데이타 캐시의 활용도를 높이는 동적 선인출 필터링 기법

(Dynamic Prefetch Filtering Schemes to enhance Utilization of Data Cache)

전 영 숙 † 김 석 일 †† 전 중 남 ††
(Young-Suk Chon) (Sukil Kim) (Joongnam Jeon)

요 약 Load/store와 같은 메모리 참조 명령어는 프로세서의 고속 수행을 방해하는 주요인이다. 캐시 선인출 기법은 메모리 참조에 따른 지연시간을 줄이는 효과적인 방법이다. 그러나 너무 적극적으로 선인출할 경우에 캐시 오염을 유발시켜 선인출에 의한 장점을 상쇄시킨다. 본 연구에서는 캐시의 오염을 줄이기 위해 동적으로 필터 테이블을 참조하여 선인출 명령을 수행할 지의 여부를 결정하는 네 가지 필터링 기법들을 비교 평가한다. 먼저 기존 연구에서의 문제점을 분석하기 위해 이진 상태 기법을 보였는데, 이 기법은 기존 연구와 같이 N:1 매핑을 사용하는 반면, 각 엔트리의 값을 1비트로 하여 두 가지 상태값을 갖도록 하였다. 비교 연구를 위해 완전 상태 기법을 제시하여 비교 기준으로 사용하였다. 마지막으로 본 논문의 주 아이디어인 정교한 필터링을 위한 블록주소 참조 기법을 제안하였다. 이 기법은 이진 상태 기법과 같은 테이블 길이를 가지며, 각 엔트리의 내용은 완전 상태 기법과 같은 항목을 가지도록 하여 최근에 미사용된 데이타의 블록주소가 필터 테이블의 하나의 엔트리와 대응되도록 1:1 매핑을 하였다. 일반적으로 많이 사용되는 일반 벤치마크 프로그램과 멀티미디어 벤치마크 프로그램들에 대하여 실험한 결과, 제안한 블록주소 참조 기법(BAL)이 기존 연구인 동적 필터 기법(2-bitSC)과 비교하여 캐시 미스율이 10.5%감소하였다.

키워드 : 캐시 메모리, 선인출 알고리즘, 필터링

Abstract Memory reference instructions such as loads or stores are critical factors that limit the processing power of processor. The prefetching technique is an effective way to reduce the latency caused from memory access. However, excessively aggressive prefetch leads to cache pollution so as to cancel out the advantage of prefetch. In this study, four filtering schemes have been compared and evaluated which dynamically decide whether to begin prefetch after referring a filtering table to decrease cache pollution. First, A bi-states scheme has been shown to analyze the lock problem of the conventional scheme, this scheme such as conventional scheme used to be N:1 mapping, but it has the two state to 1bit value of each entries. A complete state scheme has been introduced to be used as a reference for the comparative study. A block address lookup scheme has been proposed as the main idea of this paper which exhibits the most exact filtering performance. This scheme has a length of the table the same as the bi-states scheme, the contents of each entry have the fields the same as the complete state scheme recently, never referenced data block address has been 1:1 mapping a entry of the filter table. Experimental results from commonly used general benchmarks and multimedia programs show that average cache miss ratio have been decreased by 10.5% for the block address lookup scheme(BAL) compare to conventional dynamic filter scheme(2-bitSC).

Key words : cache memory, prefetch algorithm, filtering

† 정 회 원 : 충북대학교대학원 전자계산학과
yschon@hanmail.net
†† 종신회원 : 충북대학교 전기전자컴퓨터공학부 교수
ksi@cbu.ac.kr
joongnam@cbu.ac.kr
(Corresponding author임)
논문접수 : 2004년 9월 11일
심사완료 : 2007년 7월 4일

Copyright@2008 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.
정보과학회논문지: 시스템 및 이론 제35권 제1호(2008.2)

1. 서 론

메모리 참조 지연시간을 줄이기 위하여 캐시 메모리가 도입되었고, 이 경우 캐시 미스율을 감소시키는 것이 매우 중요하다. 캐시 구조의 선인출 기법[1-9]은 프로세서가 사용할 것으로 예측되는 데이타를 실제로 필요하기 전에 주기억장치에서 캐시로 인출하여 캐시 미스율을 감소시키는 방법이다. 캐시 선인출 방법들에는 OBL(one block-look-ahead) 방식[1], OBL 방식을 확장한 스트림 버퍼 선인출 방식[2], 마코프 예측기(Markov predictor)방식[3-6], 상관관계(correlation) 방식, 참조예측표(RPT: Reference Prediction Table) 방식이 있다[7].

그림 1은 하드웨어를 이용한 동적 선인출 구조[10]이다. 프로세서가 캐시로 메모리 참조 명령을 전송할 때 그 메모리 주소는 선인출기(Hardware Prefetcher)에도 전송된다. 선인출기는 그 메모리 주소를 기반으로 적절한 선인출 알고리즘에 의해 선인출 여부를 결정하고 선인출할 주소를 계산한다. 계산된 주소는 선인출 큐(Prefetch Queue)에 저장되고, 이후 해당 주소의 데이타가 메모리에서 인출되어 캐시에 저장된다.

이러한 선인출은 캐시 미스를 줄이는데 도움이 되지만 너무 적극적인 선인출은 메모리 버스 트래픽을 증가시켜 데이타 캐시의 접근을 오히려 지연시킬 수 있을 뿐만 아니라 불필요한 데이타를 선인출함으로써 캐시 오염을 일으킬 수 있는 문제점을 갖고 있다.

이러한 캐시 오염 문제를 해결하기 위한 방법으로서 선인출 데이타의 사용유무 정보를 저장하였다가 이 정보를 이용하여 불필요한 선인출 데이타는 선인출을 하지 않도록 사전에 필터링하는 방법이 있는데, 이러한 방법을 선인출 필터링 기법[11,12]이라고 한다.

그림 2는 필터링을 적용한 하드웨어 선인출 구조를 보여주고 있다. 즉, 하드웨어 선인출기로부터 생성된 모든 선인출 명령을 선인출 큐에 전달하는 것이 아니라, 이들 중 적절한 선인출만을 여과하여 실제로 유용한 선인출만을 일으키도록 선인출 필터가 추가된다. 이러한 필터링 특성은 미리 정해진 정적 필터일 수도 있고, 실행하면서 적절하게 그 특성이 변화해가는 동적 필터일 수도 있다. 이러한 선인출 필터링 기법들은 아직 많은 연구가 이루어져 있지는 않은 상태로서, Srinivasan등에 의한 정적 필터링 기법[11]과 Zhuang등의 의한 동적 필터링 기법[12]이 거의 유일한 선행 연구들이다.

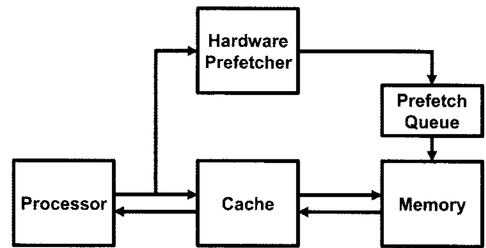


그림 1 동적 데이타캐시 선인출 구조

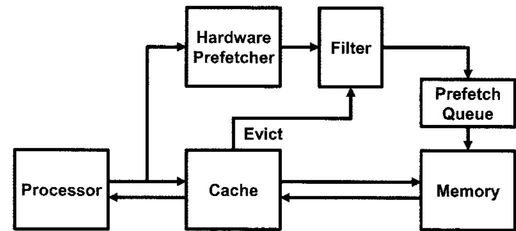
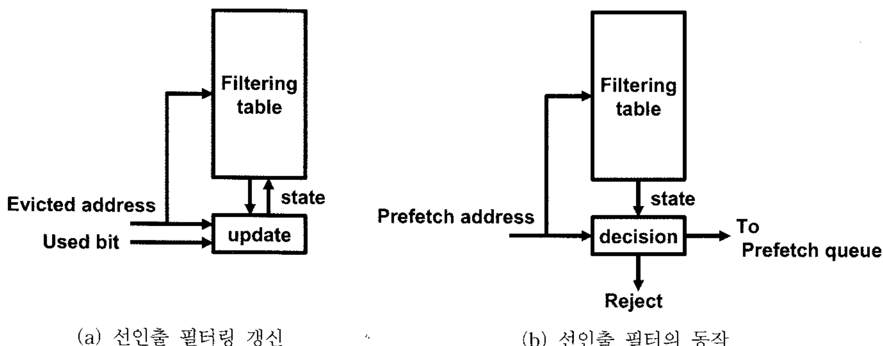


그림 2 필터링을 적용하는 하드웨어 선인출 구조

본 논문에서는 어플리케이션의 종류와 관계없이 필터 특성이 적응하는 동적 필터만을 다루기로 한다. 이러한 동적 필터의 경우, 그림 2에서와 같이 데이타 라인이 캐시에서 추출될 때 그 정보를 피드백하여 동적으로 필터를 구성한다. 그림 3(a)에서와 같이 동적 필터 내부에는



(a) 선인출 필터링 갱신 (b) 선인출 필터의 동작

그림 3 하드웨어 선인출 필터의 구조

필터링 테이블이 존재하여 캐시에서 데이타가 축출될 때마다 필터 테이블을 적절하게 갱신한다. 그림 3(b)는 선인출기가 생성한 선인출 주소를 테이블로부터 참조(lookup)하여 필터링 조건에 만족하지 않는 경우에 선인출 큐에 저장한다.

Zhuang등이 제안한 동적 필터링 방법[12]은 선인출기가 생성한 선인출 주소의 일부를 가지고 필터 테이블을 참조하여 상태를 네 가지로 관리하여 필터링을 수행한다. 이 구조는 불필요한 선인출의 수는 매우 감소시켰으나, 선인출 수에만 중점을 두었고, 전체 성능 면에서는 고려하지 않았다. 즉, 성능을 저하시키지 않고 선인출 수를 감소시키기 위해서는 유용한 선인출은 보존되면서 불필요한 선인출만 감소시켜야 되는데, Zhuang 등에 의한 방법은 유용한 선인출과 불필요한 선인출 모두를 감소시켰다. 또한, Zhuang의 논문에서는 컴파일러에 의해 삽입된 선인출 명령어와 하드웨어 기반의 동적 선인출 명령어를 모두 포함하여 이에 대한 필터링 결과를 분석하였으므로 전체적으로 필터에 입력되는 선인출 수가 증가되어 하드웨어 기반 선인출만을 사용하는 통상적인 경우에 비하여 필터링 효과가 과장된 측면이 있다.

본 논문에서는 이러한 문제점들을 해결하기 위해, 선인출 방법으로 하드웨어 기반의 선인출 명령어만을 포함시키면서, 유용한 선인출은 최대한 보존하고 불필요한 선인출만을 선택적으로 감소시켜 전체 성능을 향상시킬 수 있는 필터링 기법을 제안하고자 한다. 이를 위해서 세 가지 필터링 기법을 제시하여 기존 연구와 비교하였다.

첫째, 기존 연구에서의 문제점을 분석하기 위해 사용된 이진 상태 기법은 Zhuang등의 기법에서 하나의 상태 비트만을 사용하여 필터 테이블을 구성한다. 둘째, 성능비교를 위한 참조로 사용된 완전 상태 기법은 선인출 하였던 모든 주소에 대하여 선인출된 이후부터 캐시에서 축출될 때까지 해당 데이타가 사용되었는지 여부를 모두 필터 테이블에 저장하여, 이후에 동일 주소의 데이타가 선인출 되려고 할 때 테이블을 참조하여 사용되지 않은 선인출 주소에 대하여 선인출 명령을 수행하지 않는다. 셋째, 본 논문의 주 아이디어인 블록주소 참조 기법은 이진 상태 기법과 같은 테이블 길이를 가지면서 테이블의 각 엔트리의 내용은 완전 상태 기법과 같은 항목을 가지도록 하여, 최근에 미사용된 데이타의 블록주소가 필터 테이블의 하나의 엔트리에 대응되도록 하는 1:1 매핑 구조를 사용한 방식이다.

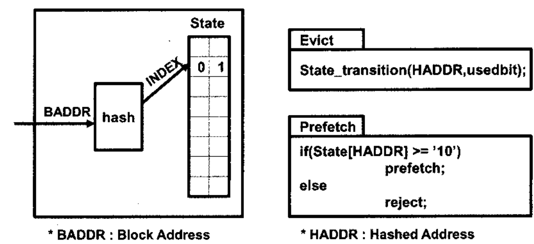
본 논문의 구성은 다음과 같다. 2장에서는 선인출 구조에 관한 기존 연구를 요약하고, 본 논문의 대상인 동적 필터 구조에 대하여 설명한다. 3장에서는 제시한 세 가지 필터링 기법에 대해 자세히 설명한다. 4장에서는 선인출 데이타의 정확도를 분석하고, 실험을 통한 성능 평가 결과를 제시한다. 마지막으로 5장에서는 본 논문에서 얻은 결과를 정리한다.

2. 관련 연구

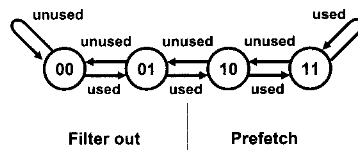
서론에서 간단히 언급한 바와 같이 적극적인 하드웨어 선인출 알고리즘에 의한 캐시 오염을 줄이는 선인출 필터링 기법에는 정적 필터 구조와 동적 필터 구조로 구분할 수 있다. Srinivasan등에 의해 제안된 정적 필터는 선인출에 의해 생성된 트래픽과 미스들에 근거하여 선인출들을 분류하는 광범위한 분류법을 사용하였다[11].

정적 필터는 먼저 프로파일을 통해 오프라인으로 오염이 되는 선인출의 정보를 수집하여 이 정보를 통해 선인출을 수행한다. 이 프로파일 정보는 주어진 입력 데이타들에 대해 정확한 정보를 제공하지만, 워킹 세트가 변할 때 수행시간 동안에 동적 적응성이 결여되는 문제점이 있다.

동적 필터는 이러한 정적 필터와 달리 어플리케이션의 종류와 관계없이 필터 특성이 적응하게 되는데, 그림 4는 Zhuang등에 의해 제안된 동적 필터 구조[12]를 보여준다. 그림 4(a)는 서로 다른 데이타 주소에 대해서 같은 인덱스 주소를 공유하는 N:1 매핑 구조를 보이고 있다. 해당 선인출 주소(BADDR : block address)는 해시 함수를 통과하여 필터 테이블에 인덱스를 한다. 인덱스된 위치의 엔트리 값은 2bit로 구성이 되어 있으며, 캐시에서 축출될 때마다 그림 4(b)처럼 상태를 변경시킨다. 선인출기가 생성한 선인출 주소의 인덱스 주소로 테이블을 참조하여 해당 엔트리의 상태값이 2 이상인



(a) 2bit saturation counter를 사용한 필터 구조



(b) 상태도

• used : 사용이 되고 캐시에서 축출된 경우, unused : 사용이 되지 않고 축출된 경우

그림 4 동적 필터 구조(2-bitSC)

경우에 선인출 명령을 수행하고, 그렇지 않은 경우에는 선인출을 하지 않는다.

이러한 기존 동적 필터 기법의 문제점은 두 가지로 요약될 수 있다.

첫째로는 해싱을 이용한 N:1 매핑을 하기 때문에 정확성이 결여된다는 점이다. 즉, 서로 다른 데이타 주소에 대해서 같은 엔트리를 공유하기 때문에 필터링을 해야 하는 경우와 하지 말아야 하는 경우의 정보가 공유되어(aliasing) 정교한 판단을 할 수 없도록 되어 있는 구조라고 할 수 있다.

둘째로는 위와 같은 aliasing 효과와 더불어 2bit 포화 카운터를 사용함으로써 결과적으로 모든 선인출이 대부분 억제되어 버리는 결과를 낳는다. 따라서 불필요한 선인출 뿐만 아니라 유용한 선인출 모두를 억제시켜 결과적으로 캐시 미스율을 오히려 증가시키게 된다. 이러한 현상은 본 논문에서는 잠김(lock) 현상이라고 정의하기로 한다.

이러한 잠김현상이 일어나게 되는 원인을 더 구체적으로 설명해 보도록 한다. 2bit 포화 카운터의 초기값은 '10' 또는 '11'로 되어 있어야 한다. 그렇지 않으면 처음부터 선인출이 금지되어 축출이 전혀 일어나지 않게 되기 때문이다.

필터링 테이블 내의 특정한 엔트리의 경우, 여러 개의 주소들이 동일한 엔트리를 공유하게 된다. 처음에는 이러한 주소들에 대하여 선인출들이 개시될 것이다. 엔트리에 따라서 유용한 선인출 주소들과 불필요한 선인출 주소들이 포함되어 있는 비율이 각각 다를 것이다. 즉, 어떠한 엔트리의 경우에는 절반 정도씩 포함되어 있을 것이고 다른 경우에는 대부분이 유용한 선인출일 수도 있고 또 다른 경우에는 대부분이 불필요한 선인출일 수도 있다. 그러나, 해싱에 의하여 유용한 선인출 주소와 불필요한 선인출 주소가 평균적으로 모든 엔트리에 대하여 대략 비슷한 비율로 섞여 있게 될 것이다. 또한, 선인출 필터링은 상당히 적극적인 선인출 알고리즘에 대하여 유용해지기 때문에 평균적으로 불필요한 선인출의 수가 더 많다고 가정할 수 있다.

따라서, 모든 엔트리에서 유용한 선인출 주소와 불필요한 선인출 주소가 반반씩 포함되어 있다고 가정하고 설명을 한다면 공평함을 잃지 않을 것이다. 유용한 선인출 및 불필요한 선인출들이 개시되고 나서, 처음에는 두 그룹에 속한 주소들이 거의 같은 비율로 축출될 것이다. 따라서, 엔트리의 2bit 포화 카운터의 값도 중간 정도의 값 ('10' 또는 '01')을 가지게 될 것이다. 그러나, 한 번 카운터의 값이 '01'로 내려가게 되면 다음 번 유용한 데이타의 축출 전까지는 선인출이 금지된다. 따라서, 잠시 동안은 해당 엔트리에 대한 선인출 빈도가 감소하게 된다. 그런데, 이러한 현상은 다른 엔트리들에 대해서도 거의 동시 다발적으로 발생하게 된다. 즉, 초기 상태로부터 시간이 얼마 지난 후에는 모든 엔트리들에 대하여 선인출 빈도가 조금 줄어들게 된다. 이렇게 되면 캐시로부터 데이타들이 축출되는 빈도도 조금 줄어들게 될 것이다. 즉, 캐시로부터 유용한 데이타들이 축출되는 빈도도 줄어들게 된다. 따라서, 2bit 카운터의 값이 다시 '10'으로 회복되는 엔트리들의 수도 줄어들게 된다. 이러한 프로세스가 누적되면 점점 더 많은 엔트리들이 선인출 금지상태가 되고, 따라서 엔트리들이 선인출 가능 상태로 회복되는 기회가 점점 줄어들게 된다. 즉, 궁극적으로는 대부분의 엔트리들이 선인출 금지 상태로 잠겨(stuck 또는 lock) 버리게 되는 것이다.

이와 같은 현상은 used 방향과 unused 방향으로의 이동이 실은 대칭적이지 않기 때문에 발생한다. 즉, 카운터의 값이 '10' 이상인 경우에는 불필요한 데이타와 유용한 데이타의 축출 비율에 비례하여 카운터의 값이 수렴하게 되나, 일단 '01'이하로 내려가게 되면 used 방향으로의 이동이 점점 더 힘들어지게 된다. 왜냐하면, 이러한 경우 선인출 금지와 데이타 축출의 감소는 서로 양의 피드백(positive feedback)을 이루기 때문이다.

또한, 확률적으로 불필요한 선인출 데이타의 축출이 연속으로 발생하게 되는 경우, 이것을 회복하기 위해서는 동일한 횟수만큼 유용한 데이타가 연속으로 축출되어야 하나, 일단 선인출 금지상태가 되었기 때문에 회복할 수 있는 확률은 극히 제한적일 수 밖에 없다. 이러한 현상은 포화 카운터의 깊이(depth)가 깊어질수록 더욱 커지게 된다. 실험 결과에 따르면, 이러한 잠김 현상은 2bit 이상의 카운터에 대해서 나타나는 것을 알 수 있었다.

Zhuang등이 원래 의도했던 바는 이 기법이 N:1 매핑을 사용하기 때문에 같은 엔트리를 공유하는 서로 다른 블록주소들의 선인출 유용도의 평균값을 구하기 위하여 2bit 포화 카운터를 사용한 것이다. 그러나 위에서 설명한 바와 같이 카운터의 값은 의도와는 달리 평균값이 '10' 이상이 아닌 점점 '00'에 가까이 가게 된다. 따라서, 거의 모든 선인출 수행이 억제되어 필터로서의 제 역할을 하지 못하게 된다. 이러한 현상이 일어나는 것은 해싱에 의한 N:1 매핑과 2bit 포화 카운터 사용이 그 궁극적인 원인이 된다.

결론적으로 해싱을 사용하는 경우 평균적으로 불필요한 선인출이 더 많이 포함되어 있는 엔트리를 찾아 내기 위해서는 단순한 N-bit 포화 카운터를 사용하는 방법은 적합하지가 않고, 보다 복잡하고 지능적인 방법을 결합해야 할 것이다. 예를 들면, 전체 엔트리의 잠겨있는 개수와 캐시 미스율의 변화 추세를 보고 동적으로 임계값(threshold)을 조정해 나가는 등의 방법이 있을

수 있겠다. 그러나, 본 논문에서는 해싱을 사용하지 않는 방법이 좋은 필터링 성능을 낼 수 있다는 것을 보이고, 그러한 추가 연구는 다음으로 미루도록 한다.

3. 선인출 필터링 방식

이 장에서는 필터링 성능을 높이기 위하여 세 가지의 기법을 제시하며, 기존 기법과의 비교를 서술하기로 한다.

이진 상태 기법(Binary state : BS)

기존 2-bitSC 기법에서는 해싱과 2bit 포화 카운터를 사용함으로 인해 전체 선인출의 총합을 많이 감소시키기는 하지만 유용한 선인출의 수 또한 줄이게 되어 전체 캐시 미스율을 늘리는 단점을 갖는다. 이는 위에서 설명한 잠김현상 때문으로써 이 문제점을 비교 분석하기 위해 이진 상태 구조(BS 기법)를 실험하였다. 이 방법은 기존 연구의 포화 카운터의 깊이(depth)를 2로 줄여서 사용이 되지 않고 축출된 경우에는 0상태를 갖고, 사용이 되고 축출된 경우는 1상태로 갖게 한다. 1상태에서 사용이 되고 축출된 데이타는 1상태를 유지하고 사용이 되지 않고 축출 된 경우에는 0상태로 가게 된다.

즉, BS 기법은 하나의 상태 비트만을 두어서 사용 미사용 여부에 따라서 즉각적으로 선인출 여부가 바뀌도록 의도한 것이다. 이 경우에도 N:1 매핑에 의한 정확도 문제는 해결되지 않지만, 한번 상태가 0이 되더라도 사용된 선인출 데이타가 하나라도 캐시에서 축출될 경우 다시 상태가 1이 되므로 위의 기존 카운터를 사용한 경우에서와 같은 필터링 상태에서의 잠김(lock) 현상이 일어날 확률이 감소하게 된다.

결론적으로 잠김현상을 줄이기 위해 2-bitSC 기법에서 used방향은 캐시에서 데이타가 축출될 때만 천이하는 것이 아니라 앞에서 언급한 것과 같이 조금 더 복잡한 알고리즘으로 처리해 주어야 할 필요가 있다

3.2 완전 상태 기법(Complete state : CS)

그림 5는 여러 가지 필터 기법들을 비교하고 가장 바람직한 방식을 유도함에 있어서 기준을 마련하기 위하여 이상적인 필터링 구조를 제시한 것이다. 이 구조는 선인출하였던 모든 주소에 대하여 선인출된 이후부터 캐시에서 축출될 때까지 해당 데이타가 사용되었는지 여부를 모두 필터링 테이블에 저장하며, 이후에 또 다시 동일 주소의 데이타가 선인출되려고 할 때 테이블을 참조하여 선인출하였으나 사용되지 않은 경우에는 선인출 명령을 제거한다.

이 구조는 데이타의 블록주소 전부를 저장해야 하므로 테이블의 크기가 너무 커져서 실제로는 구현하기가 어렵다는 문제점이 있다. 블록주소는 전체 주소에서 캐시라인 오프셋 비트를 제거한 것이며, 그림 5에서와 같이 블록주소의 비트수가 N이라 할 때 필터 테이블의 엔트리 수는 총 2N개가 필요하게 된다. 예를 들면, 전체 주소가 32bit이고 라인크기가 16byte일 때 블록주소는 32bit - 4bit = 28bit이며, 필터 테이블의 크기는 228개 × 1bit가 된다.

3.3 블록주소 참조 기법(Block Address Lookup : BAL)

완전 상태 기법은 과거의 모든 history 정보를 가지고 있는 반면, 본 논문에서 제안한 블록주소 참조 기법(BAL)은 최근에 축출된 불필요한 선인출 데이타의 블록 주소만을 저장하도록 테이블의 크기를 제한한 기법이다.

그림 6은 블록주소 참조 테이블 구조로 위에서 설명한 완전 상태 기법이 그 크기로 인하여 실제적으로는 구현이 거의 불가능하므로 대신에 사용되지 않은 블록주소만을 FIFO로 구현한 필터 테이블에 저장하며 테이블 크기를 제한한 구조이다.

선인출 데이타가 캐시에서 축출될 때 사용이 되지 않은 경우 필터 테이블에 해당 블록주소를 저장하고, 이후 선인출기가 생성한 선인출 주소가 필터 테이블에 존재하지 않는 경우에는 선인출 명령을 수행하고, 그렇지 않고 존재하는 경우에는 선인출 명령을 제거한다. 만약에 요구 미스에 의해 메모리로부터 인출한 블록주소가 테이블 내에 존재할 경우에는 해당 블록주소를 필터 테이블에서 제거한다. 이 기법은 블록주소 전부를 테이블에 저장하므로 각 엔트리의 크기가 커지는 문제점이 있는 반면, 모든 블록주소를 저장하는 완전 상태 기법과 비교하여 엔트리의 수가 크게 줄어드는 장점이 있다. 기존

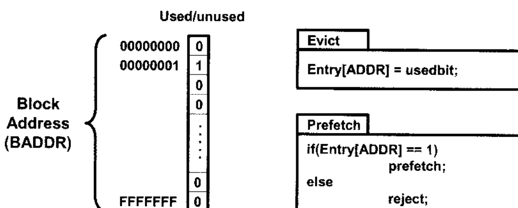


그림 5 완전 상태 구조(Complete state)

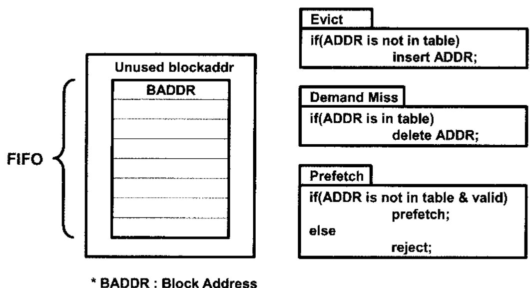


그림 6 블록주소 참조를 갖는 필터 구조(BAL)

Zhuang등에 의해 제안된 기법과 비교하면 각 엔트리에는 하나의 주소만이 할당되며 이러한 1대1 매핑에 따른 정확한 필터링을 할 수 있는 장점이 있다. 또한 미사용된 주소의 개수가 테이블 크기보다 더 커지게 되면 모든 미사용된 주소를 포함할 수 없게 되어서 LRU나 FIFO 교체정책을 써야 한다.

본 논문에서는 LRU 교체 정책을 사용하여, 선인출기가 생성한 선인출 주소가 필터 테이블에 존재하는 경우에는 해당 엔트리를 가장 앞쪽으로 이동하도록 하여, 필터링 효율이 유지되도록 하였다. 또한 테이블의 크기는 캐시의 라인 수와 동일하게 하였다. 예를 들면, 캐시 크기가 128KByte, 블록크기가 32byte인 경우 테이블은 4096개의 엔트리를 가지도록 하였다.

4. 시뮬레이션 및 성능 분석

4.1 시뮬레이션 환경

필터링을 적용한 선인출 데이타의 효과를 분석하기 위하여 DEC사의 ATOM 시뮬레이터[13]를 이용하여 load/store 명령어에 대한 트레이스 구동 시뮬레이션(trace-driven simulation)을 수행하였다. 트레이스에는 메모리 참조 명령어에 대하여 load 혹은 store인지 여부, 그리고 필요한 피연산자의 유효주소, PC값으로 구성되어 있다. ATOM 시뮬레이터의 출력 결과인 명령어 트레이스를 캐시 시뮬레이터의 입력으로 받아 캐시의 성능 및 선인출의 효과를 분석한다.

캐시 시뮬레이터는 위스콘신 대학에서 개발한 Dinero Ⅲ[14]을 기반으로 선인출 알고리즘(OBL, RPT, correlation)과 필터링 방식들을 추가하여 사용하였다. 명령어 캐시와 데이타 캐시가 분리되어 있는 캐시 구조를 실험 대상으로 하였고, 데이타 캐시의 성능만 측정하였다. 캐시 시뮬레이터는 캐시크기, 블록크기, 사상 함수등의 매개변수를 입력으로 캐시 분석 결과를 생성한다. 필터 테이블의 엔트리 수는 캐시 라인 수로 하였다. 즉, CS를 제외한 2-bitSC, BS, BAL방식들에서 엔트리 수를 동등한 크기로 하여 실험하였다.

표 1에 분석 모델을 위한 본 연구와 관련된 캐시 시뮬레이터의 매개변수들과 값을 제시하였다. 대표적인 멀티미디어 벤치마크 프로그램인 MPEG(mpeg2enc, mpeg2dec)[15]과 SpecInt2000[16]의 일반 벤치마크 프로그램인 parser, crafty, vortex, gzip을 대상으로 1천만번의 메모리 참조 명령어에 대하여 실험하였으며, 표 2는 벤치마크 프로그램들의 특징을 나타낸다.

표 1 분석 모델을 위한 매개 변수의 값

Parameter	Description
D-cache size	4K~1M
Block size	32byte
Word size	4byte
Associativity	Direct mapped
Bus width	Block size
History table entry size	Cache set size

표 2 벤치마크 프로그램 특징

Bench program	Input file	Description
vortex	bendian.raw	Object-oriented Database
gzip	tar file	compression
parser	2.1.dict	Word processing
crafty	reference input	Game Playing: Chess
mpeg2encoder	pirates.par	digital video와 audio 압축에 관한 표준 도구
mpeg2decoder	mei16v2.m2v	

그림 7은 요구인출과 선인출 명령을 발생하기 위해 필터 구조를 포함한 경우의 필터링과 갱신과정에 대한 흐름도이다. 각 번호에 대한 내용은 다음과 같다.

(1) Prefetcher : 각 선인출 알고리즘에 의해 계산된 전체 선인출할 데이타의 개수
(2) Miss_filter : 필터 테이블을 참조하여 필터링 조건에 만족하지 않은 경우의 수
(3) Hit_filter : 필터 테이블을 참조하여 필터링 조건에 만족하는 경우의 수
(4) C_M : 캐시를 참조하여 미스가 발생한 수로 실제로 선인출 명령을 발생
(5) C_H : 캐시를 참조하여 히트가 난 경우로 실제로 선인출 명령을 발생시키지 않음
(6) D_M : 프로그램이 실행하면서 데이타 액세스를 필요로 하였으나, 캐시에서 미스가 발생한 수. D_M을 프로그램의 전체 메모리 액세스 수로 나누면 캐시 미스율이다.

표 3은 OBL 하드웨어 선인출 알고리즘에 대한 여러가지 필터링 기법에서의 선인출에 따른 각 단계별 데이타 수를 측정한 결과이다. 하드웨어 선인출기로부터 생성된 선인출 수(Prefetcher: 그림 7의 1번)는 필터링 기법에 무관하게 동일한 값이다. 이들 중에서 선인출 필터에 의해 필터링 된 개수를 Hit_filter(그림 7의 3번)라 하고, 필터링 되지 않고 선인출 큐로 보내지는 개수를 Miss_filter라 하였다. 2-bitSC의 경우 Miss_filter의 수는 다른 방식들보다 가장 적으며, Hit_filter의 개수는 가장 많다. 이것은 대부분의 선인출이 필터링 된다는 것인데, 뒤에서 설명될 것과 같이 N:1 매핑에 따른 부정확성 및 카운터 동작에 의해서 거의 모든 데이타가 잠김 상태가 되어 버리기 때문이다. 이로 인하여 유용한 선인출이 대부분 억제되어 선인출을 하지 않은 경우와

표 3 벤치마크별 선인출 데이타 분석(d128k, b32, a1, OBL)

OBL		Prefetcher	Miss_filter	Hit_filter	C_M	C_H	D_M
vortex	no-prefet	0	0	0	0	0	216,032
	no-filter	6,753,207	6,753,207	0	119,904	6,633,303	214,639
	CS		4,092,281	2,660,926	40,380	4,051,901	196,819
	2-bitSC		143,008	6,610,199	4,942	138,066	213,491
	BS		460,421	6,292,786	11,383	449,038	209,879
	BAL		4,173,455	2,579,752	41,010	4,132,445	196,876
gzip	no-prefet	0	0	0	0	0	257,011
	no-filter	7,101,886	7,101,886	0	195,424	6,906,462	227,057
	CS		3,990,168	3,111,718	135,345	3,854,823	227,933
	2-bitSC		203,999	6,897,887	6,356	197,643	254,814
	BS		381,735	6,720,151	11,155	370,580	253,036
	BAL		3,994,723	3,107,163	139,656	3,855,067	228,562
crafty	no-prefet	0	0	0	0	0	131,012
	no-filter	7,454,421	7,454,421	0	51,070	7,403,351	141,345
	CS		2,962,174	4,492,247	32,378	2,929,796	127,305
	2-bitSC		791,821	6,662,600	6,172	785,649	128,781
	BS		1,205,577	6,248,844	11,048	1,194,529	127,775
	BAL		2,962,174	4,492,247	32,378	2,929,796	127,305
parser	no-prefet	0	0	0	0	0	89,232
	no-filter	8,500,820	8,500,820	0	44,192	8,456,628	67,694
	CS		6,720,280	1,780,540	31,366	6,688,914	68,055
	2-bitSC		101,739	8,399,081	6,327	95,412	84,112
	BS		2,208,910	6,291,910	18,254	2,190,656	76,084
	BAL		6,720,280	1,780,540	31,366	6,688,914	68,055
mpeg2dec	no-prefet	0	0	0	0	0	4,272
	no-filter	8,275,072	8,275,072	0	4,319	8,270,753	1,570
	CS		8,131,488	143,584	3,433	8,128,055	1,253
	2-bitSC		5,264,771	3,010,301	3,095	5,261,676	1,449
	BS		6,692,721	1,582,351	3,386	6,689,335	1,236
	BAL		8,131,488	143,584	3,433	8,128,055	1,253
mpeg2enc	no-prefet	0	0	0	0	0	30,310
	no-filter	8,333,324	8,333,324	0	31,678	8,301,646	9,002
	CS		6,085,726	2,247,598	25,250	6,060,476	7,073
	2-bitSC		335,094	7,998,230	7,724	327,370	23,227
	BS		2,469,292	5,864,032	23,837	2,445,455	8,473
	BAL		6,085,726	2,247,598	25,250	6,060,476	7,073

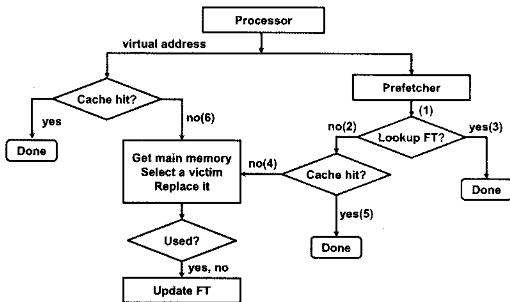


그림 7 선인출 명령 발생을 위한 필터 구조를 포함한 흐름도

거의 같은 미스율을 보인다. BS의 경우는 잠김 상태가 상대적으로 훨씬 적어서 선인출을 더 많이 발생시키며 미스율은 2-bitSC에 비해 현저히 낮아졌다.

4.2 필터링 정확도 분석

여러 가지 선인출 알고리즘 및 벤치마크에 대하여 각각의 필터링 기법을 비교하였다. 필터링 정확도라고 하는 것은 불필요한 선인출은 최대한 억제하는 대신 유용한 선인출에는 영향을 주지 않는 것이라고 할 수 있다. 따라서 각각의 필터링 기법을 적용한 결과와 필터링을 적용하지 않은 경우와 비교함으로써 필터링 정확도를 평가해 볼 수 있다.

여기에서 불필요한 선인출을 다음의 2가지로 정의할 수 있다.

첫째, 선인출을 한 뒤 캐시에서 사용되지 않고 축출된 개수, 둘째, 선인출 발생 후 해당 주소가 요구인출이 발생하지 않은 경우, 즉 첫 번째 정의에서는 계속 캐시에 남아 있었더라면 사용되었을 데이타이지만 너무 일찍 선인출하여 사용되지 않은 경우도 불필요한 선인출에 포함된다. 이 경우 필터링 정확도가 캐시의 크기 등 여러 요인에 의해 영향을 받으므로 본 논문에서는 두 번

째 정의를 사용하였다. 그러나, 두 가지 정의에 의한 상대적인 결과는 큰 차이가 없음이 실험을 통해 알 수 있었다.

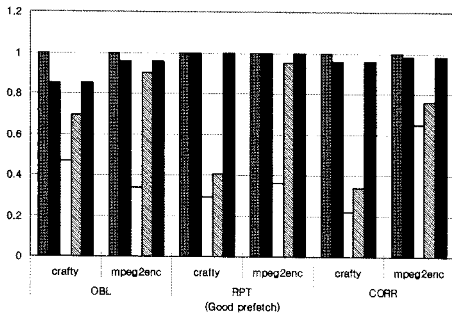
그림 8은 대표적인 두 가지 벤치마크(일반 벤치마크인 crafty와 멀티미디어 벤치마크인 mpeg2enc)에 대해서 앞에서 설명한 세가지 선인출 방식에서의 각각의 필터링 결과를 보여준다. 각각의 그래프는 유용한 선인출(good)과 불필요한 선인출(bad)의 개수를 필터링 하지 않은 결과에 대해서 정규화한 값을 보여준다.

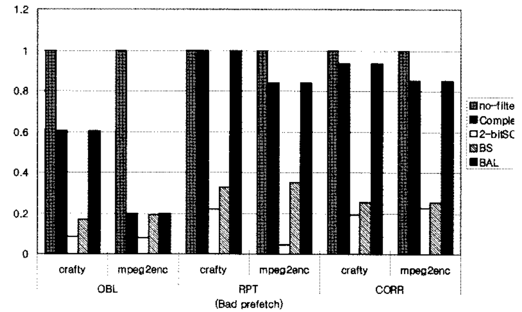
전반적으로 기존 2-bitSC 기법이 good 및 bad 선인출 모두 가장 낮은 값, 즉 가장 많이 필터링 되었음을 알 수 있다. 그러나 일반 벤치마크의 경우 bad 선인출보다 good 선인출의 개수가 더 많이 줄어들었음을 알 수 있다. CS 기법의 경우, 원래의 선인출 수가 많지 않은 RPT와 correlation에 대해서는 good과 bad 선인출 모두 크게 변하지 않았다. 그러나 선인출 개수가 많은 OBL에 대해서는 good 선인출은 대부분이(70%이상) 유지되는 반면, bad 선인출은 20~40%로 줄어드는 것을 알 수 있다. 한편 BAL 기법은 테이블 크기가 제한적임에도 불구하고 CS 기법의 경우와 거의 동일한 결과 보였다. 필터링 테이블의 크기는 CS 기법을 제외하면 모두 캐시의 라인 개수와 동일하게 실험을 하였다. 즉, 그림 9의 경우 캐시 크기는 128Kbyte이고 라인크기는 32byte이므로 필터링 테이블의 엔트리 수는 4096개로 하였다. BS 기법의 경우 OBL과 RPT의 mpeg2enc의 경우 good에 비해 bad의 개수가 상대적으로 더 많이 감소되었다.

요약하면 기존 2-bitSC 기법의 경우 전체 선인출 수는 가장 많이 감소하였으나 평균적으로 bad 선인출보다 good 선인출이 더 많이 감소하였으므로 필터링 정확도는 좋지 않다고 할 수 있다. BS 기법의 경우는 기존 방식과 유사하나 선인출 수가 많은 OBL의 경우는 정확도가 약간 높았다. 또한 CS 기법과 BAL 기법은 동일한 성능을 보이는데, 특히 선인출 수가 많은 OBL의 경우는 필터링 정확도를 가짐을 알 수 있었다.

4.3 시간에 따른 선인출 특성 분석

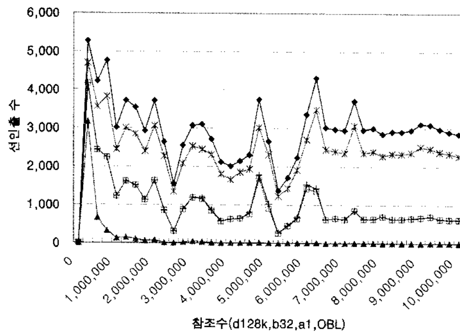
기존 2-bitSC 기법의 경우에서는 잠김 현상으로 인해 거의 모든 선인출이 일정 시간 이후 대부분 억제 될 것



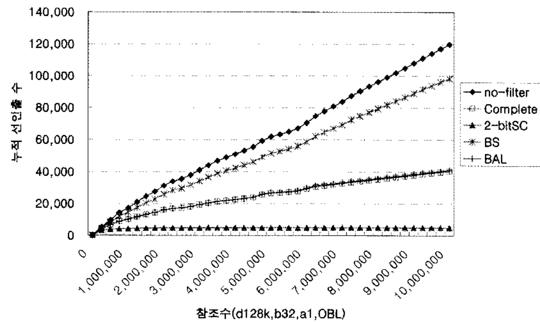


• good prefetch: 선인출 된 데이타가 사용이 되고 캐시에서 축출
• bad prefetch : 선인출 된 데이타가 사용이 되지 않고 캐시에서 축출(evict)

그림 8 good과 bad 선인출 수 비교



(a) 선인출 수



(b) 누적 선인출 수

그림 9 시간대별 선인출 수 비교(vortex, OBL)

으로 예측하였다. 이와 같은 사실을 검증하기 위하여 시간에 따른 선인출 수를 그림 9와 같이 분석하였다.

그림 9(a)에서 볼 수 있듯이 기존 2-bitSC기법은 초기부터 급격히 감소하기 시작하여 120만회의 메모리 참조 이후부터는 거의 선인출을 하지 않게 된다. 반면, 제시된 필터링 기법들에서는 선인출 수가 필터링 하지 않은 경우에 비해 전체적으로 감소되어 있을 뿐 시간에 따라서 균등한 값을 유지하고 있다. 따라서 시간이 지날수록 기존 방식에서는 선인출을 전혀 하지 않은 경우와 같이 캐시미스가 계속 늘어나서 선인출을 전혀 하지 않은 경우와 마찬가지가 될 것이다.

본 실험에서는 일천만개의 트레이스를 사용하였으나, 그 수가 더 늘어날수록 기존 방식의 bad및 good 선인출 수는 0에 가까워질 것이다. 그림 9(b)는 누적 수를 나타낸 것인데 다른 경우는 꾸준히 증가하고 있는데 기존 2-bitSC 기법은 새로운 선인출이 거의 없음을 알 수 있다. 즉 필터링 테이블의 거의 모든 엔트리가 잠김 상태에 빠져 버렸음을 알 수 있다.

기존 동적 필터 방식의 카운터 비트 수를 1로 줄인 BS 기법의 경우, 해싱을 사용함에도 불구하고 시간이

표 4 벤치마크별 캐시 미스 수

(a) d64k, b32, a1

d64k,b32,a1	OBL					
	vortex	gzip	parser	crafty	mpeg2dec	mpeg2enc
no-filter	1,062,783	448,255	107,782	148,328	2,282	15,000
CS	1,030,630	424,517	99,682	132,580	1,785	11,230
2-bitSC	1,051,410	481,254	119,961	134,540	2,393	33,050
BS	1,049,560	480,608	113,378	133,466	1,770	14,438
BAL	1,031,432	453,273	99,651	132,580	1,785	11,230

d64k,b32,a1	RPT					
	vortex	gzip	parser	crafty	mpeg2dec	mpeg2enc
no-filter	1,040,658	424,520	90,239	134,534	1,860	10,108
CS	1,040,324	424,517	90,236	134,536	1,860	10,008
2-bitSC	1,051,200	478,407	118,080	134,995	2,435	30,927
BS	1,041,618	424,959	91,777	134,623	1,835	11,587
BAL	1,040,324	424,517	90,236	134,536	1,860	10,008

d64k,b32,a1	CORR					
	vortex	gzip	parser	crafty	mpeg2dec	mpeg2enc
no-filter	1,050,731	399,355	114,018	133,776	5,290	34,577
CS	1,050,652	404,525	113,777	133,744	5,268	34,653
2-bitSC	1,052,140	480,618	118,777	136,215	4,712	35,250
BS	1,047,005	471,867	112,881	134,577	4,565	34,476
BAL	1,050,652	404,523	113,777	133,742	5,268	34,653

(b) d128k, b32, a1

128k,b32,a1	OBL					
	vortex	gzip	parser	crafty	mpeg2dec	mpeg2enc
no-filter	214,639	227,057	67,694	141,345	1,570	9,002
CS	196,819	227,933	68,055	127,305	1,253	7,073
2-bitSC	213,491	254,814	84,112	128,781	1,449	23,227
BS	209,879	253,036	76,084	127,775	1,236	8,473
BAL	196,876	228,562	68,055	127,305	1,253	7,073

128k,b32,a1	RPT					
	vortex	gzip	parser	crafty	mpeg2dec	mpeg2enc
no-filter	207,255	211,632	64,960	129,179	1,306	6,154
CS	206,961	211,626	64,959	129,178	1,306	6,107
2-bitSC	213,662	252,410	84,103	129,560	1,514	22,719
BS	207,326	211,804	65,501	129,280	1,290	6,727
BAL	206,961	211,626	64,959	129,178	1,306	6,107

128k,b32,a1	CORR					
	vortex	gzip	parser	crafty	mpeg2dec	mpeg2enc
no-filter	216,050	246,850	84,620	130,318	4,275	27,011
CS	215,775	247,782	84,615	130,309	4,271	26,944
2-bitSC	214,596	254,881	85,940	130,716	4,258	27,460
BS	211,582	253,388	84,189	130,566	4,252	26,555
BAL	215,775	247,782	84,615	130,309	4,271	26,944

지나더라도 선인출의 수가 기존 연구에서와 같이 0이 되어 버리지 않고 일정하게 유지되는 것으로 보아, 기존 기법에서의 잠김 현상이 해싱 뿐 아니라 2-bit 이상의 포화 카운터를 사용하였기 때문인 것을 알 수 있다. 따라서, 해싱을 사용하더라도 좀 더 복잡한 처리 방식을 접합할 경우 유용한 필터링 성능을 얻어낼 수도 있다는 것을 예측할 수 있다.

4.4 전체 성능 분석

위에서 분석한 선인출 특성에 의하여 캐시에 전체적인 성능이 어떻게 영향을 받는지를 분석해보았다. 표 4는 캐시 크기를 64kbyte, 128kbyte로 하고 각 선인출 기법에 대해서 벤치마크별로 캐시 미스 수를 나타낸 것이다.

그림 10은 표 4의 결과들을 가지고 필터링을 하지 않은 경우(no-filter)의 미스율을 기준으로 각 필터링 기법들의 미스율에 대한 성능 향상도를 다음의 식 (1)과 같이 정의하여 그래프로 나타낸 것이다.

$$E_{CM_ratio} = \frac{M_{filtering}}{M_{no_filter}} \qquad (1)$$

식 (1)에 보인 성능 향상도 E_{CM_ratio}의 결과가 1보다 작으면 필터링을 하지 않은 경우보다 미스율이 낮아서 성능이 향상되었음을 의미하며, 1보다 큰 경우에는 필터링을 하지 않은 경우보다 미스율이 증가되어 필터링 효과가 떨어짐을 의미한다.

실험 결과 하드웨어 선인출기에 의해 선인출이 개시되는 개수가 가장 적은 correlation의 경우 필터링 방식에 따라서 큰 차이가 나지 않는다. 단지, 2-bitSC 기법이 다른 방식들보다 미스율이 미약하게 큰 것을 알 수 있다. 다음으로 RPT의 경우에는 2-bitSC 기법은 가장 미스율이 높고, 그 다음으로 BS 기법이 약간 높으며, 나머지 기법들은 비슷한 결과를 보인다. OBL의 경우에는 벤치마크에 따라서 crafty와 mpeg2dec의 경우에는 필터링한 결과가 하지 않은 결과에 비해 모두 더 낮은 미스율을 보인다. 한편 BAL 기법과 CS 기법은 전반적으로 필터링을 하지 않은 경우에 비해 최대 22% 향상된 결과를 보인다.

표 5는 OBL 선인출 기법에서 캐시 크기를 4Kbyte부터 1Mbyte까지 증가시키면서 모든 벤치마크 프로그램들의 캐시 미스 수를 측정한 것이다.

그림 11은 표 5의 결과들을 가지고 필터링을 하지 않는 경우에 대하여 정규화를 시킨 값이다. 캐시 크기가 커짐에 따라서 필터링 한 경우의 미스율은 필터링 하지

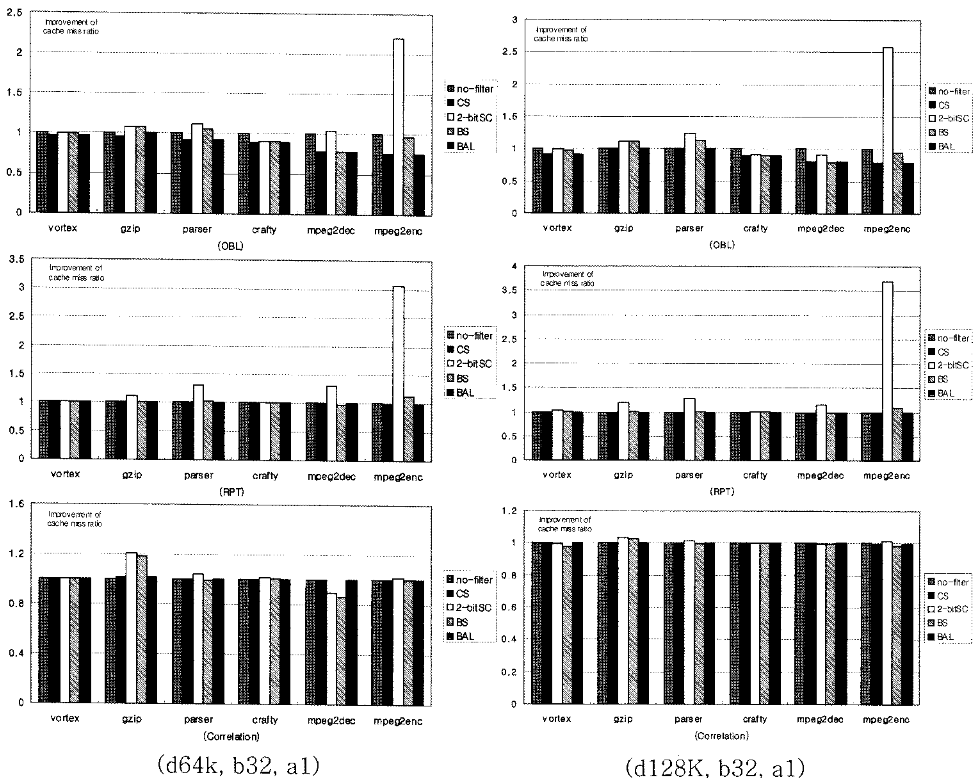


그림 10 벤치마크별 캐시 미스율

표 5 캐시 크기의 변화에 따른 미스 수

vortex	4K	8K	16K	32K	64K	128K	256K	512K	1M
no-prefetch	2,021,008	1,642,539	1,350,368	1,160,602	1,053,039	216,032	180,496	164,245	154,982
no-filter	2,175,998	1,764,016	1,429,908	1,191,879	1,062,783	214,639	171,837	153,513	145,221
CS	1,955,415	1,592,757	1,312,446	1,130,448	1,030,630	196,819	165,101	150,868	143,322
2-bitSC	2,020,838	1,642,267	1,349,824	1,159,595	1,051,410	213,491	176,306	158,160	147,134
BS	2,020,440	1,641,669	1,349,004	1,158,615	1,049,560	209,879	171,838	154,176	144,548
BAL	2,012,634	1,625,942	1,323,379	1,133,295	1,031,432	196,876	165,101	150,868	143,322

gzip	4K	8K	16K	32K	64K	128K	256K	512K	1M
no-prefetch	1,394,627	840,406	735,424	630,464	483,049	257,011	84,281	60,478	59,237
no-filter	1,680,535	878,692	748,236	600,822	448,255	227,057	73,956	57,227	55,507
CS	1,396,313	824,607	710,354	602,372	453,098	227,933	72,277	56,920	55,503
2-bitSC	1,394,459	840,130	734,836	629,429	481,582	254,814	81,022	58,608	55,717
BS	1,394,125	839,507	734,174	628,921	481,055	253,036	78,238	58,490	55,607
BAL	1,413,852	829,396	707,415	600,716	453,280	228,562	73,490	56,920	55,503

parser	4K	8K	16K	32K	64K	128K	256K	512K	1M
no-prefetch	869,741	402,864	246,929	164,468	122,315	89,232	62,821	39,106	33,004
no-filter	804,628	370,015	207,560	121,221	75,224	44,009	32,269	20,193	16,122
CS	695,527	308,907	172,320	100,486	69,586	45,551	32,519	19,958	16,010
2-bitSC	869,571	402,594	246,393	163,356	119,979	83,995	52,453	23,374	16,700
BS	799,783	368,796	209,026	121,359	76,130	45,080	32,393	20,207	16,137
BAL	696,890	307,677	171,601	99,590	69,265	45,552	32,519	19,958	16,010

crafty	4K	8K	16K	32K	64K	128K	256K	512K	1M
no-prefetch	366,546	302,527	265,869	147,237	136,479	131,012	95,539	87,123	68,118
no-filter	472,410	390,346	332,803	161,612	148,328	141,345	68,398	64,651	63,367
CS	355,908	295,013	260,492	142,765	132,580	127,305	67,418	64,501	63,404
2-bitSC	366,374	302,253	265,336	146,212	134,898	128,781	82,951	69,907	64,090
BS	365,930	301,700	264,204	144,847	133,695	127,775	70,884	66,103	63,584
BAL	360,901	296,852	260,900	142,799	132,580	127,305	67,418	64,501	63,404

mpeg2dec	4K	8K	16K	32K	64K	128K	256K	512K	1M
no-prefetch	656,644	639,201	522,030	518,921	4,793	4,272	3,486	3,486	3,486
no-filter	1,099,186	1,078,404	919,963	916,244	2,282	1,570	498	498	498
CS	706,703	690,901	576,280	573,587	1,785	1,253	498	498	498
2-bitSC	656,479	638,902	521,311	517,279	2,394	1,449	498	498	498
BS	655,632	637,944	519,660	515,993	1,776	1,236	498	498	498
BAL	706,721	690,904	576,280	573,587	1,785	1,253	498	498	498

mpeg2enc	4K	8K	16K	32K	64K	128K	256K	512K	1M
no-prefetch	424,603	87,673	58,609	43,624	36,407	30,310	22,985	21,352	17,766
no-filter	432,232	87,513	45,222	24,302	13,632	7,693	4,058	3,792	3,231
CS	459,143	65,075	34,101	18,342	10,497	6,081	3,377	3,122	2,592
2-bitSC	424,442	87,368	57,879	41,963	32,884	23,113	9,168	6,103	2,554
BS	403,111	80,500	41,872	22,523	12,713	7,603	3,863	3,484	2,923
BAL	459,047	64,962	33,942	18,342	10,497	6,081	3,377	3,122	2,592

않은 경우에 비해 점점 더 증가한다. 이것은 필터링 하지 않은 경우 불필요한 선인출로 인한 캐시의 오염이 캐시크기가 커짐에 따라 점점 그 영향이 줄어드는 반면, 필터링한 경우에서는 필터링으로 인한 유용한 선인출의 감소로 인한 미스율 증가가 두드러지게 나타나기 때문이다.

유용한 선인출의 감소가 거의 없는 BAL 기법과 CS의 경우에는 캐시 크기가 증가함에 따라 상대적으로 미스율이 증가하는 경향을 보이기는 하지만 그 값이 필터링 하지 않은 경우보다는 항상 작다. 기존의 2-bitSC 기법의 경우 캐시 크기가 작을 때에는 미사용된 선인출이 더욱 증가하여 이로 인한 대부분의 선인출이 잠김 상태에 있기 때문에 선인출하지 않은 경우와 미스율이 거의 같음을 알 수 있다. BS 기법의 경우 캐시 크기가 작을 때에는 2-bitSC와 거의 같은 값을 가지나 캐시 크기가 커짐에 따라 그 차이가 커짐을 알 수 있다. 즉, BS의 경우에도 캐시 크기가 작은 경우 미사용된 선인출이 많으면 대부분의 선인출이 잠김 상태에 있는 것을 알 수 있다. 2-bitSC의 경우 캐시 크기가 128Kbyte보다 커지게 되면 미스율이 다시 감소하는데, 이것은 축출되어 나가는 데이타 수가 급격히 감소하기 때문에 잠김 상태에 있는 엔트리가 거의 없어지기 때문이다.

일반적으로 일반 벤치마크의 경우에는 64Kbyte 이하의 캐시 크기에 대하여 BAL 기법의 필터링 방식을 사용하는 것이 유리하며, 멀티미디어 벤치마크에서는 캐시 크기에 관계없이 BAL 기법을 사용하는 것이 유리하나, 256Kbyte 이상에서는 BS 기법도 거의 같은 성능을 낼 수 있다. 한편 기존 2-bitSC 기법은 캐시크기가 1M byte로 매우 큰 경우를 제외하고는 제안된 방식에 비해 성능이 낮으며, 특히 8Kbyte에서 32Kbyte 사이의 미스율은 제안한 BAL기법이 기존 연구보다 10.5% 성능이 향상되었다.

표 6은 기존 2-bitSC 기법의 엔트리 수(4096개)에 BAL 기법의 엔트리 수를 감소시켜가면서 각 벤치마크별 캐시 미스 수를 비교한 것이다. 캐시의 크기는

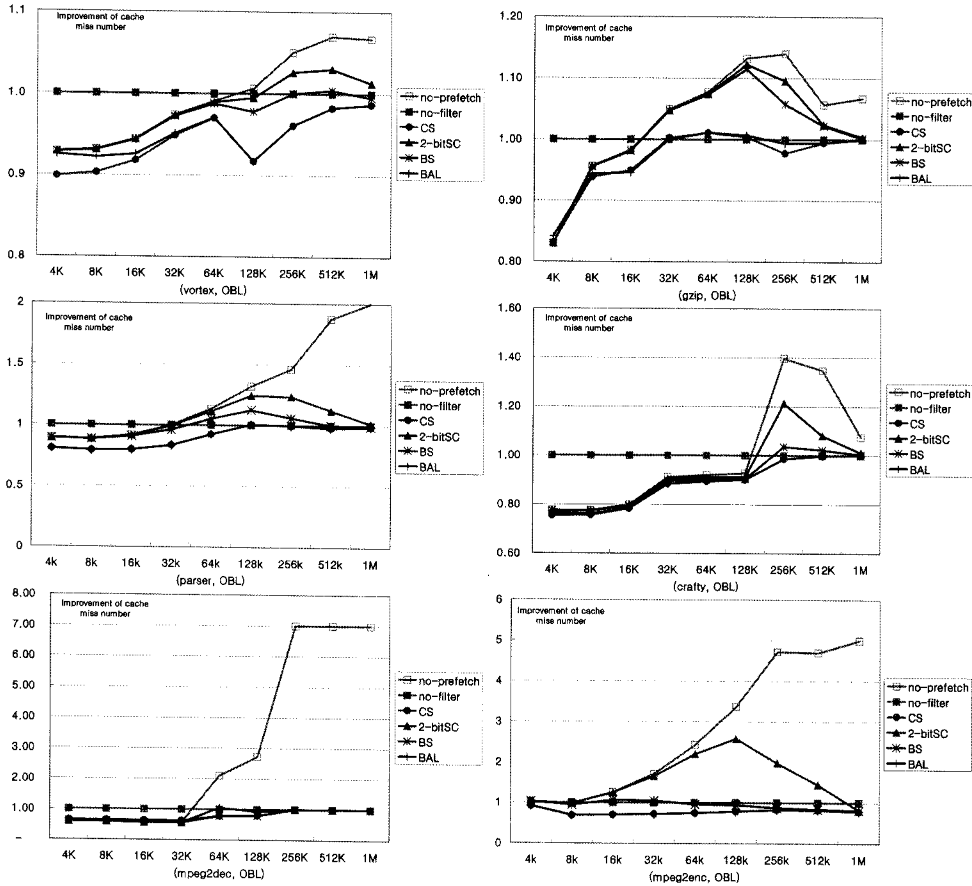


그림 11 캐시 크기에 따른 캐시 미스 수

표 6 2-bitSC 기법과 BAL 기법의 엔트리 수의 변화에 따른 캐시 미스 수 비교

d64k,b32,a1	bench / entry_no.	vortex	gzip	mpeg2dec	mpeg2enc
2-bitSC	4,096	1,050,943	481,254	2,434	31,026
BAL	4,096	1,030,823	453,273	2,182	13,122
	2,048	1,031,432	453,280	2,182	13,122
	1,024	1,032,287	452,361	2,182	13,122
	512	1,033,889	449,838	2,182	13,120
	256	1,036,888	448,126	2,182	13,089
	128	1,041,337	446,969	2,182	12,976
	64	1,046,144	446,632	2,182	12,965
	32	1,049,988	446,535	2,185	12,969
	16	1,052,912	446,485	2,187	12,967
	8	1,055,450	446,566	2,188	12,971

d128k,b32,a1	bench / entry_no.	vortex	gzip	mpeg2dec	mpeg2enc
2-bitSC	4,096	213,491	254,814	1,491	23,093
BAL	4,096	196,876	228,562	1,501	12,726
	2,048	197,270	228,573	1,501	12,726
	1,024	197,597	228,584	1,501	12,726
	512	198,257	228,344	1,501	12,726
	256	199,145	228,246	1,501	12,618
	128	200,677	227,187	1,501	12,473
	64	203,025	226,784	1,501	12,665
	32	205,494	226,642	1,501	12,672
	16	207,394	226,557	1,503	12,673
	8	209,027	226,453	1,503	12,674

64Kbyte, 128Kbyte로 하고 블록크기를 32byte로 하였다. 표에서와 같이 BAL 기법의 엔트리 수를 줄이더라도 기존 2-bitSC방식보다 캐시 미스 수가 더 적게 출력되는 것을 볼 수 있다. 이는 기존 방식과 비교하여 더 적은 수의 하드웨어를 가지고도 더 좋은 성능을 얻을 수 있음을 나타낸다.

5. 결 론

캐시 기억장치에 사용되는 데이타 선인출 기법은 프로세서가 연산을 수행하는 동안 필요한 오퍼랜드를 미리 캐시로 적재해서 메모리 액세스 시간을 줄이는 효과적인 방법이다. 그러나 너무 적극적인 선인출은 캐시의 오염을 일으켜 캐시가 갖는 본래의 이점을 상쇄시킬 수 있다. 이러한 캐시 오염문제를 해결하기 위해 선인출 필터링 방법을 사용한다.

본 논문에서는 기존의 필터링 기법이 갖는 문제점을 보완하여 세가지 필터 기법들을 제시하였다. 기존 2-

bitSC 기법은 블록주소를 해싱을 하여 필터 테이블로 인덱스를 하므로 선인출 한 데이타가 사용이 되지 않고 캐시에서 축출될 경우 동일한 인덱스를 갖는 블록주소 전체의 선인출이 금지되며, 한번 금지가 된 인덱스 주소는 2회 이상 사용이 되고 캐시에서 축출이 되어야 다시 살아나게 되므로 그러한 경우는 극히 적으므로 빠져 나오기가 어렵게 되는 즉, 잠김 상태가 됨을 시간에 대한 선인출 동작 분석을 통하여 알 수 있었다.

이러한 2-bitSC 기법의 문제점을 보이고 분석하기 위해 한번이라도 참조가 될 경우 선인출을 할 수 있도록 한 방법이 BS 기법이다. CS 기법은 제시한 필터링 기법들의 성능 비교를 위한 참조로 사용되었다. 또 다른 방법으로 본 논문의 주 아이디어로서 해싱을 하지 않는 블록주소 참조 기법을 제안하였으며, 이 기법은 사용이 되지 않은 블록주소를 필터 테이블에 저장하므로 2-bitSC기법이 N:1 매핑인데 비해 1:1 매핑이므로 더 정확하게 필터링을 하는 장점을 가지게 되며, CS 기법과 거의 동일한 성능을 가짐을 보였다.

일반 벤치마크와 멀티미디어 벤치마크 프로그램을 가지고 실험을 한 결과, 캐시 미스율은 모든 선인출 알고리즘과 벤치마크들에 대하여 2-bitSC 기법과 비교하여 제안한 BAL 기법이 10.5% 감소함을 알 수 있었으며, 이러한 결과들을 토대로 good 선인출의 수를 보존하면서 bad 선인출을 감소시키도록 하는 것이 전체 캐시 성능을 향상시킨다는 것을 보였다.

또한, 해싱을 사용하는 경우에도 기존의 방식과는 다른 좀 더 지능적인 엔트리 관리 방법을 사용하면 바람직한 필터링 성능을 낼 수도 있음을 보였으며, 이를 포함하여 PC 정보를 효율적으로 이용할 수 있는 방법들이 추가적으로 이루어져야 할 것으로 보인다.

한편, 선인출의 정확도를 높이고 캐시의 오염을 방지하기 위해 본 논문에서 제안한 선인출 필터링 기법을 응용하여 웹이나 데이타베이스 분야의 선인출에 적용하고, 모바일 컴퓨팅, 센서 네트워크에 적용을 위한 에너지 소모를 고려한 캐시 및 메모리 운영 방법에 대해 연구 되어져야 할 것으로 생각한다.

참 고 문 헌

[1] A. J. Smith, "Cache Memories," ACM Computing Surveys, 14:473-530, Sep. 1982.

[2] N. P. Jouppi, "Improving directed-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," Proc. of the 17th Annual International Symposium on Computer Architecture, pp. 364-373, May 1990.

[3] D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," IEEE Trans. on computers, Vol.48, No.2, Feb. 1999.

[4] D. Joshep and D. Grunwald, "Prefetching Using Markov Predictors," in proc. Of the 24th Annual Intl. Symp. On Computer Architecture, pp. 252-263, June 1997.

[5] J. Kim, K. V. Palem and W-F. Wong, "A Framework for Data Prefetching using Off-line Training of Markovian Predictors," in Proc. IEEE Intl. Conf. on Computer Design(ICCD), pp. 340-347, Sep. 2002.

[6] G. Hariprakash, R. Achutharaman, A. R. Omondi, "DSTRIDE : Data-cache miss-address-based stride prefetching scheme for multimedia processors," 6th Australasian Computer Systems Architecture Conference (AustCSAC'01), pp. 62-70, Jan. 29-30, 2001.

[7] Y. Solihin, J. Lee and J. Torrellas, "Correlation Prefetching with a User-Level Memory Thread," *IEEE Trans. Computers, Vol.14, No.6, June 2003.*

[8] J. L. Baer and T-Fu Chen, "An effective on-chip preloading scheme to reduce data access penalty," *In Proceedings of Supercomputing '91, pp. 176-186, Nov. 1991.*

[9] T-Fu Chen and J-L Baer, "Effective Hardware-Based data prefetching for High-Performance Processors," IEEE Trans. Computers, Vol.44, No.5, pp. 609-623, May 1995.

[10] J. H. Lee, S. W. Jeong, S. D. Kim and C. C. Weems, "An Intelligent Cache System with Hardware Prefetching for High Performance," IEEE Trans. on computers, Vol.52, No.5, May. 2003.

[11] V. Srinivasan, E. S. Davidson and G. S. Tyson, "A Prefetch Taxonomy," IEEE Trans. Computers, Vol.53, No.2, pp. 126-140, Feb. 2004.

[12] X. Zhuang and H-H S. Lee, "Hardware-based Cache Pollution Filtering Mechanism for Xggressive Prefetches," in Proc. IEEE Int. conf. on Parallel Processing, pp. 286-293, Oct. 2003.

[13] A. Srivastava and A. Eustace, "ATOM : A System for Building Customized Program Analysis Tools," Proceedings of the ACM SIGPLAN 94, pp. 196-205, 1994.

[14] M. D. Hill, "Dinero III Cache Simulator," Technical Report, Department Computer Science, University of Wisconsin, Madison. 1990.

[15] Media benchmark program : http://cares.icsl.ucla.edu/appications.html

[16] SpecInt 2000 benchmark program : http://www.spec.org/osg/cpu2000/CINT2000.

전 영 숙

1996년 한남대학교 전자계산학과(학사) 1998년 한남대학교 컴퓨터 공학과(석사) 2002년 충북대학교 컴퓨터과학과 박사수료. 관심분야는 고성능 컴퓨터 구조, 병렬 처리, 메모리 시스템

김 석 일

1975년 서울대학교 학사학위 취득. 1985년~1989년 미국 North Carolina University(공학박사). 1990년~현재 충북대학교 전기전자컴퓨터공학부 교수. 관심분야는 병렬처리컴퓨터 구조, 슈퍼컴퓨팅, RFID 응용

전 중 남

1990년 연세대학교 전자공학과 공학박사 1996년~1998년 미국 Texas A&M 연구교수. 1990년~현재 충북대학교 전기전자컴퓨터공학부 교수. 관심분야는 컴퓨터구조, 임베디드시스템