임베디드 멀티미디어 재생기에서 속도기반 미리읽기를 이용한 사용자기능 지원 파일시스템

(A File System for User Special Functions using Speed-based Prefetch in Embedded Multimedia Systems)

최 태 영 † 윤 현 주 †
(Tae-Young Choe) (Hyeon-Ju Yoon)

요 약 휴대용 멀티미디어 재생기는 기존의 멀티미디어 파일 서버와는 다른 성질들을 가지고 있다. 개인 사용자 전용, 비교적 낮은 하드웨어 성능, 사용자 기능으로 인한 순간적인 높은 부하, 그리고 짧은 개발주기 등이 이러한 성질들이다. 다양한 멀티미디어 파일 시스템은 여러 사용자의 요구는 처리하기에는 적합하지만 단일 사용자의 특수 기능을 지원하기에는 적합하지 않다. 팁과 같은 추가 정보를 응용프로그램과 파일시스템이 주고받는 방식들도 제안되었지만 프로그램의 개발주기를 증가시킬 수 있다. 본 논문에서는 파일블록배치, 버퍼-캐시, 그리고 미리읽기를 사용하여 휴대용 재생기에서 사용자 기능을 효과적으로 지원하는 파일 시스템을 디자인하고 그 성능을 평가하였다. 힌트를 사용하는 기존의 미리읽기들과는 달리 제안된 미리읽기인 SPRA (SPeed-based PRefetch Algorithm)는 힌트를 사용하지 않고 입출력 요구의 패턴을 통해서 미리 읽을 블록을 예측한다. 이는 응용프로그램이 수정되고 재컴파일 되는 과정을 제거함으로써 프로그램 개발기간을 단축시킨다. 실험결과 SPRA의 평균 반환시간은 리눅스의 추가읽기와 비교해서 4.29%~52.63%이며, 고속 재생 시 리눅스 추가읽기의 1.01~3.09배의 대역폭을 가진다.

키워드 : 임베디드 시스템, 멀티미디어, 사용자 기능, 파일 시스템, 버퍼캐시, 블록 할당, 미리읽기, 압축 데이타

Abstract Portable multimedia players have some different properties compared to general multimedia file server. Some of those properties are single user ownership, relatively low hardware performance, I/O burst by user special functions, and short software development cycles. Though suitable for processing multiple user requests at a time, the general multimedia file systems are not efficient for special user functions such as fast forwards/backwards. Some methods has been proposed to improve the performance and functionality, which the application programs give prediction hints to the file system. Unfortunately, they require the modification of all applications and recompilation. In this paper, we present a file system that efficiently supports user special functions in embedded multimedia systems using file block allocation, buffer-cache, and prefetch. A prefetch algorithm, SPRA (SPeed-based PRefetch Algorithm) predicts the next block using I/O patterns instead of hints from applications and it is resident in the file system, so doesn't affect application development process. From the experimental file system implementation and comparison with Linux readahead-based algorithms, the proposed system shows 4.29%~52.63% turnaround time and 1.01 to 3.09 times throughput in average.

Key words : Embedded System, Multimedia, User Function, File System, Buffer Cache, Block Allocation, Prefetch, Compressed data

† 정 회 원 : 금오공과대학교 컴퓨터공학부 교수
choety@kumoh.ac.kr
(Corresponding author임)
juyoon@kumoh.ac.kr
논문접수 : 2007년 12월 21일
심사완료 : 2008년 9월 1일

정보과학회논문지: 컴퓨팅의 실제 및 레터 제14권 제7호(2008.10)

1. 서 론

휴대용 멀티미디어 재생기는 휴대할 수 있을 정도로 작은 크기로 제작되어야 하기 때문에 대부분의 구성 모듈들은 작은 크기의 모델들이 사용된다. 제작비용을 고려하여 선택되는 모듈들은 개인용 컴퓨터나 그 이상의 컴퓨터 시스템에 비해서 성능이 낮다. 휴대용 재생기가 이용되는 환경은 일반적인 컴퓨터 작업환경에 비해서 다양한 작업을 하기가 힘들다. 이는 휴대용 재생기의 사용자 인터페이스는 재생기 자체의 크기 제한 때문에 버튼과 같은 사용자 인터페이스가 한정되어 있고 정보를 출력하는 화면의 크기도 제한적이기 때문이다. 이러한 제한들로 인해 사용자는 여러 작업들을 동시에 처리하지 못하게 된다. 이와 같이 제한된 상황에서 멀티미디어 동영상 재생은 하나의 미디어 스트림(media stream)에 대해서만 이루어진다고 가정하여도 일반성을 벗어나지 않는다.

멀티미디어 서비스의 효과가 다방면에서 가치가 있음이 인정됨에 따라 인터넷이나 인트라넷을 통한 멀티미디어 서비스가 확대되고 이러한 서비스를 효과적으로 지원하기 위해 많은 파일 시스템들이 제안되었다[1,2]. 파일 시스템을 구성하는 많은 모듈들 중에 디스크 스케줄링, 블록 할당, 그리고 버퍼- 캐시 관리는 파일 시스템의 성능에 많은 영향을 미친다. 하지만 동영상 재생을 위한 사용자 특수 기능들을 제공하기 위해서는 추가 기능이 도움이 된다. 사용자 특수 기능이란 일반적인 VCR(Video Cassette Recorder)이 지원하는 기본 속도 재생, 2/4/8배속재생, 천천히 재생, 뒤로 재생 등의 기능을 말한다. VCR의 이후 세대 모델인 DVD(Digital Video Disk) 재생기도 이러한 다양한 속도의 재생을 지원한다. 디스크를 저장장치로 사용하는 휴대용 멀티미디어 재생기는 이러한 다양한 속도의 재생을 얼마나 효과적으로 제공해 주는가가 소비자가 인식하는 성능 평가 중의 한 가지 요소가 된다. 멀티미디어 스트림에서 다양한 재생속도를 고려하는 것은 [3-5] 등에서 연구되어 왔다.

Ng와 Yeung은 고속 재생으로 인해 변경된 작업 특성이 실시간 스케줄상에서 허용되는 범위를 계산하고, 부드러운 고속 재생을 위해 부가적인 고속재생 동영상 정보를 디스크에 저장하고 이를 접근하는 방법을 제안하였다[3]. 이 방식은 특정한 속도의 고속 재생은 지원하지만 다른 속도는 지원하지 못하는 문제가 있다. Niranjan등은 응용프로그램이 멀티미디어 파일 시스템에 힌트를 제공함으로써 다음에 읽을 블록의 위치를 예측하는 파일 시스템을 제안하였다[4]. 이 방식은 응용프로그램이 힌트를 제공할 수 있도록 디자인 또는 코딩되어야 하며, 기존의 프로그램을 재 컴파일 해야 한다는 문제가 있다. Nam등은 의사 순차 읽기인 경우에 읽지 않고 지나치는 블록을 읽을 것인지를 판단하는 방법을 제안하였다[5]. 이 방법은 일고시간(think time)이 2 ms 이하이고 순차읽기가 아닌 경우에 적용되기 때문에 블록을 읽는 주기가 500ms 이상인 멀티미디어 스트림에서는 일반 읽기와 같은 동작을 보인다.

본 논문에서는 휴대용 멀티미디어 재생기와 같이 제한된 상황에서 사용자 기능을 효과적으로 지원하기 위해서 힌트를 사용하지 않고 입출력 패턴으로 다음 블록을 예측하는 미리읽기 알고리즘 SPRA를 제안한다. 이 알고리즘은 파일 시스템내의 모듈내부에 포함되어서 응용프로그램과 파일시스템은 변경되지 않고 적용될 수 있다. 제 2장에서는 동기 및 관련연구를 설명하고 이의 문제점과 해결방향을 제시하였다. 제 3장에서는 제안된 미리읽기 알고리즘을 소개하고 알고리즘의 특성을 설명하였다. 제 4장에서는 제안된 알고리즘의 성능을 평가하기 위하여 구성된 실험환경을 설명하고 실험 결과를 분석하였다. 마지막으로 제 5장에서 결론과 추후 과제로 끝맺는다.

2. 동기 및 관련연구

멀티미디어 재생과 같이 연속적으로 디스크에 접근(sequential access)하는 경우라면 추가읽기(read-ahead)만으로도 입출력 성능을 향상시킬 수 있다. 여기서 추가읽기 방식이란 읽어야할 블록뿐만 아니라 그 뒷부분을 같이 읽어서 버퍼에 저장해 두고 나중에 이 부분에 대한 입력요구가 발생하면 디스크를 사용하지 않고 버퍼에 있는 블록을 돌려주는 것을 말한다. 유닉스에서는 기본적으로 추가읽기를 적용하여 디스크 접근 횟수를 줄이는 효과가 있음으로써 전체 입출력 성능을 높인다. 하지만 그림 1의 (b), (c), 그리고 (d)와 같이 순차적 읽기가 아닌 의사 순차적 읽기(pseudo sequential read)가 진행되면 추가읽기의 효율이 나빠진다. 의사 순차적 읽기는 다음에 접근하는 블록이 이전에 접근한 블록 이후에 있지만 인접하지는 않은 경우를 말하며 고속 미디어 재생을 하는 응용프로그램이 생성한다. 효율이 나빠지는 이유는 이전 블록과 다음에 요구된 블록사이에 간격이 있기 때문에 미리읽기를 통해 버퍼에 저장된 부분이 활용되지 못하고 버려지는 문제가 발생한다. 이런 경우 디스크가 사용되지 않는 시간에 다음에 읽을 블록을 예측하여 미리 읽은 후 버퍼에 저장하는 미리읽기(prefetch)가 보다 효과적이다. 멀티미디어 재생에 관련된 미리읽기 알고리즘은 멀티미디어 서버의 설계와 더불어 다수가 제안되었다.

Nemesis 시스템은 클라이언트 프로그램 내에 큐

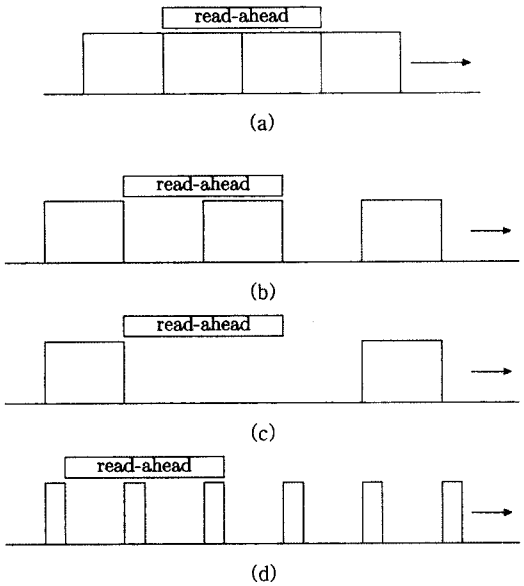


그림 1 순차적 읽기와 다양한 의사 순차적 읽기에 따른 추가읽기(read-ahead) 효과: (a) 순차적 읽기, (b) 1 블록을 건너 뛴 의사 순차적 읽기, (c) 3 블록을 건너 뛴 의사 순차적 읽기, (d) 블록의 크기와 건너뛰는 블록의 크기가 변경된 의사 순차적 읽기

(queue)를 유지하면서 다음에 재생할 프레임을 미리 큐에 넣어둔다[6]. 큐에는 유지되어야 할 프레임의 개수가 정해져 있다. 프레임 수가 감소하면 미리읽기 프로세스는 큐에 넣어야 할 프레임의 크기와 위치를 동일한 프로그램 내의 자료구조에서 얻어서 파일서버에 요청한다. Revel등이 제안한 TAP은 이전 작업들이 I/O를 처리하는데 걸린 시간과 미리읽기되어 버퍼에 있고 아직 요청이 처리되지 않은 블록들의 개수를 참조한다[7]. TAP은 이들 정보를 바탕으로 다음에 몇 개의 블록(prefetch depth)을 요청할 것인가를 결정한다. 이러한 시스템은 미리읽기되어야 할 블록은 항상 다음 블록으로 고정되어 있기 때문에 의사 순차적 읽기 같은 입출력 형태에서는 좋은 효율을 기대하기가 어렵다.

멀티미디어가 아닌 일반적인 유닉스 환경에서 TIP이라는 미리읽기 관리자를 사용하여 미리읽기의 성능을 높이는 방법도 제안되었다[8]. 응용프로그램은 가상의 디바이스를 통해서 TIP에게 힌트를 주고 TIP은 그 디바이스에서 받은 팁을 통해 어느 블록을 미리 읽을 것인지를 판단한다. Revel등은 미리읽기의 정확성을 얻기 위해 응용프로그램이 미리읽기 함수를 호출하고, 이 미리읽기 함수들은 라이브러리의 형태로 제공하여 멀티미디어 재생 프로그램의 변경을 줄이는 방법을 제안하였다[9]. 미리읽기를 요청하는 기능을 시스템 호출(system call)의 형태로 구현한 경우도 있다[10]. 이들 미리읽기는 응용프로그램이 힌트를 파일시스템에게 제공해야 하기 때문에 힌트를 사용하지 않은 기존의 응용프로그램의 소스 코드가 변경되고 컴파일 될 수 있는 상황에서만 적용 가능하다는 문제점이 있다.

Vellanki와 Chervenak은 접근된 블록열(block sequence)들을 하나의 스트링으로 보고 이를 Lempel-Ziv 스킴(scheme)으로 구성된 미리읽기 트리에 저장하는 자료구조를 구상하였다[11]. 트리의 각 노드는 가중치가 있어서 현재까지의 블록들의 접근 패턴을 바탕으로 가중치가 높은 블록을 선택하여 그 위치에 있는 블록을 미리읽기한다. 이 방식은 기존의 패턴들이 다시 발생하는 경우에는 효과가 크지만 순차적인 접근과 재생속도의 변경이 발생하는 경우에는 미리읽기의 효과를 얻지 못한다.

자료구조를 이용한 다른 미리읽기 방법은 이전에 접근했던 블록들의 주소들의 패턴을 블록테이블(Block table)이라는 자료구조에 저장한다[12]. K번 연속으로 읽은 블록들이 블록테이블 상의 한 체인에 있으면 그 체인상의 다음 블록이 미리읽기된다. 이 논문은 상당히 큰 규모(약 2백만 개의 LBN)의 블록테이블을 유지함으로써 일단 구축된 블록체인들에 대해서는 미리읽기가 상당히 좋은 성능을 보인다. 하지만 멀티미디어 스트림의 고속 재생 시와 같이 이전에 읽지 않고 인접하지 않은 블록을 읽는 경우에는 미리읽기의 효과가 거의 없다.

Nam등은 비연속적인 스트림이 주어지면 스트림의 특성을 예측하는 알고리즘을 제안했다[5]. 실측기반 성능이득 데이타베이스를 구축하여 I/O 요구의 특성으로부터 스트라입(striped)된 I/O 블록들 사이의 읽을 필요없는 블록을 읽을 것인지 무시할 것인지를 결정한다. 이 논문은 재생 속도와 같이 I/O 요구의 특성이 변화하는 상황과 I/O 블록과 블록 간 간격이 일정하지 않은 경우는 고려하지 않았다.

미디어가 재생되는 동안 재생속도가 사용자에 의해서 수시로 변경되는 경우에는 힌트를 사용하더라도 그림 2에서와 같이 사용자의 행동에 따른 다름 블록을 예측하기가 쉽지 않다. 뿐만 아니라 재생속도가 변경되었으면 다음 블록의 예측은 변경된 재생속도를 반영하여 예측할 수 있어야 한다. 다음 블록의 위치에 대한 예측을 어

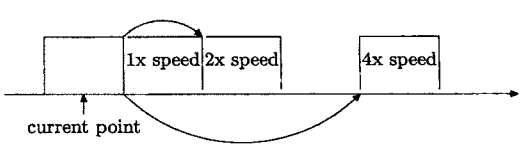


그림 2 다음 블록 읽기의 발생 가능한 위치들

렵게 하는 또 다른 문제는 다음에 발생한 블록이 예상했던 블록이 아니었다면 예상실패의 원인이 사용자가 재생속도를 바꾸어서 발생한 것인지 사용자가 재생위치를 바꾼 것인지 정확히 알기가 어렵다는 것이다.

본 논문에서는 응용프로그램에서 발생하는 I/O 요구의 패턴을 토대로 다음에 발생할 I/O의 위치를 예측한다. 예측된 블록은 미리읽기 기능에 의해 디스크의 유휴시간에 읽혀진다. 응용프로그램으로부터 힌트와 같은 정보를 받지 않기 때문에 기존 프로그램들을 수정없이 사용할 수 있다. 또한 인접하지 않은 블록을 예측을 통해 미리읽기하기 때문에 다양한 재생 속도를 가지는 미디어 재생 프로그램의 I/O 요구를 효율적으로 처리할 수 있다.

3. 제안된 미리읽기 알고리즘

임베디드 시스템에서 다양한 재생속도를 지원하기 위해서 본 논문에서는 속도에 기반을 둔 미리읽기 알고리즘(Speed-based PRefetch Algorithm, SPRA)을 제안한다. 그림 3은 SPRA 알고리즘을 의사코드(pseudo code)의 형태로 보여준다. 읽을 데이타의 디스크상의 블록 위치 (b), 블록 크기 (s), 그리고 저장될 메모리 위치 (d)가 주어질 때 미리읽기 입출력을 비동기적으로 생성한다. 여기서 $speed_0$는 직전의 SPRA 호출에서 계산된 speed 값이다. 함수 $f()$는 byte 주소를 페이지 주소로 바꾼다. SPRA는 크게 두 부분으로 이루어져 있는데,

```
Algorithm SPRA(b, s, d)
Begin
  if ((b, s) exists in buffer) then
    copy it from buffer to memory d;                    1
  else
    synch read blocks cover (b, s, d);                  2
    copy it from buffer d_b to memory d;                3
  end if
  speed ← b - b';                                       4
  if (|speed - speed_0| ≦ ε) then
    B_0 ← f(b + speed);                                 5
    B_1 ← B_0 + 1;                                      6
  else
    B_0 ← f(b + speed);                                 7
    B_1 ← f(b + speed_0);                               8
  end if
  if (acceptable I/O) then
    asynch read block B_0;                              9
    asynch read block B_1;                              10
  end if
End
```

그림 3 속도에 기반을 둔 미리읽기 (SPRA) 알고리즘

첫 번째는 주어진 I/O를 동기적으로 처리하거나 버퍼를 관리하는 버퍼 부분이고, 두 번째는 I/O들을 통해서 미리읽기할 블록의 위치를 예측하고 비동기적 I/O를 생성하는 미리읽기 부분이다.

버퍼 부분은 일반적인 버퍼캐시와 유사하다. 버퍼에 읽고자 하는 블록이 있으면 버퍼로부터 메모리 카피를 통해서 데이타를 이동시키고, 그렇지 않은 경우에는 디스크로부터 직접 읽어들인다. 이 때 디스크로부터 읽는 블록의 크기는 SPRA를 호출할 때 발생하는 사용자의 블록크기인 s가 아니고 SPRA의 페이지 크기인 s_b이다. SPRA가 요구하는 입출력 블록의 크기는 항상 s_b이며 이 크기는 향상된 입출력 속도를 얻기 위해서 트랙의 배수 또는 실린더의 크기로 정한다. 새로운 페이지를 버퍼에 읽는 경우에 빈 공간이 없으면 버퍼내의 블록들 중 하나를 쫓아내야 한다. 버퍼 페이지 교체 방식들 중 Least Recently Used(LRU) 방식이 보편적으로 사용되는 알고리즘이지만[13] 멀티미디어 스트림과 같이 연속적인 또는 단방향적인 입출력 요구에 대해서는 LRU는 성능상 이득이 없다[14]. 본 논문에서는 LRU의 부하를 없애고 동일한 성능을 얻기 위해서 FCFS 방식의 버퍼 관리를 한다. 연속된 블록을 읽거나 일정한 방향으로 블록읽기가 계속될 때 FCFS는 LRU와의 성능 차이가 없으며 추가 부하는 거의 존재하지 않는다.

미리읽기 부분은 이전 SPRA에서 사용된 디스크상의 블록 위치 b'와 지금 SPRA에서의 파라미터인 블록 위치 b를 통해 블록 요구간의 거리를 변수 $speed$에 계산한다. 변수 $speed$를 이용하여 다음 예상 블록을 포함하는 두 페이지의 위치 B_0, B_1을 구하고 비동기적으로 I/O를 요청한다. 두개의 페이지를 읽는 이유는 응용프로그램이 요구하는 블록이 압축된 데이타이기 때문인데, 압축된 데이타들은 그 위치와 크기가 일정하지 않아서 페이지보다 크기가 작더라도 두개의 페이지 상에 걸칠 가능성이 많다. 두 번째 if 조건문에서 참조한 $speed_0$는 이전 SPRA에서 계산된 $speed$이며, 이 값과 지금 SPRA의 $speed$값이 어느 정도 다르다는 것은 사용자가 재생속도를 바꾸었거나 재생되는 위치를 이동했다는 것을 의미한다. 재생속도가 바뀐 경우에는 $speed$를 통해서 예측된 블록이 정확하고, 재생되는 위치가 바뀐 경우에는 $speed_0$를 사용하여 예측하는 것이 바람직하다. 하지만 응용프로그램으로부터 힌트를 받지 않는 SPRA로서는 어떠한 경우인지 판단할 수가 없다. 따라서 각 speed에 대해서 예측한 블록을 포함하는 페이지를 하나씩 읽는다.

미리읽기는 디스크 I/O에 여유가 있을 때 하는 것이 바람직하지만 미리읽기한 페이지를 다음 읽기에서 사용하는 것이 확실하다면 입출력에 여유가 없어도 상관없

다. 하지만 미리읽기한 페이지가 다음 읽기에서 사용되지 않아 캐시 미스가 발생하면 다음 읽기는 직접 디스크 입출력을 해야 하고, 그 미리읽기로 인해 다음 읽기가 지연되면 예상 마감시간에 종료하지 못할 수 있다. 이러한 지연을 막기 위해 그림 3의 3번째 if 조건문은 I/O를 수행해도 괜찮은지를 살펴보는 'acceptable I/O' 검사를 한다. 이 검사는 두개의 블록 I/O가 다음 주기에 발생할 I/O의 마감에 영향을 끼칠 수 있는지를 예상한다. 예측한 두 페이지 I/O 시간이 유휴시간보다 작으면 I/O를 발생시킨다. 유휴시간은 다음 I/O 발생 예측시각에 지금 시각을 빼서 계산하고 페이지 I/O의 수행 시간 예측은 최근 10회의 페이지 I/O 시간들 중 최대값으로 정한다. 수행 시간 예측에 10회의 페이지 I/O를 참조한 이유는 작은 I/O 횟수의 참조는 이전 발생한 I/O를 너무 빨리 무시하는 경향이 있고, 이보다 많은 I/O 횟수의 참조는 오버헤드를 발생시킨다. 본 연구에서는 여러 개의 참조 횟수를 테스트하여 이전 I/O의 영향도 반영하면서도 수행시간에 영향을 크기 미치지 않는 횟수를 선택하였다. 10회를 초과한 참조에서는 특별한 성능상의 차이가 보이지 않았다. I/O 발생 예측시각은 최근 I/O 발생 시각에 최근 I/O 발생 시각과 이전 I/O 발생 시각을 더하여 계산한다. 그림 3의 의사코드는 간단한 if 문장이지만 구현된 코드에서는 유휴시간의 크기에 따라서 첫 번째 페이지만 처리할 것인지 두개의 페이지들을 모두 처리할 것인지를 판단한다.

미리읽기 알고리즘은 다음의 두 항목으로 그 특성이 표현된다: '어떤 블록을 읽을 것인가'와 '언제 그 블록을 읽을 것인가'이다. 우선, *어떤 블록*을 읽을 것인가를 보자. 읽을 블록은 이전의 두 I/O를 바탕으로 구한 speed 값을 통해 다음 블록을 선택한다. 즉, 그림 3의 4,5번 줄과 같이 다음 I/O 요구의 시작위치는 $2b-b'$이고 이위치를 포함하는 블록이 대상이 된다. 두 번째로 *언제 블록*을 읽을 것인가는 디스크의 예상 유휴시간이 다음 블록을 읽는데 걸리는 시간보다 클 것으로 예측되면 현재 I/O 작업이 끝나는 즉시 읽기 요구를 발생시킨다. 디스크의 예상 유휴시간이 예상 I/O 시간보다 작더라도 읽기 요구가 발생할 수 있다. 예상 I/O과 예상 유휴시간의 차이만큼 지연이 발생하더라도 다음 I/O가 자신의 작업을 마감 시각 내에 처리할 수 있을 것으로 예측되면 읽기 요구를 발생시킨다. 따라서 p를 예상 주기, t_e를 다음 I/O 예상 시각, t_c를 현재 시각, T_{io}를 블록 입출력 예상 시각이라고 하면 다음 식 (1)을 만족하면 다음 페이지에 대한 읽기 요구가 허용된다:

$$p+t_e-t_c \geq 3\times t_{io}. \quad (1)$$

여기서 좌변은 지금 주기 동안 남은 시간과 다음 주기를 합한 시간이다. 이 시간동안 한 번의 미리읽기와 미스로 인한 두 번의 직접읽기가 종료되면 다음 주기까지는 지연이 발생하지 않기 때문에 T_{io}가 최대 3번 발생하는 것을 허용하면 된다.

4. 성능 평가

4.1 실험 환경

제안된 SPRA 알고리즘은 그 성능을 평가하기 위하여 Linux Fedora 6에서 C 프로그래밍 언어를 사용하여 구현하였다. 실험이 수행된 컴퓨터는 Intel Pentium 1.2GHz이며 하드디스크의 사양은 표 1과 같다. 이 디스크는 24개의 존(zone)으로 구성되어 있어서 각 존마다 트랙당 섹터의 개수가 다르다.

표 1 HITACHI 디스크 사양

디스크 모델	HITACHI Travelstar 5K100 2.5''
용량	40 Gbytes
회전 속도	5400 RPM
최대 전송 속도	493 Mb/s
평균 탐색 시간 (읽기)	12 ms
트랙당 섹터수 (최대)	880
트랙당 섹터수 (최소)	462

그림 4는 성능 평가 실험에 사용된 SPRA 알고리즘과 Linux 모듈의 구조를 보여준다. I/O 생성자(I/O generator) 모듈은 사용자로부터 재생, 정지, 고속재생 등의 명령어를 받아 이를 스트림 형태의 주기적 I/O 블록 요구로 변환한다. 본 실험에서는 압축된 동영상을 직접 접근하지 않고 평균 300 Kbyte에 표준편차가 100 Kbyte인 지수분포 크기의 블록을 요구하도록 한다. 응용프로그램이 압축된 데이타를 읽는 상황이기 때문에

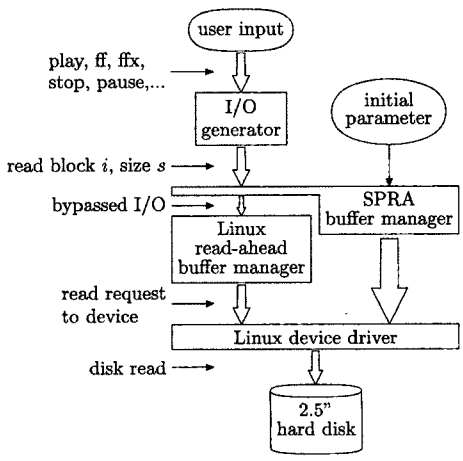


그림 4 실험에 사용된 소프트웨어/하드웨어 모듈들의 연관도

사용자의 추가적인 명령이 없더라도 요구되는 블록의 크기는 일정하지 않다. 이 모듈이 생성하는 스트림 I/O는 4.2절에서 언급할 3 가지 스트림 형태이며 실행 전에 주어진 파라미터 값에 따라 어떠한 스트림 형태로 바꿀 것인지가 정해진다. I/O 생성자가 생성한 각 I/O 요구는 SPRA 버퍼 관리자(SPRA buffer manager)에게 전달된다. 버퍼 관리자는 파라미터 값에 따라 이 요구를 자신이 처리할 것인지 아니면 Linux의 미리읽기 버퍼 관리자(Linux read-ahead buffer manager)에게 전달할 것인지가 정해진다. Linux의 미리읽기 관리자 내의 미리읽기 크기는 루트 레벨에서 수행되는 hdparm 프로그램에 의해 미리 결정되어 있다. 각 버퍼 관리자는 자신의 버퍼 내에 가지고 있지 않은 I/O 요구는 자신의 블록 요구로 바꾸어 Linux의 장치 관리자(Linux device driver)에게 전달한다.

4.2 입출력 패턴 생성

요구되는 멀티미디어 스트림은 압축된 데이타를 읽어들이기 때문에 그 크기가 가변적이다. 본 논문에서는 실제 멀티미디어 재생 프로그램으로부터 I/O 요구를 처리하는 대신 압축된 멀티미디어 스트림의 형태로 시뮬레이트하기 위해 지수분포의 블록 크기에 대한 멀티미디어 스트림 요구를 생성시켰다. 실험에서 사용된 멀티미디어 스트림은 다양한 고속 재생과 재생속도 변경시의 성능을 평가하기 위해 5초 단위로 속도가 증가하는데, 1배속에서 시작하여 두 배씩 재생속도가 증가하고, 32배속 이후에는 16배속씩 속도가 증가한다. 최대 속도는 96배속으로 한다.

일반적인 멀티미디어 응용프로그램이 생성하는 스트림 I/O의 형태는 다양하다. 재생속도가 1배속일 때 b 크기의 블록을 읽어야 한다면 n 배속일 때는 nb에 해당하는 양의 블록을 *관리*해야 한다. 가장 부드러운 영상을 재생하기 위한 단순하고도 거친 방법은 nb에 해당하는 모든 블록을 읽어서 재생하는 것이다. 하지만, 이는 많은 부하를 발생시키는데, 우선 디코딩 작업에 많은 연산 부하가 발생하고, 두 번째로 필요한 양의 블록들을 모두 디스크로부터 읽어 들이는데 많은 시간이 소요될 수 있다. 이를 보완하기 위해 여러 논문들이 제안되었는데[3], 본 논문에서는 n개의 연속된 블록들 중 첫 블록만 읽고 나머지 n-1개의 블록들은 뛰어넘는 방법과 n개의 각 블록들에서 b/n 크기의 작은 블록들만을 읽어서 처리하는 방법을 사용한다.

주기(period) p, 평균 블록크기 b, 이전 블록이 끝나는 지점과 다음 블록이 시작되는 지점의 거리를 사용하여 멀티미디어 스트림의 요구 형태를 표현할 수 있다. 블록간 거리는 평균 블록크기의 배수로 표현하기로 한다. 정상 속도 멀티미디어 스트림은 블록간 간격이 0이므로 (p, b, 0)으로 표기할 수 있으며, 재생 속도 n이 주어지면 멀티미디어 스트림은 다음과 같은 3가지 종류들 중의 하나가 된다. 여기서 재생 속도 n은 양의 정수 값을 가진다고 가정한다.

- (p/n, b, 0): 재생 속도가 증가함에 따라 요구하는 블록의 양도 비례해서 증가하는 스트림이다. n-배속이 되면 1-배속일 때 1개의 블록을 요구할 시간에 n개의 블록을 요구하게 된다. 즉, 주기가 p/n이 된다. 읽어 들이는 블록들은 인접해 있기 때문에 블록간의 간격은 0이 된다. 멀티미디어 스트림 재생 프로그램 입장에서는 연속된 고속 영상을 보여줄 수 있지만 처리에 필요한 CPU 파워가 많이 요구되고 입출력 부하가 크다. 재생속도가 빨라질수록 주기가 짧아져 지연이 발생하거나 마감을 맞추지 못할 확률이 증가한다.
- (p, b, n-1): 주기가 고정된 채 정상속도에서 읽는 크기의 블록만을 읽고 나머지 블록들은 건너뛴다. 이 패턴은 1초간 시청자가 화면을 판독할만한 시간을 주고, n-1초간을 뛰어넘는 것을 반복하는 방식이다. 한 주기 내에 n-배속으로 재생이 진행되면 n개의 블록을 처리해야 하는데, 이 스트림에서는 첫 블록만 읽고 나머지 n-1개의 다음 블록들은 읽지 않고 건너뛴다. 디스크 입출력에 부담되는 부하는 적지만 재생되는 영상은 연속성이 떨어지는 영상이 된다.
- (p/n, b/n, (n-1)/n): 재생 속도가 증가함에 따라 각 블록에서 읽는 양이 감소한다. 재생속도에 따라 적합한 위치에 있는 I 프레임이나 P 프레임만 읽는 방식이다. 주기는 p/n으로 줄어들고, 각 주기에서 읽을 양도 p/n으로 줄어든다. 이 주기 동안 1개의 b 크기의 블록을 처리하면 되는데, p/n만큼 읽고, 나머지 b(n-1)/n은 건너뛴다. 즉, 이미 읽은 블록과 다음 읽을 블록간의 거리는 (n-1)/n개의 블록이 된다. 이 경우, 블록들이 스트림 전체에 고르게 분포되어 있기 때문에 재생되는 영상이 비교적 자연스럽다. 하지만, 작은 크기의 블록들을 여러 번 읽어야 하는 부담이 있다.

그림 5는 정상속도와 2배속재생에서 각 멀티미디어 스트림의 입출력 형태에 대한 예들을 보여준다. 그림 5 (a)는 연속 할당된 멀티미디어 스트림을 1배속으로 읽는 경우에 읽혀지는 디스크 블록들을 보여준다. 본 논문에서 다루는 압축된 멀티미디어 스트림의 경우에는 동일한 크기의 블록을 읽지는 않지만 편의상 같은 크기로 표현하였다. 그림 5(b)는 2배속재생의 경우에 ($p/2$, b, 0) 스트림 형태에 의해 읽혀지는 디스크 블록들을 보여준다. 이 형태의 입력 요구는 한 번에 요구하는 블록의 크기는 b로 동일하지만 주기가 $p/2$가 된다. 멀티미디어

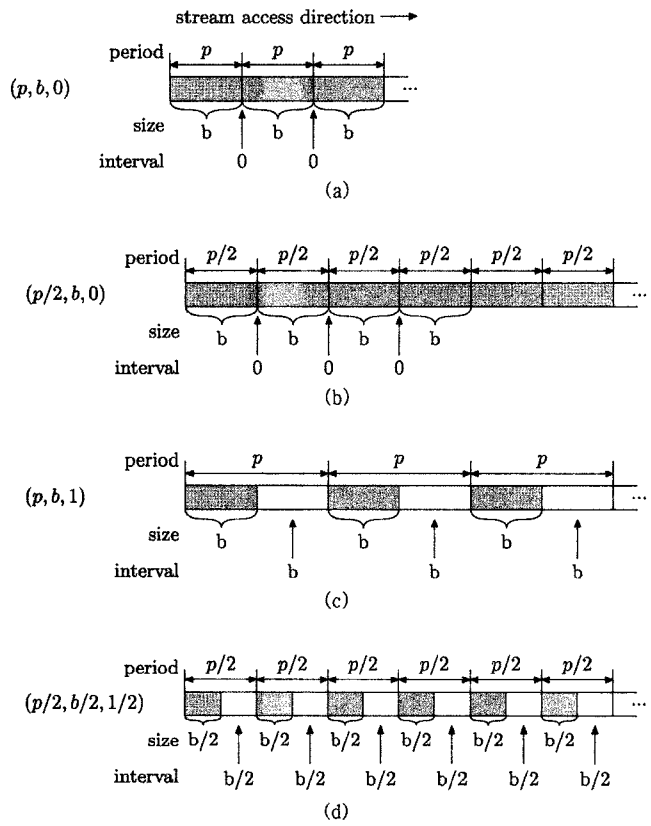


그림 5 각 입출력 요구 방식에 대한 (a) 기본 속도의 일반적인 연속읽기, (b) $(p/n, b, 0)$ 입출력 요구에서의 2배속 읽기, (c) $(p, b, n-1)$ 입출력 요구에서의 2배속 읽기, (d) $(p/n, b/n, (n-1)/n)$ 입출력 요구에서의 2배속 읽기

파일이 디스크 내에 연속적으로 배치되어 있는 경우에는 디스크의 최대 대역폭에 의해서 제공되는 최대 재생속도가 결정된다. 그림 5(c)는 2배속재생일 때 $(p, b, 1)$ 스트림 형태에 의해 읽혀지는 디스크 블록의 모습을 보여준다. 크기 b의 블록을 읽은 후에는 크기 b의 1개 블록은 읽지 않고 지나친다. 디스크로부터의 대역폭은 1배속일 경우와 동일하다. 그림 5(d)는 2배속재생일 때 $(p/n, b/n, (n-1)/n)$ 스트림 형태에 의해 읽혀지는 디스크 블록의 모습을 보여준다. 시간 p 동안에 처리해야 할 범위인 $2b$의 블록에 대해 주기를 반으로 각각 줄이고, 줄어든 주기 동안 1배속일 때 읽는 블록크기의 반인 $b/2$를 읽어 들인다. $(p/2, b, 0)$ 스트림 형태를 제외하면 모든 재생 배속에 대해 대역폭은 동일하다.

1배속일 때 읽는 블록의 평균 크기가 300Kbyte 이면 15-배속일 때 $(p/n, b/n, (n-1)/n)$ 스트림은 $(p/15$, 20 Kbyte, $14/15)$이 되어 한 번에 읽는 양이 많아야 압축된 하나의 프레임을 읽어 들이는 정도가 된다. 더 작은 크기의 블록은 읽어서 재생하는 의미가 없어지게 되므로 재생속도에 무관하게 읽는 블록의 크기에는 최소값이 존재한다. 본 실험에서는 블록 크기가 20Kbyte보다 작아지는 m배속의 재생에서는 $(p/15$, b/15, $(m-1)/15)$의 스트림으로 바꾸어서 수행한다. 즉, 재생속도 n이 15-배속 이상의 재생에서는 주기 p 동안에 mb 크기의 범위 내에서 15개의 $b/15$ 크기의 블록들이 읽혀지게 된다.

4.3 추가읽기 및 미리읽기 알고리즘

대부분의 미리읽기 관련 연구들은 응용프로그램이 파일시스템에 정보를 제공함으로써 다음에 접근할 블록을 예측하거나 같은 접근 패턴이 발생할 때 효과를 볼 수 있도록 설계되어 있어서, 재생 속도가 변화하고 접근하는 블록의 크기나 블록간의 거리가 다양한 경우는 고려하지 못한다는 문제가 있다. [5]에서 제안하는 파일 시스템은 1~3ms의 시간간격을 두는 의사 순차 읽기에서 접근 성능을 향상시킬 수 있지만, 디스크 접근 요구가 주기적으로 발생하는 경우에는 이러한 시간간격이 좀처럼 발생하지 않는다는 문제가 있다. 반면에 유닉스의 추가읽기(readahead) 알고리즘은 보편적이면서도 $(p/n, b,$

0) 스트림 형태에 적합하다. 따라서 본 논문에서는 유닉스의 추가읽기 알고리즘과 본 논문에서 제안한 SPRA 알고리즘간의 성능을 비교한다. 유닉스에서는 추가읽기 기능이 기본적으로 활성화되어 있으며 추가읽기의 크기는 'hdparm -a' 명령어를 사용하여 제어한다. 읽을 블록의 크기가 B바이트 일 때 'hdparm -a 256'이면 B+256×512 바이트를 읽게 된다. SPRA 알고리즘을 수행할 때는 'hdparm -a 0'으로 하여 추가읽기 기능을 비활성화 시킨 후 실험하였다.

데이타베이스나 큰 규모의 자료구조를 유지하는 미리읽기 알고리즘들은 적은 용량의 메모리를 가지고 있는 임베디드 시스템에서 사용하기에 적합하지 않다. Vellanki와 Chervenak가 제안한 LZ-트리를 이용한 미리읽기 알고리즘(LZT)는 비교적 작은 크기의 자료구조로 구현할 수 있기 때문에 SPRA와 비교하기에 적당하다.

실험 대상은 다음과 같이 리눅스(Linux)의 추가읽기 크기를 지정한 것, Vellanki와 Chervenak가 제안한 LZ-트리를 적용한 미리읽기 알고리즘(LZT), 그리고 SPRA 알고리즘이다:

NONE 요구된 블록만 읽으며 추가읽기를 하지 않는다.

RA256 추가읽기 크기는 256개의 섹터 (512 바이트)이다. 따라서 읽을 블록의 크기가 b Kbyte이면 $b+$128 kbyte를 읽는다.

RA2560 리눅스의 추가읽기 크기를 10배 크기로 늘였다. 따라서 읽는 크기는 b+1280 kbyte가 된다.

LZT LZ-트리를 적용하여 다음 블록을 예측하여 미리 읽는 알고리즘.

SPRA (Speed-based PRefetch Algorithm) 디스크 접근의 진행 속도를 감지하여 미리읽기를 수행할 블록을 예측하는 제안된 알고리즘.

4.4 실험 결과 분석

그림 6은 리눅스의 None, RA256, RA2560, LZT, 그리고 제안된 SPRA 알고리즘을 각각 사용하였을 경우에 (p/n, b, 0) 스트림 형태가 생성하는 I/O의 평균 반환시간을 msec 단위로 보여준다. y-축은 반환시간(turnaround time)이므로 값이 작을수록 바람직하다. x-축은 재생 배속을 의미하는데, k-배속재생의 경우에는 1배속일 때의 주기 p 동안에 k번의 입력요구가 발생하고 각 요구에 대한 결과가 출력되는 시간을 의미한다. 추가읽기를 하지 않는 None의 경우에는 반환시간이 거의 일정하게 유지되고 80 배속 이상일 때 약간 증가한다. 추가읽기를 사용하는 RA256이나 RA2560의 경우에는 48배속까지는 None보다 작은 반환시간을 가지다가 64배속부터 반환시간이 증가한다. 읽는 양이 일정함에도 불구하고 반환시간이 최대 4배까지 상승하는 이유는 읽기 수행거부로 인해 다음 읽을 블록이 인접하지 않기 때문이다. 또한 추가읽기로 인해 I/O 부하가 올라가는 것이 지연을 심화시켜 더 많은 수행거부를 발생시킨다. LZ-트리를 이용한 알고리즘은 동일한 패턴이 발견되지 않아 미리읽기되는 양이 미미하다. 대신 남은 유휴시간을 예측하여 그동안 추가읽기를 수행하도록 알고리즘을 수정하였다. 그 결과로 부하가 높은 고속읽기에서 UNIX의 추가읽기보다 반환시간이 더 줄어드는 효과를 보였다. SPRA 알고리즘으로 인한 반환시간은 16배속까지는 성공적인 추가읽기에 의해 0의 값을 가지며 모든 재생 배속에 대하여 가장 작은 반환시간을 보여준다.

그림 7은 (p, b, n-1) 스트림 형태가 생성하는 I/O에서 각 알고리즘들로 인한 평균 반환시간을 보여준다. 이 스트림 형태에서는 대부분의 알고리즘들이 4배속 이후부터는 재생 배속과 거의 무관한 I/O 성능을 보여준다. SPRA 알고리즘은 8배속부터 반환시간이 0 보다 큰 값을 가지게 된다. 이는 재생 속도가 고속으로 변경되면 추가읽기의 다음 블록 예측이 빗나가서 버퍼캐시에 미스가 발생했기 때문이다. 예를 들어 32 배속으로 주어지는 입력을 예측해서 32 배속 당시의 블록을 미리 읽었는데 64 배속으로 재생 속도가 변경되면 미리 읽은 블록은 쓸모가 없어지고 새로 해당 블록을 읽어야 한다. 재생 속도가 반환시간에 영향을 많이 미치는 (p/n, b, 0) 스트림과는 달리 (p, b, n-1) 스트림은 재생 속도의 변화는 단지 블록 위치의 변화이기 때문에 이 그림과

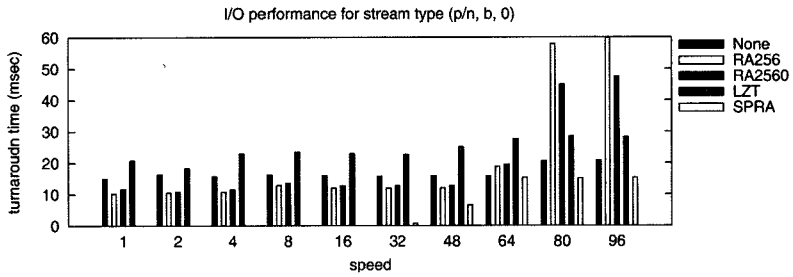


그림 6 각 미리읽기 알고리즘의 (p/n, b, 0) 스트림이 생성하는 I/O에 대한 평균 반환시간

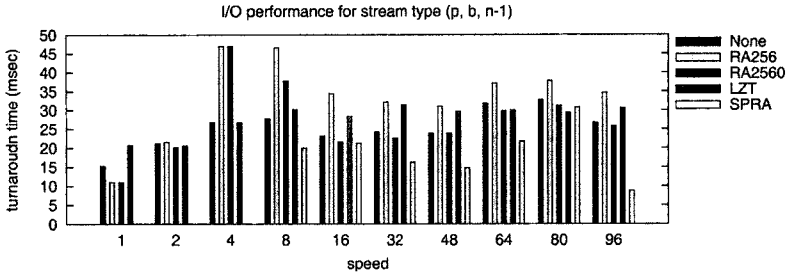


그림 7 각 알고리즘의 $(p, b, n\text{-}1)$ 스트림에 대한 평균 I/O 반환시간

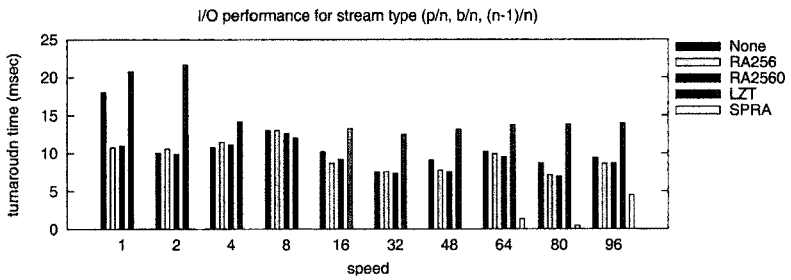


그림 8 각 알고리즘의 $(p/n, b/n, (n\text{-}1)/n)$ 스트림에 대한 평균 I/O 반환시간

같이 재생 속도와 반환시간은 거의 무관함을 알 수 있다. 유닉스의 추가읽기 알고리즘들도 인접한 블록만을 읽기 때문에 다음 블록을 읽지 못해서 추가읽기의 이익은 얻지 못하고 부하만 추가된 I/O를 수행한다.

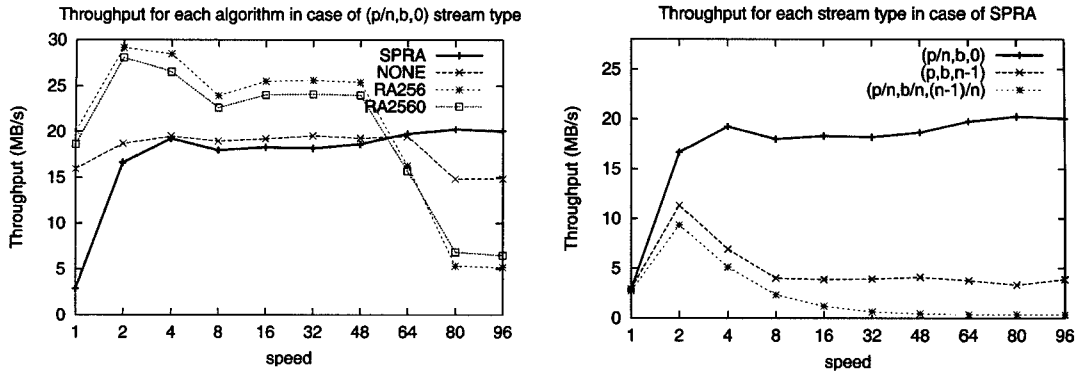
그림 8은 $(p/n, b/n, (n\text{-}1)/n)$ 스트림 형태가 생성하는 I/O에서 각 알고리즘들로 인한 평균 반환시간을 보여준다. 각 I/O의 블록 크기가 속도의 개수만큼 분할되므로 한 I/O에 대한 반환시간은 재생 속도가 높아질수록 줄어드는 경향이 있다. 하지만 인접하지 않은 블록들을 접근하기 때문에 탐색시간과 회전지연으로 인해 반환시간의 감소폭은 적은 편이다. 이는 더 고속의 재생속도는 지원이 안 될 수 있음을 의미한다. 96배속의 경우 약 10.42 msec의 주기로 I/O가 발생하는데, 반환시간이 거의 10 msec으로 주기에 거의 근접해 있다. 만약 재생속도가 더 높아진다면 지연이나 수행거부를 피할 수 없다. SPRA의 경우에는 최대 6 msec 정도의 I/O time을 가지므로 유닉스의 추가읽기에 비하면 좀 더 포용성이 있음을 알 수 있다.

표 2는 그림 6, 7, 그리고 8에서 보여준 각 알고리즘으로 인한 반환시간의 평균을 SPRA에 대한 상대값으로 보여준다. SPRA와 추가읽기 사이의 성능 차이는 $(p/n, b/n, (n\text{-}1)/n)$ 스트림의 경우에 가장 크게 보이는데, 디스크의 여러 부분에 분산되어 있는 작은 블록들을 읽을 때 추가읽기의 효과가 적기 때문인 것으로 보인다. 그럼에도 불구하고 추가읽기들 중에서는 RA2560이 가장 좋은 성능을 보이는 것으로 보아 너무 작은 블록들을 읽는 것 보다는 어느 정도 큰 범위의 블록을 읽어서 I/O가 여러 번 발생하는 것을 줄이는 것이 보다 효과적임을 알 수 있다. LZT가 NONE보다 높은 반환시간을 가지는 이유는 페이지 단위의 읽기 때문에 불필요한 부분이 읽혀지고 추가읽기의 실패 때문인 것으로 분석된다.

표 2 각 미디어 형태에 대한 알고리즘들의 반환시간에 대한 SPRA 알고리즘의 상대적 평균 반환시간

스트림 형태	NONE	RA256	RA2560	LZT
$(p/n, b, 0)$	31.55%	24.51%	26.81%	22.08%
$(p, b, n\text{-}1)$	52.63%	40.00%	49.26%	48.08%
$(p/n, b/n, (n\text{-}1)/n)$	5.98%	6.69%	6.81%	4.29%

SPRA는 백그라운드로 동작하기 때문에 응용 프로그램이 생성하는 I/O의 반환시간만으로는 성능에 대한 예측이 쉽지 않다. 그에 대한 대안으로 대역폭을 비교하는 것이 한 가지 방법이 될 수 있다. 그림 9(a)의 대역폭은 응용프로그램이 I/O를 요청한 양에 반환시간을 나눈 값이다. 리눅스의 추가읽기는 48배속까지는 SPRA보다 높은 대역폭을 가지지만 수행거부가 증가하는 64배속부터는 대역폭이 급격하게 떨어진다. 그 이유는 지연으로 인해 다음 I/O들이 수행거부가 되어버려서 추가읽기가 오히려 불필요한 부하로만 작용하기 때문이다. $(p/n, b, 0)$ 이외의 스트림 형태는 인접하지 않은 작은 블록들을 읽어야 하기 때문에 대역폭이 상당히 작게 나타난다. 그림 9(b)는 SRPA의 경우에 각 스트림 형태의 대역폭들을 보여주는데, 재생 속도가 높아짐에 따라서 블록간의 거



(a) (b)

그림 9 (*p/n, b*, 0) 스트림에서 각 재생 속도들에 대한 알고리즘들의 대역폭, (b) 각 스트림 형태에서 재생 속도들에 대한 SPRA 알고리즘의 대역폭

표 3 각 스트림 형태에서 SPRA 알고리즘의 캐시 적중률

스트림	(p/n, b, 0)	(p, b, n-1)	(p/n, b/n, (n-1)/n)
적중률	77.05%	72.00%	88.70%

리가 넓어져서 탐색시간의 비중이 늘어나고 있음을 보여준다. 표 3은 SPRA 알고리즘의 각 스트림 형태에 대한 적중률을 보여준다. (*p, b, n*-1) 스트림 형태에서는 I/O의 발생 횟수가 다른 형태들에 비해서 적어서 재생 속도가 변경될 때 발생한 미스가 전체 적중률에 많은 영향을 미친다. 또한 (*p/n, b*, 0) 스트림 형태에서는 96배속에서 다량의 수행거부 때문에 미스가 많이 발생하였다.

5. 결론 및 추후 계획

본 논문에서는 휴대용 멀티미디어 재생기에서 다양한 사용자 기능을 효과적으로 지원하기 위한 미리읽기 알고리즘을 설계하고 그 성능을 평가하였다. 대역폭을 높이기 위해 다음에 읽을 블록을 예측하는 방법으로 블록간의 거리를 사용하였다. 실험 결과 제안된 블록할당과 미리읽기 알고리즘을 사용하였을 경우 4배속 이하의 경우에는 항상 캐시 적중을 하였다. 각 스트림 형태에 대해서는 기존의 추가읽기 미리 읽기 알고리즘들에 비하여 4.29%~52.63%의 반환시간을 보인다. 또한 각 스트림 형태에 대해서 72.0%~88.7%의 적중률을 보였다.

추후 연구로는 SPRA 알고리즘을 리눅스 버퍼캐시의 한 모듈로 이식하고, 이를 호출하는 인터페이스를 구현할 계획이다. 멀티미디어 스트림을 시뮬레이션하는 대신 멀티미디어 응용프로그램으로부터의 입력을 직접 처리하여 그 성능을 평가하고자 한다.

참 고 문 헌

[1] D. P. Anderson, Y. Osawa, and R. Govindan, "A file system for continuous media," *ACM Transactions on Computer Systems*, vol. 10, pp. 311-337, November 1992.

[2] C. Wang, V. Goebel, and T. Plagemann, "Techniques to increase disk access locality in the minorca multimedia file system," in *ACM Multimedia (2)*, pp. 147-150, 1999.

[3] K. W. Ng and K. H. Yeung, "Analysis on disk scheduling for special user functions," in *ICMCS*, pp. 608-611, 1996.

[4] T. Niranjan, T. cker Chiueh, and G. A. Schloss, "Implementation and evaluation of a multimedia file system," in *ICMCS*, pp. 269-276, 1997.

[5] Y. J. Nam, D. Kim, and C. Park, "Enhancing disk i/o performance for pseudo sequential reads," in *한국정보처리학회 자료저장시스템연구회 워크숍*, July 2003.

[6] H. P. Katseff and B. S. Robinson, "Predictive prefetch in the nemesis multimedia information service," in *Proceedings of the second ACM international conference on Multimedia MULTIMEDIA '94*, pp. 201-209, October 1994.

[7] D. Revel, C. Cowan, D. McNamee, C. Pu, and J. Walpole, "Predictable file access latency for multimedia," in *IFIP 5th International Workshop on Quality of Service (IWQoS'97)*, (New York), May 1997.

[8] R. H. Patterson and G. A. Gibson, "Exposing i/o concurrency with informed prefetching," in *Proceedings of the third international conference on on Parallel and distributed information systems*, (Austin, Texas, United States), pp. 7-16, 1994.

[9] D. Revel, D. McNamee, D. Steere, and J. Walpole, "Adaptive prefetching for device independent file i/o," in *Proceedings Multimedia Computing and*

Networking 1998 (MMCN98), 1998.

[10] D. M. Huizinga and S. Desai, "Implementation of informed prefetching and caching in linux," in *International Conference on Information Technology: Coding and Computing*, pp. 443-448, March 2000.

[11] V. Vellanki and A. L. Chervenak, "A cost-benefit scheme for high performance predictive prefetching," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, January 1999.

[12] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "Diskseen: exploiting disk layout and access history to enhance i/o prefetch," in *Proceedings of 2007 USENIX Annual Technical Conference (USENIX'07)*, (Santa Clara, California), June 17-22 2007.

[13] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. Wiley, th ed., 2002.

[14] M. Stonebraker, "Operating system support for database management," Communications of the ACM, pp. 412-418, July 1981.

최 태 영

1991년 2월 고려대학교 수학교육과 졸업(이학사). 1996년 2월 포항공과대학교 대학원 컴퓨터공학과 졸업(공학석사). 2002년 8월 포항공과대학교 대학원 컴퓨터공학과 졸업(공학박사). 2002년 9월~현재 금오공과대학교 컴퓨터공학부 조교수. 관심분야는 병렬 및 분산 알고리즘, 컴퓨터 구조

윤 현 주

1988년 서울대학교 컴퓨터공학과(공학사). 1990년 한국과학기술원 전산학과(공학석사). 1997년 한국과학기술원 전산학과(공학박사). 1997년~1998년 ICASE, NASA LaRC 객원연구원. 1999년~2001년 ㈜브레인투엔티원 기술이사. 2002년~2004년 한국정보통신대학원대학교 연구교수. 2005년~현재 금오공과대학교 조교수. 관심분야는 운영체제, 임베디드 시스템, 센서 네트워크 등