NAND 플래시 메모리용 파일 시스템 계층에서 프로그램의 페이지 참조 패턴을 고려한 캐싱 및 선반입 정책

박 상 오[†] · 김 경 산[††] · 김 성 조[†††]

요 약

메인메모리와 저장장치사이의 속도차이에 대한 소프트웨어적 보완 기법으로서 캐싱 및 선반입 정책이 대부분의 시스템에서 사용되고 있다. 본 논문에서는 프로그램의 페이지 참조 패턴을 고려하지 않은 리눅스 커널의 캐싱 및 선반입 정책을 개선하고자 NAND 플래시 메모리용 파일 시스템 계층에서 동작하는 Flash Cache Core Module(FCCM)을 YAFFS 상에서 설계 및 구현하였다. FCCM은 커널의 안정성과 호환성을 지원하기위해 커널과 독립적인 구조를 가지며, 플래시 메모리의 특성을 고려한 Dirty-Last 메모리 교체 기법과 페이지의 히트 여부에 따른 선반입 페이지 대기큐를 구현하였다. FCCM은 리눅스의 캐시 및 선반입 정책과 비교해 I/O 횟수와 요구되는 선반입양이 각각 최대 55%(평균 20%) 및 최대 55%(평균 24%)까지 감소했다.

키워드: NAND 플래시 메모리, 파일시스템, 캐싱, 선반입

Caching and Prefetching Policies Using Program Page Reference Patterns on a File System Layer for NAND Flash Memory

Sang Oh Park[†] · Kyung San Kim[††] · Sung Jo Kim[†††]

ABSTRACT

Caching and prefetching policies have been used in most of computer systems to compensate speed differences between primary memory and secondary storage devices. In this paper, we design and implement a Flash Cache Core Module (FCCM) on the YAFFS which operates on a file system layer for NAND flash memory. The FCCM is independent of the underlying kernel in order to support its stability and compatibility. Also, we implement the Dirty-Last memory replacement technique considering the characteristics of flash memory, and the waiting queue for pages to be prefetched according to page hit. The FCCM reduced the number of I/Os and the amount of prefetched pages by maximum 55% (20% on average) and maximum 55% (24% on average), respectively, comparing with caching and prefetching policies of Linux.

Key Words : NAND Flash Memory, File System, Caching, Prefetching

1. 서 론

폰 노이만 구조를 갖는 컴퓨터 시스템을 구성하는 CPU, 메인메모리 그리고 디스크 등 저장장치 등 세 유닛 사이의 속도 차이는 시스템의 주요한 병목지점이 된다. 폰 노이만 구조에서 기인한 이러한 병목지점들을 해결하기 위해, 컴퓨터 시스템에서는 하드웨어/소프트웨어적으로 다양한 기법들이 제안되었다[3]. 대부분의 시스템은 메인메모리와 저장장치와의 속도 차이를 보완하기 위해 페이지 캐시 및 선반입 정책을 채택하고 있다. 또한 다양한 목적의 임베디드 시스템의 저장장치로서 낸드 플래시 메모리가 널리 사용되고 있으며, 일반 디스크 기반 저장장치와는 다른 NAND 플래시 메모리의 특성을 반영한 플래시 메모리용 파일 시스템(YAFFS, JFFS2 등)의 개발도 활발히 진행되고 있다.

본 논문에서는 프로그램의 페이지 참조 패턴을 고려하지 않은 리눅스 커널의 캐싱 및 선반입 정책을 개선하고자 NAND 플래시 메모리용 파일 시스템 계층에서 동작하는 FCCM(Flash Cache Core Module)을 YAFFS 상에서 설계 및 구현하였다. 메인메모리와 저장장치의 하나인 플래시 메모리간의 I/O 횟수를 줄이고 보다 지능적으로 캐시 및 선반

※ 이 논문은 2006년도 중앙대학교 학술연구비(일반연구비) 지원에 의한 것임.
† 정 회 원 : 중앙대학교 컴퓨터공학과 박사과정(교신저자)
†† 정 회 원 : 중앙대학교 컴퓨터공학과 석사과정
††† 정 회 원 : 중앙대학교 컴퓨터공학부 교수
논문접수 : 2006년 9월 15일, 심사완료 : 2007년 6월 4일

입된 메모리 공간을 관리하기 위해 플래시 메모리용 파일 시스템 계층에서 프로그램의 네 가지 페이지 참조 패턴(순차 참조, 반복 참조, 시간지역성 참조, 확률 참조)[9]을 이용하는 새로운 캐시 모듈 관리 기법을 제안한다. FCCM은 프로그램의 이러한 페이지 참조 패턴을 동적으로 발견하고, 이를 플래시 메모리의 특성이 고려된 캐시 메모리 관리 정책과 선반입 정책에 적용한다. FCCM은 또한 캐시 페이지의 히트 여부를 고려한 차등적인 캐시 메모리 관리 기법을 도입한다. 위의 모든 작업은 커널의 안정성과 호환성을 최대한 확보하고 새로운 커널로의 이식에 소모되는 오버헤드와 커널과의 의존성을 최소화하기 위해 파일 시스템 계층상의 독립적인 모듈로서 설계되었다.

본 논문의 2장에서는 관련 연구로서 리눅스의 캐시 및 선반입 정책, 그리고 플래시 메모리와 플래시 메모리용 파일 시스템에 대해 살펴본다. 3장은 프로그램의 페이지 참조 패턴을 고려하지 않은 리눅스 커널의 캐싱 및 선반입 정책을 개선하고자 NAND 플래시 메모리용 YAFFS 파일 시스템 계층에서 동작하는 FCCM의 설계 및 구현에 대해 상세히 설명한다. 4장에서는 FCCM의 성능과 오버헤드를 측정한 결과에 대해 기술한 후, 마지막으로 5장에서 결론과 향후 연구과제에 대해 논의한다.

2. 관련 연구

2.1 리눅스 캐싱 및 선반입 정책

리눅스에서는 캐싱의 효과를 극대화하기 위해 모든 파일 접근 및 블록 장치에서 I/O가 페이지 캐시를 통해 이루어진다. 때문에 메인메모리에 여유 공간이 있는 한 캐시로 사용되는 메인메모리 공간은 항상 증가한다. 시스템의 가용 메모리가 부족해지면 메인메모리의 회수를 위해서 Second-chance LRU[23] 기법이 적용된다. 사용 중인 페이지들은 활성 LRU 리스트와 비활성 LRU 리스트로 구분되어 유지되고, 가용 메모리가 적정 수준 이하로 내려갔을 때, 시스템이 요구한 만큼의 메모리가 비활성 LRU 리스트의 끝으로부터 회수된다.

이러한 리눅스의 메모리 할당 및 회수 정책은 두 가지 측면에서 비효율적이다. 첫째, 메모리 회수 시 LRU 기법이 일괄적으로 적용된다는 점이다. 대표적인 메모리 회수 기법에는 LRU(Least Recently Used)[9], MRU(Most Recently Used)[9], LFU(Least Frequently Used)[9]등이 있다. 실제 시스템에서 수행되는 프로그램들은 매우 규칙적인 페이지 참조 패턴을 보이며, 프로그램의 특성에 따라 서로 다른 페이지 참조 패턴을 나타낸다. 예를 들어, 멀티미디어 프로그램은 데이터에 순차적으로 접근하고, 접근된 데이터는 당분간 재사용되지 않은 패턴을 보여, MRU 기법이 더 효율적이다. 많은 과학 계산 프로그램은 블록들을 일정한 간격으로 재참조하는 경우가 많기 때문에 LRU 기법이 효율적이다. 데이터베이스를 이용하는 프로그램은 페이지를 서로 다른 확률로 참조하므로 LFU 기법이 효율적이다. 하지만 리눅스에서는 시스템에서 수행되는 다양하고 복합적인 프로그램의 페이지 접근 형태를 유연하게 활용하지 못하고 메모리 회수 시 일괄적으로 LRU 기법이 적용되는 문제가 있다. 둘째, 메모리 회수 과정에서 페이지의 Clean 여부가 고려되지 않아, Dirty 페이지가 회수되는 경우, 메모리 회수 과정에서 추가적인 I/O가 요구된다.

리눅스에서는 선반입 기법을 미리 읽기(Read-Ahead)로써 구현하고 있다. 즉, 프로그램의 저장장치 접근이 순차적일 것이라는 기대에 근거하여, 현재 접근한 페이지의 다음 페이지까지 읽어 캐시 메모리에 적재한다. 프로그램의 페이지 접근이 순차적인 것으로 판단되는 경우, 미리읽기 그룹 윈도우를 설정하여 선반입하는 페이지 수를 최대 32개 페이지까지 2배씩 증가시켜 나간다. 페이지 접근이 순차적이지 않은 경우, 새로운 미리읽기 그룹 윈도우가 재설정된다. 이러한 선반입 정책은 두 가지 측면에서 비효율적이다. 첫째, 선반입에 의해 캐시 메모리 공간에는 두 종류의 페이지가 존재 하게 된다. 하나는 프로그램이 파일에 접근하여 실제 사용한 페이지이며, 다른 하나는 선반입된 페이지로서 프로그램이 아직 사용한 페이지는 아니나 미래에 사용이 기대되는 페이지이다. 프로그램의 페이지 참조 패턴에 따라 선반입된 페이지들의 용도가 다를 수 있다. 즉, 순차 패턴의 경우, 프로그램이 선반입된 페이지를 실제 사용함으로써 시스템의 성능과 응답성을 높일 수 있다. 하지만 시간 지역성의 경우, 과거에 참조한 페이지의 재 참조 가능성이 높은 반면, 선반입 된 페이지의 경우, 히트되지 않을 가능성이 높다. 확률 패턴 역시 페이지의 공간적인 위치가 접근 가능성과 무관한 임의 접근이기 때문에 선반입된 페이지들이 히트되지 않을 가능성이 높다. 실제 통계에 의하면 프로그램에 의해 선반입된 페이지의 60% 이상[13]이 실제로 사용되지 않은 채 회수된다고 한다. 둘째, 모든 프로그램에 동일한 미리읽기 윈도우 크기를 적용하는 것이다. 시간지역성 패턴과 확률 패턴의 경우, 순차 패턴과 반복 패턴에 비해 선반입된 페이지들이 미래에 사용될 가능성이 적다. 때문에 프로그램의 페이지 참조 패턴에 따라 미리읽기 윈도우 크기를 차등 적용하는 것이 보다 효율적일 것이다. 리눅스 선반입 정책의 이러한 비효율성을 개선하기 위해, 본 논문에서는 새로운 선반입 정책을 제시한다.

본 논문에서 활용한 프로그램의 페이지 참조 패턴[9]은 페이지들이 한번 참조된 후 재참조 되지 않는 순차 참조(Sequential Reference), 모든 페이지들이 일정한 간격으로 재참조되는 반복 참조(Looping Reference), 최근에 참조된 페이지일수록 더 가까운 미래에 참조되는 시간지역성 참조(Temporally-Clustered Reference), 페이지들이 고유한 확률로 참조되는 확률 참조(Probabilistic Reference)와 같이 4가지로 분류될 수 있다. 기존 기법[9]은 사용자가 제공하는 정보 없이 프로그램의 블록 속성과 미래 참조 거리간의 관계를 기반으로 프로그램의 페이지 참조 패턴을 동적으로 발견하며, 기존 연구들에 의해 프로그램의 일반적인 페이지 참조 패턴으로 알려진 순차 참조, 반복 참조, 시간지역성 참조,

그리고 확률 참조 등을 모두 발견할 수 있으며, 페이지 참조 패턴이 실행 중에 변경된 것도 찾아낼 수 있다.

2.2 NAND 플래시 메모리

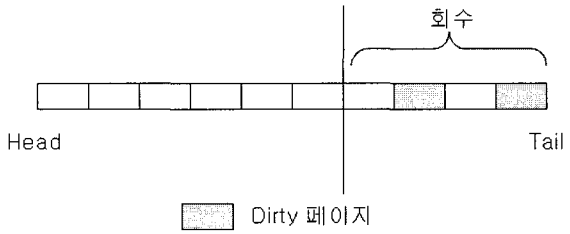
플래시 메모리는 높은 신뢰성과 집적도를 가진 반도체 형태의 저장매체로서 비용이 기타 저장매체에 비해 상대적으로 저렴하여 내장형 기기, 핸드 헬드 기기, 무선 또는 이동형 기기, 멀티미디어 기기 등에서 저장장치로 널리 이용되고 있다. 플래시 메모리는 전원이 끊어져도 데이터가 지워지지 않은 비휘발성 메모리로서 블록 단위로 내용을 지울 수 있다. 플래시 메모리는 EEPROM(Electrically Erasable Programmable ROM)의 한 형태이며, 바이트 단위로 삭제 및 수정이 가능한 EEPROM과는 달리 블록 단위로 수정되기 때문에 속도가 빠르다. 하지만 기존 데이터가 존재하는 위치에 데이터를 기록하려면 해당 블록을 지우고, 쓰기 작업을 해야 한다. 그리고 플래시 메모리의 하드웨어 특성상 플래시 메모리의 블록을 지우는 횟수에 제한이 있어 특정 블록에 대해 쓰기가 반복되면 플래시 메모리의 수명이 단축된다.

일반적으로 사용되는 플래시 메모리는 크게 NOR형과 NAND형 두 가지로 구분된다. 플래시 메모리에서 삭제 연산은 쓰기 연산의 단위인 페이지보다 큰 블록 단위로만 이루어진다. 따라서 플래시 메모리의 블록 내의 한 비트만을 수정한다고 해도 해당 블록 전체를 지우는 작업이 필요할 수도 있다. 또한, 플래시 메모리의 수명은 삭제주기(Erase Cycle)로 결정되는데 특정 블록에 지우기 작업이 집중되어 해당 블록만이 수명이 다하는 것을 방지하기 위하여, 삭제가 전체 블록에 고루 분산되도록 관리되어야 한다. NAND형 플래시 메모리는 내장형 기기에 적합한 대용량 저장장치에 대한 수요가 증가하면서 최근 각광받고 있다. 디스크와 구분되는 플래시 메모리의 하드웨어적인 특성들 때문에 FAT, EXT2/3, NTFS와 같은 디스크 기반의 파일 시스템과는 다른 파일 시스템이 요구된다. 플래시 메모리를 저장매체로 사용하기 위해서는 크게 플래시 변환 계층(Flash Translation Layer: FTL)과 같은 미들웨어를 사용하는 방법과 전용 파일 시스템을 사용하는 방법이 있다. 이중 FTL은 플래시 메모리를 RTDB(Real Time Database)와 같은 특수한 시스템에서 메인메모리로서 활용하기 위해 제안되었다[18]. 플래시 메모리를 저장장치로 사용하는 경우, FTL의 추가적인 오버헤드 때문에 FTL을 사용하는 것이 플래시 메모리용 파일 시스템을 사용하는 것에 비해 비효율적일 수 있다. 뿐만 아니라 PCMCIA 카드에 기반을 둔 형태를 제외한 FTL과 관련된 기술은 이미 특허가 등록이 되어있어 사용에 있어서 법률적인 제약이 따른다. 때문에 플래시 메모리용 파일 시스템으로서 JFFS2(Journaling Flash File System 2)와 YAFFS(Yet Another Flash File System) 등이 개발되었다[11]. YAFFS는 JFFS2를 NAND 플래시 메모리에 적합하도록 수정하려는 시도에서 시작되어 독립적인 NAND 플래시 메모리용 파일 시스템으로 발전하였다. YAFFS는 리눅스뿐 아니라 uClinux, Windows CE 등에서도 동작한다. 현재 YAFFS는 오픈 소스 형태로 계속 개발중에 있으며, 성능과 안정성이 계속 향상되고 있다[13]. 본 연구 또한 YAFFS를 수정하여 YAFFS의 플래시 메모리 파일 시스템 계층에서 프로그램의 페이지 참조 패턴에 따른 캐싱 및 선반입 정책을 구현하였다.

2.3 기존 기법

2.3.1 CFLRU(Clean-First LRU)

기존의 LRU를 기반으로 NAND 플래시 메모리에 대한 지우기와 쓰기 연산 횟수를 줄임으로써 에너지 소비를 최소화하기 위하여 CFLRU[20]가 개발 되었다. CFLRU 정책은 우선 가장 오랫동안 사용되지 않은 클린 페이지를 희수하고, 다음 가장 오랫동안 사용되지 않은 더티 페이지를 희수하는 방법이다.



(그림 1) 페이지 접근 순서에 의한 페이지 희수

그러나, CFLRU에서 더티 페이지가 희수되면, NAND 플래시 메모리의 특성상 쓰기 연산은 많은 비용이 요구된다. (그림 1)에서와 같은 페이지 접근 순서 리스트에서 하위 4개의 페이지를 희수한다고 하자. 이때 첫 번째와 세 번째 페이지의 경우 해제 이전에 저장장치 쓰기 연산이 수행되어야 한다. 시스템의 메모리 자원이 충분치 않은 시점에 저장장치 쓰기 연산까지 수행하게 되는 것은 시스템의 부하를 더욱 가중시키게 된다. 더욱이 저장장치가 NAND 플래시 메모리인 경우, 이러한 문제는 더욱 심각하게 된다. 이것은 NAND 플래시 메모리 특성상 희수될 해당 페이지 위에 바로 덮어 쓰기를 할 수가 없고, 반드시 블록 단위 삭제 연산이 수행된 후에야 페이지 단위의 쓰기 연산을 수행할 수 있기 때문이다. 또한, 삭제 연산은 단일 페이지 쓰기 연산보다 10배 가량의 시간이 소모되기 때문에 NAND 플래시 메모리 파일 시스템의 경우, 위와 같은 방식의 캐시 메모리 희수로 인하여 쓰기 연산은 시스템 응답성이 크게 저해되는 요인이 될 수 있다. 본 논문에서는 NAND 플래시 메모리에도 효율적으로 적용될 수 있도록 하는 새로운 기법인 Dirty-Last 기법을 제시하여 쓰기연산을 최대한 줄이도록 한다.

2.3.2 FAB(Flash-Aware Buffer manegement)

FAB[21]는 플래시 메모리에 대한 삭제와 쓰기 연산 횟수를 줄이기 위해서, 기존 페이지 단위 LRU 대신 블록 내에서의 페이지 사용량에 따라 희생자를 선택하는 블록 단위

LRU 정책이다. FAB는 버퍼가 가득 차게 되면, FAB 버퍼의 모든 블록 중에서 희생시킬 블록을 한 개 선택한다. 그리고 나서 선택된 블록이 가지고 있는 모든 페이지를 플래시 메모리에 쓴 후, 그 블록을 희생시킨다. FAB 정책은 우선 버퍼에 가장 많은 페이지를 가지고 있는 블록을 선택한다. 만일 이렇게 선택된 블록이 여러 개라면, LRU Tail 쪽의 블록이 선택된다. 그러나 FAB는 대용량 파일을 위해 개발되었으며, 한 페이지만 수정되어도 블록 전체가 다시 써져야 한다. FAB에서 작은 파일이 자주 사용될 경우, 큰 파일보다 빈번히 수정되게 되고[22], 이로 인하여 NAND 플래시 메모리에 대한 삭제와 쓰기 연산이 빈번히 발생되어 성능이 떨어지는 문제가 있다. 따라서, FAB의 정책은 멀티미디어 파일과 같은 큰 파일을 주로 이용하는 기기에 보다 적합하다. 본 논문에서는 파일의 특성에 따라 다른 캐시 정책들을 적용함으로써 이러한 문제점을 해결한다.

3. FCCM(Flash Cache Core Module) 설계 및 구현

본 장은 리눅스의 캐싱 및 선반입 정책이 프로그램의 다양한 페이지 참조 패턴을 반영하지 못한다는 것과 페이지 캐시가 히트 여부에 상관없이 동일하게 관리되고 있는 비효율성을 개선하고자 구현된 FCCM에 대해 설명한다. 또한 쓰기 연산 시 블록 단위의 삭제 연산을 요구하는 플래시 메모리의 특성에 따라 I/O 도중 가능한 한 쓰기 연산 횟수를 줄이고자 하였다. 효율적인 캐싱 및 선반입 정책을 개발하기 위해 프로그램 수행 시 페이지 참조 패턴을 동적으로 발견하는 기법을 구현하고, 이를 캐싱 및 선반입 관리 정책에 이용하였다. 프로그램의 페이지 참조 패턴 발견을 위해서 기 발표된 알고리즘[9]을 이용하였다. FCCM은 <표 1>과 같이 기능상 크게 세 개의 모듈로 구성된다. 패턴 탐색(Pattern Detection) 모듈은 프로그램의 실행 중에 페이지 접근 정보를 분석하여 프로그램의 페이지 참조 패턴을 찾아낸다. 캐시 코어(Cache Core) 모듈은 FCCM이 파일 시스템 계층에서 캐시 기능을 수행할 수 있도록 지원한다. 이 모듈에는 페이지 캐시 기수 트리(Page Cache Radix Tree), 메모리 회수 리스트(DLLRU, DLMRU, DLLFU), 선반입 대기 리스트(Prefetch Waiting List) 등이 포함된다. 마지막으로 리눅스와 연동하여 FCCM이 동작하기 위한 연결 모듈이다. 이 모듈은 주로 리눅스 커널의 메모리 할당/해제/관리를 담당한다.

〈표 1〉 FCCM 모듈 구성

모듈	기능
패턴 탐색 모듈 (Pattern Detection Module)	일정 주기 동안 참조된 페이지들의 속성 정보와 미래 참조 주기를 이용하여 프로그램의 페이지 참조 패턴 탐색
캐시 핵심 모듈 (Cache Core Module)	캐시기능이 수행되기 위한 필요 모듈로서 페이지 캐시 기수 트리, 메모리 회수 리스트, 선반입 대기 리스트 등을 지움
연결 모듈	메모리 할당/해제/관리 루틴

3.1 프로그램 패턴 탐색 모듈

본 절은 프로그램의 실행 중에 페이지 참조 패턴을 자동으로 발견할 수 있는 FCCM의 패턴 탐색 알고리즘에 대해 기술한다. 페이지 참조 패턴의 특성은 페이지 속성(Page Attribute)과 미래 참조 거리(Forward Distance)를 이용하여 분석된다. 페이지 속성은 페이지의 과거 참조로부터 얻어지며, 캐시 관리 정책에서 유용한 정보로서 활용된다. 페이지의 과거 참조 거리(Backward Distance), 참조 횟수(Frequency), 참조 간 간격(Inter-Reference Gap), k 번째 과거 참조 거리(k^{th} Backward Distance)등은 페이지 속성의 예이다. FCCM은 과거 참조 거리와 참조 횟수라는 두 개의 페이지 속성을 이용하여 페이지 참조 패턴을 발견한다. 페이지의 과거 참조 거리는 현재 시간과 그 페이지가 마지막으로 참조된 시간 간의 차이이다. 참조 횟수는 페이지가 현재 시간까지 참조된 횟수이다. 페이지의 미래 참조거리는 현재 시간과 그 페이지가 미래에 다시 참조될 시간 간의 차이이다. 최적의 페이지 교체 정책은 미래 참조 거리가 가장 큰 페이지를 교체하며, 일반적으로 페이지의 미래 참조거리는 캐시에서 페이지의 가치를 결정한다.

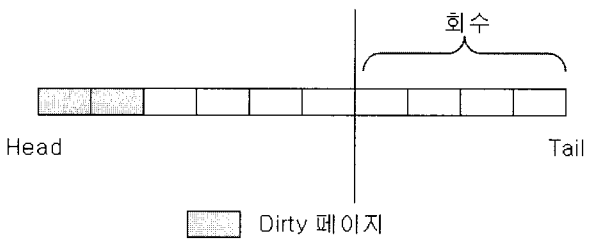
3.2 FCCM의 캐시 정책

본 절은 발견된 프로그램의 패턴 특성과 NAND 플래시 메모리의 특성을 고려하여 설계된 FCCM의 캐시 메모리 관리 정책에 대해 기술한다.

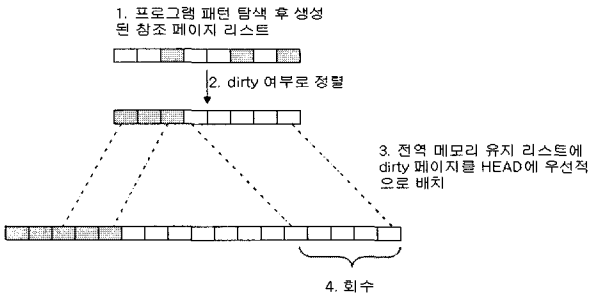
3.2.1 Dirty-Last 정책

시스템의 가용 메모리가 부족해져 페이지를 회수하게 될 때, 리눅스 커널의 PG_dirty 필드가 설정된 dirty 페이지는 저장장치에 쓰기 연산이 수행된 후에야 회수될 수 있다. FCCM은 2.3.1절에서 언급한 CFLRU[20]의 NAND 플래시 메모리 상에서의 비효율성을 개선하기 위해 기존 기법[20]을 변형하였다. 즉 페이지 회수 리스트를 유지할 때 Dirty 페이지는 가능한 한 늦게 회수 되도록 (그림 2)와 같이 배치함으로써 회수 과정에서 추가적인 NAND 플래시 메모리 쓰기 연산을 피할 수 있도록 하였다. 또한 기존 기법[20]에서 LRU에만 적용된 것을 MRU 및 LFU에도 적용을 하여 다양한 파일 특성에 따라 캐시 정책이 적용되게 하였다.

이 방법은 리스트 유지 과정에서 페이지의 Dirty 여부를 확인해야 하지만, 외부 I/O가 시스템의 응답성을 저해하는



(그림 2) Dirty-Last 페이지 회수

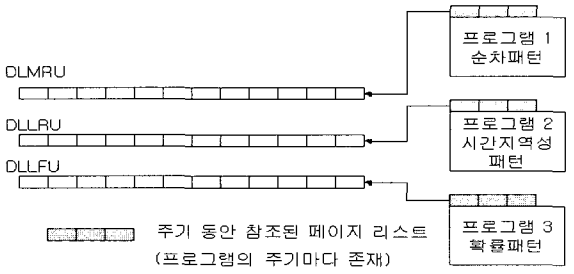


(그림 3) FCCM의 Dirty-Last 기반 메모리 회수

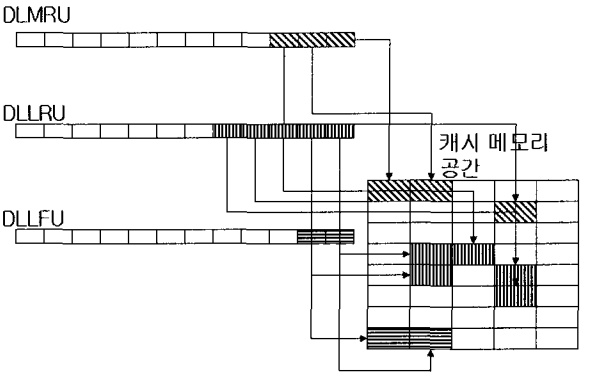
가장 큰 병목 지점임을 고려할 때 의미 있는 시도이다. 프로그램의 패턴 탐색 과정에서 FCCM의 주기 탐색 쓰레드는 주기 동안 참조된 메모리 리스트를 유지한다. 본 논문에서 제시한 Dirty-Last 기법은 Dirty 페이지가 리스트의 Head 쪽에 배치되도록 정렬한다. 정렬 후 리스트의 Head에는 Dirty페이지들의 끝을 가리킨다. 정렬된 리스트는 전역적으로 존재하는 DLLRU/DLMRU/DLLFU 리스트(3.2.2절 참조)에 추가 되는데, 이 과정에서도 Dirty 페이지를 가리키는 노드들이 회수 우선순위가 낮은 Head쪽에 배치된다. (그림 3)은 FCCM에서 Dirty 기반으로 한 메모리 회수 과정을 보여주고 있다.

3.2.2 메모리 관리 정책

FCCM은 프로그램의 페이지 참조 패턴에 따라 다른 메모리 교체 정책을 적용한다. 순차 패턴과 반복 패턴을 보이는 프로그램은 MRU 정책이 적합하고, 시간적 지역성을 보이는 프로그램은 LRU 기법이 적합하여, 확률 참조 패턴 프로그램은 LFU 기법이 적합하다[3]. 이러한 교체 정책에 3.2.1절에서 기술한 Dirty-Last 기법을 추가해 FCCM은 DLLRU, DLLFU, DLMRU 등 세 가지 메모리 유지 리스트를 관리한다. 프로그램의 페이지 참조 패턴 탐색 후 발견된 결과에 따라 (그림 4)와 같이 해당되는 메모리 유지 리스트에 패턴 탐색 주기 동안 참조된 메모리 리스트를 삽입한다. 단, 패턴 탐색이 실패한 경우에는 LRU 리스트를 이용한다. 메모리 유지 리스트는 각자의 정책에 맞는 우선순위로 정렬된다. DLMRU 리스트와 DLLFU 리스트에는 Tail에 삽입되며, DLLRU 리스트에는 Head에 삽입된다. 앞서 설명하였듯이 쓰기 연산이 수행된 Dirty 페이지들은 리스트의 Head에 우선적으로 배치된다.



(그림 4) 프로그램 페이지 참조 패턴 탐색 주기 후 주기 동안 참조된 리스트를 전역 리스트에 삽입



(그림 5) 메모리 유지 리스트에서 메모리 회수

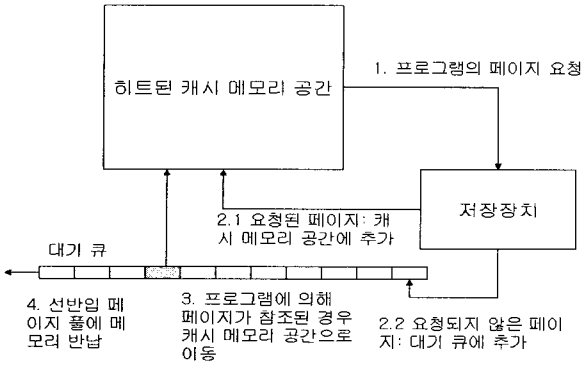
메모리 유지 리스트(DLLRU/DLLFU/DLMRU)들의 노드는 캐시 메모리 공간의 페이지를 가리키는 페이지 포인터를 포함하고 있다. 시스템의 메모리가 부족해지면 FCCM의 메모리 관리자는 메모리 유지 리스트에서 메모리 회수를 시도한다. 회수할 메모리 크기는 시스템의 상황과 설정에 따라 다르며, 시스템이 유지해야할 최소 메모리 수치만큼이 확보될 때까지 반복해서 메모리 회수를 시도한다. (그림 5)는 메모리 유지 리스트에서 메모리 해제 매커니즘을 나타낸다. 시스템이 10개의 페이지 메모리 회수를 요청했다고 가정했을 때, 각 리스트에서 가장 낮은 우선순위를 가지는 노드가 가리키는 페이지의 메모리 회수를 시도한다. FCCM은 전역적으로 프로그램들의 패턴 정보를 유지 하고 있고, 회수 시 패턴의 비율에 따라 해당 리스트에서 메모리를 회수 한다. 시스템에 존재하는 프로그램의 패턴이 각각 순차 패턴 2개, 반복 패턴 1개, 확률 패턴 2개, 시간 지역성 3개, 탐색 실패 2개 인 경우 등 10개의 메모리 페이지에 대한 해제가 요청되었을 때 DLMRU 리스트에서 3개(순차패턴 + 반복 패턴), DLLFU 리스트에서 2개(확률 패턴), DLLRU 리스트에서 5개(시간 지역성 + 탐색 실패)가 회수된다.

3.3 선반입 정책

본 절은 프로그램의 페이지 참조 패턴에 따라 적용되는 선반입 정책과 리눅스와 차별화된 선반입 대기 큐에 대해 기술한다. 2장에서 언급한 바와 같이 리눅스는 선반입 기법으로서 미리 읽기(Read-Ahead)를 채택하고 있다.

3.3.1 선반입 대기 큐

캐시 메모리 공간에 존재하는 페이지(프로그램이 실제 사용한 페이지)와 히트되지 않은 페이지(선 반입된 페이지)에 가중치를 부여하였다. 히트된 페이지는 캐시 메모리 공간에 추가되며, 히트되지 않은 페이지는 고정된 크기의 선반입 대기 큐에 삽입된다. 대기 큐에 삽입된 선반입 페이지들은 선반입이 진행됨에 따라 점차적으로 큐의 헤드로 이동하게 된다. 페이지가 큐에 존재하는 동안 프로그램에 의해 참조될 경우, 해당 페이지는 캐시 메모리 공간으로 이동하게 된



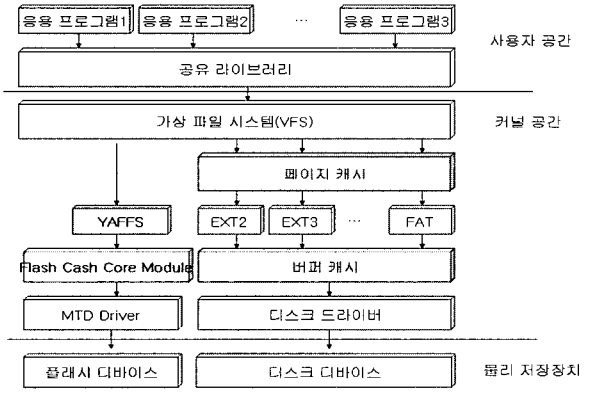
(그림 6) FCCM 선반입 정책

다. 참조되지 않고 큐에서 밀려난 페이지는 더 이상 메모리에 유지되지 않는다. (그림 6)은 FCCM의 이러한 선반입 정책에 대해 설명하고 있다.

순차 패턴 프로그램의 경우, 선반입 된 페이지가 히트될 가능성이 가장 높기 때문에 미리 읽기 윈도우 크기를 2.1절에서 설명한 리눅스의 미리 읽기 윈도우와 같게 하였다. 반복 패턴 프로그램과 확률 패턴 프로그램의 경우, 순차 패턴 윈도우 크기의 1/2로 결정하였고, 시간 지역성 패턴의 경우, 선반입 페이지의 히트 가능성이 가장 낮아, 윈도우 크기를 1/4로 하였다.

3.4 커널 독립적 설계

본 절은 리눅스 커널과의 독립성을 고려하여 설계된 FCCM의 구조에 대해 기술한다. (그림 7)은 FCCM의 전체 구조를 나타낸다. FCCM은 커널의 안정성과 호환성을 최대한 보장하도록 커널과의 의존성을 최소화하도록 설계되었다. 리눅스 커널은 (그림 7)과 같이 하드웨어 저장 장치로의 접근을 VFS, 파일 시스템, 장치 드라이버, 물리 저장장치와 같이 4단계의 계층으로 추상화될 수 있다. 이러한 추상화를 통하여 상위 계층에서는 하위 계층의 특성을 고려하지 않고 공통된 인터페이스를 통하여 하위계층을 제어할 수 있다. 디스크와 같은 일반적인 블록 장치들은 읽기 연산 수행 시 장치에서 블록을 읽어 들여 버퍼캐시에 저장하고, VFS 계층에서는 이를 하나의 페이지 단위 캐시로 추상화하여 관리한다. 하지만 오리지널 YAFFS 파일 시스템은 읽기 연산 시 버퍼 캐시 사용 없이 페이지 캐시만을 사용한다. 수정된 YAFFS 파일 시스템 계층에서 동작하는 FCCM은 리눅스의 페이지 캐시를 사용하지 않는다.



(그림 7) 커널 독립적 FCCM 설계

4. 성능 평가

본 장에서는 논문에서 구현한 FCCM(Flash Cache Core Module)의 성능과 오버헤드를 측정한 결과에 대해 기술한다. FCCM의 성능은 두 가지 측면에서 비교될 수 있다. 첫째, 얼마나 성공적으로 프로그램의 패턴을 검색하는가이다. 프로그램의 패턴에 따라 적합한 캐시 메모리 관리 정책 및 선반입 정책이 적용되므로 발견된 프로그램 패턴의 정확도가 높을수록 캐시의 히트율이 높아질 수 있다. 둘째, FCCM이 캐시 히트율을 얼마나 향상시킬 수 있는가, 즉 시스템의 I/O 횟수를 얼마나 줄일 수 있는가 이다. 성능 테스트는 리눅스 커널 2.6.10에서 S3C2440 MCU기반의 OMAP5912OSK 보드에서 수행되었다(<표 2> 참조). 성능 비교는 YAFFS에서 페이지 캐시와 FCCM에 대해 수행되었으며, YAFFS 파일시스템을 기반으로 하기 때문에 파일시스템에서 수행되는 기타 I/O 작업들은 제외하고 YAFFS 파일 시스템의 정규파일의 I/O 작업에 대해 횟수의 증감 비율과 선반입되는 메모리 량을 측정하였다.

〈표 2〉 테스트 환경

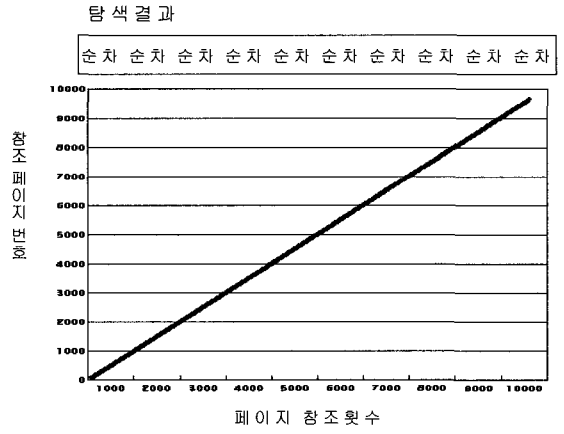
구분	항목	비고
Hardware	Board	OMAP5912
	RAM	32MB
	Flash Memory	K9F1208 64MB
Software	OS	Linux
	Version	2.6.10
	File System	YAFFS

4.1 프로그램 패턴 탐색

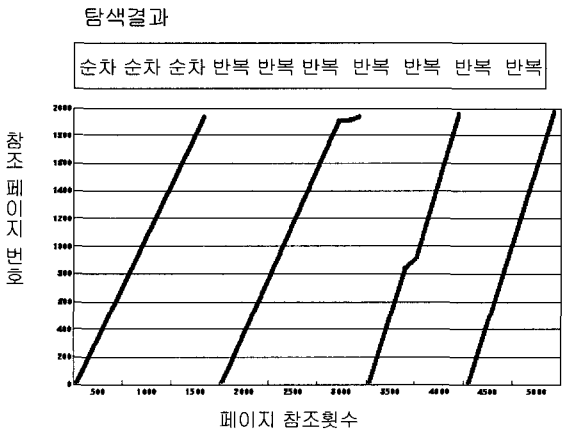
본 절에서는 FCCM을 이용하여 WC, CSCOPE, AB(Andrew file System Benchmark) 등을 대상으로 실제 리눅스 프로그램들의 패턴을 탐색한 결과를 기술한다. WC는 입력 파일의 문자 수, 단어 수, 라인 수 등을 보여주는 프로그램이고, CSCOPE는 C 소스 코드 탐색 도구로서 사용자 질의에 대해 'CSCOPE.out' 파일을 읽어서 응답한다. AB는 Andrew file System 개발 시 사용된 벤치마크 프로그램으로 여러 파일에 대한 생성, 복사, 검색, 컴파일 등을 수행한다.

FCCM을 이용해 위에 언급한 프로그램들의 페이지 참조 패턴을 탐색한 결과, 실제 프로그램의 사용 패턴과 탐색된 패턴이 잘 일치함을 알 수 있었다. WC 프로그램의 경우, FCCM은 순차 패턴으로 일관되게 검색한다(그림 8 참조). WC 프로그램은 입력파일의 문자, 단어, 라인 수를 검색할 때 순차적으로 읽는 특성을 가지기 때문이다. CSCOPE 프

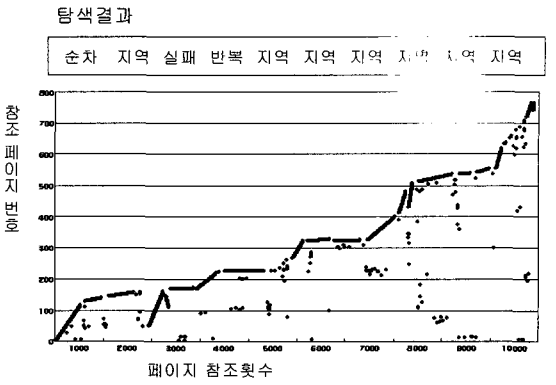
로그램의 경우, 참조 횟수가 1500 이하일 때는 순차 패턴으로 탐색되고, 그 이후에는 반복 참조로 탐색된다. 이것은 CSCOPE 프로그램이 사용자 질의에 대해 하나의 CSCOPE.out 파일을 반복적으로 참조하여 응답하는 특성 때문이다. 즉, CSCOPE.out 파일을 탐색하는 동안에는 페이지를 순차적으로 접근하게 되고, 파일을 반복해서 참조하므로 (그림 9)와 같은 그래프가 도출되었다. AB 벤치마크 프로그램의 경우, 참조 횟수가 증가함에 따라 다양한 패턴이 발견되었다. 초기에는 어느 정도 순차적으로 I/O를 수행하기 때문에, 참조 횟수 1000 정도까지는 순차 패턴으로 탐색되었다. 그 반면 페이지 참조 패턴이 변화하는 참조 횟수 2500 정도에서는 패턴 탐색에 실패하였으며, 3000 정도에는 반복 패턴이 탐색되었다(그림 10). AB 벤치마크 프로그램의 페이지 참조 패턴을 분석하면, 전체적으로 참조되는 페이지들이 시간이 지남에 따라 서서히 증가하며, 최근에 참조된 페이지들이 재 참조되는 경향을 가짐을 알 수 있다. 10000번의 참조 동안 800개 가량의 페이지들을 참조되었으며, 이는 AB 프로그램의 페이지 참조 패턴이 시간 지역성을 가짐을 보여주며, 구현된 기법이 프로그램의 페이지 참조 패턴 변화를 정확히 발견함을 의미한다.



(그림 8) WC 프로그램 패턴 탐색 결과



(그림 9) CSCOPE 프로그램 패턴 탐색 결과



(그림 10) AB 프로그램 패턴 탐색 결과

4.2 메모리 사용량 및 I/O 횟수

본 절에서는 FCCM(Flash-Cache Core Module)을 리눅스의 메모리 페이지 교체 정책에 적용하여 다양한 프로그램의 패턴에 따른 I/O 횟수와 선반입되는 메모리 사용량의 증감을 측정한 결과에 대해 기술한다. 리눅스의 페이지 캐시 모듈은 프로그램의 데이터 참조 시(페이지 크기 4KB) 요청된 페이지가 이미 캐시에 있으면 추가적인 I/O없이 캐시를 통해 요청을 처리하고, 캐시에 없으면 요청된 페이지 블록들을 저장장치에서 읽어 들인 후 처리한다.

각 응용에 대해 선반입된 메모리 크기와 입출력 횟수를 측정하기 위해서, WC는 약 12MB 크기의 텍스트 파일을 입력받아 단어 수를 카운트했고, CSCOPE는 약 34MB 크기의 인덱스 파일에 대해 질의를 4번하였다.

〈표 3〉 I/O 수행 시 선반입된 데이터양(단위/MB)

프로그램	물리적 가용 메모리양	페이지캐시+YAFFS	FCCM+YAFFS
WC	16MB	11.97	11.97
	32MB	11.97	11.97
AB	16MB	40.8	34.68
	32MB	40.8	34.68
CSC OPE	16MB	134.32	84.46
	32MB	134.21	60.24

<표 3>은 기존 페이지 캐시를 사용하는 YAFFS의 선반입 정책과 FCCM를 이용하는 변형된 YAFFS에서의 선반입 정책에서 선반입된 메모리 사용량을 각 프로그램 별로 비교한 결과이다. 선반입된 데이터양을 측정할 때, 시스템의 물리적인 가용 메모리는 16MB, 32MB로 변경시키면서 실험한 결과, 캐시의 히트율이 증가하여 선반입된 데이터양이 최대 55%(CSCOPE에서 메모리가 32MB인 경우), 평균 24% 줄어들음으로써 메모리 자원이 보다 효율적으로 관리됨을 알 수 있다. 이것은 프로그램의 패턴에 따른 차등적인 캐시 메모리 관리 및 선반입 정책이 효과적으로 사용될 수 있음을 뜻한다.

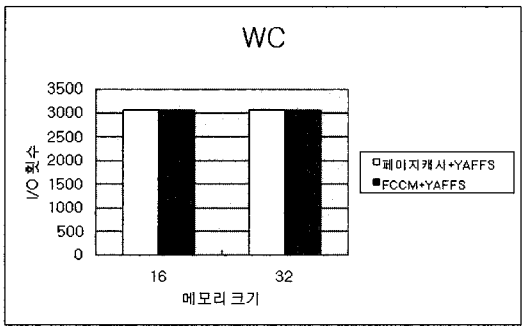
(그림 11~13)은 WC, AB, CSCOPE 프로그램들에 대해 기존 페이지 캐시를 사용하는 YAFFS의 Second-Chance LRU 기법과 FCCM의 교체 기법을 적용했을 때 NAND 플

래시 메모리 입출력 횟수를 비교한 결과이다. 그래프의 X축과 Y축은 각각 시스템의 가용 메모리크기와 I/O 횟수를 나타낸다. 시스템의 가용메모리를 16MB, 32MB로 변경시키면서 실험한 결과, 모든 경우에서 FCCM를 이용하는 변형된 YAFFS의 캐시 정책이 기존 페이지 캐시를 사용하는 YAFFS의 캐시 정책에 비해 I/O 비율을 최대 55%(CSCOPE에서 메모리가 32MB인 경우), 평균 20% 가량 줄일 수 있었다.

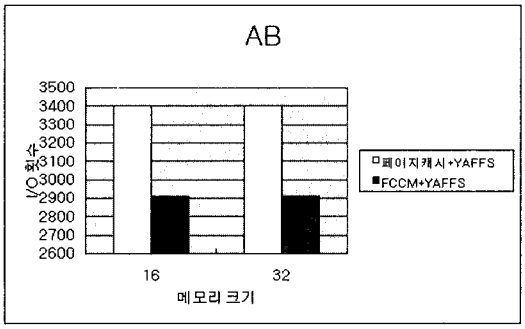
(그림 11)은 순차적인 페이지 참조 패턴을 가지는 WC 프로그램에 대한 입출력 횟수를 나타낸다. WC 프로그램은 순차적이기 때문에 캐시 메모리가 히트되는 경우가 없어 리눅스 캐시와 FCCM간의 성능 차이가 없다. 선반입된 메모리 사용량도 순차 패턴 프로그램의 경우 FCCM도 리눅스의 선반입 정책이 그대로 적용되기 때문에, 두 기법 간에 차이가 없다.

(그림 12)는 시간 지역성 패턴을 가지는 AB 프로그램에 대한 테스트 결과이다. AB의 I/O 횟수는 시스템 가용 메모리가 평균 12% 가량 감소하였다. 시간 지역성 패턴의 프로그램에 대해 FCCM은 읽기 연산 수행 시 리눅스의 선반입 정책에 비해 최대 1/4만큼의 페이지만을 선반입 한다. 이와 같이 페이지 선반입에 메모리 자원을 1/4 정도만 소모하고도 I/O 횟수가 평균 12% 감소한 것은 캐시 메모리에서의 히트율이 향상되었음을 의미한다.

<표 4>는 AB에서 발생하는 읽기, 쓰기에 대해 구분하여 I/O 횟수를 측정한 결과이다. AB의 읽기에서 15%, 쓰기에서는 10%를 줄임으로써 NAND 플래시 메모리의 쓰기 연산에서 발생되는 비용을 줄일 수 있었다.



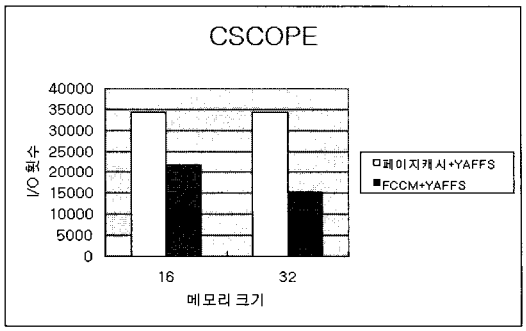
(그림 11) WC 프로그램의 I/O 횟수



(그림 12) AB 프로그램의 I/O 횟수

〈표 4〉 AB의 I/O 횟수

프로그램	연산	요청 횟수	페이지캐시+YAFFS	FCCM+YAFFS
AB	읽기	26783	10444	8878
	쓰기	15119	8243	7489



(그림 13) CSCOPE 프로그램의 I/O 횟수

(그림 13)은 반복 패턴을 보이는 CSCOPE 프로그램에 대한 성능 측정 결과이다. CSCOPE의 I/O 횟수는 시스템 가용 메모리가 32MB인 경우, 최대 55%(평균 46%) 가량 감소하였다. FCCM에서 선반입에 소모되는 메모리 자원이 기존 페이지 캐시를 사용하는 YAFFS의 선반입 정책에 비해 최대 1/2 정도만 소모하고도 I/O 횟수가 평균 46% 감소한 것은 캐시 메모리에서의 히트율이 향상되었음을 의미한다.

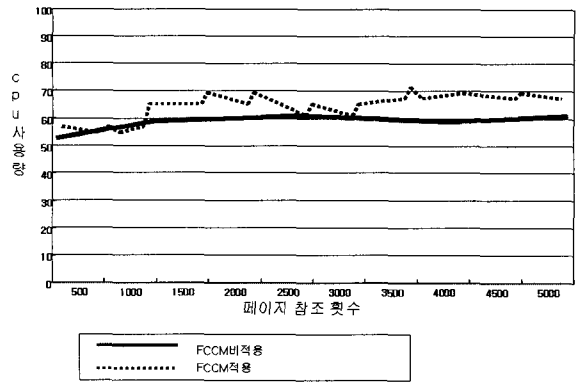
〈표 5〉 I/O 수행 시 소요시간(단위/초)

프로그램	물리적 가용 메모리양	페이지캐시+YAFFS	FCCM+YAFFS
WC	16MB	27.65	28.54
	32MB	27.72	28.56
AB	16MB	632.53	552.54
	32MB	632.28	551.75
CSC OPE	16MB	282.65	238.97
	32MB	282.34	195.16

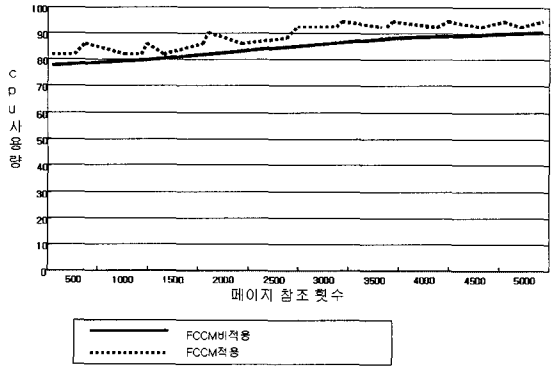
<표 5>는 각 프로그램에서 I/O 수행시 소요된 시간을 측정한 결과이다. 순차적 형식을 보이는 WC에서는 FCCM의 오버헤드(4.3절 참조) 때문에 시간이 더 걸렸다. AB와 CSCOPE는 각각 최대 13%, 31%(평균 13%, 23%) 감소하였다.

4.3 FCCM 오버헤드

본 절에서는 FCCM이 동작하기 위해 부가적으로 소모되는 시스템 부하의 원인 3가지를 기술하고 각각의 측정한 결과에 대해 기술한다. 첫째는 참조 패턴을 발견할 때 사용하는 페이지의 속성 정보(과거 참조 거리와 참조 횟수)와 미래 참조 거리 정보를 페이지가 참조될 때 마다 수집하는 부하이다. 둘째는 주기적으로 프로그램의 참조 패턴을 발견하기 위한 부하이다. 그리고 셋째는 프로그램의 패턴에 따라 캐시 및 선반입 정책을 차등적으로 적용하기 위해 필요한 오버헤드이다. 본 논문에서 다루고 있는 3 가지 프로그램들



(그림 14) WC 프로그램 CPU 사용량



(그림 15) AB 프로그램 CPU 사용량

중에서 실행 시 CPU위주의 작업을 수행하는 WC 프로그램과 AB 프로그램만을 대상으로 FCCM을 적용한 커널과 그렇지 않은 커널의 두 경우에 대해 CPU 사용량을 측정하였다. WC와 AB 응용 같이 일괄처리(Batch) 특성을 가지는 프로그램은 CPU 자원의 소모 상황에 더욱 민감하다. (그림 14)과 (그림 15)은 각각 WC와 AB 프로그램의 패턴 탐색에 소모되는 오버헤드 비율를 보여준다.

두 프로그램들의 CPU 사용량을 분석한 결과, WC 프로그램, FCCM이 적용되지 않은 커널과 적용된 커널은 각각 평균 58%(52%~61%)와 평균 69%(56%~73%)의 CPU를 사용함을 알 수 있었다. AB 프로그램의 경우, FCCM이 적용되지 않은 커널, FCCM이 적용된 커널은 각각 83%(78%~90%)와, 92%(82%~95%)의 CPU를 사용함을 알 수 있었다. 위의 분석을 통해 FCCM이 적용되었을 경우, CPU 사용률이 약 10%정도 증가함을 알 수 있다. FCCM이 적용된 경우, 페이지 참조 횟수가 500의 배수인 시점에 CPU의 사용량이 계단식으로 증가함을 보이고 있다. 이는 패턴 탐색 주기가 되어 프로그램의 패턴을 찾는 작업을 수행하는 시점이며, FCCM 오버헤드의 주요인은 프로그램의 패턴 탐색에 사용되는 시간이다. 이러한 계산 부하는 디스크 I/O에 비해 충분히 작은 수치로 판단된다. 따라서 FCCM 계산 오버헤드는 FCCM으로 인한 I/O 비율 감소와 선반입 메모리 소모량의 감소에 비해 무시할 수 있는 수치라 생각된다.

5. 결론 및 향후 연구과제

본 논문에서는 프로그램의 페이지 참조 패턴을 고려하지 않은 리눅스 커널의 캐싱 및 선반입 정책을 개선하고자 NAND 플래시 메모리 전용 파일시스템 계층에서 동작하는 Flash Cache Core Module을 설계 및 구현하였다. 리눅스 캐시 및 선반입 정책과 비교해 본 결과 논문에서 구현한 FCCM 모듈은 최대 55%(평균 20%) 가량 I/O 비율을 줄일 수 있었다. 선반입에 사용되는 메모리 자원 소모량이 정책에 따라 1/2 또는 1/4정도 감소했음에도 I/O 비율이 24% 감소한 것은 캐시의 히트 율이 증가했음을 의미하며, 캐시로 사용되는 메모리 자원이 보다 효율적으로 관리됨을 알 수 있다. 반면, CPU 의존적인 일괄처리 특성을 가지는 프로그램의 수행 시 CPU 사용량은 평균 10% 증가하였다. FCCM 모듈은 패턴 탐색을 수행하는 응용의 수가 많을수록 오버헤드가 증가하므로 다양한 프로그램이 동작하는 범용 시스템보다 특수한 목적의 프로그램들이 제한적으로 수행되는 임베디드 시스템에 적용 시 더욱 효율적으로 사용될 수 있을 것으로 기대 된다. 파일 시스템 레벨 캐시 모듈을 더욱 발전시키기 위해서는 리눅스 커널의 다양한 파일시스템들에 이식 후 안정성과 성능을 검증하기 위한 추가적인 연구가 필요하다. 또한, 패턴 탐색 후 과거와 미래의 패턴이 달라질 경우 이에 대한 추가적인 연구가 필요하다.

참 고 문 헌

[1] J. Catsoulis, *Designing Embedded Hardware*, O'reilly, October 2003.

[2] Samsung Electronics, *SmartMedia Data Book*, 2003.

[3] W. Stallings, *Computer Organization and Architecture*, Prentice-Hall, 2000.

[4] E. G Coffman and P. J. Denning, *Operating System Theory*, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

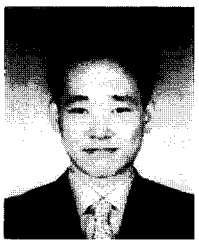
[5] K. Korner, "Intelligent Caching for Remote File System" Proceeding of 10th international Conference on Distributed Computing System, pp 220-226, May 1990.

[6] G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior", Proceeding of the 1997 ACM SIGMETRICS Conference, pp 115-126, June 1997.

[7] C. D Tait and D. Duchamp, "Detection and Exploitation of File Working Sets", Proceeding of the 11th International Conference on Distributed Computing Systems, pp 2.-9, May 1991

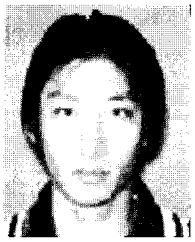
[8] A. J. Smith, "Sequential Program Prefetching in

Memory Hierarchies", IEEE Computer, 11(12):7-21(Vol. 11, No. 12, pp.7-21), December, 1978.
[9] J. Choi, and Sam. H, "Design, Implementation, and Performance Evaluation of a Detection-Based Adaptive Block Replacement Scheme", IEEE Transactions on Computers, Vol. 51, No. 7, pp.793-800, 2000.
[10] Samsung Electronics, "NAND Flash Memory," http://www.samsungelectronics.com/.
[11] JFFS2, http://sorces.redhaty.com/jffs2
[12] Aleph One Company, "Yet Another Flash File System." http://www.aleph1.co.uk/yaffs/.
[13] Patternson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching", Proceeding of the 15th Symposium on Operating System Principles, pp.1-16, 1995.
[14] C. Faloutsos, R. Ng and T. Sellis, "Flexible and Adaptable Buffer Management Techniques for Database Management Systems," IEEE Transactions on Computers, Vol. 44, pp.546-560, 1995.
[15] Daniel P.Bovet, Marco Cesati, *Understanding the LINUX KERNEL*, O'Relly, December 2002.
[16] Robert Love, *임베디드 개발자를 위한 리눅스 커널 심층 분석*, Developer's Library, 2004.
[17] 유영창, *IT EXPERT 리눅스 디바이스 드라이버*, 한빛미디어, 2005.
[18] Intel Corporation, "Understanding the Flash Translation Layer(FTL) Specification," http://www.intel.com/.
[19] 임근수, 고건, "플래시메모리 기반 저장장치의 설계기법", 한국정보과학회 03 가을학술 발표논문집(1), pp.274-276, 2003.
[20] Chanik Park, Jeong-Uk Kang, Seon-Yeong Park, and Jin-Soo Kim, "Energy-Aware Demand Paging on NAND Flash-based Embedded Storages," Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design, 2004.
[21] Heeseung Jo, Jeong-Uk Kang, Seon-Yeong Park, Jin-Soo Kim, and Joonwon Lee, "FAB: Flash-Aware Buffer Management Policy for Portable Media Players," IEEE Transactions on Consumer Electronics, Vol. 52, No. 2, pp.485-493, 2006.
[22] A.-I. A. Wang, P. Reiher, G. J. Popek, and G. H. Kuenning, "Conquest: Better Performance Through a Disk/Persistent-RAM Hybrid File System," Proceedings of the USENIX Annual Technical Conference, 2002
[23] A. S. Tanenbaum, A. S.Woodhull, Operating Systems, Design and Implementation, Prentice Hall, 1997.



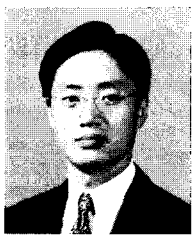
박 상 오

e-mail : sj1st@konan.cse.cau.ac.kr
2005년 중앙대학교 컴퓨터공학과(공학사)
2007년 중앙대학교 컴퓨터공학과(공학석사)
2007년~현 재 중앙대학교 컴퓨터공학과 박사과정
관심분야 : NAND 플래시 메모리 파일 시스템, 임베디드 시스템



김 경 산

e-mail : mountain@konan.cse.cau.ac.kr
2004년 중앙대학교 전기전자공학부(공학사)
2004년~현 재 중앙대학교 컴퓨터공학과 석사과정
관심분야 : 임베디드 시스템, RTOS



김 성 조

e-mail : sjkim@cau.ac.kr
1975년 서울대학교 응용수학과(공학사)
1977년 한국과학기술원 전산과(이학석사)
1977년~1980년 ADD(연구원)
1980년~현 재 중앙대학교 컴퓨터공학부 교수
1987년 Univ. of Texas at Austin(공학박사)
1887년~1988년 Univ. of Texas at Austin(Research Fellow)
1996년~1997년 Univ. California-Irvine(Visiting Professor)
관심분야 : 이동컴퓨팅, 임베디드 소프트웨어, 유비쿼터스컴퓨팅