소프트웨어 RAID 파일 시스템에 작은 쓰기와 참조 횟수를 고려한 캐쉬 교체 정책

김 종 훈[†] · 노 삼 혁[††] · 원 유 헌[††]

요 약

본 논문에서는 소프트웨어 RAID 파일 시스템에서 효율적인 캐쉬 교체 정책들을 제안한다. 그리고 이와 기존의 캐쉬 교체 정책을 소프트웨어 RAID 파일 시스템에 적용한 정책들과의 성능을 다양한 환경에서 비교한다. 실험을 통해 우선 소프트웨어 RAID 파일 시스템에서 작은 쓰기 동작은 성능을 크게 저하시키는 요소임을 확인한다. 이러한 작은 쓰기 동작을 줄이는 캐쉬 교체 정책들을 제안한다. 이러한 교체 정책들에 대한 성능 비교는 트레이스 기반 시뮬레이션에 의해 수행된다. 실험 결과를 통해 본 논문에서 제안한 교체 정책들이 기존의 정책들에 비해 효율적인 성능을 나타냄을 확인한다.

Cache Replacement Policies Considering Small-Writes and Reference Counts for Software RAID File Systems

Jong-Hoon Kim[†] · Sam-Hyuk Noh[††] · Yoo-Hun Won[††]

ABSTRACT

In this paper, we present efficient cache replacement policies for the software RAID file system. The performance of this policies is compared to two other policies previously proposed for conventional file systems and adapted for the software RAID file system. As in hardware RAID systems, we found small-writes to be the performance bottleneck in software RAID file systems. To tackle this small-write problem, we propose cache replacement policies. Using trace driven simulations we show that the proposed policies improve performance in the aspect of the average response time and the average system busy time.

1. 서 론

전통적인 클라이언트/서버 파일 시스템에서는 모든 파일 시스템에 대한 서비스를 중앙 서버가 제공해 준다[1, 2]. 이러한 클라이언트/서버 형태의 분산 파일 시스템에서는 서버에 병목현상을 초래할 수 있다. 이러한 분산 시스템 환경에서 입출력 병목현상을 해결함과 동시에 신뢰성을 지원하기 위해 제안된 개념이 Zebra와 xFS와 같은 소프트웨어 RAID 파일 시스템이다[3, 4, 5]. 소프트웨어 RAID 파일 시스템이란 네트워크로 연결된 다수의 시스템들이 보유한 각 디스크를 통해 하드웨어 RAID[6]의 기능을 소프트웨어적으로 제공하는 시스템을 의미한다. 이러한 소프트웨어 RAID 형태의 파일 시스템에서 성능을 최적화하

※본 논문은 1996년 한국학술진흥재단의 공모 과제 연구비에 의해 연구되었음.
† 준 회 원:홍익대학교 전자계산학과
†† 정 회 원:홍익대학교 컴퓨터공학과
논문접수:1997년 9월 5일, 심사완료:1997년 10월 27일

기 위해서는 캐쉬의 사용이 불가피하다. 그러나 소프트웨어 RAID 파일 시스템은 패리티를 유지해야 하므로 갱신된 블록(dirty block) 교체시 기존의 분산 파일 시스템에서의 캐싱 동작과는 달리 새로운 패리티를 계산해야 하는 오버헤드가 발생한다. 그러므로 이에 대한 연구가 필요하나 소프트웨어 RAID 파일 시스템 환경에서 이를 고려한 캐쉬에 관한 연구는 진행된 바가 없다.

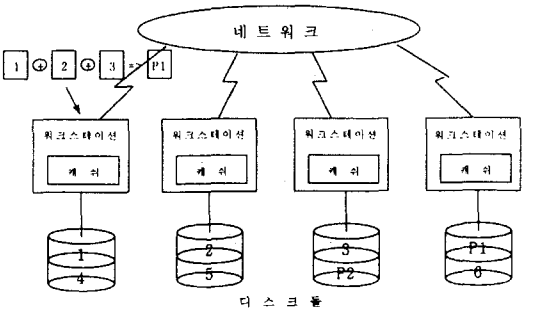
그러므로 본 논문에서는 갱신된 블록 교체를 고려한 캐쉬 교체 정책들을 제안하고 이와 기존의 교체 정책을 소프트웨어 RAID 파일 시스템에 적용한 정책들과 시뮬레이션을 통해 성능을 비교한다. 시뮬레이션 결과를 통해 본 논문에서 제안한 교체 정책들이 효율적임을 확인한다.

본 논문의 구성은 다음과 같다. 우선 2장에서는 소프트웨어 RAID 파일 시스템에 관하여 살펴보고, 3장에서 소프트웨어 RAID 파일 시스템에 캐쉬를 적용하였을 때 어떠한 변화가 생기는지를 살펴본다. 그리고 4장에서 소프트웨어 RAID 파일 시스템에 적용한 기존의 교체 정책과 본 논문에서 제안하는 교체 정책에 대해 설명한다. 5장에서는 성능 평가를 위한 실험 환경을 기술하며 실험 결과와 분석 내용을 설명한다. 마지막으로 6장에서 결론을 맺는다.

2. 소프트웨어 RAID 파일 시스템

분산 시스템 환경에서 입출력 병목현상을 해결함과 동시에 신뢰성을 지원하기 위해 도래된 개념이 소프트웨어 RAID 파일 시스템이다. 소프트웨어 RAID 파일 시스템이란 네트워크로 연결된 다수의 시스템들이 보유한 각 디스크를 통해 하드웨어 RAID의 기능을 소프트웨어적으로 제공하는 시스템을 의미한다. 하드웨어 RAID[6]란 데이터 읽기/쓰기 동작시 다수의 디스크들에 나누어 동시에 처리함으로 입출력 성능을 높일 수 있고, 패리티 정보를 함께 저장함에 의해 신뢰성을 높일 수 있는 디스크 기술이며 소프트웨어 RAID 파일 시스템은 이러한 기술을 분산 시스템 환경에 적용한 것이다. 하드웨어 RAID에 대한 내용은 본 논문의 범주를 벗어나므로 더 이상 언급하지 않기로 하겠다. 이에 대한 상세한 내용은 Chen et al.의 논문[6]을 참조하기 바란다.

소프트웨어 RAID 파일 시스템은 (그림 1)과 같은 형태로 구성되어 있다. (그림 1)에서 시스템은 초고속 통신망으로 연결되어 있는 분산 시스템 환경이며 전체 시스템들이 보유한 디스크를 통해 하드웨어 RAID의 기능을 소프트웨어적으로 제공하고 있다. 그럼으로 인해 얻어지는 장점은 하드웨어 RAID에서 얻어지는 장점과 동일한데 살펴보면 다음과 같다. 첫 번째는 파일 저장시 특정 워크스테이션의 디스크에만 저장되는 것이 아니라 저장될 파일을 여러 워크스테이션에 분산 저장함으로 입출력 동작에 있어서 성능 향상을 가져오게 된다. (그림 1)을 통해 설명하면 블록 1, 2, 3으로 구성된 파일을 디스크에 저장할 때 특정 워크스테이션에만 집중적으로 저장하는 것이 아니라 세대의 워크스테이션들의 디스크에 분산시켜 저장하게 되는 것이다. 두 번째 장점은 패리티 정보를 추가적으로 저장함으로 특정 시스템에 결함이 발생하여도 신뢰성을 유지할 수 있다는 것이다. (그림 1)에서 블록 1, 2, 3을 저장할 때 이들 블록에 대한 패리티 블록 P1을 계산하여 다른 워크스테이션의 디스크에 저장함으로 특정 워크스테이션의 결함이 발생하여도 나머지 워크스테이션들에 저장된 내용을 가지고 결함이 발생한 워크스테이션의 디스크에 저장된 정보를 복구할 수 있게 된다.



(그림 1) 소프트웨어 RAID 파일 시스템의 구성
(Fig. 1) The organization of a software RAID file system

그러면 기존의 소프트웨어 RAID 파일 시스템들을 살펴본다.

Swift[7]는 여러 파일 서버들에 파일들을 스트라이핑하여 관리하므로 입출력 성능을 향상시키는 분산

파일 시스템이다. 클라이언트들은 각 파일 입출력시 여러 서버들로 부터 동시에 서비스를 받으므로 입출력 성능의 향상을 가져오게 된다. 또한 Swift는 패리티 정보를 추가적으로 저장하므로 신뢰성이 높은 파일 서비스를 제공하게 된다. Swift는 클라이언트, 저장 에이전트, 저장 중개자, 그리고 분산 에이전트의 네 가지 요소들로 구성되었다. 클라이언트는 응용 프로그램들을 실행하며 파일들에 대한 읽기/쓰기 서비스를 받으며 저장 에이전트는 파일 데이터를 저장한다. 그리고 저장 중개자는 저장에 대한 관리와 네트워크 자원에 대한 관리를 한다. 특정 클라이언트가 파일 접근을 원할 경우 저장 중개자는 적절한 저장 방법을 계획하고 파일 전송을 위한 네트워크를 확보하는 등 전송 계획을 세운다. 또한 저장 중개자는 call-back을 이용하여 클라이언트 캐쉬의 일관성을 유지시킨다. 분산 에이전트는 저장 중개자에 의해 생성된 전송 계획에 따라 실행한다. 분산 에이전트의 가장 중요한 기능은 전송 계획에 따라 저장 에이전트들에게 파일 데이터를 스트라이핑하는 것이다. 쓰기 동작시에는 파일들을 블록 단위로 나누고 읽기 동작시에는 저장 에이전트들로 부터 블록들을 읽어서 파일로 모으는 일을 한다. 그러나 클라이언트들에 대한 모든 읽기/쓰기 동작은 분산 에이전트를 통해야만 하므로 분산 에이전트에 병목 현상이 발생하게 된다.

Zebra[5]는 LFS(Log-structured File System)[8] 개념과 RAID[6] 개념을 결합하여 분산 시스템 환경에서 동작하도록 만든 네트워크 파일 시스템이다. 작은 쓰기 동작시 패리티를 다시 계산하기 위해 옛 데이터와 옛 패리티를 읽어 오는 동작은 Zebra에 있어서 비용이 많이 든다. 그래서 이러한 비용을 줄이기 위해 쓰기 동작에 있어서 LFS의 효율적인 쓰기 동작은 Zebra의 네트워크 스트라이핑에 있어서 매우 중요하다. 각 Zebra 클라이언트들은 메모리 내에 있는 자신의 로그에 쓰기 동작을 한다. 로그에 쓰기 동작으로 인해 로그 세그먼트(log segment)라는 고정된 크기의 데이터가 저장이 되면 디스크에 이 로그 세그먼트를 쓰게 된다. 하나의 로그 세그먼트는 여러 개의 로그 단편(log fragment)들로 구성이 되어 있는데 로그 단편들을 데이터 스트라이핑 유닛이라 간주하면 되고, 패리티 단편을 패리티 스트라이핑 유닛이라 생각하면 된다. 하나의 로그 세그먼트를 구성하는 여러 개의 로그 단편들과 하나의 패리티 단편이 하나의 패리티 스트라이프가 되는데 각각의 단편들은 네트워크로 연결되어 있는 각각 다른 저장 서버(storage server)에 있는 디스크들에 분리하여 저장함으로 소프트웨어적인 RAID 개념을 얻게 된다.

xFS[4] 설계의 가장 중요한 철학은 중앙 병목 현상을 제거하는 것과 NOW(Networks of Workstations) 환경[3]에서 모든 자원들을 효율적으로 사용하자는 것이다. xFS는 여러 개의 관리자를 두어 분산시켜 관리를 하며, 저장 서버들을 다시 그룹화 하여 그룹 내의 서버들에 데이터를 분산시켜 저장하는 개념을 도입함에 의해 Zebra를 향상시킨 시스템이다. xFS 구조하에서 모든 기계들은 일반적인 데이터들 또는 메타 데이터들을 캐쉉, 관리 또는 저장에 대한 책임을 질 수 있다. 세 가지 주요 구성 요소는 클라이언트와 관리자(manager) 그리고 저장 서버(storage server)다. 클라이언트의 주요 역할은 cooperative 캐쉉[9]을 제공하는 것이다. 클라이언트는 사용자들로부터 파일에 대한 요구를 받고, 쓰기 동작을 하기 위해 저장 서버들에게 데이터를 보낸다. 그리고 캐쉬 미스가 발생했을 경우 관리자들에게 읽기 요구를 보내고, 저장 서버들 또는 다른 클라이언트들로 부터 응답을 받는다. 클라이언트는 또한 관리자로부터 데이터를 보내라는 요구를 받아 다른 클라이언트들에게 응답한다. 메타 데이터 관리자의 역할은 파일 데이터 블록들의 위치를 추적하는 것이고 클라이언트로의 요구를 적절한 목적지로 보내는 것이다. 마지막으로 저장 서버들은 공동으로 스트라이프 네트워크 디스크의 형태를 제공한다. 저장 서버는 클라이언트로부터 스트라이프드 쓰기 요구를 받는다. 또한 저장 서버는 관리자로부터 데이터를 제공하라는 요구를 받아 입출력 동작을 시작하려는 클라이언트들에게 반응한다.

그러나 이러한 소프트웨어 RAID 파일 시스템들은 캐쉬는 고려되지 않은 상태에서 연구가 이루어졌다.

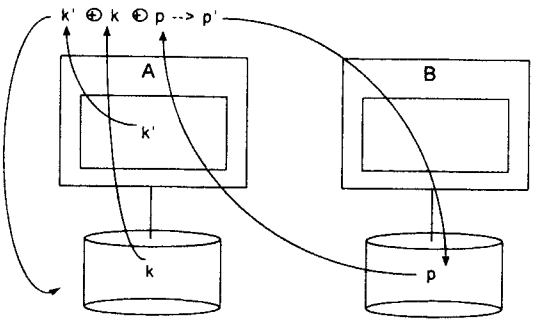
3. 소프트웨어 RAID 파일 시스템에서 캐쉉

파일 시스템의 성능을 최적화하기 위해서는 캐쉬의 사용이 불가피하므로 소프트웨어 RAID 파일 시스템에도 캐쉬를 적용해야 한다. 우선 분산 시스템 환경에서 캐쉬에 관한 연구들을 살펴본다.

분산 시스템은 클라이언트에서 캐쉬를 사용하여 네트워크 트래픽을 줄이며 서버의 부하를 감소시킨다. AT&T의 RFS(Remote File Sharing)는 초기에는 클라이언트 캐쉬가 없었으나 [10]에서 클라이언트 캐쉬를 추가하였다. AFS(Andrew File System)[11]는 클라이언트의 디스크를 캐쉬로 사용하여 파일 서버로부터 최근에 접근한 파일을 저장하며, 캐쉬 일관성 유지를 위하여 call-back를 사용한다. Sprite[2] 파일 시스템 역시 클라이언트 캐쉬를 사용한다. 특히 공유 상태가 아닌 파일의 경우 지연 쓰기(delayed-write)를 사용하여 다수의 쓰기 작업을 클라이언트 캐쉬에서 해결할 수 있다. 전통적인 분산 파일 시스템에서 클라이언트는 데이터를 접근하기 위해 먼저 자신의 캐쉬를 검색한다. 만약 이곳에서 데이터를 발견하지 못하면 서버의 캐쉬를 검색하고 마지막으로 서버의 디스크에서 데이터를 가져오게 된다. 그러나 이와 같은 파일 시스템에서는 중앙 서버에 병목현상이 발생하므로 [9, 12, 13]에서는 이러한 3-단계 캐쉬 구조에다 원격 클라이언트의 캐쉬를 추가하여 4-단계 캐쉬 구조를 형성하고, 분산 시스템에서 전체 시스템의 캐쉬 효율성을 향상시킬 수 있는 캐쉬 관리 기법들을 연구하였다. 그러므로 본 논문에서 제시한 소프트웨어 RAID 파일 시스템 모델의 캐쉬들간의 동작에도 원격 워크스테이션의 캐쉬에 직접 접근(access)이 가능하다고 가정하였다. 그러나 이러한 캐쉬에 관한 연구들은 일반적인 분산 시스템에 관한 연구이며 소프트웨어 RAID 파일 시스템에서의 연구는 아니다.

소프트웨어 RAID 파일 시스템에 캐쉬를 적용하면 기존의 분산 파일 시스템에서의 동작과 거의 동일하다. 그러나 각 워크스테이션 캐쉬의 교체 정책에서 교체 블록이 갱신된(dirty) 블록일 경우에 그 동작이 일반적인 분산 파일 시스템과는 차이가 있다. (그림 2)는 워크스테이션 A의 캐쉬에서 갱신된 블록 k′의 교체가 발생하였을 때의 처리 과정을 나타낸 것으로 그 동작을 살펴보면 다음과 같다.

(1) 교체 블록(k′)의 옛 데이터 블록(k)과 교체 블록과 같은 패리티 그룹에 속하는 옛 패리티 블록(p) 읽기
(2) 새로운 데이터(k′)와 옛 데이터 블록(k), 그리고 옛 패리티 블록(p)을 가지고 새로운 패리티 블록(p′) 계산
(3) 새로운 데이터 블록(k′)과 새로운 패리티 블록(p′) 쓰기



(그림 2) 캐쉬에서 갱신된 블록 k'의 교체 발생시의 동작
(Fig. 2) Action taken when a dirty block k' is being replaced from the cache

이러한 동작을 작은 쓰기 동작[6]이라 하고 하나의 작은 쓰기 동작을 처리하기 위해서는 옛 데이터 읽기, 옛 패리티 읽기, 새로운 데이터 쓰기와 새로운 패리티 쓰기의 네 번의 디스크 요구가 발생한다. 하드웨어 RAID 레벨 4와 5에서 이러한 동작은 응답 시간(response time)을 RAID 레벨 0에 비해 약 두 배 가량 증가시키고 처리량(throughput)에 있어서는 약 1/4 가량으로 감소시키는 것으로 알려져 있다[6]. 이러한 작은 쓰기 동작에서의 성능을 증가시키기 위해 기존의 하드웨어 RAID에서 캐쉬에 관한 연구는 [14]가 있다. [14]에서는 디스크 컨트롤러내의 비휘발성 저장 장치에 옛 데이터와 옛 패리티를 저장시켜 놓고 동작함으로 작은 쓰기 동작시 디스크에 대한 접근을 줄이는 방법이다. 소프트웨어 RAID 파일 시스템에서도 하드웨어 RAID처럼 작은 쓰기 동작이 성능을 제한할 것으로 예상되나 하드웨어 RAID에서 적용하던 위의 기법을 그대로 적용하는데는 문제가 있다. 우선 하드웨어 RAID와는 달리 소프트웨어 RAID 파일 시스템에서 각 파일 시스템들은 일반적인 데이터 블록과 패리티 블록을 구분하여 관리하기가 어렵다. 또 일반 워크스테이션을 사용하기 때문에 메모리의 일부를 비휘발성 메모리로 대체한다는 것은 현실적이지 못하다. 본 논문에서는 캐쉬를 적용한 소프트웨어 RAID 파일 시스템 환경에서 이러한 문제점을 보완

하기위해 캐쉬 교체 기법을 제안한다.

소프트웨어 RAID 파일 시스템에서 사용되는 용어는 하드웨어 RAID 시스템과 유사하다. 예를 들면, 스트라이프(stripe)란 패리티 블록과 패리티가 계산되어지는 최소의 데이터 블록들의 집합을 의미하고, 스트라이핑 유닛(striping unit)이란 스트라이프를 구성하고 있는 여러 블록중 단일 시스템에만 저장되는 한 블록을 의미한다. 본 논문에서 제안하는 소프트웨어 RAID 파일 시스템은 RAID 레벨 5 형태로 구성되어 있음을 가정한다.

4. 캐쉬 교체 정책

본 장에서는 기존의 캐쉬 교체 정책을 소프트웨어 RAID 파일 시스템에 적용한 정책을 살펴보고 본 논문에서 제안하는 교체 정책들에 대해 살펴보고자 하는데 우선 공통된 사항을 기술하면 다음과 같다.

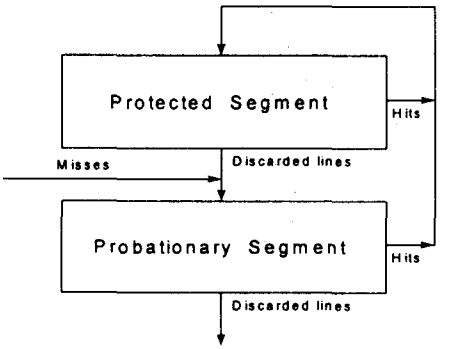
본 시스템에서는 다중 복사 캐쉉 기법[1]을 사용하지 않고 단일 복사 캐쉉 기법[12]을 사용하였는데 우선 이들 기법들을 설명하면 다음과 같다.

다중 복사 캐쉉 기법은 동일한 블록을 여러 워크스테이션에서 중복해서 저장이 가능하도록 하는 방법으로 리모트 접근을 줄이고자 하는 기법으로 NFS[1]에서 사용하는 것으로 최근 실 시스템에서 가장 보편적으로 사용되고 있는 기법이다. 반면 단일 복사 캐쉉 기법은 캐쉬 자체의 구성 형태는 다중 복사 캐쉉 기법과 동일하나 시스템 전체의 캐쉬 내에 저장되어 있는 모든 블록은 한 개만을 유지시키는 기법[12]이다. 본 기법은 동일한 블록이 중복해서 저장됨으로 그런 만큼의 다른 유용한 블록을 캐쉬에 두지 못하는 문제점을 해결하기 위해 제안되었다. 전체 캐쉬에 블록을 한 개만을 유지시킴으로 리모트 접근이 자주 발생하게되는 문제점이 있으나 최근 ATM[15]과 같은 초고속 네트워크의 출현으로 리모트 캐쉬로부터 로칼 캐쉬로 데이터를 이동시키는 것은 성능에 큰 영향을 미치지 않는다. 또한 다중 복사 캐쉉 기법에서 대두되던 캐쉬 일관성에 대한 문제는 발생하지 않게 된다.

이러한 단일 복사 캐쉉 기법과 다중 복사 캐쉉 기법들에 대한 성능 비교가 [16]에서 이루어 졌는데 단일 복사 캐쉉 기법이 다중 복사 캐쉉 기법에 비해 요구당 응답시간과 시스템 동작시간에 있어서 좋은 성능을 나타냄으로 본 시스템에서는 단일 복사 캐쉉 기법을 적용하게 되었다.

4.1 참조 횟수를 고려한 교체 정책(Reference Count Targeted Policy : RCT)

본 정책은 기존의 sLRU[17] 정책을 소프트웨어 RAID 파일 시스템에 그대로 적용한 것이다.



(그림 3) sLRU 캐쉬의 논리적인 구성과 데이터 흐름
(Fig. 3) *Logical organization and data flow of the sLRU* cache.

sLRU에서 캐쉬는 (그림 3)과 같이 임시 세그먼트(Probationary Segment)와 보호 세그먼트(Protected Segment)의 논리적인 두 영역으로 구성되어 처음으로 참조가 이루어져 캐쉬로 들어오는 블록은 임시 세그먼트에 유지시키다가 임시 세그먼트에서 히트가 발생하면 보호 세그먼트로 이동시키는 정책으로 이러한 개념이 나오게 된 동기는 다음과 같다. 블록이 참조되어 캐쉬에 올라온 후 일정한 시간 안에 두 번째 참조가 발생하지 않으면 이러한 블록은 캐쉬에서 제거될 때까지 계속해서 참조가 이루어지지 않는 경향이 있는 반면 일정한 시간 안에 두 번째 참조가 발생하면 그 블록은 빈번히 참조될 가능성이 크다. 즉 본 정책은 소프트웨어 RAID 파일 시스템에 이러한 sLRU를 그대로 적용하여 참조될 가능성이 적은 블록을 캐쉬에서 축출하여 참조 가능성이 큰 다른 블록이 캐쉬에서 오랫동안 유지될 수 있도록 해주는 정책으로 블록 요구시 동작 알고리즘은 (그림 4)와 같다.

그림에서 MRU는 most recently used로 LRU 리스트의 첫 번째 위치를 의미한다.

```
if (지역 임시 세그먼트에서 히트) then
        지역 보호 세그먼트의 MRU 블록으로 이동;
else if (지역 보호 세그먼트에서 히트) then
        지역 보호 세그먼트의 MRU 블록으로;
else if (원격 임시 세그먼트에서 히트) then
        지역 임시 세그먼트의 MRU 블록으로 이동;
else if (원격 보호 세그먼트에서 히트) then
        지역 임시 세그먼트의 MRU 블록으로 이동;
else  (* 지역 또는 원격 디스크에 존재 *)
        지역 임시 세그먼트의 MRU 블록으로 복사;
```

(그림 4) RCT 정책에서 블록 요구시의 동작 알고리즘
(Fig. 4) RCT policy's algorithm when the block is requested

4.2 작은 쓰기를 고려한 교체 정책(Small-Write Targeted Policy : SWT)

캐쉬의 형태는 앞에서와 같이 논리적인 임시 세그먼트와 보호 세그먼트로 구성된다. 참조 횟수를 고려한 교체 정책은 참조될 가능성이 높은 블록을 보호 세그먼트에 위치시킴으로 오랫동안 캐쉬에 유지될 수 있도록 해주는 반면 본 정책은 갱신된 블록을 보호 세그먼트에 위치시킴으로 캐쉬에 오랫동안 유지될 수 있게 하는 방법이다. 앞서 언급하였듯이 소프트웨어 RAID 파일 시스템에서는 갱신된 블록 교체시 네 번의 디스크 요구가 수반되는데 이는 전체 시스템의 성능에 큰 병목으로 자리잡고 있는데 본 기법에서는 갱신된 블록을 캐쉬에 오랫동안 유지시킴으로 갱신된 블록이 교체되는 횟수를 감소시킴으로 디스크로의 접근을 최소화하여 시스템의 성능을 향상시키자는 것이다.

블록 요구가 발생하여 블록이 캐쉬에 들어올 경우 갱신된 블록이거나 갱신된 블록이 될 블록은 보호 세그먼트로 위치시키고 그 외의 블록들은 임시 세그먼트에 위치시킴으로 갱신된 블록을 캐쉬에 오랫동안 유지하도록 하여 이러한 갱신된 블록이 교체되는 횟수를 감소시키게 된다. 블록 요구시 동작을 알고리즘으로 표현하고자 하는데 읽기와 쓰기 요구를 분리하여 살펴본다. 우선 읽기 요구시의 동작은 (그림 5)와 같다.

```
if (지역 임시 세그먼트에서 히트) then
    if (갱신된 블록) then
        지역 보호 세그먼트의 MRU 블록으로 이동;
    else
        지역 임시 세그먼트의 MRU 블록으로;
else if (지역 보호 세그먼트에서 히트) then
    지역 보호 세그먼트의 MRU 블록으로;
else if (원격 임시 세그먼트에서 히트) then
    if (갱신된 블록) then
        지역 보호 세그먼트의 MRU 블록으로 이동;
    else
        지역 임시 세그먼트의 MRU 블록으로 이동;
else if (원격 보호 세그먼트에서 히트) then
    지역 보호 세그먼트의 MRU 블록으로 이동;
else  (* 지역 또는 원격 디스크에 존재 *)
    지역 임시 세그먼트의 MRU 블록으로 복사;
```

(그림 5) SWT 정책에서 읽기 요구시의 동작 알고리즘
(Fig. 5) SWT policy's algorithm when the block is read-requested

(그림 6)은 쓰기 요구시의 동작을 알고리즘으로 나타낸 것이다.

```
if (지역 임시 세그먼트에서 히트) then
        지역 보호 세그먼트의 MRU 블록으로 이동;
else if (지역 보호 세그먼트에서 히트) then
        지역 보호 세그먼트의 MRU 블록으로;
else if (원격 임시 세그먼트에서 히트) then
        지역 보호 세그먼트의 MRU 블록으로 이동;
else if (원격 보호 세그먼트에서 히트) then
        지역 보호 세그먼트의 MRU 블록으로 이동;
else  (* 지역 또는 원격 디스크에 존재 *)
        지역 보호 세그먼트의 MRU 블록으로 복사;
```

(그림 6) SWT 정책에서 쓰기 요구시의 동작 알고리즘
(Fig. 6) SWT policy's algorithm when the block is write-requested

4.3 작은 쓰기와 참조 횟수를 고려한 교체 정책 (Small-Write and Reference Count Targeted Policy : SRT)

본 정책은 참조 횟수를 고려한 교체 정책과 작은

쓰기에 효율적인 교체 정책을 통합한 정책으로 캐쉬의 논리적인 구조는 위의 정책들과 동일하다. 즉 참조 가능성이 높은 블록과 갱신된 블록을 보호 세그먼트에 위치시킴으로 캐쉬에 오랫동안 유지될 수 있게 하고 나머지 블록들은 임시 세그먼트에 위치시켜 캐쉬에서 빠른 시간 내에 축출하도록 하는 정책이다. 그러므로 네 번의 디스크의 접근을 발생시키는 갱신된 블록이 교체되는 횟수를 감소시키게 되고 또한 자주 참조되는 블록을 캐쉬 내에 오랫동안 유지시키게 되므로 파일 시스템의 성능 향상을 가져오게 된다. 동작을 의사 코드로 나타내면 다음과 같다. 읽기 요구가 발생하였을 경우 (그림 7)과 같이 동작한다.

```
if (지역 임시 세그먼트에서 히트) then
        지역 보호 세그먼트의 MRU 블록으로 이동;
else if (지역 보호 세그먼트에서 히트) then
        지역 보호 세그먼트의 MRU 블록으로;
else if (원격 임시 세그먼트에서 히트) then {
        if (갱신된 블록) then
          지역 보호 세그먼트의 MRU 블록으로 이동;
        else
          지역 임시 세그먼트의 MRU 블록으로 이동;
}
else if (원격 보호 세그먼트에서 히트) then
        지역 보호 세그먼트의 MRU 블록으로 이동;
else  (* 지역 또는 원격 디스크에 존재 *)
        지역 임시 세그먼트의 MRU 블록으로 복사;
```

(그림 7) SRT 정책에서 읽기 요구시의 동작 알고리즘
(Fig. 7) SRT policy's algorithm when the block is read-requested

쓰기 요구시의 동작은 SWT 정책에서와 동일하다.

5. 실 험

본 장에서는 성능 평가를 위한 실험과 환경을 살펴보고 실험 결과와 그의 분석 내용을 설명한다.

5.1 실험 방법 및 환경

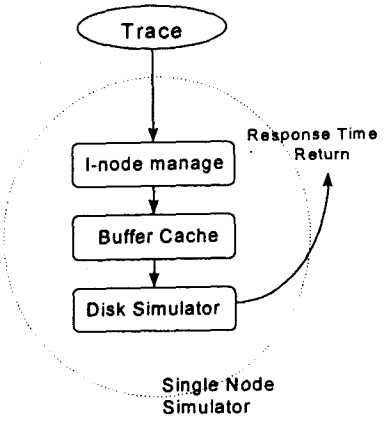
본 논문에서는 트레이스 기반 시뮬레이션을 통하여 캐싱 기법들을 평가하였다. 사용된 트레이스는 스프라이트 트레이스[18]이다. 스프라이트 트레이스는 버클리 대학에서 4대의 파일 서버와 40여대의 클라이언트에서 매일 사용하는 30여명의 사용자와 간헐적으로 사용하는 40여명의 사용자들에게서 얻어진 분산 파일 시스템의 트레이스로 분산 시스템 환경에서 가장 대표되는 트레이스이다. 본 실험에서는 여덟 대와 열 여섯 대의 워크스테이션으로 이루어진 소프트웨어 RAID 파일 시스템을 시뮬레이션하였는데 각각에 대해 여덟 개와 열 여섯 개의 트레이스를 임의로 선택하여 시뮬레이터의 작업부하로 사용하였다. 실험에 사용된 작업부하의 정보는 〈표 1〉에 있다. 이러한 트레이스들간에는 동일한 블록에 대한 요구가 포함되어 있어 파일을 공유하는 경우가 존재하며 정책들에 따라 캐쉬 일관성을 유지하며 동작을 하는데 본 시뮬레이터에서도 동일하게 동작한다.

〈표 1〉 실험에 사용된 트레이스의 정보
〈Table 1〉 Information of the trace used in the simulation

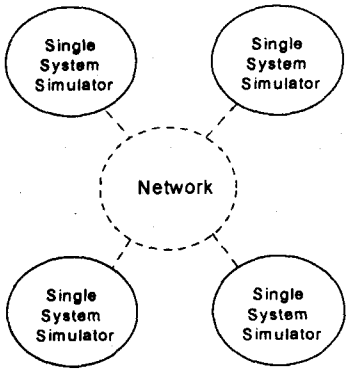
트레이스의 갯수	읽기 요구수(%)	쓰기 요구수(%)	전체 요구수
8	228,136(75)	76,045(25)	304,181
16	334,077(73)	123,563(27)	457,640

이러한 실험에서 각 워크스테이션 캐쉬를 초기화시키는데 소요된 요구를 제외한 나머지 요구를 가지고 평가하였다. 본 논문에서 시뮬레이터는 C++를 사용하여 개발하였다.

시뮬레이터는 여러 단일 노드들이 네트워크에 연결되어 있는 구성을 가진다. 단일 노드의 구성은 (그림 8(a))와 같이 트레이스에서 입출력 요구를 받아서 i-node[19]를 관리하는 부분, 버퍼 캐쉬의 관리를 위한 부분, 그리고 디스크 부분으로 나누어진다. 본 실험에서 디스크 부분은 HP 97560을 모델링한 시뮬레이터[20]를 사용하였다. 따라서, 실험에 사용된 시뮬레이터에서는 모든 노드가 같은 종류의 디스크를 가지고 있게 된다. 이와 같이 구성된 단일 노드 시뮬레이터는 C++의 클래스를 사용하여 제작되었으며, 이러한 노드들을 (그림 8(b))와 같이 네트워크에 연결되어 있는 모습으로 재구성하여 소프트웨어 RAID 파일 시스템 시뮬레이터를 제작하였으며 RAID 레벨 5 형태로 구성되었다.



(a) 단일노드



(b)소프트웨어 RAID 파일 시스템

(그림 8) 시뮬레이터의 구성
(Fig. 8) The organization of the simulator

실험에서 캐쉬 블록의 크기는 8KB이며, 8KB 데이터에 대한 메모리 접근 시간은 지역 메모리는 250μs [21], 네트워크 오버헤드는 200μs로 하고 데이터가 네트워크를 통해 전송되는데 걸리는 시간은 400μs[22]로 고정시키고 실험하였는데 이러한 시뮬레이션 파라미터들은 [9]에서와 동일하다. 이들 파라미터는 〈표 2〉에 나타나 있으며 패리티를 계산하는 시간은 무시하였다.

각 기법에 대한 성능 비교 척도는 요구당 평균 응답시간(average response time)과 요구당 시스템 동작 시간(average system busy time)이다.

요구당 평균 응답시간은 입출력 요구가 발생했을 때 블록에 대한 요구가 발생하였을 때부터 응답을 받을 때까지의 시간으로 사용자들이 실질적으로 느끼는 성능 척도이다.

〈표 2〉 시뮬레이션 파라미터
〈Table 2〉 Simulation parameters

캐쉬 블록의 크기	8KB
8KB 데이터에 대한 지역 메모리 접근 시간	250 microseconds [21]
네트워크 오버헤드	200 microseconds [22]
8KB 데이터가 네트워크를 통해 전송되는 시간	400 microseconds [22]
디스크	HP 97560 하드 디스크 시뮬레이터 [20]

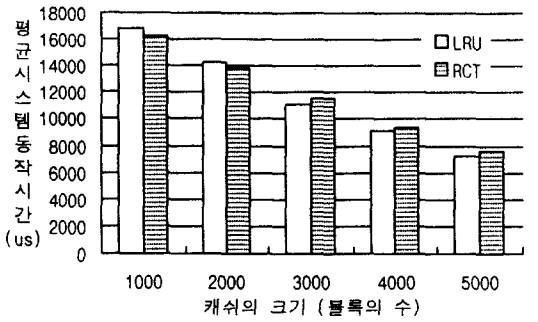
다음으로 입출력 요구당 시스템 동작 시간을 측정하였는데 이는 입출력 요구시마다 네트워크와 디스크의 전체 시스템이 동작한 시간을 합한 값을 입출력 요구의 수로 나눈 값이다. 이를 분석한 이유는 각 요구마다 전체 시스템의 오버헤드를 명확하게 측정하기 위해서다. 소프트웨어 RAID 파일 시스템에서는 오손 블록 교체시 옛 데이터 읽기와 옛 패리티 읽기 동작이 서로 다른 워크스테이션에서 동시에 발생하고 새로운 데이터 쓰기와 새로운 패리티 쓰기 동작 역시 다른 워크스테이션에서 동시에 발생하기 때문에 평균 응답 시간에서는 측정할 수 없는 전체 시스템에서의 실질적인 오버헤드를 측정하기 위해서 요구당 평균 시스템 동작 시간을 측정한다.

실험은 다양한 형태의 환경에서 각 워크스테이션 캐쉬의 크기를 변화시키면서 각 정책들의 성능을 분석하였다.

5.2 실험 결과

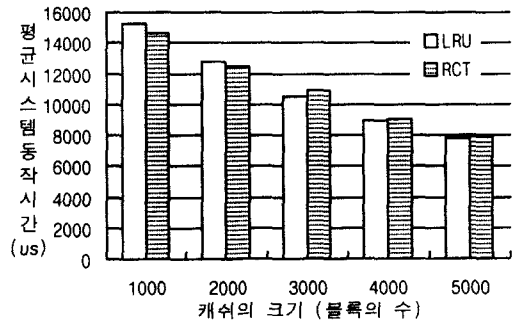
(그림 9)와 (그림 10)은 기존의 교체 정책들을 소프트웨어 RAID 파일 시스템에 적용하여 실험한 결과이다. 각 정책은 LRU와 RCT로 RCT는 sLRU를 소프트웨어 RAID 파일 시스템에 적용한 참조 횟수를 고려한 정책이다. (그림 9)는 워크스테이션이 8대로 구성된 경우이고 (그림 10)은 16대로 구성되었을 때의 평균 시스템 동작 시간이다. 그림에서 X축은 각 워크스테이션의 캐쉬의 크기를 나타내는 것으로 캐싱할 수 있는 블록(8KB)의 개수를 의미하고 Y축은

입출력 요구당 평균 시스템 동작 시간을 나타내는 것으로 단위는 microseconds이다.



(그림 9) 워크스테이션의 수가 여덟인 경우 LRU와 RCT의 평균 시스템 동작시간

(Fig. 9) Average system busy time of the LRU and RCT for an 8 workstation configuration

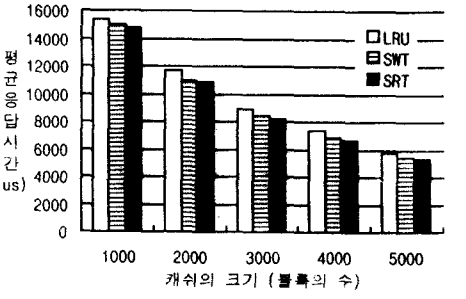


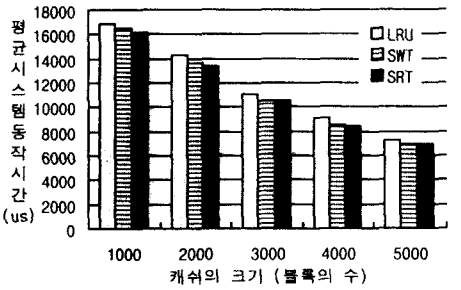
(그림 10) 워크스테이션의 수가 열 여섯인 경우 LRU와 RCT의 평균 시스템 동작시간

(Fig. 10) Average system busy time of the LRU and RCT for a 16 workstation configuration

그림의 결과에서 몇 가지 사항을 관찰할 수 있다. 우선 캐쉬의 크기가 작은 경우에는 RCT 정책이 좋은 성능을 나타내는 것을 볼 수 있다. 이는 캐쉬의 크기가 작은 경우에는 교체가 많이 발생하게 되는데 RCT 정책이 참조 가능성이 높은 블록들을 캐쉬에 오랫동안 유지시킴으로 교체 발생 횟수를 최대한 줄일 수 있기 때문이다. 그러나 캐쉬의 크기가 커짐에 따라 LRU 정책이 좋은 성능을 나타내고 있다. 이유는 다음과 같다. [17]에 나타나듯이 참조가 빈번히 발생하는 블록은 쓰기 요구에 의한 블록이라기 보다는 읽기 요구에 의한 블록이다. 그러므로 RCT 정책을 사용할 경우에는 읽기 요구에 의한 블록은 캐쉬 내에 오랫동안 유지되는 반면 쓰기 요구에 의한 블록 즉, 갱신된 블록은 캐쉬에서 이내 쫓겨나게 된다. 캐쉬의 크기가 커짐에 따라 교체 발생 빈도수는 감소하게 되는데 얼마 안되는 교체 중 갱신된 블록의 교체가 많아지게 되면 소프트웨어 RAID 파일 시스템에서는 커다란 성능 저하를 가져오게 된다. 앞서 언급하였듯이 갱신된 블록의 교체는 네 번의 디스크 요구를 수반하기 때문이다. 그러므로 소프트웨어 RAID 파일 시스템에서는 참조 횟수만을 고려하여 캐쉬를 관리하는 RCT 정책은 효율적이지 못하다.

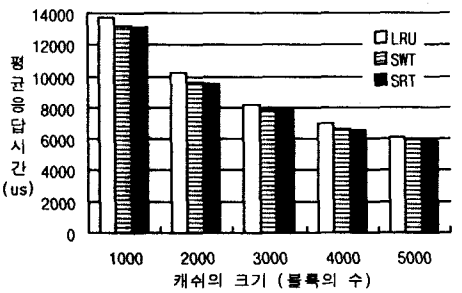
(그림 11)과 (그림 12)는 본 논문에서 제안한 정책들과 LRU 정책의 평균 응답 시간과 평균 시스템 동작 시간을 나타낸 것이다. 그림에서 SWT는 작은 쓰기 동작을 고려한 교체 정책을 의미하는 것이고 SRT는 작은 쓰기와 참조 횟수를 고려한 정책이다. 모든 경우에 있어서 본 논문에서 제안한 교체 정책들이 3-8% 가량 높은 평균 응답 시간과 평균 시스템 동작 시간을 나타내는 것을 볼 수 있다. 특히 작은 쓰기 동작만을 고려한 정책보다는 작은 쓰기와 참조 횟수를 함께 고려한 정책이 1.5% 정도 높은 성능을 나타내고 있다. 즉 소프트웨어 RAID 파일 시스템에서는 네 번의 디스크 접근을 발생시키는 갱신된 블록의 교체는 시스템의 성능을 저하시키는 요소임을 알 수 있고 본 논문에서 제안한 정책들은 갱신된 블록이 교체되는 횟수를 감소시키므로 기존의 정책들에 비해 효율적인 응답시간과 시스템 동작시간을 나타낸다. 특히 SRT는 참조 횟수도 함께 고려하여 가장 효율적인 정책임을 알 수 있다.

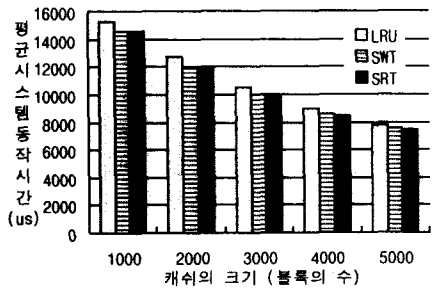




(그림 11) 워크스테이션의 수가 여덟인 경우 교체 정책들의 성능

(Fig. 11) Performance of the replacement policies for an 8 workstation configuration





(그림 12) 워크스테이션의 수가 열 여섯인 경우 교체 정책들의 성능

(Fig. 12) Performance of the replacement policies for a 16 workstation configuration

즉 소프트웨어 RAID 파일 시스템에서는 작은 쓰기 동작이 성능에 있어서 매우 큰 병목을 차지하고, 본 논문에서 제안한 교체 정책들은 이러한 문제점을 해결하는 정책으로 성능 향상을 가져오는 효율적이며 확장성 있는 정책임을 확인할 수 있다.

6. 결 론

전통적인 클라이언트/서버 형태의 파일 시스템에서는 모든 파일 시스템에 대한 서비스를 서버가 제공함으로 서버에 병목현상이 발생한다. 분산 시스템에서 이러한 중앙 서버의 병목을 해결함과 동시에 높은 신뢰성을 제공하기 위해 제안된 파일 시스템이 소프트웨어 RAID 파일 시스템이다. 소프트웨어 RAID 파일 시스템이란 네트워크로 연결된 다수의 시스템들이 보유한 각 디스크를 통해 하드웨어 RAID의 기능을 소프트웨어적으로 제공하는 시스템을 의미한다. 이러한 시스템은 기존의 파일 시스템에 비해 높은 성능과 신뢰성을 제공해준다. 본 논문에서는 소프트웨어 RAID 파일 시스템에서 효율적인 캐쉬 교체 정책들을 제안하였다. 그리고 이와 기존의 캐쉬 교체 정책들과의 성능을 다양한 환경하에서 비교하였다. 하드웨어 RAID 시스템에서와 마찬가지로 소프트웨어 RAID 파일 시스템에서도 작은 쓰기 동작은 성능을 크게 저하시키는 요소임을 확인하고 캐쉬 영역을 임시 세그먼트와 보호 세그먼트의 논리적인 두 영역으로 구성하여 블록들을 관리하는 캐쉬 교체 정책을 제안하였다. 갱신된 블록과 참조될 가능성이 높은 블록은 캐쉬에 오랫동안 유지될 수 있도록 해줌으로 성능을 향상시키자는 것이다. 이러한 교체 정책들에 대한 성능 비교는 트레이스 기반 시뮬레이션에 의해 수행되었다. 실험 결과를 통해 본 논문에서 제안한 교체 정책들이 기존의 정책들에 비해 효율적인 성능을 나타냄을 확인할 수 있었다.

참 고 문 헌

[1] R. Sandberg et al., "Design and Implementation of the SUN Network File System," In *Proceedings of the Summer 1985 USENIX Conference*, pages 119-130, Portland, Oregon, 1985.

[2] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch, "The Sprite Network Operating System," *IEEE Computer*, pages 23-36, February 1988.

[3] T. E. Anderson, D. E. Culler, and D. A. Patterson, "A Case for NOW(Networks of

Workstations)," *IEEE Micro*, 15(1):54-64, February 1995.

[4] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang, "Serverless Network File System," *ACM Transactions on Computer Systems*, 14(1):41-79, February 1996.

[5] J. H. Hartman and J. K. Ousterhout, "The Zebra striped network file system," *ACM Transactions on Computer System*, 13(3):274-310, August 1995.

[6] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID:High-Performance, Reliable Secondary Storage," *ACM Computing Surveys*, 26(2):145-185, June 1994.

[7] L.-F. Cabrera and D. D. E. Long, "Swift:Using Distributed Disk Striping to Provide High I/O Data Rates," *Computing Systems*, 4(4):405-436, Fall 1991.

[8] M. Rosenblum and J. K. Ousterhout, "The Design and Implementatio n Log-Structured File System," *ACM Tr ons on Computer Systems*, 10(1):26-52, February 1992.

[9] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson, "Cooperative Caching:Using remote client memory to improve file system performance," In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 267-280, November 1994.

[10] M. J. Bach, M. W. Luppi, and A. S. Melamed, "A Remote-File Cache for RFS," In *Proceedings of the Summer 1987 USENIX Conference*, pages 273-279, January 1987.

[11] J. Howard et al., "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, 6(1):51-81, February 1988.

[12] A. Leff, J. L. Wolf, and P. S. Yu, "Efficient LRU-Based Buffering in a LAN Remote Caching Architecture," *IEEE Transactions on Parallel and Distributed Systems*, 7(2):191-206, February 1996.

[13] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, and H. M. Levy, "Implementing Global Memory Management in a Workstation Cluster," In *Proceedings of the 15th Symposium on Operating System Principles*, pages 201-212, December 1995.

[14] J. Menon and J. Cortney, "The Architecture of a Fault-tolerant Cached RAID Controller," In *Proceedings of the 20th Annual International Symposium on Computer Architecture(ISCA '93)*, pages 76-86. May 1993.

[15] D. E. McDysan and D. L. Spohn, *ATM:Theory and Application*, McGraw-Hill, 1995.

[16] J. H. Kim, Sam H. Noh, and Y. H. Won, "An Efficient Caching Scheme for Software RAID File Systems in Workstation Clusters," In *Proceedings of High Performance Computing (HPC)* ASIA '97, pages 331-336, April 28-May 2 1997.

[17] R. Karedla, J. S. Love, and B. G. Wherry, "Caching Strategies to Improve Disk System Performance," *IEEE Computer*, 27(3):38-46, March 1994.

[18] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, "Measurements of a Distributed File System," In *Proceedings of the ACM Thirteenth Symposium on Operating Systems Principles*, pages 198-212, October 1991.

[19] S. J. Leffler, M. K. McKusick, M/J. Karels, and J. S. Quarterman, *4.3 BSD UNIX Operating System*, Addison-Wesley, 1990.

[20] D. Kotz, S. B. Toh, and S. Radhakrishnan, "A Detailed Simulation Model of the HP 97560 Disk Drive," Technical Report PCS-TR94-220, Dartmouth College, Computer Science, Hanover, NH, 1994.

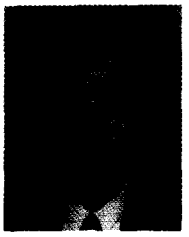
[21] R. Martin, "HPAM:An Active Message Layer for a Network of HP Workstations," In *Proceedings of the 1994 Hot Interconnects II Conference*, 1994.

[22] K. K. Keeton, T. E. Anderson, and D. A. Patterson, "LogP Quantified:The Case for Low-Overhead Local Area Networks," In *Proceedings*

of the 1995 Hot Interconnects Ⅲ *Conference*, 1995.

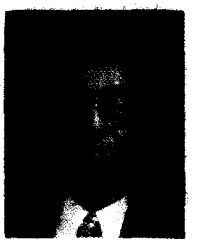
김 종 훈

1990년 목원대학교 수학교육학과(이학사)
1992년 동국대학교 대학원 통계학과 전산통계전공(이학석사)
1993년~현재 홍익대학교 대학원 전자계산학과 박사과정
관심분야:분산 시스템, 운영체제, 분산 멀티미디어 시스템



노 삼 혁

1986년 서울대학교 컴퓨터공학과(공학사)
1993년 Univ. of Maryland of College Park(공학박사)
1993년~1994년 George Washinton Univ. 객원 조교수
1994년~현재 홍익대학교 컴퓨터공학과 조교수
관심분야:분산 및 병렬 시스템, 컴퓨터 시스템



원 유 헌

1972년 성균관대학교 수학과(이학사)
1975년 한국과학원 전자계산학과(이학석사)
1985년 고려대학교 수학과 전산전공(이학박사)
1975년~1976년 한국과학기술연구소 연구원
1986년~1987년 R.P.I. 교환교수
1976년~현재 홍익대학교 컴퓨터공학과 교수
관심분야:프로그래밍 언어론, 분산 시스템, 실시간 프로그래밍