파일시스템을 내장한 저장장치의 설계, 구현 및 성능분석

(Design, Implementation, and Performance Evaluation of File System on a Chip)

안 성 준 † 최 종 무 †† 이 동 희 †††
(Seongjun Ahn) (Jongmoo Choi) (Donghee Lee)
노 삼 혁 †††† 민 상 렬 ††††† 조 유 근 †††††
(Sam H. Noh) (Sang Lyul Min) (Yookun Cho)

요 약 이동식 저장장치는 다양한 호스트에서 데이타 전달 및 공유를 위해 사용될 수 있기 때문에 상호운영성의 지원이 필수적이다. 그러나 파일시스템의 기능이 호스트에서 수행되는 경우, 서로 다른 파일시스템을 사용하는 호스트들에 대해서는 상호운영성을 지원하지 못하는 문제가 발생한다. 본 논문에서는 이동식 저장장치의 상호운영성을 향상시키기 위한 방법으로 파일시스템을 내장한 저장장치인 FSOC(File System On a Chip)를 제안하며, 이의 설계 및 구현의 예를 보인다. 또한 기존 저장장치와 FSOC의 성능 모델을 제시하고, 제시한 모델을 기반으로 기존 저장장치와 FSOC의 성능 차이를 분석하며, 구현된 FSOC를 이용한 실험을 통해 제시한 성능모델을 검증한다.

키워드 : 이동식 저장장치, 파일시스템, 상호운영성, 시스템 성능분석

Abstract Interoperability is an important requirement of portable storage devices that are used to exchange and share data among diverse hosts. However, the required interoperability cannot be provided if different host systems use different file systems. To address this problem, we propose a new type of storage device called FSOC(File System On a Chip) that contains the file system within the storage device. In this paper, we give an example of the design and implementation of a flash memory-based FSOC and propose the performance models of the conventional storage device and the FSOC. We also analyze the performance characteristics of the conventional storage device and the FSOC based on the proposed performance models, and provide several experimental results using real applications that validate the performance models.

Key words : portable storage device, file system, interoperability, performance analysis

1. 서 론

최근 들어 디지털카메라, MP3 플레이어, PDA와 같은 휴대용 정보기기의 사용이 증가함에 따라 휴대용 정보기기의 비휘발성 보조기억장치로 사용되는 플래시메모리에 기반한 콤팩트플래시, 멀티미디어카드, 스마트미디어 등의 이동식 저장장치의 사용이 증가하고 있다. 이러한 이동식 저장장치는 단일 호스트의 비휘발성 보조기억장치의 기능을 제공할 뿐 아니라 다양한 호스트 사이의 데이타 전달 및 공유를 위해서도 사용된다. 따라서 이동식 저장장치는 다양한 호스트에 대한 상호운영성을 지원하는 것이 필수적이다. 그러나 저장장치에 사용된 파일시스템과 호스트의 파일시스템이 호환되지 않는다

· 본 논문은 2002년도 서울시립대학교 학술연구조성비의 지원에 의해 연구되었음
† 비 회 원 : 서울대학교 전기컴퓨터공학부
sjahn@ssrnet.snu.ac.kr
†† 종신회원 : 단국대학교 정보컴퓨터공학부 교수
choijm@dku.edu
††† 비 회 원 : 서울시립대학교 컴퓨터과학부 교수
dhlee@venus.uos.ac.kr
†††† 종신회원 : 홍익대학교 정보컴퓨터공학부 교수
noh@cs.hongik.ac.kr
††††† 종신회원 : 서울대학교 전기및컴퓨터공학부 교수
symin@dandelion.snu.ac.kr
cho@cse.snu.ac.kr
논문접수 : 2004년 3월 30일
심사완료 : 2004년 8월 25일

면 해당 호스트는 저장장치에 저장된 데이타에 접근할 수 없다는 문제가 발생한다.

본 논문에서는 이동식 저장장치의 상호운영성을 향상시키기 위한 방법으로 파일시스템을 내장한 저장장치인 FSOC(File System On a Chip)를 제안한다. FSOC는 파일시스템의 기능을 저장장치 내부에서 수행함으로써 다음과 같은 장점을 가진다.

첫째, FSOC는 호스트에 대해 높은 상호운영성을 지원한다. 기존의 저장장치를 사용하면 호스트가 동일한 파일시스템을 사용할 때에만 저장장치에 저장된 데이타에 접근이 가능한 반면, FSOC를 사용하는 경우 호스트에 FSOC와의 인터페이스 역할을 수행하는 간단한 스텁(stub)만 추가하면 다른 파일시스템을 지원하는 호스트 간에도 데이타를 전달할 수 있기 때문에 상호운영성이 향상된다.

둘째, 저장장치에 특화된 파일시스템을 사용하여 저장장치의 효율을 높일 수 있다. 호스트에 구현된 파일시스템의 경우 다양한 특성의 저장장치를 지원해야 하므로 특정한 저장장치에 최적화된 구현이 어렵지만[1], 파일시스템을 저장장치에 내장하는 경우 해당 저장장치의 특성에 최적화된 파일시스템의 구현이 가능하다[2-4].

셋째, FSOC를 사용하면 호스트에 파일시스템을 구현할 필요가 없으므로, 호스트 시스템의 설계 및 개발의 부담을 줄이고 제품 개발 주기를 단축할 수 있다. 현재와 같이 다양한 종류의 내장형 시스템이 빠르게 개발되는 상황에서 FSOC는 시장 경쟁력을 향상시킬 수 있다.

넷째, FSOC에서는 저장장치의 파일시스템 기능과 호스트의 응용이 병렬적으로 수행되기 때문에 호스트는 응용의 수행에 더 많은 계산 시간을 할당할 수 있다.

다섯째, FSOC에서는 파일시스템 연산을 수행할 때 필요한 중간 데이타가 저장장치 외부로 전송되지 않기 때문에 호스트와 저장장치 간에 데이타 전송량을 줄일 수 있다[5].

여섯째, FSOC에서는 호스트와 저장장치의 병렬처리와 데이타 전송량의 감소에 의해 전체 시스템의 전력 소모를 줄일 수 있다. 이러한 장점은 배터리를 사용하는 휴대용 정보기기에서는 특히 중요하다.

본 논문에서는 먼저 2장에서 관련연구에 대해 기술하고, 3장에서 플래시메모리 카드에서 FSOC를 설계하고 구현한 내용을 기술한다. 4장에서는 기존 저장장치와 FSOC의 성능모델을 제시하고, 제시한 모델을 기반으로 기존 저장장치와 FSOC의 성능 차이를 분석한다. 또한 구현된 FSOC를 이용한 실험을 통해 제시한 성능모델과 실제 응용의 성능이 부합함을 보인다. 마지막으로 5장에서 결론을 맺는다.

2. 관련 연구

본 장에서는 먼저 저장장치에 프로세서와 같은 자원을 추가함으로써 저장장치의 성능 및 기능 향상을 꾀하는 관련 연구를 설명한다. 이어서 본 연구에서 FSOC를 구현한 플래시메모리 카드와 관련된 연구에 대해 설명하고, 마지막으로 병렬 수행으로 전력 소모를 줄이는 연구에 대해 설명한다.

저장장치에 자원을 추가하여 저장장치의 성능 및 기능을 향상시키는 연구 중 대표적인 것에는 능동적 디스크(Active Disk)[5]에 대한 연구와 IDISK(Intelligent Disk)[6]에 대한 연구, 그리고 OSD(Object-based Storage Device) 프로젝트[7,8]가 있다. 능동적 디스크는 호스트로부터 코드를 전달받아 수행시킬 수 있는 저장장치이다. 응용을 분할하여 데이타 접근이 많은 작업을 능동적 디스크에서 수행함으로써, 데이타 전송량을 감소시키고 작업을 병렬적으로 수행할 수 있다. IDISK는 응용을 수행할 수 있는 다수의 저장장치를 고속의 통신 채널로 연결한 구조를 가진다. 이를 통하여 응용 수행의 병렬성을 높임으로써, 의사 결정 시스템과 같이 접근해야 하는 데이타의 양이 많은 응용을 효율적으로 수행할 수 있다. FSOC와 가장 밀접하게 연관된 OSD는 파일시스템의 기능 중 일부를 수행하는 저장장치이며, 파일시스템 내의 데이타 배치를 저장장치에서 책임짐으로써 상호운영성을 향상시킨다. 그러나 OSD는 파일이 아닌 객체의 형태로 데이타를 관리하며, 파일의 명명 기능과 디렉터리의 계층적 구성 기능은 호스트에서 구현해야 한다는 점에서 FSOC와 차이가 있다.

본 연구에서 FSOC의 구현 매체로 사용된 플래시메모리는 가볍고 내구성이 높으며 전력소모가 적어 이동식 저장장치의 저장매체로 사용하기에 적합하다[9,10]. 그러나 플래시메모리는 데이타를 재쓰기(overwrite) 하기 위해서는 해당 영역을 지우는 작업(erase)을 수행해야 한다는 제약을 가지기 때문에 디스크에서 사용되는 파일시스템을 플래시메모리에 그대로 사용할 수 없다.

플래시메모리를 위한 파일시스템으로 JFFS[11]와 YAFFS[12]가 개발되었다. JFFS와 YAFFS는 리눅스 운영체제에 구현되었으며 플래시메모리의 물리적인 구조를 파일시스템이 직접 관리한다. 이러한 파일시스템들은 로그 구조 파일시스템[13]을 플래시메모리에 적합하게 변형한 것인데, 로그 구조 파일시스템에서는 저장되는 모든 데이타가 로그의 끝에 순차적으로 추가되므로 플래시메모리의 제약에 따른 성능 저하를 줄일 수 있다. JFFS와 YAFFS는 호스트에서 수행되는 것을 목표로 개발되었으며, 자원 사용량이 많아 저장장치 내에 내장하기에는 부적합하다.

콤팩트플래시[14]나 멀티미디어카드[15]와 같은 메모리 카드는 FTL(Flash Translation Layer)[16-19]을 내장하여 플래시메모리의 제약을 해결한다. FTL은 섹터 재사상 기능을 이용하여 플래시메모리를 읽고, 쓰고, 재쓰기가 가능한 연속된 섹터의 집합으로 보이게 하는 인터페이스를 제공한다.

FSOC의 사용은 호스트 프로세서의 응용 수행과 저장장치 프로세서의 파일시스템 기능 수행이 병렬 수행될 수 있기 때문에 에너지 소모를 줄일 수 있는 기회를 제공한다. Chandrakasan, Sheng, 그리고 Brodersen은 [20]에서 동일한 시간에 작업을 처리하는 경우 고전압에서 고속으로 동작하는 프로세서 하나를 사용하는 것보다 저전압에서 저속으로 동작하는 프로세서들을 여러 개 사용하여 작업을 병렬적으로 수행함으로써 에너지 소모를 줄일 수 있음을 보였다.

3. FSOC의 설계 및 구현

3.1 FSOC의 구조

본 절에서는 FSOC와 호스트의 구조를 설명한다. 그림 1은 기존 저장장치를 이용하여 파일시스템을 구현하는 경우와 FSOC의 구조를 비교하여 도시한 것이다. 기존 저장장치를 사용하는 경우 응용의 파일 접근 요청은 호스트에 있는 운영체제의 파일시스템에 의해 블록 요청으로 변환되어 장치관리자에게 전달된다. 장치관리자는 블록 요청을 섹터 요청으로 변환하여 저장장치에게 전달하게 된다.

FSOC는 저장장치에 파일시스템을 내장하고 있으며 호스트는 파일시스템 대신 FSOC와의 인터페이스 역할을 수행하는 간단한 스텁(stub)을 필요로 한다. 응용의 파일 접근 요청은 스텁을 통해 FSOC에 전달된다. 스텁은 파일 연산에 필요한 인자들을 정리하여 FSOC 요청으로 변환하는 작업과 이 요청을 FSOC에 전달하는 역할을 수행한다. 또한 저장장치가 수행한 파일 연산의 결과를 응용에게 전달하며, FSOC에서 오류가 발생한 경우 오류의 처리를 수행한다. 스텁은 독립적으로 구현될 수도 있으며, VFS(Virtual File System) 계층의 하위에 구현될 수도 있다. VFS 계층의 하위에 스텁을 구현하면 응용들은 동일한 인터페이스로 FSOC를 포함하여 다른 모든 파일시스템을 사용할 수 있다.

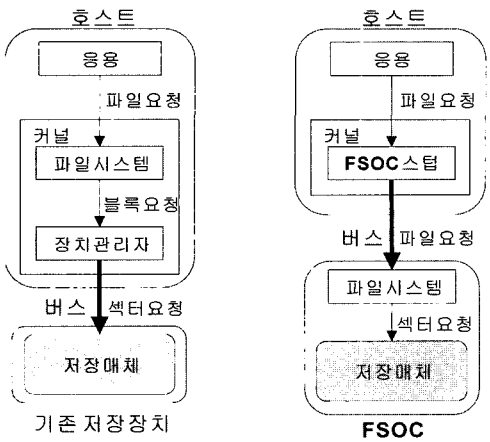


그림 1 기존 저장장치와 FSOC의 구조 비교

FSOC에서 사용되는 파일시스템은 스텁의 요청을 받아서 실질적인 파일 연산을 수행한다. 저장 매체에 새로운 데이타를 기록하거나 기존의 데이타를 읽어 들이며, 그 결과를 호스트에게 전달한다. 또한 파일 연산의 결과로 수정된 파일 관리 정보들을 갱신한다.

그림 2는 응용과 스텁, FSOC 사이의 인터페이스를 보인다. FSOC의 인터페이스는 원격 프로시저 호출[21]과 유사한 형태를 가지고 있다. FSOC는 응용이 사용하는 각 파일 연산에 대응되는 서비스 루틴을 가지고 있으며 스텁은 이에 대한 클라이언트 루틴을 가지고 있다. 예를 들어 파일의 읽기 연산을 수행하는 경우의 작업의 처리는 다음과 같이 이루어진다. 먼저 응용은 read() 시스템 호출을 통해 대상 파일의 식별자와 읽을 데이타의 양을 스텁에게 전달한다. 스텁은 read_cli() 루틴을 수행하여 파일의 식별자와 파일의 오프셋, 데이타의 양을 FSOC 요청으로 변환한 후 FSOC에 전달한다. FSOC의 read_svc() 루틴은 파일의 읽기 연산을 수행한 후 호스트의 스텁에게 결과를 전달하고 스텁은 응용의 메모리에 전달받은 데이타를 복사하는 작업을 수행한다.

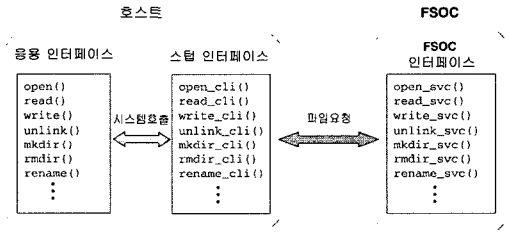


그림 2 FSOC의 인터페이스

3.2 FSOC의 구현

본 연구에서는 플래시메모리를 저장매체로 사용하여 FSOC를 구현하였다. 2장 관련연구에서 기술한 바와 같이, 플래시메모리는 가볍고 내구성이 높으며 전력소모가 적어 이동식 저장장치의 저장매체로 적합하다. 플래시메모리를 사용하는 콤팩트플래시나 멀티미디어카드 등의 메모리카드는 FTL을 내장하고 있으며, FTL 기능을 수행하기 위하여 프로세서와 메모리를 내장하고 있다. 그림 3은 삼성전자(주)에서 개발한 콤팩트플래시[22]의 구

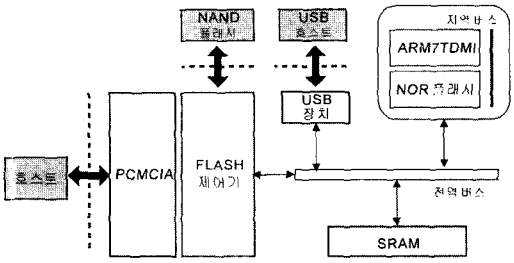


그림 3 콤팩트플래시의 하드웨어 구조

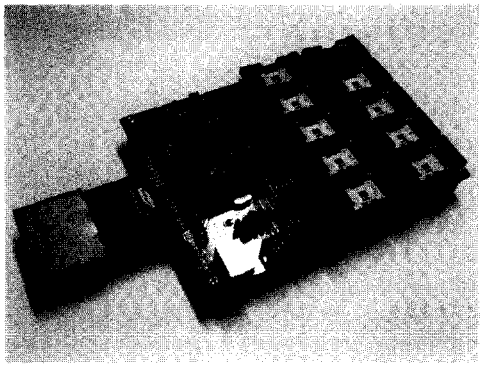


그림 4 FSOC 구현에 사용된 콤팩트 플래시 개발 보드

조를 보여준다. 내장 프로세서로는 24MHz로 동작하는 ARM7TDMI를 사용하고 있으며, 48KB 크기의 NOR형 플래시메모리에는 FTL 처리를 위한 코드가 저장되어 있다. 또한 FTL 수행 시 필요한 스택 영역, 데이타 영역, 버퍼 공간을 위하여 16KB 크기의 SRAM을 내장하고 있다. 비휘발성 저장매체로는 NAND형의 플래시메모리를 사용하며, 호스트와의 물리적인 인터페이스는 PCMCIA[23]와 USB[24]를 사용한다.

본 연구에서는 NOR형 플래시메모리에 FTL 뿐만 아니라 파일 시스템을 위한 코드도 내장함으로써 FSOC를 구현하였다. FSOC에 내장되는 파일시스템은 메모리 사용량이 작고 프로세서 성능이 낮아도 효율적으로 동작할 수 있어야 하며, 예고 없이 전력 공급이 중단되는 경우에도 신속한 무결성 회복이 가능해야 한다. 이러한 요구 조건들을 만족시키기 위하여 본 연구에서는 FAT (File Allocation Table)을 기반으로 저널링(Journaling) 기능을 추가한 파일시스템을 구현하였다. 구현된 파일시스템에서는 메모리에 유지해야 하는 정보를 최소화하여 10KB 이하의 메모리에서도 효율적으로 동작하게 하였고, 디렉터리마다 파일 이름에 대한 요약 정보를 유지하여 파일 검색에 소요되는 시간을 줄임으로써 저성능 프로세서에서도 효율적으로 동작하도록 하였다. 그리고 파일시스템 구조를 변경하는 연산을 수행하기 전에 연산의 내용을 로그에 기록함으로써 신속한 무결성 회복이 가능하도록 하였다.

또한 FTL의 효율을 높이기 위해 FSOC에 내장된 파일시스템은 삭제된 파일이 차지하던 섹터들에 대한 정보를 FTL에게 전달하며, FTL은 해당 섹터의 맵핑 정보를 삭제하고 해당 섹터가 차지하던 영역을 다른 섹터들의 재사상을 위한 공간으로 사용하도록 하였다. 이러한 최적화는 파일시스템을 저장장치에 내장함으로 인해 파일시스템과 FTL 사이의 인터페이스가 자유롭기 때문에 가능한 것이다.

이렇게 구현된 FSOC와 성능을 비교할 대상으로서, 본 연구에서는 FSOC에 내장된 FTL과 동일한 FTL을 내장하고 있는 콤팩트플래시도 구현하였다. 그리고 FSOC와 콤팩트플래시를 운용할 호스트로는 ARM920T를 장착한 내장형 시스템 개발보드를 사용하였다. 이 호스트의 운영체제는 리눅스 커널 2.4.18을 사용하였으며, 호스트가 FSOC를 운용할 수 있도록 하기 위해 FSOC에 대한 인터페이스 역할을 수행하는 스텁을 구현하여 커널에 추가하였다. 또한 FSOC에 내장된 파일시스템을 호스트의 커널에 이식하여 호스트가 콤팩트플래시에 접근할 때에는 이 파일시스템을 사용하도록 하였다. 그림 5는 구현된 호스트와 콤팩트플래시, 그리고 FSOC의 구조를 보이고 있다.

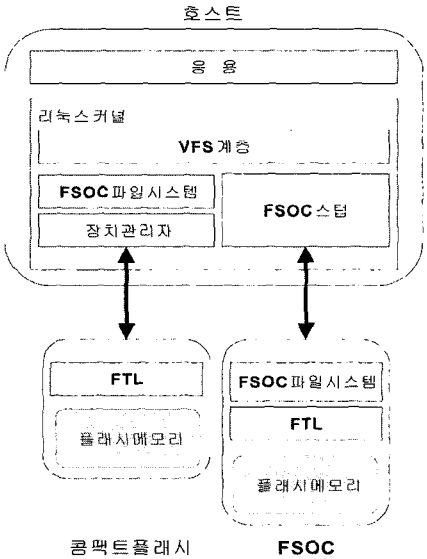


그림 5 구현된 호스트와 콤팩트플래시, FSOC

4. 성능 평가

본 장에서는 기존 저장장치와 FSOC의 성능모델을 제시하며, 여러 가지 실험을 통해 두 저장장치의 성능을 비교한다. 실험에서는 본 연구에서 구현한 콤팩트플래시

를 기존 저장장치로 사용하였으며 동일한 하드웨어 상에서 구현된 FSOC와 성능을 비교하였다. 표 1은 구현된 콤팩트플래시와 FSOC의 하드웨어 및 소프트웨어 사양을 보여준다. 3.2절에서 서술한 바와 같이 콤팩트플래시에 내장된 FTL과 FSOC에 내장된 FTL은 동일하며, 콤팩트플래시에 접근하기 위해 호스트에서 사용하는 파일시스템과 FSOC에 내장된 파일시스템 역시 동일하다.

4.1 성능모델

본 절에서는 기존 저장장치와 FSOC의 성능모델을 제시한다. 성능모델에서 사용하는 성능의 평가 기준은 저장장치의 접근시간을 포함하는 응용의 수행시간이다. 제시하는 모델은 응용의 코드를 수행하는 작업과 입출력 작업이 병렬적으로 처리된다고 가정하며, 이 때 입출력 작업은 저장장치 접근 시 저장장치에서 수행되는 모든 작업과 데이타 전송 작업으로 정의한다. 입출력 작업이 응용 코드의 수행을 봉쇄하지 않는 경우 이 가정을 만족할 수 있으며, 비동기적 쓰기(asynchronous write) 기법, 선반입(prefetching) 기법 등을 사용하면 응용 코드 수행의 봉쇄를 방지할 수 있다.

기존 저장장치를 사용하는 경우, 응용의 수행은 응용 코드의 수행, 파일시스템 코드의 수행, 장치관리자 코드의 실행, 그리고 입출력 작업으로 구성되며, 입출력 작업은 저장매체 접근과 데이타 전송으로 구성된다. 그림 6은 호스트가 기존 저장장치에 접근할 때 호스트의 프로세서와 버스, 그리고 저장장치에서 각 작업이 처리되는 과정을 보이고 있다. 기존 저장장치를 사용할 때 전체 응용 수행시간을 T_{conv}로 표기하면 전체 응용 수행시간은 식 (1)과 같이 나타낼 수 있다(식에 사용된 기호들의 정의는 표 1을 참조).

FSOC를 사용하는 경우에 응용의 수행은 응용 코드의

표 1 실험에 사용된 기존 저장장치와 FSOC의 하드웨어 및 소프트웨어 사양

	기존 저장장치 (콤팩트플래시)	FSOC
프로세서	ARM7TDMI (24MHz)	ARM7TDMI (24MHz)
SRAM	16KB	16KB
NOR형 플래시 메모리	48KB	48KB
NAND형 플래시 메모리	64MB	64MB
내장 소프트웨어	FTL	파일시스템, FTL
물리적 인터페이스	PCMCIA	PCMCIA
호스트에서 필요한 소프트웨어	파일시스템, 장치관리자	FSOC 스텁

표 2 기존 저장장치와 FSOC의 성능 모델에 사용된 기호의 정의

사용된 모델	기호	의미
공통	T_{comp}	응용 코드의 수행시간
	T_{media}	저장매체 접근시간
기존 저장장치	T_{FS_host}	호스트의 파일시스템 코드 수행시간
	T_{driver}	장치관리자 코드의 수행시간
	T_{trans_conv}	기존 저장장치와 호스트 사이의 데이타 전송시간
FSOC	T_{stub}	스텁 코드의 수행시간
	T_{FS_FSOC}	FSOC의 파일시스템 코드의 수행시간
	T_{trans_FSOC}	FSOC와 호스트 사이의 데이타 전송 시간

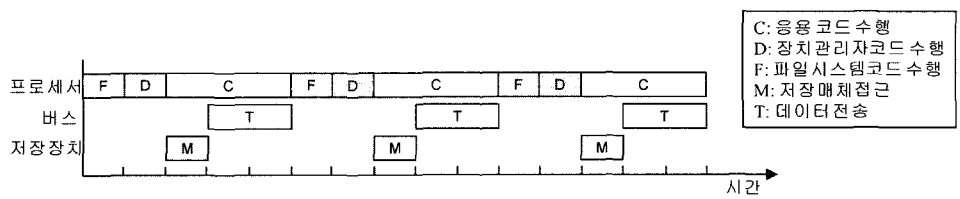


그림 6 기존 저장장치 사용 시 호스트 프로세서와 버스, 저장장치의 동작

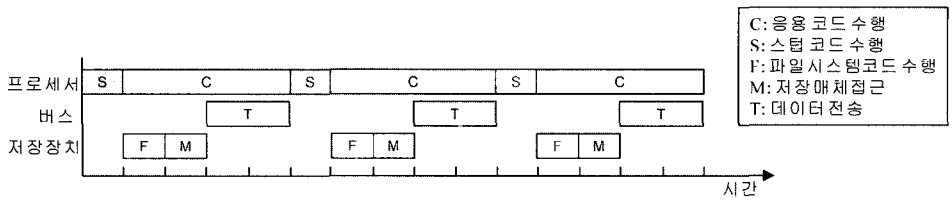


그림 7 FSOC 사용 시 호스트 프로세서와 버스, 저장장치의 동작

수행, 스텁 코드의 수행, 그리고 입출력 작업으로 구성되며, 이 중 입출력 작업은 FSOC에 내장된 파일시스템 코드의 수행과 저장매체 접근, 데이타 전송으로 구성된다. 기존 저장장치와 비교하면 호스트에서 장치관리자 대신 스텁 코드를 수행하며, 파일시스템을 호스트가 아닌 FSOC에서 수행한다는 점이 다르다. 그림 7은 호스트가 FSOC에 접근할 때 각각의 작업이 호스트의 프로세서와 버스, 저장장치에서 수행되는 과정을 보이고 있다. FSOC 사용 시 전체 응용 수행시간을 T_{FSOC}로 표기하면 전체 응용 수행시간은 식 (2)와 같이 나타낼 수 있다.

$$T_{conv} = T_{comp} + T_{FS\_host} + T_{driver} + \delta(T_{media} + T_{trans\_conv} - T_{comp}) \quad (1)$$

$$T_{FSOC} = T_{comp} + T_{stub} + \delta(T_{FS\_FSOC} + T_{media} + T_{trans\_FSOC} - T_{comp}) \quad (2)$$

$$\text{단, } \delta(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

4.2 성능 요소별 성능 비교

4.2.1 응용 코드 수행시간과 입출력 시간의 차이

본 절에서는 앞에서 설명한 성능모델을 기반으로 응용 코드 수행시간과 입출력 작업에 소요되는 시간의 차이에 따른 기존 저장장치와 FSOC의 성능을 비교한다. 이 때 스텁과 장치관리자의 역할이 유사하므로 스텁 코드 수행시간과 장치관리자 수행시간은 같다고 가정한다. 또한 분석을 간단히 하기 위해 기존 저장장치를 사용할 때와 FSOC를 사용할 때의 데이타 전송시간은 동일하다고 가정하며, 그 값을 T_{trans}로 표기한다. 이 경우 기존 저장장치와 FSOC의 성능은 응용 코드 수행시간과 입출력 시간에 따라 다음과 같은 세 가지 경우로 나누어 비교할 수 있다.

1) $T_{media} + T_{trans} > T_{comp}$: 응용 코드 수행시간이 기존 저장장치의 입출력 시간보다 작은 경우이다. 이 경우, 기존 저장장치와 FSOC 모두 전체 응용 수행시간이 입출력 시간에 의해 제한되어 일정하며, 기존 저장장치와 FSOC에서 전체 응용 수행시간의 차이는 FSOC의 파일시스템 코드 수행시간과 호스트의 파일시스템 코드 수행시간의 차이와 같다.

2) $T_{media} + T_{trans} \leq T_{comp} < T_{media} + T_{trans} + T_{FS\_FSOC}$: 응용 코드 수행시간이 기존 저장장치의 입출력 시간보다 크고 FSOC의 입출력 시간보다 작은 경우이다. 이 경우, FSOC의 전체 응용 수행시간은 입출력 시간에 의해 제한되지만, 기존 저장장치의 전체 응용 수행시간은 응용 코드 수행시간에 의해 제한된다. 따라서 FSOC의 전체 응용 수행시간은 일정한 반면, 기존 저장장치의 전체 응용 수행시간은 응용 코드 수행시간에 비례하여 증가한다.

이 경우는 기존 저장장치의 전체 응용 수행시간과 FSOC의 전체 응용 수행시간의 대소 관계에 따라 다시 다음과 같은 두 가지 경우로 나눌 수 있다. 단, 다음의 두 가지 경우는 호스트 프로세서의 성능이 FSOC에 내장된 프로세서의 성능보다 높아서 호스트의 파일시스템 코드 수행시간이 FSOC의 파일시스템 코드 수행시간보다 작을 때에만 적용된다.

(a) $T_{media} + T_{trans} \leq T_{comp} < T_{media} + T_{trans} + T_{FS\_FSOC}$

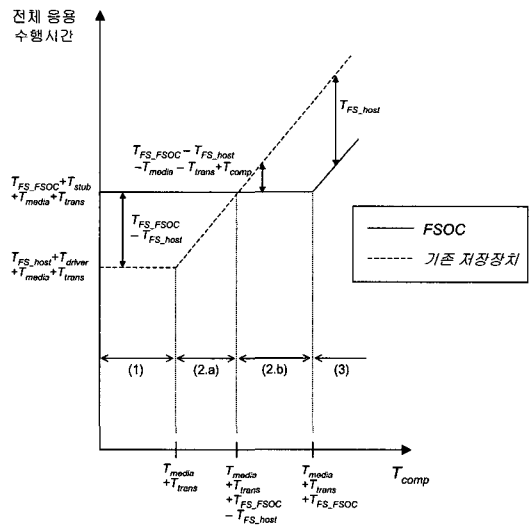
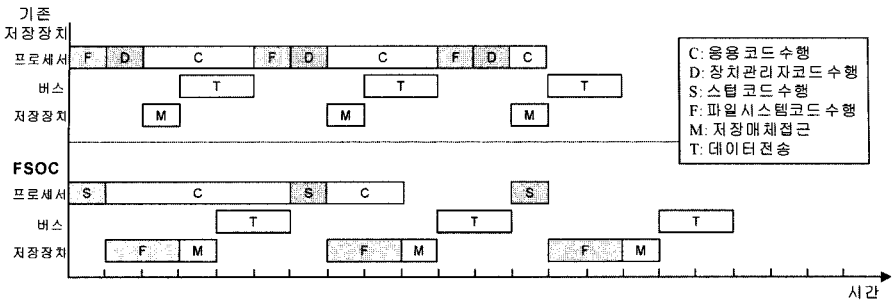
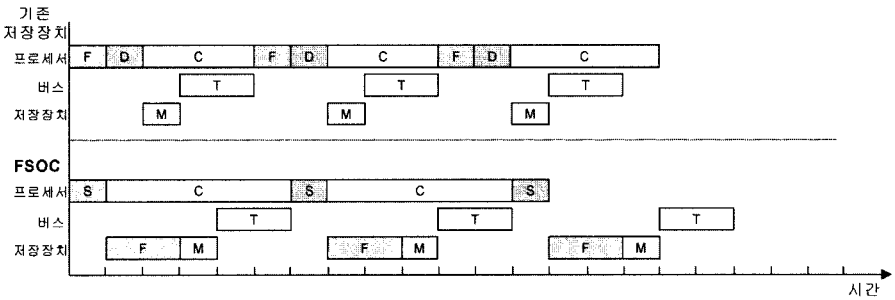


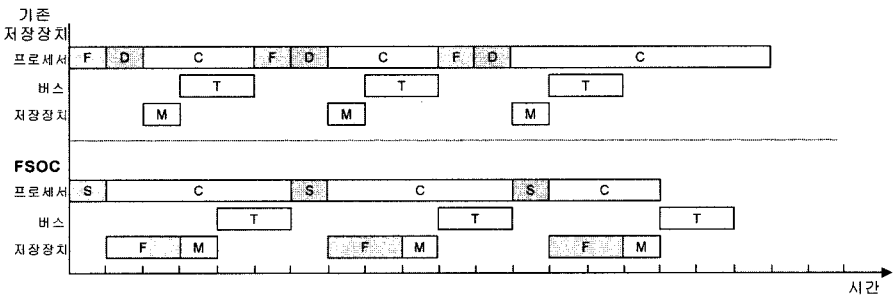
그림 8 응용 코드 수행시간과 입출력 시간의 차이에 따른 전체 응용 수행시간의 변화



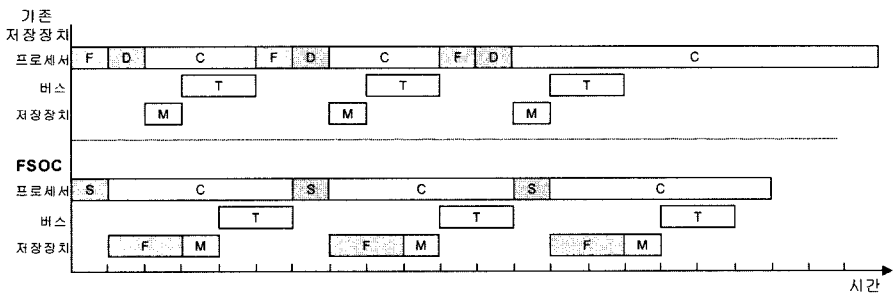
(1) $T_{media} + T_{trans} > T_{comp}$



(2.a) $T_{media} + T_{trans} \leq T_{comp} < T_{media} + T_{trans} + T_{FS\_FSOC} - T_{FS\_host}$



(2.b) $T_{media} + T_{trans} + T_{FS\_FSOC} - T_{FS\_host} \leq T_{comp} < T_{media} + T_{trans} + T_{FS\_FSOC}$



(3) $T_{comp} \geq T_{media} + T_{trans} + T_{FS\_FSOC} \geq T_{media} + T_{trans}$

그림 9 기존 저장장치와 FSOC의 수행 예

$-T_{FS\_host}$: FSOC의 전체 응용 수행시간이 기존 저장장치의 전체 응용 수행시간보다 큰 값을 가지는 경우

(b) $T_{media}+T_{trans}+T_{FS\_FSOC}-T_{FS\_host} \le T_{comp} < T_{media}+T_{trans}+T_{FS\_FSOC}$: 기존 저장장치의 전체 응용 수행시간이 FSOC의 전체 응용 수행시간보다 큰 값을 가지는 경우

3) $T_{comp} \ge T_{media}+T_{trans}+T_{FS\_FSOC} \ge T_{media}+T_{trans}$: 응용 코드 수행시간이 FSOC의 입출력 시간보다도 큰 경우이다. 이 경우 기존 저장장치와 FSOC의 전체 응용 수행시간은 모두 응용 코드 수행시간에 의해 제한되므로, 응용 코드 수행시간에 비례하여 증가한다. 이 때 기존 저장장치와 FSOC의 전체 응용 수행시간의 차이는 호스트의 파일시스템 코드 수행시간과 같다.

그림 8은 응용 코드 수행시간과 입출력 시간의 차이에 따른 기존 저장장치와 FSOC의 전체 응용 수행시간을 앞에서 설명한 (1), (2.a), (2.b), (3)의 네 구간으로 나누어 그래프로 나타내고 있고, 그림 9는 이 네 구간에 해당하는 각각의 경우에 있어서 기존 저장장치와 FSOC의 동작을 프로세서, 버스, 그리고 저장장치로 나누어 보여주고 있다.

본 연구에서는 성능모델을 검증하기 위하여 합성부하(synthetic workload)를 사용한 실험을 수행하였다. 합성부하는 먼저 파일에서 8KB의 데이타를 읽어 온 후, 카운터 변수의 값을 지정된 값으로 초기화시킨 다음, 카운터 변수의 값이 0이 될 때까지 변수의 값을 1씩 감소시키는 작업을 수행한다. 이러한 작업은 파일에서 읽은 전체 데이타가 1MB가 될 때까지 계속해서 수행되며, 이 때 카운터 변수의 초기값은 응용 코드 수행시간(T_{comp})을 조절하는 인자로 사용된다. 그림 10은 실험의 결과로서 카운터 변수의 초기값에 따른 전체 응용 수행시간을 보이고 있는데 그림에서 보면 성능모델의 그래프인 그림 8과 부합함을 확인할 수 있다.

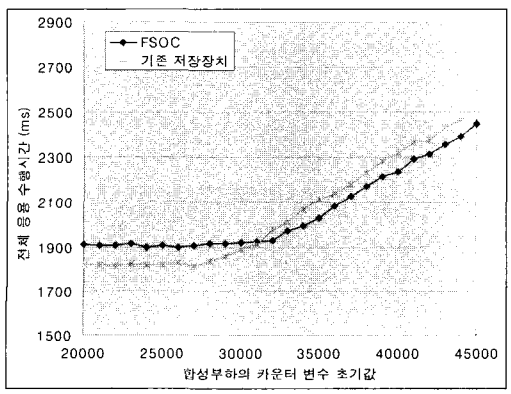


그림 10 합성부하 수행시간

4.2.2 호스트의 계산능력과 저장장치의 계산능력의 차이

FSOC와 같은 이동식 저장장치는 다양한 호스트에서 사용될 수 있으며, 호스트의 성능이 변화함에 따라 호스트에서 수행되는 응용 및 파일시스템 코드 수행시간도 변화한다. 본 절에서는 호스트에서 사용하는 프로세서와 저장장치에서 사용하는 프로세서의 상대적인 계산능력의 차이가 전체 응용 수행시간에 미치는 영향을 분석한다.

계산능력의 차이에 따른 성능을 비교하기 위해 호스트의 성능을 변화시키며 합성부하를 실행하는 실험을 수행하였다. 그림 11은 저장장치의 동작속도는 24MHz로 고정하고 호스트의 동작속도를 45MHz, 56MHz, 67MHz로 변화시킬 때 합성부하의 전체 응용 수행시간의 변화를 보여준다. 그림의 결과에서 보면 동일한 양의 응용 코드를 수행하더라도 호스트의 계산능력에 따라 기존 저장장치와 FSOC의 상대적인 성능 관계가 변화함을 알 수 있다. 예를 들어 카운터 변수의 초기값이 22000인 경우에는(그림 11의 (a) 지점) 호스트의 모든 동작속도에서 기존 저장장치와 FSOC의 전체 응용 수행시간이 모두 입출력시간에 의해 제한되어 기존 저장장치가 높은 성능을 보였다. 그러나 카운터 변수의 초기값이 29000인 경우에는(그림 11의 (b) 지점) 호스트의 동작속도에 따라 기존 저장장치의 성능과 FSOC의 성능 관계가 변화함을 관찰할 수 있다. 예를 들어 호스트가 45MHz로 동작할 때에는 FSOC가 높은 성능을 나타내었지만 호스트가 56MHz로 동작할 때와 67MHz로 동작할 때에는 기존 저장장치가 더 높은 성능을 보였다. 반면, 카운터 변수의 초기값이 39000인 경우에는(그림 11의 (c) 지점) 호스트의 모든 동작속도에서 기존 저장장치와 FSOC의 전체 응용 수행시간이 모두 응용 코드 수행시간에 의해 제한되어 FSOC가 높은 성능을 나타내었다.

합성부하를 이용하여 얻은 결과가 실제 응용에서도 그대로 적용됨을 보이기 위하여 응용 코드 수행시간과 입출력 시간의 비율이 서로 다른 cat, gzip, mpeg 세

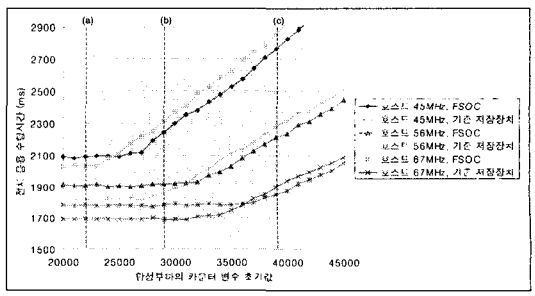
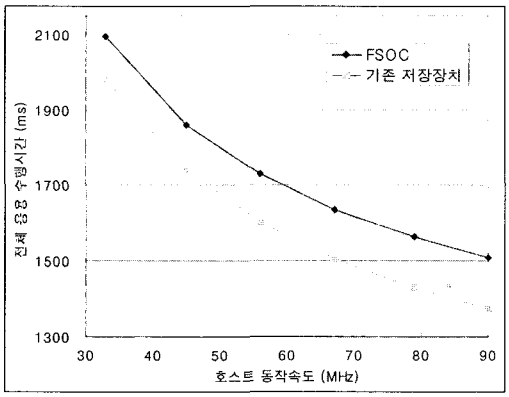


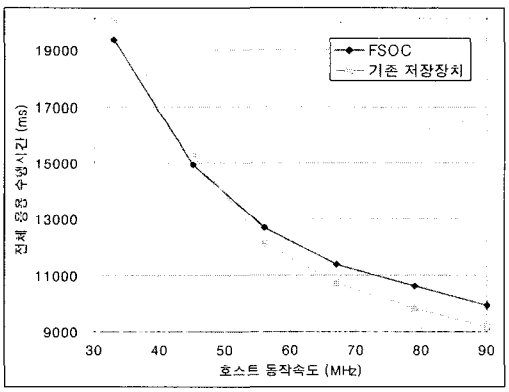
그림 11 계산능력의 차이와 합성부하 수행시간

표 3 실험에 사용된 응용

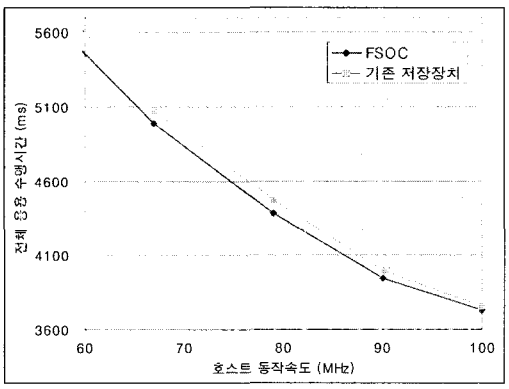
응용	설명	대상 파일 크기
cat	파일 출력 유틸리티	1.0MB
gzip	파일 압축 유틸리티	4.8MB
mpeg	MPEG 비디오 디코더	1.7MB



(a) cat



(b) gzip



(c) mpeg

그림 12 호스트 계산능력과 응용의 수행 결과

개의 응용을 이용하여 동일한 실험을 수행하였다(세 개의 응용에 관한 자세한 내용은 표 2 참고). cat은 응용 코드 수행시간에 비해 입출력 시간이 큰 응용이고, gzip은 응용 코드 수행시간과 입출력 시간의 차이가 크지 않은 응용이며, mpeg은 입출력 시간에 비해 응용 코드 수행시간이 큰 응용이다.

그림 12는 호스트의 동작속도의 변화에 따른 각 응용의 전체 응용 수행시간을 나타내고 있다. cat의 결과에서는 기존 저장장치가 높은 성능을 보였고, gzip의 결과에서는 기존 저장장치와 FSOC의 성능이 교차되는 결과를 나타내었으며, mpeg의 결과에서는 FSOC가 높은 성능을 보였다. 이 결과는 각각 그림 11의 (a), (b), (c)의 경우와 일치하고 있음을 확인할 수 있다.

4.2.3 다중 프로그래밍 영향

기존 저장장치와 FSOC를 사용하는 경우 다중 프로그래밍 영향에 따른 성능을 비교하기 위해 앞의 실험에서 사용한 cat과 함께 원주율을 계산하는 계산 중심의 응용인 pi를 동시에 병렬적으로 실행시킨 후, 두 응용이 모두 종료될 때까지의 시간을 측정하는 실험을 수행하였다. 그림 13은 cat을 단독으로 실행시켰을 때의 결과와 pi을 단독으로 실행시켰을 때의 결과, 그리고 두 응용을 동시에 실행시켰을 때의 결과를 나타내고 있는데, 이 중 cat을 단독으로 실행시켰을 때의 결과는 4.2.2 절에서 수행한 실험의 결과(그림 12 (a))를 사용하였다.

실험의 결과를 보면 pi를 단독으로 실행시켰을 때에는 저장장치를 사용하지 않기 때문에 기존 저장장치와 FSOC가 동일한 전체 응용 수행시간을 보였으며, cat을 단독으로 실행시켰을 때에는 호스트와 FSOC의 계산능력의 차이로 인해 FSOC의 전체 응용 수행시간이 기존 저장장치에 비해 5.7~9.8% 느린 결과를 보였다. 그런데 cat과 pi를 동시에 실행시켰을 때의 결과를 보면 FSOC의 전체 응용 수행시간이 기존 저장장치에 비해 오히려

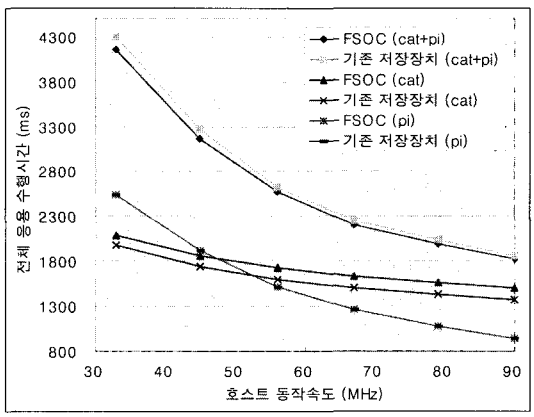


그림 13 여러 응용의 수행 결과 (cat과 pi)

1.8~3.4% 빨라짐을 확인할 수 있었다. 이 결과는 FSOC를 사용한 경우 cat의 실행 중에 호스트에서 절약된 파일시스템 코드 수행시간을 pi의 실행에 할당하여 전체 응용 수행시간을 줄이는데 사용되었음을 보여준다.

4.2.4 호스트와 저장장치간의 데이타 전송량

FSOC를 사용하면 파일 시스템 연산을 수행할 때 필요한 중간 데이타가 저장장치 외부로 전송되지 않으므로 호스트와 저장장치 사이의 데이타 전송량이 감소한다. 예를 들어 FAT 파일시스템에서 새로운 파일을 생성하기 위해서는 파일 할당 테이블과 파일이 생성되는 디렉터리의 내용을 수정해야 한다. 파일시스템이 호스트에서 수행되는 경우에는 파일 할당 테이블과 디렉터리 내용을 수정하기 위해서 변경되는 데이타가 속한 블록을 저장장치에서 호스트로 전송한 후 호스트에서 블록의 내용을 변경하고 다시 저장장치로 전송하여 쓰기 작업을 수행하여야 한다. 그러나, FSOC에서는 이러한 작업을 모두 저장장치 내부에서 수행하므로 호스트로 데이타를 전송할 필요가 없고, 단지 생성할 파일의 이름만을 호스트에서 FSOC로 전달하면 된다.

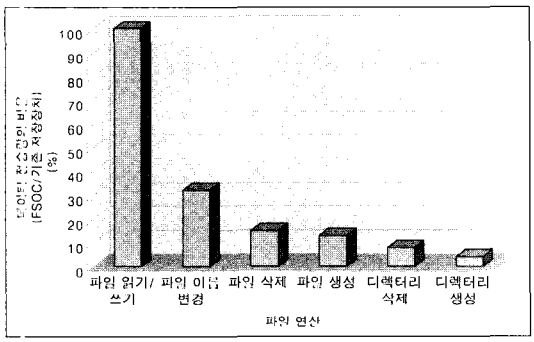


그림 14 파일 연산에 따른 기존 저장장치와 FSOC의 데이타 전송량 비율

파일 연산에 필요한 중간 데이타의 양은 파일 연산의 종류에 따라 달라진다. 그림 14는 기존 저장장치와 FSOC의 데이타 전송량의 비율을 파일 연산의 종류별로 보여준다. 파일 연산 중 파일의 읽기 및 쓰기 연산은 파일 데이타 전송량에 비해 중간 데이타의 전송량이 매우 적기 때문에 FSOC와 기존 저장장치가 거의 동일한 데이타 전송량을 보인다. 반면 디렉터리 생성 및 삭제, 파일의 생성 및 삭제 등과 같이 중간 데이타만을 사용하는 파일 연산을 수행할 때에는 FSOC의 데이타 전송량이 기존 저장장치에 비해 68~96% 감소한다.

표 4 파일 연산의 종류와 전체 응용 수행시간(단위: ms)

	FSOC	기존 저장장치
파일 읽기	1730	1603
파일 생성	8986	28926
파일 복사	4920	4961

FSOC와 기존 저장장치의 데이타 전송량의 차이로 인한 성능 차이를 실제 응용을 통하여 비교하기 위하여 세 가지 실험을 수행하였다. 첫 번째 실험은 500개의 파일을 생성하는 작업으로, 파일 연산을 위해 필요한 중간 데이타 비율이 높은 경우에 해당된다. 반면 두 번째 실험은 1MB 크기의 파일을 순차적으로 읽는 작업을 수행하는데, 이 작업은 중간 데이타를 거의 필요로 하지 않는다. 마지막으로 세 번째 실험은 파일의 생성 및 읽기, 쓰기가 혼합되어 있는 Andrew benchmark[25]를 사용하여 수행하였다. Andrew benchmark 내에서는 23B~37KB 사이의 크기를 가지고 전체 크기가 362KB인 70개의 파일을 복사하는데 이는 첫 번째 실험과 두 번째 실험의 중간 정도의 중간 데이타 비율을 가진다.

실험 결과를 보여주는 표 4를 보면, 기존 저장장치와 FSOC가 동일한 양의 데이타를 전송하는 파일 읽기 작업의 경우 호스트와 저장장치의 계산능력의 차이로 인해 FSOC가 약 8%의 성능 저하를 보였다. 반면 FSOC가 기존 저장장치에 비해 매우 적은 양의 데이타를 전송하는 파일 생성 작업의 경우 FSOC가 약 69%의 성능 향상을 보임을 확인하였다. 그리고 파일 복사 작업의 경우에는 기존 저장장치와 FSOC가 거의 동일한 성능을 보였는데 이 결과는 FSOC에서 저장장치의 계산 능력이 낮아서 발생한 성능 저하 효과와 데이타 전송량 감소로 인한 성능 향상 효과가 상쇄되고 있음을 반영하고 있다.

4.2.5 실험 결과에 대한 논의

이상의 실험 결과로부터 확인할 수 있는 사실들을 정리하면 다음과 같다.

첫째, 응용의 코드 수행시간이 입출력 시간보다 큰 경우, FSOC를 사용하면 호스트의 응용과 FSOC의 파일시스템 코드가 병렬적으로 수행되어 기존 저장장치를 사용하는 경우에 비해 전체 응용 수행시간을 단축시킨다.

둘째, 호스트와 저장장치의 계산 능력의 차이가 큰 경우, FSOC의 파일시스템 코드 수행 시간이 호스트의 파일시스템 코드 수행 시간에 비해 증가하므로 FSOC의 입출력 시간이 길어진다.

셋째, 호스트에서 실행되는 응용의 수가 많은 경우, 응용의 코드 수행과 FSOC의 파일시스템 수행이 병렬적으로 이루어질 기회가 많아지며, 따라서 FSOC를 사용할 때 전체 응용 수행시간을 단축시킬 가능성이 높아진다.

넷째, 파일 연산에 필요한 임시 데이타의 양이 많은 경우, FSOC는 호스트와 저장장치 사이의 데이타 전송량을 감소시키므로 FSOC의 입출력 시간이 단축된다.

따라서 호스트의 계산 능력이 높지 않은 휴대용 저장

장치나 내장형 시스템에서 FSOC를 사용하는 것이 적합하며, 특히 MP3 플레이어와 같이 많은 계산을 필요로 하는 멀티미디어 응용을 수행하는 장치에서 FSOC가 유리하다. 또한 동시에 여러 응용을 실행하며, 크기가 작은 여러 개의 파일에 접근하는 PDA와 같은 시스템에서도 FSOC가 효율적으로 사용될 수 있다.

5. 결 론

본 논문에서는 파일시스템을 내장한 새로운 형태의 저장장치인 FSOC(File System On a Chip)를 제안하고 플래시메모리를 저장매체로 하는 구현의 예를 보였다. 제안한 FSOC는 기존 저장장치에 비해 높은 상호운영성을 지원하며 저장장치에 특화된 파일시스템을 사용함으로써 저장장치의 효율을 높일 수 있다는 장점을 가진다. 또한 FSOC를 사용하면 호스트에서 파일시스템을 구현할 필요가 없으므로 호스트의 설계 및 개발의 부담을 줄일 수 있으며, 파일시스템 기능과 응용을 병렬적으로 수행하여 전체 응용 수행시간을 단축시킬 수 있다는 장점도 가진다. 그리고 FSOC에서는 호스트와 저장장치 간의 데이타 전송량을 줄일 수 있으며, 전체 시스템의 전력소모를 감소시킬 수 있다는 것도 FSOC의 장점이다.

본 연구에서는 또한 기존 저장장치와 FSOC의 상대적인 성능에 영향을 미치는 요소들에 기반한 성능모델을 제시하였으며 다양한 합성부하와 실제 응용을 사용한 실험을 통해 그 타당성을 검증하였다. 실험 결과 FSOC는 다수의 응용이 실행되는 다중 프로그래밍 환경에서 우수한 성능을 보였고, 단일 응용의 경우에도 응용 코드의 수행시간이 입출력 시간보다 상대적으로 큰 경우 기존 저장장치보다 우수한 성능을 보였으며, 특히 파일 연산에 필요한 중간 데이타의 비율이 높은 경우 더욱 우수한 성능을 보였다. 결론으로 FSOC는 그 정성적인 장점과 성능적인 특성 때문에 배터리로 동작되면서 다양한 멀티미디어 응용이 동시에 수행되는 휴대용 정보기기에 앞으로 널리 사용될 것으로 기대된다.

참 고 문 헌

[1] G. R. Ganger, "Blurring the Line Between OSes and Storage Devices," Tech. rep., CMU-CS-01166, Carnegie Mellon University, 2001.

[2] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger, "Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics," in Proc. the First USENIX Conference on File and Storage Technologies, pp. 259-274, 2002.

[3] R. Wang, T. E. Anderson, and D. A. Patterson, "Virtual Log-Based File Systems for a Programmable Disk," in Proc. the Third Symposium on Operating Systems Design and Implementation, pp. 29-43, 1999.

[4] C. R. Lumb, J. Schindler, and G. R. Ganger, "Freeblock Scheduling Outside of Disk Firmware," in Proc. the First USENIX Conference on File and Storage Technologies, pp. 275-288, 2002.

[5] A. Acharya, M. Uysal, and J. Saltz, "Active Disks: Programming Model, Algorithms and Evaluation," in Proc. the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 81-91, 1998.

[6] K. Keeton, D. A. Patterson and J. M. Hellerstein, "A Case for Intelligent Disks (IDISKs)," in Proc. the ACM SIGMOD International Conference on Management of Data, pp. 42-52, 1998.

[7] D. Anderson, "Object Based Storage Devices: A Command Set Proposal," Tech. rep., National Storage Industry Consortium, 1999.

[8] OSD workgroup, http://www.snia.org/tech_activities/workgroups/osd

[9] F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber, "Storage Alternatives for Mobile Computers," in Proc. the First Symposium on Operating Systems Design and Implementation, pp. 25-37, 1994.

[10] B. Marsh, F. Douglis, and P. Krishnan, "Flash Memory File Caching for Mobile Computers," in Proc. the 27th Annual Hawaii International Conference on Systems Sciences, pp. 451-461, 1994.

[11] D. Woodhouse, Red Hat, Inc., "JFFS: The Journalling Flash File System," http://sources.redhat.com/jffs2/jffs2-html/

[12] Aleph One Company, "Yet Another Flash Filing System," http://www.aleph1.co.uk/armlinux/projects/yaffs/

[13] M. Rosenblum, and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," ACM Transactions on Computer Systems, Vol.10, No. 1, pp. 26-51, 1992.

[14] CompactFlash Association, "Information about CompactFlash," http://www.compactflash.org/

[15] MultiMediaCard Association, http://www.mmca.org

[16] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," in Proc. the Winter USENIX Technical Conference, pp. 155-164, 1995.

[17] M. Wu, and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System," in Proc. the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 86-97, 1994.

[18] Intel Corporation, "Understanding the Flash Translation Layer (FTL) Specification," http://developer.intel.com

[19] MTD, "Memory Technology Device (MTD) subsystem for Linux," http://www.linux-mtd.infradead.org

[20] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power CMOS Digital Design," IEEE Journal of Solid State Circuits, Vol. 27, No. 4, pp. 473-484, 1992.
[21] A. Birrel and B. Nelson, "Implementing Remote Procedure Calls," ACM Transactions on Computer Systems, Vol. 2, No. 1, pp. 39-59, 1984.
[22] Samsung Electronics Corporation, "S3F49FAX for Compact Flash Specification Revision 1.0," http://www.samsung.com/Products/Semiconductor/SystemLSI/Microcontrollers/32_bit/MemoryCardController/S3F49FAX/S3F49FAX.htm
[23] Personal Computer Card Interface Association, "PCMCIA PC Card Standard Release 2.1," 1993.
[24] Compaq Computer Corporation, Hewlett-Packard Company, Intel Corporation, Lucent Technologies Inc, Microsoft Corporation, NEC Corporation, Koninklijke Philips Electronics, "Universal Serial Bus Specification Revision 2.0," 2000.
[25] J. K. Ousterhout, "Why Aren't Operating Systems Getting Faster As Fast as Hardware?" in Proc. the Summer USENIX Technical Conference, pp. 247-256, 1990.

안 성 준

1997년 서울대학교 컴퓨터공학과(공학사). 1999년 서울대학교 컴퓨터공학과(공학석사). 1999년~현재 서울대학교 전기컴퓨터공학부 박사과정 재학중. 관심분야는 파일시스템, 내장형 시스템, 결함허용 등

최 종 무

1993년 서울대학교 해양학과(이학사). 1995년 서울대학교 컴퓨터공학과(공학석사) 2001년 서울대학교 컴퓨터공학과(공학박사). 2001년~2003년 유비쿼스(주) 기술연구소 책임연구원. 2003년~현재 서울대학교 컴퓨터연구소 연구원. 2003년~현재 단국대학교 정보컴퓨터학부 컴퓨터과학전공 전임강사. 관심분야는 시스템 소프트웨어, 내장형 시스템, 파일 시스템, 유비쿼터스 컴퓨팅 등

이 동 희

1989년 서울대학교 컴퓨터공학과(공학사). 1991년 서울대학교 컴퓨터공학과(공학석사). 1998년 서울대학교 컴퓨터공학과(공학박사). 1999년~2001년 제주대학교 통신 컴퓨터 공학부 조교수. 2002년~현재 서울시립대학교 컴퓨터과학부 조교수. 관심분야는 운영체제, 플래시메모리 소프트웨어, 내장형 시스템, 고성능 저장장치 등

노 삼 혁

1986년 서울대학교 컴퓨터공학과(공학사). 1993년 메릴랜드대학교 컴퓨터과학과(박사). 1994년~현재 홍익대학교 정보컴퓨터공학부 부교수. 관심분야는 시스템 소프트웨어, 병렬처리 시스템, 실시간 시스템 등

민 상 렬

1983년 서울대학교 컴퓨터공학과 졸업 1985년 서울대학교 컴퓨터공학과 석사 1989년 University of Washington 전산학 박사. 1989년~90년 IBM T.J. Watson Research Center 객원 연구원. 1990년~92년 부산대학교 컴퓨터공학과 조교수. 1992년~현재 서울대학교 전기컴퓨터공학부 교수. 관심분야는 Computer Architecture, Parallel Processing, Computer Performance Evaluation

조 유 근

1971년 서울대학교 건축공학과 학사 1978년 미네소타대학교 컴퓨터과학 박사 1979년~현재 서울대학교 컴퓨터공학부 교수. 1984년~1985년 미네소타대학교 교환 교수. 1993년~1995년 서울대학교 중앙교육연구전산원장. 1999년~2001년 서울대학교 공과대학 부학장. 2001년~2002년 한국정보과학회 회장. 관심분야는 운영체제, 알고리즘 설계 및 분석, 암호학