Journal of Digital Contents Society Vol. 14 No. 2 Jun. 2013(pp. 247-254)
http://dx.doi.org/10.9728/dcs.2013.14.2.247

스토리지 클래스 메모리에서의 파일 접근 설계

박성민*, 원유집**, 강수용***

요 약

PRAM, FRAM, MRAM 등 스토리지 클래스 메모리(SCM)는 가까운 미래에 접근 속도는 DRAM에 용량은 플래시 메모리에 근접할 것으로 예상된다. 따라서 SCM이 컴퓨터 시스템에서 메모리(DRAM)뿐만 아니라 스토리지(하드디스크 혹은 플래시 메모리)를 대체할 수 있을 것이다. 이 논문에서는 SCM 기반 컴퓨터 시스템을 위한 효율적인 파일 접근 프레임워크를 제안한다. 제안하는 프레임워크는 SCM에 파일 저장을 위한 영역과 메모리 사용을 위한 영역을 구분하지 않는다. 또한 제안하는 프레임워크는 파일 관리를 위하여 단일 데이터 접근 경로, 파일 매핑을 통한 제로 카피 데이터 읽기, 카피 온 라이트 기반 데이터 쓰기, 다수 페이지 프리 폴팅 등 다양한 메모리 관련 기술들을 사용한다. 주요 실험 결과를 통해서 논문에서 제안하는 프레임워크는 SCM 기반 컴퓨터 시스템의 운영체제 디자인을 위한 초석이 될 것이다.

키워드 : 스토리지 클래스 메모리, 파일 시스템, 페이지 폴트

A Design for File Access in Storage Class Memory-based Computer Systems

Sungmin Park*, Youjip Won**, Sooyong Kang***

Abstract

Storage Class Memory(SCM), such as PRAM, FRAM and MRAM, are expected to be comparable to DRAM in terms of access speed and to Flash memory in terms of capacity in a near future. In this paper, assuming that not only the secondary storage (HDD or Flash memory) but also the primary memory (DRAM) will be replaced by SCM in the future computer systems, we propose an efficient file access framework for the SCM based computer systems. The proposed framework do not assign exclusive area in the SCM to the file system and uses various memory-related techniques, such as unified data access path, zero-copy data read using file mapping, copy-on-write, and multiple page pre-faulting for file management. Based on the preliminary experimental results, we could conclude that the proposed framework can be an efficient baseline for designing a new operating system for the SCM based computer systems.

Keywords : Storage class memory, Filesystem, Page fault

1. 서론

빠르게 발전하는 반도체 기술로 인하여, 가까운 미래에 대용량의 스토리지 클래스 메모리(SCM)가 시장에 등장할 것으로 예상한다. 블록 접근이 가능한 SCM(eg. NAND)는 이미 모바일

※ 교신저자(Corresponding Author): Sooyong Kang
접수일:2013년 05월 02일, 수정일:2013년 06월 19일
완료일:2013년 06월 27일
* 한양대학교 전자컴퓨터통신 공학과
email: syrilo@hanyang.ac.kr
** 한양대학교 컴퓨터공학부 컴퓨터학과
*** 한양대학교 컴퓨터공학부 컴퓨터학과
Tel: +82-2-2200-1725,
email: sykang@hanyang.ac.kr

▣ 본 연구는 2013년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (No. 2012-047724)

컴퓨팅 시장에서 주요 스토리지로서 자리를 잡았고 곧 PC 시장과 서버 시장에서도 기존 하드디스크를 대체할 디바이스로서 주목을 받고 있다[1]. 최근에는 바이트 접근이 가능한 SCM이 등장하여 스토리지 디바이스뿐만 아니라 메모리 디바이스로 활용될 것으로 예상되고 있다. 이러한 SCM의 예로는 PRAM, MRAM, FeRAM, Solid electrolyte, SPIN[2] 등이 있다. 이미 MRAM, FeRAM은 임베디드 시스템에서 사용되고 있다. 우리는 조심스럽게 5년 안에 무어의 법칙에 따라 바이트 접근 가능한 대용량 스토리지 클래스메모리가 등장할 것으로 예상한다. 최근에 삼성에서는 512Mbits PRAM 개발을 발표하였고[3] 또한 인텔과 ST 마이크로 일렉트로닉스에서도 4-states MLC(Multi level Cell) PRAM 기술을 발표하였다[4]. 위와 같은 사실에 따라 우리는 가까운 미래에 바이트 접근 가능한 스토리지 클래스메모리가 곧 시장에 등장할 것이라고 예상할 수 있다. 하지만 빠른 반도체 발전에 비해 아직까지 SCM이 기존 컴퓨터 시스템의 메모리 계층에서 어떻게 활용될지에 대해서는 분명하지 않다. SCM의 접근 속도는 DRAM에 가까우며 용량은 낸드기반 플래시 메모리에 가까울 것으로 예상되므로 그것은 메모리 혹은 스토리지로 대체가 가능하다. 혹은 메모리와 스토리지 사이의 새로운 레이어의 디바이스로서 그 역할을 할 수도 있다. SCM의 대부분의 연구는 SCM을 기존 스토리지의 신뢰성과 성능 향상을 위한 하이브리드 스토리지 측면에서 연구되었다. 이러한 연구에는 HeRMES[5], Conquest[6], FRASH[7]가 있다. 그리고 SCM을 스토리지로 사용하기 위한 전용 파일 시스템 연구인 PRAMFS[8]와 SCMFS[9]가 있다. 이에 반해 SCM을 메모리로서 활용한 PDRAM[10] 연구도 있다. 또한 SCM을 메모리와 스토리지로서 동시에 사용할 수 연구도 진행되었다[11].

본 논문에서는 SCM의 바이트 접근 가능한 특징과 대용량의 특징을 모두 고려하여 SCM을 메모리인 동시에 스토리지 디바이스로서 사용할 것이다. 즉 기존 DRAM과 하드디스크 같은 2차 스토리지를 대체하여 오직 SCM만이 존재하는 컴퓨팅 시스템을 가정한다. SCM만을 사용하는 컴퓨터 시스템에서는 기존 운영체제의 두 개의 데이터 접근 경로 수정을 요구한다. 기존 데이터 접근 방법으로는 load/store를 명령어를 통한 메인 메모리 접근 경로와 파일 읽기/쓰기 명령어를 통한 스토리지 접근 경로가 있다.

본 논문에서는 SCM 기반 컴퓨터를 위한 새로운 프레임워크를 제시한다. 새로운 프레임워크에서는 SCM에 파일 시스템을 위한 영역과 메모리 영역을 구분하여 할당하지 않는다. 이것은 SCM은 여러 종류의 페이지의 집합으로 구성된다고 가정하기 때문이다. 즉 메모리 데이터와 스토리지 데이터는 하나의 페이지 풀을 통해서 할당받은 페이지들에 저장된다. 새로운 프레임 워크에서 통합된 데이터 접근 경로 , 제로 카피 읽기, 카피 온 라이트 기반 파일 쓰기, 다수 페이지 프리 폴팅 기법들을 사용한다.

2. 본론: SCM 프레임워크 디자인

지금의 컴퓨터 시스템 구조는 메모리 디바이와 2차 스토리지가 구분되어 있기 때문에 운영체제에서는 메모리 접근 방법과 파일 접근 방법의 두 개의 데이터 접근 방법을 제공해 주고 있다. 하드디스크 같은 상대적으로 느린 2차 저장장치의 데이터 접근을 위하여 운영체제에서 복잡한 I/O 기술들을 제공한다. 하지만 이런 기술들 때문에 운영체제는 복잡성이 증가하였고 또한 신뢰성도 떨어지게 되었다. 만약 DRAM과 같은 메모리와 2차 저장장치가 하나의 스토리지 클래스 같은 하나의 디바이스로 통일된다면 이론적으로 두 개로 나누어져있던 데이터 경로 도 하나로 통일될 수 있다. 기존의 2차 저장장치를 접근하기 위한 파일 경로 제거한다면 운영체제로부터 복잡한 I/O 기술들을 제거함으로써 운영체제를 상당히 가볍게 만들 수 있을 것이다. 또한 SCM 컴퓨팅 시스템에서는 기존의 메모리 관리를 위한 I/O 버퍼, 가상 메모리, 페이지 캐시와 파일 시스템을 하나로 통합할 수 있을 것이다.

이 논문의 이후에는 SCM 기반 컴퓨팅 시스템을 위해 제안하는 프레임워크를 SCMFW라는 용어로 사용할 것이다.

2.1 동적 페이 할당과 단일 데이터 접근 경로

SCMFW은 세 가지 메모리 페이지 타입을 갖는다(메모리 페이지, 스토리지 페이지, 메모리 - 스토리지 페이지). 메모리 페이지는 커널과 유저 프로세스를 위한 워킹 공간으로 활용된다. 메모리 페이지는 기존 운영체제의 물리적 메모리 공간과 같은 방식으로 관리된다. 스토리지 페이지는 파일 데이터로서 영구적인 데이터를 나타낸다. 예를 들면 파일 데이터와 파일 메타데이터가 있다. 이것은 기존 운영체제의 파일 시스템의 블록과 같은 방식으로 관리된다. 메모리-스토리지 페이지는 논리적인 개념으로서 프로세스가 파일을 접근할 때 결정된다. 그것은 파일을 오픈한 프로세스가 접근하는 파일 데이터를 나타낸다. 파일을 오픈한 프로세스들만이 이 데이터를 메모리 - 스토리지 파일로 간주하게 되고 다른 프로세스는 이 데이터를 스토리지 페이지로 인식하게 된다. 프로세스가 파일을 오픈할 때, 우선적으로 파일 시스템 인터페이스를 통해서 파일 inode의 위치를 파악한다. 파일 inode에 접근 후에 프로세스는 파일 데이터가 저장된 스토리지 페이지들의 위치를 파악할 수 있다. 이 이후부터 프로세스는 파일 데이터를 읽기 위하여 그 스토리지 페이지를 메모리-스토리지 페이지로서 간주하게 된다. 그 프로세스가 파일을 닫을 때, 파일의 메모리-스토리지 페이지들은 다시 스토리지 페이지가 된다. 그러므로 프로세스가 그 파일을 다시 열게 되면 다시 위에 열거한 과정을 되풀이해야 할 것이다. 스토리지 페이지들은 공간이 요구될 때 동적으로 할당되기 때문에 메모리 공간 전체 어느 곳이나 위치할 수 있다. 동적 페이지 할당을 위해서 SCMFW의 페이지 매니져는 페이지 비트맵을 관리하고 메모리 페이지와 스토리지 페이지를 공통 페이지 풀로부터 할당한다.

SCMFW에서의 페이지 접근 경로는 기존 운영체제의 메모리 접근 경로와 같다. SCM의 모든 페이지들은 메모리 경로에 따라 접근된다. 물론 스토리지 페이지에 파일들은 파일 시스템 인터페이스를 통해서 그 위치가 파악된다. 그러나 그 위치가 파악된 후에는 그 파일을 오픈한 프로세스가 메모리 경로를 통해서 접근한다. 이때 그 프로세스들은 파일 데이터를 메모리-스토리지 페이지로서 접근하는 것이지 스토리지 페이지로서 접근하는 것이 아니다. 위에서 설명했듯이 그 페이지들은 파일 오픈 시 스토리지 페이지에서 메모리-스토리지 페이지로 변경되기 때문이다. 그러므로 어떠한 경우에도 페이지들은 스토리지 페이지일 때 접근될 수 없다. I/O 단위 관점에서 메모리 페이지들과 메모리-스토리지 페이지들은 읽을 때는 워드 단위로 접근이 되며 쓸 때는 원자적 쓰기를 위해 페이지 단위로 접근이 된다.

2.2 제로 카피 파일 읽기

기존 운영체제에서는 read() 시스템 콜에 의해서 발생된 페이지 폴트를 통해서 오픈된 파일의 데이터에 최초로 접근한다. 페이지 폴트 핸들러는 디스크로부터 페이지 캐시에 데이터를 읽고 그 후에 유저 스페이스 즉 그 프로세스 페이지 테이블에 물리적 주소가 있는 유저 버퍼로 데이터를 읽는다. 이런 과정 속에서 빈번한 시스템 콜이 발생하지만 더 이상 페이지 폴트(소프트 페이지 폴트)는 발생하지 않는다. PRAMFS[8]는 페이지 캐시를 사용하지 않고 바로 유저버퍼로 데이터를 복사하는 direct I/O를 사용한다. PRAMFS는 direct I/O를 통하여 한 번의 메모리 복사를 제거하였다. 하지만 이것은 아직까지 한 번의 데이터 복사를 발생 시킨다. 만약 SCM에서 메모리 복사 없이 파일 접근하려면 memory-mapped 파일 I/O가 훌륭한 대안이 될 것이다. memory-mapped 파일 I/O는 커널의 페이지 캐시 대부분에 파일이 있다면 매우 효율적으로 동작할 수 있다. 따라서 이미 파일 데이터가 메모리에 존재하는 SCM 컴퓨터 시스템의 파일 접근 설계를 위해 memory-mapped 파일 I/O는 훌륭한 예제가 될 수 있다. 그러나 memory-mapped 파일 I/O는 몇 가지 문제점을 가지고 있다. 첫 번째로 그 것은 페이지 캐시에서 동작한다. 프로세스가 처음으로 memory-mapped 파일의 일부분을 접근할 때 페이지 폴트 핸들러는 데이터를 페이지 캐시로 저장하고 해당 페이지의 물리적 주소를 페이지 테이블에 저장하게 된다. 그러난 SCMFW에서는 페이지 캐시를 사용하지 않는다. 두 번째로 파일 하나의 페이지에 접근할 때마다 페이지 소프트 페이지 폴트를 발생시키고 그것은 성능에 심각한 문제를 야기할 있다.

페이지 캐시 없이 제로 카피 파일 읽기를 위

해서는 SCMFW은 파일이 오픈되는 시점에 프로세스 주소 공간에 파일을 매핑하고 다수 페이지 프리 폴팅 기술을 통하여 페이지 테이블 엔트리를 채우게 된다. 빠르게 페이지 테이블을 만들기 위해서 파일 메타데이터는 파일 데이터 가진 스토리지 페이지의 물리적 주소 공간의 포인터의 집합을 관리하고 있다. 이런 방식으로 프로세스는 페이지 캐시가 아닌 파일의 메모리-스토리지 페이지로부터 직접 파일을 접근할 수 있게 된다. 따라서 제로 카피 파일 읽기가 가능해지는 것이다.

2.3 카피 온 라이트 파일 쓰기와 원자적 파일 쓰기

하드디스크 안에 블록에 대한 ECC가 있기 때문에 기존 운영체제의 파일 시스템을 위해서 블록 혹은 페이지 단위의 원자성을 제공한다. 그러나 바이트로 접근하는 DRAM 같은 메모리는 블록 혹은 페이지 단위의 ECC가 존재하지 않는다 ('buffered RAM'은 바이트 단위의 ECC를 제공 [12]). 따라서 이것은 스토리지 클래스 컴퓨터 시스템에서 파일 수정을 위한 페이지 수준 원자성을 제공하는 것을 어렵게 만든다. 그러므로 원자성을 제공하기 위해서 운영체제 수준에서 SCM에 원자적 쓰기를 제공해야 한다. 본 논문에서는 파일 데이터를 메모리-스토리지 에 쓰거나 메타데이터를 스토리지 페이지에 쓸 때 페이지 단위의 원자적 쓰기 기법을 제공한다.

메모리-스토리지 페이지를 수정할 때 원자적 수정을 제공하기 위해서 SCMFW은 카피 온 라이트 기법을 사용한다. 프로세스가 메모리- 스토리지 페이지에 쓸 때, 페이지 매니저는 한 개의 메모리 페이지를 프로세스에게 할당하고 메모리-스토리지 페이지는 새로운 페이지에 복사한다. 관련된 페이지 테이블 엔트리는 새로운 페이지 주소로 수정을 하고 실제 쓰기는 그 페이지에 발생 시킨다. 원자적 쓰기가 끝난 후에는 원본 페이지는 페이지 매니저가 프리 페이지로 반환시킨다. 이때부터 프로세스는 새로운 페이지에 읽기와 쓰기 접근을 한다. 수정의 효율성을 위해서 여러 번의 수정을 한 번의 데이터 복사로 해결한다. 이런 방법으로 SCMFW은 파일 수정을 위해 오직 페이지 단위의 원자적 쓰기만을 허용한다.

2.4 파일 생성과 삭제

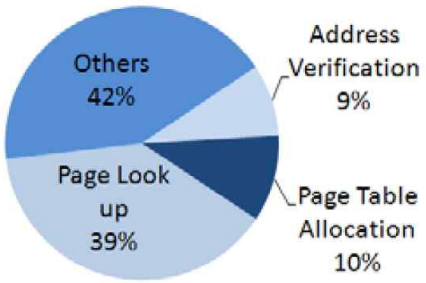
프로세스가 파일을 생성할 때, 프로세스와 파일 시스템은 파일데이터를 위해서 메모리 페이지를, 파일 메타데이터를 위해서 스토리지 페이지를 각각 페이지 매니저에게 요청한다. 이때는 쓰인 데이터가 없기 때문에 파일 데이터를 위한 스토리지 페이지가 없다. 프로세스가 메모리 페이지에 파일 데이터를 쓴 후에 원자적 쓰기 연산은 데이터를 스토리지 페이지에 저장하게 된다. 파일 메타데이터를 위한 스토리지 페이지 역시 파일 시스템이 원자적 쓰기를 이용해서 수정하게 된다.

파일이 삭제될 파일 시스템은 페이지 매니저에게 파일에 할당된 스토리지 페이지를 반환한다. 그리고 관련된 디렉토리 엔트리는 원자적 쓰기에 의해서 수정된다.

2.5 다수 페이지 프리 폴팅

SCMFW은 파일을 매핑할 때 페이지 테이블 엔트리에 페이지 프리 폴팅 기법을 사용한다. 페이지 테이블 만드는 가장 쉬운 방법은 파일이 매핑될 때 파일에 해당되는 모든 페이지에 대하여 페이지 테이블 엔트리를 채우는 것이다. 이것은 리눅스 커널 mmap()의 MAP_POPULATE 옵션과 같다. 하지만 이것은 큰 파일인 경우 파일 오픈 시 초기 지연이 문제가 된다. 그러므로 스토리지 클래스 컴퓨터 시스템에 적합한 새로운 프리 폴팅 기법이 필요하다. SCMFW에서는 하나의 페이지 폴트 루틴에 여러 개의 페이지 테이블 엔트리 만드는 다수 페이지 폴팅 기법을

(그림 1) 페이지 폴트 수행 시간 분석



(Figure 1) Page fault overhead breakdown

사용한다. 이 기법을 고안하기 위해서 우선적으로 리눅스 커널 2.6.18 32bit 커널의 페이지 폴트 비용을 분석하였다. (그림 1) 은 페이지 폴트 비용 분할 그래프이다.

페이지 폴트 핸들링 비용(PF)는 주소 검증 비용(AV)와 페이지 테이블 엔트리 할당 비용(PTA)와 페이지 검색 비용 (PL)과 모드 스위치 비용(MS)로 구성된다. 주소 검증은 해당 페이지의 주소가 적합한지를 체크하는 것이고 페이지 검색은 파일 시스템에서 해당 페이지를 찾는 작업이다. (그림 1) 로부터 다음과 같은 공식을 구할 수 있다.

PF = MS + AV + PL + PTA, where MS = 0.42PF, AV = 0.14PF, PL = 0.32PF, PTA = 0.12PF

만약 페이지 폴트 루틴이 수정할 때 한번의 루틴에 여러 페이지를 수정한다면 그 비용은 다음과 같다.

n-PF = (MS + AV + PL) + nPTA

= 0.88PF + n0.12PF (수식 1)

여기서 n-PF는 n개의 페이지를 한번의 페이지 폴트로 페이지 테이블 엔트리를 만들 때의 비용이다. 만약 프리 폴팅에 위한 성능 이 득(G)를 G = nPF/n-PF 로서 정의한다면, 이것은 G = n/(0.88 + 0.12n) 으로 나타낼 수 있고 이것은 n이 증가할수록 8.34로 수렴하게 된다. G의 값을 통해서 SCMFW를 위한 프리 폴팅 페이지 개수는 4~16pages (16 ~ 64KB)가 적당하다는 것을 알 수 있다. 이것은 많은 페이지의 프리 폴팅은 다소 성능 향상은 있게 지만 무시할 수 없을 만큼의 런 타임 지연을 가질 수 있기 때문이다.

이렇게 설계된 프리 폴팅 기법은 현재 운영체제의 프리 페칭 기법과 매우 유사함을 알 수 있다. 파일에 처음 페이지 폴트가 발생하면 네 개의 연속된 페이지에 프리 폴트를 발생 시킨다. 만약 마지막 페이지에 다음 페이지에 페이지 폴트가 발생한다면 프리 폴팅 횟수를 두 배로 증가 시킨다. 만약 그렇지 않다면 두 배 감소시킨다. 최대 프리 폴팅 개수는 16개로 제한한다. 다수 페이지 프리 폴팅 기법은 과도한 지연 없이 큰 성능 이득을 볼 수 있다.

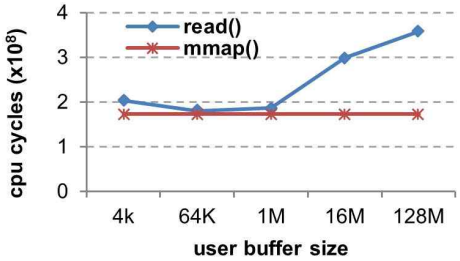
3. 실험 결과

제안하는 프레임워크의 성능을 검증하기 위하여 리눅스 2.6.18 32bit 커널과 512MB 램 위해서 몇 가지 실험을 수행하였다. PRAMFS 가 32bit 커널에만 동작하기 때문에 성능을 비교하기 위해서 32bit 커널을 사용해야만 했다. 또한 512MB 메모리를 사용한 이유는 64bit OS에서는 하이메모리를 사용하지 않기 때문에 32bit 커널에서 하이메모리를 사용하지 않으려면 메모리를 제한적으로 사용해야 하기 때문이다.

본 논문에서 제안하는 프레임워크를 에뮬레이션하기 위해서 tmpfs[13]를 사용하였다. 첫 번째로 본 논문에서는 기존의 파일 접근 방법인 read ()와 mmap()의 성능을 평가하였다. read() 시스 템 콜은 파일 시스템을 통한 전형적인 파일 접근 방법이고 mmap()은 제안하는 프레임워크 의 접근 방법을 나타낸다.

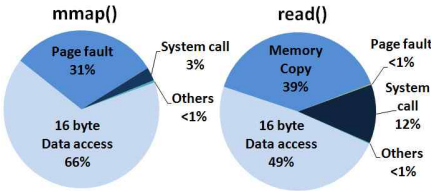
(그림 2) 는 tmpfs에 저장된 128MB 파일을 연속적으로 접근 시 소비한 CPU 사이클을 보여준다. CPU 사이클을 OProfile 툴[14]을 사용하여 측정하였다. 실험은 128MB 파일을 16byte 청크 단위로 연속적으로 접근하였다. 즉 mmap()을 위해서는 파일로부터 직접 16byte 청크를 직접 접근하였고 read()를 위해서는 파일로부터 일정한 크기의 데이터를 유저 버퍼로 복사해 온 후 16byte 청크를 접근하였다. 이 실험에서는 mmap 사용 시에 미리 페이지 폴트를 발생시키지 않았다. 즉 4Kbyte 단위로 접근할 때마다 페이지 폴트가 발생하도록 하였다. read() 접근시에는 32,768 시스템 콜이 발생한 반면에 mmap()은 단 한번의 시스템 콜만 발생하였다. 또한 read() 접근 시에는 버퍼에 최초 접근 시에만 4KB 맏다 페이지 폴트가 발생하였으며(예를 들어 유저 버퍼의 크기가 4KB이면 한번만 발생한다.) 32,768개의 페이지 복사가 페이지 캐시에서 유저 버퍼로 발생하였다. mmap() 접근 시에는 페이지 폴트만이 발생하였다. 따라서 두 실험은 시스템 콜과 메모리 복사의 비용과 페이지 폴트의 비용을 비교하는 것이라 볼 수 있다. (그림 2) 에서 볼 수 있듯이 mmap()은 read()보다 항상 성능이 우수하다. 하지만 버퍼 사이즈가 작을 때는 그 성능 차이가 크지가 않다. 이 이유는 두

(그림 2) read()와 mmap() 경로의 수행 시간



(Figure 2) Elapsed time: read() and mmap() paths

(그림 3) 두 경로의 오버헤드 분석



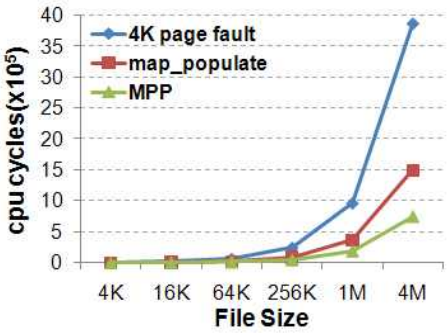
(Figure 3) Overhead breakdown: two paths

시스템 콜의 성능이 비슷한 read() 버퍼 크기가 64KB일 때의 각 시스템콜의 수행 시간을 분석한 (그림 3) 을 통해서 알 수 있다.

(그림 3) 에서 만약 두 시스템 콜의 비용이 같다고 가정한다면 read() 접근 시 16byte 데이터 접근 비용은 mmap() 보다 작아진다. 이것은 read() 접근 시에 mmap() 보다 더욱 효과적으로 동작하는 것으로 분석되었다. 위 그래프에 볼 수 있듯이 페이지 폴트 비용을 프리 폴팅을 통해서 줄이면 mmap()의 성능을 상당히 줄일 수 있다는 것을 예측할 수 있다.

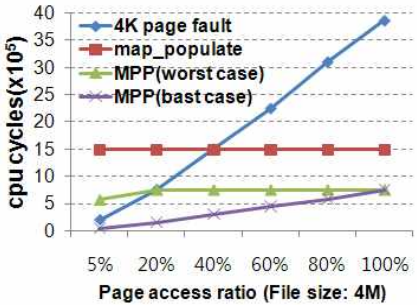
(그림 4) 와 (그림 5) 는 4MB 파일을 각각 연속적과 임의적으로 접근할 때의 페이지 폴트 비용을 보여준다. (그림 5) 의 x축은 4MB의 파일의 접근율을 나타낸다. 4K 페이지 폴트와 map_populate는 리눅스의 mmap() 시스템 콜과 MAP_POPULATE 플래그를 사용하여 측정하였다. MPP는 다수 페이지 프리 폴팅을 나타낸다. 이것은 실제 측정한 값이 아니고 (수식 1) 에 따라 예측된 값이다. map_populate는 4K page fault에 비해서 약 40% 비용만을 가진다. map_populate는 모드 스위치와 주소 검증 비용

(그림 4) 연속 접근 수행시간



(Figure 4) Sequential access overhead

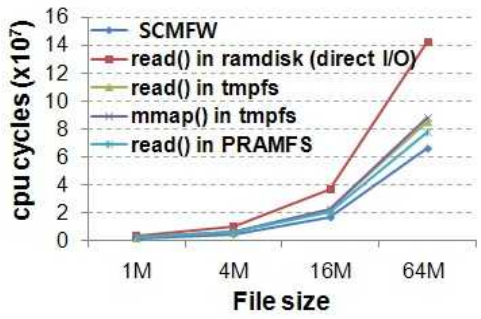
(그림 5) 임의 접근 수행시간



(Figure 5) Random access Overhead

을 줄일 수 있기 때문이다. 이에 반하여 MPP는 20%의 비용만을 가진다. 초기 3개의 페이지만을 제외하고는 16개의 페이지를 한 번에 프리 폴팅 시키기 때문이다. map_populate는 페이지 테이블을 초기에 전부 만들기 때문에 임의적 접근을 할 때에도 항상 일정한 값을 지니게 된다. 이에 반하여 4K page fault는 접근율이 증가할수록 이에 비례하여 그 값이 증가하게 된다. 파일 접근율이 40%가 넘어서게 되면 map_populate가 4K page fault 보다 성능이 나아지게 된다. MPP(worst)는 페이지가 접근될 때 6개의 페이지 중 단 하나의 페이지만이 접근될 때의 그래프이다. 즉 16개의 간격으로 페이지가 접근될 때 MPP는 최악의 성능을 가진다. 이 경우 페이지 20%만 접근되어도 전체 페이지 테이블이 만들어진다. 이에 반하여 MPP(best) 프리 폴팅한 페이지는 항상 접근될 때를 가정한 것이다. 이 경우 페이지 20%만 접근되어도 전체 페이지 테이블이 만들어진다. 실사용 컴퓨팅 환경에서는

(그림 6) 전체 수행 시간



(Figure 6) Overall Overhead

MPP(worst)와 MPP(best) 사이에서 동작할 것이다. 이 실험 결과를 통해서 MPP는 항상 MAP_POPULATE 보다 성능이 좋으며 일반적인 페이지 폴트보다 파일 접근율이 20% 보다 높을 때 우수한 성능을 보인다는 것을 알 수 있다.

(그림 6) 은 제안하는 프레임워크와 다른 시스템과의 성능을 비교한 것이다. 실험의 그래프는 각각의 파일을 연속적으로 접근했을 때의 CPU 사이클을 측정하였다. SCMFW은 제안하는 프레임워크를 나타낸다. 이 결과는 제안하는 프레임워크가 기존 시스템에 비하여 SCM 기반 컴퓨팅 시스템에서 우수한 성능을 가진다는 것을 보여준다.

4. 결론

이 논문에서는 SCM만을 가진 컴퓨팅 시스템에서 파일 접근을 위한 프레임 워크를 제안하였다. 제안한 프레임워크는 프로세스 메모리 영역과 파일 시스템 메모리 영역을 동적으로 할당한다. 프레임워크는 단일 데이터 접근, 파일 매핑을 통한 제로 카피 데이터 읽기, 카피 온 라이트, 다수 페이지 프리 폴팅 등 다양한 기술들을 사용하였다. 제안한 프레임워크는 SCM만을 가진 컴퓨팅 시스템에서 기존의 복잡하고 비효율적인 파일 접근 방법을 개선하여 효율적으로 파일을 접근한다는 것을 실험 통하여 검증하였다.

References

[1] Hoyoung Jung, Sooyong Kang, and Jaehyuk Cha. “An Asymmetric Buffer Management Policy for SSD”, Journal of digital contents society, vol. 12, no. 2, pp. 141-150, 2011.

[2] F. Freitas. Storage class memory: technology, systems and applications. In SIGMOD, 2009.

[3] http://www.samsung.com/global/business/semiconductor/news-events/press-releases/detail?newsId=4219

[4] F. Bedeschi1 et al, “A Multi-Level-Cell Bipolar-Selected Phase-Change Memory”, Proceedings of the International Solid State Circuits Conference (ISSCC’ 2008), Feb. 2008

[5] E. L. Miller, S. A. Brandt, and D. D. E. Long, "HeRMES: High-Performance Reliable MRAM Enabled Storage," In Proceedings of the 8th HotOS, pp. 95-99, 2001.

[6] A. I. A. Wang, G. Kuenning, P. Reiher, and G. Popek, "The Conquest File System: Better Performance Through a Disk/Persistent-RAM Hybrid Design," ACM Transactions on Storage, 2(3):309-348, 2006.

[7] Jaemin Jung, YoujipWon, Eunki Kim, Hyungjong Shin, and Byeonggil Jeon. “Frash:Exploiting storage class memory in hybrid file system for hierarchical storage,” ACM Transactions on Storage, 6(1):1.25, 2010.

[8] PRAMFS, http://pramfs.sourceforge.net/

[9] X. Wu and A. L. N. Reddy. “SCMFS: a file system for storage class memory,” In Proc. of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11), pp. 1-11, 2011.

[10] Gaurav Dhiman , Raid Ayoub , Tajana Rosing, “PDRAM: a hybrid PRAM and DRAM main memory system,” Proceedings of the 46th Annual Design Au

tomation Conference, July 26-31, 2009, San Francisco, California

[11] Seungjae Baek, Jongmoo Choi, "A Unified Software Architecture for Storage Class Random Access Memory," KIISE CST, Vol. 36, No. 3, pp. 171~180, 2009. 6.

[12] http://en.wikipedia.org/wiki/Fully_Buffered _DIMM

[13] P. Snyder, "tmpfs: A virtual memory file system," Proceedings of the Autumn 1990 European UNIX Users' Group Conference, Nice, France, Oct 1990.

[14] OProfile, http://oprofile.sourceforge.net/

박 성 민

2007년 : 한양대학교 전자컴퓨터통신 대학원(공학석사)
2007년 ~ 현 재: 한양대학교 전자컴퓨터통신(박사과정)

관심분야 : 플래시 메모리, 임베디드 시스템, 운영체제

원 유 집

1992년 : 서울대학교 컴퓨터 과학 대학원 (공학석사)
1997년 : 미네소타 컴퓨터 과학 대학원 (공학박사)

1997년~1999년: 인텔
1999년~현 재: 한양대학교 컴퓨터 공학부 교수
관심분야 : 멀티미디어 시스템, 네트워킹, 파일 시스템, 스토리지 시스템, 저전력 시스템

강 수 용

1998년 : 서울대학교 컴퓨터 과학 대학원 (공학석사)
2002년 : 서울대학교 컴퓨터 과학 대학원 (공학박사-분산컴퓨터 시스템)

2002~현 재: 한양대학교 컴퓨터 공학부 교수
관심분야 : 운영 체제, 멀티미디어시스템, 스토리지 시스템, 플래시 메모리, 차세대 메모리, 분산 컴퓨팅