프로세서의 재사용 정보를 이용하는 개선된 고성능 희생 캐쉬

(Advanced Victim Cache with Processor Reuse Information)

곽 종 욱 † 이 현 배 †† 장 성 태 ††† 전 주 식 ††††

(Jong Wook Kwak) (Hyunbae Lee) (Seong Tae Jhang) (Chu Shik Jhon)

요 약 최근의 단일 혹은 다중 프로세서 시스템은 일반적으로 계층적 메모리를 사용한다. 이는 프로세서의 클럭 속도와 메모리로의 데이타 접근 시간의 증가로 인한 시스템 성능 저하를 막기 위한 노력 중 하나이다. 특히 프로세서와의 속도 차이를 줄이기 위해 사용되는 캐쉬는 이단계에서 삼단계에 이르는 다양한 형태의 계층을 포함하는 메모리 시스템으로 구성된다. 이 중에서도 특히 상위 캐쉬는 프로세서와 직접 인터페이스가 이루어지기 때문에, 해당 캐쉬의 적중률은 전체 시스템의 성능을 결정하는 중요한 요소가 된다. 이러한 상위 캐쉬의 하나로써, 희생 캐쉬는 일차 캐쉬의 충돌 미스(Conflict Miss)를 줄이기 위해 추가된 모듈이다. 이는 프로세서 입장에서 보면 절차상 일차 캐쉬와 동등한 관계에서 접근이 이루어진다.

본 논문에서는 이러한 상위 캐쉬의 관리 정책 중, 기존의 일차 캐쉬와 희생 캐쉬의 구현시 배제되어 왔던 프로세서의 재사용 정보를 이용하는 캐쉬 라인의 효율적인 관리 정책을 제안하고자 한다. 이 기법은 프로세서의 데이타 사용 빈도에 의한 캐쉬 교체 정책으로, 프로세서에 의해 특정 데이타가 얼마나 자주 접근되었는가에 따라, 사용 빈도수가 높은 데이타에 대해 캐쉬에 위치시키는 시간을 연장시키는 기법이다. 본 논문에서는 제안된 메모리 시스템의 성능을 평가하기 위해, 이를 프로그램 기반 시뮬레이터인 Augmint를 통해 모델링한 후, 시뮬레이션을 수행한다. 그리고 이를 기존의 단순한 희생 캐쉬 교체 정책과 비교하여 성능상의 차이점을 비교 분석 한다. 실험 결과 제안된 L1VMR 기법은 최대 6.7%, 평균 0.5%의 성능 향상을 보였다.

키워드 : 계층적 메모리 시스템, 일차 캐쉬, 희생 캐쉬, 캐쉬 교체 정책, 재사용 정보

Abstract Recently, a single or multi processor system uses the hierarchical memory structure to reduce the time gap between processor clock rate and memory access time. A cache memory system includes especially two or three levels of caches to reduce this time gap. Moreover, one of the most important things in the hierarchical memory system is the hit rate in level 1 cache, because level 1 cache interfaces directly with the processor. Therefore, the high hit rate in level 1 cache is critical for system performance. A victim cache, another high level cache, is also important to assist level 1 cache by reducing the conflict miss in high level cache.

In this paper, we propose the advanced high level cache management scheme based on the processor reuse information. This technique is a kind of cache replacement policy which uses the frequency of processor's memory accesses and makes the higher frequency address of the cache location reside longer in cache than the lower one. With this scheme, we simulate our policy using Augmint, the event-driven simulator, and analyze the simulation results. The simulation results show that the modified processor reuse information scheme(L1VMR) outperforms the level 1 with the simple victim cache(L1V), 6.7% in maximum and 0.5% in average, and performance benefits become larger as the number of processors increases.

Key words : Hierarchical Memory System, Level 1 Cache, Victim Cache, Cache Replacement Policy, Processor Reuse Information.

† 학생회원 : 서울대학교 전기컴퓨터공학부
leoniss@panda.snu.ac.kr
†† 비 회 원 : LG전자기술원 모바일 멀티미디어 연구소
iapetus@panda.snu.ac.kr
††† 종신회원 : 수원대학교 컴퓨터학과 교수
stjhang@suwon.ac.kr
†††† 종신회원 : 서울대학교 전기.컴퓨터공학부 교수
csjhon@riact.snu.ac.kr
논문접수 : 2003년 11월 19일
심사완료 : 2004년 8월 16일

1. 서 론

최근 프로세서와 메모리 시스템간의 속도 차이는 급격하게 커져왔다. 이러한 상황은 메모리 접근 시간에 비하여 프로세서 사이클 시간이 매우 빠른 속도로 줄어들었으며, 파이프라이닝과 슈퍼스칼라(Superscalar) 프로세싱과 같은 디자인 기법이 명령어 실행에 소요되는 사이클(Cycle Per Instruction : CPI)의 평균 시간을 급격히 줄여왔기 때문에 발생하였다. 이러한 프로세서의 속도 향상에 비하여 지난 20여 년간의 메모리 접근 시간의 감소는 상대적으로 미미하다 할 수 있다[1].

이로 인한 프로세서의 사이클 시간과 메모리 접근 시간과의 차이를 극복하기 위하여 계층 메모리 시스템이 제안되었으며, 이러한 다중 캐쉬 구조의 사용으로 인하여 데이타 접근 시간을 감소시킬 수 있었다. 그러나 다중 계층 구조에서도 블록들에 대한 할당과 교체는 발생하며, 이 경우에도 데이타 접근 시간이 증가하는 문제는 남아있다. 이러한 평균 메모리 접근 시간을 줄이기 위해서 일차 캐쉬의 자원을 좀 더 정확하게 관리하고 유지되도록 하는 연구들이 있었다. 그 대표적인 예로 희생 캐쉬(Victim Cache)[2]를 이용한 참조 성공률을 높이는 방안이 제시되었다.

희생 캐쉬는 일차 캐쉬에 존재하던 라인이 프로세서의 요청에 의해서 교체 되었지만, 바로 다음 혹은 가까운 시간 내에 재참조 되는 경우, 상위 계층에서 하위 계층 캐쉬로의 참조로 인한 접근 시간 증가를 피하는 방법이다. 하지만 이러한 희생 캐쉬는 짧은 시간 내 재참조 되는 데이타에 대해서는 유효한 구조이지만, 전체 프로그램의 입장에서 보면 재사용 되는 데이타가 무시되는 경향이 있다. 일반적으로 프로그램 전체에 걸쳐서 데이타의 재사용 빈도는 응용 프로그램에 따라 다르지만, 순환구조로 되어 있는 프로그램의 경우 빈번히 요청되는 데이타 라인이 존재한다.

그림 1은 SPEC 벤치마크 프로그램 중에서의 0.26 compress의 주 순환문(Main Loop Code)이다. 특히 아래의 프로그램은 전체 수행시간의 90% 이상을 순환문 내부에서 차지한다. 순환문 내부는 해쉬 테이블로 이루어져 있는데, 아래의 코드에서 htab과 codetab이 그것이다. 특히 실제 프로그램 상에서는 비교적 큰 크기의 해쉬 테이블이 사용되고 또한 해쉬 테이블 접근시 시공간적인 국부성이 아주 낮으므로, 실제 상위 캐쉬에서의 데이타 재사용률은 미미하다고 할 수 있다[3].

표 1은 위의 SPEC 벤치마크 *026.compress* 프로그램의 주 순환문에 존재하는 해쉬 테이블의 동적 접근시에 나타나는 일차 캐쉬의 접근 실패율, 데이타의 재사용 빈도, 그리고 수행 횟수를 분석한 것이다. 단순 캐쉬의 시

```
/* Compress Main Loop Code */
while((c=getchar()) != EOF) {
    in_count++;
    fcode = (long) ((long) c << maxbits) + ent);
    i = ((c << hshift) ^ ent);
    if(htabof(i) == fcode) {          /* htab -1 */
        ent = codetabof(i);           /* codetab -1 */
        continue;
    }   else if ((long htabof(i) < 0 ) goto nomatch;
    disp = hsize_reg - i;
    if (i==0) disp = 1;
probe :
    if((i-=disp) < 0) i+= hsize_reg
    if(htabof(i) == fcode) {      /* htab -2 */
        ent = codetabof(i);       /* codetab -2 */
        continue;
    }
    if((long) htabof(i) > 0) goto probe;
nomatch:
    output((code_int) ent);
    out_count++;
    ent = c;
    if(free_ent < maxmaxcode) {
        codetabof(i) = free_ent++;
        htabof(i) = fcode;
    }
    else if((count_int) in_count >= checkpoint
    && block_compress)
    cl_block();
}
```

그림 1 *0.26 compress* Main Loop Code

표 1 해쉬 테이블의 적재(Load) 접근

Hash Table	Dynamic Execution Count	Miss Ratio	Reuse Ratio
htab-1	999999	78.9%	29.2%
codetab-1	566776	70.8%	30.0%
htab-2	1803911	91.4%	15.6%
codetab-2	182336	89.1%	11.5%

뮬레이션을 통해서 프로세서로부터 제기된 각각의 요청에 대해 일차 캐쉬의 적중 횟수를 카운트하고, 또한 캐쉬의 재사용 빈도를 추적하였다. 표를 통해 알 수 있듯이, 해쉬 테이블의 적재 접근이 상당한 참조 실패율을 야기하며, 낮은 재사용 빈도를 보인다. 또한 상대적으로 재사용 빈도가 낮을수록 해당 해쉬 테이블에 대한 적재 접근의 참조 실패율이 높다는 것을 알 수 있다.

한편, 다중 계층 메모리 시스템에서 일차 캐쉬를 비롯한 상위 캐쉬들은 나름대로의 특징에 따라 여러 가지 형태로 구현된다. 가령 시스템 온 칩(SoC) 구조로 주로 사용되는 ARM 계열 프로세서의 상위 캐쉬 구조[4]는

다음과 같다. ARM710T의 상위 캐쉬는 통합 캐쉬(Unified Cache)로 4-방향 연관 사상(4-way Set-Associativity)을 가지며, StrongARM SA-110이나 ARM920T의 경우는 32-방향 혹은 64-방향과 같은 극단적인 형태의 연관도를 가진다. 이는 캐쉬 메모리의 여러 설계 패러다임 중에서 참조 실패율을 최소로 줄여 그로 인한 참조 실패 손실(Miss Penalty)를 줄이고자 함에 그 목적이 있다. 하지만 적중 성공률(Hit Rate)보다, 조금이라도 더 빠른 적중 시간(Hit Time)을 목적으로 하는 상위 캐쉬라면 주로 2-방향 연관 사상이나 직접 사상(Direct-Map) 정책에 의한 캐쉬 라인의 관리가 이루어진다. 이러한 정책은 빠른 접근 속도를 필요로 하는 상위 계층의 캐쉬에서 갖추어야 할 조건이나, 상대적으로 높은 집합 연관(Set-Associative) 구조를 가진 캐쉬 정책에 비하여 높은 참조 실패율을 가진다는 단점이 있다. 그리고 이러한 참조 실패는 교체 및 할당 시에 해당 라인간의 충돌에 의하여 발생하게 된다. 따라서 이와 같은 실패율을 낮추기 위하여, 일차 캐쉬가 직접 사상 방식 혹은 낮은 연관도를 가지는 형태로 구성되는 경우, 일반적으로 희생 캐쉬라는 작은 엔트리 수를 가지는 버퍼를 일차 캐쉬에 분리시켜 병행하게 설치하여, 참조 실패율을 낮추는 효과를 가져왔다.

본 논문에서는 이와 같은 희생 캐쉬의 성능을 높이기 위해, 프로세서에 의한 데이타의 접근 정보를 유지하여 앞으로 해당 캐쉬 라인이 재사용 되어질 것인가 혹은 재사용 되어지지 않을 것인가에 대한 예측을 한다. 그리하여 예측에 의하여 재사용 되어질 라인에 대하여서는 일차 캐쉬와 희생 캐쉬간의 데이타 교환이 좀 더 유효한 방향으로 이루어지게 하는 교체 정책을 제안한다.

이하 본 논문의 구성은 다음과 같다. 2장에서는 메모리 시스템의 성능 향상을 위한 다양한 캐쉬 관리 기법 등의 관련 연구를 소개하고, 3장에서는 프로세서의 재사용 정보를 이용한 희생 캐쉬의 관리 정책을 설명한다. 그리고 4장에서 시뮬레이션을 통한 성능 평가를 수행하고, 끝으로 5장에서 결론을 내린다.

2. 관련 연구 및 배경 지식

2.1 계층 메모리 구조

시스템에서 메모리 구조는 일반적으로 계층 메모리 시스템(Hierarchical Memory System)으로 구성되어 있다. 또한 이와 같은 계층 구조에서 일차 캐쉬(L1 Cache)와 이차 캐쉬(L2 Cache)는 일반적으로 프로세서 칩 내에 위치하여(On-Chip Cache) 빠른 데이타 접근에 유효하도록 구성되어졌다.

2.1.1 메모리 시스템 구성

전술한 바와 같이, 시스템에서의 메모리 구조는 효율성을 높이기 위하여 계층 구조로 이루어져 있다. 프로세서의 읽기 혹은 쓰기 참조를 주 메모리에서 수행할 경우 소요되는 데이타 접근 시간은 매우 커지므로, 속도가 빠르고 작은 크기의 메모리를 상위에 구현하고 상위 계층보다 느린 메모리를 하위 계층에 두어 효율적인 데이타 참조를 할 수 있게 하였다. 일반적인 메모리 계층 구조의 구성은 그림 2와 같다.

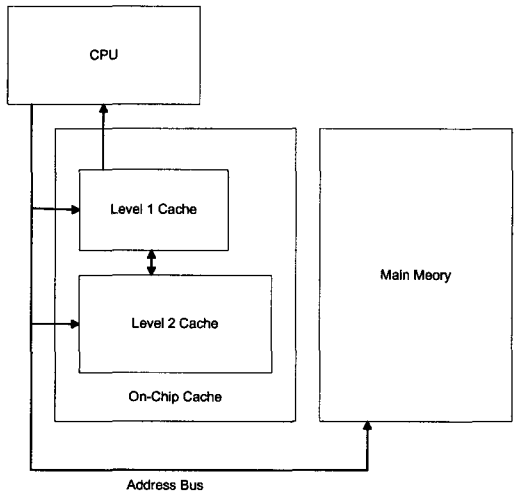


그림 2 일반적인 계층 메모리 구조

이처럼 빠른 속도의 작은 메모리는 국부성의 원리(Principle of Locality)에 의하여 프로세서에서 요구되는 데이타의 참조율을 높여, 높은 값의 적중률(Hit Rate)을 보인다. 이렇게 접근의 국부성을 활용한 구조를 캐쉬(Cache)라 부른다. 현재 대부분의 상위 캐쉬들은 프로세서와 같은 칩 상에 구현된다. 또한 빨라진 프로세서와 상대적으로 접근하는 데 긴 시간을 요구하는 메모리 사이의 성능 차이를 줄이기 위하여 고성능의 프로세서들은 두개 혹은 그 이상의 계층을 이룬 캐쉬를 사용한다. 이렇게 이차 혹은 삼차 캐쉬가 존재함으로서, 일차 캐쉬에서 비록 적중 실패가 일어난다 하더라도 주 메모리가 아닌 이차 혹은 삼차 캐쉬의 접근 시간만이 소요되므로 프로세서의 데이타 접근 시간을 줄일 수 있다. 단 최하위 레벨 캐쉬에도 데이타가 존재하지 않는다면 주 메모리로 접근해야 하는데, 이 경우 더 큰 실패 손실(Miss Penalty)이 발생한다. 그러나 실패 손실을 감안하더라도 두 개 이상의 계층으로 구성되어진 캐쉬 시스템은 데이타 접근 성공률이 높아짐에 따라 총 데이타 접근 시간이 향상되는 결과를 보인다.

또한 상대적으로 일차 캐쉬는 접근시간이 이차 캐쉬에 비하여 빨라야 하므로 그 구조를 직접 사상(Direct-Map)이나 낮은 연관도의 캐쉬로 설계하는 반면, 이차

혹은 그 이상의 하위 캐쉬는 접근 속도보다는 더 높은 적중률을 보이기 위하여 높은 집합 연관 캐쉬 구조로 설계한다. 직접 사상과 집합 연관에 의해 데이타가 캐쉬에 할당되는 모습을 그림 3에서 볼 수 있다.

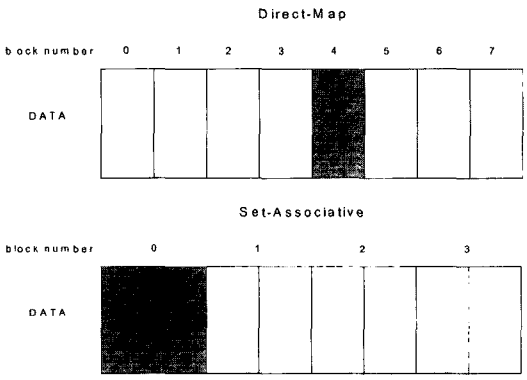


그림 3 직접 사상 및 연관 사상

2.1.2 캐쉬의 적중률을 높이기 위한 설계

빠른 데이타 접근을 위하여 일차 캐쉬를 직접 사상 혹은 낮은 연관도의 캐쉬로 설계할 경우, 캐쉬의 접근 시간(Hit Time)은 단축되나, 접근 속도에 비하여 상대적으로 적중 실패율(Miss Rate)이 높다는 단점이 있다. 이러한 적중 실패율을 낮추기 위한 기존의 방법으로, 블록을 크게 하는 경우(Larger Block Size)를 비롯하여, 높은 연관을 가지게 하는 방법(Higher Associativity), 명령어와 데이타의 하드웨어 선점 기법(Hardware Prefetching of Instruction and Data), 컴파일러에 의한 선점 기법(*Compiler-Controlled Prefetching*) 등이 제시되어 왔다. 이와 관련하여 희생 캐쉬 역시 직접 사상 혹은 낮은 연관도의 캐쉬 시스템에서 적중률을 높이기 위해 널리 사용되는 기법 중에 하나이다. 이러한 희생 캐쉬가 계층 메모리 구조에 추가된 형태를 그림 4에서 볼 수 있다.

희생 캐쉬는 일차 캐쉬에서 적중 실패가 발생했을 때, 캐쉬 라인들 중에서 선택되는 희생 라인을 하위 계층 캐쉬로 바로 옮기지 않고, 같은 계층에 있는 작은 버퍼, 즉 희생 캐쉬에 옮겨 놓는데 사용된다. 따라서 희생 캐쉬는 오로지 일차 캐쉬에서 교체가 발생하여 선택 되어진 희생 라인들만이 할당되는데, 이러한 정책을 취하는 이유는 교체되어서 다음 계층으로 이동되어질 라인이 다시 요구되는 경우가 빈번하기 때문이다. 희생 캐쉬 내에 다시 요구되는 라인이 존재하게 된다면 불필요한 접근 시간의 낭비 없이 바로 데이타를 획득 할 수 있게 된다.

Norman P. Jouppi는 하나에서 다섯 개 정도의 엔트리를 가지는 희생 캐쉬가 작은 크기의 직접 사상 캐쉬에서 발생하는 충돌 실패를 큰 수치로 감소함을 보여줬다. 4KB 직접 사상 캐쉬에서 4개의 엔트리를 가진 희생 캐쉬는 충돌 실패를 응용에 따라 20%에서 95%까지 제거함을 보였다[5].

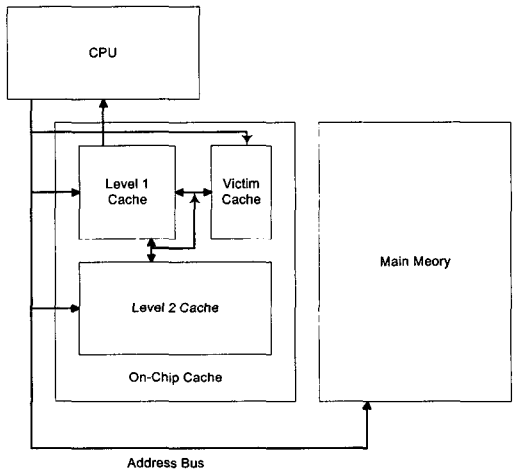


그림 4 희생 캐쉬를 추가한 메모리 시스템

2.2 재사용성을 이용한 성능 향상의 연구

희생 캐쉬 이외에도 데이타의 재사용성을 통한 성능 향상 연구는 많이 이루어졌다. 우선 캐쉬를 물리적으로 자주 참조되는 영역과 그렇지 않은 영역으로 구분하여 캐쉬 할당 자체를 분리하여 관리하는 NTS Model[6,7]이 제안되었다. 이 기법은 논문에서 제시된 국부 평면(Locality Plane)상에서 시간성에 따라 접근 영역을 각각 STS(Split Temporal/Spatial), SS/NS(Split Spatil/Non Spatial), PF(Prefetch/No Prefetch), 그리고 NTS(Non Temporal Streaming) 영역으로 구분하여 데이타를 독립적으로 할당하는 기법이다. 또한 이와 유사하게, 접근 횟수에 따라 캐쉬 블록을 자주 참조 되는 영역과 그렇지 않은 영역에 대하여 동적으로 구분지어서 사용하도록 하는 Frequent Value Locality Model[8]이 있었다. 해당 연구에서는 다른 국부성 모델과는 별도로 Frequent Value Locality 현상을 제시하고 있으며, 이에 대한 활용 기법을 소개하고 있다. 그리고 기존의 일반적인 국부성 모델인 데이타의 공간적 국부성과 시간적 국부성을 고려한 각 국부성에 따른 독립된 영역을 할당하여 각각의 캐쉬에 국부성을 높이도록 관리하는 Dual Cache[9,10] 기법도 제안되었다. 이는 캐쉬를 크게 시간적 캐쉬(Temporal Cache)와 공간적 캐쉬(Spatial Cache)로 구분하여 관리하는 기법으로, 이를 국부 예측 테이블(Locality Prediction Table)을 통해 공간의 사용 여부를 관리하는 기법이다. 한편 이와는 다르게 국

부성에 따라 구조를 나누어 관리하지 않고 각 국부성을 예측하여 하나의 캐쉬에서 할당과 교체가 일어나게 수정한 Selective Cache[11,12]와 관련된 연구도 존재한다. 이는 프로세서로부터 요청된 데이타의 추후 재사용 가능성 여부를 예측하여 필요에 따라서 캐쉬 상에서의 우회(Bypass)를 지원하는 구조이다. 이상에서 제시된 구조들은 프로세서의 메모리 접근에 있어서 나타나는 국부성과 재사용성을 기반으로 하는 기법들로서 각기 나름대로의 특징을 갖추고 있다. 하지만 이들 구조는 모두 추가적인 하드웨어 비용을 요구한다. 그 대신 시간적 혹은 공간적으로 재사용되는 데이타를 일반적인 데이타에 비하여 좀더 긴 캐쉬 할당 시간을 가지게 하여, 참조 실패율을 줄이고 적중률을 높이는 것이 목적이다. 이러한 연구의 결과는 측정을 위한 응용 프로그램의 특성에 따라 더 낳은 성능을 보여주거나, 최소한 단순 캐쉬 구조보다는 더 좋은 성능을 보여 주었다.

3. 재사용 정보를 이용한 캐쉬 교체 정책

3.1 기존의 일차 캐쉬와 희생 캐쉬간의 교체 정책

기존의 계층 메모리 구조에서 일차 캐쉬에 희생 캐쉬가 포함되어 있는 경우, 희생 캐쉬로의 데이타 할당 및 일차 캐쉬와 희생 캐쉬간의 교체 정책은 다음과 같다.

캐쉬 라인에 대한 읽기 요청 혹은 쓰기 요청이 프로세서에 의해 발생할 경우, 주소 라인의 인덱스 값에 따라 캐쉬의 해당 엔트리를 찾게 된다. 그 후 엔트리에 존재하는 캐쉬 라인의 태그 값과 어드레스 라인의 태그 값을 비교하고 유효 비트를 확인하여 현재 존재하는 데이타가 올바른지 그리고 유효한 값인지를 판단한다. 태그가 일치하지 않고 실패할 경우 하위 계층 캐쉬로 데이타 요청을 하게 된다. 희생 캐쉬를 이용하는 경우는 위와 같은 요청 데이타 라인을 일차 캐쉬에서 확인하는 동안, 이와 병행하여 희생 캐쉬를 검색하게 된다. 희생 캐쉬에 요청하는 데이타가 존재하는 경우 희생 캐쉬가 프로세서로 직접 데이타를 전송하게 되며 일차 캐쉬의 해당 엔트리에 존재하는 라인과 적중된 희생 캐쉬의 엔트리 라인을 교환(Swapping)하게 된다. 또한 일차 캐쉬와 희생 캐쉬 모두 요청하는 데이타가 존재하지 않는 경우, 요청은 하위 계층 캐쉬에게 보내지게 되며, 일차 캐쉬에서는 하위 캐쉬에서 얻어진 데이타를 할당하게 된다. 이때 충돌로 인한 교체가 발생하는 일차 캐쉬 라인은 하위 계층이 아닌 희생 캐쉬로 할당하게 된다. 이 경우 희생 캐쉬는 완전 연관(Fully Associative)에 의한 할당이 일어나게 되며, 어떠한 엔트리에도 위치 할 수 있게 된다. 그러나 일반적으로 희생 캐쉬에서도 나름대로의 효율적인 교체 정책을 위하여 선입선출(FIFO), LRU 방법들이 많이 쓰이고 있다.

그림 5의 상단은 일차 캐쉬와 희생 캐쉬 모두에게서 참조 실패가 발생하여 하위 캐쉬에서 적절한 데이타를 읽어 오는 것과 동시에 일차 캐쉬의 충돌 라인이 희생 캐쉬에 할당되는 모습이며, 하단은 희생 캐쉬에서 적중하여 일차 캐쉬와 희생 캐쉬 간에 캐쉬라인의 교체가 일어나는 모습을 보여주고 있다.

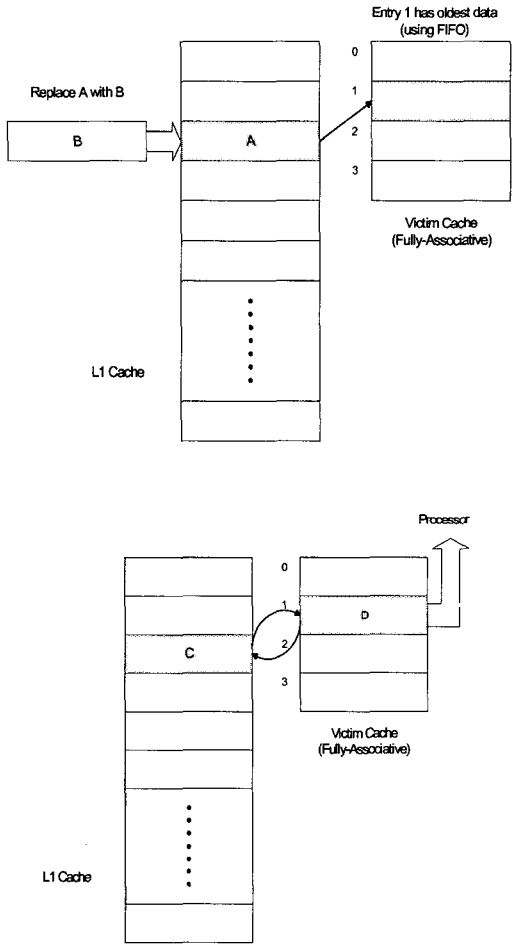


그림 5 일차 캐쉬와 희생 캐쉬의 교체

3.2 재사용 정보를 적용한 교체 정책

지금까지 소개된 일차 캐쉬는 희생 캐쉬와 함께 사용되어 질 경우, 참조 실패율을 많이 줄일 수 있어 매우 유효한 구조라고 볼 수 있다. 그러나 이러한 구조는 일차 캐쉬에서 교체되어 희생 캐쉬에 할당된 데이타가 단기간에 프로세서에 의해서 재요청되는 경우에 적중률을 높일 수 있으나, 재사용 되지 않는, 즉 프로세서에 의해 추후 요청되지 않는 데이타 역시 희생 캐쉬에 할당 될 수 있는 문제점이 있다. 희생 캐쉬에 이와 같은 불필요

한 데이타 할당을 피하기 위하여, 이전의 프로세서에 의한 데이타 요구 정보들을 이용하여 미래에 재사용 되어질 정보를 얻어내어 상위 두 캐쉬 -일차 캐쉬와 희생 캐쉬- 사이의 교체, 혹은 하위 계층 캐쉬 -일차 캐쉬와 이차 캐쉬- 로의 교체 정책을 따로 구분하여 사용한다면 좀 더 효율적인 결과를 보여 줄 것이다. 이 절에서는 재사용 정보를 획득하는 방법과, 이를 통해 얻어진 정보를 이용하여 캐쉬 교체 정책에 적용하는 경우, 그리고 또 다른 문제점인 Ping-Pong 효과를 고려한 수정된 교체 정책을 제안한다.

3.2.1 데이타 재사용에 대한 예측

프로세서의 임의의 데이타에 대한 재사용 여부를 예측하기 위해서는, 해당 데이타에 대한 과거 프로세서의 접근들을 기록하고 있어야 한다. 이를 위해서는 일반 캐쉬 라인이 가지는 유효 비트, 태그 비트, 데이타 비트 외에 추가로 접근 기록 비트를 두어야 한다.

접근 기록 비트는 프로세서에 의해 캐쉬 라인이 접근되는 경우, 그 횟수를 저장해 두는 일종의 카운터 비트(Counter Bit)이다. 이는 해당 캐쉬 라인에의 접근 횟수를 저장하며, 카운터 값은 최초 해당 라인이 캐쉬에 할당될 경우는 0, 그 후 프로세서에서 읽기 혹은 쓰기 요청이 발생하여 캐쉬 라인을 참조, 유효한 값을 가진 라인일 경우는 그 값을 증가시키도록 한다. 캐쉬 라인의 재사용 여부는 각 캐쉬 라인이 가지는 카운터의 값이 임계값에 다다르면 이전 프로세서에서 자주 사용되었던 데이타로 인정하여, 해당 라인은 미래에도 프로세서에 의해 재사용 되어질 것이라 예측한다. 일반적인 캐쉬 라인과 재사용 정보 획득을 위한 수정된 캐쉬 라인이 그림 6에 나타나 있다.

3.2.2 재사용 정보를 사용한 교체 정책 구현

3.2.1절에서 수정되어 제시된, 재사용 정보를 이용한 캐쉬 라인은 다음과 같이 읽기 동작과 쓰기 동작에 따라 교체 정책이 구현된다. 읽기 동작에 있어서 프로세서는 상위 계층 캐쉬로 데이타에 대한 요청을 보낸다. 이 경우 일차 캐쉬와 희생 캐쉬를 병렬적으로 탐색하여 해당 데이타의 존재 여부를 검색한다. 일차 캐쉬에서 적중이 될 경우, 프로세서에게 해당 데이타를 전송함으로서 읽기 동작을 마친다. 만약 일차 캐쉬가 아닌 희생 캐쉬에서 적중을 할 경우 해당 데이타는 프로세서에 보내어지고, 일차 캐쉬와 희생 캐쉬간의 데이타 교환이 일어나게 되는데, 이 때 일차 캐쉬의 해당 엔트리의 캐쉬 라인의 접근 기록 비트를 확인하여 유효한 경우에는 희생 캐쉬와 교환이 이루어지게 되나, 접근 기록 비트가 유효하지 않는 경우는 일차 캐쉬에서 하위 계층으로 교체되는 동작을 취하게 된다. 그리고 끝으로 희생 캐쉬와 일차 캐쉬 모두 해당 데이타가 존재하지 않는 경우는 하위 캐쉬로부터 읽어온 유효한 데이타 값을 일차 캐쉬의 지정된 엔트리의 캐쉬 라인과 교체하여야 하는데, 교체되는 라인의 접근 기록 비트가 유효한 값을 가지게 된다면 하위 계층으로 보내지 않고 희생 캐쉬로 보내어지게 되며 하위 계층으로 보내어지는 값은 희생 캐쉬에서 교체되는 라인을 보내게 된다. 그러나 접근 기록 비트가 유효하지 않는다면 희생 캐쉬로 보내지 않고 직접 하위 계층 캐쉬와 교체를 하도록 한다. 그림 7은 각각 프로세서로부터의 읽기 요청이 일차 캐쉬에서 참조 실패를 하고 희생 캐쉬에서 적중한 경우와, 일차 캐쉬와 희생 캐쉬 둘 다 참조 실패한 경우의 교체 정책을 보여준다.

쓰기 동작도 읽기 동작과 유사하다. 쓰기 요청 역시 일차 캐쉬와 희생 캐쉬를 동시에 검색하며, 희생 캐쉬 내에 프로세서에서 요청하는 데이타가 존재한다면 일차 캐쉬의 접근 기록 비트의 유효 여부를 판단하여 일차 캐쉬와 교환 후 쓰기 동작을 취하거나, 교환하지 않고 단순히 해당 일차 캐쉬 라인에 교체시키는 동작을 취한다. 쓰기 요청이 일차 캐쉬와 희생 캐쉬 모두 적중하지 않는다면 읽기 동작과 마찬가지로 하위 계층으로 데이타를 요청하게 되며 같은 인덱스를 가지는 일차 캐쉬 라인은 접근 기록 비트의 유효 여부에 따라 희생 캐쉬로 할당할 것인지, 하위 계층으로 보낼 것인지 판단하게 된다.

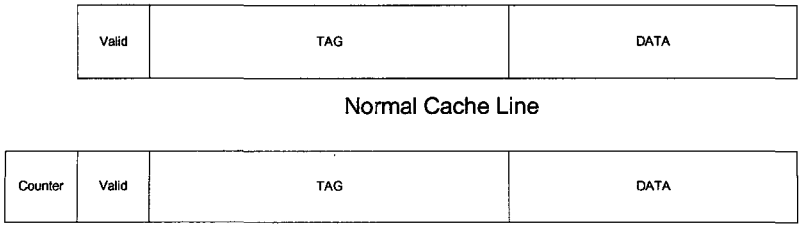


그림 6 일반 캐쉬 라인과 수정된 캐쉬 라인 구조

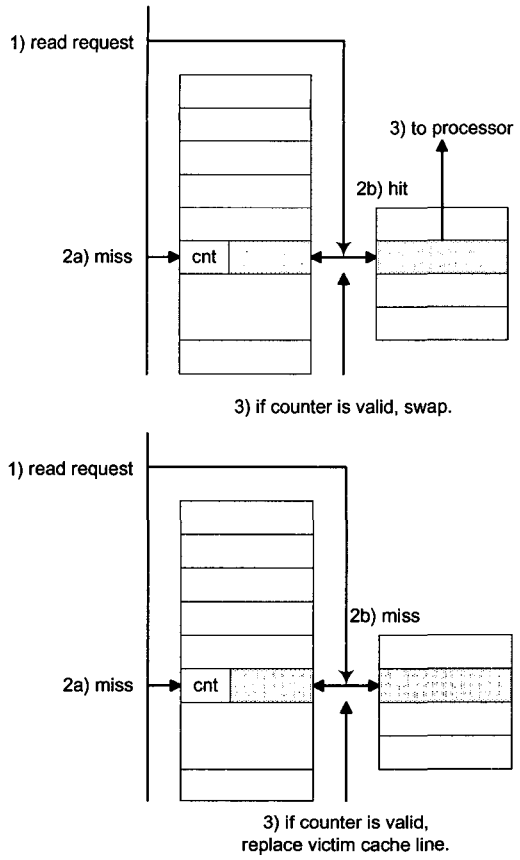


그림 7 재사용 정보를 이용한 교체(읽기 동작)

그러나 초창기 설계 당시의 희생 캐쉬는 일차 캐쉬에서 교체되어 희생 캐쉬에 할당된 데이타가 다시 프로세서에 의해 재요청되는 현상(Ping-Pong Effect)을 고려하여 설계되었다. 일반적으로 상위 캐쉬는 하위 캐쉬보다 크기가 작으므로, 하위 캐쉬에 서로 다른 영역에 할당된 라인이라 하더라도, 상위 캐쉬의 인덱싱(Indexing)에 따라 같은 집합(Set)내에 할당될 수 있으며, 이 때 집합내의 여유 공간이 없으면 캐쉬 교체 정책에 의해 특정 라인이 교체 되어야 한다. Ping-Pong 효과란 이와 같이, 비록 하위 캐쉬에서는 서로 다른 영역에 위치한다 하더라도, 상위 캐쉬의 특정 집합 내에 사상되는 라인이 프로세서에 의해서 연속적으로 요청 될 경우, 상위 캐쉬 라인 상에서 라인간의 충돌로 인해 연속적으로 발생하는 상호 교체 현상을 의미한다. 이와 같은 현상은 집합의 크기가 작은 경우에 특히 심하다. 다시 말해 재사용 정보에 의해서만 희생 캐쉬와 일차 캐쉬 간에 교체가 일어난다면 앞서 언급한 Ping-Pong 효과로 인하여 충분한 성능의 개선이 이루어지지 않을 뿐만 아니라, 더 악화 될 수도 있다. 그러므로 다음 절에서는 앞서 제시한 재사용 정보를 이용한 교체 정책에 대해서 Ping-Pong 효과를 고려한 수정안을 제시한다.

3.2.3 Ping-Pong 효과를 고려한 교체 정책 수정

재사용 정보를 이용한 교체 정책의 성능을 향상시키기 위해서는, Ping-Pong 효과를 발생시키는 라인에 대해 기존의 일반 정책을 유지하여 충돌 실패를 줄이면 된다. 이번 절에서는 이와 같은 효과를 일으킬 수 있는 후보를 선정하는 방법에 대하여 논의한다.

Ping-Pong 효과를 반영하기 위한 수정된 형태의 캐쉬 라인이 그림 8에 나타나 있다. 그림 6과 비교하여 FT 비트가 새로이 추가 되었다. 이것은 시스템이 시작된 이후 캐쉬에 아무런 초기화가 이루어지지 않은 상태에서 캐쉬 라인의 최초 사용 유무를 나타낸다. 즉 시스템이 구동된 후에 캐쉬에 초기화가 이루어질 때, 비어있는 해당 캐쉬 라인의 FT 비트에는 유효한 값을 할당하게 된다. 그 후 캐쉬 라인에 데이타가 할당되고, 추가 접근이 이루어지고 난 후, 다른 데이타로 교체 시 유효하지 못한 값을 가지게 된다. 즉 FT 비트 값이 유효하다면 최초 할당된 라인이므로, 카운트 비트가 임계값 이하라 하더라도 적중 실패를 하는 경우, 재사용 정보를 이용하여 하위 캐쉬로의 교체를 수행하지 않고 예외적으로 희생 캐쉬에 할당하도록 수정한다. 이것은 시스템이 시작된 이후, 최초로 캐쉬 라인에 할당된 경우 희생 캐쉬 역시 전체 라인에 모두 데이타를 가지고 있지 않는 문제를 해결하기 위해서 추가한 비트이다. 즉 희생 캐쉬가 비어있는 상태이기 때문에, 시스템 가동 후 초기에 한해서 조금이라도 의미 있는 값을 희생 캐쉬에 보관하기 위해서이다.

또한 Ping-Pong 현상을 반영하기 위해서, 일차 캐쉬와 희생 캐쉬간의 교체 정책 역시 수정되어야 한다. 수정 대상은 3.2.2절에서 설명되었던 읽기 연산의 시나리오 중에서, 희생 캐쉬에서만 적중하여 주소 라인의 해당 캐쉬 엔트리와 교환이 일어날 경우이다. 이 때는 재사용

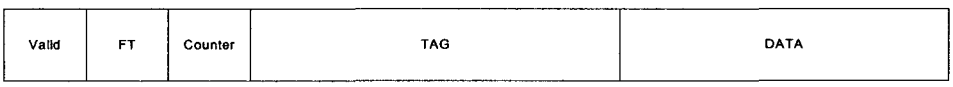


그림 8 Ping-Pong 효과를 고려한 수정된 캐쉬 라인

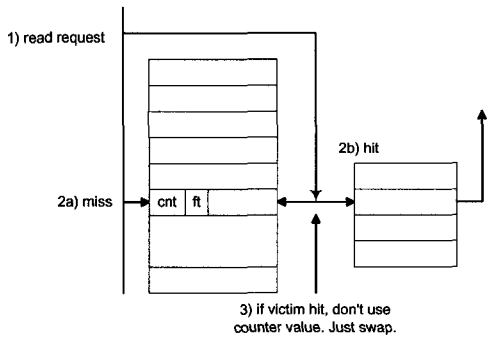
정보를 사용하지 않고 일차 캐쉬의 데이타가 희생 캐쉬로, 희생 캐쉬의 데이타는 일차 캐쉬의 해당 엔트리로 바로 교환이 이루어지도록 수정한다. 즉 재사용 정보는 캐쉬 라인과 희생 캐쉬 모두에게서 적중 실패가 발생하여 데이타가 희생 캐쉬로 할당 되어야 하는지를 결정할 경우에만 사용하도록 한다. 다시 말해 두 캐쉬 모두에게서 적중이 실패하는 경우에 재사용 정보를 사용하는 것이 원래의 희생 캐쉬 정책을 그대로 이용하면서 추가로 재사용 정보를 이용하게 되므로 더 나은 결과를 보여 줄 것이라 예상된다. 이러한 Ping-Pong 효과를 고려한 수정된 캐쉬 라인의 교체 정책 알고리즘은 그림 9와 같다.

그림 10은 수정되어진 교체 정책의 예제이다. 또한 그림 11은 FT 비트를 이용하여 최초로 캐쉬에 할당되어진 라인의 희생 캐쉬 할당 여부를 나타낸 것이다.

```
//메모리 및 시스템 초기화의 수행, 모든 캐쉬 라인의 FT 비트 세팅
System_Init();

while(Processor_Request())
{//일차 캐쉬 적중
if(HIT_L1Cache(Requested_Line))  {
    //참조된 라인의 카운트 값의 증가
    if(Requested_Line.count < Reuse_Threshold)
        Requested_Line.count++;
    Fetch_to_Processor(Requested_Line);
    }
//희생 캐쉬에서의 적중
else if(HIT_VictimCache(Requested_Line))  {
    //FT 비트의 확인
    if(L1.Target_Line.FTbit == 1) L1.Target_Line.FTbit = 0;
    //재사용 정보를 이용하지 않고, 교체 작업의 수행
    SWAP(L1.Target_Line, VC.Target_Line);
    Fetch_to_Processor(Requested_Line);
    }
//일차 캐쉬와 희생 캐쉬 모두에게서 참조 실패를 하는 경우
else{
    //FT 비트의 확인
    if(L1.Target_Line.FTbit == 1) {
        L1.Target_Line.FTbit = 0;
        SWAP(L1.Target_Line, VC.Target_Line);
        }
    Fetch_from_L2Cache(Requested_Line);
    //교체 대상 라인의 카운트 값과 재사용 기준값의 비교
    if(L1.Target_Line.Count >= Reuse_Threshold)
        SWAP(L1.Target_Line, VC.Target_Line);
        VC_Replace(Target_Line);
    else
        SWAP(L1.Target_Line, L2.Requested_Line);
    Fetch_to_Processor(Requested_Line);
    }
```

그림 9 수정된 재사용 정책을 이용한 상위 캐쉬 교체 알고리즘



Modified Read Hit in Victim Cache

그림 10 수정된 재사용 정보를 이용한 교체 정책

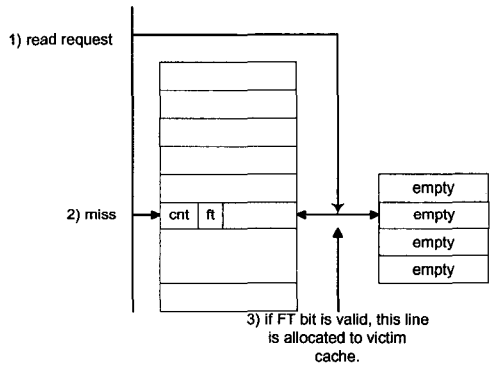


그림 11 FT 비트에 의한 희생 캐쉬 할당

4. 시스템 성능 분석

본 연구에서 제안한 기법을 평가하기 위해서 시뮬레이션을 수행하였다. 시뮬레이션에는 프로그램 구동방식의 시뮬레이터인 Augmint[13,14]를 사용하였다. Augmint는 UIUC에서 개발된 시뮬레이션 툴킷으로, 단일 혹은 다중 프로세서 환경을 메모리 구조와 더불어 프로그램 구동 방식으로 시뮬레이션 할 수 있도록 한 소프트웨어 패키지라 할 수 있다. 이러한 Augmint는 기존의 프로그램 구동형 시뮬레이터인 MINT[15]를 x86 구조에서도 수행 가능하도록 수정한 형태의 시뮬레이터이다.

4.1 시뮬레이션 환경 및 인자

시뮬레이션 환경에서 프로세서의 속도는 500MHz의 클럭이며, 이 프로세서가 100MHz의 버스에 연결되어서 동작하게 된다. 일차 캐쉬의 크기는 4KB에서 16KB까지 설정하며, 캐쉬 라인의 크기는 32Byte이다. 희생 캐쉬는 4개의 엔트리를 가진다. 자세한 내용은 표 2와 같다.

4.2 벤치마크 프로그램

시뮬레이션에서 사용되는 입력 프로그램으로는 SPLASH-2[16]에서 제시된 병렬 프로그램이다. SPLASH-

표 2 시뮬레이션 인자

인자	값
프로세서 수	1,4
일차 캐쉬의 크기	4K,8K,16K
일차 캐쉬 라인의 크기	32byte
희생 캐쉬의 크기	128byte
희생 캐쉬 엔트리 수	4
희생 캐쉬 라인의 크기	32byte
프로세서 클럭 속도	500MHz
프로세서 캐쉬 접근 시간	2 프로세서 클럭
버스 요청 명령 전송 시간	9 버스 클럭
버스의 단위 블록 데이타 전송 시간	18 버스 클럭

2 벤치마크 프로그램들은 과학, 공학, 그래픽 등의 분야에서 사용하는 계산유형을 포함하는 여러 개의 응용 프로그램을 제공한다. 본 연구에서 사용된 각 응용 프로그램의 특성은 다음과 같으며, 표 3은 입력 값의 크기이다.

- FFT : 전치 행렬을 구하는 과정에서 모든 프로세서들 간에 서로 통신을 발생시킨다.
- BARNES : Barnes-Hut Hierarchical N-body Problem 방식을 사용하여 각 입자(eg. 우주의 은하)들 사이에 상호 작용을 구하는 프로그램이다.
- MP3D : Simulated Wind Tunnel 안에서 매우 높은 입자들의 흐름을 관찰하는 프로그램이다.

표 3 벤치마크 입력 크기

응용 프로그램	인자
FFT	512K Complex Doubles
BARNES	512 Particles, 418 Seeds
MP3D	50, 30, 10 Steps

4.3 실험 결과

캐쉬 교체 정책이 바뀌었을 때의 가장 분명한 비교 수치는 프로세서 접근에 대한 실패율 혹은 적중 성공률이다. 시뮬레이션을 통해 FFT, BARNES, MP3D의 경우, 일차 캐쉬의 크기 변화에 대한 각각의 상황에 대해서 캐쉬 참조 실패율을 측정해 보았다. 희생 캐쉬의 크기는 4개의 엔트리를 가지도록 하였으며, 라인의 크기는 캐쉬와 같은 32Byte로 하였다. 일차 캐쉬의 크기는 4KB, 8KB, 16KB로 그 크기를 바꾸어 실험해 보았다. 실제 오늘날 주로 사용되는 컴퓨터 환경에서의 일차 캐쉬는 해당 시스템이 내장형 소규모 시스템인가, 데스크탑 환경인가, 아니면 워크스테이션이나 서버급 환경인가에 따라서 일반적으로 그 크기에 다소 차이가 있긴 하지만, 상대적으로 이차 혹은 삼차 캐쉬에 비해 그리 큰 사이즈가 아니다. 이는 일차 캐쉬가 프로세서와 직접 인터페이스가 이루어지기 때문에, 적절한 적중 시간(Hit Time)의 보장을 위해서 상대적으로 이차 혹은 삼차 캐쉬에 비해 공간 제약 조건이 강함에서 기인한다. 따라서 본 논문에서는 이에 대한 적절한 반영을 위해 일차 캐쉬의 크기를 4KB, 8KB, 16KB로 변화해 가면서 시뮬레이션을 수행 하였다.

본 논문에서는 이에 앞서 우선 재사용 카운터의 크기, 즉 적정 카운터 비트수를 결정하기 위한 실험을 수행한다. 이는 프로세서의 데이타에 대한 접근 횟수에 따라 캐쉬 라인이 재사용 되는지 여부를 판단하는 요소가 된다. 먼저 일반적인 희생 캐쉬보다 더 나은 성능을 보이기 위해서는 재사용 가능성을 가장 잘 예측할 수 있는 카운터의 한계치를 찾아야 한다. 그리고 그 한계치의 크기에 따라 캐쉬 라인에 추가되는 비트의 크기가 변화될 것이며, 추가 하드웨어 구성 비용도 계산 될 수 있다. 캐쉬 크기가 4KB에서 프로세서 접근 카운터의 한계치를 2에서부터 5까지 변화시키면서 실험하여 결과를 얻어내었다.

그림 12는 표 3에서 제시된 벤치마크 프로그램들에 대한 평균값으로, 세로축은 적중 실패율을, 가로축은 카운터의 한계치를 나타낸다. 그리고 비교 대상이 되는 P1, P4, P16은 각각 프로세서의 수를 의미한다. 특히 가로축의 카운트 값은 프로세서의 재사용 판단 기준으로 필요로 하는 최소한의 접근 횟수를 의미한다. 가령, 'CNT=3'의 경우는 프로세서의 재사용에 대한 판단 기준으로, 최소 3회의 접근을 필요로 함을 의미한다. 즉 프로세서에 의한 캐쉬 라인의 최초 접근시에 해당 라인의 카운터 값에 0을 할당한 후, 그 뒤 추가적인 2회의 접근이 발생할 때 재사용 라인으로 결정된다는 것을 의미한다. 다시 말해 재사용의 판단 기준값은 0, 1, 2 중에서 2가 되며, 이 때 카운터의 구현을 위해 필요한 비트수는 2 비트이다. 결과를 보면 캐쉬에 최초 할당 후에 한번의 접근이 있던 라인을 후보로 선택하는 것이 가장 낮은 적중 실패율을 가지는 것을 알 수 있다. 이러한 결과를 토대로 캐쉬 라인을 다시 수정한다면 카운터의 값으로 1 비트를 두어 최초 캐쉬에 할당하는 경우 0으로 값을 지정한 후, 프로세서에 의해 재접근 되는 경우 카운터의 값을 1로 지정하면 된다. 그 후 교체가 발생할 때 1이 저장되어 있는 라인에 한해서 희생 캐쉬에 할당

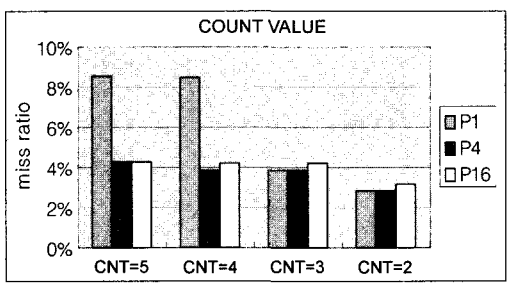


그림 12 카운터 한계치의 변화에 따른 성능 비교

한다면 추가되는 하드웨어 비용을 크게 감소시킬 수 있을 것이다.

그림 13은 일차 캐쉬의 크기가 4KB인 경우에 각 응용 프로그램에서 일차 캐쉬에 참조 실패율이 얼마나 발생하는가를 나타낸다. 가로축은 각각의 응용 프로그램을 나타내며 또한 프로세서의 개수를 나타낸다. 가로축 각 인덱스에는 네 개의 그래프가 있는데, 각각 일차 캐쉬만이 존재하는 경우(L1), 일차 캐쉬에 기존의 단순 희생 캐쉬를 추가한 경우(L1V), 희생 캐쉬 교체 정책에 재사용 정보를 추가한 경우(L1VR), 마지막으로 희생 캐쉬에 Ping-Pong 효과를 고려한 수정된 형태의 재사용 정보 교체 정책을 사용한 경우(L1VMR)를 의미한다. 특히 그림에서 보여 지듯이 L1VMR의 경우는 프로세서가 하나인 단일 프로세서인 경우보다, 다중 프로세서 시스템에서 특히 성능 향상의 결과를 보여주고 있으며, BARNES와 FFT에서 성능 향상이 두드러진다. 이것은 해당 벤치마크가 하나의 프로세서에서보다 다중 프로세서에서 측정하는 경우에 재사용 특성이 잘 나타나기 때문이며, BARNES와 FFT의 경우는 재사용뿐만 아니라 Ping-Pong 효과가 두드러지게 나타나서 L1VR이 BARNES와 FFT에서 현저히 나빠지는 현상을 볼 수 있는 반면 L1VMR은 성능의 저하가 발생하지 않음을 알 수 있다. 이상에서와 같이 4KB의 일차 캐쉬의 경우, BARNES의 경우 최대 6.7%의 성능 향상과 아울러 평균 1.5%의 전체적인 성능 향상을 가져왔다. 일반적으로 일차 캐쉬의 적중률은 여타의 캐쉬에 비해 매우 높다는 것을 감안할 때, 이러한 결과는 두드러진 성능 향상이라 할 수 있다.

그림 14는 일차 캐쉬의 크기가 8KB이다. 캐쉬 크기가 8KB인 경우에는 모든 캐쉬 구조에서 전체적으로 참조 실패율이 감소하였다. 그리고 L1V와 L1VMR은 참조 실패가 비슷하거나 L1VMR이 0.4% 성능상의 우위를 나타내었다. 한편 4KB 크기의 캐쉬와 동일하게 FFT에서는 L1VR의 성능이 좋지 않다는 결과가 나왔다. 그러나 캐쉬 자체의 크기가 커지므로 해서 4KB의 경우에 비하여 성능 차이가 줄어들었음을 알 수 있다.

그림 15는 일차 캐쉬의 크기가 16KB이다. 일차 캐쉬의 크기가 16KB일 때는 참조 실패율이 매우 낮다. 희생 캐쉬가 없는 단순 L1 구조의 경우는 충분한 성능을 보여주지 못하고 있으나, BARNES의 경우에서는 좋은 값을 보여준다. 전체적으로 L1VMR이 조금씩 좋은 결과를 보여주며, 최대 0.6%에서 평균 0.3%의 성능상 이득을 보여준다. 특히 재사용 정보만을 이용한 L1VR이 L1VMR과 비슷한 성능을 보여주는데 이것은 Ping-Pong 효과를 나타내는 캐쉬 라인이 일차 캐쉬의 크기가 커짐에 따라 충분히 일차 캐쉬 내에 그 값을 유지하고 있음을 보여준다.

이상의 결과에서 보여주듯이, 실험 결과 BARNES의

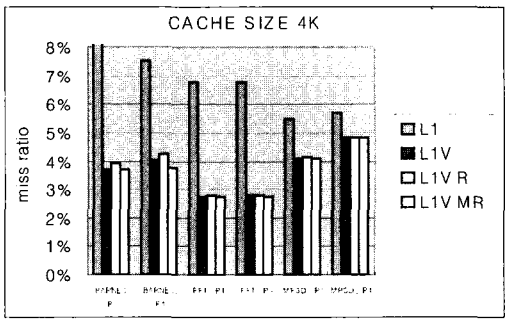


그림 13 일차 캐쉬가 4KB인 경우 참조 실패율

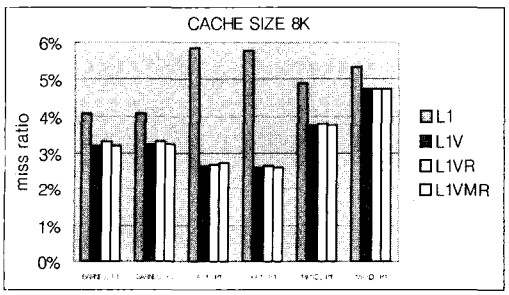


그림 14 일차 캐쉬가 8KB인 경우 참조 실패율

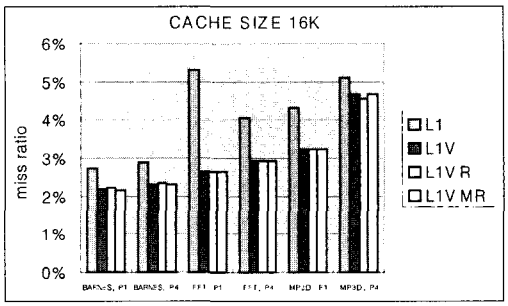


그림 15 일차 캐쉬가 16KB인 경우 참조 실패율

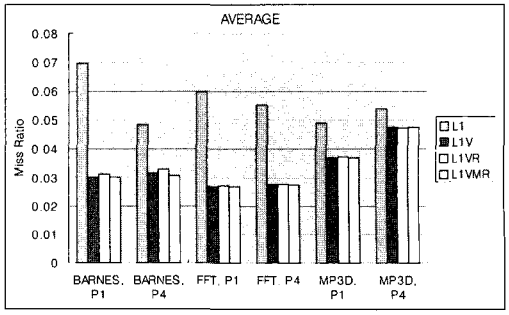


그림 16 평균 참조 실패율

경우 최대 6.7%의 성능향상을 보여주었으며, 모든 응용에 대하여 전체적으로 평균 0.5%의 성능향상을 보여주었다. 특히 빠른 접근 시간을 보장해 주는 작은 크기의 일차 캐쉬에서 L1VMR 기법이 성능면에서 더 우수한 결과를 보인다는 것을 알 수 있다. 이와 관련된 시스템 전체적인 측면에서의 통합된 평균 참조 실패율이 그림 16에 나타나 있다.

끝으로 그림 13에서 그림 15에서 보여지듯이, 특정 크기의 일차 캐쉬를 기준으로 해서 기준 크기의 두배에 해당하는 일차 캐쉬의 사용과, 기존 일차 캐쉬에 희생 캐쉬의 추가된 형태의 시스템 사이의 상관관계를 비교하여 보면 다음과 같다. 우선 그림 13과 그림 14에서 알 수 있듯이, 모든 경우의 응용에 있어서 기준 크기의 두배에 해당하는 일차 캐쉬의 사용보다는 기준 크기의 일차 캐쉬에 희생 캐쉬가 추가된 형태가 성능상에서 월등하다. BARNES의 경우 4%, FFT의 경우 51%, 그리고 MP3D의 경우 13%의 성능상 이득을 보인다. 이는 그림 14와 그림 15의 관계에 있어서도 마찬가지이다. 이러한 결과는 앞서 언급한 Norman P. Jouppi의 실험 결과인 작은 수의 엔트리를 가지는 희생 캐쉬의 추가적인 사용이 성능상에 있어 더 유리하다는 결과와 일치하는 내용이다.

그림 17에서 그림 19는 정책의 변화에 따른 수행 시간을 비교한 그림이다. 각각 캐쉬 교체 정책을 L1V, L1VR, L1VMR로 교체하였을 때 캐쉬의 크기별 수행 시간의 변화를 나타낸다. 그림에서 세로축은 수행 시간을 나타내며, 가로축은 서로 다른 캐쉬의 교체정책을 나타낸다. 그리고 그 값을 희생 캐쉬를 사용한 L1V에 대하여 상대적으로 어떠한 결과를 보여주는지를 보이기 위해, L1V를 1의 값으로 하여 상대적 값을 구하였다. L1VR은 전체적으로 L1V보다 더 많은 수행시간을 필요로 한 반면에, L1VMR의 경우에는 몇몇의 경우에는 조금 늦은 수행 시간을 보이나, 전체적으로 향상 되었다. 특히 BARNES의 경우에서는 1%-4%의 성능 향상을 가지고 왔다. 이것은 BARNES 벤치마크의 재사용 경우

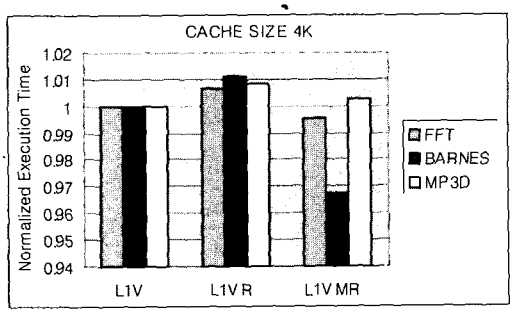


그림 17 일차 캐쉬가 4K인 경우 수행 시간

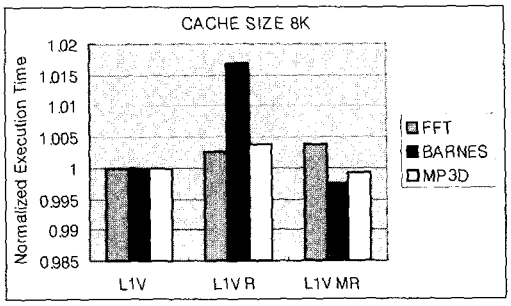


그림 18 일차 캐쉬가 8K인 경우 수행 시간

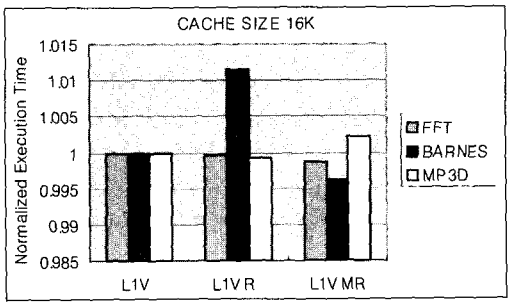


그림 19 일차 캐쉬가 16K인 경우 수행 시간

가 다른 벤치마크에 비하여 매우 많이 나타나며, 그렇기 때문에 재사용 정보를 이용하여 교체를 실시하는 경우에 성능 향상을 보이는 것이다.

5. 결 론

일반적으로 일차 캐쉬는 프로세서에서 주 메모리로의 요청을 좀 더 빠르게 처리하기 위하여 계층 메모리 구조의 상위에 위치한다. 이러한 일차 캐쉬의 성능을 향상시키기 위하여 희생 캐쉬라는 작은 수의 엔트리를 가진 버퍼를 두고, 이를 일차 캐쉬와 동시에 참조하게 하여 성능 향상을 가져왔다.

본 논문에서는 이러한 희생 캐쉬와 일차 캐쉬 상호간의 재사용 정보를 이용한 상위 레벨 캐쉬의 새로운 관리 정책을 제안하였으며, 또한 재사용 정보의 추적을 위한 접근 기록 비트의 추가를 제안하였다. 그리고 이를 통해 성능상의 향상을 추구하였다. 그 결과 프로그램의 분석을 통하여 특정 주소 영역에 대한 프로세서의 재사용 여부에 대한 예측이 가능하다는 것을 확인하였으며, 또한 일차 캐쉬와 희생 캐쉬 간의 데이타 교체가 발생할 경우, 이러한 재사용 정보를 이용할 수 있음을 보였다. 또한 희생 캐쉬 본래의 주된 사용 목적인 Ping-Pong 효과의 반영을 위해서, 해당 현상을 일으키는 라인에 대해서는 재사용 비트에 대한 정보를 이용하지 않는 수정된 형태의 프로세서 재사용 정보 정책을 제안하

였다. 그리고 이를 통해, 일반 희생 캐쉬 구조와 유사하면서 재사용 유무까지 결정할 수 있는 새로운 구조를 제시하였다. 이렇게 제안된 새로운 구조는 기존의 단순한 희생 캐쉬 관리 정책에 비해서 우수한 성능을 보이며, 또한 다중 프로세서 시스템에서 성능상의 이득이 더 크다는 것을 보였다.

본 논문의 향후 연구 과제로써, Ping-Pong 효과를 일으키는 라인, 혹은 교체시키려는 라인보다 좀 더 짧은 수명을 가진 라인에 대한 보다 더 정확한 예측 방법이 필요하며, 이를 이용하여 더 나은 성능 향상을 기대할 수도 있을 것이다.

참 고 문 헌

[1] J. L. Hennessy and D.A Patterson, Computer Architecture : A Quantitative Approach, Second Edition, Morgan Kaufmann Publishers, Inc, 1996.

[2] Norman P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," 17th Int'l Symp. on Computer Architecture, May 1990.

[3] T. Jonson and W. W. Hwu, "Run-Time Adaptive Cache Hierarchy Management vi Reference Analysis," Proc. ISCA-24, pp. 315-326, 1997.

[4] Steve Furber, "ARM : System-On-Chip Architecture", Second Edition, Addition-Wesley, 2000.

[5] Norman P. Jouppi, "Victim and miss caching as an improvement on hardware organizations," 1991.

[6] M. Prvulovic 외 3인 "Split Temporal/Spatial Cache: A Survey and Reevaluation of Performance," IEEE TCCA Newsletters, 1999.

[7] J. A. Rivers and E. S. Daividson, "Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design," Prooc. 1996 ICPP, 99. 154-163, Aug. 1996.

[8] JUN YANG and RAJIV GUPT "Frequent Value Locality and its Application," ACM Transactions on Embedded Computing Systems, Vol. 1, No. 1, November 2002, Pages 79-105.

[9] Ben Juurlink, "Unified Dual Data Caches," Euromicro Symposium on Digital Systems Design (DSD'03).

[10] A. Gonzalez, C. Aliagas, and M. Valeno, "Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality," Proc. Conf. Supercomputing, pp. 338-347, 1995.

[11] David H. Albonesi, "Selective Cache Ways: On-Demand Cache Resource Allocation," International Symposium on Microarchitecture, 2000.

[12] C. H. Chi and H. Deltz, "Improving Cache Performance by Selective Cache Bypass," Proc. 22nd Hawaii Int'l Conf. System Science, pp. 277-285, 1989.

[13] A.-T. Nguyen, M. Michael, A. Sharma, J. Torrellas, "The Augmint multiprocessor simulation toolkit for Intel x86 architectures," 1996 International Conference on Computer Design (ICCD '96).

[14] Augmint : A multiprocessor simulator, http://iacoma.cs.uiuc.edu/augmint/

[15] Jack E. Veenstra, Robert J. Fowler, "MINT : A front end for efficient simulation of shared-memory multiprocessors," In Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems(MASCOTS), pp. 201-207, 1994.

[16] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and Anoop Gupta, "Methodological Considerations and Characterization of the Splash-2 Parallel Application Suite," In Proceedings of the 22th International Symposium on Computer Architecture, pp. 24-36, May 1995.

곽 종 욱
정보과학회논문지 : 시스템 및 이론
제 31 권 제 8 호 참조

이 현 배
1998년 2월 경희대학교 컴퓨터공학과 학사. 2003년 2월 서울대학교 컴퓨터공학과 석사. 2004년~현재 LG전자기술원 모바일 멀티미디어 연구소. 관심분야는 컴퓨터 구조, 메모리 시스템

장 성 태
정보과학회논문지 : 시스템 및 이론
제 31 권 제 8 호 참조

전 주 식
정보과학회논문지 : 시스템 및 이론
제 31 권 제 8 호 참조