임베디드 환경에서 저전력 인코딩을 이용한 코드 압축 기법

이병호° 김태환 서의성 이준원
한국과학기술원 전산학과
{ bhlee@camars.kaist.ac.kr, tkim@ssl.snu.ac.kr, ses@camars.kaist.ac.kr joon@kaist.ac.kr }

Code Compression Combined with Low-Power Encoding

ByoungHo Lee° TaeHwan Kim Euiseong Seo JoonWon Lee
Dept. of Computer Science, Korea Advanced Institute of Science Technology

요 약

임베디드 시스템은 정해진 하드웨어 환경에서 특정 어플리케이션이 돌아가는 형태로 존재한다. 코드 압축은 이러한 임베디드 시스템환경에서 메모리와 프로세서 간의 소모되는 전력을 줄이는 가장 효과적인 방법으로 잘 알려져 있다. 본 논문에서는 기존과는 다른 접근방법을 통해 이 문제를 정의하고, 압축되는 명령어들의 이진코드를 결정하는 방법을 제시하고자 한다. 압축될 명령어들에 적절한 이진코드를 할당한다면 상당한 에너지를 절약할 수 있다. 이는 명령어 접근 때 발생하는 스위칭 활동이 이진코드할당에 큰 영향을 받기 때문이다. 전력 절감을 위해 이 문제를 그래프 최적화 문제로 전환을 하고, 점진적인 노드 커버링 테크닉(Incremental node covering technique)을 사용하여 부분적으로는 효율적이면서도 전체적으로는 효과적인 방법으로 해결하고자 한다.

1. Introduction

최근 이동성을 가진 기기들 위에서 동작하는 응용 프로그램의 중요성이 커짐에 따라 임베디드 시스템을 설계할 때에 전력 소모를 최소화하는 일이 큰 이슈가 되고 있다. 특히 메모리와 프로세서 간에 발생하는 에너지를 줄이는 것이 중요한데, 이는 이 부분이 임베디드 시스템의 전력소모의 대부분을 차지하고 있기 때문이다. 코드나 데이터를 압축하는 기법이 메모리와 프로세서 간에 발생하는 에너지를 줄이는 방법 중 하나로 알려져 있다. Yoshida[1]와 Benini[2]가 소개한 명령어 압축기법은 메모리와 버스의 에너지 소모에 중점을 두었다. Yoshida는 모든 개별적인 명령어를 뽑아 총 N개의 명령어를 $\lceil \log_2 N \rceil$의 이진코드로 압축을 했다. 이는 임베디드 시스템에서 작동하는 프로그램들은 특정 명령어만을 주로 쓰며 그 명령어의 개수는 코드 크기에 비해 아주 작다는 것에 착안한 결과이다. Benini는 응용 프로그램의 실행 프로파일을 분석한 결과 단지 아주 적은 개수의 명령어만이 자주 참조된다는 것을 알았다. 그는 가장 자주 쓰이는 256개의 개별 명령어를 뽑아 그 명령어만을 압축하였는데 그 이유는 여러 벤치마크 프로그램을 돌려 본 결과 특정 256개의 명령어가, 참조되는 전체 명령어의 50~80%정도를 차지한다는 사실을 알았기 때문이다. 그렇지만, 이 연구에서는 압축된 256개의 명령어들이 어떤 방법에 따라 8비트의 이진 코드를 할당받는지에 대한 언급이 없다. 명령어들에 할당되는 이진 코드의 패턴에 따라서 소모되는 전력의 양이 결정되는 것을 감안한다면 중요한 문제를 간과한 것이다. 따라서 본 논문에서는 압축을 위해 선택된 명령어들에게 이진 코드를 할당하는 문제를 언급하고 명령어 버스에서 발생하는 스위칭 활동을 최소화하는 방법에 대해 다루겠다.

2. Motivational Example

우선 [2]에서 제시한 부분적인 코드 압축이 어떤 효과가 있으며 코드를 할당하는 문제가 왜 중요한지를 알아보자. [Table I]은 각각의 명령어들과 원래의 코드, 그리고 그들의 접근횟수를 보여주고 있다. 이제 [2]에서 소개한 방법에 따라 코드를 압축해 보자. 가장 자주 등장하는 명령어는 *I1*, *I2*, *I4*이며 이를 2비트의 코드로 압축을 한다. 총 4종류의 2비트(00,01,10,11) 중 3종류의 비트는 압축된 코드를 나타내고 남은 하나는 mark를 표시한다. 이 mark가 나온 바로 다음에 등장하는 명령어들은 압축이 되지 않았음을 나타낸다. [Figure 1]에서 이러한 압축을 지원하는 아키텍쳐를 보여주고 있으며 여기에서 M은 mark를 의미한다. 이진 코드를 할당하는 문제는 결국 00, 01, 10, 11을 *I1*, *I2*, *I4*, M에 할당하는 문제로 귀결된다. [Table I]의 마지막 두 컬럼에서 보듯이 총 24가지의 이진수 할당의 경우 중 최선의 경우과 최악의 경우를 생각할 수 있으며, 이는 이진코드를 효과적으로 할당하는 것이 버스에서 발생하는 스위칭 활동을 줄이는 데에 얼마나 중요한 영향을 끼치는지 알 수 있게 해준다.

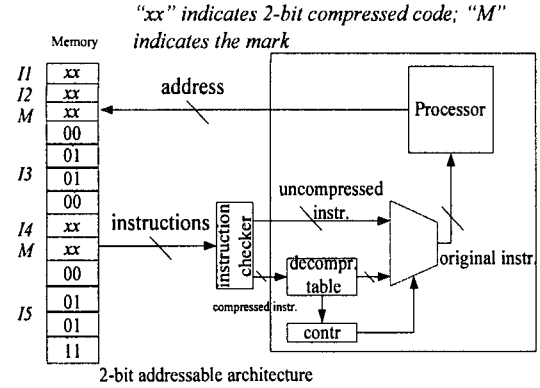
[TABLE I] A summary of distinct instructions run on a small program

Instructions	Original code	No. of access	Instructions compressed by [2]	Best code assignment	Worst code assignment
I1	00 11 10 11	6	Yes	00	11
I2	01 00 00 01	5	Yes	01	00
I3	00 01 01 00	2	No	-	-
I4	00 11 10 01	3	Yes	10	01
I5	11 01 01 00	2	No	-	-

Access sequence
S: *I1-I2-I1-I2-I3-I4-I1-I2-I3-I5-I1-I4-I2-I1-I4-I2-I1-I5*

본 연구는 정보통신부 ITRC 프로그램의 지원으로 이루어졌습니다.

[Figure1] Architecture supporting instructions in [Table I]



3. Context-Aware Code Assignment for Low Power : Problem Definition

P, S, R을 각각 실행할 프로그램, P 에서 실행할 명령어 순서, P에 존재하는 각각의 기계 명령어들의 집합이라고 하자. 또한 L (8비트로 압축)을 압축되는 명령어의 집합이라고 하며 이 때 원래의 명령어는 32비트라고 가정하자 (즉, $|L| = 255$, 메모리는 byte-addressable). 그렇다면 여기서 풀어야 할 문제는 다음과 같이 정의된다.

<u>코드 할당 문제</u>: *S, R, I(Instructions)*가 주어졌을 때 SW_{tot} (총 스위칭 카운트)의 값을 최소화하는 명령어와 mark에 할당할 8비트 이진코드를 찾아라(이 때 mapping(함수 g)은 one-to-one이고 onto mapping).

$$SW_{tot} = \sum_{\forall (I_i, I_j) \in R} f_{i,j} \cdot SW(I_i, I_j) \qquad (1)$$

이 때 $(I_i, I_j) = (I_j, I_i), f_{i,j}$는 S에서 I_i, I_j,또는 I_j, I_i의 순서를 따르는 연속된 접근의 횟수를 말하며 $SW(I_i, I_j)$는 다음과 같이 정의된다. 이 때 $H(a,b)$는 a와 b사이의 Hamming distance를 말하며 $I_S^{r1:r2}$는 명령어 I_S 에서 $r1$에서 $r2$까지 비트의 위치를 나타낸다.

$$SW(I_i, I_j) = \begin{cases} H(g(I_i), g(I_j)), \ldots\ldots I_i \in L, I_j \in L (case1) \\ H(g(I_i^{31:24}), g(I_j)), \ldots I_i \notin L, I_j \in L (case2) \\ H(g(I_i), g(M)) + H(g(M), I_j^{7:0}) + H(I_j^{7:0}, I_j^{15:8}) + H(I_j^{15:8}, I_j^{23:16}) + H(I_j^{23:16}, I_j^{31:24}), \ldots\ldots I_i \in L, I_j \notin L (case3) \\ H(g(I_i^{31:24}), g(M)) + H(g(M), I_i^{7:0}) + H(I_j^{7:0}, I_j^{15:8}) + H(I_j^{15:8}, I_j^{23:16}) + H(I_j^{23:16}, I_j^{31:24}) \ldots\ldots I_i \notin L, I_j \notin L (case4) \end{cases} \qquad (2)$$

4. The Proposed Approach

직관적으로 봐서 코드 할당의 최적의 해결책은 (1)식에서 보듯이 매핑함수 g에 대해서 모든 가능한 경우를 소모적으로 찾아 SW_{tot}의 값을 구해 보는 것이다. 그러나 여기서는 256! 만큼의 계산이 필요하며 이런 방식으로 해결책을 찾는 것이 불가능하다. 따라서, 이 문제를 좀 더 효율적으로 해결하기 위해서 그래프에 기반한 휴리스틱인 ***Compressmap-Ip***라는 것을 제안하고자 한다. ***Compressmap-Ip***는 두 단계의 해결책을 제시하는데 첫번째인 preprocessing 단계-명령어들 사이의 관계를 분석해 그래프 문제로 변환시키는 작업-와, 두번째인 code assignment 단계-압축을 위해 선택된 명령어들에게 이진 코드를 할당하는 작업-로 이루어진다. 이 두 단계에 대해 자세히 알아보자.

A. Initial Phase: Instruction Access Graph

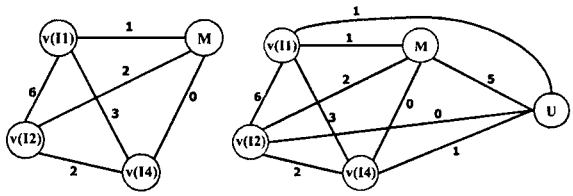
첫 단계에서는 다음과 같은 세가지 일을 한다.

(1) Instruction access graph 생성 : 접근 순서를 나타내는 S와 압축된 명령어들의 집합인 L로부터 instruction access graph인 $G(V,W)$를 만든다. $G(V,W)$는 complete weighted graph이며 노드 집합 V는 256개(255개의 명령어와 1개의 mark)로 되어 있고 만든다. 노드 v_i 와 노드 v_j 사이의 weight, $w(v_i, v_j)$ 는 두 노드간 연속적 접근횟수를 나타낸다. 즉, v_i 가 나오고 바로 뒤에 v_j 가 나오는 경우나 그 반대의 경우를 얘기하는 것이다. 아래의 [Figure 2]의 (a)는 [Table I]을 참조해 instruction access graph를 그려놓은 예제이다.

(2) Extended access graph 생성 : 실행되는 명령어들을 모두 고려하여 처음 생성한 그래프를 확장하고 새로운 두개의 노드를 포함하였다([Figure 2]의 (b)). 여기서 M은 mark를 뜻하고, U(U-node)는 압축되지 않은 노드들의 대표값을 나타낸다.

(3) U-node의 이진코드 결정 : $B_1, B_2, \ldots, B_i, \ldots B_n$을 압축되지 않은 명령어들의 8비트 벡터의 이진수 값의 순서라고 하자(즉, $B_i = (b_{i,7}, b_{i,6}, \ldots, b_{i,0})$). 이 때, $\overline{B}_U = (\overline{b}_7, \overline{b}_6, \ldots, \overline{b}_0)$로 정의되며 이때 $b_i = 1/n \sum_{j=0}^{n} b_{j,i}, i = 0,\ldots,7$ 이 된다. 결론적으로 $\overline{b}_i$ 값은 위치 n에 대해서 평균값을 나타내며 $\overline{B}_U$를 U-node의 대표값으로 쓸 수 있게 된다.

[Figure 2]



(a) Instruction access graph construction **(b) Extended access graph construction**

B. Code Assignment Phase

(1) Cost function formulation : 주어진 확장 그래프에서 식(1)을 변형하여 다음과 같이 문제를 다시 정의할 수 있다.

$$\hat{SW}_{tot} = \sum_{\forall(v_i,v_j)\in G} w_{i,j} \cdot \hat{SW}(I(v_i), I(v_j)) \quad (3)$$

여기서 $I(v_i)$는 노드 v_i에 따라 명령어, mark, 또는 U-node가 될 수 있으며 $\hat{SW}(I_i, I_j)$는 네 가지 경우가 생긴다

$$\hat{SW}(I_i, I_j) = \begin{cases} H(g(I_i), g(I_j)), \ldots\ldots I_i \in L, I_j \in L (case1) \\ H(\bar{B}), g(I_j)), \ldots\ldots I_i \notin L, I_j \in L (case2) \\ H(g(I_i), g(M)) + \\ H(g(M), B), \ldots\ldots I_i \in L, I_j \notin L (case3) \\ H(\bar{B}, g(M)) + \\ H(g(M), B) \ldots\ldots I_i \notin L, I_j \notin L (case4) \end{cases} \quad (4)$$

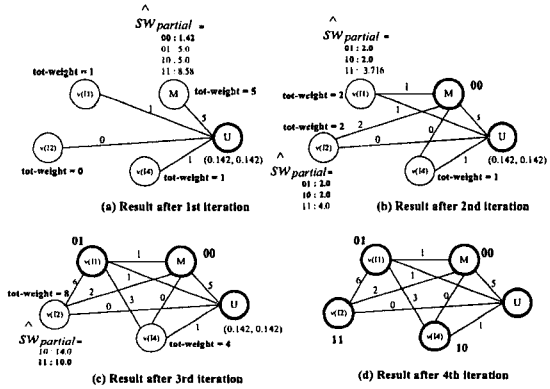
(2) Binary Code Assignment : 제안하는 알고리즘은 U-node를 시작으로 일련의 반복 과정을 거치게 된다. X를 V중에서 이미 이진 코드가 할당된 노드들의 집합이라고 하자(따라서 초기에 X = {U-node}가 된다). 한번의 반복마다 X에 존재하지 않는 v_i에 대해서 다음의 값을 계산한다.

$$tot\_weight(v_i) = \sum_{\forall v_j \in X} w_{v_i, v_j} \quad (5)$$

그리고, X에 존재하지 않는 노드들 중에 가장 큰 *tot_weight*값을 갖는 노드를 하나 선택하고 하나의 이진코드를 할당하기 위해 가능한 매핑에 대해 다음의 값을 계산한다. ($O(N)$:N=256)

$$\hat{SW}_{partial}(I(v_i), g(\cdot)) = \sum_{\forall v_j \in X} w_{v_i, v_j} \hat{SW}(I(v_i), I(v_j)) \quad (6)$$

가능한 모든 매핑에 대해 $\hat{SW}_{partial}(I(v_i), g(\cdot))$값을 구한 다음 가장 작은 $\hat{SW}_{partial}(I(v_i), g(\cdot))$값을 갖는 매핑에 해당하는 이진 코드를 할당하게 된다. 이제 X에 새로운 노드가 추가되었으므로 X를 갱신하게 된다. 이 과정을 X가 V와 같아질 때까지 계속하게 되면 결국 모든 노드에 이진 코드가 할당되게 된다. 결국 가장 큰 *tot_weight*값을 갖는 노드를 찾는 데 $O(N)$의 복잡도가 발생하고 X=V일 때까지 반복을 하게 되면 역시 $O(N)$만큼의 복잡도가 발생하게 된다. 따라서 알고리즘의 복잡도는 $O(N^2)$가 된다. 다음의 **[Example]**은 각 단계별로 알고리즘을 적용하는 과정과 결과를 보여주고 있다.



[Example] 알고리즘의 반복적인 적용 결과

5. Experimental Result

실험을 위하여 Armulator[3]를 이용하여 .trc 파일(실행되는 명령어들의 메모리 주소를 포함하는 파일)을 생성한 후 그것을 분석하여 알고리즘을 적용하였다. [표1]은 여러 개의 벤치마크 프로그램[4]을 돌렸을 때 *compressmap-lp*가 random(압축된 명령어와 mark에 순차적으로 이진 코드를 할당)과 greedy(가장 자주 등장하는 연속된 두 개의 명령어에 한 비트만 차이가 나게 Graycode를 할당)보다 더 좋은 성능을 가진다는 것을 보여주고 있다. 우리가 제시한 휴리스틱이 평균적으로는 random보다는 53%, greedy보다는 23%정도의 스위칭 활동이 줄어듦을 알 수 있다.

[표1] **Comparisons of switching activity produced by map-random, map-greedy, and our Compressmap-lp** (단위 : 백만)

BenchMark	random	greedy	Compres smap-lp	% random/ greedy
BWT	48.1	31.4	29.0	39.7 / 7.7
FILTERBANK	36.9	19.2	13.9	62.4 / 27.7
DHRYSTONE	27.1	16.2	15.1	44.3 / 6.6
MATRIXMULT	14.3	7.1	4.9	65.6 / 30.7
MATRIX_FAST	14.6	12.2	5.0	65.7 / 58.9
SIMPITERATOR	30.6	16.9	15.4	49.6 / 8.7
LOOPARY	56.2	39.0	30.8	45.2 / 20.1
Average				**53.2 / 23.0**

6. Conclusion

본 연구에서는 메모리와 프로세서 간의 명령어 버스에서 발생하는 에너지를 최소화하는 방법으로 명령어를 압축하고, 압축된 명령어들에 효과적으로 이진코드를 할당하는 기법을 소개하였다. 관련 연구들이 코드 크기를 줄이고 메모리 사용량을 최소화하는 연구에 초점을 맞춘 것에 비해, 본 논문에서는 압축된 코드에 좀 더 효율적이고 효과적인 방법으로 이진 코드를 할당하여 전력 소비를 줄여보고자 하였다.

우리는 이 문제를 분석하여 그래프 최적화 문제로 전환시켜 점진적인(반복적인) 노드 커버링 기법을 사용하여 부분적으로는 효율적인 전체적으로는 효과적인 이진 코드를 할당하였다. 여러 벤치마크 프로그램을 이용한 실험에서 그 효과가 나쁘지 않음을 알 수 있다. Random에 비해서는 평균적으로 53.2%가 greedy(Graycode based)에 비해서는 평균적으로 23%정도의 스위칭 활동 감소가 이루어진 것을 알 수 있다. 이를 통해 스위칭으로 인해 발생하는 전력소모를 줄일 수 있는 성과를 얻게 되었다.

References

[1] Y. Yoshida and *et al.*, "An object code compression approach to embedded processors," *Proc. ISLPED*, 1997.
[2] L. Benini, A. Macii, E. Macii, M. Poncino, "Minimizing memory access energy in embedded systems by selective instruction compression," *IEEE TVLSI*, vol. 10, 2002.
[3] *http://www.arm.com/products/DevTools/software_development.html*
[4] *http://www1.physik.tu-muenchen.de/jfranpsc/languagebench.html, http://datacompression.info/SourceCode.shtml, http://archi.snu.ac.kr/realtime/benchmark.*