 된다. 이와 같이 index buffer를 사용함으로써 그림 3.7에서는 3회의 주소 맵핑 테이블 갱신을 2회 쓰기 작업으로 끝낼 수 있게 된다.



(그림4)주소 맵핑 테이블 동기화

4.4 Crash Recovery

시스템 크래시나 정전과 같이 STM은 예기치 않은 상황에서 종료될 수 있다. 종료되기 전에 실행 중이던 작업을 완료하지 못하는 경우 consistency가 파괴되거나 데이터를 손실할 수 있다.

예를 들어 디스크에 데이터를 새로 쓴 후 디스크 내의 맵핑 테이블에 갱신이 이루어 지지 않는 경우 맵핑 테이블은 기존의 데이터를 가리키고 있으므로 새로 쓴 데이터를 손실하게 된다. STM에서는 index log block을 이용하여 이러한 문제를 해결하고 있다. 주기적으로 또는 일정 개수의 index가 채워질 때 마다 index buffer 내의 index들을 index log block의 형태로 SSD의 index log 영역에 쓰게 된다. SSD 내에 index log 영역은 한정되어 있으므로 이 영역이 index log block들로 모두 차게 되면 index log block을 삭제해주어야 한다. Index log block을 삭제하기 위해서는 해당 index log block과 연관된 index buffer내의 모든 index를 먼저 SSD 내의 주소 맵핑 테이블에 적용시켜 관련 index를 더 이상 유지할 필요가 없도록 하여야만 한다. 시스템이 정상적인 종료를 하게 되면 STM을 메모리에서 내릴 때 STM은 주소 맵핑 테이블을 동기화 시키고 index log 영역 내의 모든 index log block을 삭제하는 과정을 거친다. 만약 시스템이 비정상적인 종료 후 다시 시스템을 가동하고 STM을 실행하게 되면 STM은 먼저 SSD 내의 주소 맵핑 테이블을 읽어 들이고 index log영역에 유효한 index log block이 존재하는지 살펴본다. 만약 index log block이 존재한다면 이는 STM의 비정상적인 종료를 했었다고 판단하고 복구 작업을 실행한다.

STM이 재가동하게 되면 index log block을 모두 읽어 들여 index buffer를 재구성하고 가장 오래된 index부터 주소 맵핑 테이블이 다시 적용하는 roll-forward 작업을 진행한다. 주소 맵핑 테이블 동기화 작업을 통해 SSD내의 주소 맵핑 테이블을 복구함으로써 복구를 끝마친다.

이와 같은 복구 작업 덕분에 index log block에 기록된 맵핑 변경과 관련된 데이터는 잃지 않는다. 하지만 메인 메모리에 존재하는 index buffer에만 기록되고 디스크의 index log block에 저장되지 않은 테이블

변경 사항은 손실되게 된다. 따라서 index log block 기록 주기를 길게 설정하게 되면 비정상적인 STM 종료 시 더 많은 데이터를 손실할 가능성이 높다. 그러나 index log block 기록 주기를 짧게 설정한다면 index log 영역이 빨리 차게 됨으로 주소 맵핑 테이블의 동기화 작업이 잦아지고 이에 따른 부하로 전체적인 성능 저하가 나타나게 된다. 성능과 데이터 안정성은 이와 같은 트레이드오프(trade-off)관계를 가지므로 index buffer의 크기나 플러쉬 주기는 이를 고려해서 설정해야 한다.

5. 성능 평가

본 논문에서 제안한 STM의 성능을 평가고 index buffer와 주소 맵핑 테이블 동기화와 같은 요소들이 주는 영향에 대해서 실험해 보았다. 성능 평가에는 펜티엄4 2GHz CPU, 1 GB 메인 메모리 그리고 Mtron MSD-SATA 32GB SSD를 사용하였으며 리눅스 커널 2.6.23.1에서 테스트하였다. Mtron MSD 32GB SSD는 1MB의 SSD 관리 블록 크기를 갖는다. 따라서 블록 그룹의 크기는 1MB가 되도록 하였으며 STM 의 주소 맵핑 테이블은 4KB의 엔트리를 이용하였다. STM 블록 디바이스 의 성능 평가를 위해서 IOmeter [8] 벤크 마크 도구를 사용하였다.

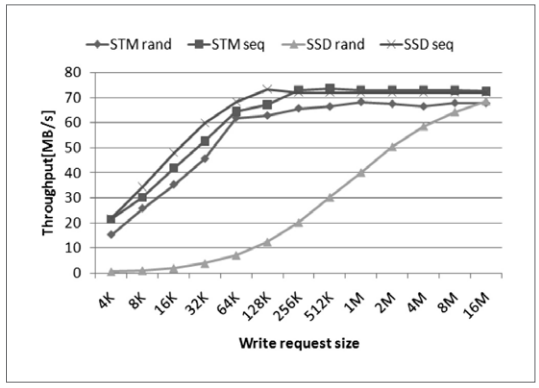
5.1 I/O 성능

Index buffer에 300개의 index가 새로 쓰여질 때마다 이 index들를 index log block으로 SSD의 index log 영역에 쓰도록 하였다. Index log 영역에는 64개의 index log block이 들어갈 수 있는 크기로 설정하여 64개의 index log block이 차면 주소 맵핑 테이블의 동기화가 발생하도록 하였다.

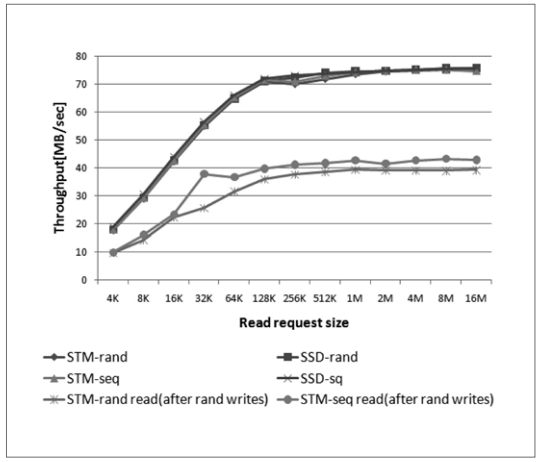
실험 결과 STM을 사용하였을 때의 임의 쓰기 성능은 그림 5와 같이 SSD의 순차 쓰기와 유사한 성능을 보이고 있다. STM을 사용하지 않고 SSD만을 사용했을 때 보다 임의 쓰기 성능은 크게 향상되었다. STM 사용 시 순차 쓰기 성능은 SSD의 순차 쓰기보다 약간 떨어지고 있다. 이것은 STM이 주소 맵핑 테이블을 갱신하고 index log block을 쓰는 부하로 인해 발생한 것으로 보인다.

그림 6은 STM의 읽기 성능을 나타낸 것이다. 순차 쓰기가 이루어 진 상황에서 읽기는 SSD의 성능과 크게 다르지 않았다. STM의 주소 맵핑 테이블 관리나 그 밖의 부하가 크지 않음을 알 수 있다. 그러나 임의 쓰기가 이루어진 뒤 읽기 성능은 SSD 보다 떨어진다. 이것은 임의 쓰기로 인해 STM이 데이터를 배치를 상위에서 인식하는 주소인 LBA순서대로 하지 않았기 때문이다.

이런 상태에서의 읽기 요청은 여러 개의 읽기 요청으로 분산되어 SSD에 전달되고 성능 저하를 일으키게 된다.



(그림5) STM의 쓰기 성능



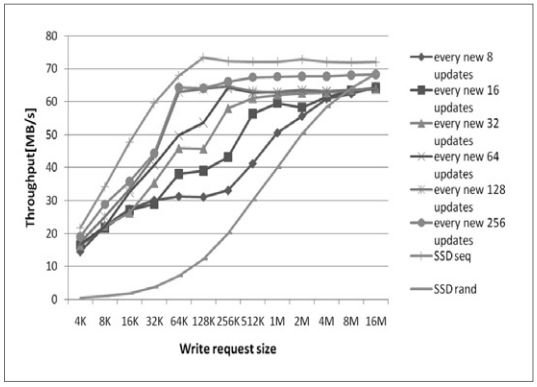
(그림6) STM의 읽기 성능

5.2 Index Log Block 기록 빈도에 따른 성능

그림 7은 주소 맵핑 테이블의 동기화 기능을 끈 채 index log block의 기록 빈도에 따른 성능을 측정 결과이다. 8번의 주소 맵핑 테이블이 변경될 때마다 index log block을 SSD에 기록하게 했을 때의 임의 쓰기 성능부터 256번의 주소 맵핑 테이블이 변경될 때마다 index log block을 쓰는 경우까지 성능을 측정하였다.

성능 측정 결과 그림 7과 같이 index log block을 낮은 빈도로 쓸수록 전반적인 성능이 증가하는 것을 볼 수 있다. 특히 32KB에서 512KB의 임의 쓰기 요청에서 index log block 기록 빈도에 큰 영향을 받았다.

큰 크기의 쓰기 요청 실험에서는 index log block 기록 이 전체 쓰기 양에 비해 비중이 작기 때문에 빈도에 큰 영향을 받지 않았다. 반면 작은 크기의 쓰기 요청에서 index log block의 기록은 큰 비중을 차지함에도 불구하고 기록 빈도의 영향을 크게 받지 않았다. 비록 빈도가 높더라도 index log block이 SSD 내의 index log 영역이라는 한정된 영역에 기록되고 자주 써지기 때문에 SSD의 내부 쓰기 캐쉬의 효과가 작용한 것으로 보인다.



(그림7) Index log block 기록 빈도에 따른 성능

Index log block을 자주 기록하는 것은 쓰기 성능을 떨어뜨리는 요인이 되지만 비정상적인 종료 시에 데이터 손실을 줄일 수 있다. 따라서 index log block은 성능과 신뢰도를 고려해서 결정되어야 한다.

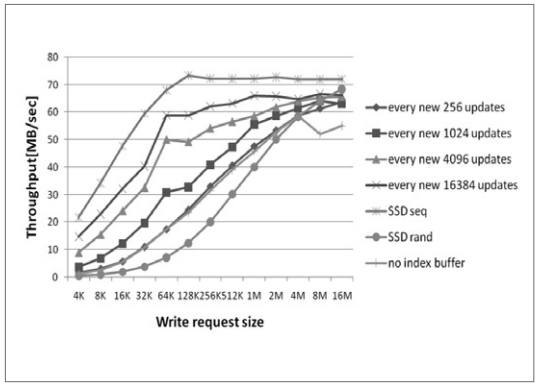
5.3 주소 맵핑 테이블 동기화 빈도에 따른 성능

그림 8과 같이 주소 맵핑 테이블의 동기화를 드물게 해주는 것이 좋은 성능을 보였다. 동기화를 하는 주기가 길어지면 index buffer에는 더 많은 index가 쌓이게 된다. 따라서 하나의 맵 블록을 업데이트 하는 것으로 더 많은 index가 처리될 확률이 높아진다.

반면 index buffer에는 많은 index가 쌓이게 되므로 이를 모두 동기화 시키는 데에는 비교적 긴 시간이 필요해진다. 따라서 동기화 중에 내려오는 입출력 요청은 바로 처리되지 못해 최대 반응 시간이 길어지게 된다.

5. 결론 및 향후 연구

본 논문에서는 SSD의 특성을 이용하여 상위 계층이나 하드웨어적인 수정 없이 새로운 계층의 추가로 SSD의 성능을 향상 시킬 수 있는 기법을 소개하였다.



(그림8) 주소 맵핑 테이블 동기화 빈도에 따른 성능

SSD로 내려오는 쓰기 요청들을 가능한 순차 쓰기 형태로 변환하여 SSD의 순차 쓰기와 비슷한 성능을 이끌어 내었다. 메모리 내의 주소 맵핑 테이블과 SSD 내의 주소 맵핑 테이블 간의 효과적인 동기화를 위하여 index buffer를 이용하였고 index log block을 통해 시스템의 비정상적인 종료가 발생하더라도 주소 맵핑 테이블을 복구할 수 있는 기법을 소개하였다.

STM과 같이 out-of-place update 방식을 이용하는 기법은 데이터들이 디스크에 LBA 순으로 배열되지 않기 때문에 임의 쓰기를 수 차례 반복한 후에는

내부의 데이터들은 순서가 뒤섞이게 된다. 따라서 차후 읽기를 하는 경우 읽기 요청이 조각나는 경우가 다수 발생해 성능 저하를 일으킨다. 따라서 유휴 시간에 이를 LBA 순으로 정렬시켜주는 작업이 이루어진다면 읽기 성능의 향상에 도움이 될 것이다. 또한 SSD의 최적화는 generic block layer와 블록 디바이스 드라이버와 같은 다양한 계층에서도 SSD에 최적화가 가능한 부분들을 찾아보고 개선할 필요가 있다.

Acknowledgment

"본 연구는 지식경제부 및 정보통신연구진흥원의 대학 IT 연구센터 지원 사업 (IITA-2009-C1090-0902- 0045)과 두뇌한국 21 사업의 지원으로 수행되었음"

참고문헌

[1] D. Woodhouse, 2001, JFFS : The Journalling Flash File System

[2] YAFFS: Yet Another Flash File System. http://www.aleph1.co.uk/yaffs/yaffs.html

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In Proceedings of the USENIX Annual Technical Conference (USENIX '08), pages 57-70, June 2008.

[4] H. Kim and S. Ahn. A Buffer Management Scheme for Improving Random Writes in Flash Storage. In Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST' 08), pages 239-252, 2008.

[5] Seon-Yeong Park, Dawoon Jung, Jeong-Uk Kang, Jin-Soo Kim, and Joonwon Lee. CFLRU: a replacement algorithm for flash memory. In CASES '06: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pages 234-241, New York, NY, USA, 2006. ACM.

[6] L.P. Chang, T.W. Kuo, An adaptive striping architecture for flash memory storage systems of embedded systems, in: Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS' 02), 2002, pp. 187-196.

[7] J. U. Kang, J. S. Kim, C. Park, H. J. Park, and J. W. Lee, A Multi-Channel Architecture for High-Performance NAND flash-based Storage System, Journal of System Architecture, Vol 52, Issue 9, 2007.

[8] Intel. Iometer, performance analysis tool. http://www.intel.com/design/servers /devtools/iometer/.

저자소개

◆김태웅

- 2007년 포항공과대학교 컴퓨터공학과 학사
- 2009 포항공과대학교 컴퓨터공학과 석사
- 관심분야 : Storage Systems, Embedded systems
- Email : ehoto@postech.ac.kr

◆류준길

- 2002년 포항공과대학교 컴퓨터공학과 학사
- 2002~현재 포항공과대학교 컴퓨터공학과 석박사 통합과정
- 관심분야 : Storage Systems
- Email : lancer@postech.ac.kr

◆박찬익

- 1983년 서울대 전기공학 학사
- 1985년 KAIST 전자전기공학 석사.
- 1988년 KAIST 전자전기공학 박사.
- 1991~1992 IBM Thomas J. Watson Research Center visiting scholar
- 1999~2000 IBM Almaden Research Center visiting professor
- 1989~현재 포항공과대학교 컴퓨터공학과 교수.
- 관심분야 : Storage Systems, Embedded systems, Pervasive Computing
- Email : cipark@postech.ac.kr