논리볼륨 관리자를 위한 자유공간관리자의 설계 및 구현

(Design and Implementation of a Freespace Manager for a Logical Volume Manager)

최 영 희* 유 재 수** 오 재 철***
(YoungHee Choi) (JaeSoo Yoo) (JaeChul Oh)

요 약 높은 가용성, 확장성, 시스템 성능의 요구를 만족시키기 위해 SAN(Storage Area Network)이 등장했다. SAN을 보다 효과적으로 활용할 수 있도록, 대부분의 SAN 운영체제들은 SAN에 부착된 물리적 저장장치들을 가상적으로 하나의 커다란 볼륨으로 보이게 하는 저장장치 가상화 개념을 지원한다. 저장장치 가상화의 핵심적인 역할을 하는 것이 바로 논리볼륨 관리자이다. 자유공간 관리자는 논리볼륨의 자유공간들에 대한 정보를 유지관리 하면서 디스크 할당요구에 적절히 디스크를 할당해주는 역할을 한다. 이때 얼마나 단편화를 최소화하면서 효과적으로 자유공간에 대한 정보를 관리하는가는 전체 볼륨관리자의 성능을 결정하는 중요한 요인이 된다. 이 논문에서는 유연한 매핑을 돕기 위해 자유공간 관리 기법을 설계하고 구현한다. 이 논문의 자유공간 관리기법은 논리블록에 대한 물리블록 할당 외에도 스냅샷과 재구성을 위한 공간할당 및 해제를 효과적으로 처리한다.

Abstract A new architecture called the Storage Area Network(SAN) was developed in response to the requirements of high availability of data, scalable growth, and system performance. In order to use SAN more efficiently, most SAN operating systems support storage virtualization concepts that allow users to view physical storage devices attached to SAN as a large volume virtually. A logical volume manager plays a key role in storage virtualization. In order for mapping managers to process snap-shots and reorganizations, an efficient freespace manager is required, and it affects the overall performance of logical volume. In this thesis, we design and implement a freespace manager for logical volume manager. The freespace manager efficiently allocates physical blocks for logical blocks. Also, it processes space allocation requests for supports snapshots and reorganizations.

1. 서 론

폭발적으로 증가하는 대량의 정보를 효율적으로 공유하고 고속으로 서비스하기 위하여 데이터를 중심으로 하는 SAN(Storage Area Network)이 등장하였다[1, 13, 15]. SAN은 서버 중심의 시스템 환경이 네트워크 상의 다른 컴퓨터내 저장 시스템을 직접 활용하는 것을 허용하지 않기 때문에 대용량의 데이터를 저장하고 관리하는데 한계를 가졌던 점을 해결해 줄 수 있는 최선의 방책으로 인식되고 있다. SAN 환경을 보다 효과적으로 활용할 수 있기 위해서는 SAN에 대한 사용자 관점의 뷰를 제공하는 것이 필요하다. 이를 위해 대부분의 SAN 운영 S/W들은 여러 물리적 디스크들을 사용자 관점

* 호원대학교 전기전자정보통신공학부 최영희 교수
** 충북대학교 전기전자및 컴퓨터공학부 유재수 교수
*** 순천대학교 컴퓨터과학과 오재철 교수

에서 바라볼 수 있는 가상 디스크 형태로 추상화 시켜주는 저장장치 가상화 개념을 지원한다. 이러한 저장장치 가상화의 핵심적인 역할을 하는 것이 바로 논리볼륨 관리자이다[1].

논리볼륨 관리자는 사용자들이 특정 논리주소 블록에 대한 I/O를 요구하면 적절히 요구한 논리블록을 실제 디스크 내의 물리블록으로 매핑한다. 매핑 수행 과정에서 필요에 따라 논리볼륨내의 실제 디스크의 각 블록의 사용유무를 관리하는 자유공간 관리자가 논리블록에 적절히 물리블록을 할당해서 매핑을 돕게된다. 이와 더불어 논리볼륨 관리자는 사용자가 원하는 시점의 볼륨이미지를 유지할 수 있는 스냅샷과 디스크를 추가할 때 시스템을 정지시키지 않고 디스크간 부하균등을 위해 변경된 내용을 포함하도록 논리볼륨을 재구성하는 온라인 볼륨 재구성 기능을 제공해야 한다. 이 두 가지 연산을 수행하기 위해서는 논리볼륨의 상위(파일시스템, 데이터베이스시스템)는 인지할 수 없는 별도의 임시 저장공간이 필요하다. 이공간의 할당 및 관리 역시 자유공간 관리자가 수행해야 하는 일이다[10, 12, 13]. 따라서, 자유공간관리자의 성능은 전체적인 논리볼륨의 성능에 직접적인 영향을 미치게 된다.

이 논문에서는 유연한 매핑을 돕기 위해 자유공간 관리 기법을 설계하고 구현한다. 이 논문에서 구현한 자유공간 관리자는 매핑을 도와 논리블록에 대한 물리블록을 할당하는 일 외에도 스냅샷과 재구성을 위한 공간할당 및 해제를 매우 효과적으로 처리한다.

이 논문의 구성은 다음과 같다. 제2장에서는 자유공간 관리자의 기존연구에 대한 분석을 기술한다. 제3장에서는 설계 및 고려사항 그리고 설계 내용으로 구분하여 기술한다. 제4장에서는 3장에서 설계했던 내용을 구현 관점에서 정의한 함수와 그들간의 호출관계에 초점을 맞추어서 설명한다. 마지막으로 제5장에서는 결론을 맺는다.관점의 뷰를 제공하는 것이 필요하다.

2.관련 연구

조사에 의하면 기존에 논리볼륨 관리자를 위한 자유공간 할당 기법에 대한 설명을 하고 있는 참고문헌은 없다. 차이는 있지만 기존의 파일 시스템에서 행해지는 자유공간 관리 기법을 분석해 논리볼륨 관리자에 도입할 수 있는 방법을 찾아 보았다.

자유 공간 관리 정책은 효율적으로 공간을 할당/해제하여 파일 시스템의 단편화를 최소화하는 것이 목적이다. 파일 시스템의 단편화란 한 파일을 이루는 블록들이 흩어져 있는 상태를 말하는데 단편화 경향이 클수록 파일을 접근하기 위해 많은 I/O를 수행해야 한다. 따라서 단편화는 파일 시스템의 성능 저하를 초래한다.

기존에 자유 공간을 할당하는 방법으로 여러 가지가 소개되었다. 대표적인 것으로 선형 리스트 구조를 이용한 sequential fit나 트리 구조를 활용한 indexed fit, 비트맵 구조를 사용한 bitmapped fit등이 있다[2]. 이러한 정책중에 GFS[3,4]나 EXT2[5]와 같은 파일 시스템은 그림 2-1 과 같이 비트맵 기반의 자유공간 관리를 하고 있다.

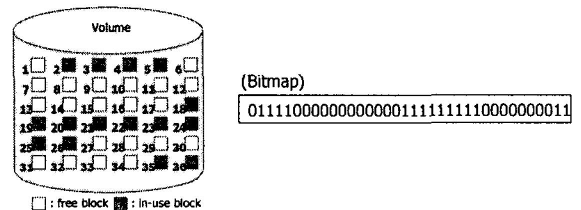

그림 2-1 비트맵 기반 자유공간 관리

XFS[6]나 Fujitsu사의 SafeFILE 제품의 파일 시스템인 HAMFS[7,8]의 경우는 B-tree를 활용한 색인 적합 방법을 사용하고 있다.

3.자유공간 관리자 설계

3.1 자유공간 관리자 설계시 고려해야 할 사항

SAN의 시스템의 구조는 그림 3-1과 같으며, 그림에서처럼 대용량의 저장 장치들이 파이버 채널이란 별도의 네트워크를 통해 연결되어 있다. SAN에서는 이 물리적 저장 장치들을 모아 하나의 논리 볼륨 형태로 상위에 제공하므로 자유 공간 정보를 어떻게 관리하느냐를 고려해야 한다. 만약 리스트나 비트맵 구조를 이용하여 관리한다며 대용량 저장 공간 하에서 자유 공간 검색 속도가 시간이 지남에 따라 느려지는 단점이 있는 반면에 에러 발생 후 복구 시 별다른 회복 절차 필요하지 않다. 색인을 활

용하여 자유 공간 정보를 관리하는 경우에는 많은 엔트리에 대해서도 빠르고 일정한 검색 속도를 지원하면서 연속 할당이나 단편화의 최소화가 가능하다. 그러나 색인 정보를 위한 추가 공간과 관리상의 복잡도가 요구되며 로깅을 위한 추가 방법이 필요하다. 이 경우 로깅의 방법으로 비트맵을 사용할 수도 있다.

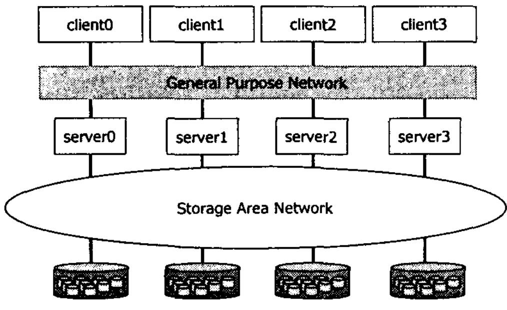

그림 3-1 SAN 시스템의 구조

SAN은 그림 3-2와 같이 여러 호스트가 SAN에 물려있는 저장 장치를 공유하므로 자유 공간 정보를 모든 호스트가 접근할 수 있어야 한다.

자유 공간 정보를 다중 호스트가 관리하는 방법을 그림 3-2와 같이 크게 세 가지 형태로 구분해볼 수 있다. A는 전체 자유 공간 정보를 관리하는 하나의 전용 호스트를 두는 방법이다. 자유 공간 정보를 한 호스트가 관리하므로 구조가 간단하지만 여러 호스트들의 자유 공간 할당 요청을 한 호스트가 처리해야 하므로 병목현상이 발생하고 호스트 부하가 커지는 단점이 있다.

B의 구성은 전체 자유 공간 정보를 여러 호스트가 중복 저장해서 관리한다. 이 경우 여러 호스트가 저장 공간을 바로 접근할 수 있으며, 한 호스트의 고장이 자유 공간 정보를 접근하는데 영향을 주지 않으므로 높은 가용성을 제공한다. 하지만 중복된 자유 공간 정보의 일관성을 유지하기 위해 전역 잠금을 빈번히 이용하므로 성능 저하를 초래한다.

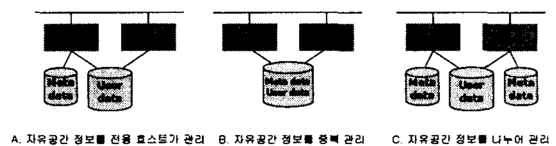

그림 3-2 자유 공간 정보를 관리하는 구성도

스냅샷은 생성할 때 매핑테이블을 복사하기 위한 공간과 스냅샷 생성 후 발생하는 쓰기 요청을 COW로 처리하기 위한 추가적인 공간이 필요하다. 또한 재구성을 할 때 데이터를 옮길 곳의 공간을 할당해주고 옮겨진 자리에 대한 공간 해제 역할을 해주어야 한다. 정상적인 데이터 저장을 위한 공간할당과 스냅샷과 재구성을 위한 공간할당이 같은 볼륨공간에 발생할 때 이들을 적절히 처리할 수 있는 방법을 생각해 보아야 한다.

3.2 자유공간 관리자의 특징 및 구조

앞서 대용량의 저장 공간을 다수의 서버가 접근 가능한 SAN환경을 고려한 자유 공간 관리의 요구 사항에 대하여 설명하였다. 이에 기초하여 설계한 자유 공간 관리 기법의 특징을 정리하면 다음과 같다. 먼저 자유 공간 관리를 위해 비트맵 구조를 사용하며 자유공간은 first fit으로 할당한다. 또한 대용량의 저장 공간을 갖는 SAN을 고려하여 다수의 서버가 자유 공간 정보를 분할 관리하도록 한다. 이때 다른 서버가 관리하는 영역에 대한 자유 공간 할당 요청은 서버간의 통신을 이용하여 할당한다. 스냅샷과 재구성을 위한 공간 할당의 경우에는 추가의 디바이스를 사용하지 않고 같은 디바이스에 할당한다. 이때 정상 공간 요청에 대해서는 디바이스의 앞부분부터 할당한다. 스냅샷이나 재구성을 위한 공간 할당 요청의 경우에는 디바이스의 뒷부분부터 함으로써 두 영역간에 간섭이 없도록 한다.

제안하는 자유 공간 관리 기법에서는 비트맵을 이용하여 자유 공간 관리를 한다. 각 블록의 사용여부를 한 비트에 대응시켜서 표시한다. SAN 환경에서 비트맵 구조를 이용하여 자유 공간 관리를 할 경우 문제점이 있다. 각 블록마다 한 비트씩의 공간을 추가로 차지하게 된다는 것이다. 소량의 저장공간에서는 무시할 수도 있는 크기이지만 대용량의 SAN환경에서는 비트맵을 위한 영역 역시 많은 공간을 차지하게 된다. 그렇게 되면 적당한 공간을 찾기 위해 탐색하는 시간이 길어진다는 문제점이 있다. 예를 들어 한 블록이 512Byte인 100G의 공간이 있

을 경우를 생각해보자. 한 블럭당 한 비트의 공간이 추가로 필요하게 되므로 100G의 경우 약 25메가바이트가 필요하게 된다. 이 경우 자유 공간 할당을 위한 시간은 시간이 지날수록 현저하게 저하된다. 따라서 자유 공간 관리를 위한 비트맵 영역을 서버들이 분할하여 관리하도록 한다. 제안하는 자유 공간 관리 기법에서 자유 공간 관리를 위한 시스템 구조가 그림 3-3에 나타나 있다. 이 그림에서 볼 수 있듯이 제안하는 방법에서는 각 서버마다 자유 공간 관리를 분할하여 처리한다. 그림에서는 서버1이 디바이스1과 디바이스2를, 서버2가 디바이스3번부터 5번까지 맡아서 관리한다. 디바이스는 실제 데이터들이 저장될 데이터 블록과, 각 디바이스마다 현재 디바이스의 데이터 블록이 사용되고 있는지 아닌지를 표시하기 위한 비트맵 영역으로 구성되어 있다.

이 경우 각 서버는 SAN을 구성하고 있는 가상 공간에 대한 접근을 할 수 있으며 필요한 공간을 할당받을 수 있다. 이때 각 서버마다 관리하고 있는 자유 공간이 있으므로 공간을 할당하고자 하는 디바이스를 만약 다른 서버가 관리하고 있다면 자유 공간할당을 위한 적절한 방법이 필요하다. 예를 들어 서버1이 자유 공간을 요청하였을 경우 디바이스3 에 데이터 블록을 할당해야 하는 경우를 가정해 보자. 이런 경우는 스트라이핑과 같이 라운드 로빈의 형태로 할당해야 하는 경우를 생각해 볼 수 있다. 만약 이럴 경우 디바이스3에 대한 자유 공간 관리는 서버2가 담당하고 있다. 따라서 이런 경우에는 서버1과 서버2간의 통신이 필요하게 된다.

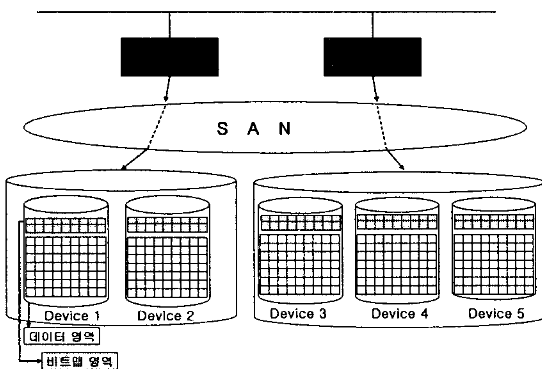

그림 3-3 비트맵 관리 구조

3.3 자유 공간 관리 기법

자유 공간의 할당 요청이 들어오는 경우를 생각해 보면 정상 공간 할당 요청과 스냅샷이나 재구성을 위한 공간 할당 요청의 경우가 있다. 제안하는 자유 공간관리 기법에서는 정상요청과 스냅샷이나 재구성을 위한 공간 할당 요청의 경우를 분리하여 공간을 할당한다. 그림 3-4는 한 디바이스 내에서 데이터 블록과 비트맵에 관한 관계 그림이다. 그림에서처럼 정상할당 요청의 경우는 앞에서부터(A영역) 데이터 블록을 할당하고, 스냅샷을 비롯한 기타의 공간 할당 요구 시에는 뒤에서부터(B영역) 하도록 한다.

자유공간 할당을 정상 요청과 스냅샷 관련 요청으로 구별하는 이유는 다음과 같다. 먼저 스냅샷을 위한 공간은 해당 스냅샷이 존재하는 동안에만 필요한 공간이다. 즉 스냅샷이 생성된 후 사용된 공간은 스냅샷 해제 시 모두 삭제되게 된다. 만약 이 공간을 정상할당 공간에 할당을 한다면 스냅샷을 위해 사용되었던 공간은 자유 공간으로 되면서 단편화가 생기게 된다. 이는 스트라이핑 볼륨의 스트라이핑 효과를 떨어뜨릴 뿐 아니라 자유 공간을 위한 탐색 시간이 길어지게 되어 자유 공간 할당을 위한 시간 역시 차츰 길어지게 된다. 또한 새로운 디바이스의 추가로 인하여 재구성이 발생하였을 경우를 생각해 볼 수 있다. 재구성 시에는 정상 할당 요청으로 인하여 할당된 블록에 대해서만 재구성을 수행해야 하는데, 만약 자유 공간 할당을 같이 한다면 재구성 엔트리 구성과 재구성 계획 수립을 위한 오버헤드가 발생하게 된다. 스냅샷을 비롯한 기타의 공간 할당을 정상 할당과 구별한다면 재구성시 보다 쉽게 재구성 엔트리를 구성하며 재구성할 수 있을 것이다. 그림 3-5 는 비트맵 탐색시간을 줄이기 위해서 추가한 자유공간 정보들이다.

3번 행의 total_block_num은 현재 디바이스에 있는 총블록의 수를 나타낸다. 4번 행의 next_avail_blk는 정상 요청의 경우 다음 번에 할당될 블록 번호를 나타낸다. 정상 요청으로 인하여 발생된 공간할당은 항상 디바이스의 앞부분부터 할당을 한다. 이런 정보를 이용하여 비트맵을 검색하지 않고도 쉽게 다음 번에 할당될 블록주소를 알 수 있다. next_avail_blk를 이용하면 현저하게 디스

크 I/O를 줄일 수 있다. 먼저 할당될 블록에 대한 비트 정보가 있는 비트맵 블록은 항상 메모리에 유지하고 변경 부분만 1로 설정하고 디스크의 해당 위치에 기록한다. 이렇게 next_avail_blk를 이용하면 한 번의 디스크 I/O만이 필요하다. 만약 next_avail_blk를 이용하지 않는다면 요청한 개수의 블록이 있는 곳을 찾기 위해 비트맵 탐색과정을 거쳐야 하며 해당하는 곳을 찾았을 경우 1로 설정한 뒤 다시 디스크에 기록하는 과정을 거쳐야 한다. 5번 행에 있는 next_avail_snapshot_blk는 스냅샷과 관련된 공간 할당 요구와 재구성시 재구성 엔트리를 저장하기 위한 공간할당 요청의 경우 처리하기 위한 것으로 next_avail_blk와 같은 기능을 수행한다. 이때 공간 할당은 뒤에서부터 이루어진다. 만약 next_avail_blk와 next_avail_snapshot_blk가 서로 만난다면 해당디바이스에는 더 이상 자유 공간이 없음을 의미한다.

일반적인 블록할당 요청의 경우에는 할당할 디바이스와 자유 공간을 찾은 후 해당 디바이스의 비트맵에 사용표시를 한 후 해당 블록의 주소를 반환해주면 된다. 정상 공간 할당 요청의 경우는 먼저 할당할 디바이스를 선정하는 단계를 수행한다. 만약 스트라이핑으로 구성되어 있는 경우에는 해당 디바이스는 라운드 로빈 방식으로 자유 공간을 할당할 디바이스를 결정한다. 따라서 마지막으로 할당한 디바이스가 어느 것인지 항상 유지하여 할당시에 순서상 이전에 할당이 이루어진 다음 번 디바이스에서 할당할 수 있도록 한다. 미러링의 경우에는 미러링을 구성하는 디바이스에 중복해서 저장해야 하므로 각 디바이스마다 공간을 할당한다.

일반적인 블록할당 요청에 의해 사용되고 있는 블록의 해제는 재구성을 수행하는 경우에 대해서만 이루어지는 것으로 가정한다. 파일 시스템 레벨에서 삭제가 발생하면 파일 시스템이 유지하고 있는 자유 공간 정보는 해제가 되지만, 그것에 대한 해제 사실을 논리볼륨에 전달하지 않는다. 이럴 경우 파일 시스템이 해당 논리주소에 대하여 새로운 공간 할당 요구를 한다면 매핑 관리자에서는 기존에 매핑되었던 물리주소를 반환하면 된다. 하지만 재구성 시에는 디바이스들간의 데이터 이동이 필요하며, 새롭게 추가된 디바이스로의 이동도 이루어진다. 이 경우 기존 디바이스들에서 사용된 블록들 중 일부는 해제되어야 한다. 정상 요청에 의해 할당되었던 데이터 블록의 해제는 재구성 수행 중 한블럭을 이동하면 해당 디바이스에 next_avail_blk를 조정하고, 해제된 블록에 대해서는 비트맵의 해당 비트를 0으로 초기화해주면 된다.

설계한 자유공간 관리자에서는 스냅샷과 관련된 자유공간 할당은 디바이스의 끝에서부터 할당한다. 스냅샷 관련 블록할당 요청은 스냅샷 생성시 매핑 테이블을 복사하기 위한 경우와 스냅샷 생성후 볼륨에 대한 변경요구로부터 스냅샷 이미지를 보호하기 위해 수행하는 COW를 위한 경우가 있다. 이들을 위한 블록할당은 이미 언급한 것처럼 디스크의 뒤족 부터 할당해나간다. 여기에서도 next_avail_snapshot_blk이라는 변수를 이용해서 공간할당을 위해 비트맵을 검색하는 시간을 줄인다.

스냅샷과 재구성을 위해 사용되었던 블록들은 연산의 종료와 함께 해제되어 자유공간으로 반환된다. 이 요청이 발생하는 경우에는 해당 블록에 대한 비트맵 정보를 0으로 초기화 해주어야 한다. 만약 해제 요청이next_avail_snap shot_blk와 같은 시작 주소를 갖는다면 next_avail_snapshot_blk의 위치를 비트맵 탐색을 통해 재조정해주는 단계가 추가로 필요하다.

재구성으로 인한 공간할당 요청일 경우에 재구성을 위한 정보를 저장하기 위해 할당요청을 할 때는 alloc_snapshot_block()을 호출한다. 재구성 수행시 실제 데이터 블록을 이동하기 위해서는 alloc_block() 함수와 release_block() 함수를 호출하여 할당 및 해제를 수행한다.

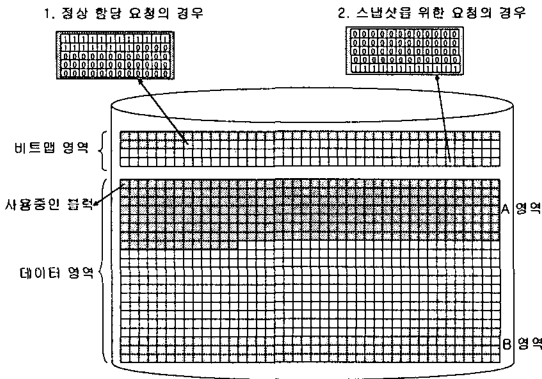

그림 3-4 디바이스내 구조

```
1   struct freesspace_manager
2   {
3        total_block_num;    // 현재 디바이스에
     있는 총 블록의 수
4        next_avail_blk;     // 정상 할당의 경우로
     다음 할당될 위치
5        next_avail_snapshot_blk; // snapshot관련
     요청에 대한 할당의 경우
6   } FS_MGR;
```

그림 3-5 자유 공간 관리 자료구조

4.자유공간 관리자의 구현

이 장에서는 제3장에서 설계한 자유공간 관리자를 구현한 내용에 대해서 기술한다. 제3장에서는 자유공간 관리에 대해서 여러 가지 방법을 제안하였다. 자유공간 관리자는 비트맵만을 이용하여 스냅샷과 일반적인 공간할당을 구분 없이 처리하는 방법과 비트맵과 다음 할당할 블록의 위치를 가리키는 포인터를 같이 사용한 방법을 모두 구현하였다. 어떤 방법을 사용할 것인지는 컴파일 시점에 선택할 수 있도록 하였다. 실험에 사용한 플랫폼은 Linux(Kernel version 2.4.9)의 운영체제와 P-3 450 , 128 Mbytes의 주기억 공간을 가지고 있다. 사용한 컴파일러는 GCC 2.7.1이다.

이 장의 구성은 다음과 같다. 먼저 4.1절에서는 자유공간 관리자를 구성하는 함수로 어떤 것들이 있고 이들이 어떤 호출관계를 갖고 있는지 설명하고 4.2절에서는 각 방법에 대한 구현 내용을 자세히 기술한다.

4.1. 함수의 구성 및 호출 관계

이 연구에서 구현한 매핑 및 자유공간 관리자의 주요함수들의 호출관계를 그림으로 표현하면 그림 4-1 과 같다. 구현한 볼륨관리자를 구성하는 주요 함수들로는 논리볼륨관리자의 유틸리티를 작성할 수 있게 해주는 pool_ioctl()과 상위 파일 시스템이나 데이터베이스 관리시스템과의 인터페이스를 위한 pool_make_reqest_fn()이 있다. 그 리고, 실제로 매핑을 수행하는 map()과 스냅샷 볼륨일 경우 COW를 수행하기 위한 copy_on_write()가 존재한다. 또한, 매핑을 처리하기 위해서 물리블록을 할당하거나 스냅샷과 재구성을 처리하기 위해서 물리블록을 할당하기 위한 자유공간 관리자 함수들로 alloc_block(),alloc_snapshot_blocks(), release_snapshot_blocks(), release_block()이 있다. 마지막으로 사용자의 스냅샷 요구나 볼륨 재구성 요구를 처리하기 위한 create_snapshot(), remove_snapshot(), resize_volume() 이 그림 4-1 과 같은 호출관계를 유지하고 있다.

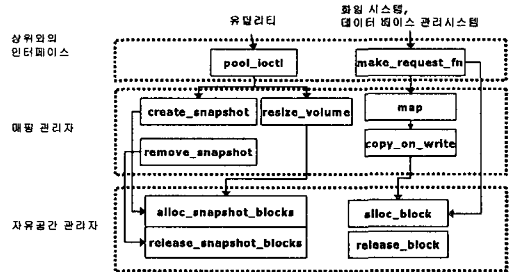

그림 4-1 구현한 자유공간 관리자 및 매핑관리자의 함수 호출 관계

4.2. 각 함수의 구현 내용

상위의 파일시스템이나 데이터베이스 관리시스템으로부터의 매핑 요구는 make_request_fn()에 의해서 볼륨관리자에 전달된다. 매핑요구로 전달된 논리주소를 보고 매핑을 위해서 어느 호스트를 호

출해야 하는지를 먼저 결정한다. 논리볼륨이 구성되면서 매핑에 참여할 서버들도 결정이 되고 각 서버가 분할된 매핑테이블의 어떤 부분을 책임질 것인지가 결정이 된다. 이 정보들은 전역적으로 유지가 되고 각 서버는 이 정보를 통해서 어느 서버가 매핑을 수행해야 할지 알 수 있게 된다. 이를 수행하는 함수가 그림 4-2 의 get_ proper_maphost() 이다. 서버가 결정이 되면 map_remote()를 호출하고 이 함수에서는 통신을 통해서 매핑 요구를 해당 서버로 전달한다. 물론, 자신이 해결할 수 있으면 바로 map_local()을 호출하여 매핑을 수행한다. 다른 서버로부터 매핑요구를 전달받은 서버 역시 map_local()을 호출해서 실제 매핑을 수행한다. 매핑이 끝나게 되면 매핑결과를 반환을 하는데 원격지 서버는 먼저 자신에게 매핑요구를 전달한 서버에게 결과를 전달하고 주 서버가 이를 반환하게 된다.

```
int get_proper_maphost(pool_t *Pool_Ptr, uint64
    Lsector, uint32 *HostId_Ptr,
    device_t *Dev_Ptr);
int map_remote( pool_t *Pool_Ptr, int RW,
    uint32 HostID, device_t *Dev_Ptr,
    uint32 Dev_BlkSize, uint64 Lsector,
    uint64 *Rsector_Ptr, kdev_t *Rdev_Ptr);
int map_local( pool_t *Pool_Ptr, device_t Dev,
    int RW, uint32 Dev_BlkSize,
    uint64 Lsector, uint64 *Rsector_Ptr,
    kdev_t *Rdev_Ptr);
```

그림 4-2 매핑 함수들

map_local() 에서는 매핑을 수행할 때 매핑테이블로부터 읽어온 논리주소에 대한 물리주소가 NULL 이고 쓰기 요청이면 그림 4-3 의 alloc_block()을 호출한다. alloc_block에서는 d_bsize 만큼의 블록을 할당해서 그 시작 물리주소를 반환한다. alloc_block() 대한 자세한 수행 절차가 그림 4-4 에 있다.

```
int alloc_block(pool_t *pp, kdev_t *d_rdev,
    uint64 *dev_sector, uint32 d_bsize);
int release_block(pool_t *pp, kdev_t d_rdev,
    uint32 d_bsize);
int alloc_snapshot_blocks(pool_t *pp,
    kdev_t *dev, uint64 *sector, uint32 size);
int release_snapshot_blocks(pool_t *pp,
    kdev_t dev,uint64 sector,uint32 size);
```

그림 4-3 자유공간 할당 함수들

```
1.  alloc_block(device, block, size)
2.  /* 상위레벨에서 자유 공간 요청이 들어온다; */
3.      device = 할당할 디바이스를 선정한다.
4.      선정된 device에서 자유 공간
        정보(FS_MGR)를 가져온다
5.      block = FS_MGR의 next_avail_blk;
6.      if (할당후 next_avail_blk
        < next_avail_snapshot_blk)
7.      {
8.          bitmap에 해당 블록에 대한 비트를 1로
            설정한다.
9.          device와 block을 반환한다
10.     }
11.     else
12.         alloc_block();
13.     return;
```

그림 4-4 정상 공간 할당 요청에 대한 처리 절차

alloc_block() 함수는 새로운 공간을 할당하여 할당한 디바이스와 블록번호를 반환하는 역할을 한다. 입력 값은 size로서 몇 개의 블록을 할당할 것인지 블록의 수를 나타낸다. 먼저 3행에 할당할 디바이스를 선정하는 단계를 거치게 된다. 만약 매핑관리자가 특정 디바이스를 요구하는 경우에는 요구한 디바이스를 선택하고 해당 가상 공간이 스트라이핑으로 구성되어 있다면 라운드 로빈 방식을 이용하여 디바이스를 선정한다. 4행에서는 선정된 디바이스에 해당하는 Freespace_Manager를 얻어온다.

Freespace_Manager로부터 해당 디바이스에 대한 총블럭의 수, 사용된 블록의 수를 얻을 수 있다. Freespace_Manager로부터 할당할 블록 번호를 얻어온다. 만약 할당후 next_avail_blk의 위치가 스냅샷 공간을 나타내는 next_ avail_snapshot_blk보다 작다면 정상적인 할당이 이루어질 수 있는 단계로

8번 행을 수행하게 된다. 8번 행에서는 해당 디바이스에서 해당 블록에 대한 비트를 비트맵 공간에 1로 설정하는 단계이다. 성공적으로 8번 행이 수행이 되면 할당한 디바이스와 블록 주소를 반환한다. 만약 해당 디바이스에 자유 공간이 없는 경우에는 alloc_block을 재 호출하여 다른 디바이스로부터 공간 할당을 요청한다.

```
int SS_snapshot(pool_t *pp, pool_t *np,
    char *ss_name);
int ss_copy_map(kdev_t dev, uint64
    destination_start, uint64 source_start,
    uint32 size);
int SS_COW_snapshot(pool_t *pp, device_t map_dev,
    uint32 offset, uint32 size,
    kdev_t p_dev, uint64 p_sector);
int SS_release_snapshot(pool_t *pp,
    uint32 snapshot_num);
```

그림 4-5 스냅샷 관련 함수들

다음은 스냅샷의 수행과정을 구현관점에서 설명한다. 스냅샷의 생성은 사용자에 요청에 의해서 시작된다. 구현한 볼륨관리자는 snapshot 이라는 유틸리티를 제공하며 'snapshot -c snapshot_name volume_name' 와 같이 입력하게 되면 스냅샷이 생성이 된다. snapshot 유틸리티는 snapshot_name과 volume_name을 ioctl()을 호출해서 pool_ioctl()로 전달한다. pool_ioctl()에서는 먼저 ss_copy_map()을 호출하여 volume_name에 해당하는 커널내부의 볼륨 자료구조를 복사해서 snapshot_name이라는 가상 볼륨을 생성한다. 그리고, 생성한 볼륨에 대한 MAJOR, MINOR 장치 번호를 다시 사용자 영역의 유틸리티 프로그램인 snapshot 으로 반환한다. snapshot에서는 장치번호를 가지고 mknod를 호출하여 snapshot_name의 디바이스를 생성한다.

pool_ioctl()에서 스냅샷을 생성할 때는 SS_snapshot()을 호출한다. 이전에서 언급한대로 스냅샷을 생성할 때는 두가지 방법이 있다. 하나는 매핑테이블을 복사하는 방법(그림 4-6)이고 다른 하나는 해쉬테이블을 이용하는 방법(그림 4-7)이다. 각 방법의 생성 과정을 그림 4-6 과 그림 4-7 에서 보여준다.

```
1  SS_snapshot()
2  {
3  모든 노드에 스냅샷의 시작을 알린다;
4
5  alloc_snapshot_block( 매핑테이블 크기 );
6
7  for ( i=0; i<매핑테이블 블록 개수; i++ )
8  {
9  원본 매핑테이블의 i번째 블록을 읽는다;
10 할당받은 위치의 i번째 블록에 기록한다;
11     }
12 }
```

그림 4-6 스냅샷-1의 SS_snapshot()

```
1  SS_snapshot()
2  {
3        모든 노드에 스냅샷의 시작을 알린다;
4
5        alloc_snapshot_block( 해쉬 헤더의 크기 );
6
7        for ( i=0; i<해쉬 키의 개수; i++ )
8        {
9            alloc_block( );
10       }
11 }
```

그림 4-7 스냅샷-2의 SS_snapshot()

그림 4-6 과 그림 4-7 에서도 볼 수 있듯이 SS_snapshot은 스냅샷을 위한 공간을 할당하기 위해서 alloc_snapshot_block() 이라는 함수를 호출한다. 이 함수는 입력 값으로 들어온 size만큼의 블록을 디바이스 뒷부분부터 할당한다. 스냅샷을 위한 공간 할당의 절차는 그림 4-8 과 같다.

alloc_snapshot_block함수는 스냅샷 시작시 스냅샷과 관련된 정보를 위해 호출되며, 스냅샷 COW시에도 호출이 된다. alloc_snapshot_block 함수는 요청 받은 블록의 수 (size)만큼의 공간을 디바이스의 뒷부분에서 할당한다. alloc_block함수처럼 alloc_snapshot_block함수도 유사한 과정을 거치게 된다. 먼저 3행에서 할당할 디바이스를 선정하는 단계를 수행한다. 만약 함수 호출시 디바이스가 선정이 된 경우는 해당 디바이스를, 그렇지 않으면 라운드 로빈으로 디바이스를 선택한다. 디바이스가

선정이 되면 5행에서 해당 디바이스에 적절한 자유 공간이 있는지 검사한다. 이때 만약 디바이스가 선정되어 호출된 경우에는 자유 공간이 없음에 대한 에러를 반환하며, 라운드 로빈으로 디바이스 선정시에는 alloc_snapshot_block 함수를 다시 호출하여 다른 디바이스에서 할당 요청을 처리한다. 8행에서 22행까지는 해당디바이스에서 자유공간을 찾고 비트맵에 설정한 뒤 찾은 디바이스와 블록 주소를 반환한다. 이 과정에서 두단계의 검사를 수행한다. 먼저 9행에서 15행까지 수행하여 이미 할당되었었던 영역들에 대해서 자유 공간이 있는지 검사한다.

9행에서 15행까지 수행하여 이 공간에 자유 공간이 있는지 검사하고 만약 존재한다면 그 블록주소를 반환하면 된다. 그렇지 않다면, 즉 이미 할당되었던 영역들중 요청된 할당 개수(size)를 만족하는 공간이 없을 경우에는 next_avail_snapshot_blk를 이용하여 할당하면 된다. 16행에서 21행까지 수행하여 자유 공간의 시작 주소는 next_avail_snapshot_blk에서 size만큼의 수를 빼준 위치에서 자유 공간 할당 요청에 대한 블록주소를 반환해주면 된다.

이렇게 next_avail_snapshot_blk를 이용한다면 bitmap만을 이용하여 자유 공간을 탐색하는 것보다 디스크 I/O를 줄일 수 있다. 먼저 요청된 개수만큼의 공간이 있는지를 쉽게 FreeSpace_Manager로부터 알 수 있다. 최악의 경우를 생각해보면 이미 할당된 영역 내에 있는지 비트맵 탐색을 통하여 검색을 한 결과 존재하지 않는 경우로, 이때는 (next_avail_snapshot_blk-size ~ next_avail_snapshot_blk) 사이의 블록을 할당하는 것으로 비트맵에 반영하는 디스크 I/O가 필요하다. 이때에도 비트맵을 이용하는 것보다 (next_avail_snapshot_blk-size ~ next_avail_snapshot_blk)사이가 빈 공간인지 검사하기 위한 탐색 시간을 줄일 수 있다.

```
1.  alloc_snapshot_block(device, block, size)
2.  {/* 상위레벨에서 자유 공간 요청이 들어온다;*/
3.  device = 할당할 디바이스를 선정한다.
4.  선정된 device에서 자유 공간 정보(FS_MGR)를
    가져온다
5.        if(디바이스내에 자유 공간을 할당한 블록이
            없는 경우)
6.            alloc_snapshot_Block(); // 다른
              디바이스에 할당 요청
7.            return;
8.        /* 이미 할당되었던 영역에 대하여 자유
            공간 검사 */
9.        if(size < FS_MGR.total_block_num-
            FS_MGR.next_avail_snapshot_blk)
10        {
11            if(자유 공간이 있는 경우)
12.               bitmap에 해당 블록들에 대한
                  비트를 1로 설정한다.
13                block = 찾은 자유 공간 주소;
14.           return ;
15.       }
16.       if(size < FS_MGR.next_avail_snapshot_blk-
            FS_MGR.next_avail_blk)
17.       {
18.           FS_MGR.next_avail_snapshot_blk -=
              size;
19.           block =
              FS_MGR.next_avail_snapshot_blk;
20.           bitmap에 해당 블록들에 대한 비트를
              1로 설정한다.
21.       }
22        return;
23  }
```

그림 4-8 스냅샷 관련 할당 요청에 대한 처리 절차

생성된 스냅샷은 다시 사용자가 스냅샷 해제요청을 하면 해제된다. 역시 snapshot 유틸리티를 이용하게 되는데 'snapshot -r snapshot_name volume_ name' 과 같이 입력한다. 이러면 다시 ioctl()을 통해서 pool_ioctl()에 이 내용을 전달하게 되고 pool_ioctl() 에서는 그림 4-5 의 SS_release_snapshot()을 호출하여 스냅샷을 해제한다. 스냅샷-1 과 스냅샷-2 각각에 대한 스냅샷 해제 과정을 간단하게 그림 4-9 와 그림 4-10 에서 보여주고 있다.

```
1   SS_release_snapshot()
2   {
3         모든 노드에 스냅샷의 해제를 알린다;
4
5         for ( i=0; i<매핑테이블의 엔트리 개수;
                i++ )
6               if ( 스냅샷 매핑테이블[i] != 원본
                      매핑테이블[i] )
7                     release_block( 스냅샷
                            매핑테이블[i]의 엔트리 );
8
9         release_snapshot_block( 매핑테이블의
                                      크기 );
10  }
```

그림 4-9 스냅샷-1의 SS_release_snapshot()

```
1   SS_release_snapshot()
2   {
3         모든 노드에 스냅샷의 해제를 알린다;
4
5         for ( i=0; i<해쉬 키의 개수; i++ )
6         {
7               for ( 현재 해쉬키의 모든 엔트리 )
8               {
9                     release_block( );
10              }
11        }
12
13        release_snapshot_block( 해쉬 헤더의
                                      크기);
14  }
```

그림 4-10 스냅샷-2의 SS_release_snapshot()

SS_release_snapshot() 에서는 스냅샷 공간을 해제하기 위해서 release_ snapshot_block()을 호출한다. release_snapshot_block() 에서는 스냅샷을 위해 할당되었던 공간을 해제하여 나중에 다시 사용될 수 있도록 한다. 해제 절차를 그림 4-11 에서 보여준다. 먼저 4행에서 해당 디바이스에서 해제할 블록에 해당하는 비트를 비트맵 블록에서 해제한다. 그런 다음 삭제할 블록의 주소가 FS_MGR.next_avail_snapshot_blk와 같은지 비교한다. 비교 결과 같다면 5행에서 9행을 통해 FS_MGR.next_avail_snapshot_blk의 위치를 조정해 준다.

```
1.  release_snapshot_block(device, block, size)
2.        선정된 device에서 자유 공간
          정보(FS_MGR)를 가져온다
3.        /* 해당 디바이스에서 block부터 size만큼의
             bitmap 영역을 0으로 초기화한다. */
4.        clear_bitmap(block, size);
5.        if block ==
             FS_MGR.next_avail_snapshot_blk
6.           /* 비트맵을 탐색하여
          FS_MGR.next_avail_snapshot_blk 이후 블록중
7.           1인 블록(blknum)을 탐색 */
8.           search_bitmap(block, &blknum)
9.           FS_MGR.next_avail_snapshot_blk  =
             blknum;
10.       return;
```

그림 4-11 스냅샷 블록에 대한 해제에 대한 처리 절차

스냅샷이 생성된 후에 해당 볼륨에 쓰기 요구가 발생하면 COW를 수행해야 한다. 해당 쓰기 요구가 스냅샷 생성 이전 시점에 있었던 블록에 대한 변경이라면 이미 앞의 '스냅샷 설계' 편에서 설명한 대로 COW를 수행해야 한다. 스냅샷-1 과 스냅샷-2 의 COW 수행절차를 그림 4-12 와 그림 4-13 에서 보여준다.

```
1   SS_COW_snapshot()
2   {
3         if ( 스냅샷 매핑테이블 엔트리 == 원본
                 매핑테이블 엔트리 )
4         {
5               alloc_block( );
6               원본 매핑테이블의 엔트리 값을
                 읽어온다;
7               할당 받은 블록에 읽어온 값을
                 기록한다;
8               스냅샷 매핑테이블 엔트리를
                 할당받은 블록으로 변경한다;
9         }
10  }
```

그림 4-12 스냅샷-1의 SS_COW_snapshot()

```
1   SS_COW_snapshot()
2   {
3       if ( hash_search(  ) < 0 )
4       {
5               alloc_block(  );
6               원본 매핑테이블의 엔트리 값을
                읽어온다;
7               할당 받은 블록에 읽어온 값을
                기록한다;
8               insert_hash(  );
9       }
10  }
```

그림 4-13 스냅샷-2의 SS_COW_snapshot()

다음은 재구성에 대한 구현 내용이다. 재구성의 수행절차는 다음과 같다. 사용자는 추가되는 디바이스에 대한 정보를 담은 구성파일과 함께 재구성 유틸리티를 실행한다. 재구성 유틸리티는 ioctl()을 호출하여 커널내부의 pool_ ioctl()로 재구성 정보를 전달하면 커널내부의 재구성이 시작된다. 먼저 재구성을 수행하기 위해 볼륨 전체에 대한 쓰기 모드의 잠금을 설정한다. 다음으로 make_mvlist()를 호출하여 재구성시 물리블록에 대한 논리주소를 반환하기 위해 물리주소 대 논리주소 테이블을 작성한다. 다음 단계는 resize_ exec()를 호출하여 재구성으로 이동될 블록이 어디서 어디로 이동해야 하는지를 기록한 재구성 테이블을 각각의 재구성 규칙에 의해 작성하고, 이 테이블을 참조하여 블록의 이동을 시작한다. 블록을 이동하면서 update_maptable()을 호출하여 매핑테이블의 내용을 같이 변경한다. 마지막으로 재구성이 끝나면 물리주소 대 논리주소 테이블과 재구성 테이블을 삭제하고 볼륨에 대한 쓰기 모드 잠금을 해제한다.

그림 4-14 는 재구성에 필요한 주요 관련 함수들을 나열한 것이다. st_ resize_volume()이 재구성에서 지원하는 인터페이스이며, 나머지 함수들은 이 인터페이스내에서 호출되어 구성이 된다. 그림 4-15 는 재구성 수행시 옮겨질 물리블록에 대한 매핑 테이블 정보를 변경해야 하는데, 물리블록에 대한 논리주소를 참고하기 위해 물리주소 대 논리주소 테이블을 구성하는 함수이다.

```
int st_resize_volume(pool_t *pp, int old_dev_num,
    int d_bsize)
int make_mvlist(pool_t *pp, int old_dev_num,
    int d_bsize)
int resize_exec(pool_t *pp, int old_dev_num,
    int add_dev_num, int d_bsize)
int update_maptable(pool_t *pp, uint64 psector,
    kdev_t dev_id, uint64 dev_sector)
```

그림 4-14 재구성관련 주요 함수들

```
make_mvlist()
{
  for(풀내의 모든 디바이스에 대해서)
  {
    if(추가된 디바이스 이면)
      break;
    매핑된 블록의 개수 계산;
    매핑된 블록에 대한 물리주소 대 논리주소
          테이블 영역 할당;
    for(디바이스내의 모든 블록에 대해서)
    {
      블록의 물리주소를 테이블내의 물리주소에
      설정;
    }
    테이블 내용을 디바이스에 저장;
  }
  for(풀내의 모든 디바이스에 대해서)
  {
    if(추가된 디바이스 이면)
      break;
    디바이스내의 매핑테이블 접근;
    for(매핑테이블내의 모든 논리주소에 대해)
    {
      논리주소에 매핑된 물리주소 획득;
      물리주소에 해당하는 물리주소 대 논리주소
      테이블을 검색;
      물리주소 대 논리주소 테이블의 나머지
      논리주소 영역 설정;
    }
    테이블 내용을 디바이스에 저장;
  }
}
```

그림 4-15 재구성테이블 생성

그림4-16 은 재구성으로 이동될 블록이 어디서 어디로 이동해야 하는지를 기록하는 재구성 테이블을 작성하고, 이 테이블을 기반으로 재구성을 실행하는 함수이다. 이때 재구성 블록을 이동하면서 매핑 테이블의 변경도 동시에 수행된다. 재구성 실행

이 완료되면 재구성 테이블 및 물리주소 대 논리주소 테이블을 삭제하면서 재구성이 끝나게 된다.

```
resize_exec()
{
  재구성 규칙에 의거해 재구성 테이블 작성;
  for(재구성 테이블의 모든 블록에 대해서)
  {
    물리주소 대 논리주소 테이블 접근;
    for(물리주소 대 논리주소 테이블의 모든 내용에
        대해서)
    {
      if(재구성 블록의 물리주소와 일치 하다면)
      {
        재구성 블록의 물리주소에 해당하는 논리주소
        획득;
        break;
      }
    }
    논리주소를 포함한 매핑 테이블 접근;
    논리주소에 매핑된 물리주소를 검색후 이동될
    물리주소로 변경;
    재구성을 테이블에 의해 재구성 블록 이동;
  }
  재구성 테이블 삭제;
  물리주소 대 논리주소 테이블 삭제;
}
```

그림 4-16 재구성 수행 절차

5.결 론

이 논문에서는 논리볼륨의 성능을 더욱 향상시키기 위한 자유공간 관리 기법을 설계하고 구현하였다. 설계하고 구현한 자유공간 관리자는 논리볼륨 관리자의 매핑수행 과정에서 필요에 따라 논리볼륨내의 실제 디스크의 각 블록의 사용유무를 관리하며 논리블록에 적절히 물리블록을 할당해서 매핑을 돕게된다. 또한 논리볼륨 관리자의 중요한 특징인 스냅샷과 온라인 볼륨 재구성 기능 수행시 필요한 공간의 할당 및 관리를 수행한다. 이 논문에서 제안한 자유공간 관리자는 자유블록에 대한 포인터를 두어 공간할당을 빠르게 하였고, 일반 공간할당 영역과 스냅샷이나 재구성 공간할당 영역을 분리하여 일반 I/O 연산의 성능을 최대한 보장할 수 있도록 하였다.

향후 연구에서는 SAN 환경에 이 논문에서 구현한 자유공간 관리자를 설치하고 다양한 실험을 통해 성능향상정도를 측정하는 성능평가를 수행하도록 하겠다.

참 고 문 헌

[1] Edward K.Lee and Chandramohan A.Thekkath, "Petal : Distributed Virtual Disks," In Proceedings of International Conference on ASPLOS, 1996, pp84-92.

[2] P. R. Wilson et al. "Dynamic storage allocation: A survey and critical review," Proc. Int'l Workshop on Memory Management. Kinross, Scotland, UK, Sep. 1995.

[3] Steven R. Soltis, Thomas M. Ruwart, Matthew T. O'Keefe. "The Global File System," Conference on Mass Storage Systems and Technologies, Sept, 1996, pp17-19.

[4] Manish Agarwal. "System Calls for Automated Online File System Maintenance Tools," Masters project. University of Minnesota, Department of Electrical and Computer Engineering, November, 1999.

[5] David A Rusling. 1996-1999, "The Linux Kernel".

[6] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto and G. Peck. "Scalability in the XFS file system," In USENIX Technical Conference, Usenix, January, 1996, pp1-14.

[7] Yoshitake Shinkai, Yoshihiro Tsuchiya and Takeo Murakami. "HAMFS File System," Fujitsu Laboratories Limited.

[8] Yoshitake Shinkai, Yoshihiro Tsuchiya and Takeo Murakami. "Alternatives of Implementing a Cluster File Systems," Fujitsu Laboratories Limited.

[9] L. W. McVoy & S. R. Kleiman. "Extent-like Performance from a UNIX File System," Sun Microsystems, Inc.

[10] C. Mohan and F. Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," Proc. 1992 ACM SIGMOD Conf., San Diego, June, 1992.

[11] Ruwart, Thomas M., Performance Characteristics of Large and Long Fibre Channel Arbitrated Loops ,

Proceedings, 16 th IEEE Symposium on Mass Storage Systems /7 th NASA Goddard Conference on Mass Storage Systems and Technologies, March 1999, IEEE Computer Society Press
[12] Thomas M. Ruwart and Matthew T. O'Keefe, "Performance Characteristics of a 100 MB/sec Disk Array," Storage and Interfaces '94, San Jose, CA
[13] Alex Elder et al., "The InTENsity PowerWall: A Case Study for a Shared File System Testing Framework," Proceedings, 17 th IEEE Symposium on Mass Storage Systems / 8 th NASA Goddard Conference on Mass Storage Systems and Technologies, March 2000, IEEE Computer Society Press
[14] Jim Lyon, "Toward Binary Compatibility of VIA Implementations on WIN32," Microsoft Corp. T11/98-435v1 FC-GS-2 rev 5.3
[15] "Qlogic SAN/Device Management API" Draft Version 1.3 05/26/00