사용자 데이터와 메타데이터에 대한 유닉스 버퍼 캐쉬의 성능 분석

최진모(국민대학교 전자공학과), 김준형(교육부 전산담당관)
성영락(국민대학교 전자공학부), 오하령(국민대학교 전과공학부)

Performance analysis of UNIX buffer cache on user data and metadata

Jin Mo Choi, June Hyoung Kim, Yeong Rak Seong, Ha Ryoung Oh

요 약

본 논문에서는 유닉스 파일 시스템에서의 버퍼캐쉬 크기에 따라 사용자 데이터와 메타데이터의 버퍼 캐쉬 히트율을 분석하였다. 그리고 메타데이터가 유닉스 운영체제 파일 시스템의 성능에 미치는 영향을 분석하고 이를 기반으로 버퍼캐쉬의 동적 특성과 성능의 장애 요인들을 분석하였다.

유닉스 운영체제에서 사용되는 사용자 데이터와 메타데이터에 대한 버퍼캐쉬의 동적인 동작을 분석하기 위하여 trace-driven 방법을 이용하였으며 이를 위하여 시뮬레이터를 작성·사용하였다. 파일시스템은 특정 유닉스 버전에 영향을 받지 않기 위해 UFS[1]에 기초하였고, 작업부하(workload)로는 Sprite-trace 데이터 중 allspice 서버에서 추출한 데이터를 사용하였다

1. 서 론

과거에 비해서 컴퓨터의 중앙 처리 장치의 속도(이하 CPU)가 향상되었으며, 가장 널리 사용되고있는 보조기억장치인 디스크의 용량도 과거에 비해 크게 증가되었다. 그러나, 시스템의 성능향상은 CPU 속도 향상에 비례하여 향상되지 않고 있다. 그것은 아무리 빠른 CPU라도 프로그램 실행 도중 필요한 데이터를 디스크에 요청할 때, 디스크 드라이브의 탐색시간, 회전지연시간 및 전송시간 때문에 CPU의 입출력 응답시간이 느려지게되며, 이 때문에 CPU의 처리량이 줄어들기 때문이다. 결과적으로 CPU와 디스크의 속도차이로 인해서 시스템의 성능이 저하되게 된다. 이러한 속도 차이를 극복하기 위한 방법 중 하나로 디스크 접근 횟수를 줄이는 방법이 있다. 이 디스크 접근 횟수를 줄이는 방법은 운영체제에서 많이 사용하는 방법으로서 한번 읽은 데이터를 주기억장치의 버퍼 캐쉬에 저장하여 재사용 할 때는 버퍼 캐쉬로부터 데이터를 읽어들여 디스크의 접근 횟수를 줄이는 방법이다.

유닉스 파일 시스템은 대용량의 디스크를 효율적으로 사용하기 위해서 메타데이터를 사용하고있다. 메타데이터란 데이터를 위한 데이터로서, 본 논문에서는 아이노드, 디렉토리, 슈퍼 블록 및 간접 블록을 합쳐 메타데이터라 부르기로 하겠다. 이 메타데이터의 크기는 비록 사용자 데이터 블록에 비하여 작지만, 디스크 입출력이 블록 단위로 이루어지고 있다는 것을 고려할 때 그 크기보다는 입출력 횟수가 시스템 성능에 더 큰 부담을 주게된다. 메타데이터의 사용 빈도가 사용자 데이터의 사용 빈도 보다 많을 경우 파일 시스템의 성능에 큰 장애요인이 될 수 있다.

그러나 이에 관한 정확한 연구분석이 부족하다. 이에 본 논문은 사용자 데이터와 메타데이터가 버퍼 캐쉬에서의 히트율 그리고 둘 사이의 디스크 입출력 사용빈도를 분석하고 유닉스 파일시스템의 성능 장애 요인들을 분석하였다.

2. 관련 연구

파일 시스템은 파일의 접근 패턴에 따라 다양하게 설계할 수 있다. 파일 접근 패턴은 과학 계산, 트랜잭션 처리 및 공학/사무용 파일 시스템으로 분류할 수 있다[2]. 유닉스 파일 시스템은 공학/사무용 파일 시스템으로 주로 사용하며, 많은 수의 작은 파일을 접근하고 동시 접근이 적은 파일 시스템이다.

지금까지의 유닉스 파일 시스템 분석의 초점은 디스크로부터의 평균 파일 접근량, 파일 접근 패턴, 전송되는 파일의 크기, 접근되는 파일의 크기 그리고 버퍼 캐쉬에서의 파일수명에 대한 분석이었다. Ousterhout[3] 등은 사용자의 파일 접근 형태와 UNIX 4.2 BSD 파일시스템의 캐쉬 동작을 분석하였고, Sprite 분산파일시스템의 동작을 분석[4]하였다 이들의

결과에 따르면 UNIX 환경에서의 파일 접근 형태는 순차적이며, 대부분 크기가 작은 파일을 사용하나, 입출력의 대부분은 매우 큰 파일의 입출력 시 발생하는 것으로 보고되어 있다 또한 이들은 수MB의 캐쉬로 상당한 디스크의 입출력을 줄일 수 있다고 예측하고 있다

3. 시뮬레이션 환경 및 방법

3.1. Trace data

시뮬레이션에 사용된 작업부하로는 캘리포니아 버클리 대학에서 추출한 Sprite-trace 데이터 중 allspice 서버에서 추출한 데이터[4,5]를 사용하였다. 이 trace 데이터는 40대의 10-MIPS 웍스테이션 중 파일서버로 4대를 나머지는 diskless clients로 구축된 환경에서 수집된 데이터이다. 4대의 파일서비 중 "allspice" 는 메인 서버로써 시스템의 대부분의 파일들을 소유하고 있는 파일서버이다 추출된 데이터의 주된 응용프로그램은 여러 종류의 대화식 편집기, 프로그램 개발 및 디버깅, 전자 메일, 문서작성 그리고 시뮬레이션이다. 이런 작업부하는 공학/사무용 파일 시스템의 성능을 분석하는 데 적합한 작업부하가 된다.

Trace 데이터는 Sprite 커널이 오픈, 닫기 및 파일 삭제와 같은 파일 시스템에서의 여러 가지 사건들(events)의 정보를 로그하여 수집하였다 이 trace 데이터는 네트웍 레벨에서의 요청이나 디스크 동작이 아닌, 커널 호출 레벨에서의 파일 동작을 기록한 것이다. Trace 데이터는 48시간을 주기로 4번에 걸쳐 수집되었으며 이를 trace1-5까지로 나누어 저장하였다. Trace 전체에 걸쳐서 작업부하가 일관성이 있다는 것은 증명하지는 않았다

3.2. 시뮬레이션 방법

본 시뮬레이션에서는 디스크 탐색시간, 회전지연시간 및 전송시간과 같은 정량적인 측정은 traces 데이터가 제공하는 정보 부족으로 인하여 이 부분에 대한 측정은 하지 않았다 그 대신 버퍼 캐쉬 히트 여부, 디스크의 읽기·쓰기 횟수 등을 측정하였다.

하나의 버퍼 블록에 해당하는 *cache_buf* 데이터 타입을 만들었으며, 버퍼 크기에 해당하는 블록 수를 구하여, 이 블록 수에 맞게 *cache_buf*를 링크드 리스트로 구현하여 사용하였다 시뮬레이션 실행은 먼저, Traces 데이터 파일로부터 레코드헤더와 레코드를 읽은 후 레코드헤더 타입에 해당하는 UFS 저 수준(low level) 알고리즘을 적용시켰다. 레코드헤더 타입은 오픈, 삭제, 닫기 등과 같은 커널 호출에 대한 명령이다 만약, 레코드헤더 타입이 데이터를 요구하는 타입이라면, 디스크로부터 블록 하나를 읽어 들이는 *bread* 함수를 호출하고 이 함수에서는 버퍼 블록을 할당하는 *getblk* 함수를 호출하게 하며, 해싱함수를 이용하여 캐쉬 히트여부를 판별하였다 예를 들어, 레코드헤더 타입이 "/a/b/c"파일을 오픈하기 위한 "SOSP-LOOKUP"이라면, *namei* 알고리즘을 적용시켰다 namei에 필요한 디렉토리 "/","a"와 "b"와 파일 "c"에 대한 정보를 아래와 같은 형식으로 lookup 레코드가 제공한다

FileID: (1,e,a,2) -> "/"
FileID: (1,e,a,16190) -> "a"
FileID: (1,e,a,1619e) -> "b"
FileID: (1,e,a,161d0) -> "c"
※ FileID: (not-used, hostID, domainID, inode)

이 정보와 bread, getblk 함수를 이용하면 디렉토리 블록을 3번 읽고, 디렉토리 블록에 해당하는 indoe를 3번 그리고 데이터 파일을 1번 읽게 된다. 이런 과정에서 캐쉬 히트여부를 판별하며, 캐쉬 미스 시에는 *disk_read* 함수를 호출하여 디스크 읽기를 적용시켰다. 파일의 타입 즉, 일반 데이터 파일인지 디렉토리 파일인지는 레코드 타입이 제공해 준다.

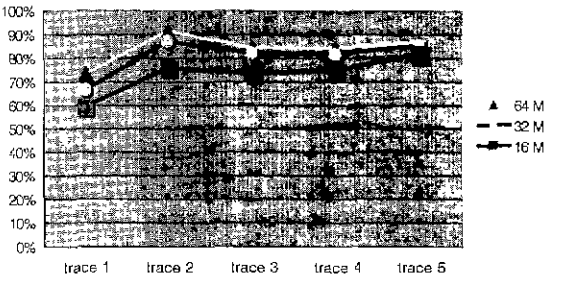
디스크 쓰기에 대한 처리를 위해서 *disk_write* 함수를 작성하였다. 디스크 쓰기는 크게 3가지 경우에 발생한다. 첫째, 레코드 타입이 파일 시스템의 구조적 정보를 변경시키는 레코드 타입일 경우는 메타데이터에 대한 디스크 쓰기가 일어난다. 둘째, 지연쓰기(delayed-write)정책을 적용시킨 사용자 데이터 블록에 대한 디스크 쓰기이다 5초마다 동기 데몬(sync-daemon)을 발생 시켜 갱신된 사용자 데이터 블록이 30초가 초과되었을 경우와 LRU에 의해 갱신된 블록이 교체될 경우 디스크 쓰기를 적용시켰다 마지막으로, 갱신된 사용자 데이터 블록을 즉시 디스크에 쓸 경우이다 이와 같은 경우는 공유되는 데이터 블록이 갱신될 때 일어난다.

디스크의 데이터 블록을 할당 및 반환하는 연산일 경우 데이터 블록 수를 관리하는 슈퍼 블록의 디스크 쓰기와 읽기도 고려하였다. 슈퍼 블록의 읽기는 슈퍼 블록이 갖고 있는 프리 블록을 모두 사용하여 다시 채워 넣어야 할 경우 1번 일어나게 된다

4. 시뮬레이션 결과 및 분석

4.1. 버퍼 캐쉬 히트율

그림1은 버퍼 캐쉬의 크기에 따른 사용자 데이터 블록의 히트율을 나타낸다



[그림1 데이터 블록 캐쉬 히트율]

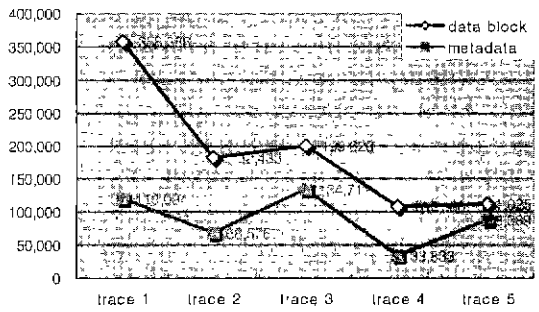
그림1에서 보듯이 버퍼 캐쉬의 크기가 커지면 그에 따른 캐쉬율도 증가됨을 볼 수 있다 그러나 그 증가폭이 크지 않음을 볼 수 있다 이것은 버퍼 캐쉬의 크기를 증가시키기만 하면 시스템의 성능이 크게 향상 될 것이라는 생각이 잘못되었음을 확인 시켜주는 결과이다.

시뮬레이션의 결과 메타데이터에 대한 캐쉬 히트율은 모든 99%이상을 보였으며, 버퍼 캐쉬가 클수록 히트율이 증가하나 아주 적은 비율이다

4.2. 디스크로부터 데이터 블록 읽기 횟수

디스크 입출력 단위는 일반적으로 디스크 블록 단위이므로 입출력의 양보다 횟수가 더 중요하다.

그림2는 버퍼 캐쉬의 크기가 16Mbytes일 때의 디스크 읽

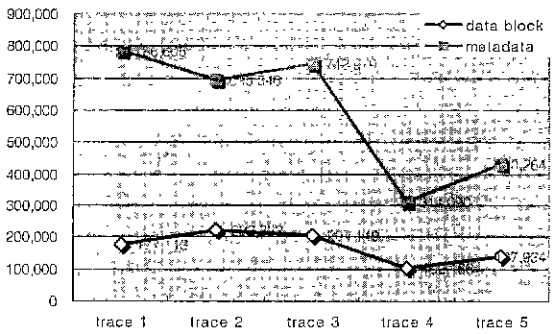


[그림2 디스크로부터 블록 읽기]

기 횟수를 나타낸 것이다 그림2를 볼 때 사용자 데이터 블록이 메타데이터보다 많이 이루어지고 있음을 알 수 있다. 이것은 그림1에서 캐쉬히트율이 16Mbytes일 때 약 60-80% 밖에 되지 않기 때문이다 그러나 메타데이터에 대한 캐쉬히트율이 99% 이상임에도 불구하고 메타데이터에 대한 디스크 읽기가 많음을 보여주고 있다. 이것은 시스템 운영에 있어서 메타데이터가 사용자 데이터보다도 많이 사용되고 있음을 보여준다.

4.3. 데이터 블록을 디스크에 쓴 횟수

디스크 쓰기에 대한 시뮬레이션의 결과는 아래 그림3으로 나타내었다.



[그림3 디스크 쓰기]

그림3의 결과로 볼 때 모든 메타데이터에 대한 디스크 쓰기가 사용자 데이터 블록보다도 많다는 것을 볼 수 있다. 이것은 사용자 데이터는 지연쓰기 정책의 사용으로 그 만큼 디스크 쓰기를 줄인 반면, 시스템의 붕괴 시나 디스크 드라이브의 고장으로 인한 데이터 손실을 막고 데이터의 일관성을 위해 write-through 정책을 사용하는 메타데이터는 연시적인 생성이나 갱신에 대해서도 디스크에 그 내용을 써야하므로 디스크 쓰기 횟수가 많음을 볼 수 있다.

그림3의 결과로 볼 때 메타데이터에 대한 디스크 쓰기가 디스크의 탐색시간 및 회전지연 시간에 큰 영향을 주게 된다. 이 결과로 메타데이터의 디스크 쓰기가 유닉스 파일 시스템의 성능 저하의 큰 원인이 됨을 알 수 있다.

5. 결 론

시뮬레이션 결과로 유닉스 운영체제에서의 버퍼 캐쉬 특성을 정리하면 다음과 같다.

첫째, 버퍼 캐쉬의 크기가 16Mbytes일 때와 64Mbytes일 때 사용자 데이터에 대한 캐쉬 히트율의 차이가 크지 않음을 알 수 있었다. 둘째, 메타데이터에 대한 읽기는 거의 모두 캐쉬에서 흡수되나 읽기의 절대량이 메타정보에 대한 것이기 때문에 캐쉬 히트율이 높다하더라도, 캐쉬에서 히트되지 않은 메타데이터의 읽기가 전체 디스크 읽기의 상당량을 차지한다는 것이다. 셋째, 사용자 데이터에 대한 쓰기가 파일서브시스템의 디스크 쓰기 중 단지 25-35% 정도만 차지한다는 것을 보여준다. 넷째, 사용자 데이터 측면에서 30초 지연쓰기를 통하여 디스크 쓰기 횟수를 줄일 수 있었으나, 겹쳐 쓰기 특히 삭제로 인한 메타데이터 쓰기의 추가적 발생과 메타데이터의 write-through 특성 및 쓰기 량의 절대량이 메타데이터 쓰기라는 사실로 미루어 판단해 보면, 30초 delayed-write를 통해서 디스크 쓰기 횟수를 크게 줄일 수는 없다 결론적으로 메타데이터의 디스크 쓰기가 UNIX 파일 시스템의 성능장애의 큰 요인으로 작용함을 알 수 있다

참 고 문 헌

[1] Maurice J Bach, "The Design of the UNIX Operating Operating System", Prentice-Hall, 1986.

[2] John K Ousterhout, Fred Douglis, "Beating the I/O Bottleneck: A Case for Log-Structured File Systems" *ACM Operating Systems Review*, Vol 23, No.1, pp. 11-28, Jan. 1989

[3] John K. Ousterhout, et al. "A Trace-Driven Analysis of the UNIX 4.2 BSD File System", *Proceedings of the 10th Symphosium on Operating System Principles,* Orcas Island, WA, 1985.

[4] Mary G. Baker, et al, "Measurements of a Distributed File System", *Proceedings of the 13th Symposium on Operating System Principles, Monterey CA, October* 1991, 198-212, Also published as Operating Systems *Review 25,5* (October 1991).

[5] John H Hartman, "Using the Sprite File System Traces", *University of California, Berkeley, Technical Report CA 94720*, May, 1993.