제19회 한국정보처리학회 춘계학술발표대회 논문집 제10권 제1호 (2003.5)

대용량 파일 시스템을 위한 디렉토리 구조 비교

김신우, 이현석, 이용규
동국대학교 컴퓨터공학과
e-mail: purian@dgu.edu

Comparison of Directory Structures for Large File Systems

Shin Woo Kim, Hyun Suk Lee, Yong Kyu Lee
Dept. of Computer Engineering, Dongguk University

요 약

최근 데이터가 대용량화됨에 따라 이를 저장할 파일시스템이 필요하게 되었다. 이에 별도의 서버를 두지 않고 분산된 클라이언트가 메타데이터를 직접 관리하면서 모든 저장 장치들에 접근할 수 있는 SAN 기반 리눅스 클러스터 파일시스템의 연구가 활발히 진행 중에 있다. 그러나, 이와 같은 대규모의 파일 시스템에서 일반 UNIX 시스템과 같은 디렉토리 구조를 갖게 되면, 파일 탐색 시 순차검색으로 인하여 많은 시간이 걸리게 된다. 그러므로, 이러한 문제를 해결하고자, 본 논문에서는 SAN 기반 리눅스 클러스터 파일 시스템을 위한 확장 해싱 디렉토리와 B+ 트리 디렉토리 구조를 설계 및 구현하고, 실행 환경에 따라 적합한 구조를 선택할 수 있도록 하기 위하여 성능평가를 통해 두 디렉토리 구조를 비교 분석한다.

1. 서론

최근 시스템에서 사용되는 데이터 파일들의 크기가 커지고 그 수가 점차 많아짐에 따라 이를 저장할 대용량의 저장 장치들이 필요하게 되었다. 따라서, 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 저장장치에는 별도의 고속 데이터 전용 네트워크인 Fibre Channel[2]을 통해 클라이언트와 저장 장치들을 연결하는 SAN(Storage Area Network)이 있다. 최근에 이를 이용한 파일 시스템으로 미네소타 대학에서 구현된 GFS(Global File System)[4]를 들 수 있으며, 국내에는 한국전자통신연구원에서 개발하고 있는 SANtopia[6]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.

한편, 대부분의 UNIX 시스템[1]에서는 디렉토리 내의 파일 이름들을 파일의 생성 순서로 유지하므로 특정 파일의 이름을 디렉토리 내에서 탐색할 때 순차적으로 검색하여야만 한다. 따라서, 많은 파일 이름들을 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 기존의 파일 시스템의 디렉토리에서의 비효율적인 순차적 검색을 극복하기 위해서 확장 해싱(Extendible Hashing)[3]을 이용하거나, B+ 트리[3]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 관리함으로 디렉토리와 파일의 정보를 순서대로 빠르게 보여줄 수 있다.

본 논문에서는 SAN 환경의 대용량 파일 시스템을 위한 확장 해싱을 이용한 디렉토리 구조와 B+ 트리를 이용한 디렉토리 구조를 설계 및 구현하고, 성능 평가를 통해 실행 환경에 따라 적합한 구조를 선택할 수 있도록 디렉토리 구조에 따른 디렉토리 연산 수행 시간을 비교한다.

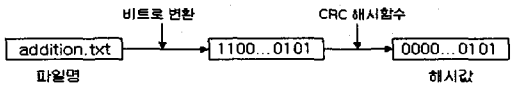
2. 확장 해싱을 이용한 디렉토리 구조

SAN 파일 시스템에서는 확장 해싱을 이용하여 디렉토리를 관리할 수 있다. 디렉토리에 삽입하고자 하

는 디렉토리 엔트리 이름을 이용하여 해시 함수를 통해 해시 값을 구하고, 이를 활용하여 직접 저장할 데이터 블록의 주소를 찾아 삽입할 수 있으며, 삭제 및 검색도 이와 같은 방법으로 할 수 있다. 확장 해싱은 데이터를 순차 접근이 아닌 직접 접근하므로 빠른 연산이 가능하다.

2.1 해시 함수

해시 함수는 데이터 통신에서 에러 검출 코드로 활용되는 CRC-32 코드(32-bit Cyclic Redundancy Check Code)[5]를 사용한다. [그림 1]은 파일의 이름을 CRC 해시 함수에 적용하여 해시 값을 알아내는 과정이다.



[그림 1] 해시 함수를 적용하는 과정

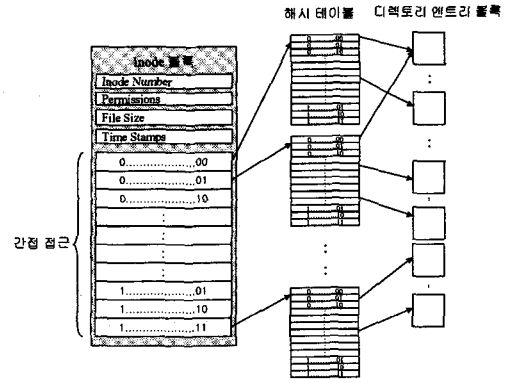
처음에 파일 이름을 입력받아서 이를 비트로 변환한다. 이 때 변환된 비트는 최소 32비트 이상이 되며, CRC-32에서 주로 사용하는 키 값인 0x04c11db7로 나눈다. 그러면 32비트 미만의 나머지가 생기고 이를 해시 값으로 사용하기 위해서 상위 비트를 0으로 채우면 32비트의 해시 값을 구할 수 있다.

2.2 확장 해싱 디렉토리

SAN 파일 시스템의 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 저장한다.

2.2.1. 디렉토리 구조

Inode 블록에 디렉토리 엔트리를 직접 저장할 수 없게되면 확장 해싱을 이용하는 해시 테이블 구조로 전환되며 [그림 2]와 같다. 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, 한 블록의 크기를 4KB, 보통 파일 이름의 길이를 8bytes라 생각할 때 166개의 디렉토리 엔트리들을 저장할 수 있다. 또, 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 × 256×1/2 × 166 ×1/2 = 1,359,872) 이상의 디렉토리 엔트리를 저장할 수 있다.



[그림 2] 확장 해싱을 이용한 디렉토리 구조

2.2.2. 디렉토리 연산

[그림 3]은 확장 해싱에서의 삽입 연산 함수이다.

```
알고리즘 exhash_insert
입력: 파일의 이름(name)
출력: 파일 저장
{ 파일의 이름을 이용하여 해시 값을 계산한다;
  해시 값을 이용하여 해시테이블에 연결된 알맞은
  디렉토리 엔트리 블록을 찾는다;
  if(찾은 디렉토리 엔트리 블록에 오버플로우 발생)
  { if(찾은 디렉토리 엔트리 블록의 링크수 == 1)
     { /* hash_table glow */
       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 3] 확장 해싱 디렉토리의 엔트리 삽입

해시 값을 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 디렉토리 크기를 2배로 커지게 하고, 그렇지 않을 경우에는 새로운 디렉토리 엔트리 블록을 추가하여 링크 포인터를 수정해 준다.

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 찾

아 삽입할 수 있고, 삭제 및 검색도 이와 같은 방법으로 할 수 있다. 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[3]. 인덱스 세트는 내부 노드로 리프에 있는 키들에 대한 경로정보를 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[3].

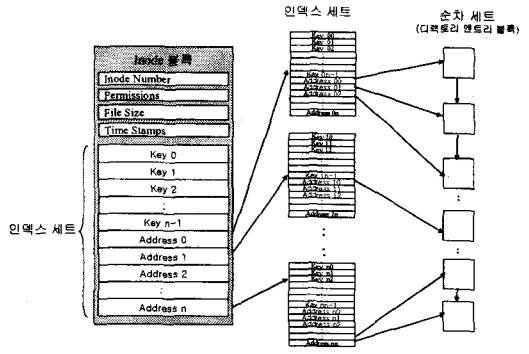
① 루트는 0, 2, ⌈ m/2⌉ ~ m 서브트리를 갖는다.
② 루트,리프 제외한 모든 노드는 ⌈ m/2⌉ ~ m 서브트리를 갖는다.
③ 모든 리프는 동일 레벨에 있다.
④ 리프가 아닌 노드의 키 값 수는 서브트리수-1이다.

3.2 B+ 트리 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 B+ 트리를 이용하는 구조로 변하여 디렉토리 엔트리를 저장한다.

3.2.1. 디렉토리 구조

Inode 블록에 새로운 디렉토리 엔트리를 삽입할 때 오버플로우가 발생하면, 디렉토리 구조는 [그림 4]와 같이 B+ 트리를 이용한 디렉토리 구조로 전환된다.



[그림 4] B+ 트리를 이용한 디렉토리 구조

Inode 블록을 접근하여 인덱스 세트에 있는 키 값과 비교하여 올바른 디렉토리 엔트리 블록을 찾아간다. 찾고자 하는 디렉토리 엔트리를 inode 블록에서부터 링크를 따라서 찾을 수도 있으며, 순차 세트인 디렉토리 엔트리 블록들을 순차적으로 검색할 수도 있다. 한편, 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.

3.2.2. 디렉토리 연산

[그림 5]는 B+ 트리에서의 삽입 연산 함수이다.

```
알고리즘 bpt_insert
입력: 파일의 이름(name)
출력: 파일 저장
{ 파일의 이름을 이용하여 B+ 트리에 연결된 알맞은 디렉토리
  엔트리 블록을 찾는다;
  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 첫번째값)
        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 5] B+ 트리 디렉토리의 엔트리 삽입

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.

4. 성능 평가

본 절에서는 SAN 파일 시스템을 위해 구현된 확장 해싱을 이용한 디렉토리 구조와 B+ 트리를 이용한 디렉토리 구조의 성능을 평가한다. 성능 실험을 위해 Redhat 6.2 버전의 LINUX를 설치하고, 커널을 6.2 버전의 기본 커널인 2.2.16을 2.2.18로 업그레이드한 환경에서 구현언어는 GCC gcs-2.91.66버전을 이용하였

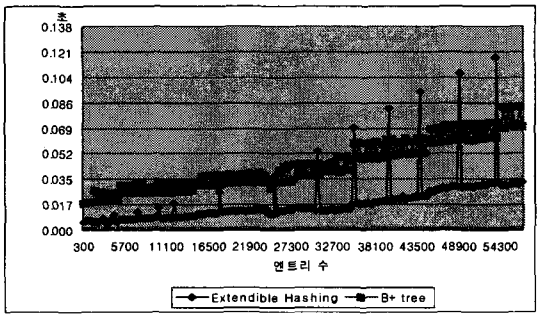
다. 실험에 이용된 디스크는 Seagate사의 ST51080A 모델[17]이며, 성능 매개변수는 [표 1]과 같다.

[표 1] 성능 매개변수와 값

매개변수	값
평균 탐색 시간	10.5 ms
평균 회전 지연 시간	5.58 ms
데이터 전송률	67.7 Mbps

4.1 디렉토리 엔트리 삽입

확장 해싱을 이용한 디렉토리 구조에서와 B+ 트리를 이용한 디렉토리 구조에서의 디렉토리 엔트리를 삽입하는 데 소요되는 평균 수행 시간의 성능 평가 결과는 [그림 6]과 같다.



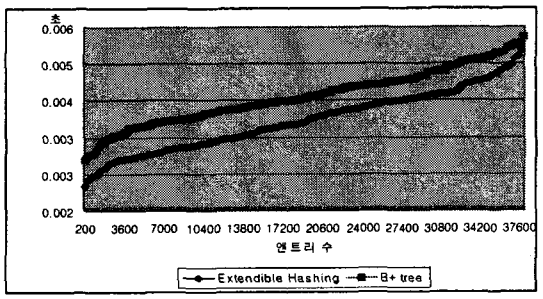
[그림 6] 디렉토리 엔트리 평균 삽입 시간

확장 해싱 디렉토리 그래프에서 갑자기 증가하는 부분은 해싱 테이블이 2배로 증가할 때를 나타내며, 이는 B+ 트리 디렉토리 그래프에서 오버플로우로 인하여 요구되는 시간보다 많은 시간이 소요된다. 그림에서 보는 바와 같이 평소에는 디렉토리 엔트리 삽입에는 확장 해싱을 이용한 디렉토리 구조의 성능이 좋으며, 오버플로우가 발생했을 경우에는 B+ 트리를 이용한 디렉토리 구조의 성능이 좋다.

4.2 디렉토리 엔트리 탐색

확장 해싱을 이용한 디렉토리 구조에서와 B+ 트리를 이용한 디렉토리 구조에서의 디렉토리 엔트리 평균 탐색 시간은 [그림 7]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 접근하는 데 비해, B+ 트리는 데이터 엔트리가 존재하는 리프 노드까지 가기 위해서 중간 노드에서 파일을 순차적으로 비교하기 때문이다.



[그림 7] 디렉토리 엔트리 평균 탐색 시간

5. 결론

본 논문에서는 대규모 SAN 파일 시스템에서 실행 환경에 적합한 방법을 선택할 수 있도록 확장 해싱 디렉토리와 B+ 트리 디렉토리 구조를 설계 및 구현하였다. 이들 디렉토리 구조는 디렉토리 엔트리의 수를 고려하여 적은 양의 디렉토리 엔트리들에 대해서 inode 블록에 직접 저장함으로써 한번의 접근으로 원하는 엔트리 정보를 검색할 수 있게 하고, 많은 양의 디렉토리 엔트리들에 대해서 확장 해싱을 이용하여 빠른 검색을 가능하게 하였으며, B+ 트리를 이용하여 엔트리들의 빠른 정렬을 가능하게 하였다.

성능평가를 통하여 엔트리를 삽입하는데 수행되는 시간은 확장 해싱을 사용한 디렉토리 구조에서 적게 소요되었으며, 엔트리를 탐색하는 시간 역시 확장 해싱을 이용한 디렉토리 구조에서 적게 소요되었음을 알 수 있다.

향후에는 선형 해싱(Linear Hashing)을 이용한 디렉토리 구조 설계 및 구현에 관해서 연구하며, 이를 포함한 확장해싱과 B+ 트리와의 성능평가가 요구된다.

참 고 문 헌

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

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

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

[4] 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.

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

[6] 이용규, 김신우, 손덕주, "SAN 환경 공유 디스크 파일 시스템의 메타데이터 관리," 정보과학회지 19권 3호, pp. 33-42, 2001. 3.

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