論文95-32B-6-4

다양한 블럭 크기를 갖는 섹터 캐시 메모리의 Trace-driven 시뮬레이션 알고리즘

(A New Trace-driven Simulation Algorithm for Sector Cache Memories with Various Block Sizes)

朴 東 圭 *

(Dong Gue Park)

요 약

본 논문에서는 서브블럭(sub_block)크기, associativity, 집합의 갯수, 블럭 크기가 서로 다른 섹터(sector)캐시 메모리의 실패율과(miss ratio)과 버스 트래픽(bus traffic)을 동시에 구할수 있는 trace-driven 시뮬레이션 알고리즘을 제안한다. 섹터 캐시 메모리의 성능을 평가하기 위하여 trace-driven 시뮬레이션 방법이 대표적으로 사용되고 있다. 그러나 기존의 trace-driven 시뮬레이션 알고리즘은 서로 다른 여러 섹터 캐시 모델의 성능을 평가 하기 위하여 매번 긴 트레이스를 가지고 시뮬레이션 해야하기 때문에 시간이 많이 소비된다. 제안된 시뮬레이션 알고리즘은 섹터 캐시 메모리의 trace-driven 시뮬레이션을 수행시 어드레스 트레이스(address trace)를 한번의 입력으로 사용함으로써 서브 블럭크기, associativity, 집합의 갯수및 블럭 크기가 서로 다른 섹터 캐시 모델을 동시에 시뮬레이션 할 수있고, 이로써 시뮬레이션 수행 시간을 감소 시킬 수 있다. 제안된 알고리즘을 C언어로서 실현하여 다양한 예제 프로그램 수행시 얻어진 어드레스 트레이스를 입력으로 trace-driven시뮬레이션을 실행한 결과, 그 실행 시간이 기존 알고리즘의 실행시간보다 상당히 감소됨을 확인 할수 있다.

Abstract

In this paper, a new trace driven simulation algorithm is proposed to evaluate the bus traffic and the miss ratio of the various sector cache memories , which have various sub-block sizes and block sizes and associativities and number of sets, with a single pass through an address trace. Trace-driven simulaton is usually used as a method for performance evaluation of sector cache memories, but it spends a lot of simulation time for simulating the diverse cache configurations with a long address trace. The proposed algorithm shortens the simulation time by evaluating the performance of the various sector cache configurations, which have various sub-block sizes and block sizes and associativities and number of sets , with a single pass through an address trace. Our simulation results show that the run times of the proposed simulation algorithm can be considerably reduced than those of existing simulation algorithms, when the proposed algorithm is implemented in C language and the address traces obtained from the various sample programs are used as a input of trace-driven simulation.

* 正會員, 順天鄕大學校 情報通信工學科
(Dept. of Information and Communcation Eng., College of Eng., Soonchunhyang Univ.)

※ 이 논문은 1993년도 한국학술진흥재단의 공모과제 연구비에 의하여 연구 되었음.
接受日字: 1994年7月13日, 수정완료일: 1995年5月27日

I. 서 론

오늘날 대부분의 컴퓨터 시스템은 CPU의 처리 속도와 메모리 엑세스 시간의 차이를 줄이기 위하여 고속의 버퍼(buffer)인 캐시 메모리를 사용하고 있다.

또한 회로 설계기술이 발달함에 따라 회로의 집적도가 높아짐으로써 프로세서 외부에 위치하던 캐시 메모리를 프로세서 내부에 위치하도록 하는 것이 가능하게 되었다. 컴퓨터 시스템에서 온칩(on-chip)캐시 메모리의 사용은 평균 메모리 엑세스 시간과 버스 트래픽을 줄임으로써 컴퓨터 시스템의 성능을 본질적으로 향상 시킨다. 따라서 온칩 캐시 메모리를 사용하는 컴퓨터 시스템의 성능을 결정하는데 있어서 온칩 캐시 메모리의 설계 방법과 실현 방법은 중요한 요인이 된다.[1][2][3][4]

일반적으로 온칩 캐시 메모리들은 한정된 회로 크기와 chip packing 제약으로 placement 알고리즘에 있어서 기존의 오프칩(off-chip)캐시 메모리들과는 달리 작은 크기의 데이타전송을 하는 섹터 매핑을 많이 사용한다.[3,4] 이러한 섹터 매핑방식을 사용하는 캐시 메모리를 섹터 캐시 메모리라고 하며, 섹터 캐시 메모리의 성능을 평가하기 위해서 trace-driven 시뮬레이션이 대표적으로 사용되고 있다. 이 방법은 어드레스 트레이스들을 외부의 입력으로 사용함으로써 제안된 캐시 메모리의 설계 변수에 따라 캐시 성능을 검토 및 분석한다.[1][3][4][5][6]

그러나 trace-driven 시뮬레이션은 긴 트레이스를 가지고 associativity와 서브 블럭의 크기, 집합의 갯수및 블럭 크기가 서로 다른 여러 섹터 캐시 모델을 시뮬레이션해야 하기 때문에 시간이 많이 소비되는 단점을 가지고 있다. 이러한 단점을 해결하기 위하여 하나의 어드레스 트레이스로 동시에 서로 다른 여러 캐시 모델의 실패율과 버스 트래픽을 구하는 알고리즘에 관한 연구가 활발히 진행되어 왔다.[1] ~ [8]

시뮬레이션 시간을 줄이기 위해서 J.G.Thompson[6]은 하나의 어드레스 트레이스로 associativity가 다른 여러 섹터 캐시 모델의 실패율을 동시에 구하는 알고리즘을 개발했다.[4] 그러나 이 알고리즘은 서브 블럭 크기가 다른 섹터 캐시 모델의 실패율을 구하기 위해서는 서브 블럭 크기가 변할 때마다 시뮬레이션을 수행해야 하기 때문에 시간이 많이걸리는 단점이 있었다.[4][6] 이러한 단점을 해결하기 위해 J.G. Thompson[6]의 알고리즘을 확장시켜 서브 블럭의 유효 레벨 inclusion개념을 사용함으로써 하나의 어드레스 트레이스로 서브 블럭및 associativity가 다른 여러 섹터 캐시 모델의 실패율을 동시에 구하는 알고리즘도 개발되었다.[4] 그러나 이 알고리즘도 블럭 크기 및 집합의 갯수가 서로 다른 섹터 캐시 메모리의 성능을 평가하기 위해서 블럭 크기및 집합의 갯수가 변할 때마다 시뮬레이션을 수행해야 하기 때문에 시간이 많이 걸리는 단점이 있었다. 그리고 M.D.Hill[1][2]은 하나의 어드레스 트레이스로 집합의 갯수 및 associativity가 서로 다른 캐시 메모리의 실패율을 동시에 구하는 all-associativity방법을 개발했다.그러나 이 알고리즘도 블럭 크기가 서로 다른 캐시 모델의 실패율을 구하기 위해서는 블럭 크기가 변할 때마다 시뮬레이션을 수행해야 하기 때문에 시간이 많이 걸리는 단점이 있었다.[8] 이러한 단점을 해결하기 위해 M.D.Hill[1][2]의 알고리즘을 확장 시켜 매 블럭 크기마다 스택 모델로 정보를 유지하고, 임의의 한 어드레스 트레이스에 대하여 그 어드레스가 발견될 때까지는 매 블럭 크기의 해당 집합에서 맨 위에서 아래쪽으로 순방향으로 all-associativity 시뮬레이션을 수행하고, 그 이후의 매 블럭 크기에서는 저장된 정보를 사용하여 아래쪽에서 윗쪽으로 역방향으로 all-associativity 시뮬레이션[1][2]을 수행함으로써 하나의 어드레스 트레이스로 블럭 크기, 집합의 갯수및 associativity가 서로 다른 캐시 메모리의 실패율을 동시에 구하는 알고리즘도 개발되었다.[8] 그러나 이 알고리즘도 서브 블럭 크기를 고려하지 않았기 때문에 섹터 캐시 시뮬레이션에는 사용할 수 없는 단점이 있었다.

또한 섹터 캐시 메모리에서는 일반적으로 버스 트래픽을 줄이기 위하여 메인 메모리 update방법으로 라이트백(write-back)을 사용하는데, 이런 라이트백 섹터 캐시 메모리의 버스 트래픽을 구하기 위해서는 라이트백 섹터 캐시 메모리의 copy-back효과를 고려해야 한다. 라이트백 섹터 캐시 메모리의 copy-back 효과를 계산하기 위하여 J.G.Thompson은 각 서브 블럭에다가 dirty 비트 대신에 dirty 레벨을 사용함으로써 associativity가 다른 라이트백 섹터 캐시 메모리 모델을 동시에 시뮬레이션하는 알고리즘을 제안하였다. 그러나 이 알고리즘은 associativity가 변하는 라이트백 섹터 캐시 모델에만 한정되는 단점이 있었다.[4][7] 이러한 단점을 해결하기 위하여 서브 블럭 크기가 서로 다른 경우에 서브 블럭의 dirty 레벨 inclusion개념을 사용함으로써 서브 블럭 크기및 associativity가 다른 라이트백 섹터 캐시 메모리 모델을 동시에 시뮬레이션하는 알고리즘도 개발되었다.[4] 그러나 이 알고리즘도 서브 블럭 크기및 associativity가 서로 다른 라이트백 섹터 캐시 모델에만 한정되는 단점이 있었다.[4]

본 논문에서는 이러한 단점을 개선하기 위하여

associativity, 서브 블럭 크기 ,집합의 갯수및 블럭 크기가 서로 다른 여러 섹터 캐시 모델의 실패율과 버스 트래픽을 동시에 구하는 시뮬레이션 알고리즘을 제안한다.

제안된 알고리즘은 섹터 캐시 메모리내의 블럭 적중과 실패가 일반 캐시 메모리의 참조와 동일하다는 성질을 사용하여 블럭 크기가 서로 다른 캐시 메모리에 적용될수 있는 시뮬레이션 알고리즘[8]을 사용하여 블럭 크기및 집합의 갯수가 서로 다른 섹터 캐시 메모리내의 블럭 적중과 실패를 결정하고, 블럭 적중시에 해당 블럭이 유효한 최소 스택 깊이를 구한다. 그 다음 섹터 캐시 메모리의 각 서브 블럭마다 각 집합 매핑 함수에 관한 유효레벨을 두고, 유효 레벨에 앞 단계의 결과를 적용함으로써, 블럭 크기및 집합의 갯수가 서로 다른 경우에 섹터 캐시 메모리의 해당 서브 블럭이 유효한 최소 레벨을 결정한다. 이 결과를 기준으로 서브 블럭의 유효레벨 inclusion개념[4]을 사용하여 서브 블럭이 서로 다른 경우에 서브 블럭의 실패와 적중을 결정함으로써 블럭 크기, 집합의 갯수, associativity 및 서브 블럭 크기가 서로 다른 섹터 캐시 메모리의 실패율을 동시에 구할 수 있다. 또한 제안된 알고리즘은 associativity, 서브 블럭 크기,집합의 갯수및 블럭크기가 서로 다른 각 경우마다 dirty 레벨을 두고, 매 write 어드레스 트레이스마다 dirty 레벨을 변경시킴으로써 라이트백 섹터 캐시 메모리의 버스 트래픽을 구할 수 있다.

제안된 알고리즘은 섹터 캐시 메모리의 trace-driven 시뮬레이션 수행시 한번의 입력 어드레스 트레이스로 associativity, 서브 블럭크기, 집합의 갯수및 블럭 크기가 다른 여러 섹터 캐시 모델의 실패율과 버스 트래픽을 동시에 구함으로써 시뮬레이션 수행시간을 줄일 수 있다.

Ⅱ. 섹터 캐시 메모리의 동작원리및 시뮬레이션

온칩 캐시 메모리들은 한정된 회로 크기와 칩 패킹 제약으로 placement 알고리즘에 있어서 기존의 오프칩 캐시 메모리들과는 달리 작은 크기의 데이타전송을 하는 섹터 매핑을 많이 사용한다.

섹터 매핑을 사용하는 섹터 캐시 메모리와 이차적인 기억장치간의 상호교환은 서브 블럭의 단위로 이루어지며, 섹터 캐시 메모리의 한 블럭내에는 그 블럭내에 서브 블럭의 데이타 존재여부를 나타내주는 각각의 서브 블럭에 대한 유효 비트가 있어야 한다.

섹터 캐시 메모리의 유효 비트들은 블럭의 어느 부분은 유효로 또 어느 부분은 무효(invalid)로 규정한다. 따라서 태그의 매치 만으로는 그 워드가 반드시 캐시내에 존재한다고 말할 수 없고, 그 워드에 대한 유효비트가 1인지를 확인해야 한다. 그러므로 섹터 매핑 기법을 사용하는 섹터 캐시 메모리에서는 블럭이 캐시와 메모리사이에서 전송되는 최소단위가 아니고, 단지 어드레스 태그와 연관된 정보의 단위로 정의 된다.

섹터 캐시 메모리에 대한 참조는 다음과 같다.프로세서로 부터 한개의 메모리 참조가 있다면, 해당 어드레스의 블럭이 섹터 캐시 메모리내에 있는지를 알기 위하여 섹터 캐시 메모리내의 어드레스 태그들을 조사함으로써 블럭 적중과 블럭 실패를 결정한다. 블럭 적중인 경우에는 해당 블럭내에 해당 어드레스의 서브 블럭이 있는 지를 알기 위하여 해당 블럭의 유효비트들을 조사함으로써 서브 블럭 적중과 서브 블럭 실패를 결정한다. 해당 서브 블럭에 대한 유효비트가 1인 경우에는 해당 서브 블럭이 섹터 캐시 메모리내에 있으므로 해당 서브 블럭 데이타가 섹터 캐시 메모리로 부터 프로세서로 전달된다. 만약 해당 서브 블럭에 대한 유효비트가 0인 경우에는 메인 메모리로 부터 해당 서브 블럭을 읽어와서 섹터 캐시 메모리내에 저장한 다음, 해당 서브 블럭에 대한 유효비트를 1로 한다. 블럭 실패가 발생하면 치환을 위하여 섹터 캐시 메모리내의 블럭들중 한개가 선택되고, 블럭 실패를 일으킨 어드레스가 선택된 블럭의 태그로 저장된다. 그 블럭내의 모든 서브 블럭들에 대한 유효비트를 0으로 하고, 해당 서브 블럭을 메인 메모리로 부터 읽어서 그 블럭에 저장한 다음 해당 서브 블럭에 대한 유효비트를 1로 한다.[10, 11]

섹터 캐시 메모리에 관한 연구는 주로 trace-driven 시뮬레이션 방법에 의존한다. Trace-driven 시뮬레이션은 이미 기록된 어드레스 트레이스를 외부의 입력으로 사용하여 제안된 컴퓨터 시스템의 모델을 평가하는 방법이다. 그러나 trace-driven 시뮬레이션은 많은 어드레스 트레이스를 가지고 행하여지기 때문에 시뮬레이션 수행시간이 오래걸리는 단점을 가지고 있으며, 이러한 단점을 해결하기 위하여 시뮬레이션 수행시간을 줄이는 방향으로 많은 연구가 되어 왔다.

Mattson[5]의 논문에서는 대치 알고리즘들이 inclusion 성질을 가질때 모든 메모리 크기에 대한 실패율을 어드레스 트레이스 한번으로 계산할 수 있다는 것을 보여주고 있다. 그러나 섹터 캐시의 스택 시뮬레이션의 문제점은 유효 비트들이 inclusion을 따르지 않는다는 것이다.[7]

이러한 문제를 해결하기 위해서 J.G.Thompson은 각각의 서브 블럭에 대한 유효 비트를 유효 레벨로 대

치시키는 알고리즘을 제안하였다.[6] 이 알고리즘에서 임의의 서브 블럭에 대한 유효 레벨은 해당 서브 블럭이 유효한 최소의 메모리 크기를 나타낸다. 유효 레벨은 블럭이 참조될 때만 갱신하는 것이 필요하며, 섹터 캐시 메모리의 실패율은 블럭의 스택 레벨에 의존하는 것이 아니고 참조되는 서브 블럭의 유효 레벨에 의해 달려 있다.그러나 J.G.Thompson이 제안한 알고리즘은 associativity가 변하는 섹터 캐시 모델에만 제한되는 단점이 있다.

이러한 단점을 해결하기 위해 J.G.Thompson[6]의 알고리즘을 확장시켜 서브 블럭의 유효 레벨 inclusion개념을 사용함으로써 하나의 어드레스 트레이스로 서브 블럭및 associativity가 다른 여러 섹터 캐시 모델의 실패율을 동시에 구하는 알고리즘도 개발되었다.[4] 그러나 이 알고리즘도 블럭 크기및 집합의 갯수가 서로 다른 섹터 캐시 메모리의 성능을 평가하기 위해서 블럭 크기및 집합의 갯수가 변할 때마다 시뮬레이션을 수행해야 하기 때문에 시간이 많이 걸리는 단점이 있었다.

이러한 단점을 개선하기 위하여 본 논문에서는 섹터 캐시 메모리내의 블럭 적중과 실패가 일반 캐시 메모리의 참조와 동일하다는 성질을 사용하여 블럭 크기가 서로 다른 캐시 메모리에 적용될수 있는 시뮬레이션 알고리즘[8] 을 사용하여 블럭 크기및 집합의 갯수가 서로 다른 섹터 캐시 메모리내의 블럭 적중과 실패를 결정하고, 블럭 적중시에 해당 블럭이 유효한 최소 스택 깊이를 구한다.

그 다음 섹터 캐시 메모리의 각 서브 블럭마다 각 집합 매핑 함수에 관한 유효레벨을 두고, 유효 레벨에 앞 단계의 결과를 적용함으로써, 블럭 크기및 집합의 갯수가 서로 다른 경우에 섹터 캐시 메모리의 해당 서브 블럭이 유효한 최소 레벨을 결정한다. 이 결과를 기준으로 서브 블럭의 유효레벨 inclusion개념[4]을 사용하여 서브 블럭이 서로 다른 경우에 서브 블럭의 실패와 적중을 결정함으로써 블럭 크기, 집합의 갯수, associativity및 서브 블럭 크기가 서로 다른 섹터 캐시 메모리의 실패율을 동시에 구할 수 있는 알고리즘을 제안한다.

Ⅲ. 실패율을 구하기 위한 섹터 캐시 메모리의 시뮬레이션 알고리즘

섹터 캐시 메모리내의 블럭 적중과 실패는 일반 캐시 메모리의 참조와 동일하며 그러므로 블럭 크기, 집합의 갯수 및 associativity가 서로 다른 섹터 캐시 메모리내의 블럭 적중과 실패는 블럭 크기가 서로 다른 캐시 메모리모델들을 동시에 시뮬레이션 할 수 있는 방법을 그대로 적용하여 결정할 수 있다.[8] 따라서 블럭크기가 서로 다른 섹터 캐시 메모리를 동시에 시뮬레이션하기 위해서 블럭 크기가 서로 다른 캐시 메모리 모델들을 동시에 시뮬레이션 할 수 있는 방법을 그대로 적용하여 블럭의 적중과 실패를 결정한다. 그리고 블럭 적중시에는 블럭 크기및 집합 갯수가 서로 다른 경우에 해당 블럭이 유효한 최소 스택 깊이를 구한다. 그리고 각 서브 블럭마다 각 집합 매핑 함수에 따른 유효 레벨을 두고, 유효레벨에 앞 단계에서 구한 각 서브 블럭이 속한 해당 블럭이 유효한 최소 스택 깊이를 적용하여 해당 서브 블럭이 유효한 최소 레벨을 결정한 다음, 서브 블럭에 서브 블럭 유효 레벨 inclusion개념[4] 을 적용하여 서브 블럭 크기가 서로 다른 경우에 서브 블럭 실패와 적중을 결정함으로써 블럭 크기, 집합의 갯수, associativity및 서브 블럭 크기가 서로 다른 섹터 캐시 메모리의 실패율을 동시에 구한다.

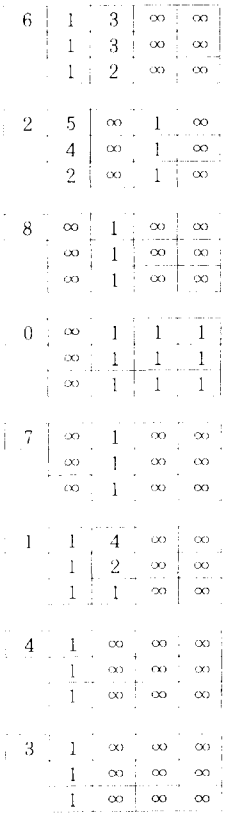

그림 1. 섹터 캐시내에 집합의 스택 모델
Fig. 1. Stack model of a set in sector cache.

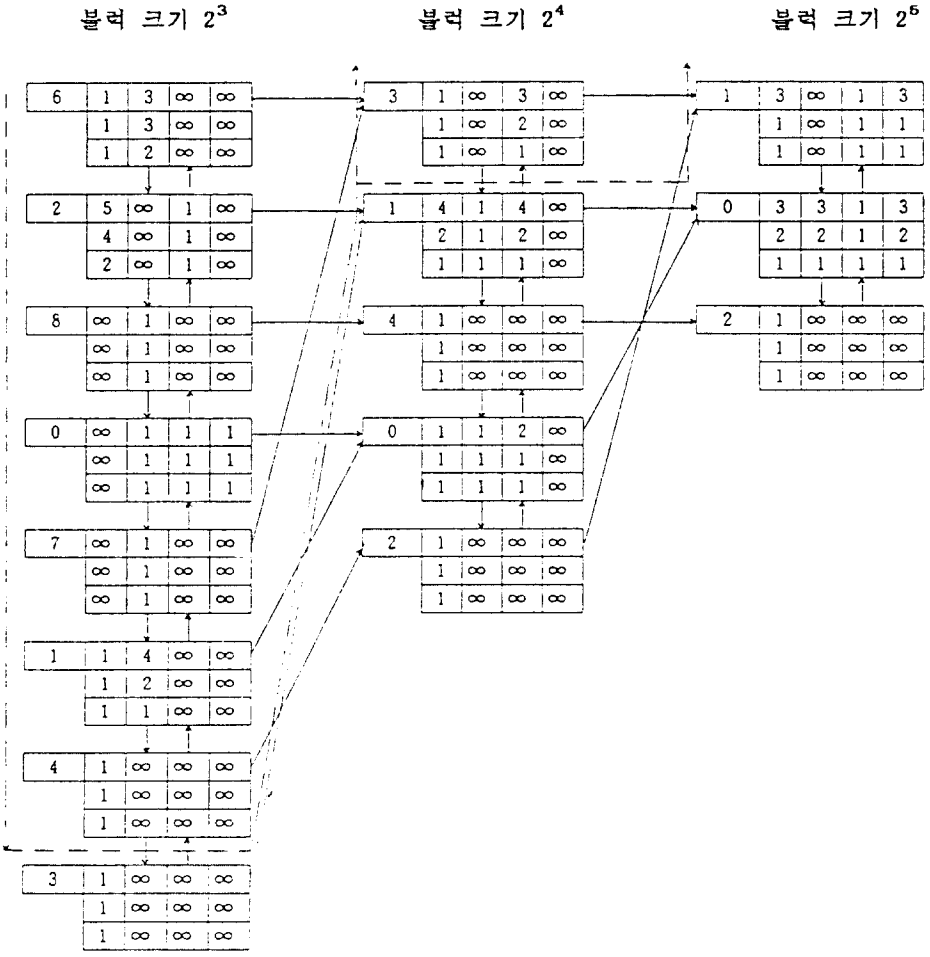

그림 2. 블럭 크기가 서로 다른 섹터 캐시 메모리에서 해당 블럭이 유효한 스택 깊이를 구하는 예

Fig. 2. Example of evaluating valid stack depth of the block in sector cache memory with various block sizes.

블럭 크기, 집합의 갯수, associativity가 서로 다른 섹터 캐시 메모리모델에서 한개의 어드레스 트레이스에 대하여, 매 블럭 크기 마다 그 어드레스에 속하는 집합내에 해당 태그의 포인터를 바로 이전 블럭 크기의 스택에서 저장한다. 그리고 블럭 크기가 변하는 모델에 대해서도 임의의 어드레스가 발견될 때까지는 매 블럭 크기에서 스택의 탑(top)에서 아래쪽 방향인 순방향으로 all-associativity 시뮬레이션[11][12]을 수행하며, 그 이후의 블럭 크기에서는 스택에 저장된 포인터를 사용하여 매 블럭 크기에서 아래쪽에서 스택의 톱(top)쪽으로 즉 역방향으로 all-associativity 시뮬레이션을 수행함으로써 해당 어드레스가 저장된 블럭의 적중과 실패를 결정하고, 해당 블럭이 유효한 최소 스택 깊이를 구할 수 있도록 한다.

예를 들어 집합 매핑함수로 bit selection을 사용하고 집합의 갯수가 2^0개인 섹터 캐시 모델에서, 서브블럭 갯수가 2^2이고 가장 작은 블럭 크기 2^3에서 집합의 스택모델이 그림 1과 같을때 블럭 크기가 서로 다른 경우에 해당 블럭이 유효한 최소 스택 깊이를 구하는 예는 다음 그림 2와 같다. 그림 1은 비어 있는 상태인 섹터 캐시 메모리에 16진수 2자리수로 구성된 입력 어드레스 트레이스 "08, 0a, 18, 20, 10, 0a, 14, 3a, 03, 04, 05, 30, 32, 42, 10, 32"가 참조된 뒤의 상태를 나타낸다.

그림 1에서 각 노드의 왼쪽은 태그를 나타내고 오른쪽의 4 열은 한 블럭을 구성하고 있는 각 서브 블럭의 유효레벨을 나타낸다. 그리고 노드에서 오른쪽이 삼행으로 구성된 것은 각 집합 매핑에 따른 유효레벨을 나타내기 때문이다.여기서각 집합 매핑 함수 마다 따로 유효레벨을 두어야 하는 이유는 각 집합 매핑 함수에 따라서 각 서브 블럭의 유효한 최소 레벨이 변하기 때문이다. 사용된 집합 매핑 함수는 $f(x) = x \text{ rem } 2^0$, $f(x) = x \text{ rem } 2^1$, $f(x) = x \text{ rem } 2^2$이다. 따라서 그림 1에서 첫번째 노드의 태그는 6이 되고 집합 매핑 함수 $x \text{ rem } 2^0$에서 각 서브 블럭 0, 1, 2, 3의 유효레벨은 1, 3, ∞,∞이며, 집합 매핑함수 x rem 21에서 각 서브 블럭 0, 1, 2, 3의 유효레벨은 1, 3, ∞, ∞이고, 집합 매핑함수 $x \text{ rem } 2^2$에서 각 서브 블럭 0, 1, 2, 3의 유효레벨은 1, 2, ∞, ∞임을 의미한다.

예를 들어 블럭 크기 2^2에서 1a의 데이타 참조가 있을 경우, 그림 2의 점선과 같은 순서로 스택을 찾아가면서 블럭 크기별로 집합 매핑 함수에 따른 해당 블럭의 스택 깊이를 구한다. 블럭 크기 2^2에서는 순방향으로 all-associativity 시뮬레이션[1 12]을 수행하고, 블럭 크기 $2^4, 2^5$에서는 역방향으로 all- associativity 시뮬레이션을 수행한다.

각 블럭 크기별로 각 집합 매핑 함수에 따라서 해당 블럭이 유효한 최소 스택 깊이를 구한 결과는 다음 표 1과 같다.

표 1. 집합 매핑 함수에 따른 섹터 캐시 메모리의 각 블럭 크기에서의 스택 깊이 계산

Table 1. Stack depth evaluation of each block of sector cache memory with various set mapping functions.

(a) 블럭 크기 2^3에서 순방향으로 all- associativity 시뮬레이션 수행한 결과

(a) Results of forward all-associativity simuLation in block 2^3

Number of LSB matched	$f(x) = x \text{ rem } 2^0$	$f(x) = x \text{ rem } 2^1$	$f(x) = x \text{ rem } 2^2$
0	1	0	0
0	2	0	0
0	3	0	0
0	4	0	0
2	4	0	1
1	4	1	1
0	5	1	1
found	5	1	2
Stack distance	5+1+2 =8	1+2 =3	2 =2

(b) 블럭 크기 2^4에서 역방향으로 all-associativity 시뮬레이션 수행한 결과

(b) Results of backward all-associativity simuation in block 2^3

Number of LSB matched	$f(x) = x \text{ rem } 2^0$	$f(x) = x \text{ rem } 2^1$	$f(x) = x \text{ rem } 2^2$
1	0	1	0
found	0	1	1
Stack distance	0+1+1 =2	1+1 =2	1 =1

(c) 블럭 크기 2^5에서 역방향으로 all-associativity 시뮬레이션 수행한 결과

(c) Results of backward all-associativity simulation in block 2^5

Number of LSB matched	$f(x) = x \text{ rem } 2^0$	$f(x) = x \text{ rem } 2^1$	$f(x) = x \text{ rem } 2^2$
1	0	1	0
found	0	1	1
Stack distance	0+1+1 =2	1+1 =2	1 =1

표 1의 (a), (b), (c)는 블럭 크기 2^3, $2^4, 2^5$인 섹터 캐시 메모리에서의 해당 블럭이 유효한 스택 깊이를 구하는 실행 결과를 나타낸다.

각 블럭 크기별로 구한 표 1의 결과를 섹터 캐시 메모리의 각 서브 블럭의 유효레벨에 적용한 결과는 표 2와 같다.

표 2. 해당 블럭내의 서브 블럭의 유효레벨 계산

Table 2. Valid Level Evaluation of sub-blocks in the block of sector cache 0memory with the stack depth of table 1.

(a) 블럭 크기 2^3 에서의 결과

(a) Results in block size 2^3

stack fully-assoc	0	1	sub-block 2	3
6	1, 1, 1	3, 3, 2	∞, ∞, ∞	∞, ∞, ∞
2	5, 4, 2	∞, ∞, ∞	1, 1, 1	∞, ∞, ∞
8	∞, ∞, ∞	1, 1, 1	∞, ∞, ∞	∞, ∞, ∞
0	∞, ∞, ∞	1, 1, 1	1, 1, 1	1, 1, 1
7	∞, ∞, ∞	1, 1, 1	∞, ∞, ∞	∞, ∞, ∞
1	1, 1, 1	4, 2, 1	∞, ∞, ∞	∞, ∞, ∞
4	1, 1, 1	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞
3	8, 3, 2	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞

(b) 블럭크기 2^4에서의 결과

(b) Resunts in block size 2^4

stack fully-assoc	0	1	sub-block 2	3
3	1, 1, 1	∞, ∞, ∞	3, 3, 1	∞, ∞, ∞
1	4,. 2. 1	2, 2, 1	4, 2, 1	∞, ∞, ∞
4	1, 1, 1	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞
0	1, 1, 1	1, 1, 1	2, 1, 1	∞, ∞, ∞
2	1, 1, 1	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞

(c) 블럭크기 2^5에서의 결과

(c) Results in block size 2^5

stack fully-assoc	0	1	sub-block 2	3
1	3, 1, 1	∞, ∞, ∞	1, 1, 1	3, 1, 1
0	3, 2, 1	3, 2, 1	2, 2, 1	3, 2, 1
2	1, 1, 1	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞

표 2에서 오른쪽의 각 서브 블럭내에 3개의 숫자는 $f(x) = x \text{ rem } 2^0$, $f(x) = x \text{ rem } 2^1$, $f(x) = x \text{ rem } 2^2$ 각 집합 매핑 함수에 따른 해당 서브 블럭의 유효레벨을 의미한다. 예를 들어 블럭 크기가 2^3인 경우에 태그가 2인 블럭내에 서브 블럭 0의 유효레벨은 집합 매핑 함수 x rem 2^0에서 5이며, 집합 매핑함수 x rem 2^1 에서 4이고, 집합 매핑함수 x rem 2^2 에서 2임을 의미한다.

표 3. 블럭 크기가 2^3인 경우에 표 2(a)의 유효 레벨을 기준으로 서로 다른 서브 블럭의 유효 레벨 계산

Table 3. Valid level evaluation of sector cache memory with various number of sub-blocks in block size 2^3 by using valid levels of table 2(a).

(a) 한 블럭당 2개의 서브블럭

(a) 2 sub-blocks per a block

stack fully-assoc	sub-block 0	1
6	1, 1, 1	∞, ∞, ∞
2	5, 4, 2	1, 1, 1
8	1, 1, 1	∞, ∞, ∞
0	1, 1, 1	1, 1, 1
7	1, 1, 1	∞, ∞, ∞
1	1, 1, 1	∞, ∞, ∞
4	1, 1, 1	∞, ∞, ∞
3	8, 3, 2	∞, ∞, ∞

(b) 한 블럭당 1개의 서브블럭

(b) 1 sub-block per a block

stack fully-assoc	sub-block
6	1, 1, 1
2	1, 1, 1
8	1, 1, 1
0	1, 1, 1
7	1, 1, 1
1	1, 1, 1
4	1, 1, 1
3	8, 3, 2

블럭 크기가 2^3인 경우에 표 2(a)의 유효 레벨을 기준으로 유효 레벨 inclusion개념[4]을 적용하면 서브 블럭 갯수가 서로 다른 경우의 유효 레벨은 표 3과 같이 구할 수 있으며, 데이타 참조 후의 결과는 표 4와 같이 모델링 될수 있다.

표 4. 블럭 크기가 2^3인 경우에 참조후의 섹터 캐시 메모리의 유효 레벨

Table 4. Valid levels of sector cache memory after an access in blocksize 2^3.

stack fully-assoc	sub-block 0	1	2	3
3	8, 3, 2	1, 1, 1	∞, ∞, ∞	∞, ∞, ∞
62	1, 1, 1	3, 3, 2	∞, ∞, ∞	∞, ∞, ∞
2	5, 4, 2	∞, ∞, ∞	1, 1, 1	∞, ∞, ∞
8	∞, ∞, ∞	1, 1, 1	∞, ∞, ∞	∞, ∞, ∞
0	∞, ∞, ∞	1, 1, 1	1, 1, 1	1, 1, 1
7	∞, ∞, ∞	1, 1, 1	∞, ∞, ∞	∞, ∞, ∞
1	1, 1, 1	4 2, 1	∞, ∞, ∞	∞, ∞, ∞
4	1, 1, 1	∞, ∞, ∞	∞, ∞, ∞	∞, ∞, ∞

블럭 크기가 2^3인 경우에 1a의 데이타 참조는 한 블럭당 서브 블럭이 4개인 경우에 블럭 3의 서브 블럭 1의 참조가 되며, 이때의 해당 서브 블럭의 유효 레벨은 그림 2와 같이 무한대이므로 실패가 된다. 그리고 블럭 크기 2^3에서 1a의 데이타 참조는 한 블럭당 서브 블럭이 2개인 경우에 블럭 3 서브 블럭 0 참조로 바뀌고 이때의 해당 서브 블럭의 유효 레벨은 표 3의 (a)에서 알수 있는 바와 같이 집합의 갯수가 한개인 섹터 캐시에서는 스택 깊이 8에서 적중되고 집합의 갯수가 두개인 섹터 캐시에서는 스택깊이 3에서 그리고 집합의 갯수가 네개인 섹터 캐시에서는 스택 깊이 2에서 적중된다. 또한 블럭 크기 2^3에서 1a의 데이타 참조는 한 블럭당 서브 블럭이 1개인 경우에 블럭 3 서브 블럭 0 참조로 바뀌고 이때의 해당 서브

블럭의 유효 레벨은 표 3의 (b)에서 알수 있는 바와 같이 집합의 갯수가 한개인 섹터 캐시에서는 스택 깊이 8에서 적중되고 집합의 갯수가 두개인 섹터 캐시에서는 스택 깊이 3에서 그리고 집합의 갯수가 네개인 섹터 캐시에서는 스택 깊이 2에서 적중된다. 블럭 크기가 24,25인 경우도 블럭 크기가 2^3인 경우와 마찬가지로 계산될수 있다.

집합 매핑 함수가 set-refinement조건을 만족할 때 집합의 갯수,블럭 크기,associativity, 서브 블럭 크기가 다른 섹터 캐시 메모리 모델의 실패율을 구하기 위하여 제안된 시뮬레이션 알고리즘은 다음 그림 3과 같다.

L = 집합 매핑 함수들의 수

f1(x),...,fL(x) = 임의의 집합매핑 함수

SB = 최대 서브 블럭크기

sb = 최소 서브 블럭크기

max_subblocks = block/sb

M = log2(SB) - log2(sb) + 1

N = 메모리 reference들의 수

max_assoc =시뮬레이션할 섹터 캐시 메모리들의 최대 associativity

distance [1:L,1:max_assoc,1:diff_blocks,1:M] = 집합 매핑 함수, associativity , 블럭 크기, 서브 블럭 크기에 따른 섹터 캐시 메모리의 실패율을 계산하기 위해 사용되는 어레이(array)

above [1:L] = 해당 어드레스 태그가 저장된 스택의 깊이를 저장

스택노드의 데이타 구조{

정수 tag

정수 vl [L] [M] [max_subblock]

}

Nunique = 메모리 reference들중 unique한 블럭들의 수

stacknodes [1:diff_blocks,1:O(Nunique)] = 집합 매핑 함수,블럭크기에 따른 스택모델의 top을 나타내는 어레이

```
For each reference x {
    integer above[1:L] — distance counters for x
    read (var x)
    N++
    found=FALSE
    for(blocksize=0;((blocksize(diff blocks)&&(NOTfound));blocksize++)
    {
      /* 매 블럭 크기에서 해당 블럭이 발견될 때까지 순방향으로 해당 블럭을 찾음 */
      for i=1 to L {above[i]=0}
      node pointer=stack[blocksize,f1(x)>>blocksize)]
      while ((NOT found) AND (node pointer!=NULL)) {
      /* 블럭의 스택 깊이를 구하기 위해 순방향으로 all-associativity 시뮬레이션 */
          y=node pointer->block number
          if (x==y) {
              found=TRUE
              above[L]++
          }
          else {
              match = FALSE
              for i=L down to 1 or match {
                  if (fi(x)==fi(y)){
                      match = TRUE
                      above[i]++
                  }
              }
              node pointer=node pointer->next
          }
      }
      if (found) {
          total above = 0
          for i=L down to 1{
      /* 해당 블럭이 적중되었기 때문에 각 집합 매핑 함수 별로 해당 블럭의 스택 깊이를 해당 블럭내의 각 서브 블럭의 유효레벨에 적용 */
              total above = total above + above[i]
              for j = 1 to max sub blocks
              {
                  if((total above+1) > node pointer->vl[i][M][j])
                      node pointer->vl[i][M][j]=total above+1
              }
      /* 앞에서 계산된 각 서브 블럭의 유효 레벨을 기준으로 유효레벨 inclusion개념을 사용하여 서브 블럭 크기가 다른 경우의 유효 레벨 계산 */
              for j = (M-1) downto 1
                  for k = 1 to (1<<(j)
                      node pointer->vl[i][j][k] =
                          min(node pointer->vl[i][j+1][2*k
                              ],node pointer->vl[i][j+1][2*k+1])
          }
      }
for i = 1 to L
      {
      /* 앞 단계에서 계산된 유효레벨을 사용하여 각 섹터 캐시 메모리 모델의 실패율을 계산하기 위해 distance 어레이 계산 */
          for j = M to 1
          {
              sub block = gblocksize,j(x)
              hitlevel = node_pointer->vl[i][j][sub_block]
              if(hitlevel <= cachep->max assoc)
                  distance[i,hitlevel,blocksize,j]++
          }
          sub block = gblocksize,M(x)
          node pointer->vl[i][M][sub block] = 1
      }
      UPDATE(x,stack number,found,node pointer,blocksize)
      if(blocksize != 0)
          stack[blocksize-1,pre set].next->nextlevel =
                  stack[blocksize,f1(x)>>blocksize].next
      pre set = f1(x)>>blocksize)
  }
  node pointer= stack[blocksize-1,f1(x)>>blocksize].next->nextlevel
  for(i = blocksize; i<(diff blocks); i++){
      /* 블럭이 스택내에서 발견된 이후의 블럭 크기에서 역방향으로 시뮬레이션 */
      for j=1 to L {above[j]=0}
      while (node pointer->pre !=NULL) {
      /* 블럭의 스택 깊이를 구하기 위해 역방향으로 all-associativity 시뮬레이션 */
          y=node pointer->block number
          match = FALSE
          for j=L down to 1 or match {
              if (fj(x)==fj(y)){
                  match = TRUE
                  above[j]++
              }
          }
          node pointer=node pointer->pre
      }
      total above = 0
      for j=L down to 1{
          total above = total above + above[j]
      /* 각 집합 매핑 함수 별로 해당 블럭의 스택 깊이를 해당 블럭내의
         각 서브 블럭의 유효레벨에 적용 */
          for k = 1 to max sub blocks
          {
              if((total above+1) > node pointer->vl[j][M][k])
                  node pointer->vl[j][M][k]=total above+1
          }
      /* 앞에서 계산된 각 서브 블럭의 유효 레벨을 기준으로 유효레벨 inclusion개념을 사용하여 서브 블럭 크기가 다른 경우의 유효 레벨 계산 */
          for k = (M-1) downto 1
              for l = 1 to (1<<(k)
                  node pointer->vl[j][k][l] =
                      min(node pointer->vl[j][k+1][2*l
                          ],node pointer->vl[j][k+1][2*l+1])
```

```
        for j = 1 to L
        {
        /* 앞 단계에서 계산된 유효레벨을 사용하여 각 섹터 캐시 메모리 모델의
           실패율을 계산하기 위해 distance 어레이 계산 */
           for k = M to 1
           {
              sub block = gblocksize,k(x)
              hitlevel = node pointer->vl [j] [k] [sub block]
              if(hitlevel <= cachep->max assoc)
                 distance [j,hitlevel,blocksize,k] ++
           }
           sub block = gblocksize,M(x)
           node pointer->vl [j] [M] [sub block] = 1
        }
        UPDATE(x,stack number,found,node pointer,i)
        node pointer = node pointer->nextlevel
     }
  }
```

그림 3. 집합의 갯수, 블럭 크기, associativity, 서브 블럭 크기가 서로 다른 섹터 캐시 메모리의 실패율을 구하는 시뮬레이션 알고리즘

Fig. 3. Simulation algorithm for set-associative sector caches with various sub-blocks, associativities, blocks and number of sets.

특히 본 알고리즘은 trace-driven 시뮬레이션 시간의 30-50%를 차지하는 입력 어드레스의 읽음을 단 한 번으로서 블럭 크기, 집합의 갯수, associativity및 서브 블럭 크기가 서로 다른 섹터 캐시 모델을 동시에 시뮬레이션한다.

Ⅳ. 라이트백 섹터 캐시 메모리의 버스 트래픽을 구하는 시뮬레이션 알고리즘

CPU가 메모리에 데이타를 write할 경우에는 주기억장치와 캐시 메모리의 내용을 동일하게 유지하여야 한다. 이러한 방법으로 캐시메모리는 라이트백과 write-through를 사용한다. 라이트백 캐시 메모리는 write-through 캐시 메모리보다 버스 트래픽을 줄일 수 있기 때문에 일반적으로 메모리 bandwidth가 제한되는 경우에는 라이트백 캐시 메모리를 사용한다. 메모리 엑세스가 N이라고 할때, 라이트백 캐시 메모리의 버스 트래픽을 구하기 위한 일반식은 다음 식(1)과 같다.

$$TR(C,Bc) = [mr(C) + mw(C)*Wf + f(C) + dp(C)]*Bc/N*Bp \quad (1)$$

여기서 mr(C)는 크기 C인 캐시 메모리에서 실패를 발생시키는 read reference들의 갯수이고 mw(C)는 실패를 발생시키는 write reference들의 갯수이다. Wf는 만약 write 페치가 사용되면 1이고 그렇지 않으면 0이며, f(C)는 프리페치의 수이다. Bp는 프로세서가 메모리 엑세스시 전달되는 바이트 양이고, Bc는 캐시 메모리내에 한 블럭의 바이트 양이다. 그리고 dp(C)는 크기 C의 캐시메모리로 부터 push되어지는 dirty블럭의 갯수이다.

섹터 캐시 메모리에서는 블럭 단위로 대치되지 않고, 서브 블럭단위로 대치되기 때문에 라이트백 섹터 캐시 메모리의 버스 트래픽을 구하기 위한 일반식은 다음식(2)와 같이 된다.

$$TR(C,Bs) = [mr(C) + mw(C)*Wf + f(C) + dp(C)]*Bs/N*Bp \quad (2)$$

여기서 Bs는 캐시 메모리내에 한 서브 블럭의 바이트 양이다.

만약 write 페치를 사용하고, 프리페치방식을 사용하지 않는다면 식(2)는 다음 식 (3)과 같이 된다.

$$\begin{aligned} TR(C,Bs) &= [mr(C) + mw(C) + dp(C)]*Bs/N*Bp \\ &= [m(C) + dp(C)]*Bs/N*Bp \end{aligned} \quad (3)$$

라이트백 섹터 캐시 메모리의 copy-back효과를 계산하기 위하여 J.G.Thompson은 각 서브 블럭 블럭에다가 dirty 비트 대신에 dirty레벨을 사용함으로써 라이트백 섹터 캐시메모리의 스택분석이 가능하도록 하였다. 여기서 dirty레벨은 그 서브 블럭이 속한 블럭이 dirty인 가장 작은 메모리를 나타낸다. 즉 이것은 그 서브 블럭이 마지막으로 write된후에 그 서브 블럭이 속한 블럭이 push된 스택내에 가장 작은 레벨이다. 그 서브 블럭이 write된 적이 없다면, 그 서브 블럭의 dirty 레벨은 무한대가 된다. 따라서 J.G.Thompson은 dirty레벨을 사용해서 각 스택에서 블럭의 매 스택 깊이(stack depth)마다 그 스택깊이에서 라이트백을 필요로 하지 않는 서브 블럭의 갯수를 write-avoid [스택깊이] 에 저장함으로써 다음과 같이 dp(C)를 계산한다. 식(4)에서 W는 모든 write reference들의 갯수이고 wa(i)는 associativity가 i인 섹터 캐시 메모리에서 write-back을 필요로 하지않는 write reference들의 갯수이다.그리고 D(C)는 크기 C인 섹터 캐시 메모리에서 스택레벨이 C보다 크지 않은 블럭에 속하고, 유효레벨과 dirty레벨이 C보다 크지않은 서브 블럭들의 갯수이다.

$$dp(C) = W - \sum_{i=1}^{C} wa(i) - |D(C)| \quad (4)$$

여기서 dp(C)는 write의 총 갯수에서 라이트백을 필요로 하지 않는 갯수를 빼는 것이다. 그러나 J.G.Thompson[6]이 제안한 방법은 associativity가 변하는 섹터 캐시에만 적용이 가능한 단점이 있다.

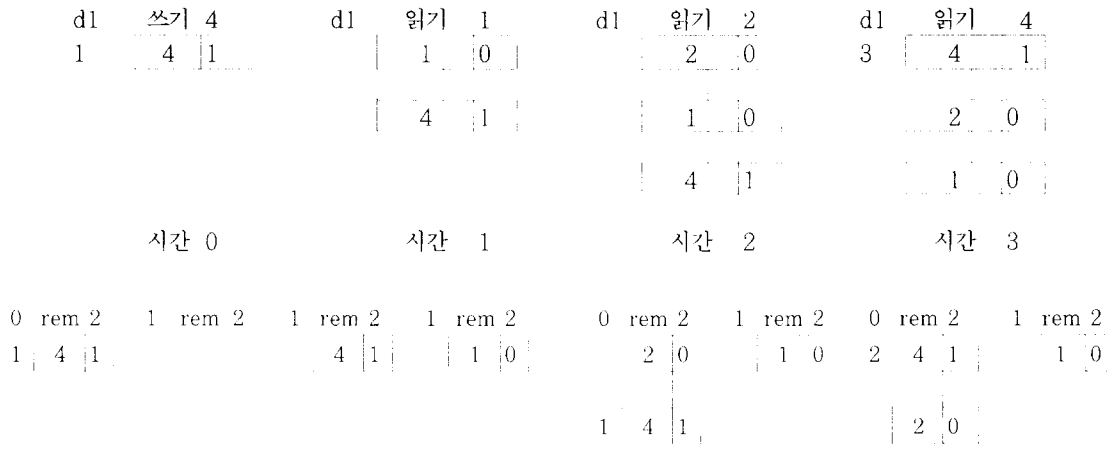

그림 4. 집합의 갯수가 서로 다른 섹터 캐시 메모리의 dirty 레벨의 예
Fig. 4. An example of dirty level of sector cache memory with various number of sets.

그리고 집합의 갯수가 달라질 경우 각 블럭의 dirty 레벨도 달라진다. 그 예는 그림 4와 같다. 집합의 갯수가 한개인 경우에는 시간 3에서 블럭 4의 dirty레벨이 3이 되나, 집합의 갯수가 두개인 경우에 시간 3에서 블럭 4의 dirty레벨은 2가 된다. 이와같은 개념을 사용하여 Wang [7]은 Thompson의 개념을 확장해서 집합의 갯수및 associativity가 서로 다른 각 경우마다 dirty 레벨을 사용함으로써 집합의 갯수및, associativity가 서로 다른 라이트백 캐시 모델의 버스 트래픽을 구할 수 있는 시뮬레이션 알고리즘을 제안하였다. 그러나 이 알고리즘도 블럭 크기가 서로 다른 여러 캐시 모델의 버스 트래픽을 동시에 구할 수 없는 단점을 갖는다. 이러한 단점을 개선하기 위하여 블럭 크기가 서로 다른 캐시 메모리의 버스 트래픽을 동시에 구하는 알고리즘이 개발되었다. [8] 그러나 이 알고리즘도 서브 블럭을 고려하지 않았기 때문에 섹터 캐시 메모리에는 적용할수 없는 단점이 있다.

이러한 단점을 개선하기 위하여 본 논문에서는 Thompson의 알고리즘을 확장해서 집합의 갯수, 블럭 크기, associativity및 서브 블럭크기가 서로 다른 섹터 캐시모델에서 라이트백 섹터 캐시 메모리의 버스 트래픽을 동시에 구할 수 있는 알고리즘을 제안 한다.

Set-associative 라이트백 섹터 캐시 메모리의 버스 트래픽을 구하기 위하여 J.G.Thompson이 제안한 방법에서 어떤 한 서브 블럭에 대한 dirty 레벨은 각 스택에서 그 서브 블럭이 발견되는 스택 깊이에 따라 변화하기 때문에, 그 서브 블럭이 속한 블럭의 스택 깊이를 구할 수 있다면 각 경우에 dirty 레벨을 둠으로써 버스 트래픽 계산을 위한 스택 분석의 확장이 가능하다. 따라서 블럭 크기, 서브 블럭 크기, associativity및 집합의 갯수가 서로 다른 라이트백 섹터 캐시 메모리의 버스 트래픽을 구하기 위하여 J.G. Thompson의 알고리즘을 확장해서 블럭 크기, 서브 블럭 크기, associativity및 집합의 갯수가 서로 다른 섹터 캐시 모델에서 각 경우마다 dirty 레벨을 두고, III장에서 제안한 알고리즘을 사용하여 해당 서브 블럭이 속한 블럭의 스택 깊이를 구함으로써 라이트백 섹터 캐시 메모리의 버스 트래픽을 쉽게 구할 수 있도록 한다.

블럭 크기, 서브 블럭 크기, associativity및 집합의 갯수가 서로 다른 라이트백 섹터 캐시 메모리의 버스 트래픽을 동시에 구하기 위한 시뮬레이션 알고리즘은 그림 5와 같다.

스택노드의 데이타 구조{
    정수 tag
    정수 vl[L][M][max_subblock]
    정수 dl[L][M][max_subblock]
}

write_avoid[1:L,1:max_assoc,1:diff_blocks,1:M] = 집합 매핑 함수, associativity , 블럭 크기, 서브 블럭 크기에 따른 섹터 캐시 메모리의 버스 트래픽을 계산하기 위해 사용되는 어레이

```
For each reference x {
    integer above[1:L] -- distance counters for x
    read (var x)
    N++
    found=FALSE
```

```
for(blocksize=0;((blocksize<diff blocks)&&(NOTfound));blocksize++)
 {
    for i=1 to L {above[i]=0}
    node pointer=stack[blocksize,f1(x)>>blocksize)]
    while ((NOT found) AND) (node pointer!=NULL)) {
       y=node pointer->block number
       if (x==y) {
          found=TRUE
          above[L]++
       }
       else {
          match = FALSE
          for i=L down to 1 or match {
             if (fi(x)==fi(y)){
                match = TRUE
                above[i]++
             }
          }
          node pointer=node_pointer->next
       }
    }
    if (found) {
       total above = 0
       for i=L down to 1{
          total above = total above + above[i]
          for j = 1 to max sub blocks
          {
             if((total above+1) > node pointer->vl[i][M][j])
                   node pointer->vl[i][M][j]=total above+1
/*  각 집합 매핑 함수 별로 해당 블럭의 스택 깊이를 해당 블럭내의
    각 서브 블럭의 dirty레벨에 적용 */
             if(total above+1) > node pointer->dl[i][M][j])
                   node pointer->dl[i][M][j]=total above+1
          }
          for j = (M-1) downto 1
          {
             for k = 1 to (1<<j)
             {
               node pointer->vl[i][j][k] =
                 min(node pointer->vl[i][j+1][2*k],
                        node pointer->vl[i][j+1][2*k+1])
/* 앞에서 계산된 각 서브 블럭의 dirty 레벨을 기준으로 dirty레벨 inclusion
   개념을 사용하여 서브 블럭 크기가 다른 경우의 dirty 레벨 계산 */
               node pointer->dl[i][j][k] =
                 min(node pointer->dl[i][j+1][2*k],
 node pointer->dl[i][j+1][2*k+1])
             }
          }
       }
       for i = 1 to L
       {
          for j = M to 1
          {
             sub block = gblocksize,j(x)
             hitlevel = node pointer->vl[i][j][sub block]
             if(hitlevel <= cachep->max assoc)
                   distance[i,hitlevel,blocksize,j]++
          }
          sub block = gblocksize,M(x)
          node pointer->vl[i][M][sub block] = 1
       }
       UPDATE(x,stack_number,found,node_pointer,blocksize)
       if(blocksize != 0)
          stack[blocksize-1,pre set].next->nextlevel =
                   stack[blocksize,f1(x)>>blocksize)].next
       pre set = f1(x)>>blocksize)
       if( x'accesstype == WRITE ){
       /* 데이타 참조가 write인 경우에 앞 단계에서 계산된 dirty레벨을 사용하여 각
        라이트백 섹터 캐시 메모리의 버스 트래픽을 구하기 위해 write-avoid 어레이
           계산 */
          for i = 1 to L
          {
             for j = M to 1
             {
                sub block = gblocksize,j(x)
                walevel = node pointer->dl[i][j][sub block]
                if(walevel <= cachep->max assoc)
                }
             sub block = gblocksize,M(x)
             node pointer->dl[i][M][sub block] = 1
          }
       }
    }
node pointer= stack[blocksize-1,f1(x)>>blocksize)].next->nextlevel
for(i = blocksize; i<diff blocks; i++){
    for j=1 to L {above[j]=0}
    while (node pointer->pre !=NULL) {
       y=node pointer->block_number
       match = FALSE
       for j=L down to 1 or match {
          if (fj(x)==fj(y)){
             match = TRUE
             above[j]++
          }
       }
       node pointer=node pointer->pre
    }
    total above = 0
    for j=L down to 1{
       total above = total above + above[j]
       for k = 1 to max sub blocks
       {
          if((total above+1) > node pointer->vl[j][M][k])
             node pointer->vl[j][M][k]=total above+1
/*  각 집합 매핑 함수 별로 해당 블럭의 스택 깊이를 해당 블럭내의
    각 서브 블럭의 dirty레벨에 적용 */
          if((total above+1) > node pointer->dl[j][M][k])
             node pointer->dl[j][M][k]=total above+1
       }
       for k = (M-1) downto 1
       {
          for l = 1 to (1<<k)
          {
             node pointer->vl[j][k][l] =
                min(node pointer->vl[j][k+1][2*l],node  pointer->v[j][k+1][2*l+1])
/* 앞에서 계산된 각 서브 블럭의 dirty 레벨을 기준으로 dirty레벨 inclusion
   개념을 사용하여 서브 블럭 크기가 다른 경우의 dirty 레벨 계산 */
             node pointer->dl[j][k][l] =
                min(node_pointer->dl[j][k+1][2*l],
  node_pointer->dl[j][k+1][2*l+1])
          }
       }
    }
    for j = 1 to L
    {
       for k = M to 1
       {
          sub block = gblocksize,k(x)
          hitlevel = node pointer->vl[j][k][sub block]
          if(hitlevel <= cachep->max assoc)
             distance[j,hitlevel,blocksize,k]++
       }
       sub block = gblocksize,M(x),
       node pointer->vl[j][M][sub block] = 1
    }
    UPDATE(x,stack_number,found,node_pointer,i)
    node pointer = node pointer->nextlevel
    if( x'accesstype == WRITE ){
    /* 데이타 참조가 write인 경우에 앞 단계에서 계산된 dirty레벨을 사용하여 각
     라이트백 섹터 캐시 메모리의 버스 트래픽을 구하기 위해 write-avoid 어레이
        계산 */
       for j = 1 to L
       {
          for k = M to 1
          {
             sub block = gblocksize,k(x)
             walevel = node pointer->dl[j][k][sub block]
             if(walevel <= cachep->max assoc)
                write avoid[j,walevel,blocksize,k]++
          }
          sub block = gblocksize,M(x)
          node pointer->dl[j][M][sub block] = 1
       }
    }
 }
```

그림 5. 집합의 갯수, 블럭 크기, associativity, 서브 블럭 크기가 서로 다른 라이트백 섹터 캐시 메모리의 버스 트래픽을 구하기 위한 알고리즘

Fig. 5. Simulation algorithm for set-associative write-back sector caches with various sub-blocks, associativities, blocks and number of sets.

V. 시뮬레이션 실행결과의 비교및 검토

본 논문에서는 다중 프로그래밍(multiprogram-

ming) 효과를 고려하기 위하여 다음과 같은 4 개의 응용 프로그램을 실행시키면서 얻은 trace를 25000개씩 round-robin방식으로 선택해서 만든 트레이스를 시뮬레이션의 입력으로 사용한다.[10]

응용 프로그램은 다음과 같다.

1) SIM 트레이스 : 명령어 레벨 시뮬레이터 (Instruction Level Simulator)인 응용 프로그램
2) HLML 트레이스 : 기계 독립적인 고급 마이크로 프로그래밍 언어.
3) SWEET 트레이스 : 게이트 레벨 시뮬레이터
4) PLAMIN 트레이스 : PLA Minimizer 프로그램.

응용 프로그램의 어드레스 트레이스는 VAX 아키텍쳐(architecture)내의 control bit인 T bit를 사용하여 매 명령 실행마다 trap을 발생시켜서 trap이 받아들여졌을때 각 명령과 관련된 어드레스를 기록함으로써 얻는다. 트레이스가 얻어진 환경은 VAX-11/750(UNIX 4.3 BSD)상에서 얻어졌다.[13]

Thompson의 알고리즘을 확장시켜 유효레벨 inclusion개념을 사용한 알고리즘[4]이 실현된 시뮬레이터와 본 논문에서 제안된 알고리즘이 실현된 시뮬레이터의 CPU시간은 다음 표 5,표 6과 같으며, trace-driven 시뮬레이션은 SUN workstation (UNIX 4.3 BSD)상에서 실행하였다.

표 5,6에서 블럭크기는 2^3부터 2^5이며, 매 블럭 크기에서 집합의 갯수는 2^6 부터 2^{10}개 이고 한 블럭당 서브 블럭의 갯수는 1, 2, 4개인 섹터 캐시들을 대상으로 하였다.

실패율을 구하기 위하여 Trace-driven 시뮬레이션을 실행한 결과 본 논문에서 제안한 시뮬레이션 알고리즘의 수행시간이 기존 시뮬레이션 알고리즘[4]의 수행시간보다 약 74.37% 감소됨을 알 수 있다.

표 5. 섹터 캐시 메모리의 실패율을 구하기 위한 Trace-driven 시뮬레이션 실행 시간

Table 5. Running times of trace-driven simulation for miss ratios of sector caches.

실행 어드레스 갯수	실행 시간	제안한 알고리즘	기존의 알고리즘[4]	실행시간 감소비율
50000		12.61(SEC)	55.65(SEC)	77.34%
100000		28.97(SEC)	95.93(SEC)	69.80%
150000		35.26(SEC)	138.01(SEC)	74.45%
200000		46.80(SEC)	186.03(SEC)	74.84%
250000		58.70(SEC)	237.11(SEC)	75.24%
300000		70.33(SEC)	276.33(SEC)	74.55%

또한 버스 트래픽을 구하기 위하여 Trace-driven 시뮬레이션을 실행한 결과 본 논문에서 제안한 시뮬레이션 알고리즘의 수행시간이 기존의 시뮬레이션 알고리즘[4]의 수행시간보다 약 66.72% 감소됨을 알 수 있다.

표 6. 섹터 캐시 메모리의 버스 트래픽을 구하기 위한 Trace-driven 시뮬레이션 실행 시간

Table 6. Running Times of Trace-Driven Simulation for Bus Traffics of Sector Caches.

실행 어드레스 갯수	실행 시간	제안한 알고리즘	기존의 알고리즘[4]	실행시간 감소비율
50000		18.88(SEC)	57.99(SEC)	67.44%
100000		33.10(SEC)	100.10(SEC)	66.93%
150000		48.10(SEC)	143.68(SEC)	66.52%
200000		63.63(SEC)	189.23(SEC)	66.37%
250000		82.22(SEC)	246.72(SEC)	66.67%
300000		96.65(SEC)	287.51(SEC)	66.38%

VI. 결 론

본 논문에서는 블럭 크기, associativity, 서브 블럭 크기및 집합의 갯수가 서로 다른 여러 섹터 캐시 모델의 실패율과 버스 트래픽을 동시에 구하는 시뮬레이션 알고리즘을 제안하였다.

제안된 알고리즘은 섹터 캐시 메모리내의 블럭 적중과 실패가 일반 캐시 메모리의 참조와 동일하다는 성질을 사용하여 블럭 크기가 서로 다른 캐시 메모리에 적용될 수 있는 시뮬레이션 알고리즘[8]을 사용하여 블럭 크기및 집합의 갯수가 서로 다른 섹터 캐시 메모리내의 블럭 적중과 실패를 결정하고, 블럭 적중시에 해당 블럭이 유효한 최소 스택 깊이를 구한다. 그 다음 섹터 캐시 메모리의 각 서브 블럭마다 각 집합 매핑 함수에 관한 유효레벨을 두고, 유효 레벨에 앞 단계의 결과를 적용함으로써, 블럭 크기및 집합의 갯수가 서로 다른 경우에 섹터 캐시 메모리의 해당 서브 블럭이 유효한 최소 레벨을 결정한다. 이 결과를 기준으로 서브 블럭의 유효레벨 inclusion개념[4]을 사용하여 서브 블럭이 서로 다른 경우에 서브 블럭의 실패와 적중을 결정함으로써 블럭 크기, 집합의 갯수, associativity및 서브 블럭 크기가 서로 다른 섹터 캐시 메모리의 실패율을 동시에 구할 수 있었다. 또한 제안된 알고리즘은 associativity, 서브 블럭 크기,집합의 갯수및 블럭 크기가 서로 다른 각 경우마다 dirty 레벨을 두고,매 write 어드레스 트레이스마다 dirty 레벨을 변경시킴으로써 블럭 크기, associativity, 서브 블럭 크기및 집합의 갯수가 서로 다른 라이트백 섹터 캐시 메모리의 버스 트래픽을 구할 수 있었다.

제안된 알고리즘을 C언어로서 실현하여 다양한 예제 프로그램 실행시 얻어진 어드레스 트레이스를 입력으로 trace-driven시뮬레이션을 실행한 결과, 그 실행시간이 Thompson의 알고리즘을 확장시켜 유효레벨 inclusion개념을 사용한 알고리즘[4]의 실행 시간에 비해 실패율을 구하는 알고리즘은 약 74 % 정도, 버스 트래픽을 구하는 알고리즘은 약 66 %정도 감소됨을 확인하였다.

제안된 trace-driven 시뮬레이션 알고리즘은 빠른 시간내에 섹터 캐시 메모리의 성능을 평가함으로써 섹터 캐시 메모리를 설계할때 유용하게 사용되리라 기대된다.

참 고 문 헌

[1] M.D.Hill, "Aspects of Cache Memory and Instruction Buffer Performance", UC Berkeley CS Division Technical Report UCB/CSD 87/381, November 1987.

[2] M.D.Hill, A.J.Smith,"Evaluating Associativity in CPU Caches", IEEE Transaction on Computers, Vol.38, no. 12, pp. 1612-1630, December 1989.

[3] M. D. Hill and a. J. Smith, "Experimental Evaluation of On-Chip Microprocessor Cache Memories", Proc. 11th International Symposium on computer Architecture, Ann Arbor, MI, June. 1984.

[4] 박동규,임인칠, "섹터 캐쉬 메모리의 Trace-Driven 시뮬레이션 알고리듬 ",전자공학회논문지 제 28권 B편 제 3 호 pp. 159-169, 3, 1991

[5] R.L.Mattson, J.Gecsei, D.R. Slutz and I.L.Traiger, "Evaluation techniques for storage hierarchies", IBM Systems Journal, Vol 9, no 2, 1970.

[6] J.G.Thompson, A.J.Smith, " Efficient (Stack) Algorithms for Analysis of Write-Back and Sector Memories ", UC Berkeley CS Division Technical Report UCB/CSD 87/358, June 1987.

[7] W.H.Wang, J.L.Baer, "Efficient Trace-Driven Simulation Methods for Cache Performance Analysis" ACM Super-Computing 1990.

[8] 박동규,임인칠, "캐쉬 메모리의 Trace-Driven 시뮬레이션 알고리듬 ", 전자공학회논문지 제 28권 B편 제 2 호 pp. 74-83, 2, 1991

[9] A.J.Smith, "Cache Memories", ACM Comput. Surveys,vol.14, pp. 473-530, September 1982.

[10] D.B.Alpert, M.J.Flynn, "Performance Trade-offs for Micorprocessor Cache Memories", IEEE MICRO , pp. 44-53, August 1988.

[11] J.L.Hennessy, D.A.Patterson, "COMPUTER ARCHITECTURE A QUANTITATIVE APPROACH ", MORGAN KAUFMANN 1990.

[12] A.Agarwal, R. L.Sites, M. Horowitz, "ATUM: A New Technique for Capturing Address Traces Using Micro- code", Proc. of 13th International Symposium on Computer Architecture, June 1986.

[13] 박동규, 허용민, 유광석, 임인칠, "Cache Memory Performance Evaluation에 관한 연구", 한국전자통신연구소, 최종보고서, 4, 1989

저 자 소 개

朴 東 圭(正會員)

1962년생. 1985년 2월 한양대학교 공과대학 전자공학과 졸업. 1988년 2월 한양대학교 대학원 전자공학과 공학석사학위 취득. 1992년 2월 한양대학교 대학원 전자공학과 공학박사학위 취득. 1992년 3월 ~ 현재 순천향대학교 정보통신공학과 조교수. 주관심 분야는 컴퓨터 시스템 성능평가, 컴퓨터 통신등임