Fips : 파일 접근 유형을 고려한 동적 파일 선반입 기법

(Fips : Dynamic File Prefetching Scheme based on File Access Patterns)

이 윤 영 † 김 재 열 †† 서 대 화 †††

(Yoon-Young Lee) (Chei-Yol Kim) (Dae-Wha Seo)

요 약 병렬 파일시스템은 클러스터 시스템에서 과도한 입출력 요청을 원활하게 지원하기 위해 사용되며, 특히 파일 선반입은 병렬 파일시스템의 성능을 개선하는데 유용하게 사용된다. 본 논문은 과학계산용 병렬 응용과 멀티미디어 서버 응용에서 효과적인 파일 접근 유형을 고려한 새로운 동적 파일 선반입 기법인 Fips를 제안한다. 본 논문이 제안하는 동적 파일 선반입 기법인 Fips는 파일의 접근 유형을 고려하여 동적으로 선반입 할 데이타 블록을 예측하고, 다양한 접근 유형에서도 데이타 블록의 선반입을 효율을 높였다. 그리고 현재의 가용 대역폭을 고려하여 선반입 시기를 결정하므로 선반입이 시스템에 과부하로 작용하는 것을 방지하도록 하였다. 병렬 파일시스템에 Fips를 적용하여 실험한 결과 다양한 작업부하에서 제안한 선반입 기법은 우수한 성능을 보여주었다.

키워드 : 파일선반입, 캐싱, 병렬파일 시스템, 운영체제

Abstract A parallel file system is normally used to support excessive file requests from parallel applications in a cluster system, whereas prefetching is useful for improving the file system performance. This paper proposes a new prefetching method, Fips(dynamic File Prefetching Scheme based on file access patterms), that is particularly suitable for parallel scientific applications and multimedia web services in a parallel file system. The proposed prefetching method introduces a dynamic prefetching scheme to predict data blocks precisely in run-time although the file access patterns are irregular. In addition, it includes an algorithm to determine whether and when the prefetching is performed using the current available I/O bandwidth. Experimental results confirmed that the use of the proposed prefetching policy in a parallel file system produced a higher file system performance.

Key words : file prefetching , caching , parallel file system , operating system

1. 서 론

CPU의 처리속도가 급속하게 향상되는 것에 비해 입출력 장치의 처리속도는 이를 따라가지 못하고 있으며 입출력 장치는 컴퓨터 시스템의 병목이 되고 있다[1].

병렬 파일시스템을 도입하여 파일의 입출력 대역폭은 상당한 향상되었으나 여전히 파일 요청 시의 디스크 접근 지연시간을 줄일 수는 없다. 이를 해결하기 위해, 이전에 사용된 데이타 블록 중 다음에 사용될 확률이 높은 데이타 블록들을 메모리에 남겨두는 캐싱 기법과 다음에 사용될 것으로 예측되는 블록을 미리 캐쉬 메모리로 읽어 들여 파일 서비스 요청 시 지연시간을 최소화하는 선반입 기법이 사용된다.

본 논문에서 제안하는 선반입 기법은 과학계산용 병렬 응용과 멀티미디어 서버 응용을 대상으로 하고 있다. 이들의 파일 접근 유형을 살펴보면, 파일 크기가 대부분

· 본 연구는 한국과학재단 목적기초연구(과제번호 : R01-2001-00333) 지원으로 수행되었음.

† 학생회원 : 넷컴스토리지
yylee@palgong.knu.ac.kr
†† 비 회 원 : 한국전자통신연구원 리눅스연구팀
gauri@etri.re.kr
††† 종신회원 : 경북대학교 공과대학 전자전기공학부 교수
dwseo@ee.knu.ac.kr

논문접수 : 2001년 4월 10일
심사완료 : 2002년 4월 4일

일반적인 버퍼 캐쉬 전체의 크기보다 크며, 한번에 요구하는 데이타의 양 또한 일반적인 파일시스템에 비해 크다[2,3]. 따라서 응용프로그램이 한번 수행되는 동안에 캐쉬 영역에 보관된 데이타 블록들은 새로 읽어온 데이타 블록으로 교체되어서, 다음에 같은 프로그램을 수행할 때에는 데이타 블록을 다시 디스크에서 읽게되어 캐슁의 효과를 볼 수 없게 된다. 하나의 응용프로그램 내에서도, 일반적인 파일의 접근 유형을 살펴보면, 하나의 데이타 블록을 반복적으로 사용하는 경우가 많지 않으므로 지역성에 의존하는 캐슁은 효과가 없다. 그러나 선반입 기법을 이용하면, 실제로 데이타 블록이 필요한 순간에 디스크에 접근하지 않고 주메모리 공간의 버퍼 캐쉬를 접근하여 데이타 블록을 가져올 수 있으므로 파일시스템의 성능을 향상시킬 수 있다.

선반입에 관한 다양한 연구들이 있었으나 대부분이 다음에 선반입할 대상 블록을 예측하는 알고리즘에 관한 것들이었다. 하지만 실제 파일시스템 성능을 감안하면 파일 요청이 과도한 경우에는 불필요한 블록을 읽을 가능성이 있는 선반입을 수행하지 않는 것이 성능에 유리한 경우도 있으므로, 시스템의 상태를 고려하여 선반입 여부를 결정할 수 있는 정책 또한 필요하다.

따라서, 본 논문에서는 가장 최근 두 번의 읽기 요청된 파일 데이타 블록에 대한 정보 - 요청한 데이타 블록의 개수, 두 데이타 블록간의 간격 -를 비교해서 접근 유형을 찾아내고, 이를 바탕으로 다음에 사용될 블록을 동적으로 예측하는 알고리즘을 소개한다. 그리고 수행 중인 응용프로그램들의 각 파일에 대한 단위 시간당 데이타 요구량을 이용해서 현재 사용 가능한 입출력 대역폭을 계산하고 이를 기준으로 파일 선반입 여부와 시기를 결정하는 *Fips(dynamic File Prefetching Scheme based on file access patterns)*를 새로운 선반입 기법으로 제안한다.

2. 배경 연구

전통적인 UNIX 플랫폼에서의 파일 접근 유형에 관한 연구들이 많이 이루어져 왔으며, 이러한 연구에 의해 일반적인 UNIX 파일시스템[9]과 Sprite 분산 파일시스템[10]에서의 대부분의 접근은 읽기이며 쓰기는 그렇게 많지 않고, 작고 일정한 크기의 접근이 주로 발생하는 것으로 밝혀졌다. 4.2 BSD UNIX 파일시스템에서의 읽기 접근이 차지하는 비율은 70% 정도로 쓰기에 비해 두 배 이상이었고 한 번에 접근하는 크기는 70-75%가 4000byte 이하였다. 또한 90%이상의 접근이 순차적인 유형이었으며, 파일을 한번 열어서 처음부터 끝까지 접근하는 경우도 70%에 가까웠다.

병렬 처리 방법을 사용하는 대용량의 데이타 입출력이 필요한 응용의 대표적인 예인 과학기술계산 분야의 파일 입출력 유형은 simple-strided 유형이 많은 것으로 조사되었다[3]. Simple-strided 유형은 일정한 크기의 접근과 역시 일정한 크기의 lseek가 교대로 일어나는 유형으로서 클러스터 시스템 등에서 병렬 처리 기법을 사용하여 대용량의 파일을 접근할 때 나타나는 접근 유형이다.

파일 입출력이 과중한 병렬 응용프로그램에 관한 Pablo 프로젝트[2]의 연구결과에 따르면 이러한 응용프로그램들의 접근 유형은 읽기와 쓰기에서 동일한 크기의 접근이 반복적으로 발생하는 경우가 대부분이었고, 그 원인을 같은 데이타 입출력 라이브러리를 사용하기 때문이라고 밝히고 있다.

이와 같은 기존의 접근 유형에 관한 연구 결과에 의해 대부분의 일반적인 응용프로그램들은 데이타 파일을 접근하는 유형이 일정한 크기의 읽기와 일정한 거리의 lseek을 반복하는 규칙적인 특성을 가지는 것을 알 수 있다.

접근 유형분석의 결과를 바탕으로 다양한 선반입 기법에 관한 다양한 연구들도 수행되었다. OBL(One Block Lookahead)[11]과 IBL(Infinite Blocks Lookahead)[11]등의 선반입 정책은 응용프로그램의 접근 유형이 순차적이라고 가정하고 요청한 블록의 다음 블록을 읽어오는 고정된 정책을 사용하였다. 고정된 정책을 사용하는 방법은 적용된 캐슁 및 선반입 정책이 목표로 하는 순차적 접근 유형이 주를 이루는 환경에서는 뛰어난 성능을 보이지만, 여러 종류의 접근 유형이 혼재하는 상황에서는 원하는 성능을 얻을 수 없다.

ISG(Interval-and-Size Graph) 알고리즘은 데이타 압축을 기반으로 하는 알고리즘을 간단하게 한 것으로써, 응용프로그램은 대부분 정규적인 형태로 파일을 액세스한다는 이론을 바탕으로 한 알고리즘이다[6]. 이 알고리즘은 현재 요청된 데이타 블록의 시작점(offset)과 다음에 사용될 데이타 블록과의 거리(interval) 그리고 사용된 데이타 블록의 크기(size)에 대한 정보를 그래프 형태로 유지하고 있다. 따라서 이 그래프가 형성된 파일에 대한 액세스가 이루어 질 경우 이 그래프를 이용하여 다음에 사용될 데이타 블록을 요청할 수가 있게 된다. 하지만, 그래프 생성과정에서 알 수 있듯이 다음에 사용될 데이타 블록을 예측하기 위한 그래프를 형성하기 위해서는 일단 한 번의 파일 액세스가 이루어져 블록 예측을 위한 그래프가 완성된 다음에 가능하다는 단

점이 있다. 또한 액세스되는 패턴의 형태가 정규적이지 않고 복잡한 경우라면 예측을 위한 그래프를 생성하는 데 걸리는 시간도 길어지고 유지해야 하는 그래프의 크기도 늘어날 것이다.

힌트기반 선반입 기법[12]은 파일시스템보다는 응용프로그램이 파일 접근 유형을 더 잘 알고 있다는 사실에 근거하고 있다. 즉, 파일 접근 유형을 응용프로그램이 파일시스템에 알려주면, 그 정보를 가지고 파일시스템은 효과적인 선반입을 수행하는 것이다. 그러나, 응용프로그램 작성 시에 접근 유형을 명시하여야 하므로, 기존 프로그램의 이식성이 떨어지게 된다. 또한, 프로그래머는 자신이 작성하는 응용프로그램의 저수준 접근 유형을 알고 있거나 예측할 수 있는 루틴을 고안해야만 하고, 표준적의 파일 입출력 API를 사용하지 않고 특별한 API를 사용해야 하기 때문에 응용프로그램 작성의 어려움이 크다.

이처럼 다양한 선반입에 관한 연구가 있었지만, 이들은 단지 다음에 사용될 데이타 블록을 예측하는 것에만 초점을 두고 있다. 그러나 응용프로그램에서 데이타의 요청이 폭주할 경우, 선반입을 위해서 실제로 사용되지 않을 수도 있는 블록을 읽는 데에 시간을 소비하여, 당장 필요한 데이타 블록의 요청이 지연되는 경우도 발생할 수 있다. 그러므로 선반입을 하기에 앞서서 입출력 상황을 파악하여 선반입을 할지 여부를 결정하는 정책에 관한 연구 또한 필요하다.

따라서 본 논문에서는 실행 시간(run-time)에 파일 접근 유형이 변경되는 것을 감지하고 변화된 접근 유형에 적응하여 데이타 블록을 미리 읽어올 수 있는 동적인 선반입 데이타 블록 예측방식을 제안함과 동시에 시스템의 입출력 요청 상태를 반영하여 선반입 여부와 시기를 적절히 결정하는 정책을 함께 제안한다.

3. 병렬 파일시스템

본 논문에서 제안하는 선반입 기법인 Fips는 리눅스 클러스터에서 운용중인 PFSL (Parallel File System for Linux clusters)을 기반으로 하였다[8]. PFSL은 분산된 입출력 하드웨어를 최대로 활용하는 것을 목적으로 한다. PFSL은 다수의 Linux 시스템이 컴퓨팅 노드 혹은 입출력 노드로 동작하면서 하나의 논리적인 병렬 파일시스템을 형성한다.

PFSL은 크게 클러스터 파일 매니저, 블록 매니저, 그리고 메타데이타 매니저로 구성된다(그림 1). 클러스터 파일 매니저는 컴퓨팅 노드에서 응용프로그램이 요청하는 파일 입출력 서비스를 담당하고, 블록 매니저는 입출력 노드들에 분산 저장된 파일의 데이타 블록의 입출력 서비스를 담당한다. 메타데이타 매니저는 분산 저장된 파일의 메타데이타를 관리하는 서버이다.

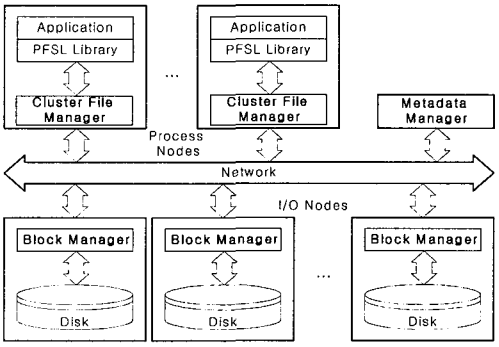


그림 1 PFSL의 구성

PFSL에서는 응용프로그램에서 요구한 데이타 블록의 입출력 시간을 줄이기 위해서 향상된 선반입 기법인 Fips가 도입되었으며, 일반적인 캐싱도 병행하여 사용하고 있다. 그림 2는 PFSL이 응용프로그램에서 요청을 받아 캐싱과 Fips를 수행하는 과정을 간단히 보여준다.

응용프로그램(application process)은 파일 입출력 서비스를 요청하고, 여기서 발생된 파일 입출력 요청들은 선반입 관리자와 연결된 큐로 들어가게 된다.

선반입 관리자(Prefetching manager)는 정해진 선반입 정책에 따라서 Fips를 이용해 다음에 사용될 블록을 예측하여 선반입을 수행하는 모듈이다. 여기서 선반입 정책은 시스템 부하 상태나 응용의 특성에 따라 결정되어 진다. 정해진 선반입 정책에 의해서 선반입 관리자는 입출력 요청을 선반입 요청 큐와 파일 요청 큐에 분산 연결한다. 즉, 실제 입력 요청과 선반입 데이타를 동시에 요청할 수도 있고 각각 따로 요청할 수도 있다.

입출력 관리자(I/O manager)는 상위의 선반입 요청 큐와 파일 요청 큐에 저장된 요청을 하나씩 처리한다. I/O 요청이 들어있는 큐는 선반입 요청만이 들어있는 큐보다 우선순위가 높다. 결과적으로 I/O 요청 큐에 대기중인 작업이 없는 경우에만 선반입 큐의 작업을 수행하게 된다. 요청된 블록은 먼저 캐쉬에서 찾게 되며 캐쉬에 없는 블록은 네트웍을 통해 I/O 노드의 블록서버에 요청한다.

캐쉬 관리자(Cache manager)는 버퍼 캐쉬를 유지하는 역할을 한다. I/O manager의 요청을 받으면 요청한 블록의 캐싱 유무를 알려주고 요청된 블록으로 LRU 정책에 따라 캐쉬 블록을 교체한다.

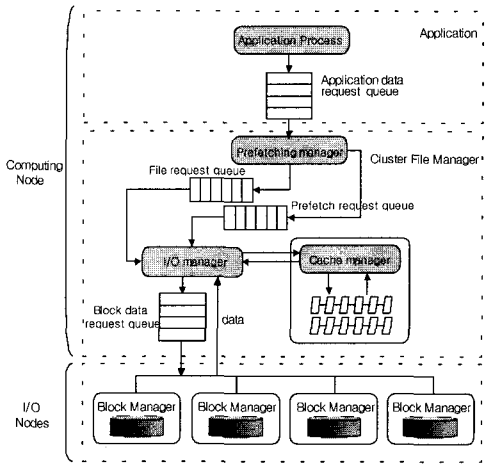


그림 2 PFSL에서 파일 캐싱과 선반입

4. Fips(dynamic File Prefetching Scheme based on access patterns)

기존의 파일 블록 선반입 연구들은 주로 다음에 사용될 데이타 블록을 예측하는 알고리즘들에 대한 것들이었다. 이들은 대부분 각 파일이나 데이타 블록의 요청 순서를 기록하고, 이러한 기록들을 바탕으로 다음에 요청될 파일이나 블록을 예측하는 방법을 사용하였다. 그러나 이러한 방법들은 이전의 파일 접근 기록을 유지하기 위해 필연적으로 복잡한 알고리즘과 데이타 정보를 가지게 된다. 하지만 이러한 정보들을 유지하고 이를 이용해 블록을 예측하기 위해서 사용하는 알고리즘들의 부하를 무시할 수 없으며, 과학 계산용 병렬 응용프로그램의 파일 접근 유형에 대해서는 복잡한 블록 예측 방법이 필요 없다. 따라서 본 논문에서 제안하는 선반입 기법인 Fips는 동적 블록 예측 알고리즘으로 간단하면서도 과학계산용 병렬 응용과 멀티미디어 서버 응용의 파일 접근 유형에서 효과적인 동적 블록 예측 알고리즘을 고안하였다. 이는 기존 선반입 알고리즘과 같이 데이타 예측정보를 유지 관리하는 것이 아니라 실시간으로 간단한 테이블을 이용하여 정보를 갱신해가면서 바로 다음 블록을 예측하는 방식이라 별도의 부하가 발생하지 않는다.

기존 선반입 기법들의 또 하나의 문제점은 현재의 데이타 블록 요청 상황 즉, 현재 얼마나 많은 파일 입출력 요청이 있는가를 고려하지 않는다는 점이다. 다시 말해, 선반입할 데이타 블록을 결정한 후, 현재의 파일시스템의 I/O 부하 정도에 상관없이 선반입을 수행한다는 것이다. 그런데 파일의 읽기/쓰기 요청이 폭주할 경우에는, 곧 사용되어진다고 보장할 수 없는 블록들을 선반입을 하는 것이 시스템의 성능을 저하시키는 요인이 된다. 이러한 문제점을 해결하기 위해서는 현재의 입출력 부하를 파악할 수 있는 방법과 이를 선반입 정책에 반영하는 알고리즘이 필요하다.

본 논문에서 제시한 Fips에서는 현재의 입출력 부하를 파악하는 방법으로 현재 사용 가능한 입출력 대역폭, 즉 '가용 입출력 대역폭'을 사용한다.

Fips는 크게 3개의 부분으로 구분된다(그림 3). 첫 번째는 데이타 블록 입출력 요청 관련 정보를 기록하고, 이를 바탕으로 다음에 요청될 데이타 블록을 예측하는 선반입 데이타 예측 단계이며, 두 번째는 입출력 부하를 파악하기 위해 가용 입출력 대역폭을 계산하는 단계, 마지막으로 얻어진 가용 입출력 대역폭을 이용하여 실제 선반입 여부와 선반입 시기를 결정하는 단계이다.

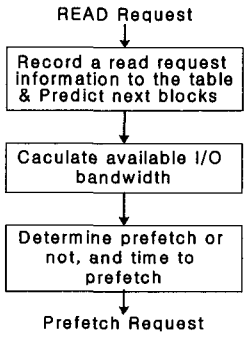


그림 3 Fips 수행 과정

4.1 Fips의 선반입 수행 과정

Fips에서, 파일시스템은 응용프로그램으로부터 파일 열기 명령이 들어오면 각 파일에 대해서 읽기 요청 시의 정보를 기록할 수 있는 동일한 구조체 형식의 테이블 두 개를 할당한다. 테이블의 데이타 구조는 다음과 같다.

```
struct  pre_table {
    int request_size          요청한 블록의 개수
    int request_interval      이전 요청과 현재 요청 사이
                              의 블록 간격
    int request_latency       이전 요청과 현재 요청 사이
                              의 시간 간격
    int start_block_number    현재 요청의 첫 블록 번호
```

응용프로그램으로부터 읽기 요청이 들어올 때마다 테이블에 요청 데이타 블록의 크기, 이전 요청과의 블록 간격, 이전 요청과의 시간 간격과 현재 요청의 블록 번호를 기록한다. 따라서 최근 읽기 요청된 두개의 데이타 블록에 대한 정보가 두 개의 테이블에 기록되어진다. 읽기 요청이 들어오면 요청에 관한 정보를 테이블에 기록하고 두 개의 테이블을 비교해 요청 크기와 요청 간격이 같은 경우 다음 블록을 예측할 수 있다. 그림 4는 파일 접근에 따른 테이블 갱신의 예를 보여 주고 있다.

선반입할 데이타 블록을 예측한 후에는 현재의 가용 입출력 대역폭을 계산한다. 가용 입출력 대역폭은 시스템이 제공할 수 있는 최대 입출력 대역폭에서 현재 응용프로그램들과 시스템이 사용하고 있는 수 있도록 했다. 계산된 가용 입출력 대역폭은 선반입 여부 혹은 선반입하는 시기를 결정하는 값으로 사용된다.

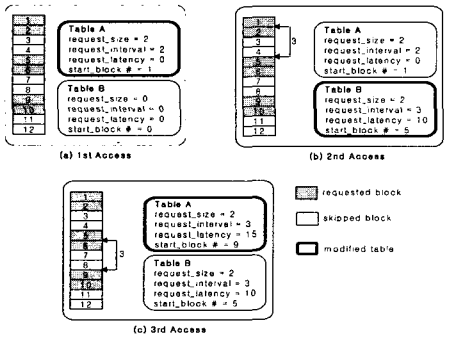


그림 4 파일 접근에 따른 테이블 갱신의 예

4.2 선반입 데이타 블록 예측

그림 5는 데이타 블록의 요청 간격과 요청 크기를 보여주고 있다. 빗금이 쳐진 블록은 응용프로그램이 읽기를 요청한 블록이며, 나머지 블록은 요청이 되지 않고 건너 뛴 블록이다. 요청 간격(interval)은 이전 요청의 마지막 블록과 현재 요청의 처음 블록과의 블록 차이를 말한다. 따라서 블록이 연속적으로(consecutive) 요청이 되는 경우는 요청 간격은 1이 된다. 요청 크기(size)는 요청하는 블록의 개수가 된다.

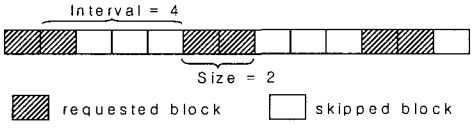


그림 5 요청 간격과 요청 크기

Fips는 ISG(Interval-and-Size Graph) 기법과 달리 테이블을 가지고 그래프를 형성하지는 않으며 두 개의 테이블을 번갈아 가면서 갱신하고 이를 비교하여 요청 간격과 요청 크기가 동일한 경우에는 다음과 같이 선반입할 데이타 블록을 예측할 수 있다.

- 선반입할 시작 블록 번호 : (*start_block_numer* + *request_size* - 1) + *request_interval*
- 선반입할 블록 개수 : *request_size*

4.3 가용 입출력 대역폭 계산

4.3.1 응용프로그램의 파일 요청 특성

일반적으로 병렬 응용프로그램에서는 메모리에 일정한 크기의 버퍼를 할당받고 이를 통하여 데이타 블록을 반복적으로 요청하여 연산 작업을 수행한다. 따라서 응용프로그램이 요구하는 데이타 블록의 크기는 병렬 응용프로그램의 파일 접근 유형에서 볼 수 있는 것과 같이 몇 개의 동일한 것들만이 나타나는 것이다. 그리고 하나의 파일 전체에 대해서 이런 동일한 크기의 데이타를 요청하지는 않지만 부분적으로는 이러한 형태를 보인다.

그림 6은 응용프로그램의 데이타 블록 요청이 시간적으로 어떻게 진행되는지를 보여준다. 요청된 데이타의 크기는 거의 일정하다. 이러한 파일 요청의 특성을 이용하여 본 논문에서는 전체 파일시스템에서 현재 사용 가능한 입출력 대역폭을 계산하여 이를 선반입 여부와 시기를 결정하는데 사용한다.

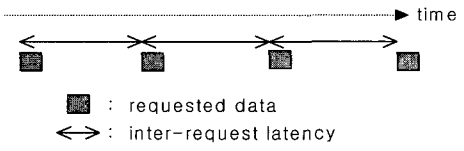


그림 6 응용프로그램의 데이타 요청

4.3.2 시간당 데이타 요구량

시스템에서 사용하고 있는 입출력 대역폭을 계산하기 위해서는 먼저 각 프로세스가 사용하고 있는 파일 각각의 시간당 데이타 요구량을 계산할 수 있어야 한다. 이렇게 계산된 각각의 요구 데이타 량을 합하면 전체 시스템이 요구하는 입출력 대역폭을 계산할 수 있다.

그림 7은 데이타 요구량 계산을 위해서 사용되는 정보를 나타내고 있다. 두 개의 값을 모두 반영하여 현재의 시간당 데이타 요구량을 계산할 수도 있다. 시간 t_{check}에서의 파일의 시간당 데이타 요구량은 S_3/τ_2 이다.

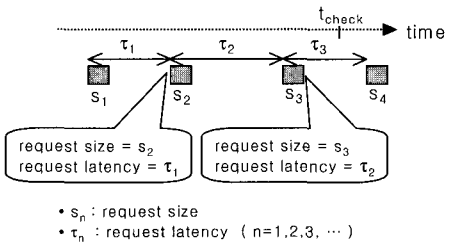


그림 7 데이타 요구량 계산

4.3.3 가용 입출력 대역폭

병렬 파일시스템의 입출력 성능이 가장 좋은 경우는 파일 블록들이 각 블록 매니저에 공평하게 스트라이핑 되어 있고, 클러스터 파일 매니저가 동시에 모든 블록 매니저로부터 데이타 블록을 읽어 들일 때이다. 이때 입출력의 병렬성이 최대가 되며, 네트웍에서 병목현상이 발생하지 않는다면, 하나의 노드에서 하나의 블록을 읽을 때 걸리는 시간으로 블록 매니저 개수만큼의 블록을 읽을 수 있다. 이렇게 파일을 외부에서 연속적으로 요청할 때 병렬 파일시스템은 최고의 입출력 성능을 보일 수 있으며, 이 때의 입출력 대역폭을 최대 입출력 대역폭(BW_{max})이라고 한다.

그리고, 현재 열려있는 모든 파일에 대해서 각각의 단위 시간당 데이타 요구량을 구하고 이들의 합을 구하면, 현재 시스템이 요구하는 입출력 대역폭을 구할 수 있는데 이것을 사용 입출력 대역폭(BW_{used})이라고 한다.

현재의 파일 요청 상황을 판단하는데 사용하는 가용 입출력 대역폭(BW_{avail})은 파일시스템의 최대 입출력 대역폭에서 현재 응용프로그램들이 요구하고 있는 사용 입출력 대역폭을 제외한 값으로 한다.

$$BW_{avail} = BW_{max} - BW_{used}$$

BW_{avail} 값이 선반입 여부를 결정하는 기준치 ($BW_{prefetch}$)보다 큰 경우에는 데이타 블록은 선반입을 선반입하게 된다. 선반입 여부를 결정하는 과정은 다음 절에서 상세하게 기술한다.

4.4 선반입 여부와 시기 결정

Fips는 매 입출력 요청에 대해서 그림 2의 선반입 관리자가 현재의 파일 요청 상황을 나타내는 가용 입출력 대역폭을 구하고 이를 바탕으로 선반입 수행 여부와 그 시기를 결정한다. 매 입출력 요청마다 연산이 요구되어 파일시스템의 성능이 저하될 수 있으나, 연산으로 인한 오버헤드가 불필요한 입출력을 수행하였을 때의 오버헤드와 비교하였을 때 무시할 수 있는 수준이었으며, 이는 실험결과를 통하여 확인할 수 있다.

선반입의 시기를 조정하기 위한 방법으로 클러스터 파일 매니저(cluster file manager) 내의 선반입 관리자(prefetch manager)가 입출력 관리자에게 요청을 전달해 주는 경로를 두 가지로 나누는 방법을 사용한다. 선반입 관리자와 입출력 관리자(I/O manager)를 연결하는 큐를 두 개 두고 각 큐의 우선 순위를 달리하여, 데이타 블록의 선반입 시기를 조절한다.

선반입 관리자는 응용프로그램으로부터 온 파일 요청을 기반으로 선반입할 데이타 블록을 결정한다. 이때 읽기 요청만을 선반입에 적용하며, 나머지 요청은 그대로 파일 요청 큐를 통해 입출력 관리자로 전달된다. 일반적으로 대부분의 파일시스템은 선반입 관리자 다음에 하나의 작업 큐만 가지고 있다. 이러한 파일 시스템에서의 선반입 관리자의 구조는 상당히 간단해 질 수 있지만, 뒤에서 기다리고 있는 파일 요청은 선반입 요청 때문에 서비스가 지연될 수 있다. 이러한 단점을 피하고자 제안하는 선반입 관리자는 입출력 관리자에게 요청을 선반입 요청 큐와 파일 요청 큐에 분산하여 요청하는 방법을 사용하였다.

파일 요청 큐는 각각 응용프로그램으로부터 온 파일 요청을 담당하며, 선반입 요청 큐는 선반입 관리자가 생성한 선반입 요청을 담당한다. 두 개의 큐는 서로 다른 우선 순위를 가지고 있어 선반입 시기를 조절할 수 있다. 즉 선반입 요청 큐보다 파일 요청 큐가 높은 우선순위를 가지고 있어, 큐 다음에 위치하는 입출력 관리자는 파일 요청 큐에 쌓인 작업이 없는 경우에만 선반입 요청 큐의 작업을 처리하게 된다.

그러나 선반입 큐를 따로 지정하는 경우 선반입 큐에서 선반입을 기다리고 있는 데이타 블록이 선반입 되기 전에 응용프로그램으로부터 호출되어 파일 요청 큐에 들어오는 경우이다. 이 경우가 있다. 파일 요청 큐가 우선순위가 높으므로 실제 응용프로그램의 파일 요청이 수행된 후에 동일한 블록이 다시 한번 선반입될 수 있다. 따라서, 선반입 요청 큐에 대기 중인 요청과 동일한 데이타 블록의 읽기 요청이 발생할 경우 선반입이 아직 실행되지 않았다면 선반입 요청 큐의 요청을 삭제한다. 그러나 선반입을 수행 중이라면 파일 요청을 큐에 넣지 않고 선반입 결과를 기다리게 한다.

이상과 같은 구조를 이용하여 선반입 관리자는 선반입의 시기를 조정할 수 있다. 그림 8은 데이타 블록의 선반입 여부와 시기를 결정하는 알고리즘이다.

①의 경우는 현재의 응용프로그램이 요구하는 입출력 대역폭이 시스템이 제공할 수 있는 능력을 넘어서는 경우이다. 이러한 경우에는 선반입이 오히려 시스템의 성

① if (BW_{avail} < 0)
    No Prefetching
② else if (BW_{avail} < $BW_{prefetch}$)
  if (request_interval == 1)
    Enqueue current read request and prefetching to
    the file request queue together
  else
    Enqueue prefetching request to the prefetching request queue
③ else
    Enqueue prefetching request to the prefetching request queue

그림 8 데이타 블록 선반입 여부 및 시기 결정 알고리즘

능을 떨어뜨릴 수 있으므로 선반입을 하지 않는다. 선반입은 현재의 파일 서비스를 방해하지 않는 범위에서 수행되어야 한다. 데이타 블록 요청이 폭주하는 상황에서 선반입을 수행하게 되면 시스템 전체의 파일 서비스에 지장을 초래할 수 있으므로 선반입을 하지 않는 것이다.

②의 경우에는 데이타 블록 요청이 폭주하는 것은 아니지만 가용 입출력 대역폭이 충분하지 않은 경우이다. 즉, BW_{avail} 이 $BW_{prefetch}$보다 작은 경우로 입출력 처리기가 응용이나 시스템이 요구하는 데이타 블록을 처리하기에 바쁜 상태를 의미한다. $BW_{prefetch}$는 시스템에 따라 상대적이므로, $\beta(0 \le \beta \le 1)$라는 튜닝 변수를 두고, $BW_{prefetch}$를 β x BW_{max}로 해서 시스템에 맞게 설정하도록 하였다. 본 논문에 적용된 시스템에서는 β를 1/2로 두고 실험하였다. 본 논문에서는 β=1/2라는 수치에 큰 의미를 부여하지 않았다. 단지 가용 입출력 대역폭의 많고 적음을 판단하기 위한 상징적인 기준으로 사용되었다.

request_interval = 1 이면 파일 요청이 연속적이라는 의미이며, 이 경우에는 연속적으로 현재 요청한 블록과 선반입할 블록을 동시에 읽을 수 있도록 우선 순위가 높은 파일 요청 큐에 요청을 삽입한다. 그 이유는 충분한 입출력 대역폭이 남아 있지 않은 상황에서는 가능한 한 파일시스템의 성능을 효과적으로 이용하기 위하여 연속된 블록은 한 번에 읽어오도록 하는 것이다. 이 때문에 뒤에서 기다리는 파일 요청 처리가 조금 지연될 수 있지만 전체 파일 서비스 시간을 감안한다면 성능에 도움이 된다. 그러나 *request_interval* ≠ 1 경우와 같이 선반입을 하기 위한 비용과 일반적인 파일 요청을 처리하는데 드는 비용이 비슷할 경우에는 바로 선반입을 하지 않고 선반입 요청을 선반입 큐에 넣어 입출력 요청이 없을 때 선반입을 수행하도록 한다.

③의 경우에는 입출력 대역폭이 충분히 남아있는 상황으로 판단하고 모든 선반입을 입출력 요청이 없을 때 수행하도록 선반입 큐에 넣어 둔다.

위와 같은 알고리즘으로 본 논문에서는 현재의 파일 시스템 상황을 고려하여 선반입 여부 및 시기를 결정하게 된다.

5. 실 험

5.1 시스템 구현

이 시스템의 구현은 리눅스 클러스터 환경에서 운용되는 병렬 파일 시스템인 PFSL에 그림 2와 같이 구현하였다. 제안된 선반입 기법은 C 언어를 이용하여 구현되었으며, 시스템 성능을 극대화하기 위해서 다중쓰래드 기법을 이용하였다. 이는 파일 서버에서의 읽기 수행과 비동기적인 수행을 가능하게 한다. 구현된 주요 부분은 다음과 같다.

- 선반입 정보의 초기화
  - 데이타 블록 캐쉬 크기
  - 최대 입출력 대역폭
  - 파일 요청 큐
  - 선반입 요청 큐
- 선반입 관리자 쓰래드 생성 및 종료
  - 파일 요청 큐 및 선반입 요청 큐 관리
  - 선반입 여부 및 시기 결정
- 입출력 관리자 쓰래드 생성 및 종료
  - 블록 서버에 데이타 블록 요청
  - 데이터 블록 요청 큐 관리
- 캐쉬 관리자 쓰래드 생성 및 종료
  - 데이타 블록 교체
  - 데이타 블록 큐 관리

그리고 모든 응용 프로그램에서 발생하는 파일 접근은 PFLS을 거쳐서 처리되도록 하였다. 이것은 각각의 입출력 부하에 따른 성능을 상대적으로 비교하기 위해서이다.

5.2. 테스트 베드 환경

테스트 베드는 4대의 노드로 구성되어 있다. 각 노드는 600MHz 펜티엄III 프로세서와 128MB의 메인 메모리, 두 개씩의 20GB의 IDE 하드디스크를 가지고 있다. 그리고 노드들은 100Mbps 이더넷 카드와 스위칭 허브를 이용하여 연결되었으며, 외부 트래픽의 영향을 배제시키기 위하여 테스트 베드만을 100Mbps 허브에 연결하였다. 운영체제는 Linux kernel version 2.2.12이고 그 위에 병렬화일시스템인 PFSL이 운용되어진다. 그리

고 성능시험 프로그램에서의 데이타 접근은 PFSL을 통해서만 가능하도록 하였다.

5.3 작업부하

실험에 사용된 작업부하는 병렬 응용프로그램의 파일 접근 특성을 분석한 연구 논문[3]의 결과를 사용하였다. 이 논문에서 제시한 sequential, interleaved, mixed의 3가지 유형을 주요 실험 대상으로 하고, 나머지는 임의의 접근 유형을 생성하도록 하여 가변적인 파일 접근 유형에 얼마나 잘 적응하는지를 실험하였다.

표 1는 실험에 사용된 작업부하를 나타낸다. 최대 데이타 요구량(maximun request rate)에 따라 LIGHT, MEDIUM, MAXIMUM으로 나누었다. 최대 데이타 요구량은 테스트 베드의 최대 입출력 대역폭을 기준으로 정하였으며, 실험에 사용된 시스템의 최대 입출력 대역폭은 약 32MB/s이다. 실험을 위해 작성된 테스트 프로그램은 작업부하에 따라 3~5개의 프로세스를 생성하고, 각 프로세스는 작업부하에 따라 데이타 블록을 요청한다.

표 1 실험에 사용된 작업부하

Workload / Characteristics	LIGHT	MEDIUM	HEAVY
Total read size (MB)	29.7	54.7	103.9
Number of total requests	700	1000	1700
Max. request rate (MB/s)	11.2	19.2	32
Avg. read size per request (kB)	48.4	56	62.6
Radom ratio (%)	13.2	23.8	3.76
Number of processes	3	4	5

6. 실험 결과

파일시스템의 성능을 측정하는 방법으로는 일반적으로 캐쉬의 적중률을 많이 사용한다. 하지만, 보다 실제적인 시스템의 성능을 검증하기 위해서는 총 입출력 시간을 보는 것이 타당하다. 왜냐하면 파일시스템에서는 파일의 블록을 읽어오기 전에 먼저 캐쉬에 버퍼 공간을 할당하고 나서 실제 데이타를 디스크로부터 읽어 오는데, 이 때 적중률은 실제적으로 데이타가 캐쉬에 들어오기 전이라도 버퍼가 할당되어 있다면 히트로 처리를 하기 때문이다. 또한 본 실험에서 사용한 파일 요청 형식은 각 요청들 사이의 상대적인 시간을 기준으로 작성이 되어 있기 때문에 실제 파일 서비스가 빨리 이루어질수록 이에 따르는 전체 읽기 시간도 줄어들므로 요청 당 평균 읽기 시간과 전체 읽기 시간을 파일시스템 성능 측정 방법으로 사용하는 것이 타당하다.

Fips의 성능을 평가하기 위해 기존에 개발된 몇 가지 선반입 기법들을 시스템에 적용하여 비교하였다.

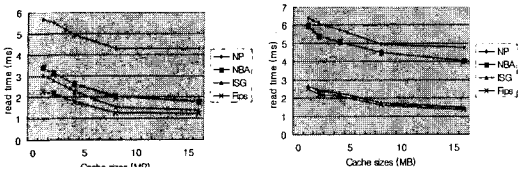
- NP : No prefetching
- NBA : N-Block Ahead
- ISG : Interval-and-Size Graph

다음의 결과는 주어진 작업부하 LIGHT, MEDIUM, HEAVY에 대해서 버퍼 캐쉬의 크기를 1MB에서 16MB까지 변화시켜 가면서 테스트 프로그램을 수행한 결과이다. 실험 결과로는 전체 읽기 시간을 전체 요청 개수로 나눈 요청 당 평균 읽기 시간을 기록하였다.

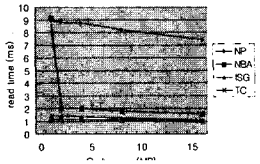
LIGHT 작업부하는 최고 데이타 요구율이 11.2MB/s이며, 이는 실험에 사용된 작업부하 중 가장 낮은 데이타 요구율이다. 실험 결과는 그림 8(a)과 표 2(a)와 같으며, 전체적으로 Fips가 다른 선반입 방식보다 우수한 결과를 보여준다. 이것은 나머지 선반입 기법과는 달리 두 개의 큐를 사용해 선반입 시기를 조절할 수 있기 때문이다. Fips는 LIGHT와 같이 요청이 빈번하지 않은 작업부하에서는 선반입 요청을 실제 응용프로그램의 파일 입출력 요청이 없는 경우에 수행함으로써 ISG에 비해 효율적으로 입출력 장치를 사용할 수 있다.

그림 8(b)와 표 2(b)는 MEDIUM 작업부하에 대한 실험 결과이다. 최대 입출력 대역폭의 절반 정도를 평균적으로 요청하는 MEDIUM의 경우에는 NP와 NBA에 비해서 ISG와 Fips가 월등한 성능 차를 나타내고 있다. 그러나 Fips가 LIGHT의 경우에 비해서 ISG에 비해 크게 향상되지는 않았지만 꾸준히 만족할만한 성능을 발휘하고 있다. NBA의 경우는 연속적인 파일 요청에 대해서 효과적이며 그 외의 strided 유형에 대해서는 상황에 따라서 성능이 차이가 난다. 이런 사실을 기준으로 보면 MEDIUM은 연속적인 파일 요청이 거의 없으며 strided 유형 또한 NBA가 효과를 볼 수 없는 형태로 이루어지고 있다고 생각할 수 있다. 이러한 이유로 NBA가 ISG와 Fips에 비해 훨씬 떨어지는 성능을 보인다.

그림 8(c)와 표 2(c)는 HEAVY 작업부하에 대한 실험 결과이다. HEAVY는 실험대상 작업부하 중 최대 데이타 요구량이 32MB로 가장 많으며 또한 요청하는 데이타의 전체 크기 또한 다른 작업부하에 비해 훨씬 크다. 전체적인 결과는 이전의 작업부하에 대한 실험결과와 비슷하다. 즉 Fips, ISG, NBA, NP의 순으로 우수한 성능을 보여주고 있다. 특이한 점은 NBA의 경우 메모리의 크기가 1MB에서 2MB로 바뀌는 경우에 급격한



(a) LIGHT (b) MEDIUM



(c) HEAVY

그림 8 요청 당 평균 읽기 시간

표 2 캐쉬 적중률

Cache size(MB)	NP(%)	NBA(%)	ISG(%)	Fips(%)
1	3.95	55.33	72.8	68.39
2	5.62	58.72	73.35	70.03
4	17.8	64.55	77.43	75.86
8	28.76	72.8	83.75	84.33
16	28.76	75.7	85.5	85.07

(a) LIGHT

Cache size(MB)	NP(%)	NBA(%)	ISG(%)	Fips(%)
1	3.45	10.51	63.74	64.21
2	7.72	16.99	68.52	68.23
4	12.5	22.4	71	71.16
8	23.2	29.96	79.47	79.69
16	26.1	34.84	84.23	84.17

(b) MEDIUM

Cache size(MB)	NP(%)	NBA(%)	ISG(%)	Fips(%)
1	0.6	1.62	93.54	79.23
2	1.84	88.29	93.6	80.08
4	2.45	89.12	94.07	82.11
8	7.52	90.24	94.49	83.51
16	13.68	91.24	96.03	85.23

(c) HEAVY

성능상의 변화를 보여주고 있는 점이다. 이는 이전의 실험결과에서는 나타나지 않았던 부분이다. 이러한 결과가 나타나는 원인은 이전의 작업부하에서는 읽은 데이타의 양이 1MB의 버퍼 캐쉬의 크기로도 어느 정도 감당할 수 있었던 반면, HEAVY의 경우에는 시간당 읽어 들이는 데이타의 양이 많고 특히 NBA의 경우 모든 읽기 요청에 대해서 선반입을 시도함으로써 다른 선반입 알고리즘에 비해 읽어들이는 데이타 양이 상대적으로 가장 많음으로서 선반입한 데이타가 읽히기 전에 버퍼 캐쉬에서 제거되는 경우가 발생할 수 있다. 이런 현상은 앞서의 작업부하에 대한 실험에서도 캐쉬 버퍼의 크기를 1MB 이하로 크게 줄이면 마찬가지로 발생하게 된다. 또 하나 주목할 만한 점은 ISG에 비해 Fips가 캐쉬 적중률이 떨어짐에도 불구하고 전체적인 성능은 더 우수하게 나왔다는 점으로 부하가 과중한 경우에는 선반입 자체가 부하로 작용할 수 있다는 것을 보여주며 이러한 경우에 대해서 현재의 파일 요청 상황을 고려한 테이블 비교 선반입 방식이 효과적임을 보여준다.

7. 결 론

병렬 파일시스템은 컴퓨터 시스템 중에서 비교적 느린 발전을 보이는 디스크의 속도를 보완하기 위해서 병렬성을 파일시스템에 도입한 것이다. 이러한 파일시스템의 성능을 결정하는 요소로는 캐쉬 기법과 선반입 기법을 들 수 있다.

본 논문이 제안하는 선반입 기법인 *Fips(dynamic File Prefetching Scheme based on file access patterms)*는 파일의 접근 유형을 고려하여 동적으로 선반입 할 데이타 블록을 예측하므로 다양하고 가변적인 접근 유형에도 정확한 데이타 블록의 선반입이 가능하며, 현재의 가용 대역폭을 고려하여 선반입 시기를 결정하므로 선반입이 시스템에 부하로 작용하는 것을 방지하도록 하였다.

이를 실제 시스템에 적용한 결과 실험을 통하여 파일 요청 상황에 맞는 효과적인 선반입 정책을 수행함을 확인할 수 있었다. Fips는 시스템에 부하를 주지 않는 간단한 알고리즘으로도 서로 다른 여러 작업부하에서 우수한 성능을 나타내었다. 특히 부하가 과중한 경우에는 캐쉬 적중률이 떨어짐에도 불구하고 전체 읽기 시간이 줄어드는 결과를 보여주는데, 이를 통하여 Fips를 사용할 경우에 선반입이 시스템의 부하로 작용하는 것을 막을 수 있다는 사실을 확인하였다. 이는 선반입 정책이 앞으로 사용될 블록을 예측하는 것뿐만이 아니라 현재의 파일 요청 상황을 파악하여 이를 선반입 정책에 반영할 수 있어야만 모든 파일 요청 부하에 대해서 선반입이 시스템에 유리하게 작용할 수 있음을 보여준다.

향후 다양한 응용에서 병렬 파일시스템의 더 큰 성능

향상을 요구할 것이며, 이를 위해 더 많은 블록 서버를 연결하고자 할 것이다. 이때, 네트웍의 속도의 한계에 의해 파일시스템의 확장성이 제한될 수 있다. 이를 극복하기 위한 대안으로 lightweight 메시징 기법을 도입하여 네트웍의 부하를 줄이는 방안에 대한 연구가 필요하다. 그리고, 다양한 작업부하에 대하여 동적으로 선반입 및 캐싱 정책 및 기법을 변경하여 항상 최적화된 입출력 성능을 나타낼 수 있는 다양한 선반입 기법에 관한 연구가 필요하다.

참 고 문 헌

[1] L. Breslau, P. Cao, L. Fan, G. Phillips and S. Shenker, "Web Caching and Zipf-like Distributions : Evidence and Implications," In Proc. of IEEE Infocom '99, pp. 126-134, March 1999.

[2] Evgenia Smirni, Daniel A. Reed, "Workload Characterization of Input/Output Intensive Parallel Applications," Proc. of the Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Springer-Verlag Lecture Notes in Computer Science, Jun. 1997, Vol. 1245, pp. 169-180.

[3] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis and M. L. Best, "File-access characteristics of parallel scientific workloads," Parallel and Distributed Systems, IEEE Transactions on Vol. 7, Oct. 1996, pp. 1075 -1089.

[4] W. B. Ligon III and R. B. Ross, "An Overview of the Parallel Virtual File System," In Proc. of the 1999 Extreme Linux Workshop, June 1999.

[5] Ekechi K. E. Nwokah, "Parallel File Access On Workstation Clusters," Ph.D. Thesis, Purdue University, West Lafayette, IN, 1999.

[6] T. Cartes, "Cooperative Caching and Prefetching in Parallel/Distributed File Systems," Ph.D Thesis, Universitat Politecnica de Catalunya, 1997.

[7] R. H. Patterson and G. A. Gibson, "Exposing I/O concurrency with informed prefetching," In Proc. Third International Conf. on Parallel and Distributed Information Systems, pp. 7-16, September 1994.

[8] J. Cho, C. Kim, and D. Seo, "A Parallel File System Using Dual Cache Scheme and Prefetching," The 2000 International Conference on Parallel/Distributed Processing Techniques and Application (PDPTA2000), June, 2000.

[9] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," In Proc. of the 10th Symposium on Operating System Principles, pp. 15-24, December 1985.

[10] M. G. Baker, J. H. Hartman, M. D. Kupper, K. W. Shirriff and J. K. Ousterhout, "Measurements of a Distributed File System," In Proc. of 13th ACM Symposium on Operating Systems Principles, Association for Computing Machinery SIGOPS, pp. 198-212, October 1991.

[11] D. Kotz and C.S. Ellis. "Practical prefetching techniques for multiprocessor file systems," Journal of Distributed and Parallel Databases, 1(1):33-51, January 1993.

[12] R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. "Informed prefetching and caching," In Proc. of the Fifteenth ACM Symposium on Operating Systems Principles, pp. 79-95, December 1995.

[13] T. M. Madhyastha and D. A. Reed, "Exploiting Global Input/Output Access Pattern Classification," In Proc. of SC'97, November 1997, CD-ROM

이 윤 영

2000년 경북대학교 전자전기공학부 졸업(학사). 2002년 경북대학교 전자공학과 졸업(석사). 2002년 ~ 현재 넷컴스토리지, 연구원. 관심분야는 병렬처리, 분산처리, 클러스터 컴퓨팅, 암호화 시스템

김 재 열

1999년 경북대학교 전자전기공학부 졸업(학사). 2001년 경북대학교 전자공학과 졸업(석사). 2001년 ~ 현재 한국전자통신연구원 리눅스연구팀 연구원. 관심분야는 SAN기반 스토리지 시스템, 하드웨어 디바이스드라이버 개발, 로우레벨 프로그래밍

서 대 화

1983년 한국과학기술원 전산학과(석사). 1993년 한국과학기술원 전산학과(박사). 1981년 ~ 1995년 한국전자통신연구소 시스템 S/W 연구실 근무. 1998년 ~ 1999년 University of California Irvine 연구 교수. 1995년 ~ 현재 경북대학교 공과대학 전자전기공학부 부교수. 관심분야는 병렬분산처리, 운영체제, 병렬처리, 컴퓨터 구조