SAN 기반 공유 파일 시스템을 위한 디렉토리 구조 설계

김신우*, 이용규*, 김경배**, 신범주**
*동국대학교 컴퓨터공학과
**한국전자통신연구원 컴퓨터시스템연구부 시스템소프트웨어연구팀

Designing Directory Structure for a SAN-Based Shared File System

Shin Woo Kim*, Yong Kyu Lee*, Gyoung Bae Kim**, Bum Joo Shin**
*Dept. of Computer Engineering, Dongguk University
**Internet Service Department, ETRI

요 약

최근 개발되고 있는 SAN 기반 리눅스 클러스터 파일 시스템들은 중앙에 파일 서버 없이 디스크를 공유하는 클라이언트들이 화이버 채널을 통하여 마치 파일 서버처럼 디스크에 자유롭게 접근할 수 있으므로, 유용성, 부하의 균형, 확장성 등에서 장점을 가진다. 본 논문에서는 ETRI에서 개발중인 SAN 기반 리눅스 클러스터 파일 시스템인 SANtopia를 위해 설계된 새로운 inode의 구조와 이 inode의 구조를 기반으로 확장 해싱(Extendible Hashing)을 이용한 새로운 디렉토리 구조의 설계에 대하여 기술하고, 성능 평가를 통하여 제안된 방법의 우수성을 보인다.

1. 서론

멀티미디어 데이터의 크기가 커지고 시스템에서 다루는 파일들의 수가 많아짐에 따라, 파일 시스템에 대용량의 데이터를 저장하는 것이 필요하다. 그러나, 기존의 단일 대형 시스템으로는 비용 대비 성능면에서 우수한 성과를 거두기 어려움에 따라, 저렴한 비용으로 작은 규모의 컴퓨터들을 클러스터로 연결하여 하나의 통합된 시스템을 구축하려는 연구가 활발히 진행중이며, 이들 파일 시스템들이 대용량의 데이터를 저장하기 위해 사용하는 저장장치로는 SAN을 들 수 있다. SAN은 기존의 네트워크 외에 별도로 고속의 데이터 전용 네트워크인 화이버 채널[3]을 통해 클라이언트들과 저장 장치들을 상호 밀접히 연결한다.

SAN 파일 시스템의 대표적인 예로 미네소타 대학에서 구현된 GFS(Global File System) [4]를 들 수 있으며, 국내에서도 한국전자통신연구원에서 SANtopia [6]를 개발하고 있다.

이와 같은 SAN을 이용한 새로운 파일 시스템들은 공통적으로 서버가 존재하지 않는 공유 디스크 파일 시스템들로, 별도의 서버 없이 각각의 분산된 클라이언트가 메타데이터를 직접 관리하면서 저장 장치들에 접근한다. 따라서, 각각의 클라이언트는 독립적으로 저장 장치들에게 데이터를 요구할 수 있으며, 하나의 서버에 업무가 집중되는 현상이 없이 어떤 클라이언트에 문제가 발생하더라도 나머지 시스템에 거의 영향을 주지 않는다는 장점이 있다.

본 논문에서는 최근에 주목을 받고 있는 SAN 파일 시스템에서의 디렉토리 구조에 대하여 살펴본다. 먼저

본 연구는 한국전자통신연구원의 2001년도 위탁과제 "SAN 기반 대규모 공유파일 시스템을 위한 디렉토리 관리 기법 연구"의 일부로 수행됨.

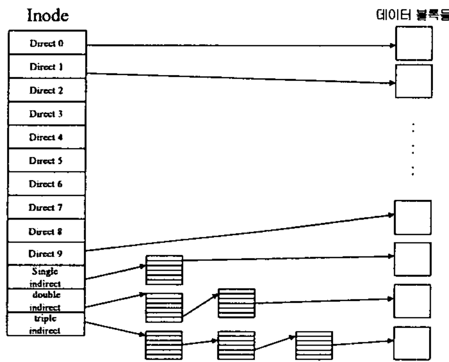
이들 중 가장 대표적인 GFS의 특징과 장단점에 대하여 살펴보고, 이의 단점을 개선하기 위해 SANtopia에서 사용되는 새로운 디렉토리 구조를 설명한다.

2. GFS의 디렉토리 구조

GFS[4]는 기존의 UNIX 파일 시스템을 개선하여 SAN 환경에서의 파일 시스템으로 사용하도록 미네소타 대학에서 최근에 개발되었다. GFS는 많은 대용량의 하일들을 저장해야하므로 디렉토리 구조에서 기존의 파일 시스템과는 매우 다르다. 본 절에서는 GFS의 디렉토리 구조에 대해서 알아본다.

2.1 Inode의 구조

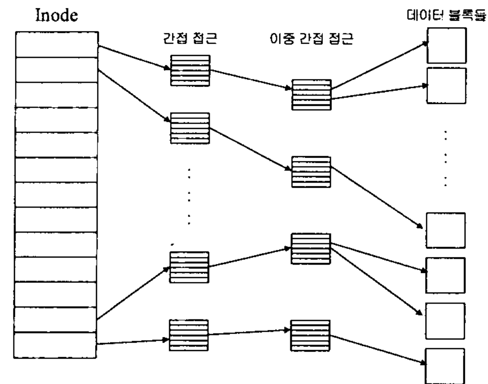
GFS는 UNIX의 단점을 보완한 시스템이다. 먼저 기존의 Unix 파일 시스템을 살펴본다..



[그림 1] UNIX inode의 구조[1]

[그림 1]에서와 같이 UNIX 시스템 V[1]는 inode를 이용하여 디스크의 데이터 블록에 접근하는데 하나의 inode는 13개의 엔트리로 구성되어 있다. 하나의 inode의 처음 10개의 엔트리는 inode에서 직접 접근하기 위한 데이터 블록들의 주소를 저장하고 있고, 11번째 엔트리부터는 데이터 블록들에 간접 접근, 이중 간접 접근, 삼중 간접 접근을 이용하도록 하고 있다. UNIX inode의 구조에는 다음과 같은 단점이 있는데, 그 중 하나는, 파일의 최대 크기가 제한된다는 점이며, 또 다른 하나는 처음 10개의 데이터 블록들은 빠른 접근이 가능하지만, 나머지 블록들은 여러 단계를 거치는 간접 접근을 하기 때문에 접근 시간이 많이 요구된다는 점이다.

GFS에서는 inode의 구조가 [그림 2]와 같은 독특한 플랫 구조를 가진다. 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 크기가 무제한으로 커질 수 있는 장점을 갖는다.



[그림 2] GFS inode의 플랫 구조

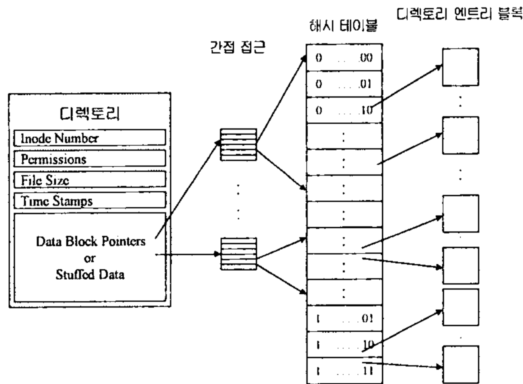
그러나, 이와 같은 장점들에도 불구하고, GFS에서는 항상 플랫 구조를 유지하여야 하므로 파일의 크기가 커질수록 데이터 블록의 평균 임의 접근 시간이 길어지는 결과를 초래한다. 예를 들어, 어떤 파일 A의 inode 트리가 정(full) *k-ary*일 경우, 즉, A의 크기가 이 트리의 높이 h로 수용할 수 있는 최대의 크기일 경우에, 이보다 조금이라도 큰 파일 B는 플랫 구조를 유지하기 위하여 트리의 높이가 $h+1$이 되어야 한다. 그러므로 이 새로운 파일 B는 A보다 데이터 블록을 임의 접근하는 데에 한번씩의 간접 접근이 추가로 필요하게 된다.

2.2 디렉토리의 구조

대부분의 UNIX 시스템에서는 디렉토리 내의 파일 이름들을 파일의 생성 순서로 유지하므로 특정 파일의 이름을 디렉토리 내에서 탐색할 때 순차적으로 검색하여야만 한다. 따라서, 많은 파일 이름들을 포함한 큰 디렉토리인 경우에는 특정 파일을 검색하는 데에 많은 시간이 요구된다. 이와 같은 UNIX 디렉토리에서의 비효율적인 순차적 검색을 극복하기 위해서 GFS는 확장 해싱(Extendible Hashing)[2]을 사용한다. 확장 해싱은 디렉토리의 파일의 수가 많고 적음에 상관없이 수용 가능하고, 비록 많은 수의 파일이 존재하더라도 해싱의 특성상 빠른 검색이 가능하게 한다.

GFS에서 디렉토리의 크기가 작을 경우에는 디렉토리의 inode 블록에 디렉토리 엔트리들을 저장하고, 파일의 수가 많아져서 디렉토리의 inode 블록에 저장할

공간이 없게 되면 [그림 3]과 같이 확장해싱을 이용하여 디렉토리 엔트리들을 저장한다. 이때, 디렉토리 구조도 크기가 커지면, inode 구조와 마찬가지로 플랫 구조를 이루며 레벨을 증가시킨다.



[그림 3] GFS의 확장 해싱을 이용한 디렉토리 구조

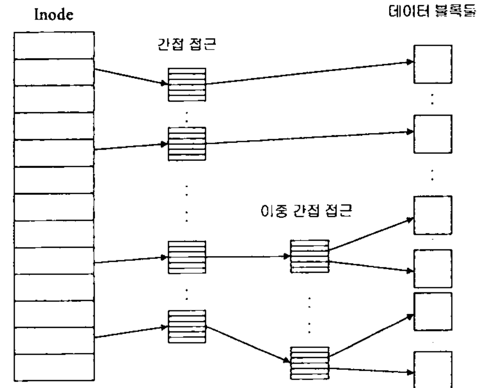
GFS의 확장 해싱에서는 CRC(Cycle Redundancy Check)[5] 라는 네트웍 상에서 시리얼 전송할 때 데이터의 신뢰성을 검증하기 위한 에러 검출 방법으로 사용되는 함수를 해시 함수로 이용한다. 확장 해싱에 대해서는 다음절에서 자세히 설명하기로 한다.

3. 새로운 디렉토리 구조

본 절에서는 GFS의 디렉토리 구조의 문제점을 개선하기 위한 새로운 디렉토리 구조를 설명한다.

3.1 inode의 구조

[그림 4]는 새로운 세미플랫(Semiflat) 구조로, 모든 데이터 블록들이 동일한 레벨에 있지 않고 트리의 높이 h와 $(h-1)$에 걸쳐 있음을 보여 준다. 세미플랫 구조에서는 모든 데이터 블록들이 파일의 크기에 따라 GFS에서처럼 플랫 구조를 가질 수도 있고 [그림 4]처럼 두 레벨에 걸쳐 있는 세미플랫 구조를 가질 수도 있다. 세미플랫 구조에서는 새로운 데이터 블록이 추가되었을 때, 플랫 구조에서처럼 트리 높이의 증가로 인하여 현재 레벨에 위치한 데이터 블록들을 다음 레벨로 전부 이동할 필요 없이 일부분만 이동하기 때문에 데이터 블록의 임의 접근에서 더 좋은 성능을 나타낸다.



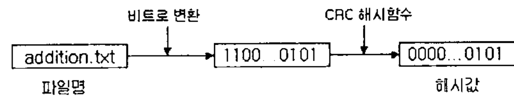
[그림 4] inode의 세미플랫 구조

3.2 디렉토리의 구조

디렉토리 구조에서는 확장 해싱이 사용되며 해시 테이블은 순차 파일로 구성된다.

가. 해시 함수

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

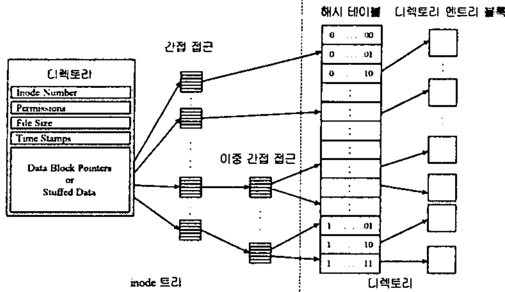


[그림 5] 해싱함수 적용하는 과정

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

나. 확장 해싱

[그림 6]은 디렉토리 구조를 보여주고 있다. 점선의 왼쪽은 inode 트리를 나타내고, 오른쪽은 해시 테이블과 디렉토리 엔트리 블록들을 나타낸다. 이때, 디렉토리 구조의 크기가 커지면 inode와 마찬가지로 세미플랫 구조를 이루며 레벨을 증가시킨다.



[그림 6] 디렉토리 구조

이와 같은 해시 디렉토리는 파일의 개수가 많을 때 활용되며, 적을 때는 디렉토리 엔트리들이 inode에 함께 저장된다.

초기의 디렉토리 구조는 inode 블록에 디렉토리 엔트리들을 저장한다. 그러나 이 블록(stuffed block)이 다 채워지면, 디렉토리 구조는 [그림 7]과 같이 make_exhash 함수를 사용하여 확장 해싱을 이용하기 위한 구조로 전환된다.

```
알고리즘 make_exhash
입력 : stuffed block의 inode 번호
출력 : 확장 해싱을 위한 디렉토리 구조
{
  할당할 해시테이블을 계산한다.;
  해시테이블을 생성한다.;

  현재 inode에 있는 디렉토리 엔트리들을 exhash_
  insert 함수를 이용하여 해시테이블에 연결된 알맞
  은 디렉토리 엔트리 블록으로 옮긴다.;

  현재 inode 블록에는 해시테이블의 주소를 저장한
  다.;
}
```

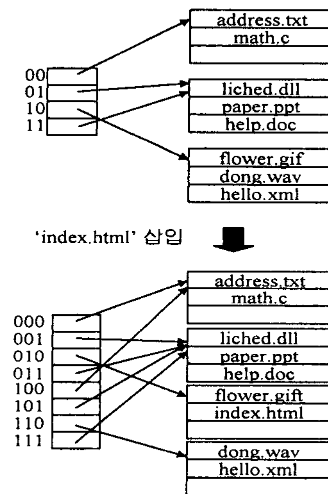
[그림 7] 해시구조 생성

변화된 디렉토리 구조로 새로운 파일을 저장하고자 할 때는 [그림 8]에서와 같이 exhash_insert 함수를 이용한다. exhash_insert 함수는 파일의 이름을 키값으로 해싱함수를 사용하여 해쉬값을 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 디렉토리를 2배로 커지게 하고, 그렇지 않을 경우에는 새로운 디렉토리 엔트리 블록을 추가하여 포인터를 수정해 준다.

```
알고리즘 exhash_insert
입력 : 파일의 이름(name)
출력 : 파일 저장
{
 파일의 이름을 이용하여 해쉬값을 계산한다;
 해쉬값을 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 8] 확장 해싱

확장 해싱에서 해시 테이블은 엔트리의 수에 따라 크기가 동적으로 확장 또는 축소된다. 다음 [그림 9]는 파일 'index.html'이 삽입될 때, '10' 버켓에서의 오버플로우로 인하여 해시 테이블의 크기가 두배로 확장되는 것을 보여준다.



[그림 9] 확장 해시 테이블

4. 성능 분석

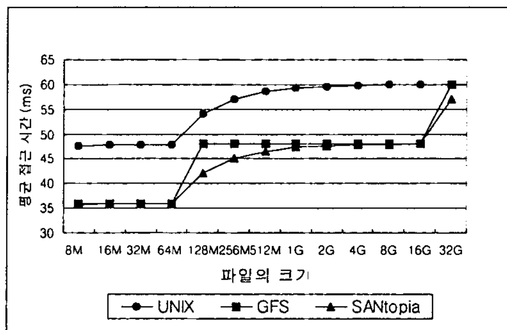
본 절에서는 앞에서 제안한 디렉토리 구조의 성능을 평가한다. 시뮬레이션에 이용된 디스크는 IBM사의 DPSS-336950 모델[7]이며, 성능 매개변수는 [표 1]과 같다.

[표 1] 성능 매개변수와 값

매개변수	값
평균 탐색 시간	6.8 ms
평균 회전 지연 시간	4.17 ms
데이터 전송률	300 Mbps
디스크 블록 크기	1 Kbytes

4.1 Inode의 구조

기존의 UNIX, GFS, 그리고 SANtopia의 inode 구조에서 파일의 크기에 따른 임의의 데이터 블록에 대한 평균 접근 시간을 비교한다. [그림 10]에서 SANtopia의 성능이 가장 좋음을 알 수 있다.

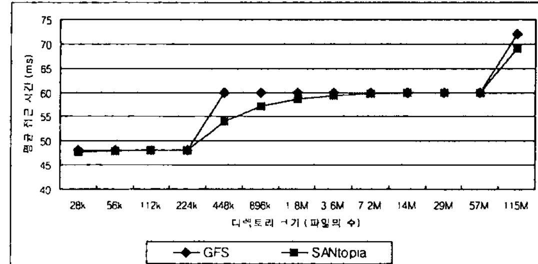


[그림 10] 임의의 블록 접근 시간

파일 크기가 증가함에 따라, SANtopia가 GFS보다 천천히 완만하게 증가하고 있음을 그래프를 통해서 확인할 수 있다. 이것은 임의의 데이터 블록 접근 시간 면에서 세미 플랫 구조가 플랫 구조보다 우수하다는 것을 보여준다.

4.2 디렉토리의 구조

GFS와 SANtopia에서의 디렉토리 엔트리 블록들의 임의의 접근 시간들을 비교한다. [그림 11]은 디렉토리 구조에서의 SANtopia의 성능이 GFS의 성능보다 좋음을 보여준다. 그림에서의 평균 접근 시간은 디렉토리 inode로부터 시작하여 하나의 디렉토리 엔트리 블록을 임의로 접근하는데 필요한 평균 시간을 말한다.



[그림 11] 디렉토리 엔트리의 접근 시간

5. 결론

본 논문에서는 SAN 환경에서 공유 디스크 파일 시스템으로 사용하기 위하여 미네소타 대학에서 최근에 개발된 GFS의 특징에 대해서 살펴보았다. GFS는 여러 가지의 장점들에도 불구하고 플랫 inode의 구조와 디렉토리 구조 등에서 개선할 점이 있었다.

본 논문에서는 이러한 문제점들을 개선하기 위해 한국전자통신연구원에서 개발중인 SANtopia를 위해 새롭게 설계된 inode 구조 및 디렉토리 구조에서 세미플랫 구조를 이용함으로써, GFS의 플랫 구조에서 파일의 크기가 커짐에 따라 급격히 증가하던 데이터 블록의 임의 접근시간을 단축하도록 하였다. 향후에는 제안된 방법의 완전한 구현과 함께 충분한 성능 실험이 필요하다.

[참고문헌]

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

[2] 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.

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

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

[7] IBM DPSS-336950 Hard-Disk Spec., http://www.storage.ibm.com/hardsoft/diskdrdl/ ultra/ul36lp.htm#Prodspecs.