# Scratch-Pad Memory를 위한 코드 변환 기법

문 대 경* 이 재 진
서울대학교 컴퓨터공학부
{daekyeong, jlee}@aces.snu.ac.kr

# Code Transformation Techniques for Scratch-Pad Memory

Daekyeong Moon* Jaejin Lee
School of Computer Science and Engineering
Seoul National University

**요 약**

전원을 전적으로 배터리에 의존하는 모바일 임베디드 시스템은 배터리 용량의 한계 때문에 효율적인 에너지의 사용이 매우 중요하다. 특히 memory subsystem은 전체 system에서 소모되는 에너지에서 큰 비중을 차지한다. 이 논문은 성능면에서 cache의 대안이 되고, cache보다 간단한 구조 때문에 전력소모가 훨씬 적은 on-chip scratch-pad memory(SPM)를 효율적으로 이용할 수 있는 소스 코드 변환 방법 및 SPM 관리 방법을 제안한다. 각 함수 단위로 코드 변환을 하며, 어떤 변수를 SPM에 할당하기 위한 소스코드 변환을 했을 때, 소스코드 분석만으로 알 수 있는 변수의 정적인 참조 횟수를 가중치로 고려하여, 코드 변환 후 메모리 참조에 의한 실행 시간과 에너지 소모를 계산하고 이를 바탕으로 SPM에 할당할 변수를 결정한 다음 실제 그 코드 변환을 적용한다. 제안된 코드 변환은 컴파일러에 의해 자동화 될 수 있다. 10개의 임베디드 벤치마크 프로그램을 이용하여 본 논문에서 제안하는 방법의 성능 평가를 한 결과, 실행 시간은 평균 23% 향상되고 에너지 소모는 평균 49% 감소함을 알 수 있다.

## 1. 서 론

일반적인 임베디드 시스템용 CPU에서 SRAM을 이용한 on-chip cache는 전체 chip 전력의 25%~45%를 소모한다. 이에 비하여, on-chip SRAM을 주소 지정이 가능한 scratch-pad memory (SPM)로 사용하면 같은 용량의 cache에 비해 34% 정도 작은 크기를 가지고 40% 정도 더 적은 전력을 소모한다[1]. 이는 SPM이 전력 소모 및 성능의 측면에서 cache의 대안이 될 수 있음을 보여준다.

하지만, 비용의 문제 때문에 SPM의 크기는 외부 DRAM보다 훨씬 작은, 수 byte에서 수 Kbyte정도이다. 따라서 프로그램의 성능을 높이면서 전력 소모를 줄이도록 SPM을 효율적으로 사용하는 것이 매우 중요하다. 이를 위하여 프로그램의 코드 부분을 SPM에 올려 실행하거나 데이터 부분을 SPM에 효율적으로 할당하기 위한 많은 연구가 있었다 [2,3,4,5,6,7,8,9,10]. 하지만 대부분의 경우 SPM에 할당 가능한 크기의 데이터만 고려하거나 [3,4,8,9], 배열과 같은 큰 데이터는 일부를 SPM에 할당하더라도 제한된 방법으로 이를 참조하는 정도였다[5]. 이렇게 크기만을 고려한 정적인 SPM 할당 방법은 사용가능한 SPM 영역이 남아 있더라도 할당하려고 하는 데이터 구조의 크기 때문에 이 영역을 잘 활용할 수 없는 문제가 발생하며, 이미 할당된 영역은 할당된 데이터가 더 이상 사용되지 않더라도 새로운 데이터를 위해 다시 사용할 수가 없다. Loop 변환(즉, loop blocking)을 이용한 SPM의 동적인 할당에 대한 연구가 있었으나 [3,6,7], 변환되는 loop에 대하여 DRAM과 SPM 간의 복사 비용만 고려하고 참조 되는 각각의 변수에 대한 가중치를 고려하지 않았기 때문에 효율적으로 데이터를 SPM에 할당하지 못한다. 또, 프로그램이 수행될 동안 계속 참조되는 작은 테이블과 같은 데이터는 동적 할당보다 프로그램의 시작부터 SPM에 할당하는 것이 좋다. [10]의 경우는 소스 코드의 변환이 아니라 프로그램의 stack 자체를 SPM에 할당하는 경우이다.

본 논문은 크기가 SPM보다 큰 배열이나 구조체를 SRAM과 SPM 간의 복사 비용뿐만 아니라 이들의 참조 비용을 함수 단위로 가중치로 고려하여 코드 변환을 하고, 이 코드 변환에 의해 SPM의 정적인 할당뿐만 아니라 동적인 할당을 통한 관리를 통해 코드의 실행 시간 및 에너지의 감소가 가능하다는 것을 보인다. 실험을 통하여 본 논문에서 제시된 방법이 프로그램의 실행시간을 평균 23%, 에너지 소모는 평균 49%까지 줄일 수 있다는 것을 보인다.

## 2. 코드 변환 방법

### 2.1 scratch-pad memory의 관리

SPM을 정적 또는 동적으로 관리하기 위하여 SPM의 처음 4 byte를 활용가능 영역 포인터로 사용한다. 이 포인터의 초기치는 SPM의 base 주소가 되며, 동적으로 SPM을 할당할 때마다 사용한 만큼 이 포인터를 증가시킨다. 호출되는 함수는 지역 변수에 SPM을 정적 또는 동적으로 할당하기 위하여 이 포인터를 이용한다. SPM을 사용한 함수에서 복귀할 경우, 이 포인터는 함수에서 사용한 SPM의 양 만큼 감소하며 이에 해당되는 코드가 함수의 끝에 삽입된다.

### 2.2 메모리 참조 비용 모델

한 개의 함수을 실행하는 동안 메모리 참조에 대한 시간 비용($T_{Access}$)과 에너지 비용($E_{Access}$)은 다음과 같은 수식으로 표현된다.

$$T_{Access} = N_{DRAM} * T_{DRAM} + N_{SRAM} * T_{SRAM} \quad (1)$$

$$E_{Access} = N_{DRAM} * E_{DRAM} + N_{SRAM} * E_{SRAM} \quad (2)$$

수식에서 $N_{DRAM}$과 $N_{SRAM}$은 각각 DRAM 및 SRAM을 참조하는 횟수를, $T_{DRAM}$과 $T_{SRAm}$은 DRAM 및 SRAM을 한번 참조할 때 걸리는 평균 시간을, $E_{DRAM}$과 $E_{SRAM}$은 DRAM 및 SRAM을 한번 참조할 때 걸리는 평균 에너지를 의미한다. 이 모델을 사용하여, 코드 변

환이 일어난 후 복사 비용을 포함하여 함수 단위로, 참조된 각 변수에 대한 정적인 비용을 계산하여 실행 시간 비용이 큰 것부터 SPM에 greedy하게 할당한다. 주로 배열 및 구조체를 나타내는 변수를 SPM에 할당한다.

### 2.3 코드 변환 및 SPM 할당

**배열의 정적인 할당.** 위에서 구한 실행시간 및 에너지 소모 비용 모델을 바탕으로, 현재 함수에서 활용 가능한 SPM의 공간이 할당하려하는 배열 변수보다 큰 경우는 정적으로 SPM을 그 변수에 할당한다. 즉, 그 변수의 선언을 활용 가능한 SPM 영역을 참조하는 포인터로 바꾼 다음 함수 내의 그 배열 변수에 대한 참조도 전부 이 포인터를 통한 참조로 바꾼다. 이 방법은 실행시 SPM과 DRAM간의 복사가 일어나지 않는다. 자주 참조되는 작은 크기의 테이블을 위한 배열 등이 이러한 방식으로 할당된다.

배열을 활용 가능한 SPM에 정적으로 모두 올릴 수 없는 경우는, 가능한 만큼만 정적으로 SPM에 올리고, 그 배열을 참조 할 때 index를 확인 하는 코드를 삽입하여 확인 결과에 따라 SPM이나 DRAM에서 참조하도록 한다 [5]. 하지만 배열을 loop를 이용하여 참조하는 경우는 매번 index 확인을 하는 것이 아니라 SPM에 올라간 부분을 참조하는 loop와 올리지 못한 부분을 참조하는 두개의 loop로 원래의 loop를 변환한다.

**배열의 동적인 할당.** Loop 내에서 참조 되는 배열의 원소들 간에 loop iteration을 통한 dependence가 없고 배열을 순차적으로 참조할 때는 원래 크기가 n인 배열을 활용 가능한 SPM에 할당 가능한 크기 n'인 배열로 나누어서 원래 loop에서 하던 작업을 크기가 n'인 배열에 대하여 $\lceil n/n' \rceil$ 만큼 반복 수행하며, 각 반복 수행마다 n'개의 원소를 DRAM에서 SPM으로 복사하고, 작업이 끝난 다음 SPM에서 DRAM으로 복사하는 코드를 삽입한다. 크기가 $(n-\lceil n/n' \rceil * n')$인 배열의 남아있는 부분에 대한 loop는 변환하지 않는다. 또, loop iteration을 통한 dependence가 존재하고 배열을 순차적으로 참조 할 경우는 원래 배열을 크기가 $n'+d$인 배열로 나누어서 앞의 경우와 마찬가지로 loop 변환을 한다. 여기서 d는 dependence distance이다. $S=\sum_{i=1}^{} A[i]$와 같은 꼴의 연산(reduction)을 수행하는 loop도 dependence가 없는 loop의 경우와 마찬가지로 배열을 SPM에 할당하고 loop를 변환 한다 [10].

**Loop Tiling (Blocking).** Matrix Multiplication등을 수행하는 loop는 tile의 크기가 현재 활용 가능한 SPM의 크기보다 작도록 loop tiling을 수행한다 [6,7,11]. 이 때, tile loop 앞에 loop가 참조하는 배열의 부분을 DRAM에서 SPM으로 복사하는 코드를 삽입하고, tile loop 종료 후에 SPM에 할당된 배열 부분을 DRAM으로 복사하는 코드를 삽입한다.

**구조체의 할당.** 크기가 큰 구조체도 배열과 마찬가지로 일부 field만 SPM 에 할당하는 것이 가능하다. 일부를 SPM에 할당한 다음 이들 field를 참조하는 원래의 코드는 SPM 영역을 참조하는 코드로 바꾸어 준다.

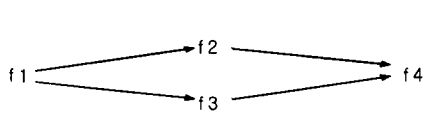


그림 1 정적 call graph

**배열이나 구조체의 주소가 함수의 인자로 전달될 때.** 그림 1의 정적 call graph를 생각하자. 배열이나 구조체가 f1에 선언되어 있고 그 주소가 f2와 f3를 통해서 f4로 전달되고, f4에서 중점적으로 참조 된다고 가정하자. 또, 배열이나 구조체는 활용 가능한 SPM의 크기 보다 작다고 가정한다. 이 경우 f4에서 그 배열이나 구조체에 SPM을 동적으로 할당하는 것보다 f1에서 정적으로 할당하는 것이 복사 비용을 줄일 수 있으므로 더 이득이다. 또, 배열의 크기가 커서 f4에서 동적인 할당을 할 경우라도 활용 가능한 SPM의 크기를 f4가 불릴 때 마다 알 수 있기 때문에 호출이 되는 경로와 무관하게 f4에서 SPM을 그 배열에 앞서와 같은 방법으로 동적으로 할당할 수 있다. 하지만 동적 할당은 복사 비용이 더 추가된다.

표 1 Memory subsystem에 관한 parameter들

| | seq. read | non-seq. read | seq. write | non-seq. write |
|---|---|---|---|---|
| DRAM | 120ns | 150ns | 120ns | 150ns |
| SRAM | 1ns | 1ns | 1ns | 1ns |

표 2 메모리 종류별 에너지 소모량

| 메모리 타입 | 메모리 크기 | 에너지 소모량 (nJ/4words) |
|---|---|---|
| SPM | 512B | 0.122063 |
| SPM | 1KB | 0.127542 |
| SPM | 2KB | 0.133936 |
| SPM | 4KB | 0.144965 |
| SDRAM(read) | 32MB | 70.2 |
| SDRAM(write) | 32MB | 51.6 |

표 3 실험에 사용한 임베디드 벤치마크 프로그램들

| 벤치마크 | 출처 | 벤치마크 | 출처 |
|---|---|---|---|
| G721 | MediaBench | DIJKSTRA | MiBench |
| ADPCM | MediaBench | CRC32 | MiBench |
| SHA | MiBench | BMM | Trimaran |
| FFT | MiBench | MM | Synthesized |
| BITCOUNT | MiBench | Bubble Sort | Synthesized |

## 3. 실험 환경

실험은 10개의 임베디드 벤치마크 프로그램들을 profiling 한 다음, hot spot이 되는 함수들에 대하여 본 논문의 코드 변환 방법을 수작업으로 적용하고, 실행시간과 메모리 참조 횟수를 구하여 변환하지 않은 원래 프로그램과 실행 시간 및 에너지 소모를 비교하였다. 프로그램의 실행에 ARMulator를 이용하였으며[12], CPU 모델은 unified cache를 가진 *100Mhz ARM720T*를 사용하였다. 여기서 프로그램의 코드 영역만 cacheable하게 설정하여 데이터영역은 cache에 올라가지 않도록 하여 cache의 효과를 배제 하였다. Memory subsystem에 관한 ARMulator의 parameter 설정은 표 1과 같다.

메모리 종류에 따른 에너지 소모량은 표 2과 같다. DRAM을 참조할 때의 에너지는 [13]의 결과를 사용하였다. 여기서 DRAM은 burst mode에서 동작하며 4개의 word가 한번에 전송된다. 실험에 사용된 SPM은 SRAM을 사용하였으며 크기를 각각 512B, 1KB, 2KB, 4KB로 설정하고 실험하였다. SPM의 에너지 소비 모델은 CACTI 3.2를 수정하여 계산하였다 [14]. 여기서 사용한 SPM의 모델은 CACTI에서 제시한 cache 모델에서 tag array의 word line, bit line, 및 sense amplifier 부분과 comparator 부분 및 valid output 부분을 제외한 것이다. SPM의 공정기술은 0.13μ을 사용하였고, block 크기는 DRAM의 burst mode와 맞추기 위해서 4 word로 설정 하였다.

실험에 사용한 임베디드 벤치마크 프로그램은 표 3과 같다. 이들

중 BMM은 double에 대한 연산을 integer에 대한 연산으로 코드를 수정해서 실험하였다. 그 외에 BubbleSort 및 MM (Matrix Multiplication)은 직접 구현한 것을 이용하여 실험하였다.

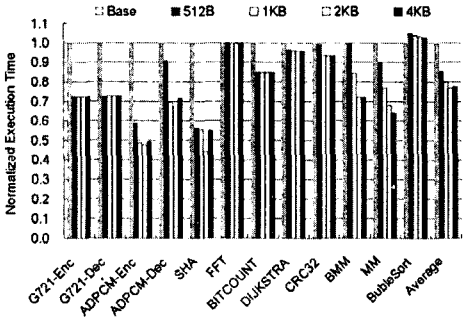


그림 2. 실행 시간

## 4. 성능 평가

### 4.1 실행 시간

실행 시간에 관한 결과는 그림 2에 나타나 있다. SPM을 사용하지 않을 때의 실행시간을 기준으로 SPM을 사용할 때의 실행시간을 정규화 하였다. 실행시간에 대한 실험 결과를 볼 때, SPM을 위해 코드 변환을 한 코드는 평균 23% 정도 실행 시간을 줄일 수 있다는 사실을 알 수 있다. FFT의 경우는 프로그램의 hot spot이 소프트웨어 부동소수점 연산 library이기 때문에 사용자 코드의 변환은 별 이득이 없으며, BubbleSort는 코드 변환에 의해 추가된 연산과 메모리 복사에 의해 CPU cycle을 더 소모하기 때문에 3~5% 정도 느려진다.

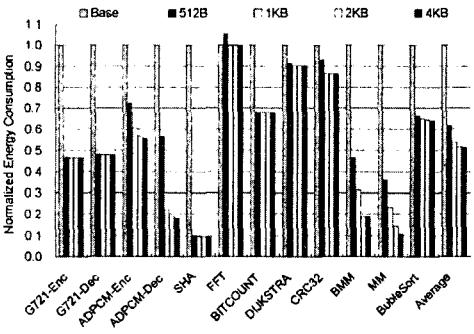


그림 3. 에너지 소모

### 4.2 에너지 소모

에너지 소모에 관한 결과는 그림 3에 나타나 있다. SPM을 사용하지 않을 때의 에너지 소모를 기준으로 SPM을 사용할 때 에너지 소모를 정규화 하였다. 실험은 메모리 종류 별로 참조가 일어날 때의 동적 에너지 소모만 고려하였다.

그림 3에서 알 수 있듯이, 본 논문에서 제안한 방법을 사용하면 에너지 소모가 평균 49% 감소한다. G721이나 BITCOUNT는 SPM에 올릴 수 있는 배열의 크기가 작기 때문에 일정 크기 이상의 SPM에서 추가적인 에너지 이득이 없다. 오히려, SPM의 크기가 커짐에 따라 메모리 참조 한번당 에너지 소모량이 커지므로 전체 에너지는 증가한다 [1]. 앞서와 마찬가지로 FFT는 hot spot이 소프트웨어 부동소수점 연산이기 때문에 별 이득이 없으며, Bubble Sort는 실행 시간이 증가하였지만 에너지 소모는 약 46% 가량 감소했다.

## 5. 결 론

본 논문은 크기가 SPM보다 큰 배열이나 구조체를 SRAM과 SPM 간의 복사 비용뿐만 아니라 이들의 참조 비용을 함수 단위로 가중치로 고려하여 SPM에 할당될 변수를 결정한 다음, 코드 변환을 하고, 이 코드 변환에 의해 SPM의 정적인 할당뿐만 아니라 동적인 할당을 통한 관리를 통해 코드의 실행 시간 및 에너지의 감소가 가능하다는 것을 보였다. 제안된 코드 변환은 컴파일러에 의해 자동화 될 수 있다. 10개의 임베디드 벤치마크 프로그램을 이용하여 본 논문에서 제안하는 방법의 성능 평가를 시뮬레이션을 통하여 한 결과, 실행 시간은 평균 23% 향상되고 에너지 소모는 평균 49% 감소함을 알 수 있으며, 이를 통해 본 논문에서 제안한 방법이 SPM의 활용에 있어 매우 효과적임을 알 수 있다.

## 6. 참고문헌

[1] Rajeshwari Banakar, Stefan Steinke, Bo-sik Lee, M.Balakrishnan, Peter Marwedel, Scratchpad Memory : A Design Alternative for Cache On-chip Memory in Embedded Systems, Proceedings of the tenth international symposium on Hardware/software codesign, pp 73-78, 2002

[2] Stefan Steinke, Nils Grundwald, Lars Wehmeyer, Rajeshwari Banakar, M. Balakrishnan, Peter Marwedel, Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory, ISSS 2002, pp 213-218, October 2-4, 2002

[3] Stefan Steinke, Christoph Zobiegala, Lars Wehmeyer, Peter Marwedel, Moving Program Objects to Scratch-Pad Memory for Energy Reduction, Technical Report No. 756, University of Dortmund, Dept. of CS XII, 2001

[4] Stefan Steinke, Lars Wehmeyer, Bo-Sik Lee, Peter Marwedel, Assigning Program and Data Objects to Scratchpad for Energy Reduction, DATE 2002, pp 409-416, March 2002

[5] Manish Verma, Stefan Steinke, Peter Marwedel, Data Partitioning for Maximal Scratchpad Usage, ASP-DAC 2003, pp 77- 83, Jan 2003

[6] M. Kandemir, J.Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, A. Parikh, Dynamic Management of Scratch-Pad Memory Space, Annual ACM IEEE Design Automation Conference, pp 690-695, June 2001

[7] M. Kandemir, A. Choudhary, Compiler-Directed Scratch Pad Memory Hierarchy Design and Management, Annual ACM IEEE Design Automation Conference, pp 628-633, June 2002

[8] P.R. Panda, N.D. Dutt, A. Nicolau, Efficient utilization of scratch-pad memory in embedded processor applications. In Proc. of Eurpocan Design and Test Conference, pp 7-11, March 1997

[9] J. Sjodin, B. Froderberg, T. Lindgren, Allocation of Global Data Objects in On-chip RAM, CASE 98, 1998

[10] Oren Avissar and Rajeev Barua, An Optimal Memory Allocation Scheme for Scratch-Pad-Based Embedded Systems, ACM Transactions on Embedded Computing Systems, Vol. 1, No. 1, pp 6-26, November 2002

[11] Randy Allen and Ken Kennedy, Optimizing Compilers for Modern Architectures, Morgan Kaufmann, 2002

[12] ARM, ARM Developer Suite 1.2: Debug Target Guide, http://www.arm.com

[13] Hyung Gyu Lee, Naehyuck Chang, Energy-Aware Memory Allocation in Heterogeneous Non-Volatile Memory Systems, ISLPED 2003, pp 420-423, August 2003

[14] S Wilton, Norm Jouppi, Cacti : An enhanced access and cycle time model, IEEE Journal of Solid State Circuits, Vol 31, Issue 5, pp 677-688, May 1996