파일 읽기 요청 크기를 이용한 리눅스 2.6.7 미리읽기 성능 향상 방안

설지훈° 이금석
동국대학교 컴퓨터공학과
{eckaiar°, kslee}@dongguk.edu

Linux 2.6.7 readahead performance improvement using read request size

Jihun Seol° Keumsuk Lee
Dept. of Computer Science, Dongguk University

요 약

미리읽기(readahead) 설계는 파일 시스템 성능에 큰 영향을 미치는 요소 중 하나이다. 기존 리눅스 커널 2.6.7 버전의 미리읽기는 임의적 파일 접근 시에 읽는 크기가 클수록 사용되지 않는 페이지들의 할당에 시간을 낭비하게 되어 전체적인 성능 저하가 일어난다. 본 논문에서는 이러한 문제를 해결하기 위해 요청되는 읽기 크기에 따라 미리읽기 크기를 조정하는 방법을 제안하며 제안된 방법으로 리눅스 커널을 수정하여 임의적 파일 접근 시에 성능 향상이 일어남을 실험을 통해 보이고자 한다.

1. 서 론

대부분의 운영체제는 가까운 미래에 필요하다고 예측되는 디스크 블록을 메모리에 미리 읽어들이려고 하며, 이러한 미리읽기를 통해 응용의 성능을 50% 이상 향상시킬 수 있다[1]. 그 이유는 큰 양의 데이터를 장치로부터 읽어들이려면 고정된 I/O 연산 시간이 소비되는데 이 I/O 처리시간동안 프로그램 수행을 중첩시킴으로서 전체적인 수행시간을 줄일 수 있기 때문이다. 하지만 미리읽기는 순차적 접근 시에는 상당한 성능 향상이 있지만, 예측할 수 없는 임의 접근 시에는 성능을 오히려 저하시킬 수 있다.

리눅스 커널 2.6.7 버전에서도 읽기 처리량을 향상시키기 위해 미리읽기를 수행하며, 이를 수행하기 위해 현재 윈도우(current window)와 미리읽기 윈도우(readahead window)를 유지한다[2]. 요청이 현재 윈도우에 해당하는지 않는지에 따라 미리 읽기 윈도우 크기를 조정하는 알고리즘을 사용한다. 이 알고리즘은 접근 패턴이 임의적이고 접근 크기가 리눅스 최대 윈도우 크기보다 같거나 클 경우 필요하지 않은 페이지를 할당하게 되며 이로 인해 전체적인 성능 저하가 일어난다.

본 논문에서는 리눅스 커널 2.6.7에서 미리읽기 윈도우를 결정할 때 작업 부하의 특징을 나타내는 여러 가지 정보 중에 읽기 요청의 크기를 이용한 알맞은 크기의 윈도우를 할당하는 알고리즘을 고안하여 불필요한 페이지의 할당을 최소화하였다.

논문의 구성은 다음과 같다. 2장에서는 기존의 미리읽기에 관련된 연구와 리눅스 미리읽기의 작동방법에 대해서 설명하고, 3장에서는 제안된 미리읽기 방법에 대해 설명하고, 4장에서는 실험 환경에 대한 설명과 실험을 통해 제안된 알고리즘의 성능을 분석하고, 5장에서는 결론을 제시한다.

2. 관련 연구

미리읽기에 관한 연구는 미리읽기의 이득을 이용하는 방향으로 시작하였으나 최근에는 파일 접근 패턴을 자동적으로 인식하여 미리읽기를 좀 더 정확하게 수행하는 방법에 대한 연구가 많이 이루어졌다[4][6]. Shriver[1]는 파일 시스템이 미리읽기를 왜 하는지에 대해 논하고 성능 측정을 위한 분석적 모델을 제시하였다. Shriver[3]은 미리읽기를 결정하는데 필요한 작업부하의 특징을 나타내는 요소들을 제안하였다. 이 연구들은 주로 파일의 접근이 순차적이라는 가정 하에 이루어졌으나 임의 접근에 대한 연구도 이루어지고 있다. Small[5]은 임의 접근 미리 읽기를 할 때 파일 인덱스 정보를 VINO 운영체제에 전달하는 방법을 사용하여 약 20%의 성능 향상을 보여준다.

Pai[2]에선 리눅스 커널 2.6.0 버전에서 2.6.7로의 미리읽기 성능을 향상시킨 방법에 대해 논하고 있다. 읽기 요청이 현재 윈도우(현재 요청된 페이지로 구성)에 해당하면 미리읽기 윈도우(예측되어 미래에 요청될 페이지로 구성)의 크기를 2페이지만큼 증가시키며 해당하지 않을 경우 2페이지만큼 감소시킨다. 요청이 현재 윈도우에 해당되지 않을 경우 새로운 미리읽기 윈도우를 설정하는데 이때 현재까지의 접근에 대한 순차성의 평균값을 계산하여 그 값과 현재 미리읽기 윈도우 크기 값 중 적은 값을 선택하여 미리읽기를 시도한다.

리눅스 커널 2.6.7의 미리읽기 페이지의 최대 크기는 32페이지(128KB)이며 최소는 0페이지(0KB)이다. 파일의 첫 접근 시에 미리읽기 윈도우는 (최대 페이지 / 2)인 16페이지로 설정되며 순차성 평균값은 16으로 설정된다. 순차적인 접근이 일어날 경우 순차성 값이 +1되며 순차성 값이 평균보다 클 경우 ((순차성 + 평균 + 1) / 2)이 평균값이 된다. 리눅스에선 미리읽기를 읽기 요청이 들어올 때 바로 시도하는 것이 아니라 순차성 평균값이 최

대 윈도우 크기보다 커질 때까지 기다렸다가 초과할 경우 미리 읽기를 시도한다. 그 이후부터 평균값이 최대값보다 클 경우 읽기 요청이 들어올 때 바로 미리 읽기를 시도한다. 만약 임의적인 접근으로 현재 윈도우에 해당되지 않는 곳을 계속 참조하게 되면 (현재 평균값 / 2)의 크기로 미리 읽기를 시도한다. 예를 들어 파일의 첫 접근부터 완전 임의적인 접근이 4번 연속해서 일어날 경우 미리읽기 윈도우 크기는 16 -> 8 -> 4 -> 2 로 줄어들게 된다.

리눅스의 미리 읽기 알고리즘은 read 시스템 콜의 읽는 크기에 영향을 받지 않고 1번에 1페이지씩 처리한다. 이 때 read 시스템 콜의 요청 크기가 최대 윈도우 크기(32 페이지)를 넘고 접근이 예측 할 수 없는 임의 접근일 경우 한 요청 당 32페이지만큼의 불필요한 페이지를 할당하게 된다. 이로 인해 전체적인 read의 성능이 저하가 일어나게 된다.

3. 제안 알고리즘

본 논문에서 제안된 알고리즘은 기존 리눅스의 미리 읽기 알고리즘을 유지하면서 임의적인 큰 블록의 읽기 요청이 일어날 때 작동하는 방식이다.

3.1 읽기 요청 크기를 이용

추가된 알고리즘은 Shriver[3]에서 제안된 작업부하의 특징을 나타내는 요소 중에 요청의 크기를 이용하였다. 리눅스는 일반적으로 읽기 요청을 할 때마다 mm/read-ahead.c 파일안의 미리읽기 함수인 page_cache_read-ahead 함수를 호출하는데 이때 읽기 요청의 크기를 매개변수로 넘겨주게끔 mm/filemap.c 의 소스 코드를 수정하였다. 그리고 page_cache_readahead의 함수에 추가된 코드는 [그림 1]과 같다.

```
if 읽기 요청의 크기가 -1이 아닐 때 then
        요청된 페이지 = 요청 크기 / 4096
        if 요청된 페이지가 1보다 클 경우 then
                큰 블록 접근 = true
        fi
fi
if 큰 블록 접근이 true 이면 then
        미리읽기 크기 = 예약된 페이지
        미리읽기 윈도우 시작 = 오프셋+예약된 페이지
        미리읽기 윈도우 크기 = 순차성 평균값
else
        미리읽기 윈도우 시작 = 0 /* 초기화 */
        미리읽기 윈도우 크기 = 0
fi
```

[그림 1] 읽기 요청 크기를 이용한 알고리즘

[그림1]의 알고리즘은 기존의 리눅스에선 읽기 요청의 크기에 상관없이 1페이지씩 참조하였던 것에 비해 요청의 크기가 1페이지보다 클 경우 요청한 만큼 바로 미리 읽기를 시도한다. 그리고 미리 읽기 크기를 순차성 평균값으로 설정함으로써 다음번에 임의적인 접근이 이루어졌을 때 할당하는 페이지의 개수를 결정한다. 이 알고리즘은 큰 블록 요청이 들어왔을 경우 그 수만큼을 미리 읽음으로써 불필요하게 할당되는 페이지의 수를 최소화한다. 만약 요청의 크기가 1페이지(4KB)보다 같거나 적을 때는 앞으로 요청될 크기를 알 수 없기 때문에 기존의 리눅스 미리 읽기 알고리즘을 사용한다.

3.2 정렬되지 않은 순차적인 접근

파일에서 순차적인 접근이 정렬되지 않고 반복적으로 일어난다면 기존의 미리읽기 알고리즘은 파일 접근 패턴을 제대로 예측하지 못한다. 예를 들어 파일 접근이 1000, 3001, 1001, 3002, 1002, 3003, ... 같은 패턴을 보인다면 부분적으로 순차적임에도 불구하고 미리읽기 윈도우의 크기는 계속 줄어들게 된다. 이것을 보완하기 위해 현재 윈도우, 미리읽기 윈도우에 이전 윈도우(previous window)를 추가하여 현재 요청이 현재 윈도우에 해당하지는 않지만 이전 윈도우에 해당한다면 윈도우 크기를 줄이지 않는 코드를 추가하였다.

3.3 그 외 수정 사항

기존 알고리즘은 파일을 처음 접근 할 때 미리 읽기 윈도우의 크기를 (최대 윈도우 크기 / 2)인 16으로 정하지만 제안된 알고리즘에선 4로 정한다. 만약 16페이지의 크기(64KB)보다 임의적인 작은 읽기 요청이 많을 경우 불필요한 페이지의 할당이 일어나기 때문이다. 이렇게 할 경우 순차적인 접근에서는 성능이 저하될 수 있다고 생각 할 수 있지만 리눅스 미리 읽기 알고리즘은 순차적인 접근이 일어날 경우 미리읽기 윈도우 크기를 매우 빠르게 증가시키기 때문에 성능 저하는 거의 일어나지 않았다.

제안된 알고리즘은 최대 윈도우 크기를 32에서 64로 조정한다. 32페이지의 크기는 큰 규모의 응용에서는 적은 크기이기 때문이다.

4. 실험

실험은 기존의 리눅스 커널 2.6.7 버전과 수정된 커널을 sysbench 프로그램[7]을 이용하여 성능을 비교, 평가했다. sysbench는 파일 I/O 성능을 테스트할 수 있는 공개 소프트웨어이다. 실험 환경은 인텔 펜티엄4 2.6GHz CPU, 512MB RAM, Gentoo Linux 1.4rc, Ext3 파일 시스템, 각 32MB 파일 128개, 총 4GB 테스트 파일로 이루어진 실험 파일들에 1000개의 임의적 읽기 요청을 블록의 크기를 16KB부터 1024KB까지 달리하여 수행하였을 때 총 걸린 시간과 초당 처리량을 측정하였다. sysbench 프로그램을 사용한 명령은 [그림 2]와 같다.

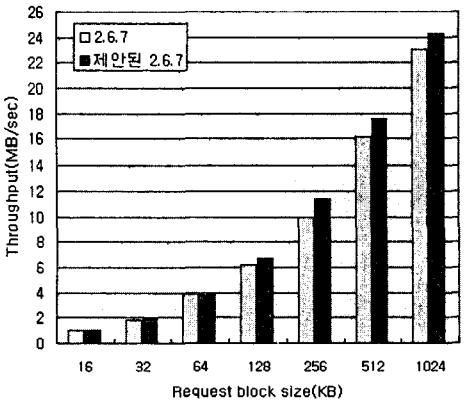
```
#sysbench --num-threads=1 --max-requests=1000
--test=fileio --file-total-size=4G --file-block-size=
[블록 크기] --file-test-mode=rndrd run
```

[그림 2] sysbench 명령어 매개변수

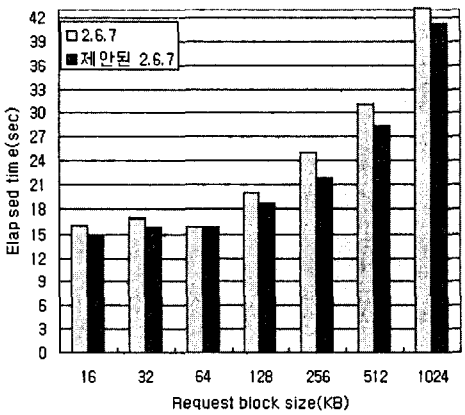
sysbench 프로그램이 파일을 임의 접근 할 때 그 접근이 같은 곳을 참조할 확률은 요청 블록의 크기가 16KB일 때 약 1%에서 ±2%의 오차 범위를 가지며 블록의

크기가 1024KB일 경우 약 14%에서 ±2%의 오차 범위를 가진다. 또한 실험 결과가 리눅스 페이지 캐시의 영향을 받지 않도록 시스템 전체를 재시동하거나 700MB 이상의 미디어 파일을 메모리에 복사하여 페이지 캐시를 초기화하였다.

[그림 3]은 초당 파일 읽기 처리량을 비교한 그래프이다. 블록 크기가 128KB(최대 윈도우 크기)와 같거나 클 경우 약 10%정도(2MB/s)의 성능 향상을 보였다. [그림 4]는 총 실행시간을 비교한 그래프이다. 총 실행시간 또한 128KB부터 10%정도의 성능 향상을 보인다.

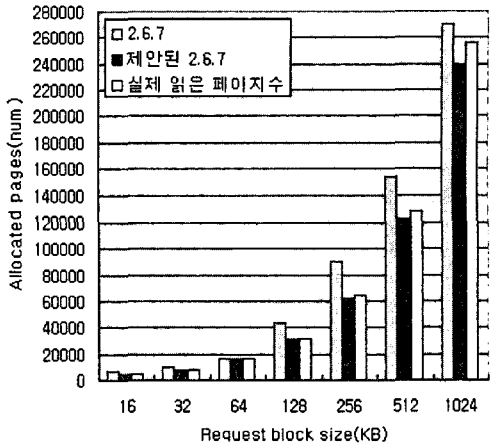


[그림 3] 초당 처리량 성능 평가



[그림 4] 총 실행 시간 비교

[그림 5]는 프로그램 수행 시에 할당된 페이지의 수를 비교한 그래프이다. 블록의 크기가 128KB일 경우 수정된 알고리즘은 약 12000개의 페이지를 더 적게 할당했으며 블록의 크기가 256KB 이상일 경우 30000개 이상의 페이지를 더 적게 할당하였다. 그러므로 약 117MB의 메모리 할당을 절약하게 된다. 그래프를 보면 실제 할당되야 하는 페이지보다 더 적게 페이지가 할당된 것을 볼 수 있는데 이는 sysbench 프로그램이 같은 블록을 참조할 확률이 블록의 크기가 커질수록 증가하기 때문이다. 예를 들어 블록의 크기가 1024KB일 경우 약 14%정도 되는 요청이 중복되었다.



[그림 5] 할당된 총 페이지 수

5. 결론 및 향후과제

제안된 알고리즘은 미리읽기의 크기를 결정할 때 읽기 요청 블록의 크기를 이용함으로써 임의적인 파일 접근의 성능을 약 10%정도 향상시킬 수 있음을 보였다. 향후 과제로는 요청 블록의 크기를 이용한 통계적 접근방법과 요청 블록 크기 외에 작업 부하의 특징을 나타내는 다른 요소도 미리읽기 알고리즘에 포함시키는 것, 그리고 미리읽기 최대 윈도우 크기가 첫 접근 시 윈도우 크기를 동적으로 변경하는 방법에 대해 연구하고자 한다.

6. 참고 문헌

[1] E. Shriver, C. Small, "Why Does File System Prefetching Work?", Proceedings of the USENIX Annual Technical Conference, June 6-11, 1999

[2] R. Pai, B. Pulavarty, M. Cao, "Linux 2.6 performance improvement through readahead optimization", Linux symposium, July 21-24, 2004

[3] E. Shriver, A. Merchant, J. Wilkes. "An analytic behavior model for disk drives with readahead caches and request reordering.", Joint International Conference on Measurement and Modeling of Computer systems, pages 182-191, June 1998

[4] T. Kroeger, "Predicting file system actions from reference patterns", University of California, Santa Cruz, December 1996

[5] C. Small, "Building an extensible operating system.", Harvard University, Bostom, MA, October 1998

[6] D. M. Ritchie and K. Thompson, "The UNIX time-sharing system", The Bell System Technical Journal, 57(6):1905-1930, July/August 1978. Part 2

[7] sysbench program, http://sysbench.sourceforge.net

[8] Cross-Referencing Linux, http://lxr.linux.no