Journal of The Institute of Electronics and Information Engineers Vol.53, NO.12, December 2016 https://doi.org/10.5573/ieie.2016.53.12.020
ISSN 2287-5026(Print) / ISSN 2288-159X(Online)

논문 2016-53-12-3

다양한 최신 워크로드에 적용 가능한 하드웨어 데이터 프리페처 구현

(Implementation of Hardware Data Prefetcher Adaptable for Various State-of-the-Art Workload)

김 강 희*, 박 태 신*, 송 경 환*, 윤 동 성*, 최 상 방**

(KangHee Kim, TaeShin Park, KyungHwan Song, DongSung Yoon, and SangBang Choi©)

요 약

본 논문에선 병렬 십진 곱셈기의 축약 단계의 면적과 지연시간을 감소시켜 성능을 향상시키기 위해 다중 피연산자 십진 CSA과 개선된 십진 CLA를 이용한 트리 구조를 제안한다. 제안한 부분곱 축약 트리는 십진수 부분곱에 대해 다중 피연산자 십진 CSA를 사용하여 빠르게 부분곱을 축약한다. 각 CSA에서는 리코딩에 입력의 범위를 제한함으로써 가장 간단한 리코더 로직을 얻는다. 그리고 각 CSA는 특정한 아키텍처 트리의 특정한 위치에서 범위가 제한된 십진수를 더하기 때문에 부분곱 축약 단계의 연산을 효율적으로 수행할 수 있다. 또한, 사용되는 십진 CLA의 로직을 개선하여 BCD 결과를 빠르게 얻을 수 있다. 제안한 십진 부분곱 축약 단계의 성능의 평가를 위해 Design Compiler를 통해 SMIC사의 180nm CMOS 공정 라이브러리를 이용하여 합성하였다. 일반 방법을 이용하는 축약 단계에 비해 제안한 부분곱 축약 단계의 지연시간은 약 15.6% 감소하였고 면적은 약 16.2% 감소하였다. 또한 십진 CLA의 지연시간과 면적이 증가가 있음에도 불구하고 전체 지연시간과 전체 면적이 감소함을 확인하였다.

Abstract

In this paper, in order to reduce the delay and area of the partial product accumulation (PPA) of the parallel decimal multiplier, a tree architecture that composed by multi-operand decimal CSAs and improved CLA is proposed. The proposed tree using multi-operand CSAs reduces the partial product quickly. Since the input range of the recoder of CSA is limited, CSA can get the simplest logic. In addition, using the multi-operand decimal CSAs to add decimal numbers that have limited range in specific locations of the specific architecture can reduce the partial products efficiently. Also, final BCD result can be received faster by improving the logic of the decimal CLA. In order to evaluate the performance of the proposed partial product accumulation, synthesis is implemented by using Design Complier with 180 nm COMS technology library. Synthesis results show the delay of the proposed partial product accumulation is reduced by 15.6% and area is reduced by 16.2% comparing with which uses general method. Also, the total delay and area are still reduced despite the delay and area of the CLA are increased.

Keywords : Parallel decimal multiplication, IEEE 754-2008, Multi-operand

Ⅰ. 서 론

1970년 이래로 마이크로프로세서 기반의 디지털 플랫폼은 무어의 법칙을 따라 2년마다 약 2배의 집적도 향상이 있었다. 그러나 마이크로프로세서 제조기술이 명령어 실행 속도 향상에 집중한 반면 메모리 제조기술은 주로 속도보다는 용량 증가에 집중되었다. 이로 인해 프로세서와 메모리 간의 성능은 크게 벌어졌고 이를 “메모리 장벽”이라고 일컫는다[1~2]. 메모리 장벽을 극복

* 학생회원, ** 평생회원, 인하대학교 전자공학과
(Dept. of Electronic Engineering, Inha University)
© Corresponding Author (E-mail: sangbang@inha.ac.kr)
※ 이 논문은 2010년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임 (2010-0020163).
Received ; July 4, 2016 Revised ; November 5, 2016
Accepted ; November 11, 2016

하기 위해 프로세서 설계자는 캐시 메모리 레벨의 계층화에 주목하였다. 메모리 접근이 국지적으로 발생한다는 사실을 바탕으로 한 캐시 메모리 레벨의 계층화는 메모리 접근 시간 단축과 프로세서와 메모리 간의 성능 격차를 줄이는 데 크게 기여하였다. 하지만 최신 소프트웨어들이 복잡한 메모리 접근 패턴을 가지게 되면서 단순한 메모리 레벨 계층화로는 성능 개선 효과를 보기 어렵게 되었다. 다양한 메모리 접근 패턴은 상위 레벨 캐시의 미스율을 크게 증가시키며 미스가 발생한 데이터를 하위 메모리 계층으로부터 얻는 동안 발생하는 지연으로 인해 프로세서는 많은 시간을 유휴 상태로 보내야 한다[3~4].

이러한 메모리 접근 지연을 효과적으로 줄이는 방법이 프리페치 기법이다. 프리페치 기법은 미래에 발생할 메모리 접근을 예측하고 프로세서가 접근하기 이전에 해당 메모리 블록을 미리 요청하여 캐시에 페치하는 것을 말한다. 예측이 정확하다면 메모리 지연을 크게 줄일 수 있으나, 예측이 틀렸을 경우 메모리 대역폭의 낭비, 캐시 오염(기존 유용한 캐시 삭제로 인한 지연 증가) 등의 악영향을 초래한다. 따라서 프리페치 기법 사용 시 정확한 예측과 적절한 시간에 프리페치를 수행하는 것이 필요하다.

전통적인 프리페치 기법은 메모리 접근의 시간 지역성 및 공간 지역성을 활용한다[1]. 시간 지역성은 최근에 접근한 패턴이 가까운 미래에 다시 반복할 가능성이 있다는 것과 관계있다. 공간 지역성은 접근 가능성이 높은 곳은 물리적으로 인접한 메모리 영역이라는 사실과 관계있다. 이러한 지역성을 이용하여 메모리 접근 지연을 효과적으로 단축시킬 수 있으나 모든 워크로드에 지역성을 적용할 수 없다는 문제가 있다. 또한 아키텍처의 발전으로 최신 프로세서는 전통적인 프리페처가 적용된 프로세서와 매우 다른 방식으로 동작한다. 프로세스 실행 속도 향상을 위해 비순차 실행, 루프 언롤링 등을 수행하면서 메모리 접근 패턴이 뒤섞인다. 그리고 파이프라인의 메모리 연산 중 발생하는 스톨을 최소화하기 위해 write back 캐시를 사용하는 데 이 때 MSHR(miss status holding register)이나 쓰기 버퍼에 대기하는 요청으로 인해 하위 메모리 계층에서 예측하기 어려운 접근 패턴이 발생한다.

이렇게 불규칙적인 접근 패턴으로부터 프리페처의 성능을 보호하기 위한 대표적인 프리페치 기법은 AMPM 프리페처[5]와 오프셋 프리페처[6~8]가 있다. AMPM(access map pattern map)은 메모리 전체의 접근 패턴을 기록하기 때문에 인접 블록에 접근이 발생하지 않는 영역에서의 프리페치를 피할 수 있다. 그러나 다수의 메모리 공간에서 발생하는 접근 패턴을 기록하기 위해 다소 큰 메모리 공간이 요구된다. 오프셋 프리페처는 학습에 의해 확률적으로 가장 빈도수가 높은 오프셋을 블록 주소와 더하여 프리페치를 요청함으로써 부정확한 프리페치를 최소화한다. 그러나 프리페치 전략이 다소 공격적이기 때문에 프리페치가 성능에 악영향을 주는 상황이 발생하는 경우 스스로 프리페치를 제한하는 기법이 추가로 요구된다.

본 논문에서는 AMPM 프리페처를 기본적으로 이용하되 조건에 따라 오프셋 프리페처를 선택적으로 사용하는 프리페처를 제안한다. 제안하는 프리페처는 현재 메모리 접근 패턴에 오프셋 프리페처의 사용이 유리하다고 판단할 때에만 활성화하며 성능에 악영향을 준다고 판단하는 경우 프리페처의 사용을 제한한다. 이를 통해 다양한 워크로드에 대하여 효율적이고 정확한 프리페치가 가능하다. 또한 MSHR에 처리가 지연되는 블록이 많을 경우 프리페치 가능 개수를 줄이거나 프리페치를 차단하여 프로세서의 요구 요청 블록이 우선적으로 처리되는 것을 보장한다. 반대의 경우 허용 프리페치 개수를 증가시켜 적극적인 프리페치를 수행할 수 있도록 한다.

본 논문은 다음과 같이 구성된다. Ⅱ장에서는 하드웨어 프리페처에 대한 개념 설명에 이어 전통적인 프리페처와 최신 프리페처에 대하여 설명하고, Ⅲ장에서는 제안하는 프리페처의 구조와 동작 과정에 대하여 설명한다. Ⅳ장에서는 실험을 통해 제안하는 하드웨어 프리페처의 성능을 검증하고, Ⅴ장에서 결론을 맺는다.

Ⅱ. 하드웨어 프리페치 기법

하드웨어 프리페치 기법은 소프트웨어 프리페치 기법처럼 인위적으로 캐시 블록을 제어하는 것이 아니라 하드웨어가 자동으로 프리페치 기능을 수행하는 것이다. 프로세서의 요구 요청(demand request) 접근이 예상되는 메모리 블록 주소 값들을 찾아내고 이 주소들이 가리키는 메모리 블록을 미리 캐시에 페치한다. 하드웨어 프리페치가 효과적으로 동작하기 위해서는 프로세서가 접근할 메모리 블록 주소들을 정확하게 예측하여 그 정보를 버퍼에 저장해야 한다. 이를 위해 next-line 프리페처를 제외한 대부분의 하드웨어 프리페치 기법은 프리페치를 하기 앞서 프로세서의 요청 메모리 주소,

프로그램 카운터, 이전 요청 메모리 주소, MSHR 정보, 캐시 힛/미스 등 캐시와 메모리 안에 존재하는 각종 정보를 이용한 학습을 선행한다.

본 논문에서는 역사가 오래되고 프리페처 구현의 기반이 되는 전통적인 프리페처들을 소개하고 최신 하드웨어 프리페처들과 함께 비교 설명한다.

2.1 전통적인 하드웨어 프리페처

2.1.1 Next-line 프리페처

Next-line 프리페처[6]는 구현의 단순함에 비해 가장 강력한 프리페치 성능을 보여준다. 만약 프로세서가 주소 A를 가지는 블록을 요청하면 그와 동시에 바로 다음 블록인 A+1에 대한 요청을 수행한다. 별도 학습으로 인한 오버헤드가 없기 때문에 접근 패턴의 공간 지역성이 높은 경우 가장 빠른 프로세스 실행 시간을 가질 수 있다. 그러나 미스율 증가, 캐시 오염 발생, 대역폭 및 MSHR의 사용 증가 등을 판단하지 않고 무조건 프리페치를 요청하기 때문에 워크로드의 메모리 접근 패턴이 불규칙하거나 시간 지역성이 높은 경우 심각한 성능 저하를 가져올 수 있다.

2.1.2 스트림 프리페처

스트림 프리페처[6]는 한 방향으로 3회 이상 연속 접근하는 스트림이 발견되면 다음 블록을 프리페치하는 방식으로 동작한다. 예를 들어 첫 요구 요청 블록 주소가 A라고 할 때, 뒤이어 A+1, A+2가 연속으로 요청되면 해당 스트림에 대한 프리페치가 유효하다고 판단하여 그 다음 블록인 A+3에 대한 프리페치를 요청한다. 이 때 한 방향으로의 연속 접근이 프리페치하기 유효한지 판단하기 위해 신뢰도 값을 설정한다. 프리페처 구동 초기에는 신뢰도 값을 0으로 초기화하며 방향이 연속되면 신뢰도 값을 1 증가시킨다. 이후 신뢰도 값이 2 이상이면 프리페치가 유효하다고 판단하여 해당 방향으로 +1 만큼 떨어진 블록에 대한 프리페치를 요청한다. 만약 방향이 변경된 경우 해당 스트림을 신뢰할 수 없다고 판단하여 신뢰도 값을 0으로 초기화하고 스트림을 다시 추적한다.

스트림 프리페처는 공간 지역성이 높은 접근 패턴을 가지는 워크로드에서 좋은 성능을 보이는 프리페처다. Next-line 프리페처와는 달리 스트림에 대한 신뢰도가 보장되었을 때만 프리페치를 요청하기 때문에 부정확한 프리페치를 방지할 수 있다. 또한 역방향 스트림에 대한 프리페치가 가능하기 때문에 next-line 프리페처에 비해 확장성이 높다. 그러나 공간 지역성이 높은 워크로드를 위하여 특화되어 있기 때문에 접근 패턴이 불규칙하거나 시간 지역성이 높은 경우 부정확한 프리페치를 하거나 낮은 신뢰도 때문에 프리페치를 수행하지 못한다.

2.1.3 스트라이드 프리페처

스트라이드 프리페처[2, 6]는 프로세서가 현재 접근한 주소와 이전에 발생한 주소 값의 차이 즉 스트라이드를 찾아내고 이를 접근 주소에 더하여 프리페치를 수행한다.

그림 1은 스트라이드 프리페처의 구조를 나타낸다. 스트라이드 프리페처에서 사용하는 검색 테이블의 엔트리는 명령어 태그 필드(Instruction Tag), 이전 주소 필드(Previous Address), 스트라이드 필드(Stride), 상태 필드(State) 네 개의 필드로 구성되어 있다. 명령어 태그 필드는 로드/스토어 명령어의 프로그램 카운터 값을 저장한다. 이전 주소 필드는 이전의 실행된 명령어의 데이터 주소를 저장한다. 스트라이드 필드는 이전 주소 필드에 저장된 주소 값과 현재 유효 주소의 차이를 계산하여 저장한다. 상태 필드는 초기(init), 과도(transient), 안정(steady), 불가(no-pred) 네 개의 상태를 가지며, 상태값에 따라 프리페치를 요청할지 여부를 결정한다. 프리페치를 수행할 주소(Prefetch Address)는 이전 주소 필드에 저장된 값과 스트라이드를 더하여 계산된다. 로드/스토어 명령어의 상태가 안정 상태이고 스트라이드 값이 0이 아니면 프리페치 명령을 요청한다. 이전에 프리페칭된 주소와 현재 액세스한 메모리 주소가 동일하다면 주소 예측에 성공한 것이고, 동일하

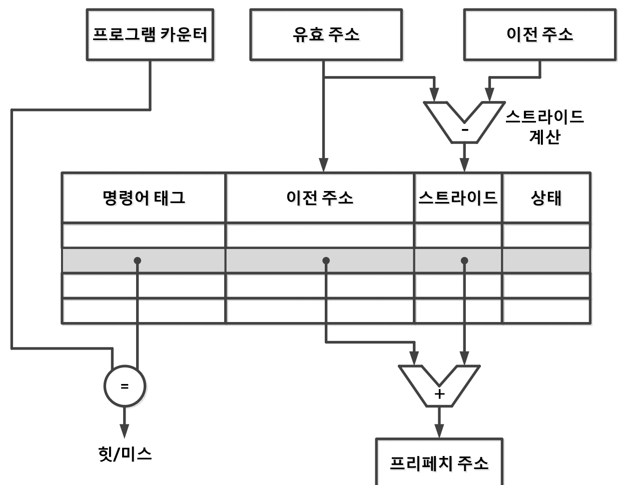

그림 1. 스트라이드 프리페처 구조
Fig. 1. The stride prefetcher structure.

지 않다면 주소 예측에 실패한 것이다.

스트라이드 프리페처는 최초 동일한 스트라이드 값이 발생하였을 때 무조건 프리페치하지 않고 2회 이상 동일한 스트라이드 값이 발생함을 확인한 후 프리페치를 수행한다. 이를 통해 불규칙한 메모리 접근 패턴에 의한 프리페치 오류를 줄일 수 있다. 또한 스트라이드 값이 다른 경우 바로 초기화함으로써 부정확한 프리페치를 방지한다.

스트라이드 프리페처는 스트림 프리페처와 마찬가지로 정방향과 역방향의 프리페치를 지원하며 스트림보다 더 다양한 거리의 프리페치가 가능하다. 그러나 스트라이드 프리페처 또한 워크로드의 메모리 접근 패턴이 공간 지역성이 높다는 것을 전제로 하여 설계되었기 때문에 접근 패턴이 동적으로 변하는 경우 스트라이드를 찾더라도 다음 요구 요청 접근에 스트라이드가 변하면서 해당 스트라이드에 대한 신뢰도가 떨어지기 때문에 프리페치가 수행되지 않는다.

2.2 최신 하드웨어 프리페처

2.2.1 AMPM 프리페처

AMPM(Access Map Pattern Matching) 프리페처[5]는 스트림, 스트라이드와 같은 국지적인 지역성을 찾아 이를 이용하는 전통적인 프리페처와는 달리 가상 메모리 전체의 접근 자취를 추적하여 이를 프리페치에 사용한다. 페이지 단위와 같이 메모리 공간을 고정 크기 영역으로 나눈 존(zone)이라는 개념을 도입하였다. 존들 중에서 최근 접근이 발생하는 존을 핫존(hot-zone)이라고 한다.

그림 2는 핫존 내에서 발생하는 접근과 프리페치 후보를 찾는 동작을 보여준다. AMPM 프리페처는 요구 요청 블록을 포함한 이전 요구 요청 접근의 스트라이드 3개를 찾아낸다. 그림 2와 같이 요구 접근이 0x05가 발생하였을 때 핫존에 기록된 이전 접근들의 정보를 살펴본다. 이 때 검색되는 스트라이드는 {0x01, 0x03, 0x05}와 {0x03, 0x04, 0x05} 두 가지이다. 이처럼 다수의 후보가 발견되는 경우 AMPM은 이 중 요구 접근과 가장 가까운 블록인 0x06에 대한 프리페치를 먼저 요청한다. AMPM 프리페처는 또한 두 개의 시프터(shifter)를 두어 정방향 스트라이드와 역방향 스트라이드를 동시 검색하여 프리페치 후보를 결정한다.

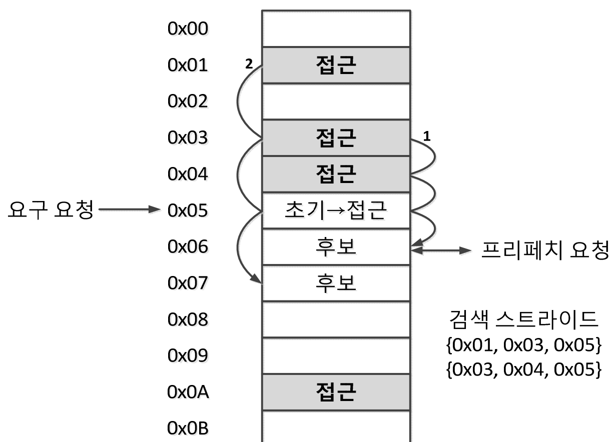

그림 2. AMPM 프리페처
Fig. 2. Access Map Pattern Matching Prefetcher.

AMPM 프리페처는 메모리 공간 전체에 대하여 요구 요청 접근을 추적하기 때문에 불규칙한 접근 패턴 또는 시간 지역성을 가지는 워크로드에 대하여 부정확한 프리페치를 요청할 가능성을 크게 낮출 수 있다. 그러나 메모리 공간 일부를 맵핑할 수 있는 존을 기록할 큰 메모리 공간이 필요하며 요구 요청 접근마다 이 거대한 공간을 탐색해야 하는 오버헤드가 추가된다.

2.2.2 오프셋 프리페처

오프셋 프리페처[6~8]는 단일 또는 다수의 오프셋을 결정하여 요구 요청 접근이 들어올 때 오프셋들을 더하여 프리페치를 요청한다. 예를 들어 요구 요청 블록 주소가 A라고 할 때 미리 설정한 오프셋이 3이면 A+3에 대한 프리페치를 요청한다. 오프셋이 1이면 next-line 프리페처와 동일하다. 오프셋 프리페처는 요구 요청 접근이 들어오면 바로 프리페치를 수행하기 때문에 다른 프리페처들에 비해 상당히 공격적인 정책을 가지고 있다. 이 때문에 프리페치가 성능에 악영향을 줄 수 있다고 판단되는 경우 프리페치 수행을 중단시킬 필요성이 있다. 이와 같이 프리페처의 사용을 제어하기 위해 오프셋 프리페처는 캐시 힛/미스율, 이용 가능한 대역폭, MSHR 사용 현황 등의 다양한 조건을 활용한다.

대표적인 오프셋 프리페처로는 FDP(Feedback Directed Prefetching)[7], 샌드박스 프리페처[8]가 있으며 이후 최근에는 샌드박스 프리페처를 수정한 다양한 프리페처들이 제안되어 왔다[6]. 그림 3은 샌드박스 프리페처의 구조이다. 샌드박스 프리페처의 특징은 실제 메모리 계층으로 프리페치를 요청하는 것이 아니라 샌드박스라는 공간에 가상으로 프리페치를 요청하는 것이다. 예를 들어 사용 중인 오프셋이 1, 2, 3이고 요구 요청 블록 주소가 A라고 할 때, 오프셋들을 더하여 A+1, A+2, A+3를 샌드박스에 저장한다. 이후 다음 접근이

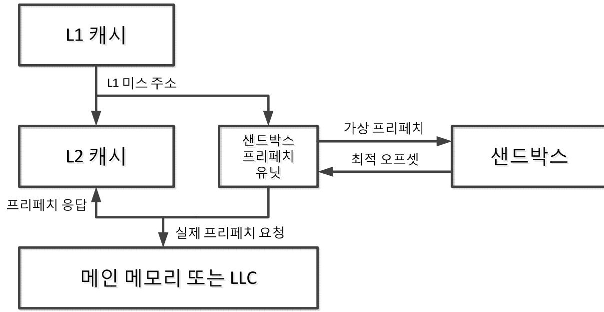

그림 3. 샌드박스 프리페처
Fig. 3. Sandbox prefetcher.

발생하였을 때 해당 요구 요청 블록의 주소가 샌드박스에 존재하는 지 확인한다. 만약 다음 접근 블록 주소가 A+2와 일치하면 오프셋 2의 점수가 1 증가한다. 이러한 과정을 미리 정의한 학습시간이 종료될 때까지 반복한다. 학습이 종료되면 프리페처를 활성화하고 다음 요구 요청 접근 블록 주소에 가장 점수가 높은 오프셋을 더하여 프리페치를 요청한다. 프리페처가 활성화되어 있는 상태에서도 지속적으로 학습을 수행하며 다른 오프셋이 점수가 높으면 오프셋을 교체한다. 그리고 최고 점수가 정의된 임계값보다 낮으면 프리페치 수행을 중단시킴으로써 부정확한 프리페치로 인한 영향을 최소화한다.

그러나 정확한 오프셋을 가지고 프리페치하더라도 하위 메모리 계층으로부터 블록 정보를 가져오는 데 걸리는 지연시간으로 인해 상위 메모리 계층의 요구 요청 이전에 프리페치 블록이 캐시에 페치되는 것은 보장하지 않는다. 이러한 문제를 해결하기 위해 BO(best offset) 프리페처[6]는 딜레이 큐를 사용한다. 딜레이 큐는 요구 요청 접근마다 발생하는 오프셋 평가를 지연시키고 프리페치를 요청한 블록이 캐시에 페치되는 시점에 오프셋이 평가되도록 조정함으로써 LLC나 메인 메모리로부터의 응답 시간까지 고려한 오프셋을 얻어낼 수 있게 한다.

스트림, 스트라이드, AMPM과 같은 프리페처들은 요구 요청 접근 히스토리를 수용할 수 있는 메모리 공간과 빠른 정보 검색을 가능하게 하기 위한 복잡한 로직이 필요하다. 그러나 오프셋 프리페처는 큰 저장공간을 요구하는 히스토리 정보가 필요하지 않다. 대신 오프셋과 점수 등의 적은 정보량만으로 프리페치를 수행한다. 오프셋 프리페처가 오프셋을 이용한 공격적인 프리페치를 수행함에도 불구하고 뛰어난 성능을 발휘하는 이유는 충분한 학습시간을 통해 얻은 가장 확률이 높은 오프셋을 선별하여 사용하기 때문이다. 다만 공격적인 프리페치로 인해 부정확한 프리페치를 수행할 확률 또한 높기 때문에 워크로드의 메모리 접근 패턴에 따라 성능 차이가 심한 단점이 있다.

2.2.3 엑스퍼트 프리페치 예측

엑스퍼트(expert) 프리페치 예측[9]은 프리페처가 생성한 프리페치 요청의 유효성을 4개의 엑스퍼트로 구성된 필터를 이용하여 결정한다. 다양한 프리페처와 결합하여 사용가능하며 프리페처의 파라미터 값의 변화나 상태 변화에 영향을 주지 않고 독립적으로 동작한다. 각 엑스퍼트는 프로그램 카운터, 블록 주소, 영역(영역은 2KB이며 블록 주소를 오른쪽으로 11번 시프트하여 얻을 수 있음), 프로그램 카운터와 블록 주소의 비트-OR를 시그니처로 사용한다. 또한 각 엑스퍼트는 가중치를 가지고 있다. 캐시에서 쫓겨난 프리페치 블록이 한번 이상 참조된 기록이 존재하면 예측이 정확한 것으로 판단하여 가중치를 증가시키며 참조 기록이 존재하지 않으면 예측이 부정확한 것으로 판단하여 가중치를 감소시킨다. 엑스퍼트의 엔트리는 포화 카운터 2bit로 구성되어 있다. 포화 카운터는 캐시에서 쫓겨난 프리페치 블록이 참조된 기록이 존재할 때 증가하며 참조된 기록이 존재하지 않는 경우 감소한다. 이와 같은 가중치 변화와 포화 카운터를 이용한 가중치-우선 필터는 그림 4와 같다.

각 엑스퍼트는 프리페치 주소에 맵핑된 엔트리의 포화 카운터 값을 확인하고 2이상이면 1을 출력하며 아니면 0을 출력한다. 이후 1을 출력한 엑스퍼트들의 가중치 값을 더한 p와 0을 출력한 엑스퍼트들의 가중치 값을 더한 n을 얻는다. 만약 p가 n보다 크면 프리페치 요청하며 그렇지 않으면 요청하지 않는다.

엑스퍼트는 이전 연구의 오염 필터[10]와 동일한 구조

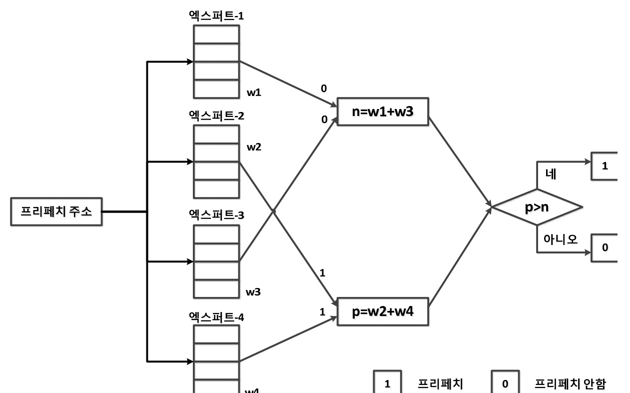

그림 4. 가중치-우선 필터
Fig. 4. Weighted-majority filter.

를 가지고 있다. 그러나 단일 필터를 사용하는 오염 필터와 달리 다수의 필터를 사용하며, 가중치-우선 알고리즘을 이용하여 예측 정확도에 따라 각 엑스퍼트의 가중치를 부여함으로써 다양한 워크로드의 패턴에 대응하고자 한다. 하지만 가중치 계산 과정에서 부동소수점 연산을 사용하므로 실제 하드웨어 구현 시 복잡도가 매우 높고, 캐시 블록에 추가되는 플래그 비트로 인해 MSHR, 쓰기 버퍼, 메모리 등 다수의 하드웨어 구조가 변경되어야 한다는 문제점이 있다.

Ⅲ. 제안하는 프리페처

이 장에서는 본 논문에서 제안하는 프리페처에 대해 설명한다. 우선 AMPM과 오프셋을 함께 사용하는 이유를 설명한다. 그리고 제안하는 프리페처의 동작 과정에 대하여 설명한다.

3.1 AMPM 프리페치와 오프셋 프리페치의 결합

공간 지역성은 존재하지만 메모리 접근 패턴이 불규칙한 경우 AMPM과 같이 전체적인 메모리 영역의 접근 패턴을 추적함으로써 프리페치 오류를 줄이는 방법이 유리하다. 반면 메모리 접근 패턴이 공간 지역성을 가질 뿐만 아니라 비교적 규칙적이고 스트림 형태를 보이는 경우 오프셋과 같이 확률적으로 빈도수가 높은 오프셋을 더하여 공격적으로 프리페치를 요청하는 방법이 유리하다. 이처럼 다양한 워크로드의 메모리 접근 패턴과 프로세스 실행 중 시시각각 변화하는 메모리 접근 패턴에 대응하기 위해서는 프리페치 전략 또한 이에 맞추어 변화해야 한다. 따라서 본 논문에서는 메모리 접근 패턴의 변화, 프리페치 정확도를 추정하여 이를 기준으로 AMPM 기법과 오프셋 기법을 적절하게 활용하는 프리페처를 제안한다.

3.2 제안하는 프로페처의 구조 및 동작과정

기존의 AMPM 프리페처는 32036 bits의 저장공간 중 접근 맵 테이블이 29147 bits를 차지는 등 많은 메모리 공간을 필요로 한다. 이로 인해 칩 크기, 검색 및 통신 오버헤드, 전력 소모 증가 등의 역효과가 발생한다. 따라서 본 논문에서는 이러한 문제를 완화시킨 AMPM의 수정 버전인 DPC-2의 AMPM lite[6]를 이용한다. AMPM lite는 기존의 AMPM에 비해 제한된 페이지 주소 공간 64 bytes 만을 추적할 수 있도록 간소화하였다. 256 bytes의 존을 사용하는 AMPM만큼의 커버리지를 얻을 수는 없지만 검색 오버헤드가 줄어들고 제한된 페이지 테이블 공간에 메모리 접근이 연속되는 경우 기존 AMPM과 동일한 성능을 얻을 수 있다. 이를 확인하기 위해 시뮬레이션을 수행한 결과 AMPM-lite는 AMPM 대비 평균 0.02% 높은 IPC 값을 가져 성능상의 차이가 거의 없음을 보여주었다.

그림 5는 메모리 접근 히스토리를 저장하는 AMPM 테이블 엔트리 구조이다.

크기	8 bytes	8 bytes	8 bytes	2 bytes
필드	페이지 주소	접근맵	프리페치맵	타임 스탬프

그림 5. AMPM 테이블 엔트리 구조
Fig. 5. A structure of AMPM table entry.

페이지 주소 필드에는 요구 요청 메모리 접근이 발생한 페이지 주소를 기록한다. 접근 맵 필드에는 캐스 미스 또는 힛이 발생한 페이지 내의 모든 요구 요청 메모리 접근을 비트로 표시한다. 프리페치 맵 필드에는 프리페치를 요청한 캐시 블록의 위치를 비트로 표시한다. 타임스탬프 필드에는 처음 엔트리를 테이블에 등록할 당시의 싸이클을 기록한다. 페이지 테이블의 엔트리 개수는 총 64개이며 LRU 방식으로 오래된 엔트리를 교체한다. 요구 요청 메모리 접근 발생 시 해당 요청 주소를 이용하여 페이지 테이블을 검색한다. 만약 일치하는 페이지 주소를 가지는 엔트리를 발견하면 프리페치 예측을 수행한다. 주소를 포함하는 블록을 A라고 할 때 이전 블록 A-1과 그 이전 블록 A-2의 접근 기록이 접근맵 상에 존재하면 A+1을 예측하여 프리페치 요청을 하위 메모리 계층으로 전송한다. 정방향 스트림에 대한 프리페치와 동일하게 역방향 스트림에 대한 프리페치 또한 수행한다. 이전 블록 A+1과 그 이전 블록 A+2에 대한 접근이 존재하면 A-1에 대한 프리페치 요청을 하위 메모리 계층으로 전송한다. 만약 접근 맵 필드 또는 프리페치 맵 필드에 해당 블록을 요청한 기록이 존재하면 프리페치를 수행하지 않음으로써 중복 프리페치 요청을 방지하고 불필요한 대역폭 사용을 줄인다.

그림 6은 제안하는 프리페처의 AMPM 동작 과정을 보여주는 예이다. 블록 크기는 64 bytes이고 페이지 크기는 4096 bytes라고 가정할 때 페이지 내 블록 개수는 총 64($4096 = 64\,\text{bytes} \times 64$)개이다. 우선 주소를 오른쪽으로 12번 ($2^{-12} = 1/4096$) 쉬프트하여 얻은 페이지 주소 0xFB062가 페이지 테이블에 있는지 확인한다. 그림 6과 같이 페이지 테이블에 페이지 주소가 존재하

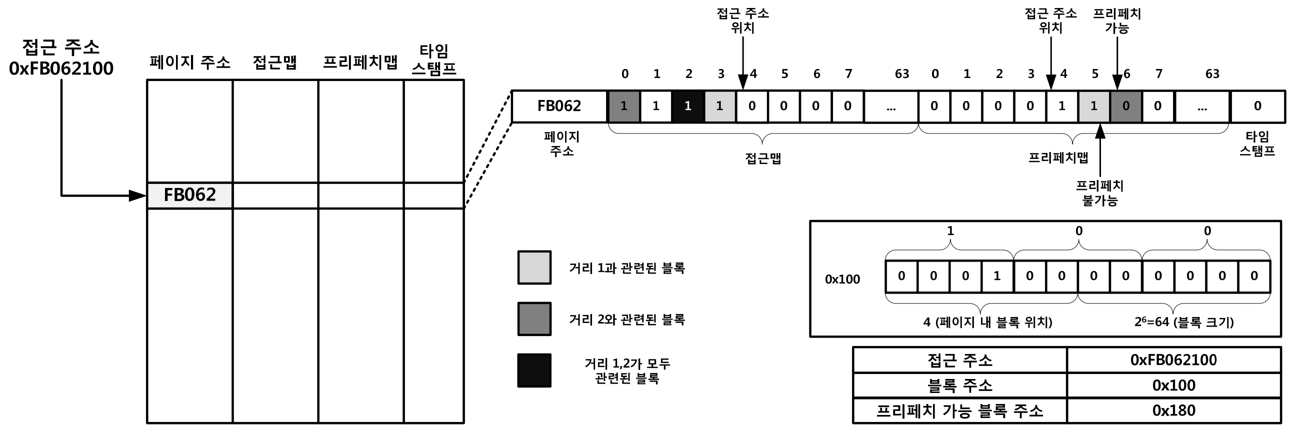

그림 6. AMPM 동작 과정
Fig. 6. AMPM Operation Process.

면 해당 페이지 주소가 가리키는 엔트리 내 접근맵과 프리페치 맵을 확인한다. 접근 주소의 블록 주소는 0x100로 접근맵의 4를 가리킨다. 페이지 내에 0x100 이전 블록들은 모두 접근이 존재하므로 페이지 경계 내에 프리페치가 가능한 거리는 1과 2가 된다. 그러나 접근 주소 위치에서 거리 1만큼 떨어진 블록 5를 통해 이미 프리페치가 수행되었다는 것을 알 수 있으므로 거리 2만큼 떨어진 블록 6만 프리페치가 가능하다. 따라서 프리페치맵 6이 가리키는 블록 주소는 0x180이므로 최종적으로 0xFB062180에 대한 프리페치를 요청한다.

제안하는 프리페처에서 평가할 오프셋은 －16부터 16까지 (0 제외) 총 32개의 오프셋을 포함한다. 예를 들어 오프셋이 1이고 요구 요청 블록 주소 A가 발생한 경우 다음 블록 주소인 A+1과 오프셋을 샌드박스에 함께 저장한다. 샌드박스는 총 128개의 엔트리를 가지고 있으며 FIFO 방식을 사용하는 큐이다. 프리페처가 사용하는 오프셋이 32개이므로 샌드박스는 총 4번 (32×4=128)의 접근에 의한 가상 프리페치 기록을 저장할 수 있다.

프리페처에 입력된 주소가 샌드박스 내에 존재하는지 확인하며 검색에 성공하면 해당 주소에 대한 알림이 캐시 힛 또는 MSHR 힛에 의한 것인지 확인한다. 캐시 힛은 상위 계층의 요구 요청 블록이 캐시에 존재하는 것을 말하며 MSHR 힛은 상위 계층의 요구 요청 블록이 캐시에 존재하지 않아 미스가 발생하였으나 MSHR에서 응답을 기다리고 있는 상태를 말한다. 만약 캐시 힛에 의한 알림이면 오프셋을 성공적으로 예측한 것이므로 해당 오프셋에 대한 점수를 1 증가시킨다. 반대로 MSHR 힛에 의한 알림이면 해당 오프셋에 의한 프리페치가 요구 요청 이전에 이뤄지지 못하고 MSHR에 대기하고 있는 것이므로 시기적절한 프리페치가 아니라고 판단하여 점수를 1 감소시킨다.

프리페처로 입력되는 횟수가 1024에 도달하면 학습이 종료되며 기록한 점수가 가장 높은 오프셋을 프리페치 후보 테이블 1순위부터 차례로 추가한다. 이 때 점수가 임계값인 LOW_SCORE 16 이상인 경우에만 후보로 등록함으로써 점수가 음의 값을 가지거나 매우 낮은 오프셋이 후보에 포함되지 않도록 한다. 프리페처 후보 테이블은 최대 4개의 엔트리를 가질 수 있다. 모든 과정이 종료되면 샌드박스의 점수를 초기화하고 학습을 재시작한다.

그림 7은 오프셋의 동작 과정을 보여주는 예이다. 주소 a,b가 접근하기 이전에 블록 주소 A의 접근으로 인해 샌드박스는 블록 A-16에서 A+16까지의 가상 프리페치 블록 주소들을 가지고 있는 상태이다. 캐시 힛이 발생한 접근 블록 주소 a와 동일한 A+3이 샌드박스에 존재하므로 스코어보드에서 해당 오프셋 +3의 점수를 1 증가시킨다. 반면 MSHR 힛이 발생한 접근 블록 주소 b와 동일한 A+1이 샌드박스에 존재하므로 스코어보드에서 해당 오프셋 +1의 점수를 1 감소시킨다. 만약 접근 횟수가 1024번을 초과하면 학습을 종료하고 스코어보드에서 점수가 높은 4개의 오프셋을 순서대로 프리페치 후보 테이블에 채운다. 프리페치 후보 테이블에 엔트리가 존재하면 이후 접근하는 블록 주소 c에 대하여 프리페처는 접근 주소와 이들 후보 오프셋들과 더하여 {c+1, c-1, c+2, c+3}에 대한 프리페치를 요청한다.

3.3 MSHR 임계값 설정

MSHR은 요구 요청들이 응답을 기다리기 위한 공간이지만 프리페치 요청들 또한 이 공간을 차지한다. 그

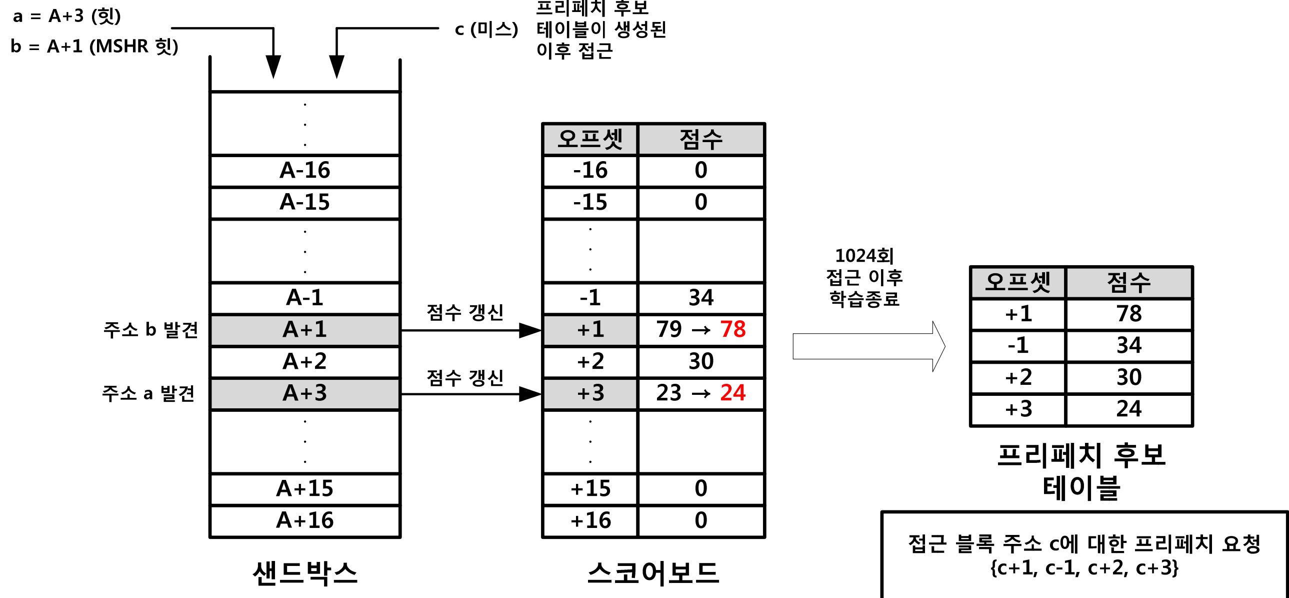

그림 7. 오프셋 동작 과정
Fig. 7. Offset Operation Process.

러나 요구 요청은 상위 계층에서 필요한 블록을 직접 요청한 것이므로 예측에 의해 블록을 요청하는 프리페치 요청보다 우선적으로 처리해야 한다. 만약 다수의 프리페치 요청들로 MSHR이 채워지면 요구 요청을 바로 처리하지 못하여 프로세스의 실행 시간이 늘어난다. 따라서 MSHR 사용량에 따라 프리페치를 제한함으로써 요구 요청을 우선적으로 처리할 수 있도록 보장해야 한다. 반대로 요구 요청이 적어 MSHR에 여유가 있다면 적극적인 프리페치를 통해 성능 이득을 최대화할 수 있다.

이를 위해 제안하는 프리페처는 MSHR 임계값을 설정한다. MSHR 사용량이 MSHR 임계값을 넘지 않으면 MSHR 임계값과 MSHR 사용량의 차이만큼 프리페치를 요청할 수 있다. 예를 들어 MSHR 임계값을 8로 설정하였을 때 현재 MSHR 사용량이 4라면 한 번에 4개의 프리페치 요청이 가능하다. 그러나 만약 현재 MSHR의 사용량이 MSHR 임계값을 초과하면 프리페처의 정확도와 관계없이 프리페치 요청을 금지함으로써 상위 계층의 요구 요청을 먼저 처리하도록 유도한다.

MSHR 임계값은 메모리 접근 패턴에 따라 동적으로 설정할 필요성이 있다. MSHR 임계값이 지나치게 높으면 MSHR의 다수 공간을 프리페치 블록이 차지해 프로세서의 요구 요청 처리가 지연될 수 있다. 반면에 MSHR에 충분한 여유 공간이 있음에도 불구하고 낮은 MSHR 임계값에 의해 프리페치가 제한받으면 프리페치에 의한 성능 이득을 볼 수 없다. 본 논문에서는 MSHR 임계값을 조절하기 위해 샌드박스의 학습기간 동안 모니터한 MSHR 힛 발생 횟수를 이용한다. 표 1은 MSHR 힛 발생 횟수에 의한 MSHR 임계값 증감을 나타낸다.

표 1. MSHR 임계값 설정
Table 1. Configuration for MSHR threshold.

MSHR 힛	증감
> 256	-2
> 128	-1
> 64	+1
>= 0	+2

만약 MSHR 힛의 개수에 따라 고정 임계값을 사용(예를 들어 MSHR 힛이 256 이상이면 4, 128 이상이면 8, 그 외는 12와 같이 고정 값을 설정)하면 학습기간 구간마다 접근 패턴이 급격하게 변하는 경우 임계값이 최댓값과 최솟값으로 편중되어 프리페치 기회를 잃거나 과도한 프리페치를 수행한다. 따라서 일시적으로 달라지는 접근 패턴에 의해 임계값이 민감하게 변하지 않도록 하기 위해 점근적으로 임계값을 증감시키는 방법을 사용한다.

MSHR 힛 발생 횟수가 많다는 것은 요구 요청 및 프리페치 요청의 처리가 지연되고 있다는 것을 의미한다. 따라서 프리페치에 의해 요구 요청의 처리가 지연되는

것을 방지하기 위해 MSHR 임계값을 줄여 프리페치 개수를 제한한다. 반대로 MSHR 힛 발생 횟수가 적으면 MSHR 임계값을 증가시켜 적극적인 프리페치 요청을 수행할 수 있도록 한다. 또한 MSHR 힛 발생 횟수에 따라 가중치를 두어 MSHR 임계값을 증감시킴으로써 현재 메모리 접근 패턴에 맞는 정확한 MSHR 임계값을 빠르게 찾을 수 있도록 한다.

L2 MSHR의 크기가 16이라고 할 때 MSHR 임계값의 최대값/최소값을 12/4로 설정하였다. 이는 요구 요청마다 프리페치 개수(degree : 4)만큼의 마진을 둠으로써 학습한 패턴과 다른 메모리 접근 패턴에 대응하기 위해서이다. MSHR의 요구 요청이 모두 처리되면 MSHR 임계값이 최소값을 가지더라도 프리페치를 수행할 수 있다. 또한 프리페치로 인해 최대 MSHR 임계값만큼 MSHR이 채워지더라도 4개의 여유공간을 통해 요구 요청을 처리할 수 있다.

3.4 프리페치 요청 필터링

프리페치 후보 테이블이 생성되어 오프셋이 활성화되면 AMPM과 오프셋을 함께 사용하기 때문에 두 프리페치에서 발생하는 요청이 중복될 가능성이 있다. 이러한 문제를 해결하기 위해 프리페처에 프리페치 큐를 추가함으로써 중복되는 블록 주소에 대한 제거 이후 프리페치 요청을 수행하도록 하였다.

3.5 전체 동작 흐름도

그림 8은 3.2, 3.3, 3.4를 포함한 제안하는 프리페처의 전체적인 동작 흐름을 보여준다. 먼저 캐시 힛/미스, MSHR 힛이 발생한 블록 주소를 AMPM과 오프셋에 각각 입력한다. AMPM은 입력 즉시 프리페치 후보를 찾아내는 작업을 수행하지만 오프셋은 AMPM의 종료 전까지 기다림으로써 AMPM에 의한 프리페치가 우선적으로 처리되도록 한다. 각 AMPM과 오프셋에 프리페치 후보가 존재하면 MSHR에 미결된 요청 블록의 개수를 확인하고 현재 MSHR 임계값과 비교한다. 만약 MSHR 임계값보다 MSHR에 미결된 요청 블록 개수가 적으면 그 차이 여유 공간 개수만큼 프리페치 후보를 프리페치 큐에 삽입하는 것이 허용된다. 이후 중복되는 요청을 필터링하여 최종적으로 프리페치 후보 블록들을 요청한다. 그와는 달리 미결 요청 블록 개수가 MSHR 임계값보다 많으면 정확도와 관계없이 프리페치를 차단함으로써 프로세서의 요구 요청이 우선적으로 처리될 수 있도록 한다.

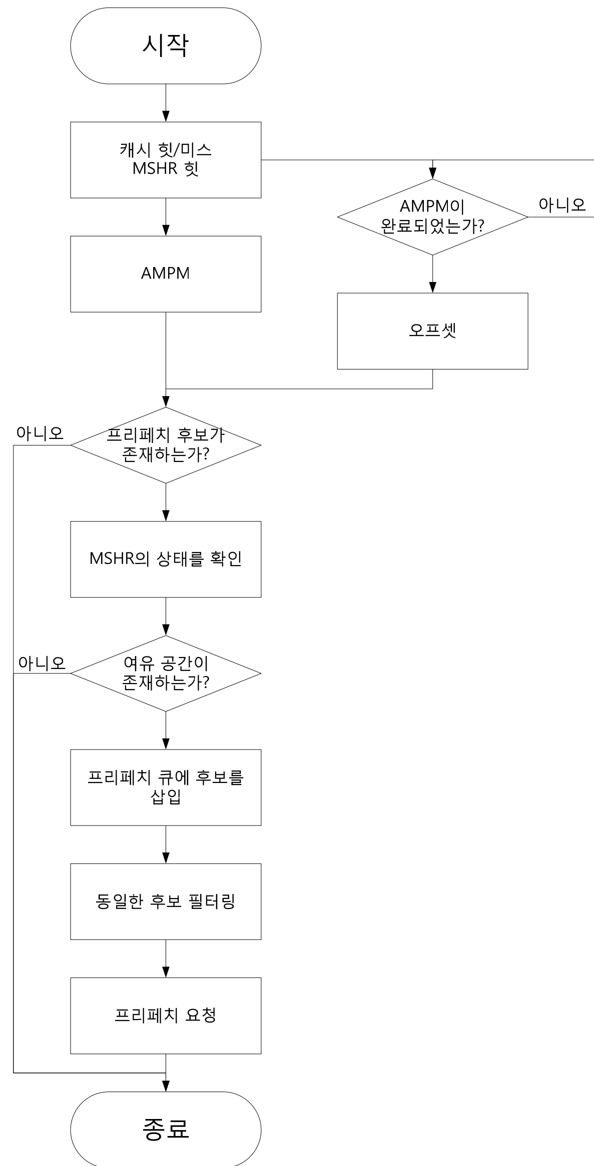

그림 8. 제안하는 프리페처 전체 동작 흐름도
Fig. 8. Overall Operation Flow Chart of Proposed Prefetcher.

Ⅳ. 성능 분석

이 장에서는 본 논문에서 제안하는 프리페처에 대한 성능을 평가하기 위한 실험환경을 정의하고 각 비교군 프리페처들의 IPC(instructions per cycle)를 측정하고, 이에 대한 분석 결과를 보인다.

4.1 실험 환경

제안하는 프리페처를 분석하기 위해 gem5 시뮬레이터[11]를 이용하여 프리페처를 구현하였으며, 벤치마크 프로그램은 PARSEC(Princeton Application Repository

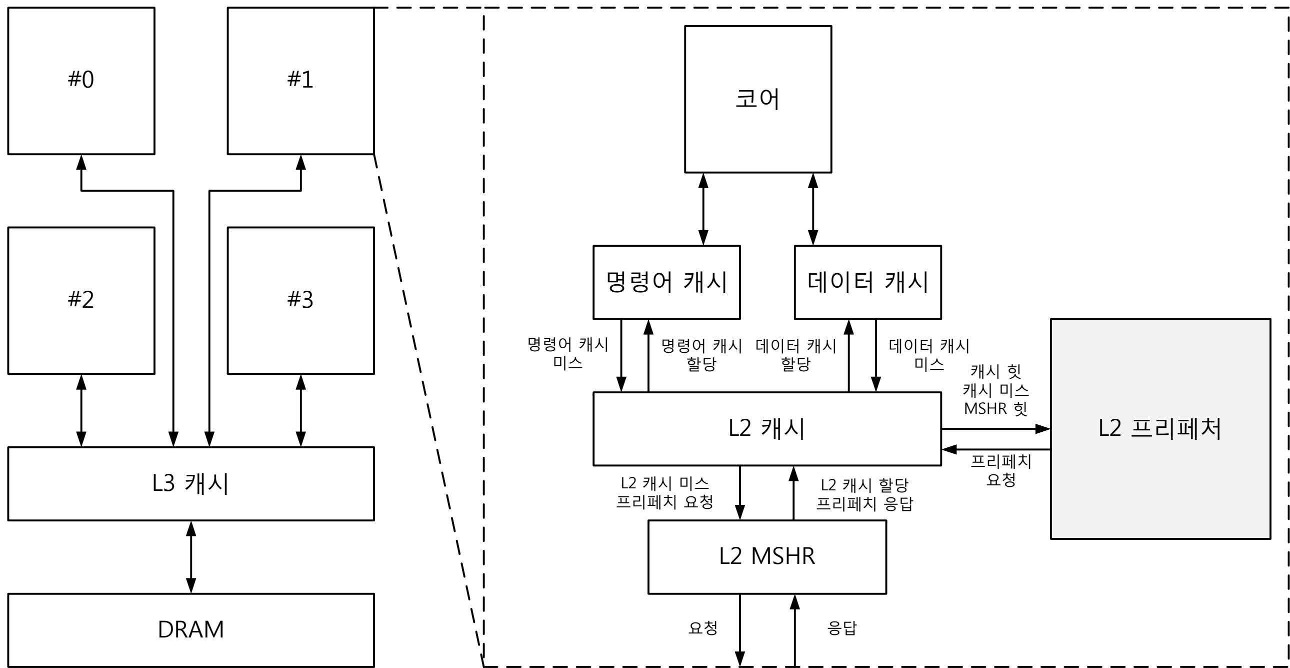

그림 9. 전체 시스템 블록도
Fig. 9. Block Diagram for whole system.

표 2. 시뮬레이션 시스템 구성
Table 2. System Configuration for Simulation.

Core	ARM ISA Quad core, 2.0GHz, OoO
Private I-Cache	32KB, 8-way 1ns hit latency
Private D-Cache	32KB, 8-way 2ns hit latency
Private L2 Cache	128KB, 8-way 12ns hit latency 16 MSHR
Shared L3 Cache	1MB, 16-way 32ns hit latency
Main Memory	2GB, 50ns access latency

for Shared-Memory Computers)[12]를 이용하였다. 그림 9는 시뮬레이션 전체 시스템의 구성을 나타낸 블록도이며 표 2는 각 시스템 컴포넌트의 상세 제원이다. CPU는 총 4개의 코어와 2GHz의 클록 속도를 가진 멀티코어 프로세서이다. 비순차 실행(OoO : Out of Order)으로 동작하며 ARM ISA(Instruction Set Architecture)를 사용한다. 공유 캐시인 L3 캐시를 제외한 L1, L2 캐시는 각 코어에 독립적으로 포함되어 동작한다. L1 캐시는 32KB 명령어 캐시와 32KB 데이터 캐시로 나뉘어 코어와 직접 통신한다. 128KB L2 캐시는 프리페처와 직접 연결되어 캐시 상에서 발생한 힛/미스/MSHR 힛 정보를 프리페처에 전달한다. 프리페처는 이 정보를 이용하여 다음 발생할 메모리 접근을 예측하고 해당 블록 요청을 수행한다. L2 MSHR은 총 16개이며 L2에서 미스가 발생한 요구 요청 또는 프리페치 요청을 저장하였다가 하위 계층으로 응답이 오면 제거한다. L3 캐시는 모든 코어가 공유하는 캐시로 1MB의 크기를 가진다. 모든 캐시의 캐시 블록 크기는 64KB로 고정되어 있다.

PARSEC은 멀티쓰레드 프로그램으로 구성된 벤치마크 모음으로 최신 워크로드에 초점을 맞추었으며 칩 멀티프로세서를 위한 차세대 공유메모리 프로그램을 대표할 수 있도록 설계되었다. 칩 멀티프로세서를 평가하기 위해 테스트용, 시뮬레이션용 등 다양한 입력 셋을 제공한다. 본 논문에서는 ARM 크로스 컴파일 문제 또는 인텔 프로세서에 의존성 문제로 인해 빌드가 불가능한 canneal, raytrace, x264을 제외한 나머지 blackscholes(재무분석), bodytrack(컴퓨터비전), dedup(기업 스토리지), facesim(애니메이션), ferret(이미지검색), fluidanimate(애니메이션), freqmine(데이터마이닝), streamcluster(데이터마이닝), swaptions(재무분석), vips(미디어프로세싱)을 사용하였다.

4.2 프리페처 저장 오버헤드

제안하는 프리페처는 코어당 18482 bits의 저장 공간을 요구한다. AMPM에서 페이지 테이블의 엔트리는 페이지 주소 64 bits, 접근 맵 64 bits, 프리페치 맵 64

bits, 타임스템프 16 bits 총 208 bits이며 테이블 엔트리의 개수는 64개이므로 페이지 테이블을 구성하는 데 필요한 저장용량은 13312 bits이다. 오프셋에서 샌드박스 엔트리는 오프셋 6 bits, 가상 프리페치 주소 32 bits 총 38 bits이며 엔트리 개수는 120개이므로 샌드박스의 저장용량은 4560 bits이다. 그리고 32개 오프셋에 대한 점수를 저장하는 스코어보드는 엔트리가 각각 오프셋 6bits와 점수 10 bits를 차지하므로 512 bits를 가진다. 프리페치 후보 테이블의 엔트리는 스코어보드 엔트리와 동일하며 4개의 엔트리를 가지므로 저장용량은 64 bits이다. 또한 프리페치 힛 카운터 10 bits, 캐시 미스 카운터 10 bits, 접근 횟수 카운터 10 bits, MSHR 임계값 4 bits 등 파라미터를 저장할 공간이 추가로 필요하다.

AMPM 프리페처는 코어당 32036 bits의 저장 공간을 필요로 한다. 256개의 접근 주소를 관리할 수 있는 접근맵을 가진 52개의 엔트리를 저장하기 위해 메모리 접근 맵 테이블에만 전체 저장 공간 오버헤드의 90%인 약 29147 bits의 저장 공간을 필요로 한다. 나머지 공간은 프리페치 MSHR, 동적 스트림 필터, 스트림 길이 히스토그램, 파이프라인 레지스터 등을 위해 사용한다. AMPM이 정확하게 동작하려면 존 내의 요구 요청 접근과 프리페치 결과를 가능한 한 많이 기록할 수 있는 공간이 필요하지만 메모리 접근 패턴이 항상 존 내의 모든 공간에 기록되는 것이 아니기 때문에 저장 공간을 비효율적으로 사용할 수 있다. 이에 반해 제안하는 프리페처의 AMPM은 존을 페이지 단위로 축소하고 오래된 페이지 정보가 쉽게 교체할 수 있도록 함으로써 AMPM 프리페처의 장점을 살리면서 저장공간 또한 효율적으로 사용할 수 있다.

BO 프리페처는 코어당 4361 bits의 매우 작은 저장공간을 필요로 한다. 프리페치 여부를 저장하기 위한 버퍼가 2048 bits, 샌드박스가 1536 bits로 높은 비중을 차지하며 나머지 공간은 각종 파라미터를 저장하기 위해 필요하다. 작은 저장 오버헤드에도 불구하고 우수한 프리페처 성능을 보일 수 있는 것은 BO 프리페처와 같은 오프셋 프리페처가 가진 가장 큰 장점이다. 이 때문에 다른 프리페처와 결합해도 저장 용량 증가가 부담되지 않는다. 제안하는 프리페처의 오프셋은 프리페치 힛 발생 횟수를 이용하여 MSHR 임계값을 조절하고 캐시 미스 발생 횟수를 이용하여 프리페치 사용을 제어함으로써 오프셋 프리페치가 성능 향상에 기여할 수 있는 메모리 접근 패턴에서만 동작하도록 구현하였다.

엑스퍼트 프리페치 예측은 코어당 32768 bits의 공간을 필요로 한다. 포화 카운터를 저장하기 위해 엔트리당 2 bits가 필요하며, 4개의 엑스퍼트가 4096개의 엔트리를 가지고 있다. 이로 인해 AMPM 프리페처와 같이 큰 저장공간을 요구하는 프리페처와 결합하는 경우 저장 오버헤드가 심각하게 가중된다는 문제가 있다. 제안하는 프리페처는 구현이 복잡하고 저장공간을 많이 차지하는 필터 시스템을 활용하는 대신 AMPM 프리페처가 생성한 프리페치 요청을 그대로 사용하되 저장공간을 적게 차지하는 오프셋 프리페처의 정보를 이용하여 프리페치 양을 조절하는 방법을 이용한다.

4.3 시뮬레이션 결과 및 분석

4.3.1 AMPM과 오프셋을 단독으로 사용하였을 때 성능 비교 분석

그림 10은 제안하는 프리페처의 AMPM과 오프셋을 단독으로 사용하였을 때의 IPC 속도 증가이다. MSHR 임계값에 의한 프리페치 제어는 동일하게 적용되었으며 오프셋의 경우 낮은 점수의 오프셋을 프리페치 후보로 사용하지 않도록 설정하였다. IPC 속도 증가값은 프리페처를 사용하지 않을 때 대비 IPC 증감을 나타내는 값으로 프리페처를 사용하지 않았을 때의 IPC는 1로 정규화된다. 실험 결과 AMPM은 오프셋보다 blacksholes (0.01), bodytrack (0.04), dedup (0.13), facesim (0.95), ferret (0.07), fluidanimate (0.07), freqmine (0.12), streamcluster (0.11), swaptions (0.04), vips (0.38) 높은 IPC 속도 증가값을 가졌다. 전체 평균에서 AMPM이 오프셋에 비해 평균 19% 높은 IPC 속도 증가값을 보였

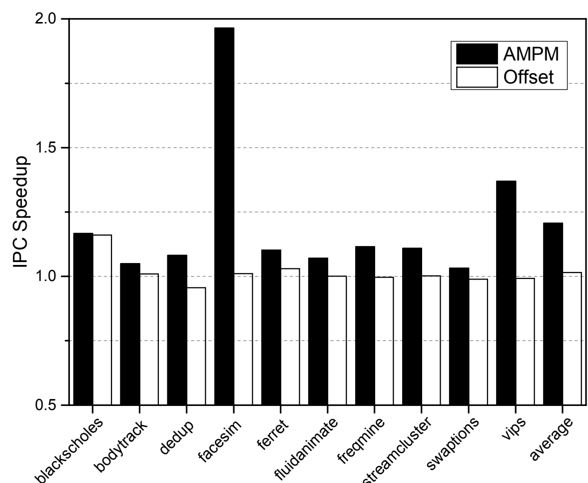

그림 10. AMPM과 오프셋 IPC 속도 증가 비교
Fig. 10. IPC Speedup Comparison between AMPM and Offset.

다. 이를 통해 공격적인 프리페치 정책을 가지고 있는 오프셋을 단독으로 사용할 경우 점수가 낮거나 부정확한 프리페치를 수행하는 등 다양한 워크로드의 메모리 접근 패턴에 대응하기 어려워 비약적인 프로세서의 성능 향상을 기대하기 어렵다는 것을 알 수 있다.

따라서 본 논문에서는 성능이 우수한 AMPM 프리페처를 기본 프리페처로 사용하되 오프셋 프리페처가 유리한 메모리 접근 패턴에서만 활성화하는 프리페처를 구현하였다.

4.3.2 MSHR 임계값 사용에 따른 성능 변화 분석

그림 11은 제안하는 프리페처에서 MSHR 임계값을 동적으로 조절하는 기법을 적용하였을 때와 그렇지 않았을 때의 IPC 속도 증가이다. MSHR 임계값은 현재 메모리 접근 패턴을 알려주는 척도이다. MSHR의 요청 처리가 지연될 때 MSHR 임계값을 낮춤으로써 필요한 요구 요청이 프리페치 요청에 의해 방해받는 것을 방지할 수 있으며 요구 요청이 적을 때 MSHR 임계값을 높임으로써 프리페치로 얻을 수 있는 이득을 최대화한다. 실험 결과 MSHR이 16개일 경우 MSHR 임계값을 사용하여 프리페처를 제어하였을 때 사용하지 않을 때보다 전체 평균 3.4% IPC 속도 증가값이 향상되었으며, MSHR이 8개일 경우 전체 평균 3.3% IPC 속도 증가값이 향상되었다. 이를 통해 제안하는 프리페처와 같이 MSHR 임계값으로 프리페치 양을 조절하는 것이 프로세서 처리 성능 향상에 기여한다는 것을 알 수 있다. 다만 MSHR 16개일 때 bodytrack에서 제안하는 프리페처는 MSHR 임계값으로 프리페처를 제어할 때보다 0.06 낮은 IPC 속도 증가값을 가졌다. Bodytrack은 사람 신체의 3D 위치를 추적하는 워크로드로 오브젝트 추출을 위해 에지 탐색을 수행한다. 이 때 에지 탐색을 위해 3x3 마스크를 사용하기 때문에 프리페처가 접근 패턴을 쉽게 인지할 수 있다. 그러나 제안하는 프리페처는 마스크에 사용되는 3개 블록을 초과하는 불필요한 프리페치를 추가하여 MSHR 힛이 늘어나고 이에 MSHR 임계값을 줄이면서 프리페치 기회를 잃게 된다. 이로 인해 MSHR 임계값을 제어하지 않을 때보다 다소 낮은 IPC 속도 증가값을 가졌다. 하지만 전체적으로 보았을 때 MSHR 임계값의 조절이 현재 메모리 접근 상황과 MSHR 용량에 맞추어 프리페치를 요청하거나 차단함으로써 성능 향상에 기여함을 보여준다.

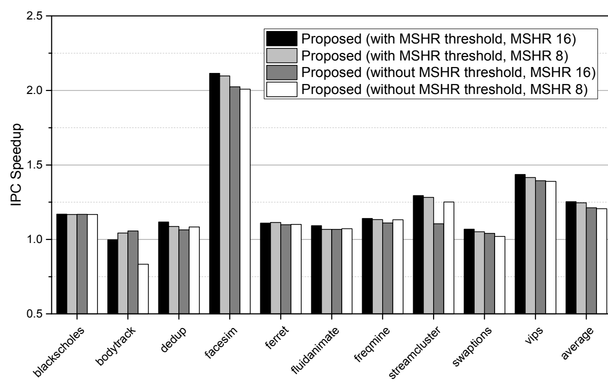

그림 11. MSHR 임계값 사용에 따른 IPC 속도 증가 비교
Fig. 11. IPC Speedup Comparison using MSHR threshold.

4.3.3 오프셋 개수에 따른 성능 변화 분석

그림 12는 제안하는 프리페처에서 스코어보드에 기록하는 오프셋의 개수를 각각 32(제안하는 프리페처에서 사용하는 개수), 16, 64개로 변화시켰을 때의 IPC 속도 증가이다. 실험 결과 오프셋 개수에 의한 성능 차이는 크게 발생하지 않았다. 오프셋 16개를 사용하였을 때 가장 높은 IPC 속도 증가값을 가졌으나 오프셋 32개를 사용할 때에 비해 0.001의 미세한 차이를 보였다. 이것은 오프셋의 커버리지와 관계없이 1, 2, -1 등 작은 오프셋의 발생 빈도가 집중되기 때문이다. 다만 성능상의 차이가 없음에도 16개 오프셋을 사용할 때보다 저장 오버헤드가 높은 32개의 오프셋을 사용하는 이유는 오프셋의 평가가 가능한 LOW_SCORE 이상의 점수를 가진 오프셋이 －16 ~ 16 범위 내에 존재하며 그 외 범위에는 존재하지 않음을 시뮬레이션을 통해 확인하였기

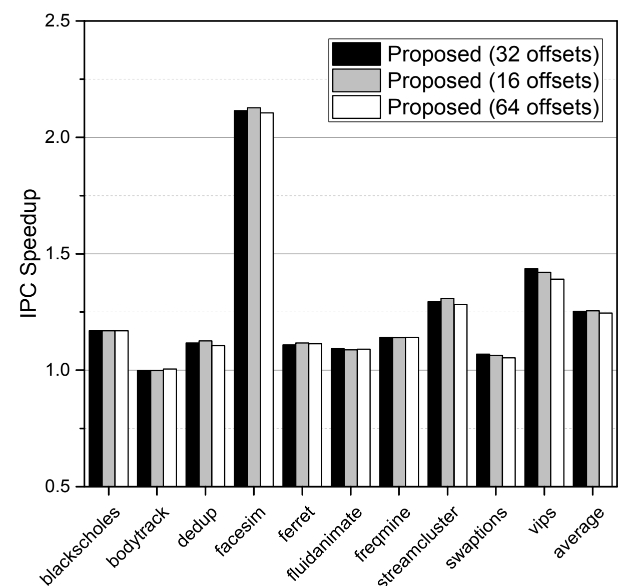

그림 12. 오프셋 개수에 따른 IPC 속도 증가 비교
Fig. 12. IPC Speedup Comparison with the number of offsets.

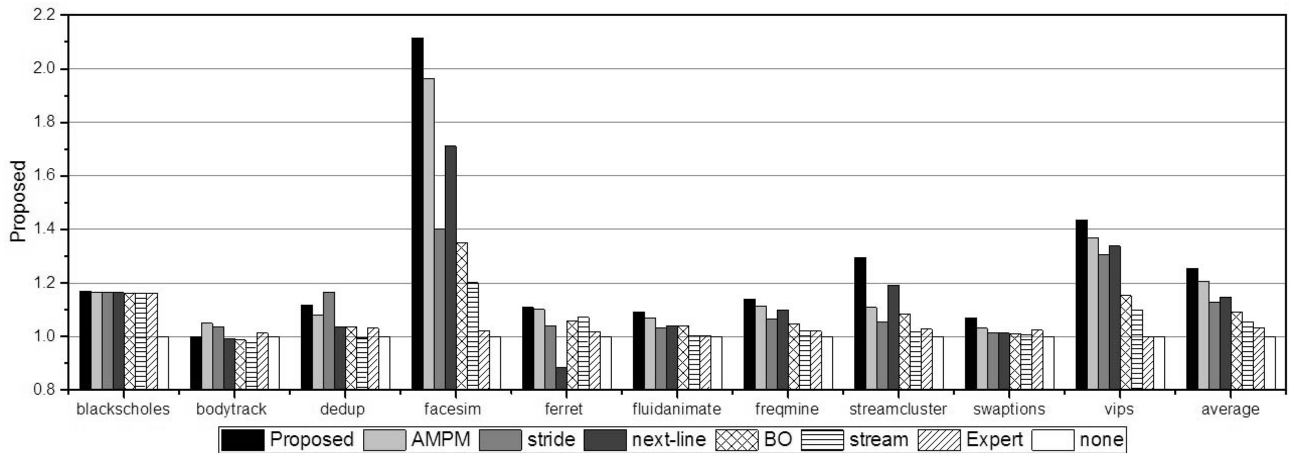

그림 13. 프리페처 사용에 따른 IPC 속도 증가 비교
Fig. 13. IPC Speedup Comparison using Prefetchers.

때문이다.

4.3.4 전통적인 프리페처, 최신 프리페처, 제안하는 프리페처의 성능 비교 분석

마지막으로 next-line, 스트림, 스트라이드, AMPM, BO, 익스퍼트 프리페치 예측 총 6개의 비교군과 제안하는 프리페처를 L2 캐시에 적용하였을 때의 IPC 속도 증가를 측정 및 분석하였다. none은 프리페처를 사용하지 않는 경우이다.

그림 13은 프리페처 사용에 따른 IPC 속도 증가를 비교한 것이다. 제안하는 프리페처는 bodytrack (-0.05)를 제외하고 AMPM 프리페처에 비해 blackscholes (0.002), dedup (0.04), facesim (0.14), ferret (0.01), fluidanimate (0.02), freqmine (0.19), streamcluster (0.04), swaptions (0.07), vips (0.07) 높은 IPC 속도 증가값을 가졌으며 전체적으로 평균 4% 높은 IPC 속도 증가값을 가졌다. 제안하는 프리페처가 AMPM 프리페처보다 적은 저장 오버헤드를 가지고 있음에도 성능이 우수한 이유는 페이지가 빈번하게 교체되거나 페이지 내에 접근이 많지 않은 경우 AMPM을 통한 프리페치가 원활히 이뤄지지 않는데 이 때 학습을 통해 가장 발생 확률이 높은 오프셋을 얻어내는 오프셋 프리페처를 통해 AMPM이 다루지 못하는 프리페치 공백을 채워줄 수 있기 때문이다.

제안하는 프리페처는 BO 프리페처에 비해 blackscholes (0.01), bodytrack (0.01), dedup (0.08), facesim (0.76), ferret (0.05), fluidanimate (0.05), freqmine (0.09), streamcluster (0.21), swaptions (0.06), vips (0.28) 높은 IPC 속도 증가값을 가졌으며, 전체적으로 평균 14.7% 높은 IPC 속도 증가값을 가졌다. BO 프리페처는 접근 당 프리페치를 한 번만 요청하기 때문에 제안하는 프리페처에 비해 공격성이 떨어져 이득을 취할 수 있는 상황에서도 최대한의 이득을 적극적으로 이끌어내지 못한다. 또한 설정된 임계값보다 점수가 낮은 경우 프리페치 요청을 수행할 수 없다. 이에 반해 제안하는 프리페처는 기본적으로 AMPM으로 동작하기 때문에 오프셋의 신뢰도가 떨어질 때에도 프리페치를 지속적으로 수행할 수 있다.

제안하는 프리페처는 bodytrack(-0.014)를 제외하고 엑스퍼트 프리페치 예측에 비해 blackscholes (0.007), dedup (0.08), facesim (1.09), ferret (0.09), fluidanimate (0.09), freqmine (0.12), streamcluster (0.27), swaptions (0.045), vips (0.43) 높은 IPC 속도 증가값을 가졌으며, 전체적으로 평균 21% 높은 IPC 속도 증가값을 가졌다. 엑스퍼트 프리페처 예측[9] 논문에서는 저자가 제안하는 가중치-우선 필터와 AMPM을 결합하였을 때 AMPM 보다 8% 높은 IPC 속도 증가값을 가지고 있다고 언급하였으나 본 논문의 실험 환경에서는 다른 프리페처들을 통틀어 가장 낮은 IPC 속도 증가값을 가졌다. 이는 [9]와 본 논문의 실험환경에 의한 차이 때문이다. [9]에서는 프로세서가 2레벨의 캐시 계층을 가지고 있으며 공유 캐시인 L2 캐시에 프리페처를 연동하고 있으나, DPC-2의 시뮬레이션 환경을 따르는 본 논문의 경우 프로세서가 3레벨의 캐시 계층을 가지고 있으며 개별 캐시인 L2 캐시에 프리페처를 연동하였다. 엑스퍼트 프리페처 예측은 교체되는 캐시 블록 정보가 존재해야 필터 정보를 업데이트할 수 있다. 따라서 [9]와 같이 공유 캐시에 프리페처를 연동하면 교체되는 캐시 블록 정보가

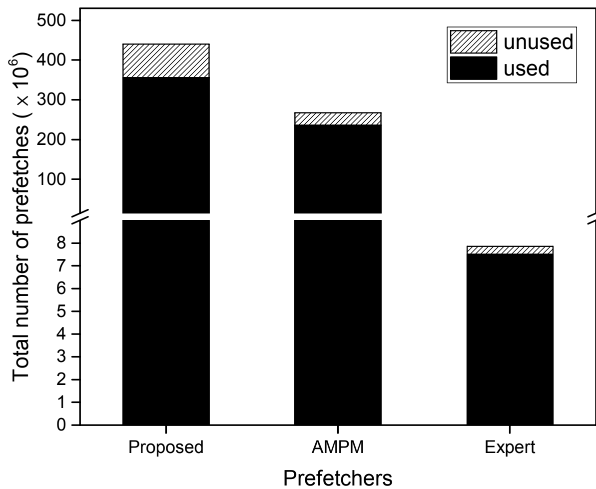

그림 14. 프리페치 블록 유용성 비교
Fig. 14. Prefetch Block Usefulness Comparison.

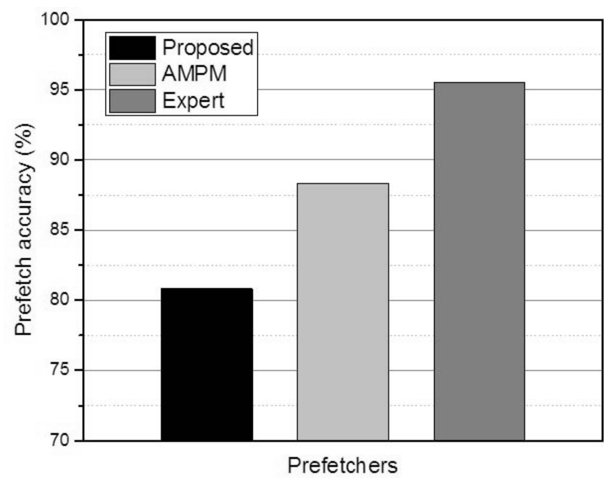

그림 15. 프리페치 정확도 비교
Fig. 15. Prefetch Accuracy Comparison.

많아 필터의 업데이트가 빠르게 이루어지지만 본 논문과 같이 개별 캐시에 프리페처를 연동하면 교체되는 캐시 블록 정보가 적어 현재 메모리 패턴을 반영하는 필터가 완성되지 않는다. 이로 인해 가중치-우선 필터가 AMPM이 생성하는 프리페치 요청을 대다수 차단하여 프리페치의 이점을 잃게 되었다.

4.3.5 프리페치 정확도 평가 및 분석

그림 14는 PACSEC 벤치마크 시뮬레이션 과정 중에 프리페처가 생성한 모든 프리페치 블록들 중 프로세서에 의해 사용된 것과 사용되지 않은 것의 총합을 나타낸 것이다. 비교 대상은 제안하는 프리페처가 기반으로 하는 AMPM과 가중치-우선 필터+AMPM을 사용하는 엑스퍼트 프리페치 예측으로 제한하였다. 제안하는 프리페처는 총 4.4억 개의 프리페치 블록 중 약 3.5억 개의 블록이 사용되었고 약 8400만개의 블록이 사용되지 않고 제거되었다. AMPM은 총 2.6억 개의 프리페치 블록 중 약 2.3억 개의 블록이 사용되었고 약 3000만개의 블록이 사용되지 않고 제거되었다. 엑스퍼트 프리페치 예측의 경우 총 785만개의 프리페치 블록 중 약 750만 개의 블록이 사용되었고 약 35만개의 블록이 사용되지 않고 제거되었다. 그림 15는 프리페치의 정확도를 나타내는 그림이다. 프리페치 정확도는 요청한 프리페치 블록 개수와 한 번 이상 참조된 프리페치 블록의 비율로 계산한다. 제안하는 프리페처는 80.8%, AMPM은 88.32%, 엑스퍼트 프리페치 예측은 95.48%의 프리페치 정확도를 가졌다. 엑스퍼트 프리페치 예측은 가중치-우선 필터를 통해 사용되지 않는 프리페치의 개수를 크게 줄이는 효과가 있다. 그러나 프리페치 정확도에 치중한 나머지 프리페치 요청을 제한하면서 프리페치로 얻을 수 있는 성능 이득을 크게 잃어버렸다. 엑스퍼트 프리페치 예측은 제안하는 프리페처의 5.6배, AMPM의 3.3배 적은 프리페치량을 보였다. 그에 반해 제안하는 프리페처는 AMPM 대비 1.7배 많은 프리페치를 요청하였고 2.8배 많은 프리페치 블록이 참조되지 않고 버려졌으나 AMPM보다 약 4% 높은 성능을 보였다. 따라서 프로세서 성능 향상을 위해서는 부정확한 프리페치 요청을 억제하는 것도 중요하지만 제안하는 프리페처와 같이 프리페치가 정확하다고 판단하면 사용 자원 여유만큼 공격적으로 프리페치를 수행함으로써 예측 범위를 확장시켜야 한다. 다만 프리페치량이 증가하면 프로세스 개수가 증가할수록 캐시 대역폭을 많이 차지하고 전력을 많이 소모하므로 실제 구현에서는 이러한 트레이드-오프 관계를 반드시 고려해야 한다.

V. 결 론

하드웨어 프리페처는 예측을 통해 프로세서의 요청 이전에 캐시에 블록을 가져옴으로써 메모리 지연을 숨긴다. 그러나 프로세서 아키텍처의 변화와 고성능 컴퓨팅 워크로드들의 등장으로 메모리 접근 패턴을 예측하기 어려워졌다. 이러한 문제를 극복하기 위해 다양한 하드웨어 프리페치 기법들이 제안되었다. 본 논문에서 제안하는 하드웨어 프리페처는 최신 프리페처인 AMPM과 오프셋 프리페처를 적절하게 결합한 것으로 AMPM의 안정적인 프리페칭 기법에 추가로 공격적인 오프셋 프리페칭을 더하여 프리페치 성능을 극대화하였다. 다만 오프셋 프리페처는 메모리 접근 패턴 변화에

민감하므로 이를 제어하기 위해 학습 기간 동안 발생한 MSHR 힛 발생 횟수를 이용하였다.

제안하는 프리페처의 성능을 평가하기 위해 gem5 시뮬레이터와 PARSEC 벤치마크를 이용하였다. 그리고 전통적인 하드웨어 프리페처들과 최신 프리페처들을 gem5 시뮬레이터에서 동작하도록 함께 구현하여 제안하는 프리페처와 성능을 비교분석하였다. 제안하는 프리페처는 AMPM 프리페처에 비해 평균 4% 높은 IPC를 보였고, BO 프리페처에 비해 평균 14.7% 높은 IPC를 보였으며, 엑스퍼트 프리페치 예측에 비해 평균 21% 높은 IPC를 보였다.

REFERENCES

[1] B. Falsafi and T. F. Wenisch, A Primer on Hardware Prefetching, Morgan & Claypool Publisher, p. 1-5. 2014.

[2] S. P. Vanderwiel and D. J. Lilja, “Data prefetch mechanisms,” *ACM, Computing Surveys.*, vol. 32, no. 2, pp. 174-199, Jun 2000.

[3] Y. S. Jeong, J. H. Kim, T. H. Cho, and S. B. Choi, “Instructions and Data Prefetch Mechanism using Displacement History Buffer,” *Journal of The Institute of Electronics Engineers of Korea*, vol. 52, no. 10, pp 82-94, Oct 2015.

[4] D. Y. Jung and Y. S. Lee, “Cache Replacement Policy Based on Dynamic Counter for High Performance Processor,” *Journal of The Institute of the Electronics Engineers of Korea*, vol. 50, no. 4, pp. 52-58, Apr 2013.

[5] The 1st JILP Data Prefetching Championship (DPC-1) Available at : http://www.jilp.org/dpc/

[6] The 2nd Data Prefetching Championship (DPC2) Available at : http://comparch-conf.gatech.edu/dpc2/

[7] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, “Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers,” *in Proc. of IEEE Conf International Symposium on High Performance Computer Architecture*, pp. 10-14, Scottsdale, USA, Feb 2007.

[8] S. H. Pugsley, Z. Chishti, C. Wilterson, P. f. Chuang, R. L. Scott, A. Jaleel, S. L. Lu, K. Chow, and R. Balasubramonian, “Sandbox Prefetching: Safe Run-Time Evaluation of Aggressive Prefetchers,” *in Proc. of IEEE Conf. International Symposium on High Performance Computer Architecture*, pp. 15-19, Orlando, USA, Feb 2014.

[9] B. Panda and S Balachandran, “Expert Prefetch Prediction: An Expert Predicting the Usefulness of Hardware Prefetchers,” *in IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 13-16, Jan.-June 1 2016.

[10] X. Zhuang and H. H. S. Lee, "A hardware-based cache pollution filtering mechanism for aggressive prefetches," *in Proc International Conference on Parallel Processing*, pp. 286-293. Kaohsiung, Oct 2003.

[11] N. Binkert, S. Sardashti, R. Sen et al, “The gem5 simulator,” *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1-7, May 2011.

[12] 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, Toronto, Canada, Oct 2008.

저 자 소 개

김 강 희(학생회원)
2011년 인하대학교 전자공학과 학사 졸업.
2013년 인하대학교 전자공학과 석사 졸업.
2013년~현재 인하대학교 전자공학과 박사과정.
<주관심분야 : 컴퓨터 네트워크, 무선 센서 네트워크, SoC>

송 경 환(학생회원)
2015년 한남대학교 컴퓨터공학과 학사 졸업.
2016년~현재 인하대학교 전자공학과 석사과정.
<주관심분야 : 컴퓨터 구조, SoC, 임베디드 시스템>

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>

박 태 신(학생회원)
2015년 인하대학교 전자공학과 학사 졸업.
2016년~현재 인하대학교 전자공학과 석사과정.
<주관심분야 : 컴퓨터 구조, SoC, 무선 센서 네트워크>

윤 동 성(학생회원)
2015년 인하대학교 전자공학과 학사 졸업.
2016년~현재 인하대학교 전자공학과 석사과정.
<주관심분야 : 컴퓨터 구조, SoC, 무선 센서 네트워크>