스트라이핑 시스템에서 디스크 추가를 위한 계산에 의한 매핑 방법

박유현[†] · 김창수[†] · 강동재[†] · 김영호[†] · 신범주[††]

요 약

최근 멀티미디어 개발 환경의 발달로 멀티미디어 서비스를 제공하는 서버에서 다루는 데이터의 양은 급격히 증가 하고 있으며, 이에 따라 저장장치의 확장성(scalability)이 요구된다. 최근 저장장치 기술에 관한 내용 중에 SAN과 같은 기술은 저장장치의 확장에 유리하며, RAID 0, 5를 통해 다중 디스크로부터 데이터를 동시에 읽을 수 있다. RAID 0 (striping), RAID 5(striping with parity)는 계산식에서 의해서 논리주소를 물리주소로 바꾸는데, 수식 기반 매핑을 수행하는 시스템에서 디스크를 확장하려면 전체 데이터를 재구성해야 하는 문제점이 발생한다. 최근에는 이러한 유연성 문제를 해결하기 위해 매핑 테이블을 쓰기도 하지만 매핑 테이블의 크기가 매우 크기 때문에 데이터를 모두 메모리에 올려놓을 수 없다. 따라서 데이터가 실제 저장된 물리주소를 알기 위해서는 추가적인 디스크 I/O가 요구되기 때문에 전체적인 성능이 떨어진다. 이 논문에서는 스트라이핑 시스템에서 저장장치의 확장을 지원하는 수식 기반 매핑 방법을 제안한다. 제안하는 방법은 SZIT라 불리는 간단한 메타데이터와 계산식을 적용하여 매핑하기 때문에 기존의 계산에 의한 매핑방법에서와 같은 빠른 매핑을 할 수 있으면서 동시에 디스크 확장의 유연성을 가질 수 있다. SZIT 기반 매핑 방법은 스트라이핑 시스템에서 저장용량을 증가시키기 위해 디스크를 추가한 경우, 시스템을 정지시킬 필요 없이 계속 서비스를 제공할 수 있는 장점을 가진다. 따라서 SZIT 기반 매핑방법은 실시간 서비스를 해야 하는 시스템에서 온라인 디스크 확장을 가능하게 한다.

The Mapping Method by Equation for Adding Disks for Striping System

BAK, Yuhyeon[†], KIM, Changsoo[†], KANG, Dongjae[†], KIM, Youngho[†] and SHIN, Bumjoo[††]

ABSTRACT

Recently, the volume of data is increasing rapidly in server for multimedia service, according to development of multimedia application environment. In recent research for storage technology, the technology like of the SAN(Storage Area Network) advantages in scalibiltiy of storage devices, and can read data from multiple disk arrays through RAID 0, 5. The RAID 0 and 5 translate logical address to physical address using equation, but in case of adding disks at the system with equation-based mapping, the problem that we must rearrange the whole data in the previous disks happens. We use the mapping table to solve this problem in recent, but we can not load the whole mapping table in main memory because it occupies too large space. Therefore the extra I/Os are demanded to evaluate real physical address of data, so total performance of the system is degraded. In this paper, we propose the mapping method that supports the scalibility in RAID 0 or 5 system. The proposing method applies small metadata, so-called SZIT and simple equation, so it is possible that we make translate logical address to physical address rapidly, and it is scalable in disk extending simultaneously. Our suggesting method, if we add disks to the striping system for expanding of storage capacity, has an advantage of never stop service. So, SZIT-based mapping method can do online-disk-expanding in real-time service.

Key words: 스트라이핑 지역정보 테이블 (SZIT), RAID, 스트라이핑, 매핑

접수일 : 2002년 2월 18일, 완료일 : 2002년 9월 19일

[†] ETRI 컴퓨터 · 소프트웨어 연구소 컴퓨터 시스템연구부 연구원

[††] 밀양대학교 컴퓨터 · 정보통신공학부 교수

1. 서 론

인터넷 사용의 대중화는 작업 환경의 급속한 변화와 함께 저장되어야 할 데이터 양의 폭발적인 증가를 가져왔다. 특히, 멀티미디어 데이터의 영향으로 기존 컴퓨터에서 다루던 데이터의 크기보다 데이터 평균적인 크기가 매우 커졌으며, 컴퓨터 사용자의 증가로 생성되는 데이터의 양 또한 증가하였다. 그러나, 하나의 서버에 접속되어 데이터를 저장하고 관리하는 클라이언트 서버 형태의 데이터 관리 시스템이나 파일서버를 기반으로 하는 네트워크 파일 시스템과 같은 기존의 자료저장 시스템들은 기하급수적으로 늘고 있는 엄청난 양의 데이터를 처리하는 데 한계가 있다[1]. 이러한 문제를 해결하기 위하여 등장한 것이 SAN(Storage Area Network)[2,3]이다. SAN은 파이버 채널(fibre channel)[4]로 연결되는 고속의 저장장치 전용 네트워크로 서버에 디스크를 직접 연결한 기존의 DAS(Direct Attached Storage)와 달리 저장 장치를 고속의 파이버 채널에 직접 연결시켜 사용할 수 있기 때문에 고성능(high performance), 고확장성(high scalability), 고가용성(high availability) 및 공유성(sharability)을 제공한다. 이 같은 SAN의 장점은 현재의 자료저장 시스템이 안고 있는 대용량 저장장치 문제들을 효과적으로 해결할 수 있게 한다.

저장장치에서 데이터를 읽어오는 성능을 향상시키고 데이터의 신뢰성을 높이기 위한 방법으로 RAID(Redundant Array of Inexpensive Disk)[5] 기술이 있는데, 그 특성에 따라 여러 단계를 제공한다. 그 중에서 RAID 0과 3, 4, 5는 디스크 배열을 구성하는 장치들에 데이터를 분산하여 저장하는 방법이다. 즉, 모든 데이터를 특정 단위(striping unit)[6]로 각 디스크에 순차적으로 써 나가는 방법으로 디스크로의 입/출력을 동시에 수행하기 때문에 병렬성이 높아진다. 일반적으로 SAN 기반의 스토리지 가상 소프트웨어는 이러한 RAID의 기능을 제공한다.

스트라이핑 방법으로 데이터를 저장하는 시스템에서 논리주소를 물리주소로 매핑 하는 방법은, 디스크 수로 모듈러 연산을 수행하여 저장할 위치 디스크를 결정하는 것이 보편적인데, 이 방법은 추가적인 데이터의 도움 없이 간단히 매핑이 가능하다는 장점을 가진다. 하지만, 사용하고 있는 도중에 새로운 디스크를 추가하고자 할 때는 시스템을 정지시키고, 물리적으로 디스크를 추가한다. 그리고 추가된 디스크를 인식하여 기존 데이터들을 재구성시킨다. 시스템을 재구성한다는 것은 전체 디스크로 분산 저장되어 있는 데이터 및 패리티 블록들을 새롭게 배치한다는 것으로, 대부분의 경우에는 재구성을 위해 시스템의 수행을 중단시킨 후 전체 디스크의 내용을 읽어서 배치 방식에 따라 다시 디스크로 쓰는 과정으로 처리된다. 따라서, 해당 디스크의 내용을 임시로 저장할 메모리에 드는 비용과 블록들의 재배치를 위한 여러 번의 디스크 읽기 및 쓰기 연산의 수행에 걸리는 시간 때문에 재구성 과정은 시스템 성능에 커다란 오버헤드가 된다. 특히 최근 활발하게 연구되고 있는 대용량 저장장치에서처럼 저장되어 있는 데이터의 양이 매우 많을 경우에는 이러한 데이터의 재배치 과정에서의 오버헤드는 엄청나게 커진다. 최근에 저장장치의 추가로 인한 시스템의 정지를 막기 위해 디스크 온라인 탈/부착에 관한 많은 연구가 되었으나, 여전히 기존 데이터에 대한 재구성은 필수적인 작업으로 남아 있다.

또 다른 매핑방법으로는 논리주소와 물리주소를 매핑 할 수 있는 테이블을 사용하는 것이 있는데, 이 방법은 디스크 수의 변화나 스냅샷(snapshot), 오류 블록에 대한 저장위치 수정 등 다양한 유연성을 제공하지만, 매핑을 위한 테이블 공간이 많이 필요하다는 단점을 가진다. 매핑 테이블의 크기가 크다는 것은 이를 메인 메모리에 둘 수 없어서 매 디스크 I/O 때마다 매핑 테이블의 접근을 위해 추가적인 디스크 I/O가 필요하다는 것을 의미한다.

이 논문에서는 스트라이핑을 하는 시스템에서 논리블록과 물리블록간의 매핑을 수식과 매핑 테이블을 사용하는 방법을 조합하여, SZIT라고 하는 매핑 정보와 간단한 수식을 통하여 디스크의 수 변화에 유연하게 적응하면서, 관리해야 하는 데이터 수를 줄이는 SZIT 기반 매핑 방법을 제안한다. SZIT 기반 매핑 방법은 수식으로 매핑 하는 방식과 같이 간단한 수식을 사용하지만 디스크 수의 변화에 유연하며, 매핑 테이블으로 매핑 하는 방법보다 유지하는 매핑 정보의 양이 현저히 적은 장점을 가진다. 매핑 테이블을 이용하는 방법이 한 블록을 쓰는데 최소 5번의 디스크 I/O가 발생하는데 반하여 이 논문에서 제안하는 SZIT 기반 매핑 방법은 단 1번의 디스크 I/O가 발생하는 성능상의 향상을 보인다. 또한 읽기 연산의 경우 매핑 테이블의 경우 최대 2번의 디스크 I/O가 발생하지만 SZIT 기반 매핑 방법은 단 1번의 디스크

I/O만으로 충분하다.

이 논문의 구성은 다음과 같다. 2장에서는 이 논문에서 다루는 RAID, 스트라이핑, 스트라이핑에서의 디스크 추가 방법 등에 관한 관련 연구를 살펴보고, 3장에서는 이 논문에서 제안하는 SZIT 기반 매핑 방법에서 테이블 구성방법과 디스크에 데이터를 읽거나 쓸 때 논리블록을 물리블록으로 매 핑하는 방법에 관하여 설명한다. 4장에서는 제안하는 매핑방법과 기존의 수식, 테이블 기반 매핑방법과의 비교를 하고, 5장에서 논문의 결론과 향후 연구의 방향을 제시한다.

2. 관련연구

2.1 RAID

전통적인 디스크 관리 기법에서는 사용자에게 최대의 가용시간, 최적의 성능, 고용량 등의 요구사항에 만족스러운 해법을 제시하지 못하였다. 이러한 문제점을 해결하고자 하드웨어 RAID 시스템이 개발되었다. 하드웨어 RAID 시스템은 여러 개의 독립적인 디스크 장치들을 모아서 RAID 컨트롤러의 제어 하에 가용성, 성능, 용량적인 측면에서 사용자의 요구에 부응하도록 다양한 RAID 레벨을 제공한다. RAID는 저장 장치에서의 데이터 성능향상, 가용성에 따라 저장방법이 달라지는데, 스트라이핑 (striping), 미러링(mirroring), 패리티를 갖는 스트라이핑(striping with parity) 등으로 구분된다. 하드웨어 RAID 시스템은 RAID의 관리를 RAID 컨트롤러가 수행하기 때문에 CPU의 활용도를 높일 뿐 아니라 버스 트래픽을 감소시킴으로써 높은 성능을 제공한다[5].

그러나, 하드웨어 RAID 시스템의 가격이 매우 비싸며, 컨트롤러에 오류가 발생할 경우, 모든 RAID 시스템을 사용할 수 없게 되는 문제점을 가진다. 이러한 하드웨어 RAID의 기능을 소프트웨어적으로 구현한 것이 소프트웨어 RAID이다. 소프트웨어 RAID는 하드웨어 컨트롤러 대신 CPU가 RAID의 관리를 수행하게 된다. 이것은 컨트롤러의 오류 발생 시 모든 것이 정지되는 문제를 해결하며, 상대적으로 저렴한 가격의 RAID 시스템을 구성할 수 있게 해준다.

소프트웨어 RAID를 구현한 시스템으로는 Swift, Zebra, xFS 등이 있는데, Swift[7]는 여러 파일서버들에 파일들을 스트라이핑 하여 관리하므로 입출력 성능을 향상시키는 분산 파일 시스템으로, 클라이언트들은 각 파일 입출력시 여러 서버들로부터 동시에 서비스를 받으므로 입출력 성능의 향상을 가져오게 된다. 또한 Swift는 패리티 정보를 추가적으로 저장하므로 신뢰성이 높은 파일 서비스를 제공하게 된다.

xFS[8]는 버클리 대학에서 수행한 시스템 개발 프로젝트들을 근간으로 만들어진 시스템이다. 버클리 대학에서는 RAID의 한계점 중의 하나인 소규모 쓰기 문제를 해결하고자 LFS(Log-Structed File System)[9]를 개발하였다. LFS에서는 모든 쓰기 연산을 먼저 주기억장치에 버퍼링한 후, 연속된 고정 크기의 로그 세그먼트로 실제 쓰기를 수행하는 방법이다. 또한 전형적인 서버 캐시 방법을 자신의 로컬 캐시가 오버플로우(overflow) 났을 때, 현재 쓰고 있지 않는 클라이언트의 메모리를 활용하는 협력 캐싱(Cooperative caching)[10] 기법으로 교체하였으며, 메타데이터와 데이터를 분할하여 분산 배치시키기 때문에 이를 활용하기 위해서 관리자 맵, Imap, 파일 디렉토리 맵, 스트라입 맵 등의 매핑 정보를 사용한다.

Zebra[11]는 LFS 개념과 RAID 개념을 결하여 분산 시스템 환경에서 동작하도록 만든 네트워크 파일 시스템이다. RAID에 있어서의 작은 쓰기 동작에 대한 비효율적인 단점을 LFS 개념을 적용시킴으로써 쓰기 동작을 효율적으로 해결하였으며, 하드웨어적인 RAID 시스템은 새로운 하드웨어를 추가시켜야 하기 때문에 기존의 디스크 시스템에 비해 상당히 비용이 많이 드는 문제점이 있었는데, Zebra는 분산 시스템 환경에서 기존의 하드웨어는 그대로 두고 소프트웨어적으로 RAID개념을 적용시킴으로써 새로운 하드웨어에 대한 추가적인 부담을 덜게 되었다.

2.2 스트라이핑

스트라이핑은 데이터를 여러 디스크에 분산시켜 저장하는 방법으로 병렬 읽기 연산의 성능이 좋아진다. 스트라이핑을 할 때, 시스템의 성능을 좌우하는 요소 중의 하나가 스트라이핑 단위크기를 얼마로 하는가 하는 문제인데, [6]에서는 입출력 작업부하의 종류(읽기, 쓰기)와 동시에 발생하는 평균 요구 수에 따라 스트라이핑 단위크기를 결정하는 부분에서 스트라이핑 단위크기를 작게 하여 가능한 RAID를 구

성하는 모든 디스크에 분산되도록 저장하여 작은 쓰기 동작의 감소를 가져오는 것이 효율적이라고 주장하였다.

스트라이핑을 할 때, 시스템의 성능을 좌우하는 또 다른 요소가 패리티의 배치방법이다. [12]는 RAID에서의 패리티 배치 종류에 따른 성능을 비교 분석하였는데, 부하가 많은 경우 읽기, 쓰기 연산에 대해서는 RAID 4의 패치가 가장 나쁜 성능을 보이며, 그 외의 배치 방법들 간에는 큰 차이를 보이지 않았다. 부하가 낮은 경우에 대한 읽기, 쓰기 연산에 있어서는 Left-Symmetric, Extended-Left-Symmetric, Flat-Left-Symmetric 방법들이 RAID 4, Right-Asymmetric, Left-Asymmetric, Right-Symmetric 방법들에 비해 좋은 성능을 보여주었다.

2.3 스트라이핑을 하는 시스템에서 디스크 추가 방법

스트라이핑 방법으로 데이터를 저장하는 시스템에서 새로운 디스크를 추가하여 용량을 늘리고자 할 때, 다음과 같은 방법을 보편적으로 사용한다.

열 추가(add column) 방법은 기존의 시스템을 구성하는 디스크의 수를 고려하지 않고 디스크를 추가하기 때문에 이미 시스템에 들어있는 데이터들을 새로 재구성해야 한다. 이것은 온라인 상에서 처리하기에는 상당히 많은 시간이 소요하게 되며 이것으로 인해 사용자의 일반적인 데이터 접근에 많은 영향을 미치게 된다.

행 추가(add row) 방법은 이미 구성하고 있는 디스크 수를 고려하여 그 배수만큼의 디스크를 논리적으로 연결하는 방법이다. 이 방법은 처음 시스템을 구성하는 디스크 수 n만큼만 데이터를 분산시킨다. 즉, 1TB 디스크 3개가 있고 3개의 디스크를 추가하는 것은 2TB 디스크 세 개로 스트라이핑을 하는 것과 동일한 효과를 가지게 된다.

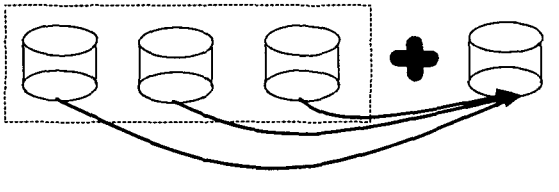


그림 1. 열 추가(add column)

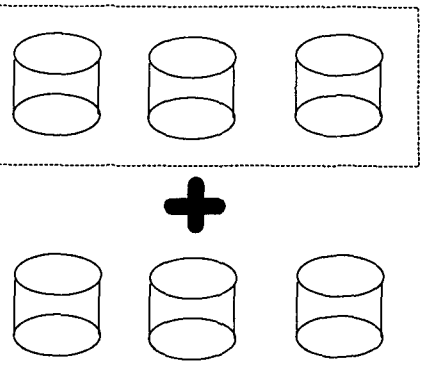


그림 2. 행 추가(add row)

열 추가의 경우 다음과 같은 데이터 재구성이 필요하며, 행 추가 방법은 현재 유지되는 디스크 수를 고려하여 그의 배수만큼의 디스크를 추가해야 하는 제약이 발생한다. 그림 3은 열 추가로 디스크를 추가한 경우의 데이터 재배치를 보여주고 있다.

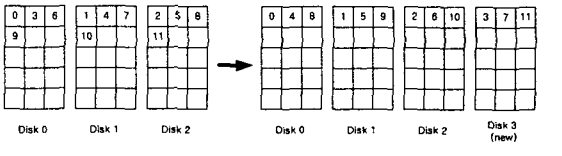


그림 3. 행 추가에서의 재배치

2.4 리눅스에서 용량이 다른 디스크를 위한 스트라이핑

리눅스에서는 서로 다른 용량을 가지는 디스크간의 스트라이핑을 제공하기 위해 스트라이핑 지역(*striping zone*) 이라는 개념을 사용한다[13]. 보기를 들어, 100 GB(D0), 150 GB (D1), 200 GB (D2)의 디스크가 있을 때, 스트라이핑 지역은 다음과 같이 나누어진다.

이 경우에 zone 0 에서는 3개의 디스크를 고려하여 스트라이핑을 하기 때문에 병행 읽기 성능을 최대로 할 수 있지만, 최악의 경우 zone 2의 경우 스트라이핑의 효과를 전혀 가질 수 없다.

표 1. 리눅스에서의 지역 정보 계산 방법의 보기

zone 0 : (D0/D1/D2) 3 ×100 GB = 300 GB
zone 1 : (D1/D2) 2 × 50 GB = 100 GB
zone 2 : (D2) 1 × 50 GB = 50 GB

2.5 논리주소와 물리주소의 매핑 방법

스트라이핑을 할 때, 논리주소와 물리주소를 매핑하는 방법은 크게 두 가지로 살펴볼 수 있다. 첫 번째 방법은 일반적으로 사용하는 수식을 사용하는 방법이다. 즉, 논리주소를 디스크 수로 모듈러 연산하여 대상 디스크를 선택하고, 디스크 내의 위치는 논리주소를 디스크 수로 나누어 얻을 수 있다.

그림 4에서 디스크의 수가 3개 일 때, 논리 블록 10의 저장 위치는 다음과 같이 구할 수 있다.

즉, 1번째 디스크의 3번째 블록에 저장된다.

이 방법은 간단한 수식을 통해서 주소가 계산되기에 장점을 가지지만 저장 디스크의 수가 달라지면 매핑 하는 방법이 달라지기 때문에 디스크가 추가될 때마다 새로운 식을 적용시켜 기존의 데이터 위치를 재조정해야 한다. 이러한 재구성 과정은 비용이 많이 드는 연산이다.

이러한 디스크 수의 변화에 유연하게 적응하기 위해 표 3과 같이 논리주소와 물리주소를 테이블의 형태로 매핑 할 수 있다[14].

하지만 이 방법은 논리블록의 수만큼의 테이블을 유지해야 하기 때문에 많은 저장공간이 필요하다. 세

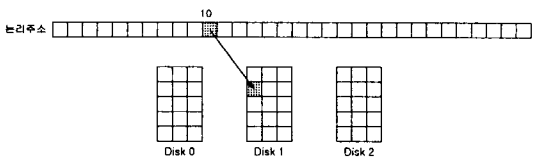


그림 4. 논리주소에서 물리주소로의 매핑

표 2. 논리블록 10에 대한 물리블록 계산 과정

저장될 디스크 위치 : 10 % 3 = 1
디스크 내에서의 위치 : 10 ÷ 3 = 3

표 3. 매핑 테이블의 구조

논리주소	디스크 번호	물리주소
0K	1	0K
1K	2	0K
2K	3	0K
3K	1	1K
4K	2	1K
5K	3	1K
:	:	:

개의 디스크가 각각 40GB이고, 블록 크기가 1K일 경우, 이러한 테이블을 유지하기 위해서는 (40GB/1K) × 한 튜플에 저장되는 데이터 크기 = 40MB × 한 튜플에 저장되는 데이터 크기가 된다.

매핑 테이블을 사용하여 물리 블록을 할당하는 것은 표 4와 같은 순서를 따른다. 즉, 쓰기 연산의 경우, 먼저 매핑 테이블을 읽어서 이미 매핑이 되어 있는지 확인하고 매핑이 되어 있지 않다면 자유공간 관리자를 호출하여 새로운 블록을 할당받는다. 자유공간관리자는 자유공간관리를 위해 별도의 메타정보를 관리하는데 이 메타데이터의 크기도 매우 크고, 시스템의 전원이 나갔다 들어오는 경우에도 일관성을 가져야 하기 때문에 디스크에 저장되어야 한다. 따라서 새로운 블록을 할당하기 위해서는 자유공간관리 정보를 읽고, 할당할 블록을 찾아 할당 표시를 한 후에 디스크에 반영해야 하기 때문에, 최소 2번의 디스크 I/O가 발생한다. 최악의 경우는 자유공간관리 정보의 블록 수만큼 읽어야만 할당할 블록을 찾을 수 있기 때문에 성능이 떨어질 수 있다. 자유공간관리자로부터 할당된 블록 정보를 전달 받으면 매핑 테이블에 할당된 블록정보를 기록하고 이를 디스크에 반영한다. 이 모든 과정이 끝난 후에 실제 데이터가 디스크에 반영된다. 따라서 새로 할당하는 블록에 대해서는 최소 5번의 디스크 I/O(실제 데이터 I/O + 매핑 테이블 읽기/쓰기, 자유공간관리 정보 읽기/쓰기)가 발생한다. 시스템의 성능에 가장 큰 영향을 미치는 것 중의 하나가 디스크 I/O 수이기 때문에 이에 대한 성능 개선 방법이 제안되고 있다.

하지만, 매핑 테이블을 사용하는 방법은 수식을 사용하는 방법에 비해서 스냅샷(snapshot)을 관리하는 방법에서 간단히 제공될 수 있다. 즉, 스냅샷이 생성된 시점에서 매핑 테이블을 복사하여, 변경되는 블록에 대해서만 실제 매핑 테이블에 반영하고 과거 데이터는 스냅샷 매핑 테이블에 유지하여 실제 매핑 테이블은 최근에 변경된 데이터 주소를 가리키고, 스냅샷 매핑 테이블은 이전 데이터 주소를 가리키면 되기에 간단히 구현될 수 있다. 수식을 사용하는 방법에서 스냅샷을 제공하기 위해서는 매핑 테이블을 사용하는 방법보다 고려해야 할 사항들이 많다.

3. SZIT 기반 매핑 방법

3장에서는 이 논문에서 제안하는 SZIT 기반 매핑

표 4. 매핑 테이블을 사용한 물리 블록 할당

```
mapping_table() {
  next_dev ← 0
  while(operation about log_addr) {
    if(write operation?) {
      read mapping block in mapping table about log_addr     // DISK OPERATION(Meta Data)
      if(is already mapped?) {
        write data block                                     // DISK OPERATION(Real Data)
      }
      else {
        alloc_addr ← alloc_freespace(next_dev)
        if(alloc_addr is DEVICE_FULL) {
          if(next_dev is max_dev_no)
            next_dev ← 0
          else
            next_dev++
        }
        else {
          write alloc_addr into mapping table about log_addr // DISK OPERATION(Meta Data)
          write data block into alloc_addr into real device  // DISK OPERATION(Real Data)
        }
      }
    }
    else {  // read operation
      read mapping block in mapping table about log_addr     // DISK OPERATION(Meta Data)
      read data block                                        // DISK OPERATION(Real Data)
    }
  }
}
alloc_freespace(next_dev) {
  read freespace block                                       // DISK OPERATION(Meta Data)
  addr ← find_freespace(next_dev)
  set addr as allocated
  write freespace block                                      // DISK OPERATION(Meta Data)
  return addr
}
```

방법과 이를 사용한 데이터 채우기 과정에 대하여 설명한다. 수식 기반 매핑방법은 매핑 속도가 빠른 반면에 디스크 수에 유연하게 대처하지 못하며, 매핑 테이블을 사용하는 경우에는 2.5에서 설명한 바와 같이 스냅샷 구성에는 유리하지만 성능이 많이 떨어지는 문제점을 가진다.

이 논문에서 제안하는 디바이스의 수에 유연하게 적응하는 수식기반 매핑 방법을 위해서는 추가 정보

표 5. 스트라이핑 지역정보 테이블 (SZIT)의 구조

지역 번호	전체 디바이스 수	참여 디바이스 수	첫 물리블록	끝 물리블록	첫 논리블록	끝 논리블록

가 필요한데, 이러한 추가 정보가 담겨진 테이블을 SZIT (Strip Zone Information Table) 라고 하고, 추가된 디바이스에만 쓰기 연산을 수행하는 것을 데이터 채우기 (data padding) 라고 한다.

3.1 스트라이핑 지역 정보 테이블 (SZIT : Strip Zone Information Table)

이 논문에서 제안하는 SZIT 기반 매핑 방법은 디바이스가 추가되는 시점에 스트라이핑 지역을 나누어 각 스트라이핑 지역에 맞는 스트라이핑을 수행한다. 즉, 디바이스를 추가한 후부터 디바이스로의 쓰기 연산은 모두 새 디바이스에서만 수행하여, 추가한 디바이스의 사용량이 기존의 디바이스의 사용량과 같을 때까지 추가된 디바이스에서만 스트라이핑을 수행한다. 이 과정을 데이터 채우기라고 한다. 데이터 채우기가 끝나면 모든 디바이스를 대상으로 스트라이핑을 수행한다. 이 논문에서 제안하는 방식을 통해 디바이스 입출력을 할 경우에 논리주소와 물리주소의 정확한 매핑관계를 알 수 있으려면 다음과 같은 추가의 정보가 필요하다.

지역번호는 스트라이핑 지역을 나타내는 번호로 첫 논리블록에 따라 순차적으로 증가한다. 전체 디바이스 수는 해당 스트라이핑 지역에서의 전체 디바이스를 나타내고, 참여 디바이스는 해당 스트라이핑 지역에 참가하는 디바이스의 수를 나타낸다. 첫 논리블록과 끝 논리블록은 해당 스트라이핑 지역의 첫 논리블록과 끝 논리블록을 나타내며, 이때의 디바이스 상의 물리적인 블록번호가 첫 논리블록의 물리블록과 끝 논리블록의 물리블록이 된다.

3.2 SZIT의 갱신

SZIT는 디바이스가 추가되는 시점마다 갱신되며, 이 정보를 통해서 논리주소를 물리주소로 변환하게 된다. 디바이스 추가가 되었을 때 SZIT를 갱신하는 방법은 다음과 같다.

최근에 개발되고 있는 많은 제품들에서 디바이스의 탈부착 상태를 온라인 상으로 검출할 수 있는 방법을 제공하고 있기 때문에, 이러한 이벤트가 발생하면 먼저 마지막으로 쓰기 연산을 수행한 논리블록 번호가 포함된 지역정보 (현재 스트라이핑 지역정보, current strip zone information) 를 수정한다. 끝 논리블록 및 물리블록은 마지막 쓰기 연산이 포함된 스트라이핑 라운드가 끝나는 블록으로 결정한다.

즉, 그림 5와 같이 3개의 디바이스에 대해서 논리블록 9까지 쓰기 연산이 수행된 후에 디바이스 추가 이벤트가 검출되었을 경우, 논리블록 9가 포함된 스트라이핑 지역 정보 0의 끝 논리주소는 스트라이핑 라운드가 끝나는 11이 된다. 논리블록 10, 11까지 쓰기 연산이 수행된 경우에도 동일하게 끝 논리주소는 11이 된다.

추가 스트라이핑 정보는 대부분 2개의 스트라이핑 지역이 추가 되는데, 하나는 새로 추가된 디스크에 데이터를 채우는 영역에 관한 정보이고, 다른 하나는 기존의 디스크와 추가된 디스크로 만들어지는

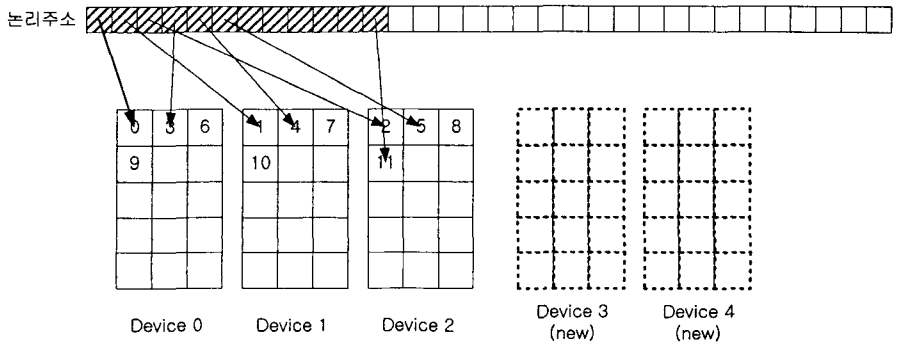


그림 5. 저장매체 추가 이전의 데이터 저장

영역에 대한 정보가 된다.

이러한 과정이 끝난 후에는 남은 영역에 대한 정보 수정이 필요하다. 하지만 이 과정은 디스크 추가가 1번 이상 발생한 경우에만 수행된다.

표 7에서는 디바이스 3개로 운영중인 시스템에서 논리블록 9~11을 쓰는 도중에 디바이스 추가가 발생한 경우의 SZIT의 모습이다. 한 디바이스에는 블록이 15개씩 있다고 가정하였다.

3.3 데이터 채우기

새로운 디바이스의 추가로 SZIT가 갱신된 이후에 발생하는 데이터는 추가된 디바이스에만 저장된다. 그림 6은 이러한 과정을 보여주며, 이과 같이 추가한 디바이스에만 데이터를 쓰는 과정을 데이터 채우기(data padding)라 한다.

그림 6은 논리블록 12번부터 19번까지 쓰여지는 과정을 보여준다. 이때 스트라이핑 지역 1에 대하여 스트라이핑에 참여하는 디바이스의 수는 추가한 디바이스인 2이다.

데이터 채우기는 이전 디바이스에 저장된 데이터의 양만큼 추가된 디바이스에 데이터가 쌓인 만큼인 논리블록 19까지만 수행되며, 그 이후로는 전체 디바이스를 대상으로 데이터를 저장한다.

그림 7은 스트라이핑 지역 1에 대한 쓰기 연산이 모두 끝난 후 논리블록 20번부터 스트라이핑 지역 2에 쓰기 연산이 수행되는 그림을 보여주며, 이때 스트라이핑에 참여하는 디바이스의 수가 전체 디바이스 수와 동일한 5개임을 보여준다.

3.4 SZIT 기반 매핑 방법

SZIT 기반 매핑 방법에서 저장장치에 저장되어 있는 데이터에 접근하기 위해서는 항상 SZIT를 참조하여 접근하려는 논리블록을 포함하는 스트라이핑 지역을 찾고, 이를 통하여 물리블록을 계산한다.

즉, 요구하는 논리주소가 SZIT에 나타나는 추가로 인해 채워진 주소 영역(12~20)인 경우가 아니면 그 범위를 고려하여 12 이전이면 원래 디바이스의 수인 3개를 고려하여 물리주소를 구하고, 20 이후라면 추가된 후의 디바이스 수인 5개를 고려하여 물리주소를 구한다. 채워진 주소 영역에 대한 디바이스 연산이 발생할 경우에는 SZIT의 메타 데이터를 참조하여 물리주소를 구하게 된다.

표 6. SZIT 갱신 과정

```
if(디바이스 추가)
{
  현재 스트라이핑 지역정보의 끝 논리블록 및
  끝 물리블록 수정
  추가 스트라이핑 지역정보에 대한 정보 추가
  추가된 지역 이후의 지역에 대한 정보 수정
}
```

표 7. 저장매체 추가 이후의 SZIT

지역번호	전체 디바이스 수	참여 디바이스 수	첫 물리블록	끝 물리블록	첫 논리블록	끝 논리블록
0	3	3	0	3	0	11
1	5	2	0	3	12	19
2	5	5	4	14	20	74

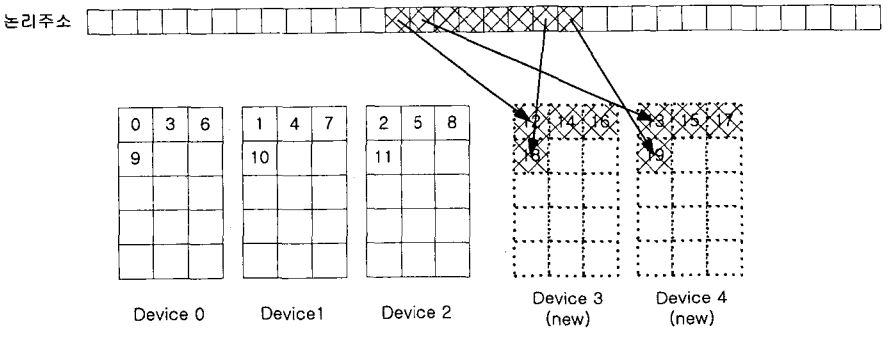


그림 6. 데이터 채우기 과정

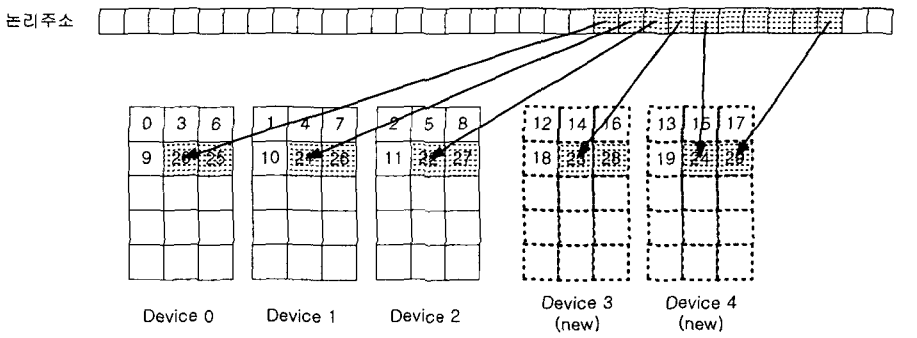


그림 7. 데이터 채우기가 끝난 이후의 전체 디바이스를 대상으로 스트라이핑

SZIT 기반 매핑은 다음과 같은 수식을 통해 논리 블럭을 물리블럭으로 변환한다.

표 8. SZIT 기반 매핑 계산식

targetDevice = ((logBlkNo) % parDevice) + (totalDevice - parDevice) targetPhyBlk = (logBlkNo - logStartNo) / parDevice + phyStartNo	
targetDevice	논리블록 n을 저장하는 디바이스
targetPhyBlk	논리블록 n에 매핑 되는 디바이스 내의 물리논리 블록 번호
logBlkNo	논리블록 n의 주소
phyStartNo	logBlkNo의 디바이스 상의 물리블록 번호
parDevice	참여하고 있는 디바이스의 수
totalDevice	전체 디바이스의 수

3.6 SZIT의 저장

SZIT를 메모리에서만 관리하게 되면 시스템의 전원이 나간 후 다시 시작하면 이러한 정보를 잃게 되므로, 이 정보는 디바이스에 직접 저장할 필요가 있다. 시스템의 재동작시에는 디바이스의 첫 부분에 저장된 스트라이핑 지역 정보를 읽어서 전체적인 시스템의 스트라이핑 지역 정보를 만들게 된다. SZIT의 내용은 시스템에 새로운 디스크가 추가될 때에만 변경되는데, 이러한 빈도는 시스템 운영중에 수차례 밖에 발생하지 않는 것이 일반적이다. 디스크가 추가되면 이러한 변화를 메모리 상의 SZIT 정보를 먼저 수정한 후, 각 디바이스의 레이블 정보를 저장하는 지역에 정보를 중복해서 기록한다.

4. 매핑 방법 비교

이 장에서는 다른 매핑방법과 SZIT 기반 매핑방법을 비교한다.

4.1 매핑 테이블 방식과 SZIT 매핑 방법의 디스크 추가 비교

매핑 테이블을 사용하는 스트라이핑 저장 방식에서 디스크가 추가 되면 재구성을 할 수도 있지만, 재구성을 하지 않고도 시스템 운영에는 아무 문제가 없다. SZIT 매핑 방법 또한 원리는 같지만 데이터를 저장하는 순서에 차이가 있다. 이를 그림으로 보면 다음과 같다.

즉, 테이블 기반 매핑은 그림 8과 같이 zone 0까지 데이터를 저장한 상태에서 디스크가 추가되면 zone 1에 데이터를 저장하게 되고 디스크 0, 1, 2의 용량 전체에 데이터가 저장되면 zone 2(디스크 3, 4)에만 저장하게 된다. SZIT 기반 매핑 방법은 그림 9와 같이 디스크가 추가된 시점부터 디스크 3, 4로 구성된 zone 1에 데이터를 저장하고 이 지역에 데이터가 모두 차게 되면 전체 디스크를 대상으로(zone 2) 데이터를 저장하게 된다.

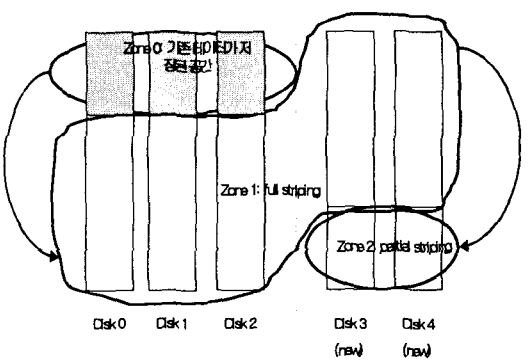


그림 8. 테이블 기반 매핑에서 디스크 추가에 따른 데이터 저장 위치 이동 과정

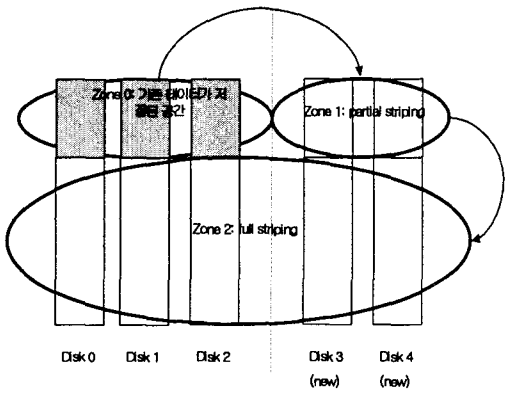


그림 9. SZIT 기반 매핑에서 디스크 추가에 따른 데이터 저장 위치 이동 과정

4.2 매핑 방법들간의 성능 비교

수식기반 매핑 방법은 논리블록을 물리블록으로 변환하는데 있어서 수식을 사용하기 때문에 따로 저장해야할 매핑 데이터가 전혀 없다. 이에 반해서 매핑 테이블에 의한 방법은 논리블록을 물리블록으로 변환하기 위한 테이블이 존재하는데, 이 테이블의 크기는(논리블록의 수 × 테이블에서의 한 튜플의 크기)가 된다. 디바이스의 크기가 클수록 이 수는 증가한다. 이 논문에서 제안하는 SZIT를 통한 매핑에서도 매핑 테이블처럼 추가의 매핑 정보를 관리하는데, 이 정보의 크기는(디바이스 추가 회수 × 테이블에서의 한 튜플의 크기)가 되므로 매핑 테이블에서 관리하는 매핑 데이터에 비해서 현저히 적은 데이터를 관리하게 된다.

수식기반 매핑 방법은 디바이스 수의 변화에 유연하지 못하기 때문에 디바이스를 추가할 경우에는 반드시 데이터 재배치를 하여야 하지만, 테이블 기반 매핑, SZIT 기반 매핑 방법은 그렇지 않다. 이 두가지 매핑 방법에서는 재배치를 하지 않을 수도 있지만, 재배치 연산을 지연시켜 수행할 수 있다.

수식기반 매핑과 테이블 기반 매핑은 항상 모든 디바이스를 대상으로 스트라이핑을 수행한다. 하지만 SZIT 기반 매핑과 테이블 기반 매핑에서는 일부 영역에 대해서 부분 스트라이핑을 해야 한다. 디바이스를 추가할 때 여러 개의 디바이스가 추가된다면 스트라이핑의 효과를 충분히 볼 수 있겠지만, 1개의 디바이스만을 추가할 경우에는 스트라이핑 효과를 전혀 볼 수 없는 영역이 존재한다.

그림 10는 매핑 각 매핑 방법들의 쓰기 연산에 대한 매핑 연산 속도를 비교한 그래프이다. 초기 시스템은 디스크 10개로 구성되어 있고, 100만 블록을 쓸 때마다 디스크를 5개씩 추가되는 것을 가정하였다. 가로축은 데이터 블록 I/O 수를 나타내며, 세로축은 실제 디스크 I/O의 수를 나타낸다. 테이블 기반 매핑 방법의 경우, 1,000만 블록의 데이터를 쓰기 위해서는 매핑 테이블, 자유공간관리 정보를 위한 디스크 접근으로 인해 최소 5,000만 블록의 디스크 I/O가 발생한다. 그래프에서 수식기반 매핑 방법과 SZIT 기반 매핑 방법의 디스크 I/O 수에 대한 차이는 거의 없어 보이지만, 실제는 디스크 추가 시 마다 디스크 수만큼의 추가 디스크 I/O가 발생한다. 그림 11은 읽기 연산에 대한 매핑 방법들의 성능 비교이다. 쓰기 연산에 비해 읽기 연산은 매핑 테이블을 한번 읽어오면 실제 데이터 주소를 알 수 있기 때문에 실제 데이터를 위한 디스크 I/O 외에 추가적으로 1번의 디스크 I/O가 더 필요하다.

4.3 디바이스 수의 변화에 따른 매핑 정보의 수 비교

매핑을 블록단위로 하고 블록의 크기를 1KB라고

표 9. 매핑 방법의 비교

		수식 기반 매핑	테이블 기반 매핑	SZIT 기반 매핑
capacity of meta data		do not need	9 Byte × 블록 수	19 Byte × Zone 수
data reorganization		must do	optional	optional
NO of disk I/O	metadata	nothing	at lest 4(write operation)	nothing
	read data	1	1	1
	additional disk I/O when disks are added	nothing	nothing	1 per total disks
striping disks		total disks	total disks or partial disks	total disks or partial disks

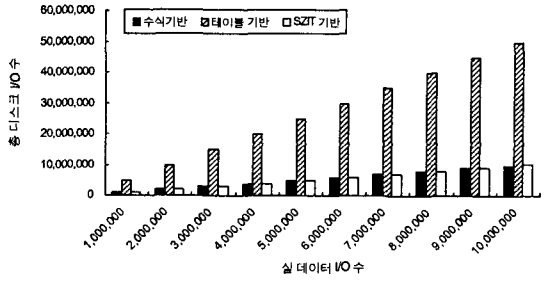


그림 10. 쓰기 연산에 따른 각 매핑 방법들의 성능비교

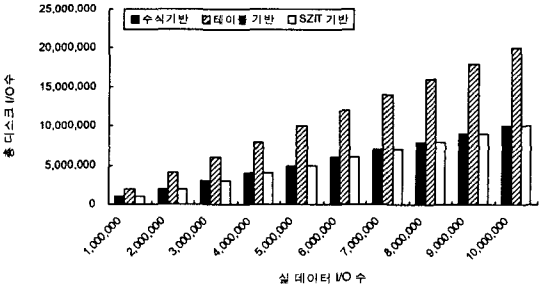


그림 11. 읽기 연산에 따른 각 매핑 방법들의 성능비교

할 때, 시스템에서 저장해야 할 매핑정보의 크기는 표 11과 같다. 여기에서 매핑 테이블에서의 한 튜플 정보의 크기는 논리블록주소(4 Byte), 디바이스 번호(1 Byte), 물리블록번호(4 Byte) 이며, 모든 디바이스의 크기는 20 GB이며, 디바이스 추가는 2개씩 이루어진다고 가정한다. 또한 데이터 채우기 방법에서 SZIT 한 튜플의 크기는 스트라이핑 지역번호(1 Byte), 전체 디바이스 수(1 Byte), 참여 디바이스 수(1 Byte), 첫 물리블록번호(4 Byte), 끝 물리블록번호(4 Byte), 첫 논리블록번호(4 Byte), 끝 논리블록번호(4 Byte)라고 가정한다. 따라서 매핑 정보 테이블의 한 튜플의 크기는 9 Byte가 되며, SZIT의 한 튜플의 크기는 19 Byte가 된다.

수식기반 매핑방법은 어떤 경우에도 추가로 유지하는 매핑 정보가 없다. 하지만 매핑 테이블을 사용하는 경우에서는 용량이 커질수록 유지하는 매핑 정보가 증가한다. 초기 상태에서 20 GB 디바이스 3개로 구성되어 있는 시스템에서 전체 용량 60 GB의 1.14%인 540 MB의 공간이 매핑 테이블을 위해 할당되어야 한다. 140 Giga Byte 정도의 용량을 가지는 시스템에서 1.2 Giga byte의 메타데이터를 메인 메모리상에 유지하는 것은 힘들기 때문에 디스크 I/O 횟수가 많아지고 이에 따라 성능이 떨어지게 된다. 이에 반해 SZIT 기반 매핑 방법은 그보다 훨씬 작은 19 Byte의 공간만 필요하며, 디바이스의 추가되는 시점에 따라 스트라이핑 지역 정보가 조금씩 달라질 수 있지만, 데이터 채우기가 끝나고 전체 디바이스를 대상으로 스트라이핑을 하고 있을 때 디바이스가 추가된다면, 첫 번째 추가 시에는 57 Byte, 두 번째 추가 시에는 95 Byte의 정보만을 유지하면 된다.

5. 결론 및 향후 연구

이 논문에서는 스트라이핑을 하는 RAID 시스템에서 디바이스를 추가할 때, 기존 데이터에 대한 재구성 작업 없이 스트라이핑의 효과를 가질 수 있는 방법을 제안하였다. 그 방법은 디바이스가 추가될 때마다 달라지는 스트라이핑 대상 디바이스의 수에 대한 정보를 테이블로 유지하여 디바이스 입/출력 요청시 간단한 수식과 테이블 정보를 이용하여 매핑을 수행한다. 이 테이블을 SZIT (Strip Zone Information Table) 라 하고, 이 테이블을 통하여 읽기, 쓰기를 할 실제 디바이스와 디바이스 내의 위치를 계산한다. 만일, 디바이스가 새로 추가된 후에 발생하는 쓰기 연산은 새로 추가된 디바이스만을 통해 스트라이핑으로 데이터를 저장하고, 기존 디바이스의 용량만큼 데이터가 채워지면 그때부터 전체 디바이스를 대상으로 스트라이핑을 수행하여 데이터를 저장한다. 이 과정을 데이터 채우기 (Data Padding) 라고 한다. 이 방법은 보편적인 매핑 테이블 방법이 과도한 메타데이터 양 때문에 성능에 심각한 문제를 일으키는 점을 보안하면서 계산식에 의한 빠른 매핑

표 10. 디바이스 수의 변화에 따른 매핑 정보의 수

	초기 디바이스	첫 번째 추가	두 번째 추가
수식기반 매핑	0	0	0
테이블 기반 매핑	540 Mega Byte	900 Mega Byte	1260 Mega Byte
SZIT 기반 매핑	19 Byte	57 Byte	95 Byte
전체 디바이스 용량	60 Giga Byte	100 Giga Byte	140 Giga Byte

을 가능케 한다. 또한 제안하는 방법에서 추가 시점 이후에 재구성을 할 수도 있는데, 대상이 되는 데이터의 수는 기존의 경우에 기존의 방법으로 스트라이핑을 할 때 발생하는 재구성 대상 데이터 수와 비슷하기 때문에 재구성 연산을 지연시키는 효과도 가질 수 있다.

즉, SZIT 기반 매핑 방법은 수식 기반 매핑 방법처럼 매핑 과정 중에 추가의 디스크 I/O가 발생하지 않기 때문에 빠른 연산속도를 낼 수 있으며, 수식 기반 매핑 방법에서 치명적인 단점으로 지적되는 디바이스 수의 변화에 유연하게 대처하지 못하는 부분을 SZIT 정보를 이용하여 극복하고 있다. SZIT의 정보는 매핑 테이블에서 매핑을 위해 관리해야 하는 데이터의 양보다 현저하게 작기 때문에 항상 메모리에 올려두고 사용할 수 있다.

SZIT 기반 매핑 방법은 스트라이핑 시스템에서 저장용량을 증가시키기 위해 디스크를 추가한 경우, 시스템을 정지시킬 필요 없이 계속 서비스를 제공할 수 있는 장점을 가진다. 따라서 SZIT 기반 매핑방법은 실시간 서비스를 해야 하는 시스템에서 온라인 디스크 확장을 가능하게 한다.

이 논문에서 제안하는 방식은 현재 모든 디바이스의 용량, 블록의 크기가 모두 동일하다는 가정을 하고 있다. 따라서 다양한 크기의 용량을 지원할 수 있는 방안과 스냅샷 지원을 위한 방법, 단순한 스트라이핑이 아닌 RAID 5와 같이 패러티 (parity) 를 계산하는 경우에 대한 처리에 대한 연구가 필요하다. 또한 메모리상의 SZIT 변경 사항을 각 디스크로 반영할 때 시스템 장애가 발생한 경우, 각 디스크에 저장된 SZIT 간의 불일치가 발생할 수도 있는데, 이를 해결하기 위한 연구도 필요하다.

참 고 문 헌

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

[2] Randy H. Katz, "*High-Performance Network and Channel Based Storage*", Proceedings of IEEE", Vol.80, No.8, 1992.

[3] Matthew T. OKeefe, "*Standard file systems and fibre channel*", In The sixth Goddard Conference on Mass Storage System and Technologies in cooperation with the Fifteen IEEE Symposium on Mass Storage Systems", Colleage Park, Maryland, March 1998.

[4] Alan F. Benner, "*Fibre Channel : Gigabit Communications and I/O for Computer Network*", McGraw-Hill, 1996.

[5] D.A. Patterson, J.L.Hennesy, R.H.Katz, "*A case for redundant arrays of inexpensive disk(RAID)*", International Conference of Management of Data(SIGMOD), pp.109-116, 1988.

[6] Perter M, Chen and Edward K. Lee, "*Striping in RAID Level 5 Desk Array*", In proceeding of the Joint International Conference on Measurement and Modeling of Computer Systems, 1995.

[7] Luis-Felipe Cabrera, Darrell D. E. Long, "*Swift : Using Distributed Disk Striping to Provide High I/O Data Rate*", Computing Systems, Fall 1991.

[8] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang, "*Serverless Network File System*", ACM Transactions on Computer Systems, February 1996.

[9] M. Rosenblum, J. K. Ousterhout, "*The Design and Implementation of a Log-Structed File System*", ACM Transactions on Coumputer of Systems, Vol. 1, Fabruary 1992, pp. 26-52.

[10] M.D.Dahlin, R.Y.Wang, T.E.Anderson, D.A.Patterson, "*Cooperative Caching : Using Remote Client Memory to Improved File System Performance*", 1st Symposium on Operating Systems Design and Implementation(OSDI), Novermber 1994, pp 267-280.

[11] John H. Hartman, John K, "*Zebra: A Striped Network File System*", In Proceedings of the USENIX File Systems Workshop, May 1992.

[12] Edward K, Lee and Randy H. Katz, "*Per-*

formance Consequences of Parity Placement in Disk Arrays", In Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System", April 1991.

[13] "*Software-RAID howto*", http://kldp.org/HOWTO/min/html/Software-RAID.html.

[14] Chang-Soo Kim, Gyoung-Bae Kim, Bum-Joo Shin, "*Volume Management in SAN Environment*", Procddeings Eighth International Conference on Parallel And Distributed Systems, Kyongju, Korea, 26-29 June 2001.

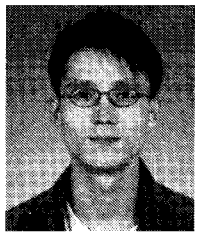
[15] 김종훈, 엄세웅, 노삼혁, 원유헌, "*스트라이핑 소프트웨어 RAID 파일 시스템의 성능에 미치는 영향*", 정보과학회논문지(A) 제 25권 제 5호, 1998. 5.

박 유 현

1996년 부산대학교 전자계산학과(학사)
1998년 부산대학교 전자계산학과(석사)
2000년 부산대학교 전자계산학과(박사과정 수료)
2000.1~2000.12년 한국국방연구원(KIDA) 자원관리연구부 연구원
2001.1~현재 ETRI 컴퓨터소프트웨어연구소 컴퓨터시스템연구부 연구원
관심분야 : 자료저장시스템, 분산시스템, 데이터베이스, 모바일 시스템
E-mail : bakyh@etri.re.kr

김 창 수

1993년 광운대학교 전자계산학과(학사)
1995년 서강대학교 전자계산학과(석사)
1995~1999년 LG 소프트(현재 LGEDS)
1999~현재 ETRI 컴퓨터.소프트웨어연구소 컴퓨터시스템연구부 선임연구원
관심분야 : 데이터베이스 시스템, 자료저장시스템, 클러스터 시스템, 분산 시스템
E-mail : cskim7@etri.re.kr

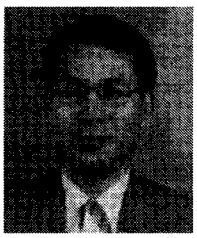


강 동 재

1999년 인하대학교 전자계산공학과(학사)
2001년 인하대학교 전자계산공학과(석사)
2001~현재 ETRI 컴퓨터·소프트웨어연구소 컴퓨터시스템연구부 연구원
관심분야 : 자료저장시스템, GIS, 데이터베이스, 시스템 프로그램
E-mail : djkang@etri.re.kr

김 영 호

1999년 충북대학교 정보통신공학과(학사)
2001년 충북대학교 정보통신공학과(석사)
2001~현재 ETRI 컴퓨터·소프트웨어연구소 컴퓨터시스템연구부 연구원
관심분야 : 자료저장시스템, 클러스터링 시스템, 내용기반 이미지 데이터베이스
E-mail : kyh05@etri.re.kr



신 범 주

1983년 경북대학교 전자공학과(학사)
1991년 경북대학교 컴퓨터공학과(석사)
1998년 경북대학교 컴퓨터공학과(박사)
1987~2002년 ETRI 컴퓨터.소프트웨어연구소 컴퓨터시스템연구부 책임 연구원, 시스템소프트웨어연구팀장
2002~현재 밀양대학교 컴퓨터.정보통신공학부 교수
관심분야 : 분산시스템, 고장감내 미들웨어, 이동컴퓨팅, 스토리지 클러스터 S/W
E-mail : bjshin@mnu.re.kr

교 신 저 자

박 유 현 (305-350) 대전광역시 유성구 가정동 161번지 한국전자통신연구원 컴퓨터·소프트웨어 연구소 컴퓨터시스템 연구부