거리 벡터(Distance-Vector)를 이용한 ARM Thumb 코드 압축

안영훈⁰, 문성림, 위영철, 김동윤
아주대학교 정보통신대학원
{amouse⁰, riea, ycwee, dykim}@ajou.ac.kr

ARM Thumb Code Compression using Distance-Vector

Younghoon Ahn⁰, Sungrim Moon, Youngchul Wee, Dongyoon Kim
Graduate School of Information and Communication of Ajou University

요 약

임베디드 시스템에서의 코드 압축은 효율성 제고를 위한 필수적인 기법이다. ARM, MIPS등 많은 프로그램 코드에서 현재 시도되고 있으나 한계를 나타내고 있다. 특히, Arm Thumb코드는 다른 코드 압축과 달리, 아직까지 15%~20%정도의 압축 효율을 보이고 있다. 본 논문은 다양한 값을 갖는 코드의 데이터 이지만, 일정 부분에서 특정 값의 발생빈도가 높은 Thumb 코드의 분포를 분석, 그 특성을 활용하였다. 즉, 현재 압축하고자 하는 필드의 값을 코드의 앞부분과 비교해 나가면서, 유사도를 분석 및 압축하고, 거리 정보를 기록하는 방식의 거리 벡터 기법의 압축방법을 고안, 적용하였고, 그 결과 압축효율이 20~25%로 기존의 방법에 비하여 약 5%정도의 효율 향상을 가져왔다.

1. 서 론

임베디드 시스템은 파워, 비용 그리고 저장 공간의 제약을 가지고 있다[5]. 특히, 한정된 저장 공간은 다 수의 기능을 구현하는데 많은 제약을 주고 있는데, 이를 해결하기 위하여, 코드 압축 방법이 제시, 사용된다. 특히, 현재 가장 활발한 연구를 보이는 부분은 핸드폰, 포켓용 전자장치 같은 휴대 장비에 성공적으로 적용된 ARM 코드의 압축이다. ARM의 컴파일 방식으로는 32bit 어셈블러로 컴파일 되는 방식과, 16bit 어셈블러로 컴파일 되는 Thumb 방식으로 구분된다. 이 중, Thumb 방식의 경우에는 압축 효율이 15~20% 정도에 머무르고 있다[1]. 이는 Thumb 코드 자체에 Opcode와 Operand가 조합된 다양한 인스트럭션 셋을 가지고 있어서, 압축에 유효한 상호 관련성을 도출해 내기 어렵기 때문이다.

본 논문은 적은 수의 Opcode와 다양한 Operand가 존재하지만, 소수의 Operand의 값이 좁은 범위에서 빈도수가 높게 나타나는 Thumb 코드의 특징을 이용한 거리 벡터 방법을 사용하였다. 이 방법은 일부분의 코드를 일정한 범위의 앞의 코드와 비교/매칭 하는 압축 방법으로, Operand 부분의 일치성을 극대화 하였다. 그 결과 기존 방법에 비해 약 5%정도의 압축효율 향상을 보였다.

2. 기존의 압축방법

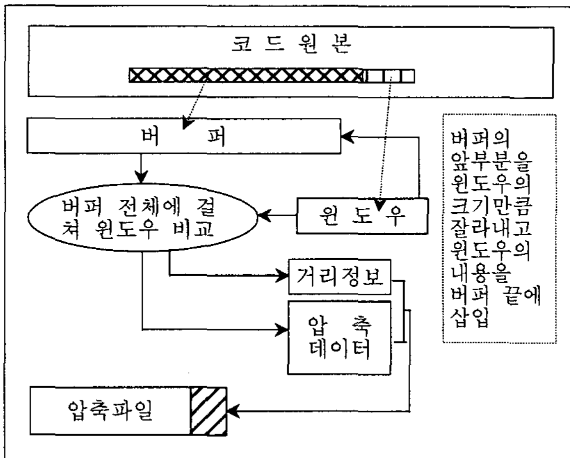
기존의 코드 압축 방법은 이미지와 데이터 압축방법을 활용한 방법으로, 코드를 블록으로 분할하여 PPMZ, LZS, ENC, XMatchRLI, XmatchVW, LZAri, Huffman 같은 다수의 압축 기법을 활용하는 Block bounded compression[4]기법으로, Thumb 코드에서 15~20% 압축 효율을 보여준다[1]. 하지만 이 방법은 많은 압축 기법을 활용하여야 하기 때문에 인코더와 디코더 프로그램이 복잡하고 압축해제 속도가 느리다는 단점이 있다.

3. 거리 벡터(Distance-Vector) 압축 기법

본 논문에서 적용된 거리 벡터 압축 방법은, 슬라이드 윈도우를 이용하여 압축하려는 코드 부분을 기준으로 앞쪽의 코드와 비교/매칭 하는 방법으로, 그 코드의 유사성과 거리를 고려한 최소값을 구하여, 그 값을 압축 코드에 저장한다. 기계어의 경우 하나의 함수를 처리하기 위해 대체적으로 2~3개의 명령어를 사용해야 하며, 그 안에 사용되는 대부분의 명령어의 종류는 매우 적으면서도 발생빈도가 높다는 특징을 이용한 것이다. 또한, 기계어에서의 Operand부분은 특정 레지스터 주소를 많이 사용하는데, 그 사용 주소 값이 동일한 값을 가진 경우가 일정부분에서 많이 나타난다.

3.1 압 축(Encoding)

압축 방식의 전체적 구조를 표현하면 아래 와 같다.



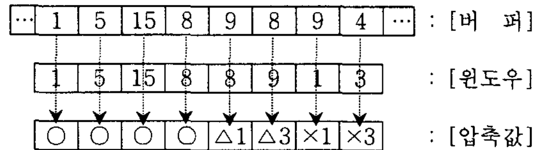
[그림 1] 전체 압축 방법의 구조

위의 그림은 코드 원본으로부터 버퍼에 사용될 데이터와 윈도우에 들어갈 데이터를 가져온 후 압축 과정을 거쳐 그 값을 압축파일에 저장하는 전체적인 모습을 나타낸 것으로, 위의 과정을 순서적으로 기술 하면 다음과 같다.

1. 원본코드에서 앞부분의 일정량을 버퍼로 가져온다.
2. 원본코드에서 버퍼 이후의 데이터를 윈도우로 가져온다.
3. 윈도우의 값을 버퍼와 비교하며 유사성이 최대로 높은 블럭을 찾는다.
4. 선택된 블럭의 거리정보와 압축데이터를 압축파일에 추가한다.
5. 버퍼의 앞부분에서 윈도우의 크기만큼을 제거 하고, 기존 윈도우 값을 버퍼의 뒷부분에 삽입한다.
6. 원본 파일로부터 다음 윈도우 값을 가져온다. 3번부터 반복.

3.1.1 압축데이터

데이터의 압축은 버퍼와 윈도우의 바이트 단위로 비교하여, 그 압축 결과를 표기하고 윈도우 내부 각 바이트들의 압축 결과 합이 윈도우의 압축된 크기가 된다. 압축데이터는 일치, 불일치 그리고 1bit가 상이한 일치 이렇게 세 가지로 분류하여 구분된다. 즉, 윈도우 안의 전체 바이트가 버퍼의 일부분에 정확하게 매칭이 되면 일치로 판정, 1bit로 유사도를 설정하고, 한 byte중 1bit가 다른 경우 1bit가 상이한 일치로 판정하여 5bit를 설정하며 그 이상, 여러 bit가 서로 상이한 값을 갖게 되면 불일치로 판정하여 10bit를 설정 하였다.



[그림 2] 압축 기법

[그림 2]에서 버퍼와 윈도우 각 칸의 숫자는 바이트 값을 의미하며, 위의 예시를 분석하면, 오른쪽부터 4개의 바이트는 일치, 그 후 2개의 바이트는 이진 표현 시 1bit가 상이한 값이므로 구분자 2bit와 상이한 값의 위치를 3bit를 이용하여 표시 한 값을 기록하고, 마지막 2바이트는 불일치 값으로, 구분자 2bit와 윈도우 안의 데이터를 기입하게 된다.

윈도우를 버퍼의 거리 정보 1부터 최대범위 값까지 이동하면서, 위의 [그림 2]와 같이 처리하여 각 거리 정보별 압축값을 계산하고 그중 최소값을 찾은 위치의 거리 정보를 저장하고 압축값을 저장한다.

이런 방식으로 전체 파일을 압축하여 압축파일을 생성한다.

3.1.2 거리 정보

거리 정보는 버퍼의 끝부분부터 앞으로 이동하면서 값을 증가시킨다. 이렇게 앞으로 이동하면서, 압축데이터의 크기가 가장 작아지는 지점을 거리 정보로 표시하게 된다. 거리 정보 값은 버퍼의 크기를 얼마로 잡는 가에 따라서 정보 저장 필드의 크기가 달라지는데, 본 논문에서는 이 거리 정보의 오버헤드를 감소시키기 위해 거리 정보 표시를 4단계로 나누었고, 그 값은 다음의 표와 같다.

[표 1] 거리 정보 표시 값

식별자	사이즈(bit)	수용범위(byte)
00	3	~8(8)
01	4	~24(16)
10	5	~56(32)
11	10	~344(256)
11	11	~600(512)
11	12	~1112(1024)
11	13	~2136(2048)
11	14	~4184(4096)

거리 정보를 표시하기 위해 식별자를 사용하여 필드 크기의 낭비를 감소시켰으며, 00~10까지는 고정적인 값으로 정하였고, 식별자 11의 값을 변화시켜 가며 압축파일 전체의 용량이 최적화 되는 지점을 찾았다.

4. 실 험

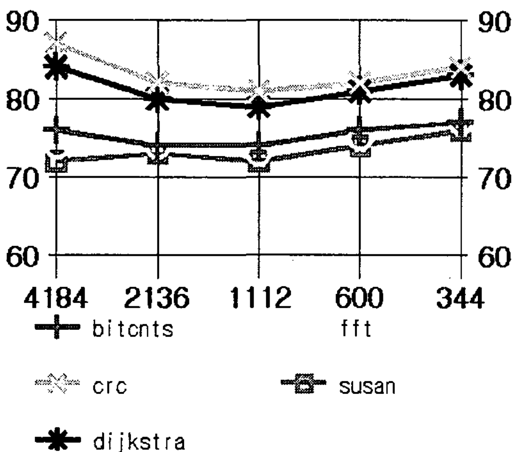
본 실험은, Intel pentium4 2.8Mhz, RAM 1Gb, WindowXP sp2의 시스템 환경에서 jdk1.5/eclipse3.1을 사용하여 인코더와 디코더를 구현 하였으며, 버퍼와 윈도우의 크기를 변경하며, 최적의 결과 값을 추출하였다. 샘플 파일은 MiBench[2]의 Arm binary image의 code 부분을 Arm Developer Suite(ADS) version 1.2를 사용하여 컴파일하여 사용하였다.

이 실험에서는 윈도우의 크기를 8로 고정시켰는데, 그 이유는 8보다 큰 경우에는 윈도우의 매칭효율이 떨어지고, 8보다 작은 경우에는 윈도우의 크기에 비해 거리 정보 값의 크기가 차지하는 비율이 많아 효율이 떨어졌다.

[표 1] MiBench 프로그램 사이즈

파일명	코드 사이즈(byte)
BitCounts	18996
CRC	9116
Dijkstra	12276
FFT	26476
Susan	43788

실험 결과는 4184, 2136, 1112, 600, 344로 버퍼 크기를 조정하며 압축파일의 효율을 측정하였으며 그 결과는 다음의 [그림 3]과 같다.



[그림 3] 복원 가능한 압축파일 크기

위의 그림은 5가지의 thumb파일을 버퍼의 크기를 변경시켜 가면서 측정한 값을 그래프로 표현한 것이다. 그림에서 볼 수 있듯이, 압축 효율은 버퍼의 크기가 1112와 2136인 지점에서 가장 효율적으로 측정되었으며, 이는 코드의 크기가 작고, 버퍼의 크기가 늘어날수록, 거리 정보 필드의 사이즈 증가로 압축 효율성이 떨어지는 것으로 사려 된다.

5. 결 론

대체적으로 현존하는 Thumb기반의 코드 압축 기법이 15~20%의 압축 효율이 나타나는데 비해, 본 논문에서 사용한 거리 벡터 기법으로 압축 시에는 20~25% 가량의 압축률이 나왔다. 코드의 특성을 사용하여 앞쪽의 코드에서 유사한 코드를 찾는 방법이 이미지나 데이터에서 사용하는 압축 방법의 단점을 극복할 수 있다. 추후, 디코딩의 수행 속도 향상을 위한 연구가 요망된다.

6. 참고문헌

1. X.H.Xu, S.R.Jones, C.T.Clarke, "ARM/THUMB Code Compression for Embedded Systems", ICM 9-11, Cairo, Egypt, Dec. 2003.

2. M.R.Guthaus, J.S.Ringenberg, D.Ernst, T.M. Austin, Trevor Mudge, Richard B. Brown, "MiBench: A free, commercially representative embedded benchmark suite", IEEE 4th Annual Workshop on Workload Characterization, Austin, TX, Decemer 2001.

3. Jens Ernst, William Evans, Christopher W. Fraser, Steven Lucco, Todd A. Proebsting, "Code Compression", PLDI '97 Las Vegas, NV, USA.

4. Andrew Wolfe, Alex Chanin, "Executing Compressed Programs on An Embedded RISC Architecture", 1992 IEEE.

5. Charles Lefurgy, Peter Bird, I-Cheng Chen, Trevor Mudge, "Improving Code Density Using Compression Techniques", In Proc. 30th International Symposium on Microarchitecture, 194-203, Dec. 1997.