SAN 논리볼륨 관리자를 위한 혼합 매핑 기법
(A Hybrid Mapping Technique for Logical Volume Manager in SAN Environments)

남 상 수 † 피 준 일 †† 송 석 일 †††
(Sang Su Nam) (Jun Il Pee) (Seok Il Song)

유 재 수 †††† 최 영 희 ††††† 이 병 엽 ††††††
(Jae Soo Yoo) (Young Hee Choi) (Byoung Yeop Lee)

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. 더불어 논리볼륨 관리자는 특정 시점의 볼륨이미지를 유지할 수 있는 스냅샷과 시스템을 정지시키지 않고 SAN에 저장장치를 추가 또는 삭제할 수 있는 온라인 재구성 기능을 지원한다.

이러한 기능을 지원하기 위해 수식 기반의 매핑 방법보다 테이블 기반의 매핑 방법이 제안되고 있다. 그러나 이 방법은 관리해야 할 데이타 양이 저장장치 용량에 비례하여 증가하고 메인 메모리에서 모두 관리할 수 없어 성능 저하의 요인이 되었다. 이 논문에서는 기존의 수식 기반의 매핑 방법을 이용하면서 스냅샷과 온라인 재구성 기능과 같은 동적인 환경을 효과적으로 지원할 수 있는 혼합 매핑 방법을 설계하고 구현한다. 제안하는 방법의 스냅샷과 재구성은 되도록이면 정상 입출력 연산에 영향을 주지 않기 위해서 별도의 예약된 공간에서 수행된다. 마지막으로, 이 논문에서 제안한 기법에 대한 성능 평가를 수행하여 제안하는 기법이 우수함을 보인다.

키워드 : SAN, 논리볼륨 관리자, 스냅샷, 온라인 재구성

Abstract A new architecture called 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 SAN more efficiently, most of SAN operating softwares 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.

A logical volume manager also supports a snapshot that preserves a volume image at certain time and on-line reorganization to allow users to add/remove storage devices to/from SAN even while the system is running. To support the snapshot and the on-line reorganization, most logical volume managers have used table based mapping methods. However, it is very difficult to manage mapping table because the mapping table is large in proportion to a storage capacity. In this paper, we design and implement an efficient and flexible hybrid mapping method based on mathematical equations. The

· 본 연구는 한국과학재단 목적기초연구(R01-2003-000-10627-0) 지원으로 수행되었음
† 비 회 원 : LG산전 전력연구소 연구원
ssnam@lgis.com
†† 비 회 원 : 충북대학교 정보통신공학과
pji@netdb.chungbuk.ac.kr
††† 비 회 원 : 한국과학기술원 전자전산학과
sisong@chungju.ac.kr
†††† 종신회원 : 충북대학교 전기전자컴퓨터공학부 교수
yjs@cbucc.chungbuk.ac.kr
††††† 비 회 원 : 순천대학교 컴퓨터과학과
choiyh7@korea.com
†††††† 비 회 원 : 배재대학교 전자상거래학부 교수
bylee@pcu.ac.kr
논문접수 : 2002년 12월 11일
심사완료 : 2003년 11월 21일

mapping method in this paper supports a snapshot and on-line reorganization. The proposed snapshot and on-line reorganization are performed on the reserved area which is separated from data area of a volume. Due to this strategy normal I/O operations are not affected by snapshot and reorganization. Finally, we show the superiority of our proposed mapping method through various experiments.

Key words : SAN, Logical Volume Manager, Snapshot, On-line Reorganization

1. 서 론

전자 상거래 등 인터넷에 기반을 둔 응용 분야의 기술 발전과, 인터넷 사용자의 확산으로 사용자에게 공유되고, 서비스되는 데이타의 양이 기하 급수적으로 증가하고 있다. 이에 따라 효과적인 대량의 데이타 관리의 중요성이 증대되고 있으며, 대량의 데이타를 효율적으로 공유하고 고속으로 서비스하기 위한 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[1,2].

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 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 매핑 기법과 테이블 기반의 매핑 기법으로 나누어볼 수 있다. 수식 기반의 매핑 기법은 계산식을 이용해 매핑을 수행한다. 이 방법은 매핑이 빠르고 구현이 간단하지만, 스냅샷이나 온라인 재구성과 같이 매핑 관계를 변경해야 하는 연산들을 유연하게 처리하기 어렵다[3].

이에 반해, 테이블 기반 매핑 기법은 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[3]. 특히, 테이블 기반의 매핑 기법은 스냅샷, 온라인 재구성 요구 시 실제 저장장치의 각 블록의 사용유무를 판단하여, 적절한 물리블록을 할당/해제할 수 있도록 해주는 자유공간 관리 기법이 필요하다.

기존 논리볼륨 관리자들의 매핑 기법은 스냅샷이나 온라인 재구성 기능을 지원하기 위해 수식 기반의 매핑 기법보다는 테이블 기반의 매핑 기법이 제안되고 있으나 관리해야 할 데이타 양이 저장장치 용량에 비례하여 증가하고 메인 메모리에서 모두 관리할 수 없어 성능 저하의 요인이 되었다.

이에 본 논문에서는 Linux상의 SAN 볼륨관리자인 GFS(Global File System)의 pool[4]의 매핑 관련 부분에, 기존의 수식 기반의 매핑 방법을 이용하면서 스냅샷과 온라인 재구성 기능과 같은 동적인 환경을 효과적으로 지원할 수 있는 혼합 매핑 방법을 설계하고 구현한다. 이 논문은 또한 유연한 매핑을 돕기 위해 자유공간 관리기법을 설계하고 구현한다. 제안하는 방법의 스냅샷과 재구성은 되도록이면 정상 입출력 연산에 영향을 주지 않기 위해서 별도의 예약된 공간에서 수행된다. 또한, 제안하는 스냅샷 방법은 위치가 변경된 원본블록만을 해쉬테이블에 유지하는 전략을 사용하기 때문에 스냅샷 생성 및 제거로 인한 영향을 최소화 하고 있다.

이 논문의 구성은 다음과 같다. 2장에서는 논리볼륨 관리자의 매핑 관리와 자유공간 관리에 대한 기존 연구에 대해 기술한다. 3장에서는 매핑 관리자의 설계 목표를 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 매

핑 기법과 자유공간 관리기법에 대해서 기술한다. 이전의 논리볼륨 관리자들의 매핑 기법을 살펴보면 크게 수식 기반의 매핑 기법과 테이블 기반의 매핑 기법으로 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 매핑 기법을 수행하는 기존의 논리볼륨 관리자에 대해 설명한다. 수식 기반의 매핑 기법을 수행하는 대표적인 SAN 논리볼륨 관리자로서 미네소타 대학에서 개발한 GFS의 pool과 을 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 Client는 SAN fabric을 구성하고 구성된 SAN fabric을 통해 저장장치에 접근한다.

GFS의 pool은 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 pool은 저장장치 추가 시 논리볼륨의 크기는 확장되지만, 저장장치의 부하 균등을 위한 저장장치간 데이타 이동이 전혀 고려되지 않았다. 추가적으로, pool은 논리볼륨에 참여한 저장장치들과 논리볼륨 정보와 같은 메타 데이타를 각 저장장치의 일정 영역에 중복 저장하며, 이런 메타 데이타의 일관성을 유지하기 위해 디바이스 락이라는 잠금 기법을 활용하고 있다.

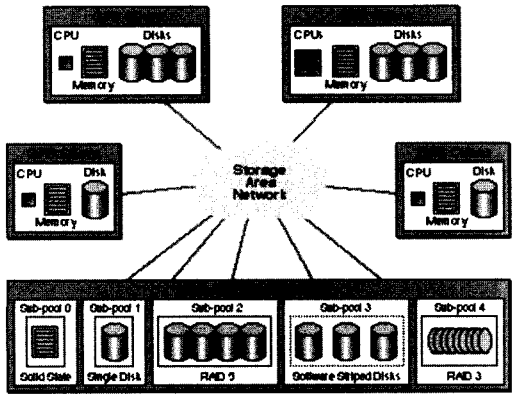

그림 1 SAN 환경에서의 GFS 모형

테이블 기반의 매핑 기법을 사용하고 있는 기존의 논리볼륨 관리자는 다음과 같다.

첫 번째로, 테이블 기반의 매핑 기법을 사용하고 있는 SAN 기반의 논리볼륨 관리자로서 SANtopia[5]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.

두 번째로, [6]에서도 테이블 기반의 매핑을 수행한다. 여기에서는 GFS pool의 매핑 관련 부분에 다양한 매핑 테이블 표현 방법을 제시하였다. 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. 여기에서는 세 가지의 매핑 테이블 표현 방법을 제시하였다.

첫 번째 방법은 매핑 테이블의 가장 기본적인 표현 방법으로 (device, sector)로 이루어지는 물리주소들이 대응되는 논리주소의 순서에 따라서 배열형태를 이루는 것이다. 이와 같이 매핑 테이블을 표현하게 되면 원하는 논리주소가 저장된 블록을 계산해 낼 수 있고 한번의 입출력으로 매핑을 수행할 수 있다. 두 번째 방법은 매핑 테이블을 물리주소를 기준으로 분할하고 여기에 희소논리주소 분할방법을 적용하는 것이다. 일반적으로 볼륨 관리자 상위에서의 입출력 요구는 논리주소가 연속적인 경향을 띄게 되고 매핑을 위해서 볼륨에 있는 저장장치들을 골고루 접근하게 된다. 마지막 방법으로 논리주소를 기준으로 매핑 테이블을 분할하되 매핑 정보를 모든 논리블록에 대해서 테이블에 기록하는 것이 아니고 범위를 이용해서 기록하는 것을 제시하였다.

[6]에서도 스냅샷과 재구성을 지원한다. 스냅샷의 경우 스냅샷 볼륨을 위한 매핑테이블을 생성하고 원본데이타에 변경이 생기면 copy-on-write(COW) 방법을 이용해 원본을 보호한다. 즉, 원본 데이타는 볼륨의 다른 영역에 복사되고 원래의 원본데이타는 변경이 된다. 그리고, 복사된 원본 데이타의 위치는 스냅샷 매핑테이블에 기록이 되어 스냅샷 볼륨은 변경되기 전 데이타를 유지하게 된다. 재구성의 경우 추가되거나 제거되는 디스크가 발생할 때 다시 전체 데이타를 디스크에 스트라

이핑이 되도록 재 구성하게 된다. 이때도 재구성되기 전 매핑테이블과 재구성된 후의 매핑테이블을 유지하면서 입/출력 요구를 처리하게 된다.

마지막으로, SAN을 기반으로 하지는 않지만 논리적 저장장치 가상화를 지원하는 Petal[7]도 테이블 기반의 매핑 기법을 이용한다. Petal은 네트워크로 연결된 여러 호스트에 부착된 저장장치를 가상화하여 하나의 논리 저장장치로 클라이언트에게 제공하는 시스템이다. 그림 2에서 Petal의 구조를 보여주고 있다. 그림에서처럼 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[2,8].

테이블 기반 매핑 기법에서는 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. [6]에서도 비트맵을 통해 자유공간을 관리한다.

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[9]. 이러한 정책 중에 GFS나 EXT2[10]와 같은 파일 시스템은 비트맵 기반의 자유공간 관리를 하고 있다. XFS[11]나 Fujitsu사의 SafeFILE 제품의 파일 시스템인 HAMFS [12,13]의 경우는 B-tree를 활용한 색인 적합 방법을 사용하고 있다.

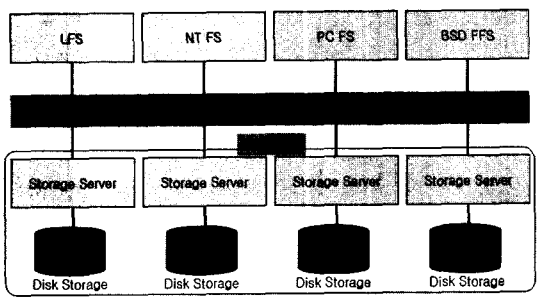

그림 2 Petal의 구조

마지막으로 수식기반 방법이지만 디스크의 온라인 추가를 효과적으로 처리할 수 있는 방법을 제안한 SZIT (Strip Zone Information Table)기반 매핑 방법[14]을 들수 있다. 이 방법은 디스크 추가시 스트라이핑 지역을 나누어 각 스트라이핑 지역에 맞는 스트라이핑을 수행하는 방법이다. 디스크가 추가되면 추가된 디스크 영역에 대한 스트라이핑 영역과 원래 스트라이핑 영역을 분리하여 각 스트라이핑 영역에 적당한 계산식을 만들어 내고 이에 따라서 매핑을 수행하는 방식이다. 이 방법은 디스크 추가와 같이 매핑 정보의 변경 회수가 적을 경우에 적합하다. 하지만 제안하는 방법은 디스크 추가의 경우는 물론 스냅샷 처럼 매핑 정보가 소량으로 매우 빈번히 변하는 경우에도 적합한 방식이다.

3. 혼합 매핑 기법의 설계

기존 테이블 기반의 매핑은 수식 기반의 매핑이 지원하지 못하는 스냅샷과 온라인 재구성을 지원하기 위한 논리볼륨 관리자의 매핑 방법으로 선호되어 왔다. 그러나 테이블 기반의 매핑 방법은 관리해야 할 데이타 양이 저장장치 용량에 비례하여 증가하고 메인 메모리에서 모두 관리할 수 없어 논리볼륨 관리자의 성능 저하 요인이 되었다. 이를 위해 이 논문에서는 수식 기반의 매핑을 통해 빠른 매핑을 수행하면서도 스냅샷과 온라인 재구성과 같은 동적인 환경에 유연하게 대처할 수 있는 혼합 매핑 방법을 설계한다.

이 장의 구성은 다음과 같다. 먼저 이 논문에서 제시하는 매핑 관리자에 대해서 기술한다. 다음에 제안하는 SAN 논리볼륨 관리자에서 지원하는 스냅샷과 온라인 볼륨 재구성 기능에 대한 설계 내용을 설명한다. 마지막으로 스냅샷과 온라인 볼륨 재구성 기능을 지원하면서 유연한 매핑을 돕기 위해 필요한 자유공간 관리자에 대한 설계 내용을 기술한다.

3.1 혼합 매핑 기법

수식 기반 매핑을 수행하는 GFS의 pool은 수식에 의해 매핑되는 블록 이외의 별도의 공간을 유지하지 않는다. 따라서 스냅샷이나 온라인 재구성이 발생하는 상황에서 이로 인해 생성되는 데이타를 유지하기 위한 별도의 공간을 할당받을 수가 없었다. 이를 위해 제안하는 방법에서는 저장장치의 영역을 정상 할당 영역과 예약 영역으로 구분한다. 정상 할당 영역은 상위의 정상적인 요청에 대해 수식 기반의 매핑으로 처리되는 공간이며, 예약 영역은 스냅샷이나 온라인 재구성 시 발생하는 데이타를 유지하기 위한 공간으로 활용된다.

예약 영역은 스냅샷이나 온라인 재구성이 발생하여 필요할 때마다 공간을 할당/해제하는 영역이기 때문에 볼륨을 사용하는 응용에 따라 예약 영역을 크게 또는

적게 할당할 수도 있다. 이런 상황에서 예약 영역의 크기를 고정하여 유지한다면 스냅샷이나 온라인 재구성 시 발생하는 데이타가 예약 영역을 다 소모하여 더 이상 공간이 없을 경우 추가되는 데이타를 유지할 수 없게 된다. 스냅샷 수행시 발생하는 데이타의 양은 유동적이며 스냅샷이 해제될 때까지 해당 데이타는 반드시 유지되어야 한다. 따라서 예약 영역을 고정하여 유지하는 것은 매핑 관리자가 스냅샷이나 온라인 재구성을 지원하는데 적합하지 않은 방법이다.

그림 3은 혼합 매핑 기법 시 저장장치의 구조를 나타낸다. 만약 스냅샷 생성 후 예약 영역을 다 소모하여 더 이상 공간이 없을 경우에는 정상 할당 영역을 활용한다. 이를 효과적으로 수행하기 위하여 정상 할당 영역의 어떤 물리블록들이 사용되는지를 비트맵을 통해 판별한 후 사용되지 않은 영역을 할당한다. 즉, 정상 할당 시 매핑은 수식을 통해 처리하고 현재 사용되는 블록을 비트맵 상에 반영하는 절차를 수행해야 한다. 따라서 비트맵에 반영하기 위하여 한번의 입출력이 발생한다. 정상 할당 영역을 예약 영역으로 활용하는 내용은 스냅샷에 대해 기술하는 부분에서 자세히 설명한다.

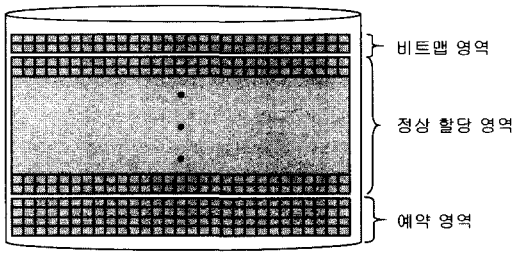

그림 3 혼합 매핑 기법 시 저장장치의 구조

3.2 스냅샷

스냅샷은 사용자가 원하는 시점의 볼륨이미지를 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. 일단 스냅샷이 생성되면 예약 영역에 해싱 테이블을 생성한다. 이때 테이블 기반의 매핑에서는 스냅샷 생성 시 테이블을 복사하지만, 이 방법에서는 해싱 테이블을 생성하는 것으로 스냅샷 생성이 끝난다. 생성된 해싱 테이블은 스냅샷이 삭제될 때까지 유지된다.

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. 그림 4는 COW를 처리하는 방법을 나타낸다. 그림에서처럼 스냅샷이 발생하면 해싱 테이블이 생성된다. 이후에 논리주소 1번에 대한 물리블록 10번 데이타 A가 B로 변경되면 COW가 발생한다. COW가 발생하면 원래 데이타(그림에서 A)는 예약 영역내의 해싱 테이블에 유지하고, 정상 할당 영역에는 최신의 데이타(그림에서 B)를 유지하는 방법을 사용한다. 스냅샷 생성 시점의 볼륨 이미지를 접근하려면 먼저 해싱 테이블을 접근해서 원하는 논리주소가 존재하는지 살펴보고, 존재하면 이를 매핑 결과로 반환하고 없으면 수식에 의해 매핑을 완성한다. 이와 같이 해싱 테이블과 수식 기반 매핑을 모두 참조해야만 스냅샷 데이타에 대한 매핑을 완료할 수 있다.

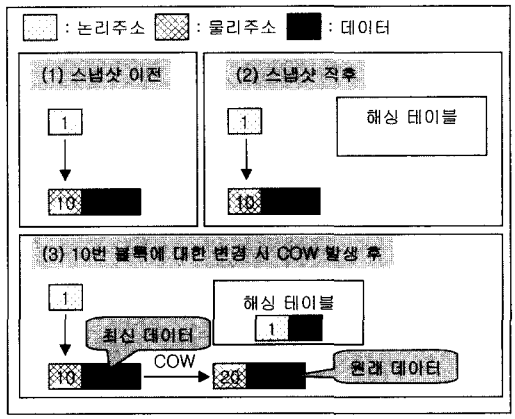

그림 4 COW 처리 방법

스냅샷 생성 후 스냅샷 당시의 이미지를 유지하기 위해 변경이 일어나는 데이타는 COW 기법을 사용하게 된다. 이때 사용자의 요구에 대한 블록이 이미 COW 처리된 블록인지 판단하는 기준이 필요하다. 이에 대한 판단 기준은 먼저 사용자 요구에 대한 논리주소가 해싱 테이블에 있는지 검색한다. 검색 결과가 존재한다면 이미 COW 처리된 블록이므로 정상 할당 영역에서 수식에 의해 매핑된 블록을 반환한다. 검색 결과가 없다면 수식에 의해 매핑된 블록은 COW 처리를 해야하는 블록이 된다.

스냅샷이 생성된 이후에 COW가 빈번히 발생하여 원래 블록의 내용을 유지하기 위한 예약 영역내 공간이 부족한 상태가 발생할 수 있다. 이때 정상 할당 영역을 예약 영역의 일부로 사용할 수 있도록 예약 영역 확장이 일어난다. 확장된 예약 영역에는 COW가 발생하여 원래 블록의 내용을 유지하는 공간으로 활용되며, 스냅샷을 위한 해싱 테이블은 확장전의 예약 영역에서 유지한다. 일단 예약 영역 확장이 발생하면 먼저 볼륨의 상태를 확장 모드로 전환한다. 확장 모드 시 사용자의 요

구에 대한 블록 처리는 확장 모드 이전의 방법과 동일하지만 추가적으로 수행할 작업이 있다. 이는 수식에 의해 매핑 처리되는 정상 할당 영역의 일부 블록을 예약 영역으로 확장하여 사용되기 때문에 이러한 블록들에 대해 사용자 요구가 내려오면 어떻게 수용할 것인지 고려해야 한다.

이를 위해 예약 영역 확장이 발생하기 전에 물리주소를 키로 하는 역 해싱 테이블을 생성한다. 이때 역 해싱 테이블 내용은 예약 영역으로 확장된 블록에 대해서만 유지한다. 논리주소를 키로 하는 COW 해싱 테이블을 유지하면서 역으로 물리주소를 키로 하는 역 해싱 테이블을 두는 것은 사용자의 요구에 대한 물리주소가 확장된 블록인지 판단하기 위해서이다. 볼륨의 상태가 확장 모드로 전환된 이후에 COW가 발생하게 되면 정상 영역의 일부를 예약 영역으로 확장하여 사용하게 되므로, 이에 해당하는 논리주소와 물리주소를 COW 해싱 테이블과 역 해싱 테이블에 삽입한다. 만일 사용자의 요구에 대한 물리주소가 이미 예약 영역으로 확장된 경우는 해당 데이타를 새로운 공간에 할당받아 복사해 놓고 원래 물리주소에 해당하는 블록은 매핑 결과로 반환한다. 이 경우 사용자의 요구에 대한 블록이 확장된 블록인지 판단하기 위해 역 해싱 테이블을 검색한다. 역 해싱 테이블을 통해 물리주소에 대한 논리주소를 찾고, 그 논리주소를 통해 COW 해싱 테이블의 물리주소를 새롭게 할당된 블록으로 변경한다. 변경된 내용은 역 해싱 테이블에도 반영한다.

그림 5는 스냅샷을 생성하고 이후에 볼륨의 상태가 확장 모드로 변경되는 상황을 나타낸다. 먼저 논리주소와 물리주소간의 매핑 관계는 P=L+20 이라고 가정한다. 또한 각각의 해싱 테이블이 차지하는 공간은 한 블록이라고 가정한다. 그림에서처럼 논리주소 4, 5, 1에 대해

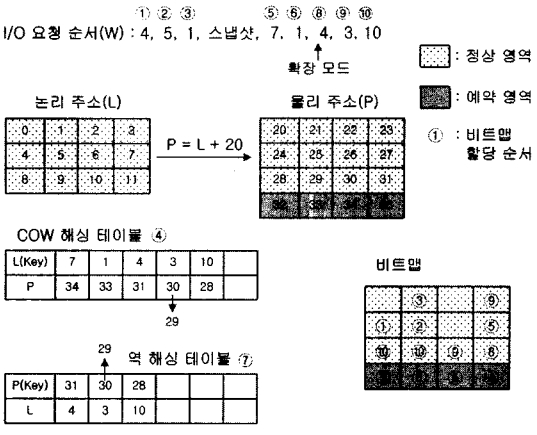

그림 5 스냅샷 생성 후 확장 모드 시 매핑 방법

쓰기 요청이 처리된 후, 스냅샷이 발생하였다. 스냅샷 생성 시 COW 해싱 테이블을 위한 공간을 할당받은 후 논리주소 7, 1에 대한 COW가 발생하여 해싱 테이블에 반영하였다. 이때 새로 할당된 블록도 COW 처리를 한다. 논리주소 4에 대한 쓰기 요청이 내려오면서 역 해싱 테이블을 위한 한 블록을 제외하면 예약 영역이 부족하여 볼륨 상태가 확장 모드로 전환된다.

이후부터 정상 영역에서 예약 영역의 확장 영역으로 이동한 블록은 역 해싱 테이블 등록된다. 물리주소 31은 논리주소 4의 원래 데이타가 복사된 블록이고, 물리주소 30은 논리주소 3의 데이타를 유지하는 블록이 된다. 이때 사용자의 요구가 논리주소 10에 발생하면 해당 물리주소 30은 역 해싱 테이블의 검색 결과 이미 확장 영역으로 사용된 블록이므로 블록의 내용을 다른 영역으로 복사해야 한다. 따라서 새 물리주소 29를 할당받아 복사한 후 COW 해싱 테이블과 역 해싱 테이블에서 해당하는 내용을 변경한다.

스냅샷 종료 후에는 COW 해싱 테이블 내용에 해당되는 블록을 해제하고 COW 해싱 테이블과 역 해싱 테이블을 삭제함으로서 볼륨의 상태가 정상으로 전환된다.

그림 6은 스냅샷을 포함한 매핑 알고리즘의 의사코드를 나타낸다.

그림 6의 1행은 해싱 테이블을 구성하는 엔트리를 나타내는 것으로 논리주소, 저장장치번호, 물리블록번호로 구성되어 있다. 2행의 map() 함수는 상위에서 입출력 요청이 발생할 때마다 호출되는 매핑 함수로서 논리주소에 대한 물리주소를 반환하는 역할을 수행한다. 6~15행은 스냅샷 데이타에 대한 입출력 요청이 발생하는 경우를 말하며, 스냅샷 데이타는 변경되지 않아야 하므로 읽기만 가능하다. 스냅샷 데이타를 읽고자 한다면 먼저 논리주소를 COW 해싱 테이블을 검색하여 존재하면 해당 물리주소를 반환하고, 그렇지 않다면 수식에 의해 물리주소를 반환한다. 16~42행은 스냅샷 풀이 아닌 일반 풀에 대한 입출력 요청에 대한 것으로 해당 논리주소에 매핑된 물리주소를 반환한다. 이때 스냅샷이 존재하면 쓰기 요청이나 확장모드의 경우를 고려해야 하며, 존재하지 않는다면 수식에 의해 매핑된 물리주소를 반환한다. 20~38행은 스냅샷 생성 후 쓰기 요청이 발생하는 경우를 나타내며, 쓰기 요청이 발생하면 먼저 COW 해싱 테이블을 검색하여 해당 논리주소가 존재하는지 판별한다. 만일 해싱 테이블에 없다면 COW 처리를 해야 하는 블록이므로 COW 처리를 한다. 확장 모드일 경우는 COW 해싱 테이블과 더불어 역 해싱 테이블도 관리한다. 만약 확장 모드 상에서 반환될 블록이 이미 확장 영역으로 사용되고 있는 경우는 해당 블록의 내용을 다른 빈 공간으로 이동시킨 다음 반환해야 하므로 COW

```
1 : hash_entry : {lsector, rdev, rsector}

2 : map(rw, lsector, rsector, rdev)
3 : {
4 :    hash_entry entry;
5 :    entry.lsector = lsector;
6 :    if(IsSnapshotPool)
7 :    {
8 :       if(rw != WRITE)        // 스냅샷 풀에 대한 write는 처리하지 않음
9 :       {
10 :         if(COW_hash_search(entry) == 1)       // 해싱에서 찾은 경우
11 :            rdev = entry.rdev, rsector = entry.rsector; // 해싱의 엔트리 내용 반환
12 :         else
13 :            equation(lsector, rdev, rsector);  // 수식에 의해 매핑 처리
14 :      }
15 :   }
16 :   else
17 :   {
18 :      if(ExistSnapshot)      // 스냅샷이 존재하면
19 :      {
20 :         if(rw == WRITE)     // 쓰기 요청 시
21 :         {
22 :            if(COW_hash_search(entry) == 0)    // 해싱에서 없는 경우
23 :            {
24 :               if(ExtendMode)         // 확장 모드인 경우
25 :               {
26 :                  if(IsAlloc(entry) == 1)      // 사용할 블록이 이미 할당된 경우
27 :                  {
28 :                     COW_blk_move(entry);      // COW 블록을 새 공간으로 이동
29 :                     COW_hash_update(entry);   // 이동한 COW 블록 정보 수정
30 :                     Inv_COW_hash_update(entry);        // 역 해싱 테이블 수정
31 :                  }
32 :                  COW_snapshot(entry);         // COW 수행
33 :                  Inv_COW_hash_insert(entry);  // 역 해싱 테이블에 삽입
34 :               }
35 :               else
36 :                 COW_snapshot(entry); // COW 수행
37 :            }
38 :         }
39 :      }
40 :      equation(lsector, rdev, rsector);        // 수식에 의해 매핑 처리
41 :   }
42 : }
```

그림 6 스냅샷을 포함한 매핑 알고리즘의 의사코드

블록을 이동시켜야 하며 COW 해싱 테이블과 역 해싱 테이블의 내용도 변경해야 한다. 그 외의 경우는 40행처럼 수식에 의해 매핑된 물리주소를 반환함으로서 map() 함수가 끝이 난다.

3.3 온라인 재구성

온라인 볼륨 재구성이란 저장장치를 추가하거나 삭제할 때 부하 균등을 위해서 시스템을 정지시키지 않고 변경된 내용을 포함하도록 논리 볼륨을 재구성하는 것을 말한다. 이 논문에서 제안하고 있는 온라인 재구성 방법은 데이타의 병행 읽기 성능을 높이기 위해 데이타를 스트라이핑 방법으로 저장하는 스트라이핑 볼륨을 대상으로 하며, 저장장치가 추가되는 경우에 대해 적용한다.

스트라이핑 볼륨에 데이타를 저장할 때 요구한 블록 번호를 볼륨에 참여하는 저장장치수로 모듈러 연산을 통해 저장할 장치의 위치를 구한다. 이때 저장장치를 추가하고자 한다면 저장장치수가 변경되므로 저장된 데이타를 추가된 장치를 포함한 전체 저장장치 수를 고려하여 부하 균등을 이루도록 재구성을 수행한다. 이러한 재구성 연산은 기존의 데이타를 대상으로 재구성이 완료될 때까지 시스템에 적잖은 부담을 주게 된다. 그러나 데이타의 양이 급속하게 증가하고 있는 상황에서 재구성 연산을 수행할 때 그 부담으로 인하여 상위의 정상 요청을 제대로 수용하지 못하는 상황이 발생한다. 따라

서 재구성 연산을 수행하면서 상위의 정상 요청을 적절히 처리할 수 있는 방법이 필요하다. 이 논문에서는 수식에 의해 매핑되는 정상 할당 영역에 대해 재구성 연산을 수행하면서 동시에 상위의 정상 요청을 효과적으로 수용하는 방법을 제안함으로서 데이타에 대한 가용성을 높인다.

이 논문에서 제안하는 온라인 재구성 방법의 특징은 다음과 같다. 첫째, 재구성을 수행할 블록들은 저장장치가 추가되기 전에 할당된 블록들에 대해서만 재구성을 수행한다. 이를 위해 저장장치가 추가되기 전에 할당된 블록들을 할당되지 않은 블록들과 구분할 필요가 있다. 따라서 재구성을 수행하기 전에 비트맵을 검색하여 블록들의 할당 유무를 관리한다. 이때 관리하는 데이타는 할당된 논리블록들로 테이블을 구축하여 논리주소 순으로 관리한다. 재구성이 시작되면 이 테이블을 참고하여 논리주소 순으로 순차적으로 해당 블록들에 대해 재구성을 수행한다. 테이블 내의 모든 블록들에 대해 재구성을 모두 수행하면 재구성 연산은 종료하게 된다.

둘째, 상위의 정상 요청을 처리하기 위해 요청된 블록이 재구성이 완료된 블록인지를 관리한다. 이는 해당 블록이 재구성이 수행되었는지에 여부에 따라 저장장치가 추가되기 전의 매핑 수식을 적용할 것인지 추가된 후의 매핑 수식을 적용할 것인지 결정되기 때문이다. 이때 요청된 논리주소에 대한 물리주소는 (저장장치번호, 물리블록번호)로 표현된다. 여기서 저장장치번호는 논리주소를 볼륨에 참여하는 전체 저장장치수로 모듈러 연산을 통해 구해지고, 물리블록번호는 논리주소를 전체 저장장치수로 나누어 구할 수 있다. 이와 같이 논리주소에 대한 물리주소는 볼륨에 참여하는 저장장치 수에 따라 달라진다. 따라서 저장장치가 추가되기 전과 후의 매핑 수식이 상이하므로 요청된 블록이 재구성이 완료된 블록인지의 여부에 따라 반환하는 물리주소가 달라진다. 이를 위해 논리주소에 대한 재구성 여부를 관리하는 재구성 테이블을 구성한다. 재구성 테이블의 내용은 논리주소와 이에 매핑된 물리블록의 재구성 완료 여부로 표현한다.

셋째, 상위의 정상 요청이 쓰기 요청일 경우 해당 블록에 대해 재구성을 수행한 후 재구성된 블록을 반환한다. 이는 [15]에서 제안한 연쇄적 재구성 방법과 동일하다. 변경될 블록이 이동할 자리에 할당된 블록이 존재할 경우는 그 블록도 재구성을 수행한다. 이런 과정이 이미 비어있는 블록으로 이동할 때까지 연쇄적으로 발생한다. 재구성될 블록이 이동할 자리가 추가된 저장장치라면 반드시 비어 있으므로 재구성이 종료되며, 기존 저장장치로 이동할 경우는 해당 자리가 비어 있느냐에 따라 재구성이 더 수행될지 종료될지 결정된다.

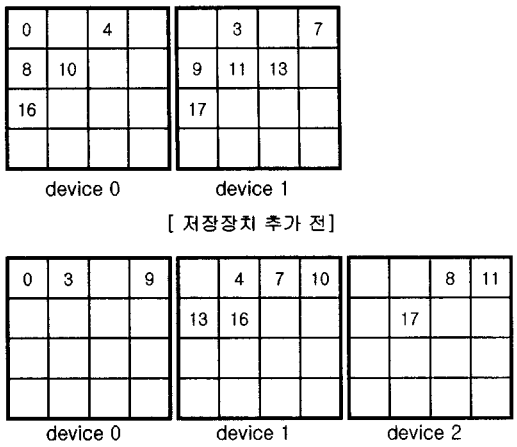

그림 7 재구성 방법

그림 7은 새로운 저장장치가 추가되었을 경우 재구성하는 방법에 대해 나타내고 있다. 그림 7과 같이 재구성을 수행하는 단계를 살펴보면 다음과 같다. 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 논리주소 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.

재구성 중 상위의 정상 요청을 처리하는 과정은 다음과 같다. 먼저 상위의 정상 요청이 내려오면 수행중인 재구성을 상위의 정상 요청을 처리할 때까지 잠시 중단한다. 이때 상위의 정상 요청이 읽기라면 재구성 테이블을 검색하여 해당 블록의 재구성 여부를 확인한다. 해당 블록이 재구성이 완료된 블록이라면 저장장치가 추가된 후의 매핑 수식을 적용하며, 반대의 경우는 저장장치가 추가되기 전의 수식을 적용하여 물리블록을 반환한다. 그림 7에서 논리주소 3에 대해 읽기가 요청되면 그림 8의 재구성 테이블을 이용해 재구성 여부를 검색한다. 이미 논리주소 3은 재구성이 완료된 블록이므로 저장장치가 추가된 후의 매핑 수식을 적용한다. 저장장치번호는 논리주소 3을 전체 저장장치 수 3으로 모듈러 연산을 통해 device 0이 구해지고, 물리블록번호는 논리주소 3을 전체 저장장치 수 3으로 나누어 1번 블록이 구해진다. 이때 논리주소와 저장장치번호, 물리블록번호는 0부터 시작한다.

재구성 중 쓰기 요청이 발생하면 재구성 테이블에서

논리주소	0	1	2	3	4	5	6	7	8	9		47
재구성 대상	1	0	0	1	1	0	0	1	1	1	0	0
재구성 여부	1	0	0	1	0	0	0	0	0	0	0	0

그림 8 재구성 테이블

쓰기 요청에 해당하는 논리주소의 재구성 여부를 확인한다. 재구성이 수행된 블록이면 저장장치가 추가된 후의 매핑 수식을 적용하여 해당 블록을 반환한다. 만일 재구성이 수행되지 않은 블록이라면 재구성을 수행한다. 이때 재구성될 블록과 연관된 블록들에 대해 연쇄적 재구성 방법이 수행된다. 그림 8과 같이 논리주소 3까지 재구성이 이루어진 상태에서 논리주소 6에 대한 쓰기 요청이 발생하면 재구성 테이블을 검색하여 논리주소 6의 재구성 여부를 확인한다. 논리주소 6은 재구성 대상이 아니므로 재구성이 안된 블록이다. 논리주소 6에 대해 재구성을 수행하면 (저장장치번호, 물리블록번호)가 (device 0, 2)로 결정된다. 이때 (device 0, 2)에는 논리주소 4에 해당하는 블록정보가 존재한다. 따라서 논리주소 4에 대해서도 재구성을 수행해야 한다. 이런 과정을 통해 재구성될 블록이 빈 블록으로 이동할 때까지 수행한다. 그림 9와 그림 10은 각각 논리주소 6에 대해 재구성이 끝난 후의 재구성 과정과 재구성 테이블을 나타낸다.

그림 11과 그림 12는 각각 재구성과 재구성 중 상위의 정상 요청을 처리하는 알고리즘의 의사코드를 나타낸다.

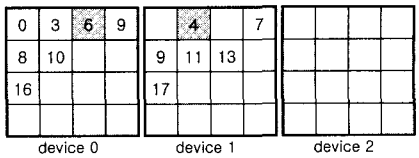

그림 9 논리주소 6에 의한 재구성 방법

논리주소	0	1	2	3	4	5	6	7	8	9		47
재구성 대상	1	0	0	1	1	0	0	1	1	1	0	0
재구성 여부	1	0	0	1	1	0	1	0	0	0	0	0

그림 10 논리주소 6의 재구성 수행후의 재구성 테이블

그림 11의 1행은 재구성 테이블을 구성하는 엔트리를 나타내는 것으로 논리주소, 할당여부(재구성 대상), 재구성 여부로 구성되어 있다. 2행의 resize_volume() 함수는 볼륨을 재구성할 때 호출되는 재구성 함수로서 재구성이 끝나면 종료한다. 5, 21행은 볼륨의 상태를 정상 상태에서 재구성 모드로 또는 반대로 전환하는 과정을 나타낸다. 6~8행은 재구성 테이블을 작성하는 과정으로서 논리주소 순으로 비트맵을 검색하여 재구성 대상인 블록을 설정하여 테이블을 구축한다. 이때 재구성 중 상위의 정상 요청을 처리하기 위해 재구성 테이블에 대한 잠금을 걸어 재구성 테이블의 일관성을 유지시킨다. 이는 재구성 시 블록을 이동할 때에서도 잠금을 획득하여 재구성 테이블을 변경시킨다. 9~19행은 논리주소 순으

```
1 : resize_map_ent : lsector, allocated, resized

2 : resize_volume()
3 : {
4 :    resize_map_ent ent[];

5 :    resizing_on = TRUE;        // 재구성 모드 지정
6 :    resize_maptable_write_lock();
7 :    make_resize_maptable(ent);          // 재구성 테이블 생성
8 :    resize_maptable_write_unlock();
9 :    while(!endofent(ent))
10 :   {
11 :      resize_maptable_read_lock();
12 :      read_cur_ent(ent);      // 재구성 테이블의 현재 엔트리 읽기
13 :      resize_maptable_read_unlock();

14 :      resize_maptable_write_lock();
15 :      move_block(ent);        // 현재 엔트리의 블록 재구성
16 :      update_resize_maptable(ent);    // 재구성 테이블 변경
17 :      resize_maptable_write_unlock();
18 :      cur_ent_inc(ent);       // 현재 엔트리 다음으로 이동
19 :   }
20 :   remove_resize_maptable(ent);        // 재구성 테이블 삭제
21 :   resizing_on = FALSE;       // 재구성 모드 해제
22 : }
```

그림 11 재구성 알고리즘의 의사 코드

```
1 : map(rw, lsector, rsector, rdev)
2 : {
3 :    resize_map_ent ent;
4 :    ent.lsector = lsector;

5 :    if(resizing_on == TRUE)   // 재구성 중
6 :    {
7 :       resize_maptable_write_lock();   // 재구성 테이블의 쓰기 락 획득
8 :       search_resize_maptable(ent);    // 재구성 테이블 검색

9 :       if(ent.resized == 0)   // 재구성이 안된 블록
10 :      {
11 :         if(rw != WRITE)     // 읽기 요청일 경우
12 :            old_equation(lsector, rdev, rsector);       // 이전 수식에 의해 처리
13 :         else       // 쓰기 요청일 경우
14 :         {
15 :            cascade_resizing(ent);    // 연쇄적 재구성 수행, 재구성 테이블 수정
16 :            new_equation(lsector, rdev, rsector);       // 새 수식에 의해 처리
17 :         }
18 :      }
19 :      else // 재구성된 블록
20 :         new_equation(lsector, rdev, rsector); // 새 수식에 의해 처리
21 :
22 :      resize_maptable_write_unlock(); // 재구성 테이블의 쓰기 락 해제
23 :   }
24 :   else
25 :   {
26 :      ...  // 정상 매핑 수행
27 :   }
28 : }
```

그림 12 재구성 중 상위의 정상 요청을 처리하는 알고리즘의 의사코드

로 해당 블록을 재구성하는 과정을 수행한다. 20행에서는 재구성이 끝나면서 재구성 테이블을 삭제한다.

그림 12에서 1행의 map() 함수는 상위의 정상 요청을 처리하는 매핑 함수이다. map() 함수는 재구성이 아닌 경우는 그림 6과 같은 정상적인 매핑을 수행한다. 재구성 중인 경우는 5~23행의 과정을 거친다. 7, 22행은 재구성 테이블의 일관성을 유지하기 위해 잠금을 획득하여 테이블을 접근한다. 8행에서는 매핑을 수행하기 전에 재구성 테이블을 검색하여 요청된 블록이 재구성 된 블록인지를 판단한다. 9~18은 검색결과 재구성이 안된 블록에 대한 처리과정으로 읽기 요청일 경우는 이전 수식을 적용해서 처리하고 쓰기 요청일 경우는 연쇄적 재구성을 수행하고 새로운 수식을 적용한다. 20행은 이미 재구성된 블록일 경우 새로운 수식을 적용하여 매핑을 처리하는 과정을 나타낸다.

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 아직 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.

3.4 자유공간 관리자

이 논문에서 제안하는 자유공간 관리기법은 비트맵 구조를 사용한다. 각 블록의 사용여부를 한 비트에 대응시켜서 표시한다. 자유공간 정보는 매핑 시 접근하는 중요한 메타 데이타이기 때문에 저장장치에 저장되며, [6]의 메타 데이타 관리 방법처럼 SAN에 참여하는 서버들에 분할하여 관리한다.

자유공간의 할당 요청이 들어오는 경우를 생각해 보면 정상공간 할당 요청과 스냅샷이나 온라인 재구성을 위한 공간 할당 요청이 있다. 정상공간 할당 요청은 수식에 의해 매핑된 블록을 할당하고, 스냅샷이나 온라인 재구성을 위한 공간 할당 요구는 예약 영역의 뒤에서부터 할당한다. 제안하는 방법에서는 스냅샷이나 온라인 재구성을 위한 공간 할당의 경우 다음 할당될 위치에 대한 정보를 추가하여 비트맵 탐색 시간을 줄인다.

4. 혼합 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, Linux상의 SAN 볼륨관리자인 GFS의 pool을 기반으로 구현되었다. FC RAID는 PowerPC 64bit RISC Microprocessor가 탑재된 JET-RA5000FF에 36GB 10,000 rpm RAID 디스크 드라이브를 3개 사용하였으며, FC Switch는 브로케이드의 8포트 루프 스위치인 Silkworm 2010을 사용하였다.

구현한 매핑 관리자는 최초 볼륨 생성 시 정상 할당 영역과 예약 영역으로 구분하여 관리한다. 정상 할당 영역은 수식에 의해 매핑이 처리되는 공간이고, 예약 영역은 스냅샷과 온라인 재구성을 위한 공간으로 활용하도록 구현하였다. 이때 예약 영역이 부족한 경우 정상 할당 영역을 활용할 수 있게 확장 가능하도록 구현하였다. 스냅샷에 대해서는 COW를 처리하기 위해 예약 영역에 해싱 테이블을 유지하고, COW 블록은 예약 영역에 할당하고, 최신의 데이타는 정상 할당 영역에 유지하여 수식에 의해 매핑 처리되도록 하였다. 온라인 재구성은 스트라이핑 볼륨에 적용하였으며, 재구성 시 정상 할당 요청을 수용하기 위해 재구성 테이블과 해싱 테이블을 유지하도록 구현하였다.

이 논문에서 제시한 매핑 관리자를 구현한 SAN 논리볼륨 관리자의 블록 설계도는 그림 13과 같다. 제안한 매핑 기법을 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 처리하기 위해서 물리블록을 할당하거나 스냅샷과 온라인 재구성을 처리하기 위해서 물리블록을 할당하기 위한 자

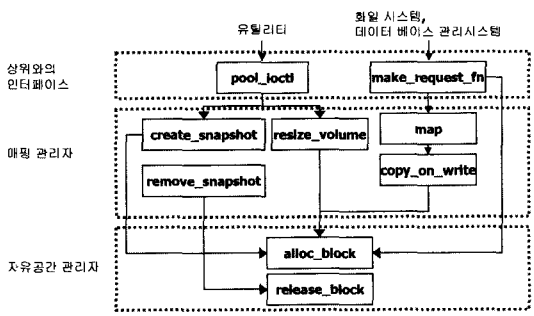

그림 13 구현한 논리볼륨 관리자의 블록 설계도

유공간 관리자 함수로 alloc_block()이 있다. 마지막으로 사용자의 스냅샷 요구나 온라인 볼륨 재구성 요구를 처리하기 위한 create_snapshot(), remove_snapshot(), resize_volume()이 그림 13과 같은 호출 관계를 유지하고 있다.

4.2 성능 평가

GFS Pool의 매핑방법, SANTOPIA 논리볼륨의 매핑 방법과 제안하는 방법을 비교를 통해 성능평가를 진행하였다. GFS의 Pool은 수식기반의 매핑기법을 이용하고 있고, SANTOPIA의 논리볼륨은 전형적인 테이블기반의 매핑기법을 이용한다. 이들 방법과 입출력 연산, 부가 저장공간 측면에서 비교를 하여 제안하는 방법의 우수성을 보인다.

기존 방법과 제안하는 매핑 방법들에 대한 비교를 해보면 표 1과 같다. 매핑 수행 속도 측면에서는 수식 기반의 GFS pool이 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. 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 공간

표 1 매핑 기법들의 성능 비교

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

이 필요하다. 혼합 매핑 기법은 스냅샷과 온라인 재구성을 위해 미리 예약된 공간을 할당해야 한다. 이를 통해 스냅샷이나 온라인 재구성을 지원할 수 있다. 그러나 이를 지원하기 위하여 비트맵 영역이 필요하며, 정상 할당 요청의 경우에도 반드시 비트맵 변경을 위해 한번의 디스크 입출력이 필요하다. 매핑 수행 속도는 GFS pool에 다소 느리지만 테이블 기반 매핑보다는 빠르다.

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 혼합 매핑 기법이 스냅샷과 온라인 볼륨 재구성 기능을 지원하면서 이를 지원하지 못하는 수식 기반의 매핑 기법에 비해 어느 정도의 매핑 속도 차가 발생하는지에 초점을 두었다. 또한 [6]에서 구현한 테이블 기반의 매핑 방법과의 매핑 속도 차도 비교하였다.

성능 평가를 하기 위해 고려한 파라미터들은 표 2와 같다. 실험에서 측정한 것은 파일의 쓰기와 읽기 연산의 응답시간이다. 파일의 읽기와 쓰기는 read(), write() 시스템 함수를 이용하였다. 실험은 다음과 같은 순서로 진행되었다. 최초에 크기가 1Mbytes, 2Mbytes, ..., 10Mbytes인 파일 10개를 생성한다. 이후에 읽기 연산의 응답시간을 측정하기 위해 4K의 읽기요구를 갖는 읽기 연산을 통해 생성한 전체 파일을 10회 반복해서 읽는다. 그리고, 생성시간을 측정하기 위해서 1~10MBytes의 파일의 생성을 10회 반복해서 수행한다. 실험은 제안하는 볼륨관리자위에 GFS를 적용하여 수행하였으며 실험에 쓰인 비교 대상은 GFS의 pool과 [6]에서 구현한 테이블 기반의 매핑을 수행하는 논리볼륨 관리자였다.

추가적으로 제안하는 논리볼륨 관리자에서 제안하는 스냅샷 방법과 테이블 기반의 매핑을 수행하는 볼륨 관리자의 스냅샷 방법에 대한 비교도 수행하였다. 측정한 부분은 스냅샷 생성의 평균 응답시간이다. 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 스트라이핑 볼륨에 대한 재구성에 초점을 두어 설계하였기 때문에 다른 방법들과 비교 평가하지 않는다.

수식 기반 매핑과 테이블 기반 매핑, 혼합 매핑 방법의 파일에 대한 쓰기 연산의 평균 응답시간은 그림 14와 같다. 성능 평가 결과 혼합 매핑 방법이 수식 기반 매핑 방법에 비해 28%의 응답시간 증가를 보이고 있고, 테이블 기반 매핑에 비해서는 30%의 응답시간 감소를 보이고 있다. 이는 쓰기 연산 시 수식에 의해 매핑을 처리하는 수식 기반 매핑 방법에 비해 혼합 매핑 방법은 예약 영역의 확장에 대비하여 수식에 의해 매핑된 블록의 사용유무를 비트맵에 반영하는 절차를 더 거치기 때문이다. 또한 혼합 매핑 방법은 수식에 의해 매핑을 처리하므로 매핑 테이블이 필요하지 않기 때문에 비트맵과 매핑 테이블을 둘다 접근하는 테이블 기반의 매핑 방법보다 나은 결과를 나타낸다.

그림 15는 수식 기반 매핑과 테이블 기반 매핑, 혼합 매핑 방법의 파일에 대한 읽기 연산의 평균 응답 시간에 대한 성능 평가이다. 평가 결과 읽기 연산의 경우는 혼합 매핑 방법과 수식 기반의 매핑 방법이 동일한 성능을 보였으며 테이블 기반 매핑 방법에 비해 혼합 매핑 방법이 26%의 응답시간 감소를 보이고 있다. 이는 읽기 연산 시 혼합 매핑 방법은 수식 기반의 매핑만 수행하므로 수식 기반의 매핑 방법과 같은 응답시간을 나타낸다. 테이블 기반 매핑 방법은 읽기 연산 시 매핑 테이블에 매핑 정보를 접근하기 때문에 다른 방법들에 비해 응답시간 증가를 보이는 것이다.

그림 16은 테이블 기반 매핑의 테이블 복사 방법과

표 2 성능 평가를 위한 파라미터들

풀 크기(GB)	1 ~ 5
파일 크기(MB)	1 ~ 10
파일 연산 종류	읽기, 쓰기

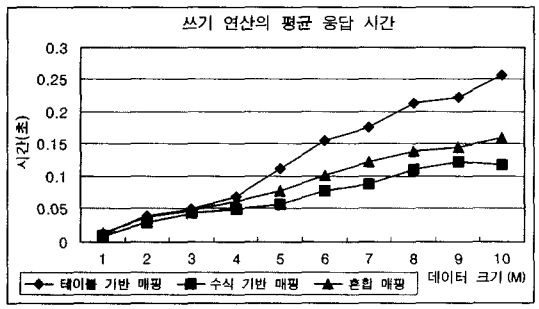

그림 14 매핑 방법에 따른 쓰기 연산의 평균 응답 시간

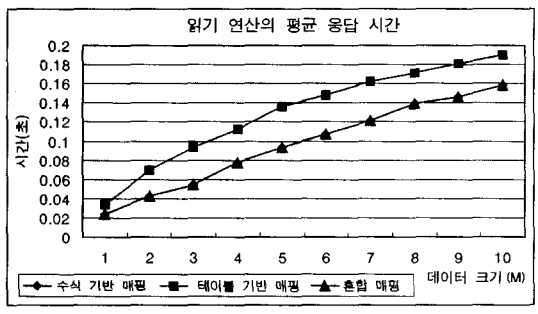

그림 15 매핑 방법에 따른 읽기 연산의 평균 응답 시간

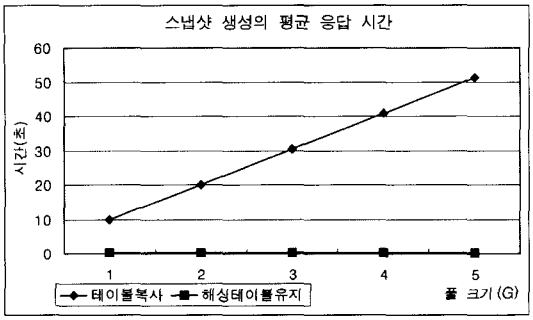

그림 16 스냅샷 방법에 따른 스냅샷 생성의 평균 응답 시간

혼합 매핑의 해싱 테이블 유지 방법에 대해서 풀 크기에 따른 스냅샷 생성의 평균 응답시간에 대해 성능 평가한 것이다. 결과를 보면 해싱 테이블 유지 방법은 풀의 크기에 상관없이 일정한 생성 응답시간을 유지하는데 반면, 테이블 복사 방법은 풀의 크기가 커짐에 따라 생성 응답시간이 일정하게 늘어남을 알 수 있다. 이는 테이블 복사 방법은 스냅샷 생성 시 매핑 테이블 전체를 다른 영역에 복사하지만, 해싱 테이블 유지 방법은 다른 영역에 해싱 테이블만 초기화하는 것으로 스냅샷 생성이 끝나기 때문이다.

마지막으로 테이블 기반 매핑 방법과 혼합 매핑 방법에서 스냅샷 데이타를 접근하는데 걸리는 시간을 비교해 본다. 테이블 기반 매핑 방법에서는 스냅샷 데이타를 접근하기 위해 별도의 공간에 복사된 매핑 테이블을 활용한다. 따라서 스냅샷 데이타에서 접근하고자 하는 논리주소에 대응하는 물리주소를 한번의 디스크 입출력을 통해 알 수 있다. 이때 매핑 테이블이 차지하는 공간은 볼륨의 크기에 비례하여 커진다.

혼합 매핑 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 한번의 디스크 입출력을 통해 알 수 있다. 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 두 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.

테이블기반 매핑기법과 제안하는 방법의 부가 저장공간 측면을 비교해 보았다. 수식기반의 GFS는 부가저장공간이 전혀 필요없는 방법이므로 비교에서 제외하였다. 볼륨의 크기는 5GBytes이고 파일시스템의 한 블록의 크기는 4KBytes로 하였다. 이때 테이블기반의 방법은 매핑테이블과 비트맵을 저장하기 위한 공간으로 약 12.6 MBytes를 소요하였다. 이 크기는 블록의 크기를 어떻게 하는가에 따라서 또 달라진다. 이에 반해서 제안하는 혼합매핑 기법은 비트맵저장을 위해 약 156 KBytes를 소요한다. 메타데이타를 저장하는데 필요한 부가공간은 제안하는 방법이 테이블기반의 방법에 비해 99% 정도를 절약한다.

하지만 제안하는 방법은 스냅샷과 재구성을 위해 볼륨의 일정크기를 예약한다. 즉, 이 예약 공간은 스냅샷과 재구성이 발생하지 않는 동안은 불필요하게 낭비되는 공간이다. 실험에서는 전체 볼륨의 5%(250MBytes)를 예약공간으로 사용하였다. 이 공간을 얼마로 하는가는 볼륨을 구성하는 사용자가 결정하는 것이며 볼륨이 사용되는 영역의 특성에 따라서 적절히 조정한다. 공간을 너무 작게 할당하게 되면 스냅샷이나 재구성 처리 시간이 길어질 것이다. 반대로 너무 크게 할당하면 불필요하게 공간을 유휴상태로 두게되는 문제가 발생한다.

지금까지의 성능 평가를 보면, GFS의 pool에 비해 제안하는 매핑 관리자의 파일에 대한 쓰기 연산의 경우 28%의 평균 응답시간 증가를 보였으며 읽기 연산은 동일했다. [6]의 테이블 기반의 매핑 방법에 비해서는 쓰기 연산과 읽기 연산의 경우 각각 30%, 26%의 평균 응답시간 감소를 보였다. 결과적으로 혼합 매핑 방법이 테이블 기반의 매핑에 비해 좋은 성능을 보이고 있으며, GFS pool에 비해서는 쓰기 연산의 경우만 약간의 성능 저하를 보이고 있다. 이는 혼합 매핑 기법에서 스냅샷이나 온라인 재구성을 지원하기 위해 필요한 메타 데이타를 유지하기 위한 오버헤드로 발생하는 차이이다.

그러나 지금과 같이 논리 볼륨에 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. 또한 블록 할당의 기본 단위를 파일 시스템에 맞게 최적화 한다면 보다 나은 성능을 기대할 수 있을 것이다. 따라서 이 논문에서 제안하는 혼합 매핑 방법은 매핑 속도 면에서 기존의 GFS pool의 수식 기반의 매핑 방법에 비해 쓰기 연산 시 다소 차이를 보였지만, GFS pool이 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 매핑 방법은 스냅샷이나 온라인 볼륨 재구성과 같은 동적인 환경에 유연하게 대처하지 못하였다. 이를 위해 제안한 매핑 관리자에서는 수식 기반의 매핑을 통해 빠른 매핑을 수행하면서도 스냅샷과 온라인 볼륨 재구성 기능을 지원할 수 있는 혼합 매핑 기법을 설계하였다.

제안한 매핑 관리자는 GFS의 pool의 매핑 관련 부분에 구현되었으며, GFS의 pool과 비교 평가하였다. 성능 평가 결과에서 보듯이 제안하는 매핑 관리자가 매핑 속도면에서 GFS의 pool에 비해 다소 성능 저하를 보였다. 이는 GFS의 pool이 지원하지 못하는 스냅샷과 온라인 볼륨 재구성 기능을 제공하기 위한 추가적인 부담이 발생하기 때문이다. 또한 이러한 추가적인 부담은 다중 호스트 환경 하에서 줄어들 수 있다. 따라서 이 논문에서 제안하는 매핑 관리자가 수식 기반의 GFS의 pool보다 SAN 환경에 적합함을 알 수 있다.

향후 연구방향으로, 이 논문에서 제안한 매핑 관리자를 보완하여 다중 호스트 환경 하에서 성능 평가하는 것이다.

참 고 문 헌

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

[2] Thomas Ruwart, "Disk Subsystem Performance Evaluation: From Disk Drives to Storage Area Networks," IEEE Symposium on Mass Storage Systems, pp. 1~24. 2000.

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

[4] 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, pp. 319~342, 1996.

[5] 김경배, 김영호, 김창수, 신범주, "SAN을 위한 전역 파일 공유 시스템의 개발," 한국 정보 과학회지, Vol. 19, No. 3, pp. 24~32, 2001.

[6] 남상수, 홍현택, 피준일, 송석일, 유재수, "SAN 논리볼륨 관리자를 위한 매핑 및 자유공간 관리기법의 설계 및 구현", 한국정보과학회 춘계학술대회논문집, Vol. 29, No. 1, pp. 4~6, 2002.

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

[8] Chandramohan A. Thekkath, Timothy Mann and Edward K. Lee, "Frangipani: A Scalable Distributed File System," In Proceedings of the sixteenth ACM symposium on Operating systems principles, volume 31, pages 224~237, 1997.

[9] 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, Vol. 986 of Lecture Notes in Computer Science, pp. 1~116, 1995.

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

[11] 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.

[12] 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.

[13] 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, 2000.

[14] 박유현, 김창수, 강동재, 김영호, 신범주, "스트라이핑 시스템에서 디스크 추가를 위한 계산에 의한 매핑 방법", 한국멀티미디어학회 논문집, Vol. 9, No. 1, pp. 15~27, 2003. 2.

[15] 박유현, 김창수, 김영호, 강동재, 김학영, "스트라이핑 시스템에서 쓰기 연산에 의한 연쇄적 재구성 방법", 한국정보처리학회 추계학술대회논문집, Vol. 9, No. 1, pp. 213~216.

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

피 준 일

1999년 충북대학교 공과대학 컴퓨터 공학과(공학사). 2001년 충북대학교 정보통신공학과(공학석사). 2002년 충북대학교 정보통신공학과(박사과정 재학 중). 관심분야는 고차원 색인 구조, 데이타 베이스 시스템, 메모리 상주형 데이타 베이스 시스템, 저장 시스템, 실시간 시스템 등

송 석 일

1998년 충북대학교 정보통신공학과(공학사). 2000년 충북대학교 정보통신공학과(공학석사). 2003년 충북대학교 정보통신공학과(공학박사). 2003년 3월~2003년 8월 KAIST 전자전산학과 박사후연구원 2003년 8월~현재 충주대학교 컴퓨터공학과. 관심분야는 데이타베이스 시스템, 트랜잭션, 저장 시스템, 멀티미디어 정보검색, XML, 정보검색 프로토콜 등

유 재 수

1989년 전북대학교 공과대학 컴퓨터공학과(학사). 1991년 한국과학기술원 전산학과(공학석사). 1995년 한국과학기술원 전산학과(공학박사). 1995년~1996년 목포대학교 전산통계학과 전임강사. 1996년~현재 충북대학교 전기전자컴퓨터공학부 부교수 및 컴퓨터·정보통신연구소 소장. 관심분야는 데이타베이스 시스템, 정보검색, 멀티미디어 데이타베이스, 분산 객체 컴퓨팅

최 영 희

호원대학교 디지털영상정보학부 교수
1983년 광운대학교 전자공학과 학사
1986년 광운대학교 컴퓨터공학과 석사
2003년 순천대학교 컴퓨터과학과 박사
관심분야는 데이타베이스시스템, 정보시스템, 프로그래밍언어

이 병 엽

1991년 한국과학기술원 전산학과 공학사 1993년 한국과학기술원 전산학과 공학석사. 1997년 한국과학기술원 경영정보공학 공학박사. 1997년~2003년 2월 대우정보시스템 eBiz 사업본부. 2003년 3월~현재 배재대학교 전자상거래학부 전임강사 관심분야는 데이타마이닝, XML, 인공지능, 멀티미디어 데이타베이스, 분산 객체 컴퓨팅 등