

Analysis of Two Code Compression Algorithms for Embedded Systems

Yuan Xie

Worldwide Design Center,
IBM Microelectronics Division
Essex Junction, VT, USA
yuanxie@us.ibm.com

Abstract

Embedded systems are space and cost sensitive. Decreasing the program size is an important goal for embedded system design. Code compression is proposed to tackle this problem. In this paper, we present analysis of two code compression algorithms based on variable-to-fixed (V2F) coding schemes: one is based on Tunstall coding and the other one is based on arithmetic coding. The paper also gives the compression lower bound by using two statistical models: one is a static model and the other one is a Markov model.

1. Introduction

Embedded computing systems are space and cost sensitive. In many embedded systems, the cost of RAM or ROM often outweighs that of the main processors, which poses serious constraints on program size. On the other hand, many embedded processors are using RISC architectures or VLIW architecture such as IBM PowerPC or TMS320C6x. Compared to CISC processors, RISC and VLIW processors have a "code bloating" problem. Figure 1 shows the code size of an MPEG2 encoder compiled for different architectures. The Intel x86 is a typical CISC architecture; Thumb is a modified RISC architecture to improve code density; SHARC is a RISC architecture from Analog Devices; TMS320c6x is a VLIW architecture from Texas Instruments; IA-64 is a VLIW instruction set adopted by Intel and HP.

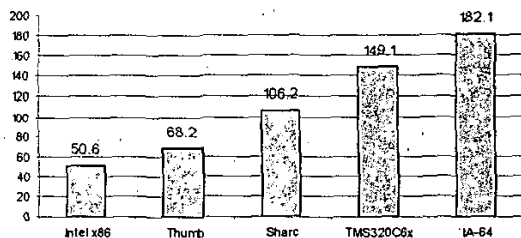


Figure 1. The code size (in KBytes) of an MPEG2 encoder for different architectures

Code compression, which refers to compressing the executable program off-line and decompressing it on-the-fly during execution, was first proposed [1] in the early 90's to address the code size constraint problem. This approach requires no change of compiler or processor architecture. It can also be combined together with compiler

techniques to achieve further code size reduction. One industry example using code compression is IBM's PowerPC embedded Processor. Figure 2 shows an IBM PowerPC 405 platform-based SOC (System-on-a-chip) design [2]. The compressed code is stored in the program memory and a decompression module called CodePack [3] is put between the high-speed memory controller (HSMC) and the Processor Local Bus (PLB). The instructions are decompressed whenever there is a cache miss. The PowerPC 405 processor fetches uncompressed instructions from the I-cache and the compression/decompression scheme operates in a manner transparent to the processor.

There have been various approaches to code compression. A comprehensive survey has been written by Lekatsas *et al.* [4]. Philips Research Lab also maintains a complete code compaction bibliography [5].

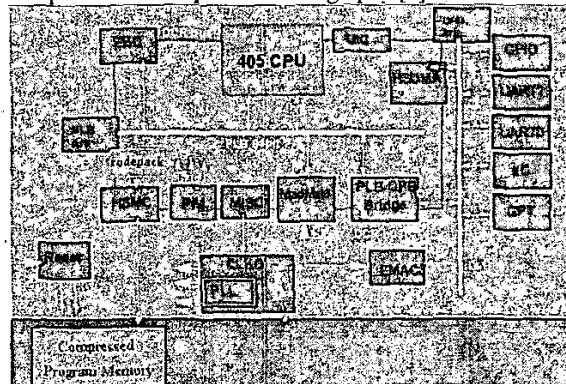


Figure 2. An IBM PowerPC 405 platform-based SOC using code compression.

2. Probability models and compression algorithms

We proposed two code compression algorithms [6][7], which were the first code compression algorithms using Variable-to-fixed coding. In this section, we briefly describe these two algorithms and the probability models.

2.1 Probability model

In our research, we use two different models for code compression. The first model is a *static iid model*, which is a fixed model that fits all applications and assumes ones and zeros in the code have independent and identical distribution (*iid*). The probability model is determined in advance from some sample applications. For example, our experimental results indicate that for TMS320C6x,

with each edge instead of a fixed probability for bit 0 and bit 1. Also, for each codebook entry, we have to indicate what the next state it is.

Each state in the Markov model has its own codebook. Therefore, for a M-state Markov model using a N-bit variable-to-fixed length codes, the entry number of all the codebooks is $M \cdot 2^N$.

3. Algorithm analysis

In section 2, we briefly describe the compression algorithms and the probability models. In this section, we discuss the compression lower bound and compare these two algorithms in terms of compression ratio and decompression unit design.

3.1 Compression lower bound

It is well known that the entropy is the lower bound for the compression. Because the entropy is a property of a probability model, the lower bound for the compression is decided by the probability model used in the compression procedure. For static iid model, the entropy is $-p(0) \cdot \log_2 p(0) - p(1) \cdot \log_2 p(1)$. For TMS320C6x, since $P(0)=0.75$, the entropy is 0.811.

For a semi-adaptive Markov model, the entropy calculation is not as straightforward as static iid models. We denote a state in the Markov model by s and let S be the set of all states in the Markov model. $P(s)$ is the probability of state s is visited, while $P(1|s)$ ($P(0|s)$) is the conditional probability of the next bit is 1(0) when current state is s . The entropy of a Markov model is represented by the following equation:

$$-\sum p(s) \cdot (p(1|s) \cdot \log_2 p(1|s) + p(0|s) \cdot \log_2 p(0|s))$$

Figure 6 shows the average entropy values of different Markov models for a set of TMS320C6x benchmarks. The instruction length for TMS320C6x is 32-bit. The experimental result shows that when the total number of states is fixed, the entropy is the lowest when the depth is set to be the instruction length. It also shows that when the model depth is fixed, the larger the width, the lower the entropy.

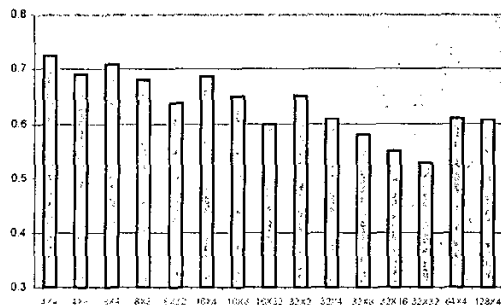


Figure 6. Entropy of different Markov model for TMS320C6x

However, in code compression, we usually cannot achieve the compression lower bound defined by the entropy. To ensure random access (i.e. decompressing prop-

erly when program changes execution flow due to branch or jump instruction), we compress the instructions block by block. For example, we choose the fetch packet (size of a VLIW instruction, 256 bits long) as the basic block for TMS320C6x. We use a coding tree, such as the one shown in Figure 4, to parse and compress each block: starting from the root node, whenever a "1" occurs, we take the right branch; otherwise, we take the left branch. Whenever a leaf node is encountered, a codeword related to that node is produced and compression procedure restarts from the initial state. For Markov V2FCC, the leaf node is associated with a Markov state, and the compression procedure jumps to the root node of the coding tree starting with that Markov state. Since we compress instructions block by block, it is very likely that the tree traversal ends at a non-unit interval at the end of the block. To avoid this problem, at the end of each block, when compression ends without reaching a unit interval, we pad extra bits to the block such that traversal can continue until a leaf node is met and a codeword is produced. During decompression, the whole block is decoded together with the extra padded bits. However, since we know the block size a priori, we simply truncate the extra bits.

To make decompression hardware simpler, and make the storage of the compressed code easier, the compressed block must be *byte aligned*. This means that if after compressing a block the result is not a multiple of 8 (in bits), a few extra bits are padded to ensure that it becomes a multiple of 8. We can thus ensure that the next compressed block will start on a byte-aligned boundary.

3.2 Compression ratio comparison

In section 3.1, we calculate the compression lower bound for two different models. Now we have to decide the codeword length N such that we can approach the lower bound as close as possible. We define L to be the average length of bit sequences represented by fixed length codewords. We denote R to be the ratio of L over the codeword length N . R defines the ideal compression ratio that the algorithm can achieve: Figure 7 plots the entropy of the input as well as the R ($R = N/L$) for different N when the probability $\text{Prob}(0)$ changes (Tunstall coding scheme using a static iid model). It shows that as N increases, R is approaching the entropy, especially when the probability is highly skewed. However, when $N > 4$, the improvement is not that significant.

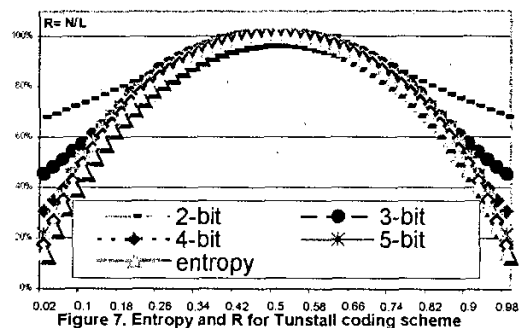


Figure 7. Entropy and R for Tunstall coding scheme

N	2	3	4	5	6
L_T	2.312	3.538	4.752	5.998	7.223
R_T	0.865	0.848	0.842	0.834	0.831
L_A	2.312	3.488	4.752	5.973	7.216
R_A	0.865	0.860	0.842	0.837	0.831
Padding	2.8%	3.1%	1.1%	2.9%	2.8%

Table 1. R and padding overhead using N-bit codeword

To compare these two V2F algorithms, we calculate the average length of the bit sequences represented by the codewords for both algorithms for TMS320C6x. The results are shown in Table 1. In the table, L_T (L_A) is defined as the average length of the bit sequences represented by an N-bit codeword using Tunstall coding (arithmetic coding) based V2F compression, and R_T (R_A) denotes the ratio of N over L_T (L_A). From these tables, we can conclude that Tunstall coding based V2F compression can always achieve the same or better compression ratio than the arithmetic coding based V2F compression, because R_T is always better than or the same as R_A . Both tables show that for Tunstall coding based V2F compression, as N increases, R is approaching the entropy, which means that the compression ratio is improved. However, the improvement is not very significant, especially after N becomes larger than four.

On the other hand, since the compression poses a byte alignment restriction for every block, by using a four-bit length codeword, the chance of padding extra bits are reduced. Table 1 also shows the padding overhead for each case, we can see that when N=4, the padding overhead is less than other cases. This explains why we achieve best compression ratio when N=4 in the experimental result shown in Figure 8.

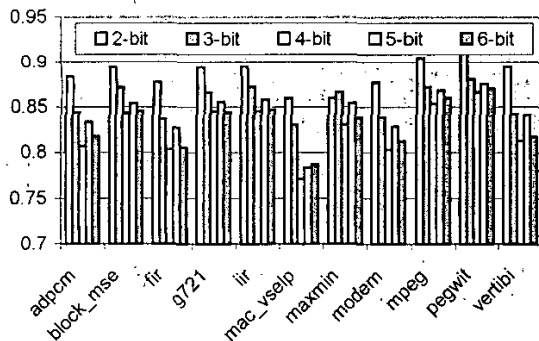


Figure 8. Compression ratio for TMS320C6x using Tunstall coding schemes and static iid model

3.3 Decompression unit design

For both algorithms, the codebook size, which is defined as the number of codewords in the codebook, increases exponentially as the codeword length N grows (codebook size = 2^N). However, the average improvement of the compression ratio is less than 1%. Intuitively, if we choose N=8, there is no need to pad extra bits

and we can get better compression ratio. Considering the codebook for N=4 has only $2^4=16$ entries, while the codebook for N=8 has $2^8=256$ entries, we conclude that the best choice for V2F code compression is to use four-bit length codewords.

Since the variable-to-fixed codebooks constructed by both algorithms have the same size, the decompression unit design has no difference and can just follow the design described in [6].

4. Conclusion

In this paper, we present analysis of two code compression algorithms that use Variable-to-fixed coding schemes. We conclude that the Tunstall-coding based algorithm is better than the arithmetic coding based algorithm in terms of compression ratio though the decompression unit complexity is the same. The compression lower bound is determined by the probability models. The depth of the Markov model should be equal to the instruction length to achieve better compression ratio. For both algorithms, using a 4-bit codeword is the best choice considering the tradeoffs between compression ratio and decompression complexity.

Reference:

- [1] T.Kemp et al. "A Decompression Core for Power PC", IBM Journal of Research and Development, September 1998
- [2] A.Wolfe and A.Chanin. "Executing Compressed Programs on an Embedded Architecture". MICRO-25 (International Symposium on Microarchitecture), 1992.
- [3] www3.ibm.com/chips/products/asics/products/soc.html
- [4] H. Lekatsas, C. Lefurgy, W. Wolf and T.Mudge. "Code Compression: methods, performance and analysis". Submitted for publication at the ACM Transaction on Embedded Computing Systems.
- [5] <http://www.extra.research.philips.com/ccb/>
- [6] Y. Xie, W. Wolf and H. Lekatsas. "Code Compression using Arithmetic Coding Based Variable-to-fixed Coding", Data Compression Conference 2003
- [7] Y. Xie, W. Wolf and H. Lekatsas, "Code Compression for VLIW using Variable-to-fixed coding". Proceedings of International Symposium on System Synthesis, 2002
- [8] B.P. Tunstall. Synthesis of Noiseless Compression Codes: Ph.D. dissertation, Georgia Institute of Technology, 1967.
- [9] P. G. Howard and J. S. Vitter. "Practical Implementations of Arithmetic Coding." invited paper in Images and Text Compression, Kluwer Academic Publishers, 1992.