리눅스 상에서 멀티미디어 데이타를 고려한 지역 버퍼 할당 기법

(A Local Buffer Allocation Scheme for Multimedia Data on Linux)

신 동 재 † 박 성 용 †† 양 지 훈 †††
(Dongjae Shin) (Sungyong Park) (Jihoon Yang)

요 약 리눅스와 같은 범용 운영체제의 버퍼 캐시(buffer cache)는 전역적(global) 블록 교체 및 미리 읽기(read ahead) 정책 등을 사용하여 파일 블록을 관리한다. 따라서, 참조의 지역성(locality)을 가지지 않고 다양한 소비율(consumption rate)을 갖고 있는 멀티미디어 데이타의 경우 캐시 시스템의 적중률이 낮을 뿐만 아니라 미리읽기의 특성으로 인하여 필요보다 과도하게 버퍼를 소비하기도 한다. 본 논문에서는 리눅스 상에서 멀티미디어 데이타를 위한 새로운 버퍼 할당 기법을 설계하고 구현하였다. 제안된 방법에서는 멀티미디어 파일마다 독립적인 미리읽기 캐시를 유지하며 미리읽기 그룹의 크기를 소비율에 비례하도록 동적으로 조절한다. 이는 공정한 자원 분배가 이루어지도록 하며, 버퍼의 소비량을 최적화되도록 한다. 본 논문에서는 구현된 시스템과 최신의 리눅스 커널 2.4.17 버전 상에서 각각 소비되는 버퍼 수와 캐시 적중률을 실험을 통하여 비교함으로써 시스템의 성능을 평가한다.

키워드 : 버퍼 할당 기법, 멀티미디어 데이타, 리눅스

Abstract The buffer cache of general operating systems such as Linux manages file data by using global block replacement policy and read ahead. As a result, multimedia data with a low locality of reference and various consumption rate have low cache hit ratio and consume additional buffers because of read ahead. In this paper we have designed and implemented a new buffer allocation algorithm for multimedia data on Linux. Our approach keeps one read-ahead cache per every opened multimedia file and dynamically changes the read-ahead group size based on the buffer consumption rate of the file. This distributes resources fairly and optimizes the buffer consumption. This paper compares the system performance with that of Linux 2.4.17 in terms of buffer consumption and buffer hit ratio.

Key words : Buffer Allocation Scheme, Multimedia Data, Linux

1. 서 론

최근 고성능 마이크로프로세서를 비롯한 하드웨어 기술의 급속한 발전은 컴퓨터 시스템의 성능을 전반적으로 향상시키고 있지만, 상대적으로 하드디스크와 같은 전통적인 저장 매체들의 성능 향상은 다른 시스템에 비하여 한계가 있는 것이 사실이다. 이와 같은 이유로 오늘날의 운영체제는 버퍼 캐시(buffer cache)를 이용하여 디스크 입출력을 가급적 줄임으로써 시스템의 성능 향상을 도모하고 있는데, 특히 LRU(Least Recently Used), LFU(Least Frequently Used), LRU-K, 2Q와 같은 버퍼 교체(buffer replacement) 알고리즘에 관한 연구들이 주목을 받아왔다[1~4]. 또한 최근에는 프리페치(prefetch)가 버퍼 캐시의 성능을 획기적으로 향상시킬 수 있다는 연구결과가 발표되면서 이를 버퍼 교체 기법과 적절히 결합한 연구들이 시도되었다[5~10]. 이러한 전략들이 어느 정도 실효를 거두어 온 것은 사실

· 본 연구는 서강대학교 산업기술연구소 지원으로 이루어졌음
† 비 회 원 : LG전자 CDMA 연구소 연구원
skipio@lge.com
†† 종신회원 : 서강대학교 컴퓨터학과 교수
parksy@ccs.sogang.ac.kr
††† 비 회 원 : 서강대학교 컴퓨터학과 교수
jhyang@ccs.sogang.ac.kr
논문접수 : 2002년 11월 6일
심사완료 : 2003년 3월 22일

이지만, 버퍼 캐시는 기본적으로 평균 수십에서 수백 킬로바이트의 텍스트(text) 데이타를 위해 설계된 시스템이므로 모든 데이타에 같은 성능을 보여 주지는 못한다. 따라서 리눅스와 같은 범용 운영체제에서 멀티미디어 데이타를 효율적으로 처리하기 위해서는 버퍼 캐시 시스템이 멀티미디어의 특성을 잘 반영할 수 있도록 설계되어야 한다.

예를 들면, 범용 리눅스 시스템에서 멀티미디어 데이타에 대한 요구가 빈번한 경우, 멀티미디어 데이타는 지연 시간이 있고, 단위 시간당 일정한 양만큼의 데이타만 필요로 하는데 소비율(consumption rate)을 고려하지 않는 리눅스의 미리읽기(read ahead)는 많은 경우에 필요보다 과도하게 버퍼를 소비한다. 또한 낮은 소비율을 갖는 스트림이 상대적으로 높은 소비율을 갖는 스트림에 버퍼를 반환할 가능성이 높고, 메모리 부족 시 미리 읽은 데이타가 한번도 참조되지 않은 채 반납될 가능성이 크기 때문에 해당 데이타를 참조할 때에는 디스크로부터 다시 읽어 와야 하는 경우가 빈번하게 발생한다. 이는 소비율이 낮은 스트림이 항상 불공정한 자원 배분을 받게 될 뿐만 아니라 시스템 전체적으로도 디스크 대역폭을 낭비하게 되어 과도하게 버퍼를 소비하기 때문에 시스템의 이용률을 저하시키는 원인이 된다. 따라서 멀티미디어 데이타를 보다 효율적으로 처리하기 위해서는 새로운 버퍼 관리 방법이 필요하다.

본 논문에서는 범용 리눅스 상에서 멀티미디어 데이타와 일반 데이타 상호간 불이익을 최소로 줄이면서 효율적인 서비스가 가능하도록 하는 새로운 버퍼 할당 기법을 제안한다[1]. 위에서 제기된 문제점들은 멀티미디어 파일의 소비율을 고려하지 않고, 버퍼 캐시를 전역적으로(globally) 관리하는 데서 비롯된다. 따라서 제안된 기법에서는 사용 중인 멀티미디어 파일마다 미리읽기 캐시(read ahead cache)를 독립적으로 유지하고, 소비율에 근거하여 동적으로 미리읽기 그룹의 크기를 조절함으로써 이러한 문제점들을 해결한다. 리눅스 커널 2.4 이후 버전(2.4.17)에서는 버퍼 캐시가 페이지 캐시(page cache)로 통합되어 메타 데이타(metadata)만을 관리하고 실제 데이타는 페이지 캐시에서 관리하게 되었다. 따라서 미리읽기 캐시는 페이지 캐시와 보다 밀접한 관련을 갖게 되고, 페이지 캐시는 가상 메모리 관리 하부구조와 긴밀한 관련을 맺고 있기 때문에 이를 고려하여 설계되어야 한다. 또한, 프로세스가 멀티미디어 파일의 블록을 읽을 때 입출력 기다림(I/O wait)이 발생하는지를 파악하여 미리읽기를 소비율에 비례하도록 재조정해야 한다.

본 논문에서는 다양한 소비율을 갖는 40개의 멀티미디어 파일을 대상으로 40개의 프로세스를 한 시간 동안 실행하여 버퍼 소비량(buffer consumption)과 버퍼 적중률(hit ratio)을 측정함으로써 구현된 시스템의 성능을 평가하였다. 버퍼 소비량이 적을수록 동시에 보다 많은 멀티미디어 스트림을 처리할 수 있고, 전역적인 LRU 교체기법으로 인하여 멀티미디어 데이타는 낮은 버퍼 적중률을 보이는 경향이 있으므로 버퍼 소비량과 버퍼 적중률을 평가 항목으로 선정하는 것은 의미가 있다.

논문의 구성은 다음과 같다. 2장에서는 멀티미디어 데이타를 위한 기존의 버퍼 관리 기법을 살펴보고 3장에서는 리눅스 커널 2.4.17에서의 버퍼 관리 기법 및 멀티미디어 서비스에 미치는 문제점을 설명한다. 4장에서는 3장에서 제기된 문제점들을 바탕으로 멀티미디어 데이타를 위한 효율적인 버퍼 할당 기법을 제안하며, 5장에서는 구현된 시스템의 성능을 비교 분석한다. 마지막으로 6장에서 결론을 맺으며 본 논문을 마무리한다.

2. 관련 연구

멀티미디어 파일의 특징을 감안한 버퍼 캐시 시스템의 구현은 크게 두 가지로 구분된다. 첫 번째는 기존의 버퍼 정책을 수정하는 것이고[11~14], 두 번째는 버퍼 캐시 시스템을 사용하지 않는 것이다[15]. 본 논문에서는 기본적으로 첫 번째 정책을 사용한다.

기존의 버퍼 정책을 수정하는 방법은 버퍼와 적절한 CPU 및 디스크 스케줄링을 통하여 멀티미디어 스트림을 지원하는 것이다. 이에는 크게 슬랙 타임(slack time)을 이용하는 방법[11]과 버퍼 소비량(buffer consumption)을 이용하는 방법[12~14]이 있다.

슬랙 타임은 스트림이 한 주기 상에서 버퍼를 소비하고 남는 여유시간이다. 다수의 스트림 중 슬랙 타임이 가장 작은 스트림부터 스케줄을 하는 것이다. 버퍼 소비량을 기준으로 하는 방법은 각 스트림의 주기 당 버퍼 소비량을 최소로 되게 스케줄링 하여 남는 버퍼를 효율적으로 사용하는 기법이다. 각 파일마다 주기 당 필요한 블록의 수를 메타 데이타로 관리하여 주기마다 스트림의 버퍼를 다시 할당한다. 이 기법을 사용하면 버퍼 소비량이 최적화되어 시스템 이용률을 극대화 할 수 있다는 장점이 있지만, 스트림마다 별도의 메타데이타를 유지해야 하는 부담이 있다.

버퍼 소비량을 기준으로 하는 방법 중 CTL(Constant Time Length)이 현재까지 연구된 방법 중 가장 효율적

1) 본 논문에서는 버퍼 관리 기법 중 버퍼 할당 기법에 초점을 맞춘다.

이라고 알려져 있다[14,16]. 이 경우 다수의 멀티미디어 스트림은 일정한 서비스 사이클 내에서 돌아가면서 서비스를 받게 된다. 또한, 메타 데이타를 이용하여 각 주기마다 스트림이 실제로 소비할 만큼의 데이타만 디스크로부터 읽어옴으로써 버퍼의 소비량을 최소화하고 있다. N개의 스트림이 서비스될 동안 만약 N+1 번째의 스트림이 서비스를 받기 위해 요청한다면 현재 시스템의 디스크 대역폭과 가용한 버퍼의 수에 의해서 허용 유무를 결정한다. 만약, N+1 번째의 스트림을 서비스하기에는 자원이 부족하지만, 서비스 사이클 내에서 일정 정도 여유 자원이 발생한다면 그 자원만큼 프리페치(prefetch) 하여 현재 서비스를 받고 있는 스트림 가운데 하나를 선정하여 혜택을 주는 방법이 있다. 이때 선정 기준으로 사용되는 것이 스트림의 버퍼 소비량이다. 버퍼 소비량이 최소가 되도록 하는 것이 동시에 보다 많은 스트림을 서비스할 수 있기 때문에 버퍼 소비량을 가급적 최소화시키는 것이 바람직하고 실제 시뮬레이션 결과에서도 이미 증명되었다[13,14]. 이 방법은 메타 데이타를 이용하여 버퍼 소비량이 최소가 되도록 하여 보다 많은 스트림이 동시에 서비스되게 함으로써 시스템 이용률을 극대화 할 수 있다는 장점이 있으나, CPU 및 디스크 스케줄링이 지연 시간(delay sensitive)을 보장해야 한다는 점 때문에 리눅스와 같은 범용 운영체제에 적용하는 것은 별 의미가 없다.

3. 리눅스의 버퍼 관리 기법

리눅스 커널 2.4 이후 버전(2.4.17)에서는 버퍼 캐시가 페이지 캐시로 통합되어 메타 데이타만을 관리하고 실제 데이타는 페이지 캐시에서 관리하게 되었다. 때문에 구현될 미리읽기 캐시는 페이지 캐시와 보다 밀접한 관련을 갖게 되었고, 페이지 캐시는 가상 메모리 관리 하부구조와 긴밀한 관련을 맺고 있기 때문에 이를 고려하여 설계되어야 한다. 3장에서는 리눅스의 버퍼 캐시 및 페이지 캐시 구조와 리눅스가 정규 파일을 읽을 때 시도하는 미리읽기 알고리즘을 살펴본다.

3.1 버퍼 캐시(buffer cache)와 페이지 캐시(page cache)

운영체제에서 페이지 캐시와 버퍼 캐시는 근본적으로 다른 관점에서 발전해왔다. 기본적으로 페이지 캐시는 메모리 매핑(memory mapping) 요구를 처리하는 가상 메모리 관리 하부구조를 위하여 설계되었고, 버퍼 캐시는 입출력 호출(I/O call)을 처리하는 입출력 하부 구조의 성능을 향상시키기 위한 방법으로 사용되었다.

파일 데이타를 처리하는 관점에서 보면 두 캐시는 서로 반-독립적(semi-independently)으로 동작하는데, 이것은 캐시 사이의 데이타 불일치를 초래할 가능성이 크므로 운영체제는 두 캐시를 동기화 해야하는 부가적인 조치를 취해야만 했다. 예를 들면, 리눅스 커널 2.2.x 버전에서는 *read()*의 단위는 페이지였지만, 내부적으로 볼 때는 일단 버퍼 캐시로 해당 파일의 데이타를 얻어온 뒤 다시 페이지 캐시로 복사하는 방식이었고, *write()*시에는 페이지 캐시를 거치지 않고 직접 버퍼 캐시를 통해서만 모든 연산이 수행되었다. 때문에 추가적인 데이타의 복사, 즉 이중 버퍼링(double buffering)이 발생하여 페이지 캐시와 버퍼 캐시 사이에 내용의 불일치(inconsistency)가 발생할 소지가 있었던 것이다. 또한, 데이타를 복사해야 하는 추가적인 비용은 시스템 성능을 저해하기에 충분했다(그림 1 참조).

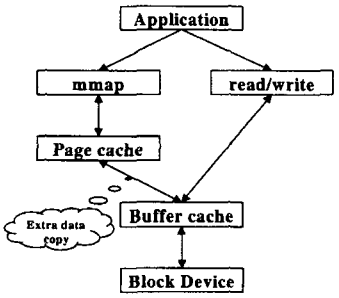


그림 1 버퍼 캐시와 페이지 캐시

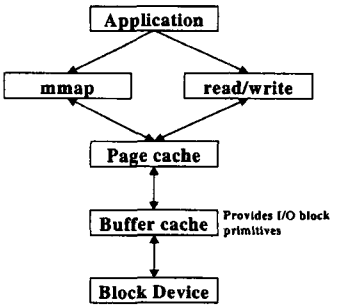


그림 2 통합된 버퍼/페이지 캐시

이러한 문제점들을 해결하기 위해서 리눅스 커널 2.4.10 이후 버전에서는 블록 *read(), write()*까지도 페이지 캐시를 사용하도록 변경되었다. 버퍼 캐시는 메타 데이타만을 유지하게 되었고, 캐시라기보다는 오히려 단순한 입출력 인터페이스로 변경되었다. 이러한 변화로 인하여 메모리 부족 시에 보다 효율적인 가상 메모리의 튜닝(tuning)이 가능해졌고, 데이타의 이중 버퍼링(double buffering)으로 인한 문제점들이 해결되었다(그

림 2 참조).

3.2 미리읽기(read ahead) 알고리즘

리눅스 상에서 미리읽기(read ahead) 알고리즘은 파일의 연속적인(contiguous) 부분에 대응하는 페이지들의 집합을 미리읽기 윈도우(read ahead window)로서 인식하여 미리 읽어 들인다. 미리읽기 윈도우는 항상 마지막 미리읽기 연산으로 수행된 페이지들을 포함하는데, 이러한 마지막 연산에 의해서 얻어진 페이지들을 미리읽기 그룹(read ahead group)이라고 한다. 다음 연산이 미리읽기 윈도우 내에 포함된다면 커널은 파일에 대한 참조가 연속적이라고 판단하여 다음 연산에서는 미리읽기 그룹의 크기를 두 배로 하여 공격적인 읽기연산을 수행한다. 한편, 참조되는 페이지가 연속적이라고 판단되었을 경우, 만약 그 페이지에 락(lock)이 걸려있고 미리읽기 그룹에 포함되어 있다면 긴(long) 윈도우를 사용하고, 락(lock)이 걸려있지 않고 미리읽기 그룹에도 포함되어 있지 않다면 짧은(short) 윈도우를 사용하여 미리읽기를 수행한다. 전체적인 미리읽기 알고리즘은 그림 3을 통하여 볼 수 있다.

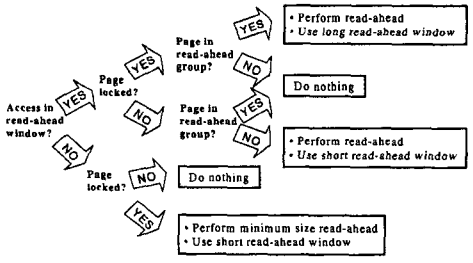


그림 3 미리읽기 알고리즘

3.3 멀티미디어 지원시의 문제점

리눅스에서는 시스템의 메모리가 부족해지면 커널은 페이지 캐시의 비활성화 리스트(inactive list)에서 유지하는 페이지를 버디 시스템(buddy system)으로 반납하여 가용한 메모리를 확보하려고 시도한다. 멀티미디어 데이타 요구가 빈번한 시스템에서 페이지 교체(page replacement)시에는 멀티미디어 스트림의 미리 읽은 페이지들이 과도하게 비활성화 리스트에 존재하게 된다. 리눅스의 미리읽기는 소비율을 고려하지 않고 파일이 연속적이라고 판단되면 공격적으로 데이타를 읽어 오기 때문이다. 따라서 미리읽기로 얻어온 멀티미디어 데이타는 필요보다 과도하게 버퍼를 소비하여 페이지 교체 시에 대상(victim)으로 선택될 가능성이 매우 높다.

예를 들면, 실제 시스템에서는 메모리 할당 요구가 매우 다양한 방식으로 이루어지는데 여기서는 문제를 단순화시키기 위하여 인터럽트나 예외 처리기(exception handler)에 의해서 발생되는 메모리 할당 요구는 제외하기로 한다. 리눅스에서는 메모리가 부족해질 때, 이용가능한 페이지 프레임(free page frame)을 확보하기 위하여 *try_to_free_pages()*라는 함수가 호출된다. 여기서는 *get_free_pages()* 함수에 의한 추가적인 메모리 할당요구 때문에 시스템에서 정한 *freepages.min*보다 가용한 메모리의 수가 적어지는 경우에만 이 함수가 호출된다고 가정한다. 또한 시스템의 전체 메모리를 128KB라고 가정하고 가용한 메모리의 양이 32KB보다 적어지는 경우에 페이지 교체가 실행된다고 하자. 참고로 리눅스에서는 페이지 크기는 기본적으로 4KB로 가정한다. 마지막으로, *try_to_free_ pages()*는 32KB(즉, 8개의 페이지)의 가용한 메모리를 확보한 후에 작업을 완료한다고 하자.

그림 4(a)는 위와 같은 시스템에서 6개의 멀티미디어 파일(f1, f2, . . . , f6)이 동시에 요청되어 수행되고 있는 상황을 보여주고 있다. 미리읽기 그룹의 크기가 최초 1에서 시작한다고 가정할 경우 그림 4(a)에서는 가용한 메모리가 32(128-96=32)KB이므로, 페이지 교체가 작동하지 않지만, 그림 4(b)처럼 또 하나의 파일이(f7) 할당됨과 동시에 시스템에서 유지하는 가용한 메모리(free page)가 32KB이하로 떨어지게 되므로, 페이지 교체를 시작한다. 이때, 가정에 의하여 최소한 32KB의 메모리를 확보할 때까지 페이지 교체가 수행되므로 그림 4(b)의 1, 2, 3, 4, 5, 6, 13, 14번 페이지가 반납될 대상으로 선정될 수 있다. 반납될 대상 중 13번과 14번을 주목해 보자. 이 페이지들은 응용프로그램에 의해서 단 한 번도

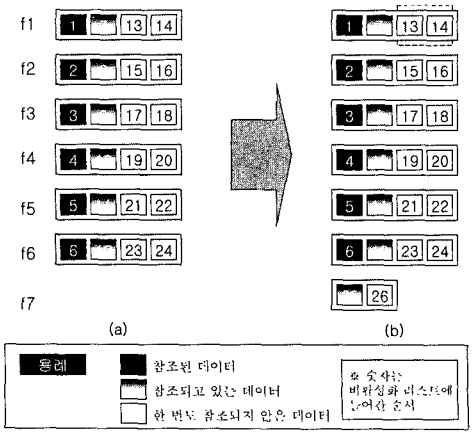


그림 4 페이지 교체 시 미리 읽기 데이타의 불이익

참조되지 않은 상태로 시스템에 반납된다. 즉, 데이타가 메모리로 올라왔다가 한 번도 참조되지 않은 상태로 반납되기 때문에 멀티미디어 파일의 특성상 가까운 미래에 반드시 읽혀져야 하므로 다시 디스크에서 읽어 와야 한다. 시간이 경과하여 미리 읽기 그룹의 크기가 커지면 반납될 대상 중 13, 14번과 같은 미리 읽기 데이타의 비율이 상당히 높아지게 된다. 멀티미디어 데이타는 이미 참조된 데이타가 다시 참조될 확률이 낮고, 오히려 앞으로 읽을 데이타가 더 중요하기 때문에 미리읽기로 읽은 데이타가 한번은 참조될 때까지 보호하는 것이 필요하다.

한편, 낮은 소비율을 갖는 스트림이 상대적으로 높은 소비율을 갖는 스트림에 버퍼를 반환할 가능성이 높기 때문에[16], 낮은 소비율을 갖는 스트림의 미리 읽은 버퍼를 실제 참조할 때에는 다시 디스크로부터 읽어 와야 하는 현상이 빈번하게 발생하여 불공정한 자원 배분을 초래할 수 있다. 이러한 문제점들은 전체 시스템의 디스크 대역폭을 낭비할 뿐만 아니라 과도하게 버퍼를 소비하게 되어 시스템 이용률을 현저하게 저하시키는 원인이 된다.

4. 멀티미디어 데이타를 고려한 지역 버퍼 할당 기법

본 논문은 멀티미디어 데이타와 일반 데이타 상호간 불이익을 최소로 줄이면서[2] 효율적인 서비스가 가능하도록 범용 리눅스의 버퍼 할당 기법을 수정한다. 범용 리눅스에서 멀티미디어 파일을 서비스할 경우 소비율을 고려할 수 없으며, 전역적인 버퍼 사용으로 인한 버퍼 반환 등의 불이익을 해결하기 위하여 멀티미디어 파일의 경우에만 지역버퍼 할당 기법을 적용하고, 할당된 버퍼 크기를 소비율에 근거하여 동적으로 조절하는데 본 논문에서는 소비율을 파악하기 위하여 입출력 기다림(I/O wait)을 사용한다. 반면, 일반 데이타는 기존의 시스템 호출을 그대로 사용하도록 하였다. 일반 파일과 멀티미디어 파일에 대한 참조를 구별하기 위한 방법으로서 본 논문에서는 사용자 프로그램이 힌트 정보(O_RDMULT)를 제공하며, 기존의 시스템 콜(*read()* etc)을 그대로 사용할 수 있도록 수정하였다. 시스템 콜이 발생된 순서에 의하여 파일 서비스가 이루어지므로 본 논문에서 멀티미디어 파일과 일반 데이타 사이의 우선순위는 없다고 볼 수 있는데 이것은 어느 특정 데이타에만 혜택을 주는 것을 가정하고 있지 않기 때문이다. 한편, 멀티미디어 파일의 경우 미리읽기 캐시를 위하여 LIFO 방식의 큐를 유지하고, 동일한 파일을 액세스하는 다른 응용프로그램을 위하여 참조된 페이지는 디스크로 바로 내려가지 않고 전역적인 버퍼로 보내서 프리페치(prefetch)용으로 사용된다.

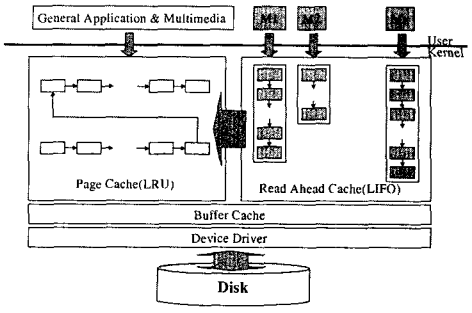


그림 5 시스템의 구조

4.1 시스템의 구조

본 논문에서는 제안된 방법을 구현하기 위하여 리눅스 커널 2.4.17을 수정하여 파일 시스템 캐시를 미리읽기 캐시와 페이지 캐시로 구분하여 설계하였다. 그림 5는 전체 시스템의 구조를 보여주고 있다.

그림 5에서 볼 수 있는 것처럼 새롭게 설계된 미리읽기 캐시는 멀티미디어 데이타를 일차적으로 저장하는 캐시로 응용프로그램에 의하여 한번도 참조되지 않은 멀티미디어 데이타들만 포함한다. 이 캐시는 열린 파일(open file)마다 독립적으로 유지되는 지역 버퍼로 볼 수 있으며 사용자 버퍼로 해당 페이지가 복사되면 본 시스템은 복사된 페이지를 미리읽기 캐시에서 페이지 캐시로 이동시킨다. 즉, 멀티미디어 데이타를 포함하는 페이지들은 일차적으로 LIFO 큐로 관리되며 참조 이후에는 일반 데이타와 마찬가지로 전역적인 LRU 교체기법으로 관리되도록 설계되었다[3]. 페이지 캐시는 리눅스 커널 2.4 버전부터 버퍼 캐시가 관리하는 실제 데이타를 관리하므로, 페이지 캐시를 LRU 교체기법에 기반한 2Q 스타일[3]로 설계하였고, 모든 프로세스에 의해서 공유되도록 하였다. 단, 미리읽기 알고리즘에 의해 메모리에 올라와 한 번도 참조되지 않은 멀티미디어 데이타는 이 캐시에서 관리되지 않는다.

이상과 같이 파일 시스템 캐시를 두 종류로 구분하여 미리 읽은 데이타를 보호함으로써 낮은 소비율을 갖는

2) 본 논문에서 제안한 바에 의하면 특히, 페이지 교체 시에 멀티미디어 데이타가 혜택을 받을 수 있다.

3) 본 논문의 초점은 버퍼 할당 기법이며, 시스템의 구현을 위하여 기존의 버퍼 교체 기법을 그대로 사용했다. 단, 성능 향상을 위하여 MRU 기법 등이 적용될 수 있다.

스트림이 상대적으로 높은 소비율을 갖는 스트림에 버퍼를 반환하는 것을 막을 수 있을 뿐만 아니라 메모리 부족 시에 멀티미디어 데이타가 우선 보호됨으로써 3.3절에서 언급한 문제점들이 많이 해결될 수 있다. 그러나 열린 파일마다 캐시를 유지하는 방법만으로는 멀티미디어를 효율적으로 서비스하는 데에 한계가 있는데 이는 소비율에 민감한 멀티미디어 데이타의 특성을 충분히 반영할 수가 없기 때문이다. 즉, 할당된 캐시의 크기가 소비율 보다 훨씬 크다면 항상 필요보다 더 많은 페이지가 메모리에 올라오게 되어 필요보다 과도하게 버퍼를 소비하게 되므로 시스템 이용률이 떨어지게 되고, 훨씬 작은 크기가 할당된다면 매번 더 많은 페이지를 확보하기 위해서 디스크 입출력을 기다려야 하므로 역시 시스템 성능이 떨어진다. 때문에 미리읽기 캐시를 열린 파일마다 독립적으로 유지하는 것과 더불어 미리읽기 그룹의 크기를 소비율에 비례하여 동적으로 유지하도록 설계하였다.

4.2 버퍼 할당 알고리즘

그림 6은 본 논문에서 제안한 버퍼 할당 알고리즘을 보여주고 있다. 멀티미디어 파일을 읽기 위한 요청이 있을 경우, 새로 설계된 시스템에서는 먼저 페이지 캐시에 데이타가 존재하는지 검사한다. 만약, 페이지 캐시에 데이타가 존재한다면 이미 이 파일의 모든 데이타가 한 번은 참조되었고 아직 버디시스템으로 반납되지 않았다는 것을 의미한다. 멀티미디어 데이타에 해당하는 페이지가 처음 읽혀진 경우 본 시스템은 미리읽기 캐시를 생성하고 참조 후에 페이지 캐시로 반납하게 되는데, 이러한 이유로 해당 페이지가 페이지 캐시에서 발견되었다면, 그 페이지는 미리읽기 캐시에 있었던 것이고 해당 페이지를 읽은 응용 프로그램이 현재 실행 중이거나 아니면 이미 실행을 완료했다는 것을 의미한다. 따라서 이후 페이지들은 미리읽기 캐시에 존재하던지 아니면 페이지 캐시에 존재할 것이다.

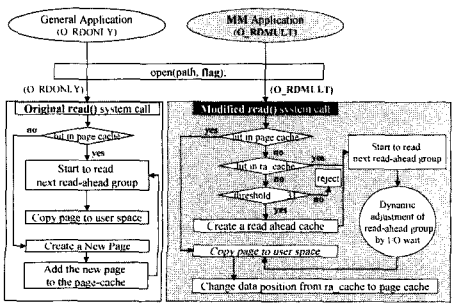


그림 6 버퍼 할당 알고리즘

만약, 페이지 캐시에 존재하지 않는다면 미리읽기 캐시를 검사하고, 여기에도 원하는 데이타가 존재하지 않는다면, 새로운 캐시가 생성되어야 한다. 생성된 캐시는 멀티미디어 파일의 미리 읽기 데이타만을 포함하는데 이것은 응용프로그램에 의해서 한 번도 참조되지 않은 페이지들이다.

미리읽기 캐시가 처음 생성되어야 할 경우 시스템에 존재하는 가용한 페이지의 수(*nr_free_pages*)에서 시스템이 최소한으로 유지해야 하는 가용한 페이지 수(*free-pages.min*)를 뺀 것이 31보다 커야한다. 만약, 이 조건을 만족할 수 없다면 캐시 생성이 허가되지 않는다. 여기서 정한 31이라는 수는 리눅스의 미리읽기 알고리즘에서 사용되는 값을 사용했다.

원하는 데이타가 사용자 프로그램으로 복사되고 난 후에, 해당 페이지는 미리읽기 캐시에서 페이지 캐시의 비활성화 리스트(inactive list)로 위치가 변경되고 그 이후부터는 일반 데이타와 같은 생명주기를 갖는다. 또한, 디스크로부터 읽어오는 미리읽기 그룹의 크기가 멀티미디어 파일의 소비율에 비례하여 동적으로 조절되므로 미리읽기 캐시는 크기가 항상 적절하게 유지된다. 수정된 *read()* 시스템 호출에서는 그림 6에서 볼 수 있듯이 원하는 페이지가 미리읽기 캐시에 존재하는 경우에만 다음 미리읽기 그룹을 읽어오려고 시도한다. 본 시스템에서 멀티미디어 데이타가 페이지 캐시에서 발견되었다는 것은 해당 페이지를 포함하는 파일의 거의 모든 데이타가 이미 메모리(미리읽기 캐시 혹은 페이지 캐시)에 존재할 가능성이 높다는 것을 의미하기 때문이다. 이 경우 미리읽기를 하는 것은 의미가 없고 오히려 자원을 낭비하는 결과를 초래한다. 마지막으로 파일을 닫을 때 미리읽기 캐시를 소멸하게 된다.

4.3 동적으로 미리 읽기 그룹 조정하기

각각의 열린 파일에 할당된 고정된 크기의 미리읽기 캐시는 해당 파일의 소비율에 비례하여 조절되어야 한다. 즉, 소비율이 높은 파일은 소비율이 낮은 파일에 비해서 더 많은 페이지가 할당되어 파일의 소비율에 비례하도록 미리읽기 그룹 크기를 조절하는 것이다. 2장에서 살펴보았듯이, CTL에 기반을 둔 최근의 버퍼 관리 기법은 메타데이타를 이용하는 것을 가정한다. 이 경우 멀티미디어 파일을 미리 스캔하여 메타데이타를 생성해 놓기 때문에 최적화된 버퍼 소비량을 달성할 수 있다. 그러나 서버에 저장된 모든 멀티미디어 파일에 대해서 메타데이타를 유지하는 것은 매우 어려우므로 이 방식은 극히 이론적이고 분석적인 모델이라고 할 수 있다.

본 논문에서는 응용프로그램이 원하는 디스크 블록을

읽을 때 입출력 기다림이 있는지를 체크함으로써 미리 읽기 그룹을 동적으로 조절한다. 최초로 멀티미디어 파일을 읽는 경우 커널은 프로세스의 소비율을 알지 못하므로 미리읽기 그룹은 고정된 크기로 할당된다. 그러나 응용프로그램이 실행됨에 따라 프로세스의 소비율과 미리읽기의 속도에 따라 입출력 기다림이 발생할 수 있다. 즉, 미리읽기 속도보다 소비율이 과다한 경우 일정 시간 이후부터 입출력 기다림이 발생하게 되고, 한동안 계속적인 입출력 기다림 후에 입출력 기다림이 없는 상태로 전이된다. 또한 처음 고정된 미리읽기 그룹의 크기가 소비율에 비해서 상대적으로 크다면, 미리읽기 그룹의 크기를 점차로 감소시키게 된다. 이 경우 한동안 입출력 기다림이 없다가 다시 입출력 기다림 상태로 전환되는 경우가 발생한다. 그러므로 한동안 계속적인 입출력 기다림 후에 입출력 기다림이 없는 상태로 전이되거나 반대로 한동안 입출력 기다림이 없다가 다시 입출력 기다림 상태로 전환되는 경우, 미리 읽기 그룹의 크기가 안정화된 것으로 볼 수 있다. 일단 안정화된 이후에는 미리읽기 그룹이 소비율에 비례하도록 설정되었다고 볼 수 있기 때문에 크기를 고정시킨다. 따라서 각각의 멀티미디어 파일마다 동적으로 최적의 미리읽기 그룹의 크기가 유지될 수 있다.

```
struct file {
    struct list_head f_list;

    struct ra_cache_head  f_rendahead_cache;
    unsigned long              f_flag_io_wait;
    int                        f_pos_io_wait;
    int                        f_fixed;

    struct dentry        *f_dentry;
    struct vfsmount      *f_vfsmnt;
    struct file_operations  *f_op;
    atomic_t       f_count;
    unsigned int         f_flags;
    mode_t               f_mode;
    loff_t               f_pos;
    unsigned long        f_reada, f_ramax, f_raend, f_ralen, f_rawin;
    unsigned int         f_uid, f_gid;
    unsigned long        f_version;
    void                 *private_data;
    long                 f_iobuf_lock;
};
```

그림 7 변경된 파일 오브젝트

제안된 알고리즘을 구현하기 위하여 리눅스 상의 VFS(Virtual File System) 내에서 열린 파일마다 유지되는 파일 오브젝트(file object) 구조체를 그림 7과 같이 수정하여 동적인 미리읽기 그룹의 관리를 위하여 사용하였다. 새로 추가된 필드 중 *f_flag_io_wait, f_fixed, f_pos_io_wait*가 미리읽기 캐쉬를 입출력 기다림에 따라 동적으로 관리하기 위하여 사용된다. *f_flag_io_wait*는 커널에서 제공하는 비트 연산(bit operation)인 *set_bit()*과 *clear_bit()*을 통하여 입출력 기다림을 기록하는 일종의 비트맵(bitmap)이라고 할 수 있고, *f_fixed*는 소비율에 따라 조절되면서 안정화되었는지의 여부를 나타내는 플래그(flag)이며, *f_pos_io_wait*는 해당 비트맵에 기록되어야 할 위치를 나타낸다.

또한, 리눅스 상에서 프로세스가 페이지를 읽을 때 입출력 기다림이 있는지의 여부는 *PG_locked* 비트를 조사함으로써 알 수 있다. 만약, 입출력 기다림이 있다면 파일 오브젝트에 유지되는 비트 변수(*f_flag_io_wait*)의 해당 위치가 1로 설정되고, 기다림이 없다면 0으로 설정된다. 아직 미리읽기 그룹의 크기가 소비율에 비례할 만큼 안정화되지 않은 경우(*f_fixed*가 0인 경우)에 기다림이 있다면 우선 미리 읽기 그룹의 크기를 1 만큼 증가시키고, 없다면 1 만큼 감소시킨다. 그림 8은 제안된 알고리즘을 보여주고 있다.

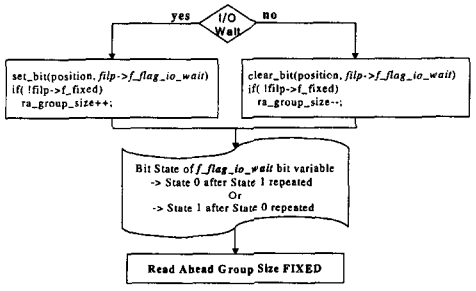


그림 8 동적으로 미리읽기 그룹의 크기 조절하기

5. 성능 평가

제안된 시스템에 대한 성능은 최신의 리눅스 시스템과 소비율을 반영하지 못하는 미리읽기 캐시 시스템, 그리고 동적으로 미리읽기 캐쉬의 크기를 반영하는 시스템 상에서 40개의 멀티미디어 파일을 동시에 실행함으로써 측정되었다. 2장에서 살펴본 CTL(Constant Time Length) 방법은 실시간 스케줄링이 완벽하게 지원되는 것을 가정하기 때문에 리눅스와 같은 범용 운영체제에서는 구현이 어렵다. 따라서 본 논문에서 구현된 시스템의 비교 대상에서는 제외되었다.

5.1 실험 환경

본 실험은 펜티엄(Pentium)II 550MHz CPU와 128 MB의 RAM을 장착하고 있는 시스템에서 수행되었다. 10KB/sec, 30KB/sec, 100KB/sec, 300KB/sec의 소비율을 갖는 총 40개의 멀티미디어 파일을 선정했다. 이것은 다양한 소비율을 갖는 여러 개의 파일이 동시에 서비스될 때 본 시스템이 어떤 작용을 하는지 파악하기

위해서 필요했다. 10개의 프로세스씩 4개의 그룹으로 나누어 각각 10, 30, 100, 300KB/sec의 소비율을 갖는 멀티미디어 파일을 1시간 동안 참조하는 방식으로 진행되었다.

제안된 방법의 효율성을 검증하기 위하여 평균 버퍼 소비량과 캐시 적중률을 평가 항목으로 선택하였다. 지금까지 살펴본 것처럼 기존 리눅스의 버퍼 캐시는 미리읽기와 LRU 교체 기법을 사용하여 전역적으로 데이타를 관리한다. 그러나 멀티미디어 데이타는 참조의 지역성을 가지지 않으며 다양한 소비율을 갖고 있기 때문에 리눅스의 캐시 시스템은 적중률이 낮을 뿐만 아니라 미리읽기의 특성으로 인하여 과도하게 버퍼를 소비한다. 버퍼 적중률이 낮으면 참조 시 디스크 연산을 초래하게 되고 과도한 버퍼 소비는 전체 시스템의 이용률을 저하시키므로 시스템의 성능을 향상시키기 위해서는 버퍼 적중률을 높여야 하며 적정 수의 버퍼만 소비하도록 해야 한다. 따라서 본 논문에서 평가항목으로 평균 버퍼 소비량과 버퍼 적중률을 선정한 것은 의미가 있다.

5.2 리눅스 커널-2.4.17과의 비교

그림 9(a)와 그림 9(b)는 각각 제안된 방식과 리눅스 2.4.17의 평균 버퍼 소비량과 캐시 적중률을 나타내고 있다.

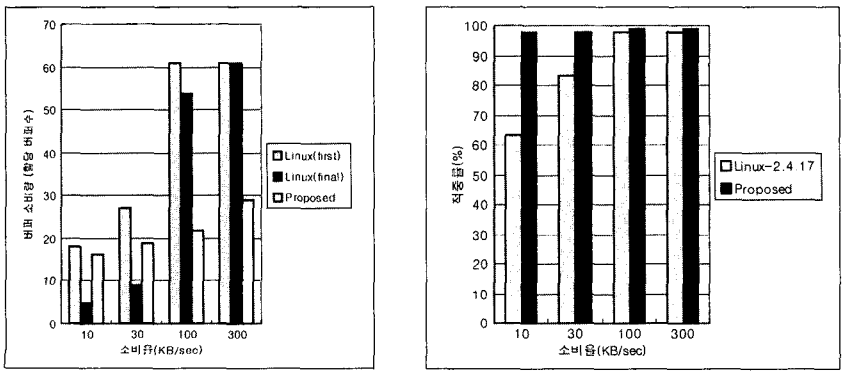
그림 9(a)에서 Linux(first)라고 표기된 것은 최초 스트림에 할당된 버퍼 소비량을 나타내며, Linux(final)은 다양한 소비율을 갖는 스트림의 상호작용으로 최종적으로 할당된 버퍼 소비량을 나타낸다. 그림에서 보는 바와 같이 리눅스 시스템에서 낮은 소비율을 갖는 멀티미디어 스트림은 최초 필요보다 약간 더 많은 버퍼를 할당받지만, 결국 높은 소비율을 갖는 스트림에 버퍼를 반환하여 버퍼 적중률이 현저히 떨어지는 것을 볼 수 있다. 높은 소비율을 갖는 스트림은 상대적으로 높은 버퍼 적중률을 보이지만 과도하게 버퍼를 소비하면서도 제안된 기법의 경우보다 버퍼 적중률이 약간 떨어지는 것을 볼 수 있는데, 이것은 페이지 교체 시에 미리 읽은 데이타가 대거 버디 시스템에 반납되기 때문이다. 그러나 제안된 방법에서는 보다 적은 수의 버퍼를 소비하면서도 거의 100%에 가까운 버퍼 적중률을 보이고 있다.

한편, 제안된 시스템의 경우도 버퍼 적중률이 완벽하게 100%에 도달하지는 않는데 최초 커널 입장에서는 멀티미디어 파일의 소비율을 알 수 없기 때문에 고정된 크기의 미리읽기 윈도우를 사용하다가 동적으로 조절되므로 안정화되기 전까지의 조절기간이 그 원인이라고 판단된다. 제안된 방법은 열린 파일마다 미리읽기 캐시를 독립적으로 유지하여 미리 읽은 멀티미디어 데이타를 보호함으로써 낮은 소비율을 갖는 스트림이 높은 소비율을 갖는 스트림에 버퍼를 반환하는 것을 근본적으로 차단하고 있기 때문에 공정한 자원 배분을 달성할 수 있을 뿐만 아니라 버퍼 적중률을 향상시키고 있다. 또한, 미리읽기 그룹의 크기가 소비율에 비례하여 동적으로 조절되어 버퍼 소비량을 최소화시키므로 동시에 보다 많은 스트림을 처리할 수 있게 되어 시스템 이용률을 향상시킨다.

5.3 미리읽기 그룹의 동적 유지의 효율성

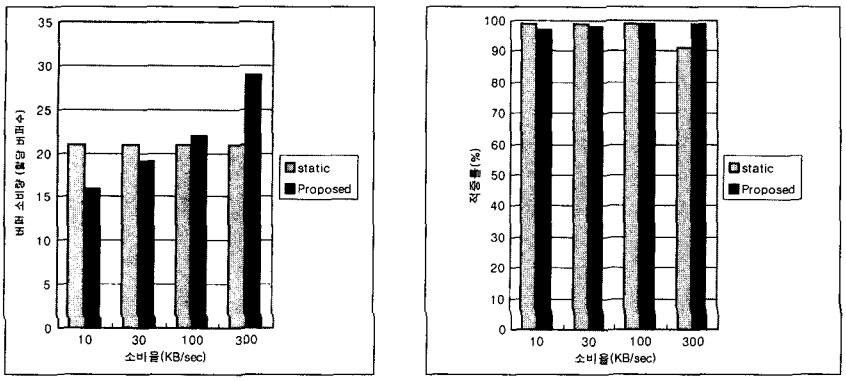
그림 10은 구현된 시스템을 정적으로 미리읽기 그룹의 크기를 유지하는 시스템과 비교함으로써 동적 유지의 필요성을 증명한다.

그림 10에서 보는 것처럼 두 경우 모두 거의 100%에 가까운 높은 캐시 적중률을 보이고 있다. 그러나 정적인 방법에서 높은 소비율(300KB/sec)을 갖는 스트림은 90%까지 버퍼 적중률이 떨어지고 있는데, 이와 같은



(a) 평균 버퍼 소비량 (b) 평균 캐시 적중률

그림 9 리눅스 2.4.17과 구현된 시스템과의 비교



(a) 사용된 평균 버퍼 수 (b) 평균 캐시 적중률

그림 10 정적으로 캐시크기 유지하는 방식과의 비교

현상은 미리읽기 그룹의 크기가 소비율을 따라가지 못하는데 기인한다고 볼 수 있다. 때문에 300KB/sec의 소비율을 갖는 스트림의 경우에는 오히려 제안된 방법이 보다 많은 버퍼를 소비한다. 이것은 높은 버퍼 적중률을 위한 고육지책(trade-off)이라고 볼 수 있다. 하지만, 소비율이 낮은 파일의 경우에는 보다 적은 버퍼를 소비하고 있고, 300KB/sec보다 높은 소비율을 갖는 파일에서는 버퍼 적중률이 더욱 낮아질 것으로 예상되므로 동적으로 미리읽기 그룹의 크기를 유지하는 것이 보다 효율적이다.

본 시스템이 보다 효율적으로 구현되기 위해서는 버퍼 수 및 캐쉬 적중률에 각각 가중치를 두어 필요에 따라서 정적으로 버퍼를 유지하거나 또는 동적으로 버퍼를 유지할 수 있도록 고려할 수 있다.

6. 결론 및 향후 연구방향

본 논문은 멀티미디어 데이타와 일반 데이타 상호간 불이익을 최소로 줄이면서 효율적인 서비스가 가능하도록 범용 리눅스의 버퍼 관리 기법을 수정하여 새로운 버퍼 시스템을 설계하고 구현하였다. 멀티미디어 데이타를 효율적으로 처리하기 위해 연구되어 온 버퍼 관리 기법 중에 대표적인 것이 CTL 방식인데, 이 방식은 여러 면에서 효율적이지만 실시간 스케줄링이 완벽하게 지원되는 것을 가정한다. 때문에 범용 리눅스에 적용하는 것은 의미가 없다. 본 논문에서는 리눅스의 미리읽기와 모든 프로세스가 공유하는 전역적인 캐시유지가 멀티미디어 데이타를 처리할 때 효율적이지 못하다는 데 착안하여 멀티미디어 파일을 위한 미리읽기 캐시를 유지하고, 미리읽기 알고리즘을 보다 효율적으로 개선하여 실시간 스케줄링이 지원되지 않는 환경에서 최대한 멀티미디어를 효율적으로 처리할 수 있도록 하였다.

시스템의 성능을 측정하여 본 결과, 특히 버퍼 소비량이 기존의 리눅스에 비하여 현저하게 개선된 것을 볼 수 있는데 이것은 동시에 보다 많은 멀티미디어 데이타를 처리할 수 있다는 것을 의미하므로 시스템 이용률 측면에서 분명한 이점으로 작용한다. 한편, 캐시 적중률에서도 구현된 방식에서는 거의 100%에 도달하므로 리눅스에서 낮은 소비율을 갖는 데이타가 높은 소비율을 갖는 데이타에 버퍼를 반환하는 불이익이 사라졌다. 따라서 모든 데이타에 비교적 공정한 자원 배분이 이루어지게 된다.

본 연구의 향후 과제로는 다양한 멀티미디어 데이타들의 요구사항, 즉 QoS를 만족시켜 줄 수 있는 버퍼 관리 기법이 개발되어야 하며, 미리읽기 캐시로 인한 기존 텍스트 데이타의 손해를 최소화 할 수 있는 방안 및 미리읽기 캐시의 크기를 동적으로 유지하는 알고리즘의 개선이 필요하다. 본 논문에서 제시한 간단한 알고리즘을 적용하더라도 성능이 향상되는 것을 볼 수는 있지만 보다 최적화된 알고리즘을 개발할 여지는 충분히 있다고 본다. 또한, 성능의 향상을 위하여 멀티미디어 데이타를 위한 효율적인 버퍼 교체 기법(예: MRU, Interval Cashing, DISTANCE 기법 등)[17~19]도 고려되어야 한다.

참 고 문 헌

[1] A. Dan and D. Towsley, "An approximate analysis of LRU and FIFO buffer replacement schemes," Proc. of the 1990 ACM SIGMETRICS Conference, pp. 143~149, 1990.

[2] E.J. O'Neil, P.E. O'Neil, and Gerhard Weikum, "The LRU-K page replacement algorithm for database disk buffering," Proc. of the 1993 ACM SIGMOD International Conferences on Management of Data, pp. 297~306, 1993.
[3] Theodore Johnson and Dennis Shasha, "2Q : A Low Overhead High Performance Buffer Management Replacement Algorithm," Proc. of the 20th VLDB Conference, pp. 439~450, 1994.
[4] Donghee Lee, Jongmoo Choi, and Sam H. Noh, "On the Existence of a Spectrum of Policies that Subsumes the LRU and LFU Policies," Proc. of the 1999 ACM SIGMETRICS Conference, pp. 134~143, 1999.
[5] James Griffioen and Randy Appleton, "Reducing file system latency using a predictive approach," Proc. of the 1994 USENIX Technical Conference (Boston, MA), pp. 197~207, 1994.
[6] R. Hugo Patterrison, "Informed prefetching and caching," Proc. of the 15th ACM Symposium on Operating System Principles, pp. 79~95, 1995.
[7] Tracy Kimbrel, Andrew Tomkins, R. Hugo Pat terson, and Pei Cao, "Trace Driven Comparison of Algorithms for parallel prefetching and caching," Proc. of the second Operating System Design and Implementation(OSDI) Symposium," 1996.
[8] Pei Cao, Edward W. Felten, Anna Karlin, and Kai Li, "Implementation and Performance of Integrated Application Controlled File Caching, Prefetching and Disk Scheduling," ACM Transaction on Computer Systems(TOCS), 1996.
[9] Hui Lei and Dan Duchamp, "An analytical approach to file prefetching," Proc. of the 1997 USENIX Annual Technical Conference, 1997.
[10] H. Seok Jeon and Sam H. Noh, "Dynamic Buffer Cache Management Scheme based on Simple and Aggressive Prefetching," Proc. of the 2000 USENIX Annual Technical Conference, 2000.
[11] D.P. Anderson, U. Osawa, and R. Govindan, "A file system for continuous media," Proc. of the 1992 ACM Transaction Computer System, pp. 311~337, 1992.
[12] S.R. Yeon and K. Koh, "A dynamic buffer management technique for minimizing the necessary buffer space in a continuous media server," Proc. of the International Conference on Multimedia Computing and System, IEEE, pp. 181~185, 1996.
[13] Kun-Lung Wu and Philp S. Yu, "Consumption-Based Buffer Management for Maximizing System Throughput of a Multimedia System," Proc. of the International Conference on Multimedia Computing and System, IEEE, 1996.
[14] I.H. Kim, J.W. Kim, S.W. Lee, and K.D. Chung, "VBR Video Data Scheduling using Window-Based Prefetching," Proc. of the International Conference on Multimedia Computing and System, IEEE, pp. 159~164, 1999.
[15] Milind M. Buddhikot, Xin Jane Chen, and Dakang Wu, "Enhancements to 4.4 BSD UNIX for Efficient Networked Multimedia in Project MARS," Proc. of the International Conference on Multimedia Computing and System, IEEE, 1998.
[16] Jongho Nang and Sungkwan Heo, "An Efficient Buffer Management Scheme for Multimedia File System," Proc. of the IEICE Transaction information and system, vol. E83~D, No 6, 2000.
[17] P.J. Shenoy, P. Goyal, S. Rao, and H.M. Vin "Symphony: An Integrated Multimedia File System," Proc. of ACM/SPIE Multimedia Computing and Networking, pp. 124~138, 1998.
[18] B. Ozden, R. Rastogi, and A. Silberschatz, "Buffer Replacement Algorithms for Multimedia Storage Systems," Proc. of IEEE International Conference on Multimedia Computing and Systems, pp. 172~180, 1996.
[19] 최종무 등, "적응력있는 블록 교체 기법을 위한 효율적인 버퍼 할당 정책", 정보과학회논문지: 시스템 및 이론, 제27권 제3호, 2000.

신 동 재
2000년 연세대학교 전산학과 졸업. 2002년 서강대학교 컴퓨터학과 석사. 2002년 8월~현재 LG전자 CDMA 연구소. 관심분야는 멀티미디어, 운영체제, 분산 시스템

박 성 용
정보과학회논문지 : 컴퓨팅의 실제 제 9 권 제 3 호 참조

양 지 훈
정보과학회논문지 : 컴퓨팅의 실제 제 9 권 제 3 호 참조