순환 참조 특성을 기반한 선반입 성능의 개선

이효정° 도인환 노삼혁
홍익대학교 컴퓨터공학과
hjlee@mail.hongik.ac.kr° {ihdoh, samhnoh}@cs.hongik.ac.kr

Performance Enhancement through Prefetching Based On Looping Reference Characteristics

Hyojeong Lee°, In Hwan Doh, Sam H. Noh
Department of Computer Engineering, Hong-Ik University

요 약

버퍼캐시에서 선반입은 교체정책과 함께 중요한 성능 향상 기법 중의 하나이다. 하지만 참조 패턴의 특성에 따라서는 선반입을 수행하면 오히려 전체 수행시간을 증가시키는 경우도 보고된 바 있다. 본 논문에서는 참조 패턴을 탐지하고 탐지된 패턴에 적절히 대응하여, 선반입의 이익은 유지하되 성능에 악영향을 미치지 않는 선반입 기법으로 순환 참조 선반입을 제안한다. 성능 평가를 위해서 리눅스에서 현재 사용되고 있는 미리 읽기 선반입과 순환 참조 선반입의 수행 시간을 비교했다. 다양한 참조 패턴을 가지는 트레이스들에 대한 시뮬레이션 성능 평가 결과, 순차 참조를 많이 포함하는 트레이스에 대해서는 순환참조 선반입이 리눅스의 미리 읽기 선반입과 유사한 정도의 3~5% 성능향상을 보였다. 뿐만 아니라, 미리 읽기 선반입 정책을 적용했을 때 오히려 40% 가량의 성능 악화를 초래하는 특정 트레이스에 대해서도 순환 참조 선반입을 적용할 경우 0.07%의 아주 미미한 성능 저하만을 유발하였다. 본 연구에서 제안하는 순환 참조 선반입 기법은 이득이 있을 때만 적극적인 선반입을 수행하여 시스템 성능을 향상시키며, 손해가 발생할 때는 선반입을 중지하여 시스템 성능 악화를 방지함을 실험을 통해 알 수 있다.

1. 서론

버퍼캐시에서 선반입(prefetch)은 교체정책과 함께 I/O 성능을 향상시키기 위한 중요한 기법으로 연구되고 사용되어 왔다. 선반입의 주 목적은 참조가 예상되는 블록을 미리 요청하여 계산시간과 I/O를 중첩하여 I/O로 인한 지연을 감추는 것이다.

선반입이 적절하게 수행되면 전체 실행 시간을 줄일 수 있지만, 선반입은 성능 악화의 위험을 내포하고 있다. 그 이유는 참조를 예상하고 선반입한 블록이 사용되지 않았을 때, 캐시 공간의 낭비와 추가적인 I/O가 발생하기 때문이다. 지금 널리 보급되어 있는 리눅스와 같은 운영체제에서 선반입은 대체로 순차적인 참조를 가정하고 순차성이 지속되는지를 판단하여, 강도를 조절하는 방식으로 동작하고 있다. 그러나 참조 패턴에 순차적인 특성이 존재하지 않는 경우 소극적인 선반입이라 해도 추가적인 I/O가 지속적으로 발생하여 수행 성능에 악영향을 미칠 수 있다. 순차적인 참조라고 해도 지나치게 적극적인 선반입을 수행할 경우, 순차적인 참조가 종료되었을 때 발생하는 I/O 대기 시간이 성능에 악영향을 줄 수 있다.

위에서 설명한 바와 같이, 참조 패턴은 선반입의 성능에 큰 영향을 준다. 그러나 지금까지 조사한 바로는 현재의 커널 선반입은 이러한 참조 패턴의 특성을 충분히 활용하지 못하고 있다. 참조 패턴을 탐지하고 이에 대응하여 선반입의 이익을 만들 수 있을 때에는 적절한 선반입을 수행하되 선반입이 악영향을 가져올 위험이 있을 때에는 선반입을 중지하는 정책이 필요하다. 본 논문에서는 참조 패턴을 탐지하여 선반입의 효과를 기대하기 어려운 참조 패턴에 대해서는 선반입을 수행하지 않고, 미래 참조의 예측 정확성이 높아 선반입의 효과가 높을 것으로 기대되는 참조 패턴에 대해 적극적인 선반입을 수행하는 순환 참조 선반입 기법을 제안한다.

기존의 선반입 연구 결과의 성능비교가 대부분 선반입 정책 간의 비교를 중심으로 이루어지고 있는데 참조 패턴이 선반입에 적절하지 않을 때 선반입이 악영향을 가져오는 문제를 해결하기 위해서는 선반입 결과와 선반입을 하지 않았을 때의 결과를 비교할 필요가 있다. 따라서 본 논문에서는 순환 참조 선반입의 성능을 다른 선반입 정책뿐 아니라 선반입을 하지 않았을 때와 비교하여, 순환 참조 선반입의 악영향이 나타나는지 살펴보도록 한다.

본 논문은 다음과 같이 구성되어 있다. 2장에서 관련 연구에 대해 설명한다. 3장에서는 순환 참조 선반입에 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 연구와 리눅스에서 현재 사용되고 있는 선반입 기법을 살펴보고, 이를 통해 선반입의 특성과 현재 선반입의 문제점을 살펴본다.

2.1 최적의 선반입을 위한 규칙

선반입은 확정되지 않은 I/O를 수행하므로 선반입된 블록이 사용되지 않았을 때, 오버헤드를 동반할 수 있다. 그러나, 선반입된 블록이 실제로 쓰인다고 해도 선반입이 언제나 성능에 이익을 가져온다고 보장할 수는 없다. 선반입은 선반입이 시작되는 순간 선반입할 블록이 저장될 캐시 공간을 선점해야 한다. 이 때, 유익한 블록이 쫓겨나 그 비용이 선반입을 통한 이익보다 크다면 선반입은 성능에 악영향을 가져올 수 있다. Cao 등은 이에 대한 근거를 밝히고 미래 참조를 모두 알고 있을 때, 최적의 선반입을 위해 아래의 4가지 규칙이 지켜져야 함을 밝혔다[1].

1. 선반입 - 캐시에 있지 않은 블록 중 가장 먼저 참조될 블록을 선반입 한다.
2. 교체 - 가장 나중에 참조될 블록을 쫓아낸다.
3. 위해 금지 - 블록 A가 B보다 먼저 참조될 블록이라면, B를 선반입 하기 위해 A를 쫓아내지 않는다.
4. 최초의 기회 - 최초의 기회에 선반입 하라. 최초의 기회는 이전의 반입이 막 끝났을 때나 다음에 교체될 블록이 참조된 직후에 발생할 수 있다.

위 규칙의 1번과 2번은 무엇을 선반입하고 무엇을 교체할지를 결정해 주고 3번과 4번은 언제 선반입을 수행할지에 대한 질문에 대답한다. 이 규칙은 미래 참조를 알아야 지킬 수 있는 것으로, 온라인으로 사용할 수 없지만, 선반입 설계에 있어 좋은 참고 사항이 된다.

2.2 응용 프로그램 힌트를 이용한 선반입

앞에서 설명한 것과 같이 잘못된 예측을 기반으로 선반입을 하면 성능 저하를 일으키게 된다. 따라서 예측의 정확성은 선반입에서 가장 중요한 요소 중 하나이다.

많은 경우에 응용 프로그램 개발자들은 어플리케이션의 참조 특성을 미리 알 수 있다. 이 점을 이용하여 어플리케이션으로부터 참조 패턴에 대한 힌트를 얻어 선반입을 하고자 하는 시도들이 있었다[2][3][4][5].

이 연구들은 응용 프로그램 개발자가 인터페이스를 통해 참조 패턴에 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 없으므로 오늘날 커널들은 일반적으로 스스로 판단하고 수행하는 선반입 방식을 사용한다.

2.3 리눅스 미리 읽기(read-ahead) 알고리즘

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.

이러한 방법은 순차적 접근이 일어날 경우 성능 향상을 가져올 수 있지만, 순차적 접근이 아닐 경우 버퍼의 낭비가 발생한다. 또, 순차적 접근이라고 해도 그 길이가 짧을 때, 지나친 미리 읽기로 불필요한 I/O대기 시간을 유발하게 된다. Butt 등의 연구에서는 비 순차적인 참조 패턴에 대해 리눅스의 미리 읽기 선반입을 수행했을 때 응답시간이 두 배로 증가한 사례를 보여주고 이러한 패턴에 대해서 선반입을 멈출 수 있는 방법의 필요성에 대해 제기하였다[6].

2.4 최근의 커널 선반입 연구

커널을 대상으로 한 최근의 선반입 연구는 리눅스의 미리 읽기 알고리즘과 마찬가지로 힌트를 사용하지 않는다. 그리고 참조 패턴이 여러 개의 스트림이 동시에 존재하는 상황으로 확장되었다. 그러나 스트림의 참조 패턴에 순차적 특성을 가정하고 있고, 주로 선반입 강도를 얼마나 효율적으로 정할지에 관한 연구가 중심이 되고 있다. 그리고 결과의 비교가 선반입 정책간의 비교로 한정되고 있다[7][8].

참조 패턴을 보고 그에 따라 적응하고자 하는 선반입에 대한 연구가 있었지만, 순차적인 참조와 비순차적 참조가 모두 스택 알고리즘을 따른다고 가정하고 있다. 그리고 역시 결과의 비교가 선반입 정책간의 비교에 한정되어 선반입의 악영향에 대해서는 언급하고 있지 않다[9].

3. 순환 참조 선반입

본 절에서는 본 연구에서 제안하는 순환 참조 선반입에 대해 설명한다. 순환 참조 선반입은 참조 패턴을 세가지로 분류하고, 그 세가지 패턴 중에 순환 참조에 대해 선반입을 수행한다. 3.1에서 세가지 패턴에 대해 기술하고, 3.2에서 순환 참조 선반입의 알고리즘을 기술한다.

3.1 세가지 참조 패턴

통합버퍼관리기법(UBM)[10]은 참조 패턴의 특성을 자동으로 검출하고 이 특성을 이용하는 버퍼캐시 관리 시스템이다.

이 기법은 참조 패턴을 세가지로 분류하고 자동으로 탐지한다. 각 참조 패턴을 위해 캐시를 분할하고 각 파티션의 한계 이익을 계산하여 그에 따라 파티션 크기를 동적으로 조정한다.

순환 참조 선반입은 위에서 제안된 바와 동일하게 참조 패턴을 세가지로 분류한다.

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

순환 참조 패턴은 일정한 주기를 가지고 반복해서 발생하는 특징을 가지고 있다. 패턴의 특성 자체가 주기적 반복을 의미하므로 이러한 패턴에 대해서는 과거 정보를 이용해서 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.

마지막으로 기타 참조의 경우에는 앞의 두 가지 참조 패턴에 비해 미래 참조 예측이 매우 어렵다. 그러나 2.3에서 설명한 바와 같이 기타 참조에 대해서는 선반입을 멈추는 것으로도 좋은 효과를 거둘 수 있다고 예측할 수 있다.

3.2 순환 참조 선반입

본 논문은 참조 패턴이 불리할 때에 성능에 악영향을 끼치지 않기 위하여 선반입의 이익이 기대되지 않을 때에는 중지하고 선반입의 이익이 높게 기대될 때는 적극적으로 선반입을 하는 기법을 제안하고자 한다. 앞에서 설명한 바와 같이 순환 참조 패턴에 대해서는 과거정보를 이용하여 미래에 일어날 참조의 형태를 큰 오버헤드 없이 상당히 높은 확률로 예측할 수 있다. 이 특성을 이용하여 다른 참조 패턴에 대해서는 선반입을 수행하지 않되 순환 참조에 대해서는 예측할 수 있는 정보를 이용하여 적극적인 선반입을 수행하도록 한다.

순환 참조 선반입은 순환 참조의 규칙성을 추출하고, 그 규칙성을 바탕으로 미래를 예측한다. 순환 참조 선반입에 사용되는 순환 참조의 규칙성은 크게 두 가지이다. 첫 번째는 순환 참조를 구성하는 순차 참조에 대한 정보로서, 그 시작과 끝을 기록한다. 두 번째는 그 순차 참조들 사이의 시간 간격이다. 이 간격은 완전하게 일정한 것이 아니므로 지수 평균을 사용하여 변화 추이를 반영했다.

선반입 알고리즘에서 결정해야 할 사항은 크게 언제, 무엇을(선반입 후보), 얼마나 많이 선반입 하며, 무엇을 쫓아낼 것인지의 네 가지로 요약할 수 있다.

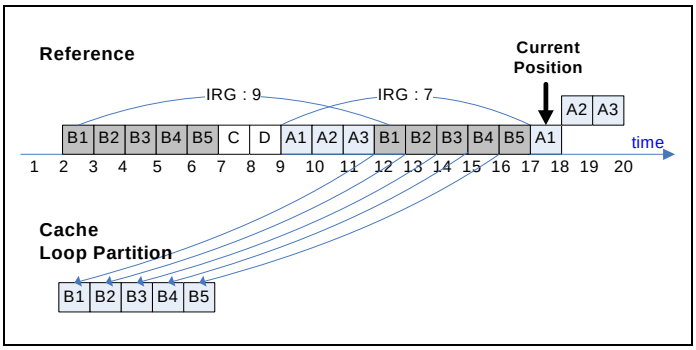
순환 참조 선반입은 이 네 가지 질문에 아래와 같이 답한다.

1) 언제: 순환 참조가 탐지되었을 때
2) 선반입 후보: 요청된 블록과 같은 순환 참조에 속하지만, 캐시에 없는 블록들
3) 얼마나 많이: 교체 대상에 적합한 블록이 존재하는 만큼
4) 교체 대상: 선반입 후보 블록보다 늦게 참조될 것으로 예상되는 블록들

3), 4)를 결정하기 위해서는 선반입 후보 블록과 교체 후보가 각각 언제 참조될 것인지를 예측할 수 있어야 한다. 선반입 후보의 다음 참조는 요청된 블록으로부터의 거리와 요청간 시간 간격을 기초로 예측한다. 교체 후보의 다음 참조는 마지막 접근 시간과 순환 참조의 주기를 기초로 예측한다. 이것은 Cao 등이 제시한 규칙의 1, 2, 3(가장 먼저 참조될 블록을 선반입하고, 가장 나중에 참조될 블록을 쫓아내고, 선반입 후보보다 먼저 참조될 교체 대상을 쫓아내지 않음)을 반영한다[1].

그림1은 순환 참조 알고리즘의 동작과 참조 시간 예측을 그림으로 설명한다. 그림의 위쪽은 시간의 흐름에 따른 참조, 아래쪽은 5개의 블록을 담을 수 있는 캐시이다. 이것은 전체 캐시가 아니라, 캐시에서 순환 참조를 보관하는 파티션만을 표현한 것이다. 이 예제에는 두 개의 순환 참조가 나타난다. 하나는 블록 A1, A2, A3으로 구성되고 참조간 간격(IRG: Inter Reference Gap)이 7인 순환 참조 A이고 다른 하나는 블록 B1, B2, B3, B4, B5로 구성되고 참조간 간격이 9인 순환 참조 B이다. 예제에서 모든 블록들은 1단위시간 간격으로 참조된다. 이것은 예제를 간단하게 도식화 하기 위한 것으로, 블록간 간격이 달라져도 알고리즘의 수행은 동일하다.

시간 17에서 블록 A1에 대한 참조 요청이 발생한다. 이 때 순환 참조가 저장되는 캐시 파티션(Cache Loop Partition)에는 블록 B1, B2, B3, B4, B5가 존재한다. 알고리즘은 블록 A1을 순환 참조 A에 속하는 것으로 판단하고 선반입을 시도한다. 알고리즘은 순환 참조 A를 구성하는 순차 참조가 A1, A2, A3 세 개의 블록으로 이루어지는 것을 알 수 있다. A1, A2, A3 중 A1 이후에 존재하는 A2, A3이 캐시에 존재하는지 살펴본다. A2, A3이 모두 캐시에 존재하지 않으므로 선반입 후보가 된다. A2, A3은 현재 참조 블록인 A1로부터 각각 1, 2만큼 떨어져 있고, 각 블록 참조 사이의 시간간격은 1이므로, 각각 시간 18, 19에 요청될 것이라고 예상할 수 있다.



[그림 1] 순환 참조 선반입의 동작

이제 캐시에 있는 블록 중에 A2, A3이 요청될 시간인18, 19보다 나중에 요청될 블록이 있는지를 살펴본다. 알고리즘은 IRG가 가장 길고 가장 마지막에 참조된 블록부터 비교 대상으로 삼는다. 가장 마지막에 참조된 블록 B5의 다음 참조 시간은 B5가 마지막으로 참조된 시간에 IRG를 더해서 예측한다. 따라서 B5의 다음 예상 참조 시간은 25가 된다. 이것은 블록 A2의 예상 참조 시간인 18보다 크므로 비교를 계속해서 진행한다. 블록 B5보다 바로 전에 접근되었던 블록 B4의 예상 참조 시간은 24가 된다. 이것 또한 블록 A3의 예상 참조 시간인 19보다 크므로 블록 A2, A3은 모두 선반입이 된다.

다만, 만일 비교 대상으로 선택된 블록이 이미 선반입을 위해 반입된 블록이고 아직 참조되지 않았다면, 이것을 쫓아내고 다른 블록을 선반입하는 것은 악영향을 유발할 수 있으므로 비교를 중단한다.

4. 성능 평가

본 절에서는 순환 참조 선반입의 성능을 리눅스의 미리 읽기 선반입과 비교하여 평가한다. 실험환경은 시뮬레이터를 사용하였다. 같은 환경에서 같은 트레이스를 사용하여 순환 참조 선반입을 수행한 경우와 선반입을 하지 않은 경우, 리눅스의 미리 읽기 선반입을 수행한 경우를 비교하였다. 일반적으로 교체 정책의 성능 지표로는 적중률을 사용하지만 선반입에서 성능 지표는 수행시간이 가장 적절함이 밝혀졌기 때문에 수행시간을 기준으로 성능을 비교하였다[6].

4.1 실험 환경

Butt 등은 리눅스 미리 읽기 선반입과 I/O를 충실히 구현하고 실행 시간을 측정할 수 있는 시뮬레이터 Accusim을 제작하고 다양한 참조 특성을 갖는 트레이스들을 제공했다[6]. Accusim은 널리 검증된 디스크 시뮬레이터인 Disksim을 기반으로 리눅스의 I/O 구조와 여러 캐시 정책을 구현한 시뮬레이터이다. 실험 환경으로 Accusim을 선택하고 트레이스 역시 이곳에서 제공된 트레이스를 사용한다.

기본적인 관리 시스템으로는 3.1절에서 설명한 통합버퍼관리기법(UBM)을 사용한다. Accusim에 UBM을 구현하고, 순환 참조 선반입을 구현하였다. 리눅스의 미리 읽기 선반입은 Accusim에 구현된 것을 사용하였다.

4.2 실험 결과

실험에서 전체 캐시 관리시스템으로는 UBM을 사용하고 기타 참조 파티션의 관리에는 LRU를 사용하였다. 선반입을 수행하지 않은 경우와 리눅스 미리 읽기 선반입이 수행된 경우, 순환 참조 선반입이 수행된 경우의 총 수행 시간을 각각 비교하였다.

사용된 트레이스는 glimpse, gcc, viewperf, tpc-h, multi3으로, 앞에서 밝힌 것처럼 [6]에서 제공된 것이다.

우선 glimpse는 인덱싱과 검색을 하는 시스템으로 550MB의 /usr 디렉토리 밑의 텍스트 파일들에서 텍스트 스트링을 검색하였고, gcc는 리눅스 커널 2.4.20을 컴파일 하였다. Cscope은 소스 코드 분석을 수행하는 어플리케이션으로 리눅스 커널 2.4.20을 분석하는데 사용했다. Viewperf는 그래픽 웍스테이션의 성능을 측정하는 데 사용되는 SPEC 벤치마크이다.

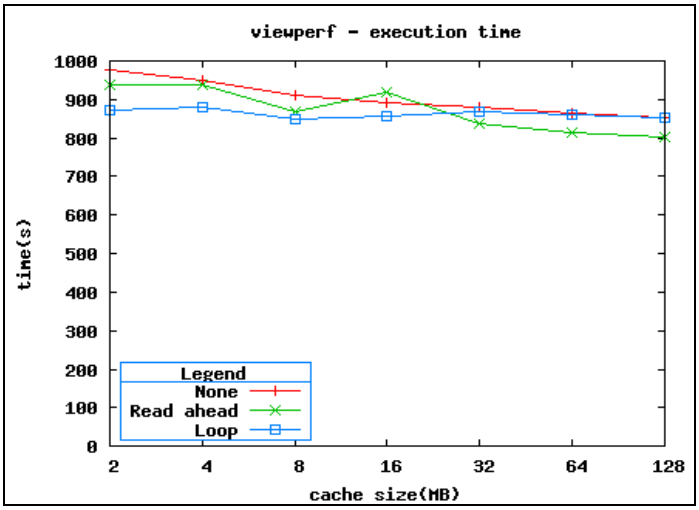
이전의 연구를 통해 트레이스들의 참조 특성과 선반입결과를 간단히 살펴보면, glimpse, gcc, viewperf는 27~99%의 순차적 특성을 가지는 트레이스이다. 다른 교체정책들과 리눅스 미리 읽기 선반입을 수행했을 때, 적중률에서는 대체로 향상을 보였으나, 수행시간 측면에서는 기대한 만큼의 이익이 반영되지 못하였다. 반면 tpc-h은 순차적 특성이 거의 존재하지 않고, multi3은 glimpse와 tpc-h가 함께 수행된 트레이스로, 이들 트레이스에 선반입을 수행하면 전체 수행시간이 상당하게 증가되는 것이 확인되었다[6].

실험은 캐시 크기를 2M에서 128M까지 2배씩 증가하여 실험하였다. 각 캐시 크기 별 수행 시간의 평균치를 [표 1]에 각 트레이스 별로 비교하여 요약하고 [그림 2], [그림 3], [그림 4], [그림 5], [그림 6]에서 각각 gcc, glimpse, tpc-h 트레이스에 대해 캐시 크기 변화에 따른 실행 시간 변화를 그래프로 표현했다.

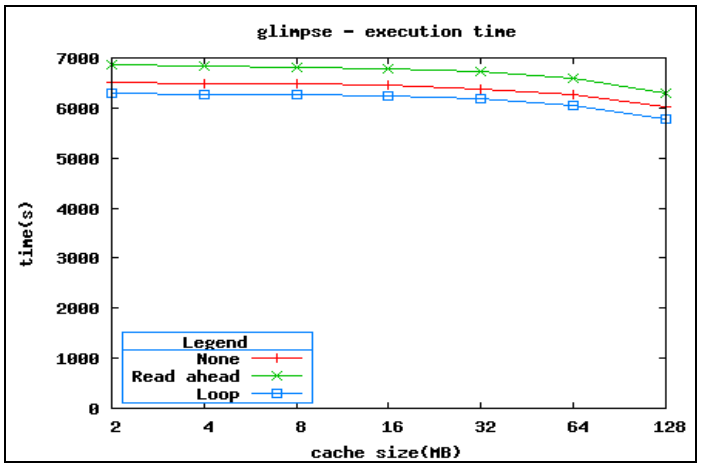
먼저 순차적인 트레이스들에 대한 결과를 살펴보면, glimpse의 경우 미리 읽기 선반입이 평균 4.86%의 수행시간 증가를 가져오는 반면 순환 참조 선반입은 평균 3.53%의 수행시간 감소를 가져온다. [그림 2]를 통해 캐시 크기 별로 수행시간이 변화하는 모습을 살펴보면 그래프는 로그스케일로 표현되어 있지만 캐시 크기에 따라 수행시간이 선형적으로 감소하고 있음을 확인할 수 있다. gcc의 경우에는 미리 읽기 선반입이 10.77%의 수행시간 감소를 가져오는 반면 순환 참조 선반입은 0.01%의 감소밖에 가져오지 못하는데, 이것은 gcc를 구성하는 순차 참조들이 대체로 짧고 특히 순환 참조를 이루는 순차 참조의 길이가 매우 짧아 선반입의 여지가 크지 않았기 때문으로 보인다. [그림 3]을 보면 순환 참조 선반입과 선반입을 하지 않은 그래프가 거의 겹쳐서 나타남을 확인할 수 있다. viewperf의 경우 미리 읽기 선반입이 평균 3.34%의 수행시간 감소를 가져왔고, 순환 참조 선반입은 평균 4.54%의 수행시간 감소를 가져왔다. [그림 4]를 보면 미리 읽기 선반입이 캐시 크기에 따라 불규칙한 결과를 보이는 데 반해, 순환 참조 선반입은 선반입의 여지가 있는 동안 선형적으로 수행 시간을 감소시키고 있음을 확인할 수 있다. 이처럼 순차적인 트레이스들에 대한 수행시간 감소 측면에서 gcc를 제외하면 미리 읽기 선반입에 비해 순환 참조 선반입이 더 나은 결과를 보였다. 순차적 특성이 적은 tpc-h의 경우 미리 읽기 선반입은 38%가 넘는 수행시간 증가를 가져오지만, 순환 참조 선반입은 0.07%의 증가만을 가져온다. [그림 5]를 통해서 순환 참조 선반입이 선반입을 거의 수행하지 않고 있음을 확인할 수 있다. Multi3의 경우에는 미리 읽기 선반입이 42.09%의 수행 시간 증가를 가져올 때, 순환 참조 선반입은 1.32%의 수행시간 감소를 가져온다.

[표 1] 수행시간 비교표

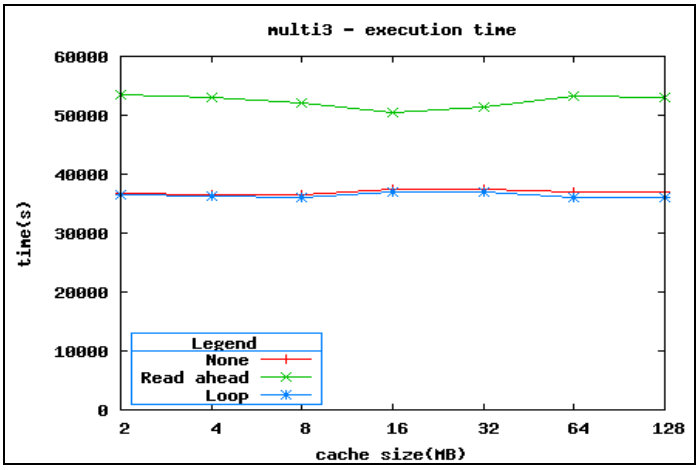
트레이스	순차 정도	선반입 없음(초)	미리 읽기 선반입(초)	순환참조 선반입(초)
glimpse	74%	6375.89	6701.92 (+4.86%)	6158.35 (-3.53%)
gcc	27%	267.51	238.69 (-10.77%)	267.48 (-0.01%)
viewperf	99%	903.95	873.80 (-3.34%)	862.89 (-4.54%)
tpc-h	3%	21826.74	30338.78 (+38.99%)	21841.82 (+0.07%)
Multi3	16%	36907.67	52440.41 (+42.09%)	36419.57 (-1.32%)



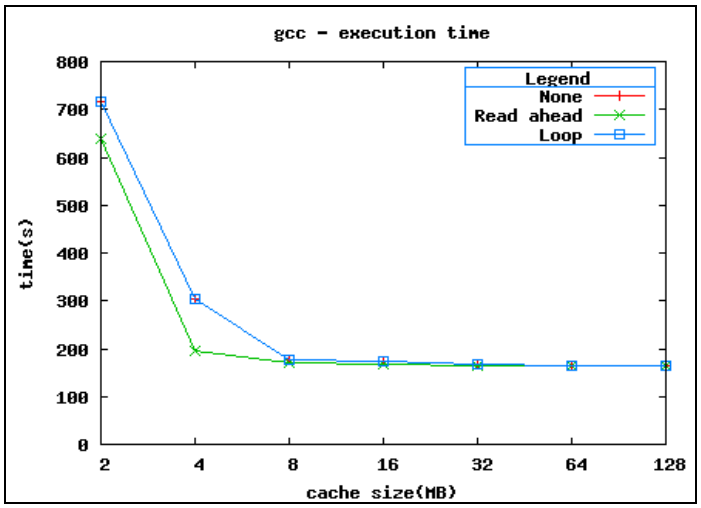
[그림 4] viewperf의 캐시 크기 변화에 따른 수행시간



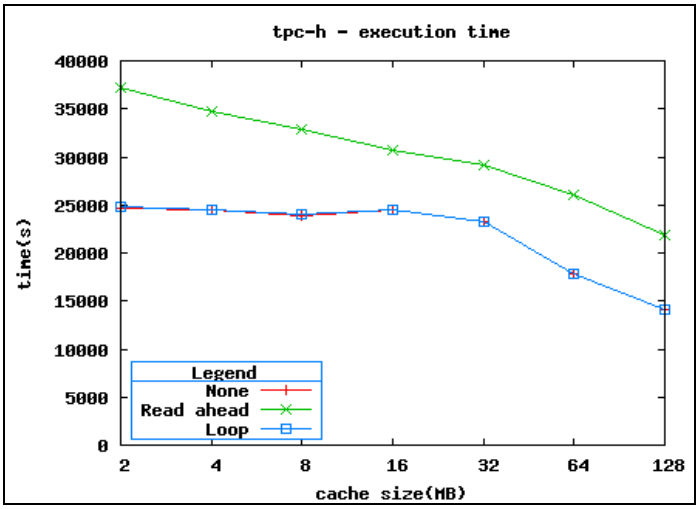
[그림 2] glimpse의 캐시 크기 변화에 따른 수행시간 변화



[그림 5] tpc-h의 캐시 크기 변화에 따른 수행시간 변화



[그림 3] gcc의 캐시 크기 변화에 따른 수행시간 변화



[그림 6] multi3의 캐시 크기 변화에 따른 수행 시간

이 실험 결과는 순환 참조 선반입이 미리 읽기 선반입에 대비해 볼 때, 순차적 특성의 참조 패턴에 대해서 효과적인 선반입을 수행할 수 있음을 보여준다. 또한 선반입을 수행하는 것이 손해를 가져올 위험이 있을 때 순환 참조 선반입이 효과적으로 선반입을 중지할 수 있음을 확인할 수 있다.

5. 결론 및 향후 과제

순차 참조 형태에서 성능을 발휘하도록 설계된 현재의 커널 선반입은 참조 패턴이 순차성을 잃을 때 성능을 저하시킬 위험이 크다. 이를 극복하기 위해 참조 특성을 탐지하고 그에 대해 적응할 수 있는 선반입 기법이 필요하다. 그러한 선반입 기법으로 순환 참조 선반입을 개발하였다.

순환 참조 선반입은 참조 패턴을 순차 참조, 순환 참조, 기타 참조의 세가지로 분류하고 온라인으로 탐지한다. 이 세가지 참조 패턴 중 순환 참조패턴은 과거 정보를 기반으로 미래 참조를 구체적으로 예측할 수 있다. 예측한 정보를 이용해서, 순환 참조에 대해서 선반입을 수행한다. 순환 참조의 길이를 이용하여 선반입 후보를 결정하고, 각 블록 요청 사이의 간격을 이용하여 후보 블록이 언제 요청될 지를 예측했다. 그리고 캐시에 존재하는 순환 참조 블록들에 대해 최종 접근 시간과 참조간 간격을 이용해 다음 요청 시간을 예측하고, 선반입 후보의 예상 요청 시간과 비교하여, 먼저 요청될 블록을 반입했다.

순환 참조 선반입의 성능을 평가하기 위해 시뮬레이션을 통해 리눅스의 미리 읽기 선반입과 비교하였다. 그 결과 순환 참조 선반입이 선반입의 효과가 발휘될 수 있을 때 선반입의 효과를 잘 발휘할 뿐 아니라, 선반입이 악영향을 낼 위험이 있을 때 선반입을 효율적으로 중지하여 성능 악화를 방지할 수 있음을 보여주었다.

본 연구의 향후 과제로는 순차 참조를 위한 선반입 기법이 개발되어야 하고 순환 참조에 대해서도 최적화의 여지가 있다고 본다. 또한 선반입이 각 블록의 가치에 미치는 영향을 고려하여 각 캐시 파티션 크기의 밸런스를 조정해야 한다.

감사의 글

본 연구는 한국과학재단 특정기초연구(R01-2004-000-10188-0) 지원으로 수행되었음.

[참고문헌]

[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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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,

[8] 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.

[9] 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.

[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.