Scratch-Pad 메모리를 위한 동적 코드 배치 기법

김지훈° 장춘기 이재진 민상렬
서울대학교 컴퓨터 공학부
{chihun°, choonki, jlee}@aces.snu.ac.kr, symin@dandelion.snu.ac.kr

Dynamic Code Placement Techniques for Scratch-Pad Memory

Chihun Kim° Choonki Jang Jaejin Lee Sang Lyul Min
School of Computer Science and Engineering, Seoul National University

요 약

SPM (Scratch-Pad Memory)을 위한 코드 배치 기법과 demand paging기법을 post pass optimizer를 사용하여 구현한다. 코드 배치 문제는 ILP (Integer Linear Programming) 문제로 변환하여 해결한다. 최적화기는 ILP 해답의 질을 높이기 위해 응용 프로그램의 프로파일 정보를 사용하고, 코드로부터 natural loop을 추출한다. 또한 SPM을 사용하여 demand paging을 할수 있도록 추가 코드를 삽입한다. 이 기법을 사용해 6개의 내장형 응용 프로그램을 실험하였고, 프로그램 크기의 20%에 해당하는 SPM에 대해 전력 소모는 75.9%로 감소하였고 성능은 54.5% 증가 하였다.

1. 서 론

Mobile embedded system에서 전력 소모를 줄이는 작업은 중요한 의미를 지닌다. SRAM을 사용한 on-chip 캐쉬는 전력 소모의 상당한 부분을 차지하기 때문에 대안으로 SPM이 개발되었다. 하지만 개발자에게 투명한(transparent) 캐쉬와는 달리 SPM은 지정 가능한 독립된 주소를 가지기 때문에, 개발자는 SPM에 코드와 데이터를 직접 할당해야하는 어려움이 따른다.

앞서 진행된 연구 [1, 2, 4, 5, 7, 8, 9, 11, 12, 13, 14]에서 코드와 데이터의 배치에 대해 언급하였으나, 자동화된 기법과 동적인 코드 배치 기법은 연구되지 않았다. 따라서 본 논문에서는 자동화된 코드 배치 기법과 프로그램의 실행 시간에 필요한 코드를 SPM에 로드 하는 동적인 코드 배치 기법에 대해 소개하도록 한다.

본 논문에서 구현한 post-pass 최적화기는 페이지 관리자를 포함한 실행 이미지를 생성한다. 응용 프로그램은 여러 개의 세그먼트로 분할되어 SPM 또는 DRAM에 정적으로 배치되거나, 실행시간에 페이지 관리자가 DRAM으로부터 SPM으로 로드 하도록 한다. 이 세 가지의 배치 방법은 프로파일 정보를 사용하여 ILP 문제를 풀어서 결정한다.

최적화기는 응용 프로그램과 라이브러리의 ELF object 파일을 입력으로 받는다. 먼저 프로그램을 구성하는 함수와 함수내의 natural loop을 찾아내어 이들을 세그먼트로 구성한다. ILP 문제는 세그먼트 단위로 코드 배치를 결정한다. 따라서 프로그램의 실행 흐름이 세그먼트의 경계를 넘어서는 함수 호출이나 리턴 명령어는 페이지 관리자를 호출하도록 변경된다. 페이지 관리자는 대상 세그먼트가 이미 SPM에 로드 되어 있는지 확인한 후에 대상 세그먼트를 버퍼에서 실행하거나, DRAM으로부터 SPM으로 복사한 후에 실행 하게 된다.

본 논문의 2절에서는 최적화기에 의해 수행되는 코드의 변환을 설명한다. 3절에서는 코드의 배치를 ILP 문제로 변경하는 모델을 제안한다. 4절과 5절은 실험 환경과 결과를 제시한다. 마지막으로 6절에서 결론을 내리도록 한다.

2. Post-pass Optimizer에 의한 코드 변환

그림1은 SNACK-pop(Seoul National university Advanced Compiler tool Kit) 으로 이름을 붙인 post-pass 최적화기의 구조이다. 최적화기는 응용 프로그램에 필요한 모든 오브젝트들을 역어셈블하고, 오브젝트 내에 있는 정보들을 이용하여 함수들의 목록으로 변환한다. 각 함수들을 분석하여 CFG(Control Flow Graph)로 변환하면서 동시에 코드 영역에 있는 상수 데이터 중에서 다른 함수에게 인자로 넘어가는 것들을 데이터 영역에 복사하고 복사한 지점을 직접 접근하도록 수정한다. 이후, CFG를 분석하여 natural loop을 추출한다.

이렇게 만들어진 실행 이미지로 프로파일 정보를 얻어낸 후 ILP를 풀면 각 함수들은 SPM, EXT, PAGED 셋 중에서 하나에 할당된다. 그림 2에서 볼 수 있듯이 위의 SPM과 EXT에 해당하는 함수들은 각각 SPM과 DRAM에 상주하고, PAGED에 해당하는 함수들은 DRAM에 있으면서 실행 시 요구에 따라 페이지 관리자에 의하여 SPM에 있는 버퍼에 로드된다.

함수가 PAGED에 속한다면 고정된 크기의 페이지로 구성된 세그먼트로 변환된다. 이 논문에서 페이지 크기는 64byte이며 세그먼트의 크기는 페이지 크기의 배수가 된다. SPM에 이러한 PAGED 함수들을 위한 버퍼를 두고, 실행 시 이들에 대한 요구가 있을 때 페이지 관리자에 의하여 버퍼에 복사되고 실행된다. PAGED 함수들은 실행 시 버퍼에 있음을 보장받을 수 없으므로, 페이지 관리자를 통하여 수행되거나 반환된다. 따라서 PAGED 함수를 호출하는 명령과 PAGED 함수로 리턴하는 명령들은 모두 페이지 관리자를 거쳐야한다. 페이지 관리자는 이를 위하여 페이지 테이블, 함수 테이블, 버퍼 테이블을 유지·관리하며 필요에 따라 함수에 해당하는 세그먼트를 버퍼에 복사하고, 버퍼가 부족한 경우에는 round-robin 정책에 따라 교체한다. 함수포인터를 이용한 호출이 있으므로 호출과 리턴 정보를 정적으로는 정확하게 알 수 없기 때문에 모든 함수포인터 호출 명령과 리턴 명령을 전부 변환해주어야만 한다. 하지만 상주하는 함수간의 호출에는 호출 명령과 리턴 명령 모두 변환할 필요가 없고, 호출하는 함수가 상주하고 대상이 PAGED인 경우 리턴 명령이 변환될 필요가 없다. 프로파일을 통하여 모든 함수 호출과 리턴에 대한 정보를 얻어내어 이 두 경우에 해당하는 경우에는 변환을 하지 않을 수 있었고, 여기에 들었던 부담을 상당히 줄일 수 있었다.

3. 코드 배치를 위한 ILP 문제의 공식화

페이지 관리 기법이 없는 SPM 코드 배치 문제는 0-1 knapsack 문제로 쉽게 변환 가능하다 [4, 6]. ILP의 목적 함수는 메모리 접근에 의한 전력 소모량을 최소화 하는 것이다. 본 논문에서는 이를 확장하여 demand paging 기법을 다룰 수 있도록 한다.

3.1 ILP의 구성

수식에 사용되는 변수는 다음과 같이 정의한다.

PS	Page의 크기
N	프로그램을 구성하는 함수의 개수
f_i	프로그램을 구성하는 i번째의 함수
S_i	함수 f_i의 크기
A_i	프로그램의 실행에 의해 f_i가 접근한 명령어의 개수

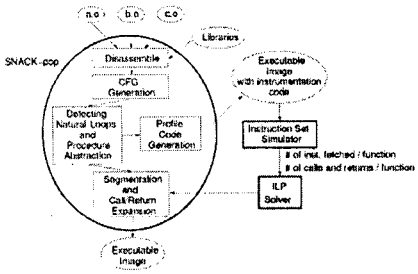

그림 1

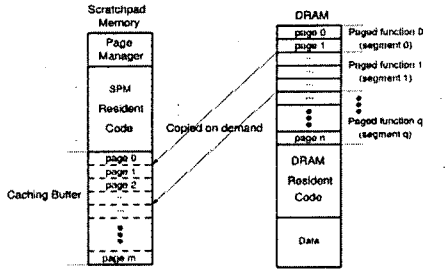

그림 2

S_{spm} SPM의 크기
E_{spm} SPM으로부터 명령어를 읽을 때 소모되는 에너지
E_{ext} DRAM으로부터 명령어를 읽을 때 필요한 에너지
C_i 함수 f_i에 대한 함수 호출 회수
R_i 함수 f_i로 리턴 되는 회수
E_c 함수호출의 변환 때문에 추가적으로 소모되는 에너지
E_r 함수리턴의 변환 때문에 추가적으로 소모되는 에너지

각 함수마다 general integer variable인 I_{spm}, I_{ext}, I_{paged}가 정의된다. 이들은 SPM, EXT, PAGED 클래스 중 어디에 포함될 것인지를 결정하는 변수이다. 추가로 버퍼에 할당되는 페이지의 개수를 결정하기 위한 integer variable을 정의한다.

$$I_{spm}(i) = \begin{cases} 1, & \text{if } f_i \in SPM \\ 0, & otherwise \end{cases}$$

$$I_{ext}(i) = \begin{cases} 1, & \text{if } f_i \in EXT \\ 0, & otherwise \end{cases}$$

$$I_{paged}(i) = \begin{cases} 1, & \text{if } f_i \in PAGED \text{and} S_i \le I_{buf} \times PS \\ 0, & otherwise \end{cases}$$

$I_{buf}(i) =$ caching buffer size in the number of pages

ILP의 목적함수는 메모리 접근에 따른 에너지의 총합이며 이 함수의 값을 최소화해야 한다. 함수는 다음과 같이 정의된다.

$$\sum_{i=1}^{N} (I_{spm}(i) A_i E_{spm} + I_{ext}(i) A_i E_{ext} + I_{paged}(i)[A_i E_{spm} + P_i])$$

P_i는 PAGED에 속한 함수가 DRAM으로부터 SPM으로 복사될 때 필요한 에너지 소모량이며 다음과 같이 정의된다.

$$P_i = (C_i + R_i)(E_{spm} + E_{ext})(\lceil S_i/PS \rceil (PS/4)) + C_i E_c + R_i E_r$$

각 함수는 하나의 클래스에만 속하므로 다음과 같은 조건을 만족하여야 한다.

$$I_{spm}(i) + I_{ext}(i) + I_{paged}(i) = 1 \quad \text{for all } 0 < i \le N$$

SPM에 정적으로 배치되는 함수의 총 크기는 SPM의 크기에서 버퍼로 사용되는 SRAM을 뺀 크기보다는 작아야 하고, 버퍼는 SPM의 크기를 넘을 수 없으므로 다음 조건을 만족해야 한다.

$$\sum_{i=1}^{N} I_{spm}(i) S_i \le S_{spm} - I_{buf} PS$$

$$0 \le I_{buf} PS \le S_{spm}$$

3.2 Natural Loop의 영향

다른 함수에 비해 적은 양의 코드를 가지지만, 많이 실행되는 함수일수록 SPM에 배치될 가능성이 높아진다. 따라서 큰 함수로부터 Natural Loop을 추출하면 SPM에 할당되는 코드가 많아져서 보다 적은 전력으로 빠르게 실행될 수가 있다.

4. 실험 환경

표 1과 같이 본 논문에서는 11개의 응용 프로그램을 실험하였다. Multi 프로그램은 *Quicksort*, *Dijkstra*, *Sha*, *ADPCM-enc*, *ADPCM-dec*, and *Bitcount*를 합쳐서 만들어진 프로그램이다. 실험은 ARMulator [3]를 사용하였다. 아키텍쳐는 캐쉬가 없는 200MHz의 ARM 920T를 사용하였다. SPM은 SRAM으로 external memory는 SDRAM으로 설정하였고, 표 2와 같은 특성을 가지는 메모리이다.

Application	Source	Code Size (Byte)
Multi	Synthetic	12,328
FFT	MIBench [15]	14,688
Epic	MediaBench [10]	21,124
Unepic	MediaBench	20,128
MPEG4-enc	www.xvid.org [16]	49,912
MPEG4-dec	www.xvid.org	44,248

표 1 테스트 프로그램

	Seq. Read	Nonseq. Read	Seq. Write	Nonseq. Write
SPM	1ns	1ns	1ns	1ns
External Memory	120ns	150ns	120ns	150ns

표 2 메모리 특성

Application	Executed Code Size	SRAM size	Execution Time (%)	Energy (code) (%)	Total Energy (%)
Multi	12KB	2	47.5	8.4	23.9
FFT	14KB	2	54.9	28.5	35.7
Epic	21KB	4	40.9	4.9	15.9
Unepic	20KB	4	13.7	3.0	14.2
MPEG4-enc	49KB	8	45.7	5.0	21.9
MPEG4-dec	43KB	8	70.4	20.6	32.8
Average	26.4KB	4.67	45.5	11.7	24.1

표 3 Demand Paging을 하였을 때의 성능 (*Normalized*)

Type	Size (KB)	Energy (nJ)
SRAM (0.13μm) (4 byte access)	1	0.127
	2	0.134
	4	0.145
	8	0.167
	12	0.189
	16	0.205
SDRAM	128,000	62.16
	128,000	55.20

표 4 SRAM과 SDRAM의 전력 소모

5. 실험결과

그림 3 은 코드 배치 기법과 SRAM의 크기에 따른 메모리 접근 회수를 보여주고 있다. 모든 경우에 대해 데이터는 SDRAM에 항상 할당되었다. *ILP* 항목은 함수로부터 Natural Loop의 추출 없이 ILP 문제를 풀어서 코드 배치를 결정하였을 경우이다. *Nloop* 항목은 Natural Loop을 추출하여 하나의 함수로 추상화시킨 다음 ILP 문제를 풀었을 경우를 나타낸다. 마지막으로 Paged는 Nloop 방식에서 Demand paging기법을 추가하였을 경우이다.

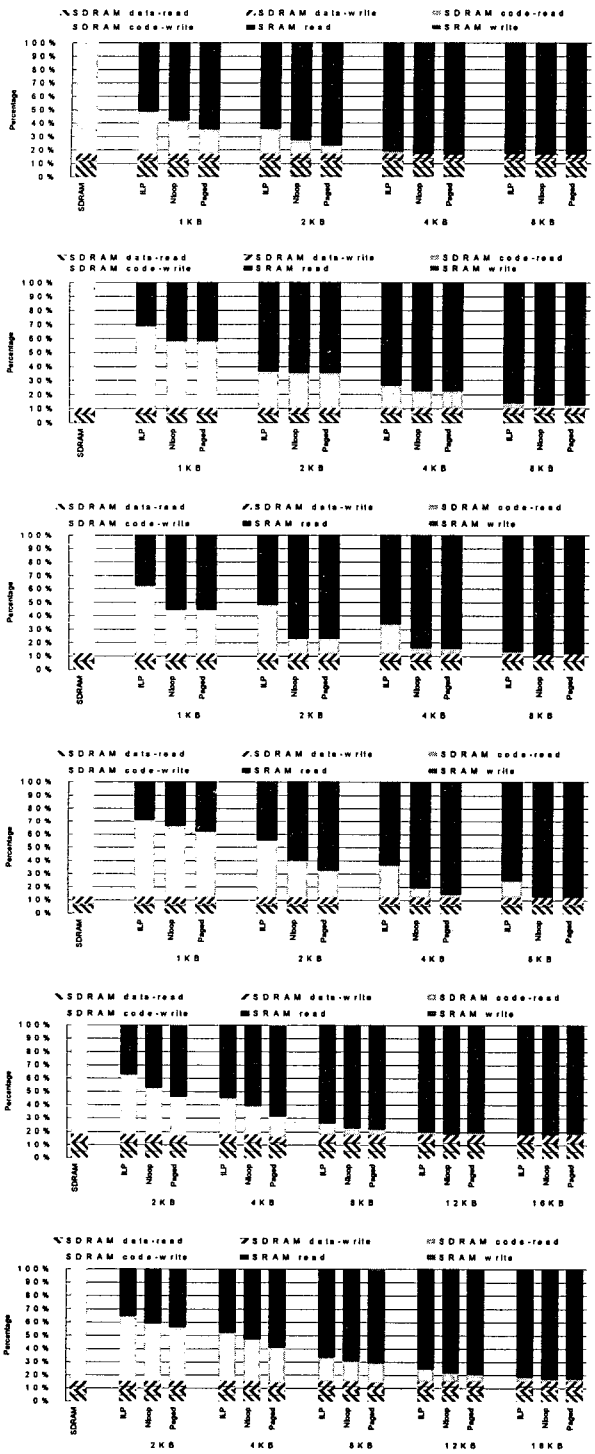

그림 3 프로그램 별 메모리 접근 회수

SRAM의 크기가 커짐에 따라 SRAM에서 실행되는 코드의 비율이 증가함을 알 수 있다. ILP의 풀이에서 실행이 많이 되는 코드가 SRAM이나 PAGED로 할당되기 때문이다. 이에 따른 실행 시간과 전력 소모의 감소는 표 3에서 알 수 있다. 실행 시간은 SDRAM으로만 실행했을 때의 평균 45.5%가 필요했으며, 에너지는 11.7%가 소모되었다. 본 논문의 기법을 적용했을 때 상당한 성능 향상과 전력 소모의 감소를 얻을 수 있었다.

6. 결 론

본 논문은 SPM을 활용할 수 있는 자동화되고 동적인 코드 배치 기법을 소개하였다. 문제 해결을 위해 Demand paging 기법과 프로파일 정보를 사용하였고, 코드 배치 결정 문제를 ILP 문제로 변환하였다. 이러한 기법은 post-pass 최적화기에 의해 응용 프로그램에 자동적으로 삽입이 되도록 하였다. 실험 결과는 본 논문의 접근 방식이 효과적이라는 것을 보여주었다. 6개의 응용 프로그램에 대해 프로그램 크기의 20%에 해당하는 SPM으로 전력 소모는 평균 75.9%로 감소하였고 성능은 평균 54.5% 향상하였다.

참고 문헌

[1]. Federico Angiolini, Luca Benini, and Alberto Caprara. Polynomial-Time Algorithm for On-chip Scratchpad Memory Partitioning. In *International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES 2003)*, October 2003.

[2]. Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini, and Mauro Oliveri. A Post-Compiler Approach to Scratchpad Mapping of Code. In *International Conference on Compilers, Architectures and Synthesis of Embedded Systems (CASES 2004)*, September 2004.

[3]. ARM. *ARM Developer Suite Version 1.2: ARM Debug Target Guide*. ARM Limited, 2001.

[4]. Oren Avissar and Rajeev Barua. An Optimal Memory Allocation Scheme for Scratchpad-Based Embedded Systems. *IEEE Transactions on Embedded Computing Systems*, 1(1):6-26, 2002.

[5]. Rajeshwari Banakar, Stefen Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems. In *The tenth International Symposium on Hardware/Software Codesign*, paged 73-78, 2002.

[6]. Tomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990.

[7]. Poletti Francesco, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, and Jose M. Mendias. An Integrated Hardware/Software Approach for Run-Time Scratchpad Management. In *The 41st Design Automation Conference (DAC 2004)*, pages 238-243, June 2004.

[8]. M. Kandemir and A. Choudhary. Compiler-Directed Scratch Pad Memory Hierarchy Design and Management. In Annul ACM IEEE Design Automation Conference, June 2002.

[9]. M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic Management of Scratch-Pad Memory Space. In *Annual ACM IEEE Design Automation Conference*, June 2001.

[10]. Chunho Lee, Miograg Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[11]. P.R. Panda, N.D.Dutt, and A. Nicolau. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In *European Design Automation and Test Conference*, March 1997.

[12]. Stefen Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In Design, Automation and Test in Europe Conference and Exposition (DATE 2002), pages 409-417, February 2002.

[13]. Manish Verma, Lars Wehmeyer, and Peter Marwedel, Cache-Aware Scratchpad Allocation Algorithm. In Design, Automation and Test in Europe Conference and Exposition (DATE 2004), pages 1264-1269, February 2004.

[14]. Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic Overlay of Scratchpad Memory for Energy Minimization. In International Conference on Hardware/Software Codesign and System Synthesis, September 2004.

[15]. Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A Free, Aommercially Representative Embedded Benchmark Suite. In Proceedings of the 4th Annual Workshop on Workload Characterization, December 1998.

[16]. http://www.xvid.org. Xvid MPEG-4 Video Codec 2004