Journal of Korea Multimedia Society Vol. 8, No. 7, July 2005(pp. 924-942)

대규모 동적 해싱 디렉토리의 구현 및 평가

김신우†, 이용규††

요 약

최근 대용량 데이터의 저장과 검색을 위해서 리눅스 클러스터 파일시스템의 디렉토리는 점차 대규모로 되고 있다. 그들 중 대표적인 GFS의 디렉토리는 동적 해싱의 하나인 빠른 검색을 지원하는 확장 해싱을 이용하고 있다. GFS의 디렉토리의 주된 특징은 모든 리프 노드들이 트리의 동일한 레벨에 놓이는 플랫 구조를 가지고 있다. 그러나 리프 노드에서 오버플로우가 발생하게 되면 레벨이 하나 증가하면서 갑자기 데이터 블록의 임의의 평균 접근 시간이 길어지는 단점이 있다. 또 다른 동적 해싱으로는 선형해싱을 들 수 있고, 이는 확장 해싱보다 파일 접근에 좋은 성능을 보여준다. 본 논문에서는 플랫구조보다 더 좋은 접근 성능을 가지는 세미 플랫 구조를 이용하여, 대규모 리눅스 클러스터 파일 시스템을 위한 확장 해싱 디렉토리와 선형 해싱 디렉토리를 설계 및 구현하고 그들의 성능을 비교한다. 성능 평가 결과, 파일의 삽입 면에서는 선형 해싱 기반의 디렉토리가 좋은 성능을 보였으나, 공간 활용 면에서는 확장 해싱 기반의 디렉토리가 좋은 성능을 보였다.

An Implementation and Evaluation of Large-Scale Dynamic Hashing Directories

Shin Woo Kim†, Yong Kyu Lee††

ABSTRACT

Recently, large-scale directories have been developed for LINUX cluster file systems to store and retrieve huge amount of data. One of them, GFS directory, has attracted much attention because it is based on extendible hashing, one of dynamic hashing techniques, to support fast access to files. One distinctive feature of the GFS directory is the flat structure where all the leaf nodes are located at the same level of the tree. But one disadvantage of the inode structure is that the height of the inode tree has to be increased to make the tree flat after a byte is inserted to a full tree which cannot accommodate it. Thus, one byte addition makes the height of the whole inode tree grow, and each data block of the new tree needs one more link access than the old one. Another dynamic hashing technique which can be used for directories is linear hashing and a couple of researches have shown that it can get better performance at file access times than extendible hashing. In this research, we have designed and implemented an extendible hashing directory and a linear hashing directory for large-scale LINUX cluster file systems and have compared performance between them. We have used the semi-flat structure which is known to have better access performance than the flat structure. According to the results of the performance evaluation, the linear hashing directory has shown slightly better performance at file inserts and accesses in most cases, whereas the extendible hashing directory is somewhat better at space utilization.

Key words: File System(파일 시스템), Dynamic Hashing(동적 해싱), Extendible Hashing(확장 해싱), Linear Hashing(선형 해싱), Semi-flat Directory Structure(세미플랫 구조)

※ 교신저자(Corresponding Author) : 이용규, 주소 : 서울시 중구 필동 3가 26번지(100-715), 전화 : 02)2260-3828, FAX : 02)2265-8742, E-mail : yklee@dgu.edu
접수일 : 2004년 9월 1일, 완료일 : 2005년 1월 4일

† 준회원, 동국대학교 컴퓨터멀티미디어공학과 강사
(E-mail : purian@dgu.edu)
†† 종신회원, 동국대학교 컴퓨터멀티미디어공학과 교수
※ 본 연구는 동국대학교 논문게재장려금 지원으로 이루어졌음.

1. 서 론

최근 기업들의 데이터가 매년 75%~150%의 비율로 증가함에 따라 기업이 요구하는 스토리지의 양은 기하급수적으로 늘어나게 되었다[1]. 따라서, 네트워크 기반 공유 저장 장치를 이용하여 작은 규모의 컴퓨터들을 클러스터로 연결한 하나의 통합된 시스템을 구축하려는 연구가 활발히 진행 중에 있다. 그 결과 xFS[2], Frangipani[3] 등의 클러스터 파일 시스템들이 구현되었고, 이들은 네트워크 연결형 파일 시스템으로 기존의 단일 시스템에 비하여 시스템 구축 비용 뿐만 아니라, 시스템의 유용성, 확장성, 고장 감내의 측면 등에서 장점을 가지고 있다.

이러한 파일 시스템들이 사용하는 대용량의 저장 장치로 별도의 고속 데이터 전용 네트워크인 화이버 채널[4]을 통해 클라이언트와 저장 장치들을 연결하는 SAN(Storage Area Network)을 들 수 있으며, 최근에 자료저장 시스템의 시장 규모가 확대되면서 업체들이 빠르게 SAN 관련 솔루션들을 제공하기 시작하고 있다. 예를 들면, IBM사의 Tivoli[5]와 컴팩사의 VersaStor[6], 그리고 베리타스사의 SANPoint [7] 등이 있는데, 이들은 여러 클라이언트들이 SAN에 부착된 저장장치들을 공유할 수 있도록 하고 있다.

또한 SAN 파일 시스템으로 기존의 상용 시스템들이 제공하는 운영체제 대신에 소스가 공개된 LINUX를 활용함으로써 클러스터 시스템의 구축 비용을 더욱 낮추려는 시도가 활발히 전개되고 있다. 그 대표적인 예로 미네소타 대학교에서 구현된 GFS(Global File System)[8,9], 한국전자통신연구원에서 개발 중인 SANtopia[10,11], 매크로임팩트의 SANique CFS[12], IBM의 GPFS(General Parallel File System)[13] 등을 들 수 있다. 이와 같은 SAN 기반 대규모 파일 시스템들은 별도의 서버를 두지 않고 분산된 클라이언트가 메타데이터를 직접 관리하면서 저장 장치들에 접근하여 모든 저장 장치들에 접근을 일정하도록 하여 하나의 서버에 업무가 집중되는 현상을 막을 수 있다[14].

SAN을 이용한 대규모 파일 시스템으로 가장 대표적인 GFS의 주된 특징으로는 inode의 플랫 파일 구조(Flat File Structure)를 가지고 있어, 모든 데이터 블록의 임의 접근 시간이 같다는 장점이 있으나 항상 플랫 구조를 유지하여야 하므로 리프 노드에서 오버플로우가 발생하게 되면 갑자기 데이터 블록의 임의의 평균 접근 시간이 길어지는 단점이 있다. 이에, 데이터 블록 당 접근 시간을 기존보다 줄이기 위해 두 레벨까지 데이터 블록을 할당하는 새로운 세미플랫 파일 구조(Semi-flat File Structure)가 제안되었다[15].

한편, 대부분의 UNIX 시스템[16]에서는 디렉토리 내의 파일 이름들을 파일의 생성 순서로 유지하므로 특정 파일을 디렉토리 내에서 탐색할 때 순차적으로 검색하여야만 한다. 따라서, 많은 파일 이름들을 포함한 SAN 기반 대규모 파일 시스템에서 UNIX 시스템과 같은 디렉토리 구조를 갖게 되면, 특정 파일을 검색하는 데 많은 시간이 소요될 수 있다. 그러므로, SAN 기반 대규모 파일 시스템에서는 파일의 수가 많고 적음에 상관없이 모두 수용 가능하며, 비록 많은 수의 파일이 존재하더라도 빠른 검색이 가능한 동적 해싱(Dynamic Hashing)을 이용하여 디렉토리를 관리한다.

동적 해싱에 관한 연구는 그동안 활발히 진행되어 왔고, 특히 확장 해싱(Extendible Hashing)[17-20]과 선형 해싱(Linear Hashing)[20-22]에 대한 비교 분석 연구[23]가 있었으나, 이는 50,000개의 엔트리를 가진 소규모 디렉토리에 대한 결과로 실제 대규모 디렉토리에서도 그 비교 분석 결과가 적용되는지는 불분명하다. 한편, 이러한 동적 해싱을 SAN 기반 대규모 파일 시스템에서 적용한 사례로는 확장 해싱을 이용하여 구현된 시스템인 GFS를 들 수 있으나, 선형 해싱을 이용하여 적용된 시스템 예를 찾아보기 힘들다.

본 논문에서는 SAN 기반 대규모 파일 시스템들 중 대표적인 GFS의 플랫 구조를 이용한 확장 해싱 디렉토리의 파일의 크기가 커질 때 갑자기 데이터 블록의 임의의 평균 접근 시간이 길어지는 단점을 보완하기 위해서, 세미플랫 구조를 이용한 확장 해싱 디렉토리 구조와 세미플랫 구조를 이용한 선형 해싱 디렉토리 구조를 설계 및 구현한다. 구현된 세미플랫 구조를 이용한 확장 해싱 디렉토리와 선형 해싱 디렉토리, 그리고 성능 비교 분석을 위해 구현된 플랫 구조를 이용한 확장 해싱 디렉토리와 선형 해싱 디렉토리의 성능 평가를 통해, 확장 해싱과 선형 해싱 자체를 비교 분석하고 세미플랫 구조를 이용한 해싱 디렉토리가 플랫 구조를 이용한 해싱 디렉토리에 비하여

대규모 디렉토리에서 삽입, 검색 등에서 우수함을 보인다.

2. 관련 연구

본 절에서는 SAN을 이용한 대규모 파일 시스템에서의 inode 구조에 대해서 알아보고, 동적 해싱인 확장 해싱과 선형 해싱에 대해서 설명한다. 그리고 최근의 SAN 파일 시스템들의 디렉토리 구조에 대해서 살펴본다.

2.1 대규모 파일 시스템에서의 Inode 구조

SAN을 이용한 대규모 파일 시스템에서의 inode 구조인 플랫 구조와 세미플랫 구조는 다음과 같다.

(1) 플랫 구조

그림 1은 GFS[8,9]의 inode의 플랫 구조(Flat Structure)로, 모든 데이터 블록들은 트리의 높이와 같은 리프 레벨에만 위치한다. 이는 모든 데이터 블록의 임의 접근 시간이 같아지도록 하며, 매우 큰 파일의 경우는 트리의 높이를 증가시키면 되므로 파일의 크기에 제한이 없어지는 장점을 갖는 반면, 항상 플랫 구조를 유지하여야 하므로 파일의 크기가 커질수록 데이터 블록의 임의의 평균 접근 시간이 길어지는 단점이 있다.

(2) 세미플랫 구조

그림 2는 새로운 세미플랫(Semi-flat) 구조[15]로, 모든 데이터 블록들이 동일한 레벨에 있지 않고 트리의 높이 h와 $(h-1)$에 걸쳐 있음을 볼 수 있다. 세미플랫 구조에서는 모든 데이터 블록들이 파일의 크기에 따라 그림 1처럼 플랫 구조를 가질 수도 있고 그림 2처럼 두 레벨에 걸쳐 있을 수도 있다. 세미플랫 구조에서는 새로운 데이터 블록이 추가되었을 때, 플랫 구조에서처럼 트리 높이의 증가로 인하여 현재 레벨에 위치한 데이터 블록들을 다음 레벨로 전부 이동할 필요가 없기 때문에 데이터 블록의 임의 접근에서 더 좋은 성능을 나타낸다.

2.2 동적 해싱

동적 해싱 방법인 확장 해싱과 선형 해싱은 다음과 같다.

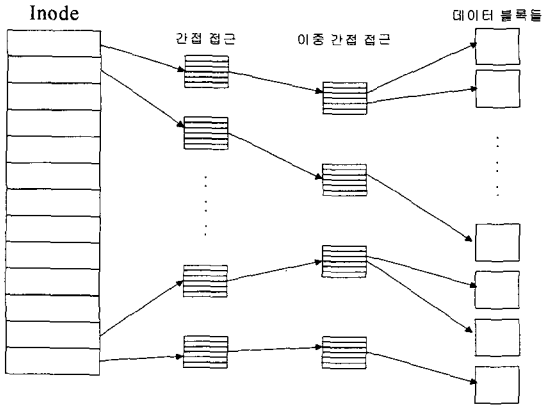


그림 1. GFS inode의 플랫 구조

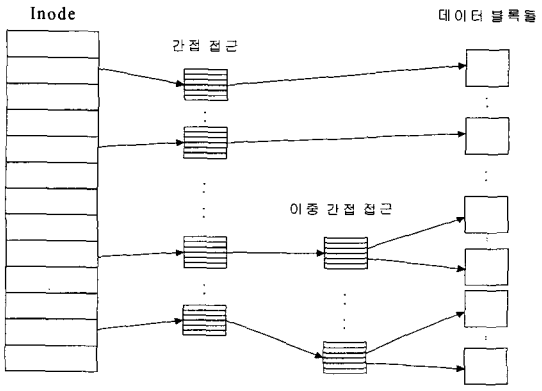


그림 2. Inode의 세미플랫 구조

(1) 확장 해싱

확장 해싱(Extendible Hashing)[17-20]은 두 단계의 조직인 해시 테이블과 리프의 집합으로 구성되어 있다. 해시 테이블은 정수 값(d)을 갖는 헤더와 리프에 대한 2^d개의 포인터로 되어 있으며, 리프는 헤더를 가지는 엔트리 블록으로 되어있는데, 헤더는 엔트리 블록에 대한 비교하는 비트의 자리수를 나타낸다.

확장 해싱은 해시 함수를 이용하여 해시 값을 알아내어, 해시 값의 처음 d비트를 해시 테이블에 대한 인덱스로 사용하여 해시 테이블에서 엔트리 블록을 찾아서 해당 엔트리에 접근한다. 이때 엔트리 블록에서 오버플로우가 발생하면 엔트리 블록이 분할되거나 해시 테이블이 증가하며 확장된다

확장 해싱은 오버플로우가 발생한 블록만 분할하므로 분할 회수를 줄이고 불록의 적재율을 높이는 장점이 있는 반면, 해시 테이블을 통해서 디렉토리

엔트리 블록에 접근해야 하는 단점이 있다.

(2) 선형 해싱

선형 해싱(Linear Hashing)[20-22]은 확장 해싱과 달리 해시 테이블의 존재 없이 리프의 집합으로 구성되어 있으며, 리프는 엔트리 블록으로 되어있다. 선형 해싱은 해시 함수를 이용하여 해시 값을 알아내어 엔트리 블록을 찾는데, 이때 엔트리 블록에서 오버플로우가 발생하면 스플릿 포인터 *sp*를 이용하여 확장한다.

선형 해싱은 별도로 해시 테이블이 존재하지 않으므로 바로 디렉토리 엔트리 블록 접근이 가능하다는 장점이 있는 반면, 오버플로우가 발생하지 않은 불록에서 분할이 되는 경우가 있어 이에 별도의 오버플로우 블록이 요구되므로 저장 공간을 많이 사용하게 되는 단점이 있다.

2.3 SAN 파일 시스템

SAN 파일 시스템들의 대부분이 대규모 데이터 수용을 고려하여 확장이 가능한 inode 구조와 확장 해싱을 이용한 디렉토리 구조를 유지하고 있으며 다음과 같다.

(1) GFS

미네소타 대학교에서 구현된 GFS(Global File System)[8,9]는 기존의 UNIX 파일 시스템을 개선하여 SAN 환경에서 LINUX의 클러스터 파일 시스템으로 사용하도록 미네소타 대학교에서 개발되었다.

GFS에서는 inode 구조를 플랫 구조로 유지하고, 확장 해싱을 이용하여 디렉토리를 관리한다. 확장 해싱은 대부분의 UNIX 시스템에서 디렉토리 내의 파일 이름들을 특별한 순서 없이 파일의 생성 순서로 유지하므로 특정 파일의 이름을 디렉토리 내에서 탐색할 때 순차적으로 검색하여야만 하는 단점을 극복하기 위해서 사용한다. 이는 디렉토리가 적은 수의 파일들로부터 많은 수의 파일들까지 자유롭게 수용할 수 있도록 하고, 또한 파일들이 많을 경우에도 해싱의 특성상 빠른 검색이 가능하게 한다.

(2) SANtopia

국내 한국전자통신연구원에서 개발한 SANtopia [10,11]는 SAN 환경 기반의 리눅스 클러스터 파일 시스템이다. SANtopia은 서로 다른 레벨을 가질 수 있는 동적 다단계 inode 구조를 유지하며, GFS와 다름없이 저장된 대규모 데이터의 빠른 검색을 위해서 확장 해싱을 이용하여 디렉토리를 관리하고 있다.

(3) SANique CFS

SANique CFS(Cluster File System)[12]는 SAN에 연결된 컴퓨터들 간에 고속의 파일 공유를 가능하게 하며, 분산 락 관리 방식을 채택하고 있는 유일한 상업용 클러스터 파일시스템이다. SANique CFS는 다중 간접 인덱싱 방식으로 inode 구조를 유지하며, 디렉토리 내부의 파일 혹은 디렉토리에 대한 정보를 신속하게 검색하기 위해서 확장 해싱을 이용한다.

(4) GPFS

GPFS(General Parallel File System)[13]는 SAN 기반 Linux 클러스터 환경의 모든 노드에서 데이터에 액세스할 수 있도록 지원하는 고성능 공유 디스크 파일 시스템으로, 대규모의 디렉토리를 지원하기 위해서 확장 해싱을 이용하여 효과적인 검색을 한다.

3. 확장 해싱을 이용한 디렉토리 구조

SAN 파일 시스템에서는 확장 해싱을 이용하여 디렉토리를 관리할 수 있다. 디렉토리에 삽입하고자 하는 디렉토리 엔트리 이름을 이용하여 해시 함수를 통해 해시 값을 구하고, 이를 활용하여 직접 저장할 데이터 블록의 주소를 찾아 삽입, 삭제 및 탐색을 할 수 있다. 확장 해싱은 데이터를 순차 접근이 아닌 직접 접근하므로 빠른 연산이 가능하다.

3.1 해시 함수

확장 해싱에서는 해시 값을 주소로 엔트리 블록에 접근하는데, 같은 해시 값이 많아지게 되면 같은 블록에 데이터가 집중(Clustering)되어 잦은 오버플로우가 발생하게 된다. 따라서, 블록들이 데이터를 균등하게 수용할 수 있도록 같은 해시 값을 적게 생성하는 해시 함수가 요구된다. 이를 위하여 본 논문에서 사용하는 해시 함수는 데이터 통신에서 에러 검출 코드로 활용되는 CRC-32 코드(32-bit Cyclic Redundancy Check Code)[24]를 사용한다. CRC-32는 다른 CRC 함수와 다르게 정밀하게 에러를 탐지할 수 있다는 장점 즉, 중복되지 않는 값을 얻을 수 있기

때문에, 연산 후 생성되는 32비트의 값을 해시 값으로 이용한다. 그림 3은 파일의 이름을 CRC 해시 함수에 적용하여 해시 값을 알아내는 과정이다.

처음에 파일 이름을 입력받아서 이를 비트로 변환한다. 이 때 변환된 비트는 최소 32비트 이상이 되며, CRC-32에서 비트 변환에 사용하는 키 값인 0x04c11db7로 나눈다. 이때, 32비트 미만의 나머지가 생기고 이를 해시 값으로 사용하기 위해서 상위 비트를 0으로 채우면 32비트의 해시 값을 구할 수 있다.

3.2 확장 해싱 디렉토리

디렉토리 구조는 파일의 수를 고려하여 결정한다. 파일의 수가 적을 때는 inode 블록에 디렉토리 엔트리를 함께 저장하고, 파일의 수가 많아져 inode 블록에서 오버플로우가 발생하면 확장 해싱을 이용하여 디렉토리 엔트리를 저장하며, 간접 접근 해시 테이블 구조로 변환되면 임의의 접근 시간이 좋은 inode의 세미플랫 구조를 이용하며 확장한다. 디렉토리는 세 경우로 나누어 관리한다.

(1) Inode 블록에 통합된 디렉토리 엔트리

그림 4는 inode 블록에 직접 디렉토리 엔트리들이 저장되어 있는 것을 보여준다. 디렉토리 엔트리 수가 적을 때는 디렉토리 엔트리들을 inode 블록에 직접 저장(Stuffing)함으로써 inode 블록 한 번의 접근으로 디렉토리 엔트리를 검색할 수 있다.

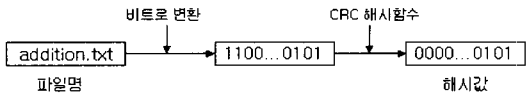


그림 3. 해시 함수를 적용하는 과정

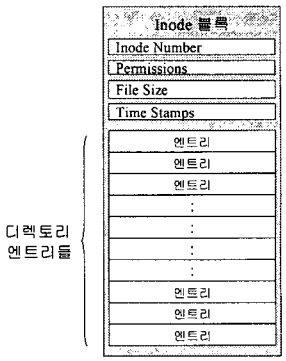


그림 4. Inode 블록에 통합된 디렉토리 엔트리

한편, 디렉토리 엔트리 블록의 헤더 크기가 112 bytes라 가정하고, 보통 파일 이름의 길이를 8bytes로 한 정보를 포함한 하나의 엔트리 크기는 24bytes로 볼 수 있다. 한 블록의 크기를 4KB로 하였다고 가정하면, 이때 166개의 디렉토리 엔트리들을 저장할 수 있다. 그러나 이 블록에 새로운 디렉토리 엔트리를 삽입하고자 할 때 오버플로우가 발생하면, 디렉토리 구조는 그림 5와 같이 확장 해싱을 이용하기 위한 단일 확장 해시 구조로 전환된다.

(2) Inode 블록에 통합된 해시 테이블 구조

Inode 블록에 통합된 디렉토리 엔트리 구조에서 새로운 디렉토리 엔트리를 삽입하고자 할 때 오버플로우가 발생하면, 디렉토리 구조는 확장 해싱을 이용하는 디렉토리 구조로 전환된다. 그림 5처럼 inode 블록에 해시 테이블을 저장하고, inode 블록에 저장되어 있던 디렉토리 엔트리들을 해시 테이블을 이용하여 각각의 디렉토리 엔트리 블록으로 이동한다. 이때 해시 테이블 엔트리들은 독립적인 디렉토리 엔트리 블록을 가질 수도 있고, 여러 해시 테이블 엔트리들이 동시에 하나의 디렉토리 엔트리 블록을 공유할 수도 있다.

이와 같은 디렉토리 구조에서는 하나의 디렉토리 엔트리 블록에서 오버플로우가 발생하면 이 엔트리 블록을 가리키는 해시 테이블 엔트리들의 링크 포인터 수를 조사하여, 둘 이상이면 디렉토리 엔트리 블록이 두 개의 블록들로 분할되고, 하나면 해시 테이블의 크기를 2배로 확장시킨다. Inode 블록에 통합된 해시 테이블 구조는 해시 테이블 블록을 따로 두어 데이터 블록에 접근하는 방법에 비해 접근하는 블록

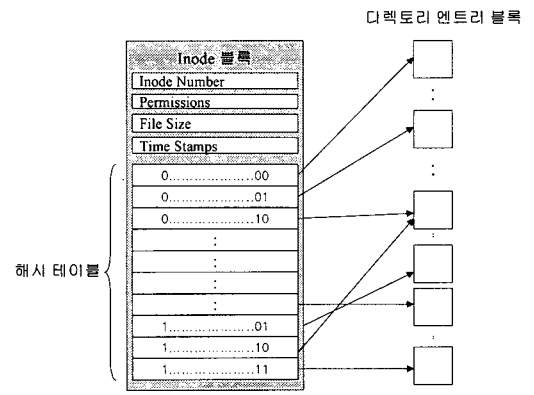


그림 5. Inode 블록에 통합된 해시 테이블 구조

의 수를 감소시킬 수 있다는 장점이 있다.

한편, inode 블록에 통합된 해시 테이블을 최대 2^8의 엔트리들을 가질 수 있다고 가정하고, 이때 166개의 디렉토리 엔트리들을 저장할 수 있는 디렉토리 엔트리 블록이 50%만 채워졌다고 생각할 때, 약 1만개((256×1/2) × (166×1/2) = 10,500) 이상의 디렉토리 엔트리를 저장할 수 있다. 이러한 디렉토리 구조에서 삽입 시에 오버플로우가 발생하면 그림 6과 같이 inode 블록을 이용하여 해시 테이블에 간접 접근하는 디렉토리 구조로 전환된다.

(3) 세미플랫 구조를 이용한 간접 접근 해시 테이블 구조

Inode 블록에 통합된 해시 테이블 구조에서 새로운 디렉토리 엔트리를 삽입할 때 오버플로우가 발생하면, 디렉토리 구조는 그림 6과 같이 간접 접근 해시 테이블 구조로 전환된다. Inode 블록을 접근하여 해당 해시 테이블 엔트리 블록을 검색하고 그 곳에서 원하는 디렉토리 엔트리 블록의 주소를 검색한다. 이와 같은 디렉토리 구조는 거대한 해시 테이블을 가질 수 있으므로 대규모의 디렉토리 엔트리들을 수용할 수 있고, 찾고자 하는 디렉토리 엔트리를 해시 값을 이용하여 빠르게 검색할 수 있다. 한편, 시스템의 간접 접근 해시 테이블은 최대 2^{16}의 크기를 가질 수 있다고 가정하고, 이때 166개의 디렉토리 엔트리들을 저장할 수 있는 디렉토리 엔트리 블록이 50%만 채워졌다고 생각할 때, 최소 135만개((256×1/2)^2× (166×1/2) = 1,359,872) 이상의 디렉토리 엔트리를 저장할 수 있다.

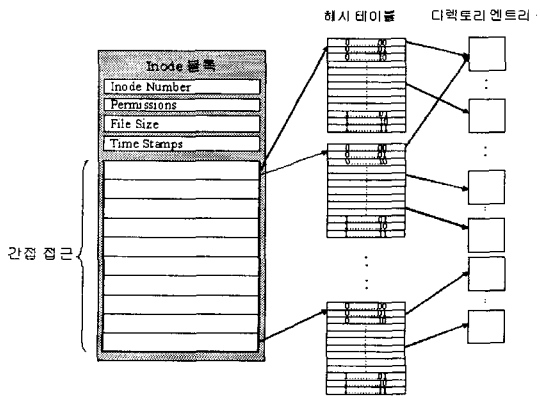


그림 6. 간접 접근 해시 테이블 구조

간접 접근 해시 테이블 구조에서 새로운 디렉토리 엔트리를 삽입할 때 오버플로우가 발생하면, 디렉토리 구조는 그림 7과 같이 세미플랫 구조를 이용한 확장 해싱 디렉토리 구조로 전환된다. 그림 7에서 실선의 왼쪽은 inode 트리를 나타내고, 오른쪽은 해시 테이블과 디렉토리 엔트리 블록들을 나타낸다. 이처럼 디렉토리 구조가 세미플랫 구조를 이루며 조금씩 확장되다보면 플랫 구조를 이루게 되고, 다시 플랫 구조에서 오버플로우가 발생하면 레벨이 하나 증가하고 세미플랫 구조로 변화되며 확장된다.

해시 값이 32비트이므로 inode 블록과 해시 테이블 블록 사이에 최대 2개의 레벨이 더 증가할 수 있다. 즉, 시스템의 해시 테이블은 최대 2^{32}의 크기를 가질 수 있으므로, 이때 166개의 디렉토리 엔트리들을 저장할 수 있는 디렉토리 엔트리 블록이 50%만 채워졌다고 생각할 때, 최소 222억개((256×1/2)^4 × (166×1/2) = 22,280,142,848) 정도의 디렉토리 엔트리를 저장할 수 있다.

3.3 디렉토리 연산

그림 8은 확장 해싱 디렉토리의 삽입 연산 함수이다. Exhash_insert 함수는 파일의 이름으로 해시 함수를 사용하여 해시 값을 구하고, 이를 이용하여 저장할 디렉토리 엔트리 블록을 찾아간다. 그러나, 할당하고자 하는 디렉토리 엔트리 블록이 오버플로우가 발생하면 그 블록에 연결된 링크의 수를 본다. 하나의 링크 포인터로 연결된 경우에는 해시 테이블의 크기를 2배로 증가하고 새로운 블록을 추가한다. 오버플로우가 발생한 블록의 내용을 새로운 블록과 나누어 분배하고, 해시 테이블의 깊이만큼의 해시 값을

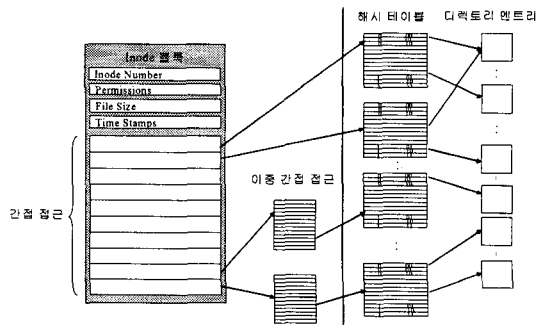


그림 7. 세미플랫 구조를 이용한 확장 해싱 디렉토리 구조

```
알고리즘 exhash_insert
입력 : 디렉토리 엔트리 정보
출력 : 삽입 성공 여부
{
 파일의 이름을 이용하여 해시 값을 계산;
 해시 값을 이용하여 해시 테이블에 연결된 알맞은 디렉토리
 엔트리 블록을 찾음;
  if(찾은 디렉토리 엔트리 블록에 오버플로우 발생) {
    if(블록 할당 실패)
    { return false; }
    if(찾은 디렉토리 엔트리 블록의 링크수 == 1) {
        /* hash_table glow */
      if(해시 테이블 블록에 오버플로우 발생) {
        레벨의 증가를 위해서 디렉토리 블록 할당;
        /* 레벨 증가 및 디렉토리 2배로 증가 */
      }
      else { /* 같은 레벨에서의 디렉토리 구조 확장 */
       해시 테이블 블록 할당, 디렉토리 구조 2배로 확장;
      }
      해시 테이블의 깊이 증가; // 2배로 증가
    }
    else { /* hash_table split */
      오버플로우 블록과 새로운 블록의 링크 포인터 변경;
    }
    오버플로우 발생한 블록의 내용을 새로운 블록과 재분배;
    해시 값을 이용하여 알맞은 디렉토리 엔트리 블록에
    데이터 저장;
  }else 찾은 엔트리 블록에 맨 뒤에 데이터 저장;
  return true;
}
```

그림 8. 확장 해싱 디렉토리의 엔트리 삽입

비교하여 알맞은 엔트리 블록을 찾아 삽입한다. 그렇지 않을 경우에는 해시 테이블 크기의 증가 없이 새로운 디렉토리 엔트리 블록만 추가하고 링크 포인터를 수정하여 오버플로우가 발생한 블록의 내용을 새로운 블록과 나누어 분배한 후, 해시 테이블의 깊이만큼의 해시 값을 비교하여 알맞은 엔트리 블록을 찾아 삽입한다.

그림 9는 확장 해싱 디렉토리의 검색 연산 함수이다. Exhash_search 함수는 파일의 이름으로 해시 함수를 사용하여 해시 값을 구하고, 이를 이용하여 검색할 디렉토리 엔트리 블록을 찾아가서 해당 디렉토리 엔트리를 찾으며, 파일의 존재 유무를 반환한다. 그림 9와 같다.

```
알고리즘 exhash_search
입력 : 파일의 이름
출력 : 파일의 존재 유무
{
 파일의 이름을 이용하여 해시 값을 계산;
 해시 값을 이용하여 해시 테이블에 연결된
 알맞은 디렉토리 엔트리 블록을 찾음;
 block_pointer = 엔트리 블록의 처음 위치;
 do{
   if(파일 엔트리가 존재) {
     print(찾았음);
     return ok;   //파일이 존재 함
   }
   else {
         block_pointer += 엔트리 길이;
   }
 }while(! 엔트리 블록의 끝);
 print(못 찾았음);
 return no;       //파일이 존재하지 않음
}
```

그림 9. 확장 해싱 디렉토리의 엔트리 검색

4. 선형 해싱을 이용한 디렉토리 구조

SAN 파일 시스템은 선형 해싱[20]을 이용하여 디렉토리 구조를 관리할 수 있다. 해시 값을 이용하여 저장할 데이터 블록에 데이터를 저장하나, 별도의 해시 테이블이 존재하지 않는 선형 해싱 구조에서는 바로 디렉토리 엔트리 블록 접근이 가능하고 삽입 시 오버플로우가 발생하여도 디렉토리 엔트리 블록만 증가하므로, 확장 해싱 구조에서의 해시 테이블이 2배 증가하는 데 소요되는 시간에 비해 데이터 삽입 시간이 적게 소요될 수 있다.

4.1 해시 함수

선형 해싱은 블록의 분할된 여부에 따라 각각 다른 해시 함수를 사용하여 해시 값을 구한다. 각 해시 함수의 결과로 나오는 범위가 전의 것보다 두 배씩 되는 해시 함수 계열(h_0, h_1, h_2, ...)을 사용한다.

h_i(해시 값) = h(해시 값) mod (2^iN)

i: 현재의 레벨을 표시, N: 초기에 필요한 엔트리 블록 수

$d_i = d_0+i$

d_i: 블록의 초기 개수

d_0: N을 표현하는데 필요한 비트 수

예를 들면, N = 32 → d_0 = 5일 때, 다음과 같다.

h_0 = h mod 32 (h_0의 생성 범위: 0-31)

h_1 = h mod 64 (h_1의 생성 범위: 0-63)

선형 해싱은 여러 라운드(Round)를 거치면서 확장하며, 해당 라운드 레벨(Level)에서는 해시 함수 h_{Level}과 $h_{Level+1}$만을 사용하고, 각 라운드가 시작될 때 있던 블록들이 첫 블록부터 마지막 블록까지 차례로 분할되어 블록의 수가 두 배로 되면 레벨이 하나 증가한다. 본 논문에서 사용한 해시 값은 확장 해싱에서 사용한 CRC 해시 함수를 사용해서 나온 값을 이용한다.

4.2 선형 해싱 디렉토리

SAN 파일 시스템의 디렉토리 구조는 파일의 수를 고려하여 결정하며, 파일의 수가 적을 때는 inode 블록에 디렉토리 엔트리를 함께 저장하고, 파일의 수가 많아져 inode 블록에서 오버플로우가 발생하면 선형 해싱을 이용하는 구조로 변하여 디렉토리 엔트리를 저장한다. 확장된 선형 해싱 구조로 변환되면 inode의 세미플랫 구조를 이용하며 확장한다. 디렉토리는 다음과 같이 세 경우로 나누어 관리한다.

(1) Inode 블록에 통합된 디렉토리 엔트리

디렉토리 엔트리 수가 적을 때는 디렉토리 엔트리들을 inode 블록에 직접 저장함으로써 inode 블록 한 번의 접근으로 디렉토리 엔트리를 검색할 수 있으며, 이는 앞에서 소개된 그림 4에서 볼 수 있다. 이 inode 블록에 새로운 디렉토리 엔트리를 삽입 하고자 할 때 오버플로우가 발생하면, 디렉토리 구조는 그림 10과 같이 선형 해싱을 이용하기 위한 확장 구조로 전환된다.

(2) 선형 해싱 구조

Inode 블록에 통합된 디렉토리 엔트리 구조에서 새로운 디렉토리 엔트리를 삽입하고자 할 때 오버플로우가 발생하면, 디렉토리 구조는 선형 해싱을 이용하는 디렉토리 구조로 전환된다. 그림 10처럼 inode 블록에서 직접 디렉토리 엔트리 블록을 접근할 수 있다.

이와 같은 디렉토리 구조에서는 하나의 디렉토리 엔트리 블록에서 오버플로우가 발생하면, 새로운 디렉토리 엔트리 블록을 할당받아 맨 마지막에 연결하고 스플릿 포인터 *sp*가 가리키고 있는 블록 내용을 해시 함수 $h_{Level+1}$을 이용하여 재분배한다. 그리고 해시 값을 이용하여 알맞은 곳에 새 디렉토리 엔트리를 삽입한다. 그러나 오버플로우가 발생한 디렉토리 엔트리 블록이 분할된 블록이 아니라면, 오버플로우 블록을 할당받아 저장한다.

Inode 블록을 이용하여 디렉토리 엔트리 블록에 접근할 수 있는 공간이 없으면 그림 11과 같이 inode 구조의 간접 접근을 위한 블록이 증가하여 레벨이 한 단계 증가한 확장된 구조로 전환된다. 선형 해싱은 해싱 테이블이 존재하지 않으므로, 오버플로우 발생률이 적을 경우에 확장 해싱을 이용한 디렉토리 방법보다 접근하는 블록수를 감소시킬 수 있다는 장점이 있다.

한편, inode 블록에 저장할 수 있는 디렉토리 엔트리 블록의 주소의 개수를 256개, 파일 이름의 길이를 8bytes, 166개의 디렉토리 엔트리들을 저장할 수 있는 디렉토리 엔트리 블록이 50%만 채워졌고 또, 50%만 채워진 오버플로우 블록이 전체 엔트리 블록의

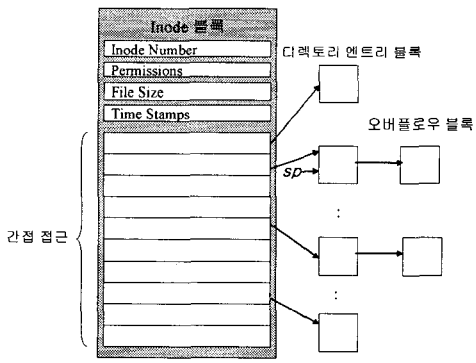


그림 10. 선형 해싱을 이용한 구조

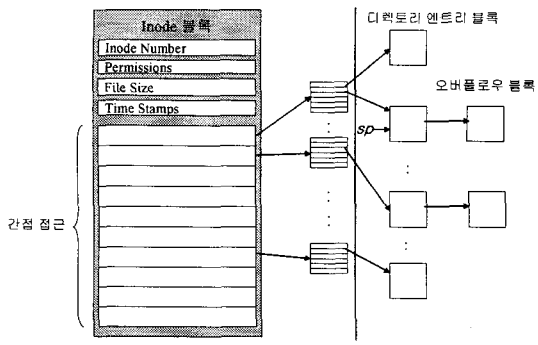


그림 11. 확장된 선형 해싱을 이용한 구조

50%만 존재한다고 가정할 때, 약 1만 5천개((256×1/2) × (166×1/2)) × 3/2 = 15,936) 이상의 디렉토리 엔트리를 저장할 수 있다.

(3) 세미플랫 구조를 이용한 확장된 선형 해싱 구조

그림 10에서 inode 블록을 이용하여 디렉토리 엔트리 블록에 접근할 수 있는 공간이 없는 상황에서 새로운 디렉토리 엔트리를 삽입할 때 오버플로우가 발생하면, 디렉토리 구조는 그림 11과 같이 확장된 선형 해싱을 이용한 구조로 전환된다. 그림 11에서 실선의 왼쪽은 inode 트리를 나타내고, 오른쪽은 디렉토리 엔트리 블록들을 나타낸다. Inode 블록을 접근하여 간접 접근 블록을 이용하여 올바른 디렉토리 엔트리 블록을 찾아간다.

한편, 확장된 선형 해싱 구조는 위와 같은 조건으로 가정하면, 최소 203만개($((256\times1/2)^2 \times (166\times1/2)) \times 3/2 = 2{,}039{,}808$) 이상의 디렉토리 엔트리를 저장할 수 있다.

확장된 선형 해싱을 이용한 구조에서 새로운 디렉토리 엔트리를 삽입할 때 오버플로우가 발생하면, 디렉토리 구조는 그림 12와 같이 세미플랫 구조를 이용한 선형 해싱 디렉토리 구조로 전환된다. 한편, 해시 값이 32비트이므로 그 때까지 디렉토리 엔트리 블록의 레벨이 최대 32까지 증가할 수 있다. 위와 같은 조건으로 가정하면, 최소 334억개($(((256\times1/2)^4 \times 166\times 1/2) \times 3/2 = 33{,}420{,}214{,}272$) 정도의 디렉토리 엔트리를 저장할 수 있다.

4.3 디렉토리 연산

그림 13은 선형 해싱 디렉토리의 삽입 연산 함수이다. 파일의 이름을 이용하여 저장할 디렉토리 엔트리 블록을 찾아간다. 그러나, 할당하고자 하는 디렉

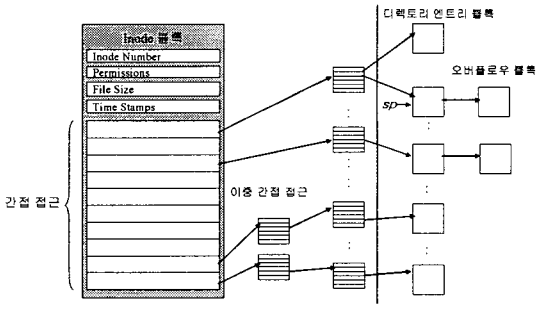


그림 12. 세미플랫 구조를 이용한 선형 해싱 디렉토리

```
알고리즘 linear_insert
입력 : 디렉토리 엔트리 정보
출력 : 삽입 성공 여부
{
 파일의 이름을 이용하여 해시 값을 계산;
 h_Level을 이용하여 알맞은 디렉토리 블록을 찾음;
 if(찾은 디렉토리 엔트리 블록에 오버플로우 발생) {
   새로운 디렉토리 엔트리 블록을 할당받아 맨 뒤에 놓음;
   if(블록 할당 실패)
   { return false; }        /* sp는 스플릿 포인터 */
    h_Level+1을 이용하여 sp가 가리키는 블록을 sp가 가리
    키고 있는 블록과 할당된 블록으로 재분배;
    sp는 다음 블록을 가리킴;
    if(sp가 h_Level의 범위 초과)
    { 레벨 증가; sp= 0; }
    if(오버플로우 발생한 블록 == 전에 sp가 가리키던 블록)
      h_Level+1을 이용하여 맞는 블록으로 삽입;
    else{
     오버플로우 발생한 블록에 새로운 오버플로우 블록
     을 할당 받아 링크로 연결;
     오버플로우 블록에 엔트리 정보 삽입;
   }
 }
 else
 {
   디렉토리 엔트리 블록에 입력으로 받은 엔트리 정보를
   블록의 끝에 삽입;
 }
 return true;
}
```

그림 13. 선형 해싱 디렉토리의 엔트리 삽입

토리 엔트리 블록이 오버플로우가 발생하면 새 블록을 할당받아 맨 뒤에 연결하고, 스플릿 포인터 *sp*가 가리키고 있는 블록 내용을 해시 함수 $h_{Level+1}$을 이용하여 재분배하며 조금씩 확장한다. 그리고 해시 값을 이용하여 알맞은 곳에 새 디렉토리 엔트리를 삽입한다. 그러나 오버플로우가 발생한 디렉토리 엔트리 블록이 분리된 블록이 아니라면, 오버플로우 블록을 할당받아 저장한다.

그림 14는 선형 해싱 디렉토리의 검색 연산 함수이다. 이 함수는 파일의 이름을 이용하여 검색할 디렉토리 엔트리 블록을 찾아가서 해당 디렉토리 엔트리를 찾는다.

```
알고리즘 linear_search
입력 : 파일의 이름
출력 : 파일의 존재 유무
{
 파일의 이름을 사용하여 해시 값을 구함;
 hLevel을 이용하여 디렉토리 블록 위치인 index를 찾음;
 if ( index < sp ) /* sp는 스플릿 포인터 */
   hLevel+1을 이용하여 디렉토리 블록 위치인 index를 찾음;
 block_pointer = 엔트리 블록의 처음 위치;
 do{
     if(파일 엔트리가 존재) {
          print(찾았음);
          return ok;   //파일이 존재 함
     }
     else {
          block_pointer += 엔트리 길이;
     }
   }while(! 엔트리 블록의 끝);
  if(연결된 오버플로우 블록 존재) {
     연결된 오버플로우 블록 전부들을 순차 탐색;
     print(찾았음);
     return ok;   //파일이 존재 함
   }
   else {
      print(못 찾았음);
      return no;        //파일이 존재하지 않음
  }
}
```

그림 14. 선형 해싱 디렉토리의 엔트리 검색

검색은 해시 값을 이용하여 디렉토리 엔트리 블록의 위치를 찾고 스플릿 포인터 *sp*값보다 작으면 해시 함수 $h_{Level+1}$을 이용하여 디렉토리 엔트리 블록의 위치를 다시 찾는다. 해당 디렉토리 엔트리 블록의 위치를 찾아 그 블록 내에서 엔트리를 검사하는데, 해당 블록에 오버플로우 블록이 연결되어 있으면 오버플로우 블록도 검사한다.

5. 구현 및 성능 실험

본 절에서는 SAN 파일 시스템을 위해 세미플랫 구조를 이용한 확장 해싱 디렉토리와 세미플랫 구조를 이용한 선형 해싱 디렉토리를 구현하고 성능의 비교 분석을 위해 플랫 구조를 가진 확장 해싱 디렉토리와 플랫 구조를 가진 선형 해싱 디렉토리도 구현하며, 이를 이용하여 성능을 비교 분석한다.

5.1 구현 환경

펜티엄 IV-1.7GHz, RAM 256Mbytes 컴퓨터에 OS는 리눅스 커널 2.6.0을 사용하며 GCC gcs-2.95.3 버전을 이용하여 해싱 디렉토리들을 구현한다. 구현 환경에서 사용되는 메모리의 용량은 성능 실험에서 고려될 삽입과 검색 연산이 그 모든 블록들을 메모리로 읽어오는 것이 아니라 해당되는 블록만을 메모리로 읽어오기 때문에 메모리의 용량과는 크게 상관이 없다. 구현 및 실험에 이용된 디스크는 Seagate사의 ST380011A 모델[25]이며 디스크의 성능은 표 1과 같다.

5.2 성능 실험

구현된 소스 코드들을 컴파일하면 확장 해싱 디렉토리 관리자를 가지고 있는 파일 시스템인 ehfs.o 파일과 확장 해싱 디렉토리 관리자를 가지고 있는 파일 시스템인 lhfs.o 파일이 생성된다. 그림 15와 같은 설

표 1. Seagate사의 ST380011A 디스크의 성능

사	양
capability	80 GB
sector size	512 Bytes
dist / head	1 / 2
transfer rate	100 Mb/sec
seek average	8.5 ms
latency average	4.16 ms
rotational speed	7,200 RPM

```
파일 시스템 설치 과정
1. 여분의 하드를 준비한다.(예: /dev/hdb라고 가정)
2. mkfs를 실행한다.
   $> mksfs /dev/hdb
3. 모듈을 등록한다.
   $> /sbin/insmod ehfs.o
4. 마운트 포인트가 될 디렉토리를 만든다.
   $> mkdir /SFS
5. 마운트를 수행한다.
   $> mount -t ehfs /dev/hdb /SFS
```

그림 15. 확장 해싱 디렉토리 관리자를 가진 파일 시스템 설치 과정

치 과정을 단계별로 진행하면 확장 해싱 디렉토리 관리자를 가진 파일 시스템이 설치된다. 선형 해싱 디렉토리 관리자를 가진 파일 시스템은 그림 14에서 'ehfs' 대신 'lhfs'으로 바꾸어 실행하면 된다.

모든 과정이 끝나면 'SFS' 디렉토리에 파일 시스템이 설치되며, 이 디렉토리로 이동하여 mkdir, rmdir, find 등의 명령어를 이용하여 파일이나 디렉토리를 삽입, 삭제, 검색할 수 있다. 성능 실험은 쉘 프로그램을 이용하여 디렉토리 엔트리들을 일정 수만큼 계속 삽입과 검색에 소요되는 시간과 사용 블록수를 측정하였다. 그리고, 두 디렉토리를 비교하기 위해서 선형 해싱에서의 오버플로우 비율을 고려한다.

$$\text{오버플로우 비율}(OR) = \frac{\text{오버플로우블록에 저장된 엔트리수}}{\text{전체 엔트리수}}$$

성능 실험을 위해서 해시 함수를 조절하여 원하는 오버플로우 비율이 나오도록 하였다. 오버플로우 비율은 오버플로우가 적은 경우에 대해서는 20%로 하고 오버플로우가 많은 경우에 대해서는 50%를 기준으로 평가하였다.

5.3 실험 결과

본 절에서는 구현된 세미플랫 구조를 이용한 확장 해싱 디렉토리와 세미플랫 구조를 이용한 선형 해싱 디렉토리에서의 디렉토리 엔트리들의 삽입과 검색에 소요되는 시간과 사용하는 블록수를 측정하여 비교하였다. 또한 세미플랫 구조와 플랫 구조의 차이를 보기 위하여 플랫 구조를 이용한 확장 해싱 디렉토리와 플랫 구조를 이용한 선형 해싱 디렉토리도 구현하여 비교하였다. 실험은 4가지의 구조에 대하여 디렉토리의 규모에 따라 두 경우로 나누어서 진행하였는데, 하나는 inode 블록에 통합된 해시 테이블 구조에서 최대 약 40,000개 정도 엔트리들을 하나의 디렉토리에서 수용 가능하므로, 약 38,000개의 엔트리까지 가질 수 있는 소규모 디렉토리에 대한 성능 실험이고, 다른 하나는 간접 접근 해시 테이블 구조에서 수용되어 사용되는 엔트리 수 중 약 100,000개 이상 수용하는 대규모 디렉토리에 대한 성능 실험이다. 본 논문에서는 각 성능실험마다 10번 실험을 통해 나온 결과 값을 기반으로 평균값을 구하였다.

(1) 디렉토리 엔트리 삽입

디렉토리 엔트리를 삽입하는 데 소요되는 엔트리 당 평균 수행 시간의 성능 평가 결과는 그림 16부터 그림 19까지와 같다. 디렉토리 엔트리 삽입 시간은 명령어 'mkdir dirname'을 한번 수행하는 데 걸리는 시간으로 한다.

그림 16과 그림 17은 선형 해싱 디렉토리에서의 오버플로우 비율 OR이 20% 정도일 때의 디렉토리 엔트리 당 평균 삽입 시간을 보여준다. x축은 삽입하는 엔트리 수를 나타내고, y축은 x축 개수의 엔트리 만큼을 삽입할 때 하나의 엔트리 당 소요되는 평균 수행시간을 나타낸다.

그림 16은 소규모 디렉토리를 보여주고 있는데, 이때는 inode 구조가 세미플랫 구조로 변화되기 전이므로 플랫 구조인 확장 해싱 그래프와 세미플랫 구조인 확장 해싱 그래프가 겹쳐져 있다. 선형 해싱의 경우도 같다. 확장 해싱 디렉토리 그래프에서 갑자기 증가하는 부분은 해시 테이블이 inode 블록에서 분리되는 지점으로 이는 다른 때보다 오버헤드가

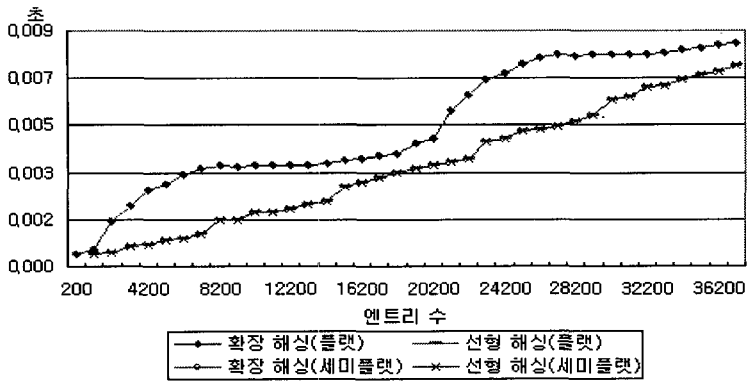


그림 16. 소규모 디렉토리 엔트리 당 평균 삽입 시간 (OR 20%)

있음을 보여주고, 선형 해싱은 전반적으로 천천히 평균 삽입 시간이 증가함을 보여준다. 오버플로우가 발생했을 경우, 해시 테이블의 증가하는 부분을 수행하는 확장 해싱 디렉토리가 선형 해싱 디렉토리보다 많은 수행 시간을 소요함을 볼 수 있다.

그림 17은 세미플랫 구조일 때의 디렉토리 구조들을 비교하기 위해서 엔트리 수 약 15,000,000개 정도 수용할 수 있는 대규모 파일 시스템의 디렉토리에 대한 것으로 위의 실험과 같은 조건으로 시뮬레이션을 통해 얻은 결과이다. 이 그림에서도 그림 16과 같이 엔트리 수가 많아짐에 따른 하나의 임의의 엔트리당 삽입시간 역시 선형 해싱 디렉토리 구조가 확장 해싱 디렉토리 구조보다 디렉토리 엔트리를 삽입하는 데에 적은 시간을 소요함을 볼 수 있다. 특히 세미플랫 구조를 가진 디렉토리 구조들이 적은 시간을 소요함을 볼 수 있다.

그림 18과 그림 19는 선형 해싱 디렉토리에서의 *OR*이 50% 정도일 때의 디렉토리 엔트리 당 평균 삽입 시간을 보여준다. 그림 18은 그림 16에 비해 오버플로우 비율이 높으므로, 전체적으로 두 디렉토리 구조가 빨리 변하고 있음을 보여준다. 삽입시간은 오버플로우의 비율에 상관없이 선형 해싱 디렉토리가 확장 해싱 디렉토리보다 좋은 성능을 보이고 있다. 이 그림에서도 inode 구조가 세미플랫 구조로 변화되기 전이므로, 플랫 구조와 세미플랫 구조 그래프가 겹쳐져 있다.

그림 19 역시 대규모 디렉토리에 대해서 그림 17의 실험과 같은 조건으로 시뮬레이션을 통해 얻은 결과이다. OR 20%와 비슷하게 선형 해싱 디렉토리 구조가 확장 해싱 디렉토리 구조보다 디렉토리 엔트리를 삽입하는 데에 적은 시간을 소요함을 볼 수 있고, 플랫구조보다 세미플랫 구조를 가진 디렉토리 구조가 성능이 좋음을 볼 수 있다.

디렉토리 엔트리를 삽입할 때 OR이 5% 정도로

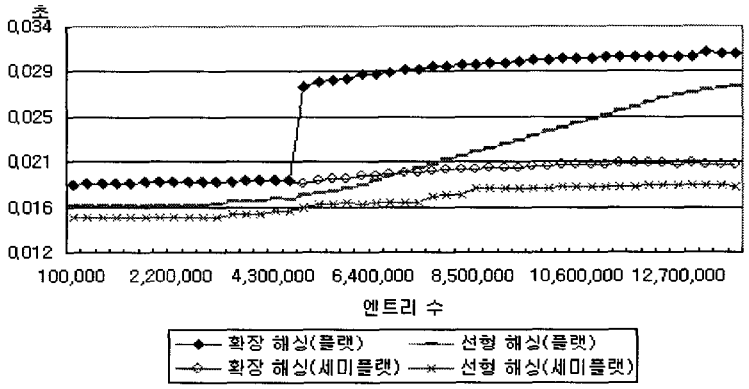


그림 17. 대규모 디렉토리 엔트리 당 평균 삽입 시간 (*OR* 20%)

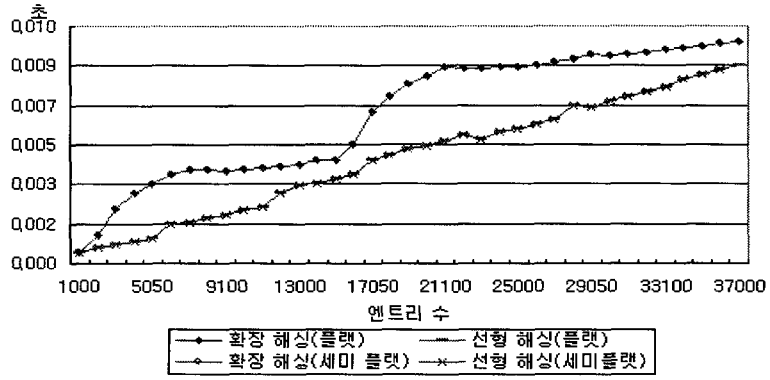


그림 18. 소규모 디렉토리 엔트리 당 평균 삽입 시간 (*OR* 50%)

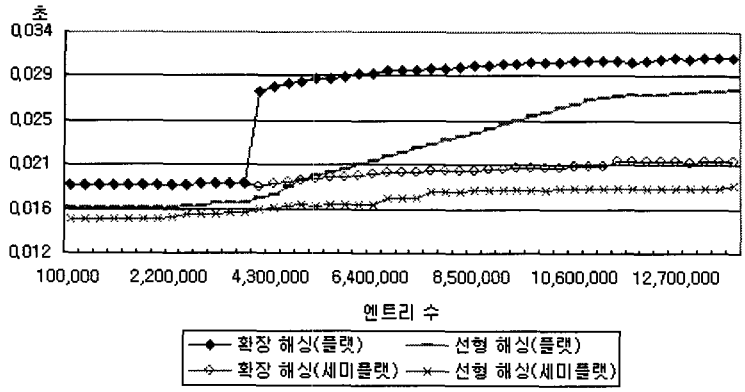


그림 19. 대규모 디렉토리 엔트리 당 평균 삽입 시간 (*OR* 50%)

낮게 되면 전반적으로 디렉토리 블록의 분할이 적어지므로 완만하게 디렉토리 구조가 변화하여 삽입 시간에 적은 시간이 소요되나, OR이 90% 정도로 높게 되면 디렉토리 블록의 분할이 많아져서 특히, 확장 해싱 디렉토리 구조는 급격하게 구조가 변화되므로 삽입 시간에 점차 더 많은 시간을 소요하게 됨을 알 수 있었고, 세미플랫 구조를 가진 해싱 디렉토리들은 inode의 완만한 증가로 인하여 삽입 시간도 천천히 증가함을 알 수 있었다.

(2) 디렉토리 엔트리 검색

디렉토리 엔트리를 검색하는데 소요되는 평균 수행 시간의 성능 평가 결과는 그림 20부터 그림 23까지와 같다. 디렉토리 엔트리 검색 시간은 명령어 'find dirname'을 한번 수행하는 데 걸리는 시간으로 한다. 그림 20과 그림 21은 선형 해싱 디렉토리에서의 *OR*이 20% 정도일 때의 디렉토리 엔트리 평균 검색 시간을 보여준다. x축은 검색할 당시에 디렉토리에 있는 엔트리 수를 나타내고, y축은 하나의 엔트리를 검색할 때 소요되는 평균 수행시간을 나타낸다.

그림 20에서도 소규모 디렉토리를 보여주고 있는데, inode 구조가 세미플랫 구조로 변화되기 전이므로 플랫 구조 그래프와 세미플랫 구조 그래프가 겹쳐져 있다. 확장 해싱 디렉토리는 해시 테이블을 거쳐서 디렉토리 엔트리 블록에 접근하고, 선형 해싱 디렉토리에서는 직접 디렉토리 엔트리 블록에 접근하므로 선형 해싱 디렉토리가 검색 시간이 적게 소요된다고 예측할 수 있으나, 앞 절에서 설명하였듯이 확장 해싱 디렉토리 구조의 소규모 디렉토리 구조에서는 해시 테이블이 inode 구조에 있기 때문에 선형 해싱 디렉토리와 차이가 없다, 오히려 선행 해싱 디렉토리 구조에서의 오버플로우 블록이 존재하므로 선형 해싱 디렉토리의 접근 시간이 더 소요함을 볼 수 있다. 그러나 선형 해싱은 확장 해싱 디렉토리 구

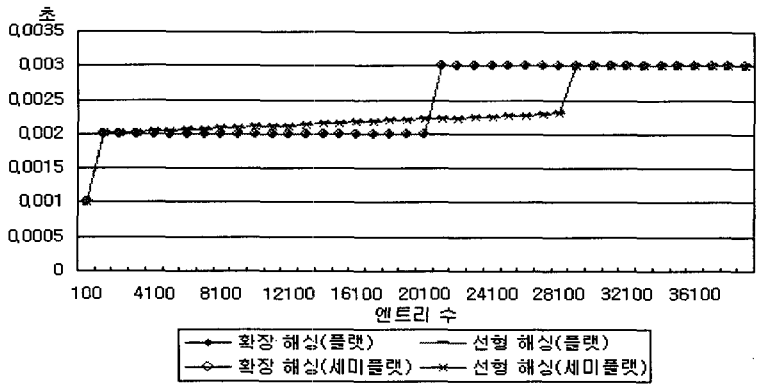


그림 20. 소규모 디렉토리 엔트리 평균 검색 시간 (*OR* 20%)

조보다 완만하게 디렉토리 구조가 확장되므로 레벨이 증가하는 지점에서는 선형 해싱 디렉토리 구조의 접근 시간이 적게 소요함을 알 수 있다.

그림 21은 대규모 디렉토리에서의 디렉토리의 성능을 시뮬레이션을 통해 얻은 결과이다. 실험을 통한 결과와 마찬가지로 많은 수의 엔트리에 대해서 플랫 구조일 때는 소규모 디렉토리인 경우와 비슷함을 알 수 있으나, 세미플랫 구조일 때는 선형 해싱 디렉토리 구조가 확장 해싱 디렉토리 구조보다 디렉토리 엔트리를 검색하는데 시간을 적게 소요함을 볼 수 있다. 전체적으로 세미플랫 구조 가진 디렉토리 구조들이 적은 시간을 소요함을 볼 수 있다.

그림 22와 그림 23은 선형 해싱 디렉토리에서의 OR이 50% 정도 되었을 때의 디렉토리 엔트리 평균 검색 시간을 보여준다. 그림 22는 오버플로우가 많이 발생하게 되면 선형 해싱 디렉토리 구조의 오버플로우 블록 접근이 많이 발생하므로, 그림 20에 비하여 선형 해싱의 평균 검색 소요 시간이 증가함과 레벨 증가가 더 빨리 나타남을 볼 수 있다. 이 그림에서도 inode 구조가 세미플랫 구조로 변화되기 전이므로, 플랫 구조와 세미플랫 구조 그래프가 겹쳐져 있다.

그림 23은 대규모 디렉토리에서의 디렉토리 성능을 시뮬레이션을 통해 얻은 결과이다. 실험을 통한 결과와 마찬가지로 그림 22처럼 많은 수의 엔트리에 대해서 플랫 구조일 경우에는 선형 해싱 디렉토리 구조에서의 검색 시간이 확장 해싱 디렉토리 구조와 거의 비슷하게 소요함을 알 수 있고, 세미플랫 구조일 경우에는 선형 해싱 디렉토리 구조에서의 검색 시간이 적게 소요됨을 볼 수 있다. 또한 세미플랫 구조 가진 디렉토리 구조들이 적은 시간을 소요함도 볼 수 있다.

디렉토리 엔트리를 검색할 때 OR이 5% 정도로 낮게 되면, 선형 해싱 디렉토리 구조는 전반적으로 적은 수의 오버플로우 블록이 존재하므로 엔트리 검

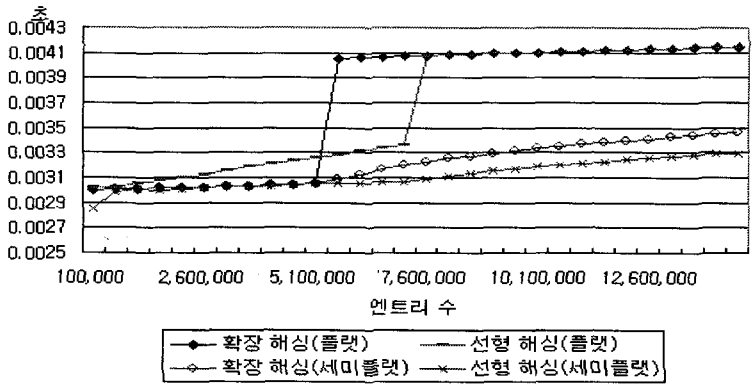


그림 21. 대규모 디렉토리 엔트리 평균 검색 시간 (*OR* 20%)

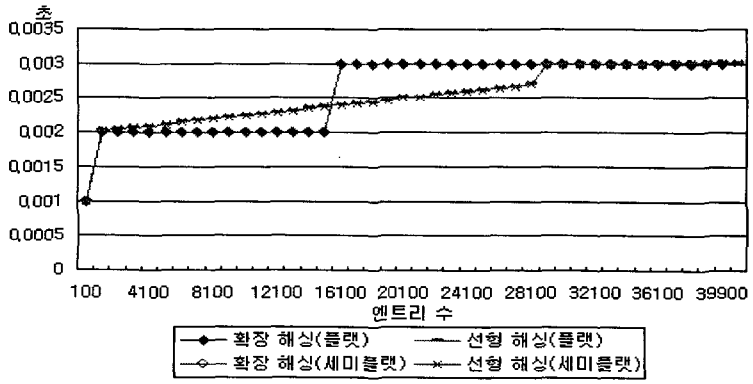


그림 22. 소규모 디렉토리 엔트리 평균 검색 시간 (*OR* 50%)

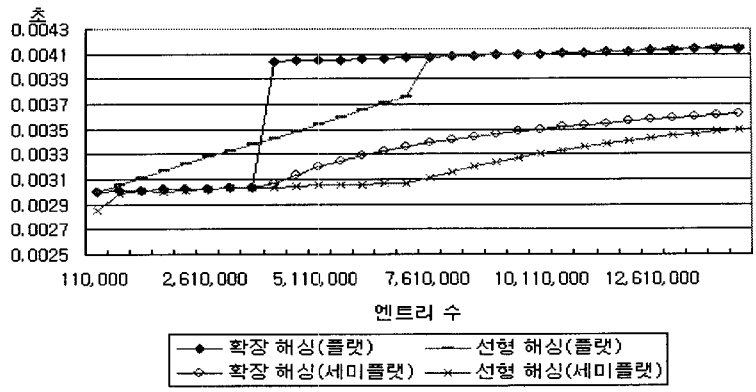


그림 23. 대규모 디렉토리 엔트리 평균 검색 시간 (*OR* 50%)

색을 위해 오버플로우 블록에 접근하는 경우가 줄어들게 됨으로 검색 시간이 적게 소요됨을 알 수 있었다. 그러나, OR이 90% 정도로 높게 되면, 오버플로우 블록이 많아지게 됨으로 선형 해싱 디렉토리 구조임에도 불구하고 확장 해싱 디렉토리 구조보다 블록을 여러번 접근하게 되는 경우가 많이 발생될 수 있어 엔트리 검색에 선형 해싱 디렉토리 구조가 확장 해싱 디렉토리 구조보다 더 많은 시간이 소요됨을 알 수 있었다. 또한, 세미플랫 구조를 가진 해싱 디렉토리들은 inode의 완만한 증가로 인하여 삽입 시간도 천천히 증가함을 알 수 있었다.

(3) 디렉토리 엔트리 블록 사용 수

해싱 디렉토리 구조에서 사용하는 디렉토리 엔트리 블록수는 그림 24부터 그림 27까지와 같다. 확장 해싱 디렉토리가 해시 테이블 블록을 이용하기는 하나, 해시 테이블의 깊이가 8일 때까지는 inode 블록에 해시 테이블을 포함하고 있어, 별도의 블록을 사용하지 않고 있는 것과 같다. 이에 반해, 선형 해싱은 별도의 오버플로우 블록을 사용하므로, 엔트리 수가 많아짐에 따라 사용하는 블록수도 많아지게 된다.

그림 24는 선형 해싱 디렉토리에서의 OR이 20% 정도 되었을 때의 사용 블록수의 현황을 보여주고 있다. 또한 소규모 디렉토리를 보여주고 있는데, inode 구조가 세미플랫 구조로 변화되기 전이므로 플랫 구조 그래프와 세미플랫 구조 그래프가 겹쳐져 있다. x축은 디렉토리에 존재하는 엔트리 수를 나타내고, y축은 해당 엔트리별로 사용하는 디렉토리 엔트리 블록수를 나타낸다. 그림에서와 같이 소규모 디렉토리에서는 사용하는 블록수의 차이가 거의 없어 보이지만, 점점 선형 해싱 디렉토리 구조에서 사용하는 블록수가 확장 해싱 디렉토리 구조보다 조금씩 증가함을 볼 수 있다.

그림 25는 대규모 디렉토리에서의 디렉토리 성능

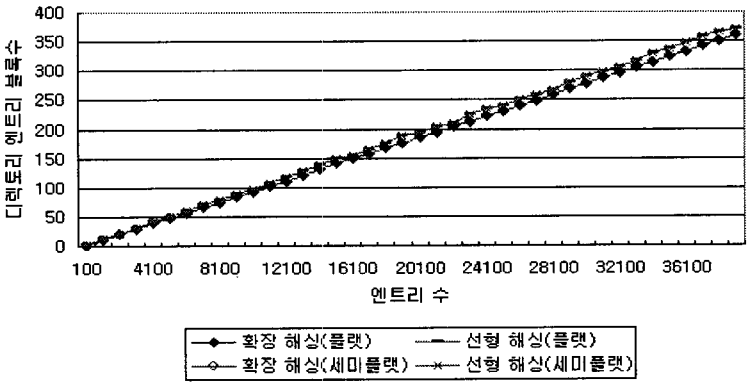


그림 24. 소규모 디렉토리 엔트리 블록 사용 수 (*OR* 20%)

을 시뮬레이션을 통해 얻은 결과이다. 좀 더 자세한 차이 표현하기 위하여 11,000,000개 이상부터 나타내었다. 그림과 같이 플랫 구조와 세미플랫 구조에서 거의 비슷하게 나타나므로, 사용 블록수는 구조와의 관련이 거의 없음을 알 수 있고, 대규모 디렉토리 구조에서는 선형 해싱 디렉토리 구조에서 사용하는 블록수가 확장 해싱 디렉토리 구조보다 많이 사용함을 볼 수 있다.

그림 26은 선형 해싱 디렉토리에서의 *OR*이 50% 정도 되었을 때의 사용 블록수의 현황을 보여주고 있다. 오버플로우 비율이 높으므로, 확장 해싱은 분할이 자주 일어나서 해시 테이블 블록을 사용하는 구조로 변화가 빨리 일어나 사용하는 블록의 수가 많아지게 되므로, 전체적으로 사용하는 블록수는 증가하고, 선형 해싱의 오버플로우 블록이 증가하므로 사용하는 블록수도 점차 많아짐을 볼 수 있다. 그림에서와 같이 소규모 디렉토리에서는 사용하는 블록수의 차이가 거의 없어 보이지만, 점점 선형 해싱 디렉토리 구조에서 사용하는 블록수가 확장 해싱 디렉토리 구조보다 조금씩 증가함을 볼 수 있다. 이 그림에서도 inode 구조가 세미플랫 구조로 변화되기 전이므로, 플랫 구조와 세미플랫 구조 그래프가 겹쳐져 있다.

그림 27은 대규모 디렉토리에서의 디렉토리의 성능을 시뮬레이션을 통해 얻은 결과이다. 그림 26과 마찬가지로 세미플랫구조와 플랫구조에 상관없이 많은 수의 엔트리에 대해서 선형 해싱 디렉토리 구조에서 사용하는 블록수가 확장 해싱 디렉토리 구조에서 사용하는 블록수보다 많이 사용함을 볼 수 있다. 그리고 그림 25에 비해 엔트리 수가 많아지면 많아질수록 오버플로우 사용 블록수는 점차 많이 늘어나게 됨으로써 전체적으로 사용하는 블록수가 증가함을 알 수 있다.

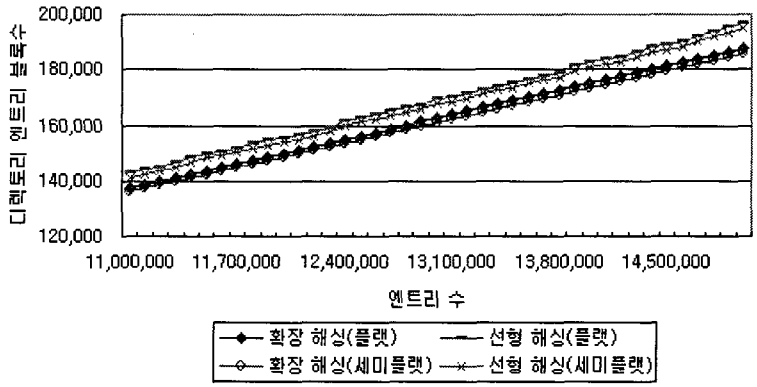


그림 25. 대규모 디렉토리 엔트리 블록 사용 수 (*OR* 20%)

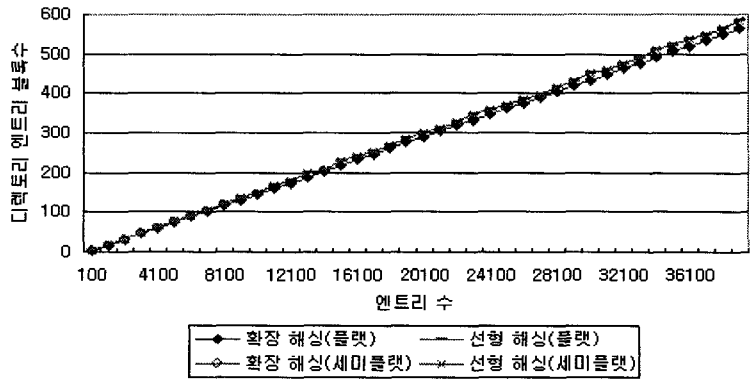


그림 26. 소규모 디렉토리 엔트리 블록 사용 수 (*OR* 50%)

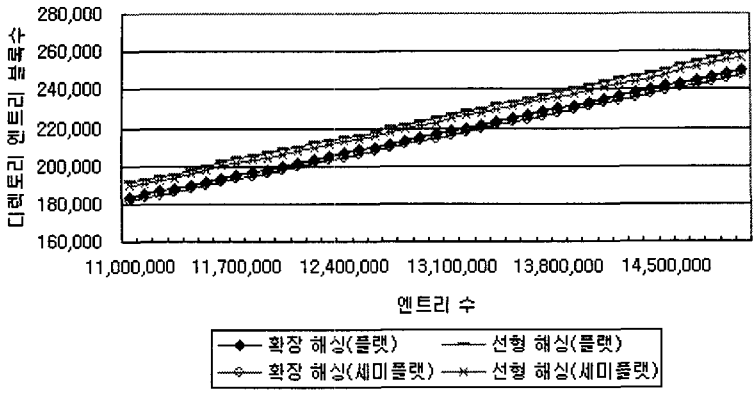


그림 27. 대규모 디렉토리 엔트리 블록 사용 수 (*OR* 50%)

디렉토리 엔트리를 검색할 때 OR이 5% 정도로 낮게 되면 전반적으로 적은 수의 오버플로우 블록이 존재하므로 선형 해싱 디렉토리 구조가 사용하는 블록수는 확장 해싱 디렉토리 구조와 거의 비슷하나, OR이 90% 정도로 높게 되면 오버플로우 블록이 많아지게 됨으로 해시 테이블 블록을 가지고 있는 확장 해싱 디렉토리 구조에서 사용하는 블록수에 비해 훨씬 많은 블록을 사용하게 됨을 알 수 있었다.

위와 같이 삽입, 검색, 사용 블록수에 대한 해싱 디렉토리 구조들에 대해서 비교 분석하였고, 전체적으로 세미플랫 구조를 가진 해싱 디렉토리가 기존의 플랫 구조를 가진 해싱 디렉토리보다 성능이 우수했음을 볼 수 있었다.

6. 결 론

본 논문에서는 SAN 기반 대규모 파일 시스템을 위해서 모든 데이터 블록의 임의 접근 시간이 같으나 파일의 크기가 커질수록 데이터 블록의 임의의 평균 접근 시간이 길어지는 플랫 구조의 단점을 보안한 세미플랫 구조를 이용하여 확장 해싱 디렉토리 구조와 선형 해싱 디렉토리 구조를 설계 및 구현하였다. 이들 해싱 디렉토리 구조는 디렉토리 엔트리의 수를 고려하여 적은 양의 디렉토리 엔트리들은 inode 블록에 직접 저장함으로써 한 번의 접근으로 원하는 엔트리 정보를 검색할 수 있게 하였고, 많은 양의 디렉토리 엔트리들에 대해서는 CRC 함수를 적용한 해시 함수를 이용하여 해싱 기법으로 세미플랫 구조를 이용한 디렉토리 공간에 저장함으로써 엔트리 정보를 빠르게 검색할 수 있게 하였다.

성능 평가를 통하여 선형 해싱 디렉토리의 오버플로우 비율에 따른 해싱 디렉토리 구조들의 성능을 비교하였다. 비교 분석 결과, 세미플랫 구조를 이용한 해싱 디렉토리가 플랫 구조를 이용한 해싱 디렉토리보다 모든 면에서 우수한 성능을 보였다. 파일의 삽입과 검색에서는 세미플랫 선형 해싱 디렉토리 구조가 우수하였으나, 공간 활용 면에서는 세미플랫 확장 해싱 디렉토리 구조가 우수한 성능을 보였다. 오버플로우 비율이 높아질수록 전체적인 평균 소요시간이 증가하였고, 극단적으로 오버플로우 비율이 90% 정도로 높게 될 때는 파일 검색 시 선형 해싱 디렉토리 구조가 확장 해싱 디렉토리 구조보다 더 많은 시간을 소요함을 알 수 있었다.

향후에는 SAN 관련 대규모 파일 시스템을 위해서 동적 해싱 디렉토리 이외의 다른 디렉토리 관리 방법에 대한 연구가 요구된다.

참 고 문 헌

[1] 스토리지 아키텍처의 재발견, http://www.hyosunginformation.co.kr/advantage/no62/docu01.htm, 2004.

[2] R.Y. Wang and T.E. Anderson, "xFS: A Wide Area Mass Storage File System," *Proceedings of the 4th Workshop on Workstation Operating Systems*, Napa, California, pp. 71-78, October 1993.

[3] C.A. Thekkath, T. Mann, and E.K. Lee, "Fran-

gipani: A Scalable Distributed File System," *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St. Malo, France, pp. 224-237, October 1997.

[4] C. Jurgens, "Fibre Channel: A Connection to the Future," *IEEE Computer*, Vol. 28, No. 8, pp. 82-90, August 1995.

[5] Tivoli, http://www-306.ibm.com/software/tivoli/, 2004.

[6] VersaStor, http://hp.co.kr/ec/product/storage/EVA8p_5.html, 2004.

[7] SANPoint, http://www.veritas.co.kr/, 2004.

[8] K.W. Preslan, etc. al, "A 64-bit, Shared Disk File System for Linux," *Proceedings of the 16th IEEE Mass Storage Systems Symposium*, San Diego, California, USA, pp. 22-41, March 1999.

[9] S.R. Soltis, T.M. Ruwart, and M.T. O'Keefe, "The Global File System," *Proceedings of the 5th NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, Maryland, pp. 319-342, September 1996.

[10] 신범주, 김경배, 김창수, 김명준, "네트워크 저장 장치를 위한 클러스터 파일 시스템 개발," *정보처리학회지*, 제8권 4호, pp. 29-41, 2001.

[11] C.S. Kim, G.B. Kim, and B.J. Shin, "Volume Management in SAN Environment," *Proceedings of the 8th International Conference on Parallel and Distributed Systems*, Kyongju City, Korea, pp. 500-505, June 2001.

[12] 오상규, "SANique CFS SAN을 위한 클러스터 파일 시스템," *한국데이터베이스학회지*, 제8권 4호, pp. 29-41, 2004.

[13] F.B. Schmuck and R.L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," *Proceedings of the FAST '02 Conference on File and Storage Technologies USENIX 2002*, Monterey, California, USA, pp. 231-244, January 2002.

[14] T.E. Anderson, M.D. Dahlin, and J.M. Neefe, "Serverless Network File Systems," *ACM Operating Systems Review*, Vol. 29, No. 5, pp. 109-126, December 1995.

[15] 김신우, 박성은, 이용규, 김경배, 신범주, "SAN 기반 리눅스 클러스터 파일 시스템을 위한 메타데이터 관리," 정보처리학회논문지, 제 8-A권 4호, pp. 367-374, 2001.

[16] M.J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[17] D.H.C. Du and S.-R. Tong, "Multilevel Extendible Hashing: A File Structure for Very Large Databases," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 3, No 3, pp. 357-370, September 1999.

[18] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong, "Extendible Hashing-A Fast Access Method for Dynamic Files," *ACM Transactions on Database Systems*, Vol. 4, No. 3, pp. 315-344, September 1979.

[19] V. Hilford, F.B. Bastani, and B. Cukic, "EH*-Extendible Hashing in a Distributed Environment," *Proceedings of the 21 Annual International Computer Software and Applications Conference*, Washington, USA, pp. 217-222, August 1997.

[20] P.E. Livadas, *File Structures: Theory and Practice*, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[21] W. Litwin, "Linear Hashing: A New Tool for File and Table Addressing," *Proceedings of the 6th International Conference on Very Large Databases*, Montreal, Quebec, Canada, pp. 212-223, October 1980.

[22] W. Litwin, M.-A. Neimat, and D.A. Schneider, "LH*-Linear Hashing for Distributed Files," *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington, USA, pp. 327-336, May 1993.

[23] A. Rathi, G. Lu, and G.E. Hedrick, "Performance Comparison of Extendible Hashing and Linear Hashing Techniques," *Proceedings of the 1990 ACM SIGSMALL*, Crystal City, Virginia, USA,

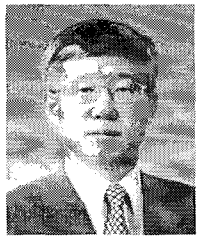
pp. 178-185, March 1990.

[24] A.S. Tanenbaum, *Computer Networks*, Prentice-Hall, Englewood Cliffs, New Jersey, 1996.

[25] Seagate ST380011A Hard-Disk Spec., http://www.seagate-asia.com/seagatefiles/korea/pdf/Cuda_7200.7-KR.pdf, 2004.

김 신 우

1997년 동국대학교 컴퓨터공학과 (학사)
2000년 동국대학교 컴퓨터공학과 (석사)
2004년 동국대학교 컴퓨터공학과 (박사)
2002년~현재 동국대학교 컴퓨터멀티미디어공학과 강사
관심분야 : XML 및 웹, 전자상거래 시스템, 스토리지 시스템, 데이터베이스



이 용 규

1986년 동국대학교 전자계산학과 (학사)
1988년 한국과학기술원 전산학과 (석사)
1996년 Syracuse University(전산학박사)
1978년~83년 정보통신부 행정직 국가공무원
1988년~93년 한국국방연구원 선임연구원
1996년~97년 한국통신 선임연구원
2002년~03년 콜로라도대학교 컴퓨터학과 방문교수
1997년~현재 동국대학교 컴퓨터멀티미디어공학과 교수
관심분야 : XML 및 웹, 전자상거래 시스템, 스토리지 시스템, 데이터베이스, 모바일 컴퓨팅