SAN 환경 대용량 파일 시스템을 위한 디렉토리 구조 비교

Comparison of Directory Structures for SAN Based Very Large File Systems

김신우(Shin Woo Kim)*, 이용규(Yong Kyu Lee)**

초 록

최근 전자상거래시스템을 비롯하여 대용량 데이터의 저장과 검색을 요구하는 정보시스템들이 광범위하게 활용되고 있다. 이에 맞추어 클라이언트가 메타데이터를 직접 관리하며 데이터에 접근할 수 있는 SAN 환경의 리눅스 클러스터 파일시스템이 연구되고 있으며, 파일의 빠른 검색을 위해 확장 해시 기반의 세미 플랫 디렉토리 구조가 제안되었다[1]. 본 연구에서는 리눅스 환경에서 확장 해시 기반의 세미플랫 디렉토리를 설계 및 구현하였으며, 구현된 시스템의 실용성을 평가하기 위하여 B+ 트리 기반의 디렉토리 구조를 함께 구현하여 성능을 비교하였다. 디렉토리의 성능을 비교 분석한 결과, 파일의 삽입, 삭제, 검색 성능에서는 확장 해싱 기반의 디렉토리가 우수하였으나, 전체 파일의 목록을 정렬하는 데는 B+ 트리 기반의 디렉토리가 더 우수한 성능을 보였다.

ABSTRACT

Recently, information systems that require storage and retrieval of huge amount of data are becoming used widely. Accordingly, research efforts have been made to develop Linux cluster file systems in the SAN environment in which clients themselves can manage metadata and access data directly. Also a semi-flat directory structure based on extendible hashing has been proposed to support fast retrieval of files[1]. In this research, we have designed and implemented the semi-flat extendible hash directory under the Linux system. In order to evaluate the practicality of the directory, we have also implemented the B+-tree based directory and experimented the performance. According to the performance comparisons, the extendible hash directory has the better performance at insert, delete, and search operations. On the other hand, the B+-tree directory is better at sorting files.

키워드 : SAN, 디렉토리 구조, 해시 함수, 확장 해싱, B+ 트리

Storage Area Network, Directory Structure, Hash Function, Extendible Hashing, B+ Tree

* 동국대학교 컴퓨터공학과

** 동국대학교 컴퓨터멀티미디어공학과 교수

1. 서 론

최근 인터넷을 통한 전자상거래의 활성화로 인하여 기업들의 데이터가 매년 75~150%의 비율로 증가하고 기업들 간의 네트워크를 통해 전달되는 데이터의 양은 9개월마다 2배로 증가함에 따라 기업이 요구하는 스토리지의 양은 기하급수적으로 늘어나고 있는 실정이다[19]. 따라서 데이터를 안전하게 보관하는 것은 물론, 저장된 데이터에 얼마나 효율적으로 접근하고 사용하는 데에 관심이 모아지고 있다. 그러므로, 대용량의 데이터 저장을 위한 저장 장치들이 필요하게 되었으며, 이를 위하여 네트워크 기반 공유 저장 장치를 이용하고 작은 규모의 컴퓨터들을 클러스터로 연결하여 하나의 통합된 시스템을 구축하려는 연구가 활발히 진행 중에 있다.

이들 파일 시스템들이 대용량의 데이터를 저장하기 위해 사용하는 저장 장치에는 별도의 고속 데이터 전용 네트워크인 Fibre Channel[8, 12]을 통해 클라이언트와 저장 장치들을 연결하는 SAN(Storage Area Network)이 있다. 최근에 이를 이용한 파일 시스템으로 미네소타 대학에서 구현된 GFS(Global File System)[15, 16]를 들 수 있으며, 국내에는 한국전자통신연구원에서 개발하고 있는 SANtopia[1, 2, 9]가 이에 해당된다. 이와 같은 SAN 기반 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[3]. 그리고 각각의 클라이언트는 독립적으로 저장 장치들에게 데이터를 요구할 수 있으며, 하나의 장치에 이상이 생겨도 나머지 장치들에 거의 영향을 주지 않는 이점을 가진다.

SAN을 이용한 대용량 파일 시스템으로 가장 대표적인 GFS[15, 16]의 주된 특징으로는 inode의 플랫 파일 구조(Flat File Structure)를 가지고 있어, 모든 데이터 블록들은 트리의 높이와 같은 리프 레벨에만 위치한다는 점이다. 이는 모든 데이터 블록의 임의 접근 시간이 같아지도록 하며, 매우 큰 파일의 경우는 트리의 높이를 증가시키면 되므로 파일의 크기에 제한이 없어지는 장점을 갖는 반면, GFS에서는 항상 플랫 구조를 유지하여야 하므로 파일의 크기가 커질수록 데이터 블록의 임의의 평균 접근 시간이 길어지는 결과를 초래한다. 이에, GFS의 플랫 파일 구조에서 데이터 블록 당 접근 시간을 기존보다 줄이기 위해 두 레벨까지 데이터 블록을 할당하는 새로운 세미플랫 파일 구조(Semi-flat File Structure)가 제안되었다[1].

한편, 대부분의 UNIX 시스템[4]에서는 디렉토리 내의 파일 이름들을 파일의 생성 순서로 유지하므로 특정 파일의 이름을 디렉토리 내에서 탐색할 때 순차적으로 검색하여야만 한다. 따라서, 많은 파일들을 포함한 대용량의 SAN 기반 파일 시스템에서 기존의 시스템과 같은 방법으로 디렉토리 구조를 갖게 되면, 특정 파일을 검색하는 데 많은 시간이 소요될 수 있다. 그러므로, SAN 기반 파일 시스템에서는 UNIX 디렉토리에서의 비효율적인 순차적 검색을 극복하기 위해서 확장 해싱(Extendible Hashing)[5, 6, 7, 10]을 이용하거

나 B+ 트리[10, 11, 13, 14]를 이용하여 디렉토리 구조를 관리한다. 확장 해싱은 디렉토리의 파일의 수가 많고 적음에 상관없이 모두 수용 가능하고, 비록 많은 수의 파일이 존재하더라도 해싱의 특성상 빠른 검색이 가능하며, B+ 트리도 디렉토리의 파일의 수가 많고 적음에 관계없이 수용 가능하고, B+ 트리의 특성상 항상 디렉토리 엔트리들을 정렬하여 관리함으로써 디렉토리와 파일의 정보를 순서대로 빠르게 보여줄 수 있다.

본 논문에서는 LINUX 환경의 SAN 기반 대용량 파일 시스템을 위해서 확장 해싱을 이용한 세미플랫 디렉토리 구조를 설계 및 구현하고, 구현된 시스템의 실용성을 평가하기 위하여 B+ 트리를 이용한 디렉토리 구조를 구현하고, 성능 평가를 통해 두 디렉토리 구조를 비교 분석한다.

2. 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 삽입할 수 있으며, 삭제 및 탐색도 이와 같은 방법으로 할 수 있다. 확장 해싱은 데이터를 순차 접근이 아닌 직접 접근하므로 빠른 연산이 가능하다.

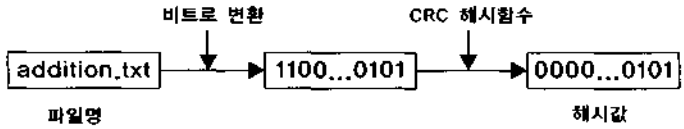
2.1 확장 해싱

확장 해싱은 두 단계의 조직인 디렉토리와 리프의 집합으로 구성되어 있다. 디렉토리는 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[10]. 확장 해싱은 해시 함수를 이용하여 해시 값을 알아내어, 해시 값의 처음 d 비트를 디렉토리에 대한 인덱스로 사용하여 디렉토리에서 엔트리 블록을 찾아서 해당 엔트리에 접근한다.

2.2 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)[17]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 1>은 파일의 이름을 CRC 해시 함수에 적용하여 해시 값을 알아내는 과정이다.

처음에 파일 이름을 입력받아서 이를 비트



〈그림 1〉 해시 함수를 적용하는 과정

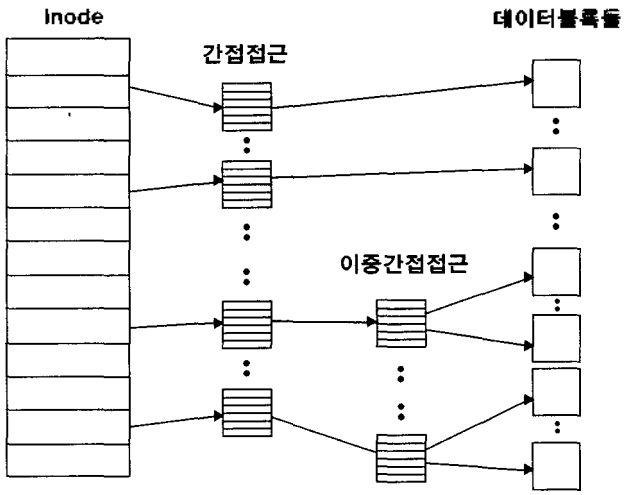
로 변환한다. 이 때 변환된 비트는 최소 32비트 이상이 되며, CRC-32에서 비트 변환에 사용하는 키 값인 0x04c11db7로 나눈다. 그러면 32비트 미만의 나머지가 생기고 이를 해시 값으로 사용하기 위해서 상위 비트를 0으로 채우면 32비트의 해시 값을 구할 수 있다.

2.3 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 inode의 세미플랫 구조를 이용하며 확장된다. 디렉토리는 다음과 같이 세 경우로 나누어 관리한다.

2.3.1 세미플랫 디렉토리 구조

〈그림 2〉는 새로운 세미플랫(Semi-flat) 구조로, 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 GFS에서처럼 플랫 구조를 가질 수도 있고 〈그림 2〉



〈그림 2〉 Inode의 세미플랫 구조

처럼 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.

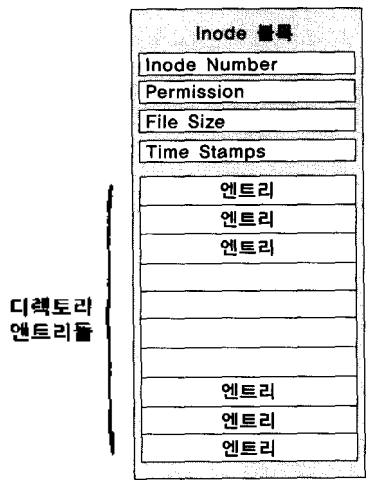
2.3.2 Inode 블록에 통합된 디렉토리 엔트리

〈그림 3〉은 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.

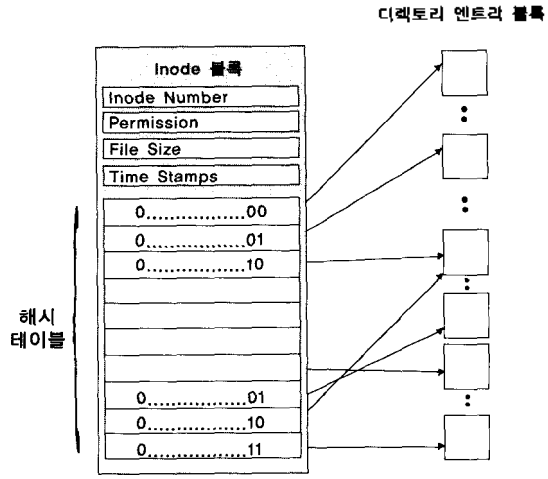
한편, 한 블록의 크기를 4KB로 하였다고 가정하면, 보통 파일 이름의 길이를 8bytes라 생각할 때 166개의 디렉토리 엔트리들을 저장할 수 있다. 그러나 이 블록에 새로운 디렉토리 엔트리를 삽입하고자 할 때 오버플로우가 발생하면, 디렉토리 구조는 〈그림 4〉와 같이 확장 해싱을 이용하기 위한 단일 확장 해시 구조로 전환된다.

2.3.3 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 4〉처럼 inode 블록에 해시 테이블을 저장하고, inode 블록에 저장되어 있던 디렉토리 엔트리들을 해시 테이블을 이용하여 각각의 디렉토리 엔트리 블록으로 이동한다. 이때 해시 테이블 엔트리들은 독립적인



〈그림 3〉 Inode 블록에 통합된 디렉토리 엔트리



〈그림 4〉 Inode 블록에 통합된 해시 테이블 구조

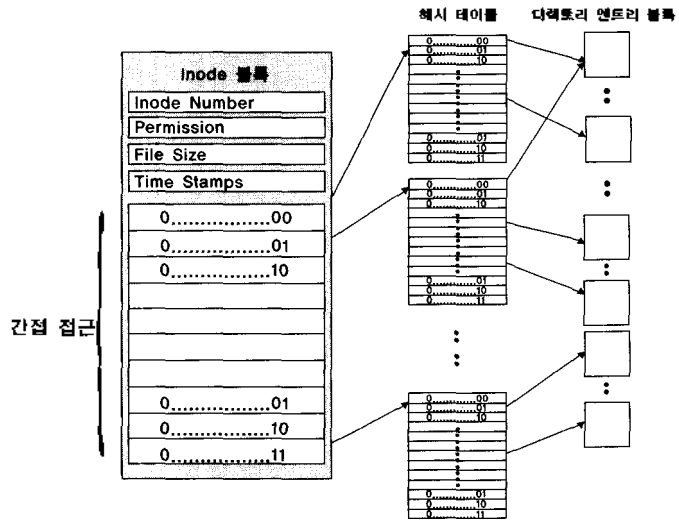
디렉토리 엔트리 블록을 가질 수도 있고, 여러 해시 테이블 엔트리들이 동시에 하나의 디렉토리 엔트리 블록을 공유할 수도 있다.

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 블록의 수를 감소시킬 수 있다는 장점이 있다.

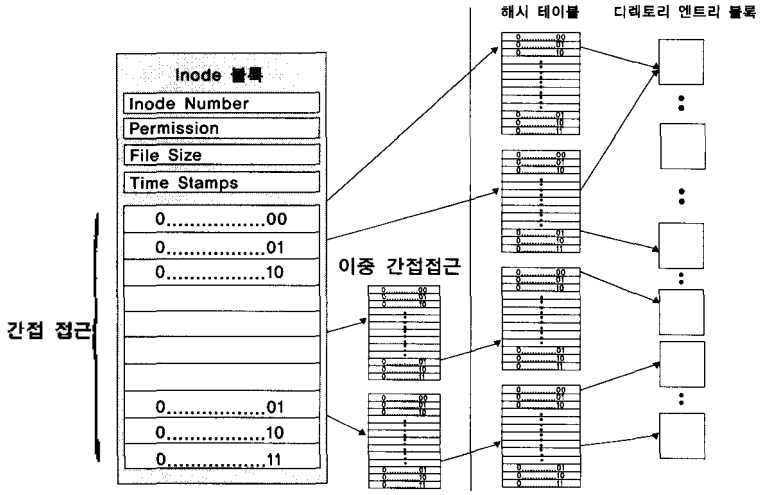
한편, inode 블록에 통합된 해시 테이블을 최대 2^8의 엔트리들을 가질 수 있다고 가정하고, 이때 166개의 디렉토리 엔트리들을 저장할 수 있는 디렉토리 엔트리 블록이 50%만 채워졌다고 생각할 때, 약 1만개 ($256 \times 1/2 \times 166 \times 1/2 = 10,500$) 이상의 디렉토리 엔트리를 저장할 수 있다. 이러한 디렉토리 구조에서 삽입 시에 오버플로우가 발생하면 〈그림 5〉와 같이 inode 블록을 이용하여 해시 테이블에 간접 접근하는 디렉토리 구조로 전환된다.

2.3.4 간접 접근 해시 테이블 구조

Inode 블록에 통합된 해시 테이블 구조에서 새로운 디렉토리 엔트리를 삽입할 때 오버플로우가 발생하면, 디렉토리 구조는 〈그림 5〉와 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 대용량의 디렉토리 엔트리들을 수용할 수 있고, 찾고자 하는 디렉토리 엔트리를 해시 값을 이용하여 빠르게 검색할 수 있다. 한편, 시스템의 간접 접근 해시



〈그림 5〉 간접 접근 해시 테이블 구조



〈그림 6〉 세미플랫 구조를 이용한 확장 해싱 디렉토리 구조

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 × 256×1/2 × 166×1/2 = 1,359,872) 이상의 디렉토리 엔트리를 저장할 수 있다.

간접 접근 해시 테이블 구조에서 새로운 디렉토리 엔트리를 삽입할 때 오버플로우가 발생하면, 디렉토리 구조는 〈그림 6〉과 같이 세미플랫 구조를 이용한 확장 해싱 디렉토리 구조로 전환된다. 〈그림 6〉에서 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 플래 구조에서 오버플로우가 발생하며 레벨이 하나 증가하고 세미플랫 구조로 변화되며 확장된다.

해시값이 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 ($(256\times1/2)^4 \times 166\times1/2$ = 22,280,142,848) 정도의 디렉토리 엔트리를 저장할 수 있다.

2.4 디렉토리 연산

변화된 디렉토리 구조로 새로운 파일을 저장하고자 할 때는 〈그림 7〉에서와 같이 exhash_insert 함수를 이용한다. 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 연

```
알고리즘 exhash_insert
입력 : 파일의 이름(name)
출력 : 파일 저장
{
 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(찾은 디렉토리 엔트리 블록의 링크수 == 1) {
       /* hash_table glow */
       if(해시 테이블 블록에 오버플로우 발생) {
        /* 레벨 증가 */
          레벨의 증가를 위해서 2개의 디렉토리 블록 할당;
          디렉토리 구조 2배로 확장;
       }
       else {  /* 같은 레벨에서의 디렉토리 구조 확장 */
          해시테이블 블록 할당, 디렉토리 구조 2배로 확장;
       }
       비교하는 비트수 증가;
       새로운 디렉토리 엔트리 블록 할당;
    }
    else {  /* hash_table split */
      새로운 디렉토리 엔트리 블록할당;
      링크 포인터 변경;
    }
  }else  찾은 공간에 데이터 저장;
}
```

〈그림 7〉 확장 해싱 디렉토리의 엔트리 삽입

```
알고리즘 exhash_delete
입력 : 파일의 이름(name)
출력 : 파일 삭제
{
 파일의 이름을 이용하여 해시 값을 계산;
 해시 값을 이용하여 해시 테이블에 연결된 알맞은 디렉토리 엔트리 블록을 찾음;
  if(찾은 디렉토리 엔트리 블록에 언더플로우 발생) {
       /* hash_table merge */
       링크 포인터 변경;
       엔트리 블록 반환;
    if(모든 디렉토리 엔트리 블록의 링크수 == 짝수) {
        /* hash_table under */
        디렉토리 구조 1/2배로 축소;
        if(해시테이블 블록에 언더플로우 발생) {
           /* 레벨 감소 */
           레벨의 감소를 위해서 디렉토리 블록 반환;
        }
       비교하는 비트수 감소;
       엔트리 블록 반환;
    }
  }else  찾은 공간에 데이터 삭제;
}
```

〈그림 8〉 확장 해싱 디렉토리의 엔트리 삭제

결된 링크의 수를 본다. 하나의 링크 포인터로 연결된 경우에는 해시 테이블 블록에 오버플로우가 발생한지 조사하여 레벨의 증가 여부를 결정하고 디렉토리 크기를 2배로 증가

한다. 그렇지 않을 경우에는 새로운 디렉토리 엔트리 블록을 추가하여 링크 포인터를 수정해 준다.

〈그림 8〉은 확장 해싱 디렉토리에서 기존의 파일을 삭제하고자 할 때 사용하는 exhash_delete 함수이다. 이 함수는 파일의 이름으로 해시 함수를 사용하여 해시 값을 구하고, 이를 이용하여 삭제할 디렉토리 엔트리 블록을 찾아가서 해당 디렉토리 엔트리를 삭제한다.

그러나, 할당하고자 하는 디렉토리 엔트리 블록에서 언더플로우가 발생하면 링크 포인터를 변경해 주고 빈 엔트리 블록을 반환한다. 그리고, 모든 디렉토리 엔트리 블록의 링크 포인터의 수를 검색하여 모두 짝수면, 디렉토리 크기를 1/2배로 축소하며, 해시 테이블 블록에 언더플로우가 발생하면 레벨이 하나 감소하고, 디렉토리 블록들은 반환한다.

3. B+ 트리를 이용한 디렉토리 구조

SAN 파일 시스템은 B+ 트리를 이용하여 디렉토리 구조를 관리할 수 있다. 디렉토리에 삽입하고자 하는 디렉토리 엔트리 이름을 인덱스 세트에 있는 키 값들과 비교하여 저장할 데이터 엔트리 블록의 주소를 찾아 삽입할 수 있고, 삭제 및 탐색도 이와 같은 방법으로 할 수 있으므로, B+ 트리도 데이터를 순차 접근하는 것에 비해 빠른 연산이 가능하다.

3.1 B+ 트리

B+ 트리는 균형된 m-원 탐색 트리(m-way search tree)로, 인덱스 세트(index set)와 순차 세트(sequence set) 두 부분으로 이루어져 있다[10]. 인덱스 세트는 내부 노드로 리프에 있는 키들에 대한 경로 정보를 제공하며, 순차 세트는 리프 노드로 모든 키 값들을 포함하고 있으며 리프 노드에 있는 모든 디렉토리 엔트리들은 순차적으로 서로 연결되어 있다.

다음은 차수가 m인 B+ 트리의 특성이다[10].

① 루트는 0, 2, $\lceil m/2 \rceil$ ~ m 서브트리를 갖는다.
② 루트, 리프 제외한 모든 노드는 $\lceil m/2 \rceil$ ~ m 서브트리를 갖는다.
③ 모든 리프는 동일 레벨에 있다.
④ 리프가 아닌 노드의 키 값 수는 '서브트리수-1' 이다.

3.2 B+ 트리 디렉토리

앞 절에서 기술한 바와 같이 디렉토리 구조는 파일의 수를 고려하여 결정한다. 해싱을 이용할 때와 같은 방법으로 파일의 수가 적을 때는 inode 블록에 디렉토리 엔트리를 함께 저장하고, 파일의 수가 많아져 inode 블록에서 오버플로우가 발생하면 B+ 트리를 이용할 수 있는 구조로 변하여 디렉토리 엔트리들을 저장한다. 디렉토리는 다음과 같이 세 경우로 나누어 관리한다.

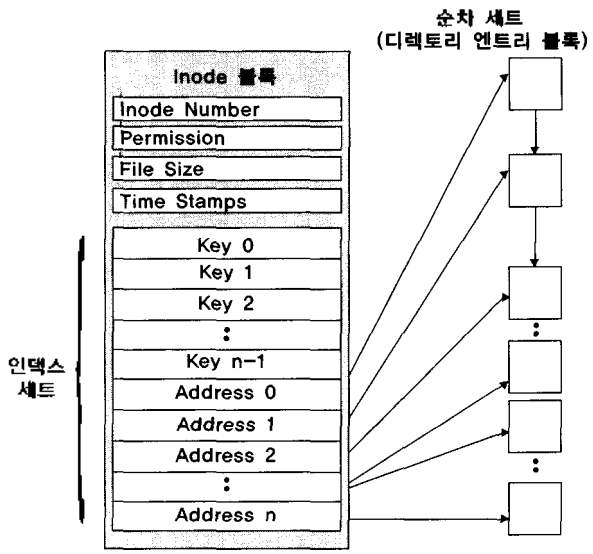
3.2.1 Inode 블록에 통합된 디렉토리 엔트리

디렉토리 엔트리 수가 적을 때는 디렉토리 엔트리들을 inode 블록에 직접 저장(stuffing)함으로써 inode 블록 한 번의 접근으로 디렉토리 엔트리를 검색할 수 있으며, 이는 앞에서 소개된 〈그림 3〉에서 볼 수 있다. 이 inode 블록에 새로운 디렉토리 엔트리를 삽입 하고자 할 때 오버플로우가 발생하면, 디렉토리 구조는 〈그림 9〉와 같이 B+ 트리를 이용하기 위한 확장 구조로 전환된다.

3.2.2 Inode 블록에 통합된 인덱스 세트 구조

Inode 블록에 통합된 디렉토리 엔트리 구조에서 새로운 디렉토리 엔트리를 삽입하고자 할 때 오버플로우가 발생하면, 디렉토리 구조는 B+ 트리를 이용하는 디렉토리 구조로 전환된다. 〈그림 9〉처럼 inode 블록에 인덱스 세트를 저장하고, inode 블록에 저장되어 있던 디렉토리 엔트리들을 인덱스 세트의 키 값과 비교하여 각각의 디렉토리 엔트리 블록으로 이동한다. 예를 들면, 새로 삽입할 디렉토리 엔트리의 값이 key0보다 작으면 Address0에 연결된 디렉토리 엔트리 블록에 저장하고, key0보다 크고 key1보다 작으면 Address1에 연결된 디렉토리 엔트리 블록에 저장한다. 그리고 디렉토리 엔트리 블록들은 차례로 연결되어 있다. 즉, 이 부분이 순차 세트이다.

이와 같은 디렉토리 구조에서는 하나의 디렉토리 엔트리 블록에서 오버플로우가 발생하면 이 디렉토리 엔트리 블록을 가리키는 인덱스 블록을 조사하여, 삽입할 공간이 있으면 새로운 디렉토리 엔트리 블록을 할당받아 연결시켜 디렉토리 엔트리들을 재분배하고, 더



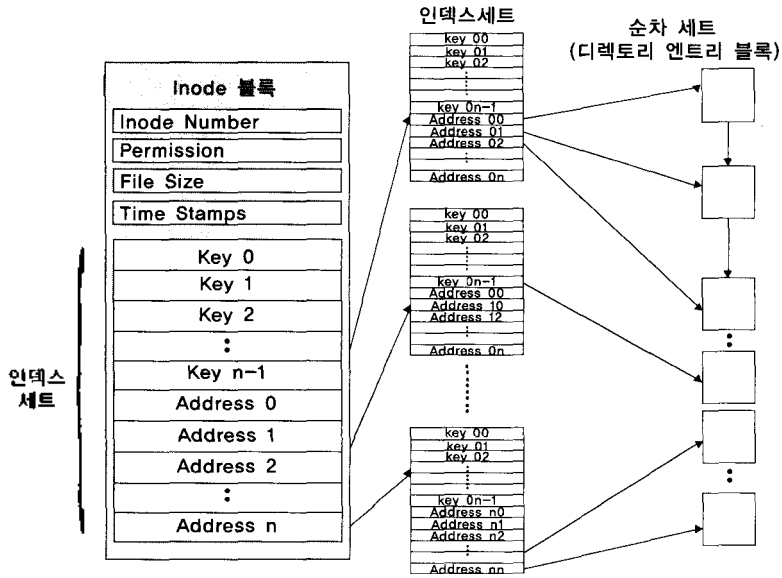
〈그림 9〉 inode 블록에 통합된 인덱스 세트 구조

이상의 삽입할 공간이 없으면 〈그림 10〉과 같이 B+ 트리의 레벨이 한단계 증가한 확장된 인덱스 세트 구조로 전환된다. Inode 블록에 통합된 인덱스 세트 구조는 인덱스 세트 블록을 따로 두어 데이터 블록에 접근하는 방법에 비해 접근하는 블록의 수를 감소시킬 수 있다는 장점이 있다.

한편, inode 블록에 통합된 인덱스 세트 구조는 키 값의 크기를 8bytes로, 주소의 크기를 8bytes로 가정할 때, 차수가 249인 B+ 트리를 만들 수 있다. 이때 디렉토리 엔트리들의 50% 정도만 채워졌다고 하면, 약 1만개 (249×1/2 × 166×1/2 = 10,000) 이상의 디렉토리 엔트리를 저장할 수 있다.

3.2.3. 확장된 인덱스 세트 구조

Inode 블록에 통합된 인덱스 세트 구조에서 새로운 디렉토리 엔트리를 삽입할 때 오버플로우가 발생하면, 디렉토리 구조는 〈그림 10〉과 같이 확장된 인덱스 세트 구조로 전환된다. Inode 블록을 접근하여 인덱스 세트에 있는 키 값과 비교하여 올바른 디렉토리 엔트리 블록을 찾아간다. 이와 같은 디렉토리 구조는 거대한 인덱스 세트 구조를 가질 수 있으므로 대용량의 디렉토리 엔트리들을 수용할 수 있고, 찾고자 하는 디렉토리 엔트리를 inode 블록에서부터 링크를 따라서 찾을 수도 있으며, 리프 노드인 디렉토리 엔트리 블록들을 순차적으로 검색할 수 있다. 한편, 확장된 인덱스 세트 구조는 키 값의 크기를 8bytes로, 주소의 크기를 8bytes로 가정할 때, 차수가 249인 B+ 트리를 만들 수 있다. 이때 높이가 3이고 디렉토리 엔트리들의 50%정도만 채워졌다고 하면, 최소 약 128만개 (249×1/2 × 249×1/2 × 166×1/2 = 1,286,520) 이상의 디렉토리 엔트리를 저장할 수 있다.



〈그림 10〉 확장된 인덱스 세트 구조

위의 구조에서 오버플로우가 발생하면, 앞에서 설명한 방법으로 B+ 트리의 인덱스 블록이 추가되며 트리의 레벨이 증가하게 된다. B+ 트리는 제한 조건이 없으므로 트리의 높이는 무한정으로 늘어날 수 있다.

3.3 디렉토리 연산

〈그림 11〉은 B+ 트리에서의 삽입 연산 함수이다. 파일의 이름을 이용하여 저장할 디렉토리 엔트리 블록을 찾아간다. 그러나, 할당하고자 하는 디렉토리 엔트리 블록이 오버플로우가 발생하면 새로운 디렉토리 엔트리 블록을 할당하고 그 공간에 찾은 디렉토리 엔트리 블록 데이터의 절반을 넣는다.

이때 원래의 삽입하려는 파일 이름이 새로운 디렉토리 엔트리 블록의 첫 번째 엔트리의 파일 이름보다 값이 작으면 찾은 디렉토리 엔트리 블록에 엔트리들을 정렬하여 저장하고, 그렇지 않은 경우에는 새로운 디렉토리 엔트리 블록에 엔트리들을 정렬하여 저장한다. 이후에 새로운 디렉토리 엔트리 블록의 첫 번째 엔트리의 파일 이름을 키로 하여 부모 노드에 삽입한다. 이때 부모 노드가 오버플로우 될 경우 분할 연산을 통해서 B+ 트리를 재구축한다.

〈그림 12〉는 B+ 트리에서의 삭제 연산 함수이다. 이 함수는 파일의 이름을 이용하여 삭제할 디렉토리 엔트리 블록을 찾아가서 해당 디렉토리 엔트리를 삭제한다.

그러나, 삭제하고자 하는 디렉토리 엔트리 블록의 엔트리 수가 1/2 이하가 되어 언더플로우가 발생하면 해당 디렉토리 엔트리 블록을 반환하고 부모 노드에서 링크하고 있던 값을 삭제한다. 이때 부모 노드에서 언더플로우가 발생할 수 있는데 병합 연산을 통해서 B+ 트리를 재구축하게 된다.

```
알고리즘 bpt_insert
입력: 파일의 이름(name)
출력: 파일 저장
{ 파일의 이름을 이용하여 B+ 트리에 연결된 알맞은 디렉토리
  엔트리 블록을 찾음;
  if(찾은 디렉토리 엔트리 블록에 오버플로우 발생)
    { 새로운 디렉토리 엔트리 블록 할당;
      찾은 디렉토리 엔트리 블록의 반을 새로운 디렉토리
      엔트리 블록으로 이동;
     if(파일의 이름(새로운 디렉토리 엔트리 블록의 첫 번째 값)
       찾은 디렉토리 엔트리 블록에 데이터를 정렬하여 저장;
     else
       새로운 디렉토리 엔트리 블록에 데이터를 정렬하여 저장;
    }
  새로운 디렉토리 엔트리 블록의 첫번째 엔트리 값을 키로 하여
  B+ 트리의 부모 노드에 삽입;
  if(부모 노드에 오버플로우 발생)
     부모 노드를 분할;
  else  찾은 공간에 데이터를 정렬하여 저장;
}
```

〈그림 11〉 B+ 트리 디렉토리의 엔트리 삽입

```
알고리즘 bpt_delete
입력 : 파일의 이름(name)
출력 : 파일 삭제
{
 파일의 이름을 B+ 트리에 연결된 알맞은 디렉토리 엔트리
 블록을 찾음;
 찾은 공간에 데이터 삭제;
  if(찾은 디렉토리 엔트리 블록에 언더플로우 발생)
 {   /* hash_table merge */
     해당 디렉토리 엔트리 블록을 반환;
    부모 노드에서 해당 디렉토리를 가르키는 값을 삭제;
    if(부모 노드에서 언더플로우 발생)
    {  /* hash_table under */
       부모 노드를 병합;
     }
  }
}
```

〈그림 12〉 B+ 디렉토리의 엔트리 삭제

4. 성능 평가

본 절에서는 SAN 기반 대용량 파일 시스템을 위해 구현된 확장 해싱 세미플랫 디렉토리 구조와 B+ 트리를 이용한 디렉토리 구조의 성능을 평가한다.

4.1 실험 환경

성능 실험에 사용된 컴퓨터는 펜티엄 3-533MHz를 사용하였으며, Redhat 6.2 버전의 LINUX를 설치하고, 커널을 6.2 버전의 기본 커널인 2.2.16을 2.2.18로 업그레이드한 환경에서 구현언어는 C언어로 하며, LINUX 6.2버전에서 지원하는 GCC gcs-2.91.66버전을 이용하였다. 실험에 이용된 디스크는 Seagate사의 ST51080A 모델[18]이며 디스크의 성능은 〈표 1〉과 같다.

성능 실험은 쉘 프로그램을 이용하여 디렉토리 엔트리들을 일정 수만큼 계속 삽입, 삭제, 탐색 그리고 정렬을 하면서 그때마다 소요되는 시간을 측정하였다. 실험 결과는 다음 절에서 설명한다.

4.2 실험 결과

본 절에서는 확장 해싱을 이용하여 구현한 디렉토리와 B+ 트리를 이용하여 구현한 디렉토리에서의 디렉토리 엔트리들의 삽입, 삭제, 탐색, 그리고 정렬하는 데에 소요되는 시간을 측정하여 비교한다.

실험 결과를 위해서 디렉토리의 규모에 따라 두 경우로 나누어서 성능 실험을 하였다. 하나는 inode 블록에 통합된 해시 테이블 구조에서 최대 약 40,000개정도 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 수용

〈표 1〉 Seagate사의 ST51080A 디스크의 성능

사 양	
capability	1.08 GB
sector size	512 Bytes
data head	64
cylinder	2,100
sector per track	63
disks	2
성 능	
rotational speed	5,376 RPM
media transfer rate	67.7 Mb/sec
latency average	5.58 ms
탐구시간	
seek average	10.5 ms
track to track	2.0 ms
full track	20.0 ms

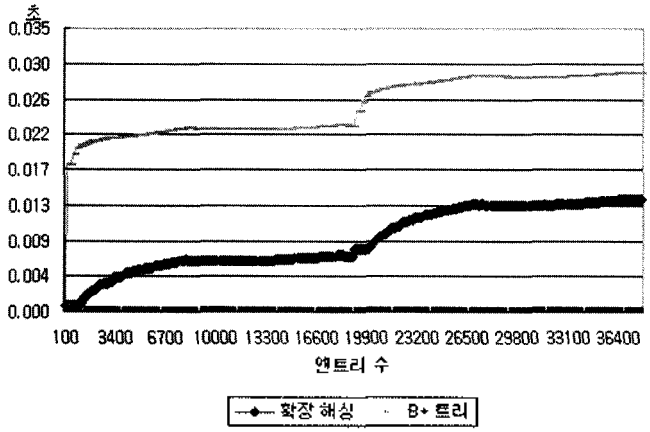
하는 대규모 디렉토리에 대한 성능 실험이다. 본 논문에서는 각 성능실험마다 10번 실험을 통해 나온 결과 값을 기반으로 평균값을 구하였다.

4.2.1 디렉토리 엔트리 삽입

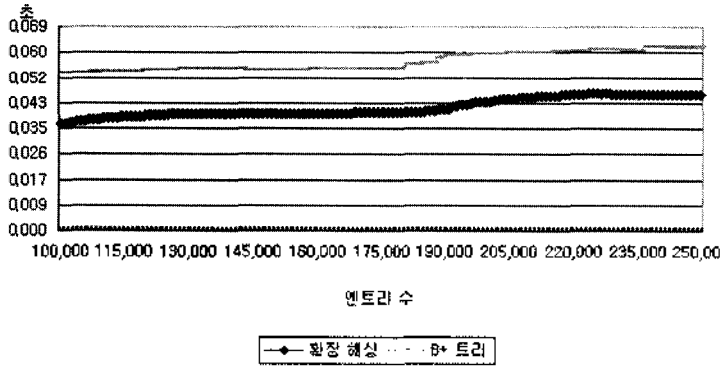
확장 해싱을 이용한 새미플랫 디렉토리 구조에서와 B+ 트리를 이용한 디렉토리 구조에서의 디렉토리 엔트리를 삽입하는 데 소요되는 엔트리 당 평균 수행 시간의 성능 평가 결과는 〈그림 13〉과 같다. 〈그림 13〉은 소규모의 디렉토리에 대한 성능 실험 결과이다.

두 디렉토리 그래프에서 갑자기 증가하는 부분은 레벨이 하나 증가하는 부분을 나타낸다. 그리고, 전체적으로 B+ 트리 디렉토리 구조가 확장 해싱 디렉토리 구조보다 디렉토리 엔트리를 삽입하는 데에 많은 시간을 소요함을 볼 수 있는데, 이는 B+ 트리 디렉토리 구조에서는 디렉토리 엔트리를 삽입할 때 확장 해싱처럼 바로 디렉토리 엔트리 블록을 찾아가는 것이 아니라, 인덱스 세트의 키 값을 비교하면서 삽입할 디렉토리 엔트리 블록을 찾아가기 때문이다.

〈그림 14〉는 엔트리 수 약 250,000개정도 수용할 수 있는 대용량 파일 시스템을 위한 대규모 디렉토리에 대한 것으로 해서 위의 실험과 같은 조건으로 시뮬레이션을 통해 얻은 결과이다. 이 그림에서도 〈그림 13〉과 같이 엔트리 수가 많아짐에 따른 하나의 임의의 엔트리 당 삽입시간 역시 B+ 트리 디렉토리 구조



〈그림 13〉 소규모 디렉토리 엔트리 당 평균 삽입 시간



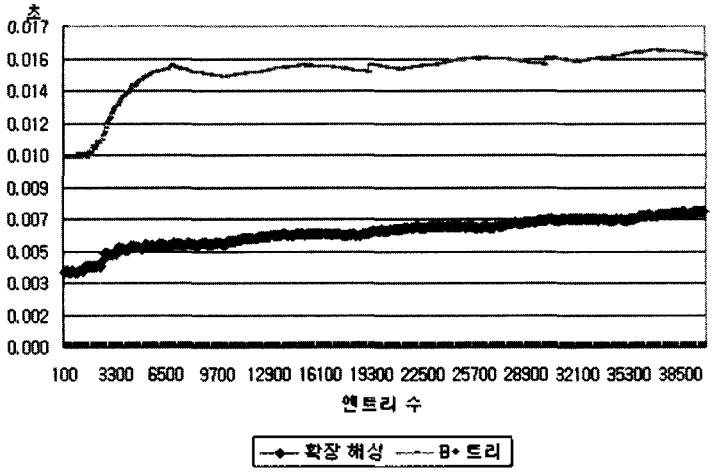
〈그림 14〉 대규모 디렉토리 엔트리 당 평균 삽입 시간

가 확장 해싱 디렉토리 구조보다 디렉토리 엔트리를 삽입하는 데에 많은 시간을 소요함을 볼 수 있다.

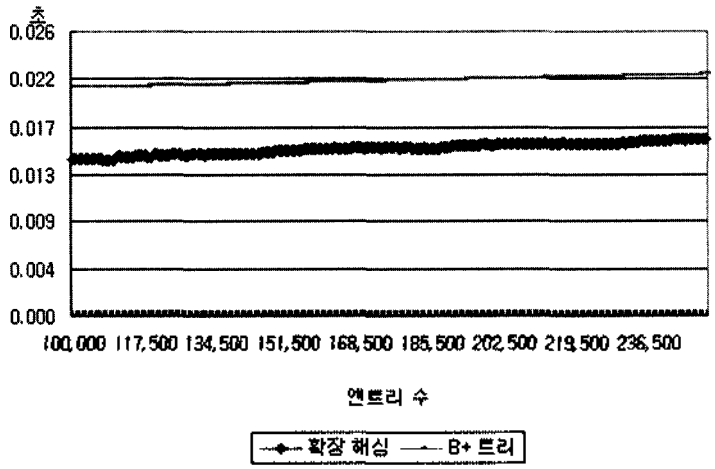
4.2.2 디렉토리 엔트리 삭제

앞에서와 동일한 방법으로 실험한 확장 해싱을 이용한 디렉토리 구조에서와 B+ 트리를 이용한 디렉토리 구조에서의 디렉토리 엔트리를 삭제하는 데 소요되는 엔트리 당 평균 수행 시간의 성능 평가 결과는 〈그림 15〉와 같다. 여기서도 디렉토리 엔트리 삭제에 확장 해싱을 이용한 디렉토리 구조가 엔트리 당 평균 삭제 시간이 적게 걸림을 알 수 있다.

확장 해싱에서는 삭제할 엔트리의 값을 해시 함수에 의해 해시 값을 알아낸 후 그 값을 주소로 하여 삭제할 위치를 정하기 때문에, 삭제할 엔트리 값을 인덱스 세트의 키 값과 크고 작음을 비교하여 삭제하고자 하는 디렉



〈그림 15〉 소규모 디렉토리 엔트리 당 평균 삭제 시간



〈그림 16〉 대규모 디렉토리 엔트리 당 평균 삭제 시간

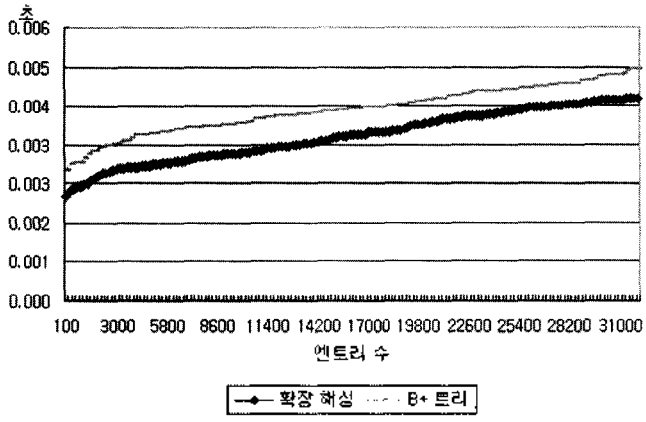
토리 엔트리가 저장된 위치를 찾는 B+ 트리에 비해서 삭제 시간이 적게 걸리게 됨을 볼 수 있다. 또, 〈그림 13〉과 비교하여 보면, 삭제 시간이 삽입 시간보다 시간을 적게 소요함을 알 수 있는데, 이는 디렉토리 엔트리를 삽입할 때, 새로운 블록을 할당받고 정리하는 일이 단순히 디렉토리 엔트리를 삭제하는 것에 비해 시간이 많이 요구되기 때문이다.

〈그림 16〉 역시 대규모 디렉토리에 대해서 〈그림 15〉의 실험과 같은 조건으로 시뮬레이션을 통해 얻은 결과이다. 엔트리 삽입 때와 비슷하게 B+ 트리 디렉토리 구조가 확장 해싱 디렉토리 구조보다 디렉토리 엔트리를 삭제하는 데에 많은 시간을 소요함을 볼 수 있다.

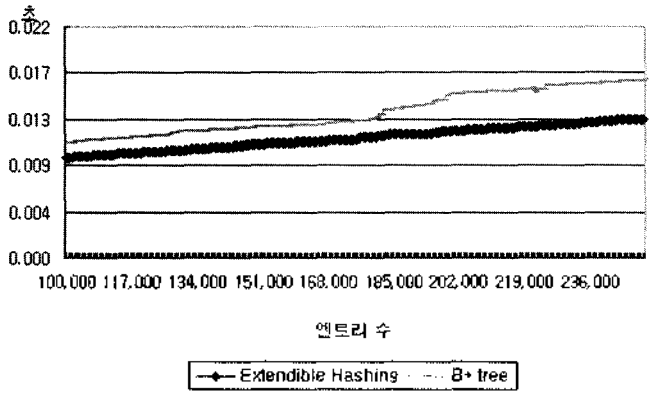
4.2.3 디렉토리 엔트리 탐색

소규모 디렉토리에서의 확장 해싱을 이용한 디렉토리 구조에서와 B+ 트리를 이용한 디렉토리 구조에서의 디렉토리 엔트리 당 평균 탐색 시간은 〈그림 17〉에서 볼 수 있다. 그림에서 보는 바와 같이, 확장 해싱을 이용한 디렉토리가 B+ 트리를 이용한 디렉토리보다 평균 파일 탐색 시간이 약간 적게 소요됨을 알 수 있다. 이것은 확장 해싱은 해시 값을 이용하여 탐색하고자하는 데이터가 존재하는 데이터 엔트리 블록에 바로 접근하는데 비해, B+ 트리는 데이터 엔트리가 존재하는 리프 노드까지 가기 위해서 중간 노드에서 파일을 순차적으로 비교하기 때문이다.

〈그림 18〉은 대규모 디렉토리를 위한 두 디렉토리 구조의 성능을 시뮬레이션을 통해 얻은 결과이다. 실험을 통한 결과와 마찬가지로 많은 수의 엔트리에 대해서도 B+ 트리 디렉토리 구조가 확장 해싱 디렉토리 구조보다 디렉토리 엔트리를 탐색하는 데에 약간의 시간을 더 소요함을 볼 수 있다.



〈그림 17〉 소규모 디렉토리 엔트리 평균 탐색 시간



〈그림 18〉 대규모 디렉토리 엔트리 평균 탐색 시간

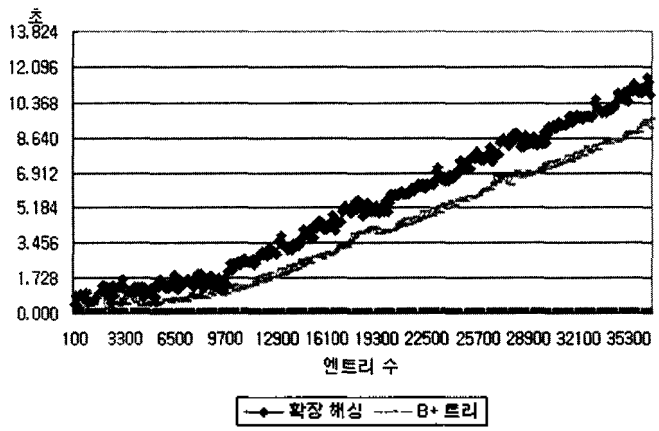
4.2.4 디렉토리 엔트리 정렬

디렉토리 엔트리 정렬은 엔트리의 수에 따라 결정한다. 적은 수의 엔트리가 있을 경우에는 메모리 내부에서 빠르게 수행되는 퀵 정렬(Quick Sort)[10]을 통하여 엔트리를 정렬하고, 많은 수의 엔트리가 있을 경우에는 메모리 밖의 외부에서 할 수 있는 합병 정렬(Merge Sort)[10]을 통하여 엔트리를 정렬한다.

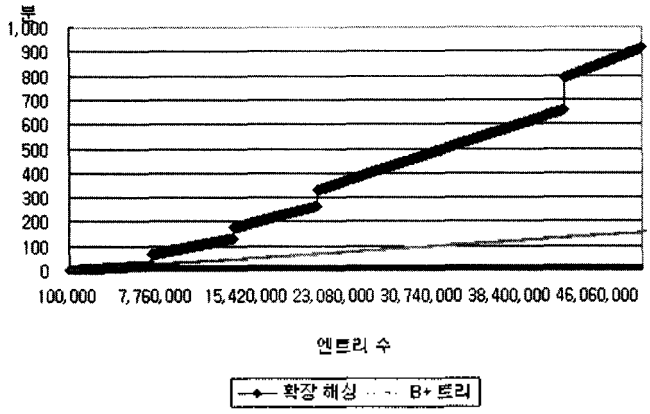
〈그림 19〉은 소규모 디렉토리에서의 확장 해싱을 이용한 디렉토리 구조에서와 B+ 트리를 이용한 디렉토리 구조에서의 디렉토리 엔트리를 정렬하는 데 소요되는 평균 수행 시간을 10번의 실험을 통해 얻은 결과이다. B+ 트리를 이용한 디렉토리 구조의 성능이 좋음을 알 수 있다.

그림과 같이, 두 그래프는 디렉토리 엔트리의 수가 증가하면 증가할수록 정렬하는데 소요되는 시간이 증가한다. 그러나 B+ 트리를 이용한 디렉토리 구조에서는 확장 해싱을 이용한 디렉토리 구조에 비해서, 디렉토리 엔트리들을 정렬하는 시간은 완만히 증가함을 볼 수 있고, 엔트리의 수가 많으면 많아질수록 두 그래프의 폭이 점차 넓어짐을 알 수 있다. 이것은 확장 해싱에서는 디렉토리 엔트리들을 블록에 삽입할 때 순서와 상관없이 흩어져서 저장되어 있으므로 이들을 전체적으로 정렬해야하지만, B+ 트리에서는 디렉토리 엔트리들을 블록에 삽입할 때 인덱스 세트의 키 값과 비교하여 순차적으로 저장하므로 이미 정렬된 순차세트 블록을 차례대로 읽어오기만 하면 되므로, 훨씬 빠르게 디렉토리 엔트리들을 정렬할 수 있는 것이다.

〈그림 20〉은 대규모 디렉토리에서의 두 디렉토리 구조의 성능을 시뮬레이션을 통해서 얻은 결과이다. 특히, 엔트리 수가 너무 많아 메모리 버퍼에서 수용하지 못할 정도의 많은 블록들이 생겼을 경우에 대해서 소요되는 정렬 시간을 분석하였다. 시뮬레이션을 위해서 메모리 크기를 256MB, 한 블록의 크기를 4KB로 가정하면, 166개의 엔트리를 저장할 수 있는 한 블록에는 2/3정도의 엔트리가 채워져 있다고 가정하고 50,000,000개의 엔트리



〈그림 19〉 소규모 디렉토리 엔트리들 정렬 시간



〈그림 20〉 대규모 디렉토리 엔트리들 정렬 시간

가 있을 경우를 분석하였다.

그림에서 보는 바와 같이, B+ 트리를 이용한 디렉토리 구조는 단순히 정렬된 데이터 블록들만을 읽어오는데 비해, 확장 해싱을 이용한 디렉토리 구조는 엔트리 수가 많아져 블록수가 많아지게 되고, 블록수가 많아져 내부 정렬이 불가능해짐에 따라 합병 정렬을 이용하게 되므로 사용되는 빈번한 블록의 I/O 작업으로 인해 확장 해싱 디렉토리 구조가 B+ 트리 디렉토리 구조보다 디렉토리 엔트리의 정렬하는 데에 많은 시간을 소요함을 볼 수 있다. 확장 해싱을 이용한 디렉토리 구조에서는 메모리에서 한번 읽어올 수 있는 데이터 블록의 수 만큼의 증가가 있을 때마다 I/O 작업의 단계별 증가가 일어남으로 그림과 같은 갑자기 요구되는 시간이 급격하게 증가하는 경우가 발생한다. 이와 같이 두 디렉토리 구조의 정렬하는데 소요되는 시간은 디렉토리 엔트리 수가 많아지면 많아질수록 B+ 트리를 이용한 디렉토리 구조의 성능이 우수함을 알 수 있다.

5. 결 론

논문에서는 대규모 SAN 기반 대용량 파일 시스템을 위한 확장 해싱(Extendible Hashing) 디렉토리 구조와 B+ 트리 디렉토리 구조를 설계 및 구현하였다. 이들 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 이용한 해시 함수를 적용하여 확장 해싱 기법으로 디렉토리 공간에 저장함으로써 엔트리 정보를 빠르게 검색할 수 있게 하거나, B+ 트리를 이용하여 디렉토리 공간에 정렬하여 저장함으로써 많은 엔트리들을 빠르게 정렬할 수 있게 하였다.

성능 평가를 통하여 엔트리를 삽입, 삭제, 그리고 탐색 성능에서는 확장 해싱 기법을 이용한 디렉토리 구조가 우수하였으며, 디렉토리 구조에 저장되어 있는 엔트리들을 정렬하

는 데 는 B+ 트리를 이용한 디렉토리 구조가 더 우수한 성능을 보였다.

향후에는 실제 시스템에서 디렉토리 관련 작업 수행 중에 전원공급 등의 오류로 인하여 문제가 발생했을 시에 이를 해결할 시스템의 회복에 관한 연구가 요구된다.

참 고 문 헌

[1] 김신우, 박성은, 이용규, 김경배, 신범주, "SAN 기반 리눅스 클러스터 파일 시스템을 위한 메타데이터 관리," 정보처리학회논문지, 8-A권 4호, pp. 367-374, 2001. 12.

[2] 신범주, 김경배, 김창수, 김명준, "네트워크 저장 장치를 위한 클러스터 파일 시스템 개발," 정보처리학회지, 8권 4호, pp. 29-41, 2001. 7.

[3] Tomas E. Anderson, Michael D. Dahlin and Jeanna M. Neefe, "Serverless Network File Systems," ACM Operating Systems Review, vol. 29, no. 5, pp. 109-126, December 1995.

[4] Maurice J. Bach, The Design of the UNIX Operating System, Prentice-Hall, 1986.

[5] David H.C. Du and Sheau-Ru 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.

[6] Ronald Fagin, et. al., "Extendible Hashing - A Fast Access Method for Dynamic Files," ACM Transactions on Database Systems, vol. 4, no. 3, pp. 315-344, September 1979.

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

[8] Clit Jurgens, "Fibre Channel: A Connection to the Future," IEEE Computer, vol. 28, no. 8, pp. 82-90, August 1995.

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

[10] Panos E. Livadas, File Structures, Prentice-Hall, 1990.

[11] Mario A. Nascimento and Margaret H. Dunham, "Indexing Valid Time Databases via B+-Trees," IEEE Transactions on Knowledge and Data Engineering, vol. 11, no 6, pp. 929-947, November 1999.

[12] Mstthew Y. O'Keefe, "Shared File Systems and Fibre Channel," Proceedings of the 15th IEEE Mass Storage Systems Symposium, pp. 1-16, College Park, Maryland, March 1998.

[13] E. Omiecinski, "Concurrent Storage Structure Conversion: from B+ Tree to

Linear Hash File," Proceedings of the 4th International Conference on Data Engineering, pp. 589-596, Los Angeles, USA, February 1988.

[14] June S. Park and V. Sridhar, "Probabilistic Model and Optimal Reorganization of B+-Tree with Physical Clustering," IEEE Transactions on Knowledge and Data Engineering, vol. 9, no 5, pp. 826-832, September 1997.

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

[16] Steven R. Soltis, et. al., "The Global File System," Proceedings of the 5th NASA Goddard Conference on Mass Storage Systems and Technologies, pp. 319-342, College Park, Maryland, September 1996.

[17] Andrew S. Tanenbaum, Computer Networks, Prentice-Hall, 1996.

[18] Seagate ST51080A Hard-Disk Spec., http://www.seagate.com/support/disc/specs/ata/ st51080a.html, 2003.

[19] 스토리지 아키텍처의 재발견, "http://www.hyosunginformation.co.kr/advantage/no62/docu01.htm" 2003. 11

저 자 소 개

김신우	(E-mail : purian@dgu.edu)
1997.	동국대학교 컴퓨터공학과(학사)
2000.	동국대학교 컴퓨터공학과(석사)
2004.	동국대학교 컴퓨터공학과(박사)
2002. ~ 현재	동국대학교 컴퓨터공학과 강사
관심 분야	XML 및 웹, 스토리지시스템, 데이터베이스

이용규	(E-mail : yklee@dgu.edu)
1986.	동국대학교 전자계산학과(학사)
1988.	한국과학기술원 전산학과(석사)
1996.	Syracuse University 전산학박사
1978 ~ 1983.	정보통신부 국가공무원
1988 ~ 1993.	한국국방연구원 선임연구원
1996 ~ 1997.	한국통신 선임연구원
2002 ~ 2003.	콜로라도대학교 컴퓨터학과 방문교수
1997 ~ 현재	동국대학교 컴퓨터멀티미디어공학과 교수
관심 분야	XML 및 웹, 스토리지시스템, 데이터베이스