런 타임에서의 캐쉬 라인 크기 선택

정삼기° 이인환

한양대학교 전자통신컴퓨터공학과

skjung@csl.hanyang.ac.kr° ihlee@hanyang.ac.kr

Dynamically Varing Cache Line Size in Merged DRAM/Logic LSIs

Samki Jung° Inhwan Lee

Dept. of Electronics and Computer Engineering, Hanyang University

요 약

DRAM과 고밀도집적회로가 병합된 시스템에서는 메모리와 프로세서간에 넓은 대역폭을 갖을 수 있다. 이런 조건에서 넓은 대역폭을 효율적으로 이용할 수 있는 D-VLS(Dynamically Variable Line Size) 캐쉬가 제안되었다. D-VLS 캐쉬는 프로그램이 실행 되면서 그 프로그램의 특성을 추적하며 적절한 캐쉬 라인 사이즈를 선택함으로써 시스템 성능향상을 목표로 한다. 본 논문에서는 D-VLS 캐쉬에서 캐쉬 라인 사이즈를 결정하는 알고리즘을 개선하고자 한다. 개선된 알고리즘을 적용한 결과 기존의 D-VLS 캐쉬보다 평균 메모리 접근 시간이 3.73% 정도 향상되었다.

1. 서 론

DRAM과 고밀도집적회로가 융합된, 즉 프로세서와 주 메모리가 집적화된 시스템에서는 일반적인 시스템에서 나타나는 기술적 제약점을 해소할 수 있는 많은 이점이 있다[1]. 특히, 대역폭이 넓은 온 칩 구조에서는 우선적으로 성능향상을 기대할 수 있다. 왜냐하면 프로세서와 메모리 사이의 데이터 전송능력을 넓은 대역폭을 이용해 효율적으로 향상시킬 수 있기 때문이다. 예를 들어 캐쉬 적중 실패(cache misses)가 되었을 때 한 번에 프로세서와 메모리 사이에 존재하는 대역폭 전체를 이용해서 프로세서가 요구하는 데이터를 전송할 수 있다[2][3][4].

이와 같이 대역폭이 넓어진 만큼 캐쉬 라인 사이즈도 커진다면 데이터를 미리 적재할 수 있는 효과가 있다. 하지만 프로그램의 공간적인 집약성이 충분하지 않고 캐쉬 적중 실패가 자주 발생한다면 오히려 시스템 성능을 저하시키는 단점이 있다. 이러한 조건일 때 캐쉬 연관성(cache associativity)을 높혀주게 되면 캐쉬 적중 실패를 줄일 수 있게 된다. 그러나 캐쉬 연관성이 높아짐에 따라 캐쉬 접근 시간(cache access time)은 늘어나게 되는 취약점이 있다[5].

이러한 복잡한 문제점을 해소하기 위해서 D-VLS 캐쉬라는 개념이 제시되었다[6]. Koji Inoue가 제안한 D-VLS 캐쉬는 프로세서가 캐쉬를 접근한 최근에 관찰된 유형을 바탕으로 적절한 캐쉬 라인 사이즈를 결정한다. 따라서 프로그램의 특성을 추적하면서 최소 크기의 캐쉬 라인 사이즈에서부터 중간 크기, 최대 크기까지 변경하게 한다. 이와 같은 선택은 추가된 하드웨어에서 하게 된다. 추가적으로 명령어 집합을 변경할 필요가 없으니 프로그램의 호환성은 고려하지 않아도 된다.

본 논문은 Koji Inoue가 제시한 캐쉬 라인 사이즈 선택 알고리즘을 개선하여 프로그램의 특성을 효율적으로 해석함으로써 보다 향상된 시스템 성능을 산출하는 것이 목표이다.본 논문에서 LSD(Line Size Determinter)를 개선한 알고리즘을 S-LSD(Sam-LSD)라 명칭한다.

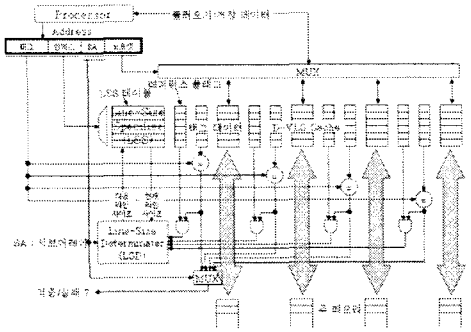


그림 1 직접 사상 D-VLS 캐쉬 구조

2. D-VLS 캐쉬의 소개

본 장에서는 D-VLS 캐쉬의 동작 원리와 LSD 알고리즘에 대해서 논하겠다. D-VLS 캐쉬에서 사용되는 용어는 Koji Inoue 논문을 참고했다[6].

2.1 D-VLS 캐쉬의 동작 원리

그림 1은 D-VLS 캐쉬 구조도이다. 프로세서에 의해서 생성된 어드레스는 128바이트 라인 크기를 가진 직접 사상 캐쉬가 동작하는 것처럼 태그, 인덱스, 오프셋 영역으로 분리된다. 따라서 인덱스에 의해서 어느 128바이트 라인을 가리킨다. 캐쉬 섹터라 불리는 128바이트 라인은 32바이트 크기인 서브라인 4개로 이루어져 있다. 각각의 서브라인마다 태그 영역과 레퍼런스 프래

그를 가지고 있다. 프로세서는 128바이트의 캐쉬 섹터를 접근하는 것이 아니고 캐쉬 섹터 중 단 하나의 서브라인을 참조한다. 그 서브라인은 레퍼런스 서브라인이라 불리고 레퍼런스 서브라인을 포함한 캐쉬 섹터는 레퍼런스 섹터라 한다. 프로세서가 접근을 하면 접근한 서브라인의 레퍼런스 플래그는 1의 값을 갖으며 레퍼런tm 섹터내에 각각의 서브라인이 갖고 있는 태그값들과 어드레스에 의해 생성된 태그값을 동시에 비교한다. 4비트의 태그비교 결과값들 중 서브어레이 값에 의해 선택된 단 하나의 비트값으로 캐쉬 적중(cache hit)인지 아닌지 결정된다. 서브어레이는 캐쉬 섹터 중 동일한 위치에 있는 서브라인들의 집합을 말한다. 위치에 따라 서브어레이 0부터 서브어레이 3번으로 불려진다.

캐쉬 적중이 되면 프로세서 명령어에 따라 레퍼런스 서브라인을 대상으로 데이터를 읽거나 쓸 것이다. 그렇지 않다면 LSS(Line Size Specifier) 테이블 내에서 레퍼런스 섹터와 동일한 위치에 있는 라인 크기에 대한 정보에 따라 메모리로부터 캐쉬 교체가 이루어진다. 교체가 된 서브라인의 레퍼런스 플래그는 0이 된다.

캐쉬 적중 여부를 알아보기 위해 구해진 4비트의 태그비교값과 레퍼런스 섹터에 있는 4개의 레퍼런스 플래그를 비트 연산한 결과는 근접 서브라인이라 불리며 LSD의 입력값으로 사용된다.

2.2 캐쉬와 메모리 사이에 데이터가 이동할 수 있는 세 가지 종류의 라인 크기

캐쉬 라인 크기는 최소 크기, 중간 크기, 최대 크기로 나눠진다. 32바이트 크기를 갖는 최소 크기 라인은 서브어레이에 따라 네 가지 전송 경우가 있다. 64바이트 크기를 갖는 중간 크기 라인은 서브어레이 0번과 1번, 서브어레이 2번과 3번으로 전송하는 단 두 가지 경우만 존재한다. 128바이트 크기를 갖는 최대 크기 라인은 하나의 캐쉬 섹터를 변경할 수 있다.

2.3 LSD 알고리즘

LSD 알고리즘은 LSS 테이블로부터 읽은 현재 라인 크기를 그림 2에서처럼 세 가지 라인 크기 중 하나를 상태전이도의 시작점으로 선정한다. 프로세서가 처음으로 캐쉬에 접근할 때는 LSS테이블로부터 최대 라인 크기값을 초기값으로 읽어온다. 그 다음 레퍼런스 서브라인과 근접 서브라인의 위치와 개수에 따라 다른 라인 크기로 이동을 할 수 있는지를 판단한다. 새롭게 결정된 라인 크기값은 LSS 테이블 내에 현재 라인 크기를 읽은 위치에 기록된다.

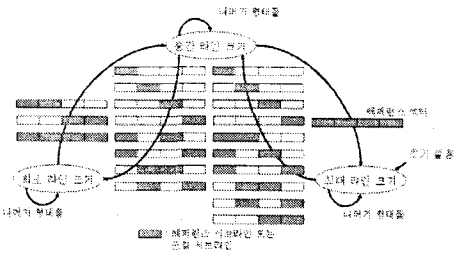


그림 2 상태전이도

3. S-LSD 알고리즘

본 장에서는 LSD와 S-LSD의 알고리즘 차이점과 동작원리 그리고 기대되는 효과에 대해서 논하겠다.

3.1 LSD와 S-LSD 알고리즘의 차이점

LSD 알고리즘은 LSS 테이블에서 읽은 현재 라인 크기값을 상태전이도의 시작점으로 한 반면에 S-LSD에서는 레퍼런스 서브라인의 값으로부터 알고리즘이 시작된다는 차이점이 있다. 따라서 현재 라인 입력값을 사용하지 않는다. 또한 알고리즘의 초기값을 필요로 하지 않는다. 왜냐하면 프로그램이 시작되면 어느 레퍼런스 서브라인이 선택될지는 예측하기 어려우나 레퍼런스 서브라인은 반드시 존재하게 되므로 자연스럽게 S-LSD 알고리즘은 시작할 수 있다.

다른 차이점은 LSD 알고리즘에서는 최소 라인 크기에서 최대 라인 크기로, 최대 라인 크기에서 최소 라인 크기로 한 번에 이동하지 않고 반드시 중간 라인 크기를 거쳐서 조건이 만족할 때만 변경된다. 이러한 조건은 프로그램의 특징을 따라가는데 신속한 대응을 하지 못한다고 생각하기에 S-LSD 알고리즘은 입력된 레퍼런스 섹터 값에 따라 조건이 만족하면 중간 라인 크기를 거치지 않고 최대 라인 크기나 최소 라인 크기로 선택 가능하게 하였다.

3.2 S-LSD 동작원리

그림 3의 (가)는 레퍼런스 서브라인이 첫 번째에 위치할 경우 레퍼런스 섹터의 상태와 상관없이 무조건 최대 크기 라인으로 결정한다. 이는 프로그램의 집약성이 높다고 기대하기 때문이다. 또한 이전에 서브어레이 0을 참조하였고 다시 동일한 레퍼런스 섹터가 접근되고 레퍼런스 서브라인이 1을 참조했을 때 캐쉬 적중 실패가 발생된다면 최대 라인 크기로 데이터가 교체될 것이다. 이는 새로운 프로그램의 흐름을 충분히 따라갈 수 있을 것이다.

그림 3의 (나)는 레퍼런스 서브라인이 두 번째를 참조한 경우 라인 크기가 어떻게 선택되는 과정이 나타나 있다. 래퍼런스 서브라인과 근접 서브라인의 개수가 4개일 때에는 최대 크기 라인을 선택한다. 왜냐하면 프로그램의 집약성이 높다는 것을 의미하기 때문이다. 래퍼런스 서브라인과 근접 서브라인이 서브어레이 0과 1을 차지하고 서브어레이 3을 선택되지 않았다면 중간 크기 라인으로 변경된다. 이는 프로그램의 집약성이 줄어들었다고 판단되기 때문이다. 나머지 형태들은 최소 크기 라인을 선택한다.

그림 3의 (다)는 그림 3의 (나)와 비슷하다. 우선 최대 크기 라인을 선택하는 과정은 동일하다. 다만 서브어레이 1과 2가 레퍼런스 섹터에서 자리를 차지하고 있고

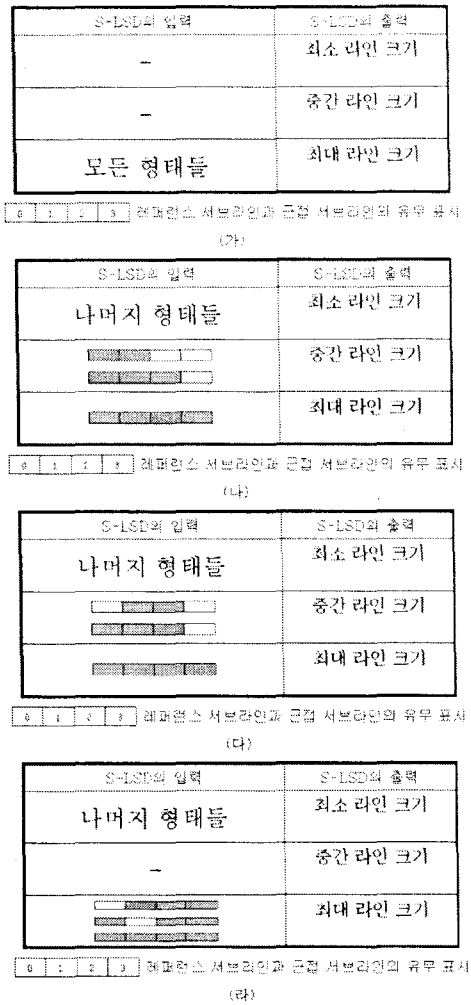


그림 3 S-LSD 알고리즘

서브어레이 3은 선택되지 않을 때만 중간 크기 라인으로 변경된다. 이런 과정의 이유도 서브어레이 1을 참조한 경우 때와 마찬가지다. 그리고 나머지 형태들은 최소 크기 라인으로 이동된다.

그림 3의 (라)를 보면 최대 크기 라인으로 변경될 수 있는 경우 세 가지를 제외하면 모두 최소 크기 라인으로 결정된다. 이는 서브어레이 3을 접근 후 서브어레이 0을 선택할 것을 예상한 것이다.

4. 성능 실험 및 결과 분석

본 장에서는 캐쉬 적중 실패율과 평균 메모리접근 시간을 구하기 위한 실험 환경 및 조건에 대해서 논한다.

4.1 실험 환경

캐쉬 적중 실패율을 구하기 위해서 Simplescalar를 사용해서 실험을 하였다[7]. Simplescalar에는 있는 여러 가지 실험 실행 파일 중 캐쉬 적중 실패율을 측정할 수 있는 sim-cache 도구를 선정했다. LSD와 D-LSD를 적용하기 위해 각각 수정된 sim-cache 이용해 실험을 하였다.

실험시 SPECint95[8][9]와 Mibench[10]을 성능 평가 프로그램으로 선정하였다. SPECint95는 정수연산 프로그램이고 Mibench는 임베디드 시스템을 위한 것이다.

실험 대상은 Inoue, Sam이란 이름으로 정했다. Inoue는 Koji Inoue가 제시한 D-VLS 캐쉬를 의미하고 Sam은 S-LSD가 적용된 D-VLS 캐쉬를 나타낸다. 본 실험에서 사용된 단일 캐쉬의 크기는 모두 16KB이다.

4.2 하드웨어 비용

Koji Inoue의 논문을 보면 실험 대상들의 하드웨어 비용을 알 수 있다[6]. S-LSD는 현재 라인 크기를 입력으로 사용하지 않으므로 LSD과 비교해서 트랜지스터의 개수가 변경될 수 있지만 다른 비용에 비해 크기가 작으므로 무시될 수 있다. S-LSD의 하드웨어는 고정된 캐쉬 라인 사이즈가 32바이트, 128바이트인 캐쉬보다 각각 약 32%, 22% 정도의 추가 비용이 요구된다.

4.3 평균 메모리접근 시간

평균 메모리 접근 시간은 캐쉬 성능 평가에 일반적으로 사용된다[6].

*평균 메모리 접근 시간 = 캐쉬 적중 시간 + 캐쉬 적중 실패율 * 캐쉬 적중 실패 부하* (1)

*= 캐쉬 적중 시간 + 캐쉬 적중 실패율 * 2 * (DRAMstup + 라인크기/대역폭)* (2)

캐쉬 교체 정책이 라이트 백(Write back)이라면 메모리에 2 번 접근을 해야할 것이다. 메모리 접근은 메모리 접근을 위한 지연시간(DRAMstup)과 주 메모리와 캐쉬 사이의 데이터 전송시간(라인크기/대역폭)로 나눠질 수 있다. 그리고 각각은 일정한 값을 유지한다. 따라서 식(1)에서 식(2)로 변경이 가능하다. 캐쉬 적중 시간은 3.476 ns이다[6]. S-LSD은 캐쉬 적중 시간과 무관하기 때문이다. 그리고 DRAMstup, 라인크기/대역폭을 각각 40ns, 10ns로 가정한다.

4.4 표준화된 캐쉬 적중 실패율 결과 및 분석

실험에 사용된 전체 30개의 프로그램들 중에서 25에서 LSD보다 S-LSD가 낮은 캐쉬 적중 실패율을 보였다. 즉, LSD보다 5.29% 향상되었다. SPECint95에서는 2.54%, Mibench에서는 8.42% 정도 개선되었으므로 S-LSD는 SPECint보다 Mibench에 효율적하다고 판단된다.

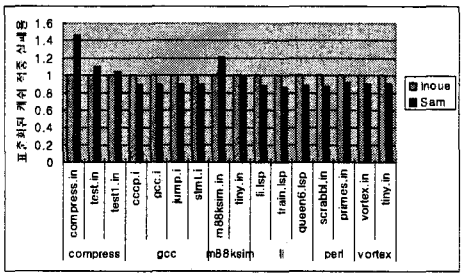
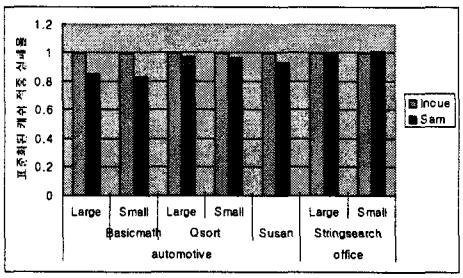


그림 4 SPECint95의 표준화된 캐쉬 적중 실패율



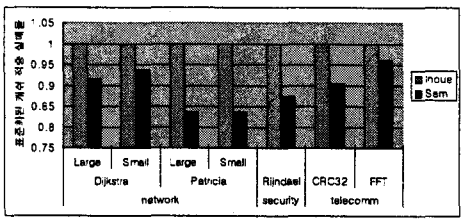


그림 5 Mibench의 표준화된 캐쉬 적중 실패율

그림 4와 그림 5를 살펴보면, SPECint95에서 compress 프로그램과 m88ksim,in을 입력파일로 하는 m88ksim 프로그램, Mibench에서 large를 입력파일로 하는 Stringsearch 프로그램을 제외한 나머지 프로그램에서 S-LSD가 우수함을 보여주고 있다. 또한 small을 입력 파일로 하는 Dijkstra 프로그램에서 가장 좋은 결과를 보인다. 이러한 결과가 산출되게된 이유는 S-LSD에서 서브어레이 값을 이용해 라인 크기를 결정하는 것이 프로그램의 특성을 파악하는데 결정적인 역할을 한다고 분석된다.

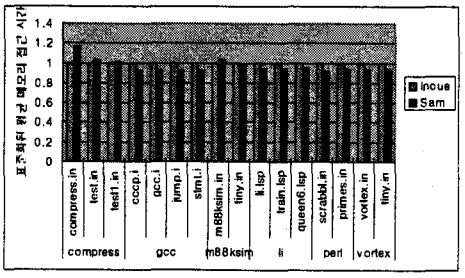
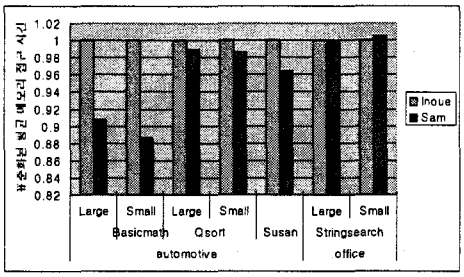


그림 6 SPECint95의 표준화된 평균 메모리 접근 시간



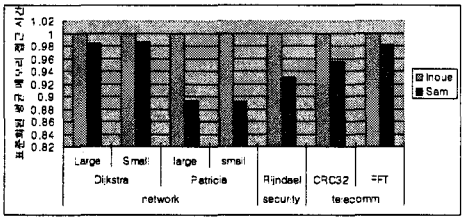


그림 7 MIbench의 표준화된 평균 메모리 접근 시간

4.5 표준화된 평균 메모리 접근 시간 결과 및 분석

전체 30개의 프로그램들 중에서 25개 프로그램에서 LSD보다 S-LSD가 낮은 평균 메모리 접근 시간을 보였다. 즉, LSD보다 3.73% 향상되었다. SPECint95에서는 2.98%, Mibench에서는 4.59% 정도 개선되었으므로 S-LSD는 SPECint보다 Mibench에 효율적하다고 판단된다.

그림 6과 그림7을 살펴보면, 표준화된 캐쉬 적중 실패와 마찬가지로 SPECint95에서 compress 프로그램과 m88ksim,in을 입력파일로 하는 m88ksim 프로그램, Mibench에서 large를 입력파일로 하는 프로그램을 제외한 나머지 프로그램에서 S-LSD가 우수함을 보여주고 있다. 또한 small을 입력 파일로 하는 Basicmath 프로그램에서 가장 좋은 결과를 보인다. 이번 평균 메모리 접근 시간 결과에서도 S-LSD가 좋은 결과가 나온 이유는 평균 메모리 접근 시간은 캐쉬 적중 실패율에 의해서 값이 결정되기 때문이다.

5. 결론

프로세서와 주 메모리가 하나의 칩으로 구현된 시스템에서 캐쉬와 메모리 사이에 넓은 대역폭을 효율적으로 이용해서 전체적인 시스템 성능 향상을 기대하고자 D-VLS 캐쉬가 제안되었다. 본 논문은 S-LSD 알고리즘을 제안해서 성능을 개선시키고자 했었다.

S-LSD는 Koji Inoue가 제안한 LSD 보다는 캐쉬 적중 실패율에서는 5.29%, 평균 메모리 접근 시간에서는 3.73%의 성능 향상이 있었다.

앞으로 8개의 캐쉬 섹터마다 라인 크기 정보를 저장할 수 있는 LSS 테이블을 대상으로 S-LSD 알고리즘을 검증할 것이고, 메모리와 프로세서간의 데이터 이동에 따른 전력소모를 측정할 것이다.

참고 문헌

[1]Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., and Yelick, K., "A Case For Intelligent RAM," IEEE Micro, vol.17, no.2, pp.34-44, March/April 1997.

[2]Murakami, K., Shirakawa, S., and Miyajima, H., "Parallel Processing RAM Chip with 256Mb DRAM and Quad Processors," ISSCC Digest of Technical Papers, pp.228-229, Feb. 1997

[3]Saulsbury, A., Pong, F., and Nowatzyk, A., "Missing the Memory Wall: The Case for Processor/Memory Integration," Proc. of the 23rd Annual International Symposium on Computer Architecture, pp.90-101, May 1996

[4]Wilson, K. M. and Olukotun, K., "Designing High Bandwidth On-Chip Caches," Proc. of the 24th Annual International Symposium on Computer Architecture, pp.121-132, June 1997

[5]Hill, M. D., "A Case for Direct-mapped Caches," IEEE Computer, vol.21, no.12, pp.25-40, Dec. 1998.

[6]Inoue, K., Koji, K., and Murakami, K., "Dynamically Variable Line-Size Cache Architecture for Merged DRAM/Logic LSIs," IEICE Transactions on Electronics, Vol.E83-D, No.5, pp.1048-1057, May 2000

[7]SimpleScalar Microarchitecture Simulator. http://www.simplescalar.com

[8] 순천향대학교 정보기술공학부 2001년 컴퓨터시스템 설계론 1 강의 자료 http://sjlee.sch.ac.kr/arch/cs-01/csystem-grad-01.html

[9]SPECint95 http://www.wizpark.co.kr/wizpark/WizFileView.php?seq=952054&View=1

[10] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. "MiBench: A free, commercially representative embedded benchmark suite." IEEE 4th Annual Workshop on Workload Characterization, December 2001. http://www.eecs.umich.edu/mibench