Bitmask-Based Code Compression for Embedded Systems

Seok-Won Seong and Prabhat Mishra, *Member, IEEE*

***Abstract*—Embedded systems are constrained by the available memory. Code-compression techniques address this issue by reducing the code size of application programs. It is a major challenge to develop an efficient code-compression technique that can generate substantial reduction in code size without affecting the overall system performance. We present a novel code-compression technique using bitmasks, which significantly improves the compression efficiency without introducing any decompression penalty. This paper makes three important contributions. 1) It develops an efficient bitmask-selection technique that can create a large set of matching patterns. 2) It develops an efficient dictionary-selection technique based on bitmasks. 3) It proposes a dictionary-based code-compression algorithm using the bitmask- and dictionary-selection techniques that can significantly reduce the memory requirement. To demonstrate the usefulness of our approach, we have performed code compression using applications from various domains and compiled for a wide variety of architectures. Our approach outperforms the existing dictionary-based techniques by an average of 20%, giving a compression ratio of 55%–65%.**

***Index Terms*—Bitmasks, code compression, decompression, embedded systems, memory.**

I. Introduction

MEMORY is one of the key driving factors in embedded-system design because 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. Fig. 1 shows the traditional code-compression and decompression flow where the compression is done offline (prior to execution) and the compressed program is loaded into the memory. The decompression is done during the program execution (online).

Compression ratio (CR), widely accepted as a primary metric for measuring the efficiency of code compression, is defined as

$$\mathrm{CR} = \frac{\text{Compressed program size}}{\text{Original program size}}. \qquad (1)$$

Dictionary-based code-compression techniques are popular because they provide both good CR and fast decompression

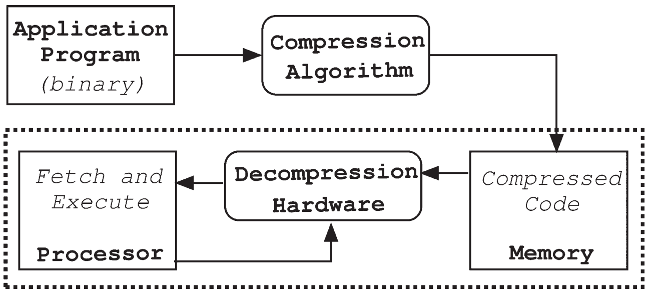

Fig. 1. Traditional code-compression methodology.

mechanism. The basic idea is to take advantage of commonly occurring instruction sequences by using a dictionary. Recently proposed techniques [3], [4] improve the dictionary-based compression by considering mismatches. The basic idea is to create instruction matches by remembering a few bit positions. The efficiencies of these techniques are limited by the number of bit changes used during compression. It is obvious that if more bit changes are allowed, more matching sequences will be generated. However, the cost of storing the information for more bit positions offsets the advantage of generating more repeating instruction sequences. Studies [4] have shown that it is not profitable to consider more than three bit changes when 32-b vectors are used for compression. There are various complex compression algorithms that can generate major reduction in code size. However, such compression scheme requires a complex decompression mechanism and thereby reduces overall system performance. It is a major challenge to develop an efficient code-compression technique that can generate substantial code-size reduction without introducing any decompression penalty (and thereby reducing performance).

We propose an efficient code-compression technique to further improve the CR by aggressively creating more matching sequences using bitmask patterns. This paper addresses various challenges in bitmask-based compression by developing efficient techniques for application-specific bitmask selection and bitmask-aware dictionary selection to further improve the CR. We have used applications from various domains (TI, Mediabench, and MiBench) and compiled them for a wide variety of architectures including TI C6x, MIPS, and SPARC. Our experimental results demonstrate that our approach outperforms the existing dictionary-based compression techniques by an average of 20% without introducing any additional decompression overhead.

The rest of the paper is organized as follows. Section II presents related work addressing code compression for embedded systems. Section III describes existing dictionary-based

Manuscript received February 15, 2007; revised June 19, 2007 and September 7, 2007. This paper was recommended by Associate Editor M. Poncino.

S.-W. Seong is with the Department of Electrical Engineering, Stanford University, Stanford, CA 94305-9505 USA (e-mail: sseong@stanford.edu).

P. Mishra is with the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611-6120 USA (e-mail: prabhat@cise.ufl.edu).

Digital Object Identifier 10.1109/TCAD.2008.917563

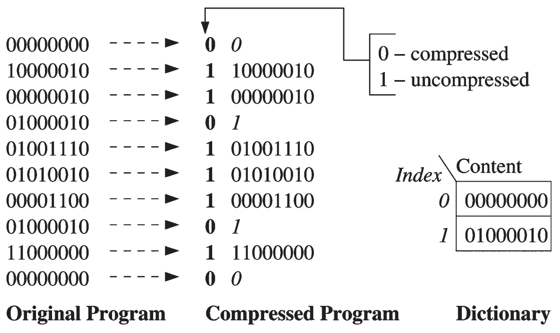

Fig. 2. Dictionary-based code compression: an example.

code-compression techniques. Sections IV and V present our code-compression algorithm and decompression mechanism, which are followed by a case study in Section VI. Finally, Section VII concludes this paper.

II. Related Work

The first code-compression technique for embedded processors was proposed by Wolfe and Chanin [5]. Their technique uses Huffman coding, and the compressed program is stored in the main memory. The decompression unit is placed between the main memory and the instruction cache. They used a Line Address Table (LAT) to map original code addresses to compressed block addresses. Lekatsas and Wolf [6] proposed a statistical method for code compression using arithmetic coding and Markov model. Lekatsas *et al.* [7] proposed a dictionary-based decompression prototype that is capable of decoding one instruction per cycle. The idea of using dictionary to store the frequently occurring instruction sequences has been explored by various researchers [9], [10]. Fig. 2 shows an example of the standard dictionary-based code compression.

The techniques discussed so far target reduced instruction set computer (RISC) processors. There has been a significant amount of research in the area of code compression for very long instruction word (VLIW) and explicitly parallel instruction computing (EPIC) processors. The technique proposed by Ishiura and Yamaguchi [11] splits a VLIW instruction into multiple fields, and each field is compressed by using a dictionary-based scheme. Nam *et al.* [12] also use dictionary-based scheme to compress fixed-format VLIW instructions. Various researchers have developed code-compression techniques for VLIW architectures with flexible instruction formats [13], [14]. Larin and Conte [13] applied Huffman coding for code compression. Xie *et al.* [14] used Tunstall coding to perform variable-to-fixed compression. Lin *et al.* [15] proposed a Lempel–Ziv–Welch (LZW)-based code compression for VLIW processors using a variable-sized-block method. Ros and Sutton [16] have used a postcompilation register reassignment technique to generate compression-friendly code. Das *et al.* [17] applied code compression on variable-length instruction-set processors.

Several techniques [3], [4] have been proposed to improve the standard dictionary-based code compression by considering mismatches. Fig. 4 shows an example of the improved dictionary-based code compression. The basic idea is to create repeating patterns from mismatches by storing the differences during code compression. These techniques are closest to our approach. We perform a detailed analysis of these techniques in Section III to demonstrate that our approach outperforms (improves the CR) the existing techniques by generating more repeating patterns without introducing any decompression penalty.

III. Dictionary-Based Code Compression

This section describes the existing dictionary-based approaches and analyzes their limitations. First, we describe the standard dictionary-based approach. Next, we describe the existing techniques that improve the standard approach by considering mismatches (hamming distance). Finally, we perform a detailed cost–benefit analysis of the recent approaches in terms of how many repeating patterns they can generate from the mismatches. This analysis forms the basis of our technique to maximize the repeating patterns using bitmasks.

A. Dictionary-Based Approach

Dictionary-based code-compression techniques provide compression efficiency as well as fast decompression mechanism. The basic idea is to take advantage of commonly occurring instruction sequences by using a dictionary. The repeating occurrences are replaced with a code word that points to the index of the dictionary that contains the pattern. The compressed program consists of both code words and uncompressed instructions. Fig. 2 shows an example of dictionary-based code compression using a simple program binary. The binary consists of ten 8-b patterns, i.e., a total of 80 b. The dictionary has two 8-b entries. The compressed program requires 62 b, and the dictionary requires 16 b. In this case, the CR is 97.5% [using (1)]. This example shows a variable-length encoding. As a result, there are several factors that may need to be included in the computation of the CR, such as byte alignments for branch targets and the address-mapping table.

B. Improved Dictionary-Based Approach

Recently proposed techniques [3], [4] improve the standard dictionary-based compression technique by considering mismatches. The basic idea is to find the instruction sequences that are different in a few bit positions (Hamming distance) and store that information in the compressed program and update dictionary (if necessary). CRs will depend on how many bit changes are considered during compression. Fig. 3 shows the encoding format used by these techniques for a 32-b program code.

It is obvious that if more bit changes are allowed, more matching sequences will be generated. However, the size of the compressed program will increase depending on the number of bit positions. Section III-C describes this topic in detail. Prakash *et al.* [3] considered only 1-b change for 16-b patterns (vectors). Ros and Sutton [4] considered a general scheme of up to 7-b changes for 32-b patterns and concluded that a 3-b change provides the best CR.

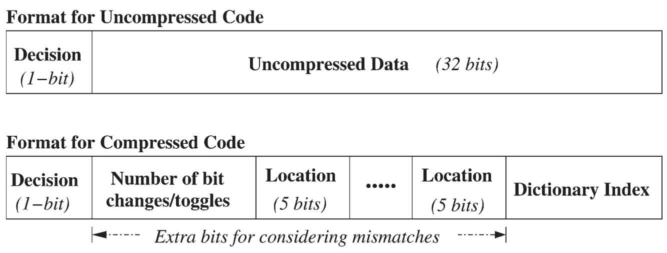
Format for Uncompressed Code
Decision (1–bit)
Uncompressed Data (32 bits)
Format for Compressed Code
Decision (1–bit)
Number of bit changes/toggles
Location (5 bits)
.....
Location (5 bits)
Dictionary Index
Extra bits for considering mismatches

Fig. 3. Encoding scheme for incorporating mismatches.

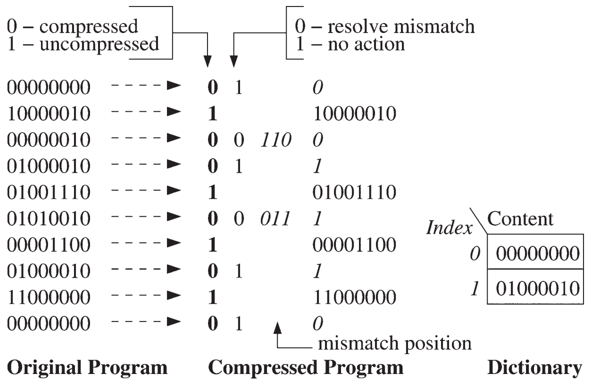

Fig. 4. Improved dictionary-based code compression.

Fig. 4 shows the improved dictionary-based scheme using the same example (shown in Fig. 2). This example considers only 1-b change. An extra field is necessary to indicate whether mismatches are considered or not. In case a mismatch is considered, another field is necessary to indicate the bit position that is different from an entry in the dictionary. For example, the third pattern (from top) in the original program is different from the first dictionary entry (index 0) on sixth bit position (from left). The CR for this example is 95%.

C. Cost–Benefit Analysis for Considering Mismatches

It is obvious that we can create more repeating patterns if we consider changes in more bit positions. For example, if we consider 2-b changes in Fig. 4, all mismatched patterns can be compressed. However, increasing more repeating patterns by considering multiple mismatches does not always improve the CR. This is due to the fact that the compressed program has to store multiple bit positions. If we consider 2-b changes for the example in Fig. 4, the CR will be worse (102.5%).

We have done a detailed study on how to match more bit positions without adding significant information in the compressed code. We have considered 32-b code vectors for compression. Clearly, the Hamming distance between any two 32-b vectors is between 0 and 32. The compression adds extra 5 b to remember each bit position in a 32-b 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-b changes, it requires extra 5 b to indicate the number of changes. As a result, this process requires extra 165 b $(32 \times 5 + 5)$ when all 32 b are different. Clearly, it is not profitable to compress a 32-b vector using extra 165 b along with a code word (index information) and other details.

TABLE I
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.

We have explored the use of bitmasks for creating repeating patterns. For example, a 32-b mask pattern is sufficient to match any two 32-b vectors. Of course, it is not profitable to store extra 32 b to compress a 32-b vector but definitely better than extra 165 b. We considered mask patterns of different sizes (1–32 b). When a mask pattern is smaller than 32 b, we need to store information related to starting bit position where the mask needs to be applied. For example, if we use an 8-b mask pattern and want to consider all 32-b mismatches, it requires four 8-b masks and extra 2 b (to identify one of the four bytes) for each mask pattern to indicate where it will be applied. In this particular case, we require extra 42 b.

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

Table I shows the summary of this analysis. Each row represents the number of changes allowed. Each column represents the size of the mask pattern. A 1-b mask is essentially the same as remembering the bit position. Each entry in the table (r, c) indicates how many extra bits are necessary to compress a 32-b vector when r number of bit changes are allowed and c is the size of the mask pattern. For example, we require extra 15 b to allow 8-b (row with value 8) changes using 4-b (column with value 4) mask patterns.

Section IV presents our code-compression technique using bitmasks, which significantly improves the CR. Consider the same example shown in Fig. 4. A 2-b mask (only on quarter-byte boundaries) is sufficient to create 100% matching patterns and thereby improves the CR (87.5%), as shown in Fig. 5. Experiments using real applications demonstrate that the CR using our approach varies between 50%–65%.

IV. CODE COMPRESSION USING BITMASKS

The motivation of this paper is based on the analysis presented in Section III-C. Our approach tries to incorporate maximum bit changes using mask patterns without adding

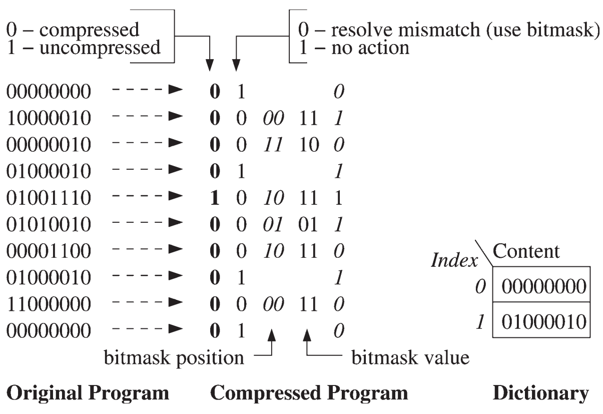

Fig. 5. Code compression using our approach.

significant cost (extra bits) such that the CR is improved. Our compression technique also ensures that the decompression efficiency remains the same compared to that of the existing techniques. Our scheme considers a 32-b program code (vector) and uses mask patterns.

Fig. 6 shows the generic encoding scheme used by our compression technique. This scheme is similar to the 32-b format shown in Fig. 3 where individual bit changes are recorded. However, as described in Section III-C, storing individual bit changes limits the number of matches. As shown in Fig. 6, a compressed code can store information regarding multiple mask patterns. For each pattern, the generic encoding stores the mask type (requires 2 b to distinguish between 1, 2, 4, and 8 b), the location where the mask needs to be applied, and the mask pattern.

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, an 8-b mask can be applied only on four places (byte boundaries). Similarly, a 4-b mask can be applied on eight places (byte and half-byte boundaries). Consider a scenario where a 32-b word is compressed by using one 4-b mask at the second half-byte boundary and one 8-b mask at the fourth byte boundary; the compressed code will appear as shown in Fig. 7.

The generic encoding scheme (shown in Fig. 6) can be further optimized. For code compression, we have found that using up to two bitmasks is sufficient to achieve a good CR. We explored various customized versions of our encoding format to figure out which encoding format works better across the target architectures. Clearly, a 32-b mask pattern is not profitable. The 16-b mask is also not useful unless there are too many mismatches which a 4- or 8-b (or combined 12-b) mask cannot capture. Fig. 8 shows three examples of customized encoding formats using 4- and 8-b masks. The first encoding (Encoding 1) uses an 8-b mask, the second encoding (Encoding 2) uses up to two 4-b masks, and the third encoding (Encoding 3) uses up to two masks where the first mask can be 4 or 8 b, whereas the second mask is always 4 b. Section IV-C describes mask selection and its challenges in detail.

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

A. Compression Algorithm

Algorithm 1 shows the four basic steps of our algorithm.

```
**Algorithm 1**: *Code Compression using Mask Patterns*
**Input**: Original code (binary) divided into 32-b 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-b vector using cost constraints.
    **Step 4**: Handle and adjust branch targets.
    **return** Compressed code and dictionary
End
```

The algorithm accepts the original code consisting of 32-b 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-b 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 I provides the cost for the choices. For example, it is costly to use two 4-b masks (cost: 15 b) if an 8-b mask (cost: 10 b) can create the match.

The second step chooses the smallest possible dictionary size without significantly affecting the CR. 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 because 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.

The third step converts each 32-b vector into compressed code (when possible) using the format shown in Fig. 6. The compressed code, along with any uncompressed ones, is 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 [5] proposed the LAT; however, it requires an extra space and degrades overall performance. Lefurgy *et al.* [9] proposed a technique which patches the original branch-target addresses to the new offsets in the compressed program. This approach does not require an additional space for the LAT nor affect the performance of the program, but it may not work on indirect branches.

Our proposed compression algorithm handles branch targets in the following manner. First, it patches all the possible branch targets into new offsets in the compressed program, and pad extra bits at the end of the code preceding branch targets to align on a byte boundary. Next, it creates a minimal mapping

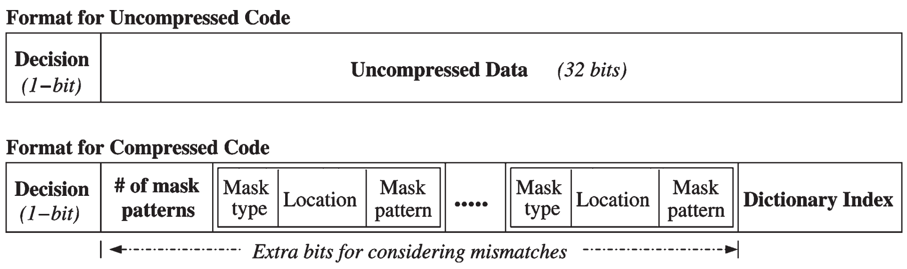

Fig. 6. Encoding format for our compression technique.

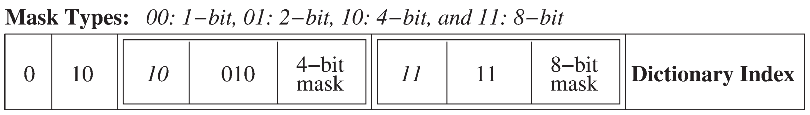

Fig. 7. Example of compressed word.

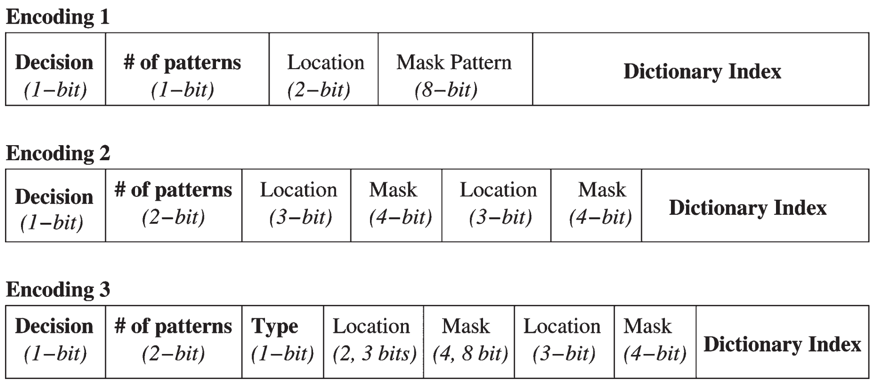

Fig. 8. Three customized encoding formats.

table to store the new addresses for 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. Our technique is very useful because more than 75% control-flow instructions are conditional branches (compare and branch) [18], 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.

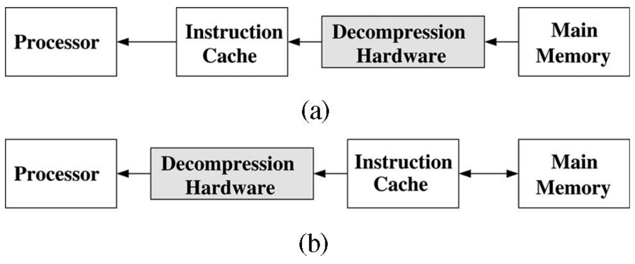

Fig. 9. Placement of decompression unit. (a) Precache placement. (b) Post-cache placement.

B. Decompression Mechanism

Embedded systems with caches can employ the decompression scheme in different ways, as shown in Fig. 9. For example, the decompression hardware can be used between the main memory and the instruction cache (pre-cache). As a result, the main memory will contain the compressed program, whereas the instruction cache will have the original program. Alternatively, the decompression engine (DCE) can be used between the instruction cache and the processor (post-cache).

The post-cache design has an advantage because the cache retains data still in a compressed form, which increases cache hits and reducing bus bandwidth, therefore achieving potential performance gain. Lekatsas *et al.* [7] reported a performance increase of 25% on average by using a dictionary-based code compression and a post-cache DCE. Decompression (decoding) time is critical for the post-cache approach. The decompression unit must be able to provide an instruction at the rate of the processor to avoid any stalling. We present a design of the dictionary-based decompression unit that handles bitmasks and uses post-cache placement of the decompression hardware. Our design facilitates simple and fast decompression and requires no modification to the existing processor core.

Our design of the decompression hardware is based on the one-cycle DCE proposed by Lekatsas *et al.* [7]. We have

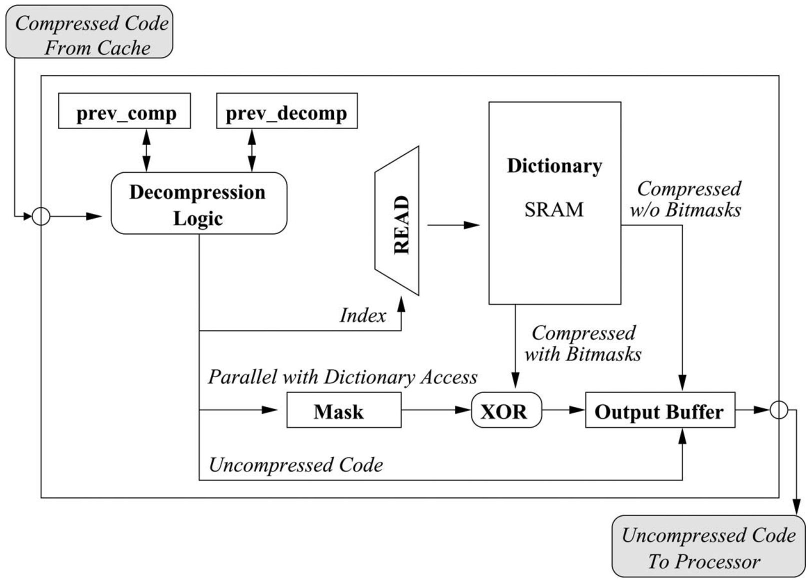

Fig. 10. DCE for bitmask encoding.

implemented the decompression hardware using VHSIC Hardware Description Language and synthesized it using Synopsys Design Compiler [20]. Our implementation is based on various generic parameters including dictionary size (index size), number and types of bitmasks, etc. Therefore, the same implementation can be used for different applications/architectures by instantiating it with appropriate set of parameters.

Fig. 10 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 bitmasks used. Using two 4-b masks (Encoding 2 in Section IV), the compression algorithm generates four different types of encoding: 1) uncompressed instruction; 2) compressed without bitmasks; 3) compressed with one 4-b mask; and 4) compressed with two 4-b masks. In the same manner, using one bitmask creates only three different types of encoding. Decoding of uncompressed or compressed code without bitmasks remains virtually identical to the previous approach. The design has prev_comp and prev_decomp registers. The prev_comp holds remaining compressed data from the previous cycle because not all of 32 b belong to the currently decoded instructions. The prev_decomp holds uncompressed data from the previous cycle. This is needed, for instance, when the DCE decompresses more than 32 b in a cycle (two or more original instructions were compressed in a 32-b code). The stored uncompressed data are sent to the CPU in the next cycle.

Our decompression unit provides two additional operations: generating an instruction-length (32-b) 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-b mask can be applied only on half-byte boundaries (eight locations). If two bitmasks were used, the two intermediate instruction-length masks need to be ORed 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-b mask does not add any additional penalty to the existing DCE.

The only additional time incurred in our design, compared with the previous one-cycle design, is in the last stage where the dictionary entry and the generated 32-b mask are XORed. We have surveyed the commercially manufactured XOR logic gates and found that many of the manufacturers produce XOR gates with propagation delays ranging from 0.09 to 0.5 ns, numerous of which are under 0.25 ns. The critical path of decompression data stream in [7] was 5.99 ns (with the clock cycle of 8.5 ns). Addition of 0.25 ns to the critical path of 5.99 ns satisfies the 8.5-ns clock-cycle constraint.

In addition, our DCE can decode more than one instruction in one cycle (even up to three instructions with hardware support). In dictionary-based code compression, approximately 50% of instructions match with each other (without using bitmasks or Hamming distance) [4]. Our technique captures additional 15%–25% using one bitmask and up to 15%–25% more using two bitmasks. Therefore, only about 5%–10% of the original program remains uncompressed.

If the code word (with the dictionary index) is 10 b, the encoding of instructions that are compressed only by using the dictionary will be 12 b or less. Instructions that are compressed with one 4-b mask have the cost of additional 7 b (a total of 18–19 b). Therefore, a 32-b stream with any combination with a 12-b code contains more than one instruction and can be decoded simultaneously. The best scenario is when a 32-b stream contains two 12-b encodings and when prev_comp holds remaining 4 b. In this scenario, the DCE engine has three instructions in hand that can be decoded concurrently.

The decompression unit, as well as the dictionary (SRAM), consumes memory space. However, the computation of the CR includes the space required for the dictionary. Therefore, when 40% code compression (60% CR) is reported, it already accounted for the area occupied by the dictionary. However, the decompression unit area is not accounted in the calculation.

Although the size of the decompression unit (excluding dictionary size) can vary based on the number of bitmasks, etc., but it ranges from 5120 to 10 240 gates. However, the savings due to code compression is significantly higher than the area overhead of the decompression hardware. For example, an MPEGII encoder has initial size of 110 kB which can be reduced to 60 kB. Therefore, a 64-kB memory is sufficient instead of a 128-kB memory. In terms of power requirement, our implementation requires 2 mW on average. A typical system-on-chip (SOC) requires several-hundred-milliwatt power. As shown by Lekatsas *et al.* [8], 50% code compression can lead to 22%–80% energy reduction due to performance improvement and memory-size reduction. Therefore, the power overhead of the decompression hardware is negligible.

C. Challenges in Bitmask-Based Code Compression

One of the major challenges in bitmask-based code-compression (BCC) technique is how to determine (a set of) optimal mask patterns that will maximize the matching sequences while minimizing the cost of bitmasks. A 2-b mask can handle up to four types of mismatches, whereas a 4-b 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 bitmask pattern is associated with a higher cost. Similarly, applying more bitmasks is not always beneficial. For example, applying a 4-b mask requires 3 b to indicate its position (eight possible locations in a 32-b vector) and 4 b to indicate the pattern (a total of 7 b), whereas an 8-b mask requires 2 b for the position and 8 b for the pattern (a total of 10 b). Therefore, it would be more costly to use two 4-b masks if one 8-b mask can capture the mismatches.

Another major challenge in bitmask-based compression is how to perform dictionary selection where existing and bitmask-matched repetitions need to be considered. In the traditional dictionary-based compression approach, the dictionary-entry selection process is simplified because it is evident that the frequency-based selection will give the best CR. However, when compressing by using bitmasks, the problem is complex, and the frequency-based selection will not always yield the best CR. Figs. 11 and 12 demonstrate this fact.

When only one dictionary entry is allowed, the pure frequency-based selection will choose 0000000, yielding a CR of 97.5% (Fig. 11). However, if 01000010 was chosen, we can achieve a CR of 87.5% (Fig. 12) for the same input program. Clearly, there is a need for efficient mask- and dictionary-selection techniques to improve the efficiency of BCC.

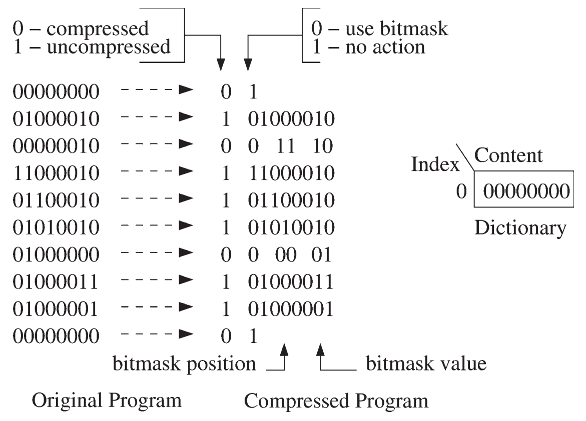

Fig. 11. Compression using frequency-based dictionary selection.

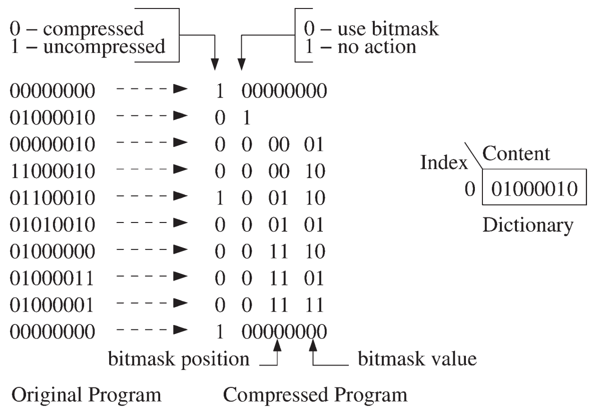

Fig. 12. Compression using a different dictionary selection.

V. Application-Aware Code Compression

This section addresses the challenges described in Section IV-C to develop an improved bitmask-based code-compression framework. We have developed application-specific bitmask-selection and bitmask-aware dictionary-selection techniques. 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- and the dictionary-selection techniques.

A. Mask Selection

As discussed in Section IV-C, 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 CR for a given application. This leads to answering the following two questions: 1) How many bitmask patterns do we need, and 2) which bitmask patterns are profitable? We will answer these questions after defining few terms related to bitmask patterns.

Table II 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-b fixed mask (referred to as $8f$) is applicable on four fixed locations (byte boundaries) on a 32-b vector. A "sliding" mask pattern can be applied anywhere. For example, an 8-b sliding mask (referred to as $8s$) can be applied in any location on a 32-b vector. There is no difference between fixed and sliding ones for a 1-b mask. We will use a 1-b sliding mask (referred to as $1s$) for uniformity.

TABLE II
VARIOUS BITMASK 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

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-b fixed mask can be applied only on four places (byte boundaries), therefore requiring 2 b. Similarly, a 4-b fixed mask can be applied on eight places (byte and half-byte boundaries) and requires 3 b for its position. A sliding pattern will require 5 b to locate the position regardless of its size. For instance, a 4-b sliding mask requires 5 b for location and 4 b for the mask itself.

If we choose two distinct bitmask patterns, 2-b fixed $(2f)$ and 4-b sliding $(4s)$, we can generate six combinations: (2f), (4f), (2f, 2f), (2f, 4f), (4f, 2f), and (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 pieces of bitmask information (position and pattern) is 15 b (if three 1-b sliding patterns are used). In addition, we need 1–5 b to indicate the mask combination and 8–14 b for a code word (dictionary index). Therefore, we require approximately 29 b (on average) to encode a 32-b vector. In other words, we save only 3 b to match 3-b differences (on a 32-b 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 bitmasks are profitable? As discussed in Section IV-C, applying a larger bitmask can generate more matching patterns. However, it may not improve the CR. Similarly, using a sliding mask where a fixed one is sufficient is wasteful because a fixed mask requires fewer number of bits (compared with its sliding counterpart) to store the position information. For example, if a 4-b sliding mask (cost of 9 b) is used where a 4-b fixed one (cost of 7 b) is sufficient, two additional bits are wasted.

We carefully studied the combinations of up to two bitmasks 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 II) produce a better CR compared with nonfactors (e.g., masks 3, 5, 6, and 7). This is due to the fact that we accept the program of 32-b vectors; therefore, nonfactor-sized bitmasks were only usable as a sliding pattern. While sliding patterns are more flexible, they are more costly than fixed patterns. These observations allowed us to reduce the 11 mask patterns in Table II to seven profitable mask patterns shown in Table III. A subset of these experiments is reported in Section VI. We analyzed the result of CRs using various mask combinations and made several useful observations that helped us to further reduce the bitmask-pattern table. We found that $8f$ and $8s$ are not helpful and that $4s$ does not perform better than $4f$. We also observed that using two bitmasks provides a better CR than using one bitmask alone. The final set of profitable bitmask patterns is shown in Table IV. Our integrated compression technique (Algorithm 3 in Section V-C) uses the bitmask patterns from Table IV.

TABLE III
PROFITABLE BITMASK PATTERNS

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

TABLE IV
FINAL BITMASK PATTERNS

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

B. Bitmask-Aware Dictionary Selection

Dictionary selection is another major challenge in code compression. The optimal dictionary selection is a nondeterministic polynomial-time-hard problem [19]. 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 [15] accelerates the compression time, it seldom matches the CR 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 [4].

We have observed that neither frequency- 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 2 shows our bit-saving-based dictionary-selection technique. The algorithm takes application(s) consisting of 32-b vectors as input and produces the dictionary as output that will deliver a good CR. It first creates a graph where the nodes are the unique 32-b vectors. An edge is created between two nodes if they can be matched using a bitmask pattern. It is possible to have multiple edges between two nodes because 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 2: Bit-Saving-Based Dictionary Selection
Inputs: 1. Application(s) consisting of 32-b instruction vectors
        2. Mask patterns
        3. A threshold value to screen deletion of nodes
Output: Optimized dictionary
Begin
  Step 1: Create a graph representation, $\mathbf{G} = (V, E)$.
          Each node ($V$) is a unique 32-b vector.
          An edge ($E$) indicates that a bitmask 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-saving distribution of all nodes.
  Step 3: Select the most profitable node $N$.
  Step 4: Remove $N$ from $\mathbf{G}$ and insert into dictionary.
  Step 5: For each node $N_i$ in $\mathbf{G}$ that is connected to $N$
          If the node profit of $N_i$ is less than certain threshold
          Remove $N_i$ from $\mathbf{G}$.
  Step 6: Repeat Steps 3–5 until dictionary is full or $\mathbf{G}$ 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, is 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 that is less than ten 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.

Fig. 13 illustrates this technique. The vertex "A" has the total savings of 10 $(5 + 5)$, "B" and "C" have 22, "D" has 5, "E" has 15, "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. The nodes "C" and "E" are also removed because they can be matched with "F" in the dictionary and bitmask(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.

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.

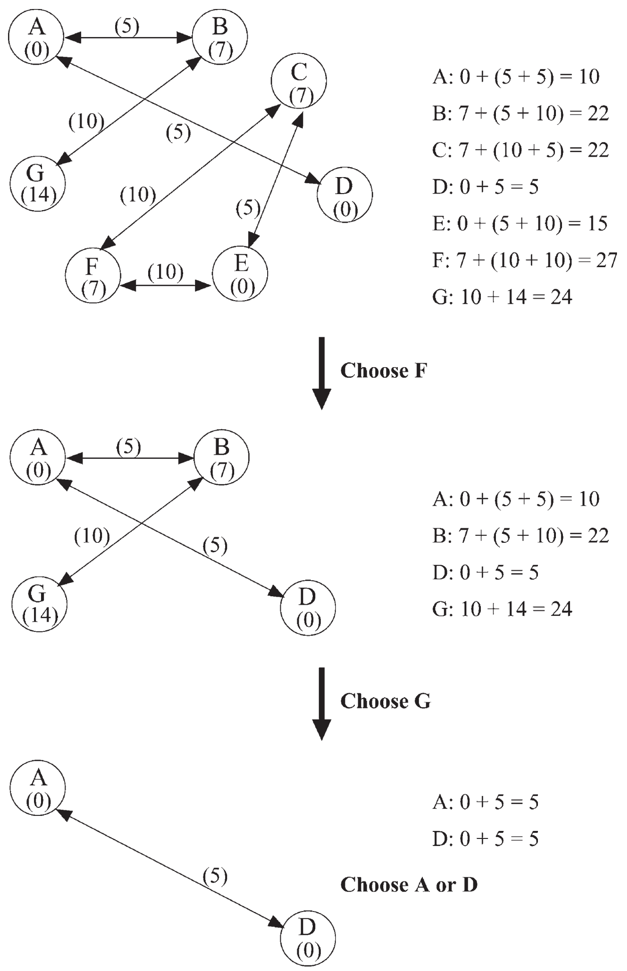

Fig.13. Bit-saving dictionary-selection method.

C. Code-Compression Algorithm

In this section, we present our code-compression algorithm that integrates our mask- and dictionary-selection methods. The goal is to maximize the compression efficiency by using the BCC technique. Algorithm 3 outlines the basic steps. The algorithm accepts the original code consisting of 32-b vectors as input and produces the compressed code and an optimized dictionary.

The algorithm begins by initializing three variables: $mask_1$, $mask_2$, and *CompressionRatio*. The profitable mask patterns are stored in $mask_1$ and $mask_2$, and *CompressionRatio* stores the best CR 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 IV. The second step selects the optimized dictionary using Algorithm 2. The third step converts each 32-b vector into compressed code (when possible). If the new CR 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.

It is important to note that this algorithm can be used as a one- 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 two most profitable mask patterns. During the second pass, the first step (two for loops) can be ignored, and the actual code compression can be performed by using real applications.

```
Algorithm 3: Code Compression using Bitmasks
Input: Original code (32-b vectors)
Outputs: Compressed code, dictionary, $\langle mask_1, mask_2 \rangle$
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-b vectors using cost constraints.
      Step 4: Update the variables if necessary.
      NewCR = (CompCode with $m_i \& m_j$) ÷ (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, $\langle mask_1, mask_2 \rangle$
End
```

VI. Experiments

We performed extensive code-compression experiments by varying both application domains and target architectures. In this section, we present our experimental results. The benchmarks are collected from TI, Mediabench, and MiBench benchmark suites: adpcm_en, adpcm_de, cjpeg, djpeg, gsm_to, gsm_un, hello, modem, mpeg2enc, mpeg2dec, pegwit, and vertibi. 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 CR using (1). Our computation of compressed program size includes the size of the compressed code as well as the dictionary and the small mapping table.

A. Results

In Section IV, we presented our generic encoding format as well as three customized formats. Encoding 1 uses one 8-b mask, Encoding 2 uses up to two 4-b masks, and Encoding 3 uses 4- and 8-b masks. Fig. 14 shows the performance of each of these encoding formats using adpcm_en benchmark for three target architectures. We used dictionary with 2048 entries for these experiments. Clearly, the second encoding format performs the best—generating a CR of 55%–65%.

Fig. 15 shows the efficiency of our compression technique for all benchmarks compiled for SPARC using dictionary sizes of 4096 and 8192 entries. We used the Encoding 2 to compress the benchmarks. As expected, we can observe three scenarios. The

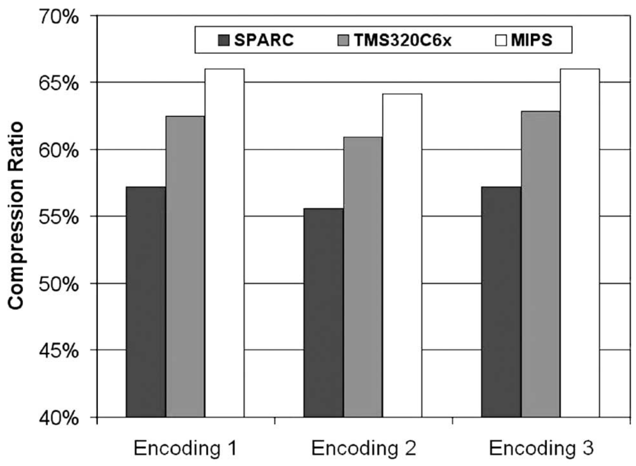

Fig. 14. CR for adpcm_en benchmark.

small benchmarks such as adpcm_en and adpcm_de performs better with small dictionary because a majority of the repeating patterns fit in the 4096-entry 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.

We experimented by varying both mask combinations and dictionary-selection methods. Fig. 16 shows the CRs of three TI benchmarks (block_mse, modem, and vertibi) compressed using all 56 different mask-set combinations[1] (both one- and two-mask combinations from $\{1s, 2f, 2s, 4f, 4s, 8f, 8s\}$). As discussed in Section V-A, 8-b mask patterns (fixed or sliding) do not provide good CR. In general, compressing with two masks achieves a better CR than using just one. Note that the CRs for three benchmarks follow a regular pattern. A similar pattern exists even with other benchmarks. It confirms our analysis in Section V-A that a small set of mask patterns is sufficient to achieve good compression. Overall, we found that the combination of 4-b fixed and 1-bit sliding or two 2-b patterns provides the best compression.

Fig. 17 compares the CRs achieved by the various dictionary-selection methods described in Section V-B. As shown in the figure, spanning-based approach is the worst compared to other dictionary-selection methods. Our bit-saving-based approach outperforms all the existing dictionary-selection methods on all benchmarks.

Fig. 18 compares the CRs between the BCC technique and the application-specific code-compression (ACC) framework. In BCC technique [1], we have experimented with customized encodings of 4- and 8-b mask combinations. In ACC framework [2], we have computed the most profitable mask pairs and applied the bit-saving-based dictionary-selection technique to further improve the CR. For example, we obtained 57% CR for adpcm_en benchmark using 4-b fixed and 1-b sliding patterns, which outperforms the BCC approach by 6%. As expected, the application-specific approach outperforms the bitmask-based technique by 5%–10%.

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

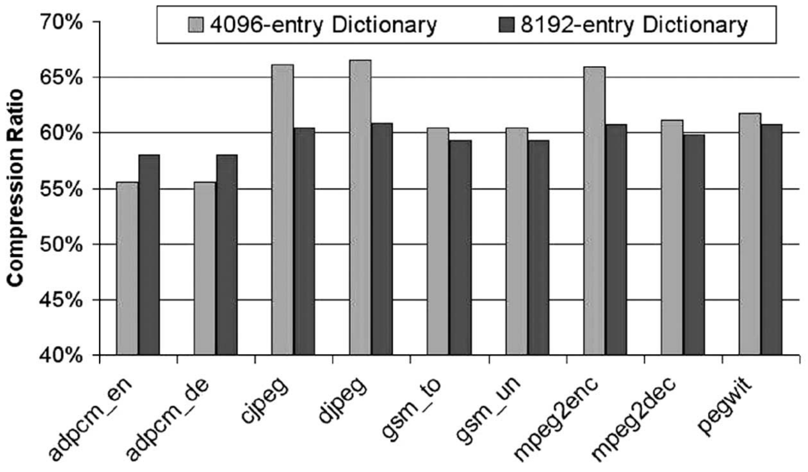

Fig. 15. CR for different benchmarks.

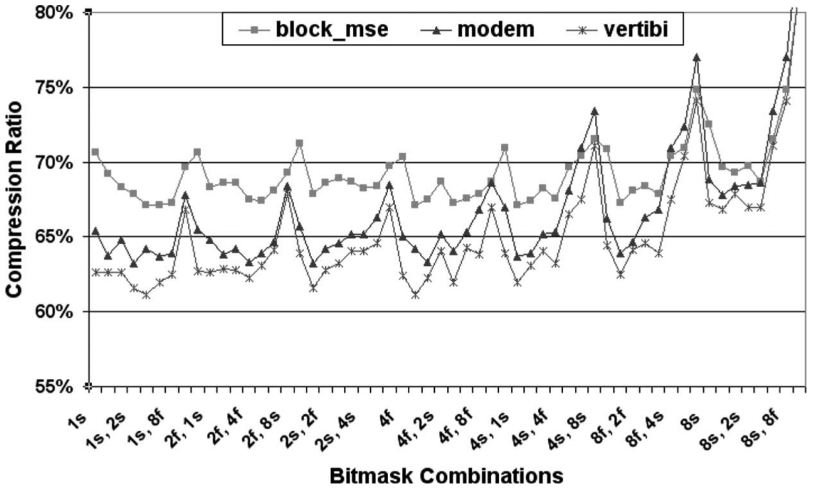

Fig. 16. Performance analysis of mask combinations.

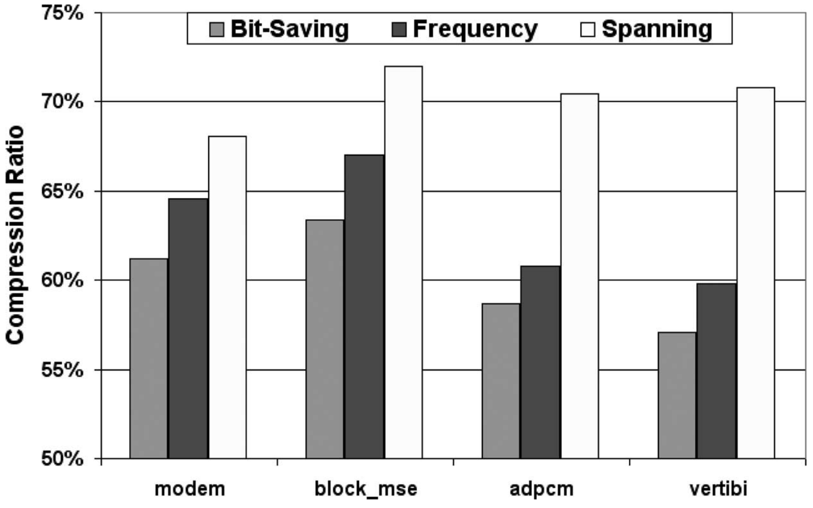

Fig. 17. Comparison of dictionary-selection methods.

Table V compares our approach with the existing code-compression techniques. Our technique improves the code-compression efficiency by 20% compared to the existing dictionary-based techniques [3], [4]. It is important to note that all the works mentioned in Table V did not use exactly the same setup. In fact, in some of them, the detailed setup information is not available except the information regarding the architecture and the average CR. However, majority of them (including all the recent researches in this area) used popular embedded-system benchmark applications from Mediabench,

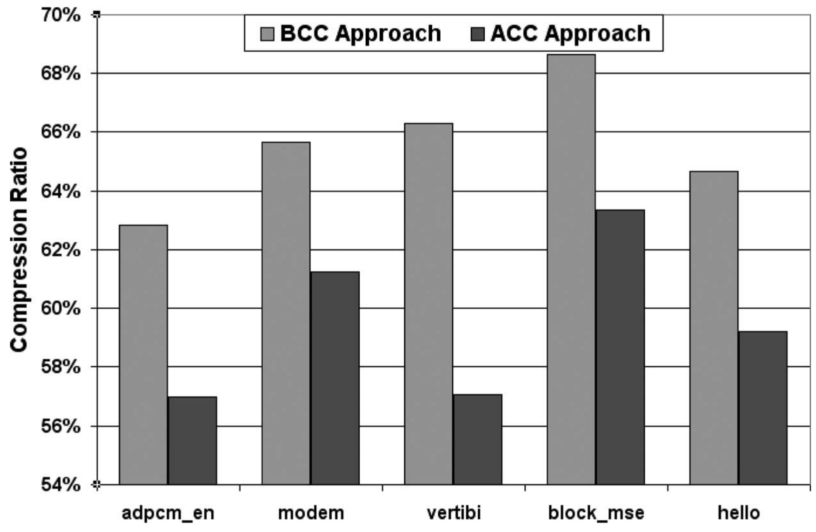

Fig. 18. CR comparison.

TABLE V
COMPARISON WITH VARIOUS COMPRESSION SCHEMES

Compression Method	Target Architecture	Compression Ratio*	Decompression Bandwidth
Wolfe [5]	MIPS	73%	8 bits
IBM [21] CodePack	PowerPC	60%	8 bits
SAMC [6]	MIPS	57%	6-8 bits
V2F [14]	TMS320C6x	70-82%	4.9-13 bits
MCSSC [15]	TMS320C6x	75%	14.5-64 bits
Prakash [3]	TMS320C6x	76-80%	N/A
Ros [4]	Itanium TMS320C6x	72-80%	N/A
Our Approach	MIPS, SPARC TMS320C6x	55-65%	32-64 bits

*Smaller compression ratio implies better compression technique.

MiBench, and TI benchmark suites, which were compiled for various architectures. We have obtained the same application binary used by Lekatsas and Wolf [6]. In other words, we have tried our best to perform a fair comparison.

The compression efficiency of our technique is comparable to the state-of-the-art compression techniques (IBM CodePack [21] and SAMC [6]). However, due to the encoding complexity, the decompression bandwidths of those techniques are only 6–8 b. As a result, they cannot 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 (Fig. 9). Moreover, those techniques do not support parallel decompression, which are, therefore, not suitable for VLIW architectures. Our decompression mechanism supports one instruction per cycle delivery as well as parallel decompression.

This code-size reduction can contribute not only to cost, area, and energy savings but also to performance of the embedded system. The ACC framework [2], due to the nature of the mask- and dictionary-selection procedures, incurs higher encoding/compression overhead than the BCC approach [1]. However, in embedded-system 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 approaches (BCC or ACC) do not introduce any decompression penalty.

VII. CONCLUSION

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 because they generate a good CR by exploiting the code repetitions. Recent techniques use bit-toggle information to create matching patterns and thereby improve the CR. However, due to the lack of an efficient matching scheme, the existing techniques can match up to 3-b differences.

We developed an efficient matching scheme using bitmasks that can significantly improve the code-compression efficiency. This paper studied various code-compression challenges in embedded systems. To address these challenges, we developed application-specific bitmask-selection and bitmask-aware dictionary-selection algorithms. We also developed an efficient code-compression technique using these algorithms to improve the code CR without introducing any decompression overhead. 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 20%, giving CRs 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.

There are two alternative ways to employ bitmask-based code compression: 1) compressing with the simple frequency-based dictionary selection and pre-customized (selected) encodings, or 2) compressing with the application-specific bitmask and dictionary selections. Clearly, the first approach is faster than the second one, but it may not generate the best possible compression. The first option is useful for early exploration and prototyping purposes. The second option is time consuming but is useful for the final system design because encoding (compression) is performed only once and millions of copies are manufactured. Therefore, any reduction in cost, area, or energy requirements is extremely important during embedded-system design.

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 plan to investigate the effects of our compression approach on overall energy savings and performance improvement.

This paper used bitmask-based compression for reducing the code size in embedded systems. This technique can also be applied in other domains where dictionary-based compression is used. For example, dictionary-based test data compression [19] is used in manufacturing test domain for reducing the test-data volume in SOC designs. This method is based on the use of a small number of channels to deliver compressed test patterns from the tester to the chip and to drive a large number of internal scan chains in the circuit under test. Therefore, it is particularly suitable for a reduced pin-count and low-cost test environment, where a narrow interface between the tester and the SOC is desirable. The dictionary-based approach not only reduces test-data volume but also eliminates the need for additional synchronization and handshaking between the SOC and the automatic test equipment. The required pin count and overall cost can be further reduced by employing the bitmask-based compression technique. Our future work includes application of bitmask-based technique for test-data compression.

References

[1] S. Seong and P. Mishra, "A bitmask-based code compression technique for embedded systems," in *Proc. ICCAD*, 2006, pp. 251–254.

[2] S. Seong and P. Mishra, "An efficient code compression technique using application-aware bitmask and dictionary selection methods," in *Proc. DATE*, 2007, pp. 1–6.

[3] J. Prakash, C. Sandeep, P. Shankar, and Y. Srikant, "A simple and fast scheme for code compression for VLIW processors," in *Proc. DCC*, 2003, p. 444.

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

[5] A. Wolfe and A. Chanin, "Executing compressed programs on an embedded RISC architecture," in *Proc. Int. Symp. MICRO*, 1992, pp. 81–91.

[6] H. Lekatsas and W. Wolf, "SAMC: A code compression algorithm for embedded processors," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 18, no. 12, pp. 1689–1701, Dec. 1999.

[7] H. Lekatsas, J. Henkel, and V. Jakkula, "Design of an one-cycle decompression hardware for performance increase in embedded systems," in *Proc. Des. Autom. Conf.*, 2002, pp. 34–39.

[8] H. Lekatsas, J. Henkel, and W. Wolf, "Code compression for low power embedded system design," in *Proc. DAC*, 2000, pp. 294–299.

[9] C. Lefurgy, P. Bird, I. Chen, and T. Mudge, "Improving code density using compression techniques," in *Proc. Int. Symp. MICRO*, 1997, pp. 194–203.

[10] S. Liao, S. Devadas, and K. Keutzer, "Code density optimization for embedded DSP processors using data compression techniques," in *Proc. Adv. Res. VLSI*, 1995, pp. 393–399.

[11] N. Ishiura and M. Yamaguchi, "Instruction code compression for application specific VLIW processors based on automatic field partitioning," in *Proc. SASIMI*, 1997, pp. 105–109.

[12] S. Nam, I. Park, and C. Kyung, "Improving dictionary-based code compression in VLIW architectures," *IEICE Trans. Fundam.*, vol. E82-A, no. 11, pp. 2318–2324, Nov. 1999.

[13] S. Larin and T. Conte, "Compiler-driven cached code compression schemes for embedded ILP processors," in *Proc. Int. Symp. MICRO*, 1999, pp. 82–91.

[14] Y. Xie, W. Wolf, and H. Lekatsas, "Code compression for VLIW processors using variable-to-fixed coding," in *Proc. ISSS*, 2002, pp. 138–143.

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

[16] M. Ros and P. Sutton, "A post-compilation register reassignment technique for improving hamming distance code compression," in *Proc. CASES*, 2005, pp. 97–104.

[17] D. Das, R. Kumar, and P. P. Chakrabarti, "Dictionary based code compression for variable length instruction encodings," in *Proc. VLSI Des.*, 2005, pp. 545–550.

[18] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 2003.

[19] L. Li, K. Chakrabarty, and N. Touba, "Test data compression using dictionaries with selective entries and fixed-length indices," *ACM Trans. Des. Autom. Electron. Syst. (TODAES)*, vol. 8, no. 4, pp. 470–490, Oct. 2003.

[20] Synopsys. [Online]. Available: http://www.synopsys.com

[21] *CodePack PowerPC Code Compression Utility User's Manual*, IBM, Armonk, NY, 1998. Version 3.0. [Online]. Available: http://www.ibm.com

Seok-Won Seong received the B.S. degree in computer and information sciences and the M.S. degree in computer engineering from the University of Florida, Gainesville, in 2003 and 2006, respectively. He is currently working toward the Ph.D. degree in the Department of Electrical Engineering, Stanford University, Stanford, CA.

Prior to joining Stanford, he was with IBM as an IT specialist. His research interests include code compression in embedded systems, data integration through transformations, context-sensitive program analysis, mobile computing, and networking.

Prabhat Mishra (S'00–M'04) received the B.E. degree in computer science and engineering from Jadavpur University, Calcutta, India, in 1994, the M.Tech. degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, India, in 1996, and the Ph.D. degree in computer science and engineering from the University of California, Irvine, in 2004.

He spent several years with various semiconductor and design automation companies including Texas Instruments, Synopsys, Intel, and Freescale (Motorola). He is currently an Assistant Professor with the Department of Computer and Information Science and Engineering, University of Florida, Gainesville. His research interests include design automation of embedded systems, multicore architectures, functional verification, and very large scale integration (VLSI) computer-aided design. He is the coauthor of the book *Functional Verification of Programmable Embedded Architectures* (Kluwer, 2005). He is also the co-editor of the forthcoming book *Processor Description Languages* (Morgan Kaufmann, 2008).

Dr. Mishra is a member of the Association for Computing Machinery (ACM). He currently serves as the Information Director of *ACM Transactions on Design Automation of Electronic Systems* and as a program committee member of several premier design automation conferences, including DATE, CODES+ISSS, ISCAS, VLSI Design, EUC, and I-SPAN. His research has been recognized by various awards, including an NSF CAREER Award in 2008, the EDAA Outstanding Dissertation Award in 2005 and the CODES+ISSS Best Paper Award in 2003.