APC: 가상 메모리 시스템에서 적응적 페이지 선반입 제어 기법

(APC: An Adaptive Page Prefetching Control Scheme in Virtual Memory System)

안 우 현 † 양 종 철 †† 오 재 원 †††
(Woo Hyun Ahn) (Jongcheol Yang) (Jaewon Oh)

요 약 가상 메모리 시스템(VM)에서 페이지 부재로 발생하는 디스크 I/O를 감소시키기 위해 페이지 선반입 기법을 사용한다. 이 기법은 부재 페이지와 함께 추가적인 페이지들을 한 번의 디스크 I/O로 미리 읽는다. 그런데, 4.4BSD와 같은 운영체제의 VM은 응용 프로그램의 페이지 참조 패턴을 고려하지 않고 항상 가능한 많은 페이지들을 선반입하고자 한다. 이 방법은 선반입된 페이지들 중 일부만 사용하는 참조 패턴에서 디스크 참조 시간을 증가시키며, 유용한 페이지들을 메모리에서 내보내는 메모리 오염을 야기한다. 이런 문제를 해결하기 위해 본 논문은 적응적 페이지 선반입 제어 기법(APC)을 제안한다. APC는 선반입 페이지들 중에서 메모리에 존재하는 동안 참조된 페이지들의 비율을 프로세스 단위로 주기적으로 측정하고, 이 비율을 사용하여 4.4BSD VM이 선반입하고자 하는 페이지의 개수를 조절한다. 그래서 실행 도중 페이지 참조 패턴이 바뀌더라도 적절한 수의 페이지를 선반입할 수 있다. 성능 검증을 위해 APC를 4.4BSD 기반의 FreeBSD 6.2에 구현하였으며, SOR, SMM, FFT 벤치마크를 통해 성능을 측정하였다. 성능 측정 결과 APC는 기존 BSD VM보다 벤치마크의 실행 시간을 최대 57% 단축하였다.

키워드 : 가상 메모리 시스템, 페이지 선반입, 4.4BSD 운영체제

Abstract Virtual memory systems (VM) reduce disk I/Os caused by page faults using page prefetching, which reads pages together with a desired page at a page fault in a single disk I/O. Operating systems including 4.4BSD attempt to prefetch as many pages as possible at a page fault regardless of page access patterns of applications. However, such an approach increases a disk access time taken to service a page fault when a high portion of the prefetched pages is not referenced. More seriously, the approach can cause the memory pollution, a problem that prefetched pages not to be accessed evict another pages that will be referenced soon. To solve these problems, we propose an adaptive page prefetching control scheme (APC), which periodically monitors access patterns of prefetched pages in a process unit. Such a pattern is represented as the ratio of referenced pages among prefetched ones before they are evicted from memory. Then APC uses the ratio to adjust the number of pages that 4.4BSD VM intends to prefetch at a page fault. Thus APC allows 4.4BSD VM to prefetch a proper number of pages to have a better effect on reducing disk I/Os, though page access patterns of an application vary in runtime. The experiment of our technique implemented in FreeBSD 6.2 shows that APC improves the execution times of SOR, SMM, and FFT benchmarks over 4.4BSD VM by up to 57%.

Key words : virtual memory system, page prefetching, 4.4BSD operating system

· 이 논문은 2010년도 광운대학교 교내 학술 연구비 지원에 의해 연구되었음
· 본 연구는 2009년도 가톨릭대학교 교비연구비의 지원으로 이루어졌음

† 정 회 원 : 광운대학교 컴퓨터소프트웨어학과 교수
whahn@kw.ac.kr
†† 정 회 원 : LG전자 MC사업본부 연구원
shining-soul@hanmail.net
††† 종신회원 : 가톨릭대학교 컴퓨터정보공학부 교수
jwoh@catholic.ac.kr
(Corresponding author임)

논문접수 : 2010년 2월 25일
심사완료 : 2010년 3월 29일

정보과학회논문지: 시스템 및 이론 제37권 제3호(2010.6)

1. 서 론

최근 물리 메모리의 대용량화에 따라 매우 많은 정보들을 메모리에 저장할 수 있게 되었다. 하지만 작업량이 큰 응용 프로그램을 실행하거나 다수의 프로그램들을 동시에 수행할 경우 여전히 메모리가 부족한 현상이 생길 수 있다. 메모리 부족 현상을 해결하기 위해 디스크의 스왑(swap) 영역을 메모리의 일부분으로 사용한다. 물리 메모리와 스왑 영역 사이의 페이지(page) 전송은 가상 메모리 시스템의 페이지-인(page-in)과 페이지-아웃(page-out) 동작에 의해 이루어지며, 페이지-인과 아웃 동작에서 디스크 I/O가 발생한다. 그런데 디스크는 I/O 속도 면에서 메모리와의 격차가 매우 크기 때문에 [1] 가상 메모리 시스템을 통해 발생하는 디스크 I/O의 횟수는 시스템 성능에 큰 영향을 미치게 된다.

가상 메모리 시스템에서 발생하는 디스크 I/O 횟수를 감소시키기 위해 여러 방법들이 연구되어 왔다. 그 중 페이지 교체(page replacement)[2-6] 기법들은 메모리가 부족할 때 가장 불필요하다고 판단되는 메모리상의 페이지를 교체함으로써 페이지 부재의 횟수를 줄인다. 이 페이지 부재의 감소는 디스크 I/O와 연관된 페이지-인 횟수를 줄어들게 한다. 또한 페이지-아웃에서의 디스크 I/O 감소를 위해, 여러 페이지들을 하나의 페이지로 압축하여 저장하는 스왑 압축 기법[7]과 한 번의 디스크 I/O로 다수의 페이지들을 스왑 영역에 저장하는 클러스터링(clustering) 기법[8,9] 등이 연구되었다.

페이지-인으로 발생하는 디스크 I/O의 횟수를 개선하기 위해 페이지 선반입(page prefetching) 기법을 사용하는 연구도 진행되었다. 선반입 기법은 페이지 부재 발생 시 스왑 영역에서 부재 페이지와 함께 추가적인 페이지들을 한 번의 디스크 I/O로 미리 읽는다. 선반입된 페이지에 대한 접근이 발생하면 요구된 페이지가 이미 메모리에 존재하므로 추가적인 I/O가 발생하지 않는다. 그동안 연구된 페이지 선반입 기법들은 응용 프로그램 자체 또는 컴파일러가 제공하는 페이지 참조 패턴에 대한 힌트를 이용하는 기법[10-13], 가상 메모리 시스템에서 페이지의 참조 패턴을 식별하여 이용하는 기법[14-17], 선반입된 페이지들을 저장할 메모리 영역의 크기를 조절하는 기법[18] 등이 있다. 하지만 이러한 기법들은 OSF/1[8], 리눅스[19], 윈도우즈[20], 4.4BSD[21]와 같은 상용 운영체제들에서 사용되지 않고 있다. 대신 이 상용 운영체제들은 부재 페이지와 가상 주소 공간에서 연속적인 페이지들 중 스왑 영역에 연속적으로 저장된 페이지들을 한 번의 디스크 I/O로 선반입한다.

기존 상용 운영체제들의 선반입 기법들은 응용 프로그램마다 페이지 참조 패턴이 동일하지 않음에도 불구하고, 항상 가능한 많은 페이지들을 선반입하는 획일적인 정책을 사용한다. 하지만, 응용 프로그램은 선반입된 페이지들 중 일부만 참조할 수 있다. 이런 참조 패턴에서 페이지-인 시 참조되지 않을 불필요한 페이지들이 함께 선반입됨으로써 디스크 참조 시간이 증가할 수 있다. 또한 메모리가 부족한 상황에서, 불필요하지만 선반입이 되는 페이지들은 오히려 유용한 페이지들을 메모리에서 내보내는 메모리 오염(memory pollution) 문제를 발생시키게 된다.

본 논문에서는 4.4BSD(이하 BSD로 칭함) 계열 운영체제의 선반입 기법이 가지는 불필요한 선반입 및 메모리 오염 문제를 해결하기 위해 적응적 페이지 선반입 제어(adaptive page prefetching control, APC) 기법을 제안한다. APC는 선반입 페이지들 중에서 메모리에 존재하는 동안 참조된 페이지들의 비율(prefetch hit ratio, PHR)을 측정하고, 이 비율을 사용하여 기존 BSD VM이 선반입하려는 페이지의 개수를 조절한다. 실행 시간에 PHR을 주기적으로 측정하여 실행 도중 페이지 참조 패턴이 바뀌더라도 적절한 수의 페이지를 선반입할 수 있다. 또한 APC는 여러 프로세스가 동시에 실행되는 상황에서 각 프로세스의 개별적인 선반입 비율을 고려하여 효과적으로 페이지들을 선반입한다.

PHR 측정 방법은 부하가 거의 발생하지 않는다. BSD VM은 페이지 교체를 위해 메모리상의 페이지들에 대해 참조 기록을 검사한다. APC는 기존의 페이지 검사 과정을 활용하여 PHR을 측정하므로 별도의 프로세스 및 추가적인 페이지 검사를 필요로 하지 않는다. 또한 APC를 기존 가상 메모리 시스템의 큰 수정 없이 추가할 수 있다.

본 논문에서는 성능 검증을 위해 APC를 4.4BSD 기반의 FreeBSD 6.2 커널에 구현하였다. FreeBSD는 현재 개인 및 서버용 컴퓨터에 널리 사용되는 운영체제이다. SOR, SMM, FFT 등 여러 벤치마크(benchmark) 프로그램을 실행하였고, 실행한 결과 기존 BSD VM에 비해 APC에서의 실행 시간이 최대 57% 단축되었다.

본 논문의 구성은 다음과 같다. 2장에서는 기존 BSD VM의 선반입 동작 과정에 대해 설명한다. 3장에서는 본 논문의 연구 동기를 설명한다. 4장에서는 BSD VM의 선반입 정책을 개선한 APC의 동작 원리와 특징에 대한 설명을 하며, 5장에서는 실험과 분석 결과에 대해 설명한다. 6장에서는 관련 연구에 대해 설명하고, 마지막 7장에서는 결론을 도출하여 정리한다.

2. 배경 지식

BSD VM은 가상 메모리상의 페이지들을 전역적으로 관리하기 위해 세 개의 페이지 리스트를 사용한다. 세

개의 리스트는 활성 리스트(active list), 비활성 리스트(inactive list), 자유 리스트(free list)이다. 활성 리스트는 최근에 활발히 참조된 페이지들을 관리하고, 그중에 오랫동안 참조되지 않은 페이지들은 비활성 리스트의 끝(tail)으로 이동되어 관리된다. 그리고 비활성 리스트의 페이지가 접근되면 다시 활성 리스트로 이동되며 자유 리스트는 메모리상의 빈 페이지를 관리한다. 자유 리스트의 빈 페이지들은 페이지 부재 처리 과정에서 새로운 페이지를 할당하기 위해 사용된다.

페이지 데몬(page-out daemon)이 활성 리스트와 비활성 리스트를 관리하며, 이 데몬은 주기적으로 수행되거나 혹은 메모리가 부족할 때 실행되어 여유 메모리를 확보한다. 페이지 데몬이 실행되면 우선 메모리에서 오랫동안 사용되지 않은 페이지들을 교체하기 위해 그림 1과 같이 비활성 리스트를 탐색한다. 이 탐색은 비활성 리스트에서 가장 오래된 페이지가 존재하는 리스트의 처음(head)부터 리스트의 끝(tail) 방향으로 진행한다. 비활성 리스트의 페이지들은 각각의 상태에 따라 다르게 처리된다. 탐색된 페이지가 참조된 기록이 없고 페이지의 내용이 변경되지 않았다면 자유 리스트로 옮겨진다. 또한 참조된 페이지들은 활성 리스트의 끝으로 옮겨진다. 만일 내용이 변경된 더티(dirty) 페이지가 참조되지 않았다면 스왑 영역으로 페이지-아웃된다. 이때 페이지-아웃될 페이지와 가상 주소 공간에서 연속적인 페이지들이 한꺼번에 묶여 디스크에 저장된다. 페이지 데몬은 비활성 리스트의 모든 페이지를 탐색하지 않고, 대신 자유 리스트가 보유해야할 빈 페이지들이 특정 임계값만큼 확보되면 비활성 리스트의 탐색을 중단한다. 따라서 비활성 리스트의 일부분의 페이지들만이 탐색되어 자유 리스트로 이동될 수 있다.

페이지 부재가 발생하면 요청된 페이지를 디스크로부터 메모리로 적재하는 페이지-인 동작이 수행된다. BSD VM은 페이지-인 과정에서 페이지 부재를 일으킨 페이지뿐 아니라 여러 페이지들을 한 번의 디스크 I/O로 함께 읽는 선반입을 수행한다. 따라서 선반입된 페이지를 접근하면 추가적인 디스크 I/O 없이 접근이 가능하다.

BSD VM의 선반입은 다음과 같은 두 가지 조건을 만족하는 페이지들을 대상으로 하며 최대 15개의 페이지를 선반입한다. 첫째, 페이지 부재를 발생시킨 페이지와 가상 주소 공간에서 연속적인 페이지 번호를 가진 페이지이다. 둘째, 페이지 부재를 발생시킨 페이지와 디스크에서 연속적으로 저장된 페이지이다. 선반입된 페이지들은 곧 접근될 가능성이 높기 때문에 비활성 리스트의 끝에 추가된다. 하지만 페이지 부재를 발생시킨 페이지는 접근 요청에 의해 적재되었기 때문에 활성 리스트에 추가된다.

그림 2는 스왑 영역에서 부재 페이지의 위치를 기준으로 선반입할 페이지를 탐색하는 과정을 나타낸다. 역방향(왼쪽)과 순방향(오른쪽) 순으로 탐색하며, 탐색 종료를 위한 네 가지 조건은 다음과 같다. 첫째는 7개(순방향은 8개)의 페이지가 탐색되었을 때이며, 둘째는 선반입 조건을 만족하지 않는 페이지가 탐색됐을 때이고, 셋째는 빈 공간이 나타났을 때이다. 마지막 네 번째는 최대 선반입 가능 페이지 수인 15개의 페이지가 탐색되었을 때이다. 디스크 스왑 영역에서 그림 2와 같이 페이지들이 배치되어 있을 때 최초 P_{13}에 대한 부재가 발생하면(①) 우선 P_6까지 역방향으로 탐색한 후(②) P_{14}부터 P_{16}까지 순방향 탐색을 한다(③). 이 때 15개의 선반입 대상 페이지를 찾지 못했다면 다시 역방향 탐색을 한 번 더 시도한다(④). 그래도 15개의 페이지를 찾지 못하고 탐색이 종료되면 선반입 대상 페이지는 현재까지 탐색된 페이지들로 결정된다. 이 경우에 선반입될 페이지의 개수는 11개이다.

BSD VM의 선반입 기법은 페이지 부재로 인한 디스크 I/O 횟수를 줄이는 장점이 있다. 하지만 선반입된 페이지들이 어느 정도 사용되는지에 대해서는 고려하지 않는다. 따라서 불필요한 선반입으로 인해 유용한 페이지를 디스크로 내보내는 메모리 오염이 유발되거나 필

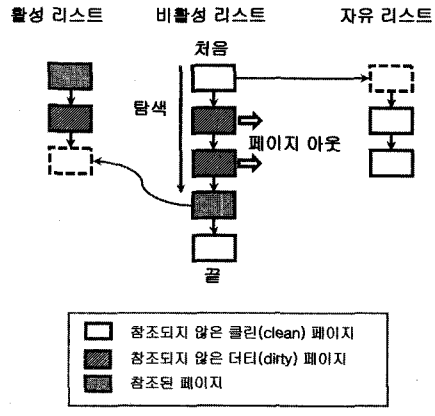

그림 1 비활성 리스트 탐색

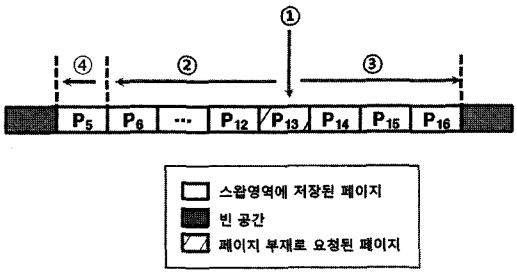

그림 2 선반입 대상 페이지 검색

요 이상으로 디스크 참조 시간이 길어지는 문제점이 있다. 이 문제들은 BSD VM과 유사한 선반입 방법을 사용하는 다른 운영체제들에서도 마찬가지이다. 하지만 기존 상용 운영체제들이 사용하고 있는 선반입 기법의 문제가 가상 메모리 시스템의 성능에 어떠한 영향을 미치는지와 어떻게 개선되어야 하는지에 대한 연구는 그동안 없었다.

3. 연구 동기

본 논문에서는 BSD VM의 선반입이 응용 프로그램의 실행에 어떤 영향을 미치는지 실험을 통해 분석하였다. 실험 환경으로 Pentium 4 3.0GHz CPU, 512MB DDR2 메모리, Seagate Barracuda 7200.9 160GB 7200RPM 하드 디스크를 사용하였고, 스왑 영역은 2GB로 설정하여 세 개의 벤치마크 SOR, SMM, FFT를 각각 실행하였다. 벤치마크들에 대한 구체적인 설명은 5장에서 언급한다.

그림 3은 벤치마크가 실행될 때 선반입된 페이지가 얼마나 효율적으로 사용되는지를 보여준다. 이 실험을 위해 측정한 첫째 요소는 선반입된 페이지들 중 메모리 상에 존재하는 동안 참조되었던 페이지들의 비율 즉, 선반입된 페이지 참조율(prefetch hit ratio, PHR)이다. 둘째는 한 번의 페이지-인 시 선반입된 평균 페이지의 개수이다. 실험 결과 벤치마크마다 PHR과 선반입된 평균 페이지 수가 모두 상이하다. SOR은 거의 100%에 가까운 PHR을 가지기 때문에 선반입의 효과가 매우 크다. 반면 SMM과 FFT는 선반입된 페이지들 중 다수의 페이지가 참조되지 않은 것으로 나타났다. 특히 0%에 가까운 PHR을 가지는 FFT에서는 선반입된 페이지가 거의 참조되지 않기 때문에 선반입의 효과가 거의 없다. 또한 SMM과 FFT의 경우 PHR이 낮음에도 불구하고 평균 선반입 페이지의 개수가 크기 때문에 선반입된 페이지가 메모리 오염을 유발할 수 있다.

그림 4는 선반입되는 페이지의 개수가 벤치마크의 실행 시간에 어떤 영향을 미치는지를 보여주는 실험 결과이다. 이 실험은 BSD VM이 결정한 선반입 페이지들 중에서 특정 비율만큼만 선반입하도록 하였다. 이 비율을 선반입 비율(prefetch ratio)이라고 정의하고, 이 값을 10~100%까지 변화시키며 벤치마크의 실행 시간을 측정하였다. 실험 결과, 최소 실행 시간을 결정짓는 최적의 선반입 비율이 벤치마크마다 다르다. 더욱 중요한 특징은 각 벤치마크에서 최적의 선반입 비율이 그림 3의 PHR과 비슷하다는 것이다. 따라서 가상 메모리 시스템이 프로세스의 PHR을 사용하여 선반입될 페이지의 개수를 조절한다면 그 성능이 향상될 수 있을 것이다.

BSD VM의 선반입 정책은 프로세스의 PHR이 높으면 매우 효과적이다. 하지만 PHR이 낮은 프로세스에

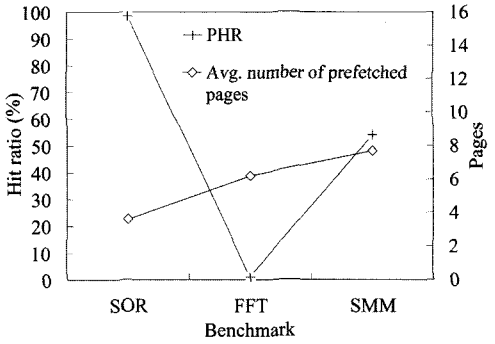

그림 3 벤치마크의 PHR과 평균 선반입 페이지의 개수

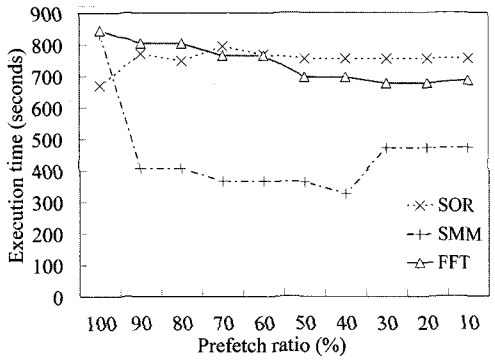

그림 4 선반입 비율에 따른 벤치마크의 실행 시간

대해 오히려 성능을 저하시킬 수 있다. 따라서 PHR이 서로 다른 프로세스에게 획일적인 선반입 정책을 적용하는 것은 부적절하며, 선반입 효과에 직접적인 영향을 미치는 PHR을 고려하여 BSD VM의 선반입 정책을 개선할 필요가 있다.

4. 적응적 페이지 선반입 제어 기법

4.1 기본 개념

페이지 참조 패턴이 프로세스가 실행하는 응용 프로그램에 따라 달라질 수 있다. 그럼에도 불구하고 그 참조 패턴에 관계없이 가능한 많은 페이지를 선반입하는 방식은 자칫 메모리 오염과 디스크 참조 시간의 증가를 야기할 수 있다. 이런 문제를 해결하기 위해 본 논문은 선반입된 페이지의 참조 패턴을 실행 시간에 주기적으로 분석하여 선반입될 페이지의 개수를 조절하는 적응적 페이지 선반입 제어(adaptive page prefetching control, APC) 기법을 제안한다. 이 기법은 각 프로세스로 선반입된 페이지들 중에 참조된 페이지의 개수, 즉, 선반입 페이지의 참조율(prefetch hit ratio, PHR)을 실행 시간에 측정하고, 그 결과를 프로세스의 다음 선반입 동작에 반영하여 선반입될 페이지 개수를 조절한다.

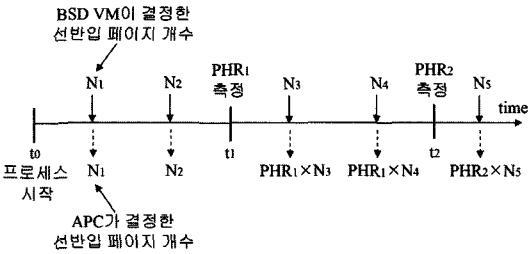

그림 5 APC 기본 동작

그림 5는 APC가 PHR를 통해 어떻게 페이지 선반입을 제어하는지 보여준다. 우선 프로세스가 시간 t_0에서 시작될 때 이 프로세스의 PHR을 알 수 없기 때문에 BSD VM이 결정한 선반입 페이지의 개수만큼 페이지를 읽는다. 이후 t_1 시점에 APC는 시간 t_0과 t_1 사이 선반입된 페이지들에 대한 참조 유무를 확인하여 PHR_1을 측정한다. 측정 완료 이후부터는 BSD VM이 이 프로세스로 선반입할 페이지를 n개로 결정하면 APC는 $PHR_1 \times n$ 개수를 선반입할 페이지의 개수로 결정한다. 예를 들면 프로세스의 PHR_1이 50%이고 BSD VM이 10개의 페이지를 선반입하려고 한다면 APC는 5개의 페이지만 선반입한다. 이후 선반입된 페이지들에 대한 PHR_2를 t_2 시점에 다시 측정하여 이다음부터 선반입할 페이지의 개수를 결정하기 위해 PHR_2를 사용한다.

APC 기법에서 각 프로세스의 PHR 측정은 다음과 같은 이유로 인해 페이지 데몬이 담당한다. 첫째 CPU의 부하를 줄이기 위해서이다. PHR을 측정하려면 선반입된 페이지들이 메모리에서 교체되기 전에 참조 유무를 확인할 필요가 있는데 페이지 데몬도 페이지 교체를 위해 비활성 리스트에서 선반입된 페이지를 포함한 페이지들을 탐색한다. 만일 이 탐색 과정에서 선반입된 페이지의 참조 유무를 확인하면 탐색하는 데몬을 추가하지 않더라도 PHR 측정이 가능하다. 둘째 간단하게 구현하여 PHR를 측정할 수 있다. 만일 비활성 리스트에 있는 페이지를 탐색하는 대신 선반입된 페이지들을 따로 관리하여 측정한다면 복잡한 자료 구조가 요구될 수 있다. 페이지 데몬을 이용하여 비활성 리스트의 페이지들을 탐색하는 방식은 복잡한 자료 구조를 도입하지 않고 페이지를 위한 기존 자료 구조에 소수의 비트(bit)만 추가하여 구현이 가능하다. 셋째 새로운 하드웨어 추가 없이 PHR을 측정할 수 있다.

4.2 PHR 측정 방법

페이지 데몬은 비활성 리스트의 페이지를 탐색할 때 각 페이지의 선반입 여부와 참조 유무를 확인하여 프로세스 단위로 PHR를 측정한다. 이 측정을 위해 프로세스의 자료 구조에 선반입된 페이지의 개수(N_{pf})와 선반입되고 아울러 참조된 페이지의 개수(N_h)를 나타내는 변수를 추가하였으며, 프로세스의 PHR은 N_h/N_{pf}로 측정된다. 이때 비활성 리스트에서 탐색된 페이지가 선반입된 페이지인지를 표시하기 위해 각 페이지의 자료 구조에 선반입 비트(prefetch bit)를 추가하였다. PHR 측정 알고리즘은 이 선반입 비트 외에 BSD VM이 관리하는 페이지의 참조 비트(reference bit)를 활용하여 선반입된 페이지가 참조되었는지를 확인한다.

BSD VM에서 페이지 부재 시 해당 페이지와 함께 선반입된 페이지들은 비활성 리스트의 끝에 배치된다. APC는 이 페이지들의 선반입 비트를 1로 설정한다. 한편, 메모리가 부족할 때 실행되는 페이지 데몬은 페이지 교체를 위해 비활성 리스트에서 가장 오랫동안 접근되지 않은 페이지가 위치한 리스트의 처음부터 끝 방향으로 탐색한다. 이때 측정 알고리즘은 탐색되고 있는 페이지의 선반입 비트를 이용하여 선반입된 페이지인지를 검사한다. 만일 선반입 페이지인 경우, 이 페이지를 포함하는 프로세스의 N_{pf}를 1만큼 증가시킨다. 또한 선반입 페이지의 참조 비트를 통해 이 페이지가 참조되었는지 확인한다. 만일 참조되었다면 이 페이지를 포함하는 프로세스의 N_h 값을 1만큼 증가시킨다.

APC는 특정 시간 동안 선반입된 페이지들을 PHR 측정을 위한 선반입 페이지 집합으로 구성하며, 이 페이지 집합의 크기가 N_{pf}에 해당된다. 이 특정 시간 구간을 선반입 간격(prefetch interval, PI)라고 하며, 이 PI의 최소 단위로 페이지 데몬의 실행 시간 간격(inter-execution time, IET)을 사용한다. 페이지 데몬이 시간 t_0, t_1, t_2,...에 수행된다고 가정하자. 만일 한 개의 IET로 PI를 설정한다면 t_0과 t_1 사이에 선반입된 페이지들을 대상 페이지 집합으로 보고 PHR을 측정한다. 그리고 PHR 측정이 완료된 시점 $t_i(t_1<=t_i)$부터 프로세스의 선반입에 PHR를 반영한다. 이때 t_i가 t_1보다 클 수 있는 이유는 t_1시점에 페이지 데몬이 비활성 리스트의 끝까지 페이지를 탐색하기 전에 자유 리스트에 있는 빈 페이지의 개수가 임계값에 도달하여 탐색이 중지될 수 있기 때문이다. t_i 이후 t_1과 t_2 사이에 선반입된 페이지 집합에 대한 PHR의 측정 과정은 앞서와 같이 수행된다. 따라서 APC는 선반입된 페이지들의 참조 패턴을 주기적으로 PHR에 반영하여 실행 도중 페이지 참조 패턴이 바뀌더라도 적절한 수의 페이지를 선반입할 수 있다.

본 논문에서는 임의의 n개의 IET로 구성된 시간 간격을 PI로 설정하여 일반화된 PHR 측정 알고리즘을 설계한다. 즉 대상 페이지 집합을 시간 t_0과 t_n 사이에 선반입된 페이지들로 보고 PHR를 측정한다. 이 측정이 완료된 시점 $t_i(t_n<=t_i)$부터 이 PHR를 사용하여 프로세스가 선반입할 페이지의 개수를 조절한다. 아울러 완료 후부터 t_n과 t_{2n} 사이에서 선반입된 페이지들을 대상 페

이지 집합으로 보고 PHR을 측정하는 과정을 앞서와 같이 수행한다.

4.3 PHR 측정 시작과 완료 시점

선반입 페이지 집합에 대한 PHR의 측정을 언제 하느냐가 측정의 정확성에 영향을 미친다. 만일 매우 과거에 선반입된 페이지를 조사한다면 최근의 페이지 참조 패턴이 PHR에 반영되지 못한다. 반면에 페이지가 선반입된 후 곧바로 참조 패턴을 조사한다면 페이지가 참조될 수 있는 충분한 시간이 주어지지 않고 PHR이 측정될 수 있다. 즉, 페이지가 가까운 미래에 참조될 수 있음에도 불구하고 선반입 후에 곧바로 참조 패턴을 PHR에 반영한다면 참조되지 않은 페이지로 측정될 것이다.

APC에서 임의의 PI에 선반입된 페이지 집합의 PHR은 페이지 데몬이 비활성 리스트에서 이 페이지 집합에 속하는 페이지를 처음으로 탐색하는 시점부터 측정이 시작된다. 그 이유는 BSD VM 페이지 교체 정책과 동일하게 선반입된 페이지들에게 PHR의 측정에 앞서 참조될 수 있는 시간적 여유를 주기 위해서이다. BSD VM에서 페이지 데몬은 비활성 리스트에 있는 페이지의 최근 참조 유무를 검사하기 위해 리스트의 처음부터 끝으로 탐색하면서 페이지의 참조 비트를 조사한다. 이 과정에서 페이지의 참조 유무에 따라 페이지를 활성 리스트 또는 자유 리스트로 이동하거나 페이지-아웃한다. 한편 APC는 이 과정에서 페이지의 선반입 여부와 참조 유무를 조사한다. 결국 APC의 PHR 측정 방식은 BSD VM의 페이지 교체 방식만큼 최근 참조 패턴을 적절히 반영하며, 선반입 페이지에게 참조될 수 있는 충분한 시간을 부여한다고 할 수 있다.

그림 6은 PI_i에서 선반입된 페이지 집합에 대한 PHR 측정의 시작과 완료 시점을 보여준다. PI는 한 개의 IET라고 가정한다. 또한 그림 6의 진행 과정 동안 활성 리스트에서 비활성 리스트로 이동하는 페이지는 없다고 가정한다. 먼저 그림 6(a)의 PI_i에서 페이지 P_0, P_4에 대해 페이지 부재가 발생하면 BSD VM은 P_1, P_2와 P_5, P_6을 그림 6(b)에 나온 시간 t_i의 비활성 리스트로 선반입한다. 그 후 페이지 데몬이 실행되는 시간 t_{i+1}에서의 비활성 리스트 상태는 그림 6(b)에 나타난다. t_{i+1}에서 페이지 데몬이 실행되는 동안 비활성 리스트에서 자유 리스트로 5개의 페이지를 이동한다고 가정하면, 페이지 데몬은 PI_{i-1}에 선반입된 페이지 집합의 마지막 페이지인 P_k를 탐색하는 시점에서 PI_{i-1}의 PHR 측정을 완료한다. 반면에 PI_i에 선반입된 페이지 집합의 첫 번째 페이지(P_1)를 탐색하는 시점 t_{i+1}에서 PI_i를 위한 PHR 측정이 시작된다. 이 측정은 시간 t_{i+2}에 실행되는 페이지 데몬이 이 집합의 마지막 페이지 P_6을 탐색하는 시점에서 완료된다.

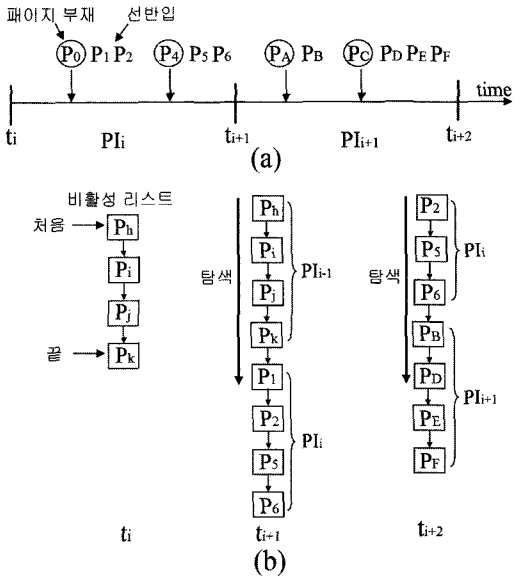

그림 6 PHR 측정 과정

페이지 데몬이 임의의 PI에 선반입된 페이지들을 모두 탐색하였을 때 PHR의 측정을 완료할 수 있도록 각 선반입된 페이지에 PI 번호를 부여한다. 초기 PI는 0이며, 다음 PI부터는 1씩 증가된 번호를 부여한다. 각 페이지는 PI 번호를 저장하는 변수를 가지며, 페이지가 선반입될 때 페이지에 현재 PI가 설정된다. 각 PI의 선반입된 페이지들은 비활성 리스트의 끝에 위치하며 데몬이 처음부터 탐색하기 때문에 페이지 데몬은 낮은 값의 PI를 갖는 페이지부터 PI 번호가 단조 증가하는 순서로 선반입된 페이지를 탐색한다. 페이지 데몬이 PI_i에 속하는 페이지 집합을 탐색하다가 1이 높은 PI_{i+1}번호로 설정된 페이지를 처음 만날 때 PI_i에 선반입된 페이지 집합을 포함하는 프로세스들의 PHR 측정을 완료한다.

4.4 APC 알고리즘

그림 7은 페이지 부재 시 BSD VM이 선반입하려는 페이지의 개수를 APC가 어떻게 조절하는지를 나타내는 알고리즘이다. 이 알고리즘은 BSD VM에서 페이지 부재를 처리하는 핸들러(handler)에서 구현되었다(줄 2-6). 이 핸들러는 우선 BSD VM의 선반입 방식을 통해 부재 페이지와 함께 선반입할 페이지의 수를 결정한다(줄번호 1). APC는 이 페이지의 개수에 PHR 측정 기법으로 결정된 해당 프로세스의 PHR를 곱한 수를 선반입할 페이지 개수로 결정한다(줄번호 2). 부재 페이지에 이 선반입할 페이지 개수를 합한 개수만큼의 페이지들이 단일 디스크 I/O로 메모리로 적재된다(줄번호 3). 향후 페이지 데몬이 해당 프로세스의 PHR를 측정할 수 있도록 선반입된 페이지의 선반입 비트를 1로, 페이지의 PI

```
// let PHRi be prefetch hit ratio of process i causing page
   fault, and PHRi is set to 1 at system booting time.
// let cur_pi be the current prefetch interval

1: set bsd_pref_page_num to the number of pages which
   BSD VM intends to prefetch
2: set pref_page_num to (bsd_pref_page_num * PHRi)
3: fetch (pref_page_num + 1) pages from disk location of
   faulted page using BSD VM routine
4: set prefetch bits of prefetched pages to 1
5: set prefetch bit of faulted page to 0
6: set PIs of prefetched pages to cur_pi
7: execute the rest of BSD VM routine to handle page fault
```

그림 7 선반입 알고리즘

```
// let smallest_pi be smallest one among PIs of pages with
   prefetch bit of 1 in inactive list
// let NPFi be the total number of pages prefetched by
   process i during a PI
// let NHi be the total number of pages that process i has
   referenced among prefetched pages during a PI
// let cur_pi be the current PI
// let cur_iet be the current IET in cur_pi
// all above parameters are set to 0 at system booting
   time
// let iet_per_pi be the total number of IETs per PI
// let threshold be the minimum number of pages in free list

1:  set scan_page to page on head of inactive list
2:  do {
3:    if (prefetch bit of scan_page == 1) then {
4:      set scan_page_pi to PI of scan_page
5:      if (scan_page_pi > smallest_pi) then {
6:        foreach (process i for pages prefetched during
                  smallest_pi) {
7:          set PHRi to (NHi / NPFi)
8:          set NHi to 0
9:          set NPFi to 0
10:       }
11:       set smallest_pi to scan_page_pi
12:   }
13:   increment NPFi by 1
14:   set prefetch bit of scan_page to 0
15:   if (reference bit of scan_page == 1) then {
16:     increment NHi by 1
17:   }
18:  }
19:  execute BSD VM routine either to page out scan_page
     or to move scan_page to active list or free list
20:  set scan_page to the next page of scan_page in
     inactive list
21: } while (the number of pages on free list < threshold
     && scan_page is not at tail of inactive list)
22: increment cur_iet by 1
23: if (cur_iet == iet_per_pi) then {
24:   increment cur_pi by 1
25:   set cur_iet to 0
26: }
```

그림 8 PHR 측정 알고리즘

를 현재 PI 값으로 설정한다(줄번호 4-6).

그림 8은 페이지 데몬이 실행될 때 수행되는 PHR 측정 과정을 보여주는 알고리즘이다. 이 알고리즘은 페이지 데몬이 비활성 리스트를 탐색하는 과정에서 수행된다. 우선 페이지 데몬은 비활성 리스트의 처음부터 각 페이지를 탐색하며, 페이지의 선반입 여부를 조사하기 위해 선반입 비트를 검사한다(줄번호 3). 이 비트가 0이면 선반입 페이지가 아니므로, 페이지는 기존 BSD VM의 페이지 관리 기법을 통해 활성 리스트나 비활성 리스트로 이동되거나 페이지-아웃된다(줄번호 19). 만일 선반입 비트가 1이면 페이지는 선반입 페이지이다. 이때는 탐색 중인 페이지에 설정된 PI동안 선반입된 페이지의 개수를 계산하기 위해 이 페이지를 포함하는 프로세스의 Npf를 1만큼 증가시킨다(줄번호 13). 선반입 여부를 검사한 페이지의 선반입 비트를 0으로 다시 설정한다(줄번호 14). 아울러 선반입 페이지의 참조 유무를 확인하기 위해 참조 비트를 조사하여 1이면 해당 프로세스의 Nh를 1만큼 증가시킨다(줄번호 15-17).

탐색중인 페이지가 선반입 페이지라면, 이 페이지의 PI를 이전까지 탐색된 페이지의 PI와 비교하여 PHR 측정의 완료 시점을 결정한다(줄번호 5). 페이지의 PI가 이전에 탐색된 페이지의 PI보다 크다면, 이전 PI의 선반입 페이지 집합이 모두 탐색되었다는 것을 의미한다. 이때 이 선반입 페이지 집합에 대한 각 프로세스의 N_{pf}와 N_h를 이용하여 PHR 측정을 완료한다(줄번호 6-7). 아울러 나중에 있을 다른 PI에 선반입된 페이지 집합의 PHR 측정을 위해, 측정이 완료된 프로세스들의 N_{pf}와 N_h를 초기화한다(줄번호 8-9).

페이지 데몬은 IET를 사용하여 선반입되는 페이지에 부여될 PI를 주기적으로 갱신한다. 이를 위해 알고리즘에서 단위 PI를 지정된 개수(iet_per_pi)의 IET로 설정한다. 페이지 데몬이 실행될 때마다 페이지 데몬이 관리하는 IET 번호(cur_iet)가 1만큼 증가되며(줄번호 22), 그 값이 iet_per_pi와 동일하면 PI를 1만큼 증가시킨다(줄번호 23-24). 이런 과정을 통해 주기적으로 PI 값이 갱신된다.

5. 실 험

5.1 실험 환경

APC의 성능을 검증하기 위해 다음과 같은 환경에서 실험하였다. CPU는 Pentium 4 3.0GHz, 디스크는 Seagate Barracuda 7200.9 160GB 7200RPM로 구성하였고, 서로 다른 메모리 크기에서 실험하기 위해 DDR2 512MB와 1GB 두 가지 메모리를 사용하였다. 운영체제는 4.4BSD 가상 메모리 시스템(BSD VM)을 탑재한 FreeBSD 6.2를 사용하였고 스왑 영역은 2GB로 설정하였다. 그리고 이 FreeBSD의 가상 메모리 시스템에 APC를 구현하여 BSD VM과 성능을 비교했다. 또한 한 개의

IET를 PI로 설정하여 성능을 측정하였다. 실험에 사용한 벤치마크와 그 설정은 다음과 같다.

- Scimark2 SOR, SMM[22]: Scimark2는 시스템 성능을 측정하기 위한 과학 계산 벤치마크 패키지이며, Java와 C 두 가지 버전의 벤치마크가 있다. 실험에서는 C 버전의 SOR(Successive Over-Relaxation)과 SMM (Sparse Matrix Multiplication) 벤치마크를 사용했으며, 실험 방법에 따라 서로 다른 파라미터를 사용하여 실험하였다. SOR의 입력 배열 크기는 각각 8000×8000과 7000×7000으로 설정했으며, 벤치마크 수행 시 메모리 사용량은 각각 약 502MB와 385MB이었다. SMM은 입력으로 10M×10M의 배열과 100M로 정의한 non-zero, 5M×5M의 배열과 50M로 정의한 non-zero를 각각 사용하였다. 벤치마크 수행 시 메모리 사용량은 각각 약 1338MB와 670MB이었다.

- FFT[23]: 고속 푸리에 변환(Fast Fourier Transform) 알고리즘으로써, 6000×6000 크기의 배열을 입력으로 설정하였다. 프로그램 수행 시 메모리 사용량은 약 1100MB이었다.

5.2 실험 결과

본 논문에서는 512MB 메모리에서 SMM, SOR, FFT를 개별적으로 실행하는 실험과, 512MB와 1GB 메모리에서 세 개의 벤치마크들을 조합하여 동시에 실행하는 실험을 수행하였다.

5.2.1 SMM, FFT, SOR 개별 실행 결과

그림 9는 512MB 메모리에서 SMM, SOR, FFT가 개별적으로 실행될 때 소요된 시간을 나타낸다. 이때 SOR의 입력 배열 크기는 8000×8000으로 설정했으며, SMM은 10M×10M의 배열을 입력으로 사용하였다. 또한 FFT에서는 6000×6000 크기의 배열을 사용하였다.

APC는 BSD VM에 비해 SMM 실행 시간을 약 57% 개선하였다. 이 실험에서 중요한 특징은 APC에서 SMM의 실행 시간이 그림 4에서 나타난 최소 실행 시간과 비슷하다는 것이다. 이를 통해 APC가 거의 정확하게 PHR

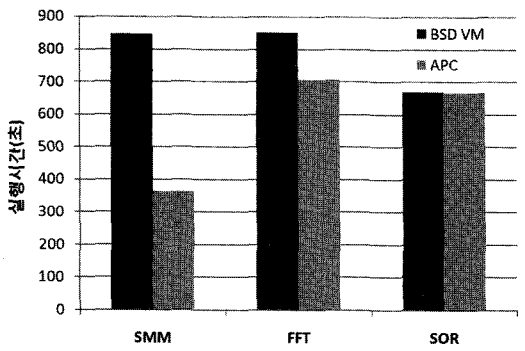

그림 9 벤치마크 실행 시간

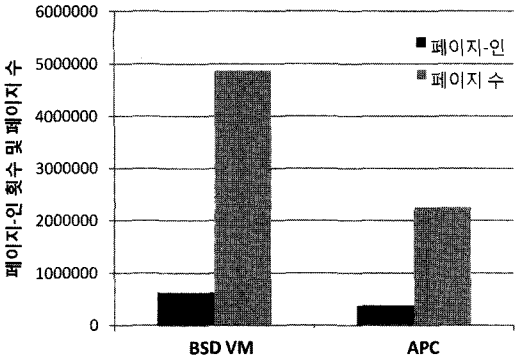

그림 10 SMM(10Mx10M) 실행 시 페이지-인 횟수 및 페이지 개수

을 측정하여 선반입에 반영하고 있음을 알 수 있다. 그림 10은 APC가 SMM의 실행 시간을 개선시킨 원인을 분석하기 위해 페이지-인 횟수와 페이지-인을 통해 읽은 페이지의 개수를 측정한 결과를 나타낸다. APC는 BSD VM보다 페이지-인으로 읽은 페이지의 수를 약 53% 줄였고, 페이지-인 횟수를 약 39% 감소시켰다. 이를 통해 APC가 불필요한 선반입을 막아 메모리 오염을 줄이고 페이지-인 횟수를 감소시켜서 SMM의 실행 시간을 개선하였음을 알 수 있다.

표 1과 그림 11은 APC가 SMM의 실행 시간을 개선시킨 또 다른 원인을 나타낸다. 표 1은 단일 페이지-인 동작에 소요되는 평균 디스크 참조 시간이며, 그림 11은 벤치마크가 실행될 때 발생하는 전체 페이지-인 동작에서 소요되는 총 디스크 참조 시간을 나타낸다. 표 1의 ①과 같이 APC에서 단일 페이지-인에 대한 디스크 참조 시간이 BSD VM보다 감소하였다. 이에 따라 그림 11처럼 APC는 전체 페이지-인 처리에 걸린 총 디스크 참조 시간도 BSD VM보다 개선하였다. 이는 APC가 PHR을 고려하여 불필요한 페이지들의 선반입을 줄이기 때문이다. 결국 디스크 참조 시간은 응용 프로그램의 실행 시간과 직결되므로 APC에서 디스크 참조 시간의 단축은 BSD VM에서보다 SMM의 실행 시간을 개선시킨 이유 중 하나가 된다.

그림 12는 FFT를 실행할 때의 페이지-인 횟수와 페이지-인을 통해 읽은 페이지들의 개수를 나타내며, FFT의 실행 시간은 그림 9에 나타난다. APC는 BSD VM에 비해 FFT의 실행 시간을 약 17% 개선했고, 페이지-인 횟수를 16% 감소시켰다. 또한 페이지-인에 의해 디스크에서 읽은 페이지의 수가 약 73% 감소했다. 그 이유는 그림 3처럼 평균 PHR이 거의 0%인 FFT가 실행될 때 APC가 BSD VM보다 선반입하는 페이지의 개수를 크게 줄였기 때문이다. APC에

표 1 단일 페이지-인에 소요되는 평균 디스크 참조 시간 (milliseconds)

벤치마크 \ 선반입 방법		512MB	
		BSD VM	APC
①	SMM	1.24	0.84
②	FFT	7.15	6.88
③	SOR	3.05	3.07

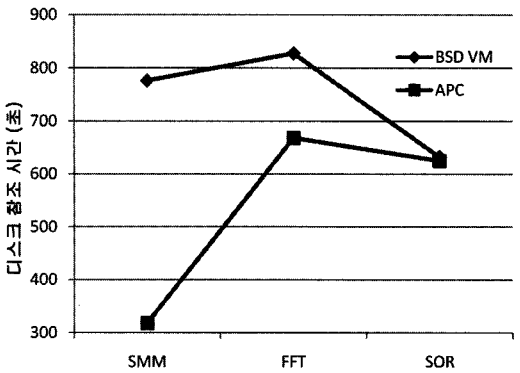

그림 11 전체 페이지-인에 소요되는 총 디스크 참조 시간

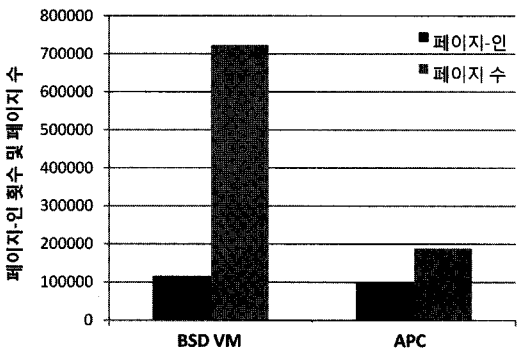

그림 12 FFT 실행 시 페이지-인 횟수 및 페이지 개수

서 FFT의 실행 시간은 그림 4에 나타난 FFT의 최소 실행 시간과 비슷하며, 이를 통해 APC가 FFT의 PHR을 정확하게 측정하고 있음을 확인할 수 있다. 또한 FFT의 실행 시간이 단축된 또 다른 이유는 표 1과 그림 11처럼 APC가 BSD VM보다 디스크 참조 시간을 개선하였기 때문이다.

그림 9에 나오듯이 SMM에 비해 FFT에 대한 APC의 성능 개선이 작다. 그 이유는 FFT의 실행 시간 대비 페이지-인 횟수가 SMM에 비해 적어서 APC를 통한 실행 시간의 개선 효과가 상대적으로 감소했기 때문이다. 결국 APC는 페이지 참조 패턴에 따라 유동적으로 프로세스의 실행 시간을 개선하며, 특히 실행 시간 대비 페이지-인 횟수가 많은 프로세스에서 APC의 성능 개선 효과가 더욱 크다.

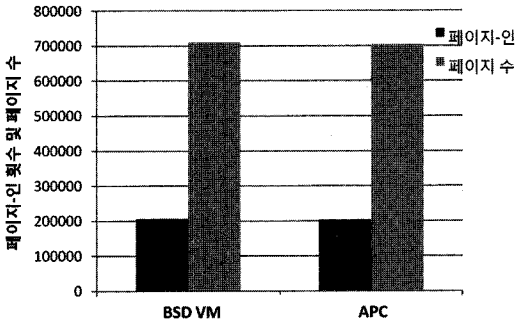

그림 13 SOR(8000×8000) 실행 시 페이지-인 횟수 및 페이지 개수

SOR의 실행 시간은 그림 9에 나오며, BSD VM과 APC에서 거의 동일하다. 아울러 SMM과 FFT의 실험 결과와 유사하게 APC에서 실행된 SOR의 실행 시간이 그림 4에서 나온 최소 실행 시간과 비슷하다. 이런 결과의 이유는 APC가 그림 3에 나온 SOR에 대한 99%의 평균 PHR만큼 정확하게 PHR을 측정하여 선반입할 페이지의 개수를 조절하며, 이 선반입된 페이지의 개수가 BSD VM과 거의 동일하기 때문이다. 이것은 그림 13처럼 APC와 BSD VM에서 거의 동일한 페이지-인 횟수와 페이지-인을 통해 읽은 페이지의 개수로 확인할 수 있다. 결국 매우 높은 PHR를 가지는 프로세스의 경우 BSD VM의 선반입 정책은 매우 효과적이며, APC는 추가적인 부하 없이 BSD VM과 대등한 선반입 성능을 보여주고 있다.

그림 9는 PI 크기를 한 개의 IET로 설정한 APC에서의 벤치마크 실행 시간을 나타낸다. 추가로 PI 크기가 성능에 어떤 영향을 미치는지 살펴보기 위해 PI 크기를 변경하며 실험을 수행하였다. PI 크기는 연속적인 2, 4, 8개의 IET로 각각 설정하였다. 실험 결과 각 경우의 성능이 모두 그림 9의 결과와 비슷하다. 이 결과는 정확한 PHR 측정을 위해 PI를 한 번의 페이지 데몬 실행 시간 간격으로 설정해도 적절하다는 것을 나타낸다.

5.2.2 SMM, FFT, SOR 동시 실행 결과

여러 프로세스가 동시에 실행되어도 APC가 각 프로세스의 PHR을 고려하여 효과적인 선반입을 수행하는지 살펴보기 위한 실험을 수행하였다. 이를 위해 SMM, FFT, SOR을 동시에 실행하여 성능을 측정했으며, 512MB와 1GB의 메모리에서 각각 실험하였다. 이때 입력 배열을 SOR에서 7000×7000, SMM에서 5M×5M, FFT에서 6000×6000으로 설정하였다.

그림 14는 세 개의 벤치마크를 동시에 수행했을 때 모든 벤치마크의 실행이 완료될 때까지 소요된 시간을 나타낸다. 또한 그림 15와 그림 16은 실행 과정에서 발

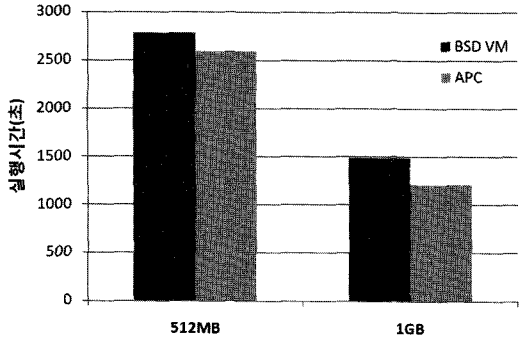

그림 14 메모리 크기별 벤치마크 동시 실행 시간

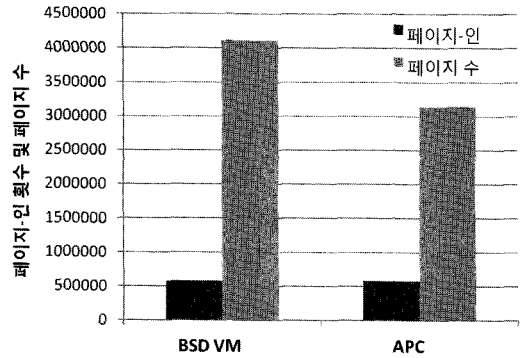

그림 15 512MB 메모리에서 벤치마크를 동시에 실행할 때의 페이지-인 횟수 및 페이지 개수

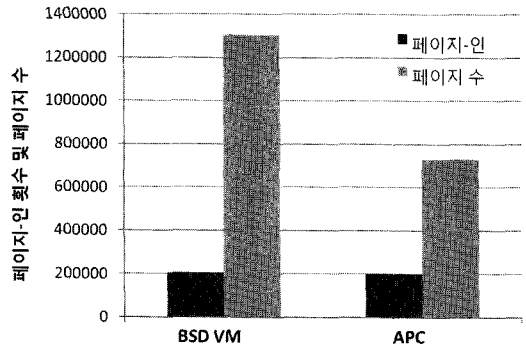

그림 16 1GB 메모리에서 벤치마크를 동시에 실행할 때의 페이지-인 횟수 및 페이지 개수

생한 페이지-인 횟수와 페이지-인을 통해 읽은 페이지 수를 나타낸다. 실험 결과 512MB에서 APC는 BSD VM에 비해 실행 시간을 약 6% 단축하였고, 1GB에서는 약 18% 개선하였다. 또한 512MB 메모리를 사용했을 때 APC에서 페이지-인으로 읽은 페이지 수가 BSD VM에 비해 23%, 1GB에서는 43% 감소했다. 이 결과는 단일 프로세스 수행뿐 아니라 여러 프로세스가 동시에 수행 하더라도 APC가 프로세스 단위로 PHR을 측정하여 선반입할 페이지의 개수를 효과적으로 조절하고 있음을 나타낸다. 또한 시스템 전체에 걸쳐 획일적인 선반입 정책을 사용하는 BSD VM보다 APC의 적응적 선반입 정책이 효과적임을 알 수 있다.

6. 관련연구

선반입 기법을 사용하여 가상 메모리 시스템의 성능을 개선하기 위한 여러 연구들이 제안되었다. 그 중 선반입할 페이지를 선택하기 위해 페이지 참조 패턴에 대한 힌트를 사용하는 방법이 있다[10-13]. 이 방법들은 응용 프로그램 자체 또는 컴파일러가 제공하는 페이지 참조 패턴에 대한 힌트를 사용하여 선반입할 페이지를 결정한다. 이처럼 응용 프로그램 수준에서 제공하는 힌트를 사용하면 보다 정확한 정보를 바탕으로 선반입이 가능하다는 장점이 있다. 그러나 힌트 파일을 커널에 제공하거나, 프로그램 실행 전에 접근 패턴을 먼저 분석하는 등의 추가적인 작업이 필요하며 커널 수준과 응용 프로그램 수준간의 빈번한 이동이 발생할 수 있는 단점이 있다. 반면 APC는 응용 프로그램 수준의 지원을 필요로 하지 않는다.

힌트를 사용하지 않고 가상 메모리 시스템 내부에서 페이지의 접근 패턴을 예측하는 기법[14-17]이 연구되었다. 이 기법들은 실시간으로 과거의 페이지 참조 기록을 분석하여 가까운 미래에 참조될 가능성이 있는 페이지를 선반입한다. 하지만 특정 페이지 참조 패턴에서만 선반입 효과가 있거나, 분석된 패턴과 다른 패턴이 실행 중에 발생하면 부정확한 선반입이 유발될 수 있다. 아울러 이 기법들은 선반입할 페이지를 선택하는 것에 초점을 두었지만, APC는 가상 메모리 시스템이 선반입할 페이지의 개수를 조절하며, 페이지 참조 패턴을 프로세스 단위로 분석한다.

OSF/1, BSD 그리고 리눅스와 같은 기존 운영체제들은 지역성(locality)과 연속성(sequentiality)을 기반으로 한 단순한 선반입 기법을 사용한다. OSF/1[8]과 BSD VM[21]은 디스크 상에서 연속적으로 저장된 페이지들 중 부재 페이지와 가상 메모리 공간에서 인접한 페이지들을 선반입한다. 리눅스 VM[19]은 단순히 부재 페이지와 디스크 상에서 연속적으로 저장된 페이지들 중 고정된 수의 페이지를 선반입한다. APC는 선반입 페이지의 수를 동적으로 조정함으로써 기존 운영체제들의 단순한 선반입 기법보다 효과적인 선반입 기능을 제공한다. 또한 기존 운영체제들이 프로세스들의 특성을 고려하지 않고 획일적인 선반입 정책을 사용하는 것에 비해 APC는 프로세스별로 PHR을 계산하여 각각 선반입의 크기를 다르게 적용하는 유연성을 제공한다.

선반입된 페이지들을 저장할 메모리 영역의 크기를 조절하는 연구도 이루어졌다[18]. 이 연구는 참조 히스토그램(hit histogram)을 사용하여 비용과 이익을 주기적으로 계산하여 선반입 페이지용 메모리 영역의 크기를 동적으로 조절한다. 하지만 선반입되어 메모리로 적재된 페이지들 중 일부에 대해서만 참조 유무를 검사하기 때문에 부정확한 예측이 발생할 수 있다. 또한 페이지 부재 시 선반입되는 페이지의 개수가 고정되어 있다는 한계가 있다. 반면 APC는 선반입할 페이지의 개수를 동적으로 조절한다.

파일 시스템에서 선반입 기법을 사용하여 디스크 I/O 횟수를 개선하는 연구들이 이루어졌다. 이들 연구 중에 응용 프로그램 자체가 제공하는 파일 참조 패턴에 대한 힌트를 사용하여 파일 시스템이 선반입할 파일 또는 블록(block)을 결정하는 기법[24,25]과 힌트의 사용 없이 파일 시스템이 파일 접근 패턴을 분석하여 선반입할 파일 또는 블록들을 결정하는 기법이 있다[26]. 아울러 리눅스와 4.4BSD 운영체제의 파일 시스템에서는 파일 내의 블록들이 순차적으로 접근되면 요구된 블록과 연속적인 블록들을 디스크에서 선반입한다[27].

7. 결 론

본 논문은 기존 운영체제의 가상 메모리 시스템이 수행하는 선반입 기법의 문제점을 제시하고 이를 개선한 적응적 페이지 선반입 제어 기법(APC)을 제안하였다. 성능 검증을 위해 APC를 BSD VM에 구현하고, 성능 측정 결과 BSD VM에 비해 최대 57%까지 실행 시간을 단축시켰다. 또한 페이지-인 요청 횟수와 페이지-인을 통해 읽은 페이지 수가 각각 최대 39%, 73%까지 감소한 결과를 보여 APC가 기존 BSD VM의 선반입 기법을 개선하였음을 입증하였다. 특히 평균 선반입 페이지의 개수가 많고 PHR이 낮은 프로세스의 경우 APC 기법이 탁월한 성능 개선 효과가 있음을 보였다. 또한 APC는 프로세스별로 PHR을 동적으로 측정하여 선반입 동작에 반영하므로 실행되는 프로세스의 메모리 사용 패턴이 각각 다르더라도 성능을 높일 수 있다. 마지막으로 APC는 구현이 간단하고 기존 가상 메모리 시스템에 쉽게 추가될 수 있다. 따라서 APC는 가상 메모리 시스템을 사용하는 여러 시스템에 큰 어려움 없이 적용될 수 있을 것으로 예상한다.

참 고 문 헌

[1] R. E. Bryant and D. R, O'Hallaron, *Computer Systems: A Programmer's Perspective*, pp.447-458, Prentice Hall, 2003.

[2] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance," *In Proc. of* the 2002 ACM *SIGMETRICS*, pp.31-42, 2002.

[3] N. Megiddo and D. S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," *In Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST 03)*, pp.115-130, 2003.

[4] S. Bansal and D. Modha, "CAR: Clock with Adaptive Replacement," *In Proc. of the 3rd USENIX Conference on File and Storage Technologies (FAST 04)*, pp.187-200, 2004.

[5] Y. Smaragdakis, S. Kaplan, and P. Wilson, "EELRU: Simple and Effective Adaptive Page Replacement," *In Proc. of SIGMETRICS · 1999*, pp.122-133, 1999.

[6] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement," *In Proc. of USENIX 2005 Annual Technical Conference*, pp.323-336, 2005.

[7] R. Cervera, T. Cortes, and Y. Becerra, "Improving Application Performance through Swap Compression," *In Proc. of USENIX 1999 Annual Technical Conference*, pp.207-218, 1999.

[8] D. Black, J. Carter, G. Feinberg, R. MacDonald, S. Mangalat, E. Sheinbrood, J. V. Sciver, and P. Wang, "OSF/1 Virtual Memory Improvements," *In Proc. of USENIX Mach Symposium*, pp.87-103, 1991.

[9] J. Yang, W. H. Ahn, J. Oh, "MOC: A Multiple-Object Clustering Scheme for High Performance of Page-out in BSD VM," *Journal of KIISE: Computer Systems and Theory*, vol.36, no.6, pp.476-487, Dec. 2009 (in Korean)

[10] A. D. Brown and T. C. Mowry, "Compiler-Based I/O Prefetching for Out-of-Core Applications," *Journal of ACM Transactions on Computer Systems (TOCS)*, vol.19, no.2, pp.111-170, 2001.

[11] S. Cho and Y. Cho, "Page Fault Behavior and Two Prepaging Schemes," *In Proc. of the 1996 IEEE 15th Annual International Phoenix Conference on Computers and Communications*, pp.15-21, 1996.

[12] K. S. Trivedi, "Prepaging and applications to array algorithms," *IEEE Transactions on Computers*, vol.25, no.9, pp.915-921, September 1976.

[13] K. S. Trivedi, "An analysis of prepaging," *Computing*, vol.22, no.3, pp.191-210, 1979.

[14] G. Dini, G. Lettieri, and L. Lopriore, "Caching and Prefetching Algorithms for Programs with Looping Reference Patterns," *The Computer Journal*, vol.49, no.1, pp.42-61, 2006.

[15] I. Song and Y. Cho, "Page Prefetching Based on Fault History," *In Proc. of the 3rd Mach Symposium on USENIX*, pp.203-213, 1993.

[16] J.-L. Baer and G. R. Sager, "Dynamic Improve-

ment of Locality in Virtual Memory Systems," *IEEE Transactions on Software Engineering*, vol.2, no.1, pp.54-62, 1976.
[17] R. N. Horspool and R. M. Huberman, "Analysis and Development of Demand Prepaging Policies," *Journal of Systems and Software*, vol.7, no.3, pp183-194, 1987.
[18] S. F. Kaplan, L. A. McGeoch, and M. F. Cole, "Adaptive Caching for Demand Prepaging," *In Proc. of the 3rd International Symposium on Memory Management (ISMM'02)*, pp.114-126, 2002.
[19] J. Knapka, Outline of Linux memory management system, http://www.thehackademy.net/madchat/ebooks/Mem_virtuelle/linux-mm/vmoutline.html.
[20] A. Siberschats, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 8th Ed., pp.405, Wiley, 2008.
[21] M. K. McKusick and G. V. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*, pp.183, Addison-Wesley, 2004.
[22] Scimark2 benchmark home page. Web site: http://math.nist.gov/scimark2.
[23] FFT home page. Web site: http://www.fftw.org.
[24] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," *In Proc. of the 15th ACM Symp. on Operating Systems Principles*, pp.79-95, 1995.
[25] P. Cao, E. Felten, A. Karlin, and K. Li, "A Study of Integrated Prefetching and Caching Strategies," *In Proc. of the 1995 ACM SIGMETRICS*, pp.188-197, 1995.
[26] H. Lei and D. Duchamp, "An Analytical Approach to File Prefetching," *In Proc. of the USENIX 1997 Annual Technical Conference*, pp.275-288, 1997.
[27] E. Shriver, C. Small, and K. A. Smith, "Why Does File System Prefetching Work?," *In Proc. of the USENIX 1999 Annual Technical Conference*, pp.71-83, 1999.

안 우 현
1996년 경북대학교 전자공학과(학사). 1998년 KAIST 전기 및 전자공학과(석사) 2003년 KAIST 전자전산학과(박사). 2003년~2005년 삼성전자 기술총괄 소프트웨어연구소 책임 연구원. 2006년~현재 광운대학교 컴퓨터 소프트웨어학과 조교수 관심분야는 운영체제, 임베디드시스템, 시스템소프트웨어

양 종 철
2008년 광운대학교 컴퓨터 소프트웨어학과(학사). 2010년 광운대학교 컴퓨터과학과(석사). 2010년~현재 LG전자 MC 사업본부 연구원. 관심분야는 운영체제, 임베디드시스템

오 재 원
1997년 서울대학교 계산통계학과(학사) 1999년 서울대학교 전산과학과(석사). 2004년 서울대학교 전기컴퓨터공학부(박사) 2004년~2007년 삼성전자 기술총괄 소프트웨어연구소 책임 연구원. 2007년~2009년 가톨릭대학교 컴퓨터정보공학부 전임강사. 2009년~현재 가톨릭대학교 컴퓨터정보공학부 조교수 관심 분야는 소프트웨어 품질, 소프트웨어 공학, 모바일 SW 플랫폼, 시스템소프트웨어