대용량 공유디스크 파일 시스템에 적합한 메타 데이타 구조의 설계 및 구현

(Design and Implementation of a Metadata Structure for Large-Scale Shared-Disk File System)

이 용 주 [†] 김 경 배 [††] 신 범 주 [†††]

(Lee Yong Ju) (Kim Gyoungbae) (Shin Bumjoo)

요 약 인터넷의 확산으로 대용량 멀티미디어 데이타에 대한 요구가 증가하고 있으며 이를 효율적으로 관리하기 위한 스토리지에 대한 연구가 진행되고 있다. 기하급수적으로 늘어나는 스토리지에 대한 요구를 해결하기 위해서 제시된 방법중의 하나가 공유디스크 환경을 제공하는 SAN(Storage Area Network)이다. SAN은 fibre channel이라는 고속 전송망을 이용해서 고속의 저장장치를 위한 네트워크를 구성한 것이다. 하지만 저장장치 네트워크의 구성만으로는 스토리지에 대한 요구는 해결하였지만 이를 사용자에게 제공하기 위한 공유디스크 환경에서의 파일시스템에 대한 연구는 미진하다. 특히 기존에 제시된 로컬 파일 시스템, 분산 파일시스템에서는 공유디스크환경에 적합하지 않으며 대용량 스토리지에 적합한 메타 데이타 구조 측면에서의 설계는 부족한 실정이다.

이를 해결하기 위해 본 논문에서는 공유디스크 환경에 적합한 메타 데이타 구조를 설계 및 구현하여 대용량 스토리지에 적합한 공유디스크 파일 시스템을 제시한다. 구현한 공유디스크 파일시스템은 SAN fabric에 참여하는 호스트들 사이의 균형적인 할당 블록을 주기 위한 파일시스템 레이아웃과 비트맵 관리 기법, 대용량 파일을 위한 효율적인 익스텐트 기반의 SEMI FLAT구조를 제안하였으며, 대용량의 디렉토리를 사용할 수 있게 확장해싱을 이용한 2단계 디렉토리 관리 구조를 설계 및 구현하였다. 또한 리눅스 커널 상에서 제시한 메타 데이타 구조에 대한 구현에 필요한 구조 정보와 알고리즘을 제시하였으며, 성능의 우수성을 보이기 위해 리눅스 환경의 대표적인 파일 시스템인 EXT2, 공유디스크 환경의 GFS와의 성능을 파일 생성, 디렉토리 생성, I/O횟수 측면에서 비교하였다.

키워드 : 메타 데이타 구조, 공유디스크 파일시스템, 클러스터, 공유디스크 저장장치

Abstract Recently, there have been large storage demands for manipulating multimedia data. To solve the tremendous storage demands, one of the major researches is the SAN(Storage Area Network) that provides the local file requests directly from shared-disk storage and also eliminates the server bottlenecks to performance and availability. SAN also improve the network latency and bandwidth through new channel interface like FC(Fibre Channel). But to manipulate the efficient storage network like SAN, traditional local file system and distributed file system are not adaptable and also are lack of researches in terms of a metadata structure for large-scale inode object such as file and directory.

In this paper, we describe the architecture and design issues of our shared-disk file system and provide the efficient bitmap for providing the well formed block allocation in each host, extent-based semi flat structure for storing large-scale file data, and two-phase directory structure of using Extendible Hashing. Also we describe a detailed algorithm for implementing the file system's device driver in Linux Kernel and compare our file system with the general file system like EXT2 and shared disk file system like GFS in terms of file creation, directory creation and I/O rate.

Key words : Metadata Structure, Shared-Disk File System, Cluster, Shared Disk Storage

---

† 정 회 원 : 한국전자통신연구원
yongju@etri.re.kr

†† 정 회 원 : 한국전자통신연구원
gbkim@etri.re.kr

††† 정 회 원 : 밀양대학교 컴퓨터.정보통신 공학부
bjshin@mnu.ac.kr

논문접수 : 2002년 1월 7일
심사완료 : 2002년 10월 14일

1. 서 론

최근 들어 멀티미디어 데이타에 대한 기하급수적인 증가와 이를 처리하기 위한 대용량 스토리지에 대한 요구가 증가하고 있다. 기존의 서버에 집중된 스토리지 관리에는 한계가 있으며 이를 해결하기 위한 여러 형태의 스토리지 관리 모델이 제시되어 왔다. 스토리지 관리 모델의 대표적인 모델이 NAS(Network Attached Storage)와 SAN(Storage Area Network)이다. NAS는 기존에 TCP/IP 프로토콜을 사용하여 파일 단위로 데이타에 접근할 수 있으며 이기종이 혼재된 환경에서 파일 접근 및 공유 서비스를 할 수 있다는 장점이 있다. 또한 SAN은 데이타 저장에 필요한 데이타 트래픽을 기존 네트워크로부터 분리하여 대역폭을 절약할 수 있으며 데이타의 접근을 물리적인 블록 단위로 수행하므로 대용량 데이타에 대한 고속 전송이 가능하게 한다.

SAN과 같은 공유디스크 환경의 가장 중요한 특징은 SAN fabric에 참여하는 호스트들 모두에게 RAID, JBOD에 물려 있는 디스크가 공통적으로 보이는 것이다. 이러한 공유디스크 환경에서는 기존에 제시된 스토리지를 관리하는 로컬 파일시스템 형태는 여러 가지 이유로 적절치 못하다. 먼저 SAN fabric에 참여하는 FC 디스크는 일반적으로 GB,TB를 넘는 단위의 디스크 형태이지만, 이를 수용하기 위한 파일시스템은 2GB정도의 파일 사이즈의 한계를 보이고 있으며, 계층적 구조를 나타내기 위한 디렉토리 또한 리스트 구조로 되어 있어 다수의 디렉토리를 만들면 효율성이 떨어진다. 아울러 SAN fabric의 여러 호스트들 사이에 디스크를 공유하므로 특정 호스트가 변경한 데이타에 대해 다른 호스트의 일관성 유지 기법이 요구되어 진다. 이러한 문제들을 해결하기 위해 익스텐트 기반의 불필요한 간접 블록의 I/O를 없앤 세미플랫 아이노드 구조를 설계 및 구현하여 파일 사이즈에 대한 한계를 없애고, 대용량 디렉토리를 지원하기 위해서 변형된 확장해싱 기법을 설계 및 구현한다. 아울러 SAN fabric에 참여하는 각각의 호스트가 사용하는 파일에 대한 일관성 유지와 디스크 영역에 존재하는 비트맵 영역의 균형적 분배 알고리즘을 설계 및 구현한다.

이에 본 논문의 2절에서는 기존에 제시된 파일시스템을 소개하고 이에 대한 문제점을 제시하고, 3절에서는 제시하는 파일시스템의 레이아웃과 비트맵 관리 기법, 아이노드와 디렉토리 관리 기법 그리고 각각의 호스트에서의 일관성 유지 기법에 대해 설명한다. 4절에서는 구체적으로 제시하는 레이아웃, 비트맵, 파일, 디렉토리에 대한 리눅스 커널 상에서의 구현에 따른 구조 정보와 알고리즘을 제시한다. 5절에서는 구현된 파일시스템에 대한 성능평가를 통해 기존에 제시된 대표적인 로컬 파일시스템인 EXT2[1], 공유디스크 환경에 적합하게 제시된 sistina의 GFS[2, 3]와 파일 생성, 디렉토리 생성, I/O횟수 측면에서 비교하여 결과를 제시한다. 마지막으로 6절에서는 제시한 파일시스템의 결론과 향후 연구 방향에 대해 제시한다.

2. 관련연구

기존 스토리지 시스템은 여러 개의 저장 장치를 파일 서버를 통해 처리하는 모습을 보여주고 있으며, 차츰 대용량의 스토리지에 대한 요구를 통해 스토리지 전용 네트워크를 구성하는 쪽으로 연구의 방향이 진행되고 있다. 먼저 NAS(Network Attached Storage)에서는 스토리지를 LAN에 부착하여 신속하게 사용할 수 있으며, 기존의 제시된 프로토콜을 채용하고 있으며 데이타의 접근 단위가 파일이다. 이를 위해 제시된 대표적인 스토리지 관리가 CMU의 NASD(Network Attached Secure Disks)[4, 5]이다. NASD에서는 디스크 관리 기능을 디스크에 내장하여 가변길이 객체 인터페이스를 통해 저장장치에 접근하도록 한다. 이 구조에서는 파일 관리자가 사용권한을 클라이언트에게 줌으로써, 클라이언트가 특정한 스토리지를 직접적으로 반복해서 접근할 수 있도록 한다. 결국 모든 제어정보가 네트워크를 통해 한번만 이동할 수 있다는 장점이 있다. [그림 1]은 CMU에서 제안된 대표적인 NASD 모델을 보여주고 있다. 그림 1의 ①의 과정에서는 클라이언트가 특정 디스크에 접근하기 위해 초기 파일 매니저에게 NASD디스크에 접근할 수 있는 권한을 부여받기 위해 요청하는 과정이다. 클라이언트의 접근 요청에 대해 파일 매니저는 여러 클라이언트에 대한 접근 제어 리스트를 보유하고 있으며, 접근이 허락될 경우 클라이언트에게 ②의 과

그림 1 CMU에서 제안하는 NASD디스크 모델

정처럼 디스크에 대한 접근을 허락한다. 접근을 허락 받은 클라이언트는 다음부터는 NASD디스크에 직접적으로 파일 매니저의 권한 부여 과정 없이 접근하여 네트워크를 통해 ③과 ④의 과정에서 보이는 것과 같이 데이타를 전송하고 받을 수 있다. 하지만 NASD의 연구는 하드웨어적인 디스크의 확장을 통해 저장공간을 관리하는 특성을 띠고 있으며 스토리지 접근을 위해 클라이언트 네트워크에 종속적인 네트워크 트래픽을 그대로 안고 있다.

다음으로 SAN은 스토리지 전용 네트워크를 구성하고 데이타 저장에 관련된 트래픽을 네트워크로부터 분리시켜 LAN의 대역폭을 넓혔으며 데이타의 백업 및 복구가 용이하고 스토리지 집중 관리를 가능하게 하였다. 또한 SAN은 Fibre Channel[6]이라는 광 전송망을 사용하게 되는데 *Fibre Channel*은 블록단위전송, 오류검출 및 복구, 흐름제어, 다양한 상위계층 프로토콜을 지원함으로써 NAS에 비해 하드웨어적인 흐름제어를 한다. 이를 통해 기존 TCP/IP프로토콜의 소프트웨어 수준에서 처리하는데 반해 처리 간접비(overhead)가 적다는 장점이 있다. 이러한 스토리지를 관리하는 공유디스크 환경에서는 여러 개의 호스트가 스토리지를 접근할 수 있게 되었지만 기존에 제시한 분산 파일 시스템은 적합하지 않다. 이를 개선하기 위해 공유디스크 환경에 적합하게 제안된 Sistina의 GFS(Global File System)가 있다.[2] GFS에서는 디바이스 락을 사용하여 일관성을 유지하며 아이노드 구조는 FULL FLAT구조를 채택하였다. 또한 NSP(Network Storage Pool)라는 소프트웨어적인 볼륨 매니저를 두고 볼륨에 전체적으로 파일시스템의 데이타를 저장하는 형태이다. GFS의 기본 구조는 각각의 리소스 그룹으로 나누어지며 각각은 RG블록, 데이타 비트맵, Dinode 비트맵, Dinode로 나누어진다.[2, 3]

하지만 기존에 제시된 GFS에서는 몇 가지 문제점을 도출하고 있다. 먼저 GFS에는 대용량 파일을 위한 구조와 디렉토리 관리 구조에서 FULL FLAT구조를 채택하여 사용하고 있다. FULL FLAT구조는 대용량 파일을 생성할 때 간접블록의 과다한 생성을 야기하여 처리 간접비를 증가시킨다. GFS의 FULL FLAT구조의 예는 [그림 2]와 같다. 그림 2에서 의 ①에서 적은 양의 데이타에 대한 아이노드 정보를 가진 루트블록에 파일 데이타를 같이 저장하는 Stuffed방식을 저장하였다가 데이타가 가득 차면 ②와 같이 Unstuffing과정을 거치게 된다. 루트블록에 저장된 파일 데이타를 새로운 블록에 카피하고 새로운 블록의 블록 넘버를 루트 데이타 영역에 저장하게 된다. 이런 과정이 계속 이루어져서 ③과 같이 1 레벨의 데이타가 모두 차게 된다. ④의 과정은 기존 ③의 과정에서 1레벨이 모두 찬 경우 새로운 데이타 쓰기 요청이 들어오면 새로운 블록을 얻어서 기존의 루트 블록에 저장된 블록넘버들을 저장하고 루트블록에는 새로 할당된 블록 넘버를 저장하는 것이다. 이러한 과정을 거쳐 균형 잡힌 트리를 구성하게 되는 것이며 파일 사이즈에 대한 제한도 구애받지 않는다. 하지만 ⑤에서와 같이 한 리프 블록 군이 다 차게 되면 Full Flat구조에서는 새로운 간접노드와 데이타를 저장하게된 블록을 할당받아 저장하는 형태이며 부가적인 간접노드 할당에 따른 I/O가 발생하게 된다. ⑥과 같이 2 레벨이 가득 찬 경우까지 오게되면 루트 블록에 저장되는 (한 블록에 저장할 수 있는 최대 블록 넘버 개수 - 1) 만큼의 간접노드에 대한 I/O가 계속 발생하는 구조이다. 더 확장되어 ⑦과 같이 3 레벨이 되는 경우는 이의 두 배로 (2 * (한 블록에 저장할 수 있는 최대 블록 넘버 개수 - 1)) 간접노드에 대한 I/O가 계속 발생하는 구조이다. Full Flat구조는 계산식에 의해 쉽게 구할 수 있는 장점이 존재하지만 레벨이 증가할수록 간접노드에 대한 I/O가 부가적으로 발생하는 현상을 초래한다. 또한 실제 저장할 수 있는 블록넘버의 개수가 제한적이어서 트리의 레벨이 지수 적으로 증가하는 단점을 가지고 있다.

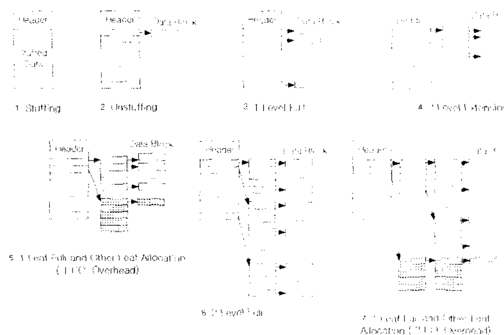


그림 2 GFS의 아이노드 확장을 위한 FULL FLAT구조

또한 GFS에서는 SAN fabric에 참여하는 호스트 사이의 비트맵 관리를 리소스 그룹을 통해 병렬화를 기했지만, 호스트 별로 데이타 요구 량의 차이를 개별 호스트의 블록 할당 정책에 반영할 수가 없다. 아울러 일관성 유지를 위해 디바이스 락을 사용하여 하드웨어적인 관리가 이루어지고 있으며 변경된 데이타에 대한 호스트 별 일관성 유지를 한 호스트가 메타 데이타를 요청하면 다른 호스트에 의해 이미 변경된 메타 데이타를 디스크에 반영하고 요청한 호스트는 다시 디스크에서

읽는 구조로 잦은 디스크 I/O를 통해 성능 저하의 원인이 되고 있다. 한 예로 GFS에서 제시하는 메타 데이타 일관성 유지에 대한 기법은 그림 3과 같다. 그림 3의 ①의 과정에서 클라이언트 1이 특정 데이타 A를 읽어서 변경하고 디스크에 아직 반영이 되지 않은 상황일 때, 다른 호스트 클라이언트 3이 ②의 과정처럼 같은 데이타 A에 대한 읽기 요청을 하면 ③의 과정에서처럼 그 데이타를 먼저 디스크에 반영하고 ④의 과정에서 디스크에서 읽는 구조이다. 이러한 구조는 메타 데이타의 일관성 유지를 디스크를 통해서 하는 대표적인 예이다.

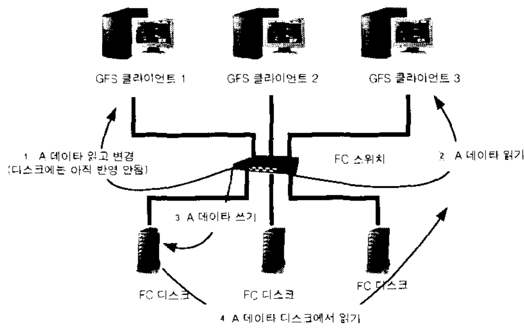


그림 3 GFS의 메타 데이타 일관성 유지기법

앞에서 제시한 GFS에서의 문제점을 해결하기 위해 제안하는 파일 시스템에서는 메타 데이타의 효율적인 저장을 위해 레이아웃과 비트맵의 관리기법을 재설계하였고, GFS의 FULL FLAT구조의 단점을 개선하기 위해 새로운 익스텐트 기반의 SEMI FLAT 구조를 제안하여 대용량 파일과 디렉토리를 다루는데 용이하게 하고, 아울러 캐시 일관성을 위해 전역버퍼에 기반한 새로운 캐쉬일관성 정책을 구현하였다.

3. 공유 디스크 파일시스템의 설계

제안하는 공유 디스크 파일 시스템은 공유 디스크 환경 하에서 스토리지 네트워크와 컴퓨터 네트워크를 분리하여 스토리지 네트워크에 적합한 파일시스템을 제공하고 사용자에게는 하나의 클러스터 컴퓨팅 파워를 제공하는 것이다. 제안하는 파일시스템인 SFS(SANtopia File System)[7, 8]가 적용되는 공유디스크 환경의 예는 그림 4와 같다.

그림 4 에서는 4-way FC(fibre channel) disk가 사용된 예이며 FC Switch에 FC Disk(RAID,JBOD)가 물려 있으며 클러스터를 구성하기 위한 노드들이 존재한다[6, 11, 12].

각각의 노드는 SAN NIC Card를 가지고 있으며 FC Switch를 통해 디스크 자원을 서로 공유하게 된다. 또한 각각의 클러스트 노드들은 로컬 파일시스템처럼 사용할 수 있으며 상위 클러스터링을 통해 사용자에게 하나의 fabric을 구성하게 된다. 전통적인 클러스터링은 서버의 디스크 자원을 사용하여 부하 분산을 하는데 반해 제안하는 시스템 환경은 하부 디스크 저장 구조를 SAN fabric으로 구성하고 상위 클러스터링 기법으로 노드들을 관리하며 사용자에게 하나의 fabric을 제시하는 것이다.

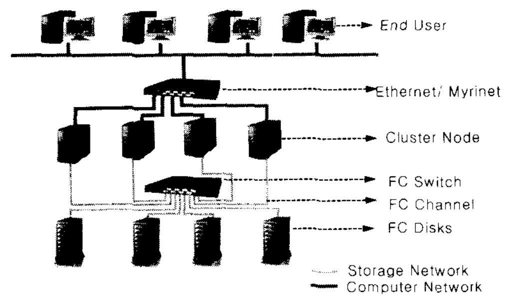
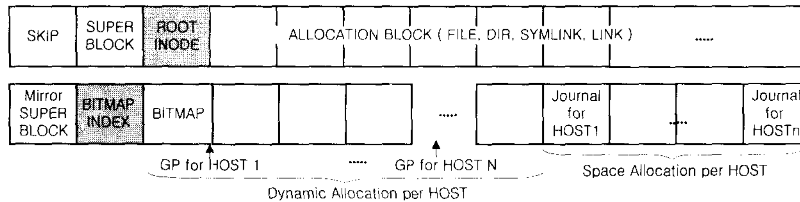


그림 4 SFS의 시스템 환경

본 절에서는 공유디스크 환경에 적합한 파일 시스템 레이아웃과 단일 블록 할당과 벌크 블록 할당을 위한 이중화 비트맵 관리를 보이며, 효율적인 아이노드 확장을 위한 익스텐트 기반의 동적 세미플랫 구조를 제안한다. 또한 확장 해싱을 이용한 2단계 디렉토리 관리 기법도 기술한다[9].

3.1 파일시스템 디스크 레이아웃

제안하는 파일시스템의 레이아웃은 그림 5와 같다. 파일시스템의 레이아웃을 설정하기 위해서는 디바이스 파일을 열어서 전체 디스크의 사이즈를 계산한 다음에 필요한 공간을 계산을 통해 얻는다. 제일 먼저 파일시스템의 디스크 사용정보, 저널 정보, 블록 사이즈 등을 저장하고 있는 슈퍼 블록은 미러링을 통해서 보호한다. 마운트 포인트가 되는 루트 아이노드 블록이 존재하며 그 다음으로는 파일시스템의 아이노드 객체인 파일, 디렉토리, 링크에 사용되어질 블록이 저장되는 할당 블록이 존재한다[10]. 다음으로 디스크의 사용정보를 비트 단위로 저장하는 비트맵이 존재하며 이를 호스트 수에 따라 동적으로 관리하게 된다. 마지막으로 호스트 수에 따라 메타 데이타의 일관성 유지를 위한 저널링에 사용될 저널 공간을 각각 저장하게 된다.



* Key Feature
- Variable extents : 1K, 2K, 4K, 8K, 16K, 32K, 64K
- Mirrored Superblock
- Dynamic bitmap alloation per host
- Journaling supports

그림 5 파일시스템 디스크 레이아웃

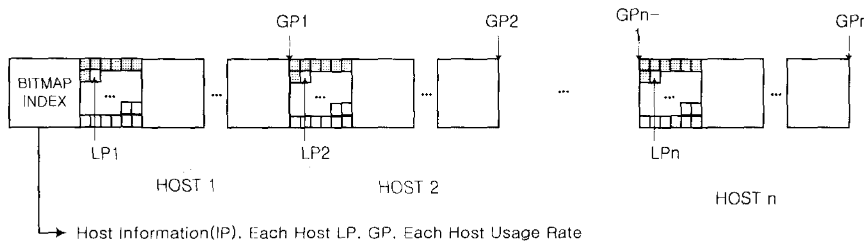


그림 6 공유디스크 환경 하에서의 비트맵 관리

3.2 데이타 요구 량에 기반한 비트맵

파일시스템에서 요구되는 비트맵의 특징은 크게 다음과 같다. 첫째, 파일시스템의 연속된 비트맵은 파일이나 디렉토리와 같은 아이노드 객체에 대한 사용 유무를 나타내며 비트맵 블록의 연속된 비트 스트림은 연속된 아이노드 블록 넘버를 가리키게 된다. 공유 디스크환경에서는 여러 호스트에서 비트맵블록을 사용하므로 이러한 연속성을 해칠 수 있게 된다. 가령 호스트 A가 파일을 만들고 호스트B가 파일을 만드는 작업을 번갈아 계속 수행하면 각각의 호스트에서 서로 디스크 상에 연속된 블록을 사용할 수 없게 되므로 디스크의 Seek Time의 부하를 일으킬 수 있다. 둘째, 비트맵도 역시 블록이므로 한 블록에 포함된 비트맵은 블록사이즈 4K, 비트맵 비트 2Bit인 경우 최대 4096 * 8 / 2=16384블록을 가리키게 되며 호스트에서 16384개의 아이노드 객체를 만들 때 아이노드에 대한 I/O와 비트맵 변경을 위한 최소 16384번의 I/O가 한 블록에 집중되게 된다. 셋째, 공유디스크 환경에서 로컬 디스크 환경과 로컬 디스크환경을 이용한 네트워크 환경과 같은 경우와는 달리 여러 호스트가 디스크를 공유함으로써 일정량의 비트맵을 할당하는 것만으로는 해결할 수 없는 문제가 발생한다. 각각의 호스트에서 작업하는 아이노드 객체의 생성, 삭제에 따른 서로 나누어 가진 비트맵의 경계값을 벗어나는 경우가 발생한다. 이러한 문제점들을 해결하기 위해 초기 파일시스템 레이아웃을 만드는 단계에서 각각의 호스트에 대한 IP정보와 호스트 수에 따라 일정량의 비트맵을 할당하고 마운트 시기에 비트맵을 일정량씩 프리리스트를 호스트별로 유지하는 정책을 사용한다. 제시하는 비트맵 레이아웃은 다음과 같다.

그림 6에서는 SAN fabric에 참여하는 호스트에 정보를 가진 비트맵인덱스 블록이 존재하며 호스트 개수별로 비트맵 영역을 각각 할당받는다. 할당된 비트맵에는 각각 LP(Local Pivot)과 GP(Global Pivot)가 존재한다. LP는 호스트 별로 사용되어지는 비트맵의 다음 위치를 나타내며 GP는 파일시스템 차원에서 사용되는 호스트 경계 값을 나타낸다. 만약 한 호스트에서 작업량의 과중으로 비트맵이 모두 찬 경우 바로 이전의 호스트에 할당된 뒤쪽 영역의 참조를 통해 호스트의 비트맵 할당에 필요한 비트맵 블록을 얻고 이전 블록의 GP를 줄이게 된다. 만약 이전 블록의 GP를 참조하여 더 이상 공간이 할당받을 수 없다면 사용되어질 Victim을 설정해야 하며 이를 위한 기준은 가장 가까운 거리의 비트맵 영역을 가진 곳을 할당하고, 이러한 조건도 없는 경우에는 비트맵 인덱스 블록을 통해 가장 많이 남은 비트맵 블

록의 뒤쪽 영역을 할당받는다.

아울러 익스텐트 단위의 할당을 위해 벌크 비트맵을 로딩하기 위한 비트맵 이중화 작업이 필요하다. 일반 파일시스템의 메타 데이타는 한 블록의 공간을 사용하며 파일의 데이타를 위해 EXTENT_FACTOR만큼의 연속된 비트맵을 할당 벌크 비트맵 할당 관리가 존재한다. 공유 디스크 환경에서의 비트맵관리 기법은 개별 호스트에서 각각의 영역의 비트맵을 할당 및 해제할 때는 최적의 조건을 제공해 주지만 서로 다른 호스트에서 할당과 해제가 빈번히 일어나는 교차작업에는 비효율적인 면이 존재한다. 가령 호스트 A 에서 비트맵을 계속 할당하고 호스트 B 에서 이전 호스트 A가 만든 아이노드 객체를 지우는 작업을 계속하면 호스트 A에 할당된 비트맵에 계속된 병목현상이 발생하게 된다.

3.3 익스텐트 기반 세미 플랫 아이노드 구조

파일시스템의 파일, 디렉토리, 링크, 특수 파일 등과 같은 모든 객체는 하나의 아이노드 형태로 블록디바이스에 저장되게 된다. 이러한 이유로 많은 파일시스템 별로 각각의 아이노드 관리 정책이 존재하며 대표적인 파일시스템인 EXT2와 GFS를 통해 아이노드 관리의 문제점을 지적하고 본 논문에서 제시하는 익스텐트 기반의 세미 플랫 구조(Extent-based Semi-Flat Structure)를 제안한다.

먼저, 리눅스 환경의 대표적인 로컬 파일시스템인 EXT2의 아이노드 확장 구조는 12개의 직접노드에 파일 데이타가 저장되는 구조이며 직접 노드구조에 데이타가 가득 찬 경우 간접노드 13번째 포인터에 새로운 블록을 할당받고 이를 데이타 블록을 가리키는 포인터들의 블록넘버를 저장하는데 사용한다. 이러한 구조를 반복하여 삼중간접 블록을 가지는 구조까지 확장될 수 있다. 결국 12개의 직접 노드와 간접, 이중 간접, 삼중 간접 블록 넘버를 저장하는 구조체 정보가 존재하며 파일의 크기가 커지면 커질수록 그에 따른 간접 노드의 I/O에 따른 오버헤드가 발생하게 된다. 이론적으로 저장할 수 있는 파일의 크기도 제한되며 블록사이즈를 4K로 했을 때 직접노드(4K*12=48K), 간접노드(4K*256=1M), 이중간접노드(4K*256256=256M), 삼중간접노드(4K*256*256*256=64G)를 가리킬 수 있다. 이러한 SKEWED된 아이노드 구조로는 대용량의 파일에 적합하지 않으며 이러한 문제를 GFS에서는 Full Flat구조로 해결하고자 하였다. 하지만 GFS에서도 역시 간접 블록에 대한 불필요한 I/O오버헤드가 존재하며 블록 단위 맵핑을 통해 레벨이 급속도록 증가하는 단점을 가진다.[2] 이에 본 논문에서는 Extent-based Semi Flat구조를 제안한다. 제안하는 구조는 다음과 같은 특징을 가진다.

- 블록단위의 할당이 아닌 익스텐트 단위로 파일 데이타를 논리적으로 매핑 한다.
- 부가적인 간접노드에 대한 접근을 막기 위해 유연한 세미 플랫 구조를 사용한다.

먼저, 기존의 파일시스템은 블록 단위의 관리를 사용하고 있으며 이는 대용량 파일에 대한 지원 측면에서 아이노드 관리 구조의 오버헤드를 초래한다. 관리 구조의 잦은 확장 원인은 블록 안에 저장할 수 있는 직접, 간접 블록 넘버의 수가 한계가 있으며 이 때문에 확장 구조가 요구되어 실제 파일의 데이타 블록에 접근하기 위한 간접 블록들의 I/O를 초래한다. 이러한 문제를 줄이기 위해서는 파일시스템 측면에서 익스텐트 형태의 관리가 필요하다. 그림 7은 기존의 블록 기반 할당 정책과 일반적인 익스텐트 할당 정책을 보인 것이다.

그림 7의 블록기반 할당은 일반적인 파일 시스템에서 사용하는 블록 기반 할당 방법이다. 파일의 데이타를 저장하기 위해 데이타 블록의 블록 넘버를 루트 블록에 저장하는 구조이다. 이는 파일의 데이타가 커지면 커질수록 급속도로 블록의 주소를 저장하는 공간이 부족하게 된다. 이를 개선하기 위해 익스텐트 기반할당 구조는 일반적인 익스텐트 기반의 할당을 보인 것이다. 가변길이 익스텐트를 사용하려면 매 익스텐트마다 연속된 블록의 개수를 저장하기 위한 length필드도 같이 루트블록에 저장하여야 한다. 또한 계산에 의해 임의의 블록을 접근하기가 어렵고 처음부터 순차적으로 length필드를

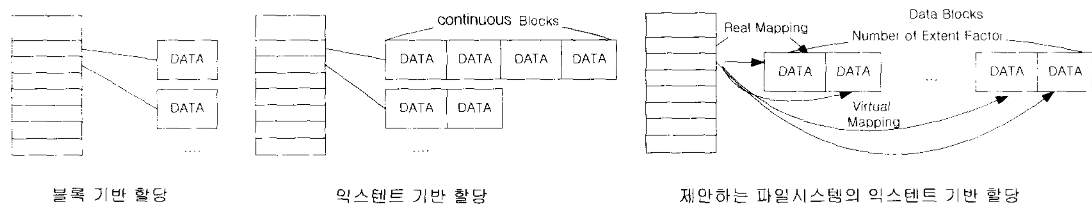


그림 7 아이노드 확장을 위한 할당 기법

더하여 찾고자 하는 블록넘버를 얻을 수 있다. 이에 제안하는 파일시스템은 고정 길이 익스텐트를 사용하여 단순한 수식에 의해 임의의 파일 데이타 블록을 얻을 수 있다. 또한 일반적인 익스텐트 기반 할당기법에서 사용하는 length필드가 불필요하게 되어 좀더 많은 블록넘버를 가상으로 루트블록에 저장할 수 있는 것이다. 그래서 제안하는 파일시스템의 익스텐트 기반 할당은 의 예에서 만약 익스텐트 계수가 4개이던 블록기반 할당에서는 4개의 블록넘버 주소공간이 사용되어지지만, 제안하는 기법에서는 하나의 주소 공간만이 사용되게 되는 것이다. 아울러 익스텐트 기반의 아이노드를 통해 얻을 수 있는 장점은 연속된 블록을 얻을 수 있어 순차적인 파일 데이타 탐색에 유리하며, 비트맵 할당의 오버헤드를 줄일 수 있다. 한 예로 익스텐트 계수가 16이고, 블록 사이즈가 4K이며 저장되는 데이타가 64K이면 저장할 때마다 비트맵에서 할당 블록을 얻어야 하므로 16번의 디스크 I/O가 발생하게 되지만 제시하는 방법은 1번의 비트맵 블록 I/O를 통해 해결할 수 있다. 이러한 익스텐트 기반의 파일 데이타 관리를 통해 제시하는 아이노드 동적 세미 플랫 구조는 그림 8과 같다.

그림 8에서 적은 양의 데이타에 대한 아이노드는 GFS와 같이 루트블록에 파일 데이타를 같이 저장하는 방식의 Stuff방식을 채택하였으며 Stuff된 데이타가 다 찬 경우 1 레벨로의 확장 시에 Extent Factor에 따라 동시에 비트맵에서 벌크 할당을 받아서 익스텐트의 맨 처음의 블록넘버를 저장하게 된다. 계속해서 확장되면 1Level이 가득 찬 경우의 예와 같다. 만약 더 큰 파일 데이타의 쓰기 요청이 있을 경우에 한 레벨이 확장되고 파일 데이타에 대한 블록 넘버는 루트 블록에 간접노드의 생성 없이 바로 블록 주소 값을 저장하게 된다. 이렇게 함으로써 Stuffed된 블록 주소 공간이 사용하고, 새로운 간접블록의 생성은 필요한 경우에만 생성되게 된다. 한 예로 구현한 파일 시스템의 파일 데이타를 저장할 경우 GFS에서는 1G정도를 1레벨에서 저장 할 수 있지만 제안하는 파일 시스템은 16G정도까지 1레벨 안에서 접근이 가능하게 되는 것이다. 아울러 그 이상의 파일사이즈를 가진 파일의 경우에도 간접 노드의 생성이 없으므로 좀더 빠른 검색이 가능하다.

3.4 2단계 디렉토리 관리

파일시스템의 디렉토리는 UFS(Unix File System)를 비롯해서 일반적으로 계층적 구조(Hierarchical structure)를 가진다[10]. 한 디렉토리에는 하부에 저장되는 파일, 디렉토리, 링크 관련 정보를 디렉토리 엔트리 형태로 저장하고 있으며 한 블록에 저장되는 디렉토리 엔트리의 수가 증가하면 새로운 블록을 할당받아 연결하는 구조를 채택하고 있다. 리눅스의 대표적 파일시스템인 EXT2의 디렉토리 구조는 그림 9와 같이 링크드 리스트 형태로 여러 블록을 잇는 구조를 가지고 있으며 이는 엔트리의 검색에 매우 비효율적인 구조이다. 한 디렉토리 밑에 엔트리의 수가 급증하면 지수 적으로 성능이 감소하는 단점을 가지고 있다.

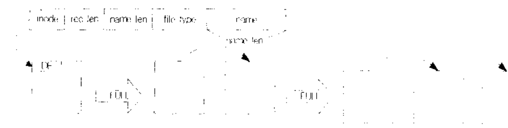


그림 9 EXT2의 링크드 리스트 디렉토리 구조

이에 GFS에서는 디렉토리 엔트리의 검색 속도를 높이기 위해서 엔트리에 저장된 이름을 이용하여 해싱을

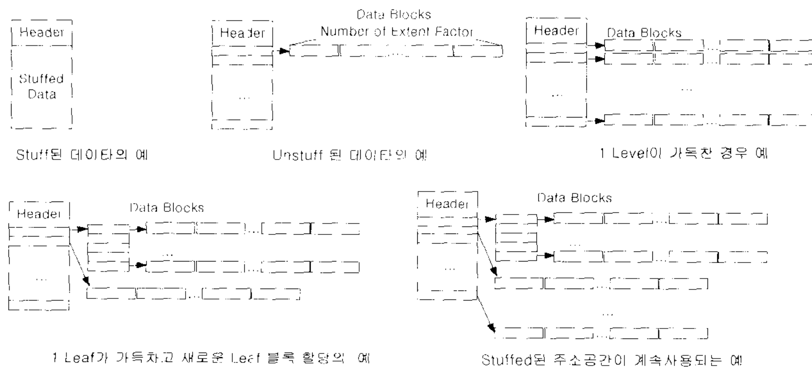


그림 8 제안하는 파일시스템의 아이노드 확장 구조

수행한다. 확장해싱을 통한 디렉토리 엔트리 저장 방법은 초기에 한 블록에 저장된 디렉토리 엔트리의 수가 가득 차게 되면 한 블록을 얻어서 기존에 저장된 블록의 엔트리를 순차적으로 검색하여 한 비트를 보고 0이면 기존 블록에 남기고 1이면 새로 할당된 블록으로 옮기는 재구성 작업을 수행하게 된다. 이를 통해 새로운 엔트리가 들어오면 해시 값에 의해 0이 저장된 블록과 1이 저장된 블록중의 한 곳에 저장이 이루어진다. 하지만 찾은 최종 블록에서의 엔트리 삽입 위치의 검색은 처음부터 순차적으로 검색하여 엔트리의 삽입 위치를 찾는다. 이는 최악의 경우 한 블록이 거의 찬 경우 맨 마지막까지 스캔을 통해 최종 저장 위치를 찾는 방법으로 매우 비효율적이다. 하지만 기존에 제시된 EXT2의 디렉토리 엔트리 저장 구조에 비해 지수적인 검색 속도 증가를 방지하였으며 전체적으로 FULL FLAT구조를 채택하였다. 하지만 앞에서 설명한 Full Flat구조 구성에 따른 불필요한 I/O와 한 블록 내에서의 검색은 여전히 순차 검색을 따른다. 그림 10에서는 GFS의 엔트리 확장 구조와 한 블록 내에서의 엔트리 순차적인 검색을 보인 것이다.[3]

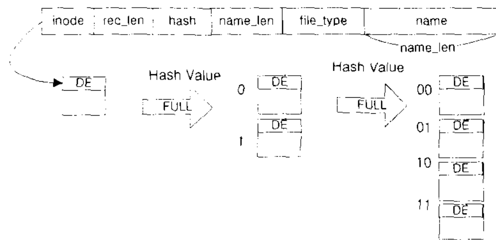


그림 10 GFS 디렉토리 구조

이에 본 논문에서 제시하는 디렉토리는 크게 두 가지의 절차(Two-Phase)로 디렉토리 엔트리의 삽입, 삭제, 검색 등을 수행한다. 앞에서 설명한 아이노드 구조를 채택하여 VFS의 논리적 블록 넘버의 요구를 Physical블록 넘버로 유연하게 변환 구성하는 세미 플랫 구조를 사용하며 마지막 단말노드의 논리적 해시 테이블을 구성하여 디렉토리 엔트리를 저장한다. 또한 한 블록내의 삽입은 scan_indicator를 두어 다음에 삽입될 위치를 미리 저장한다. 이를 통해 파일시스템의 엔트리 저장 구조에서는 한 블록 내에서 삽입되어질 엔트리의 저장 위치를 순차적으로 탐색하는 방법을 배제한다. 이를 통해 삽입과 재삽입의 성능을 높일 수 있으며 삭제 또한 삭제된 여러 곳의 엔트리의 블록 처음에서 가장 가까운 곳을 저장하며 앞부분의 스캔을 방지하여 빠른 검색을 가능하게 한다. 제시하는 디렉토리의 구조는 그림 11과 같다.

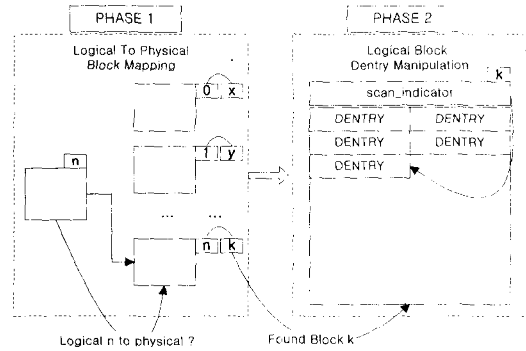


그림 11 제안하는 파일시스템의 2단계 디렉토리 구조

1단계에서는 가상 해시 테이블 맵핑 단계이다. 이 단계에서는 본 논문에서 제시한 동적 세미 플랫 구조를 채택하여 사용하였으며 맨 마지막 데이타가 저장되는 블록의 가상 해쉬 테이블 형태로 변형하였다. 이를 통해 아이노드의 확장에 따른 불필요한 간접노드의 접근을 막았으며 기존에 제시된 확장 해싱 기법을 그대로 이용할 수 있는 장점이 존재한다. 2단계에서는 scan_indicator를 통해 빠른 삽입 위치를 찾는 엔트리 삽입 단계이다. 한 블록 내에서의 순차 검색 을 통한 엔트리 삽입 방법에 비해 바로 다음에 삽입되는 엔트리의 위치를 기억하는 구조이다. 또한 엔트리의 삭제가 이루어지면 scan_indicator를 이곳으로 설정하여 바로 다음에 재삽입이나 새로운 엔트리의 삽입 시에 scan_indicator다음 위치부터 순차 검색을 통해 최소한의 엔트리의 불필요한 스캔을 방지하였다.

이러한 2단계 디렉토리 구조를 이용해서 해시 비트가 가득 찼을 때까지 가상 해시 테이블의 확장이 가능하다. 만약 해시 비트를 참조하는 비트가 모두 사용되어진 경우에는 기존의 EXT2와 같이 단말 리프 노드에 링크드 리스트 형태로 연결시킨다. 이를 통해 같은 디렉토리 내 모든 아이노드 객체인 파일, 디렉토리, 링크 등과 같은 것을 무한대로 저장할 수 있게 되는 것이다. 한 예로 실제 구현을 통해 얻은 결과로는 한 디렉토리 밑에 계속해서 서브 디렉토리를 만든다고 가정하면 한 블록 당 150가량의 디렉토리 엔트리가 들어갈 수 있으며 32비트 해시함수를 사용하므로 232 X 150 = 6천8백억개 정도의 엔트리를 모두 채우게 되면 링크드 리스트 형태로 전환된다. 이를 해결하기 위해서는 64비트 해시함수를 사용하기 위한 연구가 선행되어야 한다.

3.5 일관성 유지 기법

공유디스크 환경 하에서의 분산 파일시스템은 파일 시스템 자체적으로 모든 디스크를 공유하므로 일관성 유지가 필요하며 일관성을 유지하는 기본단위는 파일시스템의 모든 메타 데이타는 블록단위로 저장되므로 블록을 기본 단위로 한다. 블록 단위의 일관성 유지를 위해 VFS에 저장되어 있는 캐쉬와 호스트 별로 변경된 데이타에 대한 여러 기법이 존재해야 한다. 일반적인 버퍼의 coherence를 맞추기 위한 초기의 방법은 버퍼를 읽은 후 모든 버퍼를 Invalidate하는 방법이며 이는 동기화를 맞출 수는 있지만 버퍼링을 통한 I/O 오버헤드를 줄일 수 없게 된다. 기존 연구의 GFS같은 경우에는 디스크를 통해 일관성을 유지한다. 다른 호스트가 변경한 버퍼를 자신이 가져오기 위해서 디스크에 쓰게 하고 자신은 디스크에서 읽는 방법이다. 이 방법도 역시 I/O 오버헤드가 존재하게 된다.

제안하는 파일 시스템은 이 문제를 해결하기 위해 각 호스트에 버퍼를 추적하면서 잠금 메시지의 버전 넘버를 관리하여 버퍼를 직접 전송 받은 전역 버퍼 기법을 채택하였다. 구체적인 알고리즘은 그림 12와 같다.

먼저 특정 호스트가 요청하는 잠금이 SHARED나 EXCLUSIVE에 따라 나누어지며, 한 예로 SHARED인 경우에는 자기 자신이 기존의 SHARED 잠금을 가지고 있으며 재사용을 하며 다른 호스트가 가져간 경우에는 SHARED로 가져간 경우와 EXCLUSIVE로 각각 나누어 볼 수 있다. 먼저 SHARED로 가져간 경우에는 트

```
if( SHARED ) // 버퍼를 읽기 위해 개별 호스트의 잠금 요청이 SHARED인 경우
{
    if( Hold the SHARED LOCK )      // 자기 자신이 SHARED LOCK을 가지고 있는 경우
    {   // 이 경우는 이전에 SHARED LOCK을 얻어온 경우이며 아울러 다른 호스트에게 CALLBACK을 통해 뺏기지 않은 상태이다.
        Read the local host's buffer // 자기 자신이 가지고 있는 버퍼를 재 사용한다.
    }
    else                            // 자기 자신이 SHARED LOCK을 가지고 있지 않은 경우
    {                               // 이 경우는 이전에 잠금 요청이 없었거나 다른 호스트에 가져간 상태이다.
        Request the SHARED LOCK  // 전역 잠금 서버에 잠금을 요청한다.
        if( NEW LOCK )              // 이전에 어떠한 호스트가 요청하는 블록의 잠금을 한번도 요청하지 않은 경우
        {
            Store the LOCK information   // 전역 잠금 서버에 정보를 저장하고 버퍼를 읽기를 허용한다
            Read the local host's buffer  // 자기 자신이 버퍼를 처음 읽어서 사용한다.
        }
        else                        // 다른 호스트가 SHARED LOCK이나 EXCLUSIVE LOCK을 가져간 경우
        {
            if( Other hosts hold the SHARED LOCK)  // 다른 호스트가 가져간 게 SHARED LOCK인 경우
            {
                // 가져간 호스트에게 CALLBACK을 통해 요청한 호스트에게 버퍼를 직접 전송하라는 메시지를 보낸다.
                Callback request the holden host
                The holden host send the buffer to the requested host
            }
            else                                // 다른 호스트가 가져간 게 EXCLUSIVE LOCK인 경우
            {
                // 가져간 호스트에게 CALLBACK을 통해 가져간 호스트가 트랜잭션을 종료한 후 요청한 호스트에게 버퍼를
                // 직접 전송하라는 메시지를 보낸다.
                Callback request the holden host and waits for ending the transaction
                The holden host send the buffer to the requested host
else          // 개별 호스트의 잠금 요청이 EXCLUSIVE인 경우
{
    if( Hold the EXCLUSIVE LOCK )   // 자기 자신이 EXCLUSIVE LOCK을 가지고 있는 경우
    { // 이 경우는 이전에 EXCLUSIVE LOCK을 얻어온 경우이며 아울러 다른 호스트에게 CALLBACK을 통해 뺏기지 않은 상태이다.
        Read the local host's buffer   // 자기 자신이 가지고 있는 버퍼를 재 사용한다.
    }
    else                            // 자기 자신이 EXCLUSIVE LOCK을 가지고 있지 않은 경우
    {                               // 이 경우는 이전에 잠금 요청이 없었거나 다른 호스트에 가져간 상태이다.

        Request the EXCLUSIVE LOCK // 전역 잠금 서버에 잠금을 요청한다.
        if ( NEW_LOCK )             // 이전에 어떠한 호스트가 요청하는 블록의 잠금을 한번도 요청하지 않은 경우
        {
            Store the LOCK information    // 전역 잠금 서버에 정보를 저장하고 버퍼를 사용을 허용한다
            Read the local host's buffer   // 자기 자신이 버퍼를 처음 읽어서 사용한다.
        }
        else                        // 다른 호스트가 가져간 경우
        {
            // 가져간 호스트에게 CALLBACK을 통해 가져간 호스트가 트랜잭션을 종료한 후 요청한 호스트에게 버퍼를
            // 직접 전송하라는 메시지를 보낸다.
            Callback request the holden host and waits for ending the transaction
            The holden host send the buffer to the requested host
        }
    }
}
```

그림 12 메타 데이타 일관성 유지 알고리즘

랜잭션의 종료여부와 관계없이 바로 잠금 요청을 수락할 수 있으며 EXCLUSIVE로 가져간 경우에는 트랜잭션의 종료를 기다려야 한다. 잠금 요청을 수락한 후 CALLBACK요청을 받은 호스트는 요청한 호스트에게 수락 잠금 메시지와 함께 버퍼 데이타를 전송하게 된다. 이는 한번의 통신 오버헤드를 통해 잠금 수락 메시지와 버퍼 데이타를 한번에 보내게 된다. 이를 통해 약간의 통신 오버헤드를 줄일 수 있다.

한 예로 그림 13에서 SAN fabric에 3개의 호스트가 존재하며 하나의 호스트는 전역 잠금 관리자 역할을 하고 두개의 호스트는 로컬 잠금 관리자 역할을 한다고 가정한다.

그림 13 메타 데이타 일관성 유지의 예

지역 잠금 서버 1이 특정 블록에 저장된 A라는 데이타를 변경하고자 한다. ①에서와 같이 전역 잠금 서버에게 쓰기 요청을 하면, 전역 잠금 서버는 자신이 보유하고 있는 리스트를 뒤져서 없는 경우 ②와 같이 쓰기 요청을 허락한다. 지역 잠금 서버 1은 읽어온 데이타 A를 ③과 같이 변경한다. 이때 ④와 같이 지역 잠금 서버 2가 특정 블록 A가 저장된 데이타를 읽으려고 하면, 먼저 전역 잠금 서버에게 요청한다. 전역 잠금 서버는 자신이 보유한 리스트에 지역 잠금 서버 1이 데이타를 가지고 있는 것을 알고 지역 잠금 서버1에게 ⑤와 같이콜백 요청(Callback Request)을 한다. 이를 통해 지역 잠금 서버 1에게 버퍼를 지역 잠금 서버 2에게 전달할 것을 요청한다. 이를 통해 지역 잠금 서버1은 변경을 마치고 지역 잠금 서버2에게 디스크 I/O없이 바로 버퍼 데이타를 전달하게 된다. 이러한 전역 버퍼 기법을 이용하여 GFS에서와 같이 디스크에 데이타를 쓰고 다른 호스트가 디스크에서 읽는 구조보다 효율적인 구조를 제시할 수 있다.

4. 리눅스 커널 상에서의 공유 디스크 파일 시스템 구현

제안하는 파일시스템은 공유디스크 환경에서 리눅스 커널에 구현하였다. 커널 버전은 2.2대 커널을 사용하였으며 특히 Qlogic사에서 제공하는 HBA(Host Bus Adapter)가 가장 안정적인 2.2.18 커널을 기본으로 사용하였다. 파일 시스템의 레이아웃을 위해 리눅스에서는 기본적으로 파일 시스템마다 mkfs(make file system)

```
uint64 sfs_get_dev_size(commandline_t *command)
{
    uint64 device_size;
    device_size = ioctl(command->fd, BLKGETSIZE, &device_size); // 디바이스의 크기를 리턴한다.
    return device_size;
}
void sfs_create_filesystem(commandline_t *command)
{
    sfs_comput_filesystem_layout(command); // 디바이스의 사이즈를 통해 각각 블록의 위치를 계산한다.
    write_superblock(command,1);           // 첫 번째 수퍼블록에 정보를 기록한다.
    write_root(command);                   // 루트 블록에 대한 정보를 기록한다.
    write_superblock(command,2);           // Mirrored 수퍼블록에 정보를 기록한다.
    write_extent_bitmap(command);          // 비트맵 블록을 초기화한다.
    write_journal(command);                // 저널공간을 초기화한다.
}
int main(int argc, char * argv[])
{
      commandline_t command;                         // 사용자환경에서 mksfs를 위한 옵션 설정을 위한 구조체
      parse_arguments(argc,argv,&command);           // 옵션 파싱함수
      command.fd = open64(command.device_name, O_RDWR | O_LARGEFILE | O_SYNC);
      command.fs_size = sfs_get_dev_size(&command);  // RAW디바이스의 사이즈를 얻는다
      sfs_create_filesystem(&command);
      close(command.fd);
}
```

그림 14 mksfs(SANtopia make file system utility)

```
// LPk : k번째 호스트에 대한 로컬 Pivot
// GPk : k번째 호스트에 대한 전역 Pivot
// OPB : 한 블록이 가질수 있는 할당 블록의 개수
//   M : 한번의 비트맵 조정으로 얻을 수 있는 블록의 개수

if( LPk == GPk * OPB ) // 만약 호스트 K의 저장공간이 꽉 찬 경우
{
    Read the bitmap index block          // 비트맵 인덱스를 읽는다.

    Set the LPk, GPk, OPB, M             // LPk, GPk, OPB,M의 값을 설정한다.

    Check the empty bitmap      // 현재 호스트보다 아이디가 하나 적은 호스트에 대해서 공간이 있는지 여부를 확인한다.

    // 맨 처음과 맨 마지막 호스트는 라운드-로빈(Round-Robin)형태로 호스트 아이디를 얻는다.
    if( LPk-1 < ( GPk - M ) * OPB ) ........................................ ①
    {
        GPk-1 = GPk-1 - M // 비트맵 인덱스를 업데이트 한다. 관련된 인접호스트에 갱신한 정보를 Revalidate한다.

        return block number // 각각의 호스트에 저장된 현재 비트맵할당 정보를 통해 블록 넘버를 리턴한다.
    }
    // 인접한 호스트에서 여유분의 비트맵 블록을 얻어 올 수 없는 경우 전역 관리로 전환된다.
    else                        ........................................ ②
    {
        Check the nearest bitmap victim // 1. 가장 인접한 호스트에서 체크한다.

        Obtain the host's bitmap which has large amount of empty bitmap  // 2. 가장 많이 남은 호스트에서 얻어온다.

        return block number     // 각각의 호스트에 저장된 현재 비트맵할당 정보를 통해 블록 넘버를 리턴한다.
    }
}
else                // 일반적인 비트맵 할당이 가능한 경우    .................. ③
{
    return block number     // 각각의 호스트에 저장된 현재 비트맵할당 정보를 통해 블록 넘버를 리턴한다.
}
}
```

그림 15 비트맵 할당 알고리즘

utility를 제공하여야 하며 이를 위해 mksfs(make SANtopia file system)을 구현하였다. 구현 방법은 HBA에서 제공하는 드라이버를 모듈형태로 올리면 FC 디스크가 일반 호스트에서 보이며 이 디바이스 파일을 open64() 함수를 통해 64비트 모드로 열어서 ioctl()함수를 이용하여 디바이스 사이즈를 얻는다. 얻은 디바이스 사이즈를 통해 디스크 레이아웃의 위치를 각각 계산하여 순차적으로 수퍼블록, 루트블록, 비트맵블록들, 저널공간을 초기화한다. 개략적인 함수의 흐름은 아래 그림과 같다.

다음으로 실제 파일 시스템을 위한 모듈을 제작하여 커널 상에 모듈 형태로 올린다. 구현한 파일시스템 모듈은 크게 파일시스템의 메타데이타 구조를 다루기 위한 파일, 디렉토리, 아이노드, 비트맵, 슈퍼블록에 관련된 파일들로 구성되어 있으며, 일관성 유지를 위해서 전역 잠금, 로컬 잠금, 통신에 대한 파일이 존재한다. 아울러 저널링을 위한 트랜잭션, 로그, 회복 관련 코드 또한 존재한다.

먼저 비트맵 관리 모듈은 그림 15와 같은 알고리즘에 기반하여 구현되었다. 그림에서 sfs_blk_alloc()함수는 new_state를 통해 메타데이타와 일반 파일 데이타에 대해 서로 다른 방식으로 할당 정책을 취하게 되며 ①에서와 같이 Local Pivot과 Global Pivot을 통해 각각의 호스트에서의 저장 공간을 체크하게 된다. 여기서 M변수는 비트맵 블록을 이웃 호스트에게 넘겨줄 수 있는 임계값을 의미하고 OPB는 비트맵 한 블록이 가리키는 할당 블록의 양을 가리킨다. 만약 ②와 같이 저장공간이 부족한 경우 인접한 호스트에서 얻고, 만약 인접한 호스트에도 저장 영역이 모자라는 경우는 가장 많이 남은 호스트에서 얻어오는 정책을 취한다. 여기에서 사용되는 통계 정보는 비트맵 인덱스에서 얻어오게 된다. ③에서는 호스트에 여유 분의 비트맵 블록이 많은 경우의 일반적인 루틴이다.

제안하는 파일 시스템의 아이노드 구조는 아래 그림 16과 같다. 아이노드는 일반적으로 파일, 디렉토리, 링크 등과 같은 파일 시스템의 모든 객체에 공통적으로 사용되는 구조체 정보이며, 그림에서 ①은 저널링을 위해 필요한 메타 데이타 버전 정보를 나타내는 구조체이다. 버전 정보는 변경될 때마다 1씩 증가하는 버전정보 변수를 가지고 있으며 시스템 복구 시 사용되어진다. 이를

```
// 아이노드의 디스크 구조 정보
struct sfs_sinode {
    // 저널링을 위한 메타데이타 버전 정보 구조체                ........................................ ①
    sfs_meta_header_t si_header;
    // POSIX에서 규정한 일반적인 UFS에서 제공되어야 할 필드들 ........................................ ②
    uint32 si_mode;               // 파일 모드
    uint32 si_uid;                // 사용자 ID
    uint32 si_gid;                // 그룹 ID
    ....
    // 세미플랫과 가상 매핑을 위해 추가되는 필드들              ........................................ ③
    uint16 si_height;             // 세미플랫 구조를 위한 Height필드 저장
    uint32 si_entries;            // 디렉토리 엔트리를 위한 엔트리 개수 저장
    uint16 semi_flat_flag;        // 세미플랫 플래그
    uint16 semi_flat_offset;      // 세미플랫 오프셋
    uint64 prev_virtual_blkno;    // 가상매핑을 위한 논리적 블록 넘버
    uint64 prev_virtual_blknr;    // 가상매핑을 위한 물리적 블록 넘버
};
typedef struct sfs_sinode sfs_sinode_t;

// 메모리에 올려져 있는 아이노드의 메모리 구조 정보            ........................................ ④
struct sfs_sinode_info {
    uint64 ssi_num;               // 블록넘버
    atomic_t ssi_count;           // 아이노드 사용 카운트 변수
    structu inode *ssi_vno;       // VFS에서 관리하는 아이노드 포인터
    sfs_sbi_t      *ssi_sb;       // 제안하는 파일시스템의 수퍼블록 포인터
    sfs_sinode_t  ssi_sinode;     // 아이노드 디스크 구조 정보
    struct semaphore ssi_flock;   // 파일락을 위한 세마포 변수
};
typedef struct sfs_sinode_info sfs_ssi_t;
```

그림 16 아이노드 구조체 정보

```
int64 sfs_write_data(sfs_ssi_t *sip,      // 아이노드 구조체 포인터
                     const char *buf,     // 쓸려고 하는 파일 데이타 포인터
                     uint64 offset,       // 파일의 오프셋
                     uint64 size)         // 쓸려고 하는 파일의 크기
{
   // bsize          : 블록 사이즈  ( 디폴트 : 4K바이트)
   // extent_factor : 익스텐트 계수 ( 디폴트 : 16, 파일 데이타에 대한 블록 할당은 64K가 된다. )
   // meta_bh        : 메타데이타에 대한 버퍼 구조체 포인터
   // real_blkno     : 파일의 논리적인 실제 블록 넘버
   // real_blknr     : 시스템에 전달되는 실제 물리적인 블록 넘버
   // virtual_blkno  : 가상맵핑을 위한 블록 넘버
   // virtual_blknr  : 가상맵핑을 통해 얻은 실제 논리적인 블록 넘버

   real_blkno    = offset / bsize;              // 실제 블록넘버 = 파일의 오프셋 / 블록 사이즈 ................ 1
   virtual_blkno = real_blkno / extent_factor;  // 가상 블록넘버 = 실제 블록넘버 / 익스텐트 계수 ............ 2

   if( real_blkno % extent_factor == 0 )        // 리얼블록 맵핑 ( I/O가 발생한다 )................................ 3
   {
       error = sfs_write_data_block(sip.meta_bh, virtual_blkno, &virtual_blknr); // 맵핑함수 호출
       prev_virtual_blkno  = virtual_blkno;
       prev_virtual_blknr   = virtual_blknr;    // 이전 가상블록넘버를 저장한다.
                                                // 다음에 연속된 블록에 대한 쓰기 요구가 있을 때
                                                // 맵핑을 통해 간접블록에 접근없이 바로 쓸수 있게 한다.
                                                // extent_factor가 16일 경우 처음 한 블록에 대한
                                                // 간접블록 I/O만 발생하고 나머지 15개 블록은 가상맵핑을
                                                // 통해 해결할 수 있다.
   }
   else                                         // 가상 블록 맵핑                ............................ 4
   {
       if( prev_virtual_blkno == virtual_blkno ) // 가상블록 맵핑 HIT(간접블록에 대한 I/O가 없음)
       {
           virtual_blknr = prev_virtual_blknr;   // 루트 블록에 저장된 블록 넘버 할당 ........................ 5
       }
       else
       {
           error = sfs_read_data_block(sip.meta_bh, virtual_blkno, &virtual_blknr); // 맵핑 함수 호출 ..... 6
       }
   }

   real_blknr = virtual_blknr + real_blkno % extent_factor; // 리얼 블록 넘버 설정  ............................ 7

   return real_blknr;                                       // 리얼 블록 넘버 리턴
}
```

그림 17 익스텐트 기반의 가상 맵핑 함수

통해 로그 영역에 저장된 메타 데이타와 일반 디스크에 저장된 메타 데이타의 일관성을 보장하게 되는 것이다. ②에서 사용되어지는 변수들은 일반적으로 POSIX에서 규정한 필드들이며 이것들은 UFS나 리눅스 파일 시스템에 공통적으로 정의되는 것들이다.[9] ③에서 사용되어 지는 필드는 제안하는 파일시스템의 고유정보를 나타내는 것이며 세미 플랫 구조를 위한 height, entry등과 같은 구조 정보와 저장 위치를 찾기 위한 offset과 이전 블록 넘버들이 존재한다. 그림 16의 마지막 아이노드의 메모리 구조 정보는 제안하는 파일 시스템의 디스크 아이노드 구조를 메모리에 올라 왔을 때의 구조이다. 먼저 64비트의 블록 넘버를 저장하는 변수와 아이노드가 참조되는 카운트 변수 VFS에서 관리하는 아이노드 포인터, 그리고 제안하는 파일 시스템의 디스크 이미지인 구조 정보 등이 존재하게 된다.

이러한 아이노드 구조 정보를 통해 이루어지는 세미 플랫 구조의 가상 맵핑에 대한 대략적인 알고리즘은 그림 17과 같다. 파일의 데이타를 쓰는 sfs_write_data() 함수의 ①에서와 같이 파일의 실제 오프셋을 블록 사이즈로 나누면 실제 논리적인 블록 넘버를 구할 수 있다. 결국 파일의 논리적인 블록 넘버에 데이타를 쓰는 것이다. 이러한 논리적인 블록 넘버를 다시 가상 블록넘버로 바꾸기 위해 익스텐트 계수로 나누는 과정은 ②의 과정이다. 이렇게 구한 블록 넘버들을 가지고 ③과 ④의 과정에서 실제 데이타를 쓰기 위한 과정을 거치게 된다. ③의 경우에는 리얼 블록 맵핑이 이루어져야 하는 것으로 실제 맵핑 함수를 호출하고 이전 가상 넘버의 구한 블록 넘버들을 같이 저장한다. 이후에 연속된 (익스텐트

```
int sfs_dir_e_add(sfs_ssi_t *sip,          // 아이노드 구조체 포인터
                  struct qstr* filename,  // 디렉토리 엔트리에 대한 이름 구조체
                  uint64 ino,             // 삽입하고자 하는 아이노드의 블록 넘버
                  unsigned int type)      // 디렉토리 엔트리 타입(파일, 디렉토리, 심볼릭 링크, 하드링크,...)
{
grow_restart:            // 확장되는 경우 해쉬값의 한 비트가 더 적용되므로 다시 계산한다.   ..................... ①
    hsize = 1 << sip->ssi_sinode.si_depth;
    hash = sfs_dir_hash(filename);
    index = hash >> ( 32 - sip->ssi_sinode.si_depth );
split_restart:           // 분할이 일어나는 경우 분할이 적용되어 리프블록 넘버를 얻는다.   ..................... ②
    error = sfs_get_leaf_nr(sip,index,&leaf_no);   // 파일이름에 대한 해쉬 함수를 통해 인덱스 값을 얻어서 블록넘버를 가져온다
    while(1)
    {
        leaf_bh = (sfs_buf_t *)sfs_metabread(sp,leaf_no,leaf_gl);   // 리프블록을 버퍼에 읽어온다.
        ret = sfs_dentry_is_remained(sip,leaf_bh);                   // 리프블록에 엔트리가 삽입가능한지 확인한다.
        if( ret == FALSE )     // 리프 블록에 엔트리의 삽입이 가능하지 않은 경우, 2단계 확장해쉬에 적용을 받는다.
        {                                                                  .......................................③
            // 리프블록의 깊이가 전역 깊이보다 작은 경우
            if( leaf->lf_depth < sip->ssi_sinode.si_depth )      ........................................................⑤
            {
                sfs_split_leaf(sip, index, leaf_no);
                goto split_restart;
            }
            else if( sip->ssi_sinode.si_depth < 32 ) //해쉬 값은 최대 32비트를 사용한다.   ........................ ⑥
            {
                sfs_grow_leaf(sip);
                goto grow_restart;
            }
            else if( leaf->lf_next )    // 해쉬값의 32비트가 모두 찬 경우 링크드 리스트로 전환한다. .............⑦
            {
                leaf_no = leaf->lf_next;
                continue;          // 다음 링크드 블록 번호를 가지고 while(1)에서 시작하여 다음 리프블록을 얻는다.
            }
            else                          // 처음 링크드 리스트로 전환할 때 다음 블록의 할당이 요구된다. ..........⑧
            {
                 leaf->lf_next = 새로운 메타데이타 블록을 할당받아 저장한다.
            }
        }
leaf_buffer_setting:                                                                 ..........................④
        // 리프블록에 엔트리를 삽입한다.
root_buffer_setting:
        // 아이노드의 루트블록에 메타데이타(전체 엔트리 개수, 액세스한 시간, 블록수, 링크수) 등에 대한 변경 또한 일어난다.
      return 0;
    }  // End of While(1)
}
```

그림 18 2단계 확장해쉬 디렉토리 삽입 함수

계수 - 1)의 블록에 대한 쓰기 요청이 왔을 때 간접블록에 대한 접근 없이 가상 블록 맵핑에 적용을 받으므로 바로 쓰고자 하는 데이타를 저장할 블록을 바로 얻을 수 있는 것이다. ⑤와 ⑥의 과정에서는 가상 블록 맵핑이 적용되는 것이며 임의의 블록을 랜덤 하게 쓰는 경우에는 ⑥의 과정처럼 읽는 과정이 필요하게 된다. 마지막으로 구해준 가상 블록 넘버를 가지고 다시 논리적 블록 넘버를 재연하는 과정이 ⑦과 같다.

마지막으로 세미 플랫 구조의 가상 맵핑을 가상 해시 테이블에 적용한 디렉토리 관리 부분 중에서 대표적인 디렉토리 엔트리 삽입 함수는 아래 그림 18과 같다.

sfs_dir_e_add()함수는 일반적인 stuffed방식에서의 함수(sfs_dir_l_add())에서 확장되어 확장 해싱에 적용을 받는 디렉토리 엔트리 삽입을 위한 함수이다. 먼저 그림 18의 ①에서 확장이 이루어졌을 때 해시 인덱스의 값이 바뀌는 과정을 나타낸 것이다. ⑤의 과정을 체크하여 리프의 지역 깊이와 루트의 전역 깊이가 같은 경우 더 이상 분할이 이루어질 수 없으므로 확장 루틴(sfs_grow_leaf)을 통해 확장이 이루어진 후 다시 ①의 과정부터 재 시작한다.

기존의 리프 블록의 지역 깊이가 전역 깊이보다 작은 경우는 두 개 이상의 블록 주소가 같은 리프를 가리키는 경우이며, 이 경우에는 분할이 일어날 수 있다. 그림에서 ⑤의 과정에 만족하여 분할루틴(sfs_split_leaf)을 호출하고 다시 ②의 과정부터 반복하여 수행한다. 해시 인덱스를 32비트의 값을 사용하므로 최대 232개의 리프 블록을 가질 수 있는 구조이다. 해시 함수가 모든 리프 블록에 균형적인 디렉토리 엔트리를 배분한다고 가정하며, 한 리프 블록에 블록 사이즈를 4K로 하고 대략적인 엔트리 사이즈가 약 30바이트 정도 되므로 한 리프 블록에는 150개정도의 디렉토리 엔트리가 삽입 가능하다. 따라서 32비트의 값을 사용하는 해시 인덱스에서는 232 * 150 = 6천억개 정도의 엔트리가 삽입되어 질 수 있다. 만약 이보다 더 많은 엔트리가 삽입 될 경우 리프 블록의 lf_next필드에 링크드 리스트 형태로 엔트리를 삽입하는 형태를 취한다. 링크드 리스트에서 처음 엔트리 블록을 할당하는 과정이 ⑧의 과정이며, 기존의 링크드 리스트로 연결된 리프 블록을 순차적으로 탐색하는 과정이 ⑦의 과정이 된다. 마지막으로 삽입할 위치를 정확히 찾은 경우에 리프 블록에 실제 엔트리를 삽입하는 과정과 루트블록에 변경 정보를 갱신하는 과정은 ④이다. 정상적인 엔트리의 삽입 위치를 못 찾은 경우의 과정은 위와 같으며, 엔트리의 삽입 위치를 해싱 구조의 변경 없이 바로 찾은 경우는 바로 ④의 과정이 된다.

5. 성능평가

제안하는 파일시스템의 실험환경은 표 1과 같다. FC Switch[6]는 Brocade Inc의 16포트 스위치인 *Silkworm 2800*[11]을 사용하였으며 디스크 레이어에는 RAID, JBOD를 합하여 1TB의 디스크 공간을 사용하였다. 구현된 환경은 레드햇 리눅스 커널 2.2.18버전이며 4개의 서버를 연결하여 클러스트 노드로 사용하였다. 실험에서 측정한 계수는 파일연산에 대한 것으로 국한하였으며, 파일의 쓰기, 읽기 와 순차적, 임의적 접근방법을 사용하였다. 또한 파일의 사이즈는 10M,20M,30M, ...,500M의 데이타에 대해 적용하였으며 비교한 파일시스템은 리눅스의 대표적인 로컬 파일시스템인 EXT2, Sistina 사의 공유디스크환경 파일시스템인 GFS와 비교하였다[1,2]. SFS에서는 각기 저널링 기능의 ON/OFF기능, 로컬환경에서 사용할 수 있는 NO LOCK 옵션이 제공되며 이를 통해 EXT2와는 NO LOCK 옵션에 저널링 기능을 OFF하여 비교하였고 GFS에서는 저널링 기능을 ON하고 하나의 서버에서 수행하도록 NO LOCK옵션으로 수행하였다.

표 1 실험 환경

FC Switch	Brocade Silkworm 2800 (16 Ports)
FC Disks	RAID(Eurologic), JBOD(Voyager)
NIC Card	Qlogic
Node	Compaq Proliant
Node Memory	512M
O.S.	Redhat Linux 6.2
Kernel Version	2.2.18
Block Size	4K(4096Bytes)

먼저, EXT2, SFS와의 파일에 대한 읽기, 쓰기 연산의 평균응답시간을 그림 19와 같다. 데이타의 양은 10M -500M의 데이타를 보이며, 10M부터 순차적으로 500M까지 파일을 만들었을 때의 시간을 보인 것이다.

측정한 응답시간의 결과는 파일에 대한 읽기, 쓰기 연산을 각각의 데이타 크기에 따라 수행하고 평균을 낸 것이다. 한 예로, 10M 파일에 대해 각각의 파일 시스템 별로 쓰기 연산을 수행하고, 시스템을 재시작 하여 각각의 파일 시스템 별로 읽기 연산을 수행하였다. 그 이유는 리눅스의 버퍼 정책에서는 파일 시스템 별로 블록에 대해 정책이 이루어 지는게 아니라 시스템에 있는 모든 디바이스에 대해 일괄적으로 LRU정책이 적용된다. 따

라서 쓰기 연산에 따라 변경된 버퍼의 양과 디스크에 반영되는 시간에 따라 읽기 연산이 영향을 받기 때문이다. 시스템의 재시작을 하지 않고 하기 위해서는 각각의 디바이스에 적은 블록의 버퍼에 대해 모두 Invalidate하는 과정이 필요하기 때문이다. 또한 SFS의 익스텐트 개수는 16을 디폴트 값으로 사용하여 블록사이즈가 4K인 경우이므로 기본적으로 64K의 익스텐트를 사용한 것이다. 측정한 응답시간은 EXT2와 제안하는 SFS 파일시스템의 응답시간이 비슷한 것으로 나왔으며 EXT2는 저널링을 위한 추가적인 I/O가 없기 때문에 응답시간은 SFS와 비슷한 수준으로 나왔다. SFS에서는 비동기 적인 저널링 기법을 사용한다. 실험에 사용한 FC Disk는 Seagate Cheetah 10,000 RPM Ultra 160[12] 이며 일반적인 SCSI 디스크보다 성능이 우수한 디스크이다. 실험 결과 500M정도의 파일을 만드는데 두 파일시스템 모두 24초정도의 시간이 소요되는 것으로 나왔다.

다음으로, 제안하는 파일시스템과 가장 유사한 공유디스크 환경의 GFS와의 평균 응답시간을 보인 것이다.

그림 19의 오른쪽 그래프에서와 같이 GFS보다 SFS의 성능이 우수한 것으로 나왔다. 그 이유는 GFS에 비해 리소스 그룹 운영의 오버헤드가 없으며 SFS의 경우 익스텐트 기반 세미 플랫 구조를 채택하여 간접 노드에 대한 불필요한 생성 또한 탐색이 적기 때문이다. 평균적으로 1 2초 정도의 파일 연산에 대한 응답시간 감소를 보인다. 일반적으로 SCSI디스크의 I/O보다 FC 디스크에 대한 I/O속도가 빠르기 때문이 이는 상당한 I/O감소를 보인 것이다.

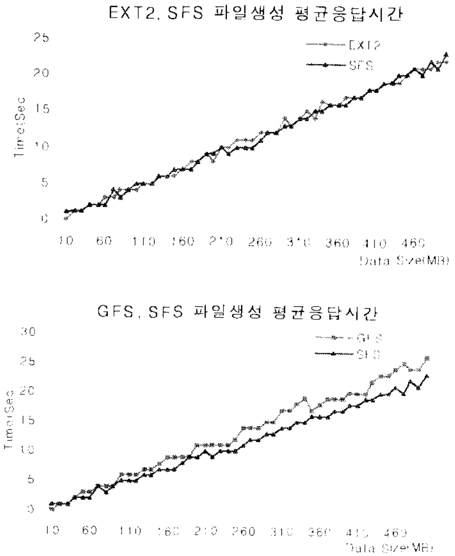


그림 19 파일 생성에 따른 평균 응답 시간

다음으로, GFS와 제안하는 파일 시스템의 실제 파일 쓰기 연산에 대한 I/O분포를 측정하기 위한 방법은 커널을 수정하여 파일 시스템이 마운트한 특정 디바이스의 I/O횟수의 누적 합계를 통해 얻을 수 있다. 리눅스 시스템에서 I/O횟수를 측정하기 위해서는 하위레벨의 Request Queue에서 디바이스에 쓰는 이벤트 함수에서

```
void make_request(int major,int rw,struct buffer_head *bh)   // 리눅스 시스템에서 Request처리 함수
{

    // 한 예로 FC Disk의 첫 번째 것인 /dev/sda를 사용하는 경우                ..................... ①
      int MY_DEVICE_MAJOR = 8;    // SCSI디스크 인 경우 리눅스에서는 8번이다.
      int MY_DEVICE_MINOR = 0;    // SCSI디스크 맨 처음 디바이스의 마이너는 0번이다.
    .....
    switch(rw)
    {
          case READ:
               ...
          case WRITE:

                 if( MY_DEVICE_MAJOR == major && MY_DEVICE_MINOR == MINOR(bh->b_dev) ) ... ②
                 {
                      my_device_write_counter++;
                 }
                 break;
    }
    ...
}
```

그림 20 리눅스 커널 상에서 I/O측정을 위한 커널 수정 부분

누적 카운트를 두고 커널을 재 컴파일 하는 방법을 취하였다. 각각의 데이타에 대해 쓰기 연산이 수행된 후에 시스템 로그파일에 적힌 누적 카운트를 계산하였다. 구체적인 함수에서의 누적 카운트 계산 방법은 아래 그림과 같다. 리눅스 시스템에서는 모든 디바이스에 대한 쓰기 연산은 ll_rw_block()함수를 통해 이루어진다. 이 함수의 하위 함수인 실제 디스크에 쓰기 위해 큐를 만드는 make_request()함수가 존재한다. 이 make_request 함수는 모든 디바이스에 공통적으로 적용되는 함수이기 때문에 특정 디바이스에 대한 I/O합계를 구하기 위해서는 새로운 누적 카운트를 위해 커널 재컴파일 과정을 거치게 된다. 그림 20의 점선 부분의 ①과 ②의 과정이 커널을 수정한 부분이다. 먼저 ①에서는 FC 디스크가 첫 번째 디스크를 사용한 경우(/dev/sda) 디바이스의 MAJOR번호와 MINOR번호를 /proc/partitions파일에서 얻어서 그 번호를 소스 상에 구현하고, 쓰기 연산에만 적용되므로 ②에서와 같이 WRITE case문에 MAJOR와 MINOR가 같은 경우에만 카운트한다.

이러한 커널 수정을 통해 측정한 GFS와 제안하는 파일시스템의 I/O 누적 합계는 그림 21과 같다.

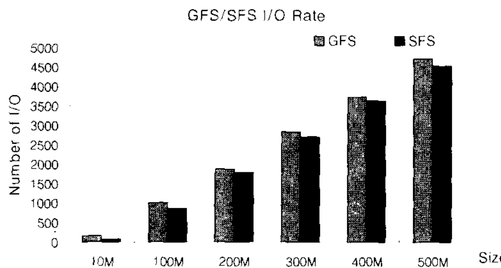


그림 21 GFS/SFS의 파일 생성에 따른 평균 응답 시간

그림에서 전반적으로 SFS의 I/O성능이 GFS보다 우수하다. 그 이유는 SFS의 경우 간접 블록 생성에 따른 오버헤드가 감소하며, 익스텐트 기반의 벌크 할당으로 비트맵 블록의 변경의 수도 감소하기 때문이다.

마지막으로, 그림 22는 각각의 파일 시스템에서 디렉토리 생성에 따른 평균 응답시간이다. 이것을 디렉토리 엔트리의 삽입에 대한 것으로 국한하였다. EXT2와 SFS에서의 디렉토리 생성 시간은 EXT2의 링크드 리스크 구조에 따른 문제점을 해결한 것을 보여 준다. EXT2의 경우 처음 2000개 이하의 엔트리에 대해서면 약간의 성능의 우수성을 보이며, 전반적으로 엔트리의 수가 증가하면서 성능이 급속도로 저하되는 것을 볼 수 있다. 또한, GFS와 SFS의 성능 비교에서는 SFS가 세미 플랫 구조를 해시 테이블로 사용하고 있으므로, 간접 블록에 대한 I/O 오버헤드가 적은 것을 보여 준다. 이것을 파일의 경우와 마찬가지로 디스크 I/O의 우수성을 보여 준다.

6. 결 론

컴퓨팅 파워의 향상으로 데이타에 대한 요구가 급증하면서 대용량 스토리지에 대한 연구가 활발히 진행되어 왔다. 이러한 스토리지에 대한 연구에 따라 기존에 제시된 스토리지 저장구조에도 많은 변화를 일으키고 있으며 이에 본 논문에서는 공유디스크 환경에서의 효율적인 분산 공유 파일시스템을 제안하였다. 제시한 분산 공유 파일시스템은 공유 디스크 환경에 적합하게 비트맵을 분리 관리하는 정책을 수행하며, 디스크 접근 시간을 최소화하기 위해 데이타 블록을 그룹화 하였고, 파일시스템 고유의 저장 구조인 아이노드를 동적 세미플랫 구조로 향상하여 불필요한 간접 I/O 오버헤드를 줄였으며 대용량 파일에 적합한 형태를 가지게 하였다. 또한 디렉토리 엔트리의 관리 구조를 효율적인 확장해싱으로 변형하였으며 저널링 기능을 지원하는 파일시스템으로 확장하였다. 추후 연구과제로는 이 기종 간의 효율적인 스토리지 관리를 위한 파일시스템 확장이 필요하다.

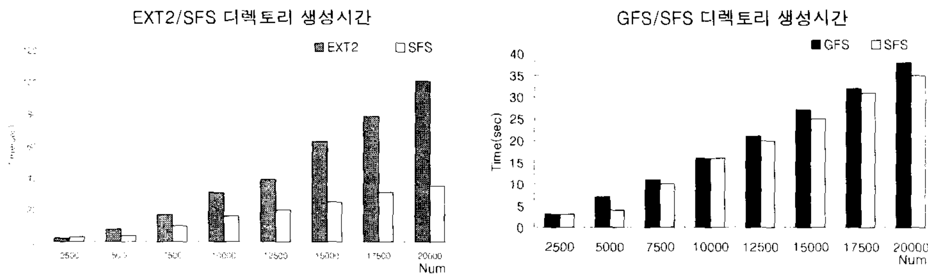


그림 22 EXT2/GFS/SFS의 디렉토리 생성 평균 응답시간

참 고 문 헌

[1] R. Card, "Design and Implementation of the Second Extended File system", Proceedings of the First Dutch International Symposium on Linux, 1995.

[2] Steven R. Soltis, Thomas M. Ruwart and Matthew T.O Keefe, "The Global File System", Proceedings of the 5th NASA Goddard Conf. On Mass Storage Systems and Technologies, College Park, MD., September 1996

[3] Kenneth W. et al., "A 64-bit Shared Disk File System for Linux", In The 7th NASA Goddard Conference on Mass Storage System and Technologies in cooperation with the 16th IEEE Symposium on Mass Storage Systems, San Diego, USA, March 1999.

[4] Garth A. Gibson, David F. Nagle, William Courtright II, Nat Lanza, Paul Mazaitis, Marc Unangst, and Jim Zelenka. "NASD scalable storage systems", In Proceedings of the USENIX '99 Extreme Linux Workshop, June 1999.

[5] Parallel Data Lab. Nasd home page. http://www.pdl.cs.cmu.edu/NASD/

[6] C. Jurgens, "Fibre Channel: A Connection to the Future", IEEE Computer, pp. 82-90, August 1995.

[7] 김신우, 이용규, 김경배, 신범주, "대용량 파일 시스템을 위한 메타 데이타 구조 설계", 한국정보과학회 추계 학술 발표 논문집, vol 27, no.2, pp.59-61, 숙명여자대학교, 2000,10

[8] 신범주, "네트워크 연결형 자료저장시스템(SANtopia) 소개", 자료저장시스템 연구회 워크샵, pp.88-105, 2000.12

[9] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, H. Raymond Strong, "Extendible Hashing -- A Fast Access Method for Dynamic Files", ACM Transactions on Database Systems, Volume 4, No. 3., September 1979, pp 315-34

[10] U. Vahalia. "UNIX Internals: The New Frontiers". Prentice-Hall, 1996.

[11] http://www.brocade.com/products/silkworm/silkworm_2400_2800/index.jhtml

[12] http://www.seagate.com/cda/products/discsales/marketing/detail/0,1081,326,00.html

이 용 주

1999년 전북대학교 컴퓨터공학과(학사). 2001년 전북대학교 컴퓨터공학과(석사). 2001년 ~ 현재 한국전자통신연구원(컴퓨터시스템연구부 연구원). 관심분야는 하부저장 시스템, 네트워크 스토리지, 분산 파일시스템, 멀티미디어 데이타베이스 등

김 경 배

1992년 인하대학교 전자계산학과(학사). 1994년 인하대학교 전자계산학과(석사). 2000년 인하대학교 전자계산학과(박사). 2000년 ~ 현재 한국전자통신연구원(컴퓨터시스템연구부 선임연구원). 관심분야는 자료저장시스템, SAN, NAS, 이동컴퓨팅, 실시간데이타베이스시스템 등

신 범 주

1983년 경북대학교 전자공학과(학사). 1991년 경북대학교 컴퓨터공학과(석사). 1998년 경북대학교 컴퓨터공학과(박사). 1987년 ~ 2002년 한국전자통신연구원(책임연구원, 시스템S/W연구팀장). 2002년 ~ 현재 밀양대학교 컴퓨터.정보통신공학부교수. 관심분야는 분산시스템, 고장감내 미들웨어, 이동컴퓨팅, 스토리지 클러스터 S/W 등