Journal of The Institute of Electronics and Information Engineers Vol.52, NO.10, October 2015 http://dx.doi.org/10.5573/ieie.2015.52.10.082
ISSN 2287-5026(Print) / ISSN 2288-159X(Online)

논문 2015-52-10-10

변위 히스토리 버퍼를 이용한 명령어 및 데이터 프리페치 기법

(Instructions and Data Prefetch Mechanism using Displacement History Buffer)

정 용 수*, 김 진 혁*, 조 태 환*, 최 상 방**

(Yong Su Jeong, JinHyuk Kim, Tae Hwan Cho, and SangBang Choi©)

요 약

본 논문에서는 변위 필드를 이용해 히스토리 레코드를 생성하는 방법과 히스토리 레코드의 기준이 되는 트리거 블록에 우선순위를 부여하여 효율적인 캐시 교체를 가능하게 하는 하드웨어 프리페치 기법을 제안한다. 히스토리 레코드의 트리거 블록을 기준으로 히스토리를 생성하기 때문에 프로그램의 시퀀스를 고려할 수 있으며, 히스토리를 변위 값으로 저장하기 때문에 트리거 주소와 변위필드에 저장된 값을 더해 빠르게 명령어 또는 데이터 주소를 프리페치 할 수 있다. 또한, 트리거 블록에 우선순위를 부여하고 캐시 교체 정책으로 랜덤 교체 방법을 사용해 캐시 공간이 가득 찼을 때 우선순위가 낮은 블록부터 랜덤하게 교체하는 방법을 제안한다. 제안하는 하드웨어 프리페처의 성능을 평가하기 위해 메모리 분석 시뮬레이터인 gem5와 PARSEC 벤치마크 프로그램을 사용하였다. 그 결과 비트벡터를 이용해 공간영역을 생성하는 기존의 하드웨어 프리페처와 비교해 L1 데이터 캐시의 미스율은 평균 약 44.5% 감소하였고 L1 명령어 캐시의 미스율은 평균 약 31% 감소하였다. 또한 IPC (Instruction Per Cycle)는 평균 약 23.7% 향상을 보였다.

Abstract

In this paper, we propose hardware prefetch mechanism with an efficient cache replacement policy by giving priority to the trigger block in which a spatial region and producing a spatial region by using the displacement field. It could be taken into account the sequence of the program since a history is based on the trigger block of history record, and it could be quickly prefetching the instructions or data address by adding a stored value to the trigger address and displacement field since a history is stored as a displacement value. Also, we proposed a method of replacing at random by the cache replacement policy from the low priority block when the cache area is full after giving priority to the trigger block. We analyzed using the memory simulator program gem5 and PARSEC benchmark to assess the performance of the hardware prefetcher. As a result, compared to the existing hardware prefecture to generate the spatial region using a bit vector, L1 data cache miss rate was reduced about 44.5% on average and an average of 26.1% of L1 instruction misses occur. In addition, IPC (Instruction Per Cycle) showed an improvement of about 23.7% on average.

Keywords : cache memory, prefetching, spatial correlation

Ⅰ. 서 론

첫 싱글 칩 마이크로프로세서는 1970년대에 발명됐다. 그 이후로 마이크로프로세서는 괄목할 만한 성장세를 보여 왔다. 각 세대를 거치면서 프로세서는 데이터를 빠르게 처리하기 위해 발전했고, 메모리는 많은 양

* 정회원, ** 정회원, 인하대학교 전자공학과
(Department of Electronic Engineering, Inha University)
© Corresponding Author(E-mail: sangbang@inha.ac.kr)
Received ; July 1, 2015 Revised ; September 16, 2015
Accepted ; September 24, 2015

의 데이터를 저장하기 위해 발전했다. 그 결과 프로세서 처리 속도는 메모리 성능과 비교했을 때 몇 십 년 동안 빠른 속도로 성장해왔고. 프로세서와 메모리 간의 성능차이가 크게 벌어졌다[1].

현 세대 프로세서는 프로그램을 실행하는데 수 사이클이 소모된다. 하지만 메모리에서 필요한 데이터를 가져오기 위해서는 수백 사이클을 기다려야하며 약 100배 차이의 속도 때문에 메모리에서 병목현상이 발생한다. 프로세서와 메모리 사이의 속도 격차를 줄이기 위해 메모리를 계층화하여 프로세서와 메모리 사이에 캐시 메모리를 추가해 속도 차이를 줄 일 수 있었다[2~3].

메모리 레이턴시를 줄이기 위한 다른 방법에는 프리페치 기법이 있다. 프리페치 기법은 프로세서가 필요한 데이터를 예측해 미리 가져오는 방법이다. 처음에는 캐시에 어떤 블록도 저장되어있지 않기 때문에 프로세서가 제일 처음 접근하는 블록에서 항상 캐시 미스가 발생한다. 이를 콜드 미스라 하며, 캐시 블록의 크기를 크게 하여 한 블록에 많은 양의 데이터를 저장하거나 명령어가 실행되기 전 프리페치를 통해 미리 블록을 가져와 캐시 미스를 줄일 수 있다. 하지만 예측이 틀린다면 소중한 메모리 대역폭을 낭비하는 결과를 초래한다. 따라서 프리페치를 요청할 때 앞으로 사용될 데이터를 정확하게 예측하여 적절한 시간에 불러오는 것이 매우 중요하다[4].

프리페치 동작은 소프트웨어 또는 하드웨어 수준에서 실행될 수 있다. 소프트웨어에 의한 프리페치 기법은 프로그래머나 컴파일러가 프리페치 명령어를 코드에 삽입하는 방법이다. 하드웨어에 의한 프리페치 기법은 프로세서가 요청한 메모리 주소, 이전 메모리주소, 프로그램 카운터, 메모리 값, 이전 메모리 값 등 여러 정보를 종합하여 프리페치하려는 메모리 주소와 타이밍을 결정한다[5].

본 논문에서는 공간적 지역성을 이용하고 프로그램의 시퀀스까지 고려하기 위해 변위를 이용해 히스토리 레코드를 생성하는 방법과 캐시 교체 정책으로 LRU (Leaset Recently Used)[6]를 사용하는 대신 히스토리 레코드의 기준이 되는 트리거 블록에 우선순위를 부여하여 효율적인 캐시 교체를 가능하게 하는 명령어 및 데이터 프리페치 기법을 제안한다.

본 논문은 다음과 같이 구성된다. Ⅱ장에서는 S. Somogyi가 제안한 데이터 프리페치 기법[7]과 M. Ferdman이 제안한 명령어 프리페치 기법[8]에 대해 알아보고, Ⅲ장에서는 제안하는 히스토리 레코드 생성 및 프리페처의 동작을 설명한다. Ⅳ장에서는 제안한 하드웨어 프리페처의 성능을 검증하기 위한 시뮬레이션 환경 설정과제안하는 방법의 성능 분석을 기술한다. Ⅴ장에서는 본 논문의 연구 내용을 요약하고 최종 결론을 맺는다.

Ⅱ. 하드웨어에 의한 프리페치 기법

2.1 Spatial Memory Streaming

Spatial Memory Streaming (SMS)은 공간 지역성을 활용하기 위해 데이터 스트림의 히스토리를 비트로 표현하여 벡터로 저장하는 하드웨어 프리페치 기법이다. 학습 시간(learning time) 과정을 통해 공간 지역성을 보이는 다수의 캐시 블록들이 비트로 저장되며 이 비트들의 집합인 공간영역(spatial region)을 만든다. 공간영역 내 속하는 임의의 캐시 블록이 데이터 캐시에서 무효화되거나 제거되었을 때 학습 시간 과정은 끝나게 된다. SMS는 필터 테이블(filter table), 누적 테이블(accumulation table), 패턴 히스토리 테이블(pattern history table) 등 3가지 하드웨어 테이블을 사용한다. 필터 테이블은 동일한 공간영역을 생성하지 않기 위해 중복되는 공간영역을 검사하며, 누적 테이블은 공간영역을 생성하기 위한 테이블로서 태그, PC/offset, 패턴 필드로 구성된다. 패턴 히스토리 테이블은 누적 테이블에서 생성한 공간영역 정보를 저장하는 버퍼 역할을 한다.

SMS는 공간적 지역성의 이점을 사용하기 위해 공간영역의 시작 주소와 인접한 데이터 주소를 비트로 표현하여 공간영역 만든다. 데이터 주소의 상위 비트가 공간 영역을 생성하기 위한 태그 값으로 태그 필드에 저장되고, 상위 비트를 제외한 나머지 하위 비트는 오프셋 필드에 저장된다. 그리고 Load/Store의 프로그램 카운터 값이 PC 필드에 저장된다. 필터 테이블에 동일한 태그 값이 있으면 동일한 공간영역이 있다는 것을 의미하기 때문에 누적 테이블에서 해당 공간영역을 찾아 새로운 정보를 업데이트한다. 동일한 공간영역이 없으면 새로운 공간영역의 태그 값, PC 값, 오프셋 값이 필터 테이블에 저장된다. 공간영역의 범위는 패턴 필드에 저장된 비트벡터의 길이에 따라 결정된다. 공간영역을 생

성하는 동안 공간 영역의 범위에 있는 블록이 L1 데이터 캐시에서 방출되거나 무효화되면, 해당하는 공간영역의 PC/offset, 비트 벡터 정보는 패턴 히스토리 테이블에 저장된다.

2.2 Proactive Instruction Fetch

Proactive Instruction Fetch (PIF)은 순차적인 접근 패턴을 보이는 명령어의 특성을 이용해 명령어 스트림의 순서를 비트로 저장하는 하드웨어 프리페치 기법이다. PIF는 압축기(compactor), 히스토리 버퍼(history buffer), 스트림 어드레스 버퍼(stream address buffer), 인덱스 테이블(index table)필터 등 4가지 하드웨어 테이블로 구성된다.

압축기는 완료된 명령어(retired instruction)의 주소를 개별적으로 저장하지 않고 동일한 명령어 블록에 속하는 연속적인 PC값을 하나의 주소로 축약해 공간영역을 만든다. 압축기는 시간적 압축(temporal compaction) 단계와 공간적 압축(spatial compaction) 단계로 구성된다. 명령어 블록 간공간적 지역성의 이점을 얻기 위해 공간적 압축기는 인접한 명령어 블록의 집합인 공간영역 내에 명령어 블록 주소를 모은다. 공간영역에서 트리거라고 부르는 첫 번째 액세스 명령어 블록에 따라 공간 압축기는 공간영역의 범위를 정의한다. 새로운 공간영역은 트리거를 기준으로 이전의 N 블록과 이후의 M 블록으로 총 N + M + 1의 명령어 블록으로 구성된다. 압축기는 새로운 공간영역의 경계를 정의하는 트리거 PC를 기록하고, 각 비트 자리가 공간영역 내의 명령어 블록을 나타내는 비트 벡터를 초기화한다. 현재 공간영역 내의 명령어로서 해당 비트는 비트 벡터에서 설정된다. 현재 공간영역의 범위 밖에 있는 명령어가 완료된다면 현재 트리거 PC값과 비트벡터를 기록하고 시간적 압축기로 보낸다. 그리고 공간적 압축기는 새로운 공간영역 내의 블록들을 감시하기 시작한다.

특정한 코드를 무수히 많이 반복하는 코드에서는 일반적으로 몇 개의 영역에 걸쳐 공간적 루프 명령 풋 프린트, 실행 된 명령의 많은 부분을 구성한다. 이상적으로는 명령어 프리페치 기법을 사용했을 경우, 루프의 모든 명령어 블록은 첫 루프를 반복하기 전에 L1 명령어 캐시로 프리페치된다. 명령어 블록이 이미 캐시에 저장되어있기 때문에 후속으로 반복하는 명령어를 예측해도 이점은 없다. 이러한 첫 루프를 실행 후에 반복하는 공간영역을 기록하는 중복을 피하기 위해 시간적 압축기는 가장 최근에 관찰 된 공간영역 레코드의 개수를 추적한다.

히스토리 버퍼는 FIFO 순으로 완료된 명령들의 순서를 저장하는 순환 버퍼이다. 각각의 저장위치에 트리거 명령어의 블록 주소와 인접한 블록들을 저장한다. 인덱스 테이블에 저장된 트리거 PC값은 맵핑된 내용을 이용해 가장 최근에 히스토리 버퍼에 기록된 내용을 가리키고 있기 때문에 히스토리 버퍼에 저장된 내용을 빠르게 검색 할 수 있다. 스트림 어드레스 버퍼는 히스토리 테이블에 있는 시퀀스를 가리키는 포인터를 유지한다. 이 포인터는 초기에 인덱스 테이블에서 설정된다. 각각의 레코드에 대해 스트림 어드레스 버퍼는 비트벡터로 인코딩된 명령어 블록들 주소를 계산하고 이 주소들에 대해 프리페치를 한다.

Ⅲ. 제안하는 프리페치 기법

이 장에서는 본 논문에서 제안하는 하드웨어 프리페처에 대해 설명한다. 제안한 하드웨어 프리페처는 프로세서가 접근한 프로그램의 순서 정보를 기록하는 히스토리 생성기(history generator)와 히스토리 생성 단계에서 기록한 정보들을 저장하는 히스토리 버퍼(history buffer)를 제안한다.

3.1 제안하는 하드웨어 프리페치 기법의 개요

제안하는 하드웨어 프리페처는 프로세서가 접근한 프로그램의 순서를 히스토리 레코드로 저장한다. 히스토리를 생성하는 과정에서 히스토리 레코드의 첫 블록 주소를 트리거 블록이라 하며 트리거 블록과 그 다음으로 미스가 발생한 블록 주소의 차이 값을 변위 필드에 저장한다. 공간영역을 나타내기 위해 변위 필드의 비트 수에 따라 저장할 수 있는 범위는 조절 가능하다. 또한 히스토리 레코드의 첫 블록 주소인 트리거 블록에 우선권을 부여하여 캐시 교체가 요청되었을 경우 우선권이 없는 블록을 먼저 교체하는 방법을 제안한다.

3.2 변위 필드가 나타낼 수 있는 블록 주소의 범위

프로세서는 데이터를 로드 할 때 해당 데이터의 인접한 데이터도 참조 할 가능성이 크다. 캐시는 공간적 지역성을 이용해 메모리에서 데이터를 로드 할 때 주변

데이터들까지 같이 로드하는데 캐시와 메모리 사이에는 블록 단위로 데이터를 교환한다. 히스토리 생성기의 변위필드에는 히스토리 레코드의 시작 블록인 트리거 블록과 트리거 블록 이후에 캐시 미스가 발생한 블록들의 주소 차이가 저장된다.

표 1은 제안하는 히스토리 생성기의 변위 필드가 비트 수에 따라 저장 할 수 있는 블록 주소의 범위를 나타낸다. 비트 수가 커질수록 두 블록의 간격 차이가 큰 블록을 저장한다. 하지만 많은 수의 변위를 저장하기 위해서는 변위 필드의 크기가 커지기 때문에 스토리지 오버헤드가 발생 할 수 있다. 반면 변위필드의 비트 수가 작을수록 트리거 주소를 기준으로 매우 인접한 블록만 변위 필드에 저장할 수 있다.

본 논문에서는 변위 필드를 4비트와 8비트로 설정한 하드웨어 프리페처를 구현하였으며. 히스토리 생성 단계에서 만들어지는 변위 필드의 전체 크기를 32비트로 설정했다. 변위 필드를 4비트로 설정하면 총 8개의 변위 필드로 구성된 히스토리 레코드가 만들어지고 변위 필드를 8비트로 설정하면 총 4개의 변위 필드로 구성된 히스토리 레코드가 만들어진다.

표 1. 비트 수에 따라 변위 필드가 나타낼 수 있는 블록 주소의 범위
Table 1. Depending on the number of bits that can represent the displacement field of the block address range.

Bits	Coverage	
	negative	positive
2	-2	+1
3	-3	+4
4	-8	+7
5	-16	+15
6	-32	+31
7	-64	+63
8	-128	+127

3.3 변위를 이용한 히스토리 레코드 생성 단계

히스토리 생성기는 캐시 미스가 발생한 주소들을 기록하여 히스토리를 만든 뒤, 프로세서가 다시 이 주소를 필요로 할 때 캐시 블록을 프리페치한다. 하나의 캐시블록에서 미스가 발생할 때마다 프리페치 요청을 하지 않고 미스가 발생한 캐시 블록들을 모아 공간적 지역성을 활용한 레코드를 만들어 프로그램의 히스토리를 만든다. 히스토리 생성기의 테이블은 히스토리 레코드의 첫 시작 블록 주소를 저장하는 트리거 주소(trigger address), 블록들 간의 주소 차이를 저장하는 변위(displacement) 등 2개의 필드로 구성되어있다.

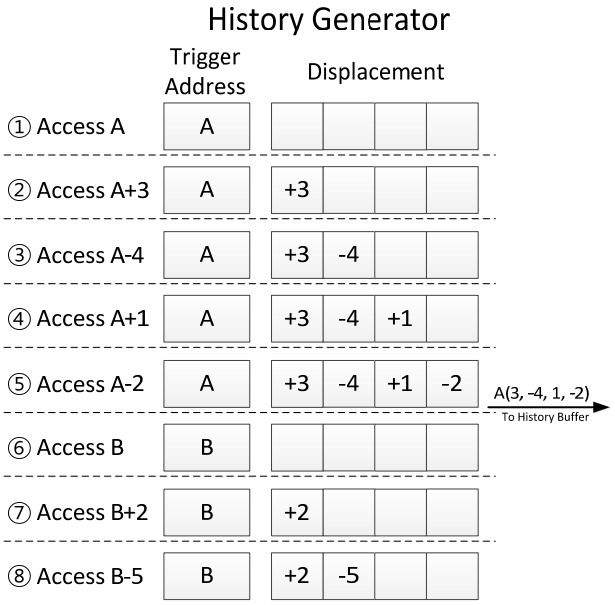

그림 1. 변위를 이용한 히스토리 레코드 생성 단계
Fig. 1. The history record generation using displacement.

그림 1은 히스토리 생성기의 자세한 동작의 예를 보여준다. 블록 A에서 캐시 미스가 발생하면, 히스토리 생성기의 트리거 주소 필드를 검색한다. 트리거 주소 필드에 저장된 값이 없다면 블록 A의 주소를 저장한다. 블록 A의 주소를 저장한다. 블록 A의 주소는 새로운 레코드의 기준이 되는 트리거 주소가 된다. 이후 블록 A+3에서 캐시 미스가 발생하면 트리거 주소 필드에 저장되어있는 트리거 블록 주소와의 차이 +3을 변위 필드에 저장한다. 이후에 미스가 발생한 블록들이 A-4, A+1, A-2이면 트리거 주소와의 차이를 변위 필드에 순서대로 -4, +1, -2를 각각의 변위 필드에 저장한다. 히스토리 레코드를 생성하는 과정에서 변위 필드의 엔트리에 변위 값들이 모두 저장이 되거나 변위 필드에 저장할 수 없는 범위의 블록이 발생한다면 히스토리 레코드의 생성을 중단하고 해당 엔트리를 히스토리 버퍼에 저장한다.

트리거 주소 블록 A에 대한 레코드를 생성하다가 변위 필드 범위 밖의 블록 B에서 캐시 미스가 발생한다

면, 이전까지 기록해놓은 트리거 주소 블록 A와 변위 값들을 히스토리 버퍼에 저장하고, 히스토리 생성기의 트리거 주소 필드와 변위 필드에 저장된 값들을 모두 초기화한다. 이후 블록 B에 대한 레코드를 생성하기 위해 트리거 주소 필드에 블록 B의 주소를 저장한다. 이후에 블록 B+2에서 캐시 미스가 발생하면 트리거 주소 필드에 저장되어있는 블록 B의 주소와의 차이를 변위 필드에 저장한다. 다음으로 블록 B-5에서 미스가 캐시 미스가 발생하면 블록 B의 주소와의 차이를 변위 필드에 저장한다.

3.4 히스토리 레코드를 이용한 예측 단계

히스토리 버퍼는 트리거 주소 필드와 변위 필드로 구성되며, 각각의 엔트리에는 히스토리 생성 단계에서 만든 히스토리 레코드가 저장된다. 트리거 액세스 필드는 히스토리 버퍼의 인덱스 역할을 한다.

그림 2는 저장된 히스토리 레코드와 트리거 액세스 정보가 일치하였을 때 동작을 나타낸다. 트리거 액세스에 일치하는 엔트리가 히스토리 버퍼에 존재하면 블록 A의 주소를 제일 먼저 프리페치한 후 변위 필드에 저장되어 있는 변위 값을 저장된 순서에 따라 블록 A의 주소 값에 더해 블록 A+3, A-4, A+1, A-2를 프리페치한다.

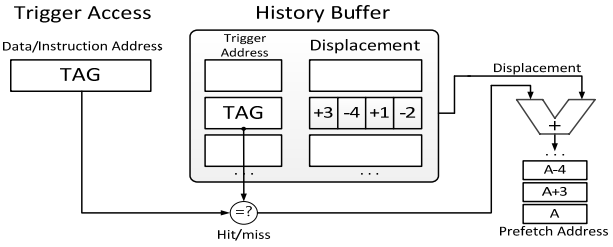

그림 2. 히스토리 버퍼와 예측 과정
Fig. 2. History Buffer and prediction process.

3.5 캐시 교체 정책의 최적화

캐시에서 캐시 적중률을 높이는데 캐시 교체 정책이 중요한 역할을 한다. Direct Mapped 캐시에서는 캐시 교체 정책을 고민할 필요가 없지만 Set-Associative 캐시에서는 최대 N개 중 하나의 캐시 블록을 제거해야하기 때문에 어떤 교체 정책을 사용할지 고민할 필요가 있다. 가장 단순한 방법에는 임의로 캐시 블록을 교체하는 Random 교체 정책과 가장 오래된 캐시 블록을 교체하는 LRU(Least Recently Used) 교체 정책이 있다.

LRU 교체 정책은 최근에 참조된 캐시 블록을 순서대로 비교할 수 있도록 LRU 스택을 개별 캐시 세트에 할당한다. 캐시 미스가 발생하면 LRU 스택을 살펴보고 가장 오래 사용되지 않은 블록을 교체하는 방법이다. 캐시 블록마다 타임스탬프를 기록하기 위해 N-way Set-Associative 캐시일 때, $Nlog2^N$-bit 디멀티플렉서 1개, $2log2^N$-bit 멀티플렉서 1개, N-bit 디멀티플렉서 1개, $Nlog2^N$-bit 멀티플렉서 1개, $Nlog2^N$-bit 비교기, $log2^N$-bit 레지스터 N개, N-bit 우선순위 인코더 1개가 필요하다. 따라서 캐시의 Set-Associativity가 높아질수록 하드웨어 복잡도가 높아진다.

제안한 트리거 블록을 기준으로 변위를 저장하는 방법에서 트리거 블록이 먼저 캐시에 저장되고 뒤이어 변위 값에 해당하는 블록들이 순서대로 저장된다. 따라서 LRU 교체 정책을 사용하면 캐시 공간이 가득 찼을 때 트리거 블록이 교체될 확률이 높다. 트리거 블록이 교체된 후, 프로세서가 교체된 트리거 블록을 필요로 했을 때 히스토리 버퍼에서 해당하는 히스토리 레코드를 찾아 트리거 블록과 변위 블록들을 프리페치한다. 따라서 캐시에는 동일한 변위 블록들이 중복되어 저장될 수 있다. LRU 교체정책을 사용했을 때의 단점을 보완하기 위해 교체할 블록을 임의로 선택하는 Random 교체 정

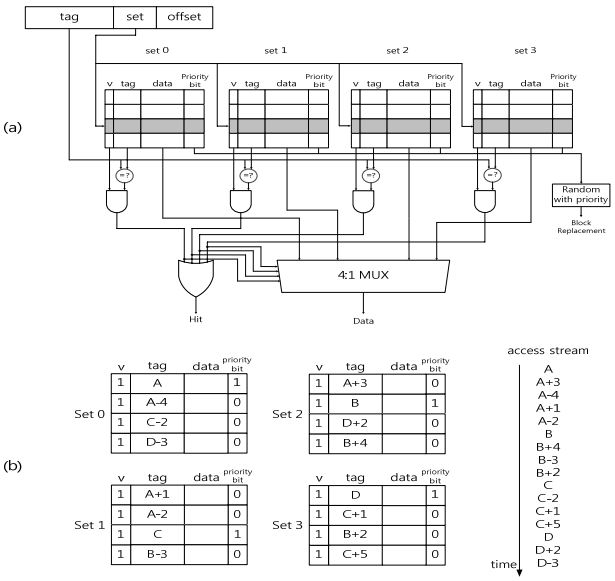

그림 3. 트리거 블록에 우선순위를 부여하는 Random 교체 정책의 동작
Fig. 3. An illustration of Random replacement policy with priority to trigger block.

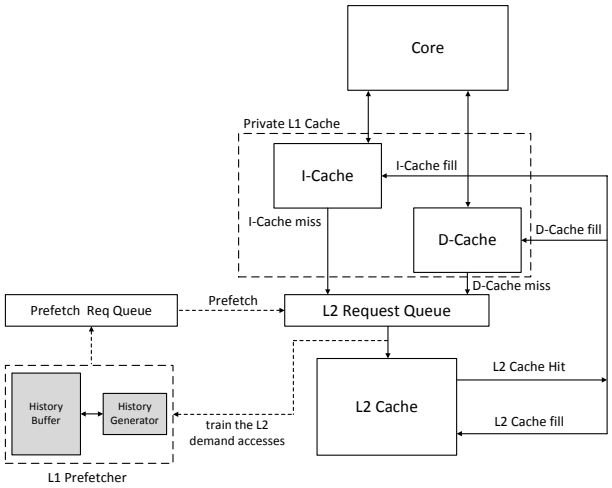

그림 4. 전체 시스템 구성도
Fig. 4. The full system block diagram.

책에 트리거 블록에 우선순위를 부여하여 히스토리 레코드의 변위 블록부터 교체하는 방법을 제안한다.

그림 3은 트리거 블록에 우선순위를 부여하고 히스토리 레코드의 변위 블록부터 교체하는 Random 교체 정책을 나타낸다. LRU 교체 정책과 달리 각각의 블록마다 타임스탬프나 접근 시간을 기록하지 않고 히스토리 레코드의 트리거 블록에만 우선순위를 기록한다. 액세스 스트림 순서대로 캐시에 저장되어 캐시 공간이 가득찼을 때, 가장 오래 사용되지 않은 블록은 블록 A이다. 하지만 블록 A는 우선순위비트 값이 1인 트리거 블록이기 때문에 블록 A를 제외한 우선순위 비트가 0인 나머지 블록들 중 임의의 블록이 새로운 블록으로 교체된다. 캐시 공간에 우선순위 비트가 0인 블록이 다 교체되고 1인 블록만 남아있다면 트리거 블록 중 임의의 블록을 제거하고 새로운 블록으로 교체한다. 트리거 블록에 우선순위를 부여하는 Random 교체 정책은 N-way set-associative 캐시에서 임의의 블록을 선택하기 위한 N개의 선형 피드백 시프트 레지스터와 트리거 블록을 구별하는 N개의 비교기가 필요하다. LRU에 비해 하드웨어 구조가 간단하며 구현하기 쉽다. 그림 4는 제안한 하드웨어 프리페처의 전체 시스템 개요를 나타낸다.

Ⅳ. 성능 분석

이 장에서는 본 논문에서 제안한 프리페처에 대해 성능을 분석하고 비교한다. 제안하는 프리페처를 검증하기 위해서 gem5 simulator를 이용해 구현하였고, 벤치마크 프로그램으로는 PARSEC (Princeton Application Repository for Shared-Memory Computers)[9]를 사용하여 성능을 분석하였다. 기존의 제안된 프리페처와 제안한 프리페처의 L1 명령어 및 데이터 캐시와 IPC (Instruction per Cycle)을 비교했다. Set-Associative 변화에 따른 L1 명령어 및 데이터 캐시 미스율의 변화도 확인하였다.

4.1 시뮬레이션 환경

제안한 프리페처의 성능을 평가하기 위해 표 2는 실험에 사용한 시스템 구성 환경을 보여준다. ALPHA ISA (Instruction Set Architecture)를 사용했고, 시뮬레이션으로 약 10억 개의 명령어를 실행하였다. L1 캐시는 private L1 캐시로 L1-D캐시와 L1-I캐시의 크기를 각각 32kB를 사용하였고 L2 캐시는 shared L2 캐시로 크기를 2MB로 고정하였다. L1과 L2 캐시 블록의 크기는 64바이트이다. 분기 예측기는 토너먼트 분기 예측기를 사용하였다. 다양한 환경에서 L1 캐시의 성능을 평가하기 위해 크기, Set-Associative의 크기와 L1 캐시의 크기를 16kB, 8kB로 바꿔가며 실험을 하였다.

PARSEC은 인텔과 프린스턴 대학에서 만들었으며, 최신 프로그램의 경향을 반영하고 다양한 workload를 지원하여 칩 멀티 프로세서를 평가할 수 있는 벤치마크이다. 칩 멀티 프로세서의 성능을 평가하기 위해 다양

표 2. 시스템 구성 환경
Table 2. System Configuration.

Core	ALPHA ISA Quad core, 3.4GHz, OoO 6-Wide Dispatch 8-Wide Commit 40-entry ROB 16-entry RAS 32-entry Issue Queue
I-Cache	32kB, 64B 1, 2, 4, 8-way
D-Cache	32kB, 2-way, 64B 1, 2, 4, 8-way
L2-Cache	2MB, 8-way, 64B 12ns hit latency
Main Memory	2GB, 54ns access latency

표 3. PARSEC 프로그램의 특성
Table 3. Characteristics of the PARSEC program.

Program		Application Domain	Problem Size	Working Set
INT	dedup	Enterprise Storage	184 MB data	unbounded
	freqmine	Data Mining	990,000 transactions	unbounded
	x264	Media Processing	128 frames, 640 × 360 pixels	medium
FP	blackscholes	Financial Analysis	65,535 options	small
	bodytrack	Computer Vision	4 frames, 4,000 particles	medium
	canneal	Engineering	400,000 elements	unbounded
	ferret	Similarity Search	256 queries, 34,973 images	unbounded
	fluidanimate	Animation	5frames, 300,000 transactions	large
	streamcluster	Data Mining	16,384 points per block, 1 block	medium
	swaptions	Financial Analysis	64 swaptions, 20,000 simulations	medium

한 크기의 Input Set을 제공한다. 표 3은 본 논문에서 사용한 PARSEC 프로그램의 특성을 보여준다. blackscholes와 swaptions는 재무분석, bodytrack은 컴퓨터 비전, canneal은 엔지니어링, dedup은 엔터프라이즈 스토리지, ferret은 이미지 검색, fluidanimate는 애니메이션, freqmine과 streamcluster는 데이터 마이닝, x264는 이미지 처리 관련 프로그램이다.

4.2 시뮬레이션 결과 및 분석

4.2.1 L1 캐시의 미스율

이번 절에서는 PARSEC 벤치마크 프로그램 별로 L1 데이터 캐시와 명령어 캐시의 미스율을 나타냈다. 세로축은 미스율을 나타내며, 가로축은 각각의 벤치마크 프로그램의 명칭이다. dedup, freqmine, x264 등 3개의 어플리케이션은 정수형 프로그램이며, blackscholes, bodytrack, canneal, ferret, fluidanimate, streamcluster, swaption 등 7개의 어플리케이션은 실수형 프로그램이다.

모든 하드웨어 프리페처는 2K의 엔트리를 가지며, 각 프로그램 별로 세 개의 막대로 표시된 미스율이 있다. 첫 번째 막대의 크기는 기존에 제안 되었던 프리페처의 미스율이며, 두 번째와 세 번째 막대의 크기는 제안한 프리페치의 미스율을 나타낸다. 두 번째 막대는

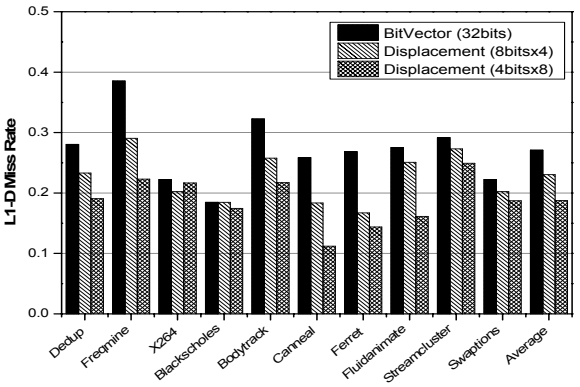

그림 5. L1 데이터 캐시 미스율 비교
Fig. 5. L1 data cache miss rate comparison.

변위 필드의 한 영역의 크기가 8비트이며, 세 번째 막대는 변위 필드의 한 영역의 크기가 4비트인 프리페처를 나타낸다.

그림 5는 L1 데이터 캐시의 미스율을 나타낸 그래프이다. BitVector(32bits)는 기존에 제안된 SMS를 나타내며, Displacement(8bitsx4)와 Displacement(4bitsx8)는 제안된 프리페처를 나타낸다. SMS의 경우 평균 약 27.1%의 미스율을 보였다. 변위 필드가 8비트인 제안된 프리페처의 경우 평균 약 23%, 변위 필드를 4비트로 설정한 제안된 프리페처의 경우 평균 약 18.7%의 미스율을 보였다. SMS와 비교했을 때 제안된 프리페처의 L1 데이터 미스율은 변위 필드의 크기를 8비트로 고정했을

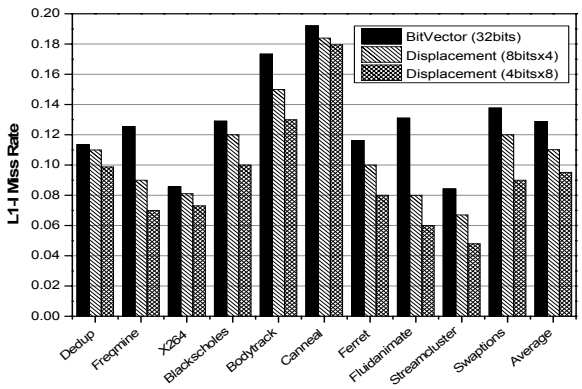

그림 6. L1 명령어 캐시 미스율 비교
Fig. 6. L1 instruction cache miss rate comparison.

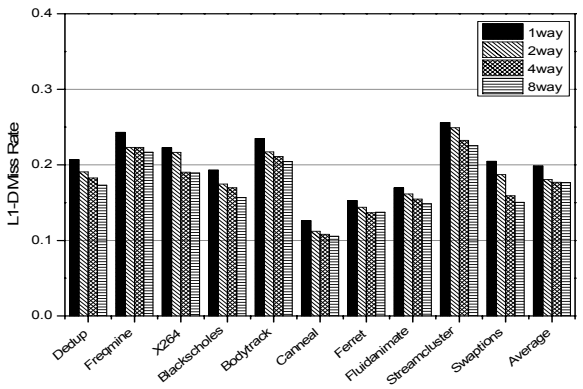

그림 7. Set-Associativity에 따른 L1 데이터 캐시 미스율 비교 : 8비트 변위 필드
Fig. 7. L1 data cache miss rates of 10 benchmark programs for each Set-Associativity : 8-bit displacement field.

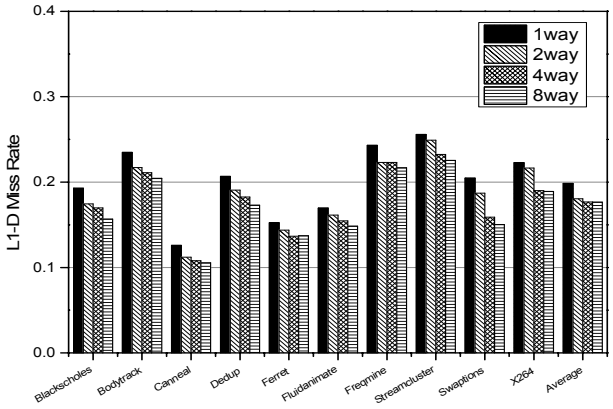

그림 8. Set-Associativity에 따른 L1 데이터 캐시 미스율 비교 : 4비트 변위 필드
Fig. 8. L1 data cache miss rates of 10 benchmark programs for each Set-Associativity : 4-bit displacement field.

경우 약 15% 감소하였고, 4비트로 고정했을 경우 약 44.5% 감소하였다. 4비트 변위 필드를 사용한 프리페처의 방법을 사용했을 경우 L1 데이터 캐시의 미스율이 가장 낮은 결과를 보였다.

그림 6은 L1 명령어 캐시의 미스율을 나타낸 그래프이다. BitVector(32bits)는 기존에 제안된 PIF를 나타내며 평균 약 12.8%의 미스율을 보였다. 변위 필드를 8비트로 설정한 제안된 프리페처의 경우 평균 약 11%, 변위 필드를 4비트로 설정한 제안된 프리페처의 경우 평균 약 9.5%의 미스율을 보였다. PIF와 비교했을 때 제안된 프리페처의 명령어 미스율은 변위 필드의 크기를 8비트로 고정했을 경우 약 14.3% 감소하였고, 4비트로 고정했을 경우 약 26.1% 감소하였다. L1 명령어 캐시도 4비트 변위 필드를 사용한 프리페치의 미스율이 가장 낮은 결과를 보였다.

명령어 액세스는 순차적이거나 반복적인 패턴을 보이는 반면에 데이터 액세스는 순회적이거나 다양한 패턴을 보인다. 제안하는 프리페처를 사용했을 때, 명령어 캐시의 미스율은 평균 31% 감소하였고 데이터 캐시의 미스율은 평균 44.5% 감소하여 데이터 캐시에서 제안된 프리페처가 더 효율적임을 확인하였다.

4.2.2 Set-Associativity에 따른 L1 캐시의 미스율 변화

그림 7에서 그림 10은 제안된 프리페처의 Set-Associativity의 크기 변화에 따른 L1 데이터 캐시와 명령어 캐시의 미스율 변화를 비교한 것이다. Set-Associativity의 크기는 1-way부터 8-way까지 바꿔가며 실험하였다.

그림 7은 변위 필드의 크기를 8비트로 설정하였으며 4개의 변위 필드가 하나의 히스토리 레코드를 이룬다. Set-Associativity를 1-way로 설정했을 때 L1 데이터 캐시의 미스율은 평균 약 25.3%, 2-way로 설정했을 때 평균 약 23%, 4-way로 설정했을 때 평균 약 21.8%, 8-way로 설정했을 때 평균 약 18.3% 미스율을 나타냈다.

그림 8은 변위 필드의 크기를 4비트로 설정하였으며 8개의 변위 필드가 하나의 히스토리 레코드를 이룬다. Set-Associativity를 1-way로 설정했을 때 L1 데이터 캐시의 미스율은 평균 약 19.8%, 2-way로 설정했을 때 평균 약 18%, 4-way로 설정했을 때 평균 약

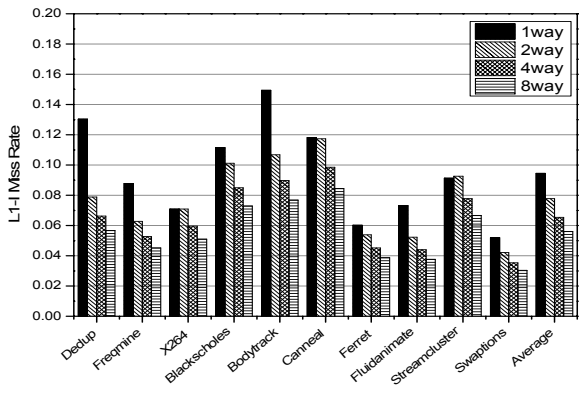

그림 9. Set-Associativity에 따른 L1 명령어 캐시 미스율 비교 : 8비트 변위 필드

Fig. 9. L1 instruction cache miss rates of 10 benchmark programs for each Set-Associativity : 8-bit displacement field.

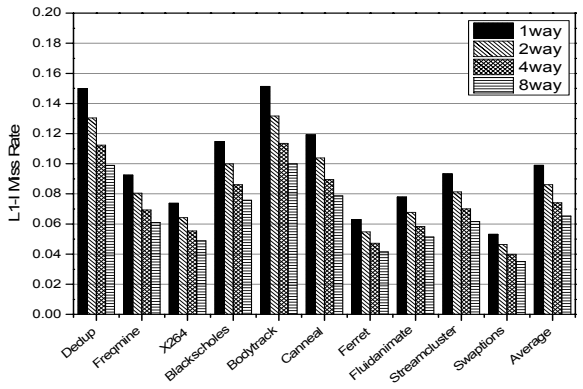

그림 10. Set-Associativity에 따른 L1 명령어 캐시 미스율 비교 : 4비트 변위 필드

Fig. 10. L1 instruction cache miss rates of 10 benchmark programs for each Set-Associativity : 4-bit displacement field.

17.6%, 8-way로 설정했을 때 평균 약 17.1% 미스율을 나타냈다.

그림 9는 변위 필드의 크기를 8비트로 설정하였으며 4개의 변위 필드가 하나의 히스토리 레코드를 이룬다. Set-Associativity를 1-way로 설정했을 때 L1 명령어 캐시의 미스율은 평균 약 9.8%, 2-way로 설정했을 때 평균 약 8.6%, 4-way로 설정했을 때 평균 약 7.4%, 8-way로 설정했을 때 평균 약 6.5% 미스율을 나타냈다.

그림 10은 변위 필드의 크기를 4비트로 설정하였으며 8개의 변위 필드가 하나의 히스토리 레코드를 이룬다. Set-Associativity를 1-way로 설정했을 때 L1 명령어 캐시의 미스율은 평균 약 9.4%, 2-way로 설정했을 때 평균 약 7.7%, 4-way로 설정했을 때 평균 약 6.5%, 8-way로 설정했을 때 평균 약 5.6% 미스율을 나타냈다. 평균적으로 8-way일 때, L1 데이터 및 명령어 캐시의 미스율이 가장 낮았다.

4.2.3 캐시 교체 정책에 따른 L1 캐시의 미스율

이번 절에서는 캐시 교체 정책에 따른 L1 캐시 미스율의 변화에 대해 알아본다. LRU replacement는 하드웨어 프리페처를 적용하지 않고 LRU교체 정책을 사용한 경우이며, Random with priority는 4비트 변위 필드를 사용하는 제안된 프리페처를 사용하고 트리거 블록에 우선순위를 가지게 하는 Random 교체 정책을 사용한 경우이다. 그림 11부터 그림 16까지 L1 데이터와 명령어 캐시의 크기를 각각 8kB, 16kB, 32kB로 설정했을 때 L1 데이터와 명령어 캐시의 미스율을 나타낸 그래프이다.

그림 11은 데이터 캐시의 크기를 8kB로 설정하여 벤치마크를 수행한 결과이다. 프리페처를 적용하지 않고 LRU 교체 정책을 사용한 경우 평균 약 74% 미스율을 보였으며, 10개 프로그램 모두 높은 미스율을 보였다. 제안된 프리페처에 우선순위 Random 교체 정책을 사용한 경우 21% 미스율을 보였으며, LRU 교체 정책과 비교해 제안한 우선순위 Random 교체 정책을 사용했을 경우 미스율은 약 70.6% 감소하였다.

그림 12는 데이터 캐시의 크기를 16kB로 설정하여 벤치마크를 수행한 결과이다. 프리페처를 적용하지 않

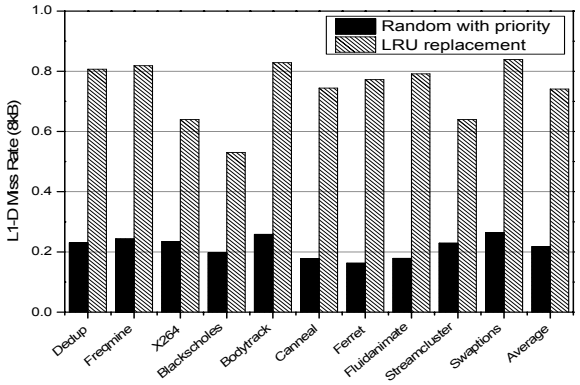

그림 11. 캐시 교체 정책에 따른 L1 데이터 캐시의 미스율 비교 : 캐시 크기 8kB

Fig. 11. In accordance with cache replacement policy, comparison of the L1 data cache miss rates : cache size 8kB.

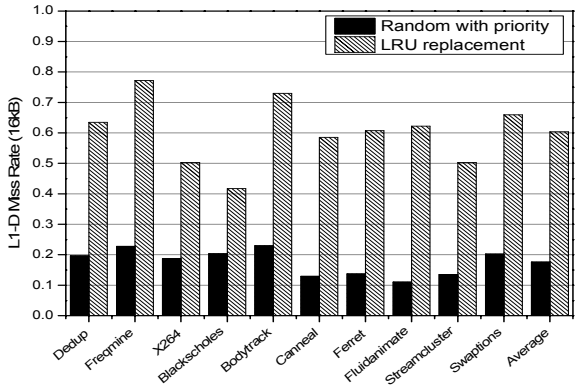

그림 12. 캐시 교체 정책에 따른 L1 데이터 캐시의 미스율 비교 : 캐시 크기 16kB

Fig. 12. In accordance with cache replacement policy, comparison of the L1 data cache miss rates : cache size 16kB.

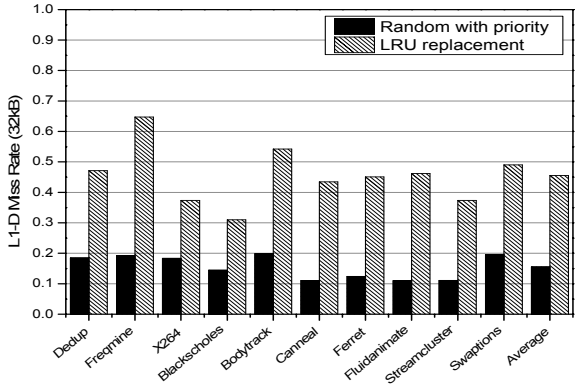

그림 13. 캐시 교체 정책에 따른 L1 데이터 캐시의 미스율 비교 : 캐시 크기 32kB

Fig. 13. In accordance with cache replacement policy, comparison of the L1 data cache miss rates : cache size 32kB.

고 LRU 교체 정책을 사용한 경우 평균 약 74% 미스율을 보였다. 제안된 프리페처에 우선순위 Random 교체 정책을 사용한 경우 17.5% 미스율을 보였으며, LRU 교체 정책과 비교해 제안된 우선순위 Random 교체 정책을 사용했을 경우 미스율은 약 70.8% 감소하였다.

그림 13은 데이터 캐시의 크기를 32kB로 설정하여 벤치마크를 수행한 결과이다. 프리페처를 적용하지 않고 LRU 교체 정책을 사용한 경우 평균 약 45.5% 미스율을 보였으며, 10개 프로그램 모두 높은 미스율을 보였다. 제안된 프리페처에 우선순위 Random 교체 정책을 사용한 경우 15.5% 미스율을 보였으며, LRU 교체 정책과 비교해 미스율은 약 65.8% 감소되었다. L1 데이

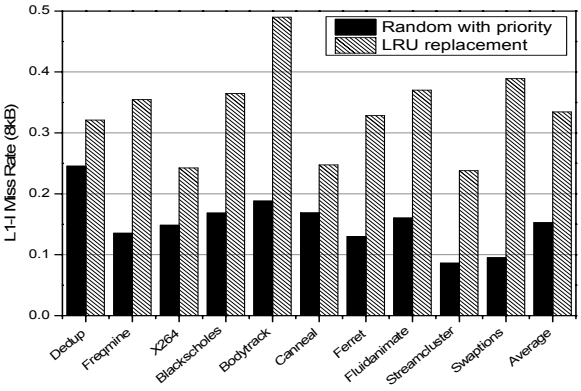

그림 14. 캐시 교체 정책에 따른 L1 명령어 캐시의 미스율 비교 : 캐시 크기 8kB

Fig. 14. In accordance with cache replacement policy, comparison of the L1 instruction cache miss rates : cache size 8kB.

터 캐시의 크기를 변경하며 미스율을 확인해본 결과, 캐시 크기를 크게 할수록 L1 데이터 캐시의 미스율은 낮아졌고, 우선순위 Random 교체 정책을 사용한 제안된 프리페처에서는 LRU를 적용한 프리페처를 사용하지 않는 방법보다 L1 데이터 미스율이 더 낮은 결과를 보였다.

그림 14는 명령어 캐시의 크기를 8kB로 설정하여 벤치마크를 수행한 결과이다. 프리페처를 적용하지 않고 LRU 교체 정책을 사용한 경우 평균 약 33.5% 미스율을 보였다. 제안된 프리페처에 우선순위 Random 교체 정책을 사용한 경우 15.2% 미스율을 보였으며, LRU 교체 정책과 우선순위 Random 교체 정책을 비교해 약 54.3% 미스율이 감소되었다.

그림 15는 명령어 캐시의 크기를 16kB로 설정하여 벤치마크를 수행한 결과이다. 프리페처를 적용하지 않고 LRU 교체 정책을 사용한 경우 평균 약 26% 미스율을 보였으며,제안된 프리페처에 우선순위 Random 교체 정책을 사용한 경우 평균 약 9.9% 미스율을 보였다. LRU 교체 정책과 우선순위 Random 교체 정책을 비교해 약 61.8% 미스율이 감소되었다.

그림 16은 명령어 캐시의 크기를 32kB로 설정하여 벤치마크를 수행한 결과이다. 프리페처를 적용하지 않고 LRU 교체 정책을 사용한 경우 평균 약 19.8% 미스율을 보였으며, 제안된 프리페처에 우선순위 Random 교체 정책을 사용한 경우 평균 약 7.7% 미스율을 보였다. LRU 교체 정책과 우선순위 Random 교체 정책을

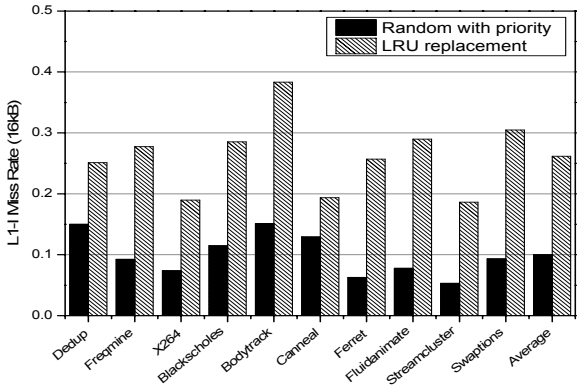

그림 15. 캐시 교체 정책에 따른 L1 명령어 캐시의 미스율 비교 : 캐시 크기 16kB

Fig. 15. In accordance with cache replacement policy, comparison of the L1 instruction cache miss rates : cache size 16kB.

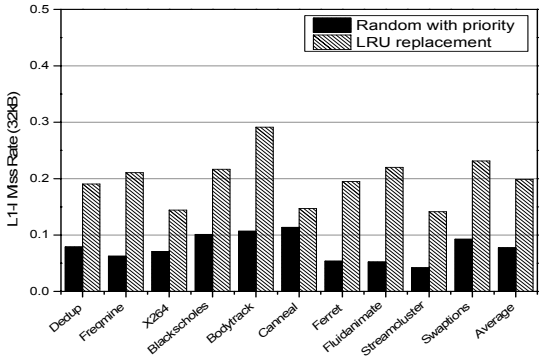

그림 16. 캐시 교체 정책에 따른 L1 명령어 캐시의 미스율 비교 : 캐시 크기 32kB

Fig. 16. In accordance with cache replacement policy, comparison of the L1 instruction cache miss rates : cache size 32kB.

교해 약 61% 미스율이 감소되었다. L1 명령어 캐시의 크기를 변경하며 미스율을 확인해본 결과, 우선순위 Random 교체 정책을 사용한 제안된 프리페처가 LRU를 적용한 프리페처를 사용하지 않는 방법보다 미스율이 더 낮은 결과를 보임을 확인했다.

4.2.4 제안된 프리페치의 IPC Speedup

그림 17은 IPC(Instruction per Cycle) Speedup을 비교한 결과이다. PIF의 경우는 평균 약 1.19이며 변위 필드를 8비트로 설정한 제안된 프리페처는 평균 약 1.38, 변위 필드를 4비트로 설정한 제안된 프리페처는 평균 약 1.48의 IPC를 나타낸다. 공통적으로 canneal 프로그램에서 3가지 방법 모두 가장 낮은 IPC SpeedUp의 결과를 보였다. 이는 canneal은 칩 디자인을 할 때 라우팅 비용을 최소로 하기 위해 수도 랜덤(pseudo-random)하게 임의의 입력 데이터를 서로 교환하기 때문이다. 또한 3가지 방법 모두 x264 프로그램에서 가장 높은 IPC SpeedUp의 결과를 보였다. PIF의 방법 대비 변위 필드의 크기를 8비트로 설정한 프리페처의 IPC는 평균 약 15%정도 IPC가 향상되었고, 변위 필드의 크기를 4비트로 설정한 프리페처의 IPC는 평균 약 24%정도 IPC가 향상되었다.

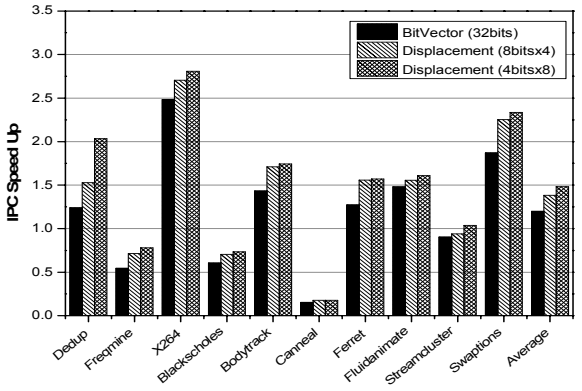

그림 17. IPC SpeedUp 성능 비교

Fig. 17. IPC SpeedUp perfomance comparison.

VI. 결 론

본 논문에서 제안한 하드웨어 프리페처는 변위 필드를 이용해 히스토리 레코드를 생성하는 단계와 히스토리 레코드의 기준이 되는 트리거 블록에 우선순위를 부여하여 효율적인 캐시 교체가 가능한 방법을 사용한다. 히스토리 레코드의 트리거 블록을 기준으로 히스토리를 생성하기 때문에 프로그램의 시퀀스를 고려할 수 있으며, 히스토리를 변위 값으로 저장하기 때문에 트리거 주소와 변위필드에 저장된 값을 더해 빠르게 명령어 및 데이터 주소를 프리페치 할 수 있다. 또한, 트리거 블록에 우선순위를 부여하고 캐시 교체 정책으로 랜덤 교체 방법을 사용해 캐시 공간이 가득 찼을 때 우선순위가 낮은 블록부터 랜덤하게 교체하는 방법을 제안하였다. 블록마다 타임스탬프를 계산하여 저장하는 LRU 교체 정책과 비교해 하드웨어 복잡도가 낮다. 그리고 트리거 블록이 LRU 위치에 있어도 우선순위가 낮은 블록부터 교체되어 동일한 트리거 블록을 액세스했을 때 해당하는 블록만 캐시로 가져오면 되기 때문에 메모리 스톨

시간을 감소시킬 수 있었다.

제안하는 하드웨어 프리페처의 성능을 평가하기 위해 메모리 분석 시뮬레이터인 gem5와 PARSEC 벤치마크 프로그램을 사용하였다. 기존에 제안된 하드웨어 프리페처와 본 논문에서 제안한 하드웨어 프리페처와 비교해 L1 데이터 캐시의 미스율은 평균 약 44.5% 감소하였고 L1 명령어 캐시의 미스율은 평균 약 31% 감소하였다. 또한 IPC(Instruction per Cycle)는 평균 약 23.7% 향상을 보였다.

REFERENCES

[1] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum, “Compiler-directed data prefetching in multiprocessors with memory hierarchies,” *In Proceedings of the 4th International Conference on Supercomputing*, pp. 354-368, Jun. 1990.

[2] C. H. Yu, K. H. Kim, S. B. Choi, “Design of Serial Decimal Multiplier using Simultaneous Multiple-digit Operations,” *Journal of The Institute of Electronics Engineers of korea*, Vol. 52, no. 3, pp. 115-124, Apr. 2015.

[3] I. K. Hwang, K. H. Kim, W. O. Yoon, and S. B. Choi, “Design of Parallel Decimal Multiplier using Limited Range of Signed-Digit Number Encoding,” *Journal of The Institute of Electronics Engineers of korea*, Vol. 50, no. 3, pp. 50-58, Mar. 2013.

[4] J. Pierce and T. Mudge, “Wrong-path instruction prefetching,” *In Proceedings of the 29th International Symposium on Microarchitecture*, pp. 165-175, Dec. 1996.

[5] R. L. Lee, P. C. Yew, and D. H. Lawrie, “Data prefetching in shared memory multiprocessors,” *In Proceedings of International Conference on Parallel Processing*, pp. 28-31, Aug. 1987.

[6] T. S. B. Sudarshan et al, “Highly efficient lru implementations for high associativity cache memory,” *In Proceedings of the 12th International Conference and Expo on advanced Ceramics and Composites*, pp 87-95, Dec. 2004.

[7] S. Somogyi, T. Wenisch, M. Ferdman, and B. Falsafi, “Spatial memory streaming,” *The Journal of Instruction-Level Parallelism*, vol. 13, Jan. 2011.

[8] M. Ferdman, C. Kaynak, and B. Falsa, “Proactive instruction fetch,” *In Proceedings of the 44th International Symposium on Microarchitecture*, pp. 152-162, Dec. 2011.

[9] C. Bienia, S. Kumar, J. P. Jaswinder, and K. Li, “The PARSEC benchmark suite: characterization and architectural implications,” *In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pp. 72-81, Oct. 2008.

저 자 소 개

정 용 수(정회원)
2013년 인하대학교 전자공학과 학사 졸업.
2015년 인하대학교 전자공학과 석사 졸업.
2015년～현재 전자부품연구원 연구원.
<주관심분야 : 컴퓨터 구조, 메모리 시스템>

조 태 환(정회원)
2001년 인하대학교 항공우주 공학과 학사 졸업.
2014년 인하대학교 전자공학과 통합과정 졸업.
2014년～현재 공군사관학교 전자공학과 조교수.
<주관심분야 : 컴퓨터 네트워크, 항공전자시스템, 항공교통관제시스템>

김 진 혁(정회원)
2009년 인하대학교 전자공학과 학사 졸업.
2011년 인하대학교 전자공학과 석사 졸업.
2011년～현재 인하대학교 전자공학과 박사과정
<주관심분야 : 멀티미디어 통신, 무선 통신, 컴퓨터 내트워크, 병렬 및 분산 컴퓨팅>

최 상 방(평생회원)
1981년 한양대학교 전자공학과 학사 졸업.
1981년～1986년 LG 정보통신(주)
1988년 University of washinton 석사 졸업.
1990년 University of washinton 박사 졸업.
1991년～현재 인하대학교 전자공학과 교수.
<주관심분야 : 컴퓨터 구조, 컴퓨터 네트워크, 무선 통신, 병렬 및 분산 처리 시스템>