http://dx.doi.org/10.3745/KIPSTA.2012.19A.1.035

개인용 SSD를 위한 선반입 및 메모리 관리 정책

백 승 훈†

요 약

기존의 운영체제에서 하드디스크의 성능을 향상시키기 위해서 사용해왔던 기술들이 SSD(Solid State Drive)에는 부정적 효과를 나타내는 경우가 많다. HDD의 기계적인 요소 때문에 접근 시간과 블록 주소의 순서가 성능에 매우 중요한 요인으로 작용하였지만, SSD는 불록 주소의 순서에 영향을 받지 않는 우수한 랜덤 읽기 성능을 제공한다. 실제 개인용 PC에서 SSD를 사용할 때에 선반입을 끄도록 권고되고 있다. 하지만 이 논문은 SSD의 내부 구조와 낸드 플래시 메모리의 특징을 고려한 선반입 및 메모리관리 정책를 결합한 방법을 제시한다. SSD에는 다수개의 낸드 플래시 메모리로 구성되어 있어 칩을 동시에 구동시키는 것이 중요하며, 낸드 플래시 메모리의 기본 입출력 단위가 계속 증가하는 방향으로 발전하고 있어서 SSD 내부의 동작 단위가 운영체제의 블록 크기보다 훨씬 커지게 되었다. 이 논문은 이러한 SSD의 특징과 경향을 수용하여, 제안하는 선반입 기법은 SSD의 동작 단위로 수행되며, 제안하는 메모리 관리 기법은 그 선반입 기법의 단점을 보완하여, 캐시 히트율과 선반입 히트율의 합이 최대가 되도록, 선반입되었지만 사용되지 않는 데이터를 적응적으로 퇴출한다. 본 기술은 리눅스 커널 모듈로 개발하였으며 실제 SSD를 사용하여 성능 평가를 실시하였다. 주어진 실험에서 제안하는 선반입 기법이 약 26%까지 성능을 향상시켰다.

키워드 : 선반입, 캐시, SSD, 플래시

A Prefetching and Memory Management Policy for Personal Solid State Drives

Sung Hoon Baek†

ABSTRACT

Traditional technologies that are used to improve the performance of hard disk drives show many negative cases if they are applied to solid state drives (SSD). Access time and block sequence in hard disk drives that consist of mechanical components are very important performance factors. Meanwhile, SSD provides superior random read performance that is not affected by block address sequence due to the characteristics of flash memory. Practically, it is recommended to disable prefetching if a SSD is installed in a personal computer. However, this paper presents a combinational method of a prefetching scheme and a memory management that consider the internal structure of SSD and the characteristics of NAND flash memory. It is important that SSD must concurrently operate multiple flash memory chips. The I/O unit size of NAND flash memory tends to increase and it exceeded the block size of operating systems. Hence, the proposed prefetching scheme performs in an operating unit of SSD. To complement a weak point of the prefetching scheme, the proposed memory management scheme adaptively evicts uselessly prefetched data to maximize the sum of cache hit rate and prefetch hit rate. We implemented the proposed schemes as a Linux kernel module and evaluated them using a commercial SSD. The schemes improved the I/O performance up to 26% in a given experiment.

Keywords : Prefetching, Cache, Solid State Drive, Flash

1. 서 론

지금까지 기계적 부품을 갖는 컴퓨터 저장장치의 성능과 컴퓨터의 프로세서 성능의 차이는 점차 벌어져 갔다. 메인 메모리와 HDD(Hard Disk Drive) 사이에는 10^5배의 응답속도 차이가 있다[1]. 병렬 디스크들을 이용하는 레이드(RAID) 기술, 선반입(Prefetching), 캐시(Cache), 입출력 스케줄링 등, 다양한 기법으로 점차 벌어지는 CPU와 저장장치의 성능 차이를 좁히려는 시도가 계속 진행되고 있다. 하지만 HDD의 기계적 한계로 인하여 시간이 갈수록 성능 격차가 벌어져만 가고 있다. 그런데, SSD(Solid State Drive)는 메인 메모리와의 성능차이를 10^2 수준의 차이로 낮출 수 있다[1].

불과 수년 사이에 컴퓨터 저장장치로서 SSD가 HDD를 치환하면서 급성장세를 타고 있다. SSD는 소음과 진동이

† 정 회 원 : 중원대학교 IT공학부 조교수
논문접수 : 2011년 5월 11일
수 정 일 : 1차 2011년 7월 4일, 2차 2011년 7월 28일
심사완료 : 2011년 8월 2일

없으며, 충격에 매우 강하고, 높은 읽기 성능으로 인하여 랩톱 컴퓨터에서 큰 경쟁력을 가지고 있다. 하지만 초기 SSD는 블록 매핑 또는 하이브리드 매핑으로 인하여 HDD의 쓰기 성능이 오히려 SSD보다 좋았다. 하지만 최근 낸드 메모리칩의 발전과 페이지 단위 미세 매핑을 도입하여 SSD의 쓰기 성능이 HDD를 훨씬 능가하게 되었다. 그리하여 모바일 단발기에서 데스크톱 컴퓨터, 기업용 서버, 및 데이터베이스 시스템에 이르기까지 SSD가 적용되는 범위가 확대되어 가고 있다[2,3,4].

SSD가 컴퓨터에 널리 사용되기 시작한지 불과 몇 년에 지나지 않았기 때문에 현재의 운영체제는 HDD의 기계적인 움직임을 고려하여 입출력 스케줄링, 선반입, 캐시 정책들을 개발해오고 있었다. 하지만 페이지 주소 단위로 임의접근이 가능한 플래시 메모리로 구성된 SSD는 HDD와는 매우 다른 특성을 지니고 있어서 현재의 운영체제가 사용해오고 있는 기법들을 SSD에 적용하지 않아야 오히려 더 좋은 성능을 얻을 수 있다[5].

2. 선행 연구

2.1 SSD관련 연구

최근에는 플래시 메모리에 특화된 파일시스템 연구가 진행되고 있고[6], 디스테이지(destage), 입출력 스케줄링, 선반입 등의 분야에서 SSD에 맞는 새로운 입출력 처리 기술들이 연구되기 시작하였다.

디스테이지란 저장장치로 저장해야할 데이터를 메모리에 저장해두었다가 적당한 시간까지 지연하여 저장장치로 그 데이터를 저장하는 기법들이다. 쓰기 캐시를 다수개의 페이지로 구성된 낸드의 블록단위로 관리하고 가장 오랫동안 사용하지 않은 블록을 디스테이지하지만 저장하지 않아도 되는 페이지들까지도 추가하여 완전한 낸드 블록을 저장하는 BPLRU[7]와; 캐시 관리 기법인 CLOCK 알고리즘에서 관리 단위를 낸드 블록단위로 설정한 LB-CLOCK[8]이 있다. 낸드에서는 다수개의 페이지로 구성된 블록 단위로 지우기와 머지가 발생하기 때문에 이러한 방식은 머지의 오버헤드를 줄임으로써 성능의 이득을 얻는다.

리눅스에서 구현된 입출력 스케줄링에는 Noop, Deadline, Anticipatory, CFQ(Complete Fair Queueing)가 있다. Noop은 단지 인접한 요구들을 하나로 만드는 것만 한다. Deadline은 Noop의 기능과 더불어 기한을 초과하여 기다린 요구들을 즉각 서비스한다. Anticipatory는 Noop의 기능과 더불어 어떤 읽기 요구를 일정시간 동안 더 기다린 후 더 많은 읽기 요청들을 모아서 서비스를 해준다. 그럼으로써 더 많은 읽기 요구들에 대해 엘리베이터 알고리즘을 적용하여 성능을 향상시킨다[9]. SSD에 적용한 입출력 스케줄링 기법에는 IRBW-FIFO(Individual Read Bundled Write First In First Out)[10] 및 BP (block-preferential) scheduler[5]가 있다. IRBW-FIFO는 읽기 요구들을 쓰기와 별도로 작은 단위로 관리하고, 쓰기들을 FIFO방법으로 서비스하되 좀 더 큰 논리 블록 단위로 관리하는 방법이다. BP scheduler는 현재 요청과 같은 낸드 블록에 속한 것을 우선 서비스하고, 다음으로 계류 중인 요청들을 가장 많이 포함하고 있는 낸드 블록을 우선으로 서비스하는 방법이다.

2.2 선반입 관련 연구

선반입은 블록이 요구되기 전에 미리 요구될 블록을 저장장치에서 메모리로 읽어서 메인 메모리와 저장장치 사이의 지연 시간을 줄이고 전체적인 처리량을 향상시킨다. 자주 언급되는 선반입의 목표는 데이터를 사용하기 전에 캐시 메모리에 데이터를 사용가능하게 하는 것이다. 그래서 연산과 디스크 입출력이 서로 중첩되어 성능이 향상될 수 있다. 선반입의 다른 목표는 연속한 블록들을 하나의 요청으로 모아서 디스크의 성능을 향상시키는 것이다. 디스크는 물리적 접근 비용이 크므로 여러 개를 별개로 요청하는 것보다 하나의 연속한 큰 블록으로 한 번에 모아서 요청하는 것이 효과적이다. 선반입은 응용프로그램-힌트-기반 선반입, 히스토리(History) 기반 선반입 및 순차 선반입으로 구분할 수 있다.

단일 또는 소수개의 프로세스에서는 입출력의 동시성이 낮다. 다르게 말하면, 한 순간에 동시에 동작 중이거나 계류 중인 입출력의 개수가 적다. 그래서 응용프로그램-힌트-기반 선반입은 응용프로그램이 가까운 미래에 사용될 블록들의 위치에 대해서 미리 힌트를 주어서 동시 입출력 개수를 증가시켜 병렬성을 높여 주어 디스크 어레이에 이득을 준다[11]. 리눅스 2.6에서부터 제공되는 비동기 입출력 인터페이스를 이용하여 힌트를 제공할 수 있다[12]. 이 기법을 위해서는 프로그래머들의 코딩 스타일과 프로그램의 구조를 바꾸어야 한다.

히스토리 기반 선반입은 정적인 과거의 접근 기록을 학습하여 미래의 접근 위치를 예측하는 것이다. 좋은 예측 정확도를 확보하기 위해서 여러 가지 기술들이 제안되어 오고 있다. 각 접근 위치 사이의 천이 빈도를 마코프 체인에 적용하는 기법, 파일 수준의 접근 기록을 분석하여 예측하는 기법, 과거에 접근한 블록들 사이의 확률을 분석하는 기법들이 있다[13,14]. Microsoft Windows의 Superfetch는 대표적인 히스토리 기반 선반입이다[15]. 다른 연구로서 응용프로그램들을 빠르게 실행하기 위해서 History정보를 사용한다[16]. 하지만 이런 기법들은 과거의 많은 접근 기록을 학습하기 위해서 많은 메모리 및 큰 연산 부하를 필요로 하고, 정적으로 동일한 작업이 반복되는 상황에서만 효과적이므로 특수한 경우에만 사용된다.

실제 시스템은 부하가 매우 적고, 많은 경우에 효과적인 순차 선반입이 사용된다. HDD의 기계적인 접근 비용이 크기 때문에 한번 접근할 때에 한꺼번에 많이 읽는 것이 유리하다. 그래서 순차 선반입은 큰 파일을 읽을 때에, 또는 순차적 접근 패턴이 나타날 때에, 한 번에 크게 미리 읽어 둔다. 더 자세히 설명하면; 순차적이지 않은 블록이 요청되면 선반입을 하지 않고, 바로 전의 접근 위치와 현재의 접근

위치가 연속적이면 현재 선반입의 크기를 과거의 선반입 크기보다 두 배 증가시킨다. 하지만 선반입 크기가 기 설정된 크기를 초과하지 않는다[17].

실제적으로 널리 사용되는 순차 선반입은 HDD의 특징에 적합하게 설계되어, SSD에는 오히려 부정적 효과를 발생시킬 수 있다. 이 논문은, 기존 기술에는 없는, SSD에 효과적이고, 부하가 적게 들고 일반적인 다양한 응용에 사용할 수 있는 선반입 기법을 제안한다.

이 논문에서 제안하는 선반입 기법은 SSD 구조, 즉 낸드 플래시 메모리 칩들의 병렬성 및 낸드 플래시 메모리 내부의 특징과 개인용 컴퓨터의 입출력 특성을 고려한다. 뿐만 아니라 그 제안하는 선반입 기법의 장점을 유지하되 단점을 낮추어 주는 메모리 관리 기법을 제시한다.

3. 동기 - SSD의 특징

이 논문에서 제시하는 선반입의 효과를 쉽게 이해하기 위해서 먼저 SSD 및 낸드 플래시 메모리의 내부 구조를 이해해야 한다. 이 절에서는 SSD의 내부 구조 및 낸드 플래시 메모리의 동작 단위와 다중-플레인 모드(multi-plane mode)에 대해서 설명한다.

3.1 SSD의 병렬성

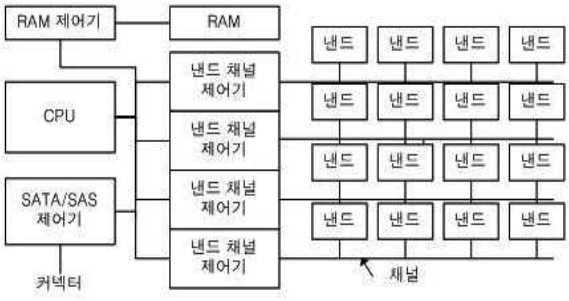
SSD는 16개에서 64개가량의 낸드플래시 메모리를 사용한다. 보통 4개의 칩이 하나의 패키지에 들어가는 Quad Flat Package(QFP)가 사용되고, 두 개의 패키지를 수직으로 적층하기도 하기 때문에 실제로 SSD에서 볼 수 있는 패키지의 개수는 4개에서 16개 정도이다.

SSD에 포함된 낸드 플래시 메모리는 USB 인터페이스의 메모리 스틱과 다르지 않다. 그래서 높은 성능을 얻기 위해 SSD는 메모리 스틱보다 더 많은 낸드 플래시 메모리를 통한 병렬화를 이용하고 있으며, 메모리 스틱보다 매우 높은 수백 MB/s의 대역폭을 제공한다.

(그림 1)은 SSD의 내부구조를 보여준다. 높은 대역폭을 얻기 위해서, 데이터는 RAID-0처럼 다수개의 낸드 플래시 메모리에 나뉘어 흩어져 배치되어 있으며, 낸드가 제어기와 데이터를 주고받기 위해 채널이라 불리는 버스가 존재하고, 성능 증대를 위해서 다수개의 채널이 존재한다. 하나의 채널에는 다수개의 낸드가 존재한다. 일반적으로 SSD에는 열 개 내외 채널이 존재한다.

낸드 플래시 메모리의 병렬화를 이용한 SSD의 최대 대역폭은 높지만, I/O의 동시성이 낮은 개인용 컴퓨터에서 작은 크기의 읽기 요구는 한 개 또는 소수의 플래시 메모리 칩만 동작시킨다. 따라서 충분히 SSD의 최대 대역폭을 활용하지 못하게 된다.

예를 들어 60μs의 페이지 읽기 시간과 4KiB의 페이지를 갖는 낸드 메모리로 600MB/s의 SATAII 대역폭을 최대한 활용하기 위해서는 약 10개의 낸드 메모리칩이 동시에 동작해야 한다. 만약 호스트가 SSD에게 4KiB(kibibyte) 이하의



(그림 1) SSD의 내부구조

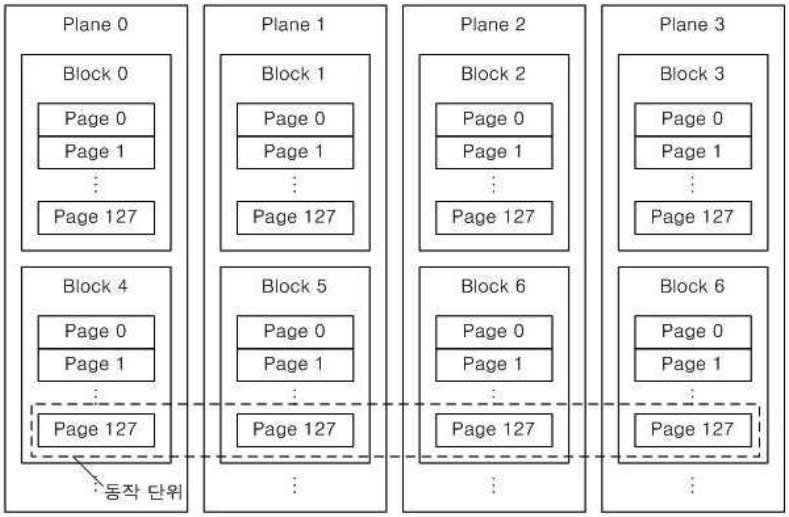
읽기를 동기식으로 요구하면 하나의 낸드 메모리칩만 동작하여 대역폭과 SSD의 최대 성능이 낭비된다.

3.2 동작 단위의 증가

낸드 플래시 메모리의 공정이 더욱 줄어들수록 셀의 특성이 나빠진다. 이것을 보완하기 위하여, 단계적으로 셀을 프로그램하는 방식인 ISPP(Incremental Step Pulse Program)의 셀 프로그램 시간 간격을 미세하게 한다. 그러면 셀의 문지방 전압 수준을 보다 정밀 설정할 수 있어서 데이트의 신뢰도를 향상시킨다[18]. ISPP란 아주 짧은 시간동안 셀을 프로그램한 뒤에 그 셀이 필요한 문지방 값에 도달하였는지 검사한다. 필요한 문지방 값까지 도달할 때까지 짧은 셀 프로그램을 반복하는 방식이다. 그런데 ISPP 시간 가격을 미세하게 할수록 일회의 프로그램에 필요한 ISPP 단계가 많아지고, 각 ISPP 단계에 필요한 오버헤드의 합이 증가하여 최종적으로 프로그램 시간이 증가하게 된다.

그래서 낸드 플래시 제조사들은 느린 단위소자의 성능을 보상하기 위해서 페이지 크기를 늘리고, 다중 플레인(multi-plane) 기능을 제공하고, 플레인의 개수를 늘려가고 있다. 플레인의 개수를 증가시키고, 페이지 크기를 증가시키면, 읽기/프로그램/지우기의 수행 단위가 커져서 대역폭이 증가한다[19]. 예를 들어서 8KiB의 페이지에 4 플레인을 사용하면 32KiB 단위로 입출력을 하게 된다. 한 개의 페이지(4KiB)에 대한 입출력 시간과, 4 플레인 모드로 4개의 페이지(즉 16KiB)를 동시에 입출력하는 시간은 거의 동일하다[20,21]. 이러한 성능 향상 효과 때문에 각 플레인의 페이지들을 합친 가상의 큰 페이지를 사용한다. 즉 페이지 크기가 커지는 효과가 발생한다.

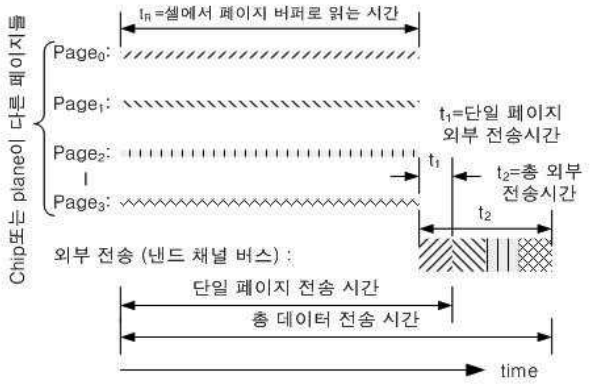
(그림 2)는 낸드 플래시 메모리 내부의 페이지, 블록, 플레인을 보여주고 있다. 이 그림에서는 낸드 플래시 메모리가 네 개의 플레인으로 구성되어 있다. 각 플레인은 동일한 개수의 블록으로 구성되어 있다. 각 플레인의 페이지들이 합쳐져 하나의 동작 단위가 되어 동시 읽기 또는 프로그램을 수행할 수 있다. (그림 2)에 동작단위가 표시되어 있다. 단, 플레인은 독립된 칩과 다르게 다음과 같은 제한사항을 갖는다. 플레인 모드 읽기/프로그램을 할 때에 각 플레인마다 선택된 페이지는 블록 내에서의 페이지 오프셋이 같아야 한다[20].



(그림 2) 낸드 플래시 내부의 페이지와 블록과 플레인(plane)의 구성을 보여준다. 각 플레인의 페이지들이 동시 읽기 또는 프로그램을 수행할 수 있다.

다중-플레인 모드로 인하여 읽기/프로그램의 동작 단위가 커지고 있다. 또한 페이지 크기도 계속 증가하는 경향을 보이고 있어서 동작 단위가 더욱 더 커지고 있다. 1세대 낸드 플래시 메모리의 페이지 크기가 512B였지만 (ECC 및 meta를 위한 여분의 공간은 제외), 1024B, 2048B, 4096B[20]를 거쳐서 8192B[21]까지 증가한 낸드 제품이 나타났다. 또한 과거의 작은 페이지의 낸드는 더 이상 생산되지 않는다.

(그림 3)은 한 페이지를 읽는 시간과 동시에 네 개의 페이지를 읽는 시간을 비교하고 있다. 즉, 네 개의 칩 또는 플레인에서 동시에 페이지를 읽을 때의 내부 전송시간 (t_R)과 외부전송시간(t_1)을 비교하였다. t_R은 셀에서 낸드 내부의 페이지 버퍼로 읽는 시간인데, 이것은 플레인 또는 칩 별로 동시에 독립적으로 진행된다. 그리고 t_R이 끝나면 외부 버스(낸드 채널)로 각 페이지 데이터가 순차적으로 전송된다. 일반적으로 내부 전송시간(t_R)은 외부전송시간(t_1)에 비하여 상당 느리다. 그렇기 때문에 (그림 3)과 같이 단일 페이지 전송 시간과 총 데이터 전송 시간은 크게 차이가 나지 않는다.



(그림 3) 네 개의 칩 또는 플레인에서 동시에 페이지를 읽을 때의 내부 전송시간 (t_R)과 외부전송시간(t_2)의 비교. 단일 페이지만 읽은 시간과 네 페이지를 동시에 읽은 시간은 크게 차이나지 않는다.

페이지 크기를 증가시키더라고 내부 전송시간(t_R)은 거의 변화가 없고, 동시 읽기를 수행하는 플레인 또는 칩의 수가 증가하더라도 내부 전송시간(t_R)은 변하지 않고, 데이터 전송을 위한 채널의 대역폭은 상대적으로 크므로, 동작 단위 크기(페이지 크기와 플레인 수의 곱) 증가는 결과적으로 최대 데이터 전송 대역폭을 향상시킨다.

다른 면으로 분석하면 동작 단위보다 작은 읽기를 하는 시간은 동작 단위 읽기 시간과의 차이가 거의 없기 때문에, 공간 지역성을 고려한다면 하나의 작은 읽기가 요청되었을 때에 SSD의 내부동작에 대응되는 (플레인과 칩의 병렬성을 활용할 수 있는) 데이터를 미리 선반입하는 것이 성능 향상에 도움이 될 수 있다.

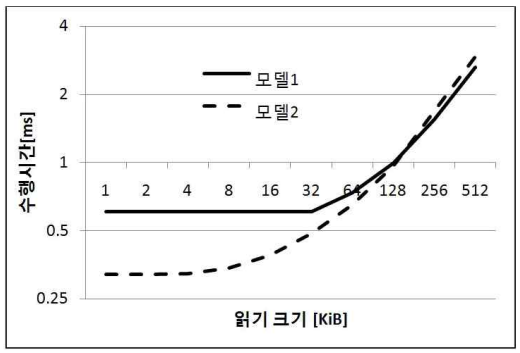
3.3 운영체제의 입출력 단위와 낸드의 동작 단위

SSD의 호스트가 되는 컴퓨터의 운영체제는 512B 또는 4096B를 입출력 단위로 사용한다. 기존의 저장장치의 섹터 크기는 512B이었지만, 저장장치 성능의 증가를 위하여 최근에는 4096B의 큰 섹터크기를 사용한다[22]. 큰 섹터는 큰 파일과 큰 용량의 저장장치에 효과적이다.

SSD의 성능 향상을 위하여 다중-플레인 모드와 큰 페이지를 사용한다. 다중-플레인 모드와 큰 페이지는 큰 파일에 효과적이지만 작은 파일이나 작은 메타 데이터에는 효과적이지 않다. 예를 들어 4 플레인과 8KiB의 낸드 페이지를 사용하면 32KiB가 기본 동작 단위가 된다. 그러면 4 플레인에 정렬된 32KiB를 읽는 시간과 섹터 크기인 4096B를 읽는 시간은 비슷하다[20]. 데이터 전송 인터페이스가 포화가 되지 않을 때까지는 32KiB 읽기는 4096B 읽기 시간과 비슷하다.

3.4 읽기 크기와 수행시간의 정량적 평가

(그림 4)는 두 가지 SSD 모델을 이용하여 읽기 크기의 변화에 따른 수행시간을 측정한 결과를 보여주고 있다. 읽기가 SSD의 내부동작단위에 정렬되도록 읽기의 오프셋을 읽기크기에 정렬하였다. 약 16KiB 또는 32KiB이하에서는 수행시간의 변화가 거의 없음을 알 수 있다. 또한 모델1의 경우에 읽기 크기가 64KiB에서 256KiB로 4배 증가하면, 수행시간은 단 2배만 증가하였다. 이 실험은 칩의 병렬성 및 페이지 크기에 관한 결과를 보여준다.



(그림 4) 두 가지 SSD 모델을 이용하여 읽기 크기의 변화에 따른 수행시간을 측정하였다. 읽기의 오프셋은 읽기 크기에 정렬하였다. 즉, 읽기가 SSD의 내부동작단위에 정렬되도록 하였다. 약 16KiB 또는 32KiB이하에서는 수행시간의 변화가 거의 없다.

이 실험은 리눅스 커널 2.6.35를 사용하였으며, 실험 전에 SSD에는 전 영역에 랜덤 데이터로 채워 넣었다. 어떤 SSD는 중복제거(deduplication)[23]나 압축 기능을[24] 가지고 있다. 만약 동일한 블록 데이터를 반복적으로 저장하면 중복제거 기능이 작동하여 SSD는 동일한 데이터를 플래시에 저장하지 않는다, 단순한 패턴이 반복되는 데이터를 사용하면 압축의 효과가 나타나서 읽기 실험에 예기치 않은 영향이 나타날 수 있다. 마지막으로 SSD 내부의 선반입을 막기 위해서 1MiB 간격으로 징검 읽기를 수행하였다.

이 실험의 결과로부터 다음의 사실을 얻을 수 있다. 공간지역성을 고려한다면 운영체제의 입출력 단위의, SSD 동작단위 보다 작은, 읽기가 요청되었을 때에 SSD의 동작 단위에 정렬된, 동작 단위 크기의, 데이터를 미리 선반입하는 것이 성능 향상에 도움이 될 수 있다.

4. 제안 기술 - 선반입 및 메모리 관리 기법

이 장은 제안하는 선반입 기법과 메모리 관리 기법을 소개한다. 선반입 기법은 SSD 구조, 즉 낸드 플래시 메모리 칩들의 병렬성 및 낸드 플래시 메모리 내부의 특징과 개인용 컴퓨터의 입출력 특성을 고려하였다. 뿐만 아니라 그 제안하는 선반입 기법의 장점을 유지하되 그것의 메모리 사용률 단점을 보완 하여 주는 메모리 관리 기법을 제시한다.

4.1 SSD를 위한 선반입 - 세그먼트 선반입

HDD기반의 선반입에는 순차 선반입 기법이 널리 이용되지만 이 기법은 HDD에 최적화되어 있으며 SSD의 동작 원리를 고려하지 않고 있으며, 순차적이거나 큰 파일을 읽을 때에만 효과적이다. 실제적으로, SSD가 설치된 개인용 컴퓨터에서는 성능향상을 위해서 선반입 기능을 중지시키는 것을 권고한다[25]. 하지만, SSD를 고려하고, 순차적인 읽기뿐만 아니라 비순차적인 읽기에 이득이 되며, 구현이 복잡하지 않아 실질적으로 사용될 수 있는, 선반입의 연구가 필요하다.

SSD에서 어느 정도의 크기 이내에서는 섹터나 운영체제의 입출력 단위보다 크게 읽어도 SSD 수행시간에는 큰 변화가 없다는 것을 2장에서 정성적 및 정량적으로 분석하였다. 그래서 이 논문에서 제안하는 기술에서는 연속적인 섹터들을 하나의 선반입 단위로 설정한다. 이 선반입 단위를 세그먼트라고 하자. 그리고 어떤 섹터를 읽으면, 그 섹터가 속해있는 세그먼트의 나머지 섹터들도 포함하여 읽는다.

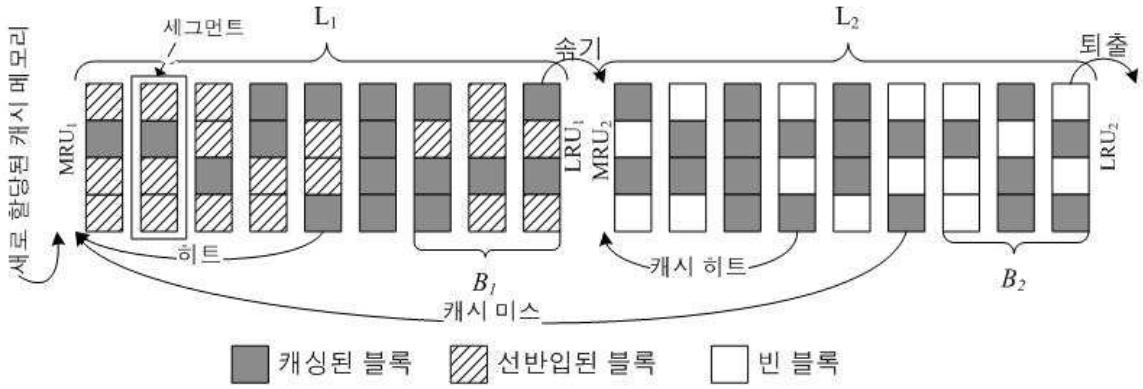
이 기법은 미래에 참조될 것 같은 데이터를 묵시적으로 선반입함으로써 공간 지역성의 원리를 이용한다. 즉 세그먼트 중의 나머지 섹트들이 미래에 요청될 가능성이 높다는 특성을 활용한다[26]. 우리는 하나의 섹터를 읽는 시간은 다수개의 섹터로 구성된 세그먼트를 읽는 시간과 비슷하다는 것을 알고 있다.

세그먼트 선반입은 순차 선반입이나 p-블록 선반입과는 구분된다. p-블록 선반입은 블록 b에 요청이 있을 때에 블록 b에서 블록 b+p까지 선반입한다. 순차 선반입은 어떤 파일을 순차적으로 읽을수록 p의 크기를 기설정된 최대값까지 배가시킨다. 세그먼트 선반입은 요청된 블록이 속한 SSD동작 단위 즉 세그먼트를 선반입하므로 요청된 블록의 앞과 뒤 블록들이 선반입된다. 그러므로 순차 선반입은 SSD동작 단위에 정렬되지 않는 읽기를 발생시키고 순차 읽기에서만 이득이 있지만, 세그먼트 선반입은 SSD의 동작단위에 정렬되어 있고 순차 읽기뿐만 아니라 공간지역성이 높은 읽기에 이득을 제공한다.

이것은 CPU의 라인 선반입과 유사하다. 예를 들어 CPU 캐시는 한 바이트를 읽어도 512B의 캐시라인 단위로 읽기를 수행하는 것과 유사하다. 캐시라인은 본 기법의 세그먼트에 대응된다. 하지만 본 기법과 라인 선반입의 차이가 있다. 본 기법은 잘못된 선반입으로 말미암아 캐시 메모리에 들어와서 캐시 메모리 공간을 낭비하는 사용되지 않는 블록들을 효율적으로 제거하는 적응적 캐시 솎기 방법을 포함하고 있다.

4.2 메모리 관리 - 적응적 캐시 솎기

세그먼트 선반입은 사용되지 않는 데이터를 선반입함으로써 메모리 효율성이 낮아져 캐시 히트율이 감소한다. 만약 선반입의 이득보다 캐시 히트율 감소로 인한 손실이 크다면 전체적인 성능이 낮아진다. 그래서 세그먼트 선반입의 단점을 보완하는 새로운 메모리 관리 정책인 적응적 캐시 솎기



(그림 5) 적응적 메모리 솎기 기법을 활용한 캐싱된 블록 데이터와 선반입된 블록 데이터의 예

를 제안한다. 적응적 캐시 솎기 정책은 세그먼트 선반입과 함께 사용된다. 이 정책은 적당한 시간에 선반입 되었지만 사용되지 않는 데이터를 솎아서 메모리에서 퇴출시킨다. 더구나, 워크로드가 변하더라도 캐시 히트율과 선반입 히트율의 합이 항상 최대가 되도록 캐시구조를 변경하는 적응적 기법을 포함한다.

캐싱된 블록에 대한 요청을 캐시 히트라고 한다. 캐싱된 블록이란 요청에 의해서 읽혀지거나 메모리 히트가 발생한 블록이다. 선반입된 블록은 요청되지 않고 선반입에 의해서 읽혀진 블록들이다. 선반입된 블록에 요청이 발생하는 사건을 선반입 히트라고 한다. 선반입 히트가 발생하면 선반입된 블록은 캐싱된 블록으로 속성이 바뀐다.

(그림 5)는 이 적응적 캐시 솎기 기법의 동작 예를 보여주고 있다. 이 그림에서는 네 개의 블록이 모여 하나의 세그먼트(선반입 단위)를 형성하였다. 즉 항상 네 개의 블록에 대한 읽기 동작을 한다.

메모리는 블록 별로 할당되지만, 선반입 및 리스트 관리는 세그먼트 단위로 이루어진다. 세그먼트는 선반입된 블록과 캐싱된 블록을 포함한다. 세그먼트의 빈 블록은 메모리가 할당되지 않은 영역이다. 메모리 할당은 세그먼트가 아닌 블록 단위로 이루어지기 때문에, 선반입된 블록과 캐싱된 블록의 합은 일정하지만, 빈 블록의 수에 따라서 세그먼트의 개수가 변한다. 즉, 세그먼트들의 빈 블록들이 많을수록 더 많은 선반입된 블록들이 퇴출되어 선반입 히트율은 감소하고 캐싱된 블록의 개수가 증가하여 캐시 히트율이 증가한다.

세그먼트와 블록은 다음과 같이 관리된다. L_1과 L_2로 구분되는 두 개의 리스트가 있고, 각각 리스트는 LRU와 비슷하게 동작한다. 각 리스트의 관리단위는 세그먼트이다. L_1에는 캐싱된 블록과 선반입된 블록이 존재할 수 있으나, L_2에는 캐싱된 블록만 존재할 수 있다.

저장장치로부터 최근에 읽은 세그먼트는 L_1의 앞(MRU_1)에 삽입된다. L_1에서 캐시 히트 또는 선반입 히트가 발생하면 그 히트된 세그먼트는 L_1의 MRU1로 이동한다. L_1의 크기는 제한을 갖기 때문에 새로 읽힌 새 세그먼트가 삽입되어 L_1의 크기가 기설정된 크기를 초과하면, 그 크기를 유지하기 위해 L_1의 끝(LRU_1)에 있는 세그먼트를 L_2의 앞(MRU_2)으로 이동시킨다.

L_2에는 선반입된 블록이 존재 하지 않기 때문에, L_1에서 L_2로 이동하는 세그먼트에서 선반입된 블록들만을 메모리에서 퇴출된다. 이 과정을 솎기라고 한다. L_2에서 캐시 히트된 세그먼트는 L_2의 앞으로(MRU_2) 이동한다. 필요한 메모리가 부족할 때에는 L_2의 끝에(LRU_2) 있는 세그먼트를 퇴출시킨다.

L_1의 끝에(LRU_1)있는 세그먼트는 L_1 중에서 참조 된지 가장 오래된 것이므로, 그 세그먼트의 선반입된 블록들이 미래에 참조될 가능성이 매우 낮다고 예측할 수 있다. 그래서 L_1의 끝에(LRU_1)있는 세그먼트를 L_2로 이동시키면서 그 세그먼트의 선반입된 블록들을 퇴출한다.

L_1에서 L_2로 이동하는 세그먼트에서 퇴출된 (솎아지는) 블록들이 사용한 메모리는 메모리 풀에 삽입되어 다른 블록에게 할당된다. 솎아지는 블록이 많을수록 L_2의 캐싱된 블록을 위해 많은 메모리를 할당할 수 있어서 캐시 히트율이 상승한다. 반대로, 메모리가 부족할 때에 L_2의 끝에(LRU_2) 있는 블록들이 퇴출된다.

L_1의 크기가 크면 선반입된 블록의 개수가 증가하여 선반입 히트율은 증가하지만, 선반입되었지만 사용되지 않은 블록이 늦게 퇴출되어 메모리 사용률이 감소하고 캐시 히트율이 감소한다. 반대로 L_1의 크기가 작으면 선반입된 블록의 개수가 감소하여 선반입 히트율은 감소하고, 선반입 블록의 개수가 감소한 만큼 캐싱된 블록의 개수가 증가하여 L_2가 커지고 캐시 히트율은 증가한다. 그러므로 L_1의 크기를 정하는 것은 매우 중요한 부분이다.

본 논문에서는 최적의 L_1 크기를 결정할 수 있는 적응적 기법을 제안한다. 핵심 원리는 다음과 같다. L_1을 키워서 얻을 수 있는 선반입 히트율의 증가분과 L_2를 키워서 얻을 수 있는 캐시 히트율의 증가분이 동일할 때에 선반입 히트율과 캐시 히트율의 합이 최대가 된다. 히트율 증가분을 여분의

사용률(Marginal Utility)이라고도 하며, 적응적 치환 캐시에 사용된 순차 선반입(SARC)[17]에서 여분의 사용률 개념을 차용하였다.

L_1의 크기를 적응적으로 최적의 값으로 설정하는 방법은 다음과 같다. (그림 5)에서 B_1은 L_1의 LRU위치에 있는 영역이며, 기설정된 개수의 세그먼트를 가지고 있으며, 그 크기는 전체 메모리의 약 5% 정도로 설정한다. B_2는 L_2의 LRU 위치에 있는 영역이며, B_1과 동일한 개수의 세그먼트를 가진다. L_1을 증가시켜서 얻는 선반입 히트율의 여분의 사용률은 단위 시간에 B_1에서 발생한 선반입 히트율이고, 이것을 ΔP이라고 하자. L_2가 커져서 얻는 여분의 이득은 단위 시간에 B_2에서 발생한 캐시 히트율이다. 이것을 ΔC라고 하자. L_1의 세그먼트 개수를 1만큼 줄이면, L_2의 세그먼트 개수는 α만큼 커진다. 즉 L_1을 증가시켜서 얻는 이득은 ΔP이고, L_1을 감소시켜서 얻는 이득은 $\alpha\Delta C$이다. 그러므로 다음 수식과 같이 단위시간마다 L_1의 세그먼트 개수 N_1을 ΔP에 비례하여 증가시키고 $\alpha\Delta C$에 비례하여 감소시키면, 선반입 히트율과 캐시 히트율이 최대가 되도록, 워크로드가 변함에 따라 N_1이 적응적으로 변하게 된다.

$$N_1 \leftarrow N_1 + S(\Delta P - \alpha\Delta C)$$

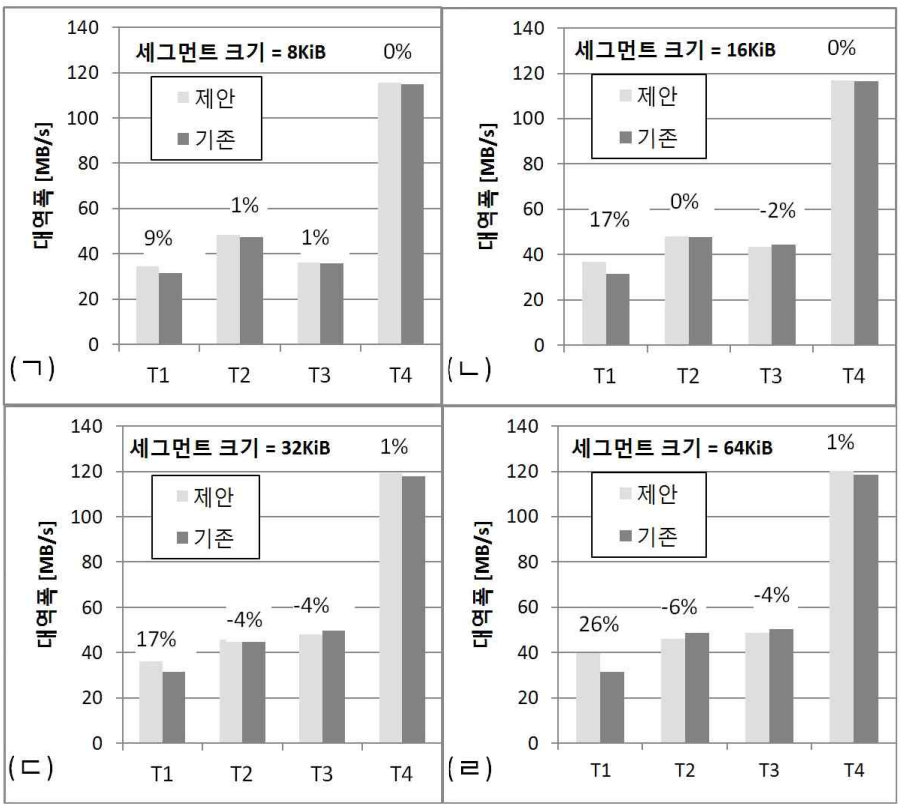
단, S는 되먹임의 속도를 조절하는 상수이고, α는 "B_1에 있는 모든 블록의 수"를 "B_2의 캐싱된 블록의 수"로 나눈 값이다. 단위시간은 일정한 시간으로 설정할 수 있고, 일정한 읽기요청개수로 선택할 수도 있다. 실험에서는 구현의 간략화를 위하여 B_1에 세 개의 캐시 히트 또는 선반입 히트가 발생할 때마다 위의 되먹임 식을 수행하였다.

5. 성능 평가

64-비트 리눅스 커널 2.6.35에서 구현된 SW 레이드 드라이버[27]를 수정하여, 제안된 그룹 선반입과 적응적 캐시 솎기를 구현하였다. 본 기법을 탑재한 레이드 드라이버의 캐시 메모리는 512MiB로 설정하였다. 레이드 드라이버에 한 개의 SSD (MMCRE28G5MXP)를 탑재하여 실험하였다. 호스트 컴퓨터는 인텔 i5 CPU 3.2GHz와 2GB의 메인 메모리를 탑재하였다.

리눅스 커널 및 기능을 변경하지 않고 제안한 기법을 탑재한 RAID 드라이버를 블록 장치 드라이버 형태로 추가하였다. (그림 6)에 나타난 실험에서, '제안'은 리눅스 커널을 거쳐 제안한 기법을 가동하는 레이드 드라이버를 통해 SSD를 가동한 것을 의미한다. '기존'은 주어진 레이드 드라이버에서 제안한 기법을 제거한 것 외에는 '제안' 방식과 동일하게 함으로써 공정한 성능 평가가 되도록 하였다.

개인 컴퓨터의 I/O 트레이스를 위해 PCMark®05 벤치마크에 있는 트레이스를 사용하였다. PCMark는 일반적인 응



(그림 6) PCMark 트레이스에서 세그먼트 크기의 변화에 따른 대역폭

용프로그램들이 사용되는 동안 디스크활동의 트레이스를 녹화해서 재생하는 벤치마크이다.

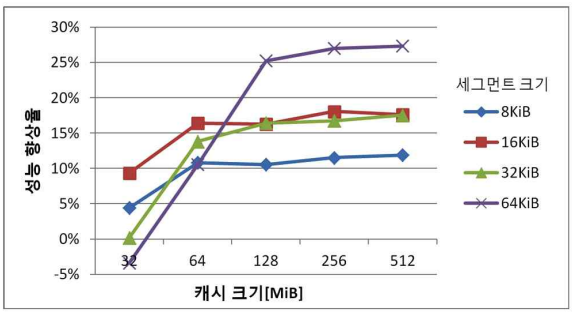
(그림 6)는 PCMark 트레이스에서 세그먼트 크기의 변화에 따른 성능을 평가한 결과를 보여준다. 이 그림에서 T1, T2, T3, T4는 각각 '일반 응용프로그램 사용', '프로그램 로딩', 'XP 시작', '바이러스 스캔'에 해당하는 트레이스이다. 운영체제의 입출력 단위(블록)는 4KiB로 설정하였으며, 세그먼트 크기를 각각 8KiB, 16KiB, 32KiB, 64KiB를 사용하였다. 세그먼트의 크기가 8KiB일 때는 (그림 6(ㄱ))에서 볼 수 있듯이 모든 트레이스에 대해서 제안하는 방식이 기존 방식(리눅스 기본 설정)에 비해서 성능이 우세함을 알 수 있다.

세그먼트의 크기가 커질수록 T1 트레이스에서는 성능이 26%까지 두드러진 성능을 나타내지만, T2, T3에서는 6%까지 성능 저하가 나타남을 볼 수 있다. 하지만 8KiB의 세그먼트에는 모든 트레이스에 대하여 성능저하가 없었다. 프로그램 로딩(T2) 및 XP 시작(T3)에 있어서; 이것들은 대부분 순차 읽기로 구성되어서, 세그먼트가 커질수록 불필요하게 선반입된 블록(요청된 순차읽기의 시작 블록의 앞부분)의 개수가 증가하여 다소 성능 저하를 일으킨다.

모든 워크로드에 대해서 성능 저하를 방지하려면; (1) (그림 6(ㄱ))처럼 작은 크기의 세그먼트를 사용하여 어떤 워크로드에서도 성능저하가 없도록 한거나, (2) 공간지역성이 낮은 프로그램 로딩(T2) 및 XP 시작(T3)과 같은 워크로드의 경우 선반입 히트율이 매우 낮다. 그러므로 선반입 히트율이 어느 수준 이하일 때에 세그먼트 선반입을 중지함으로써 문제를 해결할 수 있다.

본 실험에서는 선반입을 하지 않는 기존 방식에서도 세그먼트 크기가 성능에 미치는데, 그것은 본 실험이 이용한 레이드 드라이버에서 세그먼트가 메모리 관리 단위 및 최대 입출력 크기와 관련 있어서, 이런 것들이 성능에 영향을 미치기 때문이다. 공정한 비교를 위하여 기존 방식과 제안 하는 방식과의 메모리 관리 단위 및 최대 입출력 크기를 동일하게 설정하였다.

(그림 7)은 '일반 응용프로그램 사용' 트레이스에서 캐시 크기 변화에 따른 제안하는 기법의 성능 향상율을 보여준다. 캐시 메모리를 32MiB로 설정하였을 때에는 성능이 많이

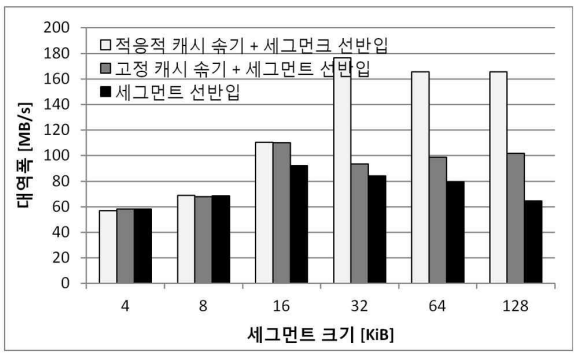


(그림 7) 일반 응용프로그램 사용 트레이스에서 캐시 크기 변화에 따른 제안하는 기법의 성능 향상율

낮아지고 64MiB이상에서 좋은 성능을 얻을 수 있었다. 특히 64KiB의 세그먼트 크기는 많은 선반입된 데이터를 위하여 더 많은 메모리를 사용하기 때문에 더 큰 캐시 메모리가 필요하다는 것도 알 수 있다.

이상의 실험 결과는 주로 세그먼트 선반입에 따른 성능이득이 대부분이다. 적응적 캐시 솎기는 사용되지 않는 선반입된 블록을 메모리에서 적절하고 적응적으로 퇴출하는 기법이다. 이 기법의 성능평가는 (그림 8)이 보여준다.

(그림 8)은 PCMark의 '일반 응용프로그램 사용' 트레이스를 다섯 번 반복하여 캐시 메모리 사용률을 평가하였다. 캐시 메모리의 크기를 128KiB로 설정하였고, 세그먼트의 크기를 4KiB에서 128KiB로 변화시켰다.



(그림 8) 캐시 크기 128KiB일 때에 캐시 메모리 필터링의 성능 평가. 고정 캐시 솎기는 적응적 캐시 솎기에서 L_1 크기를 적응적으로 변화시키지 않고 고정한 것

(그림 8)에서 고정 캐시 솎기는 적응적 캐시 솎기의 L1 크기를 적응적으로 변화시키지 않고 전체 캐시 메모리의 절반으로 고정한 방식이다. L1을 적응적으로 변화시키는 "적응적 캐시 솎기+세그먼트 선반입"과 L1의 크기를 고정한 "고정 캐시 솎기+세그먼트 선반입"을 비교하면, 그림 8에서는 세그먼트 크기가 32KiB일 때에 적응적 방법이 고정적 방법보다 최대 88% 더 높은 성능을 보였다. 즉, 큰 선반입 크기로 선반입 이득을 더 얻으면서 적절한 메모리 관리를 통하여, 큰 선반입의 단점을 극복하였다. 세그먼트가 128KiB일 때에 적응적 캐시 솎기 기법이 세그먼트 선반입만 사용하였을 때보다 성능을 2.6배까지 향상 시켰다

세그먼트 크기가 최소 입출력 단위인 4KiB이면 캐시 솎기 동작이 발생하지 않고 리스트 관리 부하만 포함하게 된다. "세그먼트 선반입"만 사용한 경우에는 리스트 관리 부하가 존재하지 않는다. 그러므로 4KiB 세그먼트 크기에서 "적응적 캐시 솎기+세그먼트 선반입"과 "세그먼트 선반입"의 성능차이가 리스트 관리 부하이다. 실험에서 약 2%의 관리 부하가 있는 것으로 측정되었다.

6. 결 론

개인용 컴퓨터의 일반 응용프로그램의 사용에 있어서는

세그먼트 선반입이 큰 효과가 나타나는 것을 보았다. 그것은 큰 페이지 크기 및 낸드 칩들의 병렬화와 관련된 것이었다. 즉, 세그먼트를 읽는 시간은 세그먼트 크기 미만을 읽는 시간과 크게 다르지 않기 때문에 큰 성능 이득이 생긴다.

적응적 캐시 솎기는 세그먼트 선반입의 메모리 사용효율 단점을 보완하는 기능을 한다. 이것은 선반입 히트와 캐시 히트의 합이 최대가 되도록 적응적 되먹임 방식으로 메모리를 관리하였다. 실제 시스템과 구현을 통한 실험에서 분명한 성능 이득을 확인하였으며, 리눅스 기반에서 본 기법을 적용하였을 때에 최대 2.6배까지 성능을 크게 향상 시켰다.

최근 SSD를 이용한 히스토리 기반 선반입은 정적이고 반복적인 워크로드에 이득이 있고, 일반적인 사용 패턴에서는 이득이 없다. 하지만 주로 사용하고 있는 순차 선반입은 HDD에 최적화되어 있어, SSD 리뷰 사이트들은 SSD가 설치된 개인 컴퓨터에서는 선반입을 사용하지 않도록 권고하고 있다. 이 논문은 SSD에 효과적이고, 부하가 적게 들고 일반적인 다양한 응용에 사용할 수 있는 실용적인 선반입 기법을 제안하였다. 리눅스 환경에서 기법을 적용하였을 때에 개인 컴퓨터의 일반 사용자 패턴에서 큰 이득을 얻었다.

제안하는 기술은 구현이 간단하고, 요구하는 메모리 및 연산 자원이 적게 들고, SSD를 구동하는 RAID 제어기 또는 SSD 내부의 선반입을 담당하는 펌웨어에 본 기술을 적용할 수 있다. 또한 실제 개인 컴퓨터를 위한 운영체제에 적용이 가능하다.

참 고 문 헌

[1] J. He, J. Bennett, and A. Snavely, “DASH-IO: an empirical study of flash-based- IO for HPC”, the 2010 TeraGrid Conference. Aug., 2010.

[2] K. Eshghi, “Enterprise SSDs with Unrivaled Perforrmance A Case for PCIe SSDs”, Flash Memory Summit 2010, Aug., 2010.

[3] A. Vasudeva, “Flash in Enterprise Storage Systems”, Flash Memory Summit 2010, Aug., 2010.

[4] Y. Wang, K. Goda, M. Nakano, M. Kitsuregawa, “Early Experience and Evaluation of Fiel Systems on SSD with Database Applications”, 5th IEEE Int'l Conf. on Networking Architecture and Storage. pp.476-476, July, 2010.

[5] M. Dunn and A.L.N. Reddy, “A New I/O Scheduler for Solid State Drives”, Tech. Rep. TAMU-ECE-2009-02, Department of Electrical and Computer Engineering, Texas A&M University, 2009.

[6] 김정기, 박승민, 김채규 “임베디드 플래시 파일 시스템을 위한 순위별 지움 정책”, 정보처리학회논문지 제9-A권 제4호 2002. 12.

[7] H. Kim and S. Ahn, “BPLRU: a buffer management scheme for improving random writes in flash storage”, the 6th USENIX Conf. on File And Storage Technologies, pp.1-14, Feb., 2008.

[8] B. Debnath, and S. Subramanya, D. Du, D.J. Lilja, “Large Block CLOCK (LB-CLOCK): a write caching algorithm for solid state disks”, IEEE Int'l Symp. on Modeling Analysis & Simulation of Computer and Telecommunication Systems. pp.1-9, Sept., 2009.

[9] S. Iyer and P. Druschel, “Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O,” Proc. USENIX Ann. Technical Conf., pp.117-130, 2001.

[10] J, Kim, Y. Oh, E. Kim, J. Choi. D. Lee, and S.H. Noh. “Disk schedulers for solid state drivers”. In Proc. EMSOFT, pp.295-304, Dec., 2009.

[11] F. Chang and G. A. Gibson, “Automatic I/O hint generation through speculative execution”, the 3rd Symposium on Operating Systems and Design and Implementation, pp.1-14, Feb., 1999.

[12] S. Bhattacharya, J. Tran, M. Sullivan and C. Mason. “Linux AIO performance and robustness for enterprise workloads”, Linux Symposium, pp.63-78, July, 2004.

[13] J. Griffioen, and R, Appleton, “Reducing file system latency using a predictive approach”, USENIX Summer Technical Conference, pp.197-208, June, 1994.

[14] D. Joseph and D. Grunwald, “Prefetching using markov predictors”, IEEE Trans. on Computers 48, 2, pp.121-133, Feb., 1999.

[15] Microsoft, “Windows PC Accelerators”, http://www.microsoft.com/whdc/system/sysperf/perfaccel.mspx, Last updated in Oct., 2010.

[16] Y. Joo, J. Ryu, S. Park, and K.G. Shin, “FAST: Quick Application Launch on Solid-State Drives”, USENIX Conf. on File and Storage Technologies, Feb., 2011.

[17] GILL, B. S., AND MODHA, D. S. SARC: Sequential prefetching in adaptive replacement cache. In Proc. USENIX 2005, pp.293 - 308, 2005.

[18] Hang-Ting Lue, Tzu-Hsuan Hs, and et al, “Study of Incremental Step Pulse Programming and STI edge effect of BE-SONOS NAND flash”, Reliability Physic Symposium, pp.693-694, April, 2008.

[19] Michael Abraham, “NAND Flash Trends for SSD/Enterprise”, Flash Memory Summit 2010 proceedings, August, 2010.

[20] Samsung Electronics, Inc. Ltd, “Flash Memory K9XXG08UXM datasheet”, 2011.

[21] Toshiba semiconductor, “NAND Interface: SmartNAND”, http://www.semicon.toshiba.co.jp/eng/product/memory/selection/nand/mlc/smartnand/index.html, 2011.

[22] “Disk sector”, Wikipedia, 2011.

[23] F. Chen, T. Luo, and X. Zhang, “CAFTL: a context-aware flash translation layer enhancing the lifespan of flash memory based solid state drives”, 9th USENIS Conf. on File and Storage Technologies, Feb., 2011.

[24] T. Makatos, Y. Klonatos, M. Marazakis, M.D.Flouris, and A. Bilas, “Using transparent compression to improve

SSD-based I/O caches", European conf. on Computer systems, April, 2010.
[25] The SSD Review, "SSD optimization guide", http://thessdreview.com/ssd-guides/optimization-guides/the-ssd-optimization-guide-2/, May, 2010.
[26] 정보성, 이정훈, "내장형 시스템을 위한 선택적 뱅크 알고리즘을 이용한 데이터 캐쉬 시스템", 한국정보처리학회 논문지 A, 제16-A권 제2호, 2009. 04.
[27] S. H. Baek and K. H. Park, "Matrix-Stripe-Cache-Based Contiguity Transform for Fragmented Writes in RAID-5", IEEE Transaction on Computers, Vol.56, No.8, pp.1040-1054, August, 2007.

백 승 훈
e-mail : shbaek@jwu.ac.kr
1997년 경북대학교 전자공학과(학사)
1999년 한국과학기술원 전기 및 전자공학과 (공학석사)
1999년~2005년 한국전자통신연구원 컴퓨터시스템연구부 연구원
2008년 한국과학기술원 전기및전자공학전공(공학박사)
2008년~2011년 삼성전자 메모리사업부 책임연구원
2011년~현 재 중원대학교 IT공학부 조교수
관심분야 : 컴퓨터저장장치, 운영체제, 컴퓨터구조 등