인접 영역 테이블을 이용한 다중 간격 프리페치 기법

심재성*, 전호윤*, 이용석*
*연세대학교 전기전자공학과
e-mail : sjshsboy, hockma24, yonglee@yonsei.ac.kr

Multi-Strided Prefetching Using Adjacent Region Table

Jae-Seong Shim*, Ho-Yoon Jun*, Yong-Surk Lee*
*School of Electrical & Electronic Engineering, Yonsei University

요 약

프로세서와 메모리 간의 속도 차이로 인해 메모리 시스템의 성능 향상이 프로세서의 성능을 높이기 위한 중요한 요인이 되었고, 이를 위해 캐시 미스율을 감소시키는 방법이 연구되고 있다. 데이터 프리페치는 캐시의 미스율을 감소시키는 기법 중 하나이며 실제로 최근 프로세서에서 메모리 시스템의 성능을 향상시키기 위해 사용된다. 데이터 프리페치를 효과적으로 수행하기 위해서 메모리 주소의 접근 패턴을 파악하는 것이 중요하며, 이를 위해 순차적으로 접근하는 경우, 한 종류의 1 보다 크거나 같은 간격(stride)으로 뛰면서 접근하는 경우, 다수의 간격이 규칙적으로 반복되며 접근하는 경우 등의 다양한 패턴을 찾는 프리페치 기법들이 등장했다. 본 논문에서 소개하는 다중 간격 프리페치의 경우, 메모리 공간을 메모리 주소의 일부 상위 비트를 통해 여러 개의 영역으로 나누고, 하나의 패턴을 하나의 영역 안에서만 학습하여, 다른 영역에 속한 메모리 주소 접근 시 현재 학습하는 패턴에 어긋나는 주소라고 여기기 때문에 학습을 방해하지 않도록 하였다. 그러나 이 방법은 영역의 크기보다 같은 패턴을 갖는 메모리 주소 스트림의 크기가 더 클 때, 접근 주소의 영역이 바뀜으로 인해 불필요한 학습을 추가적으로 해야 하는 문제점이 있다. 이에 본 논문에서 인접 영역 테이블(ART: Adjacent Region Table)을 이용하여 같은 패턴을 갖는 메모리 접근 스트림의 크기가 영역의 크기보다 클 경우, 기존의 학습된 패턴대로 프리페치를 수행할 수 있도록 하였다. 본 논문에서 제안한 알고리즘으로 실험한 결과, 기존의 다중 간격 프리페치보다 캐시 미스율을 약 6.7% 낮췄고, 시스템 전체의 성능의 지표인 IPC 의 경우, 약 5.78% 높아지는 성능 향상의 결과를 얻었다.

1. 서론

프로세서와 메모리 간의 속도차이가 커짐에 따라 프로세서의 성능 향상을 위해 메모리 시스템의 성능을 높이는 것이 큰 관건이 되었다. 메모리 시스템의 성능을 높이기 위해서 프로세서와 메인 메모리 사이에 메모리의 데이터 중 일부만 저장하는 캐시메모리가 등장하였다[1].

만약 캐시에 프로세서가 요청한 데이터가 존재하지 않는 경우, 하위 레벨 캐시나 메인 메모리로 데이터를 요청하여 가져오는 경우에 시간이 오래 걸리므로 프로세서의 성능을 낮추는 주 요인이 된다. 이를 캐시 미스라 하고, 캐시 접근 횟수 대비 캐시 미스의 비율(캐시 미스율)을 줄여 프로세서의 성능을 높이는 방법이 연구되고 있다. 데이터 프리페치는 그 연구 중 하나로 실제로 최근에 출시되는 프로세서에서 메모리 시스템의 성능을 향상시키기 위해 데이터 프리페치를 사용한다[1].

데이터 프리페치는 이전 메모리 주소의 접근 패턴을 학습하고 그 학습의 결과로 프로세서가 앞으로 요청할 가능성이 있는 주소와 데이터를 미리 하위 레벨 캐시나 메인 메모리로부터 인출하는 것을 말한다. 앞으로 사용할 주소를 정확히 예측하여 캐시에 미리 인출했다면 프로세서는 캐시 미스를 피할 수 있게 되어, 미스로 인해 겪어야 할 프로세서의 지연시간을 숨김으로써 성능을 향상시킬 수 있다.

데이터 프리페치를 수행하기 위해서는 메모리 접근 패턴을 파악하는 알고리즘이 필요하다. 가장 간단한 방법으로는 참조된 블럭의 다음 주소에 해당하는 블럭을 프리페치 하는 것이다. 이러한 방식을 순차적 프리페치(Sequential Prefetch) 혹은 단위 간격 프리페치(Unit-Strided Prefetch)라고 한다. 위의 경우는 순차적으로 캐시 블럭을 접근하는 패턴만 탐지할 수 있다. 이를 극복하기 위하여 순차적 접근뿐만 아니라, 2 칸 이상 건너 뛴 블럭을 참조하는 패턴까지 탐지할 수 있는 비단위 간격 프리페치(Non-Unit Strided Prefetch) 방식이 등장하였다. 비단위 간격 프리페치는 한 가지의 간격(stride)만 탐지하여 프리페치하기 때문에, 다수의 간격을 가진 패턴을 탐지하지 못한다. 이러한 문제를 해결하기 위해 다중 간격 프리페치 (Multi-Strided Prefetch)가 제안되었다. 다중 간격 프리페치는 다수의 간격을 저장한 후, 이를 이용해 프리페치하는 기법이다.

다중 간격 프리페치는 간격의 패턴을 학습하며 이

학습에 참여하는 메모리 주소의 범위를 메모리 공간을 일정하게 나눈 하나의 영역(Region) 이내로 제한한다. 그 이유는 다른 영역의 메모리 주소는 현재 참조되는 메모리 주소의 패턴과 상관이 없다고 판단이 되기 때문에 학습에서 배제하기 위함이다. 그러나 하나의 학습과정을 하나의 영역 이내로 제한했을 때, 패턴이 유지되는 메모리 주소의 스트림이 하나의 영역보다 크다면 영역이 바뀌는 시점부터 학습을 다시 해야 하는 문제가 발생한다.

이 문제를 해결하기 위해, 본 논문에서는 인접 영역 테이블(ART: Adjacent Region Table)을 제안하였다. ART 는 하나의 영역에 대해서 공간적으로 다음 영역과 이전 영역을 가리키는 태그값을 저장하고 이를 이용해 메모리 접근 스트림의 접근 영역이 다음 영역 또는 이전 영역으로 바뀔 때, 추가적인 학습 없이 프리페치를 진행할 수 있도록 하였다.

2. 관련 연구

2.1 태그를 이용한 프리페치(Tagged Prefetch)

태그를 이용한 프리페치[2]는 대표적인 순차적 프리페치 기법이다. 태그는 블럭마다 존재하며, 블럭이 프로세서의 요구나 프리페치에 의해서 캐시에 인출된 후 프로세서에 의해 접근이 되었는지 표시하는 한 비트를 의미한다. 하나의 블럭이 프리페치 되거나 캐시 미스로 인해 교체되었을 때, 태그 비트를 0 으로 설정한다. 프로세서에 의해 접근이 되면 태그 비트를 1 로 설정한 후, 다음 주소의 블럭을 프리페치한다. 이 알고리즘은 공간적 지역성이 뚜렷한 구간에서 효과가 있으나, 탐지할 수 있는 접근 패턴이 순차적인 접근 밖에 없다는 한계가 있다.

2.2 참조 예측 테이블을 이용한 프리페치

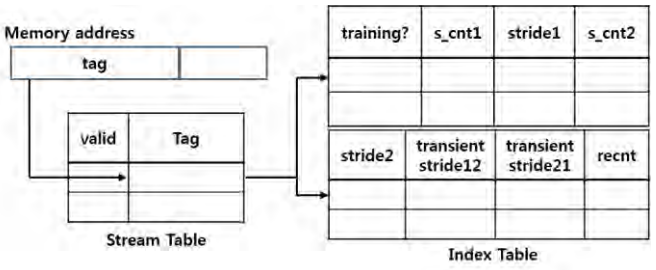
참조 예측 테이블(Reference Prediction Table)[3]의 구조는 하나의 PC 값(명령어의 주소)에 대해, 바로 이전에 접근한 주소 및 바로 이전에 접근한 주소와 현재 접근한 주소의 간격, 그리고 프리페치 상태(prefetch state)를 저장하도록 되어 있다. 프리페치 상태가 갖는 상태의 종류에는 초기상태(init state), 과도상태(transient state), 프리페치를 수행하지 않는 상태(no prefetch state), 프리페치를 수행하는 상태 (steady state) 가 있다. 참조 예측 테이블에서 참조되는 메모리 주소의 간격이 규칙적으로 나타나는지 혹은 불규칙적으로 나타나는지 관찰한 후, 규칙적이면 프리페치 상태가 프리페치를 수행하는 상태로 바뀌고, 불규칙적이면 프리페치를 수행하지 않는 상태로 바뀐다. 위의 태그를 이용하는 프리페치가 바로 다음 주소의 블럭만 프리페치를 할 수 있다면, 참조 예측 테이블은 1 이상의 간격을 뛴 주소의 블럭도 프리페치할 수 있다. 또한 프리페치 상태를 통해 학습함으로써 참조 예측 테이블에 있는 간격을 이용해 정확한 주소를 프리페치할 수 있는지 판단하고, 그렇지 않은 경우 프리페치를 중단하여 잘못된 프리페치로 인한 불필요한 메모리 트래픽의 증가를 방지할 수 있다.

3. 다중 간격 프리페치

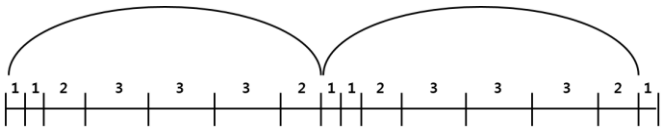
Iacobovici 가 제안한 다중 간격 프리페치[4]의 구조는 (그림 1)과 같이 stream table, index table 로 구성이 되어있다. 이 구조에서 발견하고자 하는 패턴은 (그림 2)와 같이 다수의 간격이 규칙적으로 나타나는 경우이다. 이 구조를 통해 순차적인 메모리 주소에 대한 프리페치, 1 이상의 간격만큼 떨어진 주소에 대한 프리페치를 포함하여 다수의 간격이 존재하는 경우에 대한 프리페치까지 모두 수행할 수 있다.

Iacobovici 는 메모리 주소의 일부 상위 비트인 태그(tag)를 사용해 메모리 공간을 여러 개의 영역으로 나누고, 패턴에 대한 학습을 각 영역 안에서만 수행하도록 제한하였다. 메모리 주소간의 간격을 이용해 프리페치를 하는 경우, 메모리 접근주소의 공간적 지역성을 이용하기 때문에 하나의 영역 안에서 패턴이 탐지되는 경우가 많다. 또한 패턴을 학습하는 중에 다른 영역의 주소가 접근하는 상황이 발생해도 학습을 방해하지 못하기 때문에 패턴 학습이 지속될 수 있다.

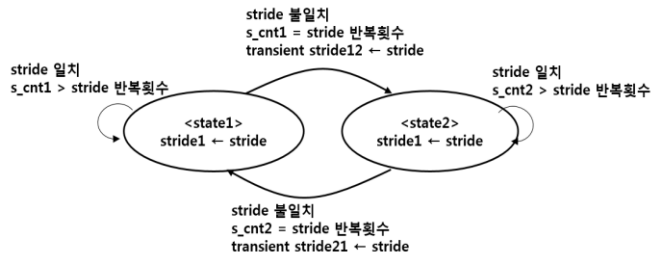
다중 간격 프리페치의 동작은 다음과 같다. 캐시 미스가 발생한 경우, 미스를 발생시킨 메모리 주소의 태그를 stream table 에 저장된 각 엔트리의 태그와 비교한다. 만약 일치하는 태그가 없다면 무효한 엔트리 혹은 접근한 지 가장 오래된 엔트리에 할당하고, 태그가 있는 경우 index table 에서 접근한 태그와 같은 위치에 있는 엔트리를 학습한다. 학습 중에 정상상태



(그림 1) stream table, index table



(그림 2) 다수의 간격이 반복되는 패턴의 예



(그림 3) 다중 간격 프리페치의 동작

간격(steady stride) 2 개와 과도상태 간격(transient stride) 2 개, 이렇게 4 개의 간격을 저장할 수 있고, 각 정상 상태에 상태카운터(s_cnt: state counter)를 두어 간격이 반복되는 횟수를 센다. 이렇게 각 상태의 간격 및 상태카운터 값이 정해지면 (그림 3)처럼 유한 상태 머신이 완성되고 실제 메모리 접근패턴이 (그림 3)과 같이 발생할 때마다 재귀카운터(recnt: recurrent counter)를 증가시키며 재귀카운터 값이 정해진 임계값을 넘어가면 학습이 끝나고 프리페치가 시작된다.

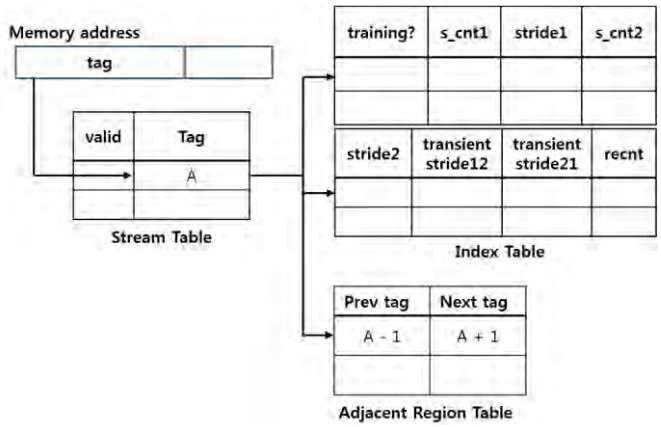
4. ART 를 이용한 다중 간격 프리페치

Iacobovici 가 제안한 다중 간격 프리페치는 다수의 간격을 학습하여 다양한 패턴을 탐지할 수 있으며 특히 영역(Region)을 사용하여 하나의 패턴을 가진 메모리 접근 스트림이 한 영역 안에서 지속적으로 프리페치가 가능하도록 하였다.

그러나 한 영역 안에서만 패턴을 찾는다면 다중 간격 프리페치를 통해 패턴이 학습된 메모리 접근 스트림의 크기가 영역의 크기보다 클 때 학습을 다시 해야 한다. 비록 다수의 영역에 걸친 메모리 접근 스트림의 패턴이 같아도 영역이 바뀔 때마다 영역을 가리키는 태그 값이 바뀌기 때문에 stream table 과 index table 의 다른 엔트리에 스트림이 다시 할당되고 패턴을 익히기 위해 다시 학습을 하게 된다. 이와 같은 불필요한 학습을 피하기 위해 (그림 4)과 같이 ART 를 제안하게 되었다.

ART 는 현재 영역을 가리키는 태그 값의 1 을 더한 값(next tag)과 1 을 뺀 값(prev tag) 을 저장하는 테이블이며 이전에 접근했던 엔트리 위치를 기억해둔다. stream table 에 A 태그로 계속 접근된 후, B 라는 태그가 접근한다고 가정해보자. 먼저 B 가 stream table 에 접근하여, 해당 태그에 대한 엔트리가 없는 것을 확인한 후, 저장된 A 태그의 엔트리 위치를 이용해, A 와 매칭되는 ART 엔트리의 next tag, prev tag 의 값과 B 를 비교한다. 일치하지 않는다면 기존과 같이 stream table 에 새로운 엔트리를 할당을 받고, 일치한다면 stream table 에서 태그 값을 A 에서 B 로 바꾸고 ART 의 next tag 에 B+1 을 prev tag 에 B-1 을 저장한다.

이 방법을 통해 같은 패턴의 메모리 접근 스트림의 크기가 영역의 크기보다 클 경우, 영역이 바뀌더라도 학습한 내용은 그대로 유지되기 때문에 추가적인 학습 없이 동일한 패턴으로 계속 프리페치할 수 있다.



(그림 4) ART 를 이용한 Multi-Strided Prefetch

5. 시뮬레이션 환경

<표 1> 시뮬레이션 환경

L1 명령어 캐시	32KB; 64B block; 8-way; 1 cycle latency
L1 데이터 캐시	32KB; 64B block; 8-way; 3 cycle latency
L2 캐시	512KB; 64B block; 8-way; 8 cycle latency
L3 캐시	2MB; 64B block; 32-way; 30 cycle latency
Tag	48 bits
stream table, index table, ART	128 entries

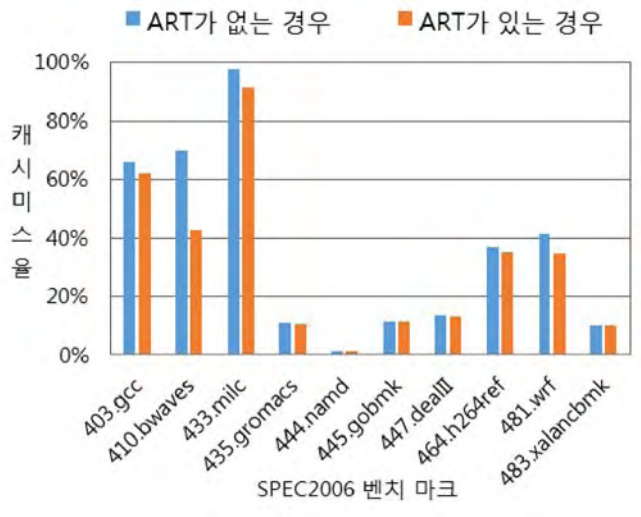
제안한 프리페치 알고리즘과 기존의 다중 간격 프리페치를 구현하기 위해서 MacSim[5] 시뮬레이터를 사용하였다. <표 1>은 시뮬레이터에서 사용한 캐시와 다중 간격 프리페치를 위한 설정이다. 제안된 프리페치 알고리즘을 평가하기 위해 SPEC2006 Benchmark[6]의 x86 바이너리를 사용하였다.

이번 시뮬레이션을 통해 L3에서 L2로의 프리페치를 관찰하였다. L1에 프리페치를 적용하지 않은 이유는 L1의 latency가 크지 않을 뿐만 아니라, Non-blocking Cache 등의 병렬화 기법으로 인해 이미 성능이 많이 개선되었고, 크기가 작기 때문에 프리페치 데이터가 정확하지 않은 경우로 인한 캐시 오염 효과가 다른 캐시보다 크게 나타나기 때문이다.

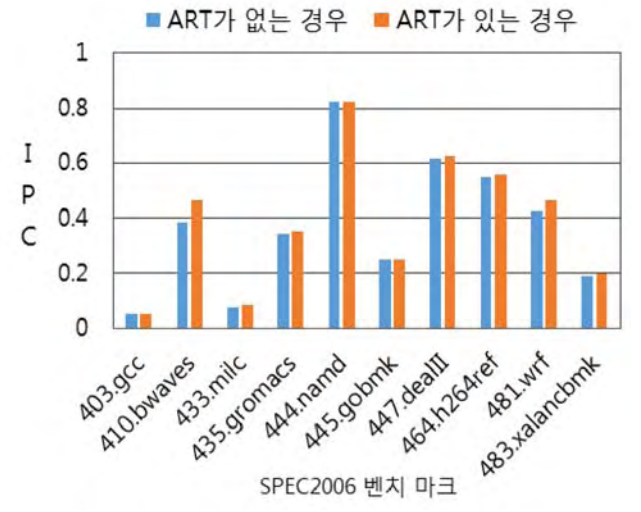
6. 시뮬레이션 결과 및 분석

이번 시뮬레이션을 통해 제안된 알고리즘이 기존의 알고리즘 대비 성능이 어떻게 변하였는지 비교 및 분석한 결과는 다음과 같다. (그림 5)는 기존의 다중 간격 프리페치와 ART 를 이용한 다중 간격 프리페치의 캐시 미스율을 그래프로 나타낸 것이고, (그림 6)은 IPC 를 그래프로 나타낸 것이다. 시뮬레이션 결과 bwaves 에서 캐시 미스율이 38%정도로 제일 감소율이 컸고, 평균적으로 약 6.7%의 캐시 미스 감소율을 보였다. 또한 프로세서의 전체적인 성능을 나타내는 IPC 를 측정했을 때, 역시 bwaves 가 약 20.6%로 가장 높은 성능 향상이 나타났고, 평균적으로 약 5.78%의 IPC 가 증가하는 것을 확인하였다.

bwaves 의 경우 메모리 접근 주소가 일정한 간격으로 계속 증가하는 형태의 패턴을 보였고, 그 패턴을 갖는 메모리 접근 스트림의 크기가 300 개 이상의 영



(그림 5) 다중 간격 프리페치에서 ART 가 없는 경우와 ART 가 있는 경우의 캐시미스율



(그림 6) 다중 간격 프리페치에서 ART 가 없는 경우와 ART 가 있는 경우의 IPC

역을 지날 정도로 매우 큰 것으로 나타났다. 기존의 다중 간격 프리페치를 적용할 경우, bwaves 에서 나타나는 메모리 접근 패턴은 계속 바뀌는 영역의 태그를 stream table 과 index table 의 새로운 엔트리에 할당하고 다시 학습을 해야 하기 때문에 그 동안 프리페치가 중단되어 성능 향상이 제한된다. 그러나 본 논문에서 제안한 ART 를 이용한 다중 간격 프리페치의 경우, 패턴이 일정한 스트림의 접근 영역이 바뀔 때 다른 엔트리에 할당하지 않고 학습을 다시 하지 않기 때문에 프리페치가 지속됨으로 인해 기존의 알고리즘에 비해 성능 향상에 대한 제한을 받지 않아 (그림 5), (그림 6)과 같이 성능이 더 향상된 것으로 나타났다.

7. 결론

기존의 다중 간격 프리페치는 다양한 메모리 접근 패턴을 탐지할 수 있는 장점이 있지만, 메모리 공간을 나눈 영역 중 하나의 영역 안에서만 학습하고 프리페치를 하여 성능 향상에 제약을 받았다. 본 논문에서 제안한 ART 를 이용한 알고리즘은 메모리 접근 스트림이 학습한 패턴을 하나의 영역에 제한하여 이용하지 않고, 인접한 영역에서도 추가적인 학습 없이 이용하게 하였고, 그 결과로 프리페치가 중단되지 않게 함으로써 성능 향상이 제한되지 않도록 하였다. 시뮬레이션을 수행한 결과, ART 를 이용한 경우에 이용하지 않은 경우보다 캐시 미스율이 평균 6.7% 감소하고, IPC 가 평균 5.78% 증가하게 되었다. 이 결과를 통해, 다중 간격 프리페치에서 패턴이 일정한 스트림이 진행될 때 ART 를 이용해 불필요한 학습을 방지하여 프리페치를 중단시키지 않는 것이 프로세서의 성능향상에 기여한다는 것을 확인하였다.

참고문헌

[1] John L. Hennessy, David A. Patterson “Computer Architecture: A Quantitative Approach”, 5th Ed. Morgan Kaufmann Publishers, pp.72-95, 2012.

[2] A. J. Smith, “Cache memories”, ACM Computing Surveys., vol. 14, no. 3, pp. 473–530, 1982.

[3] T.-F. Chen and J.-L. Baer, “Effective hardware-based data prefetching for high-performance processors”, Computers, IEEE Transactions on, vol. 44, pp. 609–623, May 1995.

[4] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham, “Effective stream-based and execution-based data prefetching”, In ICS, 2004.

[5] H. Kim, J. Lee, N. Lakshminarayana, J. Sim, J. Lim, T. Pho, “MacSim: A CPU-GPU Heterogeneous Simulation Framework. User Guide”, Georgia Institute of Technology, 2012.

[6] SPEC CPU2006 Benchmarks: http://www.spec.org/cpu2006