

임베디드 시스템을 위한 메모리 서브시스템 파라미터의 자동 검출

(Automatic Detection of Memory
Subsystem Parameters for
Embedded Systems)

하태준[†] 서상민^{††}

(Taejun Ha) (Sangmin Seo)

전보성^{††} 이재진^{†††}

(Posung Chun) (Jaejin Lee)

요약 임베디드 시스템에서 프로그램 성능을 향상시키기 위해서는 시스템의 하드웨어를 이해하고 활용하는 것이 중요하다. 특히 메모리 서브시스템에 대한 이해는 프로그램을 주어진 하드웨어에 최적화하여 성능을 향상시키는 데 큰 역할을 한다. 본 논문에서는 cache, TLB, DRAM과 같은 메모리 서브시스템의 파라미터를 자동적으로 검출하는 기존의 알고리즘을 임베디드 시스템에 적용해 보고, 새롭게 메모리 뱅크 개수 검출 알고리즘을 제안한다. 제안한 알고리즘은 실제 여러 가지 임베디드 시스템 환경에서 실험을 통해 검증하였고, 실험 결과 메모리 서브시스템의 파라미터를 정확히 검출해 낼 수 있는 것을 확인하였다.

- 본 연구는 정보통신부 및 정보통신연구진흥원의 IT신성장동력핵심기술개발사업(2006-S-040-01, Flash Memory 기반 임베디드 멀티미디어 소프트웨어 기술개발)과 한국학술진흥재단의 BK21 사업의 일환으로 수행하였습니다. 또한 이 연구를 위해 연구장비를 지원하고 공간을 제공한 서울대학교 컴퓨터 연구소에 감사드립니다.
- 이 논문은 2008 한국컴퓨터종합학술대회에서 '임베디드 시스템을 위한 메모리 서브시스템 파라미터의 자동 검출 기법'의 제목으로 발표된 논문을 확장한 것임

[†] 정회원 : Tmax Soft 연구소 전임
www381@naver.com

^{††} 학생회원 : 서울대학교 컴퓨터공학부
sangmin@aces.snu.ac.kr
posung@aces.snu.ac.kr

^{†††} 종신회원 : 서울대학교 컴퓨터공학부 교수
jlee@aces.snu.ac.kr

논문접수 : 2008년 8월 28일
심사완료 : 2009년 3월 3일

Copyright©2009 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.

정보과학회논문지 : 컴퓨팅의 실제 및 레터 제15권 제5호(2009.5)

키워드 : 임베디드 시스템, 메모리 뱅크, 하드웨어 파라미터 측정 프로그램

Abstract To optimize the performance of software programs, it is important to know certain hardware parameters such as the CPU speed, the cache size, the number of TLB entries, and the parameters of the memory subsystem. There exist several ways to obtain the values of various hardware parameters. Firstly, the values can be taken from the hardware manual. Secondly, the parameters can be obtained by calling functions provided by the operating systems. Finally, hardware detection programs can find the desired values. Such programs are usually executed on PC or server systems and report the CPU speed, the cache size, the number of TLB entries, and so on. However, they do not sufficiently detect the parameters of one of the most important parts of the computer concerning performance, namely the memory bank layout in the memory subsystem. In this paper, we present an algorithm to detect the memory bank parameters. We run an implementation of our algorithm on various embedded systems and compare the detected values with the real hardware parameters. The results show that the presented algorithm detects the cache size, the number of TLB entries, and the memory bank layout with high accuracy.

Key words : embedded system, memory bank, hardware parameter detection

1. 서 론

컴퓨터 시스템은 각각의 용도에 따라 다양한 하드웨어를 사용하여 그 성능을 향상시켜 나가고 있다. 그리고 소프트웨어 역시 다양한 하드웨어의 환경에 적합한 프로그래밍을 통해 시스템의 성능을 향상시켜 나가고 있다. 이런 환경 아래에서 하드웨어 파라미터를 정확히 아는 것은 임베디드 시스템에서 무엇보다도 중요한 요소 중의 하나이다. 특히 성능에 가장 큰 영향을 미치는 메모리 서브시스템의 파라미터를 정확히 알 수 있다면 그에 따라 소프트웨어를 작성하거나 수정하여 성능을 향상시킬 수 있을 것이다. 예를 들어, 셀프 최적화(self-optimization)는 프로그램이 실행되는 하드웨어의 특성에 맞게 소프트웨어를 변경해서 성능을 극대화한다.

하드웨어의 파라미터를 확인하는 방법에는 여러 가지가 있다. 제조사에서 제공하는 문서를 통해 확인할 수 있고, 하드웨어의 파라미터들을 알려 주는 CPU의 특정 레지스터(register)를 읽어 확인할 수 있다. OS가 설치된 시스템이라면 OS가 레지스터를 읽어서 사용자에게 알려줄 수도 있다. 그리고 하드웨어 파라미터를 찾아낼 수 있는 소프트웨어를 사용할 수도 있다. 이 소프트웨어

들은 PC나 서버 시스템에서 CPU의 하드웨어 정보를 찾는 프로그램[1]을 시작으로 TLB와 캐시(cache)의 파라미터를 찾는 프로그램[2]으로 발전해 왔다. 그리고 SPEC 벤치마크와 같은 일반적인 벤치마크나 마이크로(micro) 벤치마크 프로그램을 수행시켜 하드웨어 파라미터를 찾는 소프트웨어도 등장했다[3,4].

본 연구에서는 하드웨어를 잘 모르는 사용자도 쉽게 하드웨어 파라미터를 알아낼 수 있도록 소프트웨어를 이용하는 방법에 집중한다. 특히 PC나 서버환경이 아닌, 임베디드 시스템에서 기본적인 하드웨어 파라미터를 구할 수 있는 알고리즘을 소개하고, 프로그램의 성능에 크게 영향을 미치는 메모리 뱅크(bank)의 개수를 찾아낼 수 있는 알고리즘을 제안한다.

본 논문의 구성은 다음과 같다. 2절에서는 메모리 서브시스템의 파라미터를 검출하는데 기본이 되는 알고리즘에 대해 살펴본다. 3절에서는 기존의 캐시와 TLB의 파라미터를 검출하는 기법을 임베디드 시스템에 적용하는 방법에 대해 살펴보고, 4절에서는 메모리 뱅크 검출 알고리즘을 새롭게 제안한다. 5절에서는 실험 결과 및 분석을 제시하고, 6절에서는 제안하는 방법의 유용성에 대해 설명한다. 마지막으로 7절에서 결론을 제시한다.

2. 메모리 파라미터 검출 알고리즘

메모리 파라미터를 검출하는 가장 기본적인 방법은 벤치마크 프로그램을 실행시켜 나타난 결과를 분석해서 값을 추측하는 것이다[1]. 벤치마크 프로그램은 메모리의 특정 주소에 값을 쓰거나 읽어들이고, 이때 걸리는 시간을 측정한다. 추가로 캐시 미스(miss), TLB 미스와 같은 값도 같이 측정한다. 벤치마크 프로그램을 통해 얻은 결과를 분석하여 하드웨어 파라미터를 결정한다[2].

벤치마크 프로그램은 OS로부터 메모리를 할당 받고, 그것을 배열에 할당한다. 이 배열의 크기를 N 으로 한다. 캐시의 크기는 C , 캐시 라인 크기는 b , 그리고 캐시 연관도(associativity)는 a 라고 한다. 미스가 발생하지 않을 때 걸리는 시간을 $T_{no-miss}$, 하나의 액세스에서 미스가 발생할 때 추가로 소모되는 시간을 M 이라 한다. 기본적인 액세스 패턴은 배열의 시작 주소로부터 배열의 마지막 주소까지 간격(stride) 단위로 점프하는 것이다.

이때 간격을 s 라고 하자. 위의 값들에 따라 결과는 표 1의 4가지 경우가 발생한다. 단, 이 결과는 초기화를 위한 첫 번째 수행을 제외한 나머지 실행 결과에 해당한다.

① 1의 경우 ($1 \leq N \leq C$)

초기화를 통해 배열을 메모리로부터 캐시에 가져오고 나면 이후에 발생하는 액세스는 모두 캐시 히트(hit)가 된다. 전체 수행시간은 캐시에서 데이터를 읽는 시간만 걸리게 된다.

표 1 배열 크기와 간격에 따른 캐시 미스와 액세스 시간

배열 크기	간격	캐시 미스	액세스 시간
$1 \leq N \leq C$	$1 \leq s \leq \frac{N}{2}$	0	$T_{no-miss}$
2.a	$C < N$	$1 \leq s < b$	b/s 원소마다 미스 $T_{no-miss} + M \times \frac{s}{b}$
2.b	$C < N$	$b \leq s < \frac{N}{a}$	항상 미스 $T_{no-miss} + M$
2.c	$C < N$	$\frac{N}{a} \leq s < \frac{N}{2}$	0 $T_{no-miss}$

② 2.a의 경우 ($C < N, 1 \leq s < b$)

전체 배열이 캐시에 들어갈 수 없고, 간격이 캐시 라인 크기보다 작아서 하나의 캐시 라인에 여러 개의 액세스가 발생할 수 있다. 하나의 캐시 라인에 대해 생각해 보면, 그 라인에 대한 첫 번째 액세스는 캐시 미스가 된다. 이때 메모리에서 캐시로 데이터를 가져오는데, 캐시 라인의 크기 단위로 데이터를 가져오게 된다. 이후 캐시 미스가 발생한 라인의 다른 데이터를 액세스하는 경우 이미 데이터가 캐시에 있으므로 캐시 히트가 발생한다. 이 경우 b/s 의 액세스 당 하나의 미스가 발생한다.

③ 2.b의 경우 ($C < N, b \leq s < N/a$)

모든 액세스에 대해서 캐시 미스가 발생한다. 처음에 쓰인 데이터가 재사용되기 전에 캐시 공간의 부족으로 킥아웃(kick-out)되어 버리고 그 위치에 다른 데이터가 쓰인다. 그리고 초기화가 끝난 후 다시 배열의 처음으로 돌아와 확인하면 캐시는 미스가 발생하게 된다. 간격이 라인 크기보다 크기 때문에 하나의 라인에 최대 한 번의 액세스만 발생한다. 수행시간은 모든 액세스가 미스가 발생한 것으로 계산한다.

④ 2.c의 경우 ($C < N, N/a \leq s < N/2$)

전체 메모리 액세스는 히트가 된다. 이 경우는 간격이 크기 때문에 액세스하는 데이터의 양이 세트의 크기 C/a 보다 적다. 따라서 킥아웃이 발생하지 않고 액세스는 모두 캐시 히트가 된다. 하나의 액세스에서 수행시간은 캐시 히트일 때 수행시간과 같다.

3. 캐시 및 TLB 파라미터 검출 기법

캐시와 TLB의 파라미터 검출 방법은 PC 환경에서 사용하는 방법[1,2]을 임베디드 시스템에 적합하게 변경하였다.

3.1 캐시

캐시 크기를 구하는 방법을 살펴보자. 캐시 히트의 경우 액세스 시간을 T 라고 하면 캐시 미스는 T 보다 훨씬 큰 값이 된다. 이는 메모리 액세스가 캐시 액세스보

다 훨씬 많은 시간이 걸리기 때문이다. 따라서 배열의 크기를 점차 증가시키면서 액세스 시간을 측정하다 시간이 느려지는 지점을 찾아서 그 배열의 크기를 캐시의 크기로 나타낼 수 있다.

캐시 라인 크기를 구하는 방법은 다음과 같다. 캐시는 메모리에서 데이터를 가져올 때 캐시 라인 단위로 데이터를 가져오기 때문에 특정 캐시 라인의 첫 번째 액세스는 캐시 미스이고 라인의 나머지 액세스는 히트가 된다. 따라서 초기화를 하지 않고, 간격을 1로 한 뒤 캐시보다 작은 크기의 배열을 액세스하면서 캐시 미스를 측정한다. 이때 두 개의 미스 사이에 몇 개의 히트가 발생하였는지를 측정하면 라인 크기를 정할 수 있다.

3.2 TLB

TLB의 경우 액세스 시간을 이용해서 값을 측정할 수 있다. 측정된 액세스 시간은 캐시와 TLB 양측에 영향을 받으므로 두 요소를 동시에 고려해야 한다. 예를 들어, 배열의 시작점에서 16KB 떨어진 원소에서 액세스 시간이 증가했다고 하면, 이것이 TLB의 영향인지 캐시의 영향인지를 구분해 낼 수 있어야 한다. TLB에 의한 시간 증가는 다음 16KB 지점에서 같은 양의 시간 증가를 할 것이고, 캐시는 그렇지 않을 것이다.

4. 메모리 뱅크 검출 알고리즘

임베디드 시스템 상에서 메모리 뱅크의 파라미터를 검출하는 알고리즘은 본 논문이 새롭게 제안하는 방법으로, 뱅크 인터리빙을 하는 경우와 하지 않는 경우로 나누어 알고리즘을 설명한다.

4.1 뱅크 인터리빙(Bank Interleaving)

메모리가 여러 개의 뱅크를 가진 경우, 뱅크 인터리빙을 사용하면 성능을 향상시킬 수 있다. 뱅크 인터리빙은 각 메모리 뱅크를 파이프라인처럼 이용하는 것이다.

뱅크를 검출하려면 메모리 액세스 시간이 가장 중요한 요소이나 메모리 액세스 시간에는 TLB, 캐시의 영향이 포함되어 있다. X-ray[4]와 같은 프로그램에서는 커널 모드로 동작하여 캐시를 비활성화(disable)하고 프로그램을 동작시킨다. 가장 확실한 방법은 독립형(stand-alone) 프로그램을 이용해서 다른 프로그램이 미칠 수 있는 영향을 최소화하는 것이다. 하지만 독립형 프로그램으로 제공하는 경우 사용자의 편의성이 떨어지므로, 처음 제시했던 논문의 목적과 맞지 않는다.

커널 프로그램으로 프로그램을 만들 경우 캐시를 비활성화할 수 있고, 액세스 시간과 캐시 미스, TLB 미스 외에도 다양한 값을 읽어 들일 수 있지만, 일반 사용자가 아닌 관리자의 권한으로 프로그램을 수행해야 하는 단점을 갖는다.

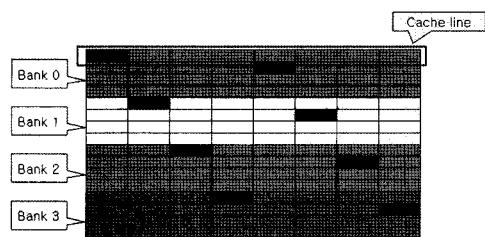


그림 1 뱅크 4개에 최적화된 액세스 패턴

4.2 뱅크 인터리빙하지 않는 경우

우선 n 개의 뱅크를 가진 메모리에 최적화된 성능을 보일 수 있는 액세스 패턴을 구한다. 이후 뱅크 번호 k 의 값을 1씩 증가시키며 실험결과를 측정하고 그 값을 분석함으로써 뱅크의 개수를 찾아낸다.

실험을 위해서 메모리 공간을 할당받고, 할당받은 메모리를 배열에 할당한다. 이 배열의 시작 주소를 S , 간격을 s (여기서 s 는 캐시 라인 크기 + $n \times$ 워드 크기)로 하고, 전체 배열의 크기를 A 라고 한다.

- ① 전체 메모리의 뱅크가 한 개라 가정하면, 모든 배열의 원소에 액세스할 때 같은 속도가 나온다.
- ② 전체 메모리의 뱅크가 두 개라 가정하면, 뱅크 0의 시작주소는 S 이고, 뱅크 1의 시작주소는 $S+A/2$ 이다. 따라서 두 개의 뱅크를 번갈아 가며 액세스 하는 것이 가장 빠르다. 그림 1의 경우는 메모리 뱅크가 4개 일 때 최적화된 액세스 패턴을 나타낸다.
- ③ 전체 메모리의 뱅크가 n 개이면, 뱅크 k 의 시작주소는 $S+A \times k/n$ 이다. 각각의 뱅크가 차례대로 액세스되는 것이 가장 빠른 속도를 가지게 된다.
- ④ 만약 뱅크가 4개일 때 뱅크 8개에 최적화된 알고리즘을 적용하면 뱅크 4개의 알고리즘보다 속도가 느려지게 된다. 이 경우 첫 번째 액세스와 두 번째 액세스의 시작주소는 S , $S+A \times 1/8$ 이다. $S+A \times 1/8$ 은 $S+A \times 1/4$ 보다 작아서 뱅크 0의 위치에 해당한다. 하나의 뱅크를 연속으로 두 번 액세스하므로 속도가 느려지게 된다. 따라서 k 를 1로 두고, 뱅크 k 에 최적화된 알고리즘을 수행하고, k 를 1씩 더할 때, 속도가 느려지는 지점을 찾는다. 속도가 느려질 때 k 의 값이 뱅크 개수가 된다. 뱅크 측정 시 캐시의 영향은 최대한 배제하여야 한다. 따라서 s 는 (캐시 라인 크기 + $n \times$ 워드 크기)로 두었다.

여기서 문제가 되는 부분은 운영체제가 있는 상황에서 가장 메모리의 연속적인 주소가 실제 메모리에서 연속적이지 않을 수 있다는 점이다. 이것을 해결하기 위해서는 유저 모드가 아니라, 관리자 권한으로 프로그램을 수행하고, 실제 메모리 주소를 이용하면 해결된다.

하지만, 뱅크 인터리빙을 사용하지 않는 경우는 한 페

이지 내에서 연속적인 메모리 접근이 있을 경우 한 뱅크에 모든 액세스를 하게 된다. 이 경우 성능저하가 발생하므로, 인터리빙을 사용하는 경우가 대부분을 차지한다.

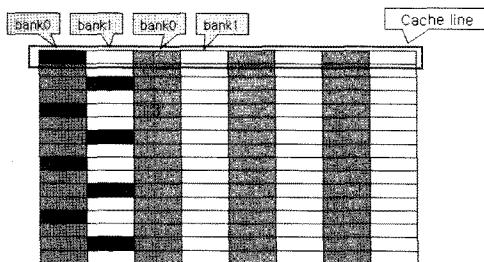


그림 2 뱅크 2개일 때 뱅크 2개에 최적화된 액세스

4.3 뱅크 인터리밍하는 경우

인터리빙을 사용하는 경우는 하나의 워드 단위로 뱅크가 배치되어 있다. 따라서 연속된 메모리를 액세스할 경우 가장 최적의 성능이 나오게 된다. 인터리빙을 사용하는 때도 TLB와 캐시, 프리패치의 영향을 최소화하며, 각 뱅크에 최적화된 알고리즘을 적용하면서 뱅크의 개수를 찾는다.

- ① 전체 메모리의 뱅크가 1개라고 가정하면 모든 영역에 대해서 같은 수행시간을 가진다.
 - ② 전체 메모리의 뱅크가 2개라고 가정하면 각 뱅크 영역이 차례대로 할당되어 있다. 차례대로 액세스하면 뱅크 2에 최적화된 액세스가 된다. 캐시의 영향을 최소화하려면 s 는 (캐시 라인 크기 $\times 2 +$ 워드 크기)가 된다. 단, 여기서 고려해야 할 점은 3번째 액세스가 $(S + 2 \times s)$ 가 아니라 $(S + 4 \times$ 캐시 라인 크기)여야 한다는 점이다. 그림 2를 보면 알고리즘은 3'이 아니라 3을 액세스한다. 실제로 3의 액세스와 3'의 액세스는 수행시간이 같게 나온다. 하지만, 뱅크 2개에 최적화된 알고리즘과 뱅크 4개에 최적화된 알고리즘을 구분하고자 이런 패턴으로 테스트하였다.
 - ③ 전체 메모리의 뱅크가 4개라고 가정하면 각 뱅크를 차례대로 액세스하는 것이 최적화된 알고리즘이다. 단, 위에서 설명한 것과 마찬가지로 5번째 액세스는 캐시라인의 첫 번째 워드를 액세스한다.
 - ④ 뱅크 k 개에 최적화된 알고리즘은 k 개의 원소를 차례대로 액세스하고 $k + 1$ 에서 캐시 라인의 처음으로 돌아와서 액세스한다. 이렇게 되면 k 개의 뱅크를 차례대로 사용하게 되어서 가장 빠른 테스트가 진행되게 된다. 인터리빙을 사용하지 않을 때는 예상 뱅크의 개수가 실제 뱅크의 개수보다 커지면 속도가 느려지는 현상이 발생하였는데 비해, 인터리빙을 사용하면 속도가 같게 나온다. 따라서 수행 시간이 같아

지는 지점이 뱅크의 개수를 나타낸다.

인터리빙을 하는 경우에는 캐시 라인 단위로 연속된 메모리 액세스가 이루어진다. 따라서 가장 메모리에 의해 연속된 페이지가 실제 물리 메모리상에 분리되어 있다고 하더라도 하나의 페이지는 연속적인 물리 메모리에 할당된다. 따라서 페이지 경계를 제외하고는 연속적인 물리 메모리 액세스가 이루어지기 때문에 올바른 결과가 도출된다. 따라서 인터리빙의 경우 유저모드로 프로그램을 수행하여도 올바른 결과를 유도해 낼 수 있다.

결과적으로 인터리빙을 사용하는 경우를 가정하고, 유저 모드로 프로그램을 수행하고, 결과가 뱅크가 1개로 나왔을 경우 관리자 모드로 인터리빙을 사용하지 않는 메모리의 뱅크축정 알고리즘을 수행하기로 한다.

5. 실험 결과

제안한 알고리즘의 정확성을 검증하기 위해 실제 임베디드 보드 상에서 캐시, TLB, 그리고 메모리 뱅크의 파라미터를 측정하였다. 실험 환경은 다음과 같다.

- CPU : FWPXA270C5 (520MHz)
 - SDRAM : K4S561632×2 (64MB)
 - OS : Embedded Linux Kernel 2.6.11
 - Hybus, XHyper270A embedded system

표 2는 캐시의 파라미터를 구하려고 앞에서 설명한 알고리즘대로 배열의 크기와 간격을 변화시키면서 실험한 결과이다. 실험 결과를 보면, 배열의 크기가 32KB가 되는 지점부터 시간의 증가가 발생하는 것을 볼 수 있다. 이를 통해 캐시의 크기를 32KB로 예상할 수 있다. 또, 간격이 32일 때와 64일 때 거의 같은 수행 시간을 보이는 것으로 보아, 표 1의 2.a와 2.b가 나누어지는 부분이라고 볼 수 있다. 따라서 캐시의 라인 크기는 32바이트라고 할 수 있다.

표 3은 TLB 파라미터를 검출하기 위한 실험 결과이다. 표 3의 결과를 살펴보면, 액세스 수가 32일 때 수행 시간이 이전보다 70배 정도 길어진 것을 알 수 있다. 이 지점이 처음 TLB 미스가 발생한 것이다. 따라서 우리가 사용할 수 있는 TLB 엔트리(entry)는 32개 미만이라는 것을 알 수 있다. 또 32개의 데이터를 액세스하는 경우라도 간격이 4128보다 작은 경우 시간이 증가하지 않는 것을 볼 수 있다. 이것은 하나의 페이지에 두 번의 액세스가 발생하기 때문에 발생한 현상이다. 따라서 폐이지 크기는 4128보다는 작고, 2080보다는 크다고 예상 할 수 있다. 이 사이에 있는 2의 몇수는 4096이므로 폐이지 크기는 4KB이다.

표 4의 결과를 보면 뱅크 16개에 최적화된 알고리즘을 적용할 때, 수행 시간이 느려지는 것을 확인할 수 있다. 따라서 현재 메모리의 뱅크 개수는 8개라 할 수 있다.

표 2 캐시 파라미터의 실험 결과

배열 크기	간격	액세스 수	수행 시간(sec)
20480	64	320	1368
20480	32	640	1369
20480	16	1280	1368
20480	8	2560	1369
24576	64	384	1369
24576	32	768	1369
24576	16	1536	1369
24576	8	3072	1368
32768	64	512	1405
32768	32	1024	2746
32768	16	2048	2132
32768	8	4096	1795

표 3 TLB 파라미터 실험 결과

배열 크기	간격	액세스 수	수행 시간(sec)
164480	8224	20	1369
82560	4128	20	1371
41600	2080	20	1369
197376	8224	24	1371
99072	4128	24	1371
49920	2080	24	1368
263168	8224	32	74757
132096	4128	32	73650
66560	2080	32	1368
328960	8224	40	73679
165120	4128	40	73597
83200	2080	40	1370

표 4 메모리 뱅크 개수에 따른 수행 시간

뱅크0	뱅크1	뱅크2	뱅크4	뱅크8	뱅크16
36086	35289	34629	33250	32903	40582

표 5 메모리 파라미터 검출 결과

		PXA270		FWP270WBXX	
		실제값	실험값	실제값	실험값
Cache	size	32KB	32KB	32KB	32KB
	line	32B	32B	32B	32B
TLB	entry	32	24	32	32
	page size	4KB	4KB	4KB	4KB
메모리	bank	8개	8개	8개	8개
		ARM926EJ-S		PXA 255	
		실제값	실험값	실제값	실험값
Cache	size	32KB	32KB	32KB	32KB
	line	32B	32B	32B	32B
TLB	entry	64	40	32	24
	page size	4KB	4KB	4KB	4KB
메모리	bank	4개	4개	4개	4개

다음은 PXA270 보드뿐만 아니라 다른 세 개의 임베

디드 보드에서 실현한 결과를 실제 하드웨어 파라미터와 비교한 것이다. 표 5를 보면 TLB 엔트리를 제외한 값들은 실제 수치와 일치하는 결과를 얻었다. TLB 엔트리의 경우 리눅스에서 특정 페이지를 피닝(pinning)하는 경우가 있기 때문에 3개의 시스템에서 부정확한 결과가 나온 것으로 판단된다.

6. 유용성

본 논문에서 제안하는 방법은 특정 하드웨어 레지스터에 종속적인 방법이 아닌 일반적인 알고리즘에 기반을 둔 것이므로, 사용자의 입장에서는 하드웨어 파라미터 검출에 더 쉬운 접근 방법이 될 것이다. 특히 하드웨어 파라미터에 대한 기술문서를 볼 수 없는 경우나, 하드웨어 파라미터를 읽을 수 있는 레지스터가 없거나, 있더라도 레지스터에 접근할 수 없는 경우 유용하게 사용될 수 있다. 실제로 본 논문의 실험에서 사용한 4개의 보드에서는 모두 메모리 뱅크에 대한 정보는 하드웨어 레지스터를 통해 구할 수가 없었다.

7. 결 론

본 논문에서 다룬 주제는 크게 두 가지이다. 하나는 기존의 메모리 서브시스템 파라미터 검출 알고리즘이 임베디드 보드 상에서 정확히 동작하는지 확인하는 것이고, 다른 하나는 메모리 뱅크의 개수를 검출할 수 있는 알고리즘을 제안하고 검증하는 것이다. 실험 결과 캐시, TLB와 같은 기본적인 메모리 서브시스템의 파라미터는 운영체제가 실행 중인 임베디드 보드에서 만족할 만한 결과를 도출할 수 있었고 뱅크의 개수도 정확히 알아낼 수 있었다.

참 고 문 헌

- [1] Rafael H. Saavedra-Barrera, "CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking," Ph.D. Thesis, UC Berkeley, Tech. Rept. No. UCB/CSD-92-684, Feb. 1992.
- [2] Rafael H. Saavedra-Barrera, "Measuring Cache and TLB Performance and Their Effect on Benchmark Run Times," UC Berkeley, Tech. Rept. No. UCB/CSD-93-767, Aug. 1993.
- [3] J. Gee and A.J. Smith, "TLB performance of the SPEC benchmark suite," paper in preparation, draft of Jan. 1992.
- [4] Kamen Yotov, Keshav Pingali, and Paul Stodghill, "X-Ray: Automatic Measurement of Hardware Parameters," Cornell Univ., Tech. Rept. TR2004-1966, Oct. 2004.