소프트웨어 RAID에서 온라인 디스크 부착이 가능한 SZIT 기반 매핑 기법

박 유 현[†]·김 창 수[††]·강 동 재[†]·김 영 호[†]·신 범 주[†††]

요 약

인터넷의 발달로 사용자는 원격지에 있는 컴퓨터에 24시간 접속할 수 있게 되었고 이에 따라 무정지 시스템에 대한 요구가 커지고 있다. 또한 이를 위해서 디스크와 호스트를 온라인으로 탈부착하는 기술에 대한 연구도 진행되고 있다. 이 논문은 스트라이핑으로 데이터를 저장하는 시스템에 온라인 디스크 부착을 할 때, 소프트웨어적인 부가적 작업을 줄일 수 있는 방법을 제안한다. 제안하는 방법은 SZIT라 불리는 메타데이터와 계산식을 이용하여 매핑하기 때문에, 전통적인 수식 기반 매핑 방법과 같이 빠른 매핑을 할 수 있으며 디스크 확장으로 인한 전체 데이터를 재구성하는 오버헤드를 제거하는 장점을 가진다.

The SZIT based-mapping method for on-line adding disks in software RAID

Yuhyeon Bak[†]·Changsoo Kim[††]·Dongjae Kang[†]
Youngho Kim[†]·Bumjoo Shin[†††]

ABSTRACT

According as the users can connect to remote hosts at anytime by using internet, the demand of non-stop system is increased. Also, the research about on-line adding/deleting disks and hosts are progressed. This paper suggests that the method of reducing additional operation when we add disks to the stripping system. That method can fast mapping like equation based mapping method because of using so-called SZIT (Striping Zone Information Table) and equations. And that method can reduce the relocation overhead in striping system at adding disks.

키워드 : 스트라이핑 영역정보 테이블 (Strip Zone information Information Table : SZIT), RAID, 스트라이핑(Striping), 매핑(Mapping)

1. 서 론

인터넷 사용의 대중화는 작업 환경의 급속한 변화와 함께 저장되어야 할 데이터 양의 폭발적인 증가를 가져왔다. 특히, 멀티미디어 데이터의 영향으로 데이터 평균적인 크기는 기존의 컴퓨터에서 다루던 데이터의 크기보다 매우 커졌으며, 컴퓨터 사용자의 증가로 생성되는 데이터의 양 또한 증가하였다. 그러나, 하나의 서버에 접속하여 데이터를 저장하고 관리하는 클라이언트 서버 형태의 데이터 관리 시스템이나 파일서버를 기반으로 하는 네트워크 파일 시스템과 같은 기존의 자료저장 시스템들은 기하급수적으로 늘어 나는 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. 이와 같은 SAN의 장점은 현재의 자료저장 시스템이 안고 있는 대용량 저장장치 문제들을 효과적으로 해결할 수 있다는 것이다.

저장장치에서 데이터를 읽어올 때, 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 스트라이핑 단위 (stri-

† 정 회 원 : 한국전자통신연구원 컴퓨터소프트웨어연구소 컴퓨터시스템연구부 연구원
†† 정 회 원 : 한국전자통신연구원 컴퓨터소프트웨어연구소 컴퓨터시스템연구부 선임연구원
††† 정 회 원 : 밀양대학교 컴퓨터공학과 교수
논문접수 : 2002년 7월 6일, 심사완료 : 2002년 11월 9일

ping unit) [6]로 디스크에 순차적으로 써 나가는 방법으로 디스크로의 입/출력을 동시에 수행하기 때문에 병렬성이 높아진다.

스트라이핑 방법으로 데이터를 저장하는 시스템에서 논리주소를 물리주소로 매핑 하는 방법은, 디스크 수로 모듈러 연산을 수행하여 저장할 위치 디스크를 결정하는 것이 보편적이다. 이 방법은 추가적인 데이터의 도움 없이 간단히 매핑이 가능하다는 장점을 가지지만, 사용하고 있는 도중에 새로운 디스크를 추가하고자 할 때는 시스템을 정지시키고 물리적으로 디스크를 추가하여 시스템을 재가동 시킨 후 추가된 디스크를 인식하여 기존 데이터를 재구성해야 한다. 시스템을 재구성한다는 것은 전체 디스크로 분산 저장되어 있는 데이터 및 패리티 블록들을 새롭게 배치한다는 것으로, 대부분의 경우, 시스템의 수행을 중단시킨 후 전체 디스크의 내용을 읽어서 배치 방식에 따라 다시 디스크로 쓰는 것이 일반적이다. 따라서 해당 디스크의 내용을 임시로 저장할 메모리에 드는 비용과 블록들의 재배치을 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 재구성 과정에서의 오버헤드는 엄청나게 커진다. 최근에 저장장치의 추가로 인한 시스템의 정지를 막기 위해 디스크 온라인 탈/부착 방법에 대해 연구되고 있지만, 여전히 기존 데이터에 대한 재구성은 필수적인 작업으로 남아 있다.

또 다른 매핑 방법으로 논리주소와 물리주소를 매핑 할 수 있는 테이블을 사용하는 것이 있는데, 이 방법은 디스크 수의 변화나 스냅샷(snapshot), 오류 블록에 대한 저장위치 수정, 성능에 따른 데이터 재분배 등의 유연성을 제공하지만, 매핑을 위한 테이블 공간이 많이 필요하다는 단점을 가진다.

이 논문에서는 스트라이핑을 하는 시스템에서 논리블록 번호를 물리블록 번호로 변환할 때, SZIT라고 하는 매핑 정보를 이용하는 매핑 방법을 제안한다. SZIT 기반 매핑 방법은 수식으로 매핑 하는 방식과 같이 간단한 수식을 사용하지만 디스크의 추가에 따른 오버헤드가 거의 없으며 매핑 정보의 양이 현저히 적은 장점을 가진다.

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 방법의 테이블 구성방법과 디스크 접근을 위해 논리블록을 물리블록으로 변환하는 방법에 관하여 설명한다. 4장에서는 제안하는 매핑 방법을 기존의 수식, 테이블 기반 매핑 방법과 비교를 하고, 5장에서 논문의 결론과 향후 연구의 방향을 제시한다.

2. 관련 연구

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 스트라이핑(striping), 미러링(mirroring), 패리티를 갖는 스트라이핑(striping with parity)과 같은 다양한 RAID 레벨을 제공한다. 하드웨어 RAID 시스템은 RAID 관리를 RAID 컨트롤러가 수행하기 때문에 CPU의 활용도를 높일 뿐 아니라 버스 트래픽을 감소시킴으로써 높은 성능을 제공한다[5].

그러나, 하드웨어 RAID 시스템의 가격이 매우 비싸기 때문에 이러한 RAID의 기능을 소프트웨어로 구현하는 시도들이 많은 연구에서 진행되었다[7, 8]. 소프트웨어 RAID는 하드웨어 컨트롤러 대신 CPU가 RAID의 관리를 수행하게 된다. 이것은 컨트롤러의 오류 발생시 모든 것이 정지되는 문제를 해결하며, 상대적으로 저렴한 가격의 RAID 시스템을 구성할 수 있게 해준다. 그렇지만 소프트웨어 RAID는 스트라이핑 기능에서 동시에 여러 디스크의 데이터를 읽어서 처리하는 면에서는 하드웨어 RAID의 성능을 내지 못한다. 리눅스의 경우, 한 시점에 한 블록(혹은 연속된 블록)만을 읽을 수 있기 때문에 다른 디스크에 있는 블록을 읽기 위해서는 순차적으로 읽을 수밖에 없기 때문이다. 하지만 다중 호스트에서 다중 디스크를 공유하는 SAN 기반 시스템과같은 시스템에서는 디스크의 집중화(bottle-neck)를 줄일 수 있는 효과를 가진다.

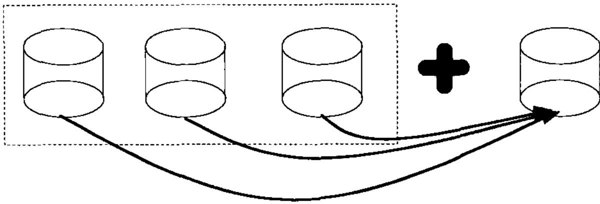
2.2 스트라이핑으로 데이터를 저장하는 시스템에서 디스크 추가 방법

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

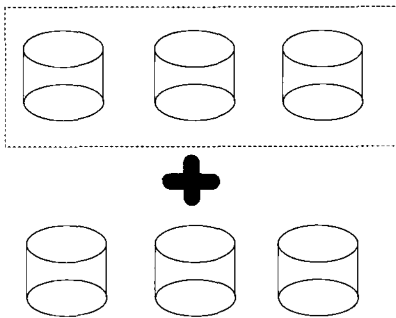
열 추가(add column) 방법은 기존 시스템을 구성하는 디스크의 수를 고려하지 않고 디스크를 추가하기 때문에 이미 시스템에 저장된 데이터들을 새로 재구성해야 한다. 온라인 상에서 재구성하는 것은 상당히 많은 시간이 소요하게 되며 이로 인해 사용자의 일반적인 데이터 접근에 많은 영향을 미치게 된다.

행 추가(add row) 방법은 이미 구성하고 있는 디스크 수를 고려하여 그 배수만큼의 디스크를 논리적으로 연결하는 방법이다. 이 방법은 처음 시스템을 구성하는 디스크 수 n 만큼만 데이터를 분산시킨다. 즉, 1TB 디스크 3개가 있고

1TB 디스크 3개를 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 (그림 3)과 같은 데이터 재구성이 필요하며, 행 추가 방법은 현재 유지되는 디스크 수를 고려하여 그의 배수만큼의 디스크를 추가해야 하는 제약이 발생한다.

2.3 리눅스에서 용량이 다른 디스크를 위한 스트라이핑

리눅스는 서로 다른 용량을 가지는 디스크간의 스트라이핑을 제공하기 위해 스트라이핑 영역(striping zone)이라는 개념을 사용한다[15]. 예를 들어, 100GB(D0), 150GB(D1), 200GB (D2)의 디스크가 있을 때 스트라이핑 영역은 다음과 같이 나누어진다.

〈표 1〉 리눅스의 영역 정보 계산 방법

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

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

2.4 논리주소에서 물리주소로의 매핑 방법

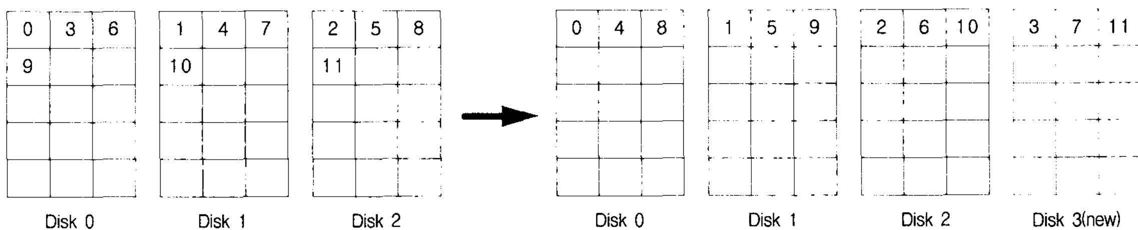
스트라이핑을 할 때, 논리주소와 물리주소를 매핑 하는 방법은 크게 두 가지로 살펴볼 수 있다. 첫 번째 방법은 일반적으로 사용하는 수식을 사용하는 방법이다. 즉, 논리주소를 디스크 수로 모듈러 연산하여 대상 디스크를 선택하고, 디스크 내의 위치는 논리주소를 디스크 수로 계산할 수 있다.

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

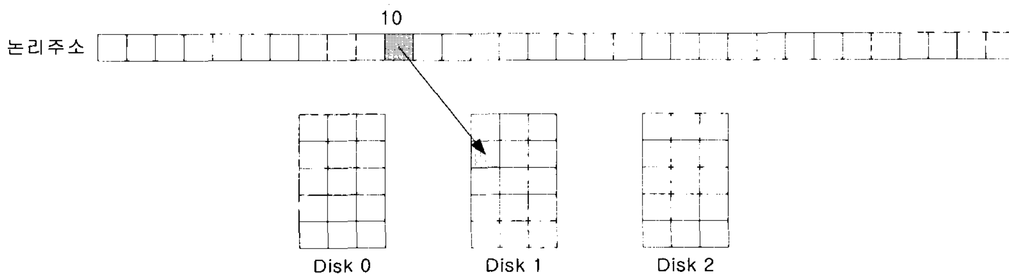
〈표 2〉 논리블록 10에 대한 물리블록 계산 과정

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

즉, 1번 디스크의 3번 물리블록에 저장된다.



(그림 3) 디스크 열 추가로 인한 데이터 재구성



(그림 4) 논리주소 10의 물리주소 변환

이 방법은 간단한 수식을 통해서 주소가 계산되는 장점을 가지지만 저장 디스크의 수가 달라지면 <표 2>의 매핑 수식이 달라지기 때문에 디스크가 추가될 때마다 새로운 식을 적용시켜 기존의 데이터 위치를 재구성해야 한다. 이러한 재구성 과정은 비용이 많이 드는 연산이다.

이러한 디스크 수의 변화에 유연하게 적응하기 위해 <표 3>과 같이 논리주소와 물리주소의 1 : 1 관계를 테이블의 형태로 저장하여 매핑 할 수 있다[14].

〈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
⋮	⋮	⋮

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

매핑 테이블을 사용하여 물리 블록을 할당하는 것은 <표 4>와 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 성능개선 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 되기 때문이다.

〈표 4〉 매핑 테이블을 사용한 물리 블록 할당

```
table_based_mapping ( ) {
   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
}
```

2.4.1 Global File System(GFS)의 매핑 기법

미네소타 대학에서 개발된 공개 SAN 파일 시스템인 GFS[7]는 별도의 매핑 테이블을 두지 않고 계산식을 이용해서 매핑을 수행한다. 위에서 언급한 대로 수학수식 기반 매핑 방법은 단순하지만 유연하지 못하다. 이 방법에서는 가상 디스크가 생성될 때 물리적 디스크들과 그들의 매핑 관계를 정해 놓아야 하며 온라인 재구성이 용이하지 못하다.

2.4.2 Petal의 매핑 기법

Petal[8]은 네트웍으로 연결된 여러 호스트에 부착된 디스크를 가상화하여 하나의 논리 디스크로 클라이언트에게 제공하는 시스템이다. Petal에서는 이 가상화를 매핑 테이블을 이용하여 실현한다. 클라이언트가 이용하는 논리주소

는 (vdiskID, offset)인데, vdiskID는 petal의 가상 디스크 번호이며 offset은 해당 vdiskID 내에서의 위치이다. 이 논리주소가 테이블을거쳐 (serverID, diskID, diskOffset)의 물리주소로 바뀐다. serverID는 해당 디스크가 어느 서버에 속해 있는지를 나타낸다. diskID는 해당 서버의 몇 번째 디스크에 해당하는가를 나타내고, diskOffset은 그 디스크 내에서의 위치를 나타낸다.

3. SZIT 기반 매핑 방법

3장에서는 이 논문에서 제안하는 SZIT 기반 매핑 방법과 이를 위한 부분 스트라이핑 과정에 대하여 설명한다. 이 논문에서 제안하는 SZIT 기반 매핑 방법은 리눅스가 용량이 다른 디스크간의 스트라이핑을 제공하기 위해 스트라이핑 영역의 개념을 사용한 것처럼, 디스크가 추가되는 시점에 스트라이핑 영역을 나누어 각 스트라이핑 영역에 맞는 스트라이핑을 수행한다. 즉, 디스크를 추가하기 전에는 이전 디스크 수를 기반으로 데이터를 나누어 저장하고, 디스크를 추가한 후에는 경우에 따라 새로운 디스크에 부분 스트라이핑으로 데이터를 저장하거나 전체 디스크 수를 기반으로 데이터를 저장하도록 한다. 이와같이 수식을 이용하면서 디스크의 수에도 유연하게 적응하기 위해서는 추가 정보가 필요한데, 이 추가정보는 SZIT(Strip Zone Information Table)라고 하는 테이블에 저장된다.

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

이 논문에서 제안하는 방식을 통해 디스크입출력을 할 경우에 논리주소와 물리주소의 정확한 매핑관계를 알기 위해서는 다음과 같은 추가의 정보가 필요하다.

〈표 5〉 스트라이핑 영역정보 테이블(SZIT)의 구조

영역 번호	전체 디스크수	참여 디스크수	첫 물리 블록	끝 물리 블록	첫 논리 블록	끝 논리 블록

영역 번호는 스트라이핑 영역을 나타내는 번호로 첫 논리블록에 따라 순차적으로 증가한다. 전체 디스크 수는 해당 스트라이핑 영역의 전체 디스크를 나타내고, 참여 디스크는 해당 영역에 참가하는 디스크의 수를 나타낸다. 즉, <표 7>에서 영역 번호 1과 같이 초기에 3개로 구성되었던 시스템에 디스크 2개가 더 추가되었다면 전체 디스크 수는 5이고, 참여 디스크는 2개이다. 영역 번호 1에 대한 스트라이핑은 새로 추가된 두 개의 디스크만을 대상으로 한다.

첫 논리블록과 끝 논리블록은 해당 스트라이핑 영역의 첫 논리블록과 끝 논리블록을 나타내며, 이때의 디스크상의 물리적인 블록번호가 첫 논리블록의 물리블록과 끝 논리블록의 물리블록이 된다.

3.2 SZIT의 갱신

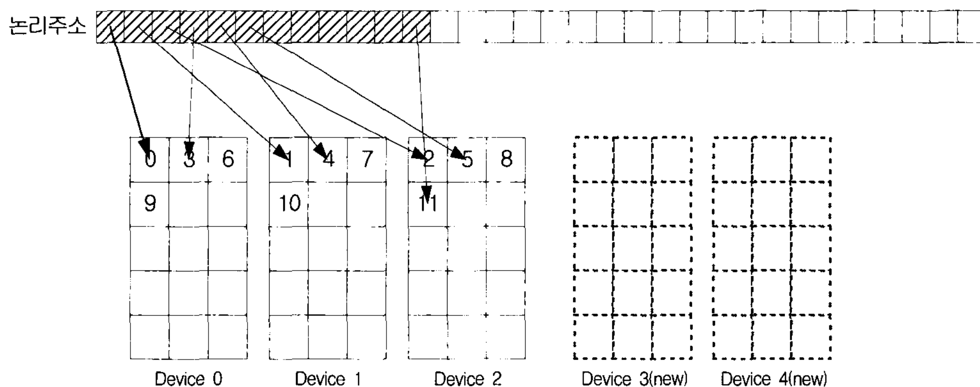
SZIT는 디스크가 추가되는 시점마다 갱신되며, 이 정보를 통해서 논리주소를 물리주소로 변환하게 된다. 디스크가 추가되었을 때 SZIT를 갱신하는 방법은 다음과 같다.

〈표 6〉 SZIT 갱신 과정

```
if (디스크 추가)
{
    현재 스트라이핑 영역정보의 끝 논리블록 및 끝 물리블록
    수정
    추가 스트라이핑 영역정보에 대한 정보 추가
    추가된 영역 이후의 영역에 대한 정보 수정
}
```

최근에 개발되고 있는 많은 제품들이 디스크의 탈부착 상태를 온라인 상으로 검출할 수 있는 방법을 제공하고 있기 때문에, 이러한 이벤트가 발생하면 먼저 마지막으로 쓰기 연산을 수행한 논리블록 번호가 포함된 영역 정보를 수정한다. 끝 논리블록 및 물리블록은 마지막 쓰기 연산이 포함된 스트라이핑 라운드가 끝나는 블록으로 결정된다.

즉, (그림 5)와 같이 3개의 디스크에 대해서 논리블록 9까지 쓰기 연산이 수행된 후에 디스크가 추가된 경우에서,



(그림 5) 디스크 추가 전의 전체 스트라이핑

논리블록 9가 포함된 스트라이핑 영역 0의 끝 논리주소는 스트라이핑 라운드가 끝나는 11이 된다. 논리블록 10, 11까지 쓰기 연산이 수행된 경우에도 동일하게 끝 논리주소는 11이 된다.

추가 스트라이핑 정보는 대부분 2개 생성되는데 하나는 새로 추가된 디스크에 부분 스트라이핑을 하는 영역에 관한 정보이고, 다른 하나는 기존의 디스크와 추가된 디스크로 만들어지는 영역에 대한 정보가 된다.

<표 6>은 디스크 3개로 운영중인 시스템이 논리블록 9~11을 쓰는 도중에 디스크가 추가된 경우의 SZIT의 모습이다. 한 디스크에는 블록이 15개씩 있다고 가정하였다.

〈표 7〉 저장매체 추가 이후의 SZIT

영역 번호	전체 디스크 수	참여 디스크 수	첫 물리 블록	끝 물리 블록	첫 논리 블록	끝 논리 블록
0	3	3	0	3	0	11
1	5	2	0	3	12	19
2	5	5	4	14	20	74

3.3 부분 스트라이핑 영역(Partial Striping Zone)

새로운 디스크의 추가로 SZIT가 갱신된 이후에 발생하는 데이터는 추가된 디스크에만 저장된다. (그림 6)은 이러한 과정을 보여준다. 추가한 디스크에 데이터가 저장되는 영역을 부분 스트라이핑 영역(Partial Striping Zone)라 한다.

(그림 6)은 논리블록 12번부터 19번까지 쓰여지는 과정을 보여준다. 이때 스트라이핑 영역 1에 참여하는 디스크의 수는 두개이다.

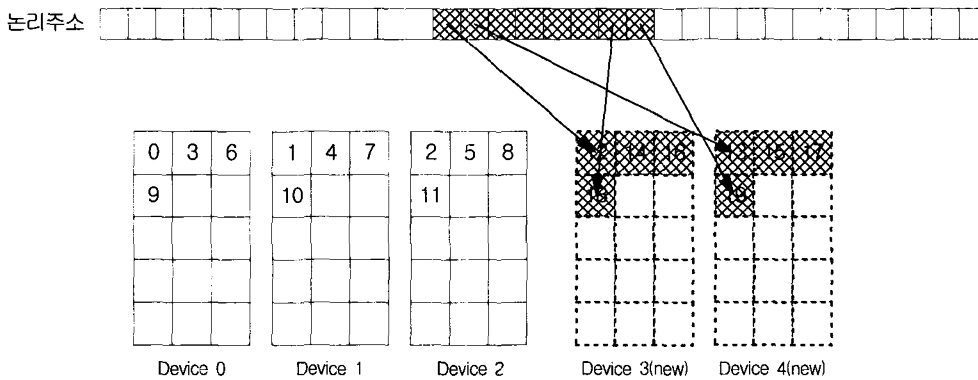
부분 스트라이핑은 기존 디스크에 저장된 데이터의 양만큼 추가된 디스크에 데이터가 저장될 때까지 수행되며, 그 이후로는 완전 스트라이핑으로 전체 디스크에 데이터를 저장한다.

(그림 7)은 스트라이핑 영역 1의 쓰기 연산이 모두 끝난 후 논리블록 20번부터 스트라이핑 영역 2의 쓰기 연산이 수행되는 것을 보여준다. 이때 스트라이핑에 참여하는 디스크의 수가 전체 디스크 수와 동일한 5개이다.

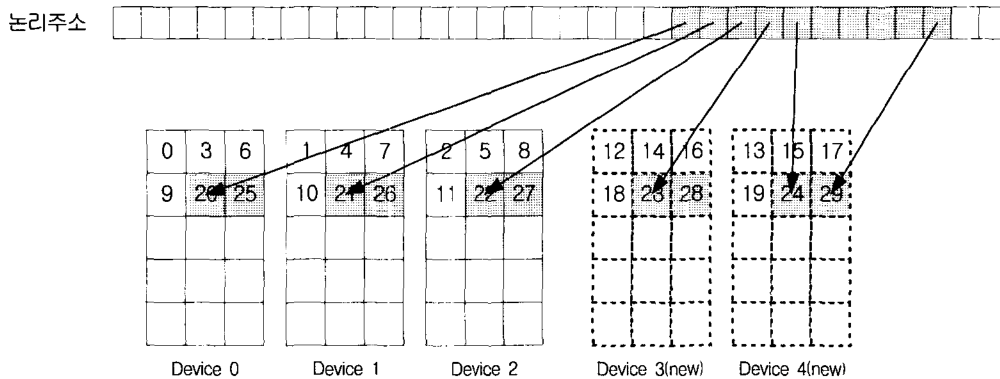
3.4 SZIT 기반 매핑 방법

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

SZIT 기반 매핑은 다음과 같은 수식을 통해 논리블럭을 물리블럭으로 변환한다.



(그림 6) 추가 디스크에 부분 스트라이핑



(그림 7) 디스크 추가 후의 전체 스트라이핑

〈표 8〉 SZIT 기반 매핑 계산식

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

<표 9>는 SZIT 기반 매핑 과정을 나타낸다. 시스템에 디스크가 추가되면 SZIT를 갱신한다. SZIT가 갱신되는 과정에서 SZIT를 참조하여 물리주소를 계산하는 경우에 잘못된 값을 가질 수 있기 때문에 잠금(lock)으로 보호한다. SZIT의 갱신 후에는 반드시 SZIT를 디스크에 반영하는데, 이는 시스템을 재가동 시킬 때도 매핑의 일관성을 유지하기 위해서이다. SZIT 갱신 이후로는 요청된 블록에 대해서 <표 8>의 계산식으로 물리 블록을 계산하여 디스크에 접근한다.

〈표 9〉 SZIT 기반 매핑 과정

```
if ( ADD_DISKS )
{
      lock_SZIT ( )
      update SZIT
      unlock_SZIT ( )
      write SZIT to disks
                        // DISK OPERATION
(METADATA)
}
if ( operation )
{
      get physical block number from logical block number
      using SZIT
      if ( operation == READ )
            get real data from disk
                        // DISK OPERATION (REAL
DATA)
      else // operation == WRITE
            write real data to disk
                        // DISK OPERATION (REAL
DATA)
}
```

3.5 RAID 5에서의 SZIT 사용

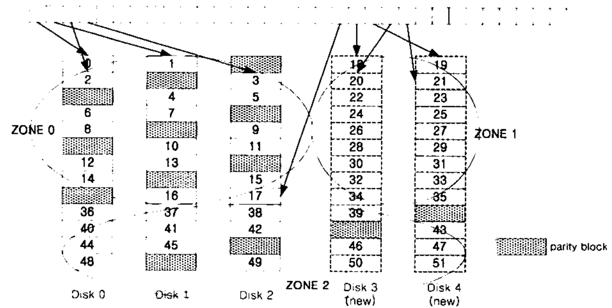
앞에서 살펴본 SZIT를 통한 매핑 방법은 RAID 0에 적용되는 방식이다. 이를 RAID 5에서 사용할 수 있도록 하려면 오류비트(parity bit)에 대한 고려를 해야 한다. 즉, RAID 0에서의 SZIT 기반 매핑과 같이 추가된 디스크에만 데이터를 저장하는 부분 스트라이핑 영역에 대해서 패러티 블록은 존재하지 않는다. 패러티 블록은 디스크가 추가되기 전에 완전 스트라이핑으로 데이터를 저장하는 영역에서 사용하는 패러티 블록을 함께 사용한다.

디스크 3개로 서비스를 제공하고 있는 RAID 5 시스템에서 디스크 2개를 추가한 경우는 다음과 같다. SZIT의 모습은 <표 9>와 같이 되고, 이에 따른 데이터 저장 순서는 (그림 8)과 같이 된다.

〈표 10〉 디스크 추가 이후의 SZIT

영역 번호	전체 디스크 수	참여 디스크 수	첫 물리 블록	끝 물리 블록	첫 논리 블록	끝 논리 블록
0	3	3	0	8	0	17
1	5	2	0	8	18	35
2	5	5	9	12	36	51

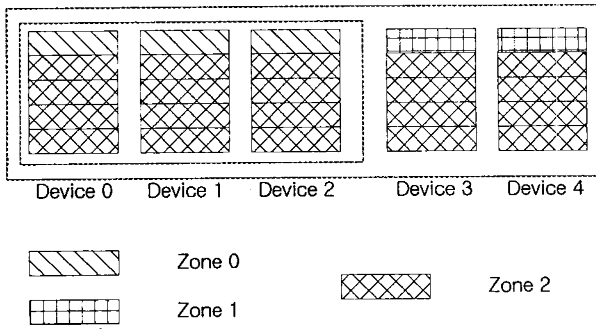
논리블록 18번 이후에 디스크가 추가되어 논리블럭 18부터 35까지는 새로운 디스크에만 저장된다. 각 블록들이 새 디스크에 저장될 때마다 이 블록에 의한 패러티 블록은 기존의 디스크들에 있는 패러티 블록을 이용한다. 즉, 18번 블록이 저장될 때는 2번 디스크의 0번 물리블록, 21번 블록이 저장될 때는 1번 디스크의 1번 물리블록을 패러티 블록으로 사용한다.



(그림 8) RAID5에서의 SZIT 기반 매핑

3.6 데이터 재구성

SZIT 기반 매핑 방법은 디스크 추가에 따른 데이터 재구성을 하지 않는 방법이 아니라, 디스크 추가 즉시 서비스를 계속할 수 있는 방법이다. 데이터 재구성의 입장에서 본다면, SZIT 기반 매핑 방법은 디스크 추가로 인한 데이터 재구성 연산을 지연시키는 효과를 가진다. (그림 9)는 모든 디스크의 크기가 같고 초기 상태에 디스크 3개에 20%의 데이터가 있고, 새로 디스크 2개를 추가하여 20%까지 쓴 후, 전체 디스크 5개로 스트라이핑을 하고 있는 시스템의 데이터가 저장된 모습이다. 이러한 디스크에서 재구성의 대상이 되는 데이터는 스트라이핑 영역 0과 1에 있는 데이터이다. 즉, 각 단위는 전체 디스크의 20%이며 이를 S라고 하면, 5×S 만큼의 데이터를 재구성해야 한다. 만일, 기존의 방법으로 재구성을 할 경우에 재구성 대상이 되는 데이터는 스트라이핑 영역 0에 있는 데이터이다. 즉, 3×S가 된다. 그러므로, 부분 스트라이핑으로 저장한 후 나중에 재배치을 하게 되면, 기존 디스크에 데이터가 많이 채워져 있을수록 디스크 추가 즉시 재구성하는 방법에 비해서 재구성 대상이 되는 데이터의 양은 많아진다.



(그림 9) SZIT 기반 매핑에서의 재구성 대상 데이터

3.7 스트라이핑 영역 정보의 저장

SZIT를 메모리에서만 관리하게 되면 시스템을 다시 시작할 때, 이러한 정보를 잃게 되므로, 이 정보는 디스크에도 저장해야 한다. 시스템의 재동작시에는 디스크의 첫 부분에 저장된 SZIT를 읽어서 시스템의 SZIT를 만들게 된다. SZIT의 내용은 시스템에 새로운 디스크가 추가될 때에만 변경되는데, 이러한 빈도는 시스템 운영중에 수차례 밖에 발생하지 않는 것이 일반적이며 시스템에 요청되는 디스크 연산(읽기, 쓰기 연산) 수에 비교 했을 때는 그 수가 아주 미미하다. 디스크가 추가되면 이러한 변화를 메모리상의 SZIT 정보를 먼저 수정한 후, 각 디스크의 레이블 정보로 중복해서 저장한다.

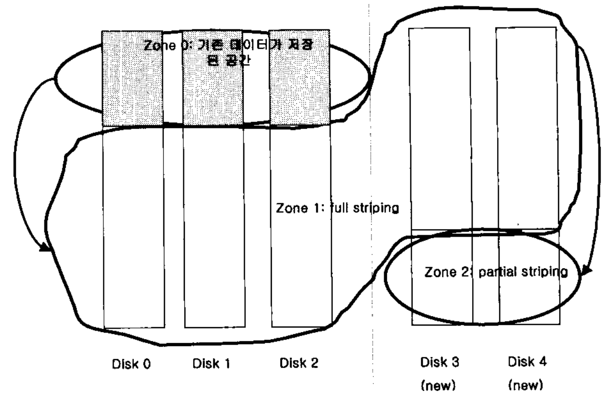
4. 매핑 방법 비교

4.1 테이블 기반 매핑 방법과 SZIT 기반 매핑 방법에서의 디스크 추가 후 데이터 저장 순서

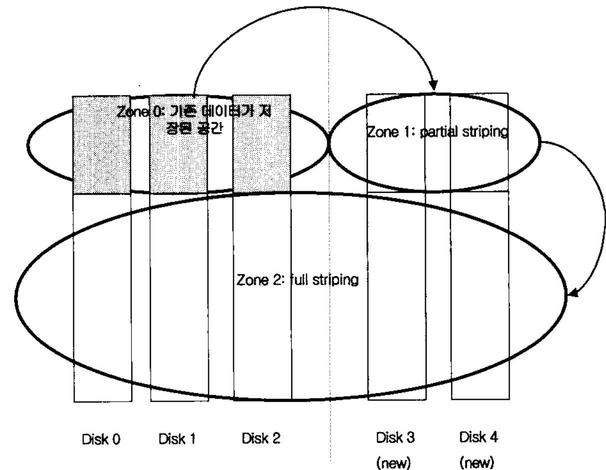
테이블 기반 매핑 방법에서 스트라이핑으로 저장할 때, 디스크가 추가되면 재구성을 하기 전에는 (그림 10)과 같이 저장한다. 즉, 디스크가 추가된 시점부터 모든 디스크에 순서대로 저장된다. 따라서, 스트라이핑으로 저장하더라도 부분 스트라이핑을 하는 영역(zone 2)이 존재한다. SZIT 기반 매핑 방법도 원리는 같지만 데이터를 저장하는 순서에 차이가 있다.

(그림 10)은 테이블 기반 매핑 방법에서 디스크가 추가되었을 때의 저장 순서이다. 테이블 기반 매핑 방법은 디스크가 추가되기 이전 영역(zone 0)을 그대로 유지하고, 디스크 추가 즉시 모든 디스크를 대상으로 데이터를 저장한다. 하지만, 특정 디스크에서 공간을 할당해야 할 때, 더 이상 할당할 공간이 없으면 다음 디스크로 할당을 요청한다. 따라서, 마지막에는 디스크 3, 4에만 데이터가 저장되고, 이 영역에서는 부분 스트라이핑을 수행하게 된다.

(그림 11)는 SZIT 기반 매핑에서 디스크가 추가되었을 때의 저장 순서이다. 테이블 기반 매핑 방법과 같이 전체 디스크를 대상으로 스트라이핑을 수행하는 영역(zone 2)과 부분 스트라이핑을 수행하는 영역(zone 1)이 존재하지만, 부분 스트라이핑을 먼저 하는 것이 차이점이다.



(그림 10) 테이블 기반 매핑에서 디스크 추가에 따른 데이터 저장 위치 이동 순서



(그림 11) SZIT 기반 매핑에서 디스크 추가에 따른 데이터 저장 위치 이동 순서

4.2 각 매핑 방법에서 유지하는 메타데이터

4.2에서는 논문에서 제안하는 SZIT 기반 매핑 방법과 수식 기반 매핑 방법, 테이블 기반 매핑 방법에서 유지해야 하는 메타데이터의 양을 비교한다. 메타데이터의 양이 많아지면 모든 메타데이터를 메인 메모리에 두고 참조할 수 없기 때문에 빈번한 디스크 접근을 유발하며, 빈번한 디스크 접근은 시스템의 성능을 저하시킨다.

수식 기반 매핑 방법은 간단한 수식을 통해서 매핑 하는 방법이다. 이 방법은 논리블록을 물리블록으로 변환하는데 있어서 수식을 사용하기 때문에 따로 저장해야할 매핑 데이터는 전혀 없다.

이에 반해서 테이블 기반 매핑 방법은 논리블록과 물리블록 간의 1 : 1 관계를 테이블로 저장한다. 따라서 디스크의 용량이 커질수록 테이블의 크기도 커지는 단점을 가진다.

논문에서 제안하는 SZIT 기반 매핑 방법은 기본적으로 수식을 사용하지만 SZIT라 불리는 적은 양의 메타데이터를 이용하여 매핑을 수행한다. SZIT 기반 매핑 방법은 디스크가 추가될 때에만 메타데이터의 양이 증가하지만, 메인 메

모리에 충분히 올려서 사용할 수 있는 크기이다. 각 매핑 방법에서 관리하는 메타데이터의 크기는 <표 11>과 같다.

〈표 11〉 각 매핑 방법의 메타데이터

	수식 기반 매핑	테이블 기반 매핑	SZIT 기반 매핑
저장할 매핑 데이터 양	불필요	디스크의 블록 수에 비례	디스크 추가 횟수에 비례

테이블 기반 매핑에서 사용하는 테이블에서 한 블록을 위해 저장하는 정보의 크기는 9Byte이다. 테이블의 정보는 논리블록주소(4Byte), 디스크 번호(1Byte), 물리블록번호(4Byte)로 구성된다. 테이블의 크기는 디스크 블록 수에 비례하기 때문에, 블록의 크기가 1KB이고 20GB 용량의 디스크 3개로 구성된 시스템에서 저장해야 할 테이블의 크기는 540MB(20GB/1KB ×3×9)가 된다. 이 크기는 전체 디스크의 0.9%이다.

SZIT 기반 매핑에서 하나의 영역(zone)을 위해 저장하는 정보의 크기는 19Byte이다. SZIT 정보는 스트라이핑 영역번호 (1Byte), 전체 디스크 수(1Byte), 참여 디스크 수(1Byte), 첫 물리블록번호(4Byte), 끝 물리블록번호(4Byte), 첫 논리블록번호(4Byte), 끝 논리블록번호(4Byte)로 구성된다. 따라서 위의 예와 같은 시스템의 경우에 저장해야 할 SZIT의 크기는 단지 19Byte이다. 이는 디스크 추가가 한번도 발생하지 않았기 때문에 전체 디스크가 하나의 영역으로 구성되기 때문이다.

〈표 12〉 디스크 수의 변화에 따른 매핑 메타데이터

	초기 디스크	첫 번째 추가	두 번째 추가
수식 기반 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 디스크 용량	60 Giga Byte	100 Giga Byte	140 Giga Byte

<표 12>은 위의 예와 같이 블록의 크기가 1KB이고, 20 GB 디스크 3개로 구성된 시스템(초기 디스크)에 20GB 크기의 디스크 두 개를 추가한 후(첫 번째 추가), 다시 같은 크기의 디스크 두 개를 추가하였을 때(두 번째 추가) 저장해야 할 메타데이터의 크기를 나타낸다. SZIT 기반 매핑 방법과 테이블 기반 매핑 방법 모두 디스크 수가 증가함에 따라 저장해야 할 메타데이터의 크기가 증가한다. 하지만 SZIT 기반 매핑 방법의 경우에 저장해야 할 메타데이터의 양은 매우 적기 때문에 메인 메모리에 모두 저장할 수 있다. 따라서 시스템을 운영하는 중에 SZIT의 참조를 위해 디스크 접근이 일어나는 경우는 거의 없기 때문에 매핑을 수행하는 속도는 수식 기반 매핑 방법과 유사하다.

4.3 각 매핑 방법에서의 디스크 접근 수

4.3에서는 각 매핑 방법에서의 디스크 접근 수를 비교한다. 디스크 접근 수는 시스템 성능을 크게 좌우하는 요소이다. 디스크 접근 수는 크게 실제 데이터를 위한 디스크 접근과 메타데이터를 위한 디스크 접근으로 나눌 수 있는데, 실제 데이터를 위한 디스크 접근은 각 매핑 방법에서 동일하게 필요하다. 이에 반해, 메타데이터를 위한 디스크 접근은 매핑 방법마다 다르며 이로 인해 성능의 차이가 발생한다.

수식 기반 매핑 방법은 어떤 경우에서도 메타데이터를 위한 디스크 접근이 발생하지 않는다.

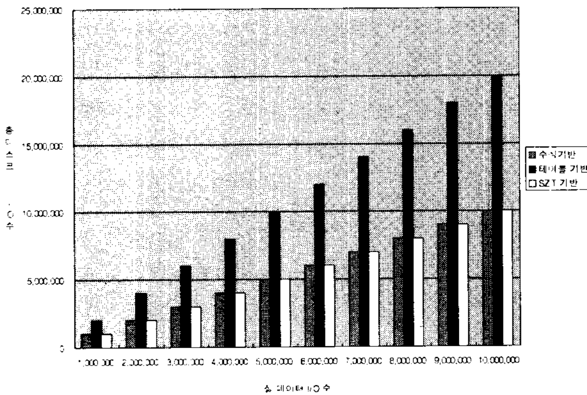
테이블 기반 매핑 방법은 경우에 따라 메타데이터를 위한 디스크 접근의 수가 다르다. 먼저, 읽기의 경우 요청된 블록에 대해 물리블록 번호를 알기 위해 매핑 테이블을 읽어야 한다. 매핑 테이블을 읽기 위해서 한번의 디스크 접근이 필요하며, 물리블록 번호를 알게 되면 이를 통해서 실제 데이터를 읽을 수 있다. 이와 같이 테이블 기반 매핑 방법은 실제 데이터를 위한 디스크 접근 외에 추가로 한번의 디스크 접근이 필요하다. 쓰기의 경우는 다시 두 가지로 나뉘는데, 하나는 이미 디스크에 저장되어 있는 데이터를 갱신하는 것이고, 다른 하나는 새로운 블록을 저장하는 것이다. 이미 디스크에 저장되어 있는 데이터를 갱신할 때는 읽기의 경우와 동일한 수의 디스크 접근이 필요하다. 하지만, 처음 디스크에 쓰기를 하는 경우에는 매핑 테이블을 읽어 매핑 정보가 없음을 확인해야 한다. 매핑 정보가 없다면 물리블록을 결정해야 하는데, 이를 위해서 자유공간관리자가 필요하다. 자유공간관리자는 보통 자유공간을 비트맵으로 관리하는데, 비트맵 블록 중에서 자유공간을 찾아서 그의 물리주소를 반환하는 역할을 한다. 자유공간 비트맵의 크기 또한 디스크의 크기가 증가함에 따라 커지기 때문에 디스크에 저장하게 된다. 따라서 자유공간 비트맵을 읽기 위해서 추가의 디스크 접근이 필요하다. 첫 번째 읽은 비트맵 블록에서 자유공간을 찾을 수도 있지만, 최악의 경우는 마지막 읽은 비트맵 블록에서 자유공간을 찾을 수도 있다. 또한 자유공간 비트맵 블록에서 자유공간을 찾으면 해당 비트를 세팅하고 디스크에 저장해야 한다. 따라서 자유공간 비트맵 블록에서 자유공간을 할당하기 위해서는 최소한 두

〈표 13〉 각 매핑 방법들의 디스크 접근 수

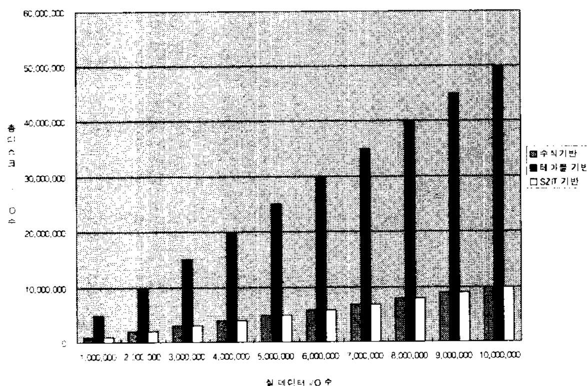
		수식 기반 매핑	테이블 기반 매핑	SZIT 기반 매핑
읽기	실제 데이터	1	1	1
	메타데이터	0	1	0
	총 디스크 접근 수	1	2	1
갱신	실제 데이터	1	1	1
	메타데이터	0	1	0
	총 디스크 접근 수	1	2	1
처음 쓰기	실제 데이터	1	1	1
	메타데이터	0	4 (최소)	0
	총 디스크 접근 수	1	5 (최소)	1

번의 디스크 접근이 필요하다. 결론적으로 테이블 기반 매핑 방법에서 처음 디스크에 저장되는 블록을 위해서는 최소한 5번의 디스크 접근 (실제 데이터를 위한 디스크 접근+테이블 읽기/쓰기, 자유공간 블록 읽기/쓰기)이 필요하다.

(그림 12)와 (그림 13)은 읽기/쓰기 연산에 따른 각 매핑 방법에서의 디스크 접근 수를 나타낸다. 초기에 디스크 3개로 구성된 시스템에서, 100만번의 데이터 블록에 대한 요청이 있을 때마다 디스크의 수가 2개씩 증가한다고 가정하였다. 따라서, 100만번의 블록 요청 후에 시스템은 5개의 디스크를 가지고, 200만번의 블록 요청 후에는 7개의 디스크를 가지게 된다. 또한, 단순한 비교를 위해 모든 블록은 접근 요청을 하였을 때 반드시 디스크로부터 수행하도록 하였다. 캐쉬의 크기와 캐쉬 정책에 따라 다소 성능의 차이가 발생할 수도 있으나, 이는 모든 방법에 동일한 효과를 내기 때문이다.



(그림 12) 읽기 연산의 디스크 접근 수



(그림 13) 쓰기 연산의 디스크 접근 수

(그림 12)는 블록 읽기에 대한 디스크 접근 수를 나타낸다. 수식 기반 매핑 방법은 요청 블록 수와 동일한 수의 디스크 접근을 수행하게 된다. 하지만 테이블 기반 매핑 방법은 요청 블록 수의 2배의 디스크 접근을 수행하게 된다. 따라서, 100만번의 블록이 요청되면 요청 블록을 위해 200만번의 디스크 접근이 필요하다. 이는 디스크에 저장되어 있는 매핑 테이블을 읽어서 요청된 논리블록의 물리블록을 얻어야 하기 때문이다. 이미 디스크에 저장되어 있는 블록에 대한 쓰기 연산 또한 동일한 그래프를 얻을 수 있다. SZIT 기반 매핑 방법은 디스크가 추가되는 시점에 메타데이터를 위한 추가의 디스크 접근이 발생할 뿐 수식 기반 매핑 방법과 동일한 디스크 접근 수를 가진다. 100만번의 블록 요청 후에는 디스크가 5개가 되므로 SZIT를 5개의 디스크에 반영한다. 따라서 100만번의 블록 때까지 발생하는 디스크 접근의 수는 1,000,005가 된다. 200만번의 블록 요청 후에는 디스크가 7개가되기 때문에 2,000,012(1,000,005+1,000,007)가 된다.

(그림 13)은 새로운 블록 쓰기에 대한 디스크 접근 수를 나타낸다. 블록 읽기에서와 마찬가지로 수식 기반 매핑 방법은 요청 블록 수와 동일한 수의 디스크 접근을 수행한다. 테이블 기반 매핑 방법은 매핑 테이블을 읽고, 자유공간 비트맵을 찾아 갱신하고, 다시 매핑 테이블의 특정 블록을 갱신하는 추가의 디스크 접근이 필요하다. 따라서, 한번의 새로운 블록 쓰기에 대해, 최소 5번의 디스크 접근이 필요하다. SZIT 기반 매핑 방법은 읽기 연산의 경우와 동일하다.

5. 결론 및 향후 연구

이 논문에서는 스트라이핑을 하는 소프트웨어 RAID 시스템에서 디스크를 추가할 때, 기존 데이터에 대한 재구성 작업 없이 즉시 서비스를 할 수 있는 매핑 방법을 제안하였다. 그 방법은 리눅스에서 서로 다른 용량을 가지는 디스크간의 스트라이핑을 위해 스트라이핑 영역을 나누어 그들간의 다른 스트라이핑을 하는 방법을 응용한 것으로, 디스크가 추가될 때마다 달라지는 스트라이핑 대상 디스크의 수에 대한 정보를 테이블로 유지하여 디스크 입/출력 요청시 간단한 수식과 테이블 정보를 이용하여 매핑을 수행한다. 이 테이블을 SZIT(Strip Zone Information Table)라하고, 이 테이블을 통하여 읽기, 쓰기를 할 실제 디스크와 디스크 내의 위치를 계산한다. 만일, 디스크가 새로 추가된 후에 발생하는 쓰기 연산은 새로 추가된 디스크만을 통해 스트라이핑으로 데이터를 저장하고, 기존 디스크의 용량만큼 데이터가 채워지면 그때부터 전체 디스크를 대상으로 스트라이핑을 수행하여 데이터를 저장한다. 이 과정이 일어나는 영역을 부분 스트라이핑 영역 (Partial Striping Zone)라고 한다. 이 방법은 보편적인 매핑 테이블 방법이 과도한 메타데이터 양 때문에 성능에 심각한 문제를 일으키는 점을 보안하면서 계산식에 의한 빠른 매핑을 가능케 한다. 또한 제안하는 방법에서 추가 시점 이후에 재구성을 할 수도 있다. SZIT 기반 매핑 시스템에서 재구성 시점에 따라 재구성 대상 데이터가 증가하지만, 재구성 연산을 지연시키는 효과를 가진다.

즉, SZIT 기반 매핑 방법은 수식 기반 매핑 방법처럼 매핑의 과정이 간단하여 빠른 연산속도를 낼 수 있으며, 수식 기반 매핑 방법에서 치명적인 단점으로 지적되는 디스크 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 두고 사용할 수 있다.

이 논문에서 제안하는 SZIT 기반 매핑 방법은 파일 시스템의 하부에 존재하는 볼륨관리자에서 구현된다. 따라서 제안하는 방법이 제대로 동작하기 위해서는 파일시스템에서 온라인 확장이 제공되어야 한다. 또한, SZIT 기반 매핑을 위해서는 데이터를 디스크에 순차적으로 저장해야 하며 볼륨관리자는 디스크가 추가되었을 때, 파일시스템으로부터 마지막으로 쓴 논리블록 번호를 알 수 있어야 한다.

현재 모든 디스크의 용량, 블록의 크기가 모두 동일하다는 가정을 하고 있다. 따라서 다양한 크기의 용량을 지원할 수 있는 방안과 데이터 블록의 오류 발생시 처리방법 및 스냅샷(snapshot) 지원을 위한 방법에 대한 연구가 필요하다. 또한 메모리상의 SZIT 변경 사항을 디스크로 반영할 때 장애가 발생한 경우에 대한 대처방안에 대한 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] Steven R. Soltis, Thomas M. Ruwart, et al, "The Global File System," Proceedings of the fifth NASA Goddard Conference on Mass Storage Systems, Sept., 1996.

[8] Edward K. Lee, Chandramohan A. Thekkath, "Petal : Distributed Virtual Disks," Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operatiing Systems, 1996.

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

[10] 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.

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

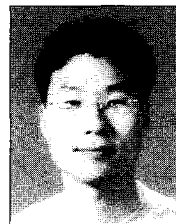
[12] Edward K, Lee and Randy H. Katz, "*Performance Consequences of Parity Placement in Disk Arrays*," In Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System, April, 1991.

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

[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, June, 2001.

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

[16] 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), pp. 267-280, Novermber, 1994.



박 유 현

e-mail : bakyh@etri.re.kr

1996년 부산대학교 전자계산학과 (학사)

1998년 부산대학교 대학원 전자계산학과 (이학석사)

2000년 부산대학교 대학원 전자계산학과 박사수료

2000년 한국국방연구원(KIDA) 자원관리연구부 연구원

2001년~현재 한국전자통신연구원 컴퓨터소프트웨어 연구소 컴퓨터시스템 연구부 연구원

관심분야 : 자료저장시스템, 분산시스템, 데이터베이스, 이동컴퓨팅, 컴퓨터 교육

김 창 수

e-mail : cskim7@etri.re.kr

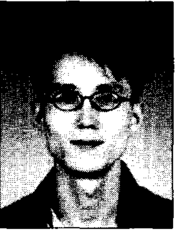
1993년 광운대학교 전자계산학과(학사)

1995년 서강대학교 전자계산학과(학사)

1995년~1999년 LG 소프트(현재 LGEDS)

1999년~현재 한국전자통신연구원 컴퓨터 소프트웨어 연구소 컴퓨터시스템 연구부 선임연구원

관심분야 : 데이터베이스 시스템, 자료저장시스템, 클러스터 시스템, 분산 시스템



강 동 재

e-mail : djkang@etri.re.kr

1999년 인하대학교 전자계산공학과(학사)

2001년 인하대학교 전자계산공학과(석사)

2001년~현재 한국전자통신연구원 컴퓨터 소프트웨어 연구소 컴퓨터시스템 연구부 연구원

관심분야 : 자료저장시스템, GIS, 데이터베이스, 시스템프로그램

김 영 호

e-mail : kyh05@etri.re.kr

1999년 충북대학교 정보통신공학과(학사)

2001년 충북대학교 정보통신공학과(석사)

2001년~현재 한국전자통신연구원 컴퓨터 소프트웨어 연구소 컴퓨터시스템 연구부 연구원

관심분야 : 자료저장시스템, 클러스터링 시스템, 내용기반 이미지 데이터베이스

신 범 주

e-mail : bjshin@mnu.ac.kr

1983년 경북대학교 전자공학과 (학사)

1991년 경북대학교 컴퓨터공학과 (석사)

1998년 경북대학교 컴퓨터공학과 (박사)

1987년~2002년 한국전자통신연구원 컴퓨터소프트웨어 연구소 컴퓨터시스템 연구부 책임연구원, 시스템 소프트웨어연구팀장

2002년~현재 밀양대학교 컴퓨터공학과 교수

관심분야 : 분산시스템, 고장감내 미들웨어, 이동컴퓨팅, 스토리지 클러스터 S/W