Code compression for ARM7 embedded systems

V. Garofalo, E. Napoli, N. Petra, A.G.M. Strollo
Department of Electronic Engineering and Telecommunication
University of Naples "Federico II", Naples, Italy
e-mail: valeria.garofalo@fastwebnet.it

Abstract—A detailed analysis of different code compression algorithms is provided in this paper. The performances of the algorithms have been tested on ARM codes whose size is below 32KB. Code compression performances have been considered including the compression overheads due to the decoding tables, to the alignment and to the tables for random access to the compressed code.

We have analyzed Huffman, Tunstall, LZ77 and Class-based techniques. Optimal performances are provided by Class Based algorithms with an average compression ratio of 64%. For this algorithm we have realized a static decompression engine that provides, after an initial latency of three clock cycles, one 32b instruction for clock cycle.

I. INTRODUCTION

Modern System On Chip (SoC) are oriented to always more complex applications and require an increased amount of RAM to provide widened functionalities. As a consequence the amount of RAM and the number of programs that can be stored in the RAM are often the restricting factor for SoC performances. Different techniques that reduce the size of the code for embedded systems have been proposed. We focused on code compression that requires dedicated hardware for the decompression but does not require any change to the compiler or the processor architecture. Many references are available for code compression (see [1] for an overview) even if new techniques are still being proposed.

In [2] the authors propose the Compressed Code RISC Processor (CCRP), reaching 73% for the compression ratio on MIPS architecture (60KB size programs). A LZW-based code compression for VLIW Embedded System is proposed in [5]. The overall compression ratio is limited to 85%-75%, with 2KB decoding table in adding. The decompression engine is implemented in TSMC 0.25μm technology and occupies 5687μm², without considering the decoding tables. In [3], the 32b PowerPC instructions are divided into two 16b streams and compressed with Class Based algorithm, with a compression ratio of about 60%. The decompression core is implemented in 0.5μm technology and occupies 1mm².

In [6] SADC and SAMC methods are proposed. The average compression for SADC is about 52% on MIPS programs and 67% on x86 programs; SAMC compresses the ARM program with a compression ratio of about 55%. A variable-to-fixed coding scheme based on either Tunstall coding or arithmetic coding is proposed in [8]. The experimental results for a VLIW embedded processor TMS320C6x show that the compression ratio is only 80% including the decoding overhead. The silicon area for the decompression unit is about 0.42mm² for TSMC 0.25μm technology.

It is worth noting that the papers presented to date consider programs whose size is often larger than 100KB, quite often without taking into account the various overhead. In this paper a detailed analysis of different code compression algorithms, applied to programs compiled for ARM7 processor, is presented, including the overhead due to the decompression tables, and any other overhead needed for the on line decompression. The emphasis is for small size programs, less than 32KB, for which the above mentioned overheads are significant. Looking for the best, in terms of performances, and the easiest, in terms of hardware, algorithm, we have tested Huffman, Tunstall, LZ77 and Class Based algorithms. We haven’t considered SAMC and SADC methods because those methods require a very complex decoding hardware and are not adapt for real time decompression. The paper is organized as follows. Section II provides the basic definitions used throughout the paper. Section III provides the performances of code compression algorithms when applied to ARM code. Section IV analyzes the Class Based algorithm optimized for ARM code and shows the implementation of the decompression engine.

II. CODE COMPRESSION DEFINITIONS

A fundamental feature of code compression algorithms is the real time decompression that requires random access to the compressed program. The solution is splitting the program into blocks that are individually compressed. The complexity of the decompression hardware benefits from byte or word alignment of the block since in this way it is easier to address the begin of the blocks. It can be demonstrated the best trade-off is byte alignment. We have compressed 64B blocks (16 instructions). This value is a compromise between compression ratio and decompression latency.

When a compressed program is used it is necessary to map the uncompressed memory addresses to the compressed memory space (as example for jump instruction). Following [10] we use a LAT (Line Address Table), that contains the correspondence from uncompressed to compressed addresses. The LAT technique is general and does not require modifications of the compiler, but must be stored with the compressed code. The overhead of the LAT on the compression ratio is:

\[
LAT_{\text{orch}}(\%) = \frac{e_{\text{length}}}{b_{\text{lock size}}} \times 100(\%)
\]  

(1)
where \( e\text{\_length} \) is the entry length and \( \text{block\_size} \) is the size of each block, both expressed in bits. In the proposed application we suppose that the address space is 15 bits for a 32KB SRAM. In order to minimize \( \text{LAT}_{\text{ovh}} \), we have chosen a \( \text{LAT} \) entry 8 bytes long. Each entry specifies the address of a first block (15 bits) and 6 bits offsets for the following 8 blocks. The offset is the length of the block, given in bytes. For every lossless compression algorithm there are data whose size is increased by the compression. It is possible to leave these data uncompressed. They are identified by the offset in the \( \text{LAT} \) that is equal to the size of the block.

Depending on the position of the decompression engine, decode architectures can be classified [11] in:

- pre-cache, in which the decompression engine is between the memory and the cache;
- post-cache, in which the decompression engine is between the cache and the CPU.

We deal with post-cache architecture that reduces the required size of the cache but places the decompression engine on the critical path of the pipeline (decompression is requested at every instruction fetch).

A compression algorithm can be evaluated in a number of different ways. We have considered the compression ratio (inclusive of the overhead) and the feasibility and the speed of the decompression engine. The Compression Ratio (CR) is defined as:

\[
CR[\%] = CR_{\text{net}} + \text{LAT}_{\text{ovh}} + \text{TABLE}_{\text{ovh}} \quad (2)
\]

\[
CR_{\text{net}}[\%] = \frac{\text{Compressed\_Progr\_Size}}{\text{Original\_Progr\_Size}} \times 100[\%] \quad (3)
\]

\[
\text{TABLE}_{\text{ovh}}[\%] = \frac{D_{\text{table}}}{\text{Original\_Progr\_Size}} \times 100[\%] \quad (4)
\]

where \( D_{\text{table}} \) is the size of decoding table.

Compression algorithms have been tested on sample C codes compiled for ARM7TDMI processor. These programs deal with error check, code correction, and memory interface between embedded flash memory and peripherals. They have been modeled with a first order probability model.

Source modeling has been conducted considering the source composed of 4b, 8b or 16b symbols. Furthermore following [3] the ARM instruction has been divided in streams (it has been observed that the fields of an ARM instruction show different statistics). For the considered models, we have calculated the theoretical Compression Ratio:

\[
CR_{\text{theo}}[\%] = \frac{\text{Entropy}}{n} \times 100[\%] \quad (5)
\]

where \( n \) is the size of the symbol. This is the theoretical limit while the actual CR will be higher due to algorithm efficiency and overheads. The results are shown in Fig. 1. The most convenient models are named A to F.

III. PERFORMANCES OF CODE COMPRESSION ALGORITHMS

A. Huffman coding

For a given probability model, Huffman algorithm produces an optimum prefix code. We have applied this algorithm to the test programs obtaining the results shown in Fig. 2a. Huffman coding results in a final size of the compressed program that is higher than the initial size (\( CR > 100\% \)), due to the size of decoding table given by: \( D_{\text{Huff}} = n2^{L_{M}} \) bits (\( L_{M} \) is the maximum length of the codeword). It is worth noting that this problem is more evident when the initial size of the program is small. Furthermore predicting in advance \( L_{M} \) is not possible. As a consequence it is not possible to design a decompression engine that can be used for every program. These drawbacks are not present for those algorithms that fix the maximum value for \( L_{M} \). In order to bind the maximum length of the code word, we established the maximum size of the decoding table obtaining the corresponding \( L_{M} \).

We have analyzed two sub-optimum algorithms, Bounded Huffman [2] and Package Merge [12], and devised a unique algorithm that merges both of them. The results are shown in Fig. 2b. The Modified Huffman algorithm gets an overall
size reduction but the performance is quite poor. This is due to the reduced size of the decoding table that hampers the performances of the compression. With reference to hardware feasibility, Modified Huffman codes are not easily implemented since the length of the codewords is obtained only after a comparison with every entry of the decoding table.

B. Tunstall coding

A different solution is represented by a class of code compression techniques called variable-to-fixed code compression (V2FCC), that maps variable-length bit sequences into fixed-length bit sequences (codeword length is fixed) [13]. Tunstall coding presents two important advantages: it requires no table to decompress the code and the decoding is very simple. If MP is the most probable bit and LP is the other bit, the decoder engine must simply fetch a codeword and decodes an output formed by L MP bits followed by one LP bit, where L is the number represented by the codeword. The codewords formed by all ones or zeros are exceptions: the first codes a unique MP bit, the latter $2^{N} - 1$ MP bits. Unfortunately the performances of Tunstall coding on the ARM test codes are not satisfactory (95%) (Fig. 3). In fact Tunstall coding performs better when long sequences of MP bits are present in the uncompressed program, a condition seldom found in executable programs.

C. LZ77 coding

The algorithm replaces a sequence of symbols with a pointer (offset+length) to a previous occurrence of the same sequence, stored in a shift register named Search Buffer. LZ77 is an adaptive algorithm that is efficient only if it owns an history from which to get sequences of symbols, so it is very inefficient at the beginning of a stream. The original algorithm has been improved in two ways: LZ77 algorithm has been implemented as suggested in [14] and named LZSS. LZSS prevents the transmission of offset and length when they are both zero by using a flag bit; in order to create an history to the decoder we store the most probable symbols in the shift register before decompressing each block.

The resulting compression ratio for the best performing test program is only 89%, using a Search Buffer whose length is 128 symbols. Furthermore the LZSS decoder is complex and provides a long latency.

IV. CLASS BASED CODING

Another solution that overcomes the problems of Huffman algorithm is the Class Based coding [3]. After sorting the source symbols with decreasing occurrence probability, this method subdivides them into different classes with a variable number of elements. The $i$-th class is identified by a tag, equal to the binary representation of $i-1$ on $\log_2(K)$ bits ($K$ is the number of classes) and is characterized by the same length of the symbol code. Each codeword consists of a tag and a symbol code. The last class is named the class of literals and is associated to the least probable symbols. The literals are not compressed and are coded by adding the class code to them. In this way the decoding table can be kept very small since the symbols with low occurrence probability are not coded. If the symbols are $n$ bits long, the size of the decoding table is therefore equal to:

$$D_{CB} = 2^{\lfloor \log_2(N_S) \rfloor} n \text{ bits, } N_S = \sum_{j=a}^{K-2+a} 2^j$$

where $a$ is the length in bits of the symbol code for the first class.

Different class based algorithms have been considered and applied to the test programs. The techniques with four classes (two bits tag) provided unsatisfactory results because the decoding table contains too few symbols and most frequent symbols are in the literal class. Techniques with more than eight classes provide a huge decoding table. The optimal choices are the class based algorithms with eight classes that correspond to a tag with three bits. Two different class based algorithms have been considered in this paper:

- **CB0**: The first class contains only one symbol and hence is coded without the symbol code. This coding is very
A static decoder (with the decoding tables optimized for one a typical code) has been implemented in TSMC 0.18µm. The area occupation of the decoding table, implemented as a standard logic ROM, is 0.024mm². The resulting silicon area occupation is about 0.06mm² and maximum clock frequency is 100 MHz.

V. Conclusion

The paper has deeply analyzed most of the proposed code compression algorithms for embedded systems and evaluated their feasibility for the compression of ARM programs whose size is smaller than 32KB. A detailed analysis of the decoding overhead demonstrates that, even if Class Based algorithms provide a net compression ratio higher than other compression techniques such as the Huffman algorithm, it reaches very good compression efficiency due to the reduced size of the decompression tables. Furthermore the decompression can be done in parallel and the decompression engine results to be fast and has reduced silicon area occupation.

REFERENCES


