DICTIONARY-BASED CODE COMPRESSION TECHNIQUES USING BIT-MASKS FOR EMBEDDED SYSTEMS

By

SEOK-WON SEONG

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2006

To my family, the greatest inspirer

ACKNOWLEDGMENTS

Most of all, I would like to thank Dr. Prabhat Mishra for his perpetual support and guidance. As parents teach their newborn's first step, he guided my first step into the research field. Also I would like to express my gratitude to Dr. Sartaj Sahni and Dr. Sanjay Ranka for serving on the supervisory committee. Their constructive comments, questions, and assistance were invaluable throughout the program. It was my great pleasure and honor to have them on the committee.

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS iv

LIST OF TABLES vi

LIST OF FIGURES vii

ABSTRACT viii

CHAPTER

1 INTRODUCTION 1

- 1.1 Overview and Motivation 1
- 1.2 Related Works 2
- 1.3 Contributions of Thesis 3

2 DICTIONARY-BASED CODE COMPRESSION 5

- 2.1 Dictionary-based Approach 5
- 2.2 Improved Dictionary-based Approach 6
- 2.3 Cost-Benefit Analysis for Considering Mismatches 7

3 CODE COMPRESSION USING BIT-MASKS 11

- 3.1 Mask Selection 12
- 3.2 Dictionary Selection 16
- 3.3 Compression Algorithm 20
- 3.4 Decompression Mechanism 21

4 EXPERIMENTS 23

- 4.1 Experimental Setup 23
- 4.2 Results 23

5 CONCLUSIONS AND FUTURE DIRECTIONS 27

- 5.1 Conclusions 27
- 5.2 Future Work Directions 28

REFERENCES 29

BIOGRAPHICAL SKETCH 31

LIST OF TABLES

Table | page
2–1 Cost of Various Matching Schemes 9
3–1 Various Bit-Mask Patterns 13
3–2 Profitable Bit-Mask Patterns 14
3–3 Final Bit-Mask Sets 14
4–1 Cost of Various Matching Schemes 26

LIST OF FIGURES

Figure		page
1–1	Traditional Code Compression Methodology	1
2–1	Dictionary-based Code Compression: An Example	6
2–2	Encoding Scheme for Incorporating Mismatches	6
2–3	Improved Dictionary-Based Code Compression	7
2–4	Code Compression Using Bit-Mask Approach	9
3–1	Encoding Format for Our Compression Technique	11
3–2	Dictionary Selection Methods	16
3–3	Compression with Different Dictionary Selection Methods	17
3–4	Bit Saving Dictionary Selection Method	20
3–5	Implementation of a Decompression Unit	22
4–1	Three Customized Encoding Formats	24
4–2	Compression Ratio for adpcm_en Benchmark	24
4–3	Compression Ratio for Different Benchmarks	25
4–4	Dictionary Selection Method Comparison	26

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

DICTIONARY-BASED CODE COMPRESSION TECHNIQUES USING BIT-MASKS FOR EMBEDDED SYSTEMS

By

Seok-Won Seong

May 2006

Chair: Prabhat Mishra
Major Department: Computer and Information Science and Engineering

Memory is one of the most restricted resource in the embedded system design. Code compression techniques have been proposed to address this issue by reducing the code size of application programs. Dictionary-based code compression techniques are popular as they offer both good compression ratio and fast decompression scheme. Recently proposed techniques improve standard dictionary-based compression by considering mismatches. We propose a bit-mask based code compression technique to aggressively minimize mismatches and improve compression efficiency. This thesis makes three important contributions: i) it proposes a compression-aware bit-mask selection procedure for creating more matching patterns, ii) it develops a dictionary selection technique to minimize the code size based on the selected bit-masks, and iii) it presents an efficient code compression algorithm using the mask selection and the dictionary selection procedures to improve compression ratio without introducing any decompression penalty. To demonstrate the usefulness of this approach, we used applications from various domains and compiled for a variety of architectures. Our technique outperforms the existing dictionary-based compression methods by an average of 15%, giving a compression ratio of 55% - 65%.

CHAPTER 1
INTRODUCTION

1.1 Overview and Motivation

Embedded systems are everywhere from household appliances to biomedical, military, geological and space equipments. The complexity of such systems is increasing at an exponential rate due to both advancements in technology and demands for sophisticated applications, in the areas of communication, multimedia, networking and entertainment. Memory is one of the key driving factors in embedded system design since a larger memory indicates an increased chip area and a higher cost. As a result, memory imposes constraints on the size of the application programs. Code compression techniques address the problem by reducing the program size.

Figure 1–1 shows the traditional code compression and decompression flow where the compression is done off-line (prior to execution) and the compressed program is loaded into the memory. The decompression is done during the program execution (online).

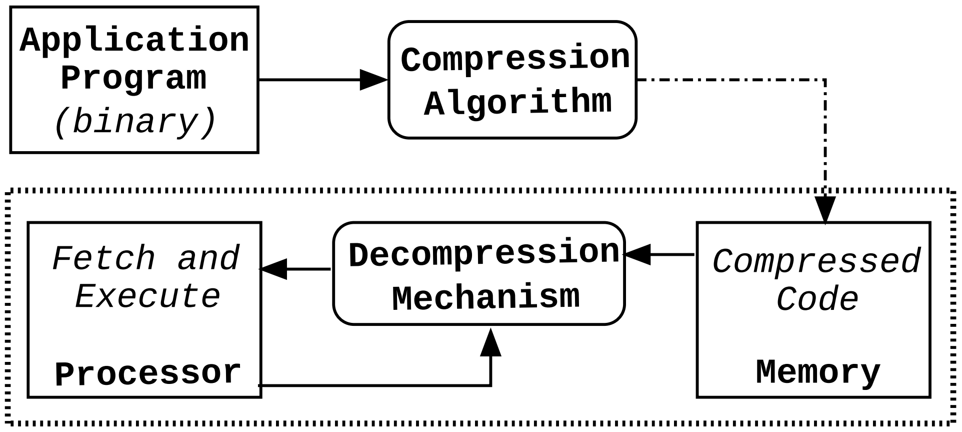

Figure 1–1: Traditional Code Compression Methodology

Embedded systems with caches can employ the decompression scheme in different ways. For example, the decompression hardware can be used between the main memory and the instruction 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 can be used between instruction cache and the processor, which increases cache hits and achieves potential performance gain.

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

$$Compression\ Ratio = \frac{Compressed\ Program\ Size}{Original\ Program\ Size} \tag{1.1}$$

Therefore, the smaller the compression ratio is, the better the compression technique. Dictionary-based code compression techniques are popular because it provides both good compression ratio and fast decompression mechanism. The basic idea is to exploit repeating instruction sequences by using a dictionary. The remainder of this chapter is organized as follows. Section 1.2 describes various code compression and decompression mechanisms available in the literature. Section 1.3 provides an overview of the problems that will be addressed in the rest of the thesis and outlines a brief summary of the thesis contributions.

1.2 Related Works

The first code compression technique for embedded processors was proposed by Wolfe and Chanin [13]. Their technique uses Huffman coding and the compressed program is stored in the main memory. The decompression unit is placed between main memory and the instruction cache. They used a Line Address Table (LAT) to map original code addresses to compressed block addresses. The idea of using dictionary to store the frequently occurring instruction sequences has been explored by various researchers [4, 7]. Lekatsas and Wolf [6] proposed a statistical method for code compression using arithmetic coding and Markov model.

The techniques discussed so far target RISC processors. There has been a significant amount of research in the area of code compression for VLIW and EPIC processors. The technique proposed by Ishiura and Yamaguchi. [2] splits a VLIW instruction into multiple fields and each field is compressed using a dictionary based scheme. Nam et al. [9] also uses dictionary based scheme to compress fixed format VLIW instructions. Various researchers have developed code compression techniques for VLIW architectures with flexible instruction formats [3, 14]. Larin and Conte [3] applied Huffman coding for code compression. Xie et al. [14] used Tunstall coding to perform variable-to-fixed compression. Lin et al. [8] proposed a LZW-based code compression for VLIW processors using a variable-sized-block method. Ros and Sutton [12] have used a post-compilation register reassignment technique to generate compression friendly code. Das et al. [1] applied code compression on variable length instruction set processors.

Several techniques [10, 11] have been proposed to improve the standard dictionary based code compression by considering mismatches. However, the efficiency of these techniques are limited by the number of bit changes (hamming distance) used during compression. The cost of storing the information for more bit positions cancels out the advantage gained by generating more repeating instruction sequences. Studies [11] have shown that it is not profitable to consider more than three bit changes when 32-bit vectors are used for compression. Prakash et al.[10] have considered mismatch in only one bit position.

1.3 Contributions of Thesis

In the thesis, I propose an efficient code compression technique to improve the compression ratio further by creating more matching sequences using bit-mask patterns. Our technique introduces many challenges to make it viable in practice. First, how to select a set of mask-patterns that will maximize the matching sequences while minimize the cost of using bit-masks? Second, how to perform efficient dictionary selection using the profitable bit-masks. This thesis develops a compression-aware bit-mask selection and dictionary section algorithms to improve the compression efficiency without introducing any decompression penalty.

I have collected applications from various domains including MediaBench and MiBench, and compiled for a wide variety of architectures including TI TMS320C6x, MIPS, and SPARC. The experimental results demonstrate that our approach outperforms the existing dictionary based compression techniques by an average of 15% without introducing any additional decompression penalty. The techniques developed in this thesis can also be applied in other domains where dictionary-based code compression is used such as test compression for manufacturing testing [**?**].

The rest of the thesis is organized as follows. Chapter 2 analyzes existing dictionary-based code compression techniques, and describes a cost-benefit analysis framework to motivate the need and usefulness of using bit-masks for code compression. Chapter 3 presents our overall code compression algorithm including a detailed description of the mask selection procedure, dictionary selection technique, and decompression framework. Chapter 4 presents the experimental results. Finally, Chapter 5 contains a summary of the thesis and a discussion of future research directions.

CHAPTER 2
DICTIONARY-BASED CODE COMPRESSION

This chapter describes existing dictionary-based approaches and analyzes their limitations. First, a standard dictionary-based approach is discussed. Next, we describe recently proposed techniques that improves the standard approach by considering mismatches (hamming distance). Finally, we present a detailed cost-benefit analysis of the recent approaches in terms of how much repeating patterns they can generate from the mismatches. This analysis forms the basis of my thesis work to maximize the repeating patterns using bit-masks.

2.1 Dictionary-based Approach

Dictionary-based code compression techniques provide compression efficiency as well as fast decompression mechanism. The basic idea is to take the advantage of commonly occurring instruction sequences by using a dictionary. The repeating occurrences are replaced by a codeword that points to the index of the dictionary that contains the pattern. The compressed program consists of both codewords and uncompressed instructions. Figure 2–1 shows an example of dictionary based code compression using a simple program binary. The binary consists of ten 8-bit patterns i.e., total 80 bits. The dictionary has two 8-bit entries. The compressed program requires 62 bits and the dictionary requires 16 bits. In this case, the compression ratio is 97.5% (using Equation (1.1)). This example shows a variable length encoding. As a result, there are many factors that also need to be included in the computation of the compression ratio including the size of the Line Address Table (LAT) [13] as well as byte alignment for branch targets.

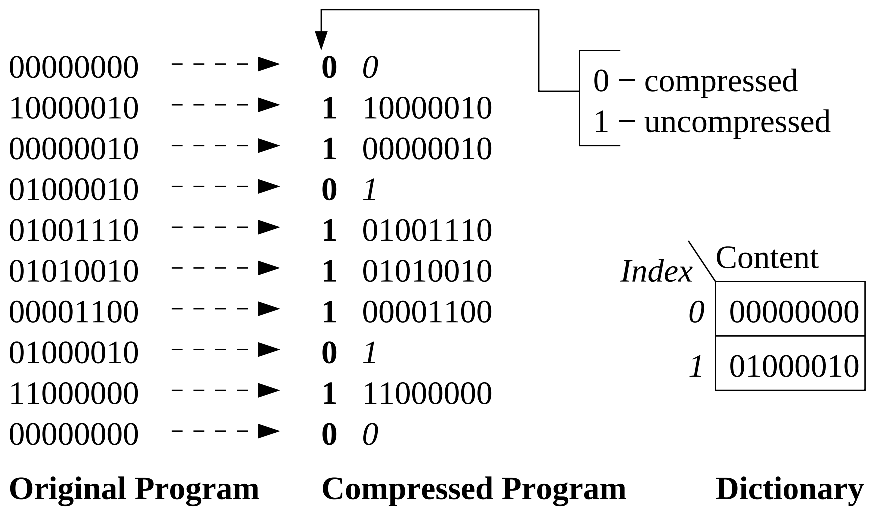

Figure 2–1: Dictionary-based Code Compression: An Example

2.2 Improved Dictionary-based Approach

Recently proposed techniques [10, 11] improve the dictionary based compression technique by considering mismatches. The basic idea is to determine the instruction sequences that are different in few bit positions (hamming distance) and store that information in the compressed program and update the dictionary (if necessary). The compression ratio will depend on how many bit changes are considered during compression. Figure 2–2 shows the encoding format used by these techniques for 32-bit program code.

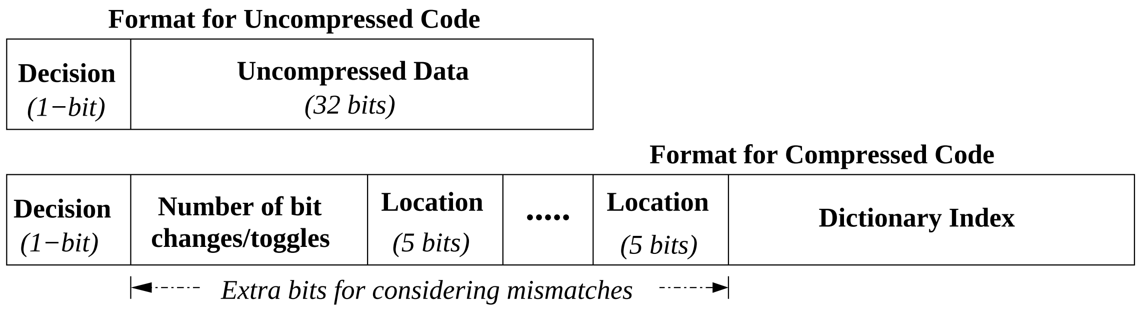
Format for Uncompressed Code
Decision (1−bit)
Uncompressed Data (32 bits)
Format for Compressed Code
Decision (1−bit)
Number of bit changes/toggles
Location (5 bits)
.....
Location (5 bits)
Dictionary Index
Extra bits for considering mismatches

Figure 2–2: Encoding Scheme for Incorporating Mismatches

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 2.3 describes this topic in detail. Prakash et al [10] considered only one-bit change for 16-bit patterns (vectors). Ros et al. [11] considered a general scheme of up to 7 bit changes for 32-bit patterns and concluded that a 3-bit change provides best compression ratio.

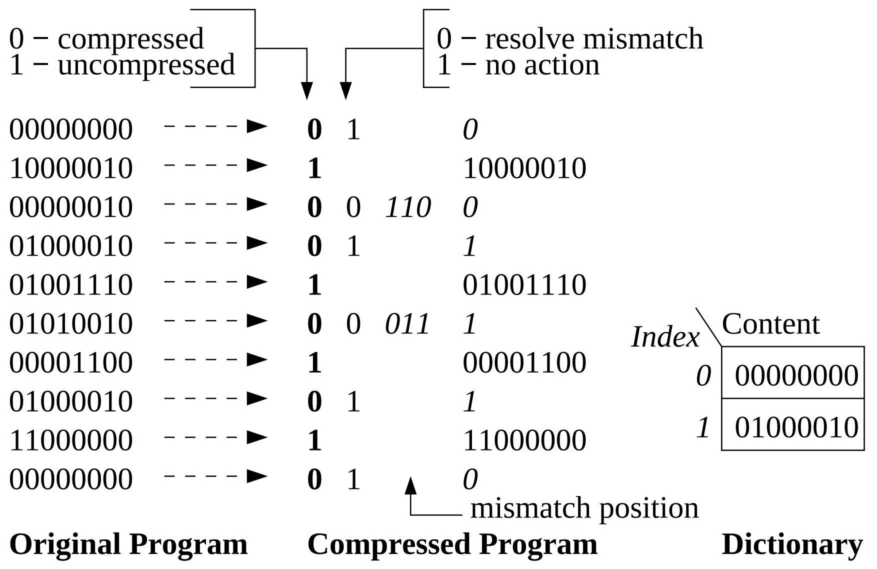

Figure 2–3: Improved Dictionary-Based Code Compression

Figure 2–3 shows the improved dictionary based scheme using the same example (shown in Figure 2–1). This example considers only 1-bit 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 the sixth bit position (from left). The compression ratio for this example is 95%.

2.3 Cost-Benefit Analysis for Considering Mismatches

We can create more repeating patterns if we consider changes in more bit positions. For example, if we consider 2-bit changes in Figure 2–3, all mismatched patterns can be compressed. However, increasing more repeating patterns by considering multiple mismatches does not always improve the compression ratio. This is due to the fact that the compressed program has to store multiple bit positions. If we consider 2-bit changes for the example in Figure 2–3, the compression ratio will be worse (102.5%).

I have done a detailed study on how to match more bit positions without adding significant information in the compressed code. I have considered 32-bit code vectors for compression. The hamming distance between any two 32-bit vectors is between 0 and 32. The compression adds extra 5 bits to remember each bit position in a 32-bit pattern. Moreover, extra bits are necessary to decide how many bit changes are there in the compressed code. For example, if the code allows up to 32 bit changes, it requires extra 5 bits to indicate the number of changes. As a result, this process requires a total of 165 extra bits ($32 \times 5 + 5$) when all 32 bits are different. Clearly, it is not profitable to compress a 32-bit vector using 165 extra bits along with a codeword (dictionary index information) and other details.

I have explored the use of bit-masks for creating repeating patterns. For example, a 32-bit mask pattern is sufficient to match any two 32-bit vectors. Of course, it is not profitable to store extra 32 bits to compress a 32-bit vector but definitely better than 165 extra bits. I considered mask patterns of different sizes (2-bit to 32-bit). When a mask pattern is smaller than 32 bits, we need to store information related to starting bit position where the mask needs to be applied. For example, if we use a 8-bit mask pattern, and want to consider all 32-bit mismatches, it requires four 8-bit masks, and extra two bits (to identify one of the 4 bytes) for each mask pattern to indicate where it will be applied. In this particular case, extra 42 bits are required

In general the dictionary contains 1024 or more entries. As a result, a code pattern will have fewer than 32 bit changes. If a code pattern is different from a dictionary entry in 8 bit positions, it requires only one 8-bit mask and its position i.e., it requires 13 (8+5) extra bits. This can be improved further if we consider bit changes only in byte boundaries. This leads to a trade-off. It requires fewer bits (8+2) but may miss few mismatches that spread across two bytes.

Table 2–1: Cost of Various Matching Schemes

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

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

Table 2–1 shows the summary of the study. Each row represents the number of changes allowed. Each column represents the size of the mask pattern. A one-bit mask is essentially same as remembering the bit position. Each entry in the table (with row r and column c) indicates how many extra bits are necessary to compress a 32-bit vector when the number of bit changes allowed is r and c is the size of the mask pattern. For example, we require 14 extra bits to allow 8-bit (row with value 8) changes using 4-bit (column with value 4) mask patterns. Clearly, the entries in the table with more than 15 extra bits are not profitable to consider during mask selection.

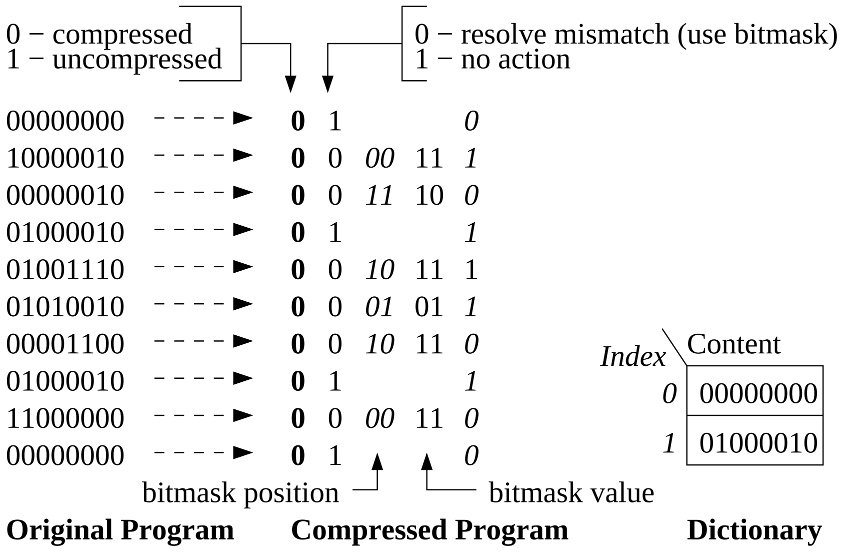

Figure 2–4: Code Compression Using Bit-Mask Approach

Section 3.1 describes our technique for selecting efficient mask patterns. Chapter 3 presents our code compression algorithm using the selected masks that significantly improves the compression ratio. Consider the same example shown in Figure 2–3. A 2-bit mask (only on quarter byte boundaries) is sufficient to create 100% matching patterns and thereby improves the compression ratio (87.5%) as shown in Figure 2–4. Experiments using real programs demonstrate that the compression ratio using our approach varies between 55-65%.

CHAPTER 3
CODE COMPRESSION USING BIT-MASKS

The motivation of the thesis is based on the analysis presented in Section 2.3. The proposed approach tries to incorporate maximum bit changes using mask patterns without adding significant cost (extra bits) such that the compression ratio is improved. Our compression technique also ensures that the decompression efficiency is improved or remains the same compared to the existing techniques.

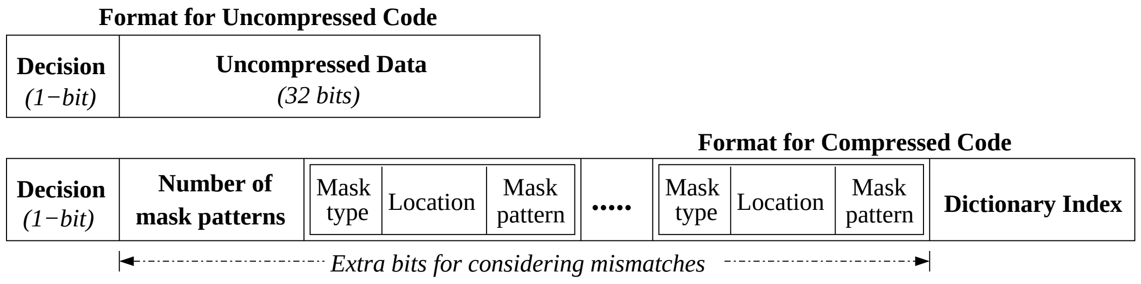

Figure 3–1: Encoding Format for Our Compression Technique

Figure 3–1 shows the generic encoding scheme used by the our compression technique. This scheme is similar to the 32-bit format shown in Figure 2–2 where individual bit changes are recorded. However, as described in Section 2.3, storing individual bit changes limits the number of matches. Our encoding format (Figure 3–1) can store information regarding multiple mask patterns. For each pattern, it stores the mask type, the location where mask needs to be applied, and the mask pattern. The generic encoding scheme can be optimized once the types and sizes of the bit-mask combinations are determined.

Code compression using bit-masks is a promising approach. However, it introduces two major challenges. First, how to select a set of mask-patterns that will maximize the matching sequences while minimize the cost of using bit-masks? Second, how to perform efficient dictionary selection using the selected bit-masks. We have developed efficient mask selection and dictionary selection technique to improve compression ratio with introducing any decompression penalty. This chapter is organized as follows. Section 3.1 describes our mask selection procedure. Section 3.2 outlines our dictionary selection technique. Section 3.3 presents our code compression algorithm using the mask selection and the dictionary selection procedures. Finally, Section 3.4 describes the decompression framework to support our bit-mask based code compression technique.

3.1 Mask Selection

A major challenge in bit-mask based code compression technique is how to choose a set of profitable mask patterns. Table 3–1 shows the mask-patterns that would create more matching patterns at an acceptable cost (based on the cost-benefit analysis in Section 2.3). A "fixed" bit-mask pattern implies that the pattern can be applied (starting position) only on fixed locations. For example, an 8-bit fixed mask (referred as *8f*) is applicable on 4 fixed locations (byte boundaries) in a 32-bit vector. A "sliding" bit-mask pattern can be applied anywhere. For example, an 8-bit sliding mask (referred as *8s*) can be applied in all locations on a 32-bit vector. There is no difference between fixed and sliding for a 1-bit mask. In this thesis, we will use 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 s can be applied on $(32 \div s)$ number of places. For example, a 8-bit fixed mask can be applied only on four places (byte boundaries) and 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 3–1: 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

Our goal in this section is to find a set of mask patterns that will deliver best code compression for a given application(s). This leads to answering two questions: i) which patterns are profitable and ii) how many patterns do we need? The answer to the second question is trivial: up to two mask patters are profitable. The reason is obvious based on cost consideration. A set of n mask patterns can generate up to $n \times 1 + n \times 2 + \cdots + n \times n$ $(n^2(n-1)/2)$ combinations. For example, if we choose two distinct mask patterns 2-bit fixed (*2f* and 4-bit sliding (*4s*), it can generate six combinations: (2f), (4f), (2f, 4f), (4f, 2f), (2f, 2f), (4f, 4f). Similarly, three distinct mask patterns can create up to 18 combinations that requires 5 bits to store the mask combination along with the cost of storing up to three masks and their positions in each compressed vector. Clearly, use of three or more mask patterns are not profitable since they require more than 10 extra bits. The answer to the first question in non-trivial. The remainder of this section tries to answer the first question.

It is evident that applying larger bit-masks will generate more matching patterns. However, doing so may not result in a superior overall compression. The reason for this is simple: a longer bit-mask pattern is associated with a larger cost. Similarly, using a sliding mask where a fixed one is sufficient is wasteful since fixed mask will require less number of bits (compared to its sliding counterpart) to store the position information. For example, applying a 4-bit fixed 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 fixed 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. Moreover if 4-bit sliding mask (cost of 9 bits) is used where 4-bit fixed (cost of 7 bits) is sufficient, two additional bits are wasted.

We also observed that the mask patterns that are factors of 32 (e.g., masks 1, 2, 4 and 8 from Table 3–1) produces 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, 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. This finding allowed us to reduce the 11 mask patterns in Table 3–1 down to 7 mask patterns shown in Table 3–2.

Table 3–2: Profitable Bit-Mask Patterns

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

We studied carefully the combinations of up to two bit-masks using various applications compiled on a wide variety of architectures. We analyzed the result of compression ratios on various mask combinations and observed that *8f* and *8s* are not helpful. We also observed that *4s* does not perform better than *4f*. The final set of bit-mask patterns are shown in Table 3–3.

Table 3–3: Final Bit-Mask Sets

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

The goal of the mask selection algorithm is to find two mask patterns from the Table 3–3 that are most profitable for the given application(s). Algorithm 1 shows our mask selection procedure. It accepts the application program(s) as input and produces a pair of mask-patterns that are most profitable for compressing the application(s). We observed that use of two masks always do better compression compared to using only one mask. There are sixteen different ways of choosing two patterns from four bit-masks. Therefore, Algorithm 1 needs to compress the application(s) sixteen times to obtain the profitable mask combination.

```
**Algorithm 1**: *Mask Selection*
**Input**: Application(s) *appl*.
**Outputs**: Two mask patterns $< mask_1, mask_2 >$.
Begin
   $mask_1$ = 1s; $mask_2$ = 1s
   compressionRatio = 100
   for each mask pattern $m_i$ in (1s, 2s, 2f, 4f)
       for each mask pattern $m_j$ in (1s, 2s, 2f, 4f)
           ratio = Compress *appl* using $< m_i, m_j >$.
           if (ratio is less than compressionRatio)
              compressionRatio = ratio
              $mask_1 = m_i$; $mask_2 = m_j$
           endif
       endfor
   endfor
   return $< mask_1, mask_2 >$
End
```

3.2 Dictionary Selection

Dictionary selection is a major challenge in code compression. Optimal dictionary selection is an NP hard problem [?]. Therefore, the techniques in the literature tries to use various heuristics based on application characteristics. Figure 3–2 classifies the dictionary selections methods used in code compression for embedded systems. Dictionary can be generated either dynamically during compression or statically prior to compression. While dynamic approaches such as LZW[8] accelerates the compression time, seldom it matches the compression ratio of the 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 vectors' frequency or spanning [11].

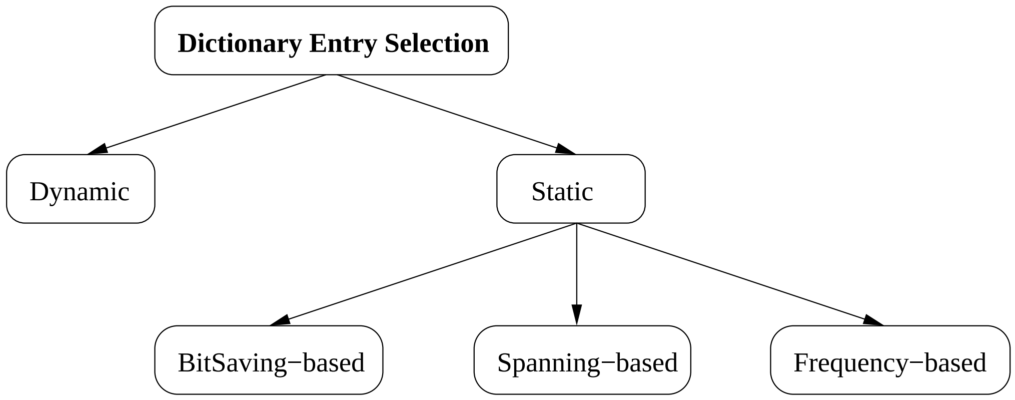

Figure 3–2: Dictionary Selection Methods

In the traditional dictionary-based compression approach, the dictionary entry selection process is simplified since it is evidently clear that the frequency-based selection will give good compression ratio. However, when compressing with bit-masks, the problem is complex and the frequency-based selection will not always yield the best compression ratio. Figure 3–3 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). Ros and Sutton [11] made similar observations in their hamming distance based code compression framework using frequency-based as well as spanning-based methods, and reported that a mixture of two achieves better results than considering one independently.

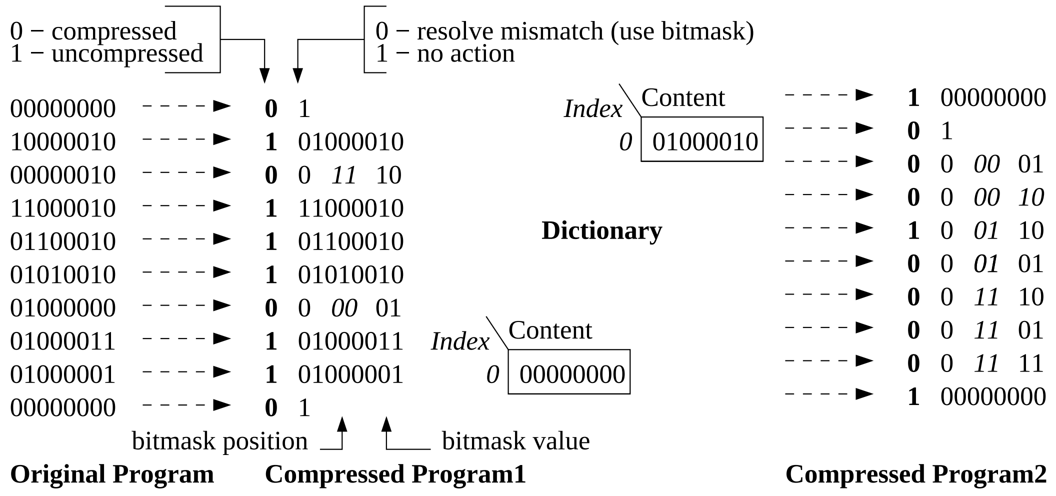

Figure 3–3: Compression with Different Dictionary Selection Methods

We have observed that neither frequency-based nor spanning-based methods can not efficiently exploit the power of bit-mask based compression to deliver good compression ratio. We developed a novel dictionary selection technique that considers bit savings as a metric to select a dictionary entry. Algorithm 2 shows out bit-savings 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 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. 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 cheapest mask (maximum savings). Also note that, this algorithm considers only the masks returned by the mask selection algorithm (Algorithm 1) and a *threshold* value to enable deletion of nodes from the graph during dictionary entry selection. 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 (mask savings) connected to that node along with the node savings (based on frequency value). Next, the node with maximum overall savings is selected as an entry for the dictionary. The selected node is deleted from the graph. we also need to delete the nodes that are connected to the selected node. However, we have observed that it is not always profitable to delete all the connected nodes, instead it is a good idea to delete nodes that has overall savings less than a particular threshold. Typically a node with frequency value less than 10 serves as a good threshold. This varies from application to application but based on our experiences a threshold value between 3 and 10 seems to work best. The algorithm terminates when either dictionary is full or the graph becomes empty.

```
Algorithm 2: Bit-Savings based Dictionary Selection
Inputs: 1. Application(s) consisting of 32-bit instruction vectors
        2. Mask patterns generated by Algorithm 1
        3. A threshold value to enable 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
```

Figure 3–4 illustrates the 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 to 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).

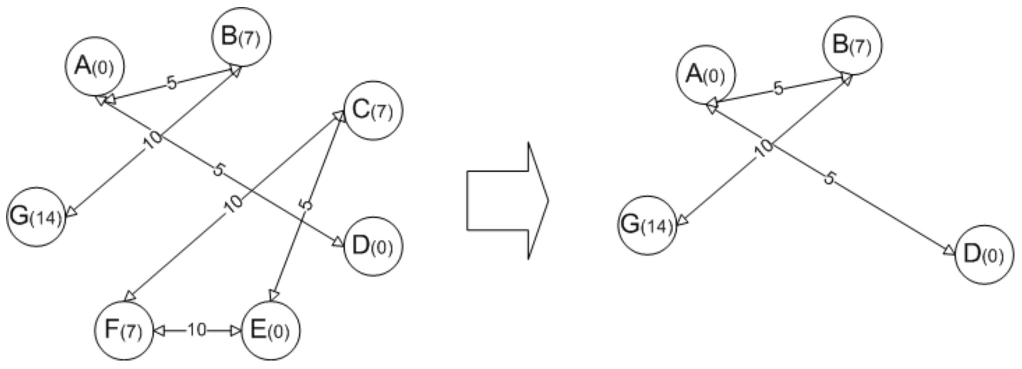

Figure 3–4: Bit Saving Dictionary Selection Method

The algorithm repeats by recalculating the savings of the vertex in the new graph and terminates when the dictionary becomes full. Our experimental Our bit-savings based dictionary selection method provides the best compression ratio and outperformed the frequency and spanning based approach.

3.3 Compression Algorithm

Algorithm 3 shows the basic steps of our compression technique using bit-masks. The algorithm accepts the original code consisting of 32-bit vectors. The first step is to determine which mask set to use for compression using Algorithm 1.

```
Algorithm 3: Code Compression using Bit-Masks
Input: Original code (binary) divided into 32-bit vectors.
Outputs: Compressed code and dictionary.
Begin
    Step 1: Select the profitable mask patterns.
    Step 2: Select the optimized dictionary.
    Step 3: Compress each 32-bit vector using cost constraints.
    Step 4: Adjust and handle the branch targets.
    return Compressed code and the dictionary
End
```

The second step is to select the optimized dictionary using Algorithm 2. In case the frequency-based dictionary selection is used, it is useful to consider larger dictionary sizes when the current dictionary size cannot accommodate all the vectors with frequency value above certain threshold. (e.g., above 5 is profitable). However, there are certain disadvantages of increasing the dictionary size. The cost of using larger dictionary is more since the dictionary index becomes bigger. The cost increase is balanced only if most of the dictionary is full with high frequency vectors. Most importantly, a bigger dictionary increases access time and thereby reduces decompression efficiency.

The third and final step of the algorithm converts each 32-bit vector into compressed code (when possible) using the format shown in Figure 3–1. The compressed code along with any uncompressed ones are composed serially to generate the final compressed program code.

3.4 Decompression Mechanism

Like other variable length encoding schemes, our scheme has one disadvantage: it produces variable length compressed code for each vector. As a result, finding a branch target during decompression becomes difficult.

To avoid this problem, several approaches have been proposed. Wolfe and Chanin [13] proposed LAT (Line Address Table), which includes the mapping between branch target addresses in the original code and compressed code. Lefurgy [4] proposed backpatching the compressed addresses to redirect the branch targets to the new locations. Extra bits are added at the end of the code that precedes the branch target to assign the branch targets on a byte boundary.

The decompression scheme for our compression technique is similar to the existing dictionary based code compression techniques. Figure 3–5 shows one possible implementation of the decompression unit in hardware when up to two ask patterns are considered. The compressed code is accessed serially and checked if it is compressed or not. If compressed, the corresponding dictionary entry is retrieved and XOR-ed with a 32-bit mask pattern. The 32-bit mask pattern is generated based on the mask patterns and their location information in the compressed code.

The decompression unit also needs to access the Line Address Table (LAT) when the address requested by the processor is not the next address (to find the branch targets).

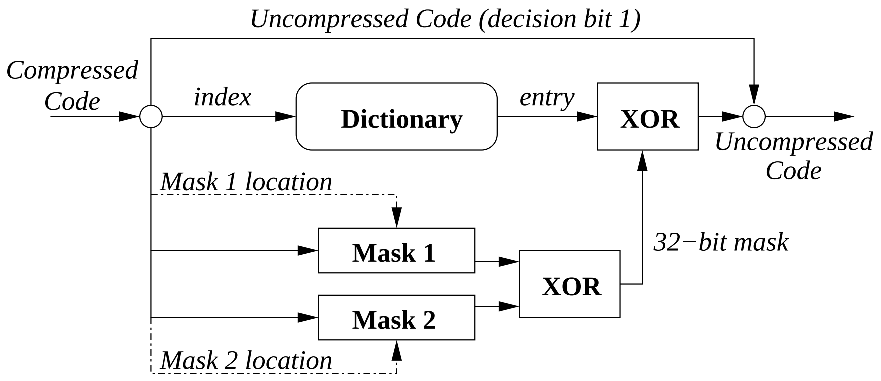

Figure 3–5: Implementation of a Decompression Unit

Since the compression technique uses a variable length encoding, the decompression needs to be done serially. However, this creates problem for VLIW architectures such as TI TMS320C6x which uses a fetch packet of up to eight 32-bit vectors. To resolve this, we can employ a stream encoding similar to the one proposed by Ros et el. [11] where a fetch packet needs to be divided into 32-bit streams during compression, and decompression can be applied to the individual streams instead of the whole program. This will require separate decompression unit as well as individual tables for each stream.

CHAPTER 4
EXPERIMENTS

4.1 Experimental Setup

We performed code compression experiments by varying both application domains and target architectures. The following section present experimental results using twelve embedded applications for three target architectures. The benchmarks are collected from TI and MediaBench & MiBench benchmark suites: *adpcm_en*, *adpcm_de*, *block_ms*, *cjpeg*, *djpeg*, *gsm_to*, *gsm_un*, *mpeg2enc*, *mpeg2dec*, *modem*, *pegwit*, and *verbiti*.

The benchmarks were compiled for three target architectures:TI TMS320C6x, MIPS, and SPARC. We used TI *Code Composer Studio* to generate binary for TI TMS320C6x and used *gcc* to generate binary for MIPS and SPARC. The compression ratio was computed using the Equation (1.1). The computation of compressed program size includes the size of the compressed code as well as that of the dictionary (and LAT where applicable).

4.2 Results

Figure 3–1 shows the general encoding format using bit-masks. We explored various customized version of the encoding format to figure out which encoding format works better across all target architectures. Clearly, 32-bit mask pattern is not profitable. The 16-bit mask is also not useful unless there are too any mismatches which a 4-bit or 8-bit (or combined 12 bit) mask cannot capture. In Section 3.1, we studied the trade off between various mack combinations and identified the set of four or five mask patterns that we would like to consider.

We explored all possible encoding scenarios using 4-bit and 8-bit masks and observed that three customized encoding formats shown in Figure 4–1 works very well across applications and target architectures. The first encoding (Encoding 1) uses a 8-bit mask, the second encoding (Encoding 2) uses up to two 4-bit masks, and the third encoding (Encoding 3) uses up to two masks where first mask can be 4-bit or 8-bit whereas the second mask is always 4-bit.

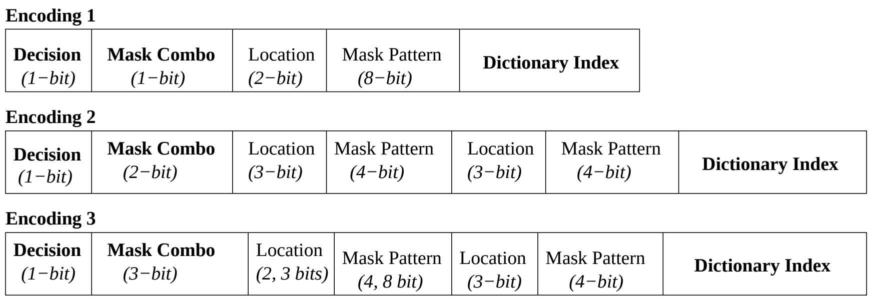

Figure 4–1: Three Customized Encoding Formats

Figure 4–2 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 - generate compression ratio of 55-65%. Our experience with other benchmarks also suggests the same trend. The remainder of the section uses the second encoding format (Encoding 2) for all the results presented.

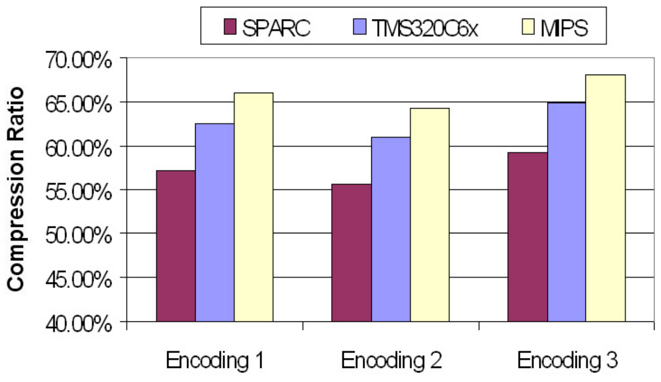

Figure 4–2: Compression Ratio for adpcm_en Benchmark

The code compression with bit-masks performs well for different dictionary sizes. Figure 4–3 shows the efficiency of the proposed compression technique for all the nine benchmarks compiled for SPARC processor using dictionary sizes of 4K and 8K. As expected, we can observe three scenarios. The small benchmarks such as *adpcm_en* and *adpcm_de* performs better with small dictionary since a majority of the repeating patterns fits in the 4K dictionary (in fact it fits in the 2K dictionary as shown in Figure 4–2). 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* does not benefit much from the bigger dictionary size. On an average, our technique generate 59% compression ratio.

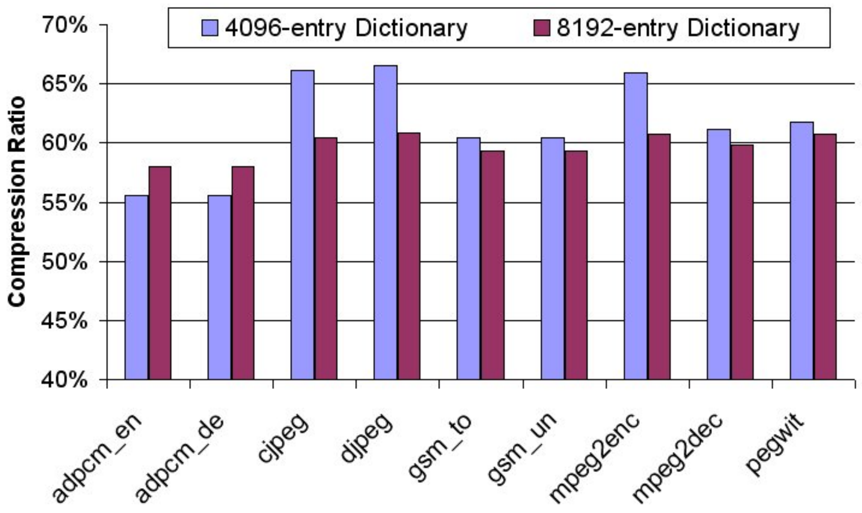

Figure 4–3: Compression Ratio for Different Benchmarks

Figure 4–4 compares compression ratios achieved by the various dictionary selection methods described in Section 3.2. We restricted the dictionary size to increase the distinction amongst four methods: frequency, spanning, and our bit-savings based approach. As shown in the figure, considering spanning alone does poorly compared to other dictionary selection methods. Our bit-savings based approach outperforms all the existing methods. This reconfirms the importance and the challenge of identifying dictionary entries to optimize the compression ratio.

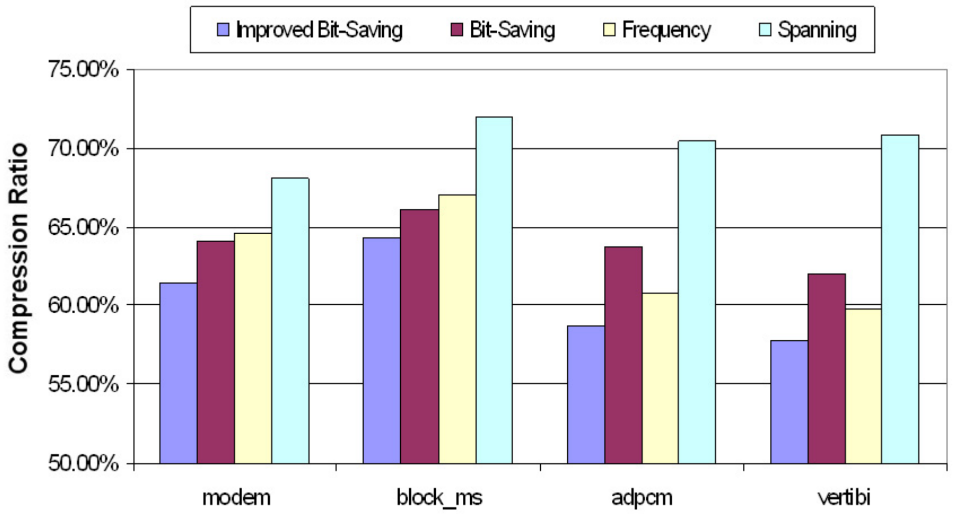

Figure 4–4: Dictionary Selection Method Comparison

Table 4–1 compares the proposed approach with existing code compression techniques. As mentioned earlier, the proposed technique improves the code compression efficiency by 15% compared to the existing dictionary based techniques [10, 11] without introducing any additional decompression penalty.

Table 4–1: Cost of Various Matching Schemes

Compression Method	Target Architecture	Compression Ratio	Hardware Overhead
Wolfe and Chanin [1]	MIPS	73%	Under 1 mm^2
V2F: Xie et al.[12]	TMS320C6x	70-82%	6-48K table
MCSSC: Lin et al.[3]	TMS320C6x	75%	30k table
Prakash et al.[5]	TMS320C6x	76-80%	Not available
Ros and Sutton [6]	Itanium TMS320C6x	72-80%	8-16K
Our Approach	MIPS, SPARC TMS320C6x	55-65%	8-16K

Smaller compression ratio implies better compression technique (see Equation 1).

CHAPTER 5
CONCLUSIONS AND FUTURE DIRECTIONS

5.1 Conclusions

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

In the thesis, I presented an efficient code compression scheme using bit-masks that can significantly increase the number of matching patterns. This approach can handle multiple bit mismatches without incurring any compression or decompression penalty. Moreover, I examined and presented the approaches for the two challenges that were introduced by the proposed technique: the choice of the mask combination and the dictionary entry selection.

I applied the technique using applications from various domains and compiled them for different architectures to demonstrate the usefulness of the approach. The experimental results show that the proposed technique reduces the original program size up to 45%. This technique outperforms all the existing dictionary based techniques by an average of 15%, giving compression ratios of 55%-65%. It also enables parallel decompression that is suitable for VLIW processors.

5.2 Future Work Directions

Code compression for embedded systems is a major problem. This thesis investigated the use of bit-masks in improving the compression efficiency. The work presented in this thesis can be extended in the following directions:

- ❒ Currently, our technique generates up to 90-95% matching sequences without losing any compression ratio. Further studies are necessary to create remaining 5-10% mismatches. One possible direction is to introduce the compiler optimizations that use hamming distance as the cost measure for generating code.
- ❒ Code compression delivers savings for area, power, and performance. This thesis primarily concentrated on area (code size) improvement without sacrificing performance. Further studies can be done to estimate power savings and area reduction by the proposed technique.
- ❒ We have investigated the use of bit-masks for architectures that uses fixed-length instruction-set. Future research needs to extend our compression technique on variable length instruction architectures.
- ❒ This thesis considered use of bit-masks for code compression. Our compression technique can be also be applied in other domains where memory is also a bottleneck such as test compression during manufacturing testing of complex and heterogeneous embedded systems.

REFERENCES

[1] Dipankar Das, Rajeev Kumar, and P.P. Chakrabarti. Dictionary based code compression for variable length instruction encodings. In *Proceedings of VLSI Design (VLSID)*, pages 545–550, 2005.

[2] N. Ishiura and M. Yamaguchi. Instruction code compression for application specific vliw processors based of automatic field partitioning. In *Proceedings of Synthesis and System Integration of Mixed Technologies (SASIMI)*, pages 105–109, 1997.

[3] S. Larin and T. Conte. Compiler-driven cached code compression schemes for embedded ilp processors. In *Proceedings of International Symposium on Microarchitecture (MICRO)*, pages 82–91, 1999.

[4] C. Lefurgy, P. Bird, I. Chen, and T. Mudge. Improving code density using compression techniques. In *Proceedings of International Symposium on Microarchitecture (MICRO)*, pages 194–203, 1997.

[5] H. Lekatsas, Jorg Henkel, and Venkata Jakkula. Design of an one-cycle decompression hardware for performance increase in embedded systems. In *Proceedings of Design Automation Conference (DAC)*, pages 34–39, 2002.

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

[7] S. Liao, S. Devadas, and K. Keutzer. Code density optimization for embedded dsp processors using data compression techniques. In *Proceedings of Advanced Research in VLSI*, pages 393–399, 1995.

[8] C. Lin, Y. Xie, and W. Wolf. Lzw-based code compression for vliw embedded systems. In *Proceedings of Design Automation and Test in Europe (DATE)*, pages 76–81, 2004.

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

[10] J. Prakash, C. Sandeep, P. Shankar, and Y. Srikant. A simple and fast scheme for code compression for vliw processors. In *Proceedings of Data Compression Conference (DCC)*, page 444, 2003.

[11] M. Ros and P. Sutton. A hamming distance based vliw/epic code compression technique. In *Proceedings of Compilers, Architectures, Synthesis for Embedded Systems (CASES)*, pages 132–139, 2004.

[12] M. Ros and P. Sutton. A post-compilation register reassignment technique for improving hamming distance code compression. In *Proceedings of Compilers, Architectures, Synthesis for Embedded Systems (CASES)*, pages 97–104, 2005.

[13] A. Wolfe and A. Chanin. Executing compressed programs on an embeded risc architecture. In *Proceedings of International Symposium on Microarchitecture (MICRO)*, pages 81–91, 1992.

[14] Y. Xie, W. Wolf, and H. Lekatsas. Code compression for vliw processors using variable-to-fixed coding. In *Proceedings of International Symposium on System Synthesis (ISSS)*, pages 138–143, 2002.

BIOGRAPHICAL SKETCH

Seok-Won Seong received his Bachelor of Science in computer and information sciences in 2003, Master of Science in business management in 2004, and Master of Science in computer engineering in 2006, from the University of Florida. He is working as a graduate researcher on code compression techniques for embedded systems with Dr. Prabhat Mishra and as a research assistant on the data integration project with Dr. Joachim Hammer.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Prabhat Mishra, Chair
Assistant Professor of Computer and Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Sartaj Sahni
Distinguished Professor and Chair of Computer and Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Sanjay Ranka
Professor of Computer and Information Science and Engineering

This thesis was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Master of Science.

May 2006

Pramod P. Khargonekar
Dean, College of Engineering

Winfred M. Phillips
Dean, Graduate School