논리볼륨 관리자를 위한 매핑 관리자의 설계 및 구현

(Design and Implementation of a Mapping 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. 논리볼륨 관리자는 논리 주소를 물리 주소로 매핑시킴으로서 저장장치 가상화를 실현한다. 이 논문에서는 논리볼륨 관리자를 위한 효율적이고 유연한 매핑 기법을 설계하고 구현한다. 이 논문의 매핑 기법은 특정 시점의 볼륨이미지를 유지할 수 있는 스냅샷과 시스템을 정지시키지 않고 SAN에 디스크를 추가 또는 삭제할 수 있는 온라인 재구성 기능을 지원한다.

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. It realizes the storage virtualization by mapping logical addresses to physical addresses. In this paper, we design and implement an efficient and flexible mapping method for logical volume manager. The mapping method in this paper supports a snapshot that preserves a volume image at certain time and on-line reorganization to allow users to add or remove storage devices to SAN even while the system is running.

1. 서 론

폭발적으로 증가하는 대량의 정보를 효율적으로 공유하고 고속으로 서비스하기 위하여 데이터를 중심으로 하는 새로운 개념의 컴퓨터 시스템 환경이 도래하였다. 가장 주목할 만한 것은 서버에 개별적지 않고 네트워크에 연결된 저장장치를 직접 접근하도록 하는 SAN을 들 수 있다. 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 방안으로 인식되고 있다[1, 2]. SAN 환경을 보다 효과적으로 활용할 수 있기 위해서는 SAN에 대한 사용자 관점의 뷰를 제공하는 것이 필요하다. 이를 위

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

해 대부분의 SAN 운영 S/W들은 여러 물리적 디스크들을 사용자 관점에서 바라볼 수 있는 가상 디스크 형태로 추상화 시켜주는 저장장치 가상화 개념을 지원한다. 이러한 저장장치 가상화의 핵심적인 역할을 하는 것이 바로 논리볼륨 관리자이다[2, 3].

논리볼륨 관리자의 가장 중요한 기능은 사용자들이 특정 논리 주소 블록에 대한 I/O를 요구하면 적절히 요구한 논리블록을 실제 디스크 내의 물리블록으로 매핑하는 것이다. 이와 더불어 논리볼륨 관리자는 사용자가 원하는 시점의 볼륨이미지를 유지할 수 있는 스냅샷과 디스크를 추가하거나 삭제할 때 시스템을 정지시키지 않고 디스크 간 부하균등을 위해 변경된 내용을 포함하도록 논리볼륨을 재구성하는 온라인 볼륨 재구성 기능을 제공해야 한다. 이에 본 논문에서는 Linux상의 SAN 볼륨관리자인 pool[1]을 기반으로 효율적이고 유연한 매핑 기법을 설계하고 구현한다. 이 논문의 매핑 기법은 스냅샷과 온라인 재구성 기능을 지원한다.

이 논문의 구성은 다음과 같다. 2장에서는 볼륨 관리자의 매핑 관리에 대한 기존 연구에 대해 기술한다. 3장에서는 매핑 관리자를 설계할 때 고려해야할 사항을 제시하고 이에 따른 설계 내용을 기술한다. 4장에서는 3장에서 설계한 매핑 관리자의 구현에 대해서 설명하고 5장에서 결론을 맺는다.

2.관련 연구

기존 논리볼륨 관리자의 매핑 방법을 살펴보면 크게 수식 기반의 매핑 방법과 매핑 테이블 기반의 매핑 방법으로 나누어진다. 수식 기반의 매핑 방법은 단순하여 매핑 요구를 빠르게 처리할 수 있지만, 스냅샷 및 볼륨 재구성과 같이 매핑 관계가 변하는 상황에 유연하게 대처하지 못한다. 반면에, 매핑 테이블 기반의 매핑 방법은 수식 기반보다 매핑 요구 처리가 느리고 SAN과 같은 대용량 환경에선 매핑 테이블의 크기가 커서 관리하기가 어렵지만, 위와 같이 매핑 관계가 변하는 상황에 유연하게 대처할 수 있는 장점이 있다. 이 장에서는 기존의 논리볼륨 관리자의 매핑 방법 및 자유공간 관리 방법에 대해 조사/분석한 내용을 기술한다.

2.1.매핑 기법

미네소타 대학에서 개발된 SAN 볼륨관리자인 pool은 별도의 매핑 테이블을 두지 않고 수식 기반의 매핑을 수행한다. 수식 기반의 매핑 방법에서는 매핑 관계가 변하는 볼륨 재구성에 유연하게 대처하지 못하므로 pool의 볼륨 재구성은 디스크의 추가시 논리볼륨의 용량이 추가될 뿐, 디스크의 부하 균등을 위한 디스크간 데이터 이동이 전혀 고려되지 않았다.

Petal[2]은 네트워크로 연결된 여러 호스트에 부착된 디스크를 가상화하여 하나의 논리 디스크로 클라이언트에게 제공하는 시스템이다. 그림 2-1에서 Petal의 구조를 보여주고 있다. 그림에서처럼 네트워크에 부착된 서버의 디스크들을 마치 하나의 디스크처럼 가상화한다. Petal에서는 이 가상화를 매핑 테이블을 이용하여 실현한다.

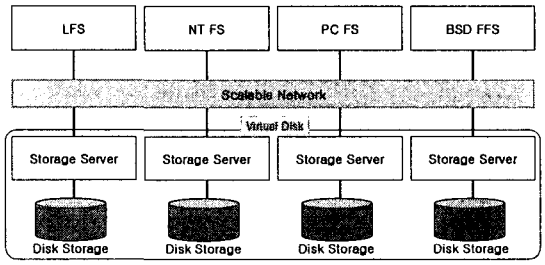

그림 2-1 Petal의 구조

Petal의 매핑 테이블 구조는 그림 2-2와 같다. 클라이언트가 이용하는 논리 주소는 <vdiskID, offset>과 같다. vdiskID는 Petal 가상 디스크의 번호이고, offset은 해당 vdiskID내에서의 위치이다. 이 논리 주소를 Petal은 그림에서 보는 단계를 거쳐서 <serverID, diskID, diskOffset>의 물리 주소로 변환한다. serverID는 해당 디스크가 어느 서버에 속해 있는지를 나타낸다. diskID는 해당 서버의 몇 번째 디스크에 해당하는 가를 나타내고, diskOffset은 그 디스크내에서의 위치를 나타낸다.

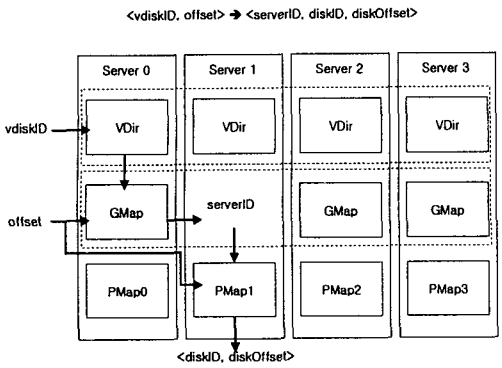

그림 2-2 Petal의 매핑 절차

그림 2-2에서 VDir과 GMap은 전역 정보이다. 즉, 모든 호스트가 접근하는 정보이다. PMap은 지역정보로써 각 서버가 자신에 부착된 디스크의 매핑 테이블을 유지/관리하는 것이다. 매핑 절차는 다음과 같다. 먼저, VDir을 접근해서 논리 주소를 GMapID로 변환한 뒤 이 GMapID를 적절히 serverID로 변환한다. 즉, 원하는 데이터가 어떤 서버에 있는지를 얻어내고, 마지막으로 serverID에 해당하는 PMap을 통해서 어떤 디스크의 어느 위치에 있는지를 얻어낸다. 이렇게 매핑 테이블을 이용해 가상화를 실현하는 Petal에서는 온라인 백업을 위한 스냅샷과 볼륨 재구성을 지원하고 있다. 하지만 Petal은 네트워크를 통해 연결된 여러 서버가 가지고 있는 저장장치들을 가상화 해주는 시스템이며 SAN과 같은 환경에 적합한 볼륨관리자는 아니다. 따라서, 위의 PMap과 같은 정보를 관리하는 방법처럼 각 서버는 자신이 가지고 있는 디스크들에 대한 물리적 변환 테이블을 가지고 있는 것이 자연스럽다.

3. 매핑 관리자 설계

이 논문에서는 논리볼륨 관리자를 위한 효율적이고 유연한 매핑 방법으로 매핑 테이블 기반의 매핑 기법을 사용한다. 이미 앞에서 언급했듯이 매핑 테이블 기반의 매핑은 수식 기반의 매핑 방법에 비해 매핑 요구 처리가 느리고 매핑 테이블을 관리하기가 어렵지만, 매핑 관계가 변하는 스냅샷과 볼륨 재구성시 유연하게 대처할 수 있다. 이 논문에서는 매핑 테이블을 쉽게 관리하면서 빠르게 매핑 요구를 처리할 수 있는 방법에 초점을 맞추어 매핑 관리자를 다양한 각도로 설계한다. 그리고 설계된 방법들의 장단점에 대해 자세히 기술하고 이 논문에서는 어떤 방법을 취했는지 설명한다.

3.1.매핑 관리자의 설계

이 논문에서는 매핑 관리자를 설계하기 위해 매핑 테이블의 관리 방법과 매핑 테이블의 표현 방법을 주로 고려하였다. 매핑 테이블 관리 방법이란 SAN 환경에서 다중 호스트가 매핑 테이블을 접근해서 변경 및 읽기를 할 때 어떻게 이를 효과적으로 수행할 것인가 관한 것이다. 매핑 테이블의 표현 방법이란 매핑 테이블의 구조를 어떻게 할 것인가를 결정하는 것이다. 일반적으로 성능향상을 위해서 매핑 테이블을 하나의 디스크에 모두 저장하지 않고 분할저장해서 부하를 분산하게 된다. 이때 분할 방법과 테이블의 엔트리의 표현 방법에 따라서 전체 볼륨 관리자의 성능이 좌우된다. 다음부터 이 두 가지 고려사항에 대해 자세히 기술하고 스냅샷과 볼륨 재구성 기능에 대한 설계 내용도 같이 설명한다.

3.1.1.매핑 테이블의 관리 방법

이 논문에서 설계하는 매핑 관리자는 테이블 기반 방법을 기본으로 한다. 테이블 기반의 매핑 관리자는 매핑 테이블을 디스크에 저장하고 매핑을 수행하거나 매핑 테이블의 내용을 변경할 때 디스크의 해당 블록을 접근한다. 여기서 생각해 보아야 할 점은 SAN에 부착된 다중 호스트들이 매핑 테이블을 어떻게 접근해서 변경 및 읽기를 수행하는 가이다. I/O를 처리하기 위해서는 반드시 매핑 테이블을 접근해야 하므로 매핑 테이블의 변경 및 읽기를 어떻게 하느냐에 따라서 I/O 성능에 중요한 영향을 미치게 된다. 다중 호스트가 볼륨을 공유하는 상황에서 매핑 테이블을 저장/관리하는 방법을 분류해보면 다음과 같이 세가지 정도로 분류해 볼 수 있다.

첫 번째 방법은 매핑 테이블을 포함한 볼륨 관리에 필요한 메타데이터만 관리하는 메타 서버를 두는 방법이다. 메타데이터를 하나의 서버가 관리하므로 구조가 간단하다. 이 방법에서는 메타서버가 여러 이유로 접근 불가능 상태가 되면 전체 볼륨 관리자가 동작할 수 없다. 이를 대비해서 메타서버는 항상 두 대 이상을 두고 고장이 발생하면 다른 한 대가 메타서버 역할을 대신한다. 하지만 여러 호스트들의 자유 공간 할당 요청을 한 호스트가 처

리해야 하므로 병목현상이 발생하고 호스트 부하가 커지는 단점이 있다. 이 방법에서 가장 문제가 되는 것은 여러 호스트들의 매핑 요청을 하나의 메타 서버가 처리해야 하므로 병목현상이 발생하고 호스트의 부하가 커진다는 점이다. 메타 서버가 과부하로 인해 처리 속도가 저하되면 전체적인 볼륨 관리자의 성능 저하로 이어진다.

두 번째 방법은 볼륨 관리에 참여하는 모든 호스트가 매핑 테이블 전체를 중복해서 관리하는 것이다. 즉, 모든 서버가 매핑 테이블 전체를 접근해서 읽기 및 변경이 가능하며 매핑을 위해서 다른 서버와 네트워크를 통한 통신을 수행할 필요가 없다. 또한 한 서버의 고장이 매핑 테이블을 접근하는데 영향을 주지 않으므로 높은 가용성을 제공한다. 하지만 이 방법에서는 모든 서버가 매핑 테이블 전체에 대해서 읽기 및 변경을 수행하므로, 중복된 매핑 테이블의 일관성을 유지하기 위해 모든 서버에 걸친 전역잠금을 수행해야 한다.

마지막 방법은 볼륨 관리에 참여하는 호스트들이 매핑 테이블을 중첩되지 않게 분할하여 관리하는 것이다. 매핑 테이블은 논리 주소의 순서에 따라서 분할이 된다. 즉, 각 분할된 매핑 테이블은 모두 연속된 논리 주소를 갖는다. 각 호스트는 자신이 관리하는 매핑 테이블만 접근할 수 있으므로 호스트들간의 전역잠금은 필요 없다. 하지만, 매핑을 수행할 때 해당 논리 주소가 자신이 관리하는 매핑 테이블의 범위를 벗어나게 되면 네트워크를 통해 다른 호스트에 매핑요구를 보내야 한다.

이 논문에서는 마지막 방법을 택한다. 일반적으로 SAN 환경에서 볼륨 관리에 참여하는 호스트들 사이에는 범용 네트워크인 LAN을 이용하지 않고 전용 네트워크를 이용하므로 통신으로 인한 매핑 지연이 크지 않을 것이고 매핑을 위해 주고받는 데이터의 양이 20Bytes 내외로 매우 작다. 또한 마지막 방법은 다른 방법들에 비해 훨씬 구현 및 관리가 간단하다.

3.1.2.매핑 테이블 표현방법

일반적으로 성능향상을 위해서 매핑 테이블을 하나의 디스크에 모두 저장하지 않고 분할저장해서 부하 분산을 시도한다. 이때 분할 방법과 테이블의 엔트리의 표현 방법에 따라서 전체 볼륨 관리자의 성능에 영향을 미치게 된다. 이 논문에서는 매핑 테이블의 표현 방법으로 네가지를 고려해 보았다.

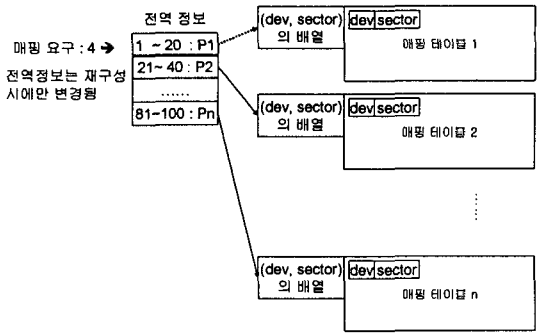

그림 3-1 매핑 테이블의 분할 관리

첫 번째로 고려해 볼 수 있는 것은 그림 3-1과 같은 매핑 테이블의 가장 기본적인 표현 방법이다. 즉, (device, sector)로 이루어지는 물리 주소들이 대응되는 논리 주소의 순서에 따라서 배열형태를 이루는 것이다. 이런 형태로 매핑 테이블을 표현하게 되면 원하는 논리 주소가 저장된 블록을 계산해 낼 수 있고 한번의 I/O로 매핑을 수행할 수 있다. 예를 들어서, 호스트1이 관리하는 매핑 테이블에는 논리 주소 100~199 까지 100개가 저장되어 있고 디스크의 한 블록에는 10개의 (device, sector)의 물리 주소가 저장될 수 있다고 가정하자. 매핑요구로 들어온 논리 주소가 155 라면 (155-100)/10 = 5가 읽어올 블록의 상대 주소가 되고 55-(5*10) = 5는 그 블록 내에서 몇 번째 물리 주소를 취해야 하는지 알 수 있다.

이 방법에 덧붙여서 희소 논리 주소 분할 방법을 들 수 있다. 즉, 일반적으로 볼륨 관리자의 상위로부터의 매핑요구는 순서적인 성향을 보이므로 위와 같이 논리 주소를 분할했을 경우 매핑요구에 대한 부하가 논리 주소의 앞부분을 담당하는 서버에 집중될 수 있다는 것이다. 이에 희소 논리 주소 분할 방법은 앞의 단점을 보완하기 위해 논리 주소 공간을 일정 범위를 가지도록 분할하는 방식이 아니라 논리 주소 공간을 띄엄띄엄 할당하는 방식이다. 하지만 희소 논리 주소 분할 방법을 적극 수용하지 못하는 이유는 일반적으로 매핑을 포함한 볼륨 관리자를 운용하기 위한 메타데이터는 디스크의 앞부분에 고정된 크기로 할당된 영역에 저장되는데, 이때 희소 분할 방법을 사용하는 상황에서 디스크가 추가되면 각 디스크에 저장되어야 할 논리 주소의 개수가 변경되면서 고정된 크기를 넘어설 수 있다는 것을 그 이유로 지적하고 있다. 이러한 문제는

각 디스크에 저장되는 매핑 테이블의 크기를 모두 같은 양으로 하지 않고 그 디스크에서 제공할 수 있는 물리 블록의 개수에 비례해서 매핑 테이블을 분할하여 해결할 수 있다. 그림 3-2가 그런 상황을 설명하고 있다. device 4 에 저장된 매핑 테이블에는 device 4가 12개의 물리블록을 가지고 있으므로 다른 디바이스에 비해서 4개의 논리 주소 매핑정보를 더 가지고 있다. 따라서, 매핑 요구가 device4에 더 집중될 수 있다.

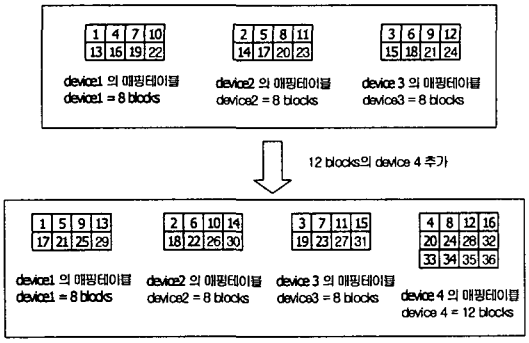

그림 3-2 희소 논리 주소 분할 방법

하지만 대부분의 파일시스템이나 데이터베이스 관리시스템에서는 논리 주소의 앞에서부터 순서적으로 사용하는 경향이 있다. 그리고, 33~36의 논리 주소들은 볼륨의 마지막 논리 주소들이므로 심각한 성능저하를 가져오지 않을 것이다.

이 방법에선 매핑 테이블이 논리 주소의 순서대로 분할되어 각 디스크에 저장되므로 특정 디스크가 어떤 이유로 일시적/영구적으로 접근 불가능의 상태가 된다면 전체 논리볼륨은 사용할 수 없게 된다. 이런 이유로 분할된 매핑 테이블을 Chained Declustering과 같은 방법을 통해 중복 저장하여 디스크 고장에 대비하여야 한다. 하지만 이것 역시 완벽한 해결책은 아니다. 중복되어 저장된 디스크가 모두 접근 불가능 상태이면 매핑은 수행될 수 없다.

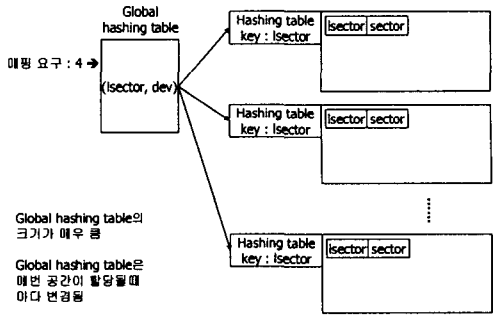

그림 3-3 물리 주소를 기준으로 매핑 테이블을 분할 관리

두 번째 방법은 위와 같은 점에 착안한 방법이다. 매핑 테이블을 분할할 때 물리 주소를 기준으로 분할을 할 수 있다. 즉, 특정 디스크에 저장되는 매핑정보들은 그 디스크 내의 물리블록들에 대한 매핑 정보만을 저장한다. 이 경우 특정 디스크에 저장된 매핑정보는 그 디스크의 물리블록들을 위한 것이므로 그 디스크가 접근 불가능 상태에 있을 때 접근 가능한 디스크에 대해서는 여전히 매핑이 가능하다. 이때 각 매핑 테이블이 연속된 논리 주소로 분할된 것이 아니기 때문에 매핑요구로 들어온 논리 주소를 매핑하기 위해 어떤 호스트에게 요청을 전달해야 하는지 알 수 없다. 이를 위해서는 이를 판별할 수 있도록 그림 3-3의 Global hashing table 같은 것이 존재해야 한다.

또한, 만일 논리 주소 4가 매핑 테이블 1에 존재해서 호스트 1에 요구를 전달했다 해도 어떤 블록을 읽어야 원하는 매핑정보를 얻을 수 있는지 알 수 없다. 이를 위해서는 다시 해싱 테이블과 같은 색인이 존재해야 한다. 해싱 테이블의 키는 논리 주소(lsector)가 된다.

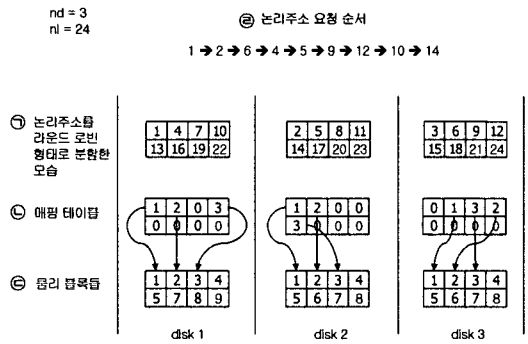

그림 3-4 희소 논리 주소 분할 방법

세 번째 방법은 두 번째 방법과 첫 번째에서 언급했던 논리 주소의 희소분할 방법을 적절히 혼합한 형태이다. 일반적으로 볼륨 관리자 상위에서의 I/O 요구는 논리 주소가 연속적인 경향을 띄게 되고 매핑을 위해서 볼륨에 있는 디스크들을 골고루 접근하게 된다. 이런 성질을 이용해서 다음과 같은 매핑 테이블 운용 방법을 생각해 볼 수 있다. 볼륨에 참여하는 디스크의 수가 nd라하고 전체 볼륨의 논리블록이 nl개라고 하자. 최초에 매핑 테이블을 구성할 때 nl개의 논리 주소를 nd개만큼 분할을 하는데 되도록 이면 인접한 논리 주소가 같은 디스크에 저장되지 않도록 디클러스터링 한다. 쉽게 생각해 볼 수 있는 것은 라운드 로빈 방법이다. 그렇게 되면 매핑 테이블이 분할된 모습은 그림 3-4와 같을 것이다. 그림에서 ㉠은 논리 주소 24 개를 라운드 로빈 방법을 기반으로 각 디스크에 분산한 모습이다. 이 상태에서 ㉣에서와 같은 순서로 논리 주소에 대한 쓰기 요청이 발생하면 자유공간 관리자는 적절히 논리 주소에 물리블록을 할당한다. 이때 제안하는 방법에서는 물리블록을 할당하는 디스크를 논리 주소가 저장된 디스크에서 할당한다. 예를 들어서 1 번 논리 주소에 대한 쓰기 요청이 최초로 발생하면 물리블록을 할당해 주어야 할 것이다. 물리블록을 할당할 디스크를 1번 논리 주소가 존재하는 disk1로부터 할당받는다. 이때 할당받은 물리블록은 1이 된다. ㉡을 보면 disk1의 매핑 테이블 영역에서 논리 주소 1에 해당하는 부분에 물리블록 1이 기록된 것을 볼수 있다.

이 방법의 장단점을 살펴보면, 우선 장점으로는 disk1에 저장된 매핑 테이블을 관리하는 호스트가 항상 disk1의 자유공간 할당도 수행하므로 추가적인 통신이 필요없다. 또한, 스트라이핑 볼륨에서 공간을 할당할 때 항상 다음에 물리블록을 할당 할 디스크가 어디인지를 알려주는 모든 호스트에 대한 전역 변수가 있어야 하는데 이에 대한 필요성도 사라진다. 또한, 일반적으로 매핑 테이블의 엔트리는 {device(2bytes), sector(8bytes)}로 이루어지며 크기는 10 bytes가 된다. 하지만 이 방법에서는 device를 별도로 기록 할 필요가 없다. 그리고, 각 디스크에 저장된 매핑 정보는 그 디스크에 대한 물리블록에 대한 것이므로 디스크 고장에 대비해서 다른 디스크에 매핑정보나 공간 할당 정보를 중복 저장할 필요가 없다.

단점으로는 먼저, 논리 주소를 디클러스터링 해놓았지만 논리 주소에 대한 요청이 순서적이지 않고 부분부분 나타날 경우 디스크 사용정도가 특정 디스크에 편중될 수 있다. 최악의 경우는 그림 3-4에서 보았을 때 논리 주소에 대한 최초 쓰기 요청이 1, 4, 7, .. 과 같이 device1에만 집중적으로 나타나는 경우이다. 이렇게 되면 스트라이핑 효과를 전혀 얻을 수 없게 된다. 하지만 실제 응용에서는 이런 상황이 아주 제한적으로 발생할 것이다.

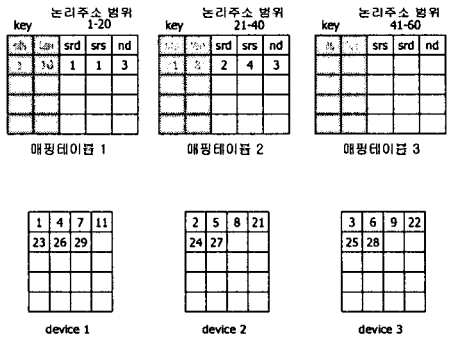

그림 3-5 범위를 이용한 매핑 테이블 표현 방법

마지막 네 번째 방법으로 논리 주소를 기준으로 매핑 테이블을 분할하되 매핑 정보를 모든 논리블록에 대해서 테이블에 기록하는 것이 아니고 범위를 이용해서 기록하는 것을 생각해 볼 수 있다.

그림 3-5는 스트라이핑 볼륨에 대해서 범위형태로 매핑 정보를 기록한 것이다. sls를 시작 논리섹터라 하고, srs를 시작 물리 섹터, srd를 시작 물리 디바이스, len을 논리 주소와 물리 주소가 연속되게 할당된 길이, dn을 논리볼륨에 포함된 디바이스 개수, devs를 스트라이핑에 참여하는 디바이스 번호들

이라고 하자. 매핑 테이블 1의 첫 번째 매핑레코드를 보면 sls=1,len=10, srd=1, srs=1, nd=3 으로 되어 있다. 이 매핑 정보가 의미하는 것은 논리 주소가 1부터 10이 연속적으로 할당되었고 이에 대응하는 물리 주소도 device 1의 1번 블록으로부터 차례로 device 2의 1, device 3의 1, device 1의 2, device 2의 2 .... 이런 순서로 10개가 연속 할당되었다는 것을 의미한다. 매핑 테이블 2의 레코드 역시 같은 형식으로 이해 할 수 있다. 이 경우에는 물리 주소가 device 2의 4, device 3의 4, device 1의 5, device 2의 5 ... 이렇게 8개가 연속적으로 할당되었다는 것을 의미한다.

매핑의 수행은 먼저 매핑요구로 들어온 논리 주소를 가지고 매핑 테이블이 저장되어 있는 디스크와 이를 관리하는 호스트를 결정한 후 매핑요구를 해당 호스트에 전달한다. 매핑요구를 받은 호스트는 매핑 테이블에 기록되어 있는 sls와 len을 키로 해서 이진탐색을 수행한 후 매핑요구 논리 주소를 포함한 레코드를 찾는다. 레코드를 찾으면 정보들을 가지고 적절한 물리 주소를 계산해 낼 수 있다. 이 때 각 호스트는 매핑 테이블의 내용을 메모리에 상주시킬 수 있다는 것이 전제가 된다. 스냅샷 생성 후 COW(copy on write)로 인한 매핑관계의 변화는 해싱 테이블을 이용해서 해결한다. 해싱 테이블은 디스크에 저장되게 되며 스냅샷 데이터 접근시에만 이용되므로 정상적인 I/O 수행에는 문제가 되지 않는다. 이 방법은 볼륨 관리자의 파일 시스템의 매핑요구로 볼륨 관리자에 전달되는 논리 주소가 어느 정도 연속적일 경우에 효과적으로 동작한다. 이 경우 각 호스트가 관리하는 매핑 테이블의 크기는 메모리에 상주시킬만큼 작을 것이고 매핑 테이블에 저장된 레코드들을 이진 탐색을 통해서 원하는 매핑 레코드를 얻을 수 있다. 만일 상위로부터의 매핑요구가 불규칙적으로 발생된다면 매핑 테이블의 크기가 커져서 첫 번째 방법에서 보다 오히려 매핑 테이블의 크기가 커진다. 스냅샷을 생성한 이후에 발생하는 COW에 대해서는 해싱 테이블을 이용해서 저장한다.

실제로 이 논문에서는 첫 번째 방법과 마지막 방법을 구현해 보았다. 마지막 방법의 경우는 이미 언급한데로 연속적인 논리 주소의 매핑요구가 발생할 때 효과적이다. 구현한 방법을 GFS 파일 시스템과 같이 사용해 보았다. GFS의 경우 논리주소 공간을 나름대로 분할하여 저널링 영역과 리소스 그룹으로 나누어 사용한다. 이 경우 매핑요구가 부분적으로 연속적이긴 하지만 전체적으로 보았을 때 일반적인 데이터 저장을 위한 매핑요구와 저널링을 위한 매핑요구가 혼재되어 나타나므로 매핑 레코드의 개수가 많아지고 메모리에 저장할 수 없는 상태까지 이르게 됨을 볼 수 있었다. 따라서 이 방법은 범용의 논리볼륨 관리자로 사용하기에는 문제가 있으며 파일시스템과 볼륨 관리자를 설계할 때 이런 부분을 고려하여 어느 정도 상호 의존적으로 구현한다면 매우 효과적으로 사용해볼 수 있을 것이다. 최종 구현에는 첫 번째 방법을 택하였다.

3.1.3.스냅샷의 설계

스냅샷 기능은 사용자가 원하는 시점의 볼륨이미지를 유지하여 추후에도 스냅샷 당시의 데이터를 참조하고자 할 경우 사용된다. 일반적으로 스냅샷을 생성할 때 원래 볼륨이 가지고 있는 매핑 테이블을 그대로 스냅샷 목적으로 사용하기 위해 복사한다. 그리고 이를 유지하기 위해서 COW를 이용한다. 앞에서 언급한 것처럼 SAN과 같은 대용량 환경에서는 매핑 테이블의 크기가 크므로 이를 복사하는데 상당한 시간이 소요된다. 스냅샷을 생성하는 동안에는 다른 I/O 연산을 허용하지 않으므로 스냅샷 생성 시간이 길어지는 것은 문제가 될 수 있다.

이에 대한 대안으로 스냅샷 생성중에 매핑 테이블을 복사하지 않고 원래 볼륨의 매핑 테이블을 공유하면서 COW에 의해서 변경되는 부분만 해싱 테이블에 저장하는 방법을 생각해 볼 수 있다. 이 방법을 이용하면 스냅샷을 생성하는 시간이 상대적으로 매우 짧고 초기의 저장공간 사용량이 작다. 하지만 스냅샷 이후에 볼륨의 모든 부분이 변경이 되면 오히려 매핑 테이블을 복사하는 것 보다 저장공간을 더 소요하게 될 수 있다. 이 외에도 스냅샷 데이터를 접근할 때 매핑 테이블과 해싱 테이블을 모두 접근해 보아야 하므로 스냅샷 데이터를 접근하는데 상대적으로 시간이 더 소요될 수 있다. 이 논문에서는 두 스냅샷 방법을 모두 설계 및 구현하였다.

3.1.4.볼륨 재구성 방법 설계

재구성 방법의 설계 내용에 대해서 설명하기 전에 이후의 재구성 방법은 모두 스트라이핑 볼륨을 기준으로 설명한 것이다. 다른 RAID level의 볼륨들은 스트라이핑 볼륨의 재구성 방법을 그대로 또는 조금 수정해서 바로 적용할 수 있다. 그리고 디스

크의 단편화가 일어나지 않도록 재구성 한다는 전제하에 수행한다.

온라인 볼륨 재구성이란 디스크를 추가하거나 삭제할 때 시스템을 정지시키지 않고 변경된 내용을 포함하도록 논리볼륨을 재구성하는 것을 말한다. 이 논문에서 설계하는 볼륨 재구성 방법은 세가지로 나눠볼 수 있다.

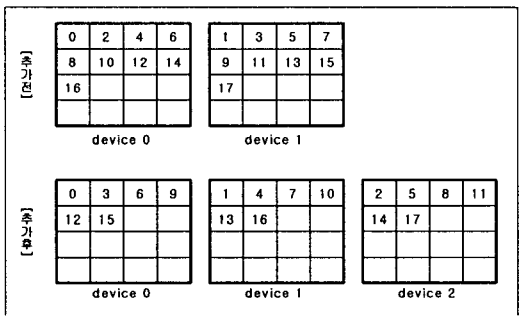

그림 3-6 재구성 방법1을 적용한 한개의 디바이스가 추가된 재구성

첫 번째 방법은 그림 3-6과 같이 기존 디바이스의 모든 데이터를 모든 디바이스내에 완전히 스트라이핑 되도록 이동시키는 방법이다. 이때 기존 디바이스내의 모든 데이터가 이동하므로 재구성 속도가 느리지만, 재구성 후의 I/O 성능 향상이 높은 방법이다. 또한 이 방법은 재구성 중 다른 I/O 요구를 수용하기 위해 해당 블록과 연관된 블록들을 처리해야 하는데, 이때 연관된 블록의 수가 많다는 단점을 가지고 있다.

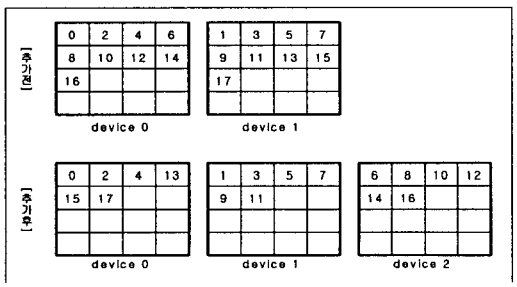

그림 3-7 재구성 방법2를 적용한 한개의 디바이스가 추가된 재구성

두 번째 방법은 그림 3-7과 같이 기존 디바이스의 일부 데이터만 이동시키는데, 가능하면 연속된 데이터가 인접하지 않도록 이동시키는 방법으로, 재구성 후의 I/O 성능 향상은 앞의 방법에 비해 못하지만 재구성 속도가 빠르고 서로 연관된 블록의 수가 적어 재구성 중 다른 I/O 요구를 수용하기가 비교적 용이한 방법이다.

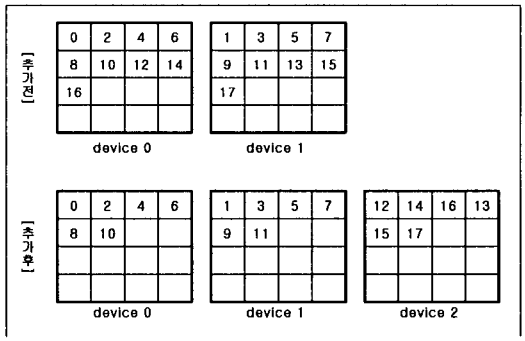

그림 3-8 재구성 방법3을 적용한 한개의 디바이스가 추가된 재구성

세 번째 방법은 그림 3-8과 같이 새 디바이스에 이동할 데이터를 기존 디바이스에서 나누어 이동시키며, 앞 디바이스의 데이터부터 순차적으로 이동한다. 그림에서 보다시피 이동할 데이터가 적기 때문에 재구성 속도가 빠르고 서로 연관된 블록이 없기 때문에 재구성 중 I/O를 수용하기 가장 쉬운 방법이다. 하지만 스트라이핑을 전혀 고려하지 않았기 때문에 재구성 후 I/O 성능향상의 정도가 매우 떨어진다.

4.매핑 관리자 구현

이 장에서는 제3장에서 설계한 매핑 관리자의 구현 방향에 대해서 기술한다. 제3장에서는 매핑 테이블을 사용하는 매핑 기법과 혼합 기법, 스냅샷, 재구성에 대한 여러 가지 방법을 제안하였다. 앞에서 언급했듯이 SAN의 논리 볼륨을 위한 매핑 관리자는 매핑관계가 동적으로 변하는 스냅샷이나 재구성과 같은 상황을 처리해 줄 수 있어야 한다. 따라서 이 논문에서 제시하고 있는 매핑 관리자는 정상 매핑뿐만 아니라 스냅샷이나 재구성을 지원하도록 구현하고자 한다. 제3장에서 언급했던 논리주소를 기준으로 매핑 테이블을 각 디스크에 분할하여 저장하고 분할된 매핑 테이블을 전담 관리하는 호스

트를 두는 매핑 방법을 적용하며 스냅샷과 재구성을 지원하도록 구현하고자 한다.

실험에 사용한 플랫폼은 Linux(Kernel version 2.4.9)의 운영체제와 P-3 450 , 128 Mbytes의 주기억 공간을 가지고 있다. 사용한 컴파일러는 GCC 2.7.1 이다.

이장에서는 매핑 관리자 및 자유공간 관리자[5, 6]를 구성하는 함수로 어떤 것들이 있고 이들이 어떤 호출관계를 갖고 있는지 설명하고 각 방법에 대한 구현 내용을 자세히 기술한다.

4.1.함수의 구성 및 호출 관계

이 논문에서 제시한 매핑 관리자를 구현하기 위한 함수들의 호출 관계를 그림으로 표현하면 그림 4-1 과 같다.

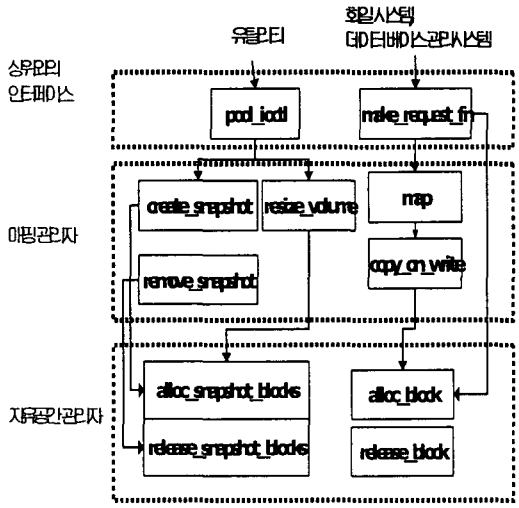

그림 4-1 구현한 매핑 관리자 및 자유공간 관리자의 함수 호출관계

설계한 매핑 테이블을 이용하는 볼륨 관리자를 구성하는 주요 함수들로는 논리볼륨 관리자의 유틸리티를 작성할 수 있게 해주는 pool_ioctl()과 상위 파일 시스템이나 데이터베이스 관리시스템과의 인터페이스를 위한 pool_make_request_fn()이 있다. 그리고, 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 그림과 같은 호출 관계를 유지하고 있다.

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 호출해야 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_block()에서는 d_bsize 만큼의 블록을 할당해서 그 시작 물리주소를 반환한다. alloc_bloc

k() 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   {
2       /* 상위레벨에서 자유 공간 요청이 들어온다;
        */
4       device = 할당할 디바이스를 선정한다.
5       선정된 device에서 자유 공간 정보(FS_MGR)를
        가져온다
6       block = FS_MGR의 next_avail_blk;
7       if (할당후 next_avail_blk
           < next_avail_snapshot_blk)
8       {
9           bitmap에 해당 블록에 대한 비트를 1로
            설정한다.
10          device와 block을 반환한다
11      }
12      else
13          alloc_block();
14      return;
15  }
```

그림 4-4 정상 공간 할당 요청에 대한 처리 절차

alloc_block() 함수는 새로운 공간을 할당하여 할당한 디바이스와 블록번호를 반환하는 역할을 한다. 입력값은 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. 5행에서는 선정된 디바이스에 해당하는 Freespace_Manager를 얻어온다.

Freespace_Manager로부터 해당 디바이스에 대한 총 블럭의 수, 사용된 블록의 수를 얻을 수 있다. Freespace_Manager로부터 할당할 블록 번호를 얻어온다. 만약 할당 후 next_avail_blk의 위치가 스냅샷 공간을 나타내는 next_avail_snapshot_blk보다 작다면 정상적인 할당이 이루어질 수 있는 단계로 8번행을 수행하게 된다. 8번행에서는 해당 디바이스에서 해당 블록에 대한 비트를 비트맵 공간에 1로 설정하는 단계이다. 성공적으로 8번행이 수행이 되면 할당한 디바이스와 블록 주소를 반환한다. 만약 해당 디바이스에 자유 공간이 없는 경우에는 alloc_block을 재 호출하여 다른 디바이스로부터 공간 할당을 요청한다.

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 그림 4-5 의 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 mknod()를 호출하여 snapshot_name의 디바이스를 생성한다.

```
int SS_snapshot(pool_t *pp, pool_t *np,
    char *ss_name);
int ss_copy_map(kdev_t dev, uint64 dest
    ination_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

pool_ioctl()에서 스냅샷을 생성할 때는 그림의 SS_snapshot()을 호출한다. 이전에서 언급한대로 스냅샷을 생성할때는 두 가지 방법이 있다. 하나는 매핑테이블을 복사하는 방법이고 다른 하나는 해쉬테이블을 이용하는 방법이다. 각 방법의 생성 과정을 그림 4-6 과 그림 4-7 에서 보여준다

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

그림 4-6 스냅샷-1의 SS_snapshot()

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

그림 4-9 스냅샷-2의 SS_release_snapshot()

스냅샷이 생성된 후에 해당 볼륨에 쓰기 요구가 발생하면 COW를 수행해야 한다. 해당 쓰기 요구가 스냅샷 생성 이전 시점에 있었던 블록에 대한 변경이라면 COW를 수행해야 한다. 스냅샷-1 과 스냅샷-2의 COW 수행절차를 그림 4-10 과 그림 4-11 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 4-10 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 4-11 스냅샷-2의 SS_COW_snapshot()

다음은 재구성에 대한 구현 내용이다. 재구성의 수행절차는 다음과 같다. 사용자는 추가되는 디바이스에 대한 정보를 담은 구성 화일과 함께 재구성 유틸리티를 실행한다. 재구성 유틸리티는 ioctl()을 호출하여 커널내부의 pool_ioctl()로 재구성 정보를 전달하면 커널내부의 재구성이 시작된다. 그림 4-12 는 재구성에 필요한 주요 관련 함수들을 나열한 것이다. st_resize_volume()이 재구성에서 지원하는 인터페이스이며, 나머지 함수들은 이 인터페이스내에서 호출되어 구성이 된다.

```
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 4-12 재구성 관련 주요 함수들

먼저 재구성을 수행하기 위해 볼륨 전체에 대한 쓰기 모드의 잠금을 설정한다. 다음으로 그림의 make_mvlist()를 호출하여 재구성 시 물리블록에 대한 논리주소를 반환하기 위해 물리주소 대 논리주소 테이블을 작성한다. 다음 단계는 그림의 resize_exec()를 호출하여 재구성으로 이동될 블록이 어디서 어디로 이동해야 하는지를 기록한 재구성 테이블을 재구성 규칙에 의해 작성하고, 이 테이블을 참조하여 블록의 이동을 시작한다. 블록을 이동하면서 그림의 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 4-13 은 재구성 수행시 옮겨질 물리블록에 대한 매핑 테이블 정보를 변경해야 하는데, 물리블록에 대한 논리주소를 참고하기 위해 물리주소 대 논리주소 테이블을 구성하는 함수이다.

```
1  make_mvlist()
2  {
3  for(풀내의 모든 디바이스에 대해서)
4  {
5  if(추가된 디바이스 이면) break;
6  매핑된 블록의 개수 계산;
7  매핑된 블록에 대한 물리주소 대 논리주소 테이블
     영역 할당;
8  for(디바이스내의 모든 블록에 대해서)
9  {
10 블록의 물리주소를 테이블내의 물리주소에 설정;
11 }
12 테이블 내용을 디바이스에 저장;
13 }
14 for(풀내의 모든 디바이스에 대해서)
15 {
16 if(추가된 디바이스 이면) break;
17 디바이스내의 매핑테이블 접근;
18 for(매핑테이블내의 모든 논리주소에 대해)
19 {
20 논리주소에 매핑된 물리주소 획득;
21 물리주소에 해당하는 물리주소 대 논리주소
     테이블을 검색;
22 물리주소 대 논리주소 테이블의 나머지 논리주소
     영역 설정;
23 }
24 테이블 내용을 디바이스에 저장;
25 }
26 }
```

그림 4-13 재구성 테이블 생성

```
1  resize_exec()
2  {
3    재구성 규칙에 의거해 재구성 테이블 작성;
4    for(재구성 테이블의 모든 블록에 대해서)
5    {
6      물리주소 대 논리주소 테이블 접근;
7      for(물리주소 대 논리주소 테이블의 모든
           내용에 대해서)
8      {
9        if(재구성 블록의 물리주소와 일치하다면)
10       {
11         재구성 블록의 물리주소에 해당하는
             논리주소 획득;
12         break;
13       }
14     }
15     논리주소를 포함한 매핑 테이블 접근;
16     논리주소에 매핑된 물리주소를 검색후 이동될
         물리주소로 변경;
17     재구성을 테이블에 의해 재구성 블록 이동;
18   }
19   재구성 테이블 삭제;
20   물리주소 대 논리주소 테이블 삭제;
21 }
```

그림 4-14 재구성 수행 절차

그림 4-14 는 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 실행이 완료되면 재구성 테이블 및 물리주소 대 논리주소 테이블을 삭제하면서 재구성이 끝나게 된다.

5.결 론

이 논문에서는 논리볼륨 관리자중에서 특히 매핑 관리자의 기능향상에 대한 방법을 찾고 이를 반영한 매핑 관리자를 설계하고 구현하였다. 기존의 수식 기반 매핑 방법은 스냅샷이나 볼륨 재구성과 같은 동적인 환경에 유연하게 대처하지 못하였다. 제안한 매핑관리자에서는 스냅샷이나 볼륨 재구성 기능을 지원하기 위해 매핑테이블 기반의 매핑 방법을 제시하였다. 스냅샷이나 볼륨 재구성 기능을 위해 매핑 테이블을 이용하는 것은 가장 일반적인 방법이지만 매핑 테이블 기반의 방법에서는 매핑정보 및 자유공간 정보 같은 메타 데이터를 관리하기 어렵다. 이를 위해 매핑 테이블을 쉽게 관리하면서 빠르게 매핑 요구를 처리할 수 있는 방법에 초점을 두어 매핑 관리자를 다양한 각도로 설계하였다. 또한 매핑 관리자의 스냅샷 및 재구성에 대해서 다양

한 방법을 제시하고 구현하여 비교해 보았다.

향후 연구에서는 본 논문에서 구현한 매핑 관리자를 설치하고 다양한 실험을 통해 성능향상정도를 측정하는 성능평가를 수행하도록 하겠다.

참 고 문 헌

[1] Steven R. Soltis, Thomas M. Ruwart and Matthew T. O'Keefe, "The Global File System," In Proceedings of the 5th NASA Goddard Conference on Mass Storage Systems and Technologies, Vol. 2, pp. 319-342, March 1996.

[2] Edward K.Lee and Chandramohan A.Thekkath, "Petal : Distributed Virtual Disks," In Proceedings of the 7th International Conference on ASPLOS, pp. 84-92, October 1996.

[3] P. R. Wilson, M. S. Johnstone, M. Neely and D. Boles, "Dynamic storage allocation: A survey and critical review," In Proceedings of International Workshop on Memory Management (IWMM'95), Vol. 986 of Lecture Notes in Computer Science, (Kinross, Scotland), pp. 1-116, September 1995.

[4] Daniel Pierre Bovet and Marco Cesati, "Understanding the Linux Kernel," pp. 686-721, O'Reilly, 2001.

[5] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto and G. Peck, "Scalability in the XFS file system," In Proceedings of the USENIX 1996 Technical Conference, pp. 1-14, January 1996.

[6] Yoshitake Shinkai, Yoshihiro Tsuchiya, Takeo Murakami and Jim Williams, "HAMFS File System," In Proceedings of 18th IEEE Symposium on Reliable Distributed Systems, pp. 190-201, October 1999.

[7] Yoshitake Shinkai, Yoshihiro Tsuchiya, Takeo Murakami and Jim Williams, "Alternatives of Implementing a Cluster File Systems," In Proceedings of the 17th IEEE Symposium on Mass Storage Systems, pp. 163-178, March 2000.