SAN 논리볼륨 관리자를 위한 매핑 기법
(A Mapping Method for a Logical Volume Manager in SAN Environment)

남 상 수 † 송 석 일 †† 유 재 수 ††† 김 창 수 †††† 김 명 준 †††††
(Sang Su Nam) (Seok Il Song) (Jae Soo Yoo) (Chang Soo Kim) (Myung Joon Kim)

요 약 높은 가용성, 확장성, 시스템 성능의 요구를 만족시키기 위해 SAN(Storage Area Network)이 등장했다. 대부분의 SAN 운영 S/W들은 SAN을 보다 효과적으로 활용하기 위해서 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 논리주소를 물리주소로 매핑 시킴으로서 저장장치 가상화를 실현한다. 이 논문에서는 논리볼륨 관리자를 위한 효율적이고 유연한 매핑기법을 설계하고 구현한다. 더불어 매핑 테이블 기반 매핑 방법에서 유연한 매핑을 돕기 위한 자유공간 관리기법을 설계하고 구현한다. 이 논문의 매핑기법은 특정 시점의 볼륨이미지를 유지할 수 있는 스냅샷과 시스템을 정지시키지 않고 SAN에 저장장치를 추가 또는 삭제할 수 있는 온라인 재구성 기능을 지원한다. 또한 이 논문에서 제안한 기법에 대한 성능 평가를 수행하여 제안하는 기법이 매핑 관리자로서 의미가 있음을 보인다.

키워드 : SAN, 논리볼륨, 매핑기법

Abstract SAN(Storage Area Network) was developed in response to the requirements of high availability of data, scalable growth, and system performance. In order to use the SAN more efficiently, most of the SAN operating software supports storage virtualization concepts that allow users to view physical storage devices of the SAN as a large volume. 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 the logical volume manager. Additionally we also design and implement a free space management method for flexible mapping. Our mapping method 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 and from the SAN even while the system is running. To justify our mapping method, we compare it with the mapping method of the GFS(Global File System) through various experiments.

Key words : SAN, logical volume, mapping method

1. 서 론

폭발적으로 증가하는 대량의 정보를 효율적으로 공유하고 고속으로 서비스하기 위하여 데이타를 중심으로 하는 새로운 개념의 컴퓨터 시스템 환경이 도래하였다. 가장 주목할 만한 것은 서버에 개별적으로 연결되던 저장 시스템을 광 채널과 같은 고속의 전용 네트워크에 직접 연결하여 중앙 집중적인 저장 시스템의 관리를 가능하게 하고, 서버를 거치지 않고 네트워크에 연결된 저장장치를 직접 접근하도록 하는 SAN을 들 수 있다[1].

SAN은 기존의 서버 중심 시스템 환경이 가지고 있는 대용량의 데이타를 저장하고 관리하는데 발생하는 문제를 해결해 줄 수 있는 최선의 방안으로 인식되고 있다. SAN 환경을 보다 효과적으로 활용할 수 있기 위해서는 SAN의 다양한 저장시스템을 사용자에게 투명하게 제공할 수 있어야 한다. 이를 위해 대부분의 SAN

† 비 회 원 : LG산전 전력연구소 연구원
ssnam@lgis.com
†† 비 회 원 : 충주대학교 컴퓨터공학과 교수
sisong@chungju.ac.kr
††† 종신회원 : 충북대학교 전기전자컴퓨터공학부 교수
컴퓨터 · 정보통신연구소 부장
yjs@cbucc.chungbuk.ac.kr
†††† 비 회 원 : 한국전자통신연구원 컴퓨터시스템연구부 연구원
cskim7@etri.re.kr
††††† 종신회원 : 한국전자통신연구원 컴퓨터시스템연구부장
joonkim@etri.re.kr
논문접수 : 2002년 10월 23일
심사완료 : 2003년 9월 15일

운영 S/W들은 물리적 저장장치들을 사용자에게 투명하게 제공하도록 저장장치 가상화 개념을 지원한다. 저장장치 가상화는 사용자가 SAN의 저장장치들의 물리적 구성을 몰라도 쉽게 저장시스템을 사용할 수 있게 한다. 이러한 저장장치 가상화는 논리볼륨 관리자에 의해서 이루어진다. SAN의 논리볼륨 관리자는 단순한 가상화 기능 외에도, 특정 시점의 볼륨이미지를 유지할 수 있는 스냅샷과 시스템을 정지시키지 않고 SAN에 저장장치를 추가 또는 삭제할 수 있는 온라인 재구성 기능을 지원해야 한다.

논리볼륨 관리자의 가장 핵심적인 기능은 사용자들이 특정 논리주소 블록에 대한 입출력을 요구하면 적절히 요구한 논리블록을 실제 저장장치의 물리블록으로 변환하는 매핑 기능이다. 기존의 논리볼륨 관리자들의 매핑 기법은 크게 수식 기반의 매핑기법과 테이블 기반의 매핑기법으로 나누어볼 수 있다. 수식 기반의 매핑기법은 계산식을 이용해 매핑을 수행한다. 이 방법은 매핑이 빠르고 구현이 간단하지만, 스냅샷이나 온라인 재구성과 같이 매핑 관계를 변경해야 하는 연산들을 유연하게 처리하기 어렵다[2].

이에 반해, 테이블 기반 매핑기법은 논리블록과 물리블록의 일대일 매핑 관계를 테이블로 저장해 놓고 이를 통해 매핑을 수행한다. 따라서 수식 기반보다 매핑 요구처리가 느리고 SAN과 같은 대용량 환경에선 매핑 테이블의 크기가 커서 관리하기 어렵지만, 매핑 관계가 변하는 상황에 유연하게 대처할 수 있다[2]. 특히, 테이블 기반의 매핑기법은 스냅샷, 온라인 재구성 요구 시 실제 저장장치의 각 블록의 사용유무를 판단하여, 적절한 물리블록을 할당/해제할 수 있도록 해주는 자유공간 관리기법이 필요하다.

SAN의 논리볼륨 관리자는 가상화, 스냅샷, 온라인 재구성 기능을 지원하기 위해 수식 기반의 매핑기법보다는 테이블 기반의 매핑기법이 선호되고 있다. 이에 본 논문에서는 리눅스상의 SAN 볼륨관리자인 GFS(Global File System) 풀(pool)[3]의 매핑 관련 부분에, 스냅샷과 온라인 재구성 기능과 같이 매핑관계가 동적으로 변하는 상황을 효과적으로 지원할 수 있는 테이블 기반의 매핑 방법을 설계하고 구현한다. 이 논문은 또한 유연한 매핑을 돕기 위해, 각 블록의 사용여부를 한 비트에 대응시켜서 표시하는 비트맵 구조의 자유공간 관리기법을 설계하고 구현한다.

이 논문의 구성은 다음과 같다. 2장에서는 논리볼륨 관리자의 매핑 관리와 자유공간 관리에 대한 기존 연구에 대해 기술한다. 3장에서는 매핑 관리자의 설계 목적과 설계 시 고려해야할 사항을 제시하고 이에 따른 설계 내용을 기술한다. 4장에서는 3장에서 설계한 매핑 관리자의 구현에 대한 내용을 기술한 후 기존의 매핑기법과의 비교를 통해 제안한 매핑 관리자의 우수성을 입증하고, 5장에서 결론을 맺는다.

2. 관련 연구

이 장에서는 기존에 제안된 논리볼륨 관리자들의 매핑기법과 자유공간 관리기법에 대해서 기술한다. 이전의 논리볼륨 관리자들의 매핑기법을 살펴보면 크게 수식 기반의 매핑기법과 테이블 기반의 매핑기법으로 나누어진다. 이에 수식 기반의 매핑과 테이블 기반의 매핑 수행하는 논리볼륨 관리자들을 분류한 후 각 논리볼륨 관리자의 특징에 대해 기술한다. 자유공간 관리기법은 기존의 논리볼륨 관리자와 기존 파일 시스템에서 이용했던 기법에 대해 조사/분석한 내용을 기술한다.

먼저, 수식 기반의 매핑기법을 수행하는 기존의 논리볼륨 관리자에 대해 설명한다. 수식 기반의 매핑기법을 수행하는 대표적인 SAN 논리볼륨 관리자로서 미네소타 대학에서 개발한 GFS 풀을 들 수 있다. GFS는 공유디스크 환경에 적합하게 제안된 리눅스 기반의 파일 시스템이다. GFS의 기본 구조는 각각의 자원 그룹으로 구성되며, 각각의 자원그룹에는 RG블록, 데이타 비트맵, Dinode 비트맵, Dinode로 나누어진다. SAN환경에 적합한 GFS모형은 그림 1과 같다. 그림 1에서 SAN에 물려있는 GFS 클라이언트(Client)는 SAN 패브릭(fabric)을 구성하고 구성된 SAN 패브릭을 통해 저장장치에 접근한다.

GFS 풀은 매핑 시 별도의 매핑 테이블을 두지 않고 수식 기반의 매핑을 수행하며, 자유공간 관리가 필요하지 않다. 따라서 빠른 매핑이 가능하고 구현이 간단하지만, 스냅샷이나 온라인 재구성 기능과 같이 볼륨 구성이 동적으로 변해서 매핑 관계가 바뀌어야 하는 환경을 수용하지 못하는 단점이 있다. 이런 이유로 풀은 저장장치

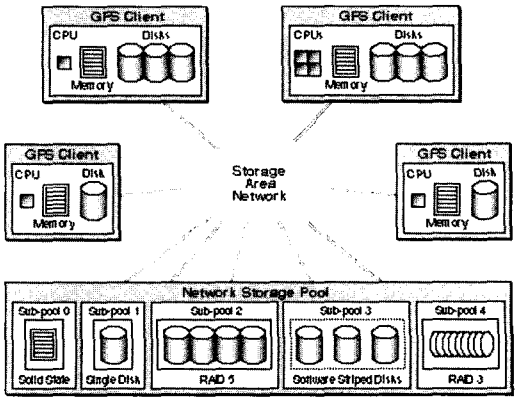

그림 1 SAN 환경에서의 GFS 모형

추가 시 논리볼륨의 크기는 확장되지만, 저장장치의 부하 균등을 위한 저장장치간 데이타 이동이 전혀 고려되지 않았다. 추가적으로, 풀은 논리볼륨에 참여한 저장장치들과 논리볼륨 정보와 같은 메타 데이타를 각 저장장치의 일정 영역에 중복 저장하며, 이런 메타 데이타의 일관성을 유지하기 위해 디바이스 락(device lock)이라는 잠금기법을 활용하고 있다.

테이블 기반의 매핑기법을 사용하고 있는 기존의 논리볼륨 관리자는 다음과 같다. 먼저, 테이블 기반의 매핑기법을 사용하고 있는 SAN 기반의 논리볼륨 관리자로서 SANtopia[4]를 들 수 있다. SANtopia의 매핑 테이블의 구조는 (device, sector)로 구성된 물리주소가 대응되는 논리주소의 순서에 따라서 배열형태를 이룬다. 예를 들어, 매핑 테이블에는 논리주소 0~99까지의 100개가 저장되어 있고 저장장치의 한 블록에는 10개의 (device, sector)의 물리주소가 저장된다고 가정하면, 논리주소 55는 5번 블록내의 6번째 물리주소에 매핑 된다. 더불어 SANtopia는 각 블록의 사용유무를 관리하여 스냅샷 및 온라인 재구성 시 적절한 물리블록을 할당 또는 해제 할 수 있도록 자유공간 관리기법을 수행한다.

또한, SANtopia에서는 논리주소의 분할방법으로 전체 논리주소 공간을 앞에서부터 순서대로 분할하는 범위 분할방법을 사용한다. 이러한 분할방법은 구현이 간단하고 쉬운 장점을 갖는다. 그러나, 볼륨 관리자 상위에서 각 볼륨을 사용하는 유형을 살펴보면 대개는 논리주소 공간을 앞에서부터 차례로 사용하는 경향이 많다. 이러한 경우 매핑 정보의 접근이 논리 파티션의 순서 상 앞쪽에 몰리는 경향이 생기게 된다. 따라서 전체 디스크의 활용도 및 해당 디스크 액세스에 할당된 호스트에 부하가 많다는 단점을 갖는다. 이런 범위 분할방법의 단점은 희소 분할방법을 적용함으로써 해결할 수 있다. 희소주소 분할방법이란 논리주소 공간을 일정 범위를 가지도록 분할하는 방식이 아니라 논리주소 공간을 사이를 두고 할당한다. 이 방식을 사용하면 범위 분할방식에서 몇 개의 논리 파티션에 액세스가 집중되거나 액세스를 위해 할당되는 호스트의 부하 집중을 방지할 수 있게된다. 단지 범위 분할할당에 비해 구현이 다소 복잡하다는 단점을 갖는다.

다음으로, SAN을 기반으로 하지는 않지만 논리적 저장장치 가상화를 지원하는 Petal[5]도 테이블 기반의 매핑기법을 이용한다. Petal은 네트워크로 연결된 여러 호스트에 부착된 저장장치를 가상화하여 하나의 논리 저장장치로 클라이언트에게 제공하는 시스템이다. 그림 2에서 Petal의 구조를 보여주고 있다. Petal은 네트워크에 부착된 서버의 저장장치들을 마치 하나의 저장장치처럼 가상화 한다. Petal에서는 매핑 테이블을 이용하여 온라인 백업을 위한 스냅샷과 온라인 볼륨 재구성을 지원하고 있다. 하지만 Petal은 네트워크를 통해 연결된 여러 서버가 가지고 있는 저장장치들을 가상화 해주는 시스템이지 SAN과 같은 저장장치를 위한 파일 시스템은 아니다[3].

테이블 기반 매핑기법에서는 실제 저장장치의 각 블록의 사용유무를 판단하여 매핑, 스냅샷, 온라인 재구성 요구 시 적절한 물리블록을 할당 및 해제할 수 있도록 자유공간 관리를 수행한다. 이미 언급한 SANtopia에서는 논리볼륨 관리자를 위한 자유공간 관리기법으로 비트맵을 이용한다.

또한, 논리볼륨 관리자를 위한 자유공간 관리기법은 아니지만 기존의 파일 시스템에서 행해지는 자유공간 관리기법을 분석해 논리볼륨 관리자에 도입할 수 있다. 기존 파일 시스템의 자유공간 관리 정책으로 여러 가지가 소개되었다. 대표적인 것으로 선형 리스트 구조를 이용한 순차적 적합, 트리 구조를 활용한 색인 적합, 비트맵 구조를 사용한 비트맵 적합 등이 있다[6]. 이러한 정

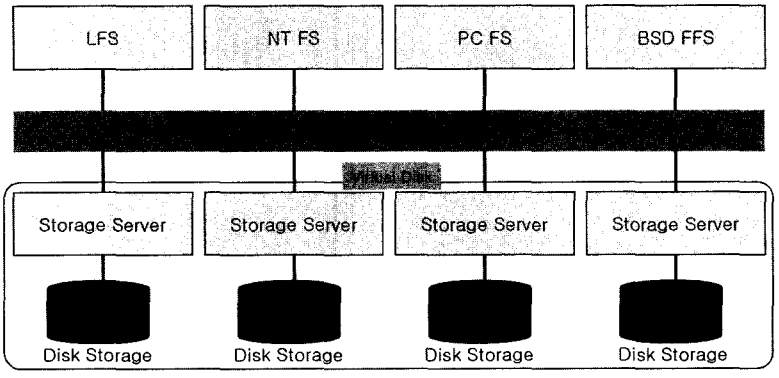

그림 2 Petal의 구조

책 중에 GFS나 EXT2[7]와 같은 파일 시스템은 비트맵 기반의 자유공간 관리를 하고 있다. XFS[8]나 Fujitsu 사의 SafeFILE 제품의 파일 시스템인 HAMFS[9]의 경우는 B-tree를 활용한 색인 적합 방법을 사용하고 있다.

3. 매핑 관리자 설계

이 논문에서는 SAN 논리볼륨 관리자를 위한 효율적이고 유연한 매핑방법으로 테이블 기반의 매핑기법을 사용한다. 테이블 기반의 매핑은 수식 기반의 매핑방법에 비해 매핑 요구 처리가 느리고 매핑 테이블을 관리하기가 어렵지만, 매핑 관계가 변하는 스냅샷과 온라인 재구성과 같은 동적인 환경에 유연하게 대처할 수 있다.

또한 SAN 환경에서는 다중 호스트가 SAN에 부착된 저장장치를 공유한다. 따라서 다중 호스트가 공유된 저장장치와 저장장치에 저장된 메타 데이타를 접근하여 변경 및 읽기를 수행한다. SAN을 위한 논리볼륨 관리자는 하나의 호스트에 위치하지 않고 여러 호스트에 분산되고, 호스트들은 서로 적절한 메시지를 주고 받으면서 논리볼륨을 유지한다. 테이블 기반의 매핑을 위해서는 매핑 테이블이나 자유공간 정보를 논리볼륨에 참여하는 여러 호스트들이 동시에 접근해서 변경 및 읽기를 수행해야 한다. 따라서 메타 데이타들에 대한 호스트들의 접근 방법 및 메타 데이타의 저장위치, 매핑 테이블의 구조를 어떻게 관리하는지에 따라 SAN 논리볼륨 관리자의 성능이 좌우된다. 이 논문에서는 이러한 점을 고려하여 메타 데이타들의 관리가 쉽고 빠르게 매핑 요구를 처리할 수 있도록 매핑 관리자를 다양한 각도로 설계한다. 또한 제안하는 매핑 관리자는 특정 파일 시스템에 의존적이지 않는 방향으로 설계한다.

3.1 메타 데이타의 유지 관리

테이블 기반의 매핑 관리자는 매핑 테이블과 자유공간 정보와 같은 메타 데이타를 저장장치에 저장하여 관리한다. 하나의 볼륨을 여러 호스트가 공유하는 상황에서는 메타 데이타 역시 각 호스트에 의해서 공유된다. 다중 호스트가 메타 데이타를 접근하여 읽거나 변경하기 때문에 메타 데이타에 대한 입출력을 어떻게 효과적으로 처리하느냐에 따라서 매핑 관리자의 성능이 좌우될 수 있다. 다중 호스트가 볼륨을 공유하는 상황에서 메타 데이타를 저장/관리하는 방법을 다음과 같이 세 가지로 나누어 볼 수 있다.

첫 번째는 매핑 테이블을 포함한 볼륨 관리에 필요한 메타 데이타만 관리하는 메타 서버를 두는 방법이다. 메타 데이타를 하나의 서버가 관리하므로 구조가 간단하다. 하지만 통신 장애와 같은 이유로 메타 서버가 접근 불가능한 상태가 되면 메타 데이타를 접근할 수 없어져서 전체 볼륨 관리자가 동작할 수 없다. 이를 대비해서 메타 서버는 항상 두 대 이상을 두고 고장이 발생하면 다른 한 대가 메타 서버 역할을 대신해야 한다. 하지만 여러 호스트들의 매핑 요청을 한 호스트가 처리해야 하므로 병목현상이 발생하고 해당 호스트의 부하가 커지는 단점이 있다. 이렇게 메타 서버가 과부하로 인해 처리 속도가 저하되면 전체적인 볼륨 관리자의 성능 저하로 이어진다.

두 번째 방법은 볼륨 관리에 참여하는 모든 호스트가 메타 데이타 전체를 중복해서 관리하는 것이다. 즉, 모든 서버가 메타 데이타를 접근해서 읽기 및 변경이 가능하며 매핑을 위해서 다른 서버와 네트워크를 통한 통신을 수행할 필요가 없다. 또한 한 서버의 고장이 메타 데이타를 접근하는데 영향을 주지 않으므로 높은 가용성을 제공한다. 하지만 이 방법에서는 모든 서버가 메타 데이타에 대해서 빈번한 읽기 및 변경을 수행하므로, 그 때마다 중복된 메타 데이타의 일관성을 유지하기 위해 모든 서버에 걸친 전역 잠금을 수행해야 한다는 단점이 있다.

마지막 방법은 그림 3에서처럼 볼륨 관리에 참여하는 호스트들이 메타 데이타를 중첩되지 않게 분할하여 관리하는 것이다. 각 호스트는 자신이 관리하는 메타 데이타만 접근할 수 있으므로 호스트들간의 전역 잠금은 필요가 없다. 하지만, 매핑을 수행할 때 요구하는 범위가 자신의 범위를 벗어나게 되면 네트워크를 통해 다른 호스트에 매핑 요구를 보내야 한다.

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

3.2 매핑 테이블

SAN 논리볼륨 관리자는 성능향상을 위해서 매핑 테이블을 하나의 저장장치에 모두 저장하지 않고 분할저장해서 부하 분산을 시도한다. 이때 매핑 테이블을 여러 저장장치에 분할하는 방법과 매핑 테이블의 엔트리 표현 방법에 따라서 전체 볼륨 관리자의 성능에 영향을 미치게 된다. 이 논문에서는 세 가지의 매핑 테이블의 표현 방법을 제시한다.

첫 번째는 매핑 테이블의 가장 기본적인 표현 방법으로 (device, sector)로 이루어지는 물리주소들이 대응되는 논리주소의 순서에 따라서 배열형태를 이루는 것이다. 그림 4에서 이를 보여준다. 이런 형태로 매핑 테이블을 표현하게 되면 원하는 논리주소가 저장된 블록을 계산해 낼 수 있고 한번의 입출력으로 매핑을 수행할

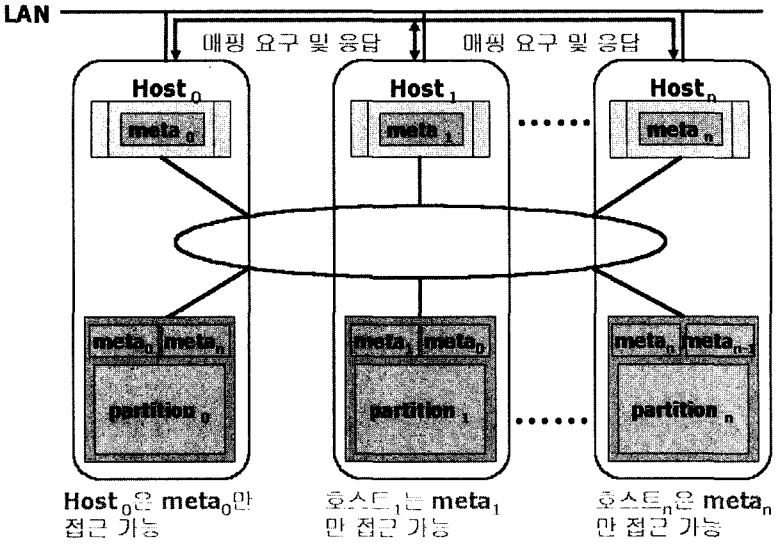

그림 3 분할 메타 데이타 관리

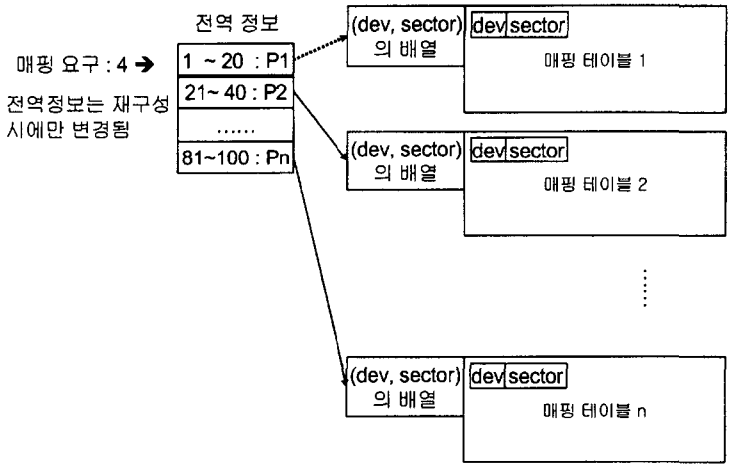

그림 4 매핑 테이블의 분할 관리

수 있다. 예를 들어서, 호스트1이 관리하는 매핑 테이블에는 논리주소 101~200까지 100개가 저장되어 있고 저장장치의 한 블록에는 10개의 (device, sector)의 물리주소가 저장될 수 있다고 가정하자. 매핑요구로 들어온 논리주소가 155라면 ⌊(155−100)/10⌋=5가 읽어올 블록의 상대 주소가 되고, (155−100)%10=5는 그 블록 내에서 몇 번째 물리주소를 취해야 하는지 알 수 있다.

일반적으로 SAN 논리볼륨 관리자에게 들어오는 매핑 요구는 순서적인 성향을 보이므로 위와 같이 논리주소를 분할했을 경우 매핑요구에 대한 부하가 논리주소의 앞부분을 담당하는 서버에 집중될 수 있다. 회소논리주소 분할방법을 적용해서 이런 점을 해소할 수 있다. 회소논리주소 분할방법에서 매핑 테이블이 논리주소의 순서대로 분할되어 각 저장장치에 저장되므로 특정 저장장치가 어떤 이유로 일시적/영구적으로 접근 불가능의 상태가 되면 전체 논리볼륨은 사용할 수 없게 된다. 이런 이유로 분할된 매핑 테이블을 Chained Declustering[10]과 같은 방법을 통해 중복 저장하여 저장장치 고장에 대비하여야 한다.

두 번째 방법은 매핑 테이블을 물리주소를 기준으로 분할하고 여기에 회소논리주소 분할방법을 적용하는 것이다. 일반적으로 볼륨 관리자 상위에서의 입출력 요구는 논리주소가 연속적인 경향을 띄게 되고 매핑을 위해서 볼륨에 있는 저장장치들을 골고루 접근하게 된다. 논

리볼륨에 참여하는 저장장치의 수가 nd라하고 전체 논리볼륨의 논리블록이 nl개라고 하자. 최초에 매핑 테이블을 구성할 때 nl개의 논리주소를 nd개만큼 분할을 하는데 되도록 이면 인접한 논리주소가 같은 저장장치에 저장되지 않도록 디클러스터링 한다. 쉽게 생각해 볼 수 있는 것은 라운드 로빈 방법이다. 그렇게 되면 매핑 테이블이 분할된 모습은 그림 5와 같다.

그림 5에서 ㉠은 논리주소 24 개를 라운드 로빈 방법을 기반으로 각 저장장치에 분산한 모습이다. 이 상태에서 ㉣에서와 같은 순서로 논리주소에 대한 쓰기 요청이 발생하면 자유공간 관리자는 적절히 논리주소에 물리블록을 할당한다. 이때 제안하는 방법에서는 물리블록을 할당하는 저장장치를 논리주소가 저장된 저장장치에서 할당한다. 예를 들어서 1번 논리주소에 대한 쓰기 요청이 최초로 발생하면 물리블록을 할당해 주어야 할 것이다. 물리블록을 할당할 저장장치를 1번 논리주소가 존재하는 disk1로부터 할당받는다. 이때 할당받은 물리블록은 1이 된다. ㉡을 보면 disk1의 매핑 테이블 영역에서 논리주소 1에 해당하는 부분에 물리블록 1이 기록된 것을 볼 수 있다. 논리주소 2는 disk2의 물리블록 1로, 논리주소 6은 disk3의 물리블록 1로 매핑되는 것을 매핑 테이블에서 볼 수 있다.

이 방법의 장점은 disk1에 저장된 매핑 테이블을 관리하는 호스트가 항상 disk1의 자유공간 할당도 수행하므로 추가적인 통신이 필요 없다. 또한, 스트라이핑 볼륨에서 공간을 할당할 때 항상 다음에 물리블록을 할당할 저장장치가 어디인지를 알려주는 모든 호스트에 대한 전역 변수가 있어야 하는데 이에 대한 필요성도 사라진다. 일반적으로 매핑 테이블의 엔트리는 {device(2bytes), sector(8bytes)}로 이루어지며 크기는 10 bytes가 된다. 하지만 이 방법에서는 device를 별도로 기록할 필요가 없다. 그리고, 각 저장장치에 저장된 매핑 정보는 그 저장장치에 대한 물리블록에 대한 것이므로 저장장치 고장에 대비해서 다른 저장장치에 매핑정보나 공간할당 정보를 중복 저장할 필요가 없다.

단점으로는 논리주소를 디클러스터링 해놓았지만 논리주소에 대한 요청이 순서적이지 않고 부분부분 나타날 경우 저장장치 사용정도가 특정 저장장치에 편중될 수 있다. 최악의 경우는 그림 5에서 보았을 때 논리주소에 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.

마지막 방법으로 논리주소를 기준으로 매핑 테이블을 분할하되 매핑 정보를 모든 논리블록에 대해서 테이블에 기록하는 것이 아니고 범위를 이용해서 기록하는 것을 생각해 볼 수 있다. 그림 6은 스트라이핑 볼륨에 대해서 범위형태로 매핑 정보를 기록한 것이다. sls를 시작 논리섹터라 하고, srs를 시작 물리섹터, srd를 시작 물리 디바이스, len을 논리주소와 물리주소가 연속되게 할당된 길이, dn을 논리볼륨에 포함된 디바이스 개수, devs를 스트라이핑에 참여하는 디바이스 번호들이라고 하자. 매핑 테이블 1의 첫 번째 매핑 레코드를 보면 sls=1, len=10, srd=1, srs=1, nd=3으로 되어 있다.

이 매핑 정보가 의미하는 것은 논리주소가 1부터 10이 연속적으로 할당되었고 이에 대응하는 물리주소도 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 이해할 수 있다.

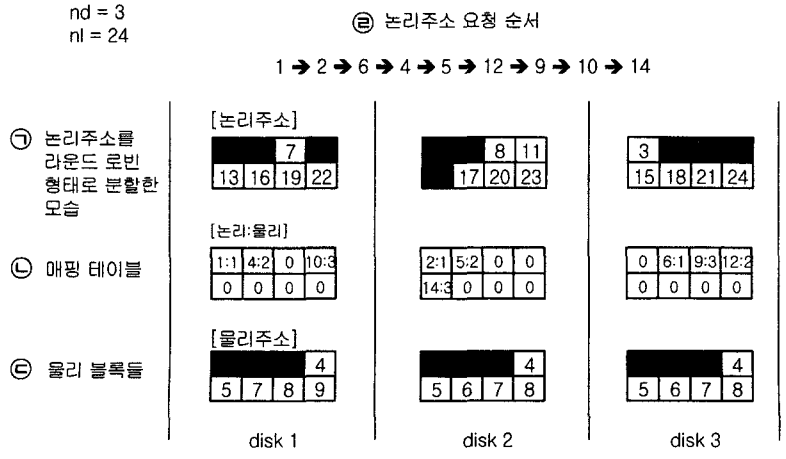

그림 5 희소논리주소 분할방법

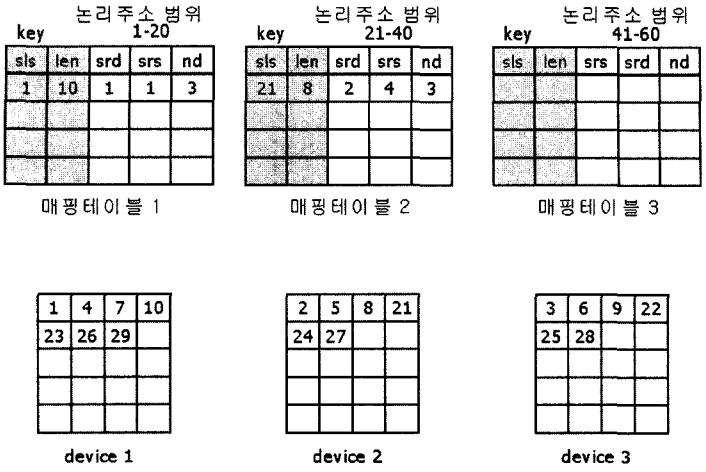

그림 6 범위를 이용한 매핑 테이블 표현 방법

이 경우에는 물리주소가 device 2의 4, device 3의 4, device 1의 5, device 2의 5 ... 이렇게 8개가 연속적으로 할당되었다는 것을 의미한다.

매핑의 수행은 먼저 매핑요구로 들어온 논리주소를 가지고 매핑 테이블이 저장되어 있는 저장장치와 이를 관리하는 호스트를 결정한 후 매핑요구를 해당 호스트에 전달한다. 매핑요구를 받은 호스트는 매핑 테이블에 기록되어 있는 sls와 len을 키로 해서 이진탐색을 수행한 후 매핑요구 논리주소를 포함한 레코드를 찾는다. 레코드를 찾으면 정보들을 가지고 적절한 물리주소를 계산해 낼 수 있다. 이때 각 호스트는 매핑 테이블의 내용을 메모리에 상주시킬 수 있다는 것이 전제가 된다.

본 논문에서는 첫 번째 방법으로 매핑 테이블을 표현한다. 첫 번째 방법은 매핑 테이블을 표현하는 가장 기본적인 방법이며, 임의의 파일 시스템에 의존하지 않고 사용할 수 있는 범용의 매핑 테이블 표현 방법이다. 두 번째 방법은 매핑 속도면에서 첫 번째 방법과 차이가 나지 않는다. 그러나 이 방법은 매핑 테이블의 공간이 절약되고, 저장장치 에러에 대비해 매핑 테이블을 중복 관리할 필요가 없다는 장점이 있다. 이 방법에 대해서는 향후 연구에서 다루어 비교 평가할 것이다.

마지막 방법의 경우는 이미 언급한데로 연속적인 논리주소의 매핑요구가 발생할 때 효과적이다. 이 방법을 GFS 파일 시스템과 같이 사용하였다. GFS의 경우 논리주소 공간을 나름대로 분할하여 저널링 영역과 리소스 그룹으로 나누어 사용한다. 이 경우 매핑요구가 부분적으로 연속적이긴 하지만 전체적으로 보았을 때 일반적인 데이타 저장을 위한 매핑요구와 저널링을 위한 매핑요구가 혼재되어 나타나므로 매핑 레코드의 개수가 많아지고 메모리에 저장할 수 없었다. 따라서 이 방법은 범용의 논리볼륨 관리자로 사용하기에는 문제가 있으며 파일 시스템과 볼륨 관리자를 설계할 때 이런 부분을 고려하여 어느 정도 상호 의존적으로 구현한다면 매우 효과적으로 사용해볼 수 있다.

3.3 스냅샷

스냅샷은 사용자가 원하는 시점의 볼륨 이미지를 유지하여 추후에 스냅샷 당시의 데이타를 참조하고자 할 경우 사용된다. 특히, 대용량의 SAN 환경에서 백업을 효율적으로 처리하기 위해서는 스냅샷 기능이 반드시 필요하다. 스냅샷 기능은 매핑 관리자에 의해 제공된다. 일반적으로 매핑 관리자는 스냅샷 요구시, 원래 볼륨이 가지고 있는 매핑 테이블을 그대로 스냅샷 목적으로 사용하기 위해 복사한다. 복사된 매핑 테이블은 스냅샷이 삭제될 때까지 유지된다. 스냅샷 기능을 위해 매핑 정보를 위한 충분한 공간이 있어야 한다.

스냅샷 생성 후 사용자의 요구에 의해서 데이타의 변경, 추가 혹은 삭제가 발생할 경우 원래 데이타를 새로운 공간에 할당받아 복사해놓고 원래 블록의 내용을 변경하는 변경 시 복제(Copy-On-Write) 기법을 사용한다. 그리고 스냅샷의 매핑 테이블에는 원래의 데이타 내용이 복사된 블록으로 매핑 정보를 변경한다. 이렇게 되면 원래 볼륨을 위한 매핑 테이블은 변경이 수행된 가장 최신의 데이타를 가리키게 되고 스냅샷을 위한 매핑 테이블은 스냅샷 생성 시점의 볼륨 이미지를 가지게 된다. 스냅샷에 대해서는 읽기만 수행될 수 있고 이에 대한 처리는 스냅샷을 위한 매핑 테이블을 이용해서 매핑을 수행한다. 이 상황에서 백업이 발생하면 백업은 스냅샷 매핑 정보를 이용하여 수행되게 됨으로서 스냅샷 시점과 일치하는 데이타의 백업을 취하게 된다.

그러나 앞에서 언급한 것처럼 SAN과 같은 대용량

환경에서는 매핑 테이블의 크기가 수십 MBytes~수 GBytes로 매우 크므로 이를 복사하는데 상당한 시간이 소요된다. 스냅샷을 생성하는 동안에는 다른 입출력 연산을 허용하지 않으므로 스냅샷 생성 시간이 길어진다. 이에 대한 대안으로 스냅샷 생성 중에 매핑 테이블을 복사하지 않고, 원래 볼륨의 매핑 테이블을 공유하면서 COW에 의해서 변경되는 부분만 해싱 테이블에 저장하는 방법을 제안한다. 이 방법을 이용하면 스냅샷을 생성하는 시간이 상대적으로 매우 짧아진다.

스냅샷 데이타를 접근하려면 먼저 해싱 테이블을 접근해서 원하는 논리주소가 존재하는지 살펴보고, 존재하면 이를 매핑 결과로 반환하고 없으면 매핑 테이블을 참조하여 매핑을 완성한다. 이처럼 매핑 테이블과 해싱 테이블을 모두 접근해 보아야 매핑을 완료할 수 있으므로 스냅샷 데이타를 접근하는데 상대적으로 더 많은 시간이 소요될 수 있다. 따라서 스냅샷 데이타에 대한 백업이 발생하면 테이블 복사 방법에 비해 백업 시간이 많이 소요된다는 단점이 있다. 그러나 스냅샷을 생성하는 동안에는 다른 정상 입출력 연산을 수행하지 못하므로, 스냅샷 생성 시간을 중요하게 고려한다면 해싱 테이블을 이용한 스냅샷 관리가 유리하다.

3.4 온라인 재구성

온라인 볼륨 재구성이란 저장장치를 추가하거나 삭제할 때 시스템을 정지시키지 않고 변경된 내용을 포함하도록 논리볼륨을 재구성하는 것을 말한다. 이 논문에서 제안하고 있는 재구성 방법은 스트라이핑 볼륨을 기준으로 한다. 다른 RAID level의 볼륨들은 스트라이핑 볼륨의 재구성 방법을 그대로 또는 조금 수정해서 바로 적용할 수 있다. 또한 저장장치 단편화가 일어나지 않도록 재구성한다는 전제 하에 수행한다.

그림 7은 새로운 저장장치가 추가되었을 경우 재구성하는 방법에 대해 나타내고 있다. 그림 7에서 첫 번째 그림과 같이 기존 두개의 저장장치를 사용하다가 새로운 저장장치가 추가되면 두 번째 그림처럼 기존 저장장치의 모든 데이타를 모든 저장장치 내에 완전히 스트라이핑 되도록 이동시키는 방법이다. 스트라이핑의 목적은 일반적으로 입출력 요구가 연속적인 논리주소에 걸쳐서 이루어지므로 연속된 주소를 되도록 서로 다른 저장장치에 저장하고 읽기를 수행할 때 여러 블록을 한번의 입출력 시간에 읽어오기 위한 것이다. 또한 스트라이핑

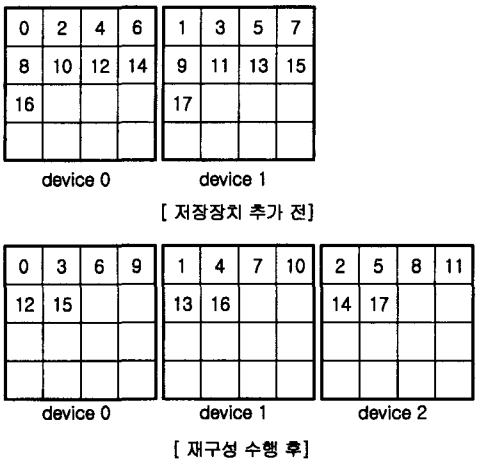

그림 7 재구성 방법에 따른 재구성 수행

을 수행하게 되면 자연스럽게 입출력 요구가 각 저장장치로 분산되어 부하균등의 효과도 얻을 수 있다. 이 방법은 재구성 시 기존 저장장치내의 모든 데이타가 이동 대상이 되므로 재구성 속도가 느린 단점이 있지만, 재구성 수행 후 입출력 성능을 향상시킬 수 있고 부하균등의 효과도 얻을 수 있다.

이 방법에서는 재구성 중 다른 입출력 요구를 수용하기 위해, 재구성 대상이 되는 블록들이 이미 재구성이 끝난 블록인지 여부를 관리해야 한다. 먼저 재구성을 수행하기 전에 비트맵 검색을 통해 저장장치가 추가되기 전 할당된 블록을 찾고 블록의 재구성 여부를 관리하기 위한 그림 8과 같은 재구성 테이블을 구성한다. 테이블은 추가된 저장장치의 블록들을 모두 포함하여 블록이 할당되는 물리주소 순으로 이루어지며 재구성 대상이 1인 블록에 대해서만 순차적으로 재구성을 수행한다. 재구성이 완료된 블록은 재구성 여부를 1로 설정한다. 이때 재구성 테이블의 재구성 대상이 1인 블록들에 대해 재구성이 완료되면 재구성이 끝나게 된다.

그림 9는 재구성 중 정상적인 입출력을 처리하는 과정을 나타낸다. 우선 정상적인 입출력 요청이 내려오기 전에 0, 1, 2, 3번째 할당된 블록이 재구성된 상태이다. 이때 할당 요청이 들어오면 device0의 7번째 블록(device0, 6)에 할당이 되어야 한다. 그러나 (device0, 6)에는 13번째 할당된 블록이 존재하므로 이 블록을 먼

물리주소	dev0, 0	dev1, 0	dev2, 0	dev0, 1	dev1, 1	dev2, 1	...	dev1, 15	dev2, 15
재구성 대상	1	1	0	1	1	0	...	0	0
재구성 여부	0	0	0	0	0	0	...	0	0

그림 8 재구성 테이블

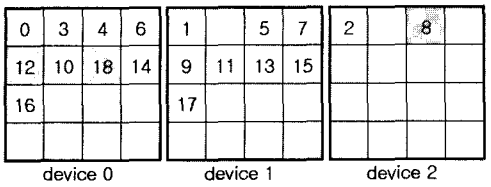

그림 9 재구성 중 19번째 블록 할당 요구에 대한 처리 방법

저 재구성해야 한다. 이런 과정을 통해 재구성될 블록이 빈 블록으로 이동할 때까지 수행한다. 따라서 9번째 할당된 블록과 13번째 할당된 블록이 각각 (device2, 2), (device0, 4) 위치로 재구성된 후 (device0, 6)에 19번째 할당될 블록을 처리한다. 이때 9, 13번째 할당된 블록은 이미 재구성된 블록이므로 재구성 테이블에서 재구성 여부를 1로 설정하여 이후의 재구성에서 제외시킨다.

이 논문에서 제안하는 온라인 재구성 방법은 재구성 테이블이라는 메타 데이타 유지로 인해 재구성 수행 중에 메타 데이타에 대한 입출력이 발생하여 재구성 성능이 떨어질 수 있다. 그러나 이로 인해 재구성을 수행하면서 동시에 상위의 정상 요청을 처리할 수 있으므로 SAN 볼륨내의 데이타에 대한 가용성을 높일 수 있다. 하지만 논문에서 제안하는 온라인 재구성 방법은 스냅샷과 동시에 수행하는 것을 고려하지 않았다. 이는 재구성 중에는 스냅샷을 생성할 수 없으며, 반대로 스냅샷을 유지하는 동안에는 재구성을 수행할 수가 없다. 재구성과 스냅샷을 동시에 수행하는 방법은 향후 연구에서 고려할 것이다.

3.5 자유공간 관리자

테이블 기반 매핑기법에서는 매핑, 스냅샷, 재구성 요구 시 논리볼륨내의 실제 저장장치의 각 블록의 사용유무를 관리하여 적절히 물리블록을 할당 및 해제하여 유연한 매핑을 도울 수 있는 자유공간 관리기법을 수행한다. 자유공간 정보는 매핑 테이블처럼 중요한 메타 데이타이기 때문에 저장장치에 저장되며, 앞에서 언급했듯이 이런 메타 데이타를 SAN에 참여하는 서버들에 분할하여 관리한다.

이 논문에서 제안하는 자유공간 관리기법은 비트맵 구조를 사용하며 자유공간은 최초적합으로 할당한다. 각 블록의 사용여부를 한 비트에 대응시켜서 표시한다. 자유공간의 할당 요청이 들어오는 경우를 생각해 보면 정상공간 할당 요청과 스냅샷이나 재구성을 위한 공간 할당 요청이 있다. 제안하는 자유공간 관리기법에서는 정상요청과 스냅샷이나 재구성을 위한 공간 할당 요청의 경우를 분리하여 공간을 할당한다.

자유공간 할당을 정상 요청과 스냅샷 관련 요청으로 구별하는 이유는 다음과 같다. 스냅샷이나 재구성을 위한 공간은 스냅샷이 존재하거나 재구성 중에만 필요한 공간이다. 즉, 스냅샷이 해제되거나 재구성이 끝나면 모두 삭제되게 된다. 만약 이 공간을 정상할당 공간에 할당한다면 스냅샷이나 재구성을 위해 할당된 공간이 해제되면서 단편화가 발생하게 된다. 이는 스트라이핑 볼륨의 스트라이핑 효과를 떨어뜨릴 뿐만 아니라 단편화로 인해 자유공간을 관리하기 어려워지며, 자유공간 탐색시간이 길어져서 자유공간 할당 요청이 느려지게 된다.

그림 10은 한 디바이스 내에서 데이타 블록과 비트맵에 관한 관계 그림이다. 그림 10에서처럼 정상할당 요청의 경우는 앞에서부터(A영역) 데이타 블록을 할당하고, 스냅샷이나 재구성의 공간 할당 요구 시에는 뒤에서부터(B영역) 할당한다. 제안하는 방법에는 현재 디바이스에 있는 총 블록의 수, 정상 할당의 경우 다음 할당될 위치, 스냅샷이나 재구성을 위한 공간 할당의 경우 다음 할당될 위치에 대한 정보를 추가하여 비트맵 탐색시간을 줄인다.

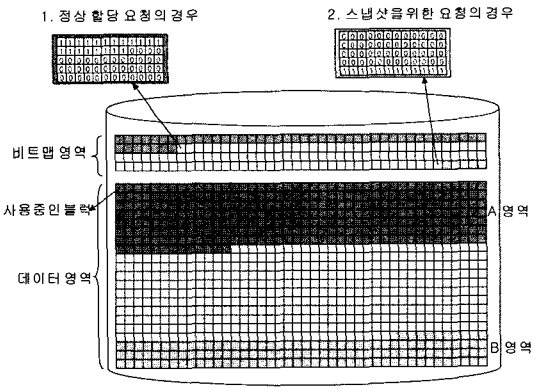

그림 10 디바이스내 공간 할당 구조

4. 매핑 관리자 구현 및 성능 평가

이 장에서는 3장에서 설계한 매핑 관리자의 구현 내용에 대해서 기술하고 기존의 매핑기법과의 성능 평가를 통해 제안한 매핑 관리자의 우수성을 입증한다.

4.1 구현

구현에는 레드햇 리눅스 커널 2.4.9 운영 체제에 Intel Pentium(R) 4 1.80GHz CPU, 256MB의 메인 메모리를 갖는 시스템이 사용되었으며, 리눅스상의 SAN 볼륨관리자인 GFS 풀의 매핑 관련 부분을 기반으로 구현되었다. FC RAID는 PowerPC 64bit RISC Microprocessor가 탑재된 JETRA5000FF에 36GB 10,000 rpm RAID 디스크 드라이브를 사용하였으며, FC Switch는 브로케

이드의 8포트 루프 스위치인 Silkworm 2010을 사용하였다.

이 논문에서는 SAN 논리볼륨 관리자를 위한 효율적이고 유연한 매핑 방법으로 테이블 기반의 매핑기법을 사용하였으며, 매핑 테이블, 스냅샷, 재구성, 자유공간 관리에 대해서 여러 가지 방법을 제안하였다.

우선 매핑 테이블에 대한 관리 방법은 논리주소를 기준으로 매핑 테이블을 각 저장장치에 분할해서 저장하고 분할된 매핑 테이블을 전담 관리하는 호스트를 두는 형태를 구현하였다. 매핑 테이블의 엔트리는 {device(2bytes), sector(8bytes)}로 이루어지며 크기는 10 bytes가 된다. 엔트리의 개수는 블록 당 하나의 엔트리를 구성하게 되어 논리볼륨의 크기에 따라 좌우된다. 예를 들어, 한 블록의 크기를 512bytes로 했을 때 논리볼륨의 크기가 100MB라면 매핑 테이블을 구성하는 엔트리 수는 195,312가 된다. 따라서 매핑 테이블의 크기는 195,312 × 10bytes = 1.95MB가 된다. 그러나 실험결과 GFS에서 요청하는 할당 단위는 8블록이었다. 8블록 당 하나의 엔트리를 구성한다면 매핑 테이블의 크기는 크게 줄어들겠지만, GFS에만 적용된다. 그러므로 할당의 단위가 서로 다른 파일 시스템에 적용하기 위해서라면 블록 당 하나의 엔트리를 구성하는 것이 범용의 매핑 관리자를 위한 매핑 테이블이라고 할 수 있다. 구현에서는 매핑 테이블을 블록 당 하나의 엔트리를 구성하였다.

스냅샷에 대해서는 매핑 테이블을 복사하는 방법과 해싱 테이블을 유지하는 방법을 모두 구현하였다. 스냅샷의 수행 과정은 다음과 같다. 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 volume_name에 해당하는 커널내부의 볼륨 자료구조를 복사해서 snapshot_name이라는 가상 볼륨을 생성한다. 스냅샷이 생성될 때 매핑 테이블을 복사하거나 해싱 테이블을 생성하기 위해 필요한 공간을 할당받는다.

스냅샷이 생성된 이후에 원본 데이타에 대한 수정이 발생하면 COW 기법을 이용해 원본 데이타를 보호한다. 이때 원본 데이타를 보호하기 위해 자유공간 관리자로부터 새로운 공간을 할당받아 복사해 놓고 원래 블록의 내용을 변경한다. 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 스냅샷이 해제된다. 이때 복사된 매핑 테이블이나 해싱 테이블이 삭제되면서 스냅샷을 위한 공간이 해제된다.

재구성 방법은 새로운 저장장치가 추가되면 기존 저장장치의 모든 데이타를 모든 저장장치 내에 완전히 스트라이핑 되도록 이동시키는 방법을 구현하였다. 재구성의 수행절차는 다음과 같다. 사용자는 추가되는 디바이스에 대한 정보를 담은 구성화일과 함께 재구성 유틸리티를 실행한다. 재구성 유틸리티는 ioctl()을 호출하여 커널내부의 재구성이 시작된다.

먼저 재구성을 수행하기 위해 볼륨 전체에 대한 쓰기 모드의 잠금을 설정하고 볼륨의 상태를 재구성 모드로 전환한다. 다음으로 비트맵 검색을 통해 저장장치가 추가되기 전 할당된 블록을 찾고 블록의 재구성 여부를 관리하기 위한 재구성 테이블을 구성한다. 재구성 시 물리블록에 대한 논리주소를 반환하기 위해 물리주소 대 논리주소 테이블도 작성한다. 다음 단계는 재구성 테이블을 참조하여 재구성 대상 블록들을 해당 위치로 이동한다. 이때 이동된 블록에 대한 매핑 정보도 같이 변경된다. 마지막으로 재구성이 끝나면 물리주소 대 논리주소 테이블과 재구성 테이블을 삭제하고 볼륨에 대한 쓰기모드 잠금을 해제한 후 볼륨의 상태를 정상 상태로 전환한다.

자유공간 관리자는 비트맵을 이용하여 스냅샷 및 재구성의 공간 할당을 정상 공간 할당과 구분하였으며, 다음 할당할 블록의 위치를 가리키는 포인터를 사용하여 구현하였다. 정상 할당 처리는 다음과 같다. 상위에서 매핑 요구가 내려오면 먼저 매핑 테이블을 검색하여 이미 매핑이 된 주소인지 확인을 한다. 매핑된 주소이면 매핑 결과를 반환하지만, 매핑 되지 않은 블록이면 비트맵을 검색하여 빈 블록을 할당하여 매핑 테이블에 반영하고 해당 블록을 반환한다. 그러나 매핑 테이블에서 한번 매핑된 주소는 해제되지 않는다. 이는 상위에서 매핑된 블록을 해제하는 요청이 내려오지 않기 때문이다. 따라서 매핑이 되어 있으면 매핑 결과를 반환하고, 안된 경우는 할당한 후 반환하는 과정을 거친다. 이와는 반대로 스냅샷 및 재구성에 필요한 공간은 매핑 테이블과 상관없이 비트맵 검색을 통해 빈 블록을 할당하고 해당 공간을 다 사용하면 해제하고 비트맵에 반영한다.

이 논문에서 제시한 매핑 관리자를 구현한 SAN 논리볼륨 관리자의 블록 설계도는 그림 11과 같다. 제안한 매핑 테이블을 이용하는 볼륨 관리자를 구성하는 주요 함수들로는 SAN 논리볼륨 관리자의 유틸리티를 작성할 수 있게 해주는 pool_ioctl()과 상위 파일 시스템이나 데이타베이스 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 처리하

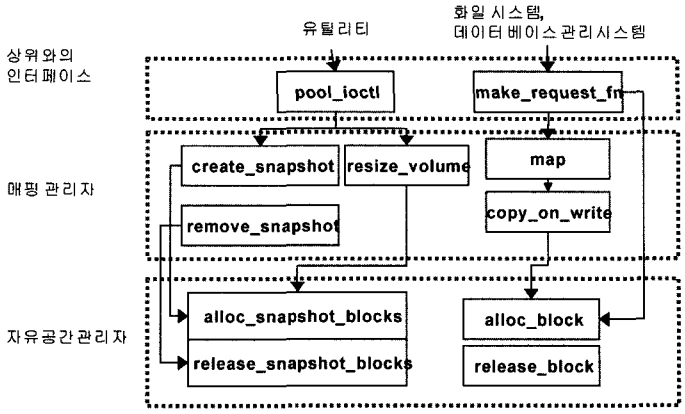

그림 11 구현한 논리볼륨 관리자의 블록 설계도

기 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 11과 같은 호출 관계를 유지하고 있다.

4.2 성능 평가

기존 방법과 제안하는 매핑 방법들에 대한 비교를 해보면 표 1과 같다. 매핑 수행 속도 측면에서는 수식 기반의 GFS 풀이 가장 빠르며 부가 사용공간이 없다. 그러나 GFS의 경우 스냅샷을 지원하지 않으며, 디스크 추가 시 온라인 재구성을 하지 않으므로 동적으로 변하는 매핑 관계를 효율적으로 처리할 수 없다. 제안하는 테이블 기반의 매핑 방법에서는 스냅샷과 온라인 재구성을 모두 지원할 수 있다. 그러나 세 가지 방법 모두 정상 입출력을 위한 매핑 요청 시 매핑 테이블을 참조하기 위한 디스크 입출력과 새로운 블록 할당을 위해 자유 공간 관리를 하기 위한 디스크 입출력이 필요하여 다소 느리다. 또한 매핑 테이블과 비트맵을 지원하기 위한 부가 공간이 필요하다.

앞에서 언급했듯이 수식 기반의 매핑 기법이 빠른 매핑 속도를 보장하고 있지만, 논리볼륨 관리자로서 반드시 제공해야 할 스냅샷이나 온라인 볼륨 재구성 기능 제대로 지원하지 못한다는 단점을 지적했었다. 따라서 실험에서는 이 논문에서 제안하는 테이블 기반 매핑 기법이 스냅샷과 온라인 볼륨 재구성 기능을 지원하면서 이를 지원하지 못하는 수식 기반의 매핑 기법에 비해 어느 정도의 매핑 속도 차가 발생하는지에 초점을 두었다.

이 논문에서 제안하는 매핑 관리자의 성능을 평가하기 위해서 제안하는 매핑 관리자와 GFS 풀의 파일에

표 1 매핑 기법들의 성능 비교

	GFS pool	테이블 기반 매핑		
		방법 1	방법 3	방법 2
매핑 속도	가장 빠름	매핑 테이블 참조 + 비트맵 공간		
		느림	느림	느림
스냅샷 지원	×	○	○	○
온라인 재구성 지원	×	○	○	○
부가 사용 공간	없음	매핑테이블+비트맵		해싱테이블 + 비트맵
		많음	적음	많음

대한 접근 성능을 비교 평가하였다. 먼저 파일의 크기를 1MB, 2MB, ..., 10MB로 변경시키면서 각 파일에 대한 접근 성능을 평가하였다. 작은 파일(0KB, 1KB, ..., 10KB)에 대한 접근 성능과 큰 파일(100MB)에 대한 접근 성능도 비교 평가하였다. 추가적으로 제안하는 논리볼륨 관리자에서 제안하는 두 가지 스냅샷 방법에 대한 비교도 수행하였다. 측정한 부분은 스냅샷 데이타를 접근하는데 걸리는 시간을 비교하였다. 온라인 재구성에 대한 성능 평가는 재구성 시간, 재구성 이후 입출력 성능 향상, 재구성 중 다른 입출력 수용에 따른 다양한 방법을 제시하여 비교할 수 있지만, 이 논문에서는 재구성 이후 입출력 성능 향상에 초점을 두어 설계하였기 때문에 비교 평가하지 않는다.

그림 12와 그림 13은 서로 다른 파일 크기에 따른 파일 접근 성능을 평가한 것이다. 측정한 방법은 1MB, 2MB, ..., 10MB 크기의 파일들을 생성하고 처음부터 순차적으로 읽음으로서 생성/읽기 위한 접근 성능을 평가하였다. 입출력 크기는 4K로 고정하였다. 그림 12를 보면, 파일 생성 시 수식 기반 매핑방법에 비해 테이블 기반 매핑방법이 85%의 성능저하를 보이고 있다. 이는 테이블 기반 매핑에서는 수식 기반 매핑에 비해 파일 생

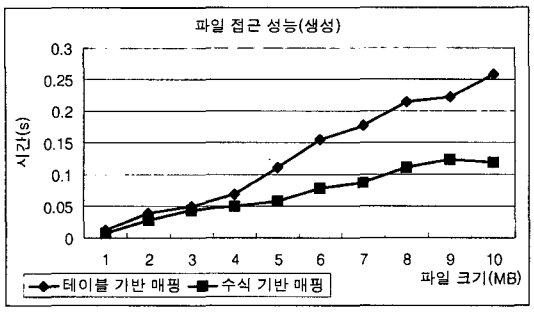

그림 12 매핑 방법에 따른 파일 접근 성능(생성)

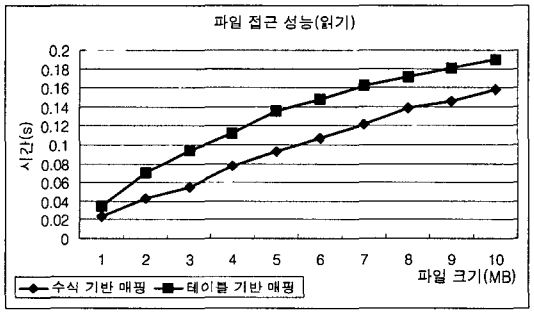

그림 13 매핑 방법에 따른 파일 접근 성능(읽기)

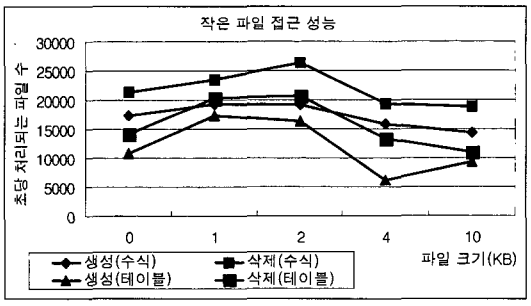

그림 14 매핑 방법에 따른 작은 파일의 접근 성능 평가

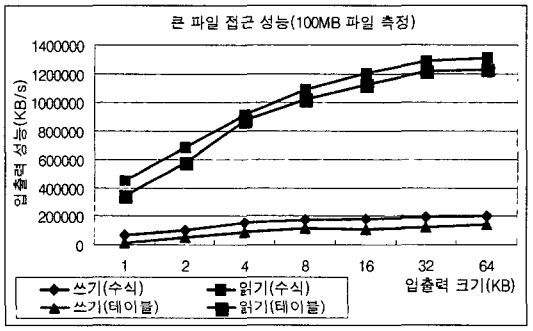

그림 15 매핑 방법에 따른 큰 파일의 접근 성능 평가

성 시 자유공간 관리자를 통해 비트맵 정보를 읽어 자유공간을 할당받고, 매핑된 결과를 매핑 테이블에 적용하는 추가된 부하가 있기 때문이다.

그림 13에서는 파일 읽기 시 테이블 기반 매핑방법이 수식 기반 매핑방법에 비해 35%의 성능저하를 보이고 있다. 테이블 기반 매핑방법이 수식 기반 매핑방법에 비해 파일 읽기 시 매핑 테이블의 매핑 정보를 읽어야 하는 입출력 부하가 더 발생하기 때문이다.

그림 14는 웹 페이지나 메일을 관리하는 시스템과 같이 저장되는 파일의 특성이 대부분 크기가 작은 파일들로 구성되며, 주로 변경이 자주 일어나는 환경을 고려하여 파일 접근 성능을 평가한 것을 나타내고 있다. 평가는 lmbench 벤치마크[11]의 lat_fs라는 프로그램을 수행하여 측정하였다. lat_fs는 다음과 같이 수행된다. 0KB, 1KB, 2KB, 4KB, 10KB 크기의 파일들에 각각 같은 크기의 파일을 1000개 생성하고, 생성된 파일을 삭제하는 과정을 통해 시간당 처리(생성/삭제)할 수 있는 파일의 수를 출력한다. 이때 생성은 create(), write(), close()시스템 함수를 1000번 수행하며, 삭제 시는 unlink() 시스템 함수를 1000번 수행한다. 그림 14에서는 테이블 기반 매핑방법이 수식 기반의 매핑방법에 비해 파일 생성은 44%, 파일 삭제는 38%의 성능저하를 보이고 있다.

그림 15는 VOD(Video on Demand) 서비스를 수행하는 시스템과 같이 저장되는 파일의 특성이 대부분 크기가 큰 파일들로 구성되며, 주로 읽기 요구가 빈번한 환경을 고려하여 파일 접근 성능을 평가한 것이다. 평가방법은 100MB 파일을 생성한 후, 파일의 처음부터 순차적으로 읽기를 수행하였다. 그림 15에서는 테이블 기반 매핑방법이 수식 기반의 매핑방법에 비해 파일 쓰기는 69%, 파일 읽기는 9%의 성능저하를 보이고 있다.

그림 14와 그림 15를 통해 파일이 작고 변경이 자주 일어나는 환경에서는 테이블 기반의 매핑방법이 수식 기반의 매핑방법에 비해 약 40%내외의 성능저하를 보이고 있다. 반면, 읽기 요구가 빈번하고 관리하는 파일의 크기가 큰 환경에서는 읽기 연산을 수행 시 테이블 기반의 매핑방법이 수식 기반의 매핑방법에 비해 9%의 성능저하를 보이고 있다. 앞의 두 가지 경우에 대해 성능차이가 다르게 나타나는 것은 파일과 관련된 메타 데이타에 대한 변경으로 인해 발생하는 입출력의 빈도에 따른 결과이다. 따라서 제안하는 매핑 관리자가 VOD 서비스를 수행하는 시스템과 같이 파일이 크고 읽기 요구가 빈번한 환경에 적합함을 알 수 있다.

마지막으로 테이블 복사 방법과 해싱 테이블 유지 방법에서 스냅샷 데이타를 접근하는데 걸리는 시간을 비교해 본다. 테이블 복사 방법에서는 스냅샷 데이타를 접근하기 위해 별도의 공간에 복사된 매핑 테이블을 활용

한다. 따라서 스냅샷 데이타에서 접근하고자 하는 논리주소에 대응하는 물리주소를 한번의 디스크 입출력을 통해 알 수 있다. 이때 매핑 테이블이 차지하는 공간은 볼륨의 크기에 비례하여 커진다.

해싱 테이블 유지 방법에서는 스냅샷 데이타를 유지하기 위해 해싱 테이블을 활용한다. 이때 볼륨의 크기에 따라 매핑 테이블이 차지하는 공간이 정해지는 것처럼 해싱 테이블도 동일하게 공간을 차지한다. 만일, 해싱 테이블을 매핑 테이블이 차지하는 공간만큼 할당해 사용한다면 스냅샷 데이타에서 접근하고자 하는 논리주소에 대응하는 물리주소를 두 번의 디스크 입출력을 통해 알 수 있다. 이는 해싱 테이블을 접근해서 원하는 논리주소가 존재하는지 살펴보고, 존재하면 이를 매핑 결과로 반환하고 없으면 매핑 테이블을 참조해야 하기 때문이다.

그러나 볼륨 전반에 걸쳐 스냅샷 데이타에 대한 변경이 일어나지 않는 한 해싱 테이블을 매핑 테이블이 차지하는 공간만큼 할당해 사용하는 것은 공간 낭비를 초래할 수 있다. 만일 매핑 테이블이 차지하는 공간의 반 정도를 할당해 유지한다면 공간 낭비를 해결할 수 있으며, 최악의 경우 세 번의 디스크 입출력을 통해 스냅샷 데이타의 위치를 알 수 있다. 따라서 해싱 테이블이 차지하는 공간으로 인해 낭비될 수 있는 공간과 디스크 입출력 회수 사이에 적절한 조율이 필요로 된다. 결과적으로 매핑 테이블 복사 방법이 해싱 테이블 유지 방법에 비해 스냅샷 데이타를 접근하는 시간이 빠르다.

지금까지의 성능 평가를 보면, GFS 풀에 비해 제안하는 매핑 관리자의 파일에 대한 접근 성능이 대용량 파일에 대한 읽기 요구를 제외하고는 다소 떨어지는 것을 알 수 있다. 이는 스냅샷이나 온라인 재구성을 지원하기 위해 필요한 메타 데이타를 유지하기 위한 부하로 발생하는 차이이다. 그러나 지금과 같이 논리볼륨에 참여하는 호스트가 하나가 아니라 실제 SAN 처럼 다중 호스트 환경이라면 매핑에 필요한 메타 데이타를 더욱 효과적으로 관리할 수 있으므로, 보다 더 좋은 결과를 보일 것이다. 따라서 이 논문에서 제안하는 테이블 기반의 매핑 방법은 매핑 속도면에서 기존의 GFS 풀의 수식 기반의 매핑 방법에 비해 다소 차이를 보였지만, GFS 풀이 제공하지 못하는 스냅샷과 온라인 재구성을 지원하고 있다.

5. 결 론

이 논문에서는 SAN 논리볼륨 관리자중에서 특히 매핑 관리자의 기능향상에 대한 방법을 찾고 이를 반영한 매핑 관리자를 설계하고 구현하였다. 기존의 수식 기반 매핑방법은 스냅샷이나 온라인 볼륨 재구성과 같은 동적인 환경에 유연하게 대처하지 못하였다. 이를 위해 이 논문에서는 매핑테이블을 이용하였고, 또한 여러 가지의 매핑테이블 표현 방법에 대해 제시하였다. 또한 매핑 관리자의 스냅샷 및 재구성에 대해서 다양한 방법을 제시하고 구현하여 비교해 보았다. 또한 자유공간 관리자는 비트맵을 기반으로 하여 정상적인 공간 할당과 스냅샷을 위한 공간 할당을 모두 효과적으로 지원할 수 있도록 하였다.

제안한 매핑 관리자는 GFS 풀의 매핑 관련 부분에 구현되었으며, 풀과 비교 평가하였다. 성능평가 결과에서 제안하는 매핑 관리자가 매핑 속도면에서 풀에 비해 다소 성능 저하를 보였다. 이는 풀이 지원하지 못하는 스냅샷과 온라인 볼륨 재구성 기능을 제공하기 위한 추가적인 부담이 발생하기 때문이다. 하지만 스냅샷이나 온라인 볼륨 재구성 기능은 무정지 시스템을 구축하기 위해 필수적인 기능이다. 무정지 시스템의 구축은 기업의 막대한 손실을 줄여주게 된다. 또한 이러한 추가적인 부담은 다중 호스트 환경 하에서 줄어들 수 있다. 따라서 이 논문에서 제안하는 매핑 관리자가 수식 기반의 GFS pool보다 무정지 시스템을 추구하는 SAN 환경에 적합함을 알 수 있다. 향후 연구에서는 이 논문에서 제안한 매핑 관리자를 보완하여 다중 호스트 환경 하에서 성능 평가를 수행한다.

참 고 문 헌

[1] 김정환, 강희일, 이동일, "SAN 기술 및 시장동향", 전자통신동향분석, pp. 24-37, 2000.

[2] Edward K. Lee and Chandramohan A. Thekkath and Chris Whitaker and Jim Hogg, "A Comparison of Two Distributed Disk Systems," Systems Research Center, Digital Equipment Corporation, 1998.

[3] 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, 1996.

[4] 김경배, 김영호, 김창수, 신범주, "SAN을 위한 전역 파일 공유 시스템의 개발", 한국 정보 과학회지, VOL.19, NO.3, pp. 24-32, 2001.

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

[6] 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, 1995.

[7] Daniel Pierre Bovet and Marco Cesati, "Understanding the Linux Kernel," pp. 686-721, O'Reilly, 2001.
[8] 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, 1996.
[9] 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, 1999.
[10] Hui-I Hsiao and David J. DeWitt, "Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines," In Proceedings of 6th International Data Engineering Conference, pp. 456-465, 1990.
[11] Larry McVoy and Carl Staelin, "lmbench: Portable Tools for Performance Analysis," In Proc. of the USENIX 1996 Technical Conference, pp. 279-294, 1996.

남 상 수
1993년 3월~2001년 2월 충북대학교 정보통신공학과 정보통신전공 학사. 2003년 3월~2003년 2월 충북대학교 정보통신공학과 정보통신전공 석사. 2003년 3월~현재 LG산전 전력연구소 전력IT연구팀 연구원. 관심분야는 데이타베이스시스템, SAN, 백업시스템, 실시간 데이타베이스

송 석 일
정보과학회논문지 : 컴퓨팅의 실제
제 9 권 제 4 호 참조

유 재 수
정보과학회논문지 : 컴퓨팅의 실제
제 9 권 제 4 호 참조

김 창 수
1989년 3월~1993년 2월 광운대학교 전자계산학과(학사). 1993년 3월~1995년 2월 서강대학교 전자계산학과(석사). 1995년~1999년 LG소프트㈜. 1999년~현재 ETRI 컴퓨터·소프트웨어연구소 컴퓨터시스템연구부 선임연구원. 관심분야는 데이타베이스 시스템, 자료저장시스템, 클러스터 시스템, 분산 시스템

김 명 준
1978년 2월 서울대학교 자연과학대학 계산통계학과(이학사). 1980년 2월 한국과학기술원 전산학과(이학석사). 1986년 5월 프랑스 Nancy(낭시) 제1대학교 응용수학 및 전산학과(이학박사). 1980년 2월~1981년 6월 아주대학교 종합연구소 연구원. 1981년 10월~1986년 5월 프랑스 Nancy 전산학연구소(CRIN) 연구원. 1986년 7월~현재 한국전자통신연구원, 선임/책임연구원, 현재 컴퓨터시스템 연구부장. 1993년 1월~1993년 12월 프랑스 Univ. of Nice Sophia-Antipolis 방문 교수. 관심분야는 데이타베이스, 분산 처리, 실시간 처리, 소프트웨어 공학