명령어 선인출 예측 정확도의 한계에 관한 연구
(A Study on the Prediction Accuracy Bounds of Instruction Prefetching)

김 성 백 [†] 민 상 렬 [††] 김 종 상 [††]
(Seong Baeg Kim) (Sang Lyul Min)(Chong Sang Kim)

요 약 선인출은 프로세서에 의해 사용될 데이타를 예측하여 미리 프로세서 근처에가져오므로써 메모리 지연 시간을 줄이는 기법이다. 선인출의 효율성은 미래에 사용될 데이타를 얼마나 정확하게 예측하는가(선인출 예측 정확도)에 따라 결정된다. 기존의 명령어 선인출에 관한 연구들은 특정 선인출 기법의 제안 및 성능 평가에 그치고 있어서 명령어 선인출의 특성이 체계적으로 분석 정리되지 못하고 있다. 이에 본 논문에서는 명령어 선인출의 예측 정확도에 대해서 이론적으로 분석하여 이의 한계를 알아보고자 한다. 그 방안으로 명령어 선인출 상한 모델이라는 이론적인 선인출 모델을 제안하고 이 모델을 기반으로 명령어 선인출에 대해 체계화된 분석을 한다. 특히 이러한 연구 결과로써 궁극적으로 시스템 성능을 효과적으로 향상시킬 수 있는 효율적인 명령어 선인출을 가능하게 하는 데 그 목적이 있으므로 주로 명령어 선인출 효율성 측면에서 분석을 시도하였다. 이러한 선인출 모델을 이용하여 본 논문에서는 SPEC 벤치마크 프로그램들의 명령어 선인출 예측 정확도의 한계를 이론적으로 분석하였다. 그 결과로 캐쉬가 없는 경우에는 선인출 정확도가 매우 높게 나타남을 보였다. 반면에 캐쉬가 있을 경우에는 캐쉬 크기가 커짐에 따라 선인출의 정확도가 급격히 떨어짐을 관찰하였다. 예를 들어 spice의 경우 블록크기가 16바이트이고 직접사상 캐쉬에서 캐쉬 크기가 2K 바이트와 16K 바이트일 때 이론적으로 가능한 최대 선인출 정확도가 각각 53%, 39%로 크게 떨어지는 것을 관찰하였다. 캐쉬의 크기가 커질수록 선인출로 메모리 지연 시간을 줄일 수 있는 명령어 참조의 많은 부분을 캐쉬가 처리하게 되고 또한 캐쉬에서 접근 실패된 명령어 참조는 그 참조 행태가 불규칙하여 예측이 어렵기 때문에 일정 크기 이상의 명령어 캐쉬를 사용하는 경우 명령어 선인출을 사용하는 것은 전체 시스템 성능의 향상에 큰 도움이 되지 않음을 이론적으로 규명하였다.

Abstract Prefetching aims at reducing memory latency by fetching, in advance, data that are likely to be requested by the processor in a near future. The effectiveness of prefetching is determined by how accurate the prediction on the needed instructions and data is. Most previous studies on prefetching were limited to proposing a particular prefetch scheme and its performance evaluation, paying little attention to theoretical aspects of prefetching. This paper focuses on the theoretical aspects of instruction prefetching. For this purpose, we propose a clairvoyant prefetch model that makes use of perfect history information. Based on this theoretical model, we analyzed upper limits on the prefetch prediction accuracies of the SPEC benchmarks. The results show that the prefetch prediction accuracy is very high when there is no cache. However, as the size of the instruction cache increases, the prefetch prediction accuracy drops drastically. For example, in the case of the spice benchmark, the prefetch prediction accuracy drops from 53% to 39% when the cache size increases from 2Kbyte to 16Kbyte (assuming 16byte block size). These results indicate that as the cache size increases, most localities are captured by the cache and that instruction prefetching based on the information extracted from the references that missed in the cache suffers from prediction inaccuracies

† 종신회원 : 제주대학교 컴퓨터교육과 교수
sbkim@educom.cheju.ac.kr
†† 종신회원 : 서울대학교 컴퓨터공학부 교수
symin@dandelion.snu.ac.kr
cskim@archi.snu.ac.kr

논문접수 : 1999년 8월 13일
심사완료 : 2000년 6월 22일

1. 서 론

반도체 기술의 발전으로 인해 프로세서의 속도는 급속히 향상되고 있다. 그러나 메모리는 집적도 면에서는 큰 진전이 있어 왔지만 접근 시간은 크게 개선되지 않고 있다. 이에 따라 프로세서와 메모리 사이에 속도 차이가 점차로 커지고 있다[6]. 이들 간의 속도 차이에서 오는 성능 저하를 줄이기 위한 방법으로 캐쉬와 선인출이 사용된다. 캐쉬는 프로세서로부터 고속으로 접근될 수 있는 메모리로서 자주 참조되는 메모리 블록을 저장하여 이에 대한 접근 시간을 줄이는 역할을 한다. 반면에 선인출은 가까운 미래에 참조될 가능성이 높은 데이타 블록을 예측하여 이를 미리 프로세서에 인접한 메모리로 가져와서 접근 시간을 줄인다.

캐쉬에 대한 연구는 그동안 광범위하게 다각도로 이루어져 왔다[12, 13]. 특히 이들 연구중에는 메모리 접근의 행태 및 캐쉬의 성능을 이론적으로 분석하려는 시도가 있었다[16, 17]. 그러나 선인출은 지금까지 캐쉬만큼 활발한 연구가 이루어지지 못하였고 특히 선인출에 대한 이론적인 연구는 거의 없었다. 최근까지 진행된 선인출에 관한 연구들은 대부분 특정 선인출 기법의 제안 및 제안한 기법의 성능 평가에 그치고 있다. 선인출이 널리 사용되기 위해서는 시스템 성능을 향상시킬 수 있는 효율적인 선인출 기법의 고안과 이를 실용화하려는 연구가 필요하다. 그러나 이에 못지않게 선인출에 대한 체계적인 이해와 정리를 위한 이론적인 연구가 필요하다.

이에 본 논문에서는 명령어 선인출 자체의 특성을 객관적으로 파악하고 이를 체계화하여 궁극적으로 효율적인 명령어 선인출을 가능하게 하고자 한다. 부연하면 명령어 선인출의 큰 특징 중에 하나는 미래를 예측하는 것이라고 볼 수 있다. 이처럼 미래를 예측해야 하는 선인출의 속성 때문에 명령어와 데이타를 포함한 모든 선인출에 있어서 매우 중요한 것이 효율성이라고 볼 수 있다. 왜냐하면 미래를 예측하여 미리 어떤 일을 하는 것은 그 예측이 잘못되는 경우 그에 따른 문제를 수반할 것이기 때문이다. 선인출의 경우에도 예외는 아니어서, 단순히 시스템 성능 향상에만 중점을 두어 프로세서에게 꼭 필요한 명령어인지를 신중하게 선별하지 않고 선인출한다면 그로 인해 야기되는 문제들 때문에 오히려 시스템 성능 저하를 초래할 것이다. 따라서 본 장에서는 명령어 선인출의 효율성을 나타내는 선인출 예측 정확도 측면에서 명령어 선인출의 특성들을 분석하고 그 분석 결과를 바탕으로 명령어 선인출의 특성들을 체계적으로 알아보고자 한다.

이를 가능하게 하기 위해 선인출에 의한 시스템 성능을 최대화할 수 있는 선인출 모델을 제안한다. 이 모델을 본 논문에서는 미래를 미리 내다봄으로써 시스템 성능을 최대화할 수 있으며 또한 선인출 특성의 한계를 보여줄 수 있다는 의미에서 선인출 상한 모델(clairvoyant prefetch model)이라고 부르기로 한다. 명령어 선인출의 궁극적인 목적은 가장 효율적으로 메모리 접근 지연 시간을 줄여 시스템 성능을 최대화하는 데 있다. 따라서 제안한 선인출 상한 모델이 선인출 본래의 목적과 일치하므로 이 모델을 바탕으로 명령어 선인출의 특성에 대해 분석한 결과들은 충분히 객관성과 타당성을 갖는다. 특히 제안한 선인출 상한 모델이 이론적으로 시스템 성능을 최대화하는 모델이므로 이 모델을 이용하여 얻어진 여러가지 결과들은 가능한 어떤 선인출 기법에 의해 얻을 수 있는 여러 결과들의 한계(limit)를 나타내는 의미를 포함하고 있다.

본 논문의 2장에서는 관련 연구를 설명하고 3장에서는 이론적인 분석을 위한 방법으로 선인출 상한 모델을 제시한다. 4장에서는 3장에서 제시한 선인출 모델을 이용하여 SPEC 벤치마크 프로그램들에 대해서 이론적인 분석 결과를 제시하고 또한 선인출 정확도에 영향을 끼치는 선인출 시기, 캐쉬 등과의 상호 관계를 설명한다. 마지막으로 5장에서는 결론을 도출한다.

2. 관련 연구

선인출은 명령어 선인출[14, 7, 8]과 데이타 선인출[2, 9, 3, 11, 10, 5]로 나눌 수 있다. 명령어는 참조 특성이 데이타에 비해 보다 규칙적이므로 예측이 용이하여 매우 효율적인 선인출이 가능하다. 그러나, 명령어 선인출에 대한 지금까지의 연구는
특정 선인출 기법의 제안 및 성능 평가에 그쳐 선인출에 대한 이론적인 연구는 거의 진행되지 않고 있다. 기존의 명령어 선인출 기법으로는 명령어의 순차적 특성을 이용한 선인출[14, 17]과 과거의 실행정보를 이용한 선인출[8]로 크게 나누어 볼 수 있다.

[14]에서는 순차적 블록을 선인출하기 위한 세 가지 기법인 Always prefetch, Prefetch on misses, Tagged prefetch 기법을 제시하고 그들의 성능을 분석하였다. Always prefetch 기법에서는 매 참조마다 다음 순차 블록을 선인출한다. Prefetch on misses 기법은 Always prefetch 기법에서 자주 발생하는 불필요한 선인출을 줄이기 위해 캐쉬 접근 실패가 발생했을 때만 다음 순차 블록을 선인출한다. Tagged prefetch 기법은

각 메모리 블록마다 하나의 태그 비트를 두어, 캐쉬 접근 실패가 나는 경우와 더불어 선인출된 블록에 의해 캐쉬 접근 적중이 되는 경우에도 다음 순차 블록을 선인출한다. 이 기법들은 바로 다음 순차 블록만을 선인출하기 때문에 메모리 지연이 큰 경우에는 현재 실행중인 블록이 완료되기 전까지 다음 블록을 선인출 할 수 없어서 큰 이득을 얻을 수 없다. 이러한 문제를 해결하기 위해 [7]에서는 *stream buffer*라는 기법을 제안하였다. 이 기법에서는 캐쉬 접근실패가 일어날 경우 그 블록으로부터 여러 개의 순차적 블록들을 선인출하여 메모리 지연이 큰 경우에도 충분한 이득을 얻을 수 있게 하였다.

이상에서 설명한 선인출 기법에서는 선인출이 순차 블록에 국한되어 있어 분기 명령이 수행이 되는 경우 성능 저하를 가져온다. 이를 해결하기 위해 [8]에서는 각 명령어 블록마다 할당된 쓰레드(포인터)에 실행 흐름 정보를 저장하고 이를 참조하여 선인출에 이용하는 쓰레드 선인출 기법을 제시하였다. 이 기법에서는 실행흐름이 매번 같은 경로를 따르는 경향을 이용하여, 과거의 수행흐름 정보를 저장하여
선인출의 정확도를 향상시킨다. 쓰레드는 다음 순차 블록을 가리키도록 초기화되며, 매 실행시 실제로 다음에 실행된 블록을 가리키도록 수정된다. 또한 각 블록이 하나 이상의 쓰레드를 가질 수 있으며 이 경우 이들 쓰레드간에 우선순위를 두어 다음에 실행될 가능성이 높은 순서대로 선인출한다. 제안한 기법에서 우선순위는 최근에 사용된 쓰레드일수록 더 높은 값을 가진다. 이 기법은 과거의 수행 정보를 사용한다는 점에서 본 연구에서 제시할 선인출 상한 모델과 관련이 깊다.
그러나, 이전 연구와 마찬가지로 이 연구에서도 단순히 새로운 선인출 기법을 제시하고 이의 성능을 평가하는데 그치고 있어 선인출 특성에 대한 이론적인 연구는 이루어지지 못하였다.

3. 이론적인 분석 모델

3.1 기본 원리

순차 선인출 기법이나 쓰레드 선인출 기법은 나름대로 효율적인 선인출을 시도하여 시스템 성능을 향상시키는 데 그 목적이 있다. 이에 이들 명령어 선인출 기법은 효율적으로 선인출하기 위해 프로그램의 실행 동안 나타나는 여러 가지 특성들을 최대한 활용한다. 그러므로 이들 선인출 기법은 시스템 성능을 향상시키려는 실제로 가능한 특정 선인출 기법으로 볼 수 있다. 그러나 본 논문에서는 특정 선인출 기법에 관한 기존의 연구와는 다르게 전반적인 선인출 특성을 체계적으로 파악해 보고자 이론적 분석을 위한 선인출 상한 모델을 제안한다.

제안하는 선인출 상한 모델은 이론적으로 시스템 성능을 최대화하기 위해 미래를 미리 알고 선인출한다. 즉 프로그램의 어떤 실행 특성을 기반으로 선인출하는 것이 아니라 주어진 프로그램의 미래 실행 행태(behavior)를 미리 내다보면서 선인출한다. 그러나 실제로 미래에 일어날 일을 미리 안다는 것은 불가능하다. 그러므로 본 논문에서는 미래를 알 수 있는 효과를 가져오기 위해서 사후 분석(postmortem analysis)이라는 기법을 사용한다. 이 기법에서는 먼저 프로그램을 처음부터 끝까지 실행하며 나타나는 실행 행태를 효율적으로 저장한다. 그런 다음에 프로그램을 처음 실행시와 동일한 조건하에서 다시 반복 실행하면서 처음 실행 동안 수집된 실행 정보를 활용하여 선인출한다. 이렇게 함으로써 실제 가능한 방법은 아니지만 미래를 미리 알고 선인출하는 효과를 가져올 수 있기 때문에 그 어떤 특정 선인출 기법보다도 선인출에 의한 시스템 성능을 최대화할 수 있다. 따라서 선인출 상한 모델에 의해 얻어진 분석 결과들은 명령어 선인출의 한계 측면을 볼 수 있도록 해준다. 또한 특정 선인출 기법에 의존적이지 않는 결과를 제시하게 됨으로써 명령어 선인출의 특성을 객관적으로 파악할 수 있게 한다.

3.2 블록 참조 행태의 추출

먼저 선인출 상한 모델이 효율적으로 활용할 수 있도록 전체 프로그램 실행 동안 각 블록별로 참조 행태(reference behavior)를 추출하고 저장하는 방법에 관해 알아본다. 본 논문에서는 블록 단위의 선인출을 기본으로 하기 때문에 프로그램의 실행에 대해 블록 단위의 참조 행태만을 고려한다. 그리고 프로그램의 실행 행태에 대한 완전한(프로그램 실행의 처음부터 끝까지를 말함) 정보를 만드는 일련의 과정을 직관적으로 이해할 수 있도록 간단한 예를 이용하여 전반적으로 설명한다.

그림 1은 일반적인 프로그램 실행시 나타나는 전형적인 참조 행태를 보여주고 있다. 그림은 각 블록별로 그

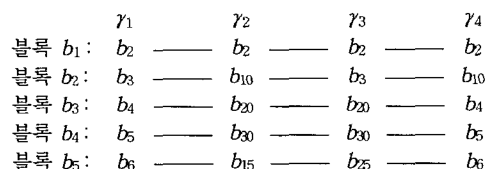


그림 1 블록의 참조 형태

블록이 참조시 그 블록 다음에 프로세서에 의해 참조되는 블록을 순서대로 나타낸 것이다. 예를 들면, 블록 b_1은 분기가 없이 순차 블록 b_2만이 참조되는 경우를 나타낸다. 부연하면 블록 b_1은 첫번째 참조 r_1시에 다음에 블록 b_2가 참조되는 것을 시작으로 참조 r_2, r_3, r_4시에도 계속해서 다음에 블록 b_2가 참조된다.

반면에 블록 b_2, b_3, b_4, b_5는 매 참조시마다 다음 참조 블록으로 순차 블록만 참조되는 것이 아니라 분기에 의한 목적지 블록도 참조된다. 구체적으로 블록 b_2의 경우에는 첫번째 참조 r_1시에는 순차 블록 b_3, 두번째 참조 r_2시에는 분기에 의해 목적지 블록 b_{10}, 세번째 참조 r_3시에는 다시 순차 블록 b_3, 네번째 참조 r_4시에는 다시 분기에 의한 목적지 블록 b_{10}이 각각 다음 참조 블록으로 나타나 있다. 이어서 블록 b_3의 경우에는 순차 블록과 더불어 목적지 블록이 다음에 참조된다는 점에서 블록 b_2의 경우와 같지만 첫번째 참조 r_1과 네번째 참조 r_4시에 순차 블록 b_4참조되고 두번째 참조 r_2와 세 번째 참조 r_3시에 분기에 의한 목적지 블록 b_{20}이 참조되는 것이 다르다. 다음으로 블록 b_4의 경우에는 블록 b_3의 경우와 동일한 참조 행태를 보인다. 즉 첫번째 참조 r_1과 네번째 참조 r_4시에 순차 블록 b_5, 두번째 참조 r_2와 세번째 참조 r_3시에 분기에 의한 목적지 블록 b_{30}이 참조된다. 마지막으로 블록 b_5의 경우에는 첫번째 참조 r_1과 네번째 참조 r_4시에 순차 블록 b_6가 참조되고 두번째 참조 r_2시에 분기에 의한 목적지 블록 b_{15}, 세번째 참조 r_3시에 분기에 의한 또다른 목적지 블록 b_{25}가 참조된다.

이들 참조 행태를 전체적으로 보면, 블록 b_1은 분기 명령어가 포함되지 않아 항상 순차 블록만이 다음에 참조되는 경우를 나타내며, 블록 b_2, b_3, b_4는 분기 명령어가 포함되어 분기 조건에 따라 순차 블록 또는 목적지 블록이 다음에 참조되는 경우를 나타냄을 알 수 있다. 특히 블록 b_5는 간접 분기 명령어 또는 한 개 이상의 분기 명령어가 블록 내에 존재하여 상이한 여러 블록이 다음에 참조된 경우를 나타낸다.

부가적으로 그림은 프로그램 실행에 따른 각 블록의 참조 행태를 모두 보여 주는 것이 아니라 매우 일부분만을 보여주고 있다는 점이다. 실제로는 프로그램마다 다르겠지만 각 블록의 참조 횟수가 적은 경우에는 한번에서 많은 경우에는 수만번 이상 될 수 있다. 물론 프로그램의 실행 행태에 따라 어떤 블록은 전혀 참조되지 않을 수도 있다. 그림에서는 각 블록의 단지 첫번째 참조부터 네번째 참조까지만을 보여주고 있다. 아울러 참조되는 상이한 블록 수가 매우 많고 이들 상이한 각 블록의 참조 행태가 각기 다르게 나타날 수 있다. 그러나 그림은 전형적인 블록 참조 행태를 반영하는 대표적인 5개의 블록만을 고려하였다.

3.3 블록 참조 행태의 참조 패턴화

제안하는 선인출 상한 모델은 먼저 프로그램의 실행에서 나타나는 각 블록의 참조 행태를 참조 패턴(reference pattern)으로 바꾼다. 여기에서 참조 패턴이란 각 블록별로 그 블록 다음에 차례대로 참조되는 블록을 각각 문자화한 것을 말한다. 부연하면 각 블록의 다음 참조 블록 중에서 가장 처음에 참조되는 블록을 문자 a로 표시하고 상이한 블록이 나올 때마다 새로운 문자로 표시함으로써 결과적으로 일련의 문자들로 이루어진 각 블록의 참조 패턴을 만들 수 있다.

그림 1의 예에 대해 참조 패턴을 만들면 다음과 같다. 블록 b_1은 다음 참조 블록으로 블록 b_2만 있으므로 문자 a만으로 이루어진 참조 패턴을 생성하게 된다. 즉 처음에 나타나는 블록 b_2를 문자 a로 표시한 후 이어서 참조되는 블록들이 이전과 항상 동일한 블록 b_2이므로 계속하여 문자 a로 표시하게 된다. 반면에 블록 b_2의 경우에는 다음 참조 블록으로 블록 b_3과 블록 b_{10}이 있으므로, 가장 처음에 참조한 b_3을 문자 a로, 다음에 참조한 b_{10}을 문자 b로 표시하게 된다. 마찬가지로 블록 b_3과 블록 b_4의 경우에는 순차 블록인 블록 b_4와 블록 b_5를 각각 문자 a로, 분기에 의한 목적지 블록인 블록 b_{20}과 블록 b_{30}을 각각 문자 b로 표시하게 된다. 블록 b_5의 경우에는 서로 다른 세 블록을 다음 참조 블록으로 가지고 있으므로, 순차 블록 b_6을 문자 a로, 두번째 나타나는 블록 b_{15}를 문자 b로, 세번째 나타나는 블록 b_{25}를 문자 c로 표시하게 된다.

그림 2는 이와 같은 방법으로 생성된 각 블록의 참조 패턴을 보여준다. 그림에서 보는 것처럼 처음 참조하는 블록에 대해서는 모든 블록에서 동일하게 문자 a로 나타냄을 알 수 있다. 이와 같이 블록 정보를 제거하고 패턴으로 바꾸어도 개개의 블록 정보를 알 수 없다는 것을 제외하면 블록 참조 행태에서와 동등한 정보를 얻을 수 있다. 예를 들면, 참조 패턴을 나타내는 그림 2에서 블록 b_1은 어떤 블록이 다음에 참조되는지는 알 수 없지만 일관되게 한 블록만이 계속하여 참조됨을 알 수 있다. 반면에 블록 b_2는 서로 다른 두 블록이 교대로 참조되고 있음을 알 수 있다. 또한 블록 b_3과 블록 b_4는 서로 다른 두 블록이 똑같은 순서로 참조됨을 알 수 있다. 이와 같이 블록 b_3과 블록 b_4의 참조 특성이 동일하다는 것은 블록 참조 행태를 참조 패턴으로 변환함으로써 직관적으로 이해할 수 있다. 한편 블록 b_5의 경우도 그림에서 서로 다른 세 블록들이 참조되고 있음을 쉽게

파악할 수 있다. 이렇게 참조 패턴으로 변환함에 따라 참조 특성이 동일한 블록과 동일하지 않는 블록을 직관적으로 구별할 수 있게 된다.

블록 b_1 : a —— a —— a —— a
블록 b_2 : a —— b —— a —— b
블록 b_3 : a —— b —— b —— a
블록 b_4 : a —— b —— b —— a
블록 b_5 : a —— b —— c —— a

그림 2 블록 참조 행태의 패턴화

3.4 참조 문자와 참조 블록 간의 사상표

앞에서 각 블록의 참조 행태를 참조 패턴으로 변환하는 것을 설명했다. 이와 같이 참조 패턴으로 변환함으로써 각 블록별로 참조 특성을 쉽게 파악할 수 있으며 또한 참조 특성이 같은 블록별로 분류할 수 있다. 같은 참조 패턴별로 분류하는 것은프로그램의 실행이 긴 경우 방대한 양의 각 블록의 참조 행태를 매우 효율적으로 유지할 수 있게 해 준다. 왜냐하면 각 블록별로 참조 패턴을 보았을 때 많은 경우에 동일하게 나타나기 때문이다. 예를 들면 순차 블록만을 다음에 참조하는 블록들과 같이 매 참조시마다 단지 하나의 동일한 다음 참조 블록만이 있는 블록들의 참조 행태를 참조 패턴으로 변환하는 경우 같은 참조 패턴으로 나타난다는 것이다. 또한 서로 다른 두 블록 이상의 참조 블록을 가지는 블록들의 경우에도 대부분 동일한 참조 순서를 보여 같은 참조 패턴을 형성한다. 결론적으로 생성된 참조 패턴들은 선인출 상한 모델에서 미래를 내다 보면서 선인출할 수 있도록 하는데 십분 활용될 수 있다.

그러나 각 블록의 참조 행태를 일련의 문자로 구성된 참조 패턴으로 변환함에 따라 각 문자에 해당하는 참조 블록이 어떤 블록인지 참조 패턴만으로는 알 수 없게 된다. 그러므로 선인출 상한 모델에서 이들 참조 패턴을 바탕으로 선인출하기 위해서는 각 블록별로 각 문자에 해당하는 블록을 알 수 있도록 해야 한다. 이를 위해 각 블록별로 각 문자에 해당하는 블록이 무엇인지를 알 수 있게 하는 별도의 사상표(mapping table)를 생성한다. 예를 들어 그림 1에서 블록 b_1의 경우 참조 패턴으로 변환시 다음 참조 블록 b_2가 문자 a로 표시되는데 이 때 블록 b_1의 경우에는 블록 b_2가 문자 a로 표시되었음을 알 수 있도록 하는 참조 문자와 참조 블록 간의 사상표를 만든다. 마찬가지로 블록 b_2의 경우에는 문자 a가 블록 b_3을 나타내고 문자 b가 블록 b_{10}을 나타냄을 사상표에 기록한다. 표 1이 위와 같은 경우에 각 블록별로 각 참조 문자에 해당하는 참조 블록을 보여주고 있다.

표에서 보는 것처럼 각 블록에서 나타나지 않는 참조 문자의 항은 '--'으로 표시된다.

표 1 참조 문자와 참조 블록 정보

블 록	참 조 문 자					
	a	b	c	d	e	…
블록 b_1	b_2	-	-	-	-	…
블록 b_2	b_3	b_{10}	-	-	-	…
블록 b_3	b_4	b_{20}	-	-	-	…
블록 b_4	b_5	b_{30}	-	-	-	…
블록 b_5	b_6	b_{15}	-	-	-	…

3.5 선인출 예측 방법

다음으로 선인출 상한 모델에서 참조 패턴을 바탕으로 선인출 예측하여 선인출하는 과정을 설명한다. 앞에서 생성된 참조 패턴은 선인출 상한 모델에서 미래를 내다보는 효과를 가져오는데 사용된다. 즉 사후 분석에서 선인출 상한 모델은 생성된 참조 패턴을 제일 앞에서부터 순서대로 따라가면서 선인출한다. 이를 위해서는 사후 분석시 각 블록은 참조될 때마다 처음 참조부터 시작하여 현재까지의 참조 패턴을 유지하는 것이 필요하다. 왜냐하면 현재까지의 참조 패턴을 알아야만 다음에 그 블록이 다시 참조되었을 때 이전까지의 참조 패턴(바로 직전까지 그 블록이 참조되면서 생성한 참조 패턴으로 이를 본 논문에서는 이전 참조 패턴(prefix)이라고 함.)을 기준으로 참조 패턴상에서 다음 참조 문자를 알아낼 수 있기 때문이다.

이와 같이 이전 참조 패턴을 유지하여 선인출하게 되는 경우 이전 참조 패턴에서 가능한 다음 참조 패턴이 하나 이상으로 나타날 수 있다. 이때 여러 개의 가능한 참조 패턴 중에서 하나를 선택해야 하는 문제가 발생한다. 이러한 경우에 제안한 선인출 상한 모델에서는 이전 참조 패턴에서 가능한 다음 참조 패턴 중에서 빈도수가 가장 큰 방향으로 선인출한다. 예를 들어, 그림 2에서와 같이 이전 참조 패턴이 a--b이면 제시하는 선인출 상한 모델에서는 문자 b에 해당하는 블록을 선인출한다. 왜냐하면 그림에서 보면 참조 패턴 a--b 다음에 가능한 다음 참조 패턴이 a--b--a, a--b--b, a--b--c인데, a--b--b는 블록 b_3, b_4 두 곳에서 나타나는 반면 a--b--a와 a--b--c는 블록 b_2와 블록 b_5에서 각각 한번씩 밖에 나타나지 않으므로 참조 패턴 a--b 다음에 가장 큰 빈도수를 나타내는 참조 패턴이 a--b--b이기 때문이다.

표 2 참조 패턴의 빈도수

이전 참조 패턴	다음 참조문자					
	a	b	c	d	e	…
a	1	4	-	-	-	…
a-a	1	-	-	-	-	…
a-b	1	2	1	-	-	…
a-a-a	1	-	-	-	-	…
a-a-b	-	-	-	-	-	…
a-b-a	-	1	-	-	-	…
a-b-b	1	-	-	-	-	…
a-b-c	1	-	-	-	-	…

이처럼 선인출 예측시 참조 패턴의 빈도수가 가장 큰 방향으로 선인출하기 위해서는 각각의 참조 패턴에 대해 빈도수를 알 수 있어야 한다. 예를 들어, 그림 2의 경우 참조 패턴 a 다음에 가능한 참조 패턴이 a--a, a--b인데 참조 패턴 a--a는 블록 b_1에서만 나타나므로 이의 빈도수는 1이고 반면에 a--b는 블록 b_2, b_3, b_4, b_5에서 나타나므로 이의 빈도수는 4가 된다. 이와 같은 방식으로 그림 2의 모든 참조 패턴에 대해서 빈도수를 구하여 표를 구성하면 표 2와 같다. 이 표는 가능한 모든 참조 패턴 뿐만 아니라 해당 참조 패턴 다음에 올 수 있는 모든 참조 패턴의 참조 빈도수를 보여준다. 단, 참조 빈도수가 0이거나 참조가 가능하지 않는 경우에는 '--'로 표기하였다. 즉 참조 패턴중에서 a--a--b는 그림 2에서 나타날 수 없으므로 모든 참조 빈도수를 나타내는 항이 '--'으로 채워진다.

예를 들어, 표 2와 같이 참조 패턴에 대한 빈도수가 주어지는 경우 선인출 상한 모델에 의해 실제로 선인출되는 참조 문자는 표 3과 같다. 표에서 보면, 이전 참조 패턴이 a인 경우를 제외하고는 이전 참조 패턴에서 가능한 다음 참조 패턴 중에서
빈도수가 가장 큰 방향으로 선인출됨을 알 수 있다. 즉 이전 참조 패턴 a--a는 문자 a, 이전 참조 패턴 a--b는 문자 b, 이전 참조 패턴 a--a--a는 문자 a 등으로 가능한 다음 참조 중에서 빈도수가 가장 큰 방향으로 선인출한다.

그러나 이전 참조 패턴이 a인 경우에는 가능한 다음 참조 패턴 중 다음 참조 패턴 a--b가 가장 큰 빈도수를 나타내지만 다음 참조 패턴 a--b가 아니라 a--a의 방향으로 선인출한다. 이는 각 블록에서 각 문자에 해당하는 블록을 알 수 있도록 하는 사상표를 유지하는 방법과 관련이 있다. 즉 제안하는 선인출 상한 모델에서는 이러한 사상표를 사후 분석시에 동적으로 생성하여 유지하기 때문에 문자 b에 해당하는 블록이 무엇인지는 그때까지 그 블록이 참조된 적이 없으므로 그 시점에서는 알 수 없다. 이 때는 차선책으로 블록을 알 수 있으면서 동시에 참조 패턴 중에서 빈도수가 가장 높은 참조 패턴 방향으로 선인출한다. 그러므로 이전 참조 패턴이 a인 경우에 다음 참조 패턴 a--a의 방향으로 선인출 예측을 하여 문자 a에 해당하는 블록을 선인출한다. 물론 이러한 경우는 전체적으로 볼 때 거의 발생하지 않으므로 크게 문제가 되지는 않는다.

표 3 선인출 참조 문자

이전 참조 패턴	다음 참조문자
a	a
a-a	a
a-b	b
a-a-a	a
a-a-b	-
a-b-a	b
a-b-b	a
a-b-c	a

3.6 전체적인 선인출 과정

지금까지 설명한 선인출 상한 모델의 전체적인 선인출 과정이 그림 3과 같다. 그림에서 보면 선인출 상한 모델이 사후 분석 기법을 사용하므로 두 단계로 처리됨을 알 수 있다. 먼저 첫번째 단계에서는 각 블록의 참조 행태를 추출하여 참조 패턴으로 변환하고 각 참조 패턴의 빈도수를 계산하여 선인출 상한 모델에서 선인출시 활용할 수 있도록 한다. 다음으로 두번째 단계에서는 첫번째 단계에서 생성한 참조 패턴과 각 참조 패턴의 빈도수를 이용하여 선인출하고 그 결과로서 선인출 예측 정확도를 계산하여 얻는다. 참고로 각 문자에 해당하는 블록 정보를 유지하는 표는 두번째 단계에서 동적으로 생성되어 활용된다.

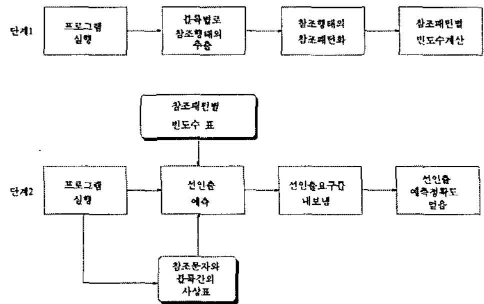


그림 3 두 단계로 처리되는 선인출 과정

이상과 같이 설명한 선인출 상한 모델의 전체적인 알고리즘은 그림 4와 같다. 그림의 앞 부분은 사후 분석 기법의 첫번째 단계를 처리하는 과정을 나타내며 뒷 부분은 사후 분석 기법의 두번째 단계를 처리하는 과정을 나타낸다. 그림의 제일 마지막 단계에서 알 수 있듯이 궁극적으로 선인출 예측 정확도를 계산하여 산출해 낸다.

```
B_p = {b_1, b_2, ... b_|B_p|}은 프로그램 P에서 사용된 블록들의 집합
L_b은 b후에 즉시 참조되는 블록들의 리스트
for each b ∈ B_p
    L'_b ← symbolize(L_b)

for 심볼릭 리스트 L'_b1, L'_b2, ... L'_b|Bp| 에 나타나는 각 이전 참조 패턴 p
    A = {a_1, a_2, ... a_k}는 p에 나타나는 참조 문자들을 원소로 하는 집합
    for each a in A
        L'_b1, L'_b2, ... L'_b|Bp| 에서 p||a의 빈도수 계산, 여기서 ||은 concatenation 연산
    a_p,max가 집합 A안의 원소중 가장 높은 빈도수를 가진 원소

n_hits ← 0
n_misses ← 0
for each b ∈ B_P
    while (L'_b 이 비어있지 않으면)
        ℓ ← null
        actual ← delete_first(L'_b) // delete_first(L)는 L에서 첫 문자를 제거
                                    // 그리고 제거된 참조 문자를 복귀
        prediction = a_ℓ,max
        if (actual = prediction) then n_hits ← n_hits + 1
                                 else n_misses ← n_misses + 1
        ℓ ← ℓ||actual

achievable_prefetch_accuracy_bound ← n_hits / (n_hits + n_misses)
```

그림 4 선인출 상한 모델의 전체적인 알고리즘

3.7 선인출의 예

제안한 선인출 상한 모델에 의해서 그림 1의 예에 대해 선인출하는 과정이 그림 5에 나타나 있다. 그림은 제안한 선인출 상한 모델에 의해 각 블록의 참조시 다음 참조 예상 블록을 선인출하는 경우에 선인출 성공과 선인출 실패를 보여준다. 다만, 그림에서 보는 것처럼 각 블록이 처음 참조시에는 다음 참조 블록을 선인출하지 않는 것으로 되어 있다. 왜냐하면 처음 참조되는 블록은 이전 참조 패턴 뿐만 아니라 각 참조 문자에 해당하는 다음 참조 블록을 아직 알 수 없기 때문이다. 그림 5의 예에서 선인출 성공의 수와 선인출 실패의 수를 알아내어 그 선인출 예측 정확도를 계산하면 다음과 같다.

● 선인출 예측 정확도= $\frac{\text{전체선인출성공의갯수}}{\text{전체선인출요구의갯수}} = \frac{9}{15} = 0.6$

이 예측 정확도는 제안하는 선인출 상한 모델에 의해 계산된 선인출 예측 정확도의 상한을 내포한다. 즉 선인출로 메모리 접근 지연 시간을 최대로 줄이는 측면에서 제안한 선인출 상한 모델보다 다른 어떤 선인출 기법에 의해서도 더 높은 선인출 예측 정확도를 얻을 수 없다는 것이다. 이러한 선인출 상한 모델의 타당성은 다음 두 가지 가정하에서 유효하다.

1. 주어진 알고리즘은 결정적(deterministic)이다.
2. 주어진 알고리즘은 각 참조 패턴을 생성한 명령어 블록별로 구별하지 않는다.

첫번째 가정은 이전 참조 패턴으로 다음 참조 패턴 중 단지 한 방향으로만 나아감을 의미한다. 두번째 가정은 각 참조 패턴과 그 빈도수는 알 수 있지만 각 참조 패턴이 어느 블록에 의해 생성되었는지는 알 수 없다는 것을 말한다.

블록 b_1 : b_2 (*no prefetch*) — b_2 (*hit*) — b_2 (*hit*) — b_2 (*hit*)
블록 b_2 : b_3 (*no prefetch*) — b_{10} (*miss*) — b_3 (*miss*) — b_{10} (*hit*)
블록 b_3 : b_4 (*no prefetch*) — b_{20} (*miss*) — b_{20} (*hit*) — b_4 (*hit*)
블록 b_4 : b_5 (*no prefetch*) — b_{30} (*miss*) — b_{30} (*hit*) — b_5 (*hit*)
블록 b_5 : b_6 (*no prefetch*) — b_{15} (*miss*) — b_{25} (*miss*) — b_6 (*hit*)

그림 5 선인출 상한 모델에 의한 선인출의 예

4. 선인출 특성 분석

4.1 분석 방법

제안한 선인출 상한 모델을 바탕으로 대표적인 SPEC 벤치마크 프로그램들에 대한 실험을 통해 선인출 예측 정확도의 이론적인 상한을 분석하여 제시한다. 분석을 위하여 SPEC 벤치마크[1] 프로그램 중에서 6개의 벤치마크(*Gcc, Xlisp, Espresso, Doduc, Spice, Fpppp*) 프로그램들을 선택하고 각각 약 1억개씩의 메모리 참조를 MIPS RS-2030 워크스테이션(workstation)으로부터 *pixie*[15] 유틸리티를 사용하여 얻어내었다. 그런 다음 이 메모리 참조(트레이스)를 입력받아 앞에서 제시한 선인출 상한 모델을 이용하여 선인출 예측 정확도의 상한선을 계산하는 프로그램을 C 언어로 작성하여 SUN SPARCstation 상에서 실행시켰다. 이러한 분석 프로그램은 선인출 상한 모델의 설명 부분에서 알아보았듯이 두단계로 나뉘어 실행된다. 첫번째 단계에서는 각 벤치마크 프로그램의 트레이스를 기반으로 각 블록의 참조 행태로부터 참조 패턴을 생성하고 각 참조 패턴의 빈도수를 나타내는 표를 구한다. 두번째 단계에서는 첫번째 단계에서 얻은 참조 패턴과 각 참조 패턴의 빈도수에 따라 선인출하여 그 결과로서 선인출 예측 정확도를 계산해 낸다. 분석시 고려한 가변 인자로는 크게 세 가지가 있다.

첫째는 캐쉬 크기가 명령어 선인출에 미치는 영향이다. 쓰레드 선인출 기법의 성능 분석 결과에서 캐쉬 크기에 의해 선인출 효율성과 선인출 성능이 크게 좌우됨을 알 수 있었다. 따라서 캐쉬가 명령어 선인출에 미

치는 영향을 체계적으로 알아보기 위해 캐쉬를 고려하지 않은 경우와 캐쉬를 고려한 경우로 구분하여 명령어 선인출의 특성을 분석하였다. 여기에서 캐쉬를 고려하지 않은 경우란 프로그램 실행에서 나타나는 각 블록의 참조 행태를 모두 추출하여 참조 패턴을 생성한 후 그 참조 패턴을 이용하여 선인출하는 경우를 말한다. 반면에 캐쉬를 고려한 경우란 캐쉬 접근 실패가 일어난 각 블록의 참조 행태에 대해서만 참조 패턴을 생성한 후 그 참조 패턴을 이용하여 선인출하는 경우를 말한다.

둘째는 캐쉬 블록의 크기가 명령어 선인출에 미치는 영향이다. 블록 단위의 선인출에 있어서 블록의 크기는 선인출 내보내는 갯수와 선인출 예측 정확도에 영향을 미친다. 즉 블록 크기가 커지면 실제로 선인출 요구의 수는 그 크기에 비례하여 감소하게 된다. 또한 블록 크기가 커지면 선인출 예측 정확도를 향상시키는 요인과 그 반대로 선인출 예측 정확도를 감소시키는 요인이 모두 발생하게 된다. 선인출 예측 정확도를 향상시키는 요인으로는 블록 크기가 커지면 선인출 예측하여 내보낸 블록에서 선인출이 성공될 가능성이 커지는 것을 들 수 있다. 왜냐하면 블록 크기가 커지면 선인출 예측이 잘못된 경우라도 실제 참조 명령어가 운좋게 선인출한 블록 안에 있을 가능성이 커지기 때문이다. 예를 들면, 분기 명령어를 기준으로 앞으로 분기(forward branch)가 되는 경우에는 그 분기가 매우 짧고 블록이 충분히 크다면 순차 선인출하는 경우에나 분기 목적지를 정확하게 선인출하는 경우에나 모두 선인출 성공을 가져오게 된다. 왜냐하면 앞으로 나가는 짧은 분기는 분기 목적지가 다음 순차 블록에 속할 가능성이 크기 때문이다. 반면에 선인출 예측 정확도를 감소시키는 요인으로는 블록이 커지면 결국 좀더 먼 미래에 참조될 블록을 선인출해야 되므로 선인출 예측이 더욱더 어려워지는 것이 있다. 이는 전진 선인출에서 선인출 정확도가 감소하는 이유와 일맥 상통한다. 그러므로 블록 크기의 변화는 이러한 상반되는 두가지 요인이 복합적으로 명령어 선인출에 영향을 줄 것이다.

셋째로 전진 선인출이 명령어 선인출에 미치는 영향이다. 앞장의 쓰레드 선인출 기법의 분석에서 메모리 접근 지연 시간이 큰 경우에는 선인출 시기를 앞으로 당기는 전진 선인출이 필요함을 알아보았다. 전진 선인출은 선인출 시기를 앞으로 얼마나 이동하는지에 따라 전진 정도를 1,2,4 등으로 표시한다. 즉 전진 정도 1은 지금 프로세서에서 참조되고 있는 블록 바로 다음에 참조될 블록을 선인출하며 전진 정도 2는 현재 프로세서에서 참조되고 있는 블록을 기준으로 실행 흐름 상에서 다음 다음에 참조될 블록을 선인출한다. 마찬가지로 전진 정도 4는 현재 참조 블록을 기준으로 실행 흐름 상으로 4 번째 블록 뒤에 참조될 블록을 선인출한다. 제안한 선인출 상한 분석 모델에서는 이와 같이 전진 선인출을 가능하게 하기 위해 각 블록의 참조 행태를 추출시에 다음 참조 블록 대신에 전진 정도만큼 후에 참조될 블록을 추출하고 이에 대해 참조 패턴을 생성한다. 이 경우 선인출 상한 모델은 실제로 전진 정도만큼 후에 참조될 블록을 선인출하게 되므로 전진 선인출 효과를 반영할 수 있다.

4.2 성능 분석 결과

그림 6은 캐쉬 고려 여부, 블록 크기 변화, 그리고 전진 선인출 정도가 선인출 예측 정확도에 미치는 영향을 종합적으로 보여주고 있다. 그림에서 보면 캐쉬 고려 여부가 명령어 선인출의 선인출 예측 정확도에 가장 크게 영향을 줌을 알 수 있다. 즉 캐쉬를 고려하지 않고 각 블록의 모든 참조 행태에 대해 참조 패턴을 가진 경우에 매우 높은 선인출 예측 정확도가 가능하다. 표 4는 각 벤치마크 프로그램에 대해 캐쉬를 고려하지 않은 경우 전진 정도와 블록 크기에 따른 선인출 예측 정확도를 보여주고 있다. 표에서 보는 것처럼 모든 벤치마크 프로그램들의 선인출 예측 정확도가 블록 크기나 전진 정도와는 거의 상관없이 99%이상으로 높게 나타남을 알 수 있다.

표 4 선인출 예측 정확도(캐쉬를 고려하지 않는 경우)

벤치마크	전진 정도 1		전진 정도 2		전진 전도 4	
	블록 크기 =16	블록 크기 =32	블록 크기 =16	블록 크기 =32	블록 크기 =16	블록 크기 =32
Gcc	99.95	99.79	99.92	99.77	99.72	99.51
Xlisp	99.99	99.95	99.98	99.96	99.94	99.70
Espresso	99.98	99.95	99.98	99.92	99.62	99.15
Doduc	99.99	99.96	99.99	99.94	99.88	99.57
Spice	99.98	99.93	99.98	99.92	99.88	99.55
Fpppp	99.99	99.99	99.99	99.99	99.96	99.93

그러나 캐쉬를 고려한 경우에는 캐쉬를 고려하지 않은 경우에 비해 선인출 예측 정확도가 급격히 떨어짐을 관찰할 수 있다. *Fpppp*와 *Doduc* 벤치마크 프로그램을 제외한 대부분의 벤치마크 프로그램들에서 비교적 작은 2K 바이트 캐쉬 크기에서도 선인출 예측 정확도가 크게 낮아진 것으로 나타났다. 특히 *Espresso*와 와 *Spice* 벤치마크 프로그램은 캐쉬 크기가 2K 바이트만 되어도 선인출 예측 정확도가 50%이하로 급격히 떨어

진다. 그 원인은 캐쉬를 고려할 경우 캐쉬가 메모리 참조의 규칙성을 무작위화(randomization)시키기 때문이다. 즉, 캐쉬를 고려할 경우 캐쉬에서 캐쉬 접근 실패가 되는 메모리 참조는 거의 규칙성이 없기 때문에 이를 기초로 하여 선인출하는 것은 선인출 상한 모델을 가정한다고 해도 선인출 예측 정확도가 그다지 높지 않다. 이는 다단계 캐쉬(multi-level cache)에서 2단계 캐쉬(second-level cache)의 캐쉬 접근 성공률이 그다지 높지 않은 것과 그 맥락을 같이 한다.

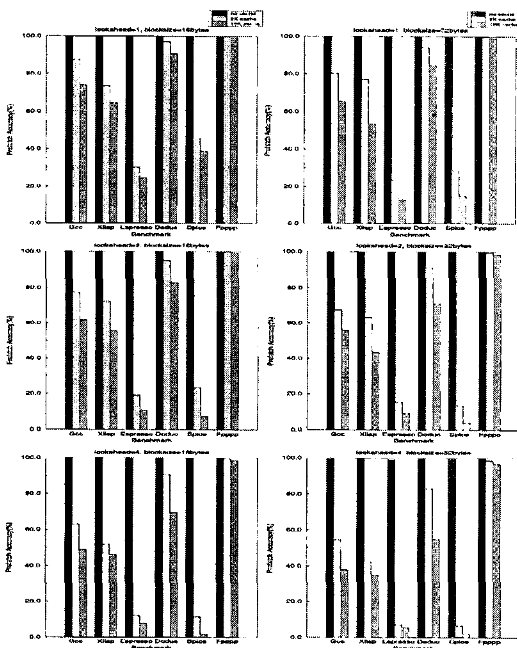


그림 6 선인출 예측 정확도

그림 7은 캐쉬가 있는 경우에 선인출 예측 정확도가 떨어지는 이유를 구체적으로 설명하고 있다. 한 예로, 프로그램 실행 흐름 상에서 블록 b_i에 대한 참조 시간 t_i부터 t_{i+6}까지의 참조 행태가 그림에 나타나 있다. 그림에서 블록 b_i는 시간 t_i, t_{i+1}, t_{i+2}, t_{i+4}, t_{i+5}에서 참조될 경우에는 블록 b_{i+1}이, 시간 t_{i+3}에서 참조될 경우에는 블록 b_j가, 시간 t_{i+6}에서 참조될 경우에는 블록 b_k가 블록 b_i의 다음 참조 블록이 됨을 알 수 있다. 이 때 처음 참조되는 시간 t_i에서 블록 b_{i+1}, 시간 t_{i+3}에서 블록 b_j, 그리고 시간 t_{i+6}에서 블록 b_k를 참조하는 경우에는 캐시 접근 실패가 발생하고, 반면에 시간 t_{i+1}, t_{i+2}, t_{i+4}, t_{i+5}에서 블록 b_{i+1}를 참조할 때에는 캐쉬 접근 성공이 발생한다고 하자. 그러면 제안한 선인출 상한 모델에서 캐쉬를 고려한 경우에는 블록 b_i의 참조 행태는 블록 b_{i+1}, 블록 b_j, 블록 b_k의 순서로 세 블록으로 나타날 것이다. 이들 참조 행태에 대한 참조 패턴을 기반으로 선인출 상한 모델에서의 선인출은 다음과 같이 이루어진다. 먼저 시간 t_i에서 블록 b_i가 참조될 때 참조 행태에 대한 참조 패턴에 의해 다음에 참조될 블록 b_{i+1}을 선인출한다. 이 때에는 실제로 프로세서에 의해 블록 b_i 다음에 블록 b_{i+1}이 참조될 것이므로 선인출 성공이 된다. 그러나 시간 t_{i+1}에서는 참조 행태에 대한 참조 패턴에 의해 다음에 블록 b_j가 캐쉬에서 캐쉬 접근 실패가 발생됨을 알게 된다. 그러므로 이 때는 선인출 상한 모델이 블록 b_j를 선인출하도록 하는 선인출 요구를 내보내게 된다. 그렇지만 실제로 프로그램 실행에서는 이때에 블록 b_j를 참조하는 것이 아니라 이미 캐쉬에 존재하는 블록 b_{i+1}을 다시 참조함을 알 수 있다. 따라서 선인출한 블록 b_j는 참조되지 못하고 결국 불필요한 선인출이 되고 만다. 이러한 현상은 시간 t_{i+2}에서도 마찬가지로 일어난다. 시간 t_{i+3}에 이르러서야 블록 b_i 다음에 블록 b_j를 참조하여 선인출한 블록 b_j가 선인출 성공이 된다. 그림은 블록 b_i가 시간 t_i에서부터 t_{i+6}까지 7번 참조되는 동안 블록 b_j와 블록 b_k를 각각 두 번씩 불필요하게 선인출함을 보여준다.

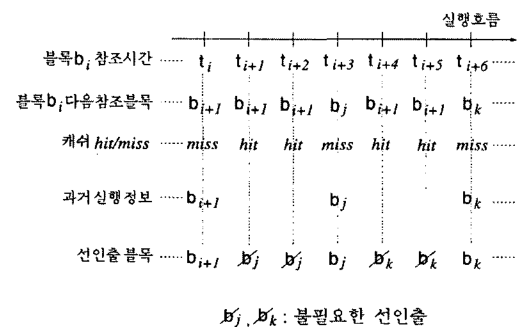


그림 7 선인출 예측 정확도 저하 원인

그러나 예외적으로 벤치마크 프로그램 중에서 *Fpppp*와 *Doduc* 벤치마크 프로그램은 캐쉬를 고려할 경우에도 다른 벤치마크 프로그램에 비해 상대적으로 높은 선인출 예측 정확도를 나타내고 있다. 이는 크게 두 가지의 요인으로 설명될 수 있다. 하나는 이들 벤치마크 프로그램이 실행중 참조하는 상이한 블록의 수가 많아서 프로그램에서 많이 사용되는 블록을 캐쉬가 모두 수용할 수 없기 때문이다[4]. 따라서 다른 벤치마크 프로그

램들과는 다르게 명령어 참조의 규칙성이 많이 보존되므로 상대적으로 높은 선인출 예측 정확도를 보이는 것이다. 다른 하나는 이들 프로그램이 매우 강한 순차적 실행 특성을 보여 선인출 예측이 매우 용이하게 되므로 선인출이 잘못되는 경우가 거의 발생하지 않는다는 것이다. 예를 들면 *Fpppp* 벤치마크 프로그램의 경우 프로그램의 실행중 참조된 전체 블록 중 순차적인 다음 참조 블록이 99% 이상임을 알 수 있었다. 마찬가지로 *Doduc* 벤치마크 프로그램의 경우에도 프로그램의 실행 동안 참조된 전체 블록 중에서 순차적인 다음 참조 블록이 약 90% 정도임을 알 수 있었다.

이상과 같이 제안한 선인출 상한 모델에서 캐쉬를 고려하여 캐쉬 접근 실패만의 참조 행태를 이용하여 캐쉬 접근 실패가 예상되는 블록을 선인출하는 것이 불필요한 선인출을 발생시킴을 알아보았다. 이러한 불필요한 선인출이 발생하는 근본적인 이유는 다음과 같이 두 가지로 설명될 수 있다.

하나는 시스템 성능을 최대화하려는 목적을 위해서는 캐쉬 고려시 참조될 가능성이 높은 방향보다는 다음에 캐쉬 접근 실패가 예상되는 방향으로 선인출해야 한다는 것이다. 즉 캐쉬가 있으면 캐쉬 접근 실패시에만 메모리 접근 지연 시간이 발생하므로 캐쉬 접근 실패를 예상하고 그를 선인출하는 것이 시스템 성능을 최대화하는 측면에서 필요하기 때문이다. 따라서 제안한 선인출 상한 모델처럼 캐쉬 접근 실패 방향으로 선인출 할 수 밖에 없다.

또 하나는 선인출 상한 모델이 캐쉬 접근 실패를 일으키는 블록들의 참조 행태 정보만을 저장하고 이를 선인출시에 활용하기 때문에, 어느 시점에서 캐쉬 접근 실패가 예상되는 블록이 어느 것인지는 알 수 있지만 실제로 캐쉬 접근 실패가 예상되는 블록이 프로세서에 의해 언제 참조될 지는 알지 못한다는 것이다. 이 때문에 캐쉬 접근 실패가 예상되는 블록이 선인출되지만 많은 경우에 캐쉬 내에서 실제로 필요한 블록을 프로세서가 참조하므로 선인출된 블록이 프로세서에 의해 즉시 사용되지 않게 된다. 이런 경우 언젠가는 사용될 것이라고 보고 선인출 버퍼에 오랫동안 저장해 두는 방법을 생각해 볼 수 있다. 그렇지만 이를 가능하도록 하기 위해서는 프로그램에 따라 다르겠지만 일반적으로 상당히 큰 선인출 버퍼가 요구된다. 왜냐하면 선인출 버퍼의 크기가 작으면 다른 블록의 참조시에 발생하는 또다른 선인출에 의해 선인출 버퍼에 오래 남아 있을 수가 없어서 실제로 사용되기도 전에 선인출 버퍼에서 방출되기 때문이다.

전체적으로 요약하면, 캐쉬를 고려할 경우에 캐쉬에서 캐쉬 접근 실패가 발생한 명령어 블록들을 기반으로 선인출하면 선인출 예측 정확도가 낮아지게 된다. 따라서 선인출에 의해 메모리 접근 지연 시간을 줄임으로써 얻어지는 시스템 성능 향상보다는 불필요한 선인출에 의한 시스템 성능 저하가 더 커져서 전체 시스템 성능이 오히려 저하되는 경우가 발생할 수 있음을 예상할 수 있다.

그림 6은 캐쉬가 명령어 선인출에 미치는 영향과 함께 블록 크기와 전진 선인출에 의한 영향을 보여주고 있다. 이들의 영향은 캐쉬에 비해 전반적으로 크지 않음을 알 수 있다. 그림에서 보는 것처럼 블록 크기의 변화가 선인출 예측 정확도에 적게 영향을 주는 것을 볼 때, 앞의 분석 방법에서 언급한 선인출 예측을 이롭게 하는 요인과 선인출 예측을 어렵게 하는 요인이 엇비슷함을 알 수 있다. 그러나 전반적으로 블록 크기가 커짐에 따라 완만하게 선인출 예측 정확도가 떨어지는 것으로 그림에서 관찰되므로 블록 크기가 커짐에 따라 선인출 예측이 더 어렵게 하는 요인이 더 크다고 볼 수 있다. 한편 전진 정도가 커지면 예상대로 선인출 예측 정확도가 감소하지만 그 정도가 작음을 볼 수 있다. 이는 제안한 선인출 상한 모델이 각 블록의 참조 행태를 모두 알고 있어서 전진 정도에 크게 관계없이 선인출 예측할 수 있기 때문이다.

5. 결 론

지금까지 대부분 명령어 선인출에 대한 연구는 특정 선인출 기법을 제시하고 그의 성능을 평가하는데 그쳐 왔다. 본 논문에서는 기존의 연구와는 달리 명령어 선인출의 선인출 예측 정확도 측면에서 이론적이고 체계적인 분석을 시도하였다. 이를 위하여 명령어 선인출 예측 정확도의 상한을 제시할 수 있는 이론적 분석을 위한 선인출 상한 모델을 제안하고 SPEC 벤치마크 프로그램에 대해 제안한 선인출 상한 모델을 적용하여 분석 결과로 선인출 예측 정확도의 상한을 제시하였다. 분석 결과에서 나타난 가장 큰 특징으로 캐쉬의 유무에 따라 선인출 예측 정확도의 상한이 크게 달라짐을 관찰할 수 있었다. 즉 캐쉬가 있는 경우 캐쉬가 메모리 참조의 규칙성을 무작위화 (randomization) 시키기 때문에 이를 기초로 하여 선인출 하는 경우에는 이론적 분석을 위한 선인출 상한 모델을 가정하여도 선인출 예측 정확도가 그다지 높지 않음을 관찰할 수 있었다. 이상과 같은 관찰 결과와 선인출 시 발생하는 오버헤드 (선인출 하고자 하는 블록의 캐쉬내 존재 유무를 확인

하기 위한 캐쉬 lookup과 불필요한 선인출에 의한 오버헤드)를 고려할 때 캐쉬를 사용하는 컴퓨터 시스템에서는 명령어 선인출이 전체 시스템의 성능을 오히려 저하시킬 수 있음을 알 수 있었다.

참 고 문 헌

[1] SPEC Newsletter, Vol. 1, 1989.

[2] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support on Programming Languages and Operating Systems*, pages 40-52, 1991.

[3] T. Chen and J. Baer. Reducing memory latency via non-blocking and prefetching caches. Techinical report, Department of Computer Science and Engineering, University of Washington, Jun. 1992.

[4] J. D. Gee and M. D. Hill and A. J. Smith. Cache Performance of the SPEC Benchmark Suite. Technical Report UCB/CSD 91/648, Computer Science Division, University of California, Berkeley, Oct. 1991.

[5] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 354-368, 1990.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative approach (second edition)*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[7] N. P. Jouppi. Improving Directed-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364-373, 1990.

[8] S. B. Kim and M. S. Park and S. Park and S. L. Min and H. Shin and C. S. Kim and D. Jeong. Threaded prefetching: An adaptive instruction prefetch mechanism, *Microprocessing and Microprogramming*, 39, 1993.

[9] A. C. Klaiber and H. M. Levy. Architecture for software-controlled data prefetching. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 43-63, 1991.

[10] R. L. Lee. *The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Multiprocessors*. PhD Thesis, University of Illinois, Urbana-Champaign, May 1987.

[11] R. L. Lee, P. C. Yew, and D. H. Lawrie. Data prefetching in shared memory multiprocessors. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages 28-31, 1987.

[12] A. J. Smith. Bibliography and readings on CPU cache memories. *Computer Architecture News*, 14(1):22-42, Jan. 1986.

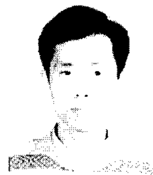
[13] A. J. Smith. Second bibliography on cache memories. *Computer Architecture News*, 19(4):154-182, Jun. 1991.

[14] A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473-530, Sep. 1982.

[15] M. D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Stanford University, Nov. 1991.

[16] D. Thiebaut. From the fractal dimension of the intermiss gaps to the cache miss ratio. *IBM Journal of Research and Development*, 32(6):796-803, Nov. 1988.

[17] D. Thiebaut and H. S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4):305-329, Nov. 1987.

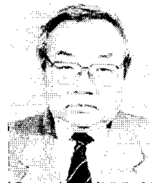


김 성 백

1989년 2월 서울대학교 컴퓨터공학과 학사. 1991년 2월 서울대학교 컴퓨터공학과 석사. 1995년 8월 서울대학교 컴퓨터공학과 박사. 1996년 8월 ~ 현재 제주대학교 컴퓨터교육과 조교수. 관심분야는 컴퓨터 구조, 컴퓨터 교육

민 상 렬

정보과학회논문지 : 시스템 및 이론
제 27 권 제 3 호 참조



김 종 상

1960년 서울대학교 공과대학 전자공학과 학사. 1965년 서울대학교 공과대학 전자공학과 석사. 1975년 서울대학교 공과대학 전자공학과 박사. 1979년 ~ 현재 서울대학교 공과대학 컴퓨터공학부 교수. 1986년 ~ 1988년 한국정보과학회 회장