# 페이지 접근 정보에 기반한 효율적인 명령어 캐쉬 선인출 기법

## (An Efficient Instruction Prefetching Scheme Based on the Page Access Information)

신 숭 현 † 김 철 홍 †† 전 주 식 †††
(Soong Hyun Shin) (Cheol Hong Kim) (Chu Shik Jhon)

**요 약** 컴퓨터 시스템의 1차 캐쉬 적중률은 시스템의 성능을 평가함에 있어 가장 중요한 요소 중 하나이다. 하위 메모리 구조로부터 1차 캐쉬로의 선인출은 1차 캐쉬의 적중률을 증가시키기 위해 사용되는 대표적인 기술 중 하나이다. 본 논문에서는 명령어 캐쉬의 선인출 효율은 높이고 선인출에 소모되는 비용은 감소시키는 재 접근 페이지 선인출 정책을 제안하고자 한다. 재 접근 페이지 선인출 정책은 수행되는 명령어들의 소속 페이지를 추적하여, 이 정보를 바탕으로 일정 횟수 이상 같은 페이지를 접근하는 경우에 한하여 선인출을 수행함으로써, 명령어 캐쉬로의 선인출 횟수는 줄이고 선인출 성공률은 향상시킨다. 또한, 일반적인 컴퓨터 시스템에서 하나의 2차 캐쉬 블록에 여러 개의 1차 캐쉬 블록이 포함되는 특성을 이용하여 미스 블록과 선인출 블록이 같은 2차 캐쉬 블록에 포함된 경우로 선인출을 한정함으로써 선인출에 소모되는 비용을 줄인다. 모의 실험에 따르면 제안하는 구조는 최대 6.7%의 성능향상을 보인다.

**키워드** : 명령어 캐쉬, 캐쉬 선인출, 메모리 페이지 접근 정보

***Abstract*** In general, the hit ratio of the first level cache is one of the most important factors in determining the performance of computer systems. Prefetching from lower level memory structure is one of the most useful techniques for improving the hit ratio of the first level cache. In this paper, we propose a prefetch on continuous same page access (CSPA) scheme which improves the prefetch efficiency of the instruction cache and reduces prefetch cost at the same time. The proposed CSPA scheme traces the page addresses of executed instructions to count how many times the same memory page is accessed continuously. To increase the prefetch efficiency, the CSPA scheme initiates prefetch only if the number of accesses to the same page exceeds the threshold value. Generally, the size of a L1 cache block is smaller than that of a L2 cache block. Therefore, one L2 cache block contains a number of L1 cache blocks. To reduce the number of unnecessary accesses to the L2 cache due to prefetch, the CSPA scheme enables prefetch only when the missed L1 block and the prefetch L1 block are in the same L2 cache block, leading to reduced prefetch cost. According to our simulations, the proposed prefetching scheme improves the performance by up to 6.7%.

**Key words** : Computer architecture, Instruction cache, Cache prefetch

## 1. 서 론

컴퓨터 시스템을 설계함에 있어서, 평균 메모리 접근 시간을 줄이는 것은 시스템의 성능을 향상시키기 위해 필요한 가장 중요한 요소 중 하나이다. 최근, 메모리 접근 시간과 프로세서 속도의 차이가 지속적으로 증가함에 따라 메모리 계층 구조(Hierarchical Memory Architecture)의 하위 계층으로의 접근 시간은 갈수록 커지게 되었다. 따라서, 하위 메모리에 비해 상대적으로 빠른 접근 시간을 제공하는 1차 캐쉬(First Level Cache)의 적중률(Hit Ratio)을 높이는 것이 컴퓨터 시스템의 성능을 결정하는 중대한 요소가 되었다.

1차 캐쉬의 적중률을 높이기 위한 연구는 그 동안 지속적으로 이루어졌다. 그 중, 메모리 계층 구조의 하위 계층에서 1차 캐쉬로의 선인출(Prefetch)은 1차 캐쉬의

† 학생회원 : 서울대학교 전기 컴퓨터공학부
shordan@panda.snu.ac.kr
cheolhong@gmail.com
†† 종신회원 : 서울대학교 전기 컴퓨터공학부 교수
csjhon@riact.snu.ac.kr
논문접수 : 2005년 7월 15일
심사완료 : 2006년 2월 15일

적중률을 높이기 위한 대표적인 기술 중 하나이다. 특히, 선인출은 데이타 캐쉬에 비해 지역성(Locality)이 높은 명령어 캐쉬에서 효과가 크다. 선인출은 소프트웨어와 하드웨어의 두 가지 방법으로 구현된다. 소프트웨어 기반 선인출 방식은 컴파일러가 프로그램을 분석한 후 적당한 위치에 선인출 명령어를 삽입하는 방법으로 이루어지고, 일반적으로 데이타 캐쉬에 자주 적용된다. 하지만, 소프트웨어 기반 선인출 기법을 사용할 경우, 컴파일 시에는 선인출에 적당한 블록을 정확하게 알기 힘들다는 이유로 많은 선인출 기회를 놓칠 수 밖에 없다는 단점이 있다. 소프트웨어 기반의 선인출과 달리 하드웨어 기반 선인출 기법은 실행 중의 정보를 이용하여 선인출 작업을 수행하게 되므로 정확성이 높다는 장점이 있다. 그러나 하드웨어 기반 선인출은 추가적인 하드웨어를 필요로 한다. 본 논문에서는, 추가적인 하드웨어를 최소로 하면서 명령어 캐쉬의 성능 향상을 위한 하드웨어 기반 선인출 기법을 제안하려 한다.

기존의 선인출 정책 중 명령어 캐쉬를 위한 하드웨어 선인출 정책 혹은 명령어 캐쉬에 적용 가능한 하드웨어 선인출 정책들은 다음과 같다. 하드웨어 선인출 기법은 순차(Sequential) 선인출과 비 순차 선인출로 구분된다. 순차 선인출 기법은 Smith 등에 의해 처음으로 제안되었다[1]. Smith는 OBL(One Block Lookahead)이라는 순차 선인출 기법을 바탕으로, 지속 선인출(Always-Prefetch), 캐쉬 미스 선인출(Prefetch-on-Miss), 태그 선인출(Tagged-Prefetch)의 세 가지 정책들을 제시하였다. 지속 선인출 기법은 모든 메모리 접근에 대해 해당 블록(Block)의 다음 블록을 메모리에서 캐쉬로 선인출을 하는 기법이다. 하지만, 이 기법은 메모리 접근이 발생할 때마다 선인출이 메모리로 요구되기 때문에 메모리 접근 수가 굉장히 많아지는 문제점이 발생한다. 캐쉬 미스 선인출 기법은 캐쉬 미스가 발생하면 미스 블록(Missed Block) 외에 그 다음 블록까지 총 두 개의 블록을 캐쉬로 옮긴다. 일반적으로 많이 사용되는 캐쉬 미스 선인출 기법은 선인출이 캐쉬 미스 시에만 발생하기 때문에 선인출 횟수가 지속 선인출 기법에 비해서는 상당히 줄어들게 된다. 태그 선인출 기법은 각 메모리 블록에 태그 비트(Tag Bit)를 하나씩 추가하여 이 태그 비트를 이용해 같은 블록이 불필요하게 여러 번 선인출되는 것을 방지한다. 캐쉬가 참조될 때 해당 블록의 태그 비트도 함께 참조되는데, 만약 태그 비트가 세트 되어 있다면 선인출이 이루어진다. 반대로 태그 비트가 세트 되어 있지 않다면, 이미 그 전에 다음 블록이 선인출되었다는 것을 의미하므로, 선인출이 이루어지지 않는다. Dahlgren 등은 선인출 거리(Prefetch Degree: 선인출 할 블록의 수)를 동적으로 조정하는 개선된 순차 선인출 법을 제안하였다[2]. 이 기법은 전체 선인출에 대해 선인출 효율을 주기적으로 계산하고, 이 비율에 따라서 선인출 거리를 조정한다. 즉, 선인출 효율이 높은 부분에서는 선인출 거리를 늘려 선인출을 충분히 이용하고, 효율이 떨어지는 부분에서는 이 거리를 줄여서 불필요한 선인출을 줄인다. Jouppi는 스트림 버퍼 큐(FIFO Stream Buffer)를 두고, 이 버퍼에 메모리에서 캐쉬로 선인출 된 블록들을 저장하는 방법을 제안하였다[3]. 이 기법은 선인출로 인한 캐쉬 폴루션(Cache Pollution)을 피할 수 있다는 장점이 있다. 버퍼에 저장된 블록은 프로세서로부터 접근이 요청되면 캐쉬로 옮겨지기 때문이다. 하지만, 캐쉬 미스가 발생한 경우, 스트림 버퍼 큐의 헤드(Head)에 해당 블록이 없다면, 이 버퍼는 완전히 비워지고 새 블록들이 이 스트림 버퍼 큐에 채워진다. 그러므로, 이 기법은 캐쉬의 폴루션을 줄일 수 있다는 장점이 있으나, 만약 캐쉬 미스가 발생하고 버퍼 큐의 헤드에 원하는 블록이 없는 경우에는 버퍼에 저장된 모든 블록들에 대한 선인출 작업이 불필요한 작업이 되어 제거된다는 단점이 있다. 더구나, 캐쉬의 연관도(Associativity)가 높아지면 캐쉬 폴루션의 피해가 점차 줄어들기 때문에 추가적인 버퍼의 필요성은 낮아진다.

Reinman 등이 제안한 분기 방향 명령어 선인출 기법(Fetch Directed Instruction)은 비 순차 선인출 기법의 하나로 분기 예측 회로(Branch Predictor)의 정보를 선인출에 이용한다[4]. 이 기법에서 선인출 작업은 분기 예측 회로가 예상한 프로그램 순서를 따라서 수행된다. 그러므로 이 기법은 분기 예측 회로의 정확도가 높을수록 더욱 효과적이다. 그러나 선인출 하드웨어가 분기 예측 회로와 연동하여야 하는 만큼 하드웨어 복잡도가 높다는 단점이 있다. 분기 방향 명령어 선인출 기법과 유사하게 프로그램의 흐름을 이용하는 정책으로는 캐쉬 미스 간의 상호 연관성을 이용하여 선인출을 수행하는 명령어 수행 기록에 따른 선인출 기법(Execution History Guided Instruction)이 있다[5]. Bacher 등이 제안한 집단 캐쉬 미스 기록 선인출 기법(Cluster Miss Prediction Scheme)은 임베디드 프로세서(Embedded Processor)의 명령어 캐쉬에 관한 정책으로, 특정 프로그램을 수행하는 실시간 네트워킹 시스템을 적용 대상으로 가정하여, 사전에 프로그램을 프로파일(Profile)하여 선인출이 필요한 부분을 미리 테이블에 기록하는 방법을 통해 선인출을 수행한다[6].

대부분의 1차 캐쉬 선인출 기법에서는, 하위 메모리 구조(2차 캐쉬 또는 메인 메모리)로의 접근을 인출 접근(Actual Lookup)과 선인출 접근(Prefetch Lookup)의 두 가지로 구분한다[1]. 인출 접근은 1차 캐쉬의 미스로 발생하는 것으로, 미스 블록을 가져오기 위한 하위 메모

리 구조의 접근이고, 선인출 접근은 프로세서에서 요구가 있기 전에 미리 1차 캐쉬에 블록을 저장해 두기 위한 접근이다. 만약 선인출의 예상이 빗나가면 선인출 접근은 필요 없는 작업이 되고, 이 선인출은 아무런 이득 없이 하위 메모리의 접근 횟수만 증가시켜 평균 메모리 접근 시간에 악영향을 미친다.

명령어 캐쉬에서는 지역성의 특성으로 인해 순차적 선인출 기법이 비순차적 선인출에 비해 더 효과적이지만[7], 일반적으로 선인출 기법은 하위 메모리 구조로의 접근 횟수를 상당히 증가시킨다. 본 논문에서는, 메모리 트래픽(Memory Traffic)의 증가를 최소화하면서 충분한 선인출 효과를 얻기 위해 하위 메모리 구조로 개별적인 선인출 접근을 요구하는 기존의 기법들과 달리, 미스 블록과 그 다음 블록을 동시에 요청하고 동시에 메모리 페이지의 접근 패턴을 추적하여 실행되는 명령어들의 페이지 정보를 선인출의 정확성을 높이는데 사용하는 재 접근 페이지 선인출 기법을 제안한다.

본 논문의 구성은 다음과 같다. 2장에서는 본 논문의 동기를 기술하고, 3장에서는 본 논문에서 제안하는 재 접근 페이지 선인출 기법을 설명한다. 4장에서는 제안하는 기법의 효율성을 측정하기 위한 모의 실험 방법과 실험 결과를 보여준다. 끝으로, 5장에서 결론을 맺는다.

## 2. 동 기

### 2.1 참조된 캐쉬 블록 간 상대 거리

일반적으로 분기 명령어(Branch Instruction)는 전체 명령어 중 0~30%를 차지하므로, 명령어는 대체로 메모리 주소의 순서를 따라 순차적으로 실행된다고 할 수 있다[8,9]. 본 논문에서는, 명령어 캐쉬에서 이러한 특성이 어떻게 나타나는지를 실험을 통해 확인하였다. 실험은 SimpleScalar[10]를 이용하여 수행하였고, 벤치마크 프로그램은 SPEC CPU2000[9]을 수행하였으며, 그림 1, 2는 전체 SPEC CPU2000을 수행한 결과의 평균이다.

그림 1은 명령어 캐쉬에서 연속적으로 참조되는 캐쉬 블록 간 상대 거리에 따른 확률을 나타낸다. 그래프의 X축인 '블록 간 상대 거리'는 연속적으로 참조된 명령어 캐쉬의 블록 간 상대 거리를 의미한다. 예를 들어, $n$번째 명령어가 수행된 후, $n$+1번째 명령어가 수행된 경우, 이 두 명령어가 같은 캐쉬 블록에 포함되었다면 캐쉬 블록 간 상대 거리는 0이고, $n$+1번째 명령어가 포함된 캐쉬 블록이 n번째 명령어가 포함된 캐쉬 블록의 다음 캐쉬 블록이라면 캐쉬 블록 간 상대 거리는 1이 된다. 그림 1에서는 같은 캐쉬 블록의 명령어가 연속적으로 수행되는 경우, 즉 블록 간 상대 거리가 0인 경우는 제외하였다. 블록 간 상대 거리가 0인 경우는, 명령어 선인출의 관점에서 볼 때 이미 필요한 블록이 명령어 캐쉬에 존재하기 때문에 무의미하기 때문이다. 그림 1에서 보는 바와 같이, 블록 간 상대 거리가 1일 확률이 70%를 넘는다는 것을 알 수 있다. 이를 통해, 명령어 캐쉬에서 이미 수행된 블록의 다음 블록은 상당히 높은 확률로 참조된다는 사실을 확인하였다. 그러므로, 명령어 캐쉬에서 미스가 발생하는 경우 요청된 블록 외에 그 다음 블록을 함께 요청한다면 명령어 캐쉬의 적중률을 높일 수 있을 것으로 기대된다. 그러나 모든 캐쉬 미스마다 두 개의 연속된 블록을 요청하는 것은 시스템 버스의 사용률을 심각하게 높일 수 있고, 이로 인해 긴급한 캐쉬 미스 처리가 지연될 수 있다.

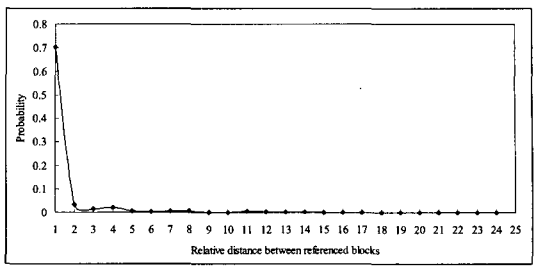


그림 1 수행 명령어 간 상대 거리에 대한 확률 분포
(명령어 캐쉬 블록 크기=32B, 메모리 페이지 크기=4KB)

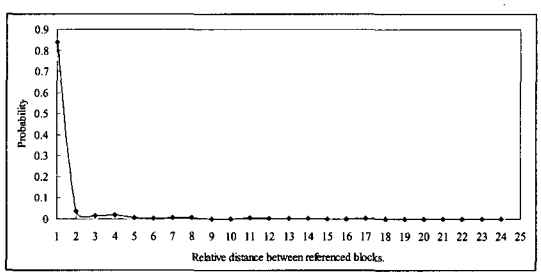


그림 2 동일 페이지 접근 시, 수행 명령어 간 상대 거리에 대한 확률 분포
(명령어 캐쉬 블록 크기=32B, 메모리 페이지 크기=4KB)

명령어 선인출의 효율성을 높이는 방법을 찾기 위해, 참조된 블록들이 같은 메모리 페이지 안에 포함된 경우의 블록 간 상대 거리를 측정하였다. 그림 2에서 보는 바와 같이, 후속 명령어가 이전 명령어와 같은 페이지에 포함된 경우에는 블록 간 상대 거리가 1일 확률이 84%이다. 그림 1과 그림 2에서 보인 결과를 종합하면, 캐쉬 미스가 발생 시 미스 블록과 다음 블록이 같은 페이지 내에 존재할 경우에만 다음 블록을 선인출 한다면, 선인출 된 블록을 프로세서가 참조할 확률이 70%에서 84%로 높아지므로 선인출의 효율성을 상당히 (14%) 높일 수 있을 것으로 기대된다. 또한 추가 실험 결과, 캐시 블록 크기와 메모리 페이지 크기를 달리하여도 비슷한 결과를 보인다는 것을 확인할 수 있었다.

### 2.2 메모리 단계 간 캐쉬 블록의 상대 크기

캐쉬 블록의 크기는 캐쉬의 목적에 따라 차이를 보인다. 큰 캐쉬 블록을 사용하는 캐쉬는 공간 지역성(Spatial Locality)을 활용하는데 효과적이고 강제 미스(Compulsory Miss)를 낮추는 효과가 있지만 용량 미스(Capacity Miss)를 높이는 단점이 있다[11]. 작은 캐쉬 블록을 사용하는 캐쉬는 큰 캐쉬 블록을 사용하는 캐쉬에 비해 미스 페널티(Miss Penalty)를 줄이고 시간 지역성(Temporal Locality)을 활용하는데 효과가 있지만, 공간 지역성 측면에서는 효율성이 상대적으로 낮아진다. 따라서 캐쉬 블록의 크기는 반드시 어느 한 쪽이 우수하다 할 수 없으며 해당 캐쉬의 목적에 따라 결정된다. 이러한 캐쉬 블록의 특성에 착안하여 캐쉬 블록의 크기가 다른 두 개의 캐쉬로 공간 지역성과 시간 지역성을 각각 이용하는 방법들이 제안되었다[12,13].

일반적으로 1차 캐쉬에서는 미스 페널티를 줄이고 시간 지역성을 높이기 위해 작은 크기의 캐쉬 블록을 사용하는 반면, 2차 캐쉬에서는 1차 캐쉬에 비해 큰 블록 크기를 사용하여 공간 지역성을 이용하게 된다. Hennessy의 저서에 따르면, 1차 캐쉬의 블록 크기로는 4~32 바이트, 2차 캐쉬의 블록 크기로는 32~256 바이트가 적당하다고 한다[11]. Alpha AXP 21064는 1차 캐쉬의 블록 크기는 64 바이트로 2차 캐쉬의 블록 크기는 256 바이트로 설계되었다[11]. 이러한 내용들을 바탕으로 본 논문에서는, 1차 캐쉬의 블록 크기로 32 바이트, 2차 캐쉬의 블록 크기로 256 바이트를 각각 가정하였다. 따라서 하나의 2차 캐쉬 블록에는 8개의 1차 캐쉬 블록이 포함되는데 2차 캐쉬 블록에 포함된 1차 캐쉬 블록들을 서브 블록[1]이라 하겠다.

1차 캐쉬로의 선인출을 수행하게 되면, 정상적인 인출 작업과 마찬가지로, 2차 캐쉬를 검색하여 요구된 블록을 넘겨 준다. 선인출 작업이 정상적인 인출 작업과 분리되는 이유는 이 두 작업이 함께 이루어진다면 미스 페널티가 높아질 수 있고, 분리함으로써 좀 더 중요한 정상적인 인출 작업에 우선권을 줄 수 있기 때문이다. 그러나 본 논문의 실험 결과에서 알 수 있듯, 같은 페이지를 접근하는 경우에는 미스 블록의 다음 블록을 선인출 하는 경우 효율성이 84% 정도로 상당히 높은 편이고, 두 작업이 함께 이루어져 높아지는 미스 페널티는 긴급 블록 우선 정책(Critical Word First Technique)을 이용하면 해결된다[11]. 따라서 수행된 명령어의 메모리 페이지 정보를 통해 선인출 효율성을 높일 수 있다면, 정상적인 인출 블록과 선인출 블록을 동시에 전송하는 기법이 선인출 비용을 줄이고 2차 캐쉬의 접근 횟수를 줄일 수 있다는 장점을 가지게 된다.

## 3. 재 접근 페이지 선인출 기법(Prefetch on Continuous Same Page Access)

본 논문에서 제안하는 재 접근 페이지 선인출 기법(prefetch on Continuous Same Page Access: CSPA)은 (1) 메모리 페이지 연속 재 접근 횟수(Continuous Same Page Hit: CSPH)가 기준치를 만족하는지, (2) 선인출 하려는 캐쉬 블록이 하위 메모리의 마지막 서브 블록이 아닌지의 두 가지 조건을 만족할 때 선인출을 수행하도록 구현된다. 제안된 구조는 그림 3, 4에서 보이는 바와 같다. 본 논문에서는 2 단계 캐쉬 구조를 가정하였고, 1차 캐쉬는 명령어 캐쉬와 데이타 캐쉬로 구분된다.

CSPA 기법에서는 측정된 CSPH 값에 따라 같은 페이지에 접근할지 여부를 예측한다. 만약 CSPH가 특정 값 이상이면 다음 명령어가 현재 수행되는 명령어와 같은 페이지에 위치할 것으로 예측하여, 명령어 캐쉬와 2차 캐쉬 사이의 선인출을 수행한다. 그림 4에서 보는 바와 같이, CSPH를 측정하기 위해 직전에 수행된 명령어의 페이지 번호는 previous page #에 저장된다. 현재 수행되는 명령어의 페이지 번호는 previous page #와 비교되고, 두 값이 일치하면 CSPH counter를 증가시키고, 그렇지 않으면 CSPH counter를 리세트 하고 현재의 페이지 번호를 새롭게 previous page #에 기록한다. 이를 통해, Fetch enable 신호는 CSPH 값이 일정 값 이상이면 세트 된다.

앞 장에서 설명한 바와 같이, 대부분의 선인출 정책에서 메모리 접근은 인출 접근(Actual Lookup)과 선인출 접근(Prefetch Lookup)의 두 종류로 구분된다. 예를 들어, 캐쉬 미스 선인출 기법의 경우, 그림 5에서와 같이 1차 캐쉬에서의 캐쉬 미스로 인해 하위 메모리로 A00 블록을 요청한다면, A00 블록은 1차 캐쉬로 인출 접근을 통해 전달되고, A00 블록의 다음 블록인, A01 블록은 선인출 접근에 의해 참조된다. 즉, 선인출을 위해 추가적인 메모리 참조가 발생하게 된다. 이 경우, 선인출을 한 블록이 사용되지 않는다면, 평균 메모리 접근 시간은 선인출 메모리 접근 횟수만큼 증가하게 될 것이다. 설령 선인출한 블록이 사용되더라도, 평균 메모리 접근 횟수가 줄어들지는 않는다. 이러한 문제점을 해결하기 위해서는 인출과 선인출이 동시에 이루어져야 하고, 인출 블록과 선인출 블록이 2차 캐쉬의 같은 블록 안에 존재하는 경우에만 선인출을 수행하여야 한다. 이를 위

1) 일반적인 서브 블록(sub-block)은 미스 페널티를 줄이기 위한 방법으로 유효 비트(valid bit)별로 블록 내부에서 나뉘어진 하부 블록을 의미한다. Write-back 캐쉬의 경우 dirty bit의 단위로 나뉘어진다[11]. 본 논문에서는 2차 캐쉬 블록 내부의, 1차 캐쉬 블록 단위로 나뉜 하부 블록의 의미로 사용된다.

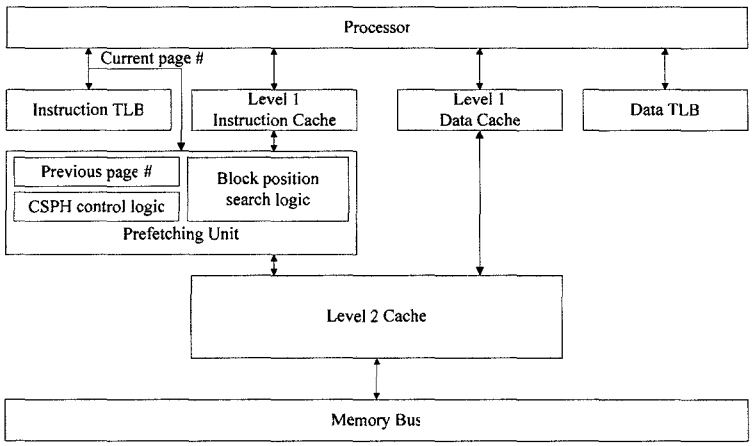


그림 3 CSPA 선인출 유닛을 포함한 전체 시스템 구조

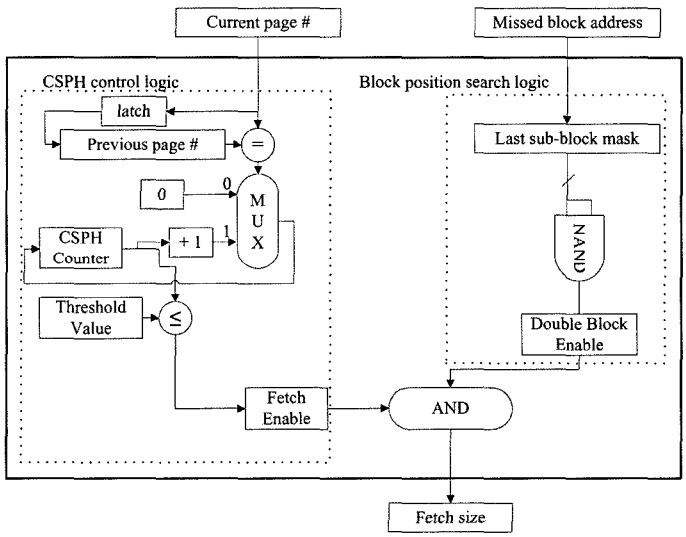


그림 4 제안하는 선인출 모듈의 구조

해, CSPA 기법에서는 A00과 A01이 한 번의 메모리 참조 요구에 의해 1차 캐쉬로 전달된다. 뿐만 아니라, 두 블록을 동시에 가져옴으로써 2차 캐쉬로의 접근이 한 번으로 줄어드는 효과를 얻을 수 있고, 지연 시간은 2차 캐쉬 접근 시간 + 1 사이클이 소요된다. 그림 6과 같이 인출되는 블록이 하위 메모리 블록의 마지막 서브블록인 경우(인출 블록과 선인출 블록이 같은 블록에 존재하지 않는 경우)에는 선인출을 수행하지 않고, A03 블록만을 인출한다 (A04 블록을 선인출하지 않는다). 그 이유는 인출 요구 블록이 하위 메모리 블록의 마지막 서브블록인 경우는 위에서 설명한 방법, 즉 인출과 선인출을 동시에 요구함으로써 얻는 이득을 얻지 못하기 때문이다. 인출 블록과 선인출 블록이 각기 다른 하위 메모리 블록, 그림 5, 6의 경우 2차 캐쉬의 서로 다른 두 블록에 존재하므로 불가피하게 두 번의 메모리 접근이 필요하며, 지연 시간은 2차 캐쉬 접근 시간의 두 배가 소요되고 하드웨어 복잡도는 높아진다.

미스 블록이 하위 메모리 블록의 마지막 서브블록인지의 여부는 간단한 방식으로 알 수 있다. 그림 7(a)와 같이 메모리 주소는 태그, 인덱스, 블록 오프셋 필드로 나뉜다. 그림 7(b), 7(c)는 본 논문에서 가정한 1차, 2차 캐쉬의 블록 오프셋 마스크를 나타낸다. 그림 7(d), 그림 7(e)는 각각 상위 블록 오프셋과 하위 블록 오프셋의 예이다. 그림 7(d), 그림 7(e)의 예와 같이, 2차 캐쉬의 블록 오프셋은 둘로 나뉠 수 있는데, 하위 부분은 1차 캐쉬의 블록 오프셋과 중복되고, 상위 부분은 2차 캐쉬

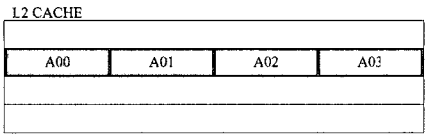


그림 5 선인출 작업이 가능한 경우

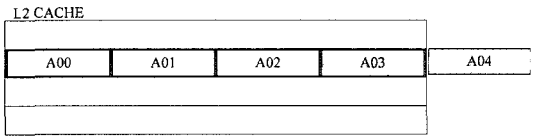


그림 6 선인출 작업이 불가능한 경우

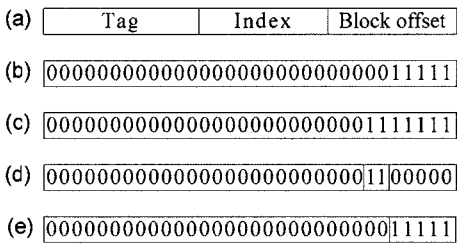


그림 7 캐쉬 주소의 구조. (a)는 일반적인 메모리 어드레스의 구조, (b),(c)는 각각 32바이트, 128바이트의 캐쉬 블록의 블록 오프셋 비트 마스크, (d),(e)는 각각 (c)의 상위 오프셋 비트 마스크와 하위 오프셋 비트 마스크를 각각 나타낸다.

내의 서브블록의 위치를 의미한다. 그림 7의 예는 1차 캐쉬 블록의 크기가 32바이트, 2차 캐쉬 블록의 크기가 128바이트인 경우로서, 각각 블록 오프셋이 5, 7 비트를 차지한다. 이 경우, 2차 캐쉬는 1차 캐쉬에서 미스된 블록을 찾기 위해, 그 블록이 속한 2차 캐쉬 블록을 찾고 2차 캐쉬의 상위 블록 오프셋을 이용해 미스된 블록을 찾아서 1차 캐쉬로 넘겨준다. CSPA 기법에서는 2차 캐쉬의 상위 블록 오프셋 값이 모두 1인 경우에는 1차 캐쉬에서 미스된 블록이 2차 캐쉬 내의 마지막 서브블록이라는 의미이므로 선인출을 수행하지 않는다.

결과적으로, 미스된 블록이 하위 메모리 구조의 마지막 서브블록이 아닌 경우에만 double block enable 신호가 세트 된다. 만약 fetch enable과 double block enable이 모두 세트 되면 fetch size 신호가 세트 되고 그림 4와 같이 선인출 모듈은 2차 캐쉬로 2 개의 연속된 1차 캐쉬 블록을 요구한다.

## 4. 성능평가

### 4.1 모의 실험 방법

모의 실험은 SimpleScalar 시뮬레이터를 수정하여 수행하였고, 벤치마크로는 SPEC2000을 이용하였다. 각 벤치마크는 종료 시까지 수행되었으며, 실험된 시스템의 특성은 표 1에서 보이는 바와 같다.

### 4.2 실험 결과

4.2.1 메모리 페이지 연속 재 접근 횟수 기준값에 따른 성능 변화

우선, 제안하는 CSPA 기법에서의 CSPH 기준값을 실험을 통해 결정하였다. 그림 8은 각기 다른 CSPH 기준값에 따른, 명령어 캐쉬 미스율과 수행시간을 나타낸다. 수행 시간은 비 선운출 정책(No-Prefetch)을 기준으로 정규화하였다. 그래프에서 CPSA-*n*은 CSPH 기준값이 n인 CSPA 기법을 의미한다. 그림 8에서 볼 수 있듯이, CSPA-*1*, CSPA-*2*, CSPA-*4*의 성능이 다른 정책에 비해 우수하다. 이 중 CSPA-*4*의 경우에는 CSPA-*1*, CSPA-*2*와 비교하여 선인출 수행횟수가 가장 적으므로, 이하의 실험에서는 CSPH 기준값으로 4를 사용한다. 기준값이 지나치게 커지면 점차 성능 저하를 보이는데, 그 이유는 그림 9에서 확인할 수 있다. 명령어 캐쉬 미스율이 떨어짐에도 불구하고 수행 시간이 길어지는 이유는 선인출의 성공 비율에 비해 중복된 선인출 비율이 급격히 높아졌기 때문이다. 선인출 성공 비율은 선인출 된 블록이 캐쉬에서 교체되기 전까지 사용되는 비율을 의미하고, 중복된 선인출 비율은 선인출 된 블록이 이미 캐쉬에 존재하는 비율을 의미한다. 그림 9에서와 같이, *n*값이 8 이상이 되면 중복 선인출 비율이 높아지는 데,

표 1 모의 실험 인자

| Parameter | value |
|---|---|
| Branch predictor | Bimodal |
| Issue | 2 issue width, out-of-order |
| Resource | 1 Integer ALU, 1 Integer mult., 1 Float ALU, 1 Float mult. |
| Misc. | Fetch queue = 8, Decode width =2, Commit width = 2,<br>RUU size = 8, LSQ size = 8 |
| iL1 cache | 16 KB, 4 way, 32 bytes cache line, 1 cycle latency |
| dL1 cache | 16 KB, 4 way, 32 bytes cache line, 1 cycle latency, write-back |
| L2 cache | Unified, 256 KB, 8 way, 256 bytes cache line, 8 cycle latency |
| Memory latency | 32 cycle latency + 1 cycle / 8 bytes, 2 port |

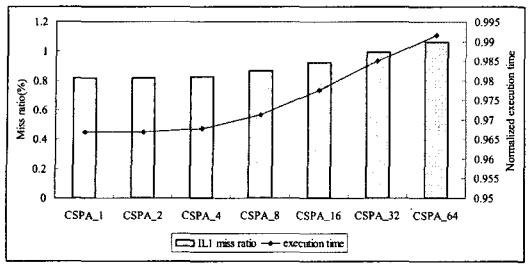


그림 8 명령어 캐쉬 미스율과 수행 시간
(수행 시간은 비선인출 정책에 정규화)

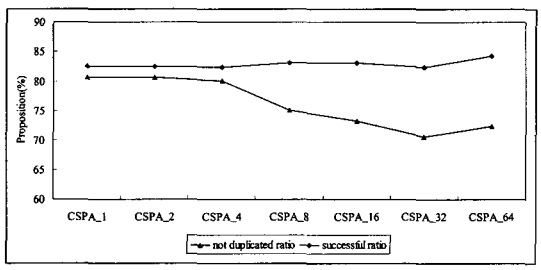


그림 9 정책에 따른 선인출의 비중복율과 성공률

그 이유는 CSPH가 증가함에 따라 선인출로 요구한 블록이 이미 캐쉬에 존재하고 있을 확률이 높아지기 때문이다. 더구나, 그림 2에서 확인한 바와 같이, 동일 페이지 내 명령어의 캐쉬 블록이 연속 접근될 확률은 84%이다. 그리고, CSPA의 선인출 성공 비율은 82~84%로서 CSPA가 선인출 효율을 극대화하기 위해 프로그램의 연속적 수행 특성을 효과적으로 사용한다는 것과, 기준값을 8 이상으로 설정하더라도 추가적인 성능 향상을 기대하기 힘들다는 사실을 알 수 있다.

4.2.2 재 접근 페이지 선인출 기법의 성능 평가

실험을 통해 CSPA와 비선인출 기법, 캐쉬 미스 선인출 기법, 그리고 스트림 버퍼 큐 기법을 비교하였다. 캐쉬 미스 선인출 기법을 비교 대상으로 선택한 이유는 제안하는 CSPA와 하드웨어 복잡도가 비슷하고 명령어 캐쉬에서는 일반적으로 순차 선인출이 비 순차 선인출에 비해 성능이 우수하기 때문이다[7]. 스트림 버퍼 큐 기법은 CSPA나 캐쉬 미스 선인출 기법에 비해 하드웨어 복잡도가 높으나 CSPA와 캐쉬 미스 선인출 기법을 제외한 다른 기법들과 비교하였을 때, 비교적 복잡도가 낮은 편으로 순차적 접근을 보이는 명령어 캐쉬의 접근에 효율적이므로 비교 대상으로 선택하였다. 모든 그래프에서 CSPA는 재 접근 페이지 선인출 기법을 의미하고, stream은 스트림 버퍼 큐를 의미한다. 스트림 버퍼 큐는 4개의 엔트리를 가정하였다[3]. 그림 11에서 stream (w buffer)와 stream(w/o buffer)는 각각 캐쉬 히트의 범위를 스트림 버퍼 큐를 포함하는지, 포함하지 않는지를 나타내며 아래에서 다시 설명하겠다.

그림 10, 11은 비 선인출 정책에 정규화한 IPC(사이클 당 수행 명령어) 값과 명령어 캐쉬의 미스율을 보여준다. 그림 10에서 보는 바와 같이, CSPA의 성능이 가장 우수함을 알 수 있다. CSPA는 비 선인출 기법에 비해 평균적으로 3.2%(그림 10의 AVG 그래프), 특히 crafty와 fma3d에서는 5.6%와 6.7%의 성능 향상을 보인다. CSPA는 또한 캐쉬 미스 선인출 기법에 비해 2.3% 성능을 향상시킨다. 비 선인출 기법의 명령어 캐쉬 미스율이 1.2%인데 비해, CSPA는 0.86%로 명령어 캐쉬 미스율도 0.3% 정도 줄어든다. 캐쉬 미스 선인출 기법의 1차 캐쉬 미스율이 CSPA에 비해 낮음에도 불구하고, CSPA의 수행 시간이 더 짧은 이유는 CSPA가 캐쉬 미스 선인출 기법에 비해 17.2% 적은 선인출 수행을 하기 때문이고, 이러한 특성은 그림 12에서 확인할 수 있다. 그림 12는 선인출 기법들의 선인출 작업 횟수를 비 선인출 기법에 정규화한 값을 보여주는 그래프이며, 모든 선인출 기법 중 CSPA가 선인출 작업을 가장

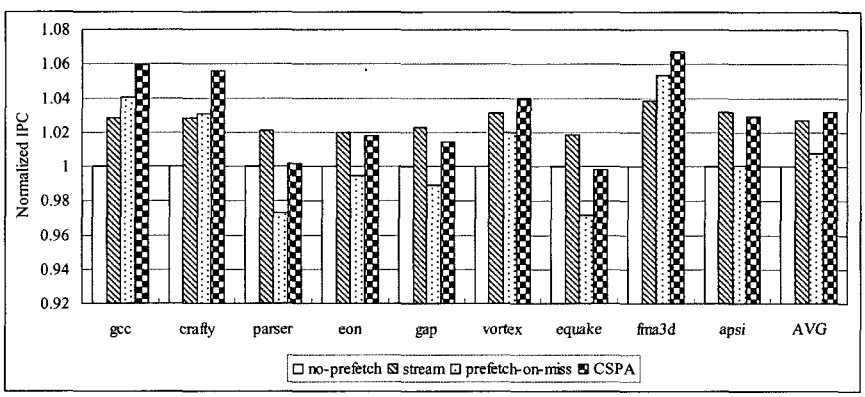


그림 10 비 선인출 기법에 정규화한 IPC 값

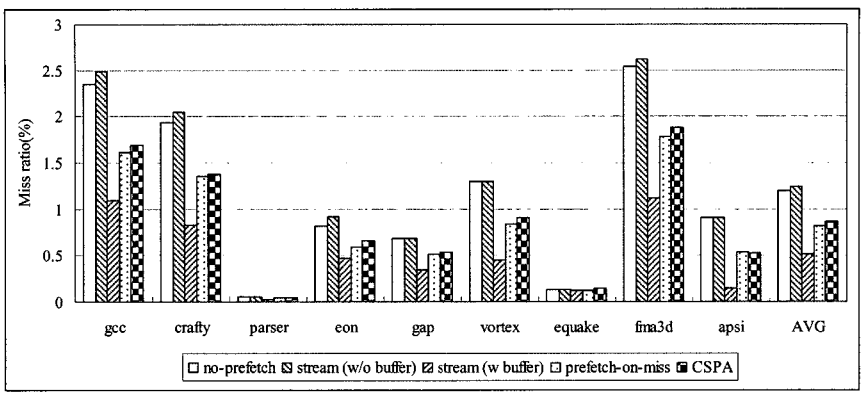


그림 11 명령어 캐쉬 미스율 비교

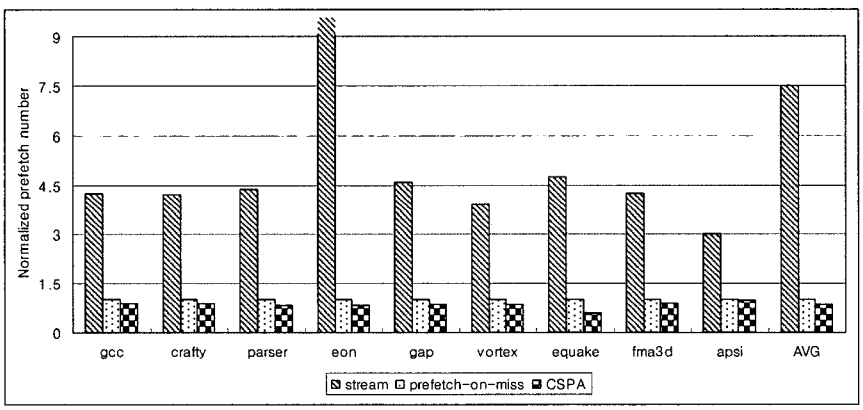


그림 12 총 선인출 작업 횟수 비율

적게 수행함을 보여준다.

그림 10에 따르면 CSPA의 스트림 버퍼 큐 기법에 대한 성능 우위는 IPC 값을 기준으로 평균 0.5% 정도에 불과하여 CSPA와 스트림 버퍼 큐 사이에 성능 상의 차이가 있다고 보기 힘들다. 이것은 스트림 버퍼 큐의 미스율이 CSPA의 미스율에 비해 낮기 때문이다. 스트림 버퍼 큐 기법에서 스트림 버퍼를 포함한 명령어 캐쉬의 미스율은 0.51%로 CSPA에 비해 0.35% 낮다. 그러나 그림 11에서 볼 수 있듯이, 스트림 버퍼를 포함한 명령어 캐쉬의 미스율에 비해 스트림 버퍼를 제외한 명령어 캐쉬 만의 미스율은 1.24%로서 CSPA보다 0.35% 높다. 그리고 선인출의 효율 면에서 CSPA는 스트림 버퍼 큐에 비해 상당한 이점이 있다. 그림 12에 따르면 스트림 버퍼 큐의 총 선인출 작업 횟수의 비율은 캐쉬 미스 선인출 기법의 7.5배에 이른다. 이는 CSPA의 9배에 달하는 선인출 작업 횟수로 스트림 버퍼 큐의 구조 상의 특징에 기인한다. 스트림 버퍼 큐는 명령어 캐쉬와 2차 캐쉬 사이에서 계속적으로 버퍼를 채우는 작업을 수행한다. 만약, 스트림 버퍼 큐에서 미스가 발생하면 큐의 내용을 초기화하고 다시 새 주소에서부터 버퍼를 채워나가는데 이로 인해 선인출 작업 횟수가 증가한다.

CSPA의 성능과 명령어 캐쉬 구조와의 관계를 분석하기 위하여, 캐쉬의 크기를 32 KB, 64 KB로 캐쉬의 연관도를 1-웨이, 16-웨이로, 그리고 캐쉬 블록의 크기를 64 바이트로 하여 모의 실험을 수행하였다. (각각의 경우, 나머지 인자는 표 1의 값을 유지한다.)

그림 13은 캐쉬 구조에 따른 IPC 값으로, CSPA는 모든 명령어 캐쉬 구조에서 가장 우수한 성능을 보였으며, 특히 작은 캐쉬와 낮은 연관도의 캐쉬에서 성능 향상이 두드러진다. 캐쉬가 작거나 연관도가 낮을 때 일반적으로 캐쉬의 미스율은 높아지므로, 명령어 선인출의 효과가 더욱 많이 나타난다. 블록의 크기가 커지면 명령어 선인출의 효과가 줄어드는데, 이는 선인출의 효과가 블록의 크기가 커지면서 이미 나타났기 때문이다. 그러나 블록의 크기가 커졌다고 선인출의 효과가 전혀 없는 것은 아니며 IPC의 증가를 보여주는데, 명령어 캐쉬의 순차적 접근성을 말해준다. 전반적으로 표 1의 인자값으

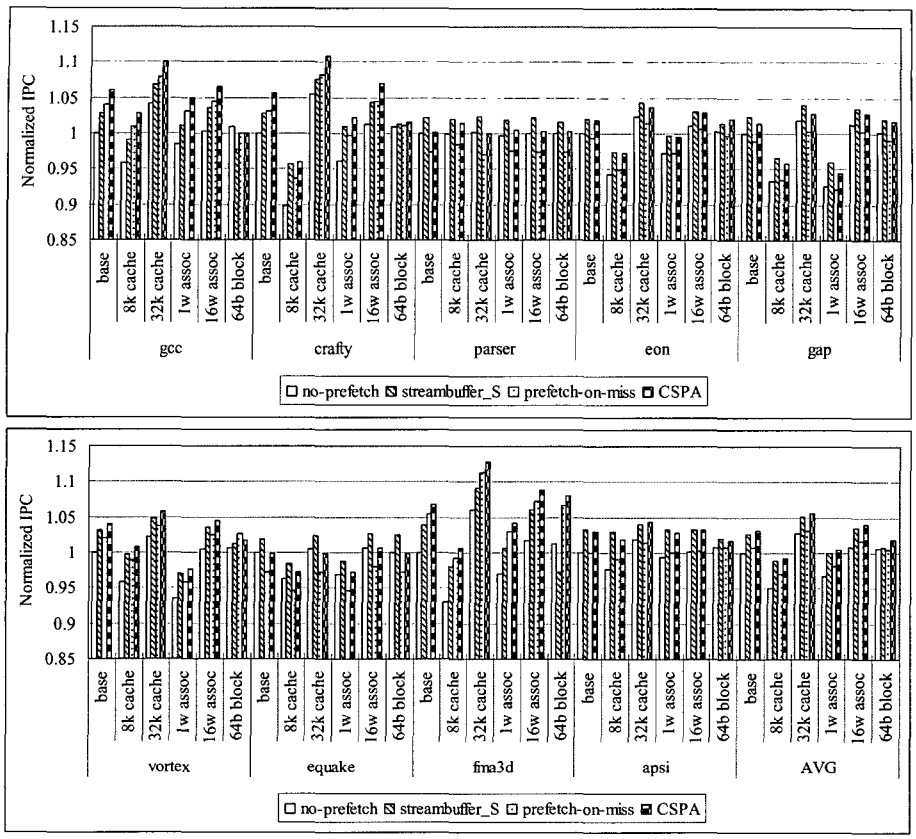


그림 13 캐쉬 구조에 따른 정규화한 IPC 값

로 구성한 시스템의 비선인출 기법의 그래프(그림 13의 base)와 캐쉬를 1-웨이 연관도(1-way associative)로 구성한 시스템의 CSPA 그래프(그림 13의 1w assoc)는 비슷한 IPC값을 보여준다. 그리고, CSPA를 적용한 기본 구조의 성능이 선인출을 하지 않는 32KB 캐쉬의 성능과 그림 13에 따르면 유사함을 알 수 있다. 따라서, CSPA 기법으로 16KB 캐쉬에서 32KB 캐쉬의 성능을 얻을 수 있으며, 1-웨이 캐쉬에서 4-웨이 캐쉬의 성능을 얻을 수 있다 할 수 있다.

## 5. 결 론

캐쉬 선인출은 캐쉬의 적중률을 높여 시스템의 성능을 향상시키는 중요한 기법이다. 특히 명령어 캐쉬의 경우 순차적 특성을 가지고 있어 선인출 기법이 더욱 효과적이다. 그러나 기존의 순차적인 선인출 작업은 선인출 된 블록이 중복되거나 효율성이 떨어지는 경우에는 성능에 악영향을 미치는 단점이 있다.

본 논문에서는, 동일 페이지 연속 재 접근 정보에 기반한 선인출 기법을 제안하였다. 재 접근 페이지 선인출 기법은 간단한 하드웨어에도 불구하고 동일 페이지 접근의 횟수를 측정하고 이를 바탕으로 선인출 여부를 결정하여 선인출 효율성을 높이고, 동시에 명령어 캐쉬와 하위 메모리 구조의 블록 간 크기의 차이를 이용하여 선인출 비용을 낮춘다. 모의 실험에 따르면 제안하는 CSPA 기법은 선인출 횟수를 낮게 유지하면서 성능 향상을 평균 3.2% 가져오는 결과를 보인다. 특히, 캐쉬 크기가 작은 경우나 캐쉬의 연관도가 낮은 경우 성능 향상이 더욱 두드러지므로, 시스템의 특성상 간단한 회로를 지향하는 경우에 효율적인 선인출을 가능하게 한다.

특히 CSPA 기법은 다른 선인출 기법에 접목하기가 간단하여 다양한 응용이 예상된다. 향후, 분기 방향 명령어 선인출이나 명령어 수행 기록에 따른 선인출 기법 등과 같은 다소 복잡한 선인출 기법들에 적용하여 성능 향상의 여지가 있는 지 연구할 계획이다.

## 참 고 문 헌

[1] Smith, A.J.: Cache Memories, Computing Surveys, Vol.14, No.3, pp.473~530, 1982.

[2] Dahlgren, F., Dubois, M. and Stenstrom, P.: Fixed and Adaptive Sequential Prefetching in Shared-memory Multiprocessors, Proc. International Conference on Parallel Processing, I-56-63, 1993.
[3] Jouppi, N.P.: Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers, Proc. 17th International Symposium on Computer Architecture, pp. 364-373, 1990.
[4] Reinman, G., Calder, B., and Austin, T.: Fetch-directed instruction prefetching, In 32nd International Symposium on Microarchitecture, pp. 16-27, 1999.
[5] Zhang, Y., Haga, S., and Barua, R.: Execution History Guided Instruction Prefetching, In Proc. of the 16th International Conference on Supercomputing, pp. 199-208, 2002.
[6] Batcher, K., and Walker, R.: Cluster Miss Prediction with Prefetch on Miss for Embedded CPU Instruction Caches, In Proc. of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pp. 24-34, 2004.
[7] Hsu, W.C. and Smith, J.E.: A Performance Study of Instruction Cache Prefetching Methods, IEEE Transactions on Computers, pp. 497-508, 1998.
[8] Lee, C., Potkonjak, M., and Mangione-Smith, W.H.: Mediabench: A tool for evaluating and synthesizing multimedia and communications systems, In Proc. of the 30th Annual International Symposium on Microarchitecture, pp. 330-335, 1997.
[9] SPEC2000 benchmarks, http://www.spec.org
[10] Burger, D., Austin, T.M., and Bennett, S.: Evaluating future micro-processors: the SimpleScalar tool set. Technical Report TR-1308, Univ. of Wisconsin-Madison Computer Sciences Dept., 1996.
[11] Hennessy, J.L. and Patternson, D.A.: Computer Architecture: A Quantitative Approach, Second Edition, Morgan Kaufmann Publishers, 1996.
[12] Jung-H. L., Seh-woong J., Shin-D. K., and Charles C. W.: An Intelligent Cache System with Hardware Prefetching for High Performance, IEEE Transactions on Computers, pp. 607-616, 2003.
[13] V. Milutinovic, M. Tomasevic, B. Markovic, and M. Tremblay: The Split Temporal/Spatial Cache: Initial Performance Analysis, Proceedings of the SCIzzL-5, Santa Clara, California, USA, pp. 72-78, 1996.

신 승 현
2000년 서울대학교 전기컴퓨터공학부(학사). 2002년 서울대학교 전기컴퓨터공학부(석사). 2002년~현재 서울대학교 전기컴퓨터공학부 박사과정. 관심분야는 컴퓨터 구조, 병렬 처리 시스템, 내장형 시스템

김 철 홍
정보과학회논문지 : 시스템 및 이론
제 33 권 제 3 호 참조

전 주 식
정보과학회논문지 : 시스템 및 이론
제 33 권 제 3 호 참조