Code compression techniques for low powered embedded systems

Utkarsh Ojha
Department of Computer Science and Engineering
MNNIT Allahabad
Email: utkarsh2254@gmail.com

Prashant Vissa
Department of Computer Science and Engineering
MNNIT Allahabad
Email: prashantvissa@gmail.com

***Abstract*—With the enhancement in embedded system technology and high performance computing, the requirement of memory is increasing to a great extent which in turn is increasing the stress on power constraints. For efficiently deploying the embedded systems in smaller devices, the power consumption has to be reduced. In this paper, we provide a detailed review of some of the recent techniques that have been used for code compression, which can lead to reduced memory access, and eventually can lead to lower power consumption. At the end, we propose our own idea of combining the idea of bit masking with a machine learning approach that can further help in reducing the number of bits required to represent some data.**

Keywords*—Code compression, Embedded systems, Power consumption.

I. Introduction

Embedded systems are becoming increasingly popular as more and more consumers wish to carry electronic devices with them. For many people, handheld devices such as cellular phones, pagers, digital cameras, PalmPilots, and PocketPCs have become like wristwatches; the owner simply would not go anywhere without them. It is no surprise that some people feel as if embedded systems will become the next major computer science research field that will follow the current boom of interest in networks, and past interest in parallel processing. Embedded devices popularity conceals the constraints that designers face when developing for these systems . Embedded systems that drive handheld devices are designed to be small so as to be comfortable to hold and carry. These systems are targeted toward large volume sales, hence they are cost-sensitive. Cost is strongly associated with size of the integrated circuits (ICs) used in a device. Much of an embedded systems IC space is devoted to memory for program code and data. Many embedded systems are further limited by the need to operate off a limited power source, usually batteries , hence power consumption is a major concern when designing their components. Larger storage, memory, and cache sizes, and faster CPUs all contribute to space, cost, and power use. Because of these factors, memory and storage space both are limited in handheld devices, affecting designers, developers and users. Hence, study of techniques... has become an important topic....

A. Code compression and its benefits

Code compression, and in general any data compression schemes try to reduce the number of bits required to represent data. This can be used to reduce the size of programs in memory, and in doing so it can reduce the size of components needed, lowering power requirements, size, and cost. Basic principle of code compression techniques is to merge multiple instructions into one single instruction, and store it in a separate database.

Code compression has many benefits. The reduced program size can be used to reduce the size of storage necessary on ICs. This can save production costs. Since memory consumes a significant amount of an embedded systems power and power consumption is proportional to IC area, battery life can be extended. Smaller chips also have lower capacitance, which lowers power consumption. When a cache miss occurs, the data brought to the instruction cache (I-cache) is loaded from main memory compressed, so less data needs to be transmitted on the bus between memory and I-cache. If the code is compressed in the I-cache, then less data is sent out from the I-cache with each instruction fetch. Thus, fewer bit toggles are necessary on bus lines, which reduces power consumption. Furthermore, if code in the I-cache is uncompressed, then less data is transmitted on the bus from memory to the I-cache, so the cache may be filled sooner. If code in the I-cache is compressed, then it is able to hold many more instructions, reducing the cache miss rate. Both of these cases cause the CPU to pause for less time due to cache misses. Since the CPU is the main consumer of power, compression can result in significant power savings. If memory and or cache are made smaller, their effective capacitance decreases further decreasing power consumption. Because there are fewer transactions and transactions are shorter, compression may also increase performance (reduce program execution time).

II. Previous code compression techniques

A. Static Huffman code based compression

Huffman coding is a bottom up method for organizing elements into a tree. The shape of the built tree determines the codeword for each element.

- A new node is created and the two smallest elements are removed from the list and made the nodes children. The probability of the new node is the sum of the probabilities of the two children.
- The new node is inserted as an element in the list of nodes, in a place corresponding to the new nodes probability. In the event of a tie, the new node is inserted before other elements of the same probability.

Thus bigger and bigger nodes are created with each iteration, until the final root node is created having the probability 1. At that point, the only element in the list represents the root node.

B. Arithmetic coding based compression

Arithmetic coding is a form of entropy encoding used in lossless data compression. Normally, a string of characters such as the words ”hello there” is represented using a fixed number of bits per character, as in the ASCII code. When a string is converted to arithmetic encoding, frequently used characters will be stored with fewer bits and not-so-frequently occurring characters will be stored with more bits, resulting in fewer bits used in total. Arithmetic coding differs from other forms of entropy encoding, such as Huffman coding, in that rather than separating the input into component symbols and replacing each with a code, arithmetic coding encodes the entire message into a single number, an arbitrary-precision fraction n where [0.0 n ¡ 1.0). It represents the current information as a range, defined by two numbers.

C. Dictionary based code compression

This is the easiest and one of the most widely technique for code compression. This method involves maintaining a separate dictionary (similar to a database). Initially, a new instruction is fetched from the code file, and checked whether it exists in the database (dictionary).If the instruction exists in the dictionary, the encoder substitutes a reference to the string’s position in the dictionary. If the instruction doesn’t exist in the dictionary then an entry for it is created in the dictionary and a code is assigned to it, which basically is the location of the instruction in the database. So the whole code file gets converted to a sequence of codes (consuming less memory than the original code file). These set of codes are then decoded and decompressed by the compiler during the run time.

D. Improved dictionary based code compression

This approach is basically an improved version of dictionary based approach. In traditional dictionary based approach, we create an entry in the database for each instruction encountered in the code file. Thus as the number of distinct instructions increases, the bits required to represent those instructions increases. In bitmask based compression technique, we just have two instructions in the dictionary, which are the most frequent instructions encountered in the code file (so as to maximize the hits while encoding). There are 3 possible cases while encoding any 8 bit instruction.

- Complete hit occurs i.e. the exact instruction is found in the dictionary. In this case, one bit is required to indicate that hit occured, one bit is required to indicate that no additional operation has to be done (explained in case 2), and one bit for the location of the matched instruction in the dictionary, making it 3 bits in total.
- Partial hit occurs: We define partial hit as the case in which modification of one bit in the 8 bit instruction will result in a perfect match with any one of the dictionary words. In this case, one bit is required to indicate partial hit has occured, one bit to indicate that additional operation has to be performed, three bits to indicate on which bit the toggling has to be performed, and finally one bit to represent the location of the matched word in the dictionary, making it 6 bit in total.
- No hit occurs: In this case, when even the modification of 1 bit won’t result in a match with any dictionary word, the 8-bit word is encoded as it is, with an additional bit to indicate no hit, taking up 9 bits to represent it.

TABLE I. DICTIONARY WORDS

Index	Word
0	00000000
1	01000010

TABLE II. ENCODING PROCESS

Input word	**Compressed word**
00000000	01 0
10000010	1 10000010
00000010	001100
01000010	01 1
01001110	1 01001110
01010010	000111
00001100	1 00001100
01000010	01 1

In this table in the compressed word column, the first bit indicates whether there is hit (complete or partial) or not (0 for hit, 1 for no hit). If the bit is 0 (i.e. hit), the second bit indicates whether any modification is required or not (0 for no modification). If there is any modification needed, the following 3 bits represent the position of the bit which needs to be toggled, followed by the bit which indicates the location of matched bit in the dictionary. If there is no hit (i.e. first bits 1), the following 8 bits represent the same 8-bits of the input word.
In this case,

- number of complete hits = 3
- number of partial hits = 2
- number of no hits = 3

Therefore, total number of bits required to represent these eight 8-bit instructions = $(3 \times 3) + (2 \times 6) + (3 \times 9) = 48$. So, on an average 6 bits are required to represent these 8-bit instructions.

E. Proposed machine learning approach combined with bit masking for code compression

While selecting the two words in the dictionary, we focus on maximizing the number of complete hit, and hence go for those two words which occur in highest frequency. While this may give optimal results in many cases, it doesn’t guarantee optimal results every time. The overall objective should be to minimize the total number of bits required to represent the code file, rather than to maximize the number of complete hits. Based on this ideology, we provide a machine learning approach in which we’ll try to minimize the cost function which is basically the total number of bits required to represent

TABLE III. DICTIONARY WORDS

Index	Word
0	00000000
1	01000010

TABLE IV. INPUT CODE FILE

Input word	Compressed word
00000000	01 0
10000010	0000111
00000010	0011100
01000010	01 1
01001110	0100000
01010010	0001011
00001100	0010110
01000010	01 1

a complete code file. Additionally, we take inspiration from the modified dictionary based technique and extend it so that modification of two bits is possible instead of just one bit, in the case of partial hit. Thus, 3 possible cases while encoding any 8-bit word in our method will look like this:

- Complete hit occurs i.e. the exact instruction is found in the dictionary. In this case, one bit is required to indicate that hit occured, one bit is required to indicate that no additional operation has to be done (explained in case 2), and one bit for the location of the matched instruction in the dictionary, making it 3 bits in total.
- Partial hit occurs: We define partial hit as the case in which modification of two consecutive bits in the 8 bit instruction will result in a perfect match with any one of the dictionary words. In this case, one bit is required to indicate partial hit has occured, one bit to indicate that additional operation has to be performed, two bits to indicate on which set of two bits the modification has to be performed, two bits to indicate what two bit number should be xor-ed with the previous two bit number, and finally one bit to represent the location of the matched word in the dictionary.
- No hit occurs: In this case, when even the modification of 2 bits won't result in a match with any dictionary word, the 8-bit word is encoded as it is, with an additional bit to indicate no hit, taking up 9 bits to represent it.

Hence, we define the cost function as follows:

$$cost = (3 \times complete{-}hits) + (7 \times partial{-}hits) + (9 \times no{-}hits) \tag{1}$$

This cost is parameterized by the bits of the two 8-bit numbers. To start with, we'll initialize the two 8-bit numbers with the two most frequent 8-bit words in the code file. We can then choose an optimizer like gradient descent or Adam optimizer, or we can manually toggle the 16 bits so as to minimize the cost as much as possible.

In this case,

- number of complete hits = 3
- number of partial hits = 5

Therefore, total number of bits required to represent these eight 8-bit instructions = $(3 \times 3) + (5 \times 7) = 44$.
So, on an average 5.5 bits are required to represent these 8-bit instructions. Compared to the previous method, in which the average number of bits required to represent same 8-bit instructions was 6, our approach performs better for this test case.

REFERENCES

[1] S.W. Seong and P. Mishra, Bitmask-Based Code Compression for Embedded Systems, 3rd ed. Harlow, England: Addison-Wesley, 1999.

[2] W.J. Wang and C.H. Lin, Code Compression for Embedded Systems Using Separated Dictionaries, 3rd ed. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 2016.

[3] D. Ingenieurwissenschaften, Huffman-based Code Compression Techniques for Embedded Systems, PhD Thesis, 2009.

[4] I. Witten, R. Neal, Arithmatic coding for data compression, Computing Practices, 2002

[5] S. Seong, Dictionary based code compression, PhD thesis, University of FLorida, 2006