DOI: 10.3745/KIPSTA.2008.15-A.5.275

디렉토리 지역성을 활용한 작은 파일들의 모아 쓰기 기법

이 경 재[†] · 안 우 현[††] · 오 재 원[†††]

요 약

Fast File System(FFS)은 디스크의 고성능 대역폭을 활용하여 대용량 파일의 여러 블록들을 한 번에 저장함으로써 파일 쓰기 성능을 향상시키고 있다. 하지만, FFS는 파일 단위로 데이터를 저장하기 때문에 작은 파일 쓰기 성능은 디스크 대역폭보다 디스크 회전 및 탐색 시간에 크게 영향을 받는다. 본 논문은 FFS에서 작은 파일 쓰기의 성능 향상을 위해 여러 개의 작은 파일들을 한꺼번에 모아서 저장하는 모아 쓰기(Co-Writing) 기법을 제안하며, 이 기법을 FFS에 적용한 CW-FFS를 구현하였다. CW-FFS의 모아 쓰기 기법은 대역폭을 활용하여 디렉토리 지역성을 가지는 작은 파일들을 모아서 연속적인 디스크 위치에 한 번의 디스크 쓰기로 저장한다. 모아 쓰기 기법은 각 파일 단위로 발생하는 디스크 회전 및 탐색 동작들을 한 번으로 감소시키기 때문에 파일 쓰기가 많은 응용 프로그램에서 작은 파일 쓰기 성능을 개선시킨다. 또한 모아 쓰기 기법이 동일 디렉토리에 포함되는 파일들 간의 디스크 공간 지역성의 저하를 야기하지 않도록 효율적인 파일 할당 방식도 함께 제안한다. CW-FFS는 성능 검증을 위해 OpenBSD 운영체제 커널에서 구현되었으며, postmark 벤치마크를 통한 성능 측정 결과는 기존 FFS 파일 시스템보다 작은 파일 쓰기 성능이 속도 측면에서 5~35%까지 개선되었음을 보여준다.

키워드 : 파일시스템, 운영체제, 디스크, 디렉토리 지역성

Co-Writing Multiple Files Based on Directory Locality for High Performance of Small File Writes

KyungJae Lee[†] · WooHyun Ahn[††] · JaeWon Oh[†††]

ABSTRACT

Fast File System(FFS) utilizes large disk bandwidth to improve the write performance of large files. One way to improve the performance is to write multiple blocks of a large file at a single disk I/O through the disk bandwidth. However, rather than disk bandwidth, the performance of small file writes is limited by disk access times significantly impacted by disk movements such as disk seek and rotation because FFS writes each of small files at a single disk write. We propose CW-FFS (Co-Writing Fast File System) to improve the write performance of small files by minimizing the disk movements that are needed to write small files to disks. Its key technique called co-writing scheme is to dynamically collect multiple small files named by a given directory and then write them at a single disk I/O to contiguous disk locations. Co-writing several small files at a single disk I/O reduces multiple disk movements that are needed for small file writes to one single disk movement, thus increasing the overall write performance of write-intensive applications. Furthermore, a file allocation scheme is introduced to prevent co-writing scheme from having a negative impact on disk spatial locality of small files named by a given directory. The measurement of our technique implemented in the OpenBSD 4.0 shows that CW-FFS increases the performance of small file writes over FFS in the range from 5 to 35% in the Postmark benchmark.

Keywords : File System, Operating System, Disk, Directory-Based Locality

1. 서 론

최근 디스크의 대역폭(Bandwidth)의 발달로 대용량 데이터의 검색 및 저장 속도가 크게 향상되었다. 하지만, 실제 디스크에 저장된 파일의 대다수를 차지하는 작은 용량의 파일들[1]을 저장할 때는 디스크 대역폭을 활용하지 못하고 빈번한 디스크 입출력(I/O) 및 탐색(seek) 때문에 파일 쓰기 성능이 개선되지 않고 있다. 특히, 작은 파일을 매우 빈번하게 다루는 이메일 서버 및 웹 서버 시스템들은 최근의 고성능 디스크를 탑재하더라도 파일 쓰기 성능을 개선하지 못하고 있다. 작은 파일의 저장 성능을 향상시키기 위해 디스크 대역폭을 활용하는 파일시스템 연구가 필요하지만 여전히 많이 진행되지 않고 있다.

※ 이 논문은 2007년도 광운대학교 교내학술연구비 지원에 의해 연구되었음.
※ 본 연구는 2008년도 가톨릭대학교 교비연구비의 지원으로 이루어졌음.
† 준 회 원 : 광운대학교 대학원 컴퓨터과학과 석사과정
†† 정 회 원 : 광운대학교 컴퓨터 소프트웨어학과 조교수
††† 정 회 원 : 가톨릭대학교 컴퓨터정보공학부 전임강사(교신저자)
논문접수 : 2008년 1월 4일
수 정 일 : 1차 2008년 3월 17일, 2차 2008년 6월 11일, 3차 2008년 7월 14일
심사완료 : 2008년 7월 26일

디스크 대역폭을 활용하기 위해 다양한 파일시스템이 제안 되었다. 그 중에서 여러 상업용 운영체제에서 많이 사용되는 파일시스템인 FFS(Fast File System)[2]가 대표적이다. FFS는 디스크를 인접한 실린더(Cylinder)의 묶음인 실린더 그룹으로 나눈 후에 한 디렉토리(Directory)에 속하는 관련 데이터를 동일 실린더 그룹에 위치시킨다. 즉, 시간적으로 동시에 접근되는 동일 디렉토리의 파일들을 디스크에서 동일 실린더 그룹에 저장한다. 이런 지역성을 활용하는 FFS의 특징 때문에 동일 디렉토리의 파일들을 저장할 때 발생하는 디스크 탐색 횟수가 감소하여 파일 쓰기 성능이 증가된다.

하지만 FFS에서는 서로 다른 디렉토리의 파일들을 번갈아 선택하여 참조할 경우, 디스크 탐색 횟수가 증가하여 파일 참조 성능이 크게 떨어질 수 있다. 고성능 서버 시스템은 복잡한 여러 응용 프로그램들을 실행하며, 이들은 여러 디렉토리들을 생성하고 파일들을 활발히 참조하는 경향이 있다[3]. 특히 각 디렉토리를 번갈아 가며 참조하는 패턴이 많다. 디렉토리 단위로 실린더 그룹(Cylinder group)을 할당하는 FFS에서 이 참조 패턴은 서로 다른 실린더 그룹들의 파일들을 번갈아 가며 참조하게 한다. 이 참조 패턴은 디스크 탐색 횟수를 증가시키기 때문에 성능을 크게 저하시킨다.

FFS는 한 파일의 블록을 생성할 때 그 파일의 다른 블록들과 연속적인 디스크 위치에 저장하는 클러스터링(Clustering) 기법[4]을 사용한다. 연속적으로 배치된 파일의 블록들을 변경(Overwrite)할 때 메모리에 일시적으로 저장한 후 디스크 대역폭을 활용하여 단일 디스크 쓰기(Single disk write)로 한 번에 저장하기 때문에 쓰기 성능이 크게 개선된다. 하지만, 이 방법은 큰 파일 쓰기에는 효과를 보이지만 수 개의 블록으로 구성된 작은 파일 쓰기에서는 디스크 대역폭을 충분히 활용하지 못하기 때문에 성능을 개선하지 못한다. 또한, 디스크가 잦은 파일 생성 및 제거로 단편화(Fragmentation)되어 있으면 빈 공간의 부족으로 클러스터링을 못 하는 단점이 있다.

최근 FFS는 파일 메타데이터의 쓰기 성능을 향상시키기 위해 Soft updates 기법[5][6]을 탑재하고 있다. 기존 FFS는 곧 변경될 파일의 메타데이터를 디스크로 곧바로 저장하여 불필요한 디스크 I/O를 야기한다. 이 문제를 해결하기 위해 Soft updates 기법은 변경된 데이터뿐만 아니라 메타데이터를 메모리에 모아두었다가 주기적으로 디스크에 저장한다. 하지만, 갑작스러운 시스템 중지로 발생하는 파일 시스템 일관성 문제를 막기 위해 파일 단위로 메타데이터와 데이터의 관계를 일관성을 유지할 수 있는 순서로 디스크에 저장한다. 그러나 파일들을 저장할 때, 이 기법이 탑재된 FFS라도 기존 FFS와 마찬가지로 여전히 디스크 대역폭을 충분히 활용하지 못한다.

파일의 동기적인 쓰기 성능을 증가시키기 위해 LFS(Log-structured File System)[7]가 제안되었다. LFS는 변경된 파일들을 원래 디스크 위치에 저장하지 않고 메모리에 저장하였다가 주기적으로 로그(Log) 기록 방식으로 빈 공간에 한 번의 디스크 쓰기로 저장하기 때문에 쓰기 성능이 크게 개선되었다. 하지만, 디스크에서 큰 빈 공간을 확보하기 위해 시스템의 성능을 크게 저하시키는 클리닝(Cleaning) 동작을 수행한다. 또한, 디렉토리의 일부 파일들만 변경될 경우, 그 파일들의 디스크 위치가 동일 디렉토리의 다른 파일들과 멀리 떨어지게 된다. 만일 이 디렉토리의 파일들을 동시에 읽을 경우 디스크 탐색 횟수가 증가하여 파일 읽기 성능이 크게 감소될 수 있다[8]. 이런 문제점들 때문에 LFS는 개인용 및 상업용 컴퓨터의 파일 시스템으로 거의 활용되고 있지 않다.

본 논문에서는 FFS에서 파일 읽기 성능을 떨어뜨리지 않으면서 작은 파일 쓰기의 성능을 향상시키기 위해 Co-Writing Fast File System(CW-FFS)을 제안한다. CW-FFS의 주요 기법은 모아 쓰기(Co-Writing) 기법이다. FFS는 생성 또는 변경된 작은 파일들이 비록 디스크에 연속적으로 배치되어 있더라도 각 파일 단위로 디스크 입출력한다. 하지만, 이 모아 쓰기 기법은 디렉토리 지역성(Locality)을 활용하여 여러 개의 작은 파일들을 묶어서 디스크의 새로운 큰 빈 공간에 모아 쓰기로 저장한다. 이때 모아 쓰기로 저장되는 파일들은 FFS처럼 디렉토리의 파일들을 순차적으로 읽을 때 디스크 탐색이 발생하지 않도록 서로 인접한 디스크 위치로 저장된다. 또한, 고정된 크기의 빈 공간이 부족할 정도로 단편화된 디스크에서는 빈 공간 상태에 따라 여러 파일들을 가변적 크기로 묶어서 모아 쓰기를 여러 번 수행한다. 따라서 빈 공간을 확보하기 위해 LFS와 같은 클리닝 동작을 사용하지 않더라도 모아 쓰기가 가능하다.

CW-FFS은 FFS에 비해 두 가지의 장점을 가진다. 첫째, FFS와 달리 여러 작은 파일들을 한 번의 모아 쓰기로 디스크에 저장하기 때문에 디스크 입출력 횟수가 감소되어 작은 파일 쓰기 성능이 증가된다. 둘째, 여러 응용 프로그램들이 서로 다른 디렉토리들의 작은 파일들에 대해 번갈아 가며 쓰기 동작을 수행할 때 파일 단위의 디스크 쓰기 대신 디렉토리 단위로 여러 파일들을 한 번에 디스크로 저장하기 때문에 디스크 탐색 횟수가 크게 감소되고 디스크 대역폭을 잘 활용할 수 있다. 이런 장점들과 더불어 LFS처럼 여러 파일들을 모아 쓰기로 저장하지만 FFS처럼 파일들을 동일한 실린더 그룹으로 저장함으로써 디스크 탐색 없이 디렉토리의 파일들을 순차적으로 읽을 수 있다.

CW-FFS를 OpenBSD 4.0 커널에 구현하였고 성능 검증을 위해 벤치마킹 프로그램과 실제 응용 프로그램을 수행하였다. Postmark 벤치마킹 프로그램의 실험 결과에서 작은 파일들의 쓰기, 수정 및 삭제 성능이 속도측면에서 FFS보다 5 ~ 35% 정도 향상되었음을 볼 수 있었으며, 반면에 파일 읽기 성능이 저하되지 않았음을 확인하였다. 또한, 실제 응용 프로그램을 사용한 실험에서는 10% 정도 성능이 향상되었다.

본 논문의 구성은 다음과 같다. 2장에서는 본 논문에 대한 관련 연구를 설명하며, 3장에서는 제안하는 파일시스템

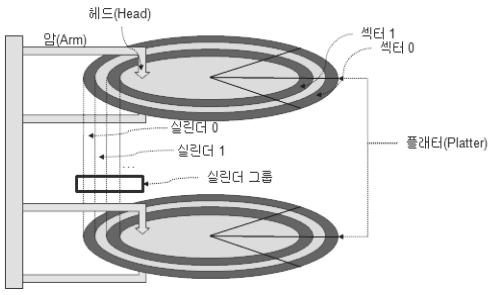
인 CW-FFS에 대하여 구체적으로 기술한다. 4장에서는 FFS에 모아 쓰기 기법을 구현하기 위해 개선 또는 변경된 부분에 대해 설명한다. 5장은 실험 및 결과 분석을 나타낸다. 마지막으로 6장에서 결론을 도출한다.

2. 관련 연구

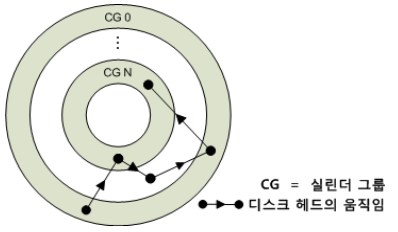
FFS는 상업용 시스템인 *SunOS/Solaris, System V Release 4, HP-UX, Tru64 UNIX, BSD* 계열의 운영체제 등에서 널리 사용 중인 파일시스템이다. FFS는 파일 접근 속도를 향상시키기 위해 디스크를 실린더 그룹 단위로 나눈다. (그림 1)처럼 각각의 실린더 그룹은 디스크 탐색 시간을 무시할 수 있는 인접한 실린더들의 묶음으로 구성된다. FFS는 시간적으로 동시에 접근하는 동일 디렉토리의 모든 파일들을 같은 실린더 그룹 안에 할당하기 때문에 동일한 디렉토리에 포함된 파일들은 인접한 위치에 저장될 수 있다. 이 디렉토리의 지역성으로 동일 디렉토리의 파일들을 순차적으로 참조할 때 디스크 탐색 없이 동일 실린더 그룹에서 파일들을 접근할 수 있기 때문에 파일 읽기 및 쓰기 성능이 향상된다.

하지만, FFS에서는 서로 다른 디렉토리들을 번갈아 선택하여 파일을 참조하면 디스크 탐색 횟수가 증가하여 성능이 매우 떨어지게 된다. (그림 2)는 FFS에서 디렉토리를 번갈아 참조할 때 디스크 헤드의 움직임을 보여준다. FFS에서 기본적으로 한 디렉토리의 데이터는 한 개의 실린더 그룹에 할당된다. 이러한 경우 각 디렉토리를 번갈아 선택하여 파일 참조를 하면 디스크 헤드의 움직임 즉, 디스크 탐색이 매우 빈번하게 발생하게 된다. 따라서 파일시스템의 성능이 크게 저하된다.

FFS는 파일에 포함되는 여러 블록들을 연속적인 디스크



(그림 1) 디스크의 구성도와 실린더 그룹



(그림 2) FFS의 동작에 의한 빈번한 디스크 헤드 움직임의 예

위치에 단일 쓰기로 저장할 수 있도록 클러스터링 기법 [4]을 사용한다. 이 기법은 파일에 새로운 블록을 추가할 때 그 파일의 다른 블록들과 디스크에서 연속되도록 배치한다. 이때, 연속적으로 배치된 블록들로 구성된 파일을 변경할 때 디스크 대역폭을 활용하여 여러 블록을 단일 쓰기로 한 번에 저장하기 때문에 파일 입출력 횟수를 줄일 수 있다. 하지만, 연속적으로 배치된 블록들로 구성된 파일들을 파일 단위로 단일 쓰기하기 때문에 큰 파일 쓰기에는 디스크 대역폭을 충분히 활용할 수 있지만 작은 파일 쓰기에서는 대역폭보다 디스크 탐색 및 회전에 전적으로 영향을 받는다. 따라서 클러스터링 기법은 작은 파일들에 대해 쓰기 성능을 개선하지 못한다. 또한, 잦은 생성 및 제거로 인해 디스크가 단편화되어 있으면 빈 공간의 부족으로 클러스터링 방식을 사용하지 못하게 된다.

디스크 단편화로 야기되는 FFS 클러스터링의 한계를 극복하기 위해 블록 재할당 방식(Block reallocation)[9]이 FFS에 구현되었다. (그림 3)은 FFS의 재할당 방식을 보여준다. 파일 F1 및 Fa는 각각 5개의 블록과 1개의 블록으로 구성되며, 이 표기 방법은 다른 파일들에게도 동일하게 적용된다. 단계 (1)의 디스크는 잦은 파일 생성 및 제거로 인해 단편화되어 있다. 단계 (2)에서 단편화된 디스크에 파일 F1이 저장된다면 인접 위치에 저장 될 수 없게 된다. 단계 (3)에서 파일 Fb가 삭제되어 충분한 연속적인 빈 공간이 확보된다. 단계 (4)에서 파일 F1이 변경될 때 파일 F1의 블록들을 클러스터링하여 인접한 위치로 재할당(또는 재배치)한다. FFS는 자주 참조되는 블록들을 메모리의 한 부분인 파일 캐쉬(File Cache)에 저장하여 빈번한 디스크 참조를 막는다. 블록 재할당 방식은 파일의 블록들을 클러스터링하기 위해 파일 캐쉬를 활용한다. 즉, 파일 F1의 블록들을 블록 단위로 디스크에 곧바로 저장하지 않고 파일 캐쉬에 저장해두었다가 그 블록들을 단일 쓰기로 디스크에 저장한다. 이 재할당 방식의 장점은 연속적으로 배치되지 않은 파일의 블록들을 재변경할 때 연속적인 빈 공간이 존재한다면 그 빈 공간으로 연속적으로 재배치하기 때문에 디스크 입출력 횟수를 줄일 수 있다는 것이다.

하지만, FFS의 재할당 방식은 다수의 블록으로 구성된 큰 파일 참조에는 성능 개선이 가능하지만 클러스터링 방식과 마찬가지로 한 개 또는 수 개의 블록들로 구성된 작은 파일을 연속적으로 재배치하는 경우 디스크 대역폭을 충분히 활용하지 못하기 때문에 쓰기 성능을 개선하지 못한다. 더욱이 거의 1~2개의 블록으로 구성된 작은 파일들을 사용하는 실제 서버 환경[1]에서는 재할당 방식으로 얻을 수 있는 파일 쓰기 성능 효과가 적을 수밖에 없다.

(그림 3) FFS의 블록 재할당 방식

3. 모아 쓰기 Fast File System

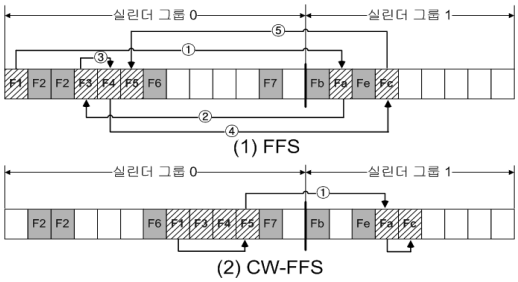
3.1 기본 개념

우리는 디렉토리 지역성을 활용하여 FFS의 작은 파일(12개 이내의 블록들을 가진 파일)의 쓰기 성능을 향상시키는 모아 쓰기(Co-Writing) Fast File System (CW-FFS)을 제안한다. CW-FFS의 주요 기술은 변경된 작은 파일들을 디스크로 저장할 때 동일 디렉토리에 포함되는 파일들을 모아서 한 번의 디스크 쓰기로 동시에 저장하는 모아 쓰기 기법(Co-Writing)이다. 대부분 파일 쓰기는 파일의 일부 블록만 변경하는 것이 아니라 전체 파일을 변경하거나 생성하는 순차적 쓰기 패턴[10]을 따른다. 이러한 순차적 파일 쓰기의 성능을 향상시키기 위해 새롭게 생성되거나 모든 블록들이 변경되는 작은 파일들에 대해 모아 쓰기 기법을 적용한다.

일반적으로 동일 디렉토리의 파일들은 디렉토리 지역성에 의해 함께 생성 또는 변경될 가능성이 높다. FFS는 이 변경된 파일들을 디스크로 곧바로 저장하지 않고 파일 캐쉬에 모아두었다가 나중에 파일 단위로 디스크에 저장한다. 반면에, CW-FFS의 모아 쓰기 기법은 파일 캐쉬에서 한 개의 파일을 디스크로 저장할 때 이 파일과 같은 디렉토리에 속하는 변경된 다른 작은 파일들까지 함께 모아서 디스크의 빈 공간에 한 번의 디스크 쓰기로 저장한다. 결국 디스크 입출력 및 탐색 횟수를 크게 감소시키기 때문에 쓰기 성능을 크게 향상시킬 수 있다.

모아 쓰기를 위해 항상 연속적인 빈 공간이 필요하며, 이 빈 공간 확보를 위해 저장될 파일들이 속한 디렉토리에 할당된 실린더 그룹의 빈 공간을 활용하는데, 그 이유는 다음과 같다. 첫째, 여러 파일들을 모아 쓰기로 디스크에 저장할 때 LFS처럼 큰 연속적인 빈 공간을 확보하기 위해 큰 부하를 야기하는 클리닝 동작이 실행되지 않아야하기 때문이다. 둘째, 모아 쓰기로 저장하는 파일들을 동일 디렉토리의 다른 파일들과 인접하게 배치하여 순차적으로 파일들을 읽을 때 성능 저하가 없어야하기 때문이다. 그런데 파일의 생성 및 제거 동작이 활발한 디렉토리에 대응하는 실린더 그룹에서는 항상 일정한 크기의 연속적인 빈 공간을 확보할 수 있는 것은 아니다. 이런 단편화된 디스크에서는 확보 가능한 연속적인 빈 공간을 검색하여 작은 연속적인 빈 공간이 확보되더라도 그 공간만큼의 작은 파일들을 모아서 저장한다. 단편화된 디스크에서 확보 가능한 연속적인 빈 공간을 검색하여 찾은 연속적인 빈 공간의 블록 수가, 모아 쓰기가 한 번에 허용하는 블록 개수 N보다 적더라도 그 공간만큼의 작은 파일들을 묶어서 저장한다.

(그림 4) (1)과 (2)는 각각 FFS와 CW-FFS에서의 파일 쓰기 동작을 보여준다. 여기서 파일 F2와 Fa는 각각 2개의 블록과 1개의 블록으로 구성되며, 이 표기 방법은 다른 파일들에게도 동일하게 적용한다. 디렉토리 1과 2는 각각 실린더 그룹 0과 1에 할당되어 있다. 디렉토리 1에 포함된 파일 F1, F3, F4, F5(빗금 친 네모)와 디렉토리 2에 포함된 파일 Fa, Fc(빗금 친 네모)를 파일 F1, Fa, F3, F4, Fc, F5



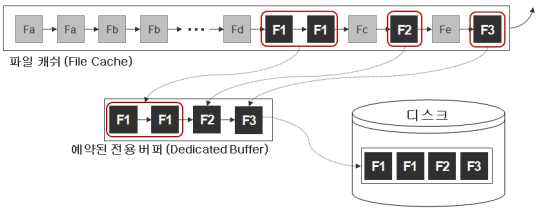
(그림 4) FFS와 CW-FFS에서의 파일 쓰기 비교

순서로 두 개의 디렉토리에 번갈아 쓰는 과정을 보여준다. 이때 F2, F6, F7, Fb, Fe(회색 네모)는 변경되지 않는 파일을 나타낸다. 기존 FFS에서는 각 파일들을 변경된 순서로 디스크에 저장하기 때문에 실린더 그룹 간 디스크 헤드의 움직임이 총 4번 발생한다. 하지만, CW-FFS에서는 파일들의 변경된 순서와 상관없이 동일 디렉토리의 파일들을 묶어서 모아 쓰기로 저장하기 때문에 파일 F1, F3, F4, F5를 디렉토리 1의 빈 공간에 모아 쓰기로 저장하고 디스크 헤드가 디렉토리 2로 넘어가서 파일 Fa, Fc를 모아 쓰기로 저장한다. 따라서 그림에서 화살표로 나타낸 디스크 헤드의 움직임과 이동 횟수로 확인할 수 있듯이 FFS에서는 5번의 디스크 입출력과 4번의 디스크 탐색에 비해 CW-FFS에서는 단 2번의 디스크 입출력과 1번의 디스크 탐색이 발생하기 때문에 성능을 크게 향상할 수 있다. 또한, 디렉토리 단위의 지역성이 유지되기 때문에 읽기 성능 저하가 발생하지 않는다.

FFS를 기반으로 모아 쓰기 기법을 지원하는 CW-FFS를 구현하기 위해 다음과 같이 FFS을 변경 또는 개선하였다. 첫째, 파일 단위로 디스크 I/O를 시도하는 동작 대신 여러 개의 파일들을 묶어서 한 번의 디스크 쓰기로 저장하는 기능을 지원할 수 있도록 파일 캐쉬 모듈을 개선하였다. 둘째, 동일 파일의 블록들을 인접하게 할당하는 방식 외에 여러 파일들을 인접하게 할당할 수 있도록 파일 블록 할당 모듈을 변경하였다. 셋째, 변경된 파일들을 마운트(Mount)된 파일 시스템 단위로 관리하지 않고 디렉토리 단위로 관리할 수 있도록 메타데이터를 관리하는 vfs/vnode[11] 모듈을 개선하였다.

3.2 모아 쓰기 기법 (Co-Writing Scheme)

모아 쓰기 기법은 파일 캐쉬의 데이터가 디스크로 저장되는 지연 쓰기(Delayed write) 시점에 실행된다. FFS는 변경된 파일들의 블록을 곧바로 디스크로 저장하지 않고 파일 캐쉬에 저장한 뒤 파일 캐쉬의 공간이 부족하여 파일 블록 교체(Block replacement)가 발생하거나 주기적으로 실행되는 싱크 데몬(Sync daemon)에 의해 디스크로 저장하는 지연 쓰기 방식을 사용한다. CW-FFS는 이 지연 쓰기 동안 파일 캐쉬에서 디스크로 저장되는 블록과 이 블록을 포함하는 파일이 속한 디렉토리 내의 다른 작은 파일들을 함께 묶은 후 모아 쓰기로 디스크에 저장한다. 이때 저장될 파일들의 새 디스크 위치는 동일 디렉토리의 파일들에 대한 순차적인 읽



(그림 5) 블록 교체 시점의 모아 쓰기 동작

기 성능이 저하되지 않도록 그 파일들이 속한 디렉토리가 할당된 실린더 그룹 내로 결정된다. 모아 쓰기는 주로 파일 캐쉬의 파일 블록 교체 시기에 이루어진다. (그림 5)는 파일 블록 교체 시기에 파일 캐쉬에 저장된 작은 파일들을 묶어서 디스크로 저장하는 과정을 보여준다. 여기서 F1, F2, F3는 동일 디렉토리의 작은 파일이다. 응용 프로그램에서 파일이 한번 참조되면 그 파일은 다음 참조 시기에 빠른 접근을 위해 파일 캐쉬에 저장된다. 파일 캐쉬는 LRU(Least Recently Used) 방식으로 가장 오래 참조되지 않은 파일 블록을 쫓아낸다. (그림 5)에서 가장 오래 참조되지 않은 파일이 F3라면 이 파일을 쫓아내는 시기에 미리 예약된 전용 버퍼(Dedicated buffer)에 동일 디렉토리에 포함된 다른 파일 F1, F2를 함께 채운다. 그 후 이 버퍼에 채워진 파일들을 디스크의 빈 공간에 한 번의 디스크 쓰기로 저장한다. 만약 파일 캐쉬에서 교체되는 블록이 큰 파일의 블록이면 모아 쓰기 과정을 거치지 않고 FFS처럼 곧바로 디스크로 저장한다.

모아 쓰기의 최대 크기는 파일시스템 파라미터에 의해 결정되며, 그 파라미터는 디스크의 최대 데이터 입출력(I/O) 전송 크기로 설정된다. 대부분 디스크는 128 KB의 최대 전송 크기를 가진다. 하지만, 임의의 디렉토리에 속하는 변경된 파일들의 양이 모아 쓰기의 최대 크기보다 클 수 있다. 이런 경우에 CW-FFS는 그 변경된 파일들을 최대 크기로 나뉘어서 모아 쓰기를 여러 번 하여 디스크로 저장한다.

```
begin
  // Let b be a block in file cache being flushed to disk
  find file f such that a block b ∈ f
  set fsize to the total number of blocks of f
  if fsize > 12 then
    store only the current block b to disk at one disk I/O
    store the metadata (i.e., inode) for f
    return
  end if
  find cylinder group cg corresponding to directory d such that f ∈ d
  find a contiguous free space, cspace in cg using the algorithm
  shown in Section 3.3.
  if cspace does not exist then
    store only the current block b to disk at one disk I/O
    store the metadata for f
    return
  end if
  get a dedicated buffer buf of the cspace size in memory
  get the smallest file fs in file cache such that fs ∈ d
  while buf is not full && fs exists do
    move fs into buf
    get the smallest file fs in file cache such that fs ∈ d
  end while
  store buf to disk at one disk I/O
  update metadata for files being co-written to disk.
end
```

(그림 6) 모아 쓰기 알고리즘

(그림 6)은 작은 파일의 한 개 블록을 지연 쓰기할 때 실행되는 모아 쓰기 알고리즘이다. 그 블록의 파일이 13개 이상의 블록을 가지는 큰 파일이라면 FFS처럼 그 블록만 디스크로 저장한다. 그렇지 않으면 모아 쓰기 동작이 시작된다. 그 파일을 포함하는 디렉토리에 할당된 실린더 그룹에서 모아 쓰기가 허용하는 최대 크기의 연속적인 빈 공간을 확보하려고 시도하고 그 공간을 확보할 수 없을 경우 3.3절에 언급될 기법을 통하여 그보다 작은 연속적인 공간을 확보하고자 한다. 만일 연속적인 빈 공간이 없다면 모아 쓰기는 중단되고 지연 쓰기될 블록만 디스크로 저장된다. 원하는 크기의 빈 공간을 획득했다면 여러 파일들을 메모리에 함께 모으기 위해 그 크기의 전용 버퍼를 할당받는다. 가능한 많은 작은 파일들을 모아 쓰기로 저장하기 위해 가장 작은 크기의 파일들을 전용 버퍼에 모은다. 전용 버퍼가 다 채워졌거나 더 이상 모아 쓰기할 파일이 없을 때 전용 버퍼의 파일들을 모아 쓰기로 디스크에 저장한다. 마지막으로 그 파일들의 특성을 관리하는 메타데이터인 inode를 디스크로 저장한다.

3.3 단편화된 디스크에서 모아 쓰기

모아 쓰기의 최대 크기만큼 파일들을 묶어서 모아 쓰기하기 위해서는 항상 큰 빈 공간이 요구된다. 하지만, 디스크가 단편화되어서 연속적인 빈 공간이 모아 쓰기가 허용하는 최대 크기보다 부족할 경우에는 최대 크기로 모아 쓰기를 할 수 없게 된다. 이 문제를 해결하기 위해서 만일 저장될 파일이 속한 디렉토리에 할당된 실린더 그룹에서 최대 크기의 빈 공간을 확보하지 못했을 경우, 그보다 작은 빈 공간들 중에서 가장 큰 빈 공간을 사용하여 모아 쓰기를 수행한다.

(그림 7)은 단편화된 디스크를 고려한, 연속적인 빈 공간을 검색하는 알고리즘을 나타낸다. 이 알고리즘은 단편화된 디스크에서 (그림 6) 알고리즘을 실행할 때 연속적인 빈 공간을 검색하기 위해 사용된다. 모아 쓰기로 저장될 파일이 속한 디렉토리가 할당된 실린더 그룹에서 블록들의 할당 정보를

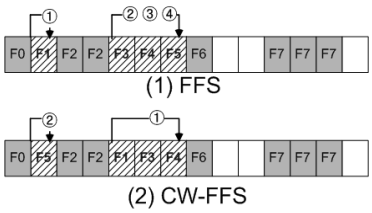
```
begin
 // Let N be the maximum number of contiguous free blocks al-
    lowed by one co-writing.
 // Let CG be a cylinder group corresponding to a given directory.
 // 최대 N 크기부터 1씩 감소하면서 빈 공간 검색한다.
 Set n to N
 while (n > 1)
   block_address = ffs_hashalloc(CG, n)
   if (block_address != -1) // n개 블록 크기의 연속적인 빈 공간 검
     색 성공
     return block_address and n
   end if
   Decrement n by 1
 end while
 // 연속적인 빈 공간의 검색을 실패했을 때  -1 값 리턴
 return -1
end
// ffs_hashalloc(CG, n): 실린더 그룹 CG에서 n개 블록 크기의
// 빈 공간을 검색하여 성공할 경우, 그 빈 공간을 할당하고 시작
// 블록 위치를 반환함. 실패하면 -1을 반환하는 커널 내의 함수임.
```

(그림 7) 최대 크기의 연속적인 빈 공간 검색 알고리즘

관리하는 실린더 그룹 맵(Cylinder group map)을 찾는다. 이 실린더 그룹 맵을 사용하여 빈 공간 검색 알고리즘이 실행된다. 우선 모아 쓰기를 할 수 있는 최대 크기 N의 빈 공간부터 검색한다. 최대 크기(128KB) 빈 공간의 블록 수를 N이라고 하면 N은 블록 크기(block size)에 따라 달라진다. 예를 들면 블록 크기가 16KB이면 N은 8이 된다. 만일 N 크기의 빈 공간이 검색되지 않으면 1씩 감소된 크기의 빈 공간을 차례로 검색한다. 만일 연속적인 빈 공간이 검색되면 그 공간의 시작 블록 번호와 크기를 전달한다. 만일 2개 이상의 연속적인 빈 공간이 검색되지 않으면 검색 실패를 알려주는 값 -1을 리턴한다. 이때 연속적인 빈 공간 검색이 실패했을 때 CW-FFS는 디스크로 단지 한 개의 블록만 저장한다.

(그림 8)은 단편화된 디스크에서 FFS와 CW-FFS의 파일 쓰기 방식을 보여준다. 파일 F1 및 F7은 각각 1개의 블록과 3개의 블록으로 구성되며, 이 표기 방법은 다른 파일들에게도 동일하게 적용한다. 모든 파일들이 동일 디렉토리에 포함되고, 파일 F1, F3, F4, F5(빗금 친 네모) 순서로 디스크 쓰기를 한다고 가정한다. 이때 F0, F2, F6, F7(회색 네모)은 변경되지 않는 파일을 나타낸다. (그림 8)(1)의 FFS에서는 한 개 블록 크기의 파일 4개를 파일 단위로 저장하기 때문에 디스크 쓰기가 4번 생긴다. 하지만, (그림 8)(2)의 CW-FFS에서는 단 2번의 디스크 쓰기로 저장이 가능하다. 즉, 변경된 파일 F1, F3, F4, F5의 4개 블록을 한 번에 저장할 빈 공간을 검색하지만 4개 블록의 빈 공간이 없기 때문에 먼저 3개의 빈 공간을 찾아 파일 F1, F3, F4를 저장하고 나머지 파일 F5를 저장한다. 결국 큰 빈공간이 부족하여 모든 변경된 파일들을 모아 쓰기로 저장하지 못할 경우, 그보다 작은 크기의 빈 공간만큼 파일들을 묶어서 모아 쓰기를 수행한다. 하지만, 여전히 모아 쓰기 동안에 여러 파일들이 함께 저장될 수 있기 때문에 FFS에 비해 디스크 I/O 횟수를 감소시킬 수 있다.

최대 크기의 빈 공간이 없을 경우, 그보다 작은 크기의 빈 공간을 활용하는 이유는 시스템 부하를 최소화하기 위해서이다. LFS가 일반 컴퓨터의 파일시스템으로 사용되지 않는 가장 큰 이유는 항상 고정된 크기의 빈 공간을 확보하기 위해 사용하는 클리닝 동작으로 인한 과부하 때문이다. 이 치명적인 단점을 가지는 클리닝 동작을 FFS에 적용할 경우, 작은 파일 쓰기 성능의 향상에 기여하기 보다는 클리닝 동작으로 발생하는 단점이 더욱 클 수 있다. 따라서 FFS 기반의 CW-FFS는 최대 크기의 빈 공간을 확보하지 못했을 경우, 그보다 작은 크기의 빈 공간을 활용하여 모아 쓰기를 실행한다. 하지만 만약에 디스크가 심각하게 단편화되어 1~2개 블록의 빈 공간만 존재 한다면 CW-FFS의 효율성은 매우 감소하게 된다. 하지만, 기존 연구[12]에 따르면 서버용 컴퓨터뿐만 아니라 개인용 컴퓨터에서 디스크의 단편화는 크지 않다. 따라서 CW-FFS에서 모아 쓰기를 못할 정도로 큰 빈 공간을 확보하지 못 할 가능성은 적을 것이다. 또한, 최근 디스크 대용화(100GB이상) 추세로 인해 CW-FFS의 이러한 단점이 극복될 수 있다.



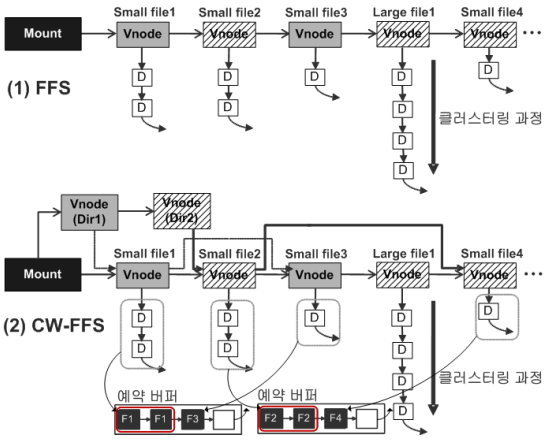
(그림 8) 단편화된 디스크에서의 모아 쓰기

4. 모아 쓰기 기법 구현

본 논문은 OpenBSD 4.0 커널의 FFS에 모아 쓰기 기법을 추가하여 CW-FFS를 구현하였다. 다음 절들은 FFS에 모아 쓰기 기법을 구현하기 위해 추가하거나 변경할 기능들을 설명한다. 우선 4.1절은 모아 쓰기할 때 동일 디렉토리에 속하는 파일을 모으기 위해 사용하는, 변경된 파일들을 디렉토리별로 관리하는 기능을 설명한다. 4.2절은 FFS와 달리 디스크로 저장될 파일들의 디스크 위치가 모아 쓰기 시점에서 할당되도록 파일시스템의 파일 쓰기 함수에 대한 변경사항을 기술한다. 마지막으로 4.3절은 모아 쓰기 동안에 수행하는 연속적인 빈 공간 검색, 모아 쓰기될 파일의 검색 및 검색된 파일을 전용 버퍼로 모으는 기능에 대한 구현을 설명한다.

4.1 디렉토리 단위의 파일 관리 기법

변경된 파일들을 디렉토리 단위의 파일들로 모아 쓰기를 하기 위해서는 파일 캐쉬에 저장된 파일들을 디렉토리 단위로 관리해야 한다. 이 기능은 다른 새로운 구조체의 추가 없이 FFS의 vfs(virtual file system)[11] 모듈에서 구현된 vnode[11]를 사용하여 구현할 수 있다. vnode는 파일시스템에 종속적인 파일 정보인 inode에 대응되는 메모리 객체(또는 구조체)로서 파일 시스템에 비종속적인 파일 정보를 관리한다. (그림 9)(1)과 (2)는 각각 FFS와 CW-FFS의 vnode 관리 기법을 보여주고 있다. 디렉토리 1은 작은 파일 1, 3을 포함하고 있고 디렉토리 2는 작은 파일 2, 4, 큰 파일 1을



(그림 9) 변경된 파일의 vnode 관리 기법

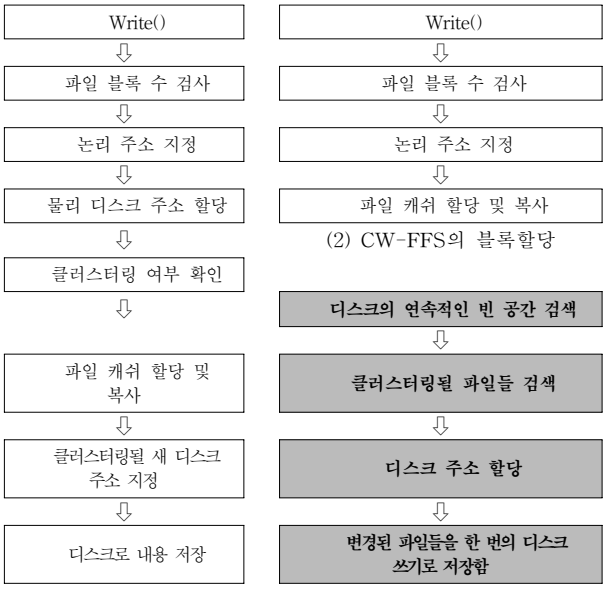
포함하고 있다. (그림 9)(1)의 FFS는 변경된 파일들의 vnode들을 디렉토리 지역성을 고려하지 않고 커널 메모리에서 연결리스트로 관리한다. 마운트 포인터(mount pointer)에 변경된 각 파일의 vnode를 연결리스트 형태로 관리하며, 이 리스트를 싱커 리스트(Syncer list)라고 한다. 그리고 각 파일의 변경된 블록들을 vnode단위로 연결리스트로 연결하며, 이를 변경 블록 리스트(dirty block list)라 한다. 변경된 파일의 블록들이 모두 디스크로 저장되면 이 파일에 대응하는 vnode를 싱커 리스트에서 제거한다. 또한, 주기적으로 실행되는 싱크 데몬(Sync daemon) 프로세스가 싱커 리스트에 연결된 vnode들을 순차적으로 가로지르면서 각 vnode의 블록들을 개별적으로 디스크로 저장한다. 만일 큰 파일에 대응하는 vnode이며 그 파일이 클러스터링된 블록들을 포함하면 그 블록들을 한 번의 디스크 쓰기로 저장한다.

(그림 9)(2)의 CW-FFS는 모아 쓰기에 앞서 동일 디렉토리에 포함되면서 변경된 작은 파일들 중에 파일 캐쉬에 저장된 파일들을 디렉토리 단위로 관리하며, (그림 9)(1)의 FFS의 vnode관리 기법을 확장하여 구현되었다. 먼저, 작은 파일의 블록들을 디렉토리 단위로 관리하기 위해 각 디렉토리의 vnode별로 변경된 파일들의 vnode를 관리하는 연결리스트를 추가한다. 이 리스트를 DF 리스트(Dirty File List)라 정의한다. 반면에 큰 파일들은 기존 FFS의 방식으로 관리될 수 있도록 마운트 포인터의 연결리스트에 연결된다. 파일 캐쉬의 블록 교체 시에 디렉토리에 대응되는 실린더 그룹에서 확보한 빈 공간만큼 변경된 파일들의 블록들을 모아서 한 번의 디스크 쓰기로 이 빈 공간으로 저장한다. 마지막으로 이 동작이 완료되면 기존 FFS처럼 마운트 포인트의 연결 리스트를 살펴보면서 큰 파일들을 디스크로 저장한다.

4.2 지연 블록 할당 기법

CW-FFS는 파일 생성 시간과 디스크 단편화를 감소시키기 위해 변경된 파일의 블록들에 대한 디스크 위치를 지연 쓰기 시점에 결정한다. 기존 FFS는 파일에 블록을 할당할 때 그 블록을 파일 캐쉬에 저장하기에 앞서 디스크 블록을 할당한다. 하지만, FFS의 방식을 CW-FFS에서 사용하면 모아 쓰기로 디스크에 저장하기 전에 이전 할당받은 디스크 위치를 새로운 디스크 위치로 변경해야 한다. 이러한 방식은 파일시스템의 성능이 저하 될 수 있기 때문에 CW-FFS에서는 변경되는 시점에 디스크 위치를 할당하지 않고 디스크로 블록들을 저장할 시점인 지연 쓰기 동안에 블록들의 위치를 할당한다.

(그림 10)은 FFS와 CW-FFS의 블록 할당 및 쓰기 과정을 보여준다. (그림 10)(1)처럼 먼저 응용 프로그램이 write 시스템 함수를 호출하면 FFS는 저장할 파일의 블록 수를 검사한다. 파일 블록들에게 물리 디스크 주소와 매핑(Mapping)되는 논리 주소를 지정하고 디스크 위치를 할당한다. 그 블록의 디스크 위치를 사용하여 그 파일의 다른 블록들과의 클러스터링 여부를 확인한다. 클러스터링이 가능하다면 그 크기만큼의 파일 캐쉬의 공간을 할당받아 블록



(그림 10) FFS와 CW-FFS의 블록 할당 및 쓰기 방식

들의 내용을 복사한다. 마지막으로 클러스터링된 블록들의 물리 디스크 주소를 연속적인 빈 디스크 영역의 주소로 변경하고, 그 블록들을 지정된 디스크 위치로 저장한다.

하지만, (그림 10)(2)처럼 CW-FFS는 단지 저장할 파일 블록 수를 검사하고 논리적인 주소를 지정하지만 실제 디스크 위치를 할당하지 않는다. 디스크 위치를 할당하기 전에 파일 캐쉬의 공간을 할당받아 저장할 블록 내용을 복사한다. 그 다음 (그림 10)(3)처럼 파일 캐쉬에서 블록이 교체될 때 그 블록의 파일이 속한 디렉토리에 대응하는 실린더 그룹에서 모아 쓰기 가능한 연속적인 빈 공간을 획득한다. 이 빈 공간 크기만큼의 작은 파일들을 교체되는 블록의 파일이 속한 디렉토리의 DF 리스트에서 검색한 후에 이 파일들의 실제 디스크 위치를 지정한다. 마지막으로 검색된 파일들을 예약된 버퍼에서 묶어 클러스터링 후에 디스크에 한 번의 디스크 쓰기로 저장한다. 이러한 CW-FFS의 블록 쓰기 할당 과정은 아래에서 구체적으로 설명한다.

4.3 지연 쓰기 시점의 디스크 쓰기 과정

(그림 10) (3)의 지연 쓰기 시점에 실행되는 모아 쓰기의 동작들에 대한 구현을 기술한다.

4.3.1 연속적인 빈 공간 검색

FFS는 수퍼블록(Superblock)에 각 실린더 그룹의 블록 할당 상태를 나타내는 실린더 그룹 맵을 관리하고 있다. 이 실린더 그룹 맵은 비트맵(Bit-map)으로 구현되어 있으며, 각 비트는 실린더 그룹 내의 블록에 대응한다. 또한 이 실린더 그룹 맵은 시스템 부팅 시점에 커널 메모리로 로드(Load)된다. 파일 한 개가 생성될 때 이 파일에 실린더 그룹의 블록이 할당되며, 실린더 그룹 맵의 그 블록 위치에 할

당 상태를 표시한다. 그 다음 FFS는 변경된 실린더 그룹 맵을 디스크로 저장한다. 이때 블록을 할당하기에 앞서 실린더 그룹 맵을 검색할 때 요구되는 CPU 부하는 무시할 정도로 작다. 그 이유는 커널 메모리에 저장된 실린더 그룹 맵을 사용하기 때문이다.

CW-FFS는 최대 가능한 연속적인 빈 공간 크기를 검색하기 위해 기존 FFS에 구현된 실린더 그룹 맵을 이용한다. 파일 캐쉬에서 교체되는 블록의 파일이 속한 디렉토리에 대응하는 실린더 그룹 맵을 (그림 7)의 알고리즘으로 검색하여 가능한 최대 크기의 빈 공간을 검색한다. 비록 전체 실린더 그룹 맵을 검색하더라도 최근의 CPU가 비트 맵 동작을 빠르게 실행할 수 있도록 최적화되었기 때문에 CPU 부하는 매우 적다.

4.3.2 클러스터링 대상 파일들 검색

모아 쓰기를 위한 연속적인 빈 공간을 검색한 뒤 그 빈 공간을 채울 작은 파일들을 찾아야한다. CW-FFS는 지연 쓰기될 파일이 속한 디렉토리에 대응하는 vnode의 DF 리스트를 순차적으로 검색하면서 빈 공간의 크기에 적합한 크기의 작은 파일들을 선택한다. 하지만, 한 파일의 일부 블록들은 모아 쓰기에 포함되는 반면 다른 일부 블록들이 모아 쓰기에 포함되지 않을 경우, 그 파일은 디스크에서 서로 분리되어 저장될 수 있다. 결국 이 파일에 대한 참조는 다중 디스크 참조를 발생시키기 때문에 디스크에서 서로 분리되지 않을 작은 파일을 선택한다. 즉, 빈 공간에 채워질 수 없는 작은 파일들은 건너뛰고 다음 파일들을 검색한다.

하지만, 이러한 순차적인 파일 검색은 CPU 부하를 증가시킬 수 있다. 이 문제를 해결하기 위해 각 DF 리스트에서는 작은 파일들의 vnode들을 블록 수로 관리 할 수 있는 해쉬 테이블(Hash table)을 사용하여 CPU 부하가 작도록 구현되었다. 또한, 빈 공간을 파일들로 채울 때 가능한 많은 파일들을 모아 쓰기로 저장하기 위해 가장 작은 파일들부터 파일 크기가 커지는 순서로 파일을 수집한다.

4.3.3 파일 블록의 디스크 위치 할당

여러 파일들을 모아 쓰기로 연속적인, 빈 공간으로 저장할 때 실린더 그룹 맵을 포함하는 수퍼블록 및 각 파일의 inode 등 메타데이터의 변경이 요구된다. 이 변경 과정은 새롭게 생성되는 블록을 가지는 파일과 변경되는 블록들을 가지는 파일에 따라 다르다. 전자에 대해서는 실린더 그룹 맵에 그 블록들이 할당되었음을 표시하며, 그 파일의 inode에 디스크 위치를 지정하면 된다. 하지만, 후자에 대해서는 다음과 같은 여러 단계로 구성된다. 첫째, 할당될 빈 공간의 블록들을 포함하는 실린더 그룹 맵에 그 블록들이 할당되었음을 표시하고, 변경된 블록들의 이전 디스크 위치를 무효화 상태로 표시한다. 둘째, 이전 디스크 위치를 지정하는 inode의 블록 포인터(Block pointer)들을 새 디스크 주로로 변경한다. 그 이유는 블록의 물리 번호를 지정하는 메타데이터와 변경될 디스크 위치와 일관성을 유지하기 위해서이다.

모아 쓰기를 통해 각 파일에 포함되는 블록들의 디스크 위치를 변경하더라도 추가적인 디스크 입출력이 발생하지 않는다. 디스크에 저장된 임의의 파일에 대응되는 inode는 파일 크기, 생성 및 변경 시간 등의 파일 속성과 파일에 속하는 블록들의 디스크 위치를 저장하기 위해 12개의 직접 블록 포인터(Direct block pointer) 및 여러 종류의 간접 블록 포인터(Indirect block point)들을 가진다. FFS에서는 파일이 변경될 때 그 디스크 위치가 변경되지 않기 때문에 inode의 블록 포인터의 갱신이 요구되지 않지만 변경 시간을 갱신하기 위해 inode를 디스크로 다시 저장한다. 한편 CW-FFS에서 모아 쓰기 동안 작은 파일의 디스크 위치가 재할당되기 때문에 inode의 블록 포인터들의 변경이 필요하다. CW-FFS는 단지 12개 이내의 블록들을 가지는 작은 파일을 모아 쓰기로 저장하기 때문에 파일의 블록의 디스크 위치가 변경되더라도 inode의 직접 블록 포인터들만 변경하면 된다. 다행히도 변경된 직접 블록 포인터들 값들은 파일 변경 시간을 갱신하기 위해 inode의 재저장 시점에 디스크로 저장하기 때문에 추가적인 디스크 입출력이 발생하지 않는다.

4.3.4 Scatter-gather 디스크 쓰기

일반적으로 저장될 블록들은 파일 캐쉬 내에서 물리적으로 흩어져 있으며, 디스크로 저장하기에 앞서 이 블록들을 미리 예약된 전용 버퍼에 수집해야 한다. 이 수집 방법 중의 하나로 블록 복사 방법이 있지만, CPU 부하를 증가시키기 때문에 CW-FFS는 Scatter-gather 디스크 쓰기 방식을 사용한다. 파일 캐쉬에서 블록을 포함하는 각 버퍼는 물리 메모리의 기본 단위인 프레임(Frame)과 매핑된 가상 메모리 페이지(Page)로 만들어진다. Scatter-gather 쓰기에 앞서, 예약된 전용 버퍼로 활용할 연속적인 가상 메모리 페이지들을 할당받는다. 이 전용 버퍼의 가상 페이지들에 블록들의 물리 프레임들을 재매핑(re-mapping)하며, 이 기법을 페이지 재매핑이라고 한다. 마지막으로, 이 전용 버퍼의 시작 메모리 주소를 디스크 드라이브로 전달하면 디스크는 물리적으로 메모리상에 흩어져있는 물리 페이지들을 메모리 복사 없이 한 번에 디스크로 이동한다.

5. 실 험

5.1 실험 환경 구축

실험을 위해 CPU 속도 1GHz의 Pentium Ⅲ, 256MB 메모리, Fujitsu MPG3307AT 30GB 하드가 탑재된 컴퓨터를 사용했고, 운영체제로 OpenBSD 4.0을 사용하였다. 추가 정보를 <표 1>에 요약하며, 블록 크기와 그룹당 실린더 개수는 OpenBSD 4.0의 기본 설정 값이다. 또한, OpenBSD 4.0은 메타데이터를 저장하기 위해 Soft updates 기법을 기본 설정으로 채택한다. 따라서 실험 대상인 FFS와 CW-FFS 모두 Soft updates 기법으로 메타데이터를 저장한다.

〈표 1〉 실험 환경 정보

디스크		파일시스템	
디스크 크기	30GB	크기	4GB
회전 속도	7200RPM	블록 크기	16KB
평균 탐색 시간	9ms	실린더 그룹 개수	25
		그룹당 실린더 개수	328

CW-FFS와 FFS의 성능을 비교하는데 있어 실제 사용되는 서버와 비슷한 파일시스템 환경을 구축하기 위해 다음과 같은 두 가지 실제 상황들을 고려하였다. 첫째, 실제 서버의 디스크 단편화 상태를 기존 연구 기법[12]으로 재현하였다. 둘째, 실제 서버들의 디스크의 데이터 저장량을 재현하기 위해 디스크 용량의 0%, 10%, 30%, 50%, 70%, 90%에서 파일 쓰기 및 읽기 성능을 테스트하였다. 이 구축된 실제 디스크에서 성능 실험을 위해 랜덤 쓰기 및 읽기 프로그램, OpenBSD 커널 빌드(Compilation), 마지막으로 파일 시스템 테스트용 벤치마크(Benchmark) 프로그램을 실행하였다. 각 실험의 구체적인 내용과 결과는 아래에서 설명한다.

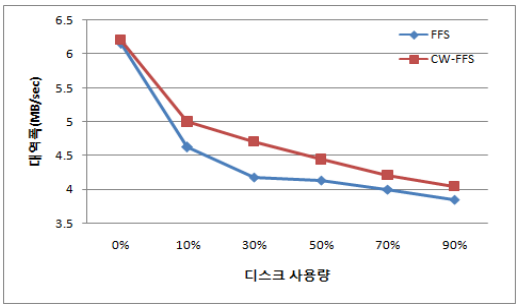
실제 파일 서버의 디스크와 비슷한 정도의 단편화를 가진 디스크에서의 성능을 측정하기 위해 기존 연구[12]에서 제안된 단편화 기법을 사용하였다. 이 연구는 디스크 단편화를 재현(Replay)하는 워크로드(Workload)와 디스크 단편화의 정량적 분석 방법을 소개하고 있다. 이 워크로드는 실제 파일 서버에서 10월간 발생한 파일 및 디렉토리 동작들을 수집하여 얻어졌다. 디스크 단편화 정도는 Layout Score로 표현되며, 파일의 모든 블록들이 연속적이면 score 1, 모두 불연속적이면 score 0인 상태를 나타낸다. 기존 연구[12]에 따르면 서버용 컴퓨터를 오래 사용하면 디스크는 평균 0.8에 가까운 Layout Score를 계속 유지한다. 따라서 본 실험에서는 실제 서버와 유사한 상황에서 실험하기 위해 실험용 디스크에 기존 워크로드를 재현하여 디스크 단편화 정도가 Layout Score 기준으로 평균 0.8이 되도록 하였다.

5.2 파일 쓰기 / 읽기 성능 측정

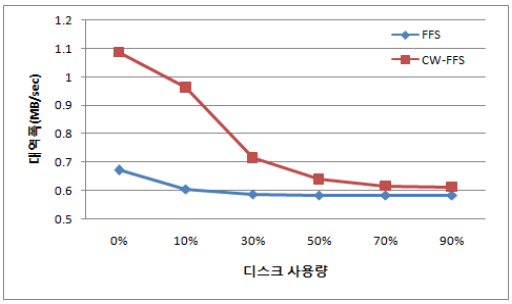
(그림 11)은 0.8의 Layout Score를 가지는 디스크에서 디스크 사용량에 따른 파일 쓰기 성능을 보여준다. 기존 연구[7]와 비슷하게 50의 디렉토리를 생성한 후 디렉토리당 1개 블록(16KB)의 파일 100개씩에 대해 순차적인 생성 및 쓰기 동작을 실행한다. 성능 비교 기준으로, 디스크 탐색 및 회전 시간 대신, 주어진 시간에 디스크로 저장되는 데이터양을나타내는 대역폭을 사용했다. 그 이유는 여러 파일 동작들을 모두 포함하는 전체 실행 시간으로 성능을 비교하면 한 개의 파일 동작이 다른 동작에 영향을 미쳐서 각 파일 동작만의 성능 측정이 불가능하기 때문이다. 이런 이유로 기존 연구 [6][7]에서도 파일 쓰기(또는 읽기 동작)에 대한 실험 척도로 대역폭을 사용했다.

FFS에 비해 CW-FFS의 파일 쓰기 성능이 전체 범위에서 1~13% 정도 개선되었음을 보여준다. 0%의 디스크 사용량에서 CW-FFS는 FFS보다 1% 정도의 성능이 개선되었다. 이처럼 성능 차이가 거의 없는 이유는 임의의 디렉토리에 대한 순차적인 파일 쓰기에서 FFS가 디스크 탐색 및 회전 지연 시간이 거의 발생하지 않도록 디스크 쓰기를 하기 때문이다. 실제 상황과 가장 유사한 30%와 50%의 디스크 사용량에서 각각 13%와 8% 정도의 큰 쓰기 성능 차이가 발생한다. 이 성능 개선은 CW-FFS가 FFS에 비해 디스크 입출력 횟수 및 디스크 회전 지연시간이 매우 감소했기 때문이다. 즉, 디렉토리 별로 100개의 파일들을 순차적으로 저장할 때 대응하는 실린더 그룹에 순차적으로 저장하기 때문에 디스크 탐색 동작보다 디스크 입출력 횟수와 디스크 회전 지연 시간이 성능에 크게 영향을 미친다. FFS의 경우 각 파일 단위로 저장하는 반면, CW-FFS는 여러 파일들을 모아 쓰기로 저장하기 때문에 예상대로 디스크 입출력 횟수 및 디스크 회전 지연 시간이 크게 감소하였기 때문에 성능이 현저히 개선되었다. 하지만, 70%와 90%에서 디스크 사용량이 증가할수록 연속적인 빈 공간이 부족해지며 모아 쓰기의 효과가 감소한다.

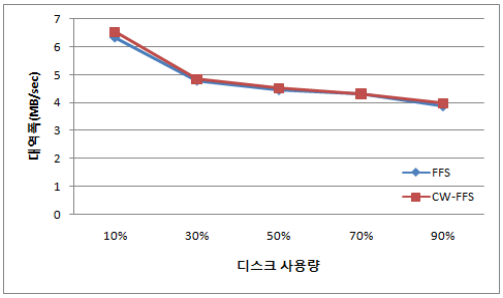
(그림 12)는 순차적 쓰기와 마찬가지로 0.8의 Layout Score를 가지는 디스크에서 디스크 사용량에 따른 랜덤 쓰기의 성능을 보여준다. 이 실험은 50개의 디렉토리를 생성한 후 랜덤하게 디렉토리를 선택하여 이 디렉토리에 파일을 생성하고 파일에 쓰는 동작을 반복적으로 수행한다. 디스크 사용량 전체 범위에서 FFS보다 CW-FFS의 쓰기 성능이 5 ~ 60% 정도 개선되었음을 볼 수 있다. 특히, 연속적인 빈 공간이 충분한 0 ~ 10%의 디스크 사용량에서 약 60%의 성능 개선이 이루어졌다. 그 성능 개선 이유는 CW-FFS가 FFS에 비해 현저히 디스크 탐색 횟수가 감소했기 때문이다. 즉, 각 파일이 디렉토리 단위로 번갈아 저장될 때 FFS는 각 디렉토리에 대응하는 실린더 그룹 단위로 번갈아 가며 파일



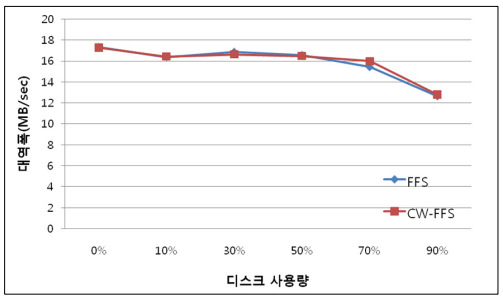
(그림 11) 순차적 파일 쓰기 성능



(그림 12) 랜덤 파일 쓰기 성능



(그림 13) 순차적인 읽기 성능 측정



(그림 14) 큰 파일 쓰기 성능

들을 저장하는 반면에 CW-FFS는 각 실린더 그룹에 번갈아 저장될 파일들을 모아서 단일 디스크 쓰기로 저장하기 때문에 디스크 탐색 횟수가 크게 감소하였다. 결국 파일 참조 시 가장 큰 시간을 소비하는 디스크 탐색을 감소시켰기 때문에 FFS와의 성능 차이가 현저히 생겼다.

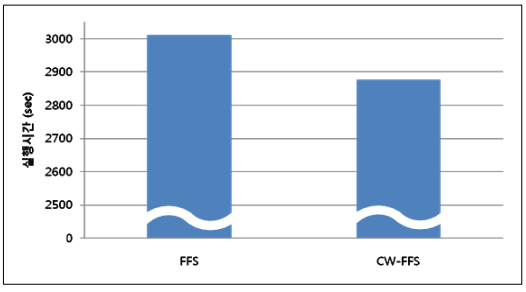
하지만, 디스크 사용량이 점차 증가함에 따라 연속적인 빈 공간의 감소로 인해 모아 쓰기 동안에 저장되는 파일 또는 블록 개수가 작아진다. 이로 인해 30%의 디스크 사용량부터 파일 쓰기 성능이 감소했으며, 특히 빈 공간이 매우 부족한 70 ~ 90%의 디스크 사용량에서 5% 정도의 적은 성능 개선이 이루어졌다. 다행히도 기존 연구[13]에서 조사된 바에 의하면 서버 및 개인용 컴퓨터의 디스크는 보통 약 40% 정도의 사용량을 가진다. 이 수치와 유사한 30%와 50%의 디스크 사용량에서 CW-FFS는 FFS에 비해 각각 22%와 10% 정도 성능이 개선되었다. 이를 통해 CW-FFS가 상용 컴퓨터에서 작은 파일 쓰기 성능을 크게 향상시킬 수 있다는 것을 확인할 수 있다.

(그림 13)은 앞의 랜덤 파일 쓰기 후의 디스크에서 사용량별로 각 디렉토리의 모든 파일들을 순차적으로 읽을 때의 성능을 보여준다. 성능 평가 기준으로 주어진 시간에 디스크에서 읽은 데이터양을 나타내는 대역폭을 사용하였다. 그 이유는 한 디렉토리에 포함된 파일들을 순차적으로 읽을 때 이 파일들이 한 실린더 그룹에 배치되느냐 아니면 여러 실린더 그룹에 흩어져 배치되어 있느냐가 디스크 대역폭에 큰 영향을 미치기 때문이다. 그 성능 결과는 CW-FFS가 FFS와 거의 동일한 성능을 가진다는 것을 보여준다. 그 이유는 CW-FFS가 모아 쓰기로 저장되는 파일들을 FFS처럼 동일 실린더 그룹에 저장하기 때문이다. 결국 CW-FFS는 모아 쓰기로 여러 작은 파일들을 저장하여 쓰기 성능 향상을 향상하였을 뿐만 아니라 기존 FFS의 실린더 그룹 개념의 장점을 그대로 유지한다.

(그림 14)는 0.8의 Layout Score를 가지는 디스크에서 디스크 사용량에 따른 큰 파일 쓰기의 성능을 나타낸다. 이 실험은 50개의 디렉토리를 생성한 후 각 디렉토리 단위로 4개의 100MB의 큰 파일들 대상으로 생성 및 쓰기 동작을 수행한다. 성능 결과로 CW-FFS와 FFS는 큰 파일 쓰기 성능이 거의 비슷함을 알 수 있다. 그 이유는 CW-FFS는 모아 쓰기로 작은 파일들의 쓰기 성능만을 개선하고 큰 파일에 대해서는 FFS의 동작을 그대로 사용하기 때문이다. 결국 CW-FFS는 동일 디렉토리의 파일들을 순차적으로 읽을 경우와 큰 파일들을 저장할 경우에 FFS의 성능을 유지하며 작은 파일의 경우는 FFS보다 성능이 향상되었다.

5.3 커널 빌드 실행의 파일 성능

(그림 15)는 OpenBSD 커널 빌드를 수행했을 때 걸린 시간을 초(sec) 단위로 보여주고 있다. 이 실험에서 총 1064개의 소스 파일들로부터 17.64 MB인 오브젝트(*.o) 파일들이 생성된다. 커널 소스 파일들은 대부분 64KB보다 작은 크기의 작은 파일들로 구성된다. 빌드 과정에서 수십 개의 디렉토리가 생성되고 각 디렉토리별로 소스 파일들을 순차적으로 빌드하여 오브젝트 파일들을 생성한다. 실험 결과로 CW-FFS가 약 5% 정도 성능이 향상된 것을 볼 수 있다. 이러한 성능 향상 이유는 순차적 파일 쓰기처럼 디스크 입출력의 횟수가 감소했기 때문이다. 즉, 생성되는 오브젝트 파일들을 각 디렉토리에 순차적으로 저장할 때 FFS는 이 파일들을 디렉토리에 대응하는 실린더 그룹에 파일 별로 저장하는 반면 CW-FFS는 여러 파일들을 모아 쓰기로 디스크에 저장하기 때문에 디스크 입출력이 감소하여 성능이 향상되었다. 하지만, 성능 향상이 크지 않은 이유는 빌드 과정에서 디렉토리들을 번갈아 가면서 오브젝트 파일을 만드는 것이 아니라 디렉토리 단위로 오브젝트 파일을 순차적으로 만들어 저장하기 때문이다.



(그림 15) 커널 빌드 성능

5.4 Postmark 벤치마크

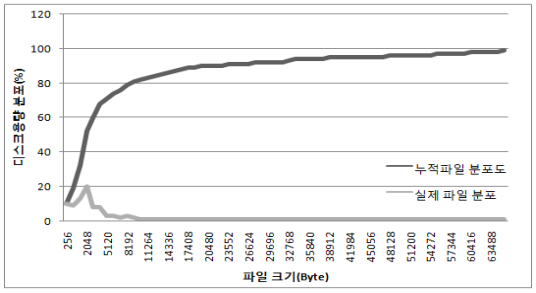
Postmark[14]는 파일 시스템 성능을 측정하는 대표적인 벤치마크 프로그램 중 하나이다. Postmark는 짧은 수명의 작은 파일들(생성 후 제거될 때까지의 시간이 짧은 파일들)의 읽기, 쓰기, 추가(append) 및 삭제 동작들에 대한 통합

〈표 2〉 Postmark의 파라미터

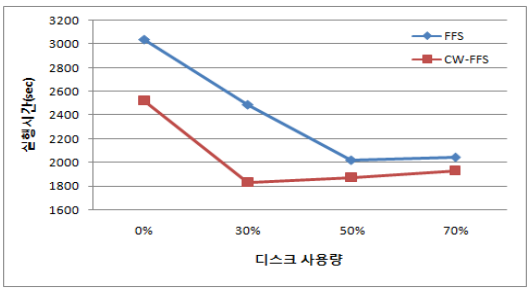
파라미터	FSL	R-FSL
파일 개수	20,000	25,000
디렉토리 개수	200	200
파일 크기	512B~10KB	실제 파일 분포
트랜잭션 개수	200,000	50,000
파일 참조 비율	동일	동일

성능(overall performance)을 측정한다. 기존 파일시스템 연구에서 주로 사용되었던 Postmark 파라미터 구성(Configuration) [15]을 본 실험에서 사용하였고, 그 파라미터들은 <표 2>와 같다.

파일시스템의 성능을 측정하는 방법으로 FSL 파라미터 구성이 주로 사용된다. 하지만, FSL의 파일 크기가 실제 서버의 디스크에 저장된 파일 크기 분포와 다르다. 실제 서버의 디스크 상태와 유사한 형태로 실험하기 위해 기존 연구 [1]에서 제안된 "실제 파일 크기 분포"를 추가한 R-FSL(Real FSL)을 구성했다. (그림 16)은 R-FSL에 사용된 파일 크기 분포도를 나타낸다. "실제 파일 분포"는 실제 사용된 파일의 크기 분포를 나타내며, "누적 파일 분포도"는 실제 파일 분포에 대한 누적 파일 분포를 보여준다. 8Kb 이하의 파일들이 79%이며 10Kb 이하의 파일들이 80%를 차지한다. 이 파일 분포도를 가지는 확률적인 함수를 Postmark에 추가하여 실험을 수행했다. 또한, R-FSL은 FSL과 유사하지만 트랜잭션(Transaction) 개수가 약간 줄었고, 초기 생성 파일의 개수가 증가되었다. R-FSL은 초기에 200개의 디렉토리들을 생성하고, 디렉토리당 25,000개의 파일을 대상으로 순차적으로 쓰기 동작을 수행한다. 이 단계를 순차적 파일 생성/쓰기 단계라고 한다. 그 후에 트랜잭션 단계에서 랜덤하게 디렉토리를 선택하여 파일 읽기, 쓰기, 추가(Append.



(그림 16) 파일 분포도 (File Distribution)



(그림 17) Postmark 측정 결과

〈표 3〉 Postmark의 트랜잭션 단계에서의 결과
각 파일 동작의 초 당 완료의 횟수

	FFS				CW-FFS			
	0%	30%	50%	70%	0%	30%	50%	70%
쓰기	9	12	14	13	12	15	15	15
읽기	9	11	13	13	12	15	15	14
추가	9	12	14	13	12	15	15	14
삭제	9	11	13	13	12	15	15	14

즉, 파일에 블록 추가 동작임), 삭제를 한다. 이때, 순차적인 파일 쓰기 생성 단계와 트랜잭션 단계의 파일 쓰기 비율은 25,000 대 50,000 (1대 2) 비율로 트랜잭션 단계에서 더 많은 파일 쓰기가 일어난다.

(그림 17)은 0.8의 Layout Score를 가지는 디스크에서 0%, 30%, 50%, 70%의 데이터 사용량에 대한 R-FSL의 파일 동작들의 통합 실행 시간을 보여준다. 실험 결과가 보여주듯이, 데이터 사용량의 전반적인 범위에서 CW-FFS가 FFS보다 6 ~ 27% 정도 실행 시간이 개선되었으며, 특히 30%의 디스크 사용량에서 27% 정도의 큰 실행 시간 차이가 발생했다. 이 실행 시간의 개선은 <표 3>에서 보여주듯이 Postmark의 트랜잭션 단계에서 파일 쓰기 외에 다른 파일 동작들의 성능이 부수적으로 개선되었기 때문이다. CW-FFS에서 여러 파일들에 대한 모아 쓰기는 예상처럼 파일 쓰기 및 파일 추가 성능을 향상시켰다. 이 성능 향상은 순차적인 파일 생성/쓰기 단계에서 디스크 입출력 횟수 및 디스크 회전 지연 시간이 감소하였고, 랜덤하게 파일들을 저장하는 트랜잭션 단계에서 디스크 탐색 횟수가 크게 감소했기 때문이다. 또한, 파일 읽기 및 제거 동작의 성능 개선 이유는 파일 읽기 및 추가 동작 중에 발생할 수 있는 디스크 탐색이 여러 파일들의 모아 쓰기로 감소했기 때문이다.

디스크 사용량이 50%와 70%의 경우 디스크에 빈 공간이 부족하여 성능 향상의 정도가 작다. 하지만, 기존 연구[13]는 실제 서버의 사용량이 약 40% 정도라고 보고하고 있다. 따라서 30%와 50% 사이의 디스크 사용량에서 최대 성능 개선을 얻을 수 있는 CW-FFS가 기존 FFS을 쓰는 것보다 실제 서버 및 개인용 컴퓨터의 파일시스템으로 유리하다는 점을 예상할 수 있다.

6. 결 론

본 논문에서 생성 및 변경된 작은 파일들을 디렉토리 단위의 지역성을 이용하여 모아 쓰기로 디스크에 저장하는 파일시스템인 CW-FFS를 제안했다. CW-FFS를 실제 서버 환경과 비슷한 디스크 상태와 파일 참조 패턴으로 실험하고 검증하였으며, 그 실험을 통해 기존 시스템보다 전반적으로 우수한 파일 쓰기 성능을 가지고 있다는 것을 입증하였다. 특히 다중 디렉토리들의 파일들을 번갈아 가며 저장하는 랜덤 파일 쓰기 실험에서 디스크 탐색 횟수의 감소 때문에 발생하는 성능 향상이 눈에 띄게 보인다. 또한 파일 쓰기 성능의 큰 향상에도 불구하고 읽기 성능의 저하가 없었다.

제안하는 CW-FFS의 구현 복잡도가 크지 않기 때문에 이 기법을 BSD 뿐만 아니라 이를 확대하여 Linux 파일시스템인 Ext2, Ext3에도 적용할 수 있고, 더 나아가 현재 최대 상업용 운영체제인 Windows의 파일시스템인 NTFS에서도 적용하여 파일시스템 성능을 향상시킬 수 있을 것으로 예상한다.

참 고 문 헌

[1] A. S. Tanenbaum, J. N. Herder and H. Bos, "File Size Distribution on UNIX Systems-Then and Now," ACM SIGOPS Operating Systems Review, Vol.40, No.1, pp.100-104, 2005.

[2] M. McKusick, W. Joy, S. Leffler and R. Fabry, "A Fast File System for UNIX," ACM Transactions on Computer Systems, Vol.2, No.3, pp.181-197, 1984.

[3] H. Huang, W. Hung and K. G. Shin, "FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption," Proceedings of the 20th ACM Symposium on Operating System Principles, pp.263-276, 2005.

[4] M. Seltzer, K. A. Smith and H. Balakrishnan, "File System Logging versus Clustering: A Performance Comparison," Proceedings of the USENIX Annual Technical Conference, pp.249-264, 1995.

[5] M. McKusick and G. Ganger, "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem," Proceedings of the USENIX Annual Technical Conference, pp.1-17, 1999.

[6] M. Seltzer, G. Ganger, M. McKusick and K. A. Smith, "Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems," Proceedings of the USENIX Annual Technical Conference, pp.71-84, 2000.

[7] M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-structured File System," Proceedings of the 13th ACM symposium on Operating Systems Principles, pp.1-15, 1992.

[8] J. M. Neefe, "Improving File System Performance with Adaptive Methods," Ph.D. Dissertation, University of California at Berkeley, 1999.

[9] K. A. Smith and M. Seltzer, "A Comparison of FFS Disk Allocation Policies," Proceedings of the USENIX Annual Technical Conference, pp.15-26, 1996.

[10] D. Roselli, J. R. Lorch and T. E. Anderson, "A Comparison of File System Workloads," Proceedings of the USENIX Annual Technical Conference, pp.41-54, 2000.

[11] S. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," Proceedings of the USENIX Annual Technical Conference, pp.260-269, 1986.

[12] K. A. Smith and M. Seltzer, "File System Aging - Increasing the Relevance of File System Bench marks," Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), pp.203-213, 1997.

[13] N. Agrawal, W. Bolosky, J. Douceur and J. Loarch, "A Five-Year Study of File-System Metadata," Proceedings of the 5th USENIX Conference on File and Storage Technologies, pp.31-45, 2007.

[14] J. Katcher, "PostMark: A New Filesystem Benchmark," Network Appliance, Technical Report TR-30-22, 1997.

[15] A. Joukov, A. Traeger, C. P. Wright and E. Zadok, "Benchmarking File System Benchmarks," Stony Brook University, Technical Report FSL-05-04, 2005.

이 경 재

e-mail : ptp777@kw.ac.kr

2007년 광운대학교 컴퓨터 소프트웨어학과 (학사)

2007년~현 재 광운대학교 일반대학원 컴퓨터과학과 석사과정

관심분야 : 운영체제, 임베디드시스템 등

안 우 현

e-mail : whahn@kw.ac.kr

1996년 경북대학교 전자공학과(학사)

1998년 KAIST 전기 및 전자공학과(석사)

2003년 KAIST 전자전산학과(박사)

2003년~2005년 삼성전자 기술총괄 소프트웨어연구소 책임 연구원

2006년~현 재 광운대학교 컴퓨터 소프트웨어학과 조교수

관심분야 : 운영체제, 임베디드시스템, 시스템소프트웨어, 모바일 SW플랫폼 등

오 재 원

e-mail : jwoh@catholic.ac.kr

1997년 서울대학교 계산통계학과(학사)

1999년 서울대학교 대학원 전산과학과 (석사)

2004년 서울대학교 대학원 전기컴퓨터 공학부(박사)

2004년~2007년 삼성전자 기술총괄 소프트웨어연구소 책임 연구원

2007년~현 재 가톨릭대학교 컴퓨터정보공학부 전임강사

관심분야 : 소프트웨어 품질, 소프트웨어 공학, 모바일 SW플랫폼, 시스템 소프트웨어 등