리눅스 클러스터 시스템에서 단일 디스크 입출력 공간을 지원하는 효율적 디스크 공유 기법

(An Efficient Disk Sharing Technique supporting Single Disk I/O Space in Linux Cluster Systems)

김 태 호 † 이 종 우 †† 이 재 원 ††† 김 성 동 †††† 채 진 석 †††††
(Taiho Kim) (Jongwoo Lee) (Jae Won Lee) (Sung-Dong Kim) (Jinseok Chae)

요 약 가격 대 성능비가 좋다는 장점으로 인해 많이 사용되고 있는 클러스터 병렬 컴퓨터 시스템에서는 여러 노드에 산재해 있는 자원들을 사용자들이 투명하게 사용할 수 있도록 지원하는 것이 필수적이다. 본 논문에서는 클러스터 시스템에서 단일 디스크 입출력 공간을 지원하는 효율적인 디스크 공유 기법을 제안한다. 응용 수준이 아닌 운영 체제 내의 블록 장치 드라이버 수준에서 디스크 공유를 지원함으로써 사용자들은 로컬 및 원격 디스크를 구분할 필요 없이 클러스터 시스템 내의 모든 디스크들을 마치 로컬 디스크 인 것처럼 투명하게 사용할 수 있다. 기반 운영체제로는 리눅스를 사용하였으며, 실험 결과 단일 디스크 입출력 공간을 성공적으로 지원함과 동시에 비교적 단순한 전역 캐쉬 일관성 정책을 사용했음에도 성능 면에서 NFS에 비해 읽기 성능은 유사, 쓰기 성능은 월등히 향상됨을 확인할 수 있었다. 본 논문이 기여하는 바는 블록 장치 드라이버 수준에서 단일 디스크 입출력 공간을 지원하는 기법을 제안함으로써 블록 장치 드라이버에 비해 비교적 구현이 쉬운 기타 입출력 장치 드라이버에 대해서도 유사한 방식으로 단일 입출력 공간을 지원할 수 있도록 도움을 줄 수 있다는 점이다.

키워드 : 단일 시스템 이미지, 리눅스, 클러스터 시스템, 가상 블록 장치 드라이버

Abstract One of very important features that are necessarily supported by clustered parallel computer systems is a single I/O system image in which users can access both the local and remote I/O resources transparently. In this paper, we propose an efficient disk sharing technique supporting a single disk I/O system image architecture. The design separates the I/O subsystem of a cluster into the file system and a set of virtual hard disk drivers. The virtual hard disk driver deals with a hard disk in the remote node as a local hard disk. All services provided by it are performed in the device driver level without any modification of file systems. Users can, therefore, access all the disks in the cluster regardless of their locations. Our virtual hard disk driver is implemented under the linux, and also tested in a linux cluster system. We find by experiments that it can successfully support a single disk I/O space, and at the same time it shows better performance than NFS. We are sure that this paper can be a guideline for single I/O space of other devices to be easily constructed.

Key words : Single System Image, Linux, Cluster System, Virtual Block Device Driver

† 비 회 원 : (주)티컴앤디티비로 부설연구소 연구원
thkim@tcom-dtvro.com
†† 종신회원 : 아이닉스소프트(주) 개발이사
jwlee44@shinbiro.com
††† 비 회 원 : 성신여자대학교 컴퓨터정보학부 교수
jwlee@cs.sungshin.ac.kr
†††† 종신회원 : 한성대학교 컴퓨터공학부 교수
sdkim@hansung.ac.kr
††††† 종신회원 : 인천대학교 컴퓨터공학과 교수
jschae@incheon.ac.kr
논문접수 : 2003년 1월 13일
심사완료 : 2003년 9월 18일

1. 서 론

오늘날의 컴퓨팅 환경에서 클러스터 시스템은 저비용으로 고성능을 제공함으로써 무한한 잠재력을 인정받고 있다. 이러한 환경에서 단일 시스템 이미지의 지원은 사용자로 하여금 클러스터 시스템을 보다 효율적으로 관리 및 활용할 수 있게 한다. 특히, 클러스터 시스템에서는 입출력 연산이 전체 시스템의 병목점이 되지 않도록 하기 위해 데이타를 분산하여 저장할 필요가 있으며 이러한 서비스를 지원하는 단일 입출력 공간을 요구하게 된다. 단일 입출력 공간이 지원되는 클러스터 내에서는

원격 노드에 장착되어 있는 입출력 장치들을 지역 노드 내의 장치인 것처럼 사용할 수 있다. 특히, 클러스터 시스템의 대표적인 응용 분야라고 할 수 있는 클러스터 웹 서버에서는 컨텐츠들이 클러스터 내의 각 서버에 분산되어 저장되기 때문에 단일 입출력 공간이 지원되지 않는다면 웹 서버 운영뿐만 아니라 성능에도 지장을 줄 수 있다.

본 논문에서는 단일 입출력 공간을 제공하기 위해 기존에 제시된 접근 방식들의 문제점을 분석하고, 이를 해결하기 위해 리눅스 클러스터 환경에서 가상 하드 디스크 드라이버(Virtual Hard Disk Driver, VHDD)를 설계 및 구현한다. 파일 시스템, 데이타베이스 등과 같은 클라이언트들은 가상 디스크들의 집합으로 VHDD를 보게 되는데, VHDD에 의해 제공되는 인터페이스는 로컬 디스크 접근 방식과 동일하기 때문에 클라이언트들은 일관된 인터페이스를 통해 지역 디스크와 가상 디스크를 접근할 수 있다. 이러한 방식은 시스템 전체의 디스크 자원을 통합하여 단일 디스크를 제공하는 SIOS[1]와 구별되며 Petal[2]에서 볼 수 있는 서비스 방식이다. VHDD는 Petal과 큰 구조를 같이하고 있지만, 서버의 사용자 수준 프로세스로의 RPC 대신 커널 스레드를 도입하여 서비스 처리의 성능 향상을 시도하였으며, Petal에서 다루어지지 않았던 캐쉬의 일관성에 대한 문제에는 멀티캐스팅 기법을 적용하여 구체적으로 해결하고자 하였다. 본 논문에서는 구현 플랫폼으로 리눅스 기반 클러스터 서버를 채택하였는데, 이는 리눅스 시스템이 가장 범용성이 높으면서도 향후 클러스터 서버로 가장 보편적으로 사용될 가능성이 높다고 판단했기 때문이다.

본 논문의 구성은 다음과 같다. 제2장에서 단일 시스템 이미지와 단일 입출력 공간에 대해 고찰하며, 단일 입출력 공간을 지원하는 기존 연구들을 살펴본다. 제3장에서는 구현된 가상 하드 디스크 드라이버의 설계와 주요 기능을 기술하며, 제4장에서는 구현된 시스템의 성능을 평가한다. 제5장에서는 결론 및 향후 연구 과제를 제시한다.

2. 단일 입출력 공간 관련 연구

단일 입출력 공간(Single I/O Space)은 클러스터 시스템에서 개별 노드에 장착된 입출력 장치 또는 저장 장치의 물리적인 위치에 대한 지식 없이 어떤 노드에서도 접근 가능하게 하는 것으로 단일 시스템 이미지의 기본 목적을 달성하기 위한 필수적인 기능이다. 디스크의 경우, 단일 입출력 공간은 사용자의 관점에서 그림 1과 같은 단일 시스템 이미지 런타임 서비스를 제공함으로써 지역 디스크와 원격 디스크에 대한 접근의 차이를 제거하며, 디스크에 저장될 데이타를 클러스터 시스템의 각 노드로 분산시킴으로써 노드간 데이타 이동을 위한 입출력 연산의 성능을 향상시킨다. 단일 입출력 공간을 지원하려는 시도들을 접근 방식에 따라 분류하면 다음과 같다[1,3].

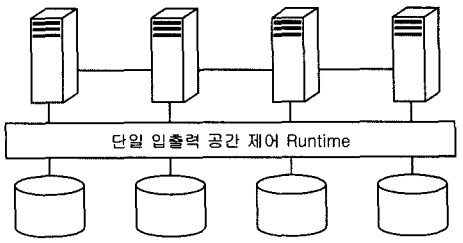

그림 1 단일 입출력 공간을 지원하는 클러스터

2.1 사용자 수준 접근

사용자 수준에서 단일 시스템 이미지 서비스를 구현하는 것은 저 수준의 접근과 비교하여 몇 가지 장단점을 갖는다. 사용자 수준 접근은 상대적으로 그 구현 비용이 적으며 설계, 구현, 적용 시 오류를 발견하기 쉽다. 또한, 구현된 시스템은 저 수준에서 구현된 시스템보다 쉽게 타 시스템으로 이식될 수 있다. 그러나, 사용자가 서비스를 제공받기 위해서 별도의 노드 구별자나 API를 사용해야 하므로 사용자 수준 접근은 완전한 단일 시스템 이미지 서비스를 제공한다고 볼 수 없다. 그리고, 사용자 수준 접근에서 필연적으로 사용하게 되는, 커널이 제공하는 파일 입출력 서비스와 네트워크 서비스는 클러스터 시스템을 위해 최적화되지 않을 수 있다. 더욱이, 그러한 서비스를 수행하기 위한 시스템 호출은 전체 시스템의 성능을 제한한다. 사용자 수준 단일 입출력 공간의 예로 Clemson 대학의 PVFS[4]와 Argonne 국립 연구소의 RIO[5] 등이 있다.

2.2 파일 시스템 수준 접근

파일 시스템 수준에서의 구현은 클러스터 시스템을 위해 파일 시스템을 최적화하고 파일의 접근과 관련하여 분산 패턴을 제어함으로써 사용자에게 완전한 단일 시스템 이미지 서비스를 제공할 수 있다. 그러나, 새로운 파일 시스템을 설계, 구현, 적용하는 것은 비용이 크고, 기존 시스템 또는 프로그램과의 엄격한 호환성을 제공하기 어렵다. 파일 시스템 수준에서 단일 입출력 공간의 예로 U. C. Berkeley의 xFS[6] 등이 있다.

2.3 디바이스 드라이버 수준 접근

디바이스 드라이버 수준 접근은 파일 시스템을 수정하지 않기 때문에, 디스크 입출력 연산에 대해 사용자뿐만 아니라 파일 시스템에게도 단일 시스템 이미지를 보여준다. 그리고 수정되지 않은 파일 시스템은 기존의 응용 프로그램과 호환성을 유지한다. 또한, 디바이스 드라이버 수준으로 제한된 접근은 설계, 구현, 적용을 용

이하게 하여 개발비용을 절감할 수 있다. 그러나, 디바이스 드라이버 수준에서는 입출력 연산의 최적화를 위하여 파일의 분산 패턴을 파일 시스템의 도움 없이는 제어할 수 없다. 디바이스 드라이버 수준에서 단일 입출력 공간의 예로 SIOS[1]와 DEC의 Petal[2] 등이 있다. SIOS와 Petal에서는 각 노드들에 장착되어 있는 하드 디스크들을 디바이스 드라이버 수준에서 하나로 묶어 마치 클러스터 전체에 하나의 디스크가 장착되어 있는 것처럼 보이도록 해준다. 이렇게 할 경우 본 논문의 VHDD와 마찬가지로 파일 시스템을 수정하지 않고도 단일 디스크 공간을 지원할 수는 있으나 전형적인 클라이언트/서버 구조가 되어 클러스터에 참여한 노드에서는 로컬 디스크를 사용하지 못한다는 단점이 있다. 본 논문의 VHDD에서는 클러스터에 참여한 노드는 공유 디스크 서버도 될 수 있음과 동시에 로컬 액세스도 가능하다는 점이 다른 점이라 하겠다.

2.4 네트워크 파일 시스템 서비스를 통한 디스크 공유

통신 인프라가 갖추어져 있지 않았던 시기의 운영 체제는 단일(stand-alone) 시스템의 보조 기억 장치를 위한 파일 시스템을 기반으로 플로피 디스크 등 이동 가능한 저장 매체를 통해 파일을 복사하는 방법으로 공유가 이루어졌다. 이후, UUCP, 직렬 포트 등을 통한 파일 전송이 가능해졌으나, 파일에 대한 동시 다중 접근을 허용하거나 실시간으로 파일의 수정을 반영하기 위해서 네트워크 환경의 분산 파일 시스템 구축이 필수적으로 요구되었다. 현재 많이 사용되고 있는 네트워크 기반 디스크 공유 서비스의 예를 들면 다음과 같다.

2.4.1 지역 네트워크에서의 파일 공유

• Samba

SMB(server message block)는 IETF에 의해 CIFS (common internet file system)라는 이름으로 표준화를 이루어가고 있는 Samba의 핵심 프로토콜이다. SMB는 전형적인 클라이언트-서버, 요청-응답 프로토콜 구조를 지니며, 파일 시스템뿐만 아니라 프린터, 직렬 포트 등의 각종 자원을 공유할 수 있게 한다. 또한, SMB는 TCP/IP, NetBEUI, IPX/SPX 등의 상이한 다중 네트워크 프로토콜 상에서도 원활하게 동작할 수 있다. Samba는 SMB 프로토콜을 구현한 서버 소프트웨어로서, SMB를 지원하는 클라이언트는 Samba를 구동 시키는 서버의 파일 시스템과 프린터 등의 자원에 쉽게 접근할 수 있다[7,8].

• NFS(Network File System)

NFS는 클라이언트-서버 구조를 가지고 있으며, UDP 프로토콜을 이용한 전형적인 Stateless 파일 시스템이다. NFS 프로토콜은 UDP를 이용하기 때문에 시스템의 부하를 줄이며, 파일에 관련된 각종 정보를 서비스 요청이 있을 때마다 반복적으로 보내어 파일의 현재 상태를 기억할 필요가 없도록 함으로써 파일 전송 시에 발생할 수 있는 손실을 최소화 시킨다. NFS는 현재 모든 내부 프로토콜 구조가 공개되어 있으며, 표준화된 RPC를 사용하고 하드웨어 독립적인 정보 교환을 위해 XDR (external data representation)이라고 불리는 데이타 정의 언어를 사용하는 등 거의 표준화가 된 여러 가지 구조적 특징들을 바탕으로 만들어졌다. Samba와는 달리 NFS는 파일 시스템의 공유에 국한되어 있으며, 클라이언트나 서버의 운영에 있어서 보다 유연함을 제공하기 때문에, 클라이언트는 원격 서버의 파일 시스템을 투명하게 사용할 수 있게 된다[7,9].

2.4.2 대규모 네트워크에서의 파일 공유

• NFS3

NFS3은 고속의 시스템과 대규모의 네트워크 환경에 적합하도록 설계되었다. NFS3은 파일의 옵셋이 이전 버전의 2배로 늘어나 최대 파일 크기의 제약을 완화시켰으며 파일의 전송 크기가 증가되어 네트워크 및 I/O 대역폭을 최대한 활용할 수 있게 하고 Write 연산 시에 완화된 조건의 새로운 Commit 규정(safe asynchronous writes)을 도입함으로써 효율을 향상시켰다. 또한, 신뢰성 있는 TCP를 사용하여 클라이언트 측의 구현을 간소화시켜 속도를 향상시켰다[7,9].

• AFS(Andrew File System)

AFS는 전형적인 분산 파일 시스템으로써 복잡한 여러 특성을 지니고 있다. AFS는 파일 작업이 RPC로 이루어지며 XDR을 사용하고 최상위 계층에 가상 파일시스템을 운영한다는 점에서 NFS와 유사하다. 그러나, AFS는 파일 시스템을 관리하는 서버측 외에 클라이언트측의 메모리와 디스크에 의한 캐싱을 제공하여 네트웍 부하를 줄이고 확장성을 향상시켰다. AFS 클라이언트의 캐쉬는 항상 서버의 내용과 일관되게 유지하기 위하여 콜백(callback) 메커니즘을 이용한다. 콜백 협약은 클라이언트가 서버에서 파일을 가져갈 경우에 성립되며, 서버는 그 파일의 변경 시에 협약을 취소하게 되고, 클라이언트는 캐싱된 파일 블록 이용 시 항상 콜백 협약을 확인한다. 이외에도, AFS는 파일 시스템의 위치 투명성, 높은 보안성 및 복구 능력, 사용 편의성, 시스템 관리 편의성 등을 제공한다[7,10,11].

• DFS(Distributed File System)

DFS는 AFS를 기반으로 만들어져 클라이언트 캐싱, 통일된 디렉터리 구조, 파일 복제 등 많은 점에서 AFS와 유사하다. 그러나, DFS에서는 콜백 메커니즘 대신 접근 토큰(access token)을 사용하여 클라이언트의 캐쉬를 관리한다. 접근 토큰을 가진 클라이언트는 특정 파일에 대한 배타적 변경(exclusive write)을 보장받는데,

동시에 여러 클라이언트가 변경을 요구할 경우에도 그 변경이 겹치지 않는다면 모두 허용한다[7,12].

3. VHDD 설계 및 구현

VHDD는 실제 디스크를 소유하고 있는 서버 노드의 커널 모듈과 공유를 원하는 클라이언트 노드의 가상 디스크 드라이버의 집합이다. 클라이언트 모듈은 원격지에 존재하는 디스크를 실제로 소유한 것과 같이 보일 수 있도록 하기 위한 디바이스 드라이버로서, 클라이언트는 이를 통해 지역 디스크와 동일하게 가상 디스크를 접근할 수 있다. 즉, 클러스터 내 사용자의 관점에서 가상 디스크조차도 디바이스 파일을 마운트하여 파일 시스템에 의해 접근된다는 점에서 투명성을 제공한다. 이와 같은 특징은 디바이스 드라이버 수준에서 접근하기 위한 시도의 Petal 디바이스 드라이버와 아주 유사한 면이 있다. 그러나, 클라이언트로부터 RPC에 의해 서비스가 요구되고 사용자 수준(user-level) 프로세스가 유닉스 raw 디스크 인터페이스를 이용하여 물리 디스크에 접근하는 Petal 서버와 달리, VHDD 서버에서는 디바이스 드라이버와 커널 모듈 간의 통신으로 클라이언트로부터 위치 정보를 받은 커널 스레드가 버퍼 캐쉬를 통해 디스크에 접근함으로써 캐싱의 이점을 살리게 된다. 아울러, 본 논문에서는 Petal에서 제시하지 않았던 클라이언트 캐쉬의 일관성 유지에 관한 문제에 공유 그룹별 멀티캐스팅 기법을 도입하여 해결하고자 하였다. 그림 2는 전체 시스템의 내부 구조와 데이타 및 제어의 주요 흐름을 보여준다.

3.1 클라이언트 모듈

VHDD 클라이언트의 동작은 읽기/쓰기 등의 요구를 서버로 전송하는 요구 함수(request function), 서버로 전송된 요구에 대한 결과를 수신하여 처리하는 RSCC (remote system call client) 데몬, 캐쉬의 일관성 유지와 관련된 MReceiver(multicast receiver) 데몬과 Flusher 데몬 등에 의해 이루어진다. VHDD 클라이언트 모듈에서 사용되는 기본 디바이스 구조체는 그림 3과 같다. vhd_drive_s 구조체는 현 구조체가 정보를 내포하고 있는 가상 드라이브의 이름 *name*, 가상 드라이브를 소유하고 있는 클라이언트의 IP 주소 *iaddr*, 서버의 공유 디스크에 대한 읽기/쓰기 등의 요구와 그 결과를 송수신하기 위한 소켓 *sock_request*, RSCC 데몬에 대한 정보 *rsc_client*, 현재 처리 중인 요구 *rq*와 그 요구를 VHDD 시스템 내부 형식으로 변환한 요구 *vhd_rq*,

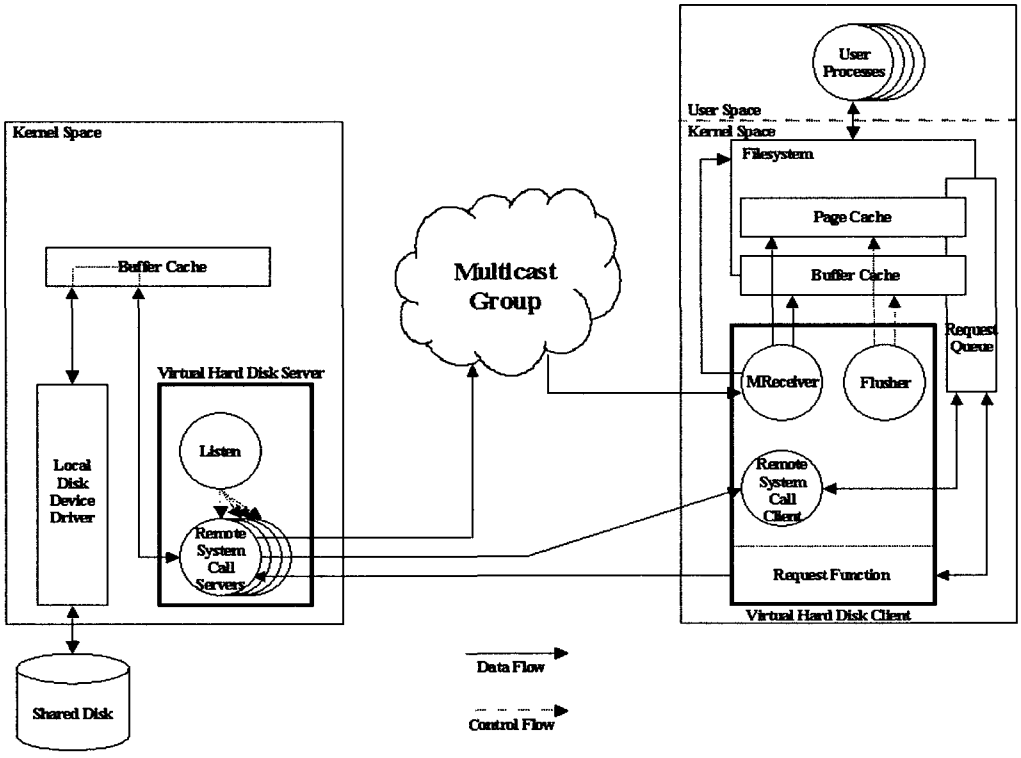

그림 2 전체 시스템의 구조

```
struct vhd_drive_s {
        char name[5];                       /* 가상 드라이브의 이름 */
        __u32 iaddr;                        /* 클라이언트의 IP 주소 */
        struct socket *sock_request;        /* 요구 또는 결과 송수신 소켓 */
        vhd_kthread_t rsc_client;           /* RSCC 데몬에 대한 정보 */
        struct request *rq;                 /* 현재 처리 중인 요구 */
        vhd_request_t vhd_rq;   /* 시스템 내부 형식의 현재 처리 중인 요구 */
        struct sockaddr_in sa_in_mcast;     /* 멀티캐스트 주소와 포트 번호 */
        vhd_kthread_t mreceiver;            /* MReceiver 데몬에 대한 정보 */
        vhd_kthread_t flusher[MAX_PARTN];/* Flusher 데몬에 대한 정보 */
        struct hd_struct *part;             /* 파티션 정보 */
        atomic_t usage;                     /* 디바이스의 사용 수 */
};
```

그림 3 클라이언트 모듈의 기본 디바이스 구조체

```
struct block_device_operations vhd_fops = {
open:           vhd_open,
release:        vhd_release,
};
```

그림 4 클라이언트가 지원하는 블록 디바이스 연산들

적절한 멀티캐스트 그룹에 참여하기 위한 멀티캐스트 주소와 포트 번호 *sa_in_mcast*, MReceiver 데몬에 대한 정보 *mreceiver*, Flusher 데몬에 대한 정보 *flusher*, 디바이스의 파티션 정보 *part*, 디바이스의 사용 수 *usage*로 구성된다.

VHDD 클라이언트는 그림 4와 같은 블록 디바이스 연산들을 지원한다. 기본 연산으로 읽기와 쓰기가 지원되지 않는 것은 VHDD를 적용한 시스템의 경우에도 기존의 리눅스 커널에서와 같이 입출력 버퍼링을 하며 입출력 서브시스템의 상위 레벨은 해당 파일시스템 연산이 담당하여 버퍼링의 혜택을 그대로 유지하기 때문이다.

VHDD 클라이언트의 open 연산에서는 먼저 서버와의 통신 소켓을 생성한 후 서버와 연결을 확립한다. 연결이 확립되면 서버로부터 공유 디스크의 각 파티션에서 기본 블록의 크기, 각 파티션의 크기, 멀티캐스트 주소와 포트 번호 등의 정보를 수신하여 디바이스를 초기화한다. 특히, 멀티캐스트 주소와 포트 번호는 동일한 디스크를 공유하는 클라이언트들을 동일한 멀티캐스트 그룹에 참여시킴으로써 쓰기 연산 시 동일 그룹 내 클라이언트들의 캐쉬 일관성 유지에 이용한다. 디바이스 초기화와 더불어 RSCC, MReceiver, Flusher 등의 데몬들 초기화도 open 연산에서 이루어진다. Flusher 데몬의 초기화를 제외한 일련의 과정은 공유의 기본 단위가 파티션이 아닌 디바이스이기 때문에 디바이스가 처음 열릴 때만 필요하며, 그 이후에는 기존의 정보와 자원을 사용하게 된다. 따라서 open 연산에서는 디바이스가 열릴 때마다 요구되는 파티션에 상관없이 가상 디바이스에 대한 사용계수를 유지하여 이미 열려진 경우라면 단지 디바이스의 사용계수만을 증가시킨다. **VHDD 클라이언트의 release 연산**에서도 open 연산과 유사하게 디바이스의 사용계수를 검사하여 디바이스를 마지막으로 닫을 경우에만 자원과 RSCC, MReceiver 데몬들을 해제하며, 그렇지 않다면 단지 요구된 파티션에 해당하는 Flusher 데몬을 해제하고 그 파티션과 연관된 캐쉬 내용에 대해 flushing 작업을 수행한 후 디바이스 사용계수를 감소시킨다.

일단 디바이스가 열리면 파일시스템을 통해 들어오는 I/O 요구들이 요구 큐에 쌓이게 되는데, 여기에 쌓여진 요구는 하위 계층 핸들러, 즉 **요구 함수**에 의해 소비된다. 요구 함수는 하나의 요구에 연결된 버퍼들을 한번의 수행에 하나씩 처리하며, 요구 큐를 비울 때까지 이 과정을 반복한다. 요구 함수는 처리할 요구에 대해 먼저 유효성 검사를 한 후, 그 요구를 그림 5와 같은 VHDD 시스템 내부 형식의 요구로 변환시킨다. 변환된 요구는 서버로 전송되고, 이후의 제어는 RSCC 데몬으로 넘어

```
struct vhd_request_s {
        __u32 iaddr;                /* 요구자 클라이언트의 IP 주소 */
        int cmd;                    /* 요구 연산 */
        int minor;                  /* 공유 디스크 내에서의 논리적 부번호 */
        unsigned long ino;          /* 요구 블록을 소유하고 있는 inode의 번호 */
        unsigned long offset;       /* inode가 포함하는 전체 파일 주소 공간 내에서
                                       요구 블록의 바이트 단위 옵셋 */
        unsigned long blocknr;      /* 파티션 내에서 요구 블록의 논리 블록 번호 */
        int blocksize;              /* 요구 블록의 크기 */
        int result;                 /* 요구 처리 결과 또는 오류 메시지 */
};
```

그림 5 시스템 내부 형식의 요구 구조체

가며 요구 함수는 세마포어를 사용해, RSCC 데몬이 서버로 전송된 요구의 결과를 수신하고 이를 처리할 때까지 대기하게 된다. RSCC 데몬으로부터 제어를 넘겨받은 후에는 더 이상 처리할 요구가 없을 때까지 위의 과정을 반복 수행한다.

RSCC 데몬은 가상 디바이스 단위로 존재하며 디바이스가 처음 열릴 때 초기화되어 수면 상태로 대기한다. 요구 함수에 의해 깨어난 RSCC 데몬은 먼저 종료 플래그를 검사하며, 계속 수행일 경우 현재의 요구에 대한 서버의 응답을 대기, 수신하여 이를 처리한 후, 제어를 요구 함수로 넘기고 다시 수면 상태로 전이한다.

3.2 서버 모듈

VHDD 서버의 동작은 클라이언트의 연결 요청을 대기·처리하는 Listen 데몬, 클라이언트들의 요구를 처리하는 RSCS(remote system call server) 데몬에 의해 이루어진다. VHDD 서버의 기본 구조체는 그림 6과 같다. VHDD 서버는 초기화시 클라이언트들의 접속 요청을 대기하도록 **Listen 데몬**을 생성한다. Listen 데몬은 클라이언트로부터 연결 요청이 오면 그 클라이언트에 대한 서비스를 담당할 새로운 RSCS 데몬을 생성하고 연결 리스트로 이를 삽입한다.

RSCS 데몬은 개별 클라이언트에 대해 공유되는 디스크 단위로 존재하며, 새로이 생성되는 데몬은 대응하는 클라이언트로 초기화를 위한 정보를 전송하고, 이 후 클라이언트로부터의 요구를 수신하여 이를 처리하며 클라이언트로부터 release가 요구될 때까지, 즉 공유 디스크 내에서 클라이언트에 의해 마운트 되었던 모든 파티션들이 언마운트 될 때까지 존재하게 된다. RSCS 데몬은 클라이언트들의 입출력 요구에 대해 파일 데이타와 메타데이타를 구별하지 않고 모두 버퍼 캐쉬(buffer cache)를 이용하여 처리한다. 이는 서버 측에서 공유가 결정된 디스크에 대해 실제 클라이언트들에 의해 공유되는 동안에는 서버에서의 마운트가 없음을 전제로 한다.

VHDD 시스템에서 단일 서버 내 복수 개의 디스크를 공유할 수도 있는데, 모듈 내에서 이들은 포트 번호에 의해 구별되며, 결과적으로 클라이언트 측에서 서버의 IP 주소와 포트 번호는 서버에 대해 공유를 요청하기 위한 최소한의 구별 조건이 된다.

3.3 VHDD와 캐쉬의 일관성

VHDD 시스템 내에 존재하는 캐쉬는 실제 공유될 디스크를 지역적으로 소유하고 있는 서버의 캐쉬와 그 디스크를 공유하는, 즉 가상의 지역 디스크를 소유하고 있는 클라이언트들의 캐쉬로 구성된다. 이들 캐쉬는 전체 시스템 내에서 전역적으로 사용되거나 관리되지 않는다. 또한, 가상 지역 디스크에 대해 입출력이 이루어지는 클라이언트에서조차도 캐쉬는 전형적인 지역 디스크 기반의 시스템 캐쉬와 동일한 방식으로 사용되며, 관리는 전적으로 커널에게 위임된다. 따라서, 이들 지역 캐쉬간의 일관성을 유지하기 위한 방안이 필수적으로 마련되어야 한다. VHDD에서는 이를 위한 기법으로 멀티캐스팅(multicasting)을 사용한다. VHDD 멀티캐스팅 기법은 크게 3단계의 과정을 거치게 된다. 제1단계는 클라이언트들의 멀티캐스트 그룹 참여이다. 하나의 디스크를 공유하는 다수의 클라이언트들은 동일한 멀티캐스트 그룹에 참여하게 되는데, 이것은 클라이언트에서 그 디바이스를 처음 열 때 서버로부터 해당 디바이스에 대한 멀티캐스트 그룹에 참여하기 위한 멀티캐스트 주소와 포트 번호를 넘겨받아 행해지며, 서버는 단일 디스크에 대해 공유하고자 하는 모든 클라이언트에게 동일한 멀티캐스트 주소와 포트 번호를 알림으로써 그들을 동일한 그룹에 참여하게 할 수 있다. 제2단계는 서버의 멀티캐스팅이다. 클라이언트는 상위 레벨로부터의 입출력 요구를 서버로 전달하는데, 서버는 이 가운데 쓰기 요구에 대해서만 동일한 내용을 멀티캐스트 그룹으로 멀티캐스팅한다. 제3단계는 클라이언트측 MReceiver 데몬의 멀티캐스트 메시지 처리이다. 서버에 의해 멀티캐스팅된 쓰기 요구 메시지는 그 요구를 서버로 전달했던 클라이언트를 포함한 그룹 내 모든 클라이언트의 MReceiver 데몬에 의해 수신되어 그 내용에 따라 적절한 캐쉬 블록에 대해 쓰기가 수행된다. MReceiver 데몬은 수신된 메시지가 현재 마운트되지 않은 영역에 대한 요구이거나 수신된 메시지의 처음 송신자가 자신일 경우 그 메시지를 무시한다. 또한 쓰기 요구된 블록이 현재 캐쉬에 존재하지 않을 경우에도 메시지가 무시된다.

```
struct vhd_server_s {
        vhd_kthread_t rsc_listen;          /* Listen 데몬에 대한 정보 */
        struct list_head rsc_server;       /* RSCS 데몬들의 연결 리스트 */
        struct sockaddr_in sa_in_mcast;    /* 서버가 허가한 디스크를 공유하는 클라이언트들을
                                              동일한 그룹으로 연결시켜줄 멀티캐스트 주소와
                                              포트 번호 */
        struct socket *sock_msender;       /* 해당 디스크의 공유 그룹상으로의 멀티캐스트를
                                              위한 소켓 */
        int *blocksize;                    /* 해당 디스크의 각 파티션에서 기본 블록의 크기 */
};
```

그림 6 VHDD 서버의 기본 구조체

멀티캐스팅 기법에서는, 임의의 클라이언트 캐쉬 내의 모든 블록들에 대해 다른 노드의 프로세스들이 쓰기 연산을 수행하기 전에 물리 디스크로 flush되는 것을 보장하는 SIOS의 *simple data consistency model*과는 대조적으로, 클라이언트의 쓰기 연산 요구가 서버와 동시에 공유 그룹 내의 모든 노드들에게도 전달되어 멀티캐스팅으로 인한 네트워크 부하는 증가하였으나 시스템의 다운 등으로 발생할 수 있는 데이타의 손실을 최소화하였다. 또한, 멀티캐스팅된 쓰기 요구가 서버를 포함하여 요구된 블록을 실제 캐싱하고 있는 노드들에서만 수행되도록 함으로써 각 노드가 수행해야 할 작업의 처리량을 감소시켰다.

또한 VHDD에서는 위와 같은 멀티캐스트 기법을 보완하기 위하여 각각의 클라이언트에 Flusher 데몬을 도입한다. 일반적으로 쓰기 요구는 즉시 디스크로 적용되기보다는 캐쉬를 이용하여 행해지는 것이 보통이다. 따라서 쓰기 요구가 발생한 시점과 실제 디스크로 적용되는 시점과는 약간의 차이를 두게 된다. VHDD에서도 이와 같은 메커니즘으로 쓰기 연산이 수행되기 때문에, 요구 함수에 의해 상위 레벨의 요구를 서버로 전달하는 시점을 가상의 지역 디스크에 쓰기 요구를 적용하는 시점으로 본다면 시점의 차이로 인한 클라이언트들 간의 캐쉬 불일치 가능성이 있음을 알 수 있다. 이러한 이유로 클라이언트에는 마운트된 파티션 단위로 Flusher 데몬이 존재하며, 이들은 단지 매초 수면 상태에서 깨어나 자신에게 할당된 파티션과 연관된 캐쉬에 대해서 flushing 작업을 수행한다. 이렇게 함으로써 앞서 언급한 두 시점의 차이를 최소화하게 된다.

지금까지의 논의의 초점은 개별 노드들이 동일한 블록을 캐싱하고 이에 대해 쓰기 연산을 수행할 경우의 캐쉬 일관성 유지 문제였다. 그런데, VHDD 시스템은 현재 단일 노드 내에서도 이와 유사한 문제를 안고 있다. 리눅스 커널은 디렉터리 엔트리(dentry)나 inode 등의 메타 데이타 객체들을 관리함에 있어서 효율성을 높이고자 버퍼 캐쉬와는 별도로 각각의 객체들로 구성된 캐쉬를 사용한다. 따라서, VHDD 시스템 내 임의의 클라이언트가 쓰기 연산을 수행한다면 그 클라이언트와 동일한 멀티캐스트 그룹에 참여하고 있는 타 클라이언트의 MReceiver 데몬이 멀티캐스팅된 쓰기 요구 메시지를 버퍼 캐쉬에 적용하게 되는데, 이 때 요구 블록의 일부 또는 전체가 메타 데이타 캐쉬에 보관되어 있을 가능성이 상존하기 때문에 두 캐쉬간에는 불일치의 가능성이 항상 존재하게 된다. 리눅스에서 사용되는 일반적인 디스크 기반의 파일시스템은 분산, 공유 등의 특수한 환경을 목적으로 개발되지 않은 범용의 파일시스템으로써, 단지 단일 노드의 효율성 제고를 위하여 캐쉬 시스템이 사용되기 때문에 캐쉬와 디스크간 flushing도 또한 '메타 데이타 캐쉬→버퍼 캐쉬→디스크'의 순방향 flushing만을 갖출 필요가 있는 것이다. 결론적으로, 앞서 언급된 일련의 조치들은 전체 시스템 내 캐쉬의 일관성을 100% 보장한다기보다는 캐쉬의 불일치 가능성을 최소화하기 위한 조치들이라 하겠다.

3.4 서버 및 클라이언트 측 VHDD 동작 예

서버 측과 클라이언트 측에서 VHDD가 각각 어떻게 동작하는지에 대한 실 예를 보이기 위해, 이번 절에서는 사용자가 마운트 등의 명령을 입력했을 때 실행 순서에 따라 서버 측과 클라이언트 측이 어떻게 상호 작용하는지를 살펴보도록 한다. 그림 7은 이 과정을 보이고 있는데, 서버 측에서 서버용 VHDD 모듈을 적재하면(①-1) Listen 데몬인 listend가 시작되고(①-2) listend는 클라이언트로부터의 요청을 기다린다. 마찬가지로 클라이언트 측에서 클라이언트 용 VHDD 모듈을 적재하면(②-

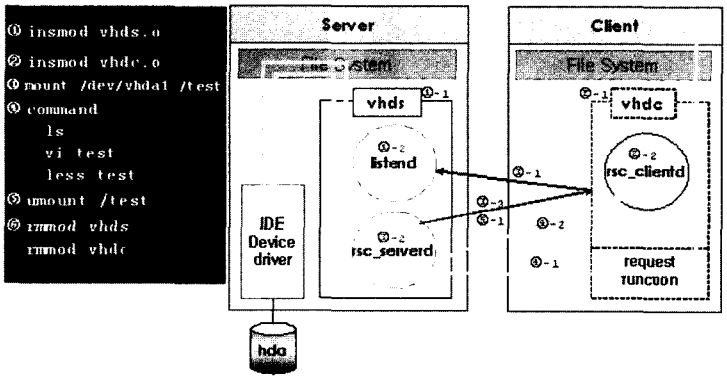

그림 7 서버와 클라이언트 측에서의 VHDD 동작 예

1) RSCC 데몬인 rsc_clientd가 동작을 시작한다(②-2). 아직 두 시스템의 VHDD들 간에는 연결이 설정되지 않은 상태이다. 이 때 클라이언트 측에서 원격 디스크를 마운트하기 위해 "mount /dev/vhdd1 /test" 명령을 입력하면 rsc_clientd는 listend에게 마운트를 요청하고(③-1) listend는 실제 서비스를 담당할 RSCS 데몬인 rsc_serverd를 띄운다(③-2). 마운트 요청의 처리 결과는 rsc_serverd가 rsc_clientd에게 직접 전달한다(③-3). 이제 클라이언트는 마운트된 파티션이 원격 파티션인지 아닌지 구별할 필요 없이 원하는 명령을 실행할 수 있다. 클라이언트가 입력한 명령들은 클라이언트 측 VHDD의 요구 함수를 거쳐 서버 측의 rsc_serverd로 전달되며(④-1), 서버 측 rsc_serverd는 결국 로컬 디스크 드라이버를 사용해 서비스를 완료한 후 결과를 클라이언트 측의 rsc_clientd에게 전달한다(④-2). 언마운트도 역시 클라이언트의 요청으로 시작하는데(⑤-1), 마운트의 반대 과정을 거치게 된다.

4. 성능 평가

4.1 성능 평가 환경

VHDD의 성능 측정은 그림 8에서 볼 수 있듯이 3-node 클러스터 서버에서 이루어졌다.

클러스터 서버의 운영 체제로는 커널 버전 2.4.13의 리눅스를 사용하였으며, 각 노드에 있는 지역 디스크의 루트 파일시스템으로는 ext2 파일시스템을 사용하였다. 그리고 클러스터 서버 내에서 임의의 한 노드를 선택, 공유 디스크로 사용될 40GB의 하드 디스크를 장착하여 나머지 노드들의 수와 같은 2개의 10GB 주 파티션(primary partition)을 생성하고 각각 ext2 파일시스템을 설치하였다. 나머지 노드들은 공유 디스크 내의 이 두 파티션을 각각 마운트한다. 위와 같이 설정된 시스템에서 클라이언트 노드 수 1대, 2대의 경우에 대해 각각 VHDD 클라이언트의 성능을 측정하였으며, VHDD 성능의 비교 대상으로서 동일 조건 하에서 NFS 버전2 클라이언트의 성능을 측정하고 임의의 클라이언트 노드에 공유 디스크로 사용된 디스크를 장착하여 로컬 디스크 성능을 측정하였다. 여기서 서버 노드의 공유 디스크와 VHDD 클라이언트 노드의 가상 디스크의 미리 읽기(read ahead)는 8KB이고, NFS 서버는 전형적인 설정인 UDP를 통하여 마운트되며, 클라이언트-서버 간에 주고받는 읽기/쓰기 블록의 크기 역시 디폴트 값인 1KB를 각각 사용하였다.

성능 측정 프로그램으로는 IOzone[13]을 사용하였으며, 실시된 실험은 다음과 같다. 각 실험은 공통적으로 최소 64KB에서 2배씩 증가하면서 최대 64MB까지의 파일 각각에 대해, 레코드 크기는 최소 4KB에서 2배씩 증가하면서 최대 16MB까지의 모든 경우에 대해 수행하되 32MB 이상의 파일에 대해서는 64KB 이상의 레코드에 대해서만 수행하였다. Read 실험은 기존 파일의 읽기 성능을 측정하고, Write 실험은 새로운 파일의 쓰기 성능을 측정한다. 마지막으로, 위의 실험에서 얻어진 측정값들을 파일 크기별 산술 평균으로 단순화한다.

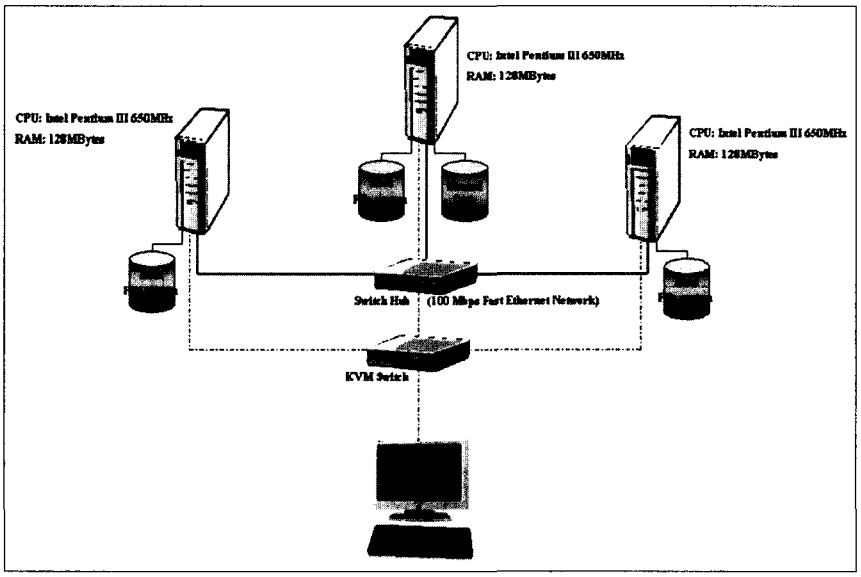

그림 8 3-node 클러스터 서버

4.2 결 과

4.2.1 Read

그림 9와 그림 10은 다양한 크기의 파일에 대해 각 시스템의 Read 성능을 클라이언트 노드수에 변화를 주며 측정한 결과이다. 각각의 그림은 전반적인 Read 연산 수행에 있어서 VHDD 클라이언트의 성능이 NFS 클라이언트의 그것에 준함을 보여준다. 또한, 클라이언트의 노드수가 증가해도 성능이 크게 떨어지지 않음을 볼 수 있다. 두 그래프에서 VHDD 클라이언트의 성능 패턴이 클라이언트 노드 내에서 측정한 지역 디스크의 성능 패턴과 유사한 것은 동일한 파일 시스템을 사용함에서 오는 결과라 할 수 있다.

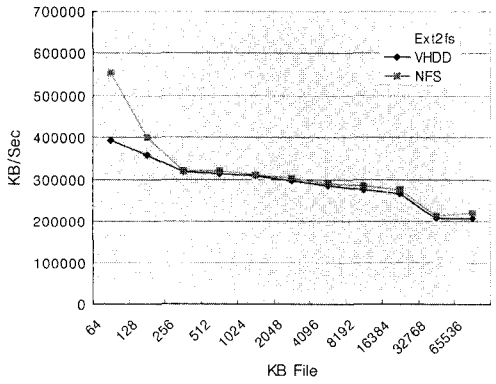

그림 9 Read 성능(클라이언트 노드 수 = 1)

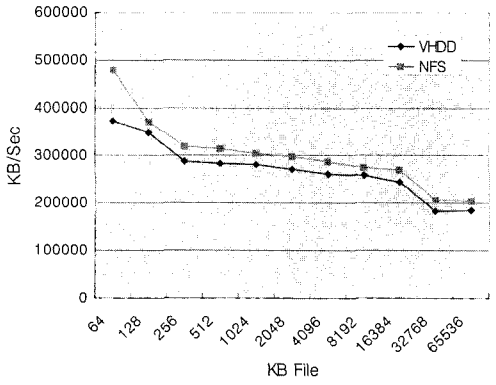

그림 10 Read 성능(클라이언트 노드 수 = 2)

4.2.2 Write

그림 11과 그림 12는 다양한 크기의 파일에 대해 각 시스템의 Write 성능을 클라이언트 노드 수에 변화를 주며 측정한 결과이다. 그림 11은 전반적인 Write 연산 수행에 있어서 32MB 이하의 파일에 대해 VHDD 클라이언트의 성능이 NFS 클라이언트의 그것보다 월등히 높음을 보여준다. 하지만 64MB 이상의 파일에서는

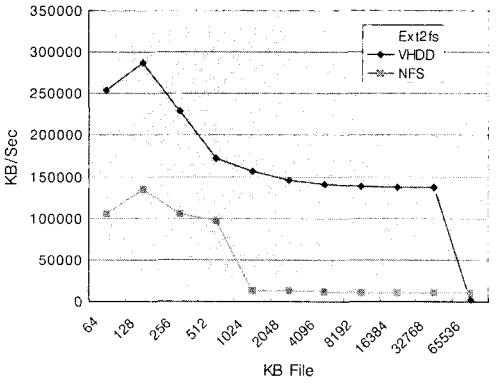

그림 11 Write 성능(클라이언트 노드 수 = 1)

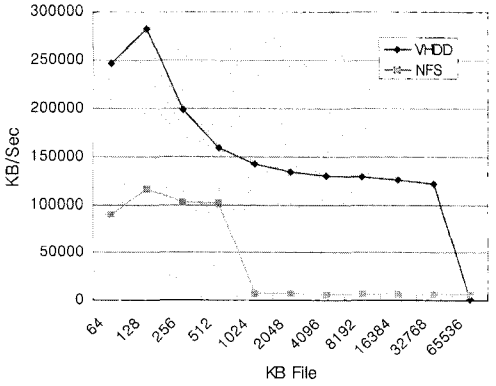

그림 12 Write 성능(클라이언트 노드 수 = 2)

VHDD 클라이언트의 성능이 급격히 떨어지는 것을 볼 수 있는데 이는 VHDD의 한계라기보다는 리눅스 커널의 한계에 기인한다고 할 수 있다. 리눅스 커널 내에서는 지연된 쓰기 정책을 유지하면서 오염 버퍼 비율이 전체 메모리의 40%를 넘어서면 이를 깨끗한 상태로 만들기 위한 작업을 수행하기 때문에, 전체 메모리 크기의 50%를 차지하는 64MB 파일에서 그 성능이 급격히 하락하게 되는 것이다. 반면, NFS 클라이언트의 성능이 1024KB 이상의 파일에서 급격히 떨어지는 이유는 NFS 프로토콜 자체의 한계성에 기인한다. NFS에서는 per-inode 요구의 수가 MAX_REQUEST_SOFT (764KB) 보다 크면 그 inode에 대해 현재 보류(pending)된 모든 쓰기 연산들이 스케줄되고 이것이 완료될 때까지 현재 요구가 대기하게 된다. 그런데, 현재 실험상에서는 단일 파일, 다시 말해, 단일 inode에 대해서만 쓰기 연산이 요구되었으므로 실험에 사용된 파일의 크기가 MAX_REQUEST_SOFT를 넘어서는 1024 KB 파일에서 급격히 성능이 하락하여 그 기조를 유지하게 된다. 그림 12에서 VHDD의 성능 패턴이 지역 디스크 성능 패턴과 유사한 것은 역시 Read에서와 같은 이유 때문인 것으로

판단된다.

5. 결론 및 향후 연구 과제

본 논문에서 제시한 가상 하드 디스크 드라이버는 리눅스 클러스터 환경에서 원격 노드에 존재하는 하드 디스크를 가상의 지역 하드 디스크로 다루고 있다. 여기서 하드 디스크로 국한한 이유는 하드 디스크가 사용 빈도가 높은 만큼 전체 시스템의 성능에 많은 영향을 주기 때문이다. 이러한 구현은 원격 디스크와 지역 디스크에 대한 위치 투명성을 주어 클러스터 시스템 사용자에게 단일 입출력 공간을 제공하며, 입출력과 관련된 하위 시스템을 파일 시스템과 가상 디스크의 집합으로 분리시킴으로써 상위 계층에서의 접근 방식에서 제시된 문제점들을 해결한다. 이와 같은 방식으로 본 논문에서 구현된 시스템은 중, 소규모 파일에 대해 NFS와 유사한 Read 성능과 NFS보다 월등한 Write 성능을 보여주었다. 그러나, 대규모 파일에 대한 성능은 개선이 요구되며, 특히 전반적인 성능이 동일한 파일시스템을 사용한 지역 디스크의 성능과 그 패턴이 유사함은 주목할 부분이다.

향후 본 논문에서 제시한 가상 하드 디스크 드라이버에 대해 캐쉬 시스템의 일관성 문제를 개선하고, 다양한 파일시스템과 입출력 패턴을 적용하여 이에 대한 프로파일링(profiling)을 통해 시스템의 문제점을 분석하여 이를 제어하고 최적화하는 것이 연구되어야 할 사항이다. 특히, 확장성(scalability)을 증대시키기 위해 노드 추가 시 발생하는 시스템 불안정성을 해소하고, 편리하게 노드를 추가/삭제하기 위한 GUI 도구 등을 개발해야 할 것이다.

참 고 문 헌

[1] Roy S. C. Ho, K. Hwang, and H. Jin, Single I/O Space for Scalable Cluster Computing, Proceedings of the 1st IEEE Computer Society International Workshop on Cluster Computing, pp.158-166, 1999.

[2] E. K. Lee and C. A. Thekkath, Petal: Distributed Virtual Disks, Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, pp.84-92, 1996.

[3] G. F. Pfister, In Search of Clusters, 2nd Ed., Prentice Hall, 1998.

[4] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, PVFS: A Parallel File System For Linux Clusters, Proceedings of the 4th Annual Linux Showcase and Conference, pp.317-327, 2000.

[5] I. Foster, D. Kohr Jr., R. Krishnaiyer, and J. Mogill, Remote I/O: Fast Access to Distant Storage, Proceedings of the 5th Annual Workshop on I/O in Parallel and Distributed Systems, pp.14-25, 1997.

[6] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang, Serverless Network File Systems, ACM Transactions on Computer Systems, Vol.14, No.1, pp.41-79, 1996.

[7] B. Callaghan, NFS Illustrated, 1st Ed., Addison Wesley, 2000.

[8] R. Sharpe, Just what is SMB?, http://samba.anu.edu.au/cifs/docs/what-is-smb.html

[9] The NFS™ Distributed File Service, Sun Microsystems, Inc., 1995.

[10] 박재호, 인터넷에서 파일을 공유하자!, WWW-KR 워크샵, 1997.

[11] 분산 파일시스템의 이용, http://dpnm.postech.ac.kr/posco-project/dfs-intro.html

[12] File Systems in a Distributed Computing Environment, Open Software Foundation, 1991.

[13] W. D. Norcott and D. Capps, Iozone Filesystem Benchmark.

김 태 호

2000년 한림대학교 정보통신공학부(학사). 2003년 한림대학교 컴퓨터공학과(석사). 2003년~현재 ㈜티컴앤디티비로 부설연구소 연구원. 관심분야는 임베디드 시스템, 멀티미디어, 유비쿼터스 컴퓨팅

이 종 우

1990년 서울대 컴퓨터공학과 졸업(학사) 1992년 서울대 컴퓨터공학과 석사과정 졸업(석사). 1996년 서울대 컴퓨터공학과 박사과정 졸업(박사). 1996년~1998년 현대전자산업(주) 과장. 1998년~1999년 현대정보기술(주) 책임연구원. 1999년~2002년 한림대학교 정보통신공학부 조교수. 2002년~2003년 광운대학교 컴퓨터공학과 조교수. 2003년~현재 아이닉스소프트(주) 개발이사. 관심분야는 운영체제, 병렬/분산 운영체제, 클러스터 시스템, 전산금융

이 재 원

1990년 2월 서울대학교 컴퓨터공학과 학사. 1992년 2월 서울대학교 컴퓨터공학과 석사. 1998년 8월 서울대학교 컴퓨터공학과 박사. 1999년~현재 성신여자대학교 컴퓨터정보학부 강의전담교수. 관심분야는 기계학습, 전산금융, 자연언어처리

김 성 동

1991년 2월 서울대학교 컴퓨터공학과 학사. 1993년 2월 서울대학교 컴퓨터공학과 석사. 1999년 8월 서울대학교 컴퓨터공학과 박사. 1999년 8월~2001년 2월 서울대학교 컴퓨터신기술공동연구소 특별연구원. 1999년 8월~현재 한성대학교 컴퓨터공학부 전임강사. 관심분야는 데이타마이닝, 기계학습, 자연언어처리

채 진 석

1990년 2월 서울대학교 컴퓨터공학과 학사. 1992년 2월 서울대학교 컴퓨터공학과 석사. 1998년 2월 서울대학교 컴퓨터공학과 박사. 1992년 3월~1997년 2월 서울대학교 공학연구소 조교. 1997년 7월~1998년 8월 한국학술진흥재단 부설 첨단학술정보센터 선임연구원. 1998년 8월~현재 인천대학교 컴퓨터공학과 조교수. 관심분야는 인터넷 소프트웨어, 마크업 언어, 한국어정보처리