DSP 내장형 시스템 설계에서 코드 스케줄링을 이용한 주소 코드 최적화

최윤서 김태환
한국과학기술원 전산학과
yschoi@vlsisyn.kaist.ac.kr tkim@cs.kaist.ac.kr

Address Code Optimization using Code Scheduling in DSP Embedded System Design

Yoonseo Choi Taewhan Kim
Dept. of EECS, Korea Advanced Institute of Science and Technology

요 약

본 논문에서는 DSP코드 생성시 어드레스 인스트럭션의 개수를 최소화하기위한 효과적인 어드레스 코드 생성 기법을 제안하였다. 기존의 방법에서는 코드 스케쥴링이 수행된 다음에 어드레스 코드가 생성되었다. 본 논문에서는 코드 스케쥴링과 어드레스 코드 생성을 결합하였고, 어드레스 인스트럭션의 개수를 줄이기 위한 효과적인 스케쥴링 방법을 제안하였다. 실험결과는 최근 연구에[6,8] 비해 23.7% 크기의 향상을 보여주었다.

1 서론

내장형 VLSI 시스템의 복잡도의 증가에 따라 전통적인 DSP 컴파일러들은 작은 코드크기와 실시간 수행이라는 조건을 충족시키지 못하고 있다. 이종의 레지스터 집합과 특정기능의 하드웨어들과 불규칙적인 데이터 경로 때문에 기존에 연구된 방법에 의해 최적화된 코드의 질이 좋지 못하다 [1]. 범용 프로세서의 경우 최종 생성된 코드의 bit중 50%, 6개의 인스트럭션 중 1개의 인스트럭션이 어드레스 코드이므로 [2]. 주소 배당문제(storage assignment), 즉 프로그램의 변수들의 메모리 상에서의 배열을 최적화하는 문제는 코드 생성에 있어서 매우 중요한 문제이다. Motorola *DSP 56000* 시리즈 용으로 컴파일된 MediaBench[4]의 경우 55%이상의 인스트럭션이 어드레스용 레지스터를 이용한다[3]. 결론적으로 변수에 메모리 주소를 배당하는 것을 최적화함으로써 코드 크기를 감소시키고 성능도 향상시킬 수 있다.

여러 가지 아키텍쳐들(VAX, TI TMS320C2x DSP family, 대부분의 내장형 컨트롤러들) 자동-증감 간접 레지스터 어드레싱 모드(auto-increment/decrement register indirect addressing mode)를 제공한다. 이들 아키텍쳐들은 그림 1에서처럼 전용 어드레스 생성 유닛(address generation unit, AGU)를 가진다. 이들은 중앙 데이터 경로와 병렬적으로 메모리의 어드레스를 빠르게 생성해낼 수 있다. 그러나, 자동-증감 어드레싱 모드를 이용하기 위해서는 변수들의 메모리상에서의 상대적인 위치가 중요하다. 따라서 선언된 순서대로, 또는 가나다순으로 변수를 메모리에 배치하는 기존 컴파일러들의 방법과는 달리, DSP용 컴파일러들은 자동-증감 어드레싱 모드를 최대로 이용하기 위해 변수들의 메모리상에서의 주소를 적절하게 배열해야 한다.

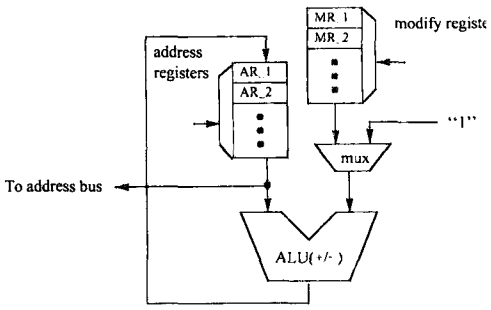

Figure 1: AGU (address generation unit) 모델.

1개의 어드레스용 레지스터만을 사용하는 경우의 주소 배당문제(simple offset assignment problem, SOA)는 Bartley [5]와 Liao 외 [6, 7]에 의해서 처음 연구되었다. Liao는 SOA는 그래프상에서 maximum weighted path cover(MWPC)를 찾는 문제와 동일하며 이 문제가 NP-complete 이라는 것을 보였고 Kruskal의 최대 스패닝 트리 알고리즘을 변형한 휴리스틱을 이용하여 이를 풀었다. 또한 그는 SOA를 여러 개의 어드레스용 레지스터들을 사용하는 경우의 주소 배당문제인 GOA (general offset assignment problem, GOA)로 확장시켰다.[6, 7] Leupers와 Marwedel[8]은 tie-breaking 휴리스틱과 변수들을 여러 개의 집합으로 분할하는 방법을 제안함으로써 Liao의 방법을 개선하였다. 또한 후처리 최적화로서 수정용 레지스터(modify

register)의 이용을 최적화하는 방법인 MRO을 연구하였다. Leupers와 David[9]는 임의의 레지스터 파일 크기와 자동 증감 범위 하에서의 GOA와 MRO를 동시에 최적화 하는 연구를 하였다. Sudarsanam외[10]는 유전자 알고리즘을 이용해서 일정 범위로 자동-증감되는 어드레싱 모드를 고려하였다. 위의 모든 연구들은 [5,6,7,8,9,10] SOA/GOA를 주어진 변수들의 사용 순서(access sequence) 하에서의 최적화 문제로 다루었다. 그러나 변수들이 메모리상에서 접근 되는 순서가 SOA/GOA의 결과의 질에 매우 크게 영향을 미치므로, 기존의 방법들은 단지 부분적인 최적화방식이라고 할 수 있다. 따라서 본 논문에서는 어드레스 코드를 더욱 효과적으로 최적화하기 위해, SOA/GOA와 코드 스케줄링을 함께 고려한다. 또한 GOA는 SOA의 확장이므로,본 논문에서는 SOA 휴리스틱[6,7,8]인 solve-SOA 을 이용한 SOA의 최적화에 초점을 맞춘다.

코드 스케줄링과 주소 배당 문제를 병합의 유용성을 간단한 예를 통해 볼 수 있다. SOA문제를 푼다고 가정하고 그림 2(a)의 C 프로그램을 보자. 그림 2(b-i)는 이 C 프로그램의 변수 들의 사용 순서를 나타낸다. Liao와 그 외는 [6,7]은 SOA문제를 프로그램 변수를 노드 $v \in V$ 로, 두 개의 변수가 사용 순서 상에서 인접한 회수를 에지 $e \in E$의 가중치(weight)로 하는 접근 그래프(access graph)로 모델링 하였다. 그림 2(b-i)의 사용순서에 해당하는 접근 그래프가 그림 2(b-ii)이다. [1] 최적의 주소 배당은 $G(V, E)$ 에서 다음의 값을 최소화하는 패스 커버(path cover)를 찾는 것이다.

$$C(G, P) = \sum_{\forall e \ of \ edges \ not \ in \ P} w(e) \qquad (1)$$

maximum weighted path cover(MWPC)를 찾는 것은 곧 C 값을 최소화 하는 것이다. 그림 2(b-ii)에서 b에서 시작해서 f에서 끝나는 굵은 선이 패스 커버를 나타낸다. 이 패스 커버는 Liao[6,7]의 알고리즘과 Leupers와 Marwedel의 tie-breaking방식[8]에 의해 구해졌다. 이에 해당하는 변수의 메모리상의 배열이 그림 2 (b-iii)에 나타나 있다. C의 값은 $w(a,c) + w(a,f) + w(b,e) + + W(e,g) = 12$ 이다.

그림 2(c) 는 새로이 스케줄된 C 코드와 그에 따른 변수의 사용 순서와 메모리 상의 메모리 배열을 나타낸다. 굵은 글씨로 나타낸 것이 새로이 스케줄 된 부분인데, 이 경우 C의 값은 10으로 12에 비해서 17%나 감소했음을 알 수 있다. 이렇듯 약간의 주소 배당 문제의 결과값은 변수의 사용 순서에 매우 민감함을 알 수 있다. 따라서, 기존의 컴파일러들과는 달리 우리는 코드 스케줄링과 결합된 효과적인 주소 배당 방식을 제안한다.

2 어드레스 인스트럭션 개수를 최소화 하기 위한 코드 스케줄링

입력 C 프로그램은 기본 블록(basic block) 들로 이루어져 있다. 우리는 기본 블록 단위에서의 어드레스 코드 최적화를 다루며 기본 블록 단위 이상의 최적화에 대해서는 고려하지 않는다. 우리의 알고리즘의 입력은 오퍼레션들간(instruction)의 디펜던시 그래프(dependency graph)이다. 디펜던시 그래프의 각 노드는 하나의 오퍼레이션을 나타낸다. 또한 노드 a로부터 b로의 에지는 a가 b보다 먼저 수행되어야 함을 나타낸다.

우리의 목표는 디펜던시 그래프의 노드들을 모두 스케줄해서 전체 변수의 사용 순서를 정하여, 결과적으로 생성되는 어드레스 코드의 개수를 최소화 하는 것이다. 제안된 알고리즘은 사용되는 변수의 순서를 한 번에 하나의 오퍼레이션씩 점

[1] 오퍼레이션 $x = y + z$의 변수들의 사용순서는 'yzx'이다.

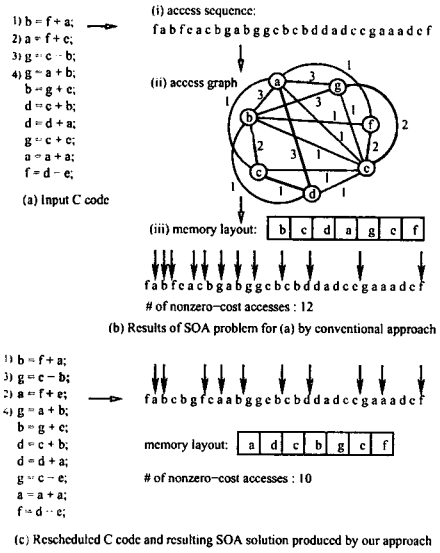

Figure 2: 어드레스 인스트럭션의 개수를 최소화하는 데 있어서의 코드 스케줄링의 효과를 보여주는 예.

차적으로 생성해 나간다. 디펜던시 그래프 상에서 모든 부모가 이미 스케줄 되고 자신은 아직 스케줄 되지 않은 오퍼레이션을 레디 오퍼레이션(ready operation)이라고 부른다. 매 수행마다 우리의 알고리즘은 가장 유망한 오퍼레이션을 골라서 현재의 실행 단계에 스케줄하고 그 오퍼레이션에 속한 변수들을 현재의 변수들의 사용 순서 뒤에 덧붙여나간다. 디펜던시 그래프상의 모든 오퍼레이션들이 스케줄 될 때까지 이 과정이 반복된다. 레디 오퍼레이션들 중에서 가장 유망한 오퍼레이션은 최소로 $C(Eq.(1))$값을 증가 시키는 오퍼레이션이다. 예를 들어, 이미 $t-1$실행 단계까지 스케줄링을 하고, 이제까지의 변수의 사용 순서가 a로 끝나있다고 가정하자. (그림 3(a)) 그림3(b)에 나온 것처럼 실행 단계 t에 스케줄할 차례라고 할 때 이때의 접근 그래프는 그림3(c)와 같다고 하자. 굵은 선으로 나타난 경로(P_{curr})가 *solve-SOA* [8]을 이용한 SOA결과이다. $C(G_{curr}, P_{curr})$ 는 6이다. 다음 실행 단계에 수행 가능한 모든 레디 오퍼레이션을 고려한 결과는 다음과 같다. 그림3(d),(e),(f)는 그림3(c)의 상태에서 오퍼레이션 1,2와 3이 각각 다음 단계에 수행된 경우에 접근 그래프의 변화를 나타낸다. 오퍼레이션 2를 스케줄 한 경우에 C값이 가장 최소로 증가되므로 오퍼레이션 2가 다음 실행 단계에 수행될 오퍼레이션으로 정해진다. 따라서 변수 'abc'가 현재의 변수의 사용 순서에 더해져서 '…aabc'가 된다. 그 다음으로 준비된 오퍼레이션들의 집합이 갱신되고 모든 오퍼레이션들이 스케줄 될 때 까지 이 과정이 반복된다. 만약 두 개 이상의 레디 오퍼레이션들이 동일한 C값을 가진다면 한 단계 이상을 더 보아서 어떤 오퍼레이션을 수행할 지 정한다. 그림4는 우리의 코드 스케줄과 결합된 CSchedule-SOA 알고리즘의 요약이다. 이는 리스트 스케줄링(list scheduling)의 한 변형이다. 이 알고리즘의 수행 시간은 m이 접근 그래프의 에지의 개수라고 할 때 $O(m \cdot \log m)$이다. 따라서 우리의 알고리즘의 총 수행 시간은 n이 오퍼레이션의 개수라고 할 때 $O(n^2 \cdot m \cdot \log m)$이다. 그러나 실제로 한 실행

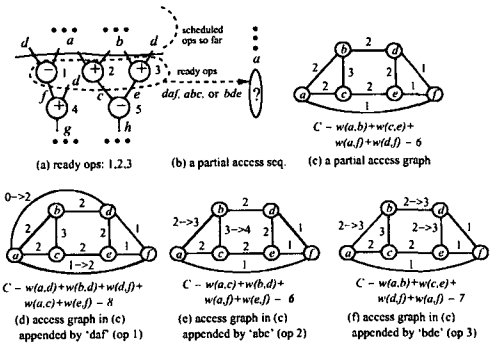

Figure 3: ready operations, to minimize C $Eq.(1)$의 C의 값을 최소화하기위한 준비된 오퍼레이션의 선택

Cschedule-SOA: SOA를 위한 코드 스케쥴링 (디펜던시 그래프)

- n = 디펜던시 그래프의 노드의 개수;
- R = 디펜던시 그래프에서 부모 노드들이 이미 모두 스케쥴된 노드들의 집합;
- $S = \phi$;

for $t = 1 .. n$ {
  foreach $op_i \in R$ {
    - op_i를 실행 단계t에 스케쥴한다;
    - op_i에 들어 있는 변수들을 더해서 접근 그래프G를 바꾼다;
    - G의 maximum weighted path cover이 P를 구한다;
    - $Eq.(1)$의 $C(G,P)$ 를 계산한다;
    - 스케쥴과 G를 원래 상태로 되돌린다;

  } **endforeach**
  - 최소의 $C(G,P)$값을 가지는 op_i를 t에 스케쥴한다;
  - op_i의 변수들을 더함으로써 변수 사용순서S를 갱신한다;
  - 접근 그래프 G와 R도 갱신한다;

} **endfor**

Figure 4: 제안된 스케쥴과 결합된 변수 주소 배당 알고리즘

단계에서의 레디 오퍼레이션의 개수는 n보다 작기 때문에 실제적으로는 $O(n \cdot m \cdot \log m)$에 가깝다.

3 실험 결과

제안된 알고리즘은 C++로 Sun Sparc20에서 실행되었다. 벤치마크 프로그램과 임의로 생성된 프로그램을 이용했다. 표1은 OFU(order of first use) 와 TB-SOA[8]와 우리의 알고리즘을 임의로 생성된 코드를 이용해서 비교한 결과이다. $|V|$와 $|S|$는 프로그램의 변수의 개수와 변수의 사용 순서의 길이를 각각 나타낸다. 우리의 알고리즘은 TB-SOA[8]에 비해서 약 14%까지의 개선을 나타내었다.

표2는 벤치마크 프로그램을 이용한 결과를 나타낸다. BIQUAD(biquad_one_section), FIR, LMS는 DSPstone벤치마크에서 취했다. COMP(complex multiplier)와 ELLIP(elliptical wave filter)은 고수준 합성의 벤치마크들이다. GAULEG , GAUHER , GAUJAC 은 [11]에서 발췌되었다. 이 결과들로부터 우리가 제안한 알고리즘이 매우 잘 작동함을 알 수 있다. 기존의 알고리즘에 비해 전체적으로 약 23%의 개선을 보였다.

|V|/|S|	# addr_instr			gain over(%)	time
	OFU	TB-SOA[8]	*ours*	OFU/[8]	(sec.)
5/10	4.43	2.33	2.33	47.4/0.0	0.01
5/20	6.67	4.51	4.51	32.4/0.0	0.01
15/20	13.56	8.00	7.67	43.4/4.1	0.03
10/50	34.21	25.00	21.5	37.1/14.0	0.07
40/50	37.00	21.71	20.04	45.8/7.7	0.39
10/100	71.66	51.67	48.00	33.0/7.1	0.15
50/100	80.8	52.5	47.5	41.2/9.5	3.84
80/100	77.42	42.5	38.25	50.6/10.0	18.92
100/200	173.59	115.33	102.67	40.9/11.0	86.86

Table 1: 임의 생성 코드를 이용한 어드레스 코드의 크기 비교

design	$\|V\|/\|S\|$	# addr_instr			gain over(%)
		OFU	TB-SOA[8]	*ours*	OFU/[8]
BIQUAD	10/38	20	16	13	35.0/18.8
FIR	5/20	7	4	3	57.1/25.0
LMS	8/30	12	7	5	58.3/28.5
COMP	10/18	13	8	5	61.5/37.5
ELLIP	45/100	84	49	36	57.1/26.5
GAULEG	23/73	25	20	17	32.0/15.0
GAUHER	11/59	23	21	16	30.4/23.8
GAUJAC	23/148	111	71	61	45.0/14.1
avg.					47.1/23.7

Table 2: 벤치마크 프로그램을 이용한 어드레스 코드의 크기 비교

References

[1] P. Marwedel, G. Goossens (Editors), *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995.

[2] G. Araujo, *Code Generation Algorithm for DSPs*, PhD thesis, Dept. of EE, Princeton University, 1997.

[3] S. Udayanarayanan and C. Chakrabarti, "Address Code Generation for Digital Signal Processors," *DAC*, pp. 353-358, 2001.

[4] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for Evaluating and Synthesizing Multimedia and Communications Systems," *Iternational Symposium on Microarchitecture*, pp. 330-335, 1997.

[5] D. H. Bartley, "Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes," *Software - Practice and Experience*, Vol. 22, No. 2, pp. 101-110, 1992.

[6] S. Liao, *et. al*, "Storage Assignment to Decrease Code Size," *ACM SIGPLAN PLDI*, pp. 186-195, 1995.

[7] S. Liao, *et. al*, "Storage Assignment to Decrease Code Size," *ACM TOPLAS*, Vol. 18, No. 3, pp. 235-253, May 1996.

[8] R. Leupers and P. Marwedel, "Algorithm for Address Assignment in DSP Code Generation," *ICCAD*, pp. 109-112, 1996.

[9] R. Leupers and F. David, "A Uniform Optimization Technique for Offset Assignment Problem," *ISSS*, pp. 3-8, 1998.

[10] A. Sudarsanam, S. Lioa, and S. Devadas, "Analysis and Evaluation of Address Arithmetic Capabilities in Custom DSP Architectures," *DAC*, pp. 287-292, 1997.

[11] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery (Editors), *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, pp. 152,154-155, 1993.