참조패턴을 이용한 선반입의 개선

(Improving Prefetching Effects by Exploiting Reference Patterns)

이 효 정 † 도 인 환 †

(Hyojeong Lee) (In Hwan Doh)

노 삼 혁 ††

(Sam H. Noh)

요 약 선반입은 I/O 성능 향상을 위해 널리 사용되는 기법 중의 하나이다. 하지만 어떤 참조패턴에 대해서는 선반입을 수행하면 오히려 전체 수행시간이 증가하는 경우가 보고된 바 있다. 본 논문은 기존의 선반입 기법에 쉽게 적용될 수 있는 프레임 IPRP(Improving Prefetching Effects by Exploiting Reference Patterns)를 제안한다. IPRP는 참조패턴을 자동으로 탐지하고 기존의 선반입을 참조패턴의 특성에 따라 조정하여 개선하고자 한다. IPRP를 리눅스 미리 읽기 선반입에 적용한 성능평가에서 리눅스 미리 읽기 선반입이 수행시간을 40%~70% 정도 증가시키는 악영향을 발휘할 때 IPRP를 적용할 경우 악영향을 완전히 방지했다. 리눅스 미리 읽기 선반입이 성능 향상을 가져오는 경우에도 리눅스 미리 읽기와 유사한 성능을 가져 왔다. 이 결과를 통해 IPRP가 기존의 선반입을 효율적으로 보완 및 개선할 수 있음을 알 수 있다.

키워드 : 선반입, 참조패턴

Abstract Prefetching is one of widely used techniques to improve performance of I/O. But it has been reported that prefetching can bring adverse result on some reference pattern. This paper proposes a prefetching frame that can be adopted on existing prefetching techniques simply. The frame called IPRP (Improving Prefetching Effects by Exploiting Reference Patterns) and detects reference patterns online and control prefetching upon the characteristics of the detected pattern. In our experiment, we adopted IPRP on Linux read-ahead prefetching. IPRP could prevent adverse result clearly when Linux read-ahead prefetching increases total execution time about 40~70%. When Linux read-ahead prefetching could bring some benefit, IPRP with read-ahead performed similar or slightly better benefit on execution time. With this result we could see our IPRP can complement and improve legacy prefetching techniques efficiently.

Key words : Prefetching, Reference Pattern

1. 서 론

디스크와 메모리의 속도 차이를 극복하기 위한 캐시에서 선반입은 교체정책과 함께 널리 연구되고 사용되어 왔다. 선반입의 목적은 참조가 예상되는 블록을 미리 요청하여 계산시간과 I/O를 중첩하여 I/O로 인한 지연을 감추는 것이다.

선반입이 성공적으로 수행되었을 때는 성능 향상을 얻을 수 있지만 선반입이 잘못 수행되었을 때는 성능에 악영향을 일으킬 위험이 있다[1]. 선반입이 부정적인 영향을 가져오게 되는 이유는 사용되지 않을 블록을 선반입할 경우 그 비용이 작을 수 있다고 해도 추가적인 I/O가 발생하기 때문이다. 또한 잘못 선반입한 블록 때문에 기존의 유용한 블록이 쫓겨나게 될 경우 쫓겨난 블록을 재 반입 하기 위한 오버헤드가 발생한다.

기존의 커널 선반입은 대체로 참조의 순차성에 근거해 선반입을 수행한다. 이러한 기법은 순차성이 클 때는 상당한 효과를 거둘 수 있지만 순차성이 떨어질 때 잘못된 선반입을 수행할 위험이 높다. 이러한 위험을 줄이기 위해 순차성의 강도에 따라 선반입의 강도를 조절하는 기법이 널리 사용된다. 그러나 그러한 점진적인 기법의 적용에도 불구하고 잘못된 선반입이 지속적으로 일어나 전체 수행시간이 최대 두 배까지 증가하는 성능 악화의 위험이 있음이 Hu의 논문을 통해 보고된 바 있다[2].

순차성이 높은 참조에서도 잘못된 선반입이 발생하여 선반입의 이익을 감소시킬 수 있다. 순차성이 높은 스트림이 반복해서 발생할 때에도 스트림의 끝에서는 순차성이 갑자기 떨어질 수 있다. 이 때 높은 순차성 때문에 사용되지 않을 블록들이 다수 선반입 될 수 있는데, 이것이 반복적으로 일어나면 선반입의 이익을 감소시킬 수 있다.

본 논문에서는 참조패턴을 온라인으로 탐지하고, 탐지

· 이 논문은 2007년도 정부(과학기술부)의 재원으로 한국과학재단의 국가지정연구실사업으로 수행된 연구임(No. R0A-2007-000-20071-0)
· 이 논문은 제34회 추계학술대회에서 '참조패턴을 이용한 선반입의 개선'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 홍익대학교 컴퓨터공학과
hjlee@mail.hongik.ac.kr
ihdoh@cs.hongik.ac.kr
†† 정 회 원 : 홍익대학교 컴퓨터공학과 교수
samhnoh@cs.hongik.ac.kr

논문접수 : 2007년 12월 6일
심사완료 : 2008년 2월 5일

된 패턴의 특성을 활용해 잘못된 선반입을 방지하여 기존의 선반입을 개선할 수 있는 프레임 IPRP(Improving Prefetching Effects by Exploiting Reference Patterns)를 제안한다. IPRP는 참조패턴이 파악될 때까지 선반입을 지연시키고 선반입이 이익을 가져올 것으로 예상되는 패턴에 대해서만 선반입을 수행하도록 한다. 그리고 순환 참조를 탐지하여 반복적인 잘못된 선반입을 방지한다. IPRP는 적은 오버헤드로 기존 선반입의 악영향을 효율적으로 방지하고 추가적인 성능 향상을 이끌어낸다.

본 논문은 다음과 같이 구성되어 있다. 2장에서 관련 연구에 대해 설명한다. 3장에서는 본 논문에서 제안하는 IRPR에 대해 구체적으로 기술하고 4장에서는 이를 이용한 실험 결과를 통해 성능을 비교한다. 끝으로 5장에서 결론을 맺고 차후 연구 과제에 대해 기술한다.

2. 관련 연구

본 절에서는 지금까지의 선반입 연구 기법을 크게 응용 프로그램 힌트를 이용한 선반입과 힌트를 이용하지 않는 선반입으로 나누어 살펴본다.

2.1 응용 프로그램 힌트를 이용한 선반입

예측의 정확성은 선반입의 성공 여부에 있어 가장 중요한 요소 중 하나이다. 많은 경우에 응용 프로그램의 개발자들은 어플리케이션의 참조 형태를 미리 알 수 있다. 이에 착안하여 어플리케이션 개발자로부터 참조형태에 대해 적절한 힌트를 얻어 선반입을 하고자 하는 시도들이 있었다[1,3-6].

이 연구들은 응용 프로그램 개발자가 인터페이스를 통해 참조패턴에 대한 힌트를 제공하도록 했다. 그리고 적절한 응용 프로그램 힌트를 기반으로 선반입을 수행하면 상당한 성능 개선을 얻을 수 있음을 보여주었다. 그러나 인터페이스가 사용하기 편리하다고 해도 응용 프로그래머에게는 개발 오버헤드가 주어진다. 게다가 응용 프로그래머가 힌트를 주지 않으면 선반입을 수행할 수 없을 뿐 아니라 악의적인 개발자가 잘못된 힌트를 줄 경우 문제가 발생할 수 있다. 또한 어플리케이션의 호환성 문제가 발생하기 때문에 힌트를 이용하는 선반입은 주로 특수한 환경에서 이용된다.

2.2 투명한 선반입

앞 절에서 설명한 바와 같이 힌트를 사용하는 선반입 기법은 일반적인 커널에서 사용하기에는 적절하지 못하다. 일반적인 커널 선반입은 응용프로그램의 힌트를 이용하지 않도록 설계된다. 이러한 선반입은 무작위적 접근을 예측하기 어렵기 때문에 참조형태에 공간적인 연속성이 존재한다는 가정하에 선반입을 수행하게 된다. 연속될 블록들이 요청되는 공간적 연속성을 순차성이라고 부른다. 순차성은 자주 나타나는 현상이며 디스크에 물리적으로 가깝게 위치해 있을 경우 많은 분량을 가져오더라도 반입에 소요되는 비용이 비교적 적으므로 이러한 선반입은 합리적이라 할 수 있다[7].

순차적 접근이 일어나지 않을 경우 손해가 발생할 수 있다는 점을 감안하여 일반적인 커널 선반입은 순차 참조를 판단하고 판단 결과에 따라 선반입의 강도를 조절하는 방식을 취한다. 리눅스도 이러한 방식을 따르는데, 리눅스에서 사용하는 선반입은 미리 읽기(read-ahead) 알고리즘이다. 미리 읽기 알고리즘은 참조가 발생하면 연속된 블록들을 미리 읽기 윈도우(read-ahead window)로 인식하여 미리 읽어 들인다. 이렇게 읽어진 페이지들을 미리 읽기 그룹(read-ahead group)이라고 한다. 다음 참조가 미리 읽기 그룹 내에서 발견되면, 커널은 파일에 대한 참조가 순차적이라고 판단하여 미리 정의된 한계치 내에서 미리 읽기 그룹의 크기를 두 배로 늘린다. BSD 역시 유사한 선반입 기법을 취한다.

이러한 점진적인 기법을 사용한다 해도 순차적 접근이 아닐 경우 캐시공간의 낭비가 발생한다. 또, 순차적 접근이라고 해도 그 길이가 짧을 때, 지나친 미리 읽기로 불필요한 I/O대기 시간을 유발하게 된다. Butt 등의 연구에서는 비 순차적인 참조패턴에 대해 리눅스의 미리 읽기 선반입을 수행했을 때 응답시간이 최대 두 배로 증가한 사례를 보여주었다[2].

순차적 접근이 일어난다고 해도 여러 개의 스트림이 병렬적으로 요청될 경우 순차성의 형태가 달라지기 때문에 선반입 기법은 기대했던 것과 다른 결과를 만들어 낼 수 있다. Gill 등과 Shen 등은 순차적인 스트림이 병렬적으로 나타날 때에 관한 연구를 진행했다[8,9].

3. 참조패턴을 이용한 선반입 개선 기법

본 절에서는 참조패턴을 이용한 선반입 개선 기법에 대해 설명한다. IPRP는 참조패턴을 세 가지로 분류한다. 3.1에서 IPRP에 사용할 패턴들과 그 탐지 방법에 대해 기술하고 3.2에서 선반입이 언제 부정적인 영향을 가져올 수 있으며 참조패턴을 이용하여 어떻게 선반입을 개선하는지 기술한다.

3.1 세 가지 참조패턴과 탐지

참조에는 패턴이 존재한다. 캐시 교체정책에서는 성능 향상을 위해 참조의 패턴을 이용한 연구가 다수 존재한다. 본 논문에서는 참조패턴의 특성을 이용하여 선반입 기법의 성능을 향상시키고자 한다.

참조패턴을 가장 단순하게 나누는 방법은 순차적인지 순차적이지 않은지로 판단하는 것이다. 통합버퍼관리기법(UBM)은 참조패턴을 보다 자세하게 세 가지로 분류했다[10]. 캐시 교체정책에서는 이 세 가지 참조패턴의 특성을 이용하여 좋은 성능을 얻어낸 사례가 있다[10,

11]. 본 논문에서도 참조패턴을 세 가지로 분류한다. 세 가지 참조패턴은 다음과 같다.

1. 일정 개수 이상의 연속된 블록 참조들이 한번만 발생되는 순차 참조(sequential references).
2. 순차 참조들이 규칙적인 간격으로 반복해서 발생되는 순환 참조(looping references).
3. 순차 참조와 순환 참조에 속하지 않는 기타 참조(other references).

이 참조패턴들을 선반입 관점에서 바라보면 순차 참조의 경우 상대적으로 선반입의 필요는 크지만, 그 길이를 알 수 없기 때문에 지나치게 적극적인 선반입을 하게 될 경우 성능 저하를 일으킬 위험이 있다.

순환 참조패턴은 일정한 주기를 가지고 반복해서 발생하는 특징을 갖는다. 이 패턴의 과거 정보를 이용하면 미래에 일어날 참조의 형태를 큰 오버헤드 없이 상당히 높은 확률로 예측할 수 있다. 따라서 효율적인 선반입을 적용할 경우 낮은 위험으로 좋은 결과를 기대할 수 있다.

기타 참조의 경우에는 앞의 두 가지 참조패턴에 비해 미래 참조 예측이 매우 어렵다. 특별한 힌트가 없는 기타 참조에 대해서는 선반입을 수행하지 않도록 하는 것이 바람직하다.

참조패턴의 탐지 방법은 UBM의 탐지 방법과 유사하다. 참조가 일어나는 파일 별로 순차성을 추적하여 기타 참조와 순차 참조를 분류하고, 순차 참조가 반복적으로 발생하게 되면 순환 참조로 분류한다.

3.2 참조패턴을 이용한 선반입 개선 기법

사용되지 않을 블록을 선반입하면 추가적인 I/O가 발생한다. 특히 그 과정에서 기존의 유용한 블록이 쫓겨나게 될 경우 쫓겨난 블록을 재 반입 하기 위한 오버헤드가 발생한다. 이와 같은 것을 캐시 오염이라고 부르며 선반입의 악영향이 발생하는 가장 큰 이유가 된다.

리눅스의 미리 읽기 선반입을 비롯한 대부분의 커널 선반입은 기타 참조에서 일어나는 캐시 오염을 줄이기 위해 순차성에 따라 선반입의 강도를 점진적으로 증가시키는 기법을 사용한다. 그러나 적은 양의 선반입을 수행한다고 해도 순차성이 낮은 참조에서는 잘못된 선반입이 반복적으로 일어나 전체 성능이 크게 떨어지게 된다. 이러한 참조패턴은 IPRP가 분류하는 참조패턴 중 기타 참조에 속한다. IPRP는 기타 참조에 대해 선반입을 금지한다.

전체적으로 순차성이 높은 참조에서도 잘못된 선반입이 발생해 선반입의 이익을 감소시킬 수 있다. 순차성이 높은 스트림이 반복해서 발생할 때도 스트림의 끝에서는 순차성이 갑자기 떨어질 수 있다. 이 때 높은 순차성 때문에 사용되지 않을 블록들이 다수 선반입 될 수 있는데, 이것이 반복적으로 일어나면 선반입의 이익을 감소시킬 수 있다. IPRP가 분류하는 세 가지 참조패턴 중 순환 참조패턴이 이에 속한다. IPRP는 순환 참조패턴에 대해서는 순환 참조를 구성하는 순차 참조의 끝을 기억하여 잘못된 선반입의 반복을 방지한다.

요약하면 IPRP는 각 참조패턴에 대해 선반입 할 때 다음과 같은 제한을 두어 선반입의 성능을 개선하고자 한다. 순차 참조가 탐지되면 기존의 선반입을 수행하도록 한다. 순환 참조가 탐지되면 기존의 선반입을 수행한다. 다만 순환 참조를 구성하는 순차 참조의 마지막 블록 이후의 블록을 선반입 하는 것을 방지한다. 위의 두 참조가 아닌 기타 참조로 판단되었을 경우에는 선반입을 금지한다.

IPRP는 위와 같은 방법으로 참조패턴이 선반입에 유리한 형태라는 것이 판단될 때까지 선반입의 수행을 지연하고 순차성이 존재할 때에도 지나치게 적극적인 선반입을 반복하는 것을 방지한다.

4. 성능 평가

본 절에서는 리눅스 미리 읽기 선반입을 토대로 IPRP의 성능을 평가한다.

4.1 실험환경

실험환경은 시뮬레이터를 사용하였다. Butt 등은 리눅스 2.4 커널의 미리 읽기 선반입을 충실히 구현하고 실행 시간을 측정할 수 있는 시뮬레이터 Accusim을 제작했다[2]. 본 논문에서 실험환경으로 사용한 Accusim은 널리 검증된 디스크 시뮬레이터인 Disksim을 기반으로 리눅스의 I/O 구조와 여러 캐시 정책을 구현했다. Accusim에 참조패턴 탐지기를 추가하고 참조패턴 탐지 결과에 따라 선반입 여부 및 선반입 정도를 조정하도록 했다. 참조패턴 탐지기는 요청되는 파일 별로 36byte 크기의 구조체를 유지하며 최대 2048개의 최근 구조체가 해시 테이블에 관리된다. 패턴탐지와 선반입 정도 조정을 위해 매 참조마다 1~3번의 비교연산과 약 10Kbyte 정도의 코드가 추가되었다.

시뮬레이터의 트레이스는 strace를 이용하여 쉽게 얻을 수 있지만 Accusim과 함께 여러 버퍼 캐시 관련 논문에서 사용된 바 있는 트레이스가 제공되었으므로 비교의 공정성을 위해 해당 트레이스들을 사용했다[12-15]. 교체 정책으로는 가장 일반적인 교체정책 중 하나인 LRU를 사용했다.

4.2 트레이스 특성

본 논문에서 사용된 트레이스는 다양한 참조패턴을 가진다. 트레이스에 대한 설명과 참조패턴의 특성을 표 1에 요약하였다.

4.3 실험 결과

실험 결과는 트레이스 특성별로 세 가지로 나누어서

표 1 트레이스 요약

트레이스	순차정도	트레이스 특성
glimpse	74%	/usr 에서 텍스트 스트링 검색
cscope	76%	리눅스 커널 2.4.20 분석
tpc-h	3%	Database MySQL
tpc-r	3%	Database MySQL
multi2	75%	cscope + gcc + viewperf
multi3	16%	glimpse + tpc-h

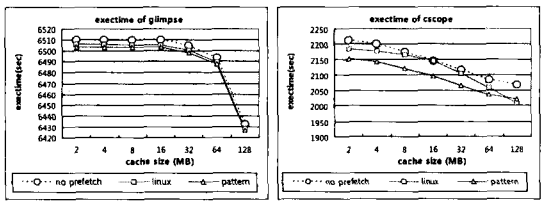

그림 1 순차적인 트레이스들의 캐시 크기 변화에 따른 수행시간 비교 - no prefetch는 선반입을 수행하 않은 결과이고, linux는 리눅스의 미리 읽기 선반입을 pattern은 IPRP를 적용한 결과이다.

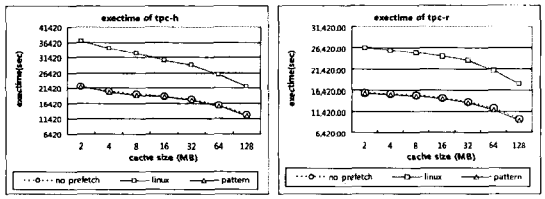

그림 2 순차성이 적은 트레이스들의 캐시 크기 변화에 따른 수행시간 비교

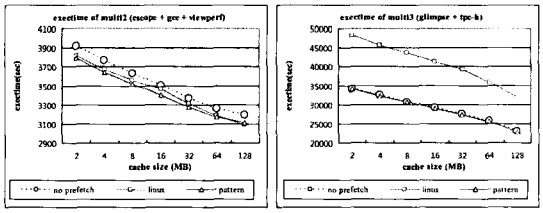

그림 3 Concurrent한 트레이스들의 캐시 크기 변화에 따른 수행시간 비교

설명할 수 있다. 모든 결과는 선반입을 수행하지 않은 경우, 리눅스의 미리 읽기 선반입을 그대로 수행한 경우, IPRP를 리눅스 미리 읽기 선반입에 적용한 경우 의 세 가지 결과를 비교했다.

• 순차성이 포함된 트레이스

순차성이 포함된 트레이스로는 glimpse와 cscope가 있다. 그림 1을 보면 기존의 리눅스 미리 읽기 선반입이 선반입을 수행하지 않았을 때에 비해 성능 향상을 가져오는 것을 확인할 수 있다. IPRP를 적용할 경우 유사하거나 약간의 추가적인 성능 향상을 얻을 수 있음을 확인할 수 있다.

트레이스 glimpse와 cscope의 경우 74~76%의 비교적 높은 순차성을 가진다. 이들은 리눅스 미리 읽기 선반입이 적절하게 동작할 수 있는 참조패턴이다. 리눅스 미리 읽기 선반입을 적용했을 때에 비해 IPRP를 적용하면 glimpse는 거의 유사한 결과(0.01% 향상)를 보이며 cscope의 경우 약 1.3%의 추가적인 향상을 가져온다. 참조패턴 기법이 지연된 선반입을 수행함에도 성능 향상을 얻을 수 있는 이유는 반복적인 잘못된 선반입을 막은 데서 온 이익에 근거한다고 볼 수 있다.

• 순차성이 포함되지 않은 트레이스

트레이스 tpc-h와 tpc-r은 약 3%정도의 낮은 순차성을 가진다.

그림 2를 보면 기존의 리눅스 미리 읽기 선반입이 점진적 이고 소극적으로 수행한다고 해도 캐시 오염과 I/O 낭비를 일으켜 매우 큰 성능 저하를 일으키는 것을 확인할 수 있다. 기존의 선반입을 그대로 적용했을 경우 tpc-h는 67%, tpc-r은 70%정도의 성능 악화가 발생한다. 그러나 IPRP를 적용할 경우 두 트레이스 모두에서 효과적으로 선반입의 악영향을 방지할 수 있을 뿐 아니라 각각 0.9%, 1.28%의 성능 향상도 가져왔다.

• 병행하는(concurrent) 트레이스

앞서 언급한 바와 같이 선반입은 병행하는 트레이스에서는 단일 어플리케이션 트레이스에서와 다른 결과를 보일 수 있을 뿐 아니라 실제 상황은 일반적으로 병행하는 트레이스에 가까울 수 있으므로 병행하는 트레이스에 대한 실험은 선반입 연구에서 중요한 문제이다.

그림 3은 참조패턴을 적용한 기법이 병행하는 트레이스에서도 효과적으로 악영향을 방지하고 선반입 성능을 향상시킬 수 있음을 보여준다. IPRP를 적용할 경우 리눅스 미리 읽기가 성능 향상을 가져오는 multi2에 대해서는 0.83%의 추가적인 성능 향상을 가져왔고 리눅스 미리 읽기가 약 41%의 성능 악화를 유발하는 multi3에 대해서는 성능악화를 방지할 뿐 아니라 약 0.3%의 성능 향상을 가져왔다.

5. 결론 및 향후 과제

잘못된 예측에 기반한 선반입은 성능을 저하시킬 위험이 크다. 이를 해결하기 위해 참조패턴을 활용하여 적은 오버헤드로 성능 저하의 위험을 효과적으로 방지할 수 있는 프레임 IPRP를 개발했다.

본 논문에서 제안한 IPRP는 참조패턴을 순차 참조, 순환 참조, 기타 참조의 세 가지로 분류하고 온라인으로 탐지한다. 성능 악화의 위험을 방지하기 위해 참조패턴

이 탐지될 때까지 선반입의 시작을 지연시키고 순차성이 존재하여 선반입의 성능 향상을 기대할 수 있는 순차 참조와 순환 참조에 대해서만 선반입을 수행하도록 제한하였다. 순환 참조의 경우 순환의 끝을 예측할 수 있으므로 필요치 않은 블록은 선반입하지 못하도록 제한하였다.

성능을 평가하기 위해 시뮬레이션을 통해 리눅스의 미리 읽기 선반입에 IPRP를 적용하고 적용전과 비교하였다. 그 결과 본 논문에서 제안한 참조패턴을 이용한 선반입 개선 기법을 적용할 경우 잘못된 선반입으로 인한 40~70%의 성능 악화를 효과적으로 방지할 뿐 아니라 기존의 선반입이 효과를 발휘하는 상황에서도 유사한 성능 혹은 약 1%의 추가적인 성능 향상을 가져올 수 있는 것을 확인하였다. 또한 기존의 선반입에 쉽게 적용될 수 있다.

본 논문에서 제시한 기법은 참조패턴을 소극적으로 활용한 기법으로서 참조패턴을 보다 적극적으로 이용하면 더욱 높은 성능 향상을 기대할 수 있을 것으로 생각된다. 이에 관한 연구를 향후 과제로 진행하고자 한다.

참 고 문 헌

[1] P. Cao, E. W. Felten, A. R. Karlin and K. Li, "A Study of Integrated Prefetching and Caching Strategies," In *Proceedings of the ACM International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS)*, pp. 188-197, 1995.

[2] A. R. Butt, C. Gniady, and Y. C. Hu, "The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms," In *Proceedings of the ACM International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS)*, pp. 57-168, 2005.

[3] P. Cao, E. W. Felten, A. R. Karlin and K. Li, "Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling," *ACM Transactions on Computer Systems, Vol.14*, No.4, pp. 311-343, 1996.

[4] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pp. 79-95, 1995.

[5] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. Felten, G. Gibson, A. R. Karlin, and K. Li, "A trace-driven comparison of algorithms for parallel prefetching and caching," In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pp. 19-34, USENIX Association, 1996.

[6] A. Tomkins, R. H. Patterson, and G. A. Gibson, "Informed Multi-Process Prefetching and Caching," In *Proceedings of the ACM International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS)*, pp. 100-114, 1997.

[7] B. S. Gill and D. S. Modha, "SARC: Sequential prefetching in adaptive replacement cache," In *Proceedings of the USENIX Annual Technical Conference*, pp. 293-308, 2005.

[8] B. S. Gill and L. D. Bathen, "AMP: Adaptive Multi-stream Prefetching in a Shared Cache," In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 07)*, pp. 185-198, 2007.

[9] C. Li, K. Shen and A. Papathanasiou, "Competitive prefetching for concurrent sequential I/O," In *Proceedings of 2nd European Conference on Computer Systems (EuroSys 2007)*, Mar, 2007.

[10] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "A Low-Overhead, High-Performance Unified Buffer Management Scheme That Exploits Sequential and Looping References," In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 119-134, 2000.

[11] C. Gniady, A. R. Butt, and Y. C. Hu, "Program Counter Based Pattern Classification in Buffer Caching," In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 395-408, 2004.

[12] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. "Towards application / file-level characterization of block references: a case for fine-grained buffer management," In *Proceedings of the ACM International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS)*, June 2000.

[13] S. Jiang and X. Zhang, "LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance," In *Proceedings of the ACM International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS)*, June 2002.

[14] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE Transactions on Computers, Vol.50,* No.12, pp. 1352-1360, 2001.

[15] N. Megiddo and D. S. Modha, "ARC: A Self-tuning, Low Overhead Replacement Cache," In *Proceedings of the 2th USENIX Conference on File and Storage Technologies (FAST 04)*, Mar. 2003.