SSD에 대한 리눅스 페이지 캐시의 성능 평가

이주환[O]*, 김정현*, 김홍준*, 이재진*, 최재영**, 임선영**
* 서울대학교 컴퓨터공학부, ** 삼성전자
{joohwan, junghyun, hongjune}@aces.snu.ac.kr, jlee@cse.snu.ac.kr,
{anton, sylim829.lim}@samsung.com

Performance Evaluation of Linux Page Cache on Solid-State Disk

Joohwan Lee[O]*, Junghyun Kim*, Hongjune Kim*, Jaejin Lee*, Jaeyoung Choi**, Sunyoung Lim**
* School of Computer Science and Engineering, Seoul National University
** Samsung Electronics, Korea

요 약

플래시 메모리의 집적도가 높아지고 가격이 저렴해 짐에 따라 낸드 플래시 기반의 SSD 의 사용이 확산되고 있다. 플래시 메모리 기반 SSD는 기존의 하드디스크와 비교하여 여러 가지 장점을 가지지만 덮어쓰기가 불가능한 특성상 쓰기 공간 확보를 위해 가비지 컬렉션이 수행되어야 하는 단점을 가진다. 이러한 단점을 개선하기 위해 다양한 연구들이 제안되었다. 이 중, 운영체제의 페이지 캐시에 대한 연구가 상반된 주장을 보이고 있는 점[11,12,13]에 착안하여 실험을 통해 이를 재확인하였다. 실험 결과, 큰 용량의 페이지 캐시가 SSD를 스토리지로 갖는 시스템에서 파일 입출력 성능을 크게 향상시키는 것을 확인할 수 있었다.

1. 서론

CPU의 성능은 지속적으로 향상되었지만, 전체 시스템의 성능은 CPU의 성능 향상에 비례하여 향상되지 않고 있다. 이는 여러 가지 이유가 있지만 CPU가 지속적으로 계산을 하지 못하고 데이터를 얻기 위하여 CPU가 기다리는 오버헤드가 CPU의 성능이 향상될수록 점차로 증가하고 있기 때문이다. 특히 파일 입출력의 느린 속도가 점차로 시스템 성능의 주된 병목으로 작용하고 있다.

최근 플래시 메모리의 집적도가 높아지고 가격이 저렴해 짐에 따라, 낸드 플래시 기반의 SSD의 사용이 확산되어 기존의 하드디스크를 대체하고 있다. SSD는 하드디스크와는 달리 외부 충격에 강하고 전력소모가 적고 휴대성에 장점을 가지며 읽는 속도가 매우 빠르고 특히 하드 디스크와 비교하여 랜덤 읽기에 대해서도 매우 빠른 속도를 보인다. 하지만 쓰기 속도가 읽기 보다 느리고 특히 랜덤 쓰기는 연속적 쓰기에 비하여 매우 느린 속도를 보인다[1].

SSD의 성능, 특히 느린 랜덤 쓰기 성능을 향상시키기 위하여 다양한 솔루션이 제안 되었다. 해결책으로 운영체제에서의 캐싱, 플래시에 특화된 파일 시스템, SSD에서의 쓰기 캐싱, 병렬 블록 삭제의 솔루션이 제시되었다[2].

운영체제에서의 캐싱은 운영체제가 디스크로의 파일 입출력 요청에 대해서 메인 메모리에 캐시를 제공하는 것이다. 이를 통해 애플리케이션은 파일 입출력을 빠르게 처리할 수 있다. 플래시에 특화된 파일 시스템은 플래시의 물리적 특성을 고려한 파일 시스템이다. 그 예로 JFFS와 YAFFS등이 존재한다. 이와 같은 파일 시스템은 플래시의 느린 랜덤 파일 쓰기를 파일 시스템에서 고려하여 이를 해결하고자 한다. 하지만 기존의 파일 시스템을 사용하지 못한다는 단점이 있다.

일부 드라이브는 쓰기요청에 대하여 SSD 내부에 존재하는 메모리 버퍼를 사용한다. 이는 운영체제의 캐싱과 유사하다. 하지만 급작스런 전원 차단의 경우에 쓰기 요청을 내부의 배터리 등을 이용하여 보호하는 차이가 있다. 하지만 버퍼에 사용되는 메모리의 가격 등의 문제로 큰 용량을 확보하는 것의 한계가 존재한다. 따라서 입출력 집중적인 애플리케이션에 대해서는 금방 캐시 사이즈를 초과하는 문제가 나타난다. 병렬적으로 블록을 삭제하는 솔루션은 SSD 의 낮은 랜덤 쓰기 성능이 가비지 컬렉션의 오버헤드에 따른다는 것에 착안하여 여러 개의 블록을 동시에 지워서 블록 삭제의 오버헤드를 최소화하고자 한다[2].

본 논문은 SSD의 성능을 개선하기 위한 여러 솔루션 중 운영체제에서의 페이지 캐시에 대하여 고찰한다. 기존의 SSD의 성능 향상을 위한 연구에서는 대용량의 페이지 캐시에 대한 충분한 연구가 이루어지지 않았다. 따라서 본 논문에서는 페이지 캐시가 효과적으로 파일 입출력을 향상시키는 워크로드와 페이지 캐시의 크기에 따른 성능 차이에 대해 분석하였다. 분석 결과 큰 용량의 페이지 캐시가 시스템 성능을 크게 향상시키는 것을 확인할 수 있었다. 2장에서는 플래시 기반의 SSD의 구조를 설명한다. 3장에서는 리눅스의 페이지 캐시의 동작과 페이지 캐시의 크기에 따른 시스템 성능에 대한 차이를 설명하고 4장에서는 기존의 연구에 대해 설명한다. 5장에서는 SSD를 이용한 실험과 그 결과를 분석하며 6장에서 결론을 맺으며 마무리 한다.

2. SSD

플래시 메모리는 여러 개의 블록으로 이루어 지며 각 블록은 일반적으로 64개의 페이지로 구성된다. 페이지는 읽기/쓰기의 기본 단위 이며 일반적으로 4KB의 크기로 구성되고 블록은 지우기의 기본 단위이다. 하드디스크의 경우에는 원래 데이터가 저장 된 곳에 업데이트가 가능하지만 플래시는 원래 저장된 곳에 업데이트(in-place update)가 불가능하여 새로운 데이터를 저장할 때는 기존의 데이터를 지우고 쓰거나 다른 빈 공간에 데이터를 써야 한다.

플래시는 각 연산에 대해 효율적으로 성능을 나타낼 수 있도록 디스크 외부로 보이는 블록 위치와 물리적인 플래시 블록을 다르게 처리 한다. 플래시 메모리 주소 변환 계층(FTL)은 플래시의 논리적 블록과 물리적 블록간의 사상을 처리하는 역할을 한다. FTL 기법들은 주소 사상 단위에 따라 페이지 기반, 블록 기반, 그리고 혼합 기법으로 구분할 수 있다[3]. 페이지 기반 FTL은 사상 단위로 플래시의 읽기/쓰기 기본 단위인 페이지를 사용한다. 블록 기반 FTL은 사상의 기본 단위로 플래시의 삭제연산의 기본 단위인 블록을 사용한다. 혼합 기법은 위 두 방법의 장점만을 살린 방법으로 로그 버퍼 기반의 FTL이 대표적이다. 혼합 기법은 적은 메모리 자원으로 사상 정보를 유지하면서 좋은 성능을 보이므로 SSD에서 많이 사용된다.

로그 버퍼 기반의 FTL 기법은 플래시 블록들을 데이터 블록과 로그 블록으로 나눈다[4]. 데이터 블록은 블록 단위로 사상되며 데이터 저장을 위해 사용되고 로그 블록은 쓰기 요청 시 요청된 데이터를 쓰기 위한 공간으로 페이지 단위로 사상된다. SSD에서는 드라이브 외부로 보이는 스토리지 용량보다 더 큰 용량의 플래시 메모리를 가지고 외부에 보이지 않는 플래시를 로그 블록으로 할당하여 쓰기 요청을 처리한다. 하지만 플래시의 특성 상 다시 쓰기가 불가능하고 로그 블록의 크기에 한계가 존재하기 때문에 SSD는 새로운 데이터의 쓰기를 받아들일 수 있는 영역을 지속적으로 확보할 필요가 있다. 이를 위해 블록 합병을 통한 가비지 컬렉션을 통해 새롭게 쓰기 버퍼 영역을 확보한다. 가비지 컬렉션은 업데이트 후에 불필요한 데이터를 저장하고 있는 페이지를 삭제하여 다시 쓰기가 가능하게 하는 과정이다. FTL은 쓰기가 가능한 페이지가 없거나 특정 포인트 이하로 내려가면 가비지 컬렉션을 수행한다.

블록 합병에는 스위치 합병, 부분 합병, 전체 합병이 존재한다. 스위치 합병은 희생 블록이 모두 유효하지 않은 데이터를 가지고 있는 경우로 기존의 유효한 데이터를 복사하는 오버헤드가 존재하지 않고 단순히 기존의 데이터 블록을 삭제하고 사상 테이블을 수정하는 것만으로 완료된다. 부분 합병과 전체 합병은 희생 블록이 유효한 데이터를 가지고 있어 기존의 데이터를 새롭게 할당될 데이터 블록에 카피하는 오버헤드가 존재한다. 따라서 가비지 컬렉션 시 스위치 합병이 많이 일어날수록 SSD가 좋은 쓰기 성능을 보인다. 순차 쓰기의 경우 스위치 합병이 많이 일어나지만 랜덤 쓰기의 경우 전체 합병이 많이 일어나게 된다.

가비지 컬렉션이 매우 큰 수행시간을 가지므로 프로세스의 파일 입출력 수행 중 가비지 컬렉션의 횟수를 최대한 줄일 때 SSD 의 성능이 좋아질 수 있음을 알 수 있다. 프로세스의 수행 중 가비지 컬렉션이 발생하면 CPU 는 가비지 컬렉션이 완료될 때까지 기다려야 하므로 이는 애플리케이션 성능에 큰 오버헤드로 나타날 수 있다. SSD 로의 요청이 적게 이루어지는 구간에 가비지 컬렉션을 통해 지속적으로 로그 블록을 확보하여 프로세스의 쓰기 요청 중간에 가비지 컬렉션이 발생하지 않으면 SSD 는 좋은 성능을 보일 것이다. SSD 의 이러한 특성은 log structured file system 과 유사하다[5].

3. 페이지 캐시

사용자의 디스크 블록의 액세스 패턴에는 지역성이 존재한다. 컴퓨터를 사용할 때 사용자는 특정 작업을 하게 되고 따라서 동일한 파일을 연속적으로 읽게 된다. 따라서 디스크 블록의 정보를 메모리 공간에 저장해두면 파일 입출력 작업 시에 응답시간을 줄일 수 있다.

리눅스는 디스크의 느린 입출력을 보완하기 위해 페이지 캐시를 사용한다. 페이지 캐시는 액세스한 파일을 운영체제의 페이지 단위로 분할하여 메인 메모리에 유지하는 버퍼이다. 페이지 캐시는 일반적으로 커널의 메모리 관리 루틴과 함께 구현되며 애플리케이션에는 투명하게 구현된다. 페이지 캐시에 저장된 페이지는 커널의 페이지 교체 정책에 따라 희생될 수 있다. 리눅스에서 파일 읽기를 하면 해당 데이터는 페이지 캐시로 복사된 후 사용자 어드레스 스페이스로 복사된다. mmap 시스템 콜의 경우에는 사용자 어드레스 스페이스로 복사되지 않고 페이지 캐시의 페이지가 해당 프로세스의 페이지 테이블에 직접 사상된다[6].

사용자의 파일 액세스의 지역성으로 인해 페이지 캐시는 빠른 파일 입출력을 가능하게 한다. 응용프로그램의 파일 읽기 요청은 대부분 디스크로 도달하지 않고 페이지 캐시를 통해 만족된다. 또한 페이지 캐시는 빠른 파일 쓰기를 가능하게 한다. 응용 프로그램의 파일 쓰기 요청이 발생하면 운영체제는 해당 블록이 페이지 캐시에 존재하는 지를 검색하고 만약 해당 블록이 페이지 캐시에 존재하면 응용프로그램의 쓰기요청은 페이지 캐시에 쓰여지는 것으로 완료된다. 따라서 응용프로그램은 파일이 디스크에 쓰여질 때까지 기다리지 않아도 된다. 또한 파일 쓰기단위가 운영체제의 페이지 크기와 일치하면 디스크로부터의 읽기가 필요하지 않아 페이지 캐시에 쓰는 것으로 파일 쓰기가 가능해진다. 페이지 캐시를 통해 사용자에게 파일 쓰기에서 발생할 수 있는 지연(하드디스크의 경우에는 탐색시간, SSD의 경우에는 가비지 컬렉션 등)을 숨길 수 있다.

커널의 기존의 페이지 교체 정책(LRU)을 수정하여 파일 입출력 성능을 향상시키고자 하는 연구가 존재하여 ARC, MQ, 2Q, LRU-2, LRFU, LIRS 등이 제안되었다[7]. 하드디스크에서 페이지 캐시가 히트되는 확률을 높이고자 LRFU와 LIRS가 제안되었다. LRFU는 최근에 액세스 했는지의 여부와 빈도를 함께 고려하여 캐시 적중률을 향상시키고자 하였다[8]. LIRS는 IRR(inter-reference recency)를 이용하여 캐시 적중률을 향상시키고자 하였다[9]. CFLRU[10] 은 기존의 하드디스크의 캐시 정책이 캐시 적중률을 높이고자 했던 데 반해 플래시에서 각 연산이 가지는 오버헤드가 다른 것에 착안한 방법이다. 하드디스크는 연산의 종류에 상관없이 비용이 동일하다. 따라서 버퍼 영역에 보관된 페이지 중 교체 대상 페이지를 선정할 때 연산의 종류는 무관하다. 하지만 플래시는 쓰기 연산이 읽기 연산에 비해 느리고, 쓰기 연산으로 인하여 가비지 컬렉션이 발생할 수 있다. CFLRU는

캐시를 Working First Region과 Clean First Region으로 분리하여 수정된 페이지에 대해 플래시에 쓰는 시점을 늦춘다.

페이지 캐시가 효과적으로 동작하기 위해서는 캐시의 크기가 커야 할 필요성이 존재한다. 페이지 캐시의 크기가 너무 작으면, 다시 이용될 데이터임에도 불구하고 새로 읽히는 데이터를 위해서 캐시에서 비워질 가능성이 높다. 따라서 그로 인해 반복적인 디스크 입출력이 발생할 수 있다. 이와 같은 디스크 입출력은 페이지 캐시의 공간이 충분하였다면 발생하지 않는다. 최근 대용량 멀티미디어 파일과 같이 대용량 파일의 사용이 빈번해지고 있다. 따라서 페이지 캐시의 크기가 사용자의 요구에 따라 커져야 함을 알 수 있다. 따라서 리눅스는 동작중인 프로세스에 할당되지 않은 모든 물리 메모리를 일반적으로 페이지캐시로 이용하고 있다. 하지만 고정된 사이즈의 메인 메모리에서 페이지 캐시의 크기가 너무 크면 프로세스에 할당된 메모리의 부족을 야기하여 프로세스 페이지의 많은 스왑을 발생시켜 프로세스의 수행이 늦어질 수 있다. 리눅스는 자동적으로 프로세스들이 많은 메모리를 필요로 할 때는 페이지 캐시의 크기를 줄여서 이 문제를 해결하고 있다. 이와 같은 상반된 요구에 따라 대용량의 메인 메모리로 시스템을 구성하는 것이 필요하다는 결론을 내릴 수 있다.

4. 관련 연구

사용자의 파일 입출력 패턴에 대해서는 기존의 여러 연구가 존재한다. 하지만 큰 용량의 페이지 캐시가 필요한지에 대해서는 각각의 의견이 다르다. 큰 용량의 페이지 캐시가 디스크로의 파일 입출력을 효과적으로 감소시킬 수 있을 것이라는 의견도 있지만, 캐시 사이즈가 일정 사이즈 이상이 되면 더 이상 캐시 사이즈가 커지는 것이 무의미하다는 의견도 존재한다.

Ousterhout[11] 등은 UNIX 4.2 BSD 파일시스템에서 사용자의 파일 액세스 형태와 파일 시스템의 캐시 동작을 분석하였다. 이 논문에서는 사용자의 파일 액세스의 시간적 지역성과 공간적 지역성으로 인해 페이지 캐시가 디스크 액세스를 효과적으로 줄인다고 주장한다. 대부분의 파일 읽기가 순차적이고 대부분의 새로 쓰여진 파일은 덮어쓰기로 인해 금방 삭제되었다고 주장한다. 이들의 결론에 따르면 대용량의 캐시를 통해 대부분의 읽기 요청이 캐시로 만족될 것이고 대부분의 쓰기 요청은 덮어쓰기로 인해 삭제될 것이므로 대용량의 캐시를 통해 디스크로의 입출력 요청이 크게 줄어들 것이라고 주장하고 있다.

반면 Chris Ruemmler[12]와 Drew Roselli[13]등은 대용량의 캐시가 효과적이지 않다고 주장하고 있다. [12]에 따르면 작은 양의 캐시로도 충분하며 캐시의 크기를 증가시킬 때 캐시 미스율의 감소가 매우 작았다고 주장하였다. [13]에 따르면 평균 블록의 생존시간이 대부분 파일 시스템의 쓰기 지연인 30초보다 길어서 쓰기 요청이 덮어쓰기로 인해 삭제되는 비율이 낮다고 주장하였다.

기존의 연구가 상충된 주장을 함에 따라 대용량의 캐시가 효과적인지에 대해 검증을 할 필요가 존재하였다. SSD의 성능 향상을 위한 기존 연구에서도 대용량의 캐시에 대한 충분한 연구가 진행되지 않았다. 따라서 우리는 사용자의 파일 액세스 패턴을 분류하고 어떤 워크로드에서 대용량의 페이지 캐시가 효과적이고 그렇지 않은지를 분석하고 메모리 크기가 파일 입출력 성능에 미치는 영향을 분석하였다.

5. 실험 결과 및 성능 평가

5.1. 실험 환경

본 실험을 진행한 실험 환경은 다음과 같다. CPU 는 Intel(R) Xeon(R) CPU E5520 2.27GHz 이고 RAM은 DDR3 메모리로 2GB, 4GB, 8GB로 변경하여 실험하였다. 운영체제는 openSUSE 10.3으로 커널 버전은 2.6.22.5-31 이다. SSD는 Samsung 100GB SATA SLC 300Mbps MCCOE1HG5MXP-0VB 를 사용하였다.

페이지 캐시 관련 주요 속성은 다음과 같다.

표 1 실험 환경 페이지 캐시 관련 주요 속성

속성	설정 값
/proc/sys/vm/nr_pdflush_threads (시스템에서 pdflush 를 수행하는 스레드 수)	2
/proc/sys/vm/dirty_writeback_centisecs (pdflush 가 깨어나는 주기)	5 s
/proc/sys/vm/dirty_expire_centiseconds (데이터가 페이지 캐시에서 dirty 상태로 유지될 수 있는 시간)	30 s
/proc/sys/vm/dirty_background_ratio (pdflush 가 디스크로의 쓰기를 시작하는 dirty 페이지의 비율)	5 %
/proc/sys/vm/dirty_ratio (dirty 페이지가 전체 메모리에서 차지할 수 있는 최대 비율)	10 %
/proc/sys/vm/swappiness (프로세스 메모리가 페이지 캐시를 위해 스왑되는 정도를 설정)	60

본 논문을 위해 사용한 벤치마크는 Iozone[14]이다. Iozone 은 읽기, 쓰기, 랜덤 읽기, 랜덤 쓰기 등 각 파일 연산의 성능을 측정할 수 있는 파일시스템 벤치마크로 각 연산의 성능 측정에서 사용하는 파일 크기와 입출력 단위를 조정할 수 있어 다양한 워크로드에 대한 검증이 가능하다. 본 논문에서는 사용자의 파일 액세스 패턴을 과학 계산, 트랜잭션 처리, 사무용 처리, 멀티미디어 처리로 구분하여 파일 입출력 성능을 측정하였다.

5.2. 과학계산 프로그램

과학계산 프로그램은 대용량의 파일을 순차적으로 읽고 쓰는 액세스 패턴을 가진다[5]. 이를 실험하기 위해 Iozone을 이용하여 대용량 파일에 대한 순차 읽기, 순차 쓰기, 다시

읽기, 다시 쓰기 성능을 측정하였다. 파일의 크기를 점차로 키워서 파일 입출력 성능이 떨어지는 파일의 크기를 추출하였다. 다시 읽기, 다시 쓰기는 이미 읽거나 쓴 파일에 대해서 다시 순차적으로 쓰고 읽는 연산이다. 읽고 쓰는 단위는 4KB이다. 성능 평가의 단위는 단위 시간당 파일 입출력 처리량이다.

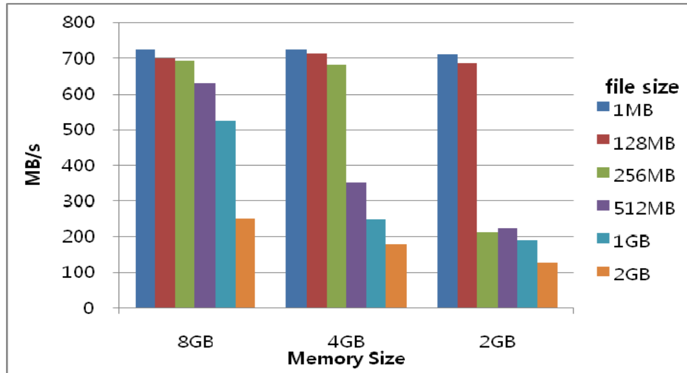

그림 1 Iozone 순차 쓰기(쓰기 단위 4KB)

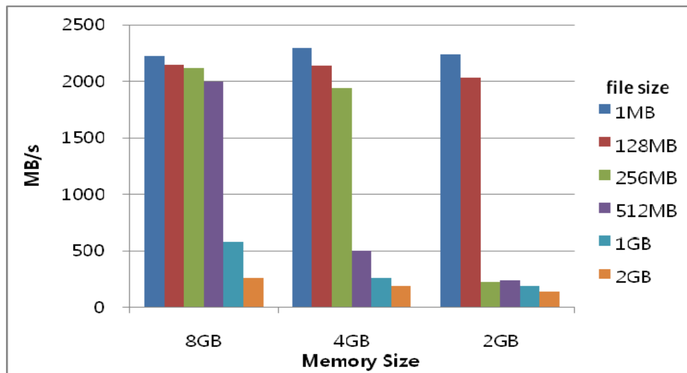

그림 2 Iozone 순차 다시 쓰기 (쓰기 단위 4KB)

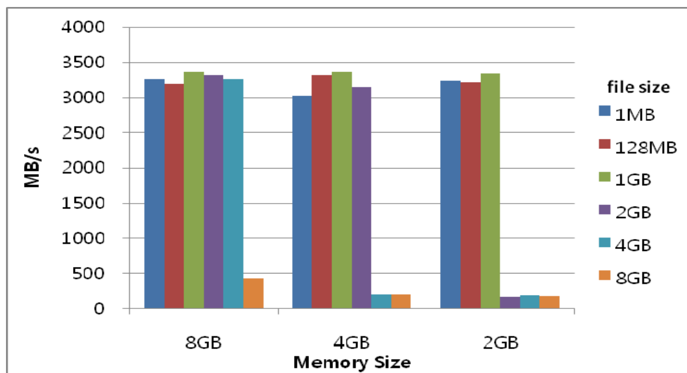

그림 3 Iozone 순차 읽기(읽기 단위 4KB)

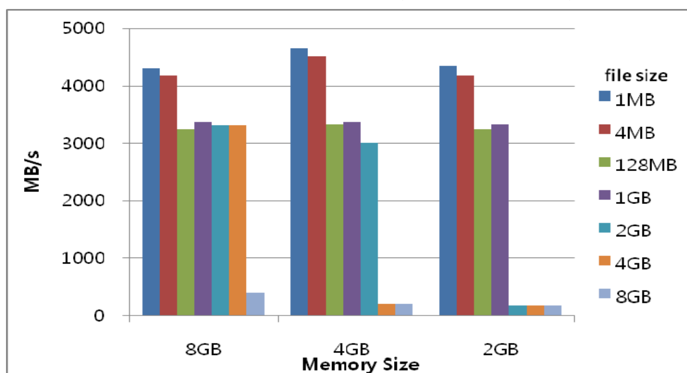

그림 4 Iozone 순차 다시 읽기 (읽기 단위 4KB)

먼저 쓰기와 다시 쓰기가 성능의 차이를 보여주는 것을 알 수 있다. 이것은 쓰기의 경우 파일 데이터뿐 아니라 파일의 메타 데이터를 쓰는 추가적인 오버헤드가 존재하기 때문이다.

실험 결과를 통해 파일 쓰기 성능이 파일 크기가 시스템 메모리의 10%보다 커지는 지점을 기점으로 급격하게 떨어지는 것을 확인할 수 있다. 이는 /proc/sys/vm/dirty_ratio가 10%로 설정되어 이를 넘는 크기의 파일에 대하여 프로세스가 페이지 캐시에 쓰는 것으로 쓰기 요청이 완료되지 못하고 SSD로의 쓰기가 완료되는 것을 기다려야 하기 때문이다. 파일 읽기 성능은 파일 크기가 시스템 메모리 크기와 같아지는 지점을 기점으로 떨어지는 것을 확인 할 수 있다. 이를 통해 리눅스가 동작중인 프로세스에 할당되지 않은 메모리를 페이지 캐시로 이용하는 것을 확인할 수 있다.

큰 크기의 파일에 대하여 읽기/쓰기 연산 모두 페이지 캐시의 크기가 성능에 큰 영향을 주는 것을 확인할 수 있다. 페이지 캐시의 크기가 클수록 더 큰 크기의 파일에 대하여 높은 입출력 성능을 보였다. 또한 작은 크기의 파일에 대해서는 페이지 캐시의 크기의 차이에 상관없이 비슷한 성능을 보이는 것을 확인할 수 있다. 이는 큰 파일에 대해서 연산을 할 때 페이지 캐시의 크기가 작다면 SSD에서의 액세스가 빈번히 발생하여 파일 입출력 성능이 떨어지기 때문이다. 페이지 캐시의 크기가 충분하다면 SSD로의 액세스가 줄어든다.

5.3. 트랜잭션 프로그램

트랜잭션 프로그램은 항공사 예약과 같은 시스템으로 웹 서버, 데이터베이스와 같은 프로그램이다. 이러한 시스템은 동시에 다수의 병렬 요청을 처리한다. 다수의 병렬 요청은 각기 다른 블록을 액세스하므로 파일 접근 패턴이 임의로 분포된다.

실험 1. 대용량 파일 랜덤 읽기/쓰기

랜덤 파일 액세스 패턴에 따른 성능을 실험하기 위해 Iozone 을 이용하여 대용량 파일의 랜덤 읽기/쓰기 성능을 측정하였다.

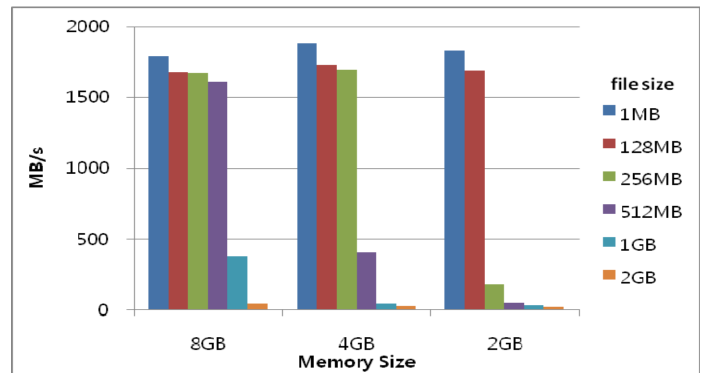

그림 5 Iozone 랜덤 쓰기 (쓰기 단위 4KB)

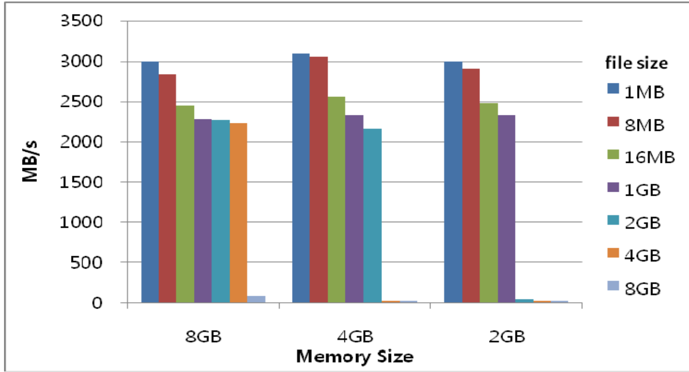

그림 6 Iozone 랜덤 읽기 (읽기 단위 4KB)

랜덤 읽기/쓰기 성능이 순차 다시 읽기 성능, 순차 다시 쓰기 성능과 비슷한 경향을 보이는 것을 확인할 수 있다. 랜덤 쓰기 성능이 순차 다시 쓰기 성능과 비슷한 경향을 보이는 것은 페이지 캐시가 SSD 로의 쓰기 요청에 대한 버퍼역할을 하였기 때문이다. 페이지 캐시가 버퍼 역할을 함으로서 프로세스는 실제 SSD 로의 쓰기가 완료되기를 기다리지 않고 계속 수행이 가능하다. 실제 SSD 로의 쓰기는 비동기적으로 발생한다. 따라서 쓰기의 재 순서화를 통해 가비지 컬렉션의 오버헤드가 감소하고 SSD 의 대역폭을 효율적으로 사용할 수 있다. 또한 가비지 컬렉션으로 인한 지연을 프로세스에게 숨길 수 있다. 하지만 시스템 메모리의 용량이 적은 경우 비동기적으로 쓰기를 하지 못하고 프로세스가 실제 SSD 에 쓰기가 완료되기를 기다리게 된다. 이 경우 프로세스의 성능은 가비지 컬렉션으로 인한 지연에 직접 영향을 받는다.

실험 2. 병렬 요청으로 인한 랜덤 파일 읽기/쓰기

트랜잭션 프로그램의 랜덤한 파일 액세스 패턴에 따른 성능을 좀 더 정확하게 검증하기 위해 Iozone을 여러 개의 스레드를 이용하여 실행하였다. 스레드의 개수는 16, 64, 256개로 변경하며 실험하였고 각 스레드는 랜덤한 위치에 파일 입출력을 수행한다. 그림에서 “File Size”는 각 스레드가 액세스하는 파일 크기이다. 따라서 동시에 접근하는 파일 크기는 (스레드의 수) X (각 스레드 별 액세스하는 파일크기) 이다. 워크로드에서 파일 입출력 단위는 16KB이다. 측정 결과는 각 스레드 별 처리량을 나타낸다.

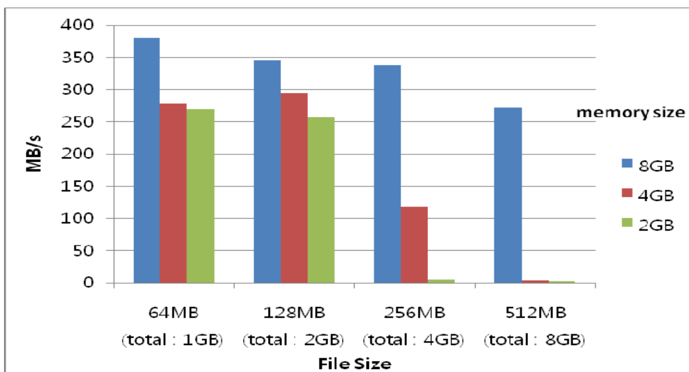

그림 7 Iozone 랜덤 읽기/쓰기 (스레드 16개, 입출력 단위 16KB)

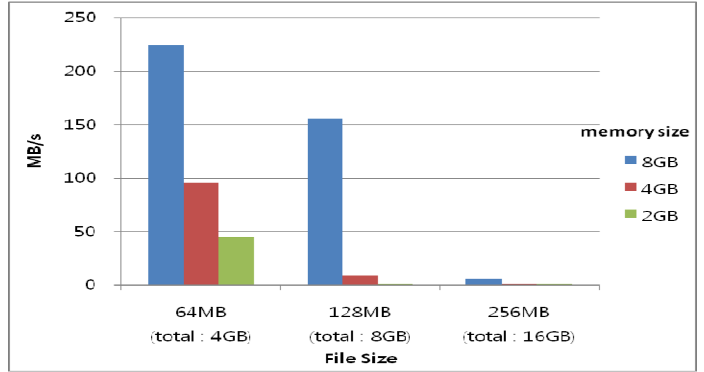

그림 8 Iozone 랜덤 읽기/쓰기 (스레드 64개, 입출력 단위 16KB)

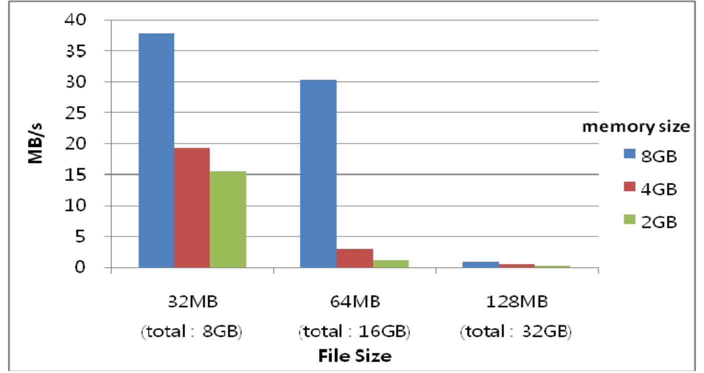

그림 9 Iozone 랜덤 읽기/쓰기 (스레드 256개, 입출력 단위 16KB)

페이지 캐시의 크기가 증가할수록 성능이 향상되는 것을 확인할 수 있다. 동시에 액세스하는 파일에 대해 페이지 캐시가 충분히 크지 못하여 SSD로의 요청이 빈번하게 발생할 때 성능이 떨어지는 것을 확인할 수 있다. 스레드의 수가 증가함에 따라 성능 감소가 점차로 작은 파일에서 나타났는데 이는 동시에 액세스하는 파일의 크기가 (스레드의 수) X (각 스레드가 액세스하는 파일 크기)이기 때문이다.

트랜잭션 시스템에서 동시에 여러 요청을 처리할 때 페이지 캐시의 크기가 충분하지 않을 경우 파일 데이터가 캐싱이 되지 못하여 파일 입출력 요청이 SSD에서 처리되는 것을 기다려야 한다. 이 때 CPU가 다음 서버 요청을 처리하기 위하여 현재 SSD의 큐에 있는 모든 입출력 요청이 처리 될 때까지 기다려야 함에 따라 서버의 성능은 현저하게 떨어질 것임을 알 수 있다.

5. 4. 사무용 프로그램

사무용 프로그램은 대부분 작은 크기의 파일을 액세스하고 파일 액세스 패턴이 시간적 지역성을 가진다[5]. 이와 같은 프로그램에서는 파일 입출력의 크기가 작고 동시에 트랜잭션 시스템과는 달리 동시에 여러 프로그램이 수행되지 않는다. 이를 검증하기 위해 16개의 스레드를 이용하여 랜덤 입출력 성능을 측정하였다. 측정 결과는 각 스레드 별 처리량을 나타낸다. 측정 결과 페이지 캐시의 크기가 성능에 크게 영향을 미치지 않은 것을 확인할 수 있다.

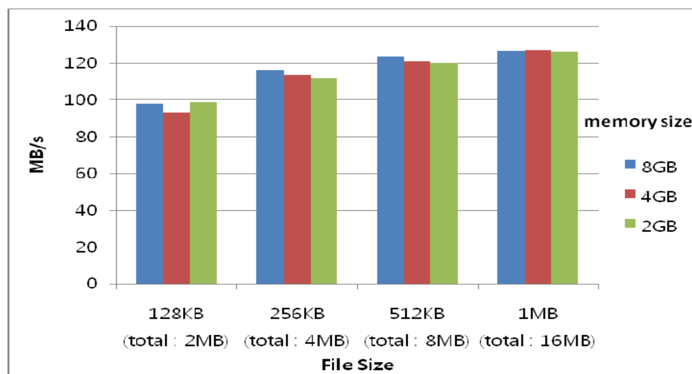

그림 10 Iozone 랜덤 읽기/쓰기
(스레드 16개, 입출력 단위 4KB)

5. 5. 멀티미디어(디지털 비디오/오디오)

멀티미디어 파일은 일반적으로 대용량으로 멀티미디어 이들에 대한 주된 파일 연산은 순차적이다[15]. 따라서 과학 계산 프로그램과 같이 큰 사이즈의 페이지 캐시가 효과적임을 알 수 있다.

6. 결론

본 논문에서는 리눅스 환경에서 페이지 캐시의 크기가 SSD 의 성능에 미치는 영향을 실험을 통해 분석하였다. 실험 결과, 대용량 파일을 액세스하는 경우, 또는 여러 요청을 병렬로 처리하는 트랜잭션 시스템과 같이 파일 입출력의 부하가 큰 환경에서 대용량의 페이지 캐시가 시스템 성능을 효과적으로 향상시킴을 확인하였다. 특히 SSD 는 랜덤 쓰기가 느린 특성을 가지는데, 이 단점을 효과적으로 보상할 수 있었다. 이는 페이지 캐시 계층을 통해 SSD 로의 비동기적 쓰기가 가능하여 프로세스가 SSD 로의 쓰기가 완료되기까지 대기하지 않고 진행이 가능하기 때문이다. 따라서 가비지 컬렉션과 같은 지연의 영향을 받지 않고 SSD 의 대역폭을 효율적으로 사용할 수 있다.

최근 메인 메모리로 사용되는 DRAM 의 집적도가 증가하여 저렴한 가격에 큰 용량의 메인 메모리를 구축하는 것이 가능하게 되었다. 그에 따라 기존에는 페이지 캐시의 크기를 일정 이상 할당할 수 없었지만 메모리의 가격이 내려감에 따라 큰 용량의 페이지 캐시가 가능해지게 되었다. 이러한 큰 용량의 페이지 캐시가 SSD 와 함께 파일 입출력이 집중적으로 이루어지는 시스템에서 시스템 성능을 크게 향상시킬 수 있을 것이라고 기대한다.

7. 참조논문

[1] Andrew Birrell, Michael Isard, Chuck Thacker, Ted Wobber, “A design for high-performance flash disks”, ACM SIGOPS Operating Systems Review, v.41 n.2, p.88-93, 2007

[2] Douglas Dumitru, “Understanding Flash SSD Performance”, http://managedflash.com/news/papers/easyc o-flashperformance-art.pdf, 2007

[3] Chung T. S., Park D. J., Park, S. W., Lee, D. H., Lee, S. W., Song, H. J, “System software for flash memory: a survey”, In Proceedings of the 2006 IFIP International Conference on Embedded And Ubiquitous Computing, 2006

[4] S.W. Lee, W. K. Choi, D. J. Park, “FAST: an efficient flash translation layer for flash memory”, The 1st International Workshop on Embedded Software Optimization, 2006

[5] John Ousterhout, Fred Douglis, “Beating the I/O bottleneck: a case for log-structured file systems”, ACM SIGOPS Operating Systems Review, v.23 n.1, p.11-28, 1989

[6] Daniel P.Bovet, Marco Cesati, “Understanding the LINUX KERNEL”, O'Reilly, 2002

[7] Megiddo, N., Modha, D., “Outperforming LRU with an adaptive replacement cache”, IEEE Computer, pp. 58–65, 2004

[8] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, C. S. Kim, “LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies”, IEEE Transactions on Computers, vol.50, issue 12, 2001

[9] S. Jiang, X. Zhang, “LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance”, Proceedings of the ACM SIGMETRICS conference, 2002

[10] S. Y. Park, D. Jung, J. U. Kang, J. S. Kim, J. Lee, "CFLRU: a replacement algorithm for flash memory" Proceedings of the international conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), pp.234-241, 2006

[11] John K. Ousterhout, H. Da Costa, David Harrison, John A. Kunze, Mike Kupfer, James G. Thompson, “A trace-driven analysis of the UNIX 4.2 BSD file system”, Proceedings of the tenth ACM symposium on Operating systems principles, p.15-24, 1985

[12] Chris Ruemmler, John Wilkes, “UNIX disk access patterns”, In Proceedings of the Winter 1993 USENIX Conference, pages 405-420, 1993

[13] Drew Roselli, Jacob R. Lorch, Thomas E. Anderson, “A comparison of file system workloads”, Proceedings of the annual conference on USENIX Annual Technical Conference, p.4-4, 2000

[14] D. Capps, W. D. Norcott, “Iozone filesystem benchmark”, http://www.iozone.org

[15] P. Venkat Rangan, Harrick M. Vin, “Designing file systems for digital video and audio”, Proceedings of the thirteenth ACM symposium on Operating systems principles, p.81-94, 1991

ACKNOWLEDGEMENT

본 연구는 교육과학기술부/한국과학재단창의적연구진흥사업(매니코어프로그래밍연구단, 0421-20090025)의 지원으로 수행되었습니다. 이 연구를 위해 연구장비를 지원하고 공간을 제공한 서울대학교 컴퓨터연구소에 감사 드립니다.