An Efficient Code Compression Technique using Application-Aware Bitmask and Dictionary Selection Methods

Seok-Won Seong
sseong@cise.ufl.edu

Prabhat Mishra
prabhat@cise.ufl.edu

Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611, USA.

Abstract

Memory plays a crucial role in designing embedded systems. A larger memory can accommodate more and large applications but increases cost, area, as well as energy requirements. Code compression techniques address this problem by reducing the size of the applications. While early work on bitmask-based compression has proposed several promising ideas, many challenges remain in applying them to embedded system design. This paper makes two important contributions to address these challenges by developing application-specific bitmask selection and bitmask-aware dictionary selection techniques. We applied these techniques for code compression of TI and MediaBench applications to demonstrate the usefulness of our approach.

1 Introduction

Demands for sophisticated and complex embedded applications have soared drastically in recent years. The current trend requires a larger and faster memory and imposes major challenges in embedded system design since memory has significant impacts on the size, power and cost of the entire system. Code compression techniques address this issue by reducing the size of the applications before they are loaded into the onchip memory. During execution (runtime) compressed instructions pass through the decompression hardware and executed by the processor. Therefore, decompression time is a critical aspect in code compression for embedded systems.

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. Various techniques [12, 13] improve the dictionary-based compression technique by considering mismatches. The basic idea is to create instruction matches by remembering a few bit positions where they differ. However, the efficiency of these techniques is limited by the number of bit changes used during compression. The cost of storing the information for more bit positions offsets the advantage of generating more repeating instruction sequences. Our previous work on bitmask-based code compression [14] addressed this issue by creating more matching patterns using bitmasks to improve the compression ratio. **Compression ratio** is a widely used metric to measure the efficiency of code compression. It is defined as the ratio (CR) between the compressed program size (CS) and the original program size (OS) i.e., CR = CS / OS. Therefore, *a smaller compression ratio implies a better compression technique*.

The bitmask-based code compression is very promising but it poses various practical challenges. For instance, it is very difficult to determine the optimal mask set prior to compression. Furthermore, the conventional dictionary selection (such as frequency- or spanning-based) approaches do not provide the best possible compression using bitmasks. This paper addresses these challenges by developing efficient techniques for application-specific bitmask selection and bitmask-aware dictionary selection to improve the compression ratio further without introducing any decompression penalty. The rest of the paper is organized as follows. Section 2 presents related work addressing code compression for embedded systems. Section 3 describes the bitmask-based code compression and its challenges. Section 4 presents our code compression technique that addresses these challenges followed by a case study in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

Wolfe and Chanin first proposed the Huffman-coding based code compression scheme for the MIPS architecture [15]. They used a Line Address Table (LAT) to map compressed block addresses to the original code addresses. Based on a similar concept, IBM introduced CodePack for PowerPC [2] architecture. Liao [9] and Lefurgy [5] explored dictionary-based compression techniques. Lekatsas and Wolf [7] proposed a statistical method for code compression using arithmetic coding and Markov model. Ishiura and Yamaguchi [3] proposed a technique that splits a VLIW instruction into multiple fields and compresses each field using a dictionary-based scheme. Nam et al. [11] presented a dictionary-based method for an isomorphic VLIW instruction word scheme. Lie et al. [10] proposed a LZW-based code compression for VLIW instructions using variable-sized blocks. Lekatsas et el. [6] proposed a dictionary-based decompression prototype that is capable of decoding one instruction per cycle.

Recently, various techniques have been proposed to improve the standard dictionary-based compression. The basic idea is to create more repeating patterns by storing the mismatch information during encoding. Prakash et al. [12] considered one-bit change for 16-bit vectors. Ros et al. [13] considered a generic scheme for 32-bit vectors and reported that up to 3-bit changes are profitable. Previously, we developed a bitmask-based code compression technique that significantly improves the compression ratio compared to the exisiting techniques. Section 3 briefly describes our previous work on bitmask-based code compression [14] and outlines various challenges in applying this technique to embedded system design.

3 Bitmask-Based Code Compression

This section describes bitmask-based code compression [14] using a simple example and compares with existing dictionary-based compression techniques. Figure 1 shows an example of the standard dictionary-based code compression. The sample program is made up of ten 8-bit binaries (total 80 bits). The dictionary has two 8-bit entries. Each repeating pattern is replaced with a dictionary index. The final compressed program is reduced to 62 bits and the dictionary requires 16 bits. In this case, the compression ratio is 97.5%.

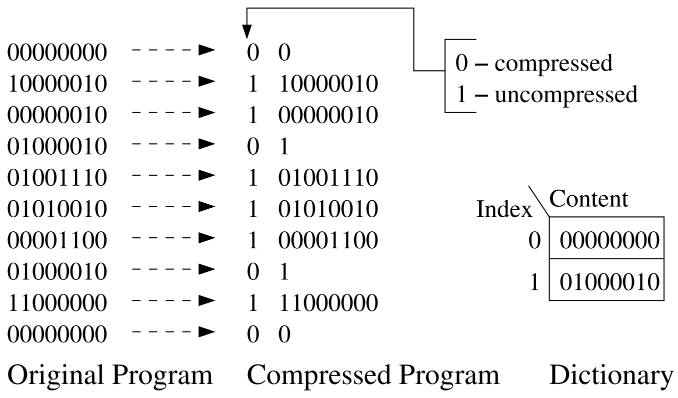

Figure 1. Dictionary-based Code Compression

We use the same example in Figure 3 to illustrate how bitmask-based code compression improves the compression ratio by using bitmasks. First, we outline the bitmask-based compression algorithm in Section 3.1 and the decompression scheme in Section 3.2. Next, Section 3.3 describes various challenges in applying bitmask-based compression technique.

3.1 Compression (Encoding) Framework

Figure 2 shows the generic encoding scheme of the bitmask-based compression technique. A compressed code stores information regarding the mask type, the location where the mask is applied, and the mask pattern itself. The mask can be applied on different places on a vector (binary) and the number of bits required for indicating the position varies depending on the mask type. For instance, an 8-bit mask applied on only byte boundaries requires 2 bits, since it can be applied on four locations (on a 32-bit vector). If we do not restrict the placement of the mask, it will require 5 bits to indicate any starting position on a 32-bit vector.

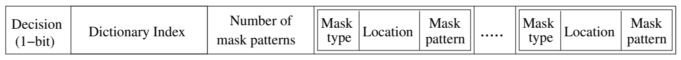

Figure 2. Generic Encoding Format

Figure 3 illustrates the bitmask-based code compression technique using the same binary shown in Figure 1. A 2-bit mask (only on quarter byte boundaries) is sufficient to create 100% matching patterns. The bitmask-based approach improves the compression ratio to 87.5% for this example.

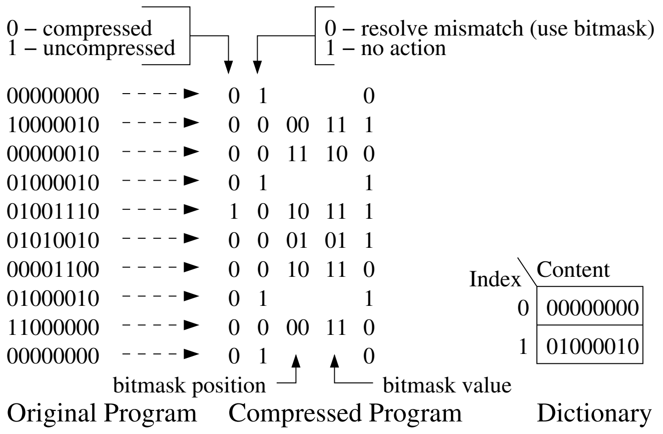

Figure 3. Bitmask-based Code Compression

3.2 Decompression (Decoding) Mechanism

The design of the bitmask-based decompression engine (BDE) is based on the one-cycle dictionary-based decompression hardware by Lekatsas et el. [6]. Figure 4 shows the design of the decompression engine. The bitmask-based decompression unit provides two additional operations (compared to the existing decompression engine) to support compressed encodings using bitmasks: i) it generates an instruction-length mask from the encoding, and ii) it XORs the generated mask and the entry in the dictionary to restore the original instruction. Creating an instruction-length mask is done in parallel with accessing the dictionary, therefore it does not introduce any additional time. Moreover, such design can support parallel decompression to enable decoding of multiple instructions per cycle.

3.3 Challenges in Bitmask-Based Code Compression

One of the major challenges in bitmask-based code compression is how to determine (a set of) optimal mask patterns that will maximize the matching sequences while minimizing the cost of bitmasks. A 2-bit mask can handle up to 4 types of mismatches while a 4-bit mask can handle up to 16 types of mismatches. Clearly, applying a larger bitmask will generate more matching patterns, however, doing so may not result in better compression. The reason is simple. A longer bit-mask pattern is associated with a higher cost. Similarly, applying more bitmasks is not always beneficial. For example, applying a 4-bit mask requires 3 bits to indicate its position (8 possible locations in a 32-bit vector) and 4 bits to indicate the pattern (total 7 bits) while an 8-bit mask requires 2 bits for the position and 8 bits for the pattern (total 10 bits). Therefore, it would be more costly to use two 4-bit masks if one 8-bit mask can capture the mismatches.

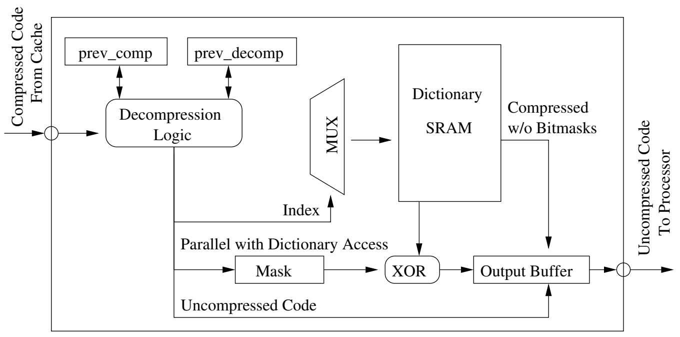

Figure 4. One-Cycle Decompression Engine

Another major challenge in bitmask-based compression is how to perform dictionary selection where existing as well as bitmask-matched repetitions need to be considered. In the traditional dictionary-based compression approach, the dictionary entry selection process is simplified since it is evident that the frequency-based selection will give the best compression ratio. However, when compressing using bitmasks, the problem is complex and the frequency-based selection will not always yield the best compression ratio. Figure 5 demonstrates this fact. When only one dictionary entry is allowed, the pure frequency-based selection will choose "*0000000*", yielding the compression ratio of 97.5% (Compressed Program 1). However, if "*01000010*" was chosen, we can achieve the compression ratio of 87.5% (Compressed Program 2). Clearly, there is a need for efficient mask selection and dictionary selection techniques to improve the efficiency of bitmask-based code compression.

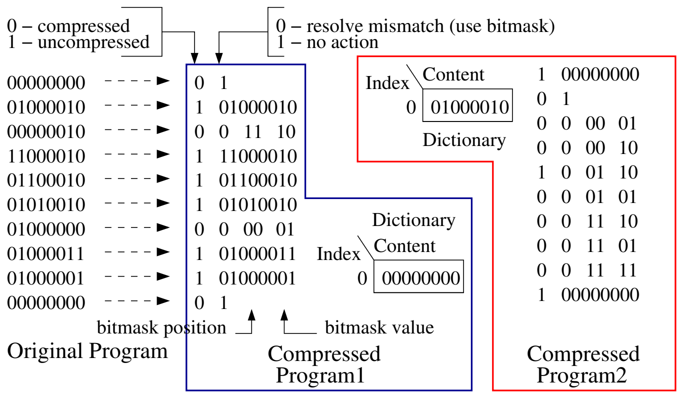

Figure 5. Different Dictionary Selection Methods

4 Application-Aware Code Compression

Our work is motivated by the challenges described in Section 3.3. We have developed two techniques to address these challenges: i) application-specific mask selection, and ii) bitmask-aware dictionary selection. First, we describe our mask selection approach. Next, we present our bitmask-aware dictionary selection technique. Finally, we present our code compression framework integrating the mask selection and the dictionary selection techniques.

4.1 Mask Selection

As discussed in Section 3.3, mask selection is a major challenge. Our goal in this section is to develop a procedure to find a set of bitmask patterns that will deliver the best compression ratio for a given application(s). This leads to answering two questions: i) how many bitmask patterns do we need? and ii) which bitmask patterns are profitable? We will answer these questions after defining few terms related to bitmask patterns.

Table 1 shows the mask patterns that can generate matching patterns at an acceptable cost. A "fixed" bitmask pattern implies that the pattern can be applied only on fixed locations (starting positions). For example, an 8-bit fixed mask (referred as *8f*) is applicable on 4 fixed locations (byte boundaries) on a 32-bit vector. A "sliding" mask pattern can be applied anywhere. For example, an 8-bit sliding mask (referred as *8s*) can be applied in any location on a 32-bit vector. There is no difference between fixed and sliding for a 1-bit mask. We will use a 1-bit sliding mask (referred as *1s*) for uniformity.

The number of bits needed to indicate a location will depend on the mask size and the type of the mask. A fixed mask of size x can be applied on $(32 \div x)$ number of places. An 8-bit fixed mask can be applied only on four places (byte boundaries), therefore requires 2 bits. Similarly, a 4-bit fixed mask can be applied on eight places (byte and half-byte boundaries) and requires 3 bits for its position. A sliding pattern will require 5 bits to locate the position regardless of its size. For instance, a 4-bit sliding mask requires 5 bits for location and 4 bits for the mask itself.

Table 1. Various Bit-Mask Patterns

Bit-Mask	Fixed	Sliding
1 bit		X
2 bits	X	X
3 bits		X
4 bits	X	X
5 bits		X
6 bit		X
7 bit		X
8 bit	X	X

If we choose two distinct bit-mask patterns, 2-bit fixed (*2f*) and 4-bit sliding (*4s*), we can generate six combinations: (2f), (4f), (2f, 2f), (2f, 4f), (4f, 2f), (4f, 4f). Similarly, three distinct mask patterns can create up to 39 combinations. Now we can try to answer the two questions posed at the beginning of this section. It is easy to answer the first question: up to two mask patterns are profitable. The reason is obvious based on the cost consideration. The smallest cost to store the three bit-mask information (position and pattern) is 15 bits (if three 1-bit sliding patterns are used). In addition, we need 1-5 bits to indicate the mask combination and 8-14 bits for a codeword (dictionary index). Therefore, we require approximately 29 bits (on average) to encode a 32-bit vector. In other words, we save only 3 bits to match 3 bit differences (on a 32-bit vector). Clearly, it is not very profitable to use three or more bitmask patterns.

Next we try to answer the second question i.e., which bit-masks are profitable? As discussed in Section 3.3, applying a larger bitmask can generate more matching patterns. However, it may not improve the compression ratio. Similarly, using a sliding mask where a fixed one is sufficient is wasteful since a fixed mask require fewer number of bits (compared to its sliding counterpart) to store the position information. For example, if a 4-bit sliding mask (cost of 9 bits) is used where a 4-bit fixed (cost of 7 bits) is sufficient, two additional bits are wasted.

We carefully studied the combinations of up to two bit-masks using several applications compiled on a wide variety of architectures. We observed that the mask patterns that are factors of 32 (e.g., masks 1, 2, 4 and 8 from Table 1) produce a better compression ratio compared to non-factors (e.g., masks 3, 5, 6, and 7). This is due to the fact that we accept the program of 32-bit vectors, therefore non-factor sized bit-masks were only usable as a sliding pattern. While sliding patterns are more flexible, they are more costly than fixed patterns. The above observations allowed us to reduce the 11 mask patterns in Table 1 down to 7 profitable mask patterns: 1s, 2f, 2s, 4f, 4s, 8f, 8s. A subset of these experiments is reported in Section 5. We analyzed the result of compression ratios using various mask combinations and made several useful observations that helped us to further reduce the bit-mask pattern table. We found that *8f* and *8s* are not helpful and *4s* does not perform better than *4f*. We also observed that using two bitmasks provide a better compression ratio than using one bitmask alone. The final set of profitable bitmask patterns are shown in Table 2. Our integrated compression technique (Algorithm 2 in Section 4.3) uses the bitmask patterns from Table 2.

Table 2. Final Bitmask Patterns

Bit-Mask	Fixed	Sliding
1 bit		X
2 bits	X	X
4 bits	X	

4.2 Bitmask-Aware Dictionary Selection

Dictionary selection is another major challenge in code compression. The optimal dictionary selection is an NP hard problem [8]. Therefore, the dictionary selection techniques in literature try to develop various heuristics based on application characteristics. Dictionary can be generated either dynamically during compression or statically prior to compression. While a dynamic approach such as LZW [10] accelerates the compression time, seldom it matches the compression ratio of static approaches. Moreover, it may introduce extra penalty during decompression and thereby reduces the overall performance. In the static approach, the dictionary can be selected based on the distribution of the vectors' frequency or spanning [13].

We have observed that neither frequency-based nor spanning-based methods can efficiently exploit the advantages of bitmask-based compression. Moreover, due to lack of a comprehensive cost metric, it is not always possible to obtain the optimal dictionary by combining frequency and spanning-based methods in an ad-hoc manner.

We have developed a novel dictionary selection technique that considers bit savings as a metric to select a dictionary entry. Algorithm 1 shows our bit-saving based dictionary selection technique. The algorithm takes application(s) consisting of 32-bit vectors as input and produces the dictionary as output that will deliver a good compression ratio. It first creates a graph where the nodes are the unique 32-bit vectors. An edge is created between two nodes if they can be matched using a bit-mask pattern(s). It is possible to have multiple edges between two nodes since they can be matched by various mask patterns. However, we consider only one edge between two nodes corresponding to the most profitable mask (maximum savings).

```
Algorithm 1: Bit-Saving based Dictionary Selection
Inputs: 1. Application(s) consisting of 32-bit instruction vectors
        2. Mask patterns
        3. A threshold value to screen deletion of nodes.
Output: Optimized dictionary
Begin
  Step 1: Create a graph representation, G=(V,E).
              Each node (V) is a unique 32-bit vector.
              An edge (E) indicates a bit-mask can match the nodes.
  Step 2: Allocate bit-savings to the nodes and edges.
              Frequency determines the bit-savings of the node.
              Mask used determines the bit-savings by that edge.
  Step 3: Calculate the bit-savings distribution of all nodes.
  Step 3: Select the most profitable node N.
  Step 4: Remove N from G and insert into dictionary
  Step 5: For each node $N_i$ in G that is connected to N
            If the node profit of $N_i$ is less than certain threshold
            Remove $N_i$ from G.
  Step 6: Repeat Steps 3 - 5 until dictionary is full or G is empty.
  return Dictionary
End
```

Once the bit-savings are assigned to all nodes and edges, the algorithm computes the overall savings for each node. The overall savings is obtained by adding the savings in each edge (bitmask savings) connected to that node along with the node savings (based on the frequency value). Next, the algorithm selects the node with the maximum overall savings as an entry for the dictionary. The selected node as well as the nodes that are connected to the selected node are deleted from the graph. However, we have observed that it is not always profitable to delete all the connected nodes. Instead, we set a particular *threshold* to screen the deletion of nodes. Typically a node with a frequency value less than 10 is a good candidate for deletion when the dictionary is not too small. This varies from application to application but based on our experiments a threshold value between 5 and 15 was most useful. The algorithm terminates when either the dictionary is full or the graph is empty.

Figure 6 illustrates this technique. The vertex "A" has the total saving of 15 (10+5), "B" and "C" have 22, "D" has 5, "E" has 10, "F" has 27, and "G" has 24. Therefore, "F" is chosen as the best candidate and gets inserted into the dictionary. Once "F" is inserted into the dictionary, it gets removed from the graph. "C" and "E" are also removed since they can be matched with "F" in the dictionary and bit-mask(s). Note that if the frequency value of the node "C" was larger than the threshold value, it would not be removed in this iteration.

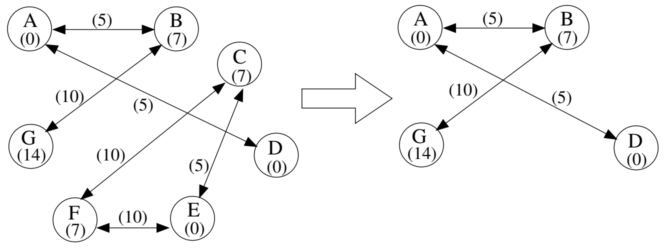

Figure 6. Bit-Saving Dictionary Selection Method

The algorithm repeats by recalculating the savings of the vertex in the new graph and terminates when the dictionary becomes full or the graph is empty. Our experimental results show that the bit-saving based dictionary selection method outperforms both frequency and spanning based approaches.

4.3 Code Compression Algorithm

In this section, we present our code compression algorithm that integrates our mask selection and dictionary selection methods. The goal is to maximize the compression efficiency using the bitmask-based code compression. Algorithm 2 outlines the basic steps. The algorithm accepts the original code consisting of 32-bit vectors as input and produces the compressed code and an optimized dictionary.

```
Algorithm 2: Code Compression using Bitmasks
Input: Original code (32-bit vectors)
Outputs: Compressed code, dictionary, $< mask_1, mask_2 >$
Begin
  $mask_1$ = 1s; $mask_2$ = 1s; CompressionRatio = 100%
  Step 1: Select the mask patterns.
  for each mask pattern $m_i$ in (1s, 2s, 2f, 4f)
      for each mask pattern $m_j$ in (1s, 2s, 2f, 4f)
          Step 2: Select the optimized dictionary.
          Step 3: Compress 32-bit vectors using cost constraints.
          Step 4: Update the variables if necessary.
          NewCR = (CompCode with $m_i$&$m_j$)$\div$(OrigCode)
          if (NewCR $<$ CompressionRatio)
             CompressionRatio = NewCR
             $mask_1 = m_i$; $mask_2 = m_j$
          endif
      endfor
  endfor
  Step 5: Adjust and handle the branch targets.
  return Compressed code, dictionary, $< mask_1, mask_2 >$
End
```

The algorithm begins by initializing three variables: $mask_1$, $mask_2$, and *CompressionRatio*. The profitable mask patterns are stored in $mask_1$ & $mask_2$ and *CompressionRatio* stores the best compression ratio at each iteration. The first step is to pick a pair of mask patterns from the reduced set of (1s, 2s, 2f, 4f) from Table 2. The second step selects the optimized dictionary using Algorithm 1. The third step converts each 32-bit vector into compressed code (when possible). If the new compression ratio is better than the current one, the fourth step updates the variables. The final step of the algorithm resolves the branch instruction problem by adjusting branch targets. The algorithm returns the compressed code, optimized dictionary and two profitable mask patterns. Note that this algorithm can be used as a one-pass or two-pass code compression technique. In a two-pass code compression approach, the first pass can use synthetic benchmarks (equivalent to the real applications in terms of various characteristics but much smaller) to determine the most profitable two mask patterns. During second pass the first step (two *for* loops) can be ignored and the actual code compression can be performed using real applications.

5 Experiments

We performed various code compression experiments by varying both mask combinations and dictionary selection methods. We have used benchmarks from various application domains. In this section, we present experimental results using applications from TI and Mediabench suites. We used TI *Code Composer Studio* to generate binaries for TI TMS320C6x architecture.

5.1 Results

Figure 7 shows compression ratios of three benchmarks (*block_mse*, *modem*, and *vertibi*) compressed using all 56 different mask set combinations[1] (both one-mask and two-mask combinations from {1s, 2f, 2s, 4f, 4s, 8f, 8s}). As discussed in Section 4.1, 8-bit mask patterns (fixed or sliding) do not provide good compression ratio. In general, compressing with two masks achieves a better compression ratio than using just one. Note that the compression ratios for three benchmarks follows a regular pattern. A similar pattern exists even with other benchmarks. It confirms our analysis in Section 4.1 that a small set of mask patterns are sufficient to achieve good compression. Overall, we found that the combination of 4-bit fixed and 1-bit sliding or two 2-bit patterns provides the best compression.

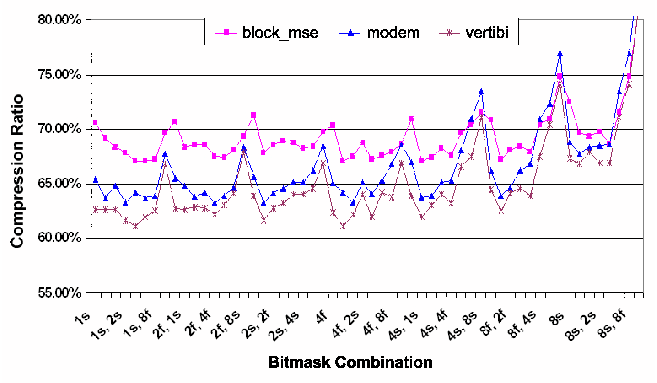

Figure 7. Performance Analysis of Mask Combinations

Figure 8 compares compression ratios achieved by the various dictionary selection methods described in Section 4.2. As shown in the figure, spanning-based approach is the worst compared to other dictionary selection methods. Our bit-savings

[1]In order of (1s), (1s,2f), (1s,2s), (1s,4f), (1s,4s), (1s,8f), (1s,8s), (2s) ...

based approach outperforms all the existing dictionary selection methods on all benchmarks.

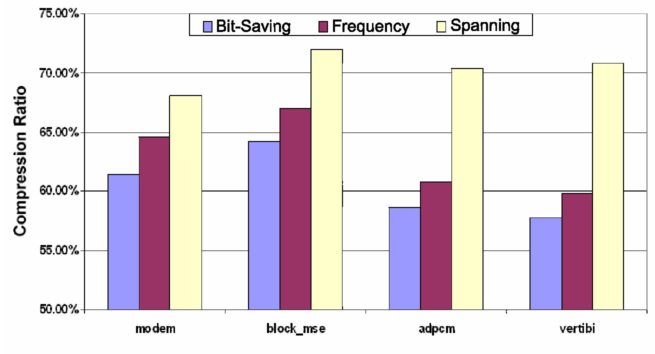

Figure 8. Comparison of Dictionary Selection Methods

Figure 9 compares the compression ratios between the existing bitmask-based code compression (BCC) technique and our approach. The existing approach (BCC [14]) experimented with customized encodings of 4-bit and 8-bit mask combinations and reported that using two 4-bit masks provided the best compression ratio. We have computed the most profitable mask pairs and applied the bit-saving based dictionary selection to improve the compression ratio further. For example, we obtained 57% compression ratio for *adpcm_en* benchmark using 4-bit fixed and 1-bit sliding patterns that outperforms the BCC approach by 6%. On an average, our approach outperforms the existing bitmask-based technique by 5 - 10%. A detailed comparison of bitmask-based compression with other code compression techniques is available in [14].

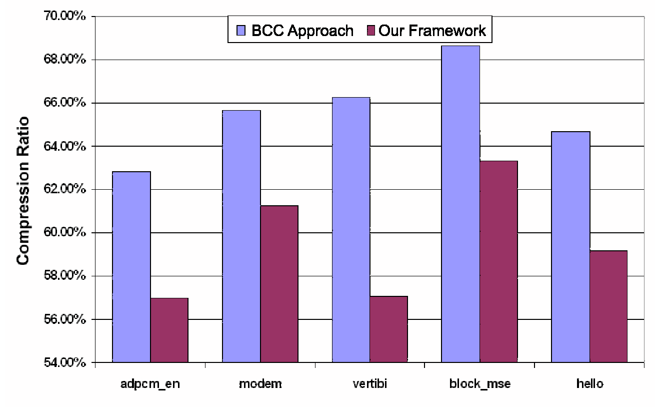

Figure 9. Compression Ratio Comparison

This code size reduction can contribute not only to cost, area, energy savings but also to the performance enhancement of the embedded system. Our approach, due to the nature of the mask and dictionary selection procedures, incurs higher encoding/compression overhead than [14]. However, in embedded systems design using code compression, encoding is performed once and millions of copies are manufactured. Any reduction of cost, area, or energy requirements is extremely important. Moreover, our approach does not introduce any decompression penalty.

6 Conclusions

In recent years, applications for embedded systems have become exponentially complex and it imposes major constraints in system design. The existing bitmask-based compression technique proposed a promising approach to address the memory requirement issue but poses various practical challenges.

This paper studied these challenges in detail and developed application-specific bitmask selction and bitmask-aware dictionary selection algorithms to address them. We developed an efficient code compression technique using these algorithms to improve code compression efficiency without introducing any decompression overhead. In the future, we plan to investigate the effects of our compression approach on overal energy savings and performance improvement. We also plan to apply our technique in other domains such as data compression for manufacturing testing.

References

[1] H. Lekatsas et al. Design and simulation of a pipelined decompression architecture for embedded systems. *ISSS*, 2001.

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

[3] N. Ishiura and M. Yamaguchi. Instruction code compression for application specific VLIW processors based of automatic field partitioning. *SASIMI*, 105–109, 1997.

[4] C. Lefurgy, P. Bird, I. Chen, and T. Mudge. Improving code density using compression techniques. *MICRO*, 194–203, 1997.

[5] C. Lefurgy and T. Mudge. Code compression for DSP. CSE-TR-380-98. Technical report, University of Michigan, 1998.

[6] H. Lekatsas et al. Design of an one-cycle decompression hardware for performance increase in embedded systems. *DAC* 2002.

[7] H. Lekatsas and W. Wolf. SAMC: A code compression algorithm for embedded processors. *IEEE TCAD*, 18(12), 1999.

[8] L. Li, K. Chakrabarty, and N. Touba. Test data compression using dictionaries with selective entries and fixed-length indices. *ACM TODAES*, Vol.8:470–490, October 2003.

[9] S. Liao, S. Devadas, and K. Keutzer. Code density optimization for embedded DSP processors using data compression techniques. *Advanced Research in VLSI*, 393–399, 1995.

[10] C. Lin, Y. Xie, and W. Wolf. LZW-based code compression for VLIW embedded systems. *DATE*, 76–81, 2004.

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

[12] J. Prakash et al. A simple and fast scheme for code compression for VLIW processors. *DCC* 2003.

[13] M. Ros and P. Sutton. A hamming distance based VLIW/EPIC code compression technique. *CASES*, 132–139, 2004.

[14] S. Seong and P. Mishra. A bitmask-based code compression technique for embedded systems. *ICCAD*, 2006.

[15] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. *MICRO*, 81–91, 1992.