공격적인 선인출 및 직접 사상 필터링을 이용한 L1 캐시 선인출 기법

(An L1 Cache Prefetching Scheme using Excessively Aggressive Prefetchering and a Small Direct-mapped Filtering Cache)

전 영 숙 †

(Chon Young Suk)

요 약 본 논문에서는 공격적인 선인출 및 직접 사상 필터링을 이용한 L1 캐시 선인출 기법을 제안한다. 이를 위하여 캐시 선인출의 역효과에 대한 정량적 분석 방법을 제안하였고 이를 이용하여 다양한 벤치마크에서의 공격적 선인출 효과를 분석하였다. 분석 결과를 바탕으로 최적 선인출 필터 구조 및 알고리즘을 도출하였고 독자적으로 개발된 타이밍 기반 캐시 시뮬레이터를 사용하여 전체 시스템 성능을 추출하였다. 실험 결과는 제안된 L1 선인출 기법을 사용하여 다양한 벤치마크에 대하여 시스템 성능을 평균적으로 18% 향상시킬 수 있음을 보인다.

키워드 : 캐시 메모리, 캐시 선인출, 하드웨어 선인출, 선인출 필터링

Abstract This paper proposes an L1 cache prefetch scheme using an excessively aggressive hardware prefetcher and a hardware prefetch filter having a small direct-mapped filtering cache. A quantitative analysis method has been introduced and applied to analyze nonideal effects of aggressive cache prefetching. From those analysis results, the structure and algorithm of a prefetch filter has been derived and simulated, and the overall system performance has been measured using a cycle-by-cycle cache simulator. Experimental results show that the proposed scheme improves the overall system performance by 18% on the average over several benchmarks

Key words : cache memory, cache prefetch, hardware prefetcher, and prefetch filter

1. 서 론

캐시 선인출은 캐시 미스에 의해 유발되는 메모리 접근 지연시간을 숨김으로써 시스템 성능을 향상시키기 위한 목적으로 사용된다. 이러한 캐시 선인출은 소프트웨어[1,2] 및 하드웨어로 구현될 수 있다[3-8]. 하드웨어로 구현된 캐시 선인출기는 시스템 전체 성능을 극대화하기 위하여 그 알고리즘을 특정 시스템의 구조에 맞게 최적화 시킬 수 있다는 장점이 있다. 반면에 하드웨어 선인출기의 알고리즘을 특정한 응용 프로그램에 대하여 최적화 시킬 수는 없다. 그러므로, 하드웨어 선인출기는 다양한 응용 프로그램에 대한 평균적인 성능이 극대화 될 수 있도록 설계되어야 한다.

현재까지 여러 가지 하드웨어 선인출 알고리즘들이 연구되어져 왔는데 그것들은 주소 증분(stride) 기반과 상관관계(correlation) 기반의 두 가지로 분류할 수 있다. 단일블록 참조(one block lookahead; OBL)[3], 순차적 선인출(next sequential prefetch; NSP)[4], 네이버(neighbor)[5], 스트림 버퍼(stream buffer)[6], 참조예측표(reference prediction table; RPT)[7,8] 등의 방식들이 바로 주소 증분 기반 하드웨어 선인출 알고리즘에 해당한다. 그러한 선인출 알고리즘은 대부분의 메모리 접근(access)들이 고정된 또는 적어도 동적으로 추적이 가능한 주소 증분을 가질 것이라는 가정에 근거를 둔다. 이러한 가정은 for-문이나 while-문 등의 대규모 반복 계산을 많이 포함하고 있는 주로 과학 계산용 응용 프로그램들에 대해서는 유효할 수 있다. 그러나, 대부분의 메모리 접근 행태가 간단하거나 균일하지 않은 다른 응용 프로그램들에서는 단순한 주소 증분을 갖지 않는 접근 패턴들이 더 많은 부분을 차지하게 된다. 상관관계 기반의 선인출 알고리즘들은 그러한 경우에 있어서의

† 학생회원 : 충북대학교 컴퓨터과학과
yschon@hanmail.net
논문접수 : 2006년 3월 27일
심사완료 : 2006년 7월 22일

메모리 접근 패턴을 예측하기 위한 시도들이다[10-14]. 이러한 방식들은 연속된 미스 데이타(missed data) 주소들 간의 순서쌍을 이용하여 참조를 예측하게 된다. 이러한 관점에서, 접근된 주소와 직후에 미스가 된 주소 간의 관계를 이용하는 섀도우 디렉토리 선인출(shadow directory prefetch)[15] 또한 같은 범주에 속한다고 할 수 있다.

그러나, 이러한 하드웨어 기반 선인출 방식은 포인터 추적(pointer chasing) 등의 더욱 복잡한 메모리 접근 행태를 많이 포함하는 몇몇 응용 프로그램들에서는 그다지 유용하지 않을 수 있다. 그러한 응용 프로그램들의 메모리 접근 패턴은 어떠한 복잡한 알고리즘에 의해서도 정확하게 예측이 되지 않는다. Luk와 Mowry는 어떠한 벤치마크들의 경우에서는 매우 정교하게 고안된 복잡한 선인출 알고리즘들도 매우 높은 미스 율을 보임을 보고한 바 있다[2].

적절한 필터와 결합된 과도하게 공격적인 선인출 방법이 그러한 경우를 위한 효과적인 대안이 될 수 있다. 예측의 정확도 자체는 매우 낮을 것임에도 불구하고, 선인출하는 총량 또는 모수(母數)가 많기 때문에 결과적으로 적중되는 예측의 절대적인 수는 더 클 수 있기 때문이다. 거의 모든 응용 프로그램들이 공간적 국소성(spatial locality) 및 시간적 국소성(temporal locality)을 어느 정도 포함하고 있다고 가정할 때에, 이러한 알고리즘이 시스템 성능을 향상시킬 수 있는 가능성이 어느 정도 있다고 할 수 있는 것이다. 물론, 최종적인 예측의 정확도를 높이기 위해서는 불필요한 선인출들을 적절하게 걸러낼 수 있는 필터링 기법이 또한 수반되어야 한다. 지금까지 발표된 선인출 필터링 방식은 정적 필터와 동적 필터의 두 가지로 구분된다. Zhuang과 Lee에 의해 제안된 동적필터는 변화하는 환경에 동적으로 적응하기 위하여 필터의 특성을 실시간으로 갱신한다[17]. 반면에 정적 필터는 이러한 동적 적응성을 갖지 못한다[15]. 그러나, Zhuang과 Lee는 전체 선인출 수의 감소 및 선인출 정확도의 증가에만 중점을 두었고, 유용한 선인출의 감소로 인한 전체 시스템의 성능 저하를 인식하지 못하였었다.

본 논문에서는 전체 시스템의 성능을 향상시킬 수 있는 L1 캐시 선인출 방식이 제안된다. 제안된 방식에서는 과도하게 공격적인 선인출 알고리즘과 간단하고 효과적인 필터를 결합시킨다. 필터링 알고리즘은 선인출 정확도 뿐만 아니라 시스템 성능이 극대화 되도록 최적화될 것이다. 그러한 최적의 구조 및 알고리즘은 선인출이 시스템 성능에 미치는 역효과를 분석한 결과로부터 도출된다. 본 논문은 다음과 같이 구성된다. 2장에서는 하드웨어 캐시 선인출 구조를 설명하고, 3장에서는 선인출에 의한 역효과를 분석한다. 분석된 결과를 바탕으로 하여 4장에서는 선인출 필터의 구조를 도출하고, 5장에서 제안된 구조의 시뮬레이션 결과를 제시할 것이다. 6장에서는 제안된 구조의 한계점을 논의한 후, 마지막으로 결론을 맺기로 한다.

2. 하드웨어 L1 캐시 선인출 구조

그림 1은 본 논문에서 가정한 하드웨어 L1 선인출 구조를 보여 준다. CPU로부터 캐시 접근 요청이 L1 캐시로 전달될 때에 이 하드웨어 선인출기는 요청된 데이타의 주소를 분석하여 선인출 여부를 판단하고 적절한 선인출 주소를 생성하여 이를 선인출 큐에 저장한다.

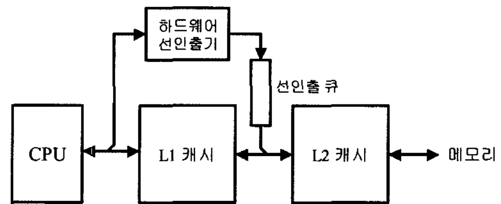


그림 1 하드웨어 L1 캐시 선인출 구조

이 때에 선인출 개시 여부 및 선인출 주소의 생성은 선인출 알고리즘에 따라 각각 다르다. 한편, 선인출 요청을 선인출 큐에 저장할 때에 먼저 선인출 큐를 검사하여 동일한 주소로의 선인출 요청이 있을 시에는 큐에 저장하지 않는다. 이와 같이 캐시 선인출은 캐시 미스를 방지함으로써 메모리 지연시간을 숨기기 위하여 어떤 데이타가 요구되기 전에 미리 해당 데이타를 페치(fetch)한다. 그런데, 이러한 동작은 한편으로 캐시에 오염을 유발하여 캐시의 용량 미스(capacity miss)의 발생율을 높이고 또한 메모리 트래픽을 증가시키는 역효과를 가져온다. 만일 메모리 지연시간을 숨김으로 인한 이득이 역효과에 의한 손실보다 작다고 하면, 시스템의 성능은 선인출을 하지 않은 경우에 비해서 오히려 더 나빠질 수도 있게 된다.

선인출로 인한 이득은 얼마나 많은 캐시 미스들이 선인출로 인하여 방지될 수 있었는가에 비례할 것이다. 만일 선인출된 데이타가 최소한 하나 이상의 요구 미스(demand miss)를 방지할 수 있었다면 그 선인출은 좋은 선인출(good prefetch)이라고 간주하기로 한다. 그렇지 않은 경우에는 불필요한 또는 나쁜 선인출(bad prefetch)로 간주하기로 한다. 그러므로, 선인출로 인한 이득은 좋은 선인출의 총량에 비례할 것이고, 나쁜 선인출은 캐시 오염과 메모리 트래픽을 증가시키기만 할 뿐이므로, 선인출의 역효과로 인한 손실은 나쁜 선인출의 총량에 비례할 것이다.

선인출 해 온 데이타가 이미 캐시 내에 존재하고 있었다면(선인출 히트), 그 선인출로 인하여 캐시 오염이 유발되지는 않을 것이다. 그렇지만, 여전히 일정 시간 동안 메모리 버스를 점유하였을 것이므로 다른 메모리 접근의 처리는 그 시간만큼 지연되었을 것이므로 평균 메모리 접근 지연 시간(memory access delay time; MADT)을 증가시키게 될 것이다. 따라서, 선인출의 역효과로 인한 손실은 선인출 미스와 선인출 히트를 모두 포함하는 선인출 페치 수에 비례할 것이다.

캐시 선인출은 L1 캐시 또는 L2 캐시에 모두 적용될 수 있다. L1 캐시는 그 접근 속도가 매우 빨라야 하는데, 선인출기를 추가할 경우 이로 인한 L1 캐시로의 접근 부하가 증가하여 전체 시스템 성능을 크게 저하시킬 가능성이 크다. 또한, L1 캐시는 그 용량이 매우 작으므로 위에서 언급한 바와 같은 캐시 오염에 의한 선인출 역효과가 L2 캐시에 비해 심각하여 매우 정교한 선인출 알고리즘을 제외하고는 실제적으로 전체 시스템 성능을 개선시키기 힘들다. 이러한 이유로 대부분의 기존 연구에서는 L2 캐시 선인출을 대상으로 하고 있는데, L2 캐시 선인출은 그 성능이 아주 우수하다고 하더라도 전체 시스템 성능을 크게 개선시키지는 못한다는 한계가 있다[17].

본 논문에서 가정하는 그림 1의 구조의 특징적인 부분으로는, L1 캐시에의 하드웨어 선인출기를 추가함으로 인한 부하의 증가를 최소화하기 위하여 모든 선인출들은 선인출 큐를 거쳐서 다음 수준의 캐시로 직접 요청되도록 한다는 점이다. 이렇게 함으로써 선인출 요청은 CPU로부터의 요구 요청(demand request)들과 서로 L1 포트를 차지하기 위해서 경쟁하지 않아도 된다. 따라서, 이 구조를 비경쟁(non-competeing) L1 선인출 구조라고 부르기로 한다.

이러한 비경쟁 구조는 CPU의 메모리 요청들에게 더 많은 사용 가능한 L1 포트들이 제공될 수 있다는 것을 의미하므로 시스템 성능에 더 이롭게 된다. 반면에, 선인출 시에 캐시를 먼저 검색(lookup)하지 않으므로 선인출 미스 뿐 아니라 선인출 히트들 까지도 트래픽 증가에 기여하기 때문에, 선인출로 인한 메모리 트래픽의 증가가 더욱 커질 가능성이 있다. 그러나 이러한 트래픽의 증가분은 본 논문에서 제안하는 선인출 필터를 이용하여 효과적으로 감소될 수 있는데 그림 2는 이러한 선인출 필터를 가지는 하드웨어 L1 선인출 시스템의 구조를 보여준다.

선인출 필터는 하드웨어 선인출기로부터 개시된 선인출 요청을 좋은 선인출과 나쁜 선인출로 구분하여 좋은 선인출은 통과시키고 나쁜 선인출은 차단시키는 필터의 역할을 한다. 이렇게 함으로써 선인출 트래픽을 감소시키고 또한 L1 캐시에의 오염을 줄이게 되어, 선인출 역효과를 최소화할 뿐 아니라 L1 개시 선인출을 가능하게 할 수 있다. 단, 이러한 필터가 적절한 필터링 특성을 가져야만 선인출 이득을 최대한 보존하면서 선인출 역효과를 최소화할 수 있을 것이다.

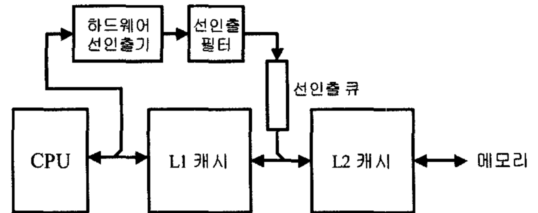


그림 2 선인출 필터를 갖는 하드웨어 L1 캐시 선인출 구조

필터링 특성은 선인출 필터의 구조 및 알고리즘과 밀접한 관련이 있는데, 다음 장에서는 최적의 필터가 가져야 할 특성과 조건을 도출하기 위하여 L1 선인출 역효과에 대한 정성적 및 정량적 분석을 먼저 수행하도록 한다.

3. L1 선인출 역효과에 대한 분석

선인출 역효과에 대한 정량적인 분석을 위하여 먼저 시스템 성능을 정량화하기 위한 FOM(figure of merit)을 정의하고 가상 시스템이라는 개념을 도입하기로 한다.

3.1 가상 시스템 및 실험적인 CPI 정의

시스템 성능은 대개 싸이클 당 명령어 처리 수(instructions per cycle; IPC) 또는 명령어 당 소요 싸이클 수(cycles per instruction; CPI) 중 하나로 표현되어 진다. 본 논문에서는 CPI를 사용하였는데 그 이유는 CPI는 그것에 영향을 주는 각각의 인자들의 합으로 표현될 수 있기 때문이다. 다양한 선인출 알고리즘과 필터링 방식들에 대한 CPI를 추출하기 위하여 캐시 타이밍 시뮬레이터를 개발하여 적용하였다. 시뮬레이터는 L1 캐시, 하드웨어 선인출기, 선인출 큐, 선인출 필터, L2 캐시 및 메모리 모델을 포함하고 있다. 선인출 역효과의 각각의 성분을 평가하기 위하여 표 1에서와 같이 가상 시스템의 실험적인 CPI들을 정의하였다.

실제 시스템의 성능을 나타내는 CPI 외에도, 무한대의 L1/L2 캐시 크기 및 무한대의 메모리 전송 대역폭을 가지는 가상 시스템의 성능을 나타내는 CPI(∞,∞)와, 유한한 캐시 크기와 무한대의 메모리 전송 대역폭을 갖는 시스템에 대한 성능을 CPI(∞)를 각각 추가로 정의하였다. 그러한 모든 CPI들은 선인출을 하는 경우와 하지 않는 경우 각각에 대하여 따로 정의되었다.

표 1 가상 시스템의 CPI들에 대한 정의

Parameter	L1 Cache	# MEM Banks	MEM Bandwidth	Prefetch
요구 인출 CPI	8KB	16	4.3GB/s	no
가장 선인출 CPI	8KB	16	4.3GB/s	fake
선인출 CPI	8KB	16	4.3GB/s	yes
요구 인출 CPI(∞)	8KB	∞	∞	no
선인출 CPI(∞)	8KB	∞	∞	yes
요구 인출 CPI(∞,∞)	∞	∞	∞	no
선인출 CPI(∞,∞)	∞	∞	∞	yes

마지막으로, 특별한 경우인 가장 선인출(fake prefetch) CPI를 정의하였는데, 그것은 모든 선인출 과정이 실제와 동일하게 수행되다가 최종적으로 선인출된 데이타를 실제로 L1 캐시에 저장하는 동작만 일어나지 않는 가상적인 경우에 대한 CPI를 나타낸다. 이러한 가상적인 경우는 선인출로 인하여 메모리 트래픽이 증가되는 효과만을 순수하게 추출하기 위한 것으로서, L1 캐시의 내용을 변화시키지 않은 채로 (즉, 캐시 오염이 없도록) 선인출로 인한 메모리 및 L2 버스 트래픽의 증가를 재현하도록 할 수 있게 하여 준다.

표 2 이상적이지 않은 캐시 선인출에서의 CPI 성분 항목

Parameter	Equation
요구 인출 FCA	= *요구 인출 CPI(∞,∞) – 요구 인출 CPI(∞)*
선인출 FCA	= *선인출 CPI (∞,∞) – 선인출 CPI(∞)*
캐시 오염 손	= *선인출 FCA – 요구 인출 FCA*
트래픽 증가 손실	= *가장 선인출 CPI – 요구 인출 CPI*
선인출 이득	= *선인출 CPI (∞,∞) – 요구 인출 CPI(∞,∞)*
최적 선인출 CPI	= *요구 인출 CPI – 선인출 이득*
예상 선인출 CPI	= *최적 선인출 CPI + 캐시 오염 손실 + 트래픽 증가 손실*
선인출 감소 이득	= *예상 선인출 CPI – 선인출 CPI*

표 2는 최종적으로 선인출로 인한 두 가지 역효과, 즉 캐시 오염 손실과 트래픽 증가 손실을 추출하기 위하여 이상적이지 않은 캐시 선인출에서의 CPI를 구성하는 각각의 항목들을 도출하는 수식들을 요약해서 보여준다. 예를 들면, 선인출로 인한 캐시 오염 손실은 선인출 하지 않는 경우와 선인출을 하는 경우 각각에 대한 '유한 캐시 CPI 증가분(FCA)'들의 차이가 된다[19].

트래픽 증가 손실은 가장 선인출 CPI와 요구 인출 CPI의 차가 된다. 또한 선인출 이득은 최적 선인출 CPI와 요구 인출 CPI로부터 계산할 수 있다. 그러므로, 요구 인출 CPI와 선인출 이득 및 두 가지의 선인출 손실을 이용하여, 시스템의 CPI를 실제로 측정하지 않고도 계산할 수 있다. 이렇게 하여 계산된 예상 선인출 CPI는 특히 공격적인 선인출 알고리즘의 경우 실제 측정된 선인출 CPI보다 더 큰 값이 될 수가 있다. 이러한 오차는 추출된 선인출 CPI 증가분 (FCA)이 무한대의 메모리 전송 대역폭을 가지는 조건 하에서 추출되었기 때문에 실제보다 더 과장된 값을 가지기 때문이다.

메모리 전송 대역폭이 유한한 경우에는, 선인출 큐에 저장되는 모든 선인출 요청이 L2 캐시로 전송되지는 않는다. 그 이유는 선인출 요청은 요구 인출에 의한 요청보다 더 낮은 우선순위를 갖도록 되어 있어서 L2 캐시는 이전의 요구 인출들을 모두 처리하고 난 후에야 비로소 다시 요청 가능한 상태로 되기 때문에, 선인출 큐에 저장되는 선인출 요청들 중 많은 수가 새로운 선인출 요구가 접수됨에 따라 선인출 큐로부터 축출되어 사라져 버릴 것이기 때문이다.

표 3 가상 시스템의 실험적인 CPI 및 선인출 손실 성분에 대한 실험 결과

Benchmark		176.gcc		181.mcf	
Prefetch scheme		A[1:1]	A[1:2]	A[1:1]	A[1:2]
Measured	*요구 인출 CPI*	2.077		1.932	
	가장 선인출 CPI	2.847	3.054	2.290	2.514
	선인출 CPI	2.428	2.467	2.136	2.347
	요구 인출 CPI(∞)	2.076		1.930	
	선인출 CPI(∞)	2.066	2.246	1.896	1.994
	요구 인출 CPI(∞,∞)	1.385		1.108	
	선인출 CPI(∞,∞)	1.205	1.130	1.043	1.021
Estimated	*최적 선인출 CPI*	1.898	1.822	1.867	1.845
	캐시 오염 손실	0.170	0.425	0.032	0.151
	트래픽 증가 손실	0.770	0.977	0.358	0.582
	예상 선인출 CPI	2.838	3.224	2.256	2.578
	선인출 감소 이득	0.410	0.758	0.120	0.230

결론적으로, 계산된 예상 선인출 CPI와 실제 선인출 CPI간의 오차는 그러한 선인출들을 잃어버림으로 인하여 감소된 캐시 오염 성분이라고 할 수 있다. 매우 정교한 선인출 알고리즘의 경우에는 이 오차가 음의 값을 가질 수도 있다.

표 3은 두개의 SPEC2000 벤치마크들에 대한 두 가지의 과도하게 공격적인 선인출 알고리즘에서의 실험적인 CPI들에 대한 실험 결과를 보여 준다. A[1:N]은 입력된 주소의 다음 블록부터 N개의 연속적인 블록들을 선인출 하도록 한 매우 공격적인(Aggressive) 인출 방식이다. 즉, A[1:1]은 기존의 OBL[3] 또는 비 태그방식(non-tagged) NSP에 해당한다[4]. 또한, A[1:2]의 경우에는 입력된 블록 주소를 b라고 할 때에 (b+1)과 (b+2)의 블록주소의 데이타들을 순차적으로 선인출 한다.

실험 결과들은 선인출 역효과의 두 가지 성분들, 즉 캐시 오염 손실과 트래픽 증가 손실 모두 좀 더 공격적인 선인출 알고리즘에 대해서 더욱 증가함을 보여준다. 또한, 더욱 공격적인 선인출에서는 캐시 오염 손실은 급속히 증가되는 반면 트래픽 증가 손실은 상대적으로 거의 증가하지 않는 것을 볼 수 있는데, 이것은 잃어버린 선인출들의 수가 더 많아지기 때문에 실제로 트래픽 증가는 거의 포화되기 때문이다.

한편 최적 선인출 CPI는 선인출 알고리즘이 더욱 공격적이 됨에 따라 선인출 성능도 지속적으로 좋아질 것임을 보여 준다. 이것은 극도로 공격적인 선인출 알고리즘이 적절하게 고안된 선인출 필터와 함께 결합될 수 있다면 매우 우수한 시스템 성능을 가질 수 있을 것이라는 가능성을 보여주는 것이라고 할 수 있다.

3.2 선인출 손실 성분에 대한 분석

표 4는 모든 메모리 접근 명령에 대하여 2개의 선인출을 개시하도록 되어 있는 매우 공격적인 선인출 알고리즘인 A[1:2]에 대한 선인출 손실을 시뮬레이션으로 분석한 결과를 보여준다.

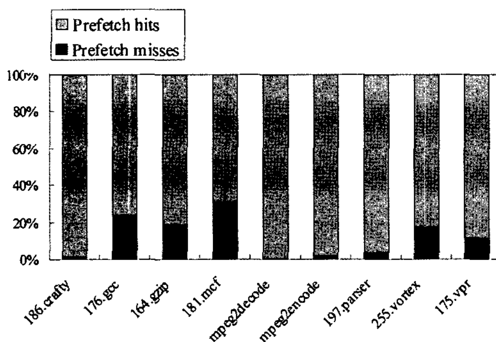
9가지의 벤치마크들 중에서, 4개의 벤치마크들의 선인출 CPI는 선인출 손실들로 이하여 요구 인출 CPI보다도 오히려 나쁜 것을 볼 수 있다. 또한, mpeg2decode를 제외한 거의 모든 벤치마크에 있어서 공통적으로 트래픽 증가 손실이 전체 선인출 손실의 대부분의 비율을 차지한다. 한편, 트래픽 증가 손실은 전체 선인출 페치수에 비례하게 된다.

그림 3(a)는 전체 선인출 페치의 대부분의 비율을 선인출 히트가 차지함을 보여 준다. 따라서, 선인출 히트 수를 감소시키는 것이 적절한 선인출 필터가 가져야 할 첫번째 요건이라고 할 수 있다. 선인출 히트 수를 감소시킬 수 있는 또 다른 방법이 있을 수 있는데, 그것은 L2 캐시로 선인출을 요청하기 전에 먼저 L1 캐시를 룩업 하는 통상적인 방법이다. 이러한 경우에는, 모든 선인출 히트들이 미연에 방지될 것이고 따라서 선인출 히트로 인한 트래픽의 증가도 없을 것이다. 그러나, 선인출이 발생할 때마다 항상 L1 캐시를 룩업 하게 되면 모든 선인출 요청들이 사용 가능한 L1 포트들 중 하나를 항상 점유하게 되고 이로 말미암아 시스템 성능은 특히 공격적인 선인출의 경우 극도로 저하되게 된다.

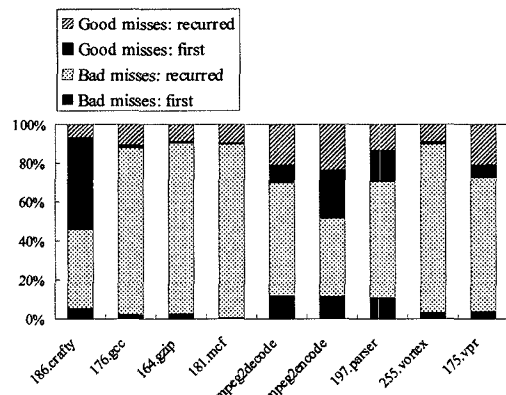
선인출 큐에 적재되어 있는 선인출 요청이 하나라도

표 4 선인출 A[1:2]의 역효과 성분 분석 결과

Benchmark	요구 인출 CPI	선인출 CPI	최적 선인출 CPI	선인출 손실	
				캐시 오염	트래픽 증가
186.crafty	1.666	1.205	1.158	1.9 %	98.1 %
176.gcc	2.077	2.466	1.822	30.3 %	69.7 %
164.gzip	1.605	1.901	1.576	37.0 %	63.0 %
181.mcf	1.932	2.347	1.845	20.6 %	79.4 %
mpeg2decode	1.044	1.025	1.013	52.1 %	47.9 %
mpeg2encode	1.155	1.057	1.035	8.4 %	91.6 %
197.parser	1.442	1.266	1.190	7.5 %	92.5 %
255.vortex	1.704	2.002	1.559	33.5 %	66.5 %
175.vpr	1.787	1.574	1.467	19.3 %	80.7 %



(a) Prefetch fetches



(b) Prefetch misses

그림 3 A[1:2]에 대한 (a) 선인출 페치 및 (b) 미스의 성분 분석

있고 또한 비어 있는 L1 포트가 있을 때에 그 포트는 선인출 요청을 위하여 할당될 것인데, 공격적인 선인출의 경우 거의 항상 요구 인출 또는 선인출 요청에 의하여 모든 L1 포트가 점유되고 있을 것이므로 CPU의 입장에서는 사용 가능한 L1 포트 의 수가 평균적으로 매우 줄어들게 되기 때문이다. 만일 선인출 필터가 선인출 히트도 또한 줄여줄 수가 있다면, 선인출 시에 L1 캐시를 먼저 룩업 하지 않고 바로 다음 단계의 캐시로 선인출 요청을 전송하는 것이 전체 시스템 성능에 대하여 훨씬 이득이 될 것이다. 별도의 실험 결과에 따르면, 이러한 필터링을 사용하지 않는 경우에 대해서도, 선인출 시에 L1 캐시를 먼저 룩업 하는 것은 그렇지 않은 경우에 비하여 CPI를 평균적으로 30% 악화시킨다.

선인출 미스의 경우 캐시로 저장된 해당 데이타가 그 뒤에 CPU에 의해 요구되어 사용이 되었었는지 여부에 따라서 좋은 선인출과 나쁜 선인출로 분류된다. 만일 선인출된 데이타가 사용이 되었었다면, 그것은 그 데이타가 캐시로 선인출 되지 않았을 경우에 발생하였을 요구 미스를 최소한 한 개 이상 방지하였다는 것을 의미한다. 좋은 또는 나쁜 선인출의 수는 새로운 요구 데이타 또는 선인출 데이타가 캐시로 저장될 때마다 캐시로부터 축출되는 데이타들을 산출함으로써 측정될 수 있다. 즉, 축출된 데이타가 사용되지 않은 채 축출된 선인출 데이타라면 나쁜 선인출 미스 수가 1 증가되고 그렇지 않으면 좋은 선인출 미스 수가 1 증가된다. 선인출 미스 수의 전체 개수는 L1 캐시 라인의 개수에 비해서 훨씬 더 많기 때문에, 시뮬레이션이 끝난 시점에서 아직 축출되지 않고 L1 캐시 내에 남아 있는 선인출 데이타들은 무시할 수 있다.

각각의 선인출 미스 수를 측정하기 위하여, 모든 캐시 라인에는 추가적으로 2개의 비트가 더 저장이 되어야 하며, 이들 추가적인 데이타는 다음 장에서 다룰 선인출 필터를 위한 정보를 위해서도 필요하다. 추가되어야 할 첫번째 비트는 해당 라인이 요구 인출에 의하여 캐시로 들어오게 된 것인지 아니면 선인출 페치에 의하여 오게 된 것인지를 가리키는 비트로서 축출된 데이타가 선인출 데이타인지 요구 데이타인지를 구분할 수 있게 하여 주고, 또 다른 비트(used_bit)는 해당 데이타가 캐시 내에 있을 동안 CPU에 의해서 실제로 사용이 되었었는지 여부를 나타낸다[17].

한편, 동일한 데이타가 축출되었다가 다시 캐시로 들어오는 일이 반복해서 일어날 수가 있다. 왜냐하면, 좋은 선인출 데이타의 경우에는 시간적 국소성에 의해서 축출된 이후 CPU가 또다시 그 데이타를 요구할 가능성이 높기 때문이고, 나쁜 선인출 데이타의 경우에도 해당 선인출을 발생하도록 한 요구 데이타를 또다시 CPU가 요청하게 되면 하드웨어 선인출기는 이전과 동일한 선인출을 다시 발생시킬 것이기 때문이다. 이와 같은 동일한 데이타의 반복적인 축출 여부를 시뮬레이션이 진행되는 동안 감시하여 첫번째 축출과 재발된 축출 각각의 회수를 구분할 수 있도록 하였다.

그림 3(b)는 선인출 미스의 각각의 성분을 보여준다. 좋은 선인출 미스와 나쁜 선인출 미스 수 간의 비율은 선인출의 정확도를 대변한다. 전체 나쁜 선인출 미스 중에서 재발된 미스가 상당히 높은 비중을 차지하고 있다는 것은 매우 주목할 만하다. LRU(least recently used) 정책에 의하여 좋은 선인출 데이타는 나쁜 선인출 데이타보다 캐시 내에서 훨씬 더 긴 생존 시간을 가질 것이다. 좋은 선인출 데이타는 CPU에 의하여 여러 번 사용이 될 것이기 때문에 동일한 set 내에서 축출 우선 순위가 가장 낮은 곳으로 재차 옮겨질 것이므로 축출되지 않고 오랫동안 생존할 수 있을 것이다.

그림 4는 좋은 선인출 미스들과 나쁜 선인출 미스들 각각의 생존 시간 - 페치 되었을 때부터 축출될 때까지의 싸이클 수 - 의 분포를 보여 준다. 좋은 미스의 평균 생존 시간은 나쁜 미스의 생존 시간보다 약 2.5배 길다는 것을 보여 준다. 참고로, 좋은 미스 데이타가 캐시 내에서 생존 시간 동안 CPU에 의해 사용된 평균 회수는 여러 벤치마크에 대하여 6회에서 60회 정도인 것을 알 수 있었다.

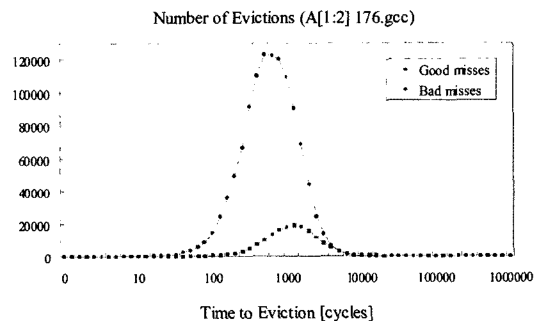


그림 4 선인출 데이타들의 생존 기간 분포도

결론적으로, 나쁜 선인출 미스들은 그들의 짧은 생존 시간에 의해서 여러 번 반복되어 발생되는 정도가 좋은 선인출 미스들에 비해서 더 많고, 또한 그것들은 전체 나쁜 선인출 미스의 거의 대부분을 차지한다. 그러므로, 적절한 선인출 필터가 가져야 할 두 번째 요건은 나쁜 선인출 미스가 재발되는 것을 방지하는 것이며 이러한 재발을 방지하는 것만으로도 나쁜 선인출 미스의 거의 대부분을 제거할 수 있을 것임을 알 수 있다.

한편, 좋은 선인출 미스는 오히려 계속해서 재발되는 것을 허용하여야 하는데 그 이유는 그것들은 순전히 제

한된 캐시 크기로 인하여 축출되었을 뿐이며 따라서 축출된 뒤 CPU로부터 또다시 요구되기 전에 가급적 빨리 캐시로 다시 진입하는 것이 바람직하기 때문이다.

4. 직접 사상 필터링 캐시를 이용한 선인출 필터

이 장에서는 앞에서 도출된 요건들을 바탕으로 적절한 선인출 필터 가 구현된다. 적절한 선인출 필터의 요건을 다시 요약해 보면 다음과 같다.

- 트래픽 증가를 최소화하기 위하여 선인출 히트를 감소시킬 수 있어야 함
- 캐시 오염을 최소화하기 위하여 나쁜 미스가 재발되는 것을 방지해야 함

그림 5는 하드웨어 선인출기의 뒤 단에 직렬로 연결된 하드웨어 선인출 필터를 갖는 하드웨어 L1 캐시 선인출 구조를 보여 준다.

선인출 필터의 필터링 특성을 규정하는 내부 변수들은 수행 중에 수집된 정보를 이용하여 실시간으로 동적으로 갱신된다. 이렇게 함으로써 변화하는 환경에 대하여 필터를 최적으로 유지할 수 있다. 미스 또는 히트 뿐 아니라 축출되는 데이타 등 L1 캐시 제어기로부터의 모든 정보들이 필터를 갱신하기 위하여 사용될 수 있다.

반면에, Zhuang과 Lee는 축출 데이타 정보만을 사용하여 필터를 갱신하였다[17]. 그들의 필터는 일련의 엔트리들로 구현이 되는데, 전체 메모리 공간이 전체 엔트리들로 일대일 사상(mapping)이 된다. 그것을 위하여, 전체 주소공간을 엔트리의 수와 같은 수의 부분 공간들로 나눈다. 각각의 엔트리들은 고유의 값을 가지고 있게 되는데 그 값은 해당되는 부분 주소공간 내에 좋은 선인출 주소와 나쁜 선인출 주소들이 평균적으로 얼마나 많이 포함되어 있는지를 나타낸다. 그 값은 처음에는 최소값과 최대값의 중간 값으로 설정이 되고 만일 좋은 선인출 데이타가 축출되면 1씩 증가가 되고 나쁜 데이타의 경우 1씩 감소된다. 해당 엔트리의 값이 중간 값보다 크거나 같으면 해당되는 부분 주소공간에 속하는 주소를 갖는 선인출 요청은 선인출 필터에 의해 패스가 되고, 중간 값보다 작은 경우에는 차단이 된다.

해싱(hashing) 함수는 임의의 주소를 특정한 부분 주소공간으로 사상하는 역할을 한다. 그러므로 이후부터 이러한 필터링 방식을 'HAFT'(Hashing and Averaging Filtering Table) 방식이라고 부르기로 한다. HAFT는 같은 해시 주소를 갖는 주소들에 대해서 좋은 선인출 축출 수와 나쁜 선인출 축출 수의 평균을 취하면 그것이 그 주소들의 평균 선인출 정확도를 대변할 것이라는 가정에 기반하고 있다. 그러나 필터링-선인출-축출 루프가 선형적인 부의 피이드백(negative feedback) 루프가 아닌 비선형/비대칭적인 정의 피이드백(positive feedback) 루프임을 고려하면(그림 6 참조), 그러한 평균값이 그들 주소에 대한 선인출을 패스 또는 차단시키기

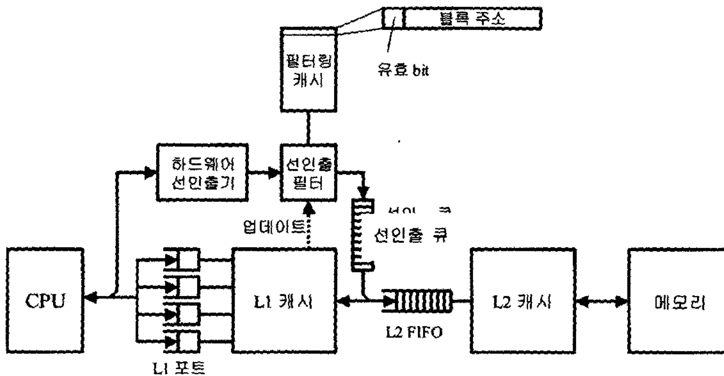


그림 5 선인출 필터를 가지는 비경쟁 L1 선인출 시스템 구조

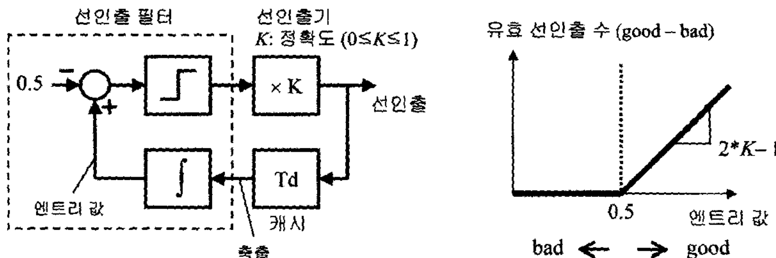


그림 6 HAFT 방식의 블록도 및 전달 함수

위한 선택의 기준으로 쓰이도록 피이드백 되는 경우에 그러한 가정은 더 이상 유효하지 않게 된다.

즉, 엔트리 값이 일시적으로 bad 영역으로 이동한 이후에는 다시 good 영역으로 되돌아가기가 거의 불가능해지기 때문이다. 왜냐하면, 그 엔트리에 대응되는 모든 주소들의 선인출이 완전히 차단되기 때문에, 아직 축출되지 않고 캐시 내에 남아 있는 경우를 제외하고는 좋은 선인출 데이타의 축출을 원천적으로 기대할 수가 없기 때문이다. 이렇게 bad 영역에 갇히게 되는 현상은, 축출이라는 것이 이산 시간적인 사건이고 또한 실제 구현에서는 엔트리 값도 양자화된 불연속적인 값([17]의 경우 4가지 값)을 가지기 때문에 실제로 매우 쉽게 일어날 수 있다. 이러한 이유로 인하여, 대부분의 엔트리들은 해당 주소공간 내에 나쁜 선인출 주소보다 좋은 선인출 주소가 훨씬 더 우세하게 많은 특수한 경우를 제외하고는 거의 모두 '필터링' 상태에 있게 된다. 그림 7과 그림 8은 그러한 상황을 도식적으로 설명하여 준다.

그림 7의 각각의 픽셀은 각각의 부분 주소공간 또는 필터링 엔트리에 해당한다. 각 픽셀의 밝기는 좋은 선인출 수와 나쁜 선인출 수 간의 비율, 즉 선인출 정확도에 해당이 되고 더 상세하게는 각 픽셀의 그레이 코드는 다음 수식과 같이 계산된다.

$$Gray\ code = 128 + 64 * log_2(good / bad) \quad (1)$$

그러므로, 흰색에 가까운 픽셀은 좋은 선인출 수가 지배적이라는 것을 의미하며 검은색에 가까운 픽셀은 나쁜 선인출 수가 지배적이라는 것을 의미한다. 좋은 선인출과 나쁜 선인출의 비율이 대략 0.5에서 2 사이인 픽셀들은 회색 근처의 색을 갖게 된다. 여러 벤치마크들에 대하여 전반적으로 회색 픽셀들이 많다는 것은 해시 함수를 사용하여 부분 주소공간을 구분할 경우, 좋은 영역과 나쁜 영역이 잘 구분될 수 없다는 것을 의미한다.

더욱이, 그림 8은 시뮬레이션이 끝난 뒤의 각 엔트리의 최종 상태를 보여주는데, 흰색 픽셀은 '패스' 상태, 검은 픽셀은 '필터링' 상태임을 뜻한다. 그림 8을 보면, 앞에서 예측된 바와 같이 그림 7에서 거의 흰색이었던 픽셀을 제외한 모든 픽셀들의 상태가 '필터링' 상태에 들어가 있음을 알 수 있다. 이것은 해싱으로 주소공간을 나누고 동일 부분공간 내의 주소들 간에 평균값을 취하는 것은 각 부분공간의 평균 선인출 정확도를 나타낼 수 없으며 또한 거의 모든 부분공간 들의 선인출을 모두 차단하게 될 것이라는 것을 알려 주는 것이다.

해싱 또는 평균을 취하는 등의 방법으로 전체 메모리 주소공간을 모두 관장하려고 시도하는 대신에, 본 논문에서 제안하는 필터 구조는 전체 주소 중에서 극히 일부의 주소만을 선택하여 리스트에 저장한다. 저장되는 주소들은 필터에 의해서 차단되어져야 하는 선인출 데이타들의 주소들이다. 주소 리스트는 제한된 크기를 갖는데 그것은 그 리스트가 물리적인 테이블의 형태로 구현될 것이기 때문이며, 그러한 테이블은 궁극적으로 직접 사상(direct mapped) 캐시로서 구현되어진다. 선인출 요청이 필터로 입력되면, 입력된 블록 주소의, 예를 들면 하위 10비트를 이용하여 특정 엔트리가 지목이 된다 (또는 블록 주소의 전체 비트를 해싱한 결과인 10비트를 이용할 수도 있다). 그리고 지목된 엔트리의 유효 비트가 검사되고 또한 엔트리의 태그가 입력된 주소의 해당 부분과 비교된다. 만일 유효 비트가 활성화되어 있고 태그가 일치한다면, 입력된 선인출은 차단되고 선인출 요청은 무시된다. 유효 비트가 활성화되어 있지 않거나 태그가 일치하지 않는다면 선인출 요청은 다음 수준인 L2 캐시로 전달이 된다.

표 5는 여러 가지 벤치마크에 대하여 다양한 선인출 알고리즘에서 시뮬레이션 동안 출현하는 모든 구별 가

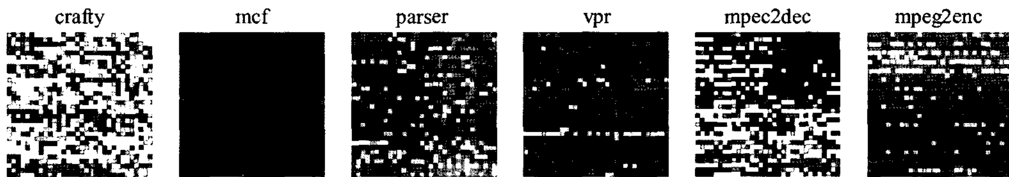


그림 7 2차원 해시 주소 상의 선인출 정확도 지도

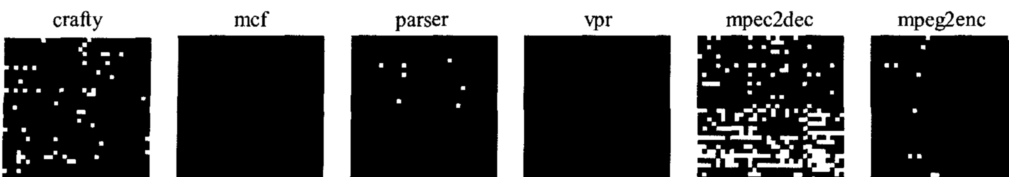


그림 8 2차원 해시 주소 상의 필터링 상태 지도

표 5 공격적인 선인출기에 대한 선인출 주소의 가짓수 (백만 트레이스 당)

Benchmark	A[1:1]	A[1:2]	A[1:4]	A[1:8]	A[1:16]
186.crafty	6,352	6,493	6,715	6,775	6,891
176.gcc	3,819	4,226	4,862	5,024	5,370
164.gzip	5,253	5,869	7,063	7,624	12,267
181.mcf	157,782	151,895	152,076	152,124	303,109
mpeg2decode	349	366	401	412	417
mpeg2encode	1,665	1,722	1,771	1,784	1,793
197.parser	3,106	3,219	3,397	3,445	3,518
255.vortex	12,134	12,497	13,115	13,271	19,293
175.vpr	3,978	4,139	4,401	4,473	4,572

능한 선인출 블록 주소들의 수를 보여 주는데, 출현하는 선인출 주소의 가짓수는 백만 트레이스 당 수천 개에 불과할 정도로 예상보다 그리 많지 않음을 알 수 있다.

A[1:1]의 경우는 선인출 주소의 수가 바로 요구 주소의 수를 나타낸다. A[1:N]의 선인출 수는 N에 대하여 직접적으로 비례하지는 않는다. 왜냐하면, 공간적 국소성으로 인하여 많은 선인출 주소들이 서로 대부분 겹쳐 있기 때문이다. 아무튼, 단지 수천개의 엔트리를 갖는 필터링 테이블만으로도 출현하는 모든 선인출 주소들을 낱낱이 통제할 수가 있음을 알 수 있다.

표 6은 필터링 테이블, 즉 직접 사상 필터링 캐시(Direct-Mapped Filtering Cache; 이하 DMFC 구조)의 파라미터들을 보여준다. 메모리 주소가 32비트로 표현이 되고 8KB L1 캐시의 블록 사이즈를 32바이트라고 가정하면, 블록 주소는 27비트의 길이를 가지게 된다. 필터링 테이블이 1024개의 엔트리를 가진다고 하면, 태그의 길이는 17비트가 되고 각 엔트리의 크기는 유효비트를 포함하여 18비트가 된다. 제안된 필터 구조에서는 각 엔트리가 특정한 하나의 주소로만 대응되기 때문에 HAFT 구조에서와 같은 일종의 알리아싱(aliasing)에 의한 문제는 발생되지 않을 것이다.

표 6 직접 사상 필터링 캐시 (DMFC)의 파라미터 값

Filtering Cache	
Capacity	2.3KB
# Entries	1024
Entry size	18bits
Associativity	1
# Ports	1
Lookup [cycles]	1

제안된 필터는 자신의 특성을 갱신하기 위하여 L1 캐시로부터의 축출, 미스 및 히트 정보를 이용하게 된다. 표 7에서와 같이 필터가 갱신될 수 있는 모든 조건은 7가지로 구분된다.

표 7 직접 사상 필터링 캐시(DMFC) 방식의 필터 갱신 조건

Event	Condition		Action	Result
축출	요구 데이터		-	-
	선인출 데이터	미사용	리스트에 추가	선인출하지 않음
		사용	리스트에서 제거	선인출함
요구 페치	미스		리스트에서 제거	선인출함
	히트		-	-
선인출 페치	미스		-	-
	히트		리스트에 추가	선인출하지 않음

그 7가지 조건들 중에서 4가지의 경우에만 실제로 필터가 갱신되도록 하였다. 먼저 다음의 2가지 경우는 앞에서 이미 언급된 바와 같다.

1. 사용되지 않은 (나쁜) 선인출 데이타가 캐시로부터 축출되면, 이후의 캐시 오염을 줄이기 위하여 그 주소가 필터링 리스트에 추가되어 동일한 선인출의 재발을 방지한다.
2. 어떤 선인출 요청이 선인출 히트인 것으로 밝혀지면, 이후의 트래픽 증가를 억제하기 위하여 그 주소가 필터링 리스트에 추가되어 동일한 주소로의 선인출 요청을 차단한다.

또한, 다음과 같은 2가지 조건을 더 추가하였다.

3. 사용되었던 (좋은) 선인출 데이타가 축출되면, 그 주소가 필터링 리스트에 등재되어 있는지를 검사하여 만일 있다면 리스트에서 제거한다.

이 조건은 해당 주소가 유익한 선인출 주소임이 틀림없는데도 불구하고 유한한 캐시 용량으로 인하여 축출이 되었기 때문에 다시 캐시로 진입하는 것을 허용하기 위해서 필요하다. 마지막 조건은 다음과 같다.

4. 요구 미스가 발생하면, 그 주소가 필터링 리스트에 등재되어 있는지를 검사하여 만일 있다면 리스트에서 제거한다.

이 조건은 유익하지만 불운한 선인출, 즉 조금 일찍 또는 조금 늦게 선인출 되었거나 일시적으로 캐시 미스가 증가되어 많은 데이타들이 캐시로 대거 유입되었기 때문에 사용되기 전에 축출이 된 잠재적으로 좋은 선인출 데이타를 구원하기 위해서 필요하다. 이 조건이 없다면, 필터는 여러 가지 이유로 인하여 한 번 잘못 등재된 엔트리를 다시 회복시킬 수가 없게 된다. 또한 이 조건은 변화하는 환경에 대하여 필터를 동적으로 적응시키기 위해서도 필요하다고 할 수 있다.

그림 9는 제안된 선인출 및 필터링 알고리즘의 전체 흐름도를 보여준다.

5. 실험 결과

5.1 시뮬레이션 환경

캐시 시뮬레이션을 위한 시스템 파라미터들을 표 8에

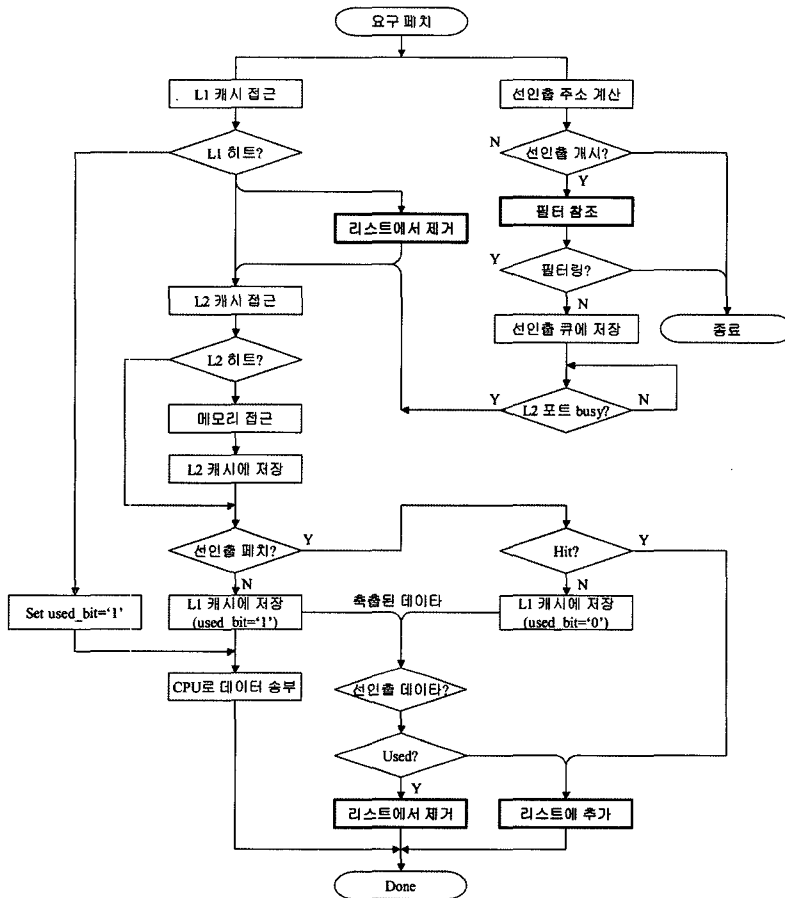


그림 9 제안된 필터링 알고리즘의 흐름도

표 8 캐시 시뮬레이터의 시스템 파라미터

L1 cache	
Capacity	8KB
Line size	32bytes
Associativity	4
# Ports	4
Lookup [cycles]	1
Latency[1)] [cycles]	2
Replacement	LRU
Prefetch queue depth	32

L2 cache	
Capacity	512KB
Line size	64bytes
Associativity	8
# Ports	1
Lookup [cycles[2)]]	6
Latency[1)] [cycles[2)]]	18
Replacement	LRU
Miss FIFO depth	8

Memory	
Capacity	4GB
Line size	64bytes
# Banks	16
Latency [cycles[2)]]	218
Burst length [cycles[2)]]	32
Bus width	8bytes
Per-pin data rate	533Mbps
I/O bandwidth	4.27GB/s

1) includes lookup and transfer cycles. 2) CPU cycles.

요약하였다.

CYSIM이라는 타이밍 기반 트레이스 구동 캐시 시뮬레이터를 본 연구를 위하여 새로이 개발하여 적용하였다. 트레이스는 ATOM[20] 시뮬레이터를 이용하여 SPEC2000 중 7개의 벤치마크들과 2가지의 대표적인 멀티미디어 벤치마크들에 대하여 추출하였다. CPU는 메모리 접근 명령을 제외하고는 모든 명령어들을 한 싸이클 내에 처리할 수 있으며 또한 이상적인 명령어 캐시를 가지고 있다고 가정하였다[8]. 그러므로, CPI는 시스템의 메모리 접근 성능에 의해서만 좌우될 것이다. L1 및 L2 캐시의 기본 파라미터들은 인텔 사의 Xeon [21] 프로세서의 경우를 사용하였으며 CPU 클록 주파

스는 2.13GHz로 가정하였고 이것은 266MHz의 메모리 클록 주파수의 8배가 된다.

주 메모리로서 533Mbps DDR2 SDRAM 시스템을 가정하였다. 주 메모리는 64bit 단일 채널을 가지면 2개의 2GByte DIMM(Dual In-line Memory Module)로 구성되는데 각각의 DIMM은 16개의 1Gb DDR2 SDRAM 칩을 가진다고 가정하였다. 1Gb 이상의 DDR2 메모리 칩의 경우 총 8개의 온-칩 뱅크(bank)를 가지게 되기 때문에, 4개의 외부 뱅크 또는 랭크(rank)를 포함하면 전체적인 메모리 뱅크의 개수는 32개가 된다. 그러나, 1Gb이라는 높은 집적도로 인한 뱅크 인터리빙(bank interleaving) 시의 타이밍 제약(tFAW; four bank activation window 등의)을 없애기 위하여 시뮬레이션에서는 전체 메모리 뱅크의 수를 16이라고 가정하였다.

한 번의 메모리 접근에 대한 데이타 전송 시간은 4개의 메모리 클록 싸이클 수 또는 더블 데이타 전송 기준으로 BL(burst length)는 8이 되며 이것은 32개의 CPU 클록 싸이클 수가 된다.

L1 및 L2 캐시 모두 비차단(non-blocking) 캐시를 가정하였다. L1 캐시는 표 8에 나와 있는 바와 같이 4개의 포트를 가질 것이므로, 3개의 미스를 처리하고 있는 동안 1개의 히트를 처리할 수 있다. 4개의 미스를 처리하고 있는 동안에는, 4개의 미스 데이타 중 적어도 한 개의 데이타가 도착되어 CPU로 송부되고 해당 포트의 상태가 idle 상태로 바뀌기 전까지는 캐시로의 접근이 차단된다. L2 캐시는 단 한 개의 포트를 가짐에도 불구하고 비차단 캐시로서 간주할 수 있는데, 그것은 내부의 FIFO를 이용하여 여러 개의 접근 동작들이 서로 겹쳐질 수 있도록 되어 있기 때문이다.

캐시로의 각각의 접근 요청은 캐시 내부에서 완전한 파이프라인 방식으로 처리되어, 예를 들면 한 포트가 캐시로부터 데이타를 읽어 내고 있는 동안 또 다른 포트가 캐시를 lookup 하는 것이 가능하다. 다수개의 포트를 갖는 비차단 L1 캐시라는 것은 CPU의 OOO(out-of-order) 실행을 지원한다는 의미가 된다.

그러나, CPU의 내부 레지스터 정보를 포함하는 트레이스를 구하는 것이 불가능하였기 때문에, 명령어들 간의 데이타 의존성을 정확하게 고려할 수는 없었다. 대신에, 임의의 두 명령어 간에 데이타 의존성이 존재하지 않게 되는 명령어 간의 최대 거리를 모든 벤치마크에 대하여 4행으로 제한하였다. 즉, 어떤 명령어 행에 대해서 미스가 발생하였고 그 이전에 아직 미처리된 미스는 없다고 가정하면 최대 3개까지의 명령어들이 계속해서 처리될 수 있다. 그렇지만, 미스가 된 데이타가 도착할 때 까지 다음 4번째의 명령어는 처리되지 않고 CPU는 정지될 것이다. 이러한 일률적인 단순화는 시뮬레이션된 CPI와 실제 시스템에서의 CPI 간에 차이를 유발할 것이지만, 동일한 벤치마크에 대하여 서로 다른 선인출 또는 필터링 방식들간의 상대적인 성능 비교는 여전히 유효할 것이다.

5.2 시뮬레이션 결과

과도하게 공격적인 하드웨어 선인출기와 직접 사상 필터링 캐시(DMFC)를 이용한 하드웨어 선인출 필터를 갖는 제안된 L1 캐시 선인출 방식이 시뮬레이션을 통하여 실험되었고, 특히 필터링 방식은 기존의 해싱 및 평균(HAFT) 방식 및 필터링 하지 않는 경우와의 비교를 수행하였다.

그림 10은 제안된 DMFC 필터링은 프로그램 수행 시간이 경과함에 따라서 필터링 하지 않는 경우에 비해 일정하게 감소된 선인출 요청 수를 유지하는 반면, 기존의 HAFT 방식은 일정 시간이 지난 뒤에는 거의 모든 선인출들을 차단하게 된다는 것을 보여준다.

이것은 앞에서 언급한 것과 같은 알리아싱에 따른 과도한 필터링 효과로 인한 것이다. 또한 제안된 방식에서는 수백만 싸이클 이내에 새로운 어플리케이션 환경에의 적응이 완료된다는 것도 또한 보여 준다.

그림 11은 9개의 벤치마크 및 8가지의 선인출 알고리즘에 대하여 4가지 필터링 방식의 CPI를 비교한다(no filter, HAFT, DMFC, 이상적인 선인출). 공격적인 선인출 알고리즘들로서 N이 1부터 16까지의 값을 갖는 A[1:N]를 사용하여 조금씩 다른 선인출 강도를 갖는 경우에 대한 비교를 할 수 있도록 하였다.

기존의 공격적인 선인출기로서 네이버를 포함하였는데, 이것은 이미지 프로세스 어플리케이션을 위하여 주변에 인접한 8개의 데이타를 선인출하는 상당히 공격적인 선인출 알고리즘이다[5]. RPT는 그다지 공격적이지 않은 정교한 하드웨어 선인출기의 일종으로서 선택되었다[7][8]. 수정된 RPT(M-RPT)는 원래 RPT의 정교한 주소간격 예측 메커니즘을 최대한 이용하면서 좀 더 많은 선인출 요청을 발생하도록 수정을 가해서 만든 공격적인 선인출 알고리즘이다[23]. 그림 11의 CPI 값들은 각각의 벤치마크에 대한 요구 인출 CPI 값으로 정규화한 값들이다.

다음의 특징적인 결과들이 주목할 만하다.

1. HAFT 방식의 CPI는 요구 인출 CPI와 거의 동등하다. 이것은 HAFT 필터링이 나쁜 선인출 뿐만 아니라 좋은 선인출까지도 대부분 제거해 버림으로써 선인출로 인한 대부분의 이득을 잃어버리게 되기 때문이다.
2. 제안된 DMFC는 모든 경우에 있어서 필터링 하지 않는 경우보다 항상 더 좋은 성능을 보인다.
3. DMFC의 CPI는 일부의 경우를 제외하고는 최적 선인출 CPI와 유사한 값을 갖는다. (예외: A[1:4]이상

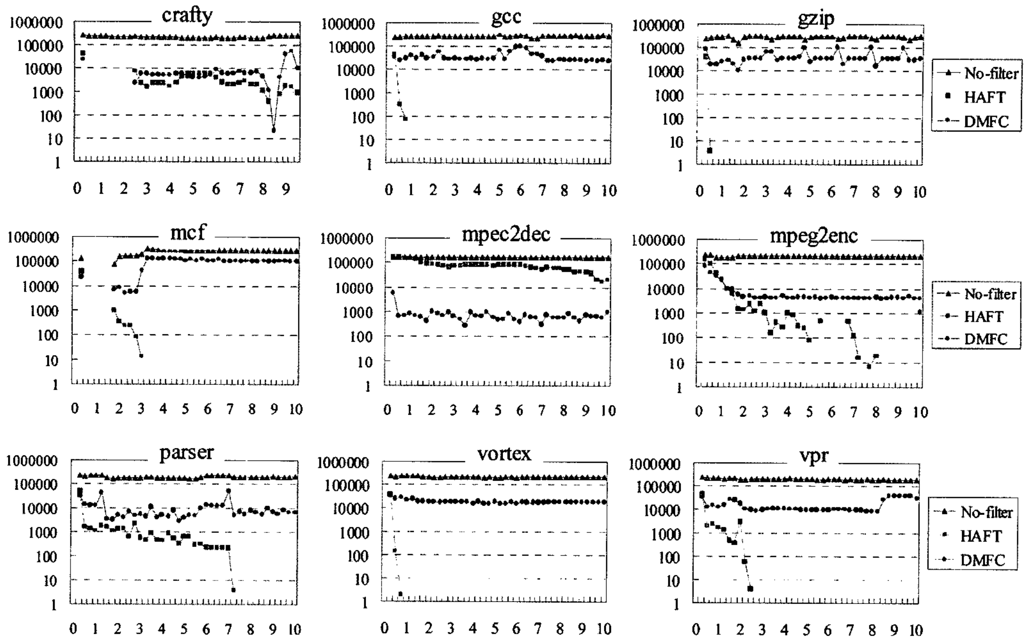


그림 10 3가지 필터링 방식에 대한 A[1:2]의 시간에 따른 선인출 행태
(X-축: 백만 트레이스, Y-축: 필터를 통과한 선인출 요청 수)

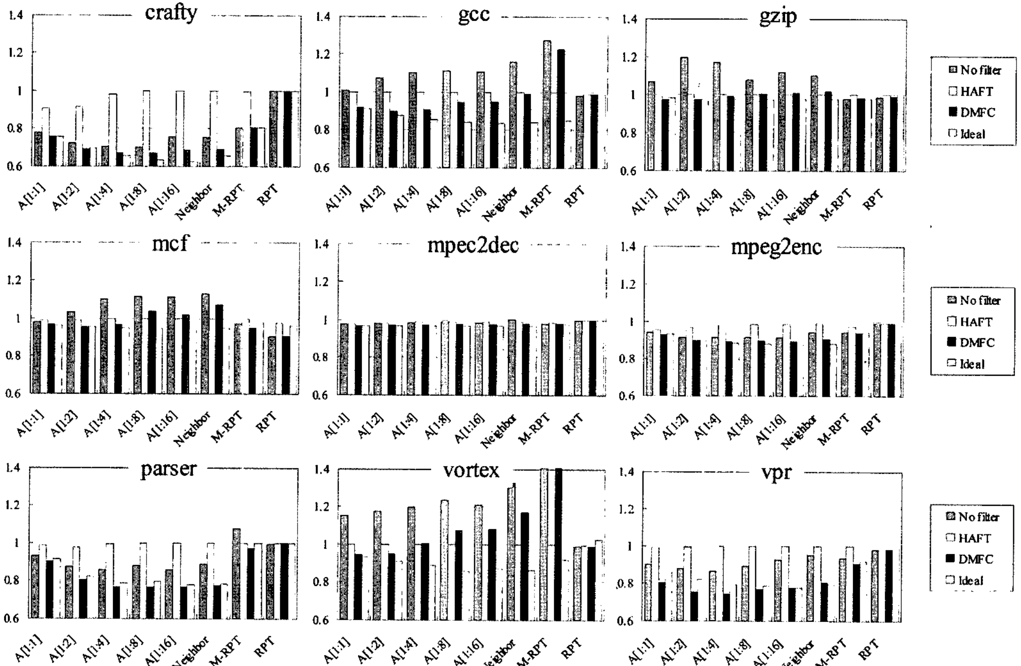


그림 11 다양한 필터링 방식에 대한 CPI 비교

의 매우 공격적인 선인출에서의 mcf 및 vortex의 경우)

4. A[1:2]와 DMFC의 조합이 대체적으로 최적의 성능을 갖는다.

따라서, 표 9에서는 A[1:2] 선인출의 경우에 대한 성능을 따로 요약해 보았다. 표에서 볼 수 있듯이 DMFC의 성능이 이상적인 선인출의 성능과 유사하다는 사실

표 9 A[1:2] 선인출에 대한 여러 필터링 방식들 간의 CPI 비교

Benchmark	*요구 인출 CPI*	*선인출 CPI : no-filter*	*선인출 CPI : HAFT*	*선인출 CPI : DMFC*	*최적 선인출 CPI*
186.crafty	1.666	1.205	1.523	1.154	1.158
176.gcc	2.077	2.466	2.076	1.863	1.822
164.gzip	1.605	1.901	1.605	1.565	1.576
181.mcf	1.932	2.347	1.919	1.844	1.845
mpeg2decode	1.044	1.025	1.027	1.016	1.013
mpeg2encode	1.155	1.057	1.118	1.037	1.035
197.parser	1.442	1.266	1.408	1.165	1.190
255.vortex	1.704	2.002	1.704	1.620	1.559
175.vpr	1.787	1.574	1.780	1.135	1.467

은 선인출로 인한 역효과, 즉 캐시 오염 및 트래픽 증가 손실이 제안된 필터링 방식에 의해서 성공적으로 제거되었음을 증명해 준다.

요구인출 CPI와 비교할 때에 A[1:2]와 DMFC의 조합은 최소 2.6%(gzip)에서 최대 57%(vpr)의 성능 향상을 보인다. 9개의 벤치마크들에 대해서 평균 18% 성능이 향상되었다. 기존의 선인출 알고리즘으로는 메모리 접근 패턴의 예측이 매우 어려운 것으로 알려져 있는 mcf의 경우에도 성능이 4.8% 향상되었다는 것은 매우 주목할 만한 결과이다.

제안된 필터링 방식의 효율성을 심층적으로 분석하였다. 그림 12는 3가지 필터링 방식에 대한 총 선인출 요청 수를 보여주는데 각각의 값들은 필터링 하지 않은 경우에 대해서 정규화 된 값들이다. DMFC의 총 선인출 요청 수는 필터링 하지 않는 경우에 비해서 현저하게 감소되었으나, 전반적으로 HAFT의 경우보다는 다소 많음을 볼 수 있다.

HAFT의 경우에는 2가지 벤치마크(mpeg2decode 및 mpeg2encode)를 제외하면 대부분의 선인출들이 과도한 필터링 효과로 인하여 거의 완전하게 억제되었음을 볼 수 있다. 예외적인 두 가지 벤치마크의 경우에는, 공간적 국소성이 매우 높아서 필터링 하지 않은 원래의 선인출 정확도가 매우 높기 때문에 HAFT내의 거의 모든 엔트리들이 그림 8에서와 같이 '패스' 상태로 있게 된다. 물론 그림 8의 mpeg2decode의 경우에 모든 픽셀들이 흰색은 아니지만, 그림의 검은색 픽셀들은 그것들이 '필터링' 상태임을 의미하는 것이 아니라, 멀티미디어 벤치마크의 특성 상 주소공간 내의 모든 영역이 계산에 쓰이지 않기 때문에 그러한 픽셀들로는 선인출 요청 자체가 한번도 일어나지 않았기 때문에 검은색으로 표시된 것이다. 여하튼, HAFT는 그러한 벤치마크들에 대해서는 선인출 미스 뿐 아니라 선인출 히트마저도 전혀 감소시킬 수가 없게 된다.

DMFC의 경우의 총 선인출 요청 수 중의 선인출 미스와 선인출 히트의 각각의 비중을 그림 13에 보였다. 그림 3(a)와 비교하였을 때 선인출 히트의 비중이 전체의 반 이하가 될 정도로 현저하게 감소된 것을 알 수 있다. 다양한 벤치마크들에 대해서 40%에서 90%의 선인출 히트 수가 DMFC 방식에 의해서 제거되었음을 알 수 있다.

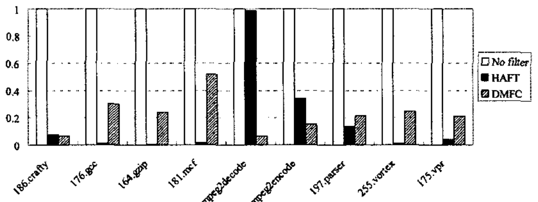


그림 12 3가지 필터링 방식에 대한 총 선인출 수 비교 (A[1:2])

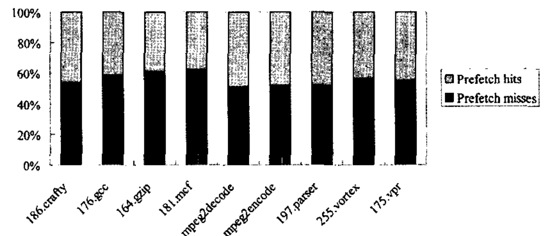


그림 13 DMFC를 적용한 경우의 A[1:2] 선인출 요청의 성분 분석

그림 14는 선인출 미스 수의 각각의 성분들을 분석한 결과를 보여 준다. 대부분의 좋은 선인출 미스 수가 거의 유지되면서 나쁜 선인출 미스 수는 대폭 줄어들었다는 사실에 주목할 필요가 있다. 나쁜 미스 수가 감소하게 된 것은 주로 재발된 나쁜 미스들이 줄어들었기 때문이다. 어떠한 필터도 최초의 나쁜 선인출 데이타가 처음으로 축출되기 전에는 그것이 나쁜 선인출인지 아닌지를 알 수가 없기 때문에 최초의 나쁜 미스 수는 줄일 수가 없다. 재발된 나쁜 미스 수는 일부 벤치마크(gcc,

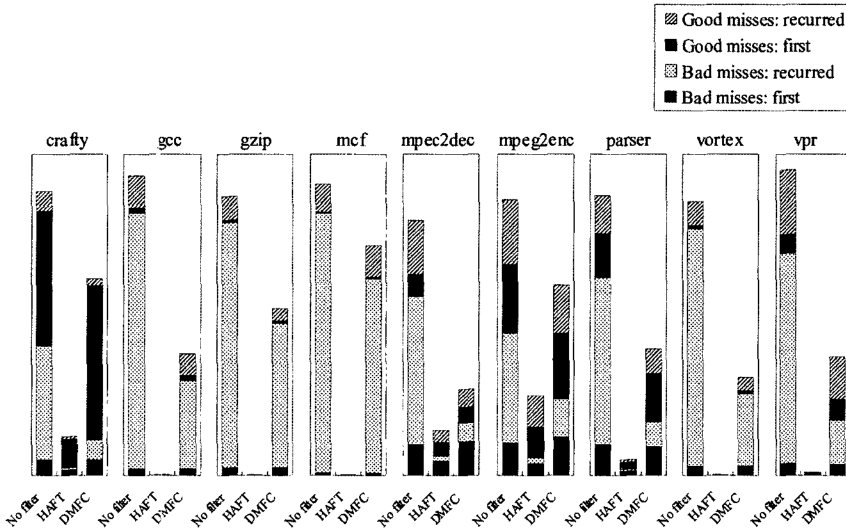


그림 14 A[1:2] 선인출 및 3가지 필터링 방식에 대한 선인출 미스 성분

gzip, mcf 및 vortex)의 경우 그렇게 대폭 감소된 것으로는 보이지 않는데, 그것은 필터링 테이블(필터링 캐시)의 크기가 충분하지 못하기 때문이다.

그림 15는 두 가지의 벤치마크에 대하여 HAFT와 DMFC 각각에 있어서의 필터링 테이블 크기가 필터링 성능에 미치는 영향을 보여준다. Crafty에서는 CPI 및 DMFC의 선인출 미스 수는 1K의 테이블 크기에서도 이미 포화 되어 필터링 테이블 크기를 더 증가시켜도 CPI가 향상되지 않는다. Gcc에서는, DMFC가 최대의 성능을 갖기 위해서는 테이블의 크기가 8K 이상 되어야 함을 알 수 있다. 또한, HAFT의 성능은 필터링 테이블의 크기가 증가함에 따라 CMFC의 성능에 근접해 간다는 사실을 알 수 있다. 이것은 테이블의 크기가 매우 커지게 되면 주소를 해싱 하는 의미가 거의 없어져서 좋은 선인출 주소를 구별하는 능력이 향상되어 알리아싱 효과가 점점 줄어들게 되기 때문이다.

그림 16에서는 다양한 L1 캐시 크기에 대한 CPI를 비교하였다. 8KB보다 작은 캐시 크기에 대해서는, HAFT 필터링 방식이 요구 인출 CPI와 거의 같은 값을 갖는다. 이것은 캐시 크기가 작으면 축출되는 선인출 데이타의 수도 많아져서 HAFT의 거의 모든 엔트리들이 '필터링' 상태가 될 것이기 때문이다. 반대로, 캐시 크기가 128KB보다 크면, 선인출 성능은 필터링 방식과 무관하게 되는데, 그것은 첫째로, 과도한 선인출에 따른 캐시 오염이 무시할 만 하게 되고, 둘째로, 큰 캐시 크기로 인하여 요구 미스 수 또한 현저히 줄어들게 되어 선인출로 인하여 증가된 메모리 트래픽으로 인하여 증가된 요구 미스의 지연시간이 시스템 성능에 거의 영향

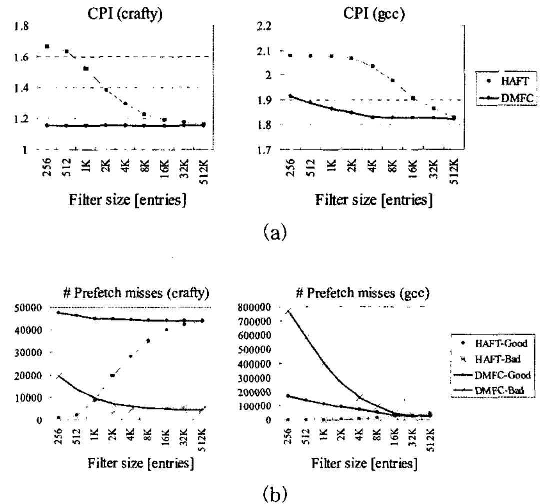


그림 15 필터링 테이블 크기에 따른 (a) CPI, (b) 선인출 미스 수

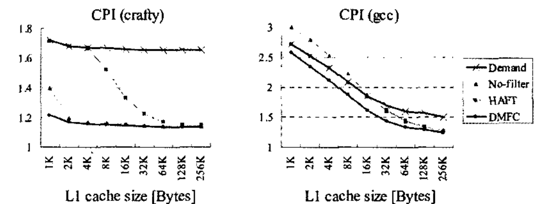


그림 16 L1 캐시 크기에 따른 각 방식의 CPI 비교

을 주지 않기 때문이다. 중간 이하의 L1 캐시 크기에 대해서는 DMFC가 HAFT 또는 필터링 하지 않는 경

우에 비해서 반 정도의 L1 캐시 크기로도 오히려 더 나은 성능을 보인다.

6. 제안된 구조의 한계

제안된 선인출 필터링 방식은 다음과 같은 이론적인 한계를 지닌다.

1. **엔트리 간의 충돌.** 직접 사상 필터링 캐시는 통상의 캐시의 충돌 미스와 같은 문제를 겪는다. 이 문제는 associativity를 도입하거나 테이블의 크기를 증가시킴으로써 최소화 시킬 수 있으나, 이 경우에 테이블의 복잡도 또는 오버헤드가 증가된다.
2. **필터링 리스트의 길이 부족.** 유한한 크기의 테이블은 동상의 캐시의 용량 미스와 같은 문제를 겪는다. 이 문제도 마찬가지로 테이블의 크기를 증가시킴으로써 최소화 시킬 수 있으나, 오버헤드가 증가된다.
3. **주소에 대한 분해능이 L1 캐시의 블록 크기에 의해 제한됨.** 서로 다른 선인출 주소들이 같은 블록 주소를 가질 수 있다. 이것은 테이블에 블록 주소가 아닌 전체 주소를 저장함으로써 해결될 수 있으나, 이를 위해서는 L1 캐시에도 전체 주소가 저장되어야 하므로 캐시 면적이 크게 증가한다.
4. **다른 명령어에 의해 개시되었으나 동일한 주소를 갖는 선인출 요청**들은 데이타 주소만으로는 구별될 수 없다. 이것은 데이타 주소뿐 아니라 프로그램 카운터의 전체 또는 일부 값도 L1 캐시 및 필터링 테이블에 저장함으로써 해결될 수도 있겠지만, 역시 L1 캐시 오버헤드가 크게 증가한다.
5. **선인출 거리.** 엄밀하게 말하면, 일찍 또는 늦게 선인출 되었기 때문에 사용되지 못하고 축출된 선인출 데이타도 나쁜 선인출이다. 그러나 현재의 알고리즘은 요구 미스가 요청될 때에 그러한 선인출들도 또한 필터링으로부터 구제하도록 되어 있다. 어떠한 선인출 요청이 제 때에 발생되었는지 여부를 판단할 수 있는 방법이 없다.

위에서 나열한 요소들로 인하여, 제안된 필터의 필터링 정확도는 제한적이 된다. 첫번째 및 두 번째 요인은 테이블 크기를 증가시킴으로써 완화될 수 있다. 그림 17은 필터링 테이블의 크기에 따라서 좋은 선인출과 나쁜 선인출을 명확하게 구별할 수 있는 선인출 판별도를 보여 준다.

시뮬레이션이 수행되는 동안 축출되는 데이타들의 고유 주소들의 수를 산출하였다. 동일한 주소에 대하여 한번은 좋은 선인출로 또 다른 때에는 나쁜 선인출로 기록되는 주소들은 혼성(mixed)으로 분류하였다. 시뮬레이션 내내 좋은 선인출로만 기록된 주소들은 엄격히 좋음(strictly good)으로, 또한 계속해서 나쁜 선인출로만

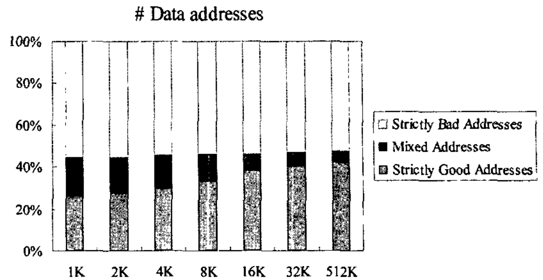


그림 17 필터 테이블 크기에 따른 선인출 판별도 성능

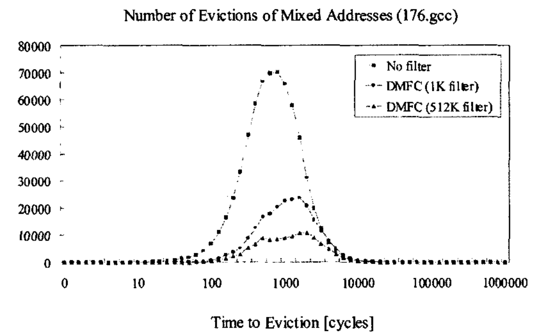


그림 18 혼성 주소를 가지는 선인출 데이타들의 생존 시간의 분포도

기록된 주소들은 엄격히 나쁨(strictly good)으로 분류하였다. 그림 17은 이와 같은 3가지 분류에 따른 주소들의 수의 비중을 보여 준다. 혼성 주소들의 수는 테이블의 크기가 증가되면서 점점 줄어드는 것을 알 수 있다. 이것은 테이블의 충돌 또는 용량 미스가 감소하여 uncertainty가 줄어들었기 때문이다. 그와 더불어 높아진 필터링 정확도로 인한 캐시 미스의 감소로 '너무 일찍 선인출된' 데이타들의 축출이 감소하였을 것이기 때문이기도 하다. 혼성 주소의 수는 32K개의 엔트리 수(72Kbyte) 이상의 크기를 갖는 테이블에 대해서는 거의 포화 된다.

그림 18은 혼성 주소를 갖는 모든 축출된 데이타의 생존 시간을 보여 준다. 이번에는 동일 주소에 대한 여러 회수의 축출을 모두 반복하여 산출하였다. 512K의 필터 테이블 크기에 대해서는 두개의 봉우리가 관측된다.

512K 정도의 큰 테이블 크기에서는 충돌 또는 용량 미스 문제가 거의 없을 것이기 때문에, 이들 두 봉우리는 각각 나쁜 선인출 데이타들의 축출과 좋은 선인출 데이타들의 축출에 대응된다. 바로 이들은 위에서 언급한 5가지 제한적 요인 중에서 큰 테이블 크기에 의해 제거되고 남은 3가지의 요인으로 인하여 필터가 구별해내지 못한 선인출들에 해당된다.

7. 결 론

본 논문은 과도하게 공격적인 선인출과 효율적인 선인출 필터링을 결합함으로써 임의의 응용 프로그램에 대한 선인출 커버리지 및 정확도를 높임으로써 시스템 성능을 향상시킬 수 있다는 가정을 기반으로 한다. 이러한 가정을 검증하기 위하여 과도한 선인출로 인한 시스템 성능에의 역효과를 정량적으로 측정하기 위한 가상 시스템의 CPI를 도입하여 선인출 역효과를 측정하였다.

또한 비경쟁 L1 캐시 선인출 구조를 사용함으로써 선인출로 인한 시스템 성능의 향상을 극대화 시킬 수 있도록 하였다. 비경쟁 L1 캐시 선인출 구조 하에서의 과도한 선인출로 인한 손실 성분을 정밀하게 분석하였으며, 이 결과를 바탕으로 선인출 손실을 최소화할 수 있는 필터의 구조 및 알고리즘을 도출하였다.

비경쟁 L1 캐시 선인출 구조 하에서의 과도하게 공격적인 하드웨어 선인출기와 결합되어 선인출 효율 및 시스템 성능을 극대화 시킬 수 있도록 직접 사상 필터링 캐시(DMFC)를 이용한 선인출 필터링 구조를 제안하였다. 또한, 기존의 필터링 방식의 한계를 분석함으로써 제안된 필터의 당위성을 입증하였다. 자체적으로 개발한 정교한 타이밍 기반 시뮬레이터를 이용하여 제안된 구조의 성능을 분석하였다. 실험 결과들은 제안된 구조 및 선인출 필터링 방식은 과도한 선인출 알고리즘에 대해서 전체 시스템의 성능을 향상시킴을 보여 준다. 또한, 제안된 직접 사상 필터링 캐시(DMFC) 방식은 선인출로 인한 전체 시스템 성능을 이론적인 최대값에 근접하게 할 수 있을 정도로 매우 우수한 필터링 성능을 가짐을 보여주었다.

DMFC 필터를 A[1:2] 하드웨어 선인출기와 결합한 경우에 벤치마크에 따라 최소 2.6%에서 최대 57%의 성능 향상 및 평균적으로는 18%의 이득을 얻을 수 있었다. 제안된 구조에서의 선인출 결과들을 분석함으로써 선인출 필터의 효율성을 검토하였다. 추가적으로, DMFC 방식의 이론적인 한계에 대한 분석을 수행하고 논의하였으며 이러한 점들을 극복하기 위해서는 더욱 개선된 필터 구조에 대한 연구가 필요할 것으로 보인다.

참 고 문 헌

[1] D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching," *in Proc. Fourth Int. Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 40-52, Apr. 1991.

[2] C.-K. Luk and T. Mowry, "Compiler Based Prefetching for Recursive Data Structures," *in Proc. Seventh Int. Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 222-233, Oct. 1996.

[3] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, Vol. 14, No. 3, pp. 473-530, Sep. 1982.

[4] J. D. Gindele, "Buffer Block Prefetching Method," *IBM Technical Disclosure Bull.*, vol. 20, no. 2, pp. 696-697, July 1977.

[5] R. Cucchiara, M. Piccardi and A. Prati, "Hardware Prefetching Technique for Cache Memories in Multimedia Applications," *in Proc. IEEE Intl. Workshop on Computer Architectures for Machine Perception (CAMP)*, 2000

[6] N. P. Jouppi, "Improving Directed-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *in Proc. of the 17th Annual International Symposium on Computer Architecture*, pp. 364-373, May 1990.

[7] J. L. Baer and T.-F. Chen, "An Effective On-chip Preloading Scheme to Reduce Data Access Penalty," *in Proc. of Supercomputing '91*, pp. 176-186, Nov. 1991.

[8] T.-F. 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.

[9] J. Pomerene, T. Puzak, R. Rechtschaffen and F. Sparacio, "Prefetching System for a Cache Having a Second Directory for Sequentially Accessed Blocks," US Patent 4,807,110, Feb. 1989.

[10] M. Charney and T. Puzak, "Prefetching and Memory System Behavior of the SPEC95 Benchmark Suite," *IBM J. Research and Development*, vol. 41, no. 3, pp. 265-286, May 1997.

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

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

[13] 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.

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

[15] V. Srinivasan, G. Tyson and E. Davidson, "A Static Filter for Reducing Prefetch Traffic," Technical Report CSE-TR-400-99, University of Michigan, 1999.

[16] V. Srinivasan, E. S. Davidson and G. S. Tyson, "A Prefetch Taxonomy," *IEEE Trans. Computers*,

Vol. 53, No. 2, pp. 126-140, Feb. 2004.

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

[18] O. Mutlu, H. Kim, D. N. Armstrong and Y. N. Patt, "Cache Filtering Techniques to Reduce the Negative Impact of Useless Speculative Memory References on Processor Performance," *in Proc. 16th Symposium on Computer Architecture and High Performance Computing* (SBAC-PAD'04), pp. 2-9, 2004.

[19] P. G. Emma, A. Hartstein, T. R. Puzak and V. Srinivasan, "Exploring the Limits of Prefetching," *IBM J. Research and Development*, Vol. 49, No. 2-3, pp. 127-144, Jan. 2005

[20] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *in Proc. ACM SIGPLAN 94*, pp. 196-205, 1994.

[21] Y. Ruan, V. S. Pai, E. Nahum and J. M. Tracey, "Evaluating the Impact of Simultaneous Multithreading on Network Servers using Real Hardware," *in Proc. ACM Int. Conf. on Measurement and Modeling of Computer Systems*, pp. 315-326, 2005.

[22] 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.

[23] 전영숙, 문현주, 김석일, 전중남, "단속적 불규칙 주소 간격을 갖는 멀티미디어 데이타를 위한 하드웨어 캐시 선인출 방법", 정보과학회논문지, 제31권, 제11호, pp. 658-672, 2004.

전 영 숙

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