플래시메모리 기반 컴퓨터시스템을 위한 고속 부팅 기법의 설계 및 성능평가

(Design and Evaluation of a Fast Boot-up Technique for Flash Memory based Computer Systems)

임 근 수 † 김 지 홍 †† 고 건 †††
(Keun Soo Yim) (Jihong Kim) (Kern Koh)

요 약 최근 플래시메모리에 기반한 내장형 컴퓨터시스템의 사용이 급증하고 있다. 이러한 내장형시스템은 일반적으로 빠른 부팅시간을 제공해야 한다. 하지만 부팅과정에서 플래시메모리용 파일시스템을 초기화하는 마운팅 시간이 플래시메모리의 크기에 따라 1-25초가량 소요된다. 현재 플래시메모리 단일 칩의 용량은 매년 2배씩 증가하는 추세에 있기 때문에 플래시메모리용 파일시스템을 마운트하는 시간이 내장형 시스템의 부팅시간을 지연시키는 중요한 요인이 될 것이다. 본 논문에서는 플래시메모리용 파일시스템의 메타데이타를 언마운팅 시점에 플래시메모리에 기록하고 이후에 마운팅하는 시점에 빠르게 읽어 들임으로써 마운팅 시간을 크게 단축하는 메타데이타 스냅샷 기법들을 NOR형과 NAND형 플래시메모리의 특성에 맞춰 설계한다. 파일시스템이 정상적으로 언마운트되지 않은 경우에는 이를 자동으로 인식하고 빠르게 에러를 복구할 수 있는 새로운 기법들을 사용한다. 성능평가를 통해서 제안하는 기법들은 대표적인 플래시메모리용 파일시스템인 JFFS2와 비교하여 마운팅 시간을 100배가량 단축시킴을 보인다.

키워드 : 고속 부팅, 플래시메모리, 고속 마운팅, 메타데이타 스냅샷

Abstract Flash memory based embedded computing systems are becoming increasingly prevalent. These systems typically have to provide an instant start-up time. However, we observe that mounting a file system for flash memory takes 1 to 25 seconds mainly depending on the flash capacity. Since the flash chip capacity is doubled in every year, this mounting time will soon become the most dominant reason of the delay of system start-up time. Therefore, in this paper, we present instant mounting techniques for flash file systems by storing the in-memory file system metadata to flash memory when unmounting the file system and reloading the stored metadata quickly when mounting the file system. These metadata snapshot techniques are specifically developed for NOR- and NAND-type flash memories, while at the same time, overcoming their physical constraints. The proposed techniques check the validity of the stored snapshot and use the proposed fast crash recovery techniques when the snapshot is invalid. Based on the experimental results, the proposed techniques can reduce the flash mounting time by about two orders of magnitude over the existing de facto standard flash file system, JFFS2.

Key words : Fast booting, flash memory, flash mounting, and metadata snapshot

1. 서 론

내장형 컴퓨터시스템은 빠른 부팅시간을 제공해야 한다[1]. 일반적으로 이러한 내장형시스템에서는 플래시메모리를 보조 기억장치로 활용한다. 이는 플래시메모리의 작은 크기와 물리적인 충격에 강하며 저전력으로 동작하는 특성들 때문이다. 그러나 부팅과정에서 수행되는 플래시메모리용 파일시스템의 초기화 작업인 마운팅(mounting)은 비교적 긴 시간을 요구한다. 예를 들어, 그림 1은 리눅스 운영체제를 탑재한 PDA의 부팅시간으로 전체 14초의 부팅시간 중에 4초가 16MB 용량의 플래시메모리를 마운팅하는데 사용된다. 이 마운팅 시간은

† 정 회 원 : 삼성종합기술원 컴퓨팅랩
keunsoo.yim@samsung.com
†† 종신회원 : 서울대학교 컴퓨터공학부 교수
jihong@davinci.snu.ac.kr
(Corresponding author임)
kernkoh@oslab.snu.ac.kr
논문접수 : 2004년 11월 25일
심사완료 : 2005년 8월 5일

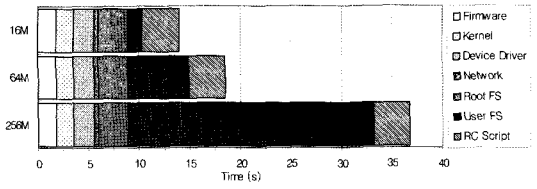


그림 1 플래시메모리 크기에 따른 리눅스 기반 PDA의 부팅시간

플래시메모리의 용량와 저장된 데이타의 양에 비례함을 실험을 통해서 확인하였다. 그런데 현재 플래시메모리의 단일 칩 용량이 매년 2배씩 증가하는 추세에 있다[2,12]. 따라서 플래시메모리용 파일시스템의 마운팅 시간은 내장형시스템의 부팅시간을 지연시키는 가장 중요한 요인이 될 것이다. 뿐만 아니라 개인용 컴퓨터에서도 플래시메모리 저장장치를 USB 포트를 통해서 장착하는 경우에 이를 인식하고 파일시스템을 마운팅하는데 이와 유사한 길이의 시간이 소요된다. 즉, 내장형 및 개인용 컴퓨터시스템의 성능을 위해서 플래시메모리용 파일시스템을 빠르게 마운팅하는 기법을 연구 및 개발할 필요가 있다.

이러한 긴 마운팅 시간은 플래시메모리의 물리적인 제약들과 로그구조의 파일시스템의 특성에서 기인한다[2]. 세부적으로 EEPROM의 일종인 플래시메모리에서는 삭제 연산이 쓰기 연산에 선행되어야 한다. 이 삭제 연산은 쓰기 단위인 페이지보다 큰 블록 단위로 이뤄지며 수 밀리 초에서 수 초사이의 시간이 소요된다. 이 삭제 연산을 상위의 응용프로그램에 대해서 감추기 위해서 JFFS2[3] 및 YAFFS2[4]와 같은 기존 플래시메모리용 파일시스템들은 쓰기 연산을 외부 갱신(out-place update)기법을 통해 처리한다. 외부 갱신기법은 갱신 연산을 기존에 저장된 데이타에 직접 수행하지 않고 기 삭제된 다른 블록에 수행하고 이 곳에 관련 정보를 유지하는 것이다. 그러나 원 데이타를 사용해 갱신된 페이지의 위치를 직접 유추할 수 없기 때문에 마운팅 시점에 전체 플래시메모리 영역을 읽어드려 가장 최신에 갱신된 데이타의 위치를 수집한다. 이렇게 수집된 메타데이타는 메모리 상에서 효율적으로 사용될 수 있도록 재구성된다. 이 두 단계가 기존 파일시스템들의 마운팅 시간을 크게 지연시키는 요인이다.

이러한 문제점을 해결하기 위해서 본 논문에서는 플래시메모리용 파일시스템을 고속으로 마운팅하는 기법을 개발한다. 제안하는 기법은 언마운팅(un-mounting) 시점에 주 메모리상의 메타데이타를 플래시메모리에 저장하고 마운팅 시점에 이를 빠르게 읽어드리는 메타데이타 스냅샷 기법의 일종이다. 본 논문에서는 스냅샷 기법을 NOR형과 NAND형 플래시메모리의 물리적인 특성에 맞춰서 세부적으로 설계한다. 첫째, 플래시메모리의 각 블록을 삭제할 수 있는 최대 횟수는 종류에 따라서 10만 번에서 100만 번사이로 제한된다. 따라서 모든 블록을 균등하게 삭제하는 삭제 균등화(wear-leveling)를 보증해야 모든 블록이 유사한 시점에 삭제한계에 도달하게 되어 사용기간 중의 가용용량을 최대화할 수 있다. 이를 위해 제안하는 스냅샷 기법은 가변길이의 메타데이타 스냅샷을 자주 삭제되지 않은 블록들에 저장하고 이를 연결 리스트를 사용해 관리한다. NOR형 플래시메모리를 위한 기법의 경우 플래시메모리의 첫 번째 블록에 기 저장된 스냅샷의 블록 주소(pointer)를 시간순으로 정렬해 유지함으로써 가장 최신에 저장된 스냅샷의 위치를 빠르게 찾는다. 둘째, 페이지 단위의 입출력만을 지원해 작은 크기의 주소를 저장할 경우 심각한 공간상의 비효율성을 초래하는 NAND형 플래시메모리를 위해 이 기법을 확장한다[5,6]. NAND형 플래시메모리를 위한 기법도 최신에 저장된 스냅샷을 빠르게 찾으며 삭제 균등화를 보증하는 특성을 갖는다.

셋째, 제안하는 스냅샷 기법들은 마운팅 시점에 가장 최근에 저장된 메타데이타 스냅샷과 실제 파일시스템 데이타사이의 일관성 검사를 빠르게 수행한다. 만약 언마운팅이 정상적으로 수행되지 않아 이 스냅샷이 유효하지 않으면 세 가지 새롭게 제안하는 고속 에러 복구 기법들을 사용해 메타데이타를 재구성한다. 넷째, 데이타 압축기법을 사용해 스냅샷을 압축함으로써 스냅샷을 저장하기 위한 플래시메모리상의 기억공간의 크기를 줄이고 이를 저장하는데 소요되는 언마운팅 시간도 단축한다.

성능평가 결과 제안하는 스냅샷 기법들은 플래시메모리의 삭제 균등화를 보증하며, 리눅스에 탑재된 플래시메모리용 파일시스템인 JFFS2와 비교해 마운팅 시간을 100배가량 단축함을 보인다. 예를 들어, 플래시메모리의 용량이 128MB이고 저장된 데이타의 양이 100MB인 경우에 10초 이상인 JFFS2의 마운팅 시간을 100 미리 초 이하로 단축한다. 그리고 오류가 발생한 파일시스템의 메타데이타 재구성시에 제안하는 고속 오류 복구 기법들을 사용하면 입출력 시간을 크게 단축시킴을 보인다. 또한 스냅샷 압축기법을 사용함으로써 제안하는 스냅샷 기법들의 언마운팅 시간을 50% 이상 단축시킴을 보인다.

본 논문의 구성은 다음과 같다. 2장에서는 플래시메모리용 파일시스템의 기본 구조와 기존 메타데이타 스냅샷 기법의 특성과 문제점에 대해 기술한다. 3장에서는 제안하는 NOR형과 NAND형 플래시메모리들을 위한 메타데이타 스냅샷 기법들에 대해 설명하며, 4장에서는

성능평가 결과를 기록한다. 그리고 5장에서는 결론을 맺는다.

2. 관련 연구

플래시메모리는 가볍고 물리적인 충격에 강하며 저전력으로 동작하는 내장형시스템의 저장장치로서 적합한 특성들을 가지고 있다. 플래시메모리의 종류에는 NOR형과 NAND형 두 가지가 있다. NOR형 플래시는 워드 단위의 입출력을 제공하고 빠른 읽기 성능을 보여 주로 코드용 메모리로 사용된다. NAND형 플래시는 512바이트 또는 2048바이트 단위의 페이지 입출력만을 지원하고 쓰기 연산의 속도가 비교적 빠르기 때문에 주로 데이타 저장장치로 사용된다.

플래시메모리에서는 삭제 연산이 쓰기 연산을 선행해 수행되어야 하며, 각 블록에 최대로 수행 가능한 삭제 연산의 횟수에는 물리적인 제한이 있다. 호스트로부터 이러한 삭제 연산을 감추고 삭제 균등화를 보증하기 위해서 호스트에서 생성된 논리 주소를 플래시메모리상의 물리 주소로 변환하는 플래시 변환 계층(Flash Translation Layer, FTL)[7-9]이 개발되어 사용되고 있다. FTL을 사용하면 하드 디스크와 같은 저장장치 인터페이스 제공할 수 있어서 호스트에서는 FAT 또는 ext2 [10]와 같은 하드 디스크용 파일시스템을 사용해 FTL 기반 플래시메모리 저장장치를 제어한다. 이러한 이점으로 인하여 CompactFlash, SmartMedia 카드 등과 같은 휴대용 플래시메모리 저장장치에서는 FTL을 사용하고 있다. 그러나 FTL 관련 기술은 국제 특허들로 등록되어 있어서 사용에 제한이 따르며 쓰기 작업을 빈번하게 수행하는 시스템에 사용되면 성능이 저하될 수 있는 것으로 알려져 있다[13].

따라서 내장형시스템에서는 해당 시스템의 설계 목적에 따라서는 데이타를 로그 형태로 저장하는 저널링(journaling) 기법에 기반한 JFFS2 또는 YAFFS2와 같은 플래시메모리용 파일시스템들을 사용해서 직접 플래시메모리를 제어한다.[1] 그러나 그림 2에 제시된 것과 같은 저널링에 기반한 플래시메모리용 파일시스템들은 긴 부팅시간을 초래할 수 있다.[2] 그림에서 단계 I은 루트 디렉토리를 위한 inode[10]와 디렉토리 엔트리 그리고 이 엔트리에는 등록된 inode 번호가 2인 파일 A의 inode와 데이타를 저장하고 있는 상태를 나타낸다. 이 파일의 inode에는 버전인 1값과 첨부 데이타의 범위인

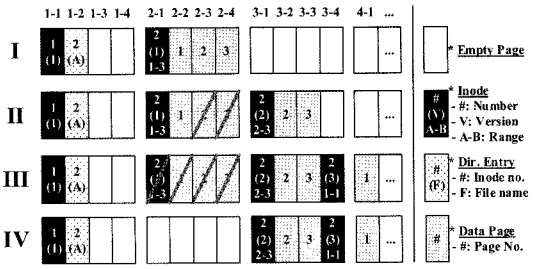


그림 2 저널링 기법을 사용한 플래시메모리용 파일시스템의 데이타 관리방식

1-3값이 기록되어 있다. 단계 II는 이 파일에 대한 갱신 연산이 수행되고 난 경우로 저널링 기법에 의해서 신규 버전의 inode를 생성하고 여기에 갱신 데이타인 2-3 페이지를 첨부해 저장한 것을 보인다. 이 연산이 수행되고 나면 기존 inode에 첨부되어 있던 페이지 2-3은 더 이상 유효하지 않게 된다. 만약 플래시메모리상의 여분의 공간비율이 특정 값 - 예를 들면 20% - 이하이면 새로운 빈 공간 확보를 위하여, 단계 III과 같이 유효하지 않은 페이지의 비율이 가장 높은 블록 2에서 유효한 페이지들을 다른 위치로 복사하고 단계 IV와 같이 블록 2를 삭제한다. 단계 III과 IV는 백그라운드 작업으로 수행되어 저널링 기법을 사용한 플래시메모리용 파일시스템들의 경우 삭제 연산을 응용프로그램으로부터 감출 수 있다. 그리고 이 파일시스템들은 신규 inode와 데이타 저장을 위한 블록을 해당 블록의 기 삭제 횟수를 고려해 선택함으로써 삭제 균등화 속성을 보증한다.

저널링 기법에 기반한 파일시스템들은 최종적으로 갱신된 데이타에 대한 inode들을 수집하기 위해서 마운팅 시점에 전체 플래시메모리 공간을 읽어 메타데이타를 구성해 메모리 상에 유지한다. JFFS2의 경우 메모리상에 메타데이타를 그림 3과 같이 구성한다. 이때 inode 번호는 디렉토리 또는 파일을 지칭하는 기호이다. inode는 해당 디렉토리 또는 파일의 핵심 메타데이타로서

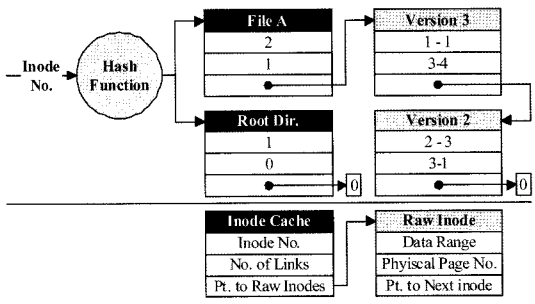


그림 3 저널링 플래시 파일시스템의 메모리상의 메타데이타 구조

1) Microsoft사의 TFAT(Transactional FAT)와 같은 저널링 기법에 기반하지 않는 플래시메모리 파일시스템도 존재한다.

2) JFFS2의 경우 그림 1에 제시된 것과 같은 긴 부팅 시간을 초래하며, YAFFS의 경우 이를 상당 부분 개선하였으나 아직까지도 마운팅 시간이 1-5초 가량 소요되어 내장형시스템의 고속 부팅에 충분하지 않다.

inode 번호와 해당 디렉토리 또는 파일의 수정된 횟수를 의미하는 inode의 버전 그리고 파일의 경우 포함하고 있는 데이타의 범위를 가진다. inode 다음에는 디렉토리의 경우 디렉토리 엔트리가 파일의 경우 데이타가 기록된다. 디렉토리 엔트리에는 해당 디렉토리 내부에 저장되어 있는 자식 디렉토리와 파일의 inode 번호와 이름이 저장되며, 데이타 부분에는 해당 쓰기 연산에 의해서 수정된 페이지만을 저장하고 있다. 예를 들어 파일 A의 두 번째 페이지에 접근하려는 경우, 이 파일의 inode 번호를 사용해 메타데이타 상의 inode 캐시를 검색하고 이를 통해 찾은 inode 캐시에 저장되어 있는 포인터를 따라서 두 번째 페이지를 포함한 물리(raw) inode를 검색한다. 최종적으로 이렇게 찾은 물리 inode에 기록된 물리 페이지 번호를 사용해서 플래시메모리 상에 저장되어 있는 해당 페이지에 접근한다.

그러나 플래시메모리의 전체 기억공간을 읽어드리고 이를 바탕으로 메모리 상의 메타데이타를 구성하는 것은 각각 긴 입출력 시간과 CPU 시간을 필요로 한다. 이러한 특성은 특히 플래시메모리의 용량이 크거나 저장된 데이타의 양이 많은 경우에 기존 플래시메모리용 파일시스템들의 마운팅 시간을 크게 지연시키는 요인이 되고 있다.

플래시메모리용 파일시스템에 사용되고 있는 저널링 기법은 하드 디스크 파일시스템의 하나인 로그구조 파일시스템(Log-structured File System, LFS)[11]에서 사용되던 것으로, LFS에서는 메타데이타 스냅샷 기법을 통해 마운팅 시간 줄이고 빠르게 예외상황으로부터 복구된다. 세부적으로 LFS는 그림 4(a)와 같이 주기적으로 메모리 상의 메타데이타를 하드 디스크에 저장하고 저장위치를 첫 번째 섹터에 위치한 슈퍼 블록에 기록한다. 하지만 이 방식을 플래시메모리에 적용하는데 다음과 같은 두 가지 문제점들이 있다. 하나는 데이타 일관성(integrity) 문제로 슈퍼 블록을 읽고 이를 삭제한 이후에 새로운 메타데이타 위치를 반영한 슈퍼 블록을 쓰는 비교적 긴 수행시간 도중에 전원이 차단되는 것과 같은 예외상황이 발생하면 슈퍼 블록이 손상되는 것이다. 다른 하나는 슈퍼 블록이 빈번하게 삭제되어 삭제 한계 횟수에 다른 블록에 비해 빠르게 도달하게 되어 삭제 균등화 특성을 깨뜨리는 것이다.

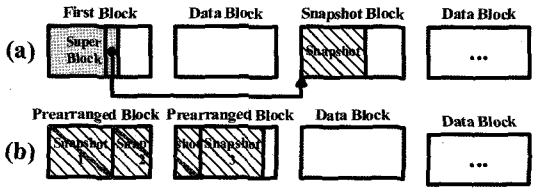


그림 4 LFS(a)와 FTL(b)의 메타데이타 스냅샷 기법

FTL에서도 유사한 목적으로 메타데이타 스냅샷 기법과 유사하게 메타데이타를 로그 형태로 저장한 사례가 있다[5]. Log-Block 구조의 FTL의 경우 메타데이타는 크기가 512 바이트와 같이 비교적 작은 크기로 고정되어 있다. 그렇기 때문에 그림 4(b)와 같이 특정 영역을 스냅샷 저장에 할당하고 이 영역을 스냅샷 크기로 분할한 후에 분할된 영역에 라운드로빈(round-robin) 방식으로 스냅샷을 저장한 후에, 마운팅 시점에는 이 영역을 전부 읽어드려 가장 나중에 저장된 스냅샷을 찾는 방식을 사용한다[8]. 이를 통해서 FTL에서 사용되는 스냅샷 기법은 데이타 일관성 문제를 해결하고 삭제 균등화를 보증한다. 그러나 이 기법을 플래시메모리용 파일시스템들에 적용하기는 어려운데 그 이유는 이 파일시스템들은 메타데이타의 크기가 100KB에서 1MB사이의 가변적인 값이어서 삭제 균등화를 보증하지 못하기 때문이다. 예를 들어, 만약 스냅샷 크기가 할당된 영역의 크기와 비교해 비교적 큰 경우에 이 영역은 삭제 한계에 빠르게 도달할 것이며, 역으로 스냅샷의 크기가 작은 경우에는 할당된 영역외의 부분이 빠르게 삭제 한계에 도달할 것이다. 뿐만 아니라 이 기법은 메타데이타가 바뀌는 경우마다 이를 스냅샷 형태로 저장하는데 이 파일시스템들에서는 메타데이타의 크기가 커서 이러한 방식을 사용하면 입출력 성능을 크게 저하시킬 수 있다. 다음 장에서는 이러한 문제점을 해결하고 그 외의 몇 가지 새로운 장점을 갖는 새로운 메타데이타 스냅샷 기법에 대해 기술한다.

3. 제안하는 고속 마운팅 기법

이 장에서는 고속 마운팅을 위한 NOR형과 NAND형 플래시용 메타데이타 스냅샷 기법들과 세 가지 고속 오류 복구 기법들에 대해서 설명하고 제안하는 스냅샷 기법의 언마운팅 시간을 단축시키는 스냅샷 압축기법에 대해서 소개한다.

3.1 NOR형 플래시를 위한 스냅샷 기법

제안하는 스냅샷 기법의 설계 목표는 삭제 균등화를 보증하며 동시에 가장 최근에 저장된 스냅샷을 빠르게 찾는 것이다. 이를 위하여 제안하는 기법은 메타데이타 스냅샷을 연결 리스트를 사용해 가변 길이의 분산된 영역에 저장하고 트리 자료구조를 사용해 저장된 스냅샷의 위치를 유지한다.

제안하는 기법은 그림 5와 같이 플래시메모리의 첫 블록을 루트 블록으로 예약하고 해당 블록에 스냅샷 헤더 블록의 주소를 순차적으로 저장한다. 일반적으로 NOR형 플래시메모리의 블록 크기(B_{size})는 128KB이고 물리 블록에 대한 주소의 크기(P_{size})는 2 바이트이기 때

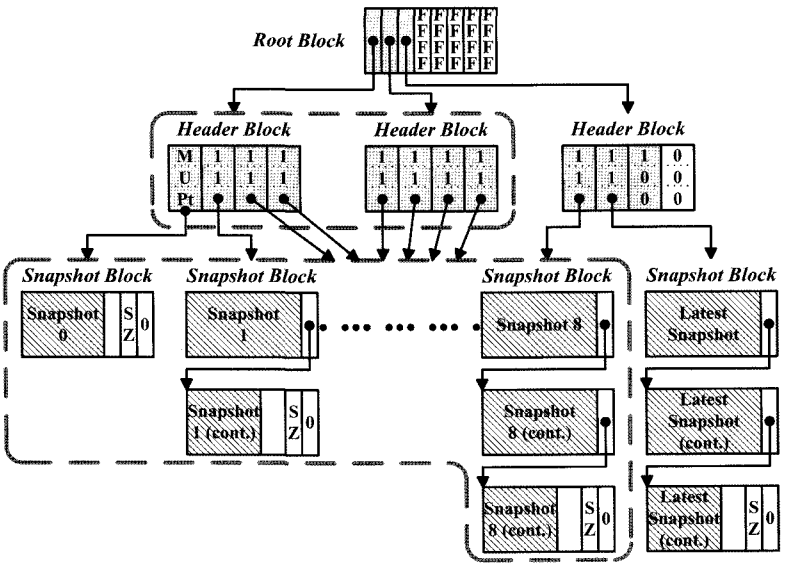


그림 5 NOR형 플래시를 위한 메타데이타 스냅샷 기법

문에 루트 블록에 저장할 수 있는 해더 블록의 주소의 개수는 B_{size} / P_{size} = 2^{16}개이다. 이때 블록 주소는 루트 블록의 첫 번째 워드부터 시작해 순차적으로 기록하며, 만약 루트 블록내의 공간이 가득 찬 경우에는 이 블록을 삭제하고 이후 주소들을 다시 첫 워드부터 시작해 순차적으로 기록한다. 마운팅 시점에는 가장 마지막에 저장된 주소를 찾는 방법으로 순차 또는 이진 검색 기법을 사용한다. 순차 검색은 블록의 첫 워드부터 순차적으로 값을 읽어 만약 값이 $FFFF_{(16)}$이면 그 이전에 읽었던 워드의 값이 가장 최근에 저장된 주소로 고려하는 것이다. 예를 들면, 그림 5의 루트 블록은 해더 블록에 대한 8개의 포인터를 가지고 있다. 순차 검색 방법을 사용하면 첫 번째 포인터부터 4번째 값이 $FFFF_{(16)}$인 포인터까지 총 4개의 포인터를 읽어야 세 번째 포인터가 가장 최근에 저장된 해더 블록을 가리키고 있음을 알 수 있다. 이때 현재 저장되어 있는 해더 블록 포인터의 개수에 따라 최소 2개에서 최대(B_{size} / P_{size})개까지 포인터를 읽어야 하기 때문에 평균적으로 {2 + (B_{size} / P_{size})} / 2개의 포인터를 읽게 되어 시간 복잡도로 O(B_{size} / P_{size})를 갖는다.

그리고 이진 검색은 루트 블록을 두 개의 서브 블록들로 나누고 이 서브 블록들의 경계에 있는 워드의 값을 읽어 만약 이 값이 0이면 좌측 서브 블록을 선택하고 그렇지 않으면 우측 서브 블록을 선택해 이 과정을 재귀적으로 반복하여 가장 최근에 저장된 주소를 찾는 방법이다. 세부적으로 그림의 루트 블록의 경우 이진 검색을 사용하면 중간 위치의 4번째 포인터를 읽고 이 값이 $FFFF_{(16)}$임을 알고 1-3번째 포인터에 해당 하는 좌측 서브 블록을 선택한다. 이후 좌측 서브 블록의 중앙인 2번째 포인터를 읽고 이 값이 $FFFF_{(16)}$이 아니기 때문에 이 값을 기준으로 우측 서버 블록을 선택한다. 우측 서버 블록에는 3번째 포인터만이 존재하기 때문에 이 값을 읽어 $FFFF_{(16)}$임을 확인할 수 있다. 이미 4번째 포인터가 유효하지 안다는 사실을 알고 있기 때문에 가장 최근에 저장된 해더 블록을 3번째 포인터의 값을 사용해 접근할 수 있다. 이때 시간 복잡도는 이진 검색이므로 O(lg(B_{size} / P_{size}))이다.

그림 5에 제시된 것과 같이 루트 블록에 저장되어 있는 각 주소는 이에 대응하는 하나의 해더 블록을 가리킨다. 이때 그림 상에서 점선으로 된 영역으로 둘러싸인 해더 블록들과 이와 관련된 데이타들은 가장 최근에 저장한 것이 아니기 때문에 삭제하여도 무방하다. 하나의 해더 블록은 여러 개의 스냅샷 해더로 구성되며 각 스냅샷 해더는 마운팅 표시(M)와 언마운팅 표시(U) 그리고 스냅샷 블록의 주소(Pt)로 구성된다. 일반적으로 스냅샷 해더의 크기(H_{size})는 4바이트이기 때문에 하나의 해더 블록에 저장할 수 있는 스냅샷 해더의 개수는 B_{size} / H_{size} = 2^{15}개 이다. 해더 블록 역시 스냅샷 해더를 순차적으로 저장하며 앞서 언급한 순차 및 이진 검색 기법을 사용해 가장 최근에 저장한 스냅샷 해더를 찾는다. 최종적으로 이렇게 찾은 스냅샷 해더에 저장되어 있는 스냅샷 블록의 주소를 사용해 실제로 메타데이타 스냅샷에 접근한다.

플래시메모리용 파일시스템의 메타데이타 스냅샷은

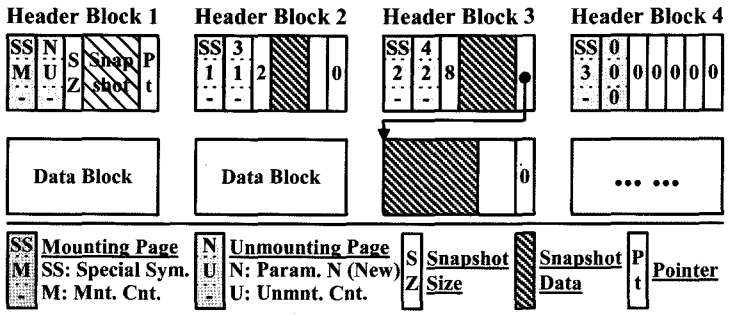


그림 6 NAND형 플래시를 위한 메타데이타 스냅샷 기법(N = 4)

크기가 가변적이기 때문에 제안하는 기법에서는 이 스냅샷을 연결 리스트를 사용해 다수개의 임의의 위치에 존재하는 블록에 저장한다. 이때 스냅샷과 해더 블록을 저장할 블록을 기 삭제 횟수를 고려해 선택함으로써 삭제 균등화를 보증한다. 연결 리스트를 구성하기 위해서 각 블록의 마지막 워드는 다음번 블록의 주소를 저장하는데 사용되며 이 워드의 값이 0이면 리스트의 마지막 블록이다. 마지막 블록은 추가적으로 크기 필드(SZ)를 사용해 해당 블록에 저장된 스냅샷의 크기를 유지한다.

이상의 내용을 요약해 보면 제안하는 스냅샷 기법은 평균적으로 $\lg(B_{size} / P_{size}) \times P_{size} + \lg(B_{size} / H_{size}) \times H_{size}$ = 92바이트만을 읽어서 가장 최근에 저장된 스냅샷 데이타의 위치를 찾아낸다. 이후 실제 스냅샷을 읽는데 시간이 소요되지만 스냅샷이 크기는 일반적으로 100KB에서 1MB사이로 비교적 작기 때문에 고속 마운팅 기능을 지원할 수 있다. 또한 제안하는 스냅샷 기법에서 루트 블록은 매 $B_{size} / P_{size} \times B_{size} / H_{size} = 2^{31}$ 번의 스냅샷 저장시마다 한번씩 삭제되며 그 외 해더 및 스냅샷 블록은 기 삭제 횟수를 고려해 선택된 블록에 저장되기 때문에 삭제 균등화 특성을 보증한다.

파일시스템이 마운트된 이후에 쓰기 연산에 의해서 처음으로 메모리상의 메타데이타가 수정된 경우 가장 최근에 사용된 해더 블록 다음번에 위치한 해더 블록의 M 값을 1로 설정한다. 언마운팅 시점에는 이 M 값이 1로 설정되어 있는 경우에만 스냅샷을 저장하며 스냅샷 저장이 완료되면 이 해더 블록의 U 값도 1로 설정한다. 이를 통해서 이후 마운팅 시점에 M 값이 설정되어 있고 U 값이 설정되어 있지 않으면 파일시스템이 안전하게 언마운트 되지 않았음을 알아낼 수 있다.

3.2 NAND형 플래시를 위한 스냅샷 기법

NOR형 플래시메모리를 위한 제안하는 스냅샷 기법을 NAND형 플래시에 적용할 경우 심각한 공간상의 낭비를 야기한다. 왜냐하면 NAND형 플래시는 페이지 단위의 입출력만을 지원하기 때문이다. 세부적으로 2바이트 크기의 주소를 하나의 페이지에 기록하면 2바이트를 제외한 페이지 내부의 모든 공간이 내부파편이 되는데 이 내부파편 공간에 다른 데이타를 저장하기 위해서는 이 페이지를 포함한 블록에 대한 삭제 연산을 선행해야 한다[5,6]. 따라서 NAND형 플래시에서 내부파편의 비율을 효율적으로 줄이기 위해서 이를 위한 스냅샷 기법을 제안한다.

NAND형 플래시를 위한 제안하는 스냅샷 기법은 플래시메모리상의 첫 N개의 블록들을 해더 블록으로 예약해 사용한다. 그림 6에 제시된 것과 같이 각 해더 블록은 마운팅 페이지와 언마운팅 페이지 그리고 스냅샷 데이타로 구성된다. 메모리상의 메타데이타가 수정된 경우 이 영역의 블록 중의 하나를 라운드-로빈 방식으로 선택하고 이 블록을 삭제하고 이 블록과 연결된 스냅샷 데이타 블록들이 있는 경우 이 블록들도 삭제한다. 이후 이 블록의 마운팅 페이지에 특수 기호(SS)와 해더 블록의 크기인 N값 그리고 매 스냅샷 저장시마다 1씩 증가되면 마운팅 횟수(M)를 기록한다. 이때 특수 기호는 해당 블록이 해더 블록임을 표시하는 목적으로 사용되며 마운팅 페이지의 여분의 영역(out-of-band area[3])에 저장된다.

언마운팅 시점에 만약 메타데이타가 수정되었으면 메타데이타 스냅샷의 크기(SZ)와 데이타를 선택된 해더 블록에 저장한다. 만약 스냅샷의 크기가 해당 해더 블록의 여분공간보다 크면 이를 연결 리스트를 사용해 다수개의 블록에 저장한다. 이때 주소 필드(Pt)를 사용해 다음번 블록을 가리키며 이 값이 만약 0이면 해당 블록이 마지막 블록이다. 이 작업이 완료되면 최종적으로 마운팅 횟수와 같은 값을 가지는 언마운팅 횟수(U) 값을 언마운팅 페이지에 저장한다.

즉, 제안하는 기법에서는 마운팅 페이지와 언마운팅 페이지에서만 내부파편이 발생한다. 스냅샷 크기와 데이

3) NAND형 플래시의 페이지에는 여분의 영역(out-of-band 또는 spare area)이 존재하며 이 영역의 크기는 페이지의 크기가 512바이트와 2048바이트인 경우에 각각 16바이트와 64바이트이다.

타 그리고 주소 필드들은 한 시점에 순차적으로 저장되기 때문에 이 영역에서는 내부파편이 발생하지 않는다. 또한 제안하는 기법에서는 한번의 스냅샷 저장 시에 스냅샷 데이타의 크기와 무관하게 오직 하나의 해더 블록만이 삭제되기 때문에 해더 블록들은 서로 균등하게 삭제되는 특성을 갖는다. 그러나 해더 블록의 삭제 횟수와 그 외의 데이타 블록간의 삭제 횟수에는 입출력 패턴에 따라서 차이가 있을 수 있다. 이러한 문제를 해결하기 위해서 제안하는 기법은 수식 1을 바탕으로 N값을 동적으로 조정하는데 이때 B_{size}, F_{size}, E_{count}는 각각 플래시메모리 블록의 크기, 플래시메모리의 용량, 그리고 해당 마운팅 시점에 수행된 전체 삭제 연산의 횟수를 의미한다. 수식 1은 해더 블록들의 평균 삭제 횟수와 그 외 데이타 영역의 평균 삭제 횟수를 현재의 입출력 패턴 하에서 균등하게 유지할 수 있는 N' 값을 계산하는 것이다. 언마운팅 시점에 스냅샷 데이타를 저장한 이후에 수식 1을 사용해 N' 값을 계산하고 만약 이 값이 N보다 크면 N 값을 1만큼 증가시키고 역으로 N 값보다 작으면 N 값을 1만큼 감소시킨다. 이때 N 값의 범위는 최소 2에서 최대 전체 블록의 개수의 10%로 제한된다. 이렇게 결정된 N 값은 언마운팅 페이지에 기록되어 이후 마운팅 시점부터 해더 블록의 크기로 사용된다.

$$\frac{1}{N'} = \frac{E_{count}}{F_{size}/B_{size} - N'} \rightarrow N' = \left\lceil \frac{F_{size}}{B_{size}(E_{count}+1)} \right\rceil \quad (1)$$

마운팅 시점에는 모든 해더 블록의 마운팅 횟수 값을 읽어 가장 최근에 저장된 스냅샷을 찾는다. 이때 해더 블록의 크기인 N 값을 알지 못하기 때문에 앞에서부터 순차적으로 해더 블록을 읽고 특수 기호의 존재 여부를 확인하며 이 작업은 특수 기호가 저장되지 않은 블록을 읽거나 max{N} 값인 전체 블록의 10%를 읽은 경우까지 수행된다. 이렇게 마운팅 횟수를 모두 읽은 이후에는 마운팅 횟수 값이 가장 큰 블록을 선택함으로써 가장 최근에 저장된 스냅샷을 찾고 이 블록에 저장된 마운팅 및 언마운팅 횟수 값을 비교함으로써 파일시스템이 안전하게 언마운트 되었는지 여부를 검사한다. 만약 이 값이 서로 같으면 안전하게 언마운트 되었음을 의미하기 때문에 해당 블록에 저장되어 있는 스냅샷 데이타를 메모리로 읽어드려 메타데이타로 사용함으로써 마운팅을 완료한다. 즉, 최악의 경우 제안하는 기법은 max{N} x P_{size} + P_{size} 바이트를 읽음으로써 가장 최근에 저장된 스냅샷 데이타를 찾을 수 있다.

3.3 고속 오류 복구 기법들

만약 파일시스템이 안전하게 언마운트되지 않았으면 이는 전원 차단 또는 운영체제 오류와 같은 예외 상황이 발생했음을 의미한다. 이 경우 메모리상의 버퍼에 저장되어 있던 데이타는 플래시메모리에 저장되지 않아 유실되었을 수 있다. 하지만 저널링 기법에 기반한 플래시메모리용 파일시스템들은 쓰기 연산을 시간 순서대로 순차적이며 일관된 형태로 수행하기 때문에 저장되어 있는 데이타 자체는 일관성을 유지한다. 그렇기 때문에 기존 파일시스템들이 부팅 과정에서 메타데이타를 구성하는 것과 같은 방식을 통해서 일관성 있는 메타데이타를 재구성할 수 있지만 이는 비교적 긴 시간이 소요된다. 따라서 본 논문에서는 이 경우 빠르게 오류를 복구할 수 있는 세 가지 기법들을 제안한다.

첫째, 그림 7(a)와 같이 플래시메모리상에 저장되어 있는 데이타 페이지는 오류 복구 과정에서 읽지 않는다. 이 기법은 각 inode는 이에 첨부된 데이타 페이지의 위치와 크기를 가지고 있다는 특성을 활용한 것으로 이 정보를 바탕으로 모든 데이타 영역을 읽지 않을 수 있다. 기존 파일시스템들의 경우 데이타의 영역의 오류 검사를 - 예를 들면 CRC 검사 - 위해서 데이타 영역을 마운팅 시점에 읽었는데 제안하는 기법에서는 이를 실제로 데이타가 사용되는 시점으로 지연시킨다. 둘째, 그림 7(b)에 제시된 것과 같이 특정 블록에 데이타를 저장할 때 첫 페이지부터 순차적으로 기록함으로써 뒷부분에 위치한 빈 페이지들을 읽지 않는다. 블록 내에서 데이타를 순차적으로 기록하는 정책을 사용해 데이타를 저장한 경우 모든 블록에서 처음으로 빈 페이지를 읽으면 그 이후의 페이지도 모두 빈 페이지라는 사실이 항상 성립하기 때문에 이후 페이지를 읽지 않아도 된다. 마지막으로 이 두 가지 기법을 그림 7(c)와 같이 병행해 사용하면 inode와 빈 페이지들 중의 첫 페이지만을 읽어 메모리상의 메타데이타를 재구성함으로써 오류 상황 복구시의 입출력 시간을 크게 단축할 수 있다.

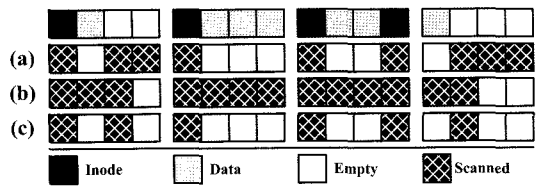


그림 7 제안하는 고속 오류 복구 기법들

3.4 압축을 통한 고속 언마운팅 기법

제안하는 스냅샷 기법은 메모리상의 메타데이타가 변경된 경우 언마운팅 시점에 이를 플래시메모리에 저장한다. 그런데 플래시메모리는 쓰기 속도가 비교적 느려서 스냅샷의 크기가 큰 경우에 언마운팅 시간이 길어질 수 있다. 따라서 본 논문에서는 언마운팅 시간을 줄이기 위해서 메타데이타를 압축해 저장하는 기법을 사용한다 [5,6]. 이때 압축에 걸리는 시간이 스냅샷을 플래시메모

리에 저장하는데 걸리는 시간에 비해 크게 짧기 때문에 압축기법은 언마운팅 시간을 크게 줄이고 스냅샷을 저장하는데 사용되는 공간도 줄이는 효과를 갖는다. 이렇게 저장된 스냅샷은 마운팅 시점에 복원 과정을 거쳐 사용된다.

4. 성능 평가

제안하는 기법들의 성능은 펜티엄-III 850MHz를 장착한 컴퓨터에서 동작하는 리눅스 시스템을 바탕으로 이에 탑재된 플래시메모리용 파일시스템인 JFFS2와 비교해 평가한다. 다양한 종류와 용량의 플래시메모리에서의 성능을 평가하기 위해서 표 1과 같은 특성을 갖는 가상 디바이스 드라이버를 개발하였다. 이 드라이버는 입출력 시간만큼 CPU를 잡고 있는 방식으로 사용해 DMA를 사용하지 않는 실제 플래시메모리 장치와 동일하게 동작한다. 이 드라이버는 상위 플래시메모리용 파일시스템들에 일반적인 메모리 디바이스의 인터페이스(Memory Technology Device, MTD)[3]를 제공하며 수행된 입출력 연산의 횟수를 기록해 추후 분석과정에 사용가능하게 한다. 추가적으로 실험 시에 플래시메모리용 파일시스템에서 소요된 CPU 시간도 측정하였다.

첫째, JFFS2의 마운팅 시간을 분석하였다. 그림 8에 제시된 것과 같이 JFFS2의 마운팅 시간은 플래시메모리의 용량과 정비례 관계에 있다. 이러한 특성은 JFFS2의 경우 마운팅 시점에 NOR형과 NAND형 플래시들에 대해서 전체 용량의 103%와 125%에 해당하는 양의 데이타를 읽어들이기 때문이다. 이 비율이 100%를 상회하는 이유는 마운팅 과정에서 이미 메모리상의 버퍼 캐시에서 제거된 데이타를 재 참조하는 경우가 발생해서이다. 또한 이 비율이 NAND형 플래시에서 더 높은 이유는 페이지 단위의 입출력만이 가능해 한 페이지 내에 있는 불필요한 데이타도 함께 읽어들여서이다. 마운팅 시간은 NAND형의 경우 NOR형에 비해서 길게 나타났는데 이 이유는 NAND형의 읽기 속도가 NOR형의 그것보다 2-3배가량 느리기 때문이다. 비록 반도체 기술의 발달에 인해 플래시메모리의 입출력 속도가 점진적으로 개선되고는 있기는 하지만 앞으로도 하나의 플래시메모리 칩의 전체 기억공간을 읽어드리는데 단일 칩의 용량이 매년 2배씩 증가하는 추세에 있기 때문에 5-20초가량이 소요될 것으로 예상된다. 또한 마운팅 과정에서 플래시메모리에서 읽은 페이지의 유형을 판단하고 inode 캐시를 구성하는 등의 작업은 플래시메모리에 저장된 데이타의 용량이 1MB이하인 경우에 1초 이하의 시간이 소요됨을 확인하였다[10].

다음으로 JFFS2의 마운팅 시간을 저장된 데이타의 양을 변화시키며 측정하였다. 이 실험에서는 128MB 용량의 플래시메모리를 사용하였으며 저장용 데이타로는 리눅스 커널의 소스를 사용하였다. 그림 9에 제시된 것과 같이 저장된 데이타의 양이 증가하면 마운팅 과정에서의 CPU 시간이 비례해 증가하며 입출력 시간은 거의 변화하지 않는다. 또한 CPU 시간은 플래시메모리의 용량과는 무관함을 확인하였다. 예를 들어 256MB 플래시메모리를 사용해 동일한 데이타를 저장한 경우에 그림 9와 거의 동일한 CPU 시간을 보였다. 그러나 이 CPU 시간은 호스트의 CPU의 계산능력에 크게 의존하는 특성을 보이는데, 이 실험에서는 내장형시스템에서는 일반적으로 사용하는 CPU보다 계산능력이 뛰어난 CPU를 사용였기 때문에 실제 내장형시스템에서 마운팅에 소요되는 CPU 시간은 이 실험에서 제시된 것보다 클 수 있다. 또한 만약 하드웨어적으로 DMA 입출력을 지원한다면 이 CPU 시간의 상당 부분을 입출력 시간과 겹쳐 수행할 수 있을 것이다.

따라서, JFFS2의 마운팅 시간 중에서 입출력 시간은 플래시메모리의 용량에 비례하고 CPU 시간은 저장된 데이타의 양에 비례한다. 제안하는 스냅샷 기법들은 작은 크기의 스냅샷만을 읽어 부과적인 처리 없이 바로 메모리상의 메타데이타로 사용하기 때문에 마운팅 과정

표 1 플래시메모리의 입출력 특성

Type	Model	Read		Write		Erase
NOR	Unit	2B	512B	2B	512B	128KB
	Speed	75ns	19.2μs	14μs	3.58ms	1.2s
NAND	Unit	512B		512B		16KB
	Speed	35.9μs		226μs		2ms

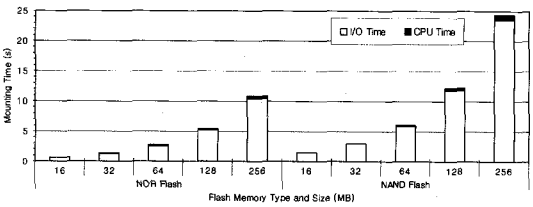


그림 8 플래시메모리 종류와 용량에 따른 JFFS2의 마운팅 시간

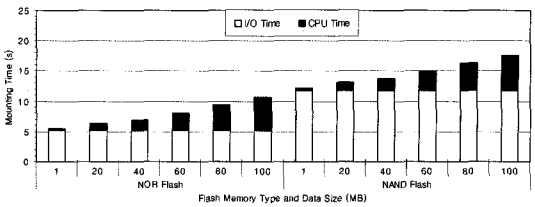


그림 9 저장된 데이타의 크기에 따른 JFFS2의 마운팅 시간

에서 소요되는 입출력 및 CPU 시간을 모두 줄일 수 있다. 이 점이 입출력 시간만을 줄이는 기존 고속 마운팅 기법과의 차이점이다[4].

다음으로 제안하는 기법의 마운팅 시간을 정확하게 이해하기 위해서 JFFS2의 메모리상의 메타데이타를 분석하였다. 그림 10은 128MB 용량의 NOR형 플래시를 사용한 경우 메타데이타 스냅샷의 크기이다. 메타데이타 스냅샷은 크게 inode 캐시, 물리 inode, 그리고 삭제 블록 정보로 나뉜다. 그림을 통해서 저장된 데이타의 양이 많으면 보다 많은 양의 inode 캐시와 물리 inode를 사용함을 알 수 있다. 또한 삭제 블록 정보는 각 물리 블록관련 데이타를 유지해 삭제 연산 시에 사용하는 것으로 이 데이타의 크기는 플래시메모리의 크기와 정비례 관계에 있다. NAND형 플래시를 사용한 실험에서도 유사한 특성을 확인할 수 있었다. 이 실험에서 저장된 데이타의 양이 증가하면 스냅샷의 크기도 비례해 커지는 특성이 나타났는데 이러한 스냅샷 크기는 그림 10에 제시된 것과 같이 JFFS2와 유닉스에서 널리 사용되는 zlib 압축 알고리즘을 사용해 약 절반가량으로 줄일 수 있다.

둘째, 제안하는 스냅샷 기법의 마운팅 시간을 분석하였다. 그림 11은 128MB 용량의 NOR형 플래시상의 유효한 스냅샷이 존재하는 경우 제안하는 기법의 마운팅 시간으로 JFFS2의 그것에 비해서 100분의 1이하로 단축되었다. 이와 같은 성능 향상은 제안하는 기법이 플래시메모리의 전 영역을 읽어드리지 않고 기 생성된 메타데이타 스냅샷만을 읽어 부과적인 처리 없이 바로 메타데이타로 사용하기 때문이다. 세부적으로 저장된 데이타의 양이 많으면 마운팅 시의 입출력 시간이 길어지는데 이는 스냅샷의 크기가 증가되었기 때문이다. 이 입출력 시간은 스냅샷 압축기법을 사용한 경우 절반가량으로 단축되며, 이 경우 압축된 스냅샷을 복원하는데 비교적 짧은 CPU 시간이 소요된다. 또한 실험을 통해서 NAND형 플래시를 위한 제안하는 스냅샷 기법의 경우 장치의 특성상 입출력 시간이 다소 지연되지만 전반적으로 유사한 성능을 보임을 확인하였다.

하지만 제안하는 스냅샷 기법은 JFFS2와 비교해 상대적으로 긴 언마운팅 시간을 갖는다. 그림 12는 NOR형 플래시를 위한 제안하는 기법의 언마운팅 시간으로 이 시간은 저장된 데이타의 양에 비례하는 특성을 보이는데 이는 언마운팅 과정의 대부분이 플래시메모리에 대한 스냅샷의 쓰기 연산이기 때문이다. 또한 스냅샷 압축기법을 사용한 경우 언마운팅 시간 중에서 입출력 시간은 약 절반으로 감소하고 CPU 시간은 근소하게 증가한다. JFFS2의 경우 언마운팅 시간이 0.5초 이하이기 때문에 NOR형 플래시를 위한 제안하는 스냅샷 기법은 비교적 긴 언마운팅 시간을 갖는다. 다행이도 NOR형 플래시메모리는 주로 코드용 메모리로 사용되어 스냅샷이 빈번하게 변화하지 않기 때문에 언마운팅 시에 스냅샷을 새로 저장할 필요가 없는 경우가 많아 이와 같이 긴 언마운팅 과정을 수행하지 않아도 된다. 그리고 언마운팅을 수행해야 하는 경우에도 일반적으로 이 작업은 운영체제의 종료 과정의 일부로 다른 모듈들의 종료 작업과 병렬로 수행될 수 있다. 또한 NAND형 플래시를 위한 스냅샷 기법의 경우 그림 12에 제시된 것보다 10배 이상 짧은 언마운팅 시간을 보였는데 이는 NAND형 플래시의 쓰기 연산의 속도가 NOR형 플래시에 비해서 15배 이상 빠르기 때문이다. NAND형 플래시를 위한 스냅샷 기법의 언마운팅 시간은 1초 이하이다.

셋째, 제안하는 스냅샷 기법과 기존의 LFS 및 FTL에서 사용된 스냅샷 기법들의 삭제 균등화 특성을 분석하였다. 그림 13(a)에 제시된 것과 같이 LFS의 경우 슈퍼 블록으로 사용되는 첫 블록이 쓰기 트레픽의 양과

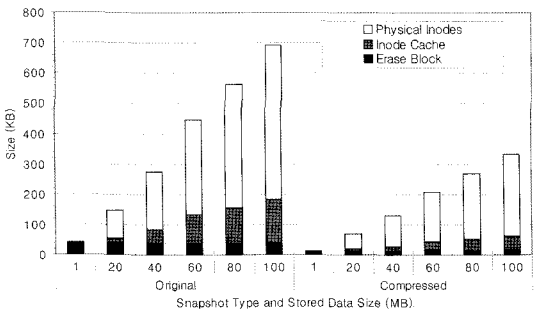


그림 10 저장된 데이타의 양과 압축기법의 사용유무에 따른 스냅샷 크기

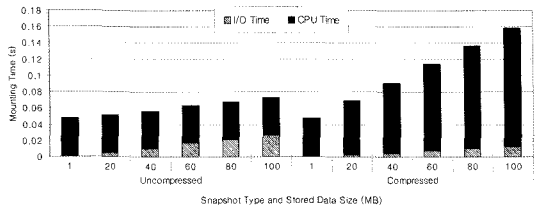


그림 11 NOR형 플래시를 위한 제안하는 스냅샷 기법의 마운팅 시간

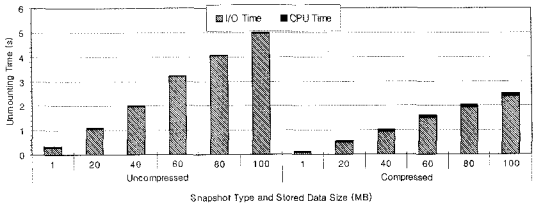
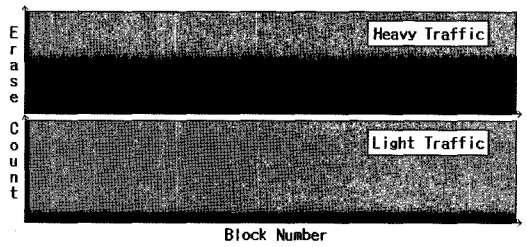
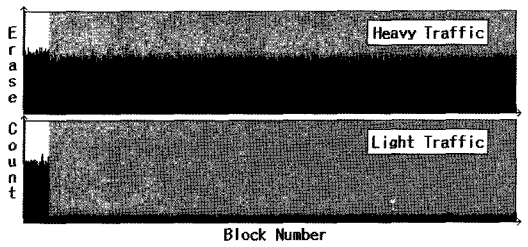


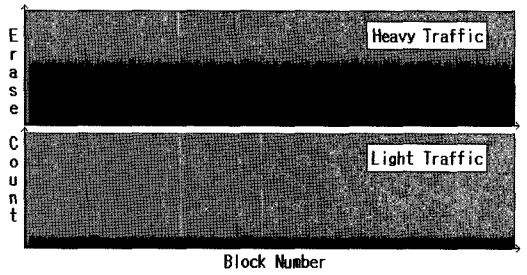
그림 12 NOR형 플래시메모리를 위한 제안하는 스냅샷 기법의 언마운팅 시간



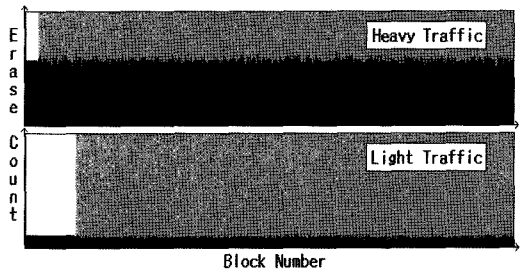
(a) LFS의 스냅샷 기법



(b) FTL의 스냅샷 기법



(c) NOR형 플래시를 위한 스냅샷 기법



(d) NAND형 플래시를 위한 스냅샷 기법

그림 13 쓰기 트레픽의 양에 따른 다양한 스냅샷 기법의 삭제 균등화 특성

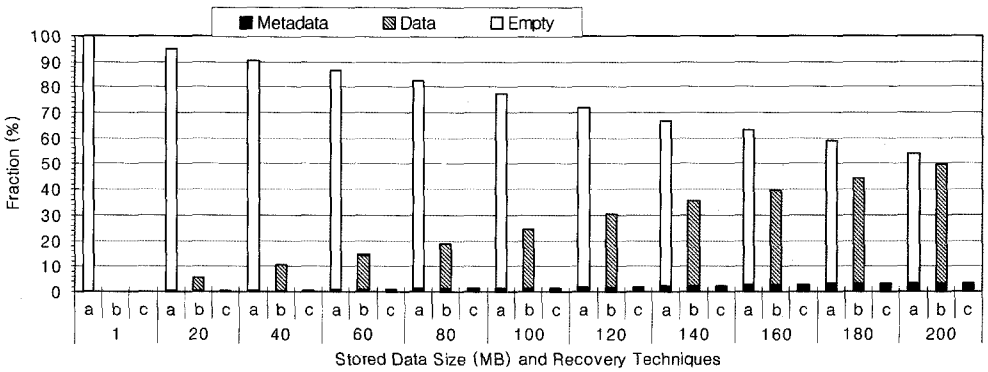


그림 14 제안하는 에러 복구 기법들을 사용한 경우 읽혀진 공간의 비율

무관하게 빠르게 삭제 한계에 도달한다. 그리고 FTL의 경우 그림 13(b)에 제시된 것과 같이 쓰기 트레픽의 양에 따라서 흰색 배경으로 표시된 스냅샷을 저장하는 영역과 회색 배경으로 표시된 데이타를 저장하는 영역간의 삭제 횟수에 차이가 생길 수 있다. 예를 들어, 이 실험에서는 비교적 작은 크기의 스냅샷 저장영역을 사용하였는데 이는 쓰기 트레픽의 양이 적은 경우 스냅샷 저장영역이 보다 빠르게 삭제 한계에 도달하는 문제를 발생시켰다. 또한 저장된 데이타의 양이 많은 경우 스냅샷의 크기가 커져 스냅샷 저장영역이 보다 빈번하게 삭제되는 특성을 보였다.

반면에 그림 13(c)에 제시된 것과 같이 NOR형 플래시를 위한 제안하는 스냅샷 기법의 경우 루트 블록으로 사용되는 첫 번째 블록의 삭제 횟수가 그 외 데이타 영역의 그것에 비해 현저하게 낮게 나타나는 특성을 보이며 전체적으로는 삭제 균등화를 보증한다. 그리고 NAND형 플래시를 위한 기법의 경우 그림 13(d)에 제시된 것과 같이 동적으로 스냅샷 저장영역의 크기를 제어함으로써 전체적으로 삭제 균등화를 보증한다. 이는 쓰기 트레픽의 양이 많은 경우 스냅샷 저장영역을 축소하고 반대의 경우 영역을 확장하여 이 영역과 데이타 영역간의 삭제 횟수를 균등화시키기 때문이다.

마지막으로 제안하는 고속 오류 복구 기법들의 성능을 평가하였다. 그림 14는 제안하는 복구 기법들을 사용한 경우 저장된 데이타의 양에 따라서 128MB 용량의 NOR형 플래시에서 읽어드린 플래시메모리의 영역의 비

율을 나타낸다. 그림 7(a)에 제시된 기법을 사용하면 데이타 영역을 읽지 않을 수 있다. 실험에서 플래시메모리에 저장된 데이타의 크기는 압축으로 인하여 원 데이타 크기의 30%정도였다. 그리고 그림 7(b)에 소개된 기법을 사용하면 빈 영역을 대부분을 읽지 않을 수 있는데 파일시스템 상에서 빈 영역이 비교적 많은 공간을 차지함을 알 수 있다. 또한 그림 7(c)에 제시된 이 두 가지 기법들을 병합한 기법의 경우 물리 메타데이타와 빈 영역의 첫 번째 페이지만을 읽어 대부분의 경우 전체 기억공간의 5% 이하만을 읽게 된다 여기서 물리 메타데이타는 inode와 디렉토리 엔트리 그리고 노드 해더 등을 지칭한다. 즉, 제안하는 복구 기법들을 사용해 오류가 발생한 이후의 마운팅 시간 중에서 입출력 시간을 95% 이상 줄일 수 있다. 이 경우에도 읽어드린 데이타를 분석해 메모리상의 메타데이타를 구성해야 하기 때문에 CPU 시간은 줄어들지 않는다. 또한 NAND형 플래시의 경우 이보다 약간 많은 영역을 읽는 특성을 보였는데 이는 NAND형 플래시가 페이지 단위의 입출력만을 지원하기 때문이다.

5. 결 론

본 논문에서는 언마운팅 시점에 메모리상의 메타데이타 스냅샷을 플래시메모리에 저장하고 이를 마운팅 시점에 빠르게 읽어드리는 고속 마운팅 기법들을 NOR형과 NAND형 플래시메모리들의 특성에 맞춰서 설계하였다. 이 기법들은 저장된 스냅샷을 일관성을 검사하는 기능을 가지고 있어서 만약 파일시스템이 안전하게 언마운팅되지 않은 경우에는 새로운 세 가지 오류 복구 기법들을 사용해 빠르게 메타데이타를 재구성한다. 실험결과 제안하는 기법들은 기존 파일시스템(JFFS2)과 비교해서 마운팅 시간을 100배가량 단축함을 보였다. 이와 같은 큰 성능 향상은 제안하는 기법이 입출력 시간과 CPU 시간을 모두 단축시킬 수 특성에 기인한다. 따라서 앞으로 제안하는 기법들은 플래시메모리 기반의 내장형시스템의 부팅시간을 단축시키는데 효율적으로 사용될 수 있다.

참 고 문 헌

[1] T. R. Bird, "Methods to Improve Bootup Time in Linux," *In Proceedings of the Ottawa Linux Symposium (OLS)*, Sony Electronics, 2004.

[2] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti, "Introduction to Flash Memory," *In Proceedings of the IEEE*, Vol. 91, No. 4, pp. 489-502, April 2003.

[3] D. Woodhouse, "JFFS: The Journaling Flash File System," *In Proceedings of the Ottawa Linux Symposium (OLS)*, RedHat Inc., 2001.

[4] Aleph One Company, "The Yet Another Flash Filing System (YAFFS)," http://www.alephl.co.uk/yaffs/.

[5] K. S. Yim, H. Bahn, and K. Koh, "A Flash Compression Layer for SmartMedia Card Systems," *IEEE Transactions on Consumer Electronics*, Vol. 50, No. 1, pp. 192-197, 2004.

[6] Samsung Electronics, "256M x 8Bit / 128M x 16Bit NAND Flash Memory," http://www.samsungelectronics.com/.

[7] L.-P. Chang and T.-W. Kuo, "An Efficient Management Scheme for Large-Scale Flash-Memory Storage Systems," *In Proceedings of the ACM Symposium on Applied Computing (SAC)*, pp. 862-868, 2004.

[8] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A Space-Efficient Flash Translation Layer for CompactFlash Systems," *IEEE Transactions on Consumer Electronics*, Vol. 48, No. 2, pp. 366-375, 2002.

[9] M. Wu and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Stroage System," *In Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 86-97, 1994.

[10] U. Vahalia, *UNIX Internals, The New Frontiers*, Ch. 8-9, Prentice Hall Inc., 1996.

[11] 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. 26-52, 1992.

[12] Samsung Electronics, "Advantages of SLC NAND Flash Memory," http://www.samsungelectronics. com/.

[13] A. Ban, "Flash File System," United State Patent, No. 5,404,485, 1995.

임 근 수
정보과학회논문지 : 시스템 및 이론
제 32 권 제 4 호 참조

김 지 홍
1986년 서울대학교 계산통계학과 학사
1988년 Univ. of Washington 컴퓨터과학과 석사. 1995년 Univ. of Washington 컴퓨터과학 및 공학과 박사. 현재 서울대학교 컴퓨터공학부 부교수. 관심분야는 컴퓨터구조, 내장형 시스템, 저전력 시스템, 멀티미디어 시스템임

고 건
정보과학회논문지 : 시스템 및 이론
제 32 권 제 4 호 참조