University of Cincinnati

Date: 8/20/2012

I, Lijo John, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Engineering.

It is entitled:
Implementation of bitmask based code compression algorithm on MSP430 microcontroller

Student's name: Lijo John

This work and its defense approved by:

Committee chair: Carla Purdy, PhD
Committee member: Wen Ben Jone, PhD
Committee member: George Purdy, PhD
Implementation of bitmask based code compression algorithm on MSP430 microcontroller

A Thesis submitted to the Graduate School

Of

The University of Cincinnati

In partial fulfillment of the requirements for the degree of

Master of Science

in the

Department of Electrical and Computer Engineering

of the

College of Engineering and Applied Sciences

August 2012

by

Lijo John

Committee Chair: Dr. Carla Purdy
ABSTRACT

Embedded systems today have become ubiquitous and they play a very significant role in all aspects of our lives. Due to their compact size, low cost and simple design aspects, embedded systems have become very popular and affect almost all aspects of our lives. They are found everywhere from kitchen appliances to space craft. They span all aspects of modern life from various applications such as biomedical instrumentation, communication systems, peripheral controllers of a computer and industrial instrumentation to scientific applications.

Since larger memory is always associated with higher cost, increased chip area and increased power consumption, the primary goal has always been to keep the memory at a minimum. One way to use the available memory more efficiently is to compress the program code and store it in a compressed form. The compressed code is decompressed on the fly as and when needed by the processor. In this thesis, we have discussed one code compression technique known as bitmask based code compression algorithm in detail and have implemented it on Texas Instrument’s MSP430 microcontroller. Using this technique, we were able to achieve a compression ratio of about 62-76%. We also compared our results to other compression technologies in the literature.
ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Carla Purdy for her support and advice over the past 1 year without which this thesis would not have been possible. I am also indebted to Bhavik Shah for his valuable suggestions which gave me better ideas for my thesis and implementation.

I am grateful to my friends Chaitanya Mantena, Manas Minglani, Shaival Jhaveri, Krishnakumar Iyer, Rajbir Chadha, Jasman Kaur and Harmandeep Kaur Bhutta for making my graduate life a great experience.

Most importantly, I dedicate this thesis to my brother Linu Alex who had been the greatest support for me throughout my graduate life.
# TABLE OF CONTENTS

1. **INTRODUCTION** .................................................................................................................. 1
   
   1.1 Motivation .......................................................................................................................... 1
   
   1.2 Thesis goals ...................................................................................................................... 2
   
   1.3 Outline ............................................................................................................................ 2
2. **BACKGROUND** .................................................................................................................. 3
   
   2.1 Code compression ............................................................................................................ 3
   
   2.2 Significance of code compression .................................................................................... 3
   
   2.3 Previous work .................................................................................................................. 5
   
   2.4 Thesis approach .............................................................................................................. 16
3. **METHODOLOGY** ............................................................................................................... 17
   
   3.1 Flow of the work .............................................................................................................. 17
   
   3.2 Bitmask based code compression ...................................................................................... 17
   
   3.2.1 Standard dictionary-based approach [1] .................................................................... 17
   
   3.2.2 Mismatch-based dictionary approach ........................................................................ 19
   
   3.2.3 Bitmask-based dictionary approach [1] .................................................................... 21
   
   3.3 Compression algorithm .................................................................................................. 23
   
   3.4 Decompression core ...................................................................................................... 25
   
   3.5 Target processor ............................................................................................................ 26
   
   3.6 Code outline ................................................................................................................... 27
4. **EXPERIMENTS** ................................................................................................................ 29
   
   4.1 Experimental setup ....................................................................................................... 29
LIST OF FIGURES

Figure 2.1 Code compression approach [1]........................................................................3
Figure 2.2 Decompression Schemes [1].............................................................................4
Figure 2.3 Memory organization [3]..................................................................................7
Figure 2.4 2-bit Tunstall coding tree and codebook [6].......................................................10
Figure 2.5 Dictionary code example [2].............................................................................14
Figure 3.1 Dictionary based code compression....................................................................18
Figure 3.2 Mismatch-based dictionary code compression....................................................20
Figure 3.3 Encoding format for incorporating mismatches [1].............................................20
Figure 3.4 Bit-Mask-based dictionary code compression [1]................................................22
Figure 3.5 Encoding format for incorporating bitmasks [1]...................................................23
Figure 3.6 Possible implementation of a decompression core.............................................25
Figure 3.7 Code outline.......................................................................................................28
Figure 4.1 Encoding format used in our implementation....................................................29
Figure 4.2 Compression ratio for ADC10_ex1.....................................................................32
Figure 4.3 Compression ratio for ADC10_ex2.....................................................................33
Figure 4.4 Compression ratio for ADC12_ex1.....................................................................34
Figure 4.5 Compression ratio for ADC12_ex2.....................................................................34
Figure 4.6 Compression ratio for comp_ex2.......................................................................35
Figure 4.7 Compression ratio for comp_ex5.......................................................................36
Figure 4.8 Compression ratio for crc_ex1..........................................................................37
Figure 4.9 Compression ratio for crc_ex1_DCOSetup........................................................37
Figure 4.10 Compression ratio for dac12_ex1.....................................................................38
Figure 4.11 Compression ratio for dma_ex1_repeatedBlock..............................................39
Figure 4.12 Compression ratio for FLASH_example1..........................................................40
Figure 4.13 Compression ratio for FRAM_ex1_write..........................................................40
Figure 4.14 Compression ratio for ramcontroller_example1 .................................................................41
Figure 4.15 Compression ratio for TIMER_continousModeWithCCR0Interrupt .............................42
Figure 4.16 Compression ratio for UART_TX ..................................................................................42
Figure 4.17 Comparison between the two implementations ............................................................45
LIST OF TABLES

Table 2.1 Summary of code compression algorithms ............................................................. 5
Table 2.2 LZW coding example [8] .......................................................................................11
Table 4.1 Sample code programs .........................................................................................30
Table 4.2 Summary of results .................................................................................................43
Table 4.3 Mean, max-min, standard deviation and standard error .......................................44
1. INTRODUCTION

1.1 Motivation

Embedded systems span all aspects of our modern life today. Embedded systems play a pivotal role in real life applications such as telecommunication systems, consumer electronics, transportation systems, medical equipment, security systems and household appliances [1]. With embedded systems gaining an undisputed importance in a world which is miniaturizing day by day, the goal has always been to keep memory resources minimal. Due to considerations such as power consumption, space and weight, the memory is typically limited in embedded systems. Large memory is associated with increased chip area and higher cost, thus imposing various constraints on the size of the application programs. For an efficient utilization of the limited memory in embedded systems, researchers have been trying to fit more program code into the available space by means of code compression. As the cost of an integrated chip is directly related to the die size and the memory size is also directly proportional to the die size, a lot of research has been going on for the last twenty five years to fit the program into the smallest memory possible. Code compression techniques are gaining popularity due to their capability to reduce the memory requirements of embedded systems [1].

bitmask-based code compression technique which is an improvement over the dictionary-based code compression technique.

1.2 Thesis goals
This thesis aims to study the bitmask-based code compression technique proposed by Seong and Mishra [1]. In this thesis, we implement the bitmask-based code compression algorithm on Texas Instrument’s MSP430 microcontroller and we analyze the compression ratio achieved by using this technique. This thesis also consists of a tutorial on a step-by-step implementation process of bitmask based code compression algorithm on MSP430 microcontroller.

1.3 Outline
The thesis is organized as follows:

Chapter 2 provides the background on various code compression techniques. It also discusses the code compression techniques in brief and gives a comparison of the compression ratios achieved by various code compression techniques.

Chapter 3 discusses the bitmask-based code compression algorithm in detail and provides the various steps carried out to implement the bitmask-based code compression on Texas Instrument’s MSP430.

Chapter 4 provides the experimental setup and the results of the implementation.

Chapter 5 presents the conclusions and future work that can be done.

Appendix A consists of a tutorial on a step-by-step implementation process of bitmask based code compression algorithm on MSP430 microcontroller.
2. BACKGROUND

2.1 Code compression

Memory is one of the key driving factors in an embedded system design. As larger memory is associated with higher cost, increased power consumption and increased chip area, the aim is to use the available memory as efficiently as possible. Memory imposes various limitations on the size of the application programs. Thus, various code compression techniques have gained popularity by addressing this issue by reducing the program size. The idea is to compress the application program offline and store it in the memory in a compressed form. When the processor requires the program, the code is decompressed and fed back to the processor in its original form.

Thus, various code compression techniques are being explored due to their capability to reduce the memory requirements of embedded systems [1]. The first code compression algorithm dates back to the year 1992 [1]. Since then, there has been an intensive research in this field for the best compression algorithm.

2.2 Significance of code compression

Figure 2.1 Code compression approach [1]
Our aim in code compression is to reduce the size of application programs via various compression techniques. We store the program code in compressed format in the memory and we decompress it at run-time as and when the application program is required by the processor. As shown in Figure 2.1 [1], the application program is compressed offline by the code compression algorithm and then stored into the memory. The compressed code is then fetched and decompressed at run-time by the processor before the execution phase.

As shown in Figure 2.2 [1], the decompression scheme can be implemented in two different ways in embedded systems with caches. The decompression hardware can be placed in between the main memory and the instruction cache as a result of which the main memory will contain the compressed program and the instruction cache will contain the original program. This is known as the pre-cache architecture as shown in Figure 2.2(a) [1]. Alternatively, the decompression hardware can be placed in between the instruction cache and the processor as a result of which both the main memory and the instruction cache will contain the compressed program and the original program will be fed to the processor upon request after decompression. This is known as the post-cache architecture as shown in Figure 2.2(b) [1]. In this architecture, we can increase the cache hits and achieve a potential performance gain.
Earlier approaches for code compression algorithms have been borrowed from the field of text compression and fall into two categories: statistical and dictionary. In statistical compression methods, the frequency of the single characters determines the length of the codeword used to replace the character. Frequent characters are replaced by shorter codewords in order to minimize the overall code size. One such statistical compression method is Huffman encoding [2]. In dictionary compression, frequently occurring phrases are replaced by codewords which are used to index into a dictionary which in turn contains the original phrase. Compression is achieved by means of short codewords which replace the frequent long phrases [2].

Several criteria are used to select the best compression algorithm. Two very important factors are the decompression efficiency and compression ratio. Decompression efficiency is defined as the amount of effort required to expand the codeword back into its original form. In order to compute the compression efficiency, compression ratio (C.R) is accepted as a primary metric [1]. It is defined as:

\[
Compression \ Ratio \ (C.R) = \frac{Compressed \ Program \ Size}{Original \ Program \ Size} \quad (2.1) \ [1]
\]

Thus, the smaller the compression ratio, the better the code compression technique is.

2.3 Previous work

There are many code compression algorithms in the literature. The following table summarizes major code compression algorithms which we will be discussing in brief in this chapter. The first code compression algorithm was proposed by Wolfe and Chanin [3], who used Huffman coding for compressing the program code. A pre-cache architecture was utilized by them to place the decompression engine in between the main memory and the instruction cache.
Table 2.1 Summary of code compression algorithms

<table>
<thead>
<tr>
<th>Year</th>
<th>Reference</th>
<th>Target</th>
<th>Method</th>
<th>Compression Ratio</th>
<th>Hardware Overhead</th>
<th>Decompression Bandwidth</th>
<th>Parallel Decompression</th>
</tr>
</thead>
<tbody>
<tr>
<td>1992</td>
<td>Wolfe and Chanin [3]</td>
<td>MIPS</td>
<td>Huffman</td>
<td>73%</td>
<td>Under 1 mm2</td>
<td>1 byte per iteration</td>
<td>No</td>
</tr>
<tr>
<td>1999</td>
<td>Lekatsas and Wolf [4]</td>
<td>MIPS</td>
<td>SAMC</td>
<td>57%</td>
<td>4K Table + control logic</td>
<td>Not available</td>
<td>No</td>
</tr>
<tr>
<td>2002</td>
<td>Xie [6]</td>
<td>TMS320 C6x, IA-64</td>
<td>V2F</td>
<td>70-82%</td>
<td>6-48k table + control logic</td>
<td>Up to 13 bits per iteration</td>
<td>Only iid (identical and independent distribution) could</td>
</tr>
<tr>
<td>2004</td>
<td>Xie and Lin [8]</td>
<td>TMS320 C6x</td>
<td>LZW</td>
<td>83-87%</td>
<td>&lt;0.005 mm2 control logic</td>
<td>Average 1.36-1.72 bytes</td>
<td>Yes</td>
</tr>
<tr>
<td>2008</td>
<td>Seong and Prabhat Mishra [1]</td>
<td>MIPS, SPARC, TMS320 C6x</td>
<td>Bitmask</td>
<td>55-65%</td>
<td>No of Mismatches + location of mismatches + dictionary overhead</td>
<td>32-64 bits</td>
<td>Yes</td>
</tr>
</tbody>
</table>

A Line Address Table (LAT) is used for mapping the original block addresses on to compressed block addresses and is stored along with the main memory. They also use another small cache known as the Cache Line Address Lookaside Buffer (CLB) to hold the most recently used addresses from the LAT. To handle the CLB, a least recently used (LRU) replacement algorithm is used. A block which cannot be compressed is left in its original form and is flagged in the corresponding LAT pointer. Figure 2.3 [3] shows the memory organization as below:
Whenever an instruction cache miss occurs, a cache refill penalty (which also includes the CLB refill penalty) is added. These penalties are dependent on the memory model as well as the compressed line size. Wolfe and Chanin implemented the code compression algorithm on a MIPS processor (a 32-bit RISC processor) which executes the R2000 instruction set. The processor has a 24-bit physical address space and a 32-bit data bus. A trace driven system simulator was used to compile the programs which are all versions of C and FORTRAN programming language from a DECstation 3100 workstation. A compression ratio (C.R) of 73% was achieved upon the implementation of this method on the MIPS processor.

**Lekatsas and Wolf** [4] proposed a statistical scheme known as Semi Adaptive Markov Compression (SAMC) based on arithmetic coding and a Markov model. Using arithmetic coding in combination with a precalculated Markov model, they developed the SAMC algorithm which is adapted to the application and the instruction set. Arithmetic coding is implemented in this algorithm using a lookup table generation technique developed by Howard and Vitter [5]. In a compression technique, we decompose an instruction set into a sequence of events and then we encode the events using as few bits as possible. The basic idea of arithmetic coding is to assign short codewords to more probable events and longer codewords to less probable events. This is similar to the technique used in Huffman coding [3]. Thus, code compression is possible whenever some events are more likely than others. Statistical coding techniques use estimates of the probabilities of the events in order to assign the codewords. A statistical coder works in
conjunction with a modeler to estimate the probability of each possible event at each point in the coding. The probability model has to provide a probability distribution for the events. These probabilities need not necessarily be accurate but the more accurate they are, the better the compression will be. If the probabilities are wildly inaccurate, the file will be expanded rather than compressed but the data can still be recovered. So in order to obtain maximum compression of the file, we need a good probability model and an efficient way of representing the probability model. In SAMC algorithm, the probability model is semi-adaptive which means that it uses a preliminary pass of the input file to gather the statistics and it uses a Markov model to compute subsequent probabilities.

SAMC Algorithm was implemented on two different architectures, Analog Devices Sharc and ARM's ARM and Thumb instruction sets. It compressed eight benchmark programs (DES*, Fft, Lucas, Mpegencode*, Mpegdecode*, Reed-Solomon and Xlisp). SAMC algorithm gives an average compression ratio of 45%.

Xie and Lekatsas [6] proposed a code compression algorithm for Very Long Instruction Word (VLIW) processors by using a variable-to-fixed (V2F) length encoding which was proposed by Tunstall [7]. V2F algorithms translate variable-length sequences into fixed-length sequences. Since the codeword length is fixed, it is easier to index into the compressed data which is one of the main advantages of V2F coding. There are two variants of this V2F algorithm, known as the Memoryless V2F coding algorithm and Markov V2F coding algorithm.

1. Memoryless V2F coding algorithm:

   Assuming that the 1s and 0s in the executable code have independent and identical distribution (iid), we calculate the probabilities for 1s and 0s. For example, in TMS320 Instruction set Architecture (ISA), the probability of 0 is 75% and 1 is about 25%. For this
algorithm, we first make a tree with the root node which has probability 1. Then, we attach 1 and 0 to the root, which leads in two leaf nodes each having the probability of occurrence as 0 and 1, i.e. P (0) and P (1) respectively. The leaf node which has the highest probability is divided into two branches with 1 and 0 as the label. The number of leaf nodes increment by 1 after the division and this is reiterated until the total number of nodes is $2^N$. And finally we assign equal length codeword having length N to the leaf nodes arbitrarily which is stored in a codebook. After the coding tree and the codebook have been constructed, the compression of the instructions is done block by block to assure random access. For compressing each block, we start from the root node and if we encounter a ‘1’, we take the right branch and if a ‘0’, we take the left branch. Whenever we come across a leaf node, the codeword representing that node is assigned and we re-start from the root node.

Figure 2.4 [6] shows the construction of a 2-bit Tunstall coding tree and a codebook for a binary stream with iid probabilities. The probability that a bit is 0 is 0.7 and the probability that a bit is 1 is 0.3. The code tree will expand till there are 4 leaf nodes and a 2-bit codeword is assigned randomly to every leaf node. Thus, the compression of the binary stream is very straightforward once the codebook is built. For example, for the binary stream 01 000 001, we start traversing through the coding tree as per the binary stream until we end up on a leaf node. The code corresponding to that leaf node is then assigned as the codeword and it will generate 01 11 10.
2. Markov V2F coding algorithm:

In this algorithm, we combine the memoryless V2F coding algorithm with a Markov model. Using the memoryless V2F algorithm, we create a $N$-bit variable-to-fixed length coding tree and codebook for each node in the Markov model. Thus, for an $L$-state Markov model, we have $L$ codebooks and each codebook has $2^N$ codewords using the $N$-bit variable-to-fixed length codes.

Implementation was carried out on Texas Instrument’s TMS320C6x and Intel’s IA-64 architecture. Most of the benchmarks were provided by Texas Instruments or part of Mediabench, which are for general embedded systems and applications that have a strong DSP component. TMS320C6x benchmarks (adpcm, block_mse, fir, g721, iir, mac_vselp, maxmin, modem, mpeg, pegwit and vertibi) are compiled using the Code Composer Studio IDE from Texas Instruments and IA-64 benchmarks (3d, adpcmenc, cpr, diesel, epic, g721, key, mpeg, pegwit, smc and trick) were compiled using IA-64 Linux Developer’s Kit from HP. Xie and
Lekatsas achieve a compression ratio of 82.5% using the memoryless V2F coding algorithm and 56% using the Markov V2F coding algorithm respectively on TMS320C6x. They achieve a compression ratio of 72% using the memoryless V2F coding algorithm and 70% using the Markov V2F coding algorithm respectively on IA-64.

**Lin, Xie and Wolf** [8] proposed the LZW code compression algorithm for compressing branch blocks. Branch blocks are instructions between two consecutive possible branch targets and they provide larger blocks for compression. This method is fully adaptive and generates a coding table on the fly during compression and decompression. Whenever a branch target is encountered, we clear the coding table to ensure correctness. Since branch blocks are used as the basic compression unit, programs in the memory would be compressed and decompressed on the fly as and when the branch blocks are needed. While applying LZW algorithm for code compression, we use the byte as our basic element. A new entry is generated every iteration. During the compression phase, the compressor finds the longest phrase in the table, and sends the corresponding codeword to the output and then adds the phrase as a new entry with the next byte. When the table gets full, the compressor keeps using the existing table to compress upcoming data.

**Table 2.2 LZW coding example [8]**

<table>
<thead>
<tr>
<th>Index</th>
<th>Phrase</th>
<th>Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>Initial</td>
</tr>
<tr>
<td>1</td>
<td>y</td>
<td>Initial</td>
</tr>
<tr>
<td>2</td>
<td>xx</td>
<td>0 + x</td>
</tr>
<tr>
<td>3</td>
<td>xy</td>
<td>0 + y</td>
</tr>
<tr>
<td>4</td>
<td>yx</td>
<td>1 + x</td>
</tr>
<tr>
<td>5</td>
<td>xyx</td>
<td>3 + x</td>
</tr>
<tr>
<td>6</td>
<td>xyxx</td>
<td>5 + x</td>
</tr>
</tbody>
</table>
Table 2.2 [8] shows a LZW coding example. Suppose we have two symbols x, y and the branch block consists of the sequence “xxyxyxyxxx”. Initially, the LZW coding table consists of all possible symbols and phrases (series of symbols) will be added to the table during the runtime. During the compression phase, the compressor searches for the longest phrase. When it finds the phrase, it outputs the corresponding index and adds this phrase with the next symbol as a new entry to the coding table. At the start, the longest phrase is ‘x’ and hence code 0 is generated and then as ‘x’ comes, the code 0 is again generated and ‘xx’ will be added in entry 2. As the next symbol is ‘y’, code 1 is generated and ‘xy’ will be added in entry 3. Now, the next symbol is ‘xy’ and it finds it in the table as code 3. Also ‘xyx’ is added to the table as entry 5. As the next symbol is ‘xyx’, code 5 is generated. After that the symbol is ‘xx’ and hence code 2 is generated. Thus, as the compression goes on, code “001352” will be generated. When the decompressor gets the first ‘0’, it will decode it as ‘x’ but no entry is generated since it has no idea what the next codeword is. “xx” will be added only after the next codeword arrives.

Whenever a codeword is read from memory, we first check if it is a branch target. If it is a branch target, then we shift out the padding from the buffer, reset the coding table and restart the decompression at a byte aligned position. If it is a not a branch target, the decompressor core gets a codeword, looks it up in the table, outputs the content and then adds the old phrase with the first element of next phrase as a new entry. Thus, the LZW coding table has all the possible symbols initially and new phrases will be added eventually during the runtime. During compression, the compressor searches for the longest phrase in the table and outputs the index and then adds this new phrase with the next symbol as a new entry. Although both pre-cache and post-cache architecture can be employed for decompression, the LZW algorithm works better with post-cache architecture due to its larger bandwidth. Also, the post-cache architecture has more area and power savings. The LZW algorithm is a variable-to-fixed code compression scheme. It can be used for parallel decompression by using a look-ahead scheme where it can
start decompressing a codeword when the newly generated codeword is not used. The coding table used by the LZW algorithm for compression and decompression is determined by the decompression bandwidth and the codeword length. Since this algorithm has wider decompression bandwidth, it is most suited for VLIW architecture.

Experiments are performed on TI’s TMS320C6x VLIW processor. The benchmarks are collected from Texas Instruments and Mediabench, which are general embedded system applications with a strong DSP component. The benchmarks (adpcmdec, adpcm, block_mse, codebook, fir_filter, hellodsp, hello, idct, iir_cas, iir_filter, maxminave, min_err, modem, mpeg2enc, vertibi and average) are compiled using Code Composer Studio IDE from TI. A compression ratio (C.R) of about 83-87% is achieved by implementing the LZW algorithm on TI's TMS320C6x VLIW processor.

Lefurgy [2] proposed the dictionary based coding algorithm which basically takes advantage of repeated sequences in a program by using a dictionary. The repeated sequence could be a byte, an instruction field, a whole instruction or a group of instructions. This algorithm replaces the commonly repeated occurrences by a codeword which point to an index in the dictionary that comprises the repeated pattern. Thus the compressed instruction will now contain both the uncompressed instruction and the codeword.

Figure 2.5 [2] shows an example of the dictionary coding algorithm. This method exploits the fact that most of the instructions in a program are highly repetitive. By replacing these patterns by shorter codewords, it can remove repetition and improve the code density of the program. During the execution of the program, the codewords are expanded back into their original form and executed in a normal manner. The algorithm uses indices into the instruction dictionary as codewords. As long as the index is shorter than the repeating sequence it replaces, the compression will be in effect and the overhead of the dictionary will not be huge. Since all
repeating sequences are replaced by a single codeword which points to an index into the dictionary, this algorithm alters the location of instructions in the program. This creates a big problem for branch instructions because the branch targets may change as a result of code compression. To tackle this problem, this algorithm does not compress relative branch instructions. These branch instructions are the ones which contain an offset field which is used to compute a branch target. In a processor, the instruction decoder proceeds directly with an uncompressed instruction. If it encounters a codeword, it first translates it to the original instruction sequence and then executes the instruction. Codewords can be of fixed or of variable length. Variable length codewords lead to a higher decoding penalty but they provide better compression than fixed length codewords. Code compression is most effective on large programs rather than small programs, the reason being that smaller programs have less scope for commonly repeating occurrences and hence compressible sequences are less frequent.

![Diagram](image)

Figure 2.5 Dictionary code example [2]
This compression technique is integrated into the PowerPC, ARM, and i386 instruction sets. For PowerPC and i386, SPEC CINT95 benchmarks (gcc, compress, go, ijpeg, ii, m88ksim, perl and vortex) were compiled with GCC 2.7.2 using -02 optimization. SPEC CINT92 and SPEC CINT95 benchmarks (expresso, li, eqntott and compress) were compiled for ARM6 using the Norcroft ARM C compiler v4.30. The dictionary coding algorithm gives a compression ratio of about 64-87% upon the implementation on PowerPC, ARM and i386 instruction sets.

Seong and Prabhat Mishra [1] proposed the Bitmask based code compression algorithm. This algorithm takes advantage of the commonly occurring sequences in the program and is similar to the dictionary based coding algorithm. This algorithm goes one step further by considering mismatches and tries to increase the commonly occurring sequences in a program so that more compression can be achieved. The basic idea is to store the mismatches by remembering a few bit positions in order to increase the number of instruction matches. This algorithm, using bitmasks, significantly improves the compression efficiency and does not introduce much of a decompression penalty. However, the efficiency of this algorithm is constrained by the number of bit changes which are considered during compression. It is obvious that, if we consider more bit changes, it will create more instruction matches. But there will be a point where the cost of storing the extra bit positions will offset the advantage of increasing the commonly occurring sequences in the program. The bitmask algorithm incorporates maximum bit changes by using bit mask patterns. For a 32 bit instruction, the mask type may be of 1, 2, 4 or 8 bits. So the extra information which needs to be stored for every mismatch is the mask type, the position where the mask needs to be applied and the mask pattern.

Implementation was carried out on MIPS, SPARC and TI’s TMS320C6x architecture. The benchmarks are collected from TI, Mediabench and MiBench benchmark suites. The benchmarks (adpcm_en, adpcm_de, djpeg, gsm_to, gsm_un, cjpeg, modem, helo, mpeg2enc, pegwit and vertibi) are compiled using TI’s Code Composer Studio for TMS320C6x and gcc for
MIPS and SPARC. The bitmask code compression algorithm improves the compression ratio to about 55-65% upon the implementation on MIPS, SPARC and TMS320C6x processors. We discuss this algorithm in detail in Chapter 3.

2.4 Thesis approach

From Table 2.1, it is very clear that the bitmask based coding algorithm is capable of achieving a very low compression ratio (55-65%) and it is suitable for parallel decompression as well. Hence, in this research, we have chosen the bitmask based coding algorithm. In this thesis, we will be discussing the bitmask based coding algorithm in detail and will implement it on Texas Instrument’s MSP430. The benchmarks can be collected from the Texas Instruments website and compiled using the Code Composer Studio IDE v4.0 compiler to generate the binary. The software code for the bitmask based coding algorithm will be developed in C++ and then implemented on the target architecture. The compression ratio for measuring the efficiency of the compression can be computed using equation (2.1). The compressed code program size will also comprise of the size of the dictionary and a mapping table used for branches.
3. METHODOLOGY

The underlying purpose of this thesis is to understand the bitmask code compression algorithm in detail and to implement it on target architecture to analyze how much code compression can be achieved by using this technique. This chapter discusses the flow of the work in detail.

3.1 Flow of the work

The basic structure of this work is as below:

1. Explain the bitmask based code compression algorithm in detail.
2. Develop the software code for the bitmask based code compression algorithm.
3. Collect the sample code programs from the Texas Instruments website for the specific target architecture MSP430.
4. Compile the sample code programs using the TI Code Composer Studio v4.0 to generate the binary for the target architecture MSP430.
5. Implement the bitmask based code compression algorithm on the binary generated in step 4.
7. Analyze the results for different code examples for MSP430.

3.2 Bitmask based code compression

Before we discuss the bitmask based code compression algorithm in detail, it is necessary to understand previous dictionary based code compression approaches. This section first discusses previous dictionary based approaches in brief and then the bitmask based code compression approach in detail.

3.2.1 Standard dictionary-based approach [1]

The main idea behind a dictionary based approach was to exploit frequently occurring instruction sequences in a program. It has been observed that most of the times, 90% of the
execution time of a program is spent in 10% of its code. In dictionary based approach, we try to exploit this property of a program. The tendency of programs to reuse instructions which have been used recently is known as the principle of locality [9]. The idea of putting frequently occurring instructions in a dictionary and replacing them by the dictionary index as a shorter codeword gained a lot of popularity since then. Figure 3.1 shows an example of dictionary based code compression. The program binary consists of twelve 8-bit patterns. The program binary occupies a total of 96 bits. The dictionary has two 8-bit patterns. The compressed program requires 73 bits and the dictionary requires 16 bits. The compression ratio is 92%.

```
00000000  ----->  0 0
10000010  ----->  1 10000010
00000010  ----->  1 00000010
01000010  ----->  0 1
01001110  ----->  1 01001110
01010010  ----->  1 01010010
00001100  ----->  1 00001100
01000010  ----->  0 1
11000000  ----->  1 11000000
00000000  ----->  0 0
00000010  ----->  1 00000010
01000010  ----->  0 1

<table>
<thead>
<tr>
<th>Index</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>01000010</td>
</tr>
</tbody>
</table>
```

Figure 3.1 Dictionary based code compression
3.2.2 Mismatch-based dictionary approach

There have been a lot of improvements in the dictionary-based code compression approach [10, 11]. One such improvement over the dictionary-based code compression approach is to increase the number of frequently occurring instruction sequences by considering mismatches. As the number of frequently occurring instructions in a program may be limited, this technique is aimed towards increasing the frequencies by storing the mismatches in the program along with the dictionary. The basic principle of this approach is to find out the instruction sequences which differ in only a few bit positions and store this information in the compressed program. The compression ratio will change depending upon the number of bit changes that are considered for this compression. The number of bits required to store the mismatch position depends on the number of bits in one instruction sequence. An 8-bit instruction sequence will require 3 bits to store the mismatch position.

Figure 3.2 shows an example of mismatch-based code compression (same example as Figure 3.1). This example uses a 1-bit change for 8-bit vectors. The program binary consists of twelve 8-bit patterns. The program binary occupies a total of 96 bits. The dictionary has two 8-bit patterns. The compressed program requires 72 bits and the dictionary requires 16 bits and the compression ratio is 91% (using the expression 2.1).

Figure 3.3 [1] shows an example of the encoding format used for mismatch-based code compression for an 8-bit program code. There is one decision bit in both the compressed and uncompressed code to indicate if it is compressed or not. If it is compressed, then there is another bit to indicate the number of bit changes considered for compression. Depending on the number of bit changes considered, there are respective mismatch positions in the compressed code. The compressed code ends with one final bit for indicating the dictionary index.
Figure 3.2 Mismatch-based dictionary code compression

Uncompressed code format

<table>
<thead>
<tr>
<th>Decision (1-bit)</th>
<th>Uncompressed code (8-bits)</th>
</tr>
</thead>
</table>

Compressed Code Format

<table>
<thead>
<tr>
<th>Decision (1-bit)</th>
<th>Number of bit changes (3-bits)</th>
<th>Location (3-bits)</th>
<th>..........</th>
<th>Location (3-bits)</th>
<th>Dictionary index</th>
</tr>
</thead>
</table>

Extra bits for incorporating mismatches

Figure 3.3 Encoding format for incorporating mismatches [1]
3.2.3 Bitmask-based dictionary approach [1]

From the previous dictionary-based code compression approaches as discussed above, it is evident that we can increase the number of frequently occurring instruction sequences by considering the change in bit positions. Thus, if we consider 2-bit changes in the example shown in Figure 3.2, we can cover almost all instruction sequences. But it does not guarantee a good compression ratio because we need additional bits to store multiple bit changes in the instruction sequence. The compression ratio would become worse if we consider 2-bit changes for the same example as in Figure 3.2. The bitmask-based code compression approach makes the use of bitmasks for storing the bit changes and thus, requires lesser bits to capture the bit changes between two instruction sequences. For example, if there is an 8-bit change in two 32-bit instruction sequences, we can use an 8-bit mask to capture the mismatch. Thus, it will only require an additional (8+5) 13 bits to capture the mismatch with the additional 5 bits to store the start position of the bitmask and 8 bits to indicate the bitmask. The main idea of a bitmask-based approach is to incorporate maximum bit changes such that the compression ratio is significantly improved.

Figure 3.4 [1] shows an example of a bitmask-based code compression algorithm. In his example, we have created 100% matching patterns by considering a 2-bit mask. We have considered the 2-bit mask only on byte boundaries at the quarter in this example. This requires even lesser bits but it may skip the mismatches which extend across two byte boundaries. In this example, the bit mask is applied only at bit positions 0, 2, 4 and 6 in the 8-bit instruction sequence where the bitmask positions 00, 01, 10 and 11 correspond to bit position 0, 2, 4 and 6 respectively. The program binary consists of twelve instruction sequences. The program binary occupies a total of 96 bits. The dictionary has two 8-bit patterns. The compressed program requires 64 bits and the dictionary requires 16 bits. The compression ratio is 83% (using the expression 2.1). Thus, the compression ratio is significantly improved by using bitmask-based
approach.

Figure 3.5 [1] shows the generic encoding format used by the bitmask-based code compression scheme. As shown in this figure, information regarding multiple mask types can also be stored in this scheme. For example, for a 32-bit instruction sequence, we can apply 1-b, 2-b, 4-b or 8-b mask and we need 2-b to differentiate between the various mask types. The mask location is the start position in the instruction sequence where the bit mask needs to be applied and the index points to the dictionary entry on which the bitmask needs to be applied.
Uncompressed code format

| Decision (1-bit) | Uncompressed code (32-bits) |

Compressed code format

<table>
<thead>
<tr>
<th>Decision (1-bit)</th>
<th>Number of mask patterns</th>
<th>Mask type</th>
<th>Location</th>
<th>Mask pattern</th>
<th>Dictionary index</th>
</tr>
</thead>
</table>

Extra bits for incorporating bitmasks

Figure 3.5 Encoding format for incorporating bitmasks [1]

3.3 Compression algorithm

The algorithm shown below lists the basic steps of the bitmask-based code compression technique [1]. The original code (binary) is divided into 16-bit vectors and fed as an input to the algorithm.

Algorithm: Bitmask-based code compression

Input: Original code (binary) divided into 16-bit vectors.

Outputs: Compressed code and dictionary.

Begin

Step 1: Create the frequency distribution of the 16-bit vectors.

Step 2: Create the dictionary based on Step 1.

Step 3: Compress each 16-bit vector using the bitmask.
Step 4: Handle and adjust branch targets.

return Compressed code and the dictionary

End

The first step of the algorithm is to create a frequency distribution of the 16-bit vectors which are an input to the algorithm. The number of times a particular instruction sequence repeats in a program is termed as the frequency for that instruction sequence. Based on this frequency computation, it builds up a dictionary in the second step. It is always desirable to have the smallest possible dictionary size which can accommodate all high frequency vectors. But if there are a large number of high frequency vectors, then it increases the dictionary size. A large dictionary size can be a disadvantage because it increases the access time and also reduces the decompression efficiency. In such cases, we build a dictionary of high frequency vectors only above certain threshold. The third step is to compress each 16-bit vector into a compressed code format as shown in Figure 3.5.[1]. The compressed code along with the uncompressed code is composed in a serial form to form the final compressed code.

The final step of the algorithm is to handle and adjust branch targets. As we try to compress the entire program using bitmasks, it may give rise to a branch-instruction problem. This is because the compression changes the branch target addresses in case of conditional as well as unconditional branches in the original code. To resolve this problem, various approaches have been suggested. The LAT (Line Address Table) approach was proposed by Wolfe and Chanin [3] where they mapped the branch target addresses in the original code onto new target addresses in the compressed code. Another back-patching approach was suggested by Lefurgy [2] which includes patching of the original branch targets into new offsets in the compressed code. Also, additional bits were added at the end of the code which preceded branch targets so that it adjusts the byte boundary.
### 3.4 Decompression core

The decompression core can be employed in two different ways in embedded systems. For example, the decompression engine can be placed in between the main memory and the instruction cache which is known as pre-cache architecture. Alternatively, the decompression engine can be placed in between the processor and the instruction cache which is known as post-cache architecture. As the instruction cache contains the compressed code in post-cache architecture, it has an advantage over pre-cache architecture because it increases the cache hit ratio and the bandwidth.

Figure 3.6 shows one possible implementation for the decompression core in hardware when a single mask pattern is used for the bitmask-based code compression algorithm. The first bit i.e. the decision bit in the compressed code indicates if the instruction sequence is compressed or not. If it is compressed, then the corresponding entry from the dictionary is retrieved and it is XOR-ed with a 16-bit mask pattern. The information regarding the bitmask pattern as well as the bitmask position is encoded in the compressed code as per the compressed code format in Figure 3.5.

![Figure 3.6 Possible implementation of a decompression core](image)
The decompression core has to perform two main functions. The first function is to generate the 8-bit mask pattern based on the encoded information and the second function is to XOR the dictionary entry and the 8-bit mask pattern. The generation of the 8-bit mask pattern and the retrieval of the dictionary entry can be done in parallel as they are completely independent of each other and thus, fasten the decompression process. The decompression core may also need to access the LAT (Line Address Table) in case of branches to fetch the correct branch target address.

3.5 Target processor

We will be implementing the bitmask-based code compression algorithm on Texas Instrument’s 16-bit MSP430 microcontroller [12]. The features of the microcontroller are listed below:

- The MSP430 is a 16-bit microcontroller which is used for a wide range of portable and low-power applications.
- MSP430 MCUs have an excellent mix of intelligent peripherals, low cost and low power consumption for many applications.
- The key applications of MSP430 MCUs include metering devices, portable medical devices, data logging and wireless communications, capacitive touch sense IOs, energy harvesting, motor control and security and safety devices.
3.6 Code outline

The program code is divided into 16-bit vectors in binary format and is read into a binary search tree. The binary search tree has two fields namely data and frequency. The frequency of each 8-bit input vector is computed while reading into the binary search tree. Based on the frequencies, a threshold is chosen and a dictionary is created for all 16-bit input vectors above that threshold. The dictionary has an index which points to the 16-bit input vector and which is used as a shorter codeword while compressing the input.

During the compression phase, as the input vectors are read, the dictionary is searched for a matching entry. If the entry is found, the 16-bit vector is replaced by the index as a shorter codeword. If there is no matching entry in the dictionary, then we check if the bitmask can be applied such that it matches up to some dictionary entry or not. If the bitmask can be applied, then the information regarding the bitmask location and mask pattern along with the dictionary entry is stored with the compressed code. If the bitmask cannot be applied, then it cannot be compressed at all and stored as it is.

We developed the software code in C++ to implement the bitmask-based code compression algorithm.
The flowchart in Figure 3.7 shows the code outline for the same.

Figure 3.7 Code outline
4. EXPERIMENTS

4.1 Experimental setup

We performed the bitmask based code compression experiments by varying the sample code programs on target architecture MSP430. The following section presents the experimental results obtained by using fifteen sample code programs for Texas Instrument's MCU MSP430. The sample code programs are collected from the TI website [13]. The sample programs were compiled for target architecture MSP430 using the TI Code Composer Studio v4.0 to generate the binary. The compression ratio was calculated using the equation (2.1). The computation of the compressed program size also comprises the size of the dictionary.

Figure 4.1 shows the encoding format used for the bitmask code compression algorithm in our implementation. The first bit is a decision bit which indicates if the code is compressed or not. The second bit is a mask bit which indicates if the bitmask is applied or not. The next bit indicates the location in the code where the bitmask is to be applied. The mask pattern bit indicates the bitmask pattern. The dictionary index field points to the original code entry in the dictionary.

<table>
<thead>
<tr>
<th>Decision (1-bit)</th>
<th>Mask (1-bit)</th>
<th>Location</th>
<th>Mask pattern</th>
<th>Dictionary index</th>
</tr>
</thead>
</table>

Figure 4.1 Encoding format used in our implementation

The dictionary size can be fixed in this algorithm but it does not guarantee the best compression ratio as it would miss out on many frequently occurring instruction sequences. In our implementation, we have kept the dictionary size variable and we fix a frequency threshold
instead. The dictionary is created on the fly of all instruction sequences above a fixed frequency threshold.

Table 4.1 shows the complete set of sample code programs available on the TI website with their respective code sizes.

<table>
<thead>
<tr>
<th>Impl?</th>
<th>Sample programs</th>
<th>Description</th>
<th>Code size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>ADC10_ex1</td>
<td>Analog-to-digital converter</td>
<td>5185</td>
</tr>
<tr>
<td>Yes</td>
<td>ADC10_ex2</td>
<td>Analog-to-digital converter</td>
<td>8993</td>
</tr>
<tr>
<td>No</td>
<td>ADC10_ex4</td>
<td>Analog-to-digital converter</td>
<td>5423</td>
</tr>
<tr>
<td>No</td>
<td>ADC10_ex5</td>
<td>Analog-to-digital converter</td>
<td>5440</td>
</tr>
<tr>
<td>Yes</td>
<td>ADC12_ex1</td>
<td>Analog-to-digital converter</td>
<td>7055</td>
</tr>
<tr>
<td>Yes</td>
<td>ADC12_ex2</td>
<td>Analog-to-digital converter</td>
<td>4828</td>
</tr>
<tr>
<td>No</td>
<td>ADC12_ex3</td>
<td>Analog-to-digital converter</td>
<td>4233</td>
</tr>
<tr>
<td>No</td>
<td>ADC12_ex6</td>
<td>Analog-to-digital converter</td>
<td>4777</td>
</tr>
<tr>
<td>No</td>
<td>comp_ex1</td>
<td>Compares two voltages and switches its output to indicate which is larger</td>
<td>4539</td>
</tr>
<tr>
<td>Yes</td>
<td>comp_ex2</td>
<td>Compares two voltages and switches its output to indicate which is larger</td>
<td>5695</td>
</tr>
<tr>
<td>Yes</td>
<td>comp_ex5</td>
<td>Compares two voltages and switches its output to indicate which is larger</td>
<td>19958</td>
</tr>
<tr>
<td>Yes</td>
<td>crc_ex1</td>
<td>Build signature and rebuild to test</td>
<td>3910</td>
</tr>
<tr>
<td>Yes</td>
<td>CS_ex1_DCOSetup</td>
<td>Configure the clock for 8MHz operation</td>
<td>17969</td>
</tr>
<tr>
<td>Yes</td>
<td>dac12_ex1</td>
<td>Digital-to-analog converter</td>
<td>2567</td>
</tr>
<tr>
<td>Yes</td>
<td>dma_ex1_repeatedBlock</td>
<td>Repeated Block Transfer to-and-from RAM</td>
<td>5916</td>
</tr>
<tr>
<td>Yes</td>
<td>FLASH_example1</td>
<td>Writes data into FLASH memory</td>
<td>7922</td>
</tr>
<tr>
<td>Yes</td>
<td>FRAM_ex1_write</td>
<td>Writes long word into 512 byte blocks of FRAM</td>
<td>9367</td>
</tr>
<tr>
<td>Yes</td>
<td>ramcontroller_example1</td>
<td>RAM controller turn off RAM sector</td>
<td>5338</td>
</tr>
<tr>
<td>Yes</td>
<td>TIMER_continuousModeWithCCR0Interrupt</td>
<td>Continuous Mode Timer</td>
<td>5304</td>
</tr>
<tr>
<td>Yes</td>
<td>UART_TX</td>
<td>Configures the UART module to send data in UART mode and read echoed data</td>
<td>33966</td>
</tr>
</tbody>
</table>
Out of the sample code programs listed in Table 4.1, we have chosen fifteen code programs for our experiments on the basis of their varying code size and redundancies. For example, the code programs ADC10_ex1, ADC10_ex2, ADC10_ex4, ADC10_ex5, ADC12_ex1, ADC12_ex2, ADC12_ex3 and ADC12_ex5 are for analog-to-digital converters with slight variations in their program. In order to remove the redundancy, we have chosen a few of them on the basis of their code size. The first column (Impl) in Table 4.1 indicates if the respective sample code program is considered for implementation or not (Yes/No).

In our experiments, we have implemented three test cases:

Test Case 1: Applied a 2-bit mask

Test Case 2: Applied a 4-bit mask

Test Case 3: Applied a 8-bit mask

It is very evident that a 16-bit mask pattern will not be profitable as it will require more bits than the uncompressed 16-bit input vector and will give a worse compression ratio. For a 2-bit mask, we require 3 bits to store the location of the bitmask (8 locations where we can apply the 2-bit mask) and 2 bits for the mask pattern in addition to the first 2 bits for the decision and mask bit. Thus, we require 7 bits for storing the 2-bit mask. Depending on the dictionary size, additional bits are required for the storing the index. Similarly, for a 4-bit mask, we require 2 bits to store the location (4 locations to apply the 4-bit mask) and 4 bits for the mask pattern. Thus, we require 8 bits for storing the 4-bit mask. In case of an 8-bit mask, we require 1 bit to store the location (2 locations to apply the bitmask) and 8 bits for the mask pattern. Thus, we require 11 bits for storing the 8-bit mask.

In our implementation, we have used a constant 4-bit location so that we can apply the bitmask anywhere in the 16-bit input vector and not limit the bitmask to the definite byte boundaries.
Thus, we require 8 bits, 10 bits and 14 bits for storing the 2-bit, 4-bit and 8-bit mask patterns respectively.

4.2 Results

The below results show the performance of all three test cases on the fifteen sample code programs.

1. ADC10_ex1

This is a sample program for an Analog-to-Digital converter (ADC). It takes sample input at node A0 and the reference voltage at AVcc Ref. A single sample is made on A0 with reference to AVcc. Software sets the ADC to start sample and conversion. The internal oscillator times the sample (16x) and conversion.

Figure 4.2 shows the performance of this sample program on the implementation of a 2-bit, 4-bit and an 8-bit mask pattern.

![Figure 4.2 Compression ratio for ADC10_ex1](image)

Figure 4.2 Compression ratio for ADC10_ex1
2. ADC10_ex2

This is a sample program for an Analog-to-Digital converter (ADC). It takes sample input at node A0 and the reference voltage at AVcc Ref. A single sample is made on A0 with reference to internal 1.5 Vref. Software sets the ADC to start sample and conversion. The internal oscillator times the sample (16x) and conversion. Figure 4.3 shows the performance of this sample program on the implementation of a 2-bit, 4-bit and an 8-bit mask pattern.

![Compression graph](image)

Figure 4.3 Compression ratio for ADC10_ex2

3. ADC12_ex1

This is a sample program for a Analog-to-Digital converter (ADC). It takes sample input at node A0 and the reference voltage at AVcc Ref. A single sample is made on A0 with reference to internal 1.5 Vref. Software sets the ADC to start sample and conversion. The internal oscillator times the sample (16x) and conversion. Figure 4.4 shows the performance of this sample program on the implementation of a 2-bit, 4-bit and an 8-bit mask pattern.
4. ADC12_ex2

This is a sample program for an Analog-to-Digital converter (ADC). This program uses the internal 1.5V reference and performs a single conversion on channel A0. It takes sample input at node A0 and the reference voltage at AVcc Ref. Figure 4.5 shows the performance of this sample program on the implementation of a 2-bit, 4-bit and an 8-bit mask pattern.
5. comp_ex2

This is the sample program for a comparator. It takes a CB interrupt and the input is compared against 1.5V internal reference. It uses CB interrupt and an internal reference to determine if the input is high or not. Figure 4.6 shows the performance of this sample program on the implementation of a 2-bit, 4-bit and an 8-bit mask pattern.

![Compression ratio for comp_ex2](image)

6. comp_ex5

This is the sample program for a comparator. It takes two inputs with varying duty cycle and determines which input is higher. Figure 4.7 shows the performance of this sample program on the implementation of a 2-bit, 4-bit and an 8-bit mask pattern.
This sample program builds a signature using a seed and multiple data values. This signature is considered as the checksum and can be sent by a UART connection along with the data to verify the correct data has been sent. This program also tests CRC checksum of the data that has been created, by recreating the same checksum and comparing it to the first checksum.

Figure 4.8 shows the performance of this sample program on the implementation of a 2-bit, 4-bit and an 8-bit mask pattern.
8. **CS_ex1_DCOSetup**

This sample program sets the internal clock for 8MHz operation. Figure 4.9 shows the performance of this sample program on the implementation of a 2-bit, 4-bit and an 8-bit mask pattern.

![Figure 4.8 Compression ratio for crc_ex1](image)

**Figure 4.8 Compression ratio for crc_ex1**

![Figure 4.9 Compression ratio for crc_ex1_DCOSetup](image)

**Figure 4.9 Compression ratio for crc_ex1_DCOSetup**
9. **dac12_ex1**

This sample program is for a Digital-to-Analog converter (DAC). It converts a signal from digital form to analog form.

Figure 4.10 shows the performance of this sample program on the implementation of a 2-bit, 4-bit and an 8-bit mask pattern.

![Compression Ratio](chart.png)

**Figure 4.10 Compression ratio for dac12_ex1**

10. **dma_ex1_repeatedBlock**

This sample program is for a Direct Memory Access (DMA) repeated-block-transfer to-and-from RAM. A 16 word block is transferred from one memory location to another memory location in a burst mode.

Figure 4.11 shows the performance of this sample program on the implementation of a 2-bit, 4-bit and an 8-bit mask pattern.
11. FLASH_example1

This sample program is for dummy data calibration into a memory location. 16 locations are written with char data, 8 locations with int data and 4 locations with word data.

Figure 4.12 shows the performance of this sample program on the implementation of a 2-bit, 4-bit and an 8-bit mask pattern.
12. FRAM_ex1_write

This sample program is for writing long words into FRAM. This program writes to 512 byte blocks of FRAM. Figure 4.13 shows the performance of this sample program on the implementation of a 2-bit, 4-bit and an 8-bit mask pattern.
13. ramcontroller_example1

This sample program is for a RAM controller. During low power modes, this program shuts off the RAM sector instead of retention state. Figure 4.14 shows the performance of this sample program on the implementation of a 2-bit, 4-bit and an 8-bit mask pattern.

![Figure 4.14 Compression ratio for ramcontroller_example1](image)

14. TIMER_continuousModeWithCCR0Interrupt

This sample program is for a continuous mode timer. It toggles every 50000 clock cycles.

Figure 4.15 shows the performance of this sample program on the implementation of a 2-bit, 4-bit and an 8-bit mask pattern.

15. UART_TX

This sample program is for configuring the UART module to send data in UART mode and read the echoed data.
Figure 4.16 shows the performance of this sample program on the implementation of a 2-bit, 4-bit and an 8-bit mask pattern.

Figure 4.15 Compression ratio for TIMER_continuouModeWithCCR0Interrupt

Figure 4.16 Compression ratio for UART_TX
4.3 Summary of results

Table 4.2 summarizes the performance of all fifteen sample code programs for a 2-bit, 4-bit and 8-bit mask.

<table>
<thead>
<tr>
<th>Program</th>
<th>Original code size (bytes)</th>
<th>Compressed code size (bytes)</th>
<th>Dictionary size (bytes)</th>
<th>Compression ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2-bit</td>
<td>4-bit</td>
<td>8-bit</td>
<td>2-bit</td>
</tr>
<tr>
<td>ADC10_ex1</td>
<td>5185</td>
<td>3043</td>
<td>2968</td>
<td>3078</td>
</tr>
<tr>
<td>ADC10_ex2</td>
<td>8993</td>
<td>5470</td>
<td>5431</td>
<td>5643</td>
</tr>
<tr>
<td>ADC12_ex1</td>
<td>7055</td>
<td>4132</td>
<td>4095</td>
<td>4252</td>
</tr>
<tr>
<td>ADC12_ex2</td>
<td>4828</td>
<td>3035</td>
<td>2963</td>
<td>3108</td>
</tr>
<tr>
<td>comp_ex2</td>
<td>5695</td>
<td>3443</td>
<td>3383</td>
<td>3519</td>
</tr>
<tr>
<td>comp_ex5</td>
<td>19958</td>
<td>12960</td>
<td>13039</td>
<td>13592</td>
</tr>
<tr>
<td>crc_ex1</td>
<td>3910</td>
<td>2308</td>
<td>2285</td>
<td>2347</td>
</tr>
<tr>
<td>CS_ex1_DCOSetup</td>
<td>17969</td>
<td>11163</td>
<td>11143</td>
<td>11446</td>
</tr>
<tr>
<td>dac12_ex1</td>
<td>267</td>
<td>1743</td>
<td>1712</td>
<td>1720</td>
</tr>
<tr>
<td>dma_ex1_repeatedBlock</td>
<td>5916</td>
<td>3431</td>
<td>3402</td>
<td>3520</td>
</tr>
<tr>
<td>FLASH_example1</td>
<td>7922</td>
<td>5037</td>
<td>5026</td>
<td>5239</td>
</tr>
<tr>
<td>FRAM_ex1_write</td>
<td>9367</td>
<td>5923</td>
<td>5907</td>
<td>5142</td>
</tr>
<tr>
<td>ramcontroller_example1</td>
<td>5338</td>
<td>3396</td>
<td>3373</td>
<td>3470</td>
</tr>
<tr>
<td>TIMER_continuousMode</td>
<td>5304</td>
<td>3313</td>
<td>3268</td>
<td>3374</td>
</tr>
<tr>
<td>WithCCR0Interrupt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UART_TX</td>
<td>33966</td>
<td>22946</td>
<td>23207</td>
<td>24022</td>
</tr>
</tbody>
</table>

It is clear from Table 4.2 that the compression ratio is best for a 4-bit mask on sample code programs which have a code size less than 10000 bytes. For sample code programs with code size more than 10000 bytes, the compression ratio is best for a 2-bit mask and the performance of a 4-bit and 8-bit mask degrades.
This is because as the code size increases, the dictionary size also increases as we have not fixed the dictionary size in our implementation. Hence, the index size also increases because of the large number of entries in the dictionary. Hence, for an 8-bit mask, we need more number of bits to store the mask pattern along with the dictionary index whereas for a 2-bit mask, we need less number of bits to store the mask pattern. Thus, the code compression degrades in performance for larger code sizes for larger bit masks.

Table 4.3 shows the mean, max-min, standard deviation and standard error for the fifteen sample code programs.

<table>
<thead>
<tr>
<th>Function</th>
<th>2-bit mask</th>
<th>4-bit mask</th>
<th>8-bit mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean (%)</td>
<td>67.45</td>
<td>67.26</td>
<td>69.06</td>
</tr>
<tr>
<td>Minimum (%)</td>
<td>63.54</td>
<td>62.10</td>
<td>64.22</td>
</tr>
<tr>
<td>Maximum (%)</td>
<td>72.98</td>
<td>73.75</td>
<td>76.14</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>2.914</td>
<td>3.413</td>
<td>3.203</td>
</tr>
<tr>
<td>Standard error</td>
<td>0.752</td>
<td>0.881</td>
<td>0.827</td>
</tr>
</tbody>
</table>

Previous work by Seong and Mishra [1] show that the implementation of the bitmask based code compression algorithm on TI’s TMS320C6x gives a compression ratio of about 55-65%. This thesis implements the bitmask based code compression algorithm on TI’s MSP430 MCU and it gives a compression ratio of about 62-76%. Figure 4.17 shows the comparison between the ranges of the compression ratios achieved by both the implementations.
Figure 4.17 Comparison between the two implementations
5. CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

Embedded systems are severely constrained by the memory size. Various code compression approaches have been proposed to overcome this limitation by minimizing the size of the program code. Dictionary-based code compression techniques have been widely used as they provide good compression by taking advantage of the repeating patterns. Bitmask-based code compression technique creates more repeating patterns by taking advantage of the bitmask patterns and provides a better compression ratio compared to previous dictionary-based approaches.

In this thesis, we briefly discussed various code compression algorithms. We then discussed in detail the bitmask based code compression algorithm. A step by step procedure to implement the bitmask based code compression algorithm was presented. The implementation was carried out on Texas Instrument’s 16-bit MSP430. MSP430 is a 16-bit RISC based microcontroller which is designed specifically for ultra-low power applications. We implemented a 2-bit, 4-bit and 8-bit mask on MSP430 and compressed the fifteen sample code programs using the bitmask-based code compression algorithm. We presented the results indicating the compression ratios for each sample program. We found that for programs with code size below 10000 bytes, 4-bit mask pattern gave the best compression ratio whereas for programs with code size above 10000 bytes, a 2-bit mask pattern gave the best compression ratio. Thus, we can conclude that the code compression degrades in performance for larger code sizes with the application of a larger bit mask pattern.

We also attached a tutorial on a step-by-step implementation process of bitmask based code compression algorithm on MSP430 microcontroller in Appendix A.
We achieved a compression ratio of about 62-76% using the bitmask based code compression technique. Based on the results, we can conclude that a good amount of compression can be achieved by using the bitmask based code compression algorithm.

5.2 Future work

This work implements the bitmask based code compression algorithm on TI's MSP430 and shows that a good compression ratio of about 62-76% can be achieved using this technique. In this thesis, we only considered the performance of the bitmask based code compression algorithm in terms of how much code is compressed using this technique. We only analyze the compression ratios in the result section. Further future work can include analysis on the power savings and the area savings achieved by using this technique. Also, we implemented this technique on a fixed instruction set. Further future work can be extended to variable sized instruction sets.
REFERENCES


APPENDIX A

Implementation of bitmask based code compression algorithm: A tutorial

This tutorial is intended to provide a step-by-step description of the implementation process of the bitmask based code compression algorithm on TI's MSP430 MCU. The steps involved are:

1. Install the Code composer studio v4.0 [14]. There is a 30 day free evaluation license option available for this tool. After 30 days, you will have to upgrade it to a full license.
2. This free license option supports only MSP430 and C28x. It supports MSP430 for a code size of 16KB and C28x up to a code size of 32KB.
3. Collect the sample code programs for MSP430 from the TI website [13] and compile them using the code composer studio into a binary format.
4. Run the code for the bitmask based code compression algorithm for a 2-bit mask, 4-bit mask and 8-bit mask respectively on each of these compiled sample code programs in binary.
5. Compute the compression ratio for each of these sample programs for a 2-bit mask, 4-bit mask and 8-bit mask respectively.
6. Compare and analyze the compression ratios for each of these sample programs.

This tutorial is based on the version 4.0 of Texas Instrument’s Code Composer Studio. MSP430Ware, a collection of code examples and the MSP430 driver library is available free when you download Code Composer Studio v4.0. We will use these code examples to implement the bit mask based code compression algorithm.

A.1 Bitmask based code compression algorithm

This section describes the bitmask based code compression algorithm. This algorithm takes advantage of the commonly occurring sequences in the program and is similar to the dictionary based coding algorithm. This algorithm goes one step further by considering mismatches and
tries to increase the commonly occurring sequences in a program so that more compression can be achieved. The basic idea is to store the mismatches by remembering a few bit positions in order to increase the number of instruction matches. This algorithm, using bitmasks, significantly improves the compression efficiency and does not introduce much of a decompression penalty. However, the efficiency of this algorithm is limited by the number of bit changes used during compression. It is obvious that, if we consider more bit changes, it will create more instruction matches. But there will be a point where the cost of storing the extra bit positions will offset the advantage of increasing the commonly occurring sequences in the program. The bitmask algorithm incorporates maximum bit changes by using bit mask patterns. For a 32 bit instruction, the mask type may be of 1, 2, 4 or 8 bits. So the extra information which needs to be stored for every mismatch is the mask type, the position where the mask needs to be applied and the mask pattern.

The bitmask-based code compression approach makes the use of bitmasks for storing the bit changes and thus, requires lesser bits to capture the bit changes between two instruction sequences. For example, if there is an 8-bit change in two 32-bit instruction sequences, we can use an 8-bit mask to capture the mismatch. Thus, it will only require an additional (8+5) 13 bits to capture the mismatch with the additional 5 bits to store the start position of the bitmask and 8 bits to indicate the bitmask. The main idea of a bitmask-based approach is to incorporate maximum bit changes such that the compression ratio is significantly improved.

**A.2 Implementation**

This section demonstrates a step-by-step implementation process of the bitmask based code compression algorithm on one of the code examples on TI's MCU MSP430. We will use ADC10_ex1 as our code example. It is the program code for an Analog-to-digital converter and it converts a continuous quantity into its digital form.
We will be implementing the bitmask-based code compression algorithm on Texas Instrument’s 16-bit MSP430 microcontroller [12]. The features of the microcontroller are listed below:

- The MSP430 is a 16-bit microcontroller which is used for a wide range of portable and low-power applications.
- The MSP430 is basically designed for low cost and low power consumption embedded applications.
- The MSP430 can use six different low-power modes in which unneeded clocks and CPU are disabled to reduce the power consumption.
- MSP430 MCUs have an excellent mix of intelligent peripherals, low cost and lowest power consumption for many applications.
- The key applications of MSP430 MCUs include metering devices, portable medical devices, data logging and wireless communications, capacitive touch sense IOs, energy harvesting, motor control and security and safety devices.

**Workspace**

- Code Composer Studio creates a “workspace”, that is a folder where projects are stored.
- The default directory is:
  
  Desktop -> My Documents -> Workspace.

**Create new folder in workspace**

- Using Windows Explorer, create a new folder
Naming the new Project

- Name this folder MyFirstProject
Figure A.2 Naming the new project

Copy the files

- Copy the project files to MyFirstProject folder
Start code composer studio

- Start code composer studio
Creating a new project

- Select File -> New -> CCS Project
Figure A.5 Creating a new project

**Naming the new project**

- For project name, use MyFirstProject.
- Click Next >
Figure A.6 Naming the new project

**Project types available**

- Default is ARM. Select Project Type MSP430 from the list.
- Click on Next >

**Additional project settings**

- No references to other projects. Click Next >
Select MSP430F5515

- From the pull-down menu, select MSP430F5515

Active project

- MyFirstProject is the active project for Debug.
Show files in MyFirstProject

- Click on + to expand the list of files.
Figure A.9 Show files in MyFirstProject

View main.c

- Double click on main.c to open the main.c file.
Select project properties

- Select Project -> Properties
Figure A.11 Select project properties

Properties for MyFirstProject
Figure A.12 Properties for MyFirstProject

C/C++ build – basic settings

- Double clock on C/C++ build
Figure A.13 C/C++ build – basic settings

Basic options

- Click on basic settings
Figure A.14 Basic options

Setup silicon version

- Type MSP430 in for (--silicon_version, -v)

Runtime model options

- Double click on Runtime model options
A.15 Runtime model options

Setting the pointer size

- Set `(--ptrdiff_size) to 16 (default)`

Selecting the memory model

- It is important to use the “small” memory model
Apply predefined step

- Choose the option as TI-TXT
- This TI-TXT format is the binary format for the compiled COFF file which is created as the output file after the sample code program is compiled.

Run the Code for bitmask based code compression algorithm

- The TI-TXT file is in hex format and hence we run the below programs to convert this file in a binary format which can be fed as an input to the bitmask based code compression algorithm. The bitmask based code compression algorithm is then run and the compressed file is formed.
  - clearfiles.cpp
  - hex1.cpp
  - hex2.cpp
Compute the compression ratio

- Compute the compression ratio for each sample code program as below:

\[
\text{Compression Ratio (C.R)} = \frac{\text{Compressed Program Size}}{\text{Original Program Size}}
\]  

(2.2) [1]
Exercise questions:

1. Implement the bitmask based code compression algorithm on target processor TMS320C5x which has a 32-bit instruction set. Modify the code compression algorithm to compress a 32-bit input vector and apply 2, 4, 8 and 16-bit mask patterns and compare the compression ratio in each case.

2. Implement the bitmask based code compression algorithm on target processor ARM. Apply 2, 4, 8 and 16-bit mask patterns and compare the compression ratio in each case.

3. Modify the bitmask based code compression algorithm to apply a combination of 2, 4, 8 or 16-bit mask patterns on each input vector as and when the combination is profitable i.e. minimum number of bits are required to store the mask patterns.

4. Develop an algorithm which can give the best bit mask combination for every input vector to be compressed and implement the same.