SAN 환경 공유 디스크 파일 시스템의 메타데이터 관리

동국대학교 이용규* · 김신우**
한국전자통신연구원 손덕주

1. 서 론

멀티미디어 데이터의 크기가 커지고 시스템에서 다루는 파일들의 수가 많아짐에 따라, 파일 시스템에 대량의 데이터를 저장하는 것이 필요하다. 그러나, 기존의 단일 대형 시스템으로는 비용 대비 성능 면에서 우수한 성과를 거두기 어려움에 따라, 저렴한 비용으로 작은 규모의 컴퓨터들을 클러스터(Cluster)로 연결하여 하나의 통합된 시스템을 구축하려는 노력이 시도되었다. 그 결과 xFS[1,12], Frangipani[6,10] 등의 클러스터 파일 시스템들이 구현되었고, 이들은 네트워크 연결형 파일 시스템으로 기존의 단일 시스템에 비하여 시스템 구축 비용뿐만 아니라, 시스템의 유용성, 확장성, 고장 감내의 측면 등에서 장점을 가지고 있다.

이들 파일 시스템들이 대용량의 데이터를 저장하기 위해 사용하는 저장장치로는 NAS(Network Attached Storage)와 SAN(Storage Area Network)을 들 수 있다. NAS는 네트워크에 부착된 저장장치로 이더넷(Ethernet)이나 TCP/IP와 같은 전통적인 LAN 프로토콜을 사용하며, NAS 서버는 파일의 입출력을 처리한다. 한편, 최근에 주목을 받고 있는 SAN은 기존의 네트워크 외에 별도로 고속의 데이터 전용 네트워크인 Fibre Channel[5]을 통해 클라이언트들과 저장 장치들을 상호 밀접히 연결한다.

최근에 자료저장 시스템의 시장규모가 확대되면서 업체들이 앞다투어 SAN 관련 솔루션들을 제공하기 시작하고 있다. 예를 들면, IBM사의 Tivoli [18]와 Compaq사의 VersaStor[19], 그리고 Veritas 사의 SANPoint[15] 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 대학에서 구현된 GFS(Global File System)[7,8]를 들 수 있으며, 국내에서도 한국전자통신연구원에서 SANtopia [13,14,16]를 개발하고 있으며 SANUX사에서도 SANUX[17]를 개발 중에 있다.

이와 같은 SAN을 이용한 새로운 파일 시스템들은 공통적으로 서버가 존재하지 않는 공유 디스크 파일 시스템들로, 별도의 서버를 두지 않고 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 각광을 받고 있는 Linux를 운영체제로 사용하는 SAN 파일 시스템에서의 메타데이터 관리 방법에 대하여 살펴본다. 먼저 이들 중 가장 대표적인 GFS의 특징과 장단점에 대하여 살펴보고, 이의 단점을 개선하기 위해 SANtopia에서 사용되는 새로운 메타데이터 관리 방안을 설명한다. 특히, 메타데이터의 여러 구성요소들 중 디렉토리 및 inode의 구조, 데이터 블록 할당 방

* 종신회원
** 학생회원

법, 빈 공간 관리 방법, 그리고 메타데이터의 저널링에 중점을 두기로 한다.

2. GFS에서의 메타데이터 관리

GFS[7,8]는 기존의 UNIX 파일 시스템을 개선하여 SAN 환경에서 Linux의 클러스터 파일 시스템으로 사용하도록 미네소타 대학에서 최근에 개발되었다. 비디오나 오디오 등의 멀티미디어 데이터를 지원하는 것을 목적으로 한 GFS는 대용량의 많은 파일들을 저장해야 하기 때문에, 메타데이터를 관리하는 면에서 기존의 파일 시스템들과는 매우 다르다. 본 절에서는 GFS의 메타데이터 구조와 저널링 메카니즘에 대해서 알아본다.

2.1 디렉토리 및 inode의 구조

GFS는 UNIX의 단점을 보완한 시스템으로, inode는 독특한 플랫 구조(Flat Structure)를 이루고 있다. 우선 UNIX의 inode 구조를 알아보고, GFS의 inode 구조를 이와 비교하여 설명한다.

그림 1에서와 같이, UNIX 시스템 V[2]는 inode를 이용하여 디스크의 데이터 블록에 접근하는데 하나의 inode는 13개의 엔트리로 구성되어 있다. 한 inode의 처음 10개의 엔트리는 inode에서 직접 접근하기 위한 데이터 블록들의 주소를 저장하고 있고, 11번째 엔트리부터는 그림 1에서 보는 것처럼 데이터 블록들에 간접 접근, 이중 간접 접근, 삼중 간접 접근을 하도록 한다.

UNIX inode의 구조에는 몇 가지 단점이 있다.

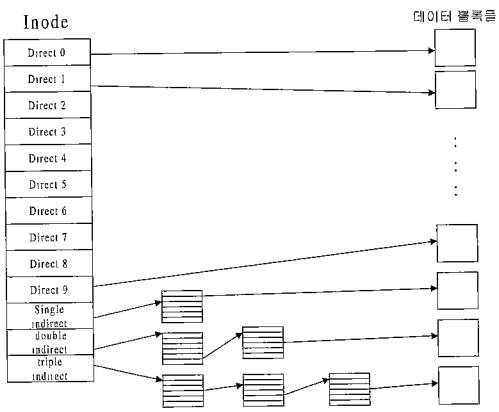


그림 1 UNIX inode의 구조 [2]

그 중 하나는, 파일의 최대 크기가 제한된다는 점이다. 예를 들어, 한 블록의 크기를 1 KBytes라고 가정할 때, 수용할 수 있는 최대 파일의 크기가 약 16 GBytes로 제한된다. 또 다른 하나는, 큰 파일인 경우 처음의 10 개의 데이터 블록들은 직접 접근이 가능하므로 빠른 접근이 가능하지만, 파일의 나머지 블록들은 여러 단계를 거쳐 간접 접근을 하기 때문에 접근 시간이 많이 걸리게 된다는 점이다. 즉, 파일의 앞부분은 데이터 블록의 빠른 임의 접근이 가능하나, 파일의 뒷부분으로 갈수록 데이터 블록의 임의 접근 시간이 더 걸리게 된다.

GFS에서는 inode의 구조가 UNIX와는 달리 그림 2와 같은 독특한 플랫 구조를 가진다. 그림에서 보는 바와 같이 파일의 모든 데이터 블록들은 트리의 높이(height)와 같은 리프(leaf) 레벨에 위치한다. 즉, 파일의 크기에 따라 트리의 높이가 정해지며, 모든 데이터 블록들은 예외 없이 동일한 리프 레벨에 위치하게 된다.

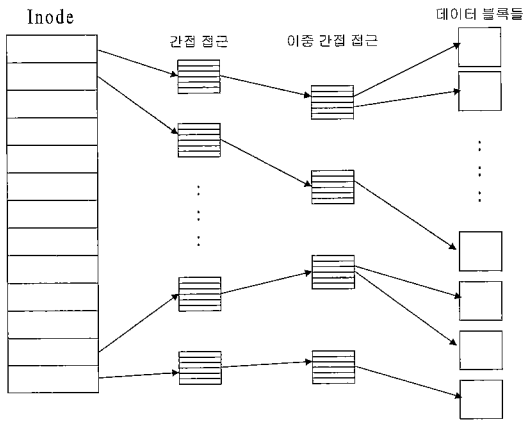


그림 2 GFS inode의 플랫 구조

이는 UNIX에서 데이터 블록의 파일내의 위치, 즉, 앞부분과 뒷부분 어느 곳에 위치한 블록인지에 따라 임의 접근 시간이 달라지는 것을 방지하여, GFS에서는 모든 데이터 블록의 임의 접근 시간이 같아지도록 하며, inode 트리의 구조를 단순화하기 위함이다. 또한 매우 큰 파일의 경우는 inode 트리의 높이를 증가시키면 되므로 파일의 크기에 제한이 없어지는 장점을 갖는다.

그러나, 이와 같은 장점들에도 불구하고, GFS에서는 항상 플랫 구조를 유지하여야 하므로 파일의

크기가 커질수록 데이터 블록의 평균 접근 시간이 길어지는 결과를 초래한다. 예를 들어, 어떤 파일 A의 inode 트리가 정(full) k-ary일 경우, 즉, A의 크기가 이 트리의 높이 h로 수용할 수 있는 최대의 크기일 경우에, 이보다 조금(1 byte)이라도 큰 파일 B는 플랫 구조를 유지하기 위하여 트리의 높이가 h+1이 되어야 하므로 이 새로운 파일 B는 A보다 데이터 블록을 임의 접근하는 데에 한번씩의 간접 접근이 추가로 필요해진다. 따라서 파일 A와 B는 파일 크기에서의 약간의 차이에도 불구하고, B가 A보다 데이터 블록의 임의 접근 시간이 길어지게 된다.

또한, GFS에서는 UNIX의 디렉토리 구조를 개선하였다. 대부분의 UNIX 시스템에서는 디렉토리 내의 파일 이름들을 특별한 순서 없이 파일의 생성 순서로 유지하므로 특정 파일의 이름을 디렉토리 내에서 탐색할 때 순차적으로 검색하여야만 한다. 따라서, 많은 파일 이름들을 포함한 디렉토리인 경우에는 특정 파일을 검색하는 데에 많은 시간이 걸리게 된다. 이와 같은 UNIX 디렉토리에서의 비효율적인 순차적 검색을 극복하기 위해서 GFS는 확장 해싱(Extendible Hashing)[4]을 사용하는데, 이는 디렉토리가 적은 수의 파일들로부터 많은 수의 파일들까지 자유롭게 수용할 수 있도록 하고, 또한 파일들이 많을 경우에도 해싱의 특성상 빠른 검색이 가능하게 한다.

2.2 데이터 블록의 크기

GFS에서도 다른 파일 시스템들과 마찬가지로 파일 시스템 생성 시에 데이터 블록의 크기를 정한다. 예를 들어 데이터 블록의 크기를 8 KBytes로 정하면, 시스템 내의 모든 파일들이 블록의 크기로 이를 사용하여야 한다. 이는 파일 시스템에서 공간 관리가 간편해지는 장점이 있다.

그러나, 실제 통합 환경에서는 크고 작은 많은 파일들이 파일 시스템 내에 공존한다[9]. 만약 블록의 크기를 크게 정한다면, 파일의 마지막 블록에 매우 큰 내부 단편화(Internal Fragmentation)가 발생하게 되며, 평균적으로 내부 단편화로 인해 낭비되는 공간은 파일 당 1/2 블록이 된다. 예를 들어, 한 블록의 크기를 64 KBytes라고 했을 때, 평균적으로 시스템 내의 파일마다 32 KBytes가 낭비된다. 반면에, 블록의 크기를 작게 정한다면, 내부 단편화로 낭비되는 공간은 줄일 수 있지만, 큰 파일인 경우에는 매우 많은 블록들이 필요하게 된다. 예를 들어, 1 GBytes 크기의 파일은 1 KBytes 블록들이 약 1,000,000 개가 필요하므로, 이 파일의 관리를 위해 매우 큰 inode 공간이 필요하게 된다.

2.3 빈 공간 관리

GFS는 다른 시스템들과 별 차이 없이 파일들에게 데이터를 블록 단위로 할당하거나 회수하며, 이를 위해 빈 공간을 그림 3과 같은 비트맵을 이용하여 관리한다. 비트맵에서는 디스크의 데이터 블록이 사용중인지 아닌지를 하나의 비트로 표시한다.

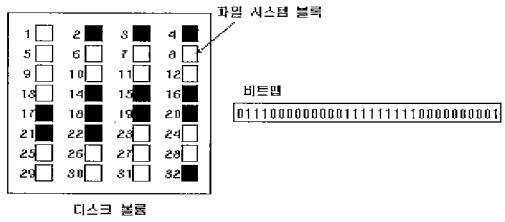


그림 3 빈 공간 관리

그러나, 파일 시스템의 크기가 매우 클 경우에는 이에 더불어 매우 큰 비트맵 공간을 필요로 한다. 예를 들어, 전체 디스크의 크기가 1 Peta Bytes이고 한 블록의 크기가 1 KBytes라고 가정한다면, 이 시스템은 2^{40}개의 블록들이 존재하므로 1 Tbits, 즉, 128 GBytes의 비트맵 공간을 필요로 한다.

이처럼 파일 시스템의 크기가 커짐에 따라 많은 데이터 블록들과 그에 따른 큰 크기의 비트맵을 요구하게 된다. 따라서, 파일에 빈 공간을 할당하고자 할 때 비트맵으로부터 빈 공간을 찾는 데 오랜 시간이 걸리게 되며, 파일 시스템 내에 빈 공간이 부족한 경우에는 수 많은 비트맵 블록들을 찾아다녀야 하는 문제가 발생한다. GFS에서도 이러한 문제를 해결하기 위하여 비트맵 대신에 빈 블록의 시작 위치와 연속된 빈 블록의 개수의 쌍들을 유지함으로써 공간의 할당을 빨리 할 수 있도록 하는 방안을 강구하고 있다.

2.4 메타데이터의 저널링

GFS의 또 다른 특징은 메타데이터의 회복을 위해 데이터베이스 시스템에서 데이터의 회복을 위해

이용되고 있는 저널링(Journaling)을 사용하는데 있다. 파일 시스템에 문제가 발생될 때 UNIX에서는 FSCK(파일 시스템 체크 루틴)를 이용하여 비일관성의 원인을 검사하고 해결한다. 그러나, 이 FSCK를 수행하는 데는 파일 시스템의 크기에 비례하여 많은 시간이 필요하며, 회복 중에 시스템의 오프라인 상태를 요구하기 때문에 서비스가 중단되는 단점이 있다. 따라서, GFS와 같은 공유 디스크 파일 시스템에서는 회복시간을 줄이고 회복 중 시스템의 온라인 상태를 유지하기 위하여 저널링을 사용한다. 그러나 메타데이터 이외의 일반 데이터에는 저널링을 수행하지 않는다.

GFS는 메타데이터의 수정에 원자성(Atomicity)을 보장하기 위해서 데이터베이스의 트랜잭션 개념을 사용하며, 저널링을 위해 로그 우선(Write-Ahead) 프로토콜[3]을 사용한다. 로그 우선 프로토콜이란 그림 4와 같이 수정된 데이터를 디스크에 기록하기 전에 로그에 우선적으로 기록하는 것을 말한다. 이렇게 함으로써, 로그의 기록 후에 만약 시스템에 문제가 발생하더라도 로그를 활용하여 메타데이터를 다시 회복할 수 있다.

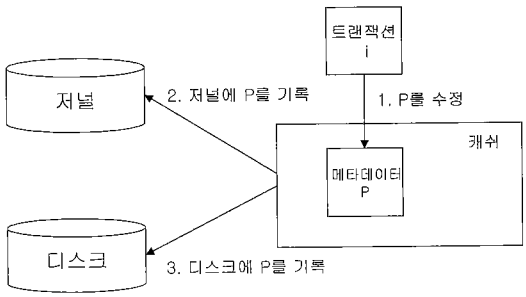


그림 4 로그 우선 프로토콜

GFS에서는 각각의 클라이언트가 자신의 저널 공간을 독자적으로 가지고 있으며, 이 공간은 락에 의해 다른 클라이언트들로부터 보호된다. 그러나, 어느 클라이언트에 장애가 있을 때에는 다른 클라이언트들이 이 클라이언트의 저널에 접근하여 회복할 수 있도록 한다.

GFS에서는 트랜잭션 관리자와 저널 관리자를 이용하여 저널링을 구현한다. 트랜잭션 관리자는 디스크에서 필요한 메타데이터를 사용하고자 할 때, 트랜잭션을 생성하고 디스크의 메타데이터에 락을 걸어 다른 클라이언트들이 접근하지 못하도록 한 후, 메타데이터를 수정할 수 있도록 하는 역할을 한다. 저널 관리자는 수정된 메타데이터를 디스크 저널에 기록한 후 이것이 다시 원래의 디스크에 기록되도록 하는 역할을 담당한다. 이와 같이 디스크 저널에 기록이 끝난 후에 다른 클라이언트가 동일한 메타데이터를 사용하려고 하여도 디스크에 기록이 끝나기 전까지는 이를 사용할 수 없으며, 다른 클라이언트는 디스크 기록 후에 반드시 디스크로부터 다시 읽어서 사용하여야 한다. 이는 저널링을 단순화하려는 의도에서 구현되었지만, 메타데이터의 수정 후에 디스크에 두 번(저널과 원래의 디스크) 기록된 후 다른 클라이언트가 사용할 수 있으므로, 메타데이터의 수정에 오랜 시간이 걸리게 된다. 이러한 GFS의 저널링 메카니즘은 Frangipani [10], Ext2fs[11] 등과 같은 파일 시스템들의 저널링과 비슷하다.

이처럼 저널링은 디스크에 수정된 메타데이터를 기록하기 전에 저널에 우선 기록함으로써, 저널에 기록 후 시스템에 문제가 발생하여도 이 저널을 이용하여 빠른 회복을 수행할 수 있다. 물론 저널에 기록되지 않은 트랜잭션들은 무시하면 된다. 그러나, 동일한 블록을 두 클라이언트가 요구할 경우에도 한 클라이언트가 디스크 저널에 기록한 후 원래의 디스크에 기록한 다음에야 다른 클라이언트가 디스크에 접근하여 사용할 수 있기 때문에 불필요한 디스크 접근이 발생하는 단점이 있다. 이 경우 메타데이터를 디스크 저널에 기록 후 다른 클라이언트에게 전송하여 사용할 수 있도록 한다면, 최소 한번 이상의 디스크 접근을 생략할 수 있게 된다. 이에 대하여는 다음절에서 상세히 설명한다.

3. 메타데이터 관리 방법의 개선

앞에서 살펴본 바와 같이, GFS의 장점에도 불구하고 메타데이터를 관리하는 데에는 몇 가지 개선할 사항들이 있다. 본 절에서는 GFS의 메타데이터 관리 방안의 문제점을 개선하기 위해 SANtopia에서 사용되는 새로운 메타데이터 관리 방안을 설명한다.

3.1 디렉토리 및 Inode의 구조

GFS는 리프의 데이터 블록들이 모두 트리의 동일한 레벨에 존재하는 플랫 구조를 이용한다. 이것은 파일의 크기가 커지면서 기존의 모든 데이터 블

록들도 추가적인 간접 접근이 필요하게되는 결과를 초래한다.

그림 5는 SANtopia의 새로운 세미플랫 구조(Semiflat Structure)이다. 그림을 보면 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 5처럼 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 세미플랫 구조가 플랫 구조보다 데이터 블록의 임의 접근에서 더 좋은 성능을 나타낸다. 그러므로 접근 시간의 불필요한 낭비가 없게 되어 플랫 구조에 비해 임의의 데이터 블록의 접근시간을 단축할 수 있다.

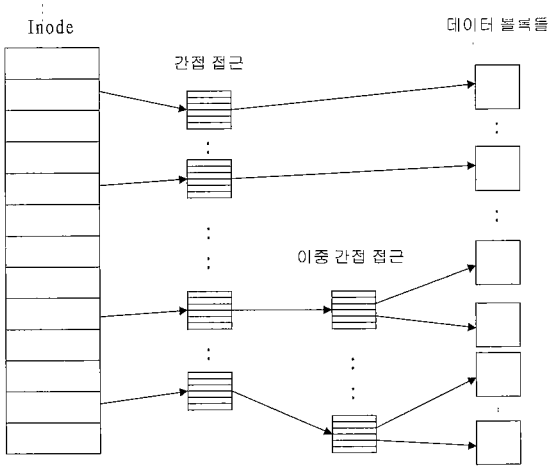


그림 5 inode 세미플랫 구조

SANtopia에서도 GFS에서와 같이 UNIX 디렉토리에서의 비효율적인 순차적 검색을 극복하기 위해서 확장 해싱을 사용한다. 이는 한 디렉토리가 많은 파일들을 수용할 수 있게 하고 파일의 빠른 검색을 가능하게 한다.

3.2 데이터 블록의 크기

GFS에서는 파일 시스템 생성 시에 데이터 블록의 크기를 정한다. 만약 한 블록의 크기를 크게 정한다면 파일의 마지막 블록에 큰 내부 단편화가 발생하게 되고, 만약 한 블록의 크기를 작게 정한다면 큰 파일인 경우 파일 당 많은 수의 블록들을 할당하기 위해 큰 inode 트리의 공간이 필요하게 되는 문제점이 있다.

이러한 문제를 해결하기 위해서, SANtopia에서는 파일의 크기가 클 경우에는 연속된 블록들을 익스텐트(Extent) 단위로 하여 할당하고, 파일의 크기가 작을 경우에는 일반적인 블록 단위로 할당하는 방법을 사용한다. 예를 들면, 큰 크기의 파일은 64 KBytes의 익스텐트 단위로 할당하고, 일반적인 보통의 크기를 가진 파일들은 1 KBytes 블록 단위로 할당할 수 있다. 파일의 크기가 1,000 KBytes라고 가정하면, 15 개의 익스텐트들과 40 개의 블록들을 파일에 할당하면 된다. 그러므로, 앞에서 언급한 내부 단편화 문제를 해결할 수 있다. 그러나, 빈 공간 관리를 위해 두 종류의 비트맵이나 빈 리스트들을 유지해야 하기 때문에 관리가 복잡해지게 된다. 하지만, 이것은 익스텐트 내부의 단편화된 부분을 작은 블록들로 사용할 수 있기 때문에 큰 문제가 되지 않는다. 물론, 익스텐트와 블록의 크기는 파일 시스템 생성 시에 정하도록 하며, 익스텐트를 연속된 블록들로 할당하는 이유는 디스크의 탐색시간을 줄여서 익스텐트의 접근 시간을 단축하기 위한 것이다.

3.3 빈 공간 관리

GFS는 빈 공간을 비트맵을 이용하여 관리한다. 이 방법은 큰 파일 시스템에서는 많은 디스크 블록들과 그에 따른 큰 크기의 비트맵을 요구하게 되어, 파일에 공간을 할당하고자 할 때 빈 공간을 찾는데 오랜 시간이 걸리는 문제점이 발생한다. 그러므로, 이러한 문제의 해결책으로 SANtopia에서는 독창적으로 디스크의 특정 영역에 존재하는 빈 공간 주소들의 모임인 빈 공간 지갑(Free Space Purse)을 사용하여 빈 공간을 관리한다.

지갑은 현재 할당 가능한 블록들의 디스크 시작 주소와 연속된 개수의 쌍들로 구성되며 대부분의 할당과 회수가 지갑에서 가능하도록 충분한 크기를 갖는다. 만일, 지갑에 있는 블록들이 전부 할당된 후에는 비트맵에 있는 빈 블록들을 찾아 파일의 나머지 부분을 할당한다. 빈 공간 지갑에서 할당과 회수가 되풀이되어 어느 순간에 빈 블록의 수가 미리 정해진 한도 이하가 되면 비트맵의 빈 블록들을

가져와 지갑을 채운다. 이 때, 비트맵에는 할당된 것으로 표시한다.

지갑은 메모리 캐쉬와는 성격이 다르다. 캐쉬에 있는 블록들은 디스크 블록들을 복사하여 저장하고 있으므로 데이터 일관성의 문제가 따르는데 반해, 지갑에 있는 빈 공간은 비트맵에 미리 사용 중인 것으로 표시하여, 파일에 빈 공간을 할당할 때 일일이 빈 공간을 찾는 시간과 비트맵에 표기하는 시간을 줄이도록 하는데 목적이 있다. 이는 마치 우리가 은행 예금 중 일부를 인출하여 지갑에 넣고 사용하는 방식과 유사하다.

그림 6은 비트맵에서 빈 공간 지갑을 사용하는 예로 10,000 개의 빈 블록 주소들을 지갑이 가지고 있다고 가정한 것이다.

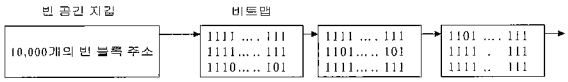


그림 6 빈 공간 지갑을 이용한 빈 공간 관리(비트맵)

비트맵을 사용하는 대신 빈 공간 연결 리스트를 이용하여 빈 블록들을 관리할 수도 있다. 그림 7은 연결 리스트에서 지갑을 사용하는 것을 보여준다.

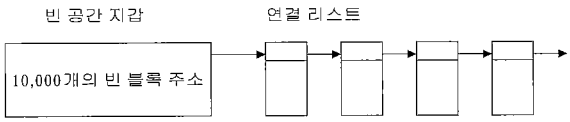


그림 7 빈 공간 지갑을 이용한 빈 공간 관리(연결 리스트)

익스텐트에 대해서도 앞에서 살펴본 빈 공간 지갑을 이용하여 빈 블록들을 관리하는 방식을 적용한다.

3.4 메타데이터의 저널링

GFS는 저널링을 사용하여 시스템에 문제가 발생했을 때, 빠른 회복을 수행한다. 그러나, 한 클라이언트가 디스크의 메타데이터를 수정하는 중에 다른 클라이언트가 동일한 블록을 사용하고자 할 때, 먼저 사용하고 있는 클라이언트가 메타데이터를 디스크 저널에 기록한 후 원래의 디스크에 기록한 다음에야, 다른 클라이언트가 디스크에 접근하여 사용할 수 있기 때문에 불필요한 디스크 접근이 발생하는 단점이 있다. 이러한 문제점을 해결하기 위해서 SANtopia에서는 두 클라이언트가 동일한 블록을 사용하고자 할 때, 한 클라이언트가 메타데이터를 디스크 저널에 기록하고 난 다음, 디스크에 기록하기 전에 곧바로 다른 클라이언트가 메타데이터를 사용할 수 있도록 함으로써 GFS에 비하여 최소한 한 번 이상의 디스크 접근을 생략하도록 한다. 또한, SAN 환경에서는 네트워크를 통한 전송시간이 디스크 접근 시간보다 빠르기 때문에, 전체적인 트랜잭션 처리 시간을 감소시킬 수 있다.

본 절에서는, 메타데이터가 디스크 저널에 기록된 후 다른 클라이언트가 바로 사용 가능하도록 하는 메타데이터 저널링과 회복 메카니즘에 대해서 설명한다.

3.4.1 저널링 레이아웃

SANtopia에서는 클라이언트마다 자신의 독립적인 저널 공간을 가지고 있으며, 저널 공간은 락에 의해 다른 클라이언트들로부터 보호된다. 그러나, 어느 클라이언트에 장애가 있을 시에는 다른 클라이언트들이 이 클라이언트의 저널에 접근하여 회복할 수 있도록 한다. 그림 8은 5개의 클라이언트를 가정한 레이아웃이다.

Lock 0	Super Block
Lock 1	Journal 0
Lock 2	Journal 1
Lock 3	Journal 2
Lock 4	Journal 3
Lock 5	Journal 4
Locks 6-1000	Resource Group 0
Locks 1001-2000	Resource Group 1

그림 8 저널링 레이아웃

3.4.2 저널링 자료구조

저널링을 구현하기 위해서는 디스크와 메모리에서 메타데이터를 관리할 각각의 자료구조가 요구된다. 디스크에 존재하는 저널을 관리하기 위한 자료구조는 그림 9와 같이 저널 헤더, 디스크립터 테이블, 메타데이터 블록으로 구성되어 있다.

저널 헤더는 저널의 위치 정보를 저장하는 것으

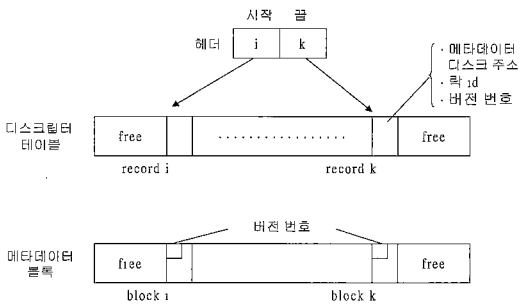
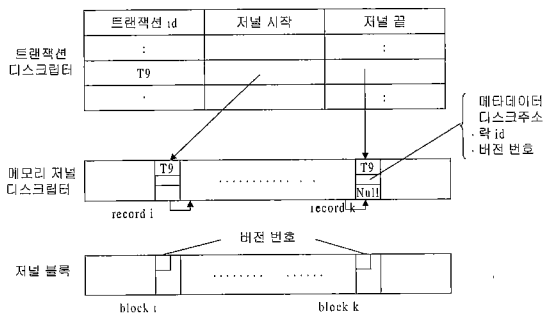


그림 9 디스크 저널 자료구조

로 저널의 시작과 끝을 기록하고 있다. 디스크립터 테이블은 메타데이터의 정보를 기록하기 위하여 메타데이터의 디스크 주소와 락 id, 그리고 버전 번호 등을 기록한다. 여기서, 버전 번호는 디스크에 기록된 메타데이터가 실패한 클라이언트에 의해 변경되었는지를 테스트하기 위해서 회복작업을 하는 동안에 사용된다. 그리고, 메타데이터 블록은 수정된 데이터를 저장하고 있다.

메모리에 있는 저널을 관리하기 위한 자료구조는 그림 10과 같이 트랜잭션 디스크립터, 메모리 저널 디스크립터, 저널 블록, 그리고 캐쉬 디스크립터로 구성되어 있다.



캐쉬 디스크립터

캐쉬 페이지 번호	디스크 페이지 번호	더티 비트	핀 마크	딜레이 카운터	클라이언트 id	저널 페이지 번호	락 id	Waiting Client Queue
.	:	:	:	:	:	:	:	:
21	P3	0	1	0	null	null	null	c3
22	P5	1	1	1	c0	1210	L5	c1 c2
23	P9	1	0	0	null	1320	null	null
:	:	:	:	:	.	.	.	.

그림 10 메모리 저널 자료구조

트랜잭션 디스크립터는 트랜잭션 정보를 가지는 것으로서 트랜잭션 id, 저널의 시작과 끝을 기록하고 있으며, 메모리 저널 디스크립터는 디스크의 디스크립터 테이블과 비슷한 역할을 하며 메타데이터의 디스크 주소와 락 id, 그리고 버전 번호를 기록하고 있다. 저널 블록은 실제의 데이터를 저장하고, 캐쉬 디스크립터는 로깅을 위해서 필요한 모든 정보 즉, 캐쉬 페이지 번호, 디스크 페이지 번호, 더티 비트(Dirty Bit), 핀 마크(Pin Mark), 딜레이 카운터(Delay Counter), 클라이언트 id, 저널 페이지 번호, 락 id, 그리고 대기 클라이언트 큐를 저장한다. 이중 더티 비트는 메모리에 로드된 후에 페이지가 변경된 것을 나타내고, 핀 마크는 클라이언트가 페이지를 사용 중이기 때문에 디스크에 기록하지 못하도록 금지하는 것이다.

딜레이 카운터는 다른 클라이언트에게 자신이 디스크 저널에 기록한 메타데이터를 보낸 후 그로부터 다시 저널에 기록되었다는 메시지를 받을 때까지 기다리는 시간을 측정하는 것이다. 만약 한 클라이언트가 다른 클라이언트에게 메타데이터를 보내고 그로부터 메타데이터의 로깅 메시지를 받는다면, 이 클라이언트는 수정된 메타데이터를 자신이 디스크에 기록하지 않아도 된다. 그것은 메타데이터가 다른 클라이언트에 의해 다시 수정되었으므로 그가 디스크에 최근의 버전을 기록하면 되기 때문이다. 이렇게 함으로써 불필요한 디스크 접근을 줄일 수 있게 된다. 그러나 일정 시간이 지나도 다른 클라이언트로부터 메타데이터의 로깅 메시지를 받지 못한다면, 무한정 기다리지 않고 저널에 기록된 메타데이터를 디스크에 기록하여 자신의 수정사항이 디스크에 반영되도록 한다.

대기 클라이언트 큐는 현재 이 메타데이터를 요구하고 사용이 끝나기를 기다리고 있는 클라이언트들을 표시한다.

3.4.3 저널링

SANtopia에서 저널링은 트랜잭션을 사용하여 파일 시스템 상태를 변화하며, 각각의 저널링 엔트리는 관련 있는 메타데이터의 락들을 가지고 시스템의 일관된 상태를 유지한다. 저널링에는 GFS에서처럼 두 개의 관리자가 사용되는데, 하나는 트랜잭션 관리자로 트랜잭션을 관리하고, 다른 하나는 저널 관리자로 실질적인 저널링을 한다. 트랜잭션 관리자는 다음과 같은 단계로 트랜잭션이 실행되도록 한다.

(1) 필요한 락을 획득

(2) 메모리의 메타데이터 버퍼들을 디스크에 기록되지 않도록 함(Pin)
(3) 메타데이터 수정

저널 관리자는 다음과 같은 단계로 메모리의 수정된 메타데이터를 디스크 저널에 기록한다

(1) 디스크 저널에 기록
(2) 다른 클라이언트로부터 메타데이터를 받았다면, 로깅 메시지를 보내줌
(3) 메타데이터를 기다리는 클라이언트가 있으면, 수정된 메타데이터와 락을 넘겨줌. 없으면, (6)을 수행
(4) 딜레이 카운터를 설정하고, 딜레이 시간 측정을 시작함
(5) 메타데이터를 받은 클라이언트로부터 로깅 메시지가 도착하면 메타데이터가 있는 메모리 버퍼를 해제하고 트랜잭션 종료. 일정 시간(딜레이 시간)이 지나도 아무런 메시지가 없으면 (6)을 수행
(6) 메타데이터의 디스크 기록 금지 해제(Unpin)
(7) 버퍼관리자에 의한 디스크 커미트(Commit) 수행

3.4.4 회 복

SANtopia에서는 시스템 실패 시 회복 관리자를 이용하여 시스템을 회복한다. 회복 관리자는 실패한 클라이언트의 id와 저널 락을 획득하면서 회복을 시작한다. 저널 회복 단계는 다음과 같다.

(1) 저널의 처음과 끝의 엔트리를 찾음
(2) 부분적으로 커미트된 엔트리들은 무시
(3) 각각의 저널 엔트리에 대해 회복 관리자는 그 엔트리와 관련 있는 모든 락들을 획득
(4) 디스크에 있는 버전 번호와 회복하려는 클라이언트 저널의 버전 번호를 비교하여 디스크의 버전이 저널의 직전 버전일 때는 회복을 수행하고, 그렇지 않으면 이전의 버전이 디스크에 커미트될 때까지 해당 저널의 회복을 늦춤 (메타데이터의 여러 버전들이 순차적으로 회복됨)

4. 결 론

본 논문에서는 SAN 환경에서 공유 디스크 파일 시스템으로 사용하기 위하여 미네소타 대학에서 최근에 개발된 GFS의 특징과 장단점에 대해서 살펴보았다. GFS는 여러 가지의 장점들에도 불구하고 inode의 플랫 구조, 디스크 블록의 할당 방법, 빈 공간의 관리 방법, 그리고 메타데이터의 저널링 등에서 몇 가지 개선할 점들이 있었다.

본 논문에서는 이러한 GFS의 문제점들을 개선하기 위해 한국전자통신연구원에서 개발중인 SANtopia에서 사용되는 새로운 메타데이터 관리 방안을 설명하였으며, 이를 다음과 같이 요약할 수 있다. 첫째, 세미플랫 구조를 이용함으로써 GFS의 플랫 구조에서 파일의 크기가 커짐에 따라 급격히 증가하던 데이터 블록의 임의 접근시간을 단축하도록 하였다. 둘째, 파일의 크기에 따라 연속된 블록들인 익스텐트와 블록의 두 가지 단위로 공간을 할당하도록 함으로써 내부 단편화를 줄이고 inode 트리의 크기를 줄이도록 하였다. 셋째, 파일에 블록을 할당할 때 빈 공간을 찾는 데 오랜 시간이 걸리지 않도록 하기 위해서, 빈 공간 지갑을 두어 빈 공간의 주소를 관리함으로써 빠르게 빈 공간을 할당하거나 회수할 수 있도록 하였다. 마지막으로, 저널링에서는 메타데이터가 수정되고 저널에 기록된 후에는 디스크에 기록되기 전에라도 다른 클라이언트가 사용할 수 있도록 함으로써 디스크 접근 횟수를 줄이도록 하였다.

참고문헌

[1] Thomas E. Anderson, et. al., "Serverless Network File Systems," Proceedings of the 15th ACM Symposium on Operating Systems Principles, pp. 109-126, Copper Mountain Resort, Colorado, December 1995.
[2] Maurice J. Bach, The Design of the UNIX Operating System, Prentice-Hall, 1986.
[3] Philip A. Bernstein and Eric Newcomer, Principles of Transaction Processing, Morgan Kaufmann Publishers, 1997.
[4] 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.
[5] Clit Jurgens, "Fibre Channel: A Connection to the Future," IEEE Computer, vol. 28, no.

8, pp. 82-90, August 1995.

[6] Edward K. Lee and Chandramohan A. Thekkath, "Petal: Distributed Virtual Disks," Proceedings of the 7th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 84-92, Cambridge, Massachusetts, October 1996.

[7] 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.

[8] Kenneth W. Preslan, et. al., "Implementing Journaling in a Linux Shared Disk File System," Proceedings of the 17th IEEE Mass Storage Systems Symposium, pp. 351-378, College Park, Maryland, March 2000.

[9] Prashant J. Shenoy and Harrick M. Vin, "Cello: A Disk Scheduling Framework for Next Generation Operating Systems," Proceedings of the SIGMETRICS, pp. 44-55, Madison, Wisconsin, 1998.

[10] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee, "Frangipani: A Scalable Distributed File System," Proceedings of the 16th ACM Symposium on Operating Systems Principles, pp. 224-237, St. Malo, France, October 1997.

[11] Stephen C. Tweedie, " Journaling the Linux ext2fs File system," Proceedings of the LinuxExpo'98, http://www.nondot.org/sabre//os/S3FileSystems/journal-design.pdf, 1998.

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

[13] 김신우, 이용규, 김경배, 신범주, "대용량 파일 시스템을 위한 메타데이터 구조 설계," 한국정보과학회 추계 학술발표논문집, vol 27, no. 2, pp. 59-61, 숙명여자대학교, 2000. 10.

[14] 김신우, 이용규, 김경배, 신범주, "공유 디스크 파일 시스템을 위한 메타데이터 저널링 구조 설계," 한국멀티미디어학회 추계 학술발표논문집, pp. 534-537, 연세대학교, 2000. 11.

[15] 민병준, "System Management," 제 1 회 자료 저장 시스템 워크샵 발표 자료집, pp. 106-126, 제주도, 2000. 12.

[16] 신범주, "네트워크 연결형 자료저장시스템 SANtopia 소개," 제 1 회 자료 저장 시스템 워크샵 발표 자료집, pp. 88-105, 제주도, 2000. 12.

[17] 오상규, "SANUX 개발 현황," 제 1 회 자료 저장 시스템 워크샵 발표 자료집, pp. 127-138, 제주도, 2000. 12.

[18] 이강욱, "IBM SAN의 기술동향," 제 1 회 자료 저장 시스템 워크샵 발표 자료집, pp. 17-22, 제주도, 2000. 12.

[19] 정대규, "Compaq의 SAN과 NAS 솔루션," 제 1 회 자료 저장 시스템 워크샵 발표 자료집, pp. 23-40, 제주도, 2000. 12.

이 용 규

1986 동국대학교 전자계산학과(학사)
1988 한국과학기술원 전산학과(석사)
1996 Syracuse University(전산학 박사)
1978~1983 정보통신부 국가공무원
1988~1993 한국국방연구원 선임연구원
1996~1997 한국통신 선임연구원
1997~현재 동국대학교 컴퓨터멀티미디어공학과 교수
관심분야:XML 및 웹, 저장시스템, 데이터베이스, 정보검색
E-mail:yklee@dgu.edu

김 신 우

1997 동국대학교 컴퓨터공학과(학사)
2000 동국대학교 컴퓨터공학과(석사)
2000~현재 동국대학교 컴퓨터공학과(박사과정)
관심분야:XML 및 웹, 저장시스템, 데이터베이스, 정보검색
E-mail:purian@dgu.edu

손 덕 주

1976 서울대학교 수학교육과(학사)
1978 한국과학기술원 전산학과(석사)
1978~현재 한국전자통신연구원(책임연구원, 인터넷서비스연구부장)
관심분야:이동컴퓨팅, 이동단말 소프트웨어, 데이터베이스 시스템, 네트웍 자료저장 시스템 등
E-mail:djson@etri.re.kr

• 한국 데이타베이스 학술대회 2001 •

- 일 자 : 2001년 6월 1~2일
- 장 소 : 한국과학기술회관
- 논문제출마감 : 2001년 4월 9일(월)
- 주 최 : 데이타베이스연구회
- 문 의 처 : 전남대학교 컴퓨터정보학과 이도헌 교수
  Tel. 062-530-3427/0110
  E-mail : dhlee@dbcc.chonnam.ac.kr

• 알고리즘과 계산이론에 관한 한·일 공동 워크샵 •

- 일 자 : 2001년 6월 28~29일
- 장 소 : 부산대학교
- 논문제출마감 : 2001년 4월 21일(토)
- 주 최 : 컴퓨터이론연구회
- 문 의 처 : 서강대학교 컴퓨터학과 장직현 교수
  Tel. 02-705-8491
  E-mail : jchang@alglab.sogang.ac.kr