A Bitmask-based Code Compression Technique for Embedded Systems

Seok-Won Seong
Dept. of Computer & Information Sc. & Engg.
University of Florida, Gainesville, FL 32611, USA
sseong@cise.ufl.edu

Prabhat Mishra
Dept. of Computer & Information Sc. & Engg.
University of Florida, Gainesville, FL 32611, USA
prabhat@cise.ufl.edu

ABSTRACT

Embedded systems are constrained by the available memory. Code compression techniques address this issue by reducing the code size of application programs. Dictionary-based code compression techniques are popular because they offer both good compression ratio and fast decompression scheme. Recently proposed techniques [8, 9] improve standard dictionary-based compression by considering mismatches. This paper makes two important contributions: i) it provides a cost-benefit analysis framework for improving the compression ratio by creating more matching patterns, and ii) it develops an efficient code compression technique using bitmasks to improve the compression ratio without introducing any decompression penalty. To demonstrate the usefulness of our approach we have used applications from various domains and compiled for a wide variety of architectures. Our approach outperforms the existing dictionary-based techniques by an average of 15%, giving a compression ratio of 55% - 65%.

1. INTRODUCTION

Memory is one of the key driving factors in embedded system design since a larger memory indicates an increased chip area, more power dissipation, and higher cost. As a result, memory imposes constraints on the size of the application programs. Code compression techniques address the problem by reducing the program size. Figure 1 shows the traditional code compression and decompression flow where the compression is done off-line (prior to execution) and the compressed program is loaded into the memory. The decompression is done during the program execution (online).

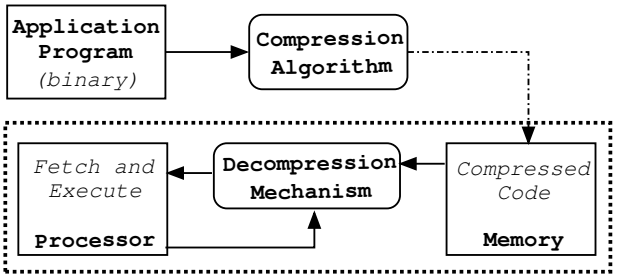


Figure 1: Traditional Code Compression Methodology

The first code compression technique for embedded processors was proposed by Wolfe and Chanin [1]. The idea of using a dictionary to store the frequently occurring instruction sequences has been explored by various researchers [2, 12]. Lekatsas and Wolf [6] proposed SAMC, a statistical method for code compression using arithmetic coding and Markov model. There has been a significant amount of research in the area of code compression for VLIW and EPIC processors. The technique proposed by Ishiura and Yamaguchi [10] splits a VLIW instruction into multiple fields and each field is compressed using a dictionary-based scheme. Nam et al. [13] also uses a dictionary-based scheme to compress fixed format VLIW instructions. Xie et al. [14] used Tunstall coding to perform a variable-to-fixed compression. Lin et al. [3] proposed a LZW-based code compression for VLIW processors using a variable-sized-block method.

Dictionary-based code compression techniques are popular because they provide both good compression ratio and fast decompression mechanism. The basic idea is to take advantage of commonly occurring instruction sequences by using a dictionary. Recently proposed techniques [8, 9] improve the dictionary-based compression technique by considering mismatches. The basic idea is to create instruction matches by remembering a few bit positions. However, the efficiency of these techniques are limited by the number of bit changes (hamming distance) used during compression. The cost of storing the information for more bit positions offsets the advantage of generating more repeating instruction sequences. Studies [9] have shown that it is not profitable to consider more than three bit changes when 32-bit vectors are used for compression. Section 2 presents a detailed cost-benefit analysis for creating matching instructions. *Compression ratio*, widely accepted primary metric for measuring the efficiency of code compression, is defined as,

$$Compression\ Ratio = \frac{Compressed\ program\ size}{Original\ program\ size} \quad (1)$$

We propose an efficient code compression technique to improve the compression ratio further by aggressively creating more matching sequences using bitmask patterns. We use the decompression engine between the instruction cache and the processor that increases cache hits and reduces bus bandwidth. Our design of the decompression unit is analogous to the one-cycle decompression hardware proposed by Lekatsas et al. [4] except one additional XOR at the output to handle the use of bitmasks. We have used applications from various domains (Mediabench and MiBench) and compiled them for a wide variety of architectures including TI TMS320C6x, MIPS, and SPARC. Our experimental results demonstrate that our approach outperforms the existing dictionary-based compression techniques by an average of 15% without introducing any additional decompression penalty.

The rest of the paper is organized as follows. Section 2 describes our cost-benefit analysis framework for creating more repeating patterns. Section 3 presents our code compression algorithm and decompression mechanism followed by a case study in Section 4. Finally, Section 5 concludes the paper.

2. COST-BENEFIT ANALYSIS

We have studied how to match more bit positions without adding significant information in the compressed code. We have considered 32-bit

ICCAD '06 November 5-9, San Jose, CA

code vectors for compression. Clearly, the hamming distance between any two 32-bit vectors is between 0 and 32. The compression adds extra 5 bits to remember each bit position in a 32-bit pattern. Moreover, extra bits are necessary to decide how many bit changes are there in the compressed code. For example, if the code allows up to 32 bit changes, it requires extra 5 bits to indicate the number of changes. As a result, this process requires the total of 165 extra bits (32x5 + 5) when all 32 bits are different. Clearly, it is not profitable to compress a 32-bit vector using 165 extra bits along with a codeword (index information) and other details.

We have explored the use of bit-masks for creating repeating patterns. For example, a 32-bit mask pattern is sufficient to match any two 32-bit vectors. Of course, it is not profitable to store extra 32 bits to compress a 32-bit vector but definitely better than 165 extra bits. We considered mask patterns of different sizes (1-bit to 32-bit). When a mask pattern is smaller than 32 bits, we need to store information related to starting bit position where the mask needs to be applied. For example, if we use a 8-bit mask pattern, and want to consider all 32-bit mismatches, it requires four 8-bit masks, and extra two bits (to identify one of the 4 bytes) for each mask pattern to indicate where it will be applied. In this particular case, we require extra 42 bits.

In general a dictionary contains 256 or more entries. As a result, a code pattern will have fewer than 32 bit changes. If a code pattern is different from a dictionary entry in 8 bit positions, it requires only one 8-bit mask and its position i.e., it requires 13 (8+5) extra bits. This can be improved further if we consider bit changes only in byte boundaries. This leads to a tradeoff - requires fewer bits (8+2) but may miss few mismatches that spread across two bytes. Our study uses the latter approach that uses fewer bits to store a mask position.

Table 1: Cost of Various Matching Schemes

Bit Changes	Size of the Mask Pattern					
	1-bit	2-bit	4-bit	8-bit	16-bit	32-bit
32 bits	165	100	59	42	35	32
16 bits	84	51	30	21	17	
8 bits	43	26	15	10		
4 bits	22	13	7			
2 bits	11	6				
1 bit	5					

An entry is left blank when that combination is not possible.

Table 1 shows the summary of our study. Each row represents the number of changes allowed. Each column represents the size of the mask pattern. A one-bit mask is essentially same as remembering the bit position. Each entry in the table (r, c) indicates how many extra bits are necessary to compress a 32-bit vector when r number of bit changes are allowed and c is the size of the mask pattern. For example, we require 15 extra bits to allow 8-bit (row with value 8) changes using 4-bit (column with value 4) mask patterns. This analysis forms the basis of our bitmask-based code compression technique as described in Section 3.

3. CODE COMPRESSION USING BITMASKS

The motivation of our work is based on the analysis presented in Section 2. Our approach tries to incorporate maximum bit changes using mask patterns without adding significant cost (extra bits) so that the compression ratio is improved. Our compression technique also ensures that the decompression efficiency remains the same compared to the existing techniques. Our scheme considers a 32-bit program code (vector) and uses mask patterns. Figure 2 shows the generic encoding scheme used by our compression technique. A compressed code can store information regarding multiple mask patterns. For each pattern, the generic encoding stores the mask type, (requires two bits to distinguish between 1-bit, 2-bit, 4-bit, or 8-bit), the location where mask needs to be applied, and the mask pattern.

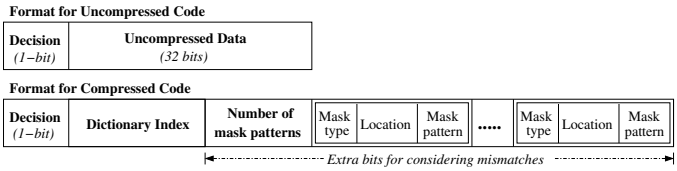
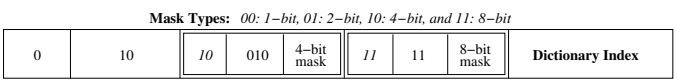


Figure 2: Encoding Format for Our Compression Technique

The number of bits needed to indicate a location will depend on the mask type. A mask of size s can be applied on $(32 \div s)$ number of places. For example, a 8-bit mask can be applied only on four places (byte boundaries). Similarly, a 4-bit mask can be applied on eight places (byte and half-byte boundaries). Consider a scenario where a 32-bit word is compressed using one 4-bit mask at second half-byte boundary, and one 8-bit mask at fourth byte boundary, the compressed code will appear as shown below.



The generic encoding scheme can be further optimized. For code compression, we have found that using up to two bitmasks is sufficient to achieve a good compression ratio. We explored various customized version of our encoding format to figure out which encoding format works better across the target architectures. Clearly, a 32-bit mask pattern is not profitable. The 16-bit mask is also not useful unless there are too many mismatches which a 4-bit or 8-bit (or combined 12 bit) mask cannot capture. We explored all possible encoding scenarios using 4-bit and 8-bit masks and observed that three customized encoding formats shown in Figure 3 work very well across applications and target architectures. The first encoding (Encoding 1) uses a 8-bit mask, the second encoding (Encoding 2) uses up to two 4-bit masks, and the third encoding (Encoding 3) uses up to two masks where first mask can be either 4-bit or 8-bit whereas the second mask is always 4-bit.

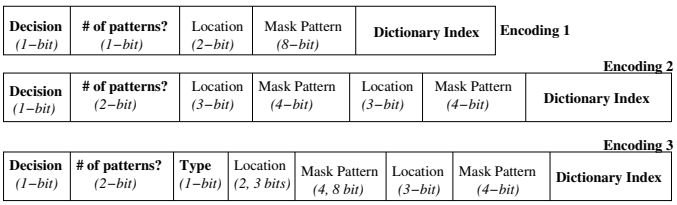


Figure 3: Three Customized Encoding Formats

We first explain our code compression algorithm. Next, we present our decompression mechanism. In Section 4, we report performance of these customized encoding formats.

3.1 Compression Algorithm

Algorithm 1 shows the four basic steps of our code compression algorithm. The algorithm accepts the original code consisting of 32-bit vectors. The first step creates the frequency distribution of the vectors. We consider two types of information to compute the frequency: repeating sequences and possible matching sequences by bitmasks. First, it finds the repeating 32-bit sequences and the number of repetition determines the frequency. This frequency computation is similar to any dictionary-based code compression scheme and provides an initial idea of the dictionary size. Next, all the high frequency vectors are upgraded (or downgraded) based on how many new repeating sequences they can create from mismatches using bitmasks with cost constraints. Table 1 provides the cost for the choices. For example, it is costly to use two 4-bit masks (cost: 15 bits) if an 8-bit mask (cost: 10 bits) can create the match. The second step chooses the smallest possible dictionary size without significantly affecting the compression ratio. It is useful to consider larger dictionary sizes when the current dictionary size cannot accommodate all the vectors with frequency value above certain

threshold. However, there are certain disadvantages of increasing the dictionary size. The cost of using a larger dictionary is more since the dictionary index becomes bigger. The cost increase is balanced only if most of the dictionary is full with high frequency vectors. Most importantly, a bigger dictionary increases the access time and thereby reduces decompression efficiency.

Algorithm 1: *Code Compression using Mask Patterns*
Input: Original code (binary) divided into 32-bit vectors
Outputs: Compressed code and dictionary
Begin
Step 1: Create the frequency distribution of the vectors.
Step 2: Create the dictionary based on Step 1.
Step 3: Compress each 32-bit vector using cost constraints.
Step 4: Handle and adjust branch targets.
return Compressed code and dictionary
End

The third step converts each 32-bit vector into compressed code (when possible) using the format shown in Figure 2. The compressed code along with any uncompressed ones are composed serially to generate the final compressed program code. The final step of the algorithm resolves the branch instruction problem by adjusting branch targets. Wolfe and Chanin [1] proposed the LAT, however, it requires an extra space and degrades overall performance. Lefurgy [2] proposed a technique which patches the original branch target addresses to the new offsets in the compressed program. This approach does not require any additional space but it is not suitable for handling indirect branches. Our technique handles branch targets by:

- patching all the possible branch targets into new offsets in the compressed program, and padding extra bits at the end of the code preceding branch targets to align on a byte boundary,
- creating a minimal mapping table to store the new addresses for the ones that could not be patched.

This approach significantly reduces the size of the mapping table required, allowing very fast retrieval of a new target address. This technique is very useful since more than 75% control flow instructions are conditional branches (compare and branch) and they are patchable. It leaves only 25% for a small mapping table. Our experiments show that more than 95% of the branches taken during execution do not require the mapping table. Therefore, the effect of branching is minimal in executing our compressed code.

3.2 Decompression Mechanism

Decompression time is critical since decompression is done at runtime. The decompression unit must be able to provide an instruction at the rate of the processor to avoid any stalling. Our design of the decompression engine is based on the one-cycle decompression engine (DCE) presented by Lekatsas et al. [4]. Figure 4 shows the design of our bitmask-based decompression unit. To expedite the decoding process, the DCE is customized for efficiency, depending on the choice of bit-masks used. Using two 4-bit masks (Encoding 2 in Section 3), the compression algorithm generates 4 different types of encodings: i) uncompressed instruction, ii) compressed without bitmasks, iii) compressed with one 4-bit mask, and iv) compressed with two 4-bit masks. In the same manner, using one bitmask creates only 3 different types of encodings. Decoding of uncompressed or compressed code without bitmasks remain virtually identical to the previous approach.

For compressed encodings using bitmasks, our decompression unit provides two additional operations: generating an instruction-length (32-bit) mask, and XORing the mask and the dictionary entry. The creation of an instruction-length mask is straightforward as done by applying the bitmask on the specified position in the encoding. For example, a 4-bit mask can be applied only on half-byte boundaries (8 locations). If two bitmasks were used, the two intermediate instruction length masks need to be OR-ed to generate one single mask. The advantage of our design is that generating an instruction length mask can be done in parallel with accessing the dictionary, therefore generating a 32-bit mask does not add any additional penalty to the existing DCE.

The only additional time incurred in our design, compared to the previous one-cycle design, is in the last stage where the dictionary entry and the generated 32-bit mask are XOR-ed. We have surveyed the commercially manufactured XOR logic gates and found that many of the manufactures produce XOR gates with the propagation delay ranging from 0.09ns - 0.5ns, numerous under 0.25ns. The critical path of decompression data stream in [4] was 5.99ns (with the clock cycle of 8.5 ns). Additional 0.25ns satisfies the 8.5ns clock cycle constraint.

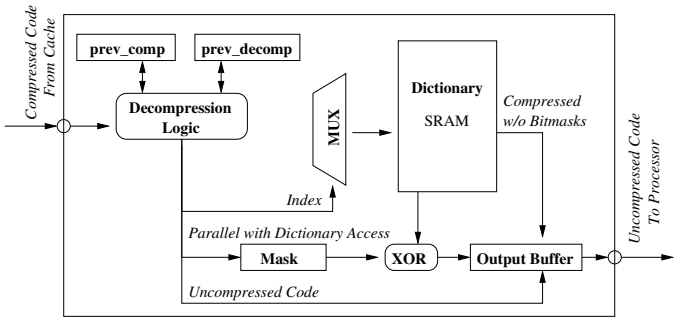


Figure 4: Decompression Engine for Bitmask Encoding

Our DCE can decode multiple instructions per cycle (with hardware support). If the codeword (with the dictionary index) is 10 bits, the encoding of instructions compressed only using the dictionary will be 12 bits or less. Instructions compressed with one 4-bit mask has the cost of additional 7 bits (total 18-19 bits). Therefore a 32-bit stream with any combination with a 12-bit code contains more than one instruction and can be decoded simultaneously. The best case is when a 32-bit stream contains two 12 bit encodings and *prev_comp* register holds 4 bits of the compressed data from the previous cycle, the DCE engine has three instructions in hand that can be decoded concurrently.

4. EXPERIMENTS

We performed various code compression experiments by varying both application domains and target architectures. In this section, we present experimental results using nine embedded applications for three target architectures. The nine benchmarks are collected from Mediabench and MiBench benchmark suites: *adpcm_en*, *adpcm_de*, *cjpeg*, *djpeg*, *gsm_to*, *gsm_un*, *mpeg2enc*, *mpeg2dec*, and *pegwit*. We compiled the benchmarks for three target architectures:TI TMS320C6x, MIPS, and SPARC. We used TI *Code Composer Studio* to generate binary for TI TMS320C6x. We used *gcc* to generate binary for MIPS and SPARC. We computed the compression ratio using the Equation (1). Our computation of a compressed program size includes the size of the compressed code as well as the dictionary and the small mapping table.

4.1 Results

In Section 3, we presented our generic encoding format as well as three customized formats. Encoding 1 uses one 8-bit mask, Encoding 2 uses up to two 4-bit masks, and Encoding 3 uses 4-bit and 8-bit masks. Figure 5 shows the performance of each of these encoding formats using *adpcm_en* benchmark for three target architectures. We used dictionary with 2K entries for these experiments. Clearly, the second encoding format performs the best by generating a compression ratio of 55-65%. Our experience with other benchmarks also suggests the same trend. We use the second encoding format (Encoding 2) for all the results presented in the remainder of this section.

Our technique performs well for different dictionary sizes. Figure 6 shows the efficiency of our compression technique for all the nine benchmarks compiled for SPARC using dictionary sizes of 4K and 8K entries. As expected, we can observe three scenarios. The small benchmarks

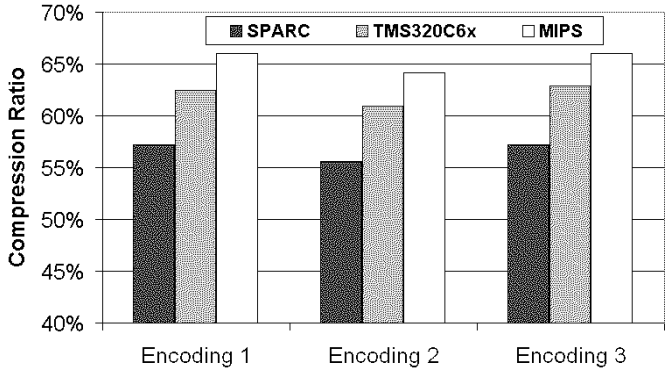


Figure 5: Compression Ratio for adpcm_en Benchmark

such as *adpcm_en* and *adpcm_de* perform better with a small dictionary since a majority of the repeating patterns fits in the 4K dictionary. On the other hand, the large benchmarks such as *cjpeg*, *djpeg*, and *mpeg2enc* benefit most from the larger dictionary. The medium sized benchmarks such as *mpeg2dec* and *pegwit* do not benefit much from the bigger dictionary size. On an average, our technique generates 59% compression ratio.

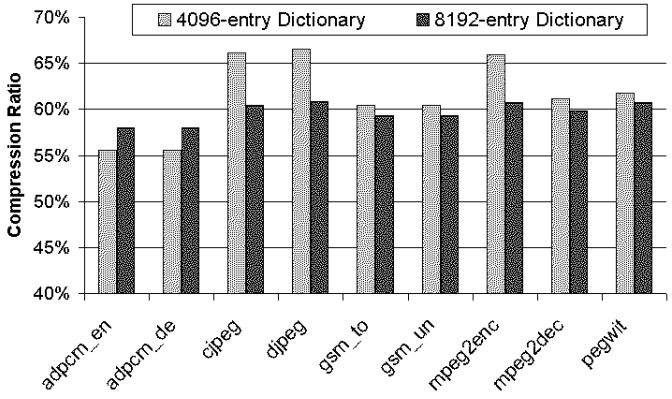


Figure 6: Compression Ratio for Different Benchmarks

Table 2 compares our approach with the existing code compression techniques. Our technique improves the code compression efficiency by 15% compared to the existing dictionary based techniques [8, 9]. The compression efficiency of our technique is comparable to the state-of-the-art compression techniques (IBM CodePack[7] and SAMC[6]). However, due to the encoding complexity, the decompression bandwidth of those techniques are only 6-8 bits. As a result, they can not support one instruction per cycle decompression and it is not possible to place the DCE between the cache and the processor to take advantage of the post-cache design. Our decompression mechanism supports one instruction per cycle delivery as well as parallel decompression.

Table 2: Comparison with Various Compression Schemes

Compression Method	Target Architecture	Compression Ratio	Decomp Bandwidth	Parallel Decomp
Wolfe [1]	MIPS	73%	8 bits	No
IBM [7] CodePack	PowerPC	60%	8 bits	No
SAMC [6]	MIPS	57%	6-8 bits	No
V2F [14]	TMS320C6x	70-82%	4.9-13 bits	No
MCSSC [3]	TMS320C6x	75%	14.5-64 bits	Yes
Prakash [8]	TMS320C6x	76-80%	N/A	Yes
Ros [9]	Itanium TMS320C6x	72-80%	N/A	Yes
Our Approach	MIPS, SPARC TMS320C6x	55-65%	32-64 bits	Yes

Smaller compression ratio implies better compression technique.

5. CONCLUSIONS

Embedded systems are constrained by the memory size. Code compression techniques address this problem by reducing the code size of the application programs. Dictionary-based code compression techniques are popular since they generate a good compression ratio by exploiting code repetitions. Recent techniques use bit toggle information to create matching patterns and thereby improve the compression ratio. However, due to lack of an efficient matching scheme, the existing techniques can match up to three bit differences.

We developed an efficient matching scheme using bitmasks that can significantly improve the code compression ratio. We applied our technique using applications from various domains and compiled them for different architectures to demonstrate the usefulness of our approach. Our experimental results show that our approach reduces the original program size by up to 45%. Our technique outperforms all the existing dictionary-based techniques by an average of 15%, giving compression ratios of 55%-65%. We also proposed the design of a simple and fast decompression unit that is capable of decoding an instruction per cycle as well as performing parallel decompression.

Currently, our technique generates up to 95% matching sequences. We plan to investigate further in terms of possibilities in creating more matches with fewer bits (cost). One possible direction is to introduce the compiler optimizations that use hamming distance as a cost measure for generating code. We also plan to study the power saving and performance improvement by our technique, as it reduces the code size and simplifies the decompression process.

6. REFERENCES

[1] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. *MICRO* 1992.

[2] C. Lefurgy, P. Bird, I. Chen and T. Mudge. Improving code density using compression techniques. *MICRO*, 1997.

[3] C. Lin, Y. Xie and W. Wolf. LZW-based code compression for VLIW embedded systems. *DATE* 2004.

[4] H. Lekatsas and J. Henkel and V. Jakkula. Design of an one-cycle decompression hardware for performance increase in embedded systems. *DAC* 2002.

[5] H. Lekatsas and J. Henkel and W. Wolf. Design and simulation of a pipelined decompression architecture for embedded systems. *ISSS* 2001.

[6] H. Lekatsas and W. Wolf. SAMC: A code compression algorithm for embedded processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(12):1689–1701, 1999.

[7] IBM. *CodePack PowerPC Code Compression Utility User's Manual. Version 3.0*, 1998.

[8] J. Prakash, C. Sandeep, P. Shankar and Y. Srikant. A simple and fast scheme for code compression for VLIW processors. *DCC* 2003.

[9] M. Ros and P. Sutton. A hamming distance based VLIW/EPIC code compression technique. *CASES* 2004.

[10] N. Ishiura and M. Yamaguchi. Instruction code compression for application specific VLIW processors based of automatic field partitioning. *SASIMI* 1997.

[11] S. Larin and T. Conte. Compiler-driven cached code compression schemes for embedded ILP processors. *MICRO* 1999.

[12] S. Liao, S. Devadas and K. Keutzer. Code density optimization for embedded DSP processors using data compression techniques. *Advanced Research in VLSI* 1995.

[13] S. Nam, I. Park and C. Kyung. Improving dictionary-based code compression in VLIW architectures. *IEICE Trans. Fundamentals*, E82-A(11):2318–2324, 1999.

[14] Y. Xie, W. Wolf and H. Lekatsas. Code compression for VLIW processors using variable-to-fixed coding. *ISSS* 2002.