DECODING-AWARE COMPRESSION TECHNIQUES FOR RECONFIGURABLE SYSTEMS

By

CHETAN MURTHY

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

2008

© 2008 Chetan Murthy

To my father, Murthy M.C.; my mother, Shashikala R.; and my brother, Mohan M. for their constant care, love, and encouragement

ACKNOWLEDGMENTS

First, I would like to thank my thesis supervisor Dr. Prabhat Mishra for giving me an opportunity to solve interesting and complex problems, helping me learn new technologies, and recognizing the potential in me to positively contribute in ongoing research in the Embedded Systems Lab. My sincere thanks go to Dr. Ye Xia and Dr. Alireza Entezari for being my thesis committee members, and giving me valuable feedback and constructive comments on my thesis. I would like to acknowledge the contributions of Mr. Xioke Qin in helping me to setup the experimental framework for FPGA-based decompression. I would also like to thank all computer science department faculty for offering advanced courses to augment my knowledge and inspiring me to apply those concepts to solve numerous complex issues.

I extend my profound gratitude to research members in the Embedded System Lab who were there at all times providing a joyous environment, listening to all my problems and assisting in solving them with great ease.

Last but by no means least, I would like to convey my heartfelt thanks to my family who, at all times have been a great source of positive spirit and inspiration, encouraging me to take bold decisions and accomplish them successfully and make them proud with this achievement

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS . . . 4

LIST OF TABLES . . . 7

LIST OF FIGURES . . . 8

ABSTRACT . . . 10

CHAPTER

1 INTRODUCTION . . . 12

1.1 Introduction . . . 12
1.2 Study Contributions . . . 13

2 BACKGROUND . . . 15

2.1 Introduction to FPGA . . . 15
2.1.1 Architecture . . . 15
2.1.2 Configurable Logical Block . . . 15
2.1.3 Reconfiguration . . . 16
2.1.4 Placement of Decompressor Engine . . . 17
2.2 Related Work . . . 19
2.3 Cost Benefit Analysis of Dictionary Based Compression Algorithms . . . 20
2.3.1 Dictionary Based Compression . . . 20
2.3.2 Bitmask Based Dictionary Compression . . . 22
2.3.3 Decoding Engine of Bitmask Encoded Bitstreams . . . 24

3 DECODING AWARE CONFIGURATION BITSTREAM COMPRESSION . . . 26

3.1 Decoding-Aware Bitstream Compression . . . 26
3.1.1 Parameter Selection for Dictionary Based Compression . . . 26
3.1.2 Decoding-Aware Bitmask Based Compression . . . 27
3.1.3 Efficient Dictionary Selection . . . 30
3.1.4 Run Length Encoding of Compressed Words . . . 31
3.2 Efficient Bitstream Decompression . . . 33
3.2.1 Decode Friendly Parameter Selection . . . 34
3.2.2 Decoding-Aware Placement of Compressed Bitstreams . . . 35
3.2.3 Decompression Engine . . . 37
3.3 Experiments . . . 38
3.3.1 Decoding-Aware Parameters for Benchmarks . . . 38
3.3.2 Compression Efficiency . . . 39
3.3.2.1 Decoding aware vs. bitmask based compression . . . 40
3.3.2.2 Decoding-aware vs. bitstream compression techniques . . . 41
3.3.3 Decompression Efficiency . . . 44

4 CONTROL WORD COMPRESSION 47
4.1 Architecture of No Instruction Set Computer 47
4.2 Control Word Compression 48
4.2.1 Bitmask Based Compression using Multiple Dictionaries 49
4.2.2 Bitmask Aware Don't Care Resolution 50
4.2.3 Smart Encoding of Least Frequently Changing Bits 53
4.3 Decompression of Control Words 55
4.3.1 Decompression Engine 55
4.3.2 Branch Target Look Up Table 56
4.4 Experiments 56
5 OPTIMAL REPRESENTATION OF BITMASKS 58
5.1 Optimal Encoding of n Bits 58
5.2 Proof for n-1 Bit Representation 59
5.3 Experiments 60
6 CONCLUSION AND FUTURE WORK 61
6.1 Conclusion 61
6.2 Future Research Directions 62
REFERENCES 63
BIOGRAPHICAL SKETCH 65

LIST OF TABLES

Table page

3-1 Parameter values for best compression ratio 38

3-2 Operating speed and look up table usage of decoders 44

3-3 Decompression cycles for fixed length decoder 45

3-4 Decompression time in milliseconds for FFT benchmark 46

LIST OF FIGURES

Figure		page
1-1	Traditional FPGA reconfiguration with compression	12
2-1	Field programmable gate array architecture	15
2-2	Configurable logic block and a simple function implementation	16
2-3	Reconfiguration architecture of Virtex family	17
2-4	Decompression hardware placement	18
2-5	Compressed word formats of dictionary and bitmask based compression techniques	20
2-6	Dictionary based compression	21
2-7	Decompression engine for bitmask encoded bitstream	25
3-1	Effect of word length, dictionary size and number of bitmasks on compression ratio	28
3-2	Dictionary selection in bitmask-based compression	31
3-3	Run length encoding with bitmask based compression	32
3-4	Decoding aware placement of encoded bits	33
3-5	Decompression engine	37
3-6	Comparison of compression ratio with bit mask based code compression technique	39
3-7	Comparison of compression ratio with LZSS-8 on Dirk et al. benchmarks	41
3-8	Comparison of compression ratio with LZSS-8 on Pan et al. benchmarks	42
3-9	Comparison of compression ratio with difference vector compression technique on Pan et al. benchmarks	43
3-10	Comparison of decompression time for FFT benchmark	46
4-1	No instruction set computer architecture and decoder placement	48
4-2	Bitmask aware don't care resolution	51
4-3	Removal of constant and least frequent bits in control words	53
4-4	Control word compression methodology	54
4-5	Multi-dictionary based decompression engine	55
4-6	Branch target look up table for compressed control words	56

4-7 Comparison of compression ratio with dictionary based compression technique on MiBench benchmark 57

5-1 The n-1 encoding of n bit bitmask 58

5-2 Comparison of compression ratio with and without using n-1 bit encoding scheme 60

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

DECODING-AWARE COMPRESSION TECHNIQUES FOR RECONFIGURABLE SYSTEMS

By

Chetan Murthy

December 2008

Chair: Prabhat Kumar Mishra
Major: Computer Engineering

In recent years FPGAs are widely used in reconfigurable systems. Field programmable gate arrays (FPGAs) are configured using bitstreams often loaded from memory. Configuration data is reaching megabytes because of multiple versions of an IP core are configured on a single FPGA and sometimes because of the multiple IP cores. Limited configuration memory restricts the number of IP core bitstreams that can be stored. Moreover slower memory and limited communication bandwidth limit how frequently IP cores can be configured. One promising direction is to compress these bitstreams. Most of the compression techniques exploit redundancies to compress multiple bitstreams but are not suitable for realtime decompression. Other techniques focus on accelerating decompression but compromises compression efficiency. It is a major challenge to design a compression technique which efficiently reduces bitstream size, meanwhile keeping decompression overhead minimal. Our study proposes a novel bitstream compression technique efficiently combining bitmask and run length encoding for better compression ratio and smart rearrangement of compressed bits for fast decoding. Our study's main contributions are i) decoding aware dictionary selection to increase dictionary coverage, ii) run length encoding of repetitive patterns to reduce compressed size, iii) efficient encoding scheme for storing least frequently changing bits and, iv) smart rearrangement of compressed bits into fixed length words that can significantly decrease the decompression overhead. Hard to compress benchmarks are chosen which are widely used IP cores

from image processing and encryption domain to show the usefulness of this technique. The proposed technique outperforms the compression ratio of existing techniques by 5 to 15% and decompression hardware is capable of operating at 200 MHz, the best known operating speed for FPGA based decompressor. Our study also analyzes the application of an enhanced version of proposed bitstream compression technique to compress no instruction set computer (NISC) control words. Results show an improvement in compression ratio by 15 to 20% without adding any decompression overhead.

CHAPTER 1
INTRODUCTION

1.1 Introduction

Field programmable gated arrays (FPGA) store configuration bitstream on memories which are usually limited in capacity and bandwidth. As FPGA are commonly used in reconfigurable systems and application specific integrated circuits (ASIC), configuration memory becomes a key factor in determining the number of IP cores that a system can be configured and the delay in configuration. The bitstream compression algorithms solve memory constraint issue by reducing the size of the bitstreams, whereas decompression accelerators increases the decoding speed by simple decoding logic. But there are very few algorithms that offers both efficient compression ratio and fast decompression. Figure 1-1 shows the typical flow of compressed FPGA bitstream configuration. Bitstreams generated by vendor specific bit generation programs are compressed and stored on a persistent memory. The decompression hardware decodes and transfers the compressed bits from memory to configuration hardware which is then transferred to configurable logic blocks (CLB) memory.

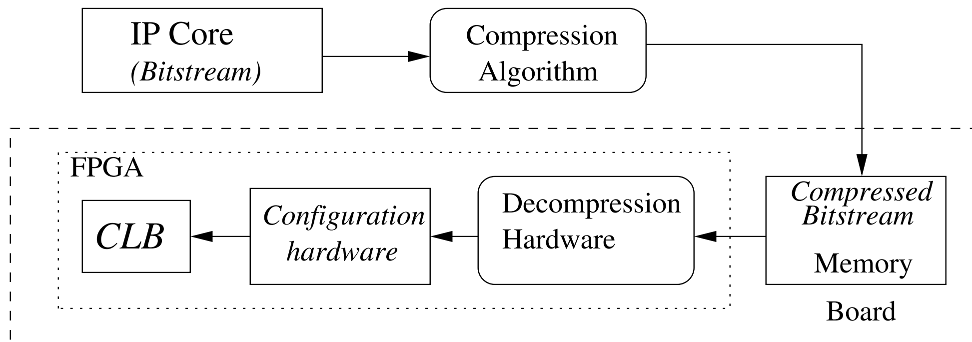

Figure 1-1. Traditional FPGA reconfiguration with compression

Compression ratio (η) is the metric commonly used to measure effectiveness of a compression technique, defined in Equation 1–1.

$$CompressionRatio(\eta) = \frac{CompressedSize}{UncompresssedSize} \tag{1–1}$$

We can classify the existing bitstream compression techniques into two categories: those having good compression ratio but unacceptable decompression overhead and complexity [1] [2] [3] [4], and others which accelerate decompression but compromises compression ratio [5]. The main idea of these algorithms is to store frequently occurring sequence of bits using a static [6] or sliding [5] dictionary or to use FPGA specific features (partial reconfiguration or readback [7]) to obtain repetitive patterns. One of the promising compression technique is bitmask based code compression because of its good compression ratio and simple decompression logic. The direct application of this algorithm is not flexible enough in choosing word length, bitmasks, and dictionary size for FPGA bitstreams. It is obvious that unlimited use of these results in better matches but will result in multiple variable length encodings. However such encodings are not profitable as they result in slower and complex decompression hardware. Hence it is a major challenge to develop an efficient compression technique which significantly reduces the bitstream size without sacrificing decompression performance.

1.2 Study Contributions

Our study makes these major contributions: i) efficiently combining decode aware compression and run length encoding of repetitive patterns, ii) novel encoding of least frequently changing bits, iii) optimal representation of n bit changes by only using $n-1$ bits, and iv) smart rearrangement of compressed bits to obtain fixed length encoded words.

Our study proposes an efficient decoding aware compression technique for compressing FPGA configuration bitstreams to improve compression ratio and decrease decompression overhead. This is accomplished by efficiently choosing decoding aware parameters (word length, dictionary size, and number and type of bitmasks) combined with smart run length encoding of repetitive words. The decompression overhead is reduced by efficiently reorganizing compressed bits to form fixed length words. We have chosen very hard to compress and commonly used IP cores from Opencore repository [8] and benchmarks used

by Koch et al. [9] to demonstrate the effectiveness of our technique. Our experimental results show that our approach improves compression ratio by 5 to 15% over existing bitstream compression techniques and decompression hardware is capable of running at 200 MHz. Furthermore, the decompression time to configure FPGA is decreased by 15 to 20% over decompression accelerator proposed by Koch et al. [5].

Our study also analyzes the application of the above decoding aware compression algorithm to compress no instruction set computer (NISC) control words. The direct application of the compression technique is not beneficial in code size reduction. Whereas using multiple dictionaries results in acceptable reduction in code size comparable to technique proposed by Gorjiara et al. [10]. A novel technique is proposed in which least frequently changing and constant bits are smartly encoded to significantly reduce the code size with no extra decompression overhead. Results show an improvement of 15 to 20% in compression ratio with minimal decompression overhead. The dictionaries stored on block RAM (BRAM) are fixed and limited to maximum of 2. Finally, our study proposes an optimal encoding scheme to represent n bit changes in an input stream by only using $n-1$ bits.

The rest of the thesis is organized as follows: Chapter 2 discusses the FPGA architecture and related frameworks. It also describes the existing bitstream compression techniques to compress FPGA configuration bitstreams. A cost benefit analysis of existing dictionary based compression techniques for compressing configuration bitstreams is also discussed in this chapter. Chapter 3 describes our proposed decode aware compression technique and smart rearrangement of compressed bits to achieve fast decompression. This chapter also describes an efficient technique to combine bitmask based compression with run length encoding of repetitive words in a bitstream. Chapter 4 analyzes the application of an enhanced version of proposed technique to compress NISC control words. Chapter 5 describes an optimal representation of n bit changes using only $n-1$ bits. Finally, Chapter 6 concludes the thesis exploring possible future research directions.

CHAPTER 2
BACKGROUND

2.1 Introduction to FPGA

This section describes the FPGA architecture, the process of configuring FPGA with configuration bitstream and the effect of decoder placement on configuration time.

2.1.1 Architecture

The Xilinx Virtex FPGA comprises of two types of configurable elements: configurable logic blocks (CLB) and input/output blocks (IOB). CLB provides the basic functional and logical elements for IP implementation and IOB provides the interface to connect CLB and package pins as shown in Figure 2-1. Each CLB contains multiple logic cells (LC) which implements 4 input function generator and equipped with carry forward logic (switching fabric). Each LC contains a look up table (LUT) stored on static RAM (SRAM) cells which holds the function generator implementation. Each LC also contains storage elements which can be configured as flip flops or latches. Surrounding each CLB are large blocks of vertically arranged block select RAM (BRAM) which can be used as memory storage elements.

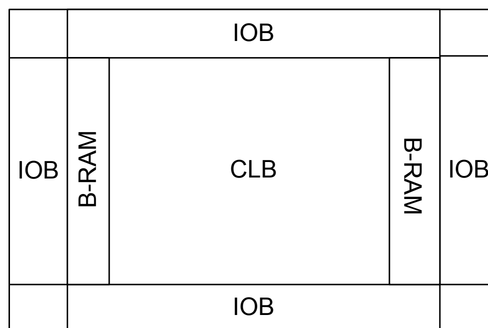

Figure 2-1. Field programmable gate array architecture

2.1.2 Configurable Logical Block

The configurable logic block (CLB) forms the basic reconfigurable fabric element which can be programmed to function as a hardware. Depending on vendor each CLB consists of multiple logic cells. Each LC consists of four input LUT based function

implementation as shown in Figure 2-2(A). The function output values (16 values) are stored on volatile SRAM cells. The output of this function is then routed to subsequent LC via switching fabric. The output is also saved in a latch next to LUT module.

A simple function *p*0 implemented is shown in Figure 2-2(B). It must be noted that configuration bitstreams contains both CLB's function implementation (SRAM) content and the routing information.

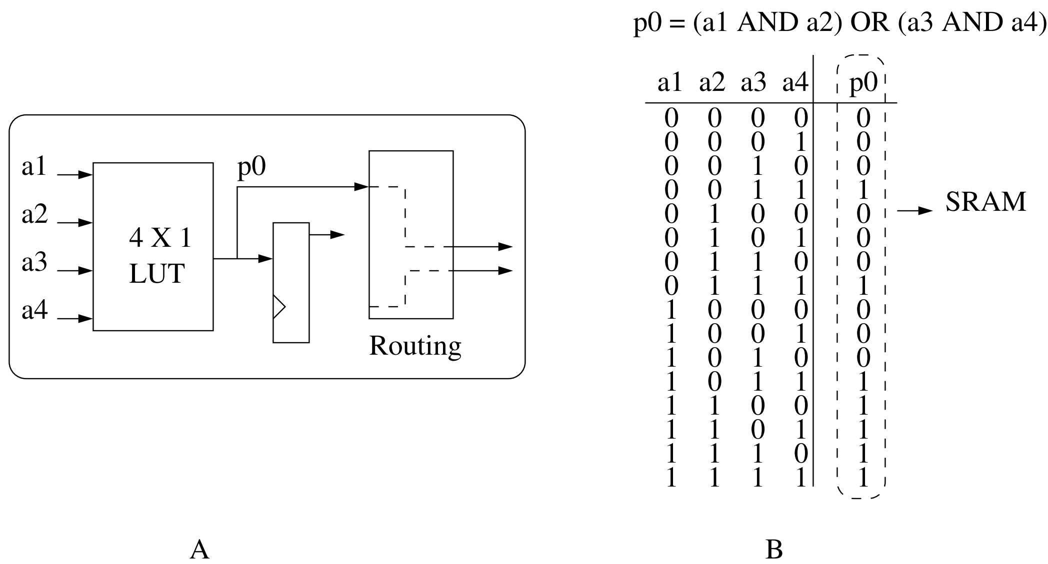

Figure 2-2. Configurable logic block and a simple function implementation. A) Logic Cell inside a CLB, B) Function generator for function p0

2.1.3 Reconfiguration

These are the steps followed to program or configure an IP core using Xilinx Virtex FPGA [11].

1. **Design entry**: The logic of the IP core is written in any hardware description language (HDL) is converted to a net list representation.

2. **Implementation**: The net list representation is then converted to bitstreams or other supported formats by a vendor specific bit generator program. The size of the bitstream is dependent on the type of FPGA family and vendor on which the IP core will be loaded.

3. **Configuration or programming**: In this step the configuration bitstream is stored directly on to FPGA memory (SRAM) or on an external flash based EEPROM. The

configuration bitstream stored on external memory is transferred to SRAM using complex programmable logic device (CPLD) or using an onboard microprocessor.

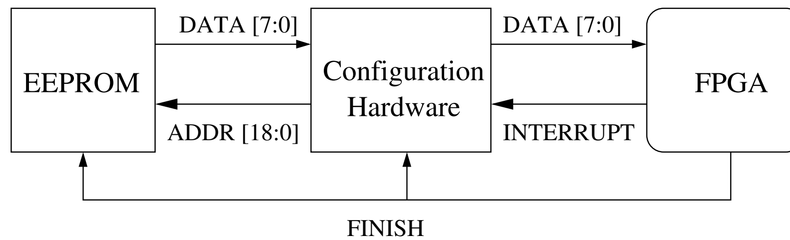

Figure 2-3. Reconfiguration architecture of Virtex family

Figure 2-3 describes the process of configuring FPGA using EEPROM and configuration hardware which controls the process of loading bitstream from memory to FPGA. The bitstream generated from vendor specific tool is transferred onto EEPROM through a serial or JTAG interface. The saved bitstream is then loaded on FPGA's SRAM to function as a hardware. In most of Xilinx Virtex family, FPGA have a 8 bit communication bus from configuration hardware to FPGA. This is potentially a bottleneck while transferring large designs from EEPROM to FPGA. A promising approach is to transmit compressed bitstreams to alleviate this bottleneck.

2.1.4 Placement of Decompressor Engine

The placement of decompression engine plays a crucial role in determining the decompression speed and time required to configure FPGA. Placing decompression engine near configuration hardware is the most promising approach. This placement increases the communication bandwidth and bridges the operational speed difference between FPGA and external memory. Figure 2-4 describes four possible ways of placing the decompressor in a FPGA system. These are described as follows:

a. **Custom hardware decompressor:** Figure 2-4(A) shows the placement in which decompressor is a part of configuration hardware. This is an efficient placement but also the costliest implementation. This placement requires ASIC design and replacement of existing hardware.

b. **FPGA based decompressor:** The placement shown in Figure 2-4(B) requires decompressor to have direct access to SRAM memory in order to configure FPGA.

Such functionality is provided as frame data input register (FDRI) in Xilinx Virtex family.

c. **FPGA master-slave mode:** In this placement the decompressor resides on a master FPGA as shown in Figure 2-4(C). The compressed bitstream is first transferred to master, decompressed and transmitted onto slave FPGA for configuration [7].

d. **Processor based decompressor:** This placement is feasible on some of the FPGA that has onboard processor as show in Figure 2-4(D). The processor executes the software implementation of the decompressor and transmits the decompressed data to configure FPGA. The disadvantage of such an implementation is that it does not solve the communication bottleneck and can potentially slow down the configuration process.

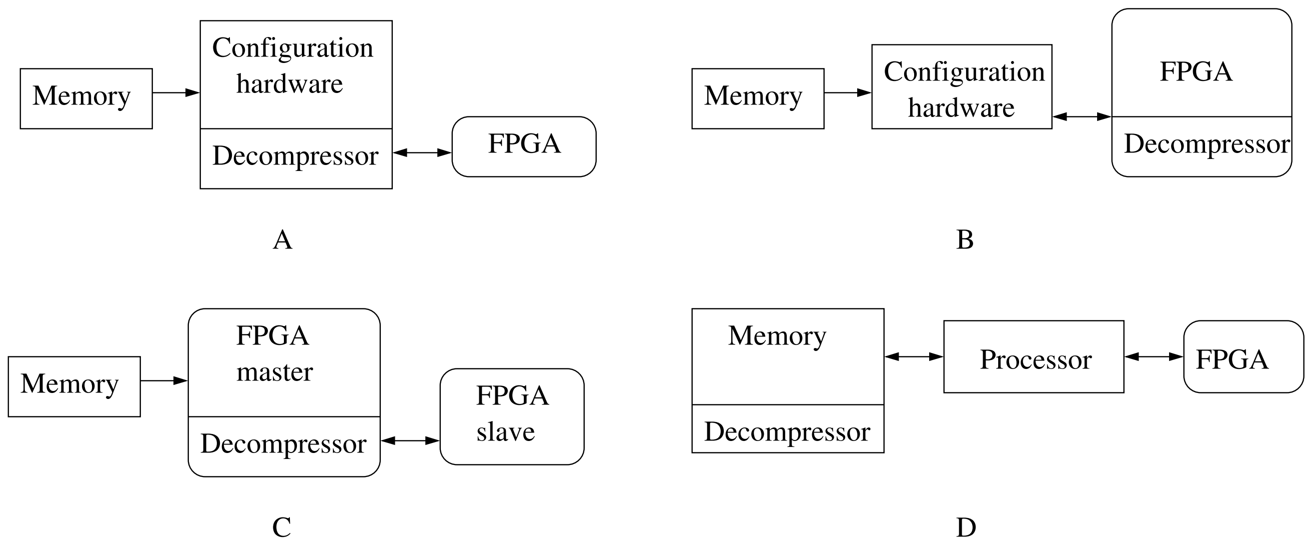

Figure 2-4. Decompression hardware placement. A) custom hardware decompressor, B) FPGA based decompressor, C) FPGA Master Slave Decompressor, D) processor based decompressor

From the above possible placements we find decompression engine placed within FPGA (Figure 2-4(B)) as the most feasible and cost effective placement. This placement reduces the communication bottleneck by transferring compressed bitstream to FPGA first and then decompressing just before configuration. Another advantage of this placement is that the decoder can run at FPGA operating speed.

2.2 Related Work

There are numerous compression algorithms that can be used to compress configuration bitstreams. These techniques can be classified into two categories based on how the redundancies are exploited: format specific compression and generic bitstream compression. The compression techniques in the first category exploit the local redundancies in a single or multiple bitstreams by reading back the configured data and storing the differences by performing exclusive-OR (XOR) operation. These algorithms requires FPGA to support partial reconfiguration and frame readback functionality. Pan et al. [1] uses frame reordering in the absence of readback facility on FPGA. In this technique frames are reordered such that the similarity between subsequent frames configured is maximum. The difference between consecutive frames (difference vector) is then encoded using either Huffman based run length encoding or LZSS based compression. Another method proposed in the same article organizes and reads back the configured frames. The frames are organized such that compressed bitstream contains minimal number of difference vectors and maximal readback of configured frames thus reducing the compressed frames significantly. Such complex encoding schemes tend to produce excellent compression ratio. However, decompression is a major bottleneck and is not addressed by Pan et al. [1].

The generic bitstream compression techniques uses complete bitstream to extract the redundancies within small window (usually 32 bytes) and encode the information. An advantage of these techniques is that no special FPGA support is required for decompression. Parameterized LZSS [5] chooses efficient parameters suitable for bitstream compression and decompression. The compression focuses on the most repeating lengths in the matched strings to encode partial set of those lengths using less number of bits and the rest is encoded using canonical representation. The decompression hardware is fairly simple and is able to decode at acceptable speed. LZ77 algorithm proposed in [12] also works similarly by matching the redundant symbols in a small window.

In summary, the compression technique in [1] achieves significant compression but incurs drastic decompression overhead. On the other hand, the approaches in second category [5] [7] try to maintain decompression overhead in an acceptable range but compromises on compression efficiency. Our technique tries to consider decompression overhead during the compression of bitstreams. The compression parameters are chosen such that compressed bitstreams are decode friendly while maintaining a good compression ratio.

2.3 Cost Benefit Analysis of Dictionary Based Compression Algorithms

In this section, we briefly describe two dictionary based compression techniques to evaluate their suitability in compressing configuration bitstreams. These techniques are very promising for their simple and fast decompression but have never been used for configuration bitstream compression. This analysis forms the basis of our approach.

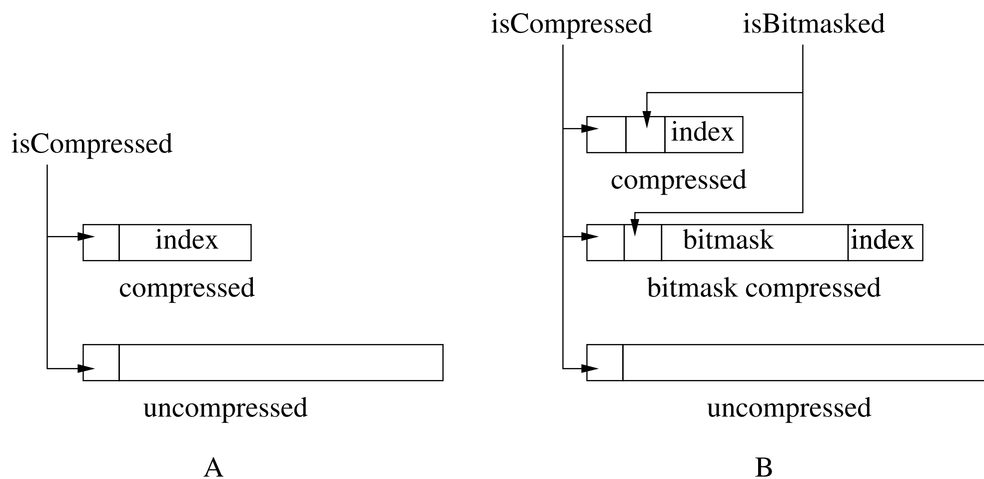

Figure 2-5. Compressed word formats of dictionary and bitmask based compression techniques. A) Dictionary compressed word format B) Bitmask compressed word format

2.3.1 Dictionary Based Compression

In a traditional dictionary based compression algorithm, the input data with length N bits is divided into n words of each length w bits i.e, $N = n \times w$. A list of all unique words are then sorted in descending order of their occurrences. The most frequently occurring words are stored in a dictionary. The input data is then encoded using indices to

words stored in a dictionary or a part of input words stored in restricted dictionary. The uncompressed words are indicated by compressed flag (1 bit) as shown in Figure 2-5(A). The efficiency of this algorithm depends on selected parameters: word length (w) and the number of dictionary entries (d). A wider word length (w) results in lesser matches resulting in larger dictionary size and a large compressed code. However, an advantage is that there are less number of words (n) to compress. A smaller word length results in smaller dictionary size with many words to compress. Here, the advantage is that the probability of obtaining redundant words is much higher in this case.

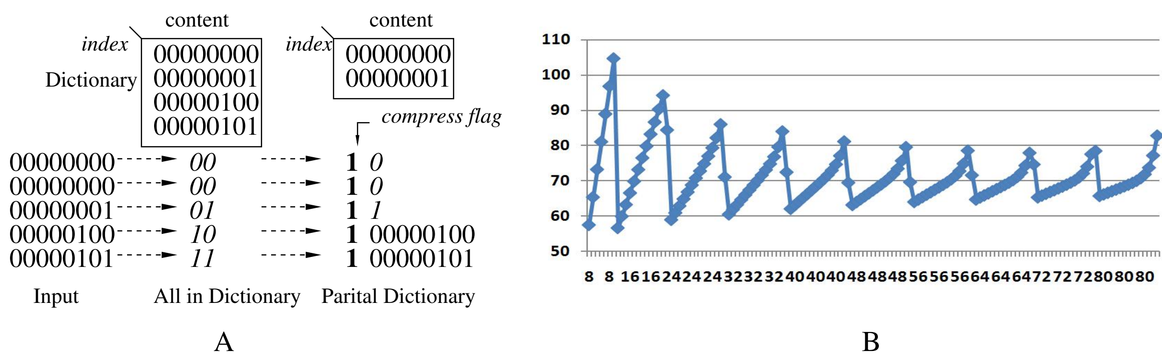

Figure 2-6. Dictionary based compression. A) Dictionary based compression B) Compression ratio histogram

Figure 2-6(A) shows a simple dictionary based compression with all words in dictionary or a partial set of words in a limited dictionary. To obtain optimum compression ratio an intuitive approach would be to evaluate all possible values of word length (w) and dictionary size (d). Such an algorithm takes exponential time to reach efficient parameter combination. Figure 2-6(B) shows the compression ratio of a sample benchmark (FFT) with varying word lengths (w) and variable dictionary size (d). The x axis represents different word length and y axis shows the compression ratio achieved. The spikes in the graph represent a large dictionary size with wide index bits resulting in degraded compression ratio.

$$\eta_{complete} = \frac{w * d_{all} + n * \lceil log_2(d_{all}) \rceil}{n * w} \quad (2\text{–}1)$$

$$\eta_{partial} = \frac{(w * d) + (1 + \lceil log_2(d) \rceil) * n_m + (1 + w) * (n - n_m)}{n * w} \quad (2\text{–}2)$$

Section 3.1.1 discusses a linear time algorithm to select efficient parameters which yields better compression ratio. The compression ratio achieved when all the words are in dictionary (d_{all}) is given by Equation 2–1. In a generic case when part of input words are stored in dictionary the compression ratio is given by Equation 2–2. Here $n * w$ is the input size, $w * d$ is the dictionary size used, $\lceil log_2(d) \rceil$ is the compressed dictionary index bits, n_m is the number of entries matched with dictionary and $(n - n_m)$ is the number of uncompressed words. The algorithm reaches worst case compression ratio when number of matched words (n_m) is equal to dictionary size (d) i.e. each dictionary entry is used exactly once. An efficient algorithm should choose one of these options with appropriate parameter values for word length (w) and dictionary size (d) to obtain the best compression ratio.

2.3.2 Bitmask Based Dictionary Compression

The bitmask based compression proposed by Seong et al. [13] is a promising technique that has better compression ratio over dictionary based compression technique. This technique uses extra bits to record the differences in unmatched words from the dictionary entries and encode them as usual dictionary indices. This technique uses limited dictionary to match many words with small bit changes. The different compressed word format is depicted in Figure 2-5(B). This technique uses b number of bitmasks, $B = \{B_1, B_2, .., B_b\}$. Each bitmask B_i is defined as $< s_i, t_i, o_i >$ where i) s_i is size of the bitmask, ii) t_i is the type of bitmask ($FIXED$: for fixed location encoding or $SLIDING$: for sliding bitmask) which can encode differences occurring at fixed or any location within the word and, iii) o_i bits to encode offset within the word ($\lceil log_2(w) \rceil$ bits for sliding bitmask and $\lceil log_2(w)/s_i \rceil$ bits for fixed location bitmask).

$$\eta_{lower} = \frac{dict_{size} + match_{size} + bitmasked_{size}}{n * w} \tag{2–3}$$

During compression, each word is either compressed with full match encoded as $['0', '1', \lceil log_2(d) \rceil]$ or compressed using b bitmasks encoded as $['0', '0', \lceil log_2(b-1) \rceil, [\sum_{i=0}^{b} (o_i,$

$s_i)], log_2(d)]$ or uncompressed word encoded as $['1', w]$. Clearly the best compression ratio (lower bound) that we can obtain is with minimum dictionary entries (d_{min}) we can match all the words using direct match (n_m words) or using one bitmask ($n - n_m$ words). Then compression ratio n_{lower} is given by Equation 2–3. Here, $dict_{size}$ is the total dictionary size ($d_{min}*w$), $match_{size}$ is the total fully matched compressed words size $(2+\lceil log_2(d_{min})\rceil)*n_m$, and $bitmaksed_{size}$ is total bitmasked words size $((2 + s_0 + l_0) * (n - n_m))$. An important point to note is that for the same input, dictionary size (d_{min}) for bitmask based technique will be much smaller than generic dictionary based compression. The worst compression ratio (upper bound) possible is when there are no matches with dictionary and none of the words can be bitmasked with any of the dictionary entry then the compression ratio is given by Equation 2–4. Here, $d * w$ is the dictionary size, $(2 + \lceil log_2(d)\rceil) * d$ is the size of entries matched with dictionary, and $(1 + w) * (n - d)$ is the size of uncompressed size.

$$\eta_{upper} = \frac{d * w + (2 + \lceil log_2(d)\rceil) * d + (1 + w) * (n - d)}{n * w} \tag{2–4}$$

On an average if there are n_m entries matched with dictionary, n_b entries matched with one or more bitmasks and n_u entries uncompressed then the compression ratio is given by Equation 2–5. where

1. $dict_{size}$ is the total dictionary size $(d * w)$
2. $match_{size}$ is the total size of fully matched words $(2 + log_2(d)) * n_m$
3. $bitmasked_{size}$ is the total size of bitmasked words $(2 + log_2(b) + \sum(s_i + log_2(w))) * n_b$
4. $uncompressed_{size}$ is the total uncompressed code $(1 + w) * n_u$

$$\eta = \frac{dict_{size} + match_{size} + bitmasked_{size} + uncompressed_{size}}{n * w} \tag{2–5}$$

The efficiency of a compression algorithm which is determined by choosing appropriate word length (w) and dictionary size (d) is also dependent on the number and type of bitmasks selected. A large number of bitmasks results in smaller dictionary size but at

the same time need more bits to encode bitmasks. Less number of bitmasks, results in large dictionary size but at the same time needs less bits to encode bitmask compressed words. Hence it is challenging to design a compression technique that selects the efficient parameter combination (w, d, b and B) for compression that results in best compression efficiency close to η_{lower}.

2.3.3 Decoding Engine of Bitmask Encoded Bitstreams

The compressed bitstreams generated by dictionary based techniques are usually not aligned to byte boundary and generally do not have fixed length distance between any two compressed words. For example Figure 2-7(A) shows the compressed words and Figure 2-7(B) shows the compressed words arranged in byte boundary. The parameters used for this example are word length $w = 16$, dictionary size $d = 16$, number of bitmask $b = 1$ and type of bitmask used is $B_1 =< 2, SLIDING, 4 >$. To decompress, the decoder needs to check each bit before reading in the next required compressed word.

The decompression engine shown in Figure 2-7(C) describes a decoder with control unit reading variable length of data from input buffer and based on compressed flag retrieves the subsequent variable length data (uncompressed/ dictionary matched/ bitmask compressed). It is evident that this approach results in slower and serial decompressor leading to unacceptable decompression overhead. Hence, it is a major challenge to find a compression algorithm that produces compressed output which can be aligned in byte boundaries to decompress at a faster rate while keeping the compression efficiency close to optimal compression ratio (η_{lower}).

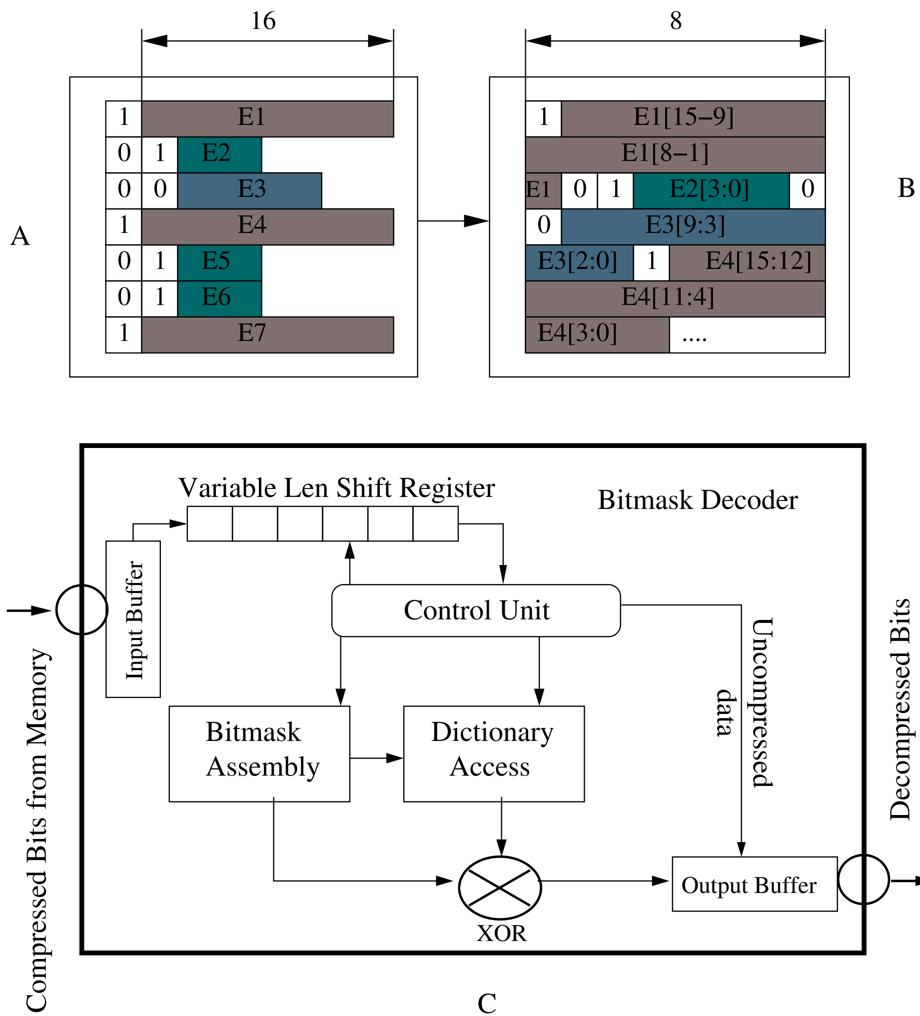

Figure 2-7. Decompression engine for bitmask encoded bitstream. A) Compressed words B) Compressed words in memory C) Bitmask decoder

CHAPTER 3
DECODING AWARE CONFIGURATION BITSTREAM COMPRESSION

3.1 Decoding-Aware Bitstream Compression

We first begin with a technique to choose efficient parameters for generic dictionary based compression. Next we propose a decoding-aware bitmask based compression to select efficient parameters. An efficient parameter based dictionary selection is designed to obtain better dictionary coverage. Later we propose a run length encoding scheme to intelligently encode repetitive compressed words to improve compression and decompression performance. Finally we show how compressed bits are transformed to fixed length encoded bytes for faster decompression.

Algorithm 1: Dictionary Based Parameter Selection

Input: Input Bitstream I
Output: parameters P
$P = \phi$
forall w *in* $\{8, 16, 24, 32, ..8 * k\}$ **do**
  $calc_frequencies(I, w)$
  forall d *in* $\{1, 2, 4, .., 2^{w-1}\}$ **do**
    calculate η using Equation 2–2
    if η *is minimum* **then**
      $P = \{w, d\}$
  end
end
return P

3.1.1 Parameter Selection for Dictionary Based Compression

To improve compression ratio using partial or full dictionary we need to choose suitable parameters: word length (w) and number of dictionary entries (d). Algorithm 1 illustrates parameter selection that yields efficient compression ratio. Since memory and communication bus are designed in multiple of byte size (8 bits), storing dictionaries or transmitting data other than multiple of byte size results in under utilization of memory and communication bus lines. This limits the search space for word length (w) within multiples of 8 up to k iterations. Now with this selected word length, we can easily evaluate the dictionary sizes which yields the best compression ratio. Dictionary size

dictates the size of the index bits. For the word to be compressed, it is evident that these index bits have to be at least one bit less than the word length (w) itself. Thus we can find the efficient dictionary size for a given word length (w) by incrementally changing the index bits from 1 to ($w-1$). In other words dictionary size ranges from 1 to 2^{w-1}. With these parameters the algorithm now calculates the compression ratio by using the Equation 2–2. The number of matched words (n_m) can be determined by sorting the unique words in descending order of their occurrences. The cumulative sum until i^{th} word provides the number of matched words from 1 to i entries in the dictionary. Our experiments found that choosing word length (w) more than 80 bits results in less redundant input. This limits the maximum iterations (k) to choose word length limit to 10.

Algorithm 2: Decode Aware Compression

Input: Input Bitstream I
Output: parameters P
$P = \phi$ **forall** w *in* $\{8, 16, 24, 32, .., 8 * k\}$ **do**
  $W = calc_frequencies(I, w)$
  forall d *in* $\{1, 2, 4, .., 2^{w-1}\}$ **do**
    $B = get_possible_bitmasks(w, d)$
    forall b *in* B **do**
      $generate_dictionary(W, \{w, d, b, B\})$
      calculate η_i using Equation 2–5
    end
  end
  $P = min\{\eta_i\}$
end
return P

3.1.2 Decoding-Aware Bitmask Based Compression

In bitmask based compression method, efficiency is not only determined by word length (w) and dictionary size (d), but also by the number of bitmasks (b) and type of each bitmask t_i used. From Equation 2–5 it is evident that more the number of bitmasks used smaller dictionary size is sufficient. This requires less bits to index the dictionary but to store these bitmasks we need large offset and difference bits. The entries in the

dictionary selected determines the effectiveness of matching uncompressed words with less differences based on proximity of the bit differences that an entry in the dictionary can match. The application specific bitmask compression method proposed in [13] suggests feasible bitmask size and type of bitmask, and a graph based dictionary selection algorithm for better compression ratio. The direct application of this algorithm results in compressed code that is complex and variable length as described in Figure 2-7(B). Section 3.2.2 discusses the type of bitmasks that can be used such that compressed code can be converted to fixed length compressed words without significantly sacrificing the compression efficiency.

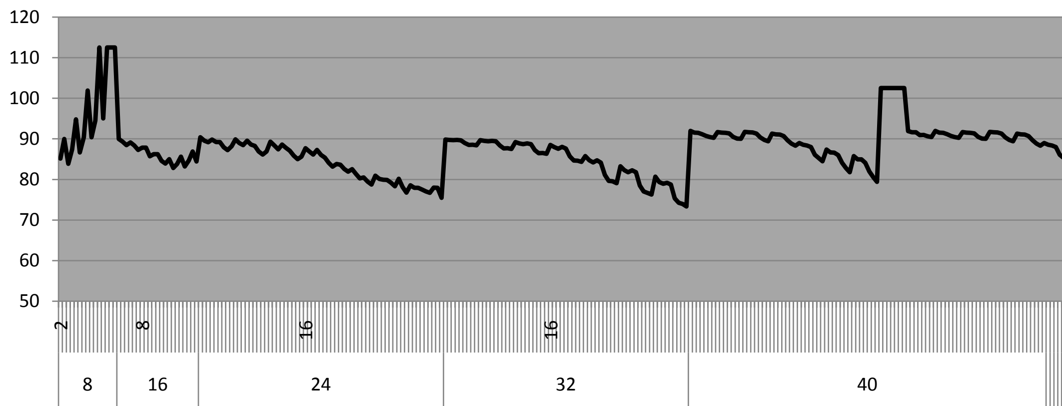

Figure 3-1. Effect of word length, dictionary size and number of bitmasks on compression ratio

Algorithm 2 describes our decode aware parameter (word length w, dictionary size d, number of bitmasks b, size and type of each bitmask $\{s_i, t_i\}$) selection. The range of word length (w) and dictionary size (d) remains the same as in Algorithm 1. A list of bitmask combination is proposed based on its feasibility to align in a fixed byte boundary as discussed in Section 3.2.2. An efficient dictionary selection proposed in Section 3.1.3 is used to select dictionary which covers most of the words using minimal number of bitmasks. The compression ratio is calculated using formula 2–5. The parameter combination which results in minimal compression ratio is used during final compression.

Figure 3-1 shows the compression ratio obtained by applying Algorithm 2 on RSAXCV100 benchmark. The compression ratio obtained is dependent on the input data's entropy. A high entropy input requires larger dictionary and wider bitmasks to obtain better compression efficiency. It can be noted that as word length increases the compression ratio reaches 90% (higher the value lesser the bitstream is compressed). This is because wider words results in less redundancy and dictionary chosen covers less number of words. The effect of increasing dictionary size also improves the compression ratio only to a certain point. Any increase in dictionary size further increases the compression ratio because of the larger index bits used to access the dictionary. An increase in the number and type of bitmask for a given word length and dictionary size improves with lesser number of bitmasks depending on word length selected (one bitmask for 16 bit words, two bitmasks for 32 bit words). To obtain the range of parameters for a new benchmark we need to run the proposed algorithm with all possible values (with word length ranging up to 80 bits as discussed earlier).

Algorithm 3: Efficient Dictionary Selection

Input: input words $W = \{w_i, f_i\}$ where w_i:i^{th} word with frequency f_i
Parameters $P = \{w, d, b, B\}$ as defined in Section 2.3
Output: Dictionary D
$D = \phi$
Construct a graph $G = \{V, E\}$
such that $V = f_i$,
$E(u, v)$ = word u can be bitmasked with v using at most all Bitmasks
$C(E)$ = cost of edge is number of bitmask used
while *($|D|$ is d or $|G|$ is ϕ)* **do**
    find v_{max} of all $v_{savings}$ in V such that
    $v_{savings} = \{f_u * saving_made[0] + ((f_v) * saving_bitmask[C(u,v)]) - w\}$
    remove v_{max} from G and edges incident upon it.
    $D \leftarrow v_{max}$
    forall *(u, w) adjacent nodes of v_{max}* **do**
        if *$num_bitmask(v_{max}, u)$ eq $num_bitmask(v_{max}, w)$ && is geq $num_bitmask(u, w)$* **then**
            remove edges among neighbors (u, w) of v_{max}
    end
end
return D

3.1.3 Efficient Dictionary Selection

Our dictionary selection approach is motivated by application specific bitmask based code compression proposed in [13]. The dictionary is selected for given parameters: word length (w), dictionary size (d), number of bitmasks (b) and size and type of each bitmask (B). Algorithm 3 describes how dictionary is selected based on the savings made by each uniquely occurring word. The dictionary selection is primarily governed by a word's capability to match other word using minimal number of bit masks and covers most of the input words. The input is divided into unique words with each word associated with frequency (f_i). A graph (G) is created in which each vertex represents word with frequencies as its weight. Two vertices are connected via an edge if the two words represented by them can be bitmasked using at most all the bitmasks in B. Each edge (u, v) has the number of bitmasks used to match vertex u and vertex v as its weight. The savings made for each vertex is calculated based on the sum of savings made by itself in the dictionary and savings made by bitmask matching with other vertices indicated by the edges incident on it.

$$savings_made[i] = (1 + w) - \lceil log_2(d) \rceil - \sum_{j=0}^{i}(s_j + l_j) \tag{3–1}$$

Equation 3–1 is used to calculate the savings made (*savings_made*) by each vertex u using i bitmasks. The *savings_made* is an array which holds the savings for different number of bitmasks (from $0, 1, 2,$ to b). This array is then used to calculate the total savings of vertex u. The final savings of a vertex is simply the sum of all the savings of incident vertices including itself, calculated using Equation 3–1 indexed by weight on each edge. Note that $savings_made[0]$ indicates savings using no bitmask or direct indexing. A winner vertex with maximum savings is selected and inserted in the dictionary. All incident edges are removed from the graph (G). To avoid savings conflict among multiple vertices. The edges between the adjacent vertices of winner vertex are also removed if the current saving with the winner vertex is more beneficial than the edge between them.

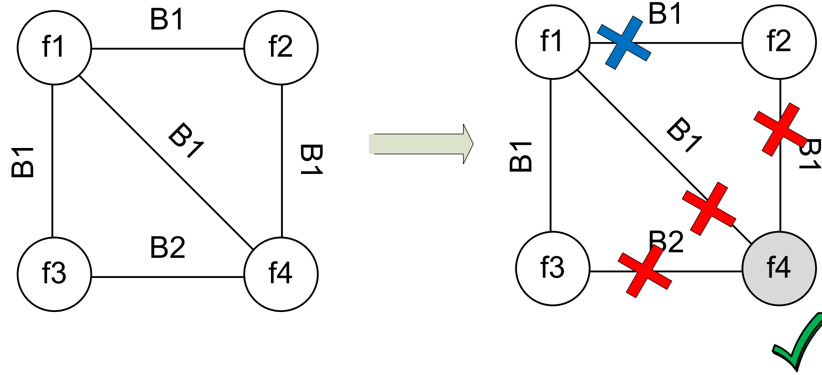

Figure 3-2. Dictionary selection in bitmask-based compression

Figure 3-2 demonstrates an iteration of dictionary selection. Let $f1$, $f2$, $f3$ and $f4$ be the frequencies of the four most frequently occurring elements and $B1$ (Bitmask 1) and $B2$ (Bitmask 2) be the number of bitmasks used for matching. The total savings made by each vertex (u) is calculated by the product of frequency and savings made by each edge ($f_u * savings_made_u$). Then a winner with highest savings is selected. Suppose f_4 is the winner then all the incident edges are removed from the graph. Note that once the winner f_4 is selected the incident edge between vertex f_1 and f_2 is also removed because $f1$ is already covered by $f4$ using $B1$ bits. This ensures that savings are not claimed by multiple vertices which are already in the dictionary. This maximizes the total savings made by the selected dictionary.

The dictionary selection technique proposed by Seong et al. [13] heuristically removes adjacent vertices that have arbitrary threshold incident edges on it along with the winner vertex. The idea behind this is to reduce the dictionary size selected (thus index bits). Our proposed algorithm eliminates this heuristics by providing a fixed dictionary size. The dictionary selected covers maximum words directly or using minimal bitmasks thus ensuring better dictionary coverage.

3.1.4 Run Length Encoding of Compressed Words

Careful analysis of the bitstream pattern reveals that the input bitstream contains consecutive repeating patterns of words. The algorithm proposed in previous section

encodes such patterns using same repeated compressed words. Instead we use a method in which repetition of such words are run length encoded (RLE). Such repetition encoding results in an improvement in compression performance by around 10 to 15% using the benchmarks of Koch et al. [9]. To represent such encoding no extra bits are needed. This is due to the observation that bitmask value 0 is never used, because this value means that it is an exact match and would have encoded using zero bitmasks. Using this as a special marker, these repetitions can be encoded. This smart encoding reduces the extra bit that is required to indicate RLE.

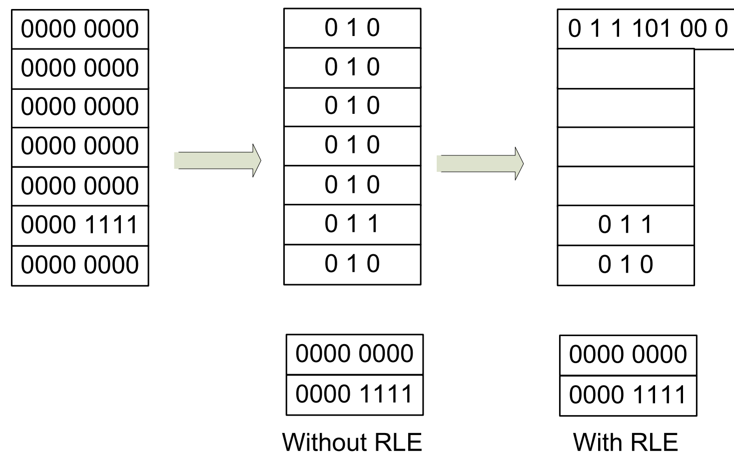

Figure 3-3. Run length encoding with bitmask based compression

Another advantage of such run length encoding is that it alleviates the decompression overhead by providing the decompressed word instantaneously to the decoder to send it to the configuration hardware in the same cycle. This ensures the full utilization of the configuration hardware bandwidth and reduces the bottleneck of communication channel between memory and decoder. Figure 3-3 describes the RLE-based compression. The compressed words are run length encoded only if the savings made by RLE word encoding is greater than the actual encoding. That is if there are r repetition of compressed words and cost of representing each word is x bits and the number of bits required to encode run length is y bits then RLE is used only if $x * r < y$ bits.

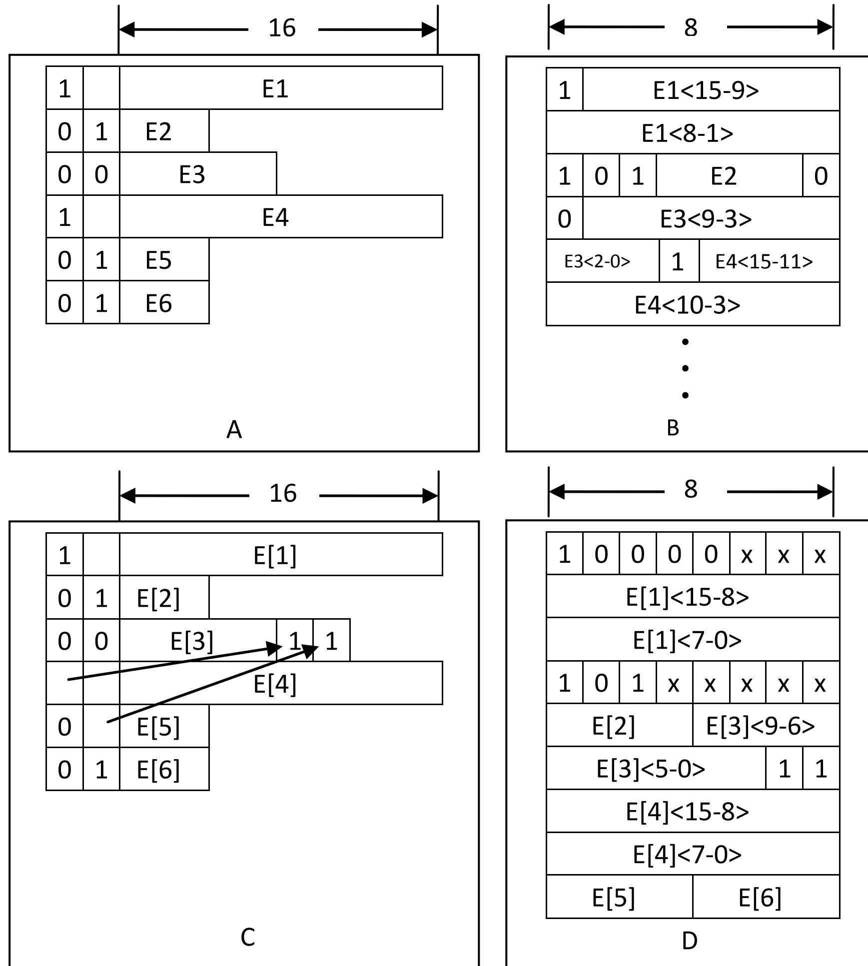

Figure 3-4. Decoding aware placement of encoded bits. A) Variable length bitmask compressed words B) Bitmask compressed words aligned in a byte boundary C) Relocation of compress and match flags to a bitmask compressed word D) Fixed length encoded compressed words

3.2 Efficient Bitstream Decompression

Our work in this direction is motivated by previous bitstream compression framework for high speed FPGA [5] [14]. Generally, when variable length coding approaches are used to improve the compression ratio, they also set two obstacles for the design of high speed decompression engines. For example, Figure 3-4(A) gives a sample output of the bitstream compression algorithm. Figure 3-4(B) is its placement in a 8 bit-width memory using a naive placement method. It can be easily seen that: i) the start position of the next compressed entry usually cannot be determined unless we decode the previous entry;

ii) the input buffer within the decompression engine must be shifted for a variable length within each cycle. Both of them have a negative impact on the length of the critical path within the decompression engine, and therefore limit the maximum operational speed. The LZSS decompression technique in Koch et al. [5] uses one interesting way to attack this problem: place the encoded bits in a way that they can be treated as fixed length encoding. In other words, the encoded bits should have two properties: i) the start position of each compressed entry should be easily identifiable. ii) the number of possible shift length of input buffer should be as small as possible. These lead to our approach for high speed decompression of variable length coding. The following subsections give a detailed description on parameter selection which leads to smart rearrangement and how variable length compressed words are transformed to fixed length compressed bitstreams.

Algorithm 4: Decode Aware Parameter Selection

```
Input: w - word length, d - number of dictionary entries
Output: Bitmasks B
B = φ, indexBits = ⌈log2(d)⌉
if indexBits OR indexBits + 1 OR indexBits + 2 is not power of 2 then
    return φ
forall type in {SLIDING, FIXED} do
    forall bitmask in {1, 2, 3, 4} do
        if type==0 then
            offsetBits = ⌈log2(w)⌉
        else
            offsetBits = ⌈log2(w/b)⌉
        forall offset in offsetBits do
            if offset + bitmask OR offset + bitmask + 1 OR offset + bitmask + 2
            is power of 2 then
                B = B + {bitmask, offset, type}
        end
    end
end
return B
```

3.2.1 Decode Friendly Parameter Selection

The three different types of compressed words (uncompressed, compressed with exact match and compressed with bitmask) can be converted to fixed length encoded words

by following these three steps. First, the compressed and bitmasked flags are stripped from compressed words. Next, these flags are then arranged together to form byte aligned word. Finally, the remaining content of the compressed words are arranged only if they satisfy the following conditions. Each of the uncompressed words needs to be multiple of 8 as described in Section 3.1.1. The dictionary index of compressed words and the sum with either of the flags should be equal to power of 2. This condition ensures that the dictionary index bits can be aligned to byte boundary. The mask information (offset and bit changes) of a bitmask compressed word is also subjected to similar condition.

Algorithm 4 describes a bitmask suggestion technique before compressing the bitstream such that they meet the above constraints. The bitmasks and type of bitmask explored are limited $(1, 2, 3, and 4$ bits) by the study described in Seong et al. [13]. Both $SLIDING$ and $FIXED$ bitmask types are suggested for these possible bitmask sizes. Figure 3-4(A) describes a bitstream compressed with parameters word length $w = 16$, dictionary size $d = 16$, number of bitmask $b = 1$ and bitmask used B = $\{s_0 = 2, t_0 = SLIDING, l_0 = 4\}$. Here two dictionary indices $(4 + 4$ bits) are combined to encode as a single byte. The two dictionary indices can belong to a fully matched compressed word or to a bitmask compressed word. The offset and mask $(4+2)$ of bitmask compressed word are then encoded with next words compressed flag (1 bit) and bitmask flag (1 bit) making the total number of bits aligned to a byte boundary. These extra bits serves two purposes: i) one padding the holes caused by misaligned offset bits, and ii) refills the flag bits that were used to decode this bitmask compressed word. Note that adding these extra flag bits refills the used flag bits but never overflows the flag register. A detailed strategic placement algorithm is discussed in the next subsection.

3.2.2 Decoding-Aware Placement of Compressed Bitstreams

The placement algorithm merges all compressed entries into a single bitstream for storage. Given any input entry list with format described in previous section, our algorithm passes through the entire list three times to generate the final bitstream. In the

first pass, we will try to attach two bits to each entry which is compressed with bitmask or RLE, so that the length of all entries (neglect flag bits) are either 4, 12 or 16. In the second pass, we simply extract the flags of each 8 successive words, then store them as a separate "flag entry" before these 8 words. Finally, we rearrange all the words so that all of them fit into 8 bit slots.

Algorithm 5: Placement of Encoded Bits

Input: Compressed Entry List L of size s
Output: Output Bitstream B
forall *entry* e *of* L **do**
  if e *is compressed using BM or RLE* **then**
    Remove compression flag of e_1 and matching flag of e_2 and append to e
end
Create all flag entries
forall *entry* e *of* L **do**
  if e *is a flag entry* **then**
    Put e in $B[\lceil f(e) \rceil]$
  if e *is not compressed* **then**
    Put e in $B[\lceil f(e) \rceil]$ and $B[\lceil f(e) \rceil] + 1$
  if e *is fully matched* **then**
    Put e in the lower or higher half of $B[\lfloor f(e) \rfloor]$ depending on $f(e)$
  else
    if $f(e)$ *is integer* **then**
      Put the high 4 bits of e in the higher half of $B[f(e)]$ Put the rest of e in of $B[f(e)] + 1$
    else
      Put the high 4 bits of e in the lower half of $B[\lfloor f(e) \rfloor]$ Put the rest of e in of $B[\lfloor f(e) \rfloor] + 1$
end
here $f(e) = 2n_u + 0.5n_m + 1.5n_b$, where n_u, n_m and n_b are the number of not compressed, fully matched and other entries before e respectively.

The entire algorithm is given in Algorithm 5. Figure 3-4(C) and (D) describe our bitstream rearrangement procedure using Figure 3-4(A) as input. In the first pass, the compression flag of entry E4 and matching flag of E5 are attached to the end of E3 (Figure 3-4(C)). Each entry now has a length of 4, 8 or 12. Then the remaining compression flags and matching flags are extracted as flag entries (byte 1 and 4 in Figure

3-4(D)) in the second pass. After that, we can easily rearrange all the bits and make them fit into the 8 bit-wide memory, as shown in Figure 3-4(D).

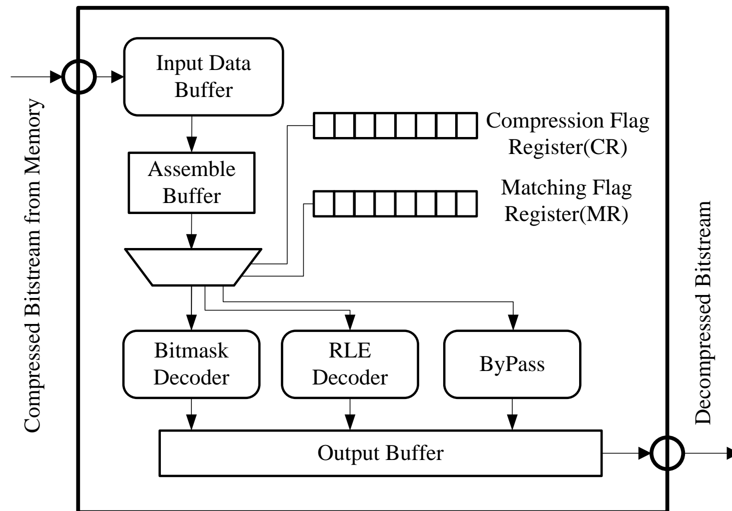

Figure 3-5. Decompression engine

3.2.3 Decompression Engine

The structure of our decompression engine is given in Figure 3-5. The compression flags and the matching flags are stored in corresponding shift registers: compress register (CR) and match register (MR). CR[0] and MR[0] indicate the flags for next compressed entry. In each cycle, the new incoming data is first classified using their flags, assembled into a complete compressed entry, then decoded by bitmask (BM), RLE or output directly. Our implementation of the BM and RLE decoder is based on the proposed design in Seong et al. [13] and Koch et al. [5]. If current entry is compressed with bitmask or RLE, the last two bits of this entry is directly sent to CR[0] and MR[0] (these two bits are indeed the flags of next compressed entry, which are rearranged to their current position by our placement algorithm). Otherwise, CR and MR are shifted by one bit. When CR or MR is empty, they are reloaded immediately using next incoming byte, which exactly corresponds to the flags of next 8 compressed entries (this is guaranteed by our placement algorithm). Since all encoded bits are carefully placed, we avoid the shift operation of the input buffer completely. Besides, the boundary between different compressed entries can be easily

Table 3-1. Parameter values for best compression ratio

w	d	b	B_1	B_2
16	1,2,4,8,**16**,32,64,128,256,512	1	1s,**2s**,3s,4s,1f,2f,3f,4f	-
24	1,2,4,8,16,32,64,128,256,512	1	1s,2s,3s,4s,1f,2f,3f,4f	-
24	1,2,4,8,16,32,64,128,256,512	1,2	1s,2s,3s,4s,1f,2f,3f,4f	1s,2s,3s,4s,1f,2f,3f,4f
32	1,2,4,8,16,32,64,128,256,512	1	1s,2s,3s,4s,1f,2f,3f,4f	-
32	1,2,4,8,16,32,64,128,256,**512**	1,**2**	1s,**2s**,3s,4s,1f,2f,3f,4f	1s,2s,**3s**,4s,1f,2f,3f,4f
40	1,2,4,8,16,32,64,128,256,512	1	1s,2s,3s,4s,1f,2f,3f,4f	-
40	1,2,4,8,16,32,64,128,256,512	1,2	1s,2s,3s,4s,1f,2f,3f,4f	1s,2s,3s,4s,1f,2f,3f,4f

identified. Therefore, the maximum operational speed of the corresponding hardware is not hampered by our variable length coding technique. The detailed experimental results can be found in Section 3.3.3

3.3 Experiments

We use two sets of hard to compress IP core bitstreams chosen from image processing and encryption domain (derived from Koch et al. [9] and Pan et al. [1]) to compare our compression and decompression efficiencies. All the benchmarks are in readable binary format (.rbt), each word length of 32 bit binary ASCII representation, or binary (.bin) format later converted to rbt format. All rbt files are then converted to specified word lengths listed in Section 3.1.2. We also use Xilinx Virtex-II family IP core benchmarks to analyze the results in this article, the same results are found applicable to other families and vendors too.

3.3.1 Decoding-Aware Parameters for Benchmarks

Table 3-1 shows the different parameter values used by the Algorithm 2 as described in Section 3.1.1, to evaluate the best possible compression ratio. Each column value is permutated with every other column. The parameters with best compression ratio is chosen for the final compression. The values highlighted are the final selected values for Koch et al. [9] and Pan et al. [1] benchmarks. The benchmarks in Koch et al. [9] can be efficiently compressed using 16 bit words, with 16 entry dictionary and a 2-bit sliding mask for storing bitmask differences. The benchmarks in Pan et al. [1] can be efficiently

compressed with 32 bit words, 512 entry dictionary entries and two bitmasks with a 2-bit and 3-bit sliding bitmasks. Note that if two bitmasks are used in order to reorganize the compressed bits, the bits indicating the number of bitmasks are stripped to form another 8 bit vector similar to compress and bitmask flags described in Section 3.2.1. This facilitates other fields to be arranged on a byte boundary.

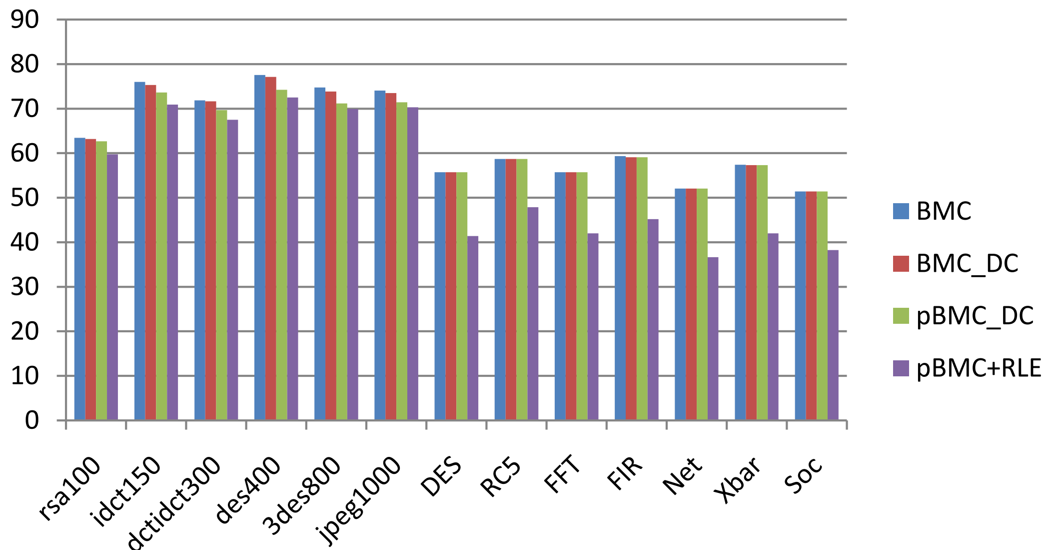

Figure 3-6. Comparison of compression ratio with bit mask based code compression technique

3.3.2 Compression Efficiency

We analyze the compression efficiency of our proposed approach compared to bitmask based compression technique proposed in Seong et al. [13] with respect to improved dictionary selection, decoding aware parameter selection and run length encoding of repetitive patterns proposed in our study. Our efficient dictionary selection is found to select dictionary entries improving the bitmask coverage by at least 5% for benchmarks which requires larger dictionary. We observed that in benchmarks that have large consecutive redundancy, run length encoding outperforms other techniques by at least 10 to 15%. The compression ratio is also evaluated with existing compression techniques proposed by Koch et al. [5] and Pan et al. [1]. Our proposed technique is found to outperform Koch et al. [5] by around 5% on [9] benchmarks and around 15% on [1]

benchmarks. The proposed decode aware compression technique produce close enough compression (with 5 to 10% variation) compared to Pan et al. [1].

3.3.2.1 Decoding aware vs. bitmask based compression

Bitmask based compression technique proposed in [13] is compared with enabling all three techniques proposed by our study. Figure 3-6 shows the compression ratio for all the benchmarks. These are the four different types of compression techniques that are compared: i) BMC - bit mask compression technique proposed in Seong et al. [13], ii) BMC_DC - bit mask compression along with efficient dictionary selection technique, iii) pBMC_DC - our proposed decode aware compression described in Section 3.1, and iv) pBMC+RLE - our proposed decode aware compression combined with run length encoding. The following are the observations and results for each of the techniques proposed.

Optimized dictionary selection: This compares the dictionary selection algorithm over the technique proposed in [13]. Figure 3-6 shows that for smaller benchmark, dictionary selection algorithm has little effect on improving compression ratio. The dictionary size is very small to reflect the optimization developed. This optimization becomes significant as the dictionary size increases. This can be seen from the compression ratio of benchmarks in Pan et al. [1]. These benchmarks require large dictionaries for better compression ratio (size up to 1K entries). The main advantages of our approach is that for any generic benchmark we don't have to find threshold value manually. Another advantage is that the optimization adds no additional decoding overhead. The optimized dictionary selection generates dictionary which improves the compression ratio by 4 to 5% on benchmarks that uses large dictionaries.

Decode aware parameter selection: This compares the decode aware bitmask based compression with efficient dictionary selection against bitmask based compression. In Figure 3-6 the pBMC column shows the behavior of decode aware parameter selection over the Seong et al. [1] method. Since decode aware compression technique explores more

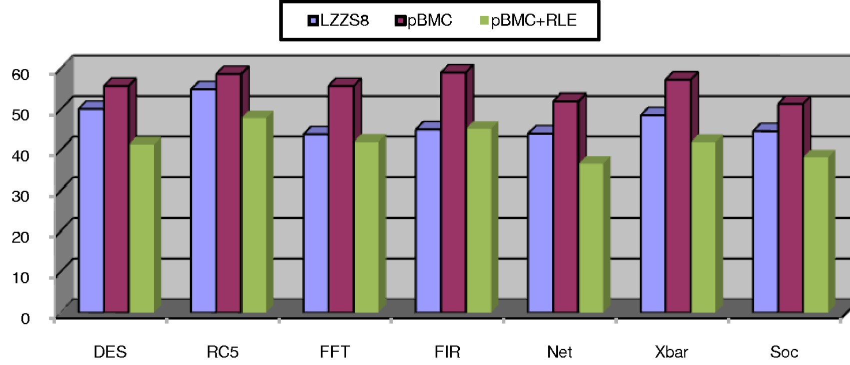

Figure 3-7. Comparison of compression ratio with LZSS-8 on Dirk et al. benchmarks

word lengths and dictionary size the proposed technique is found to choose parameters which gives best compression ratio and at the same time produces decode friendly compressed bitstreams. It is found that the proposed technique improves the compression ratio by at least 7 to 9% over bitmask based compression (BMC).

Run length encoding: This compares the run length encoding improvement along with other techniques to show the improvement of our proposed technique. The column pBMC+RLE in Figure 3-6 shows an improvement on all the benchmarks. This technique has the most improvement of all the techniques we proposed on improving the compression ratio. Most of the repetitive pattern is smartly encoded without adding any overhead in compression or during decoding the compressed bits. On an average we found 5 to 7% improvement over bitmask based compression for Pan et al. [1] benchmarks and 15% improvement on Koch et al. [5] benchmarks.

3.3.2.2 Decoding-aware vs. bitstream compression techniques

Now we compare the compression efficiency with existing bitstream compression techniques: LZSS technique proposed by Koch et al. [5] and distant vector based compression technique proposed by Pan et al. [1]. The distant vector compression technique uses format specific features to exploit redundancy thus benchmarks used in Koch et al. [5] could not be used.

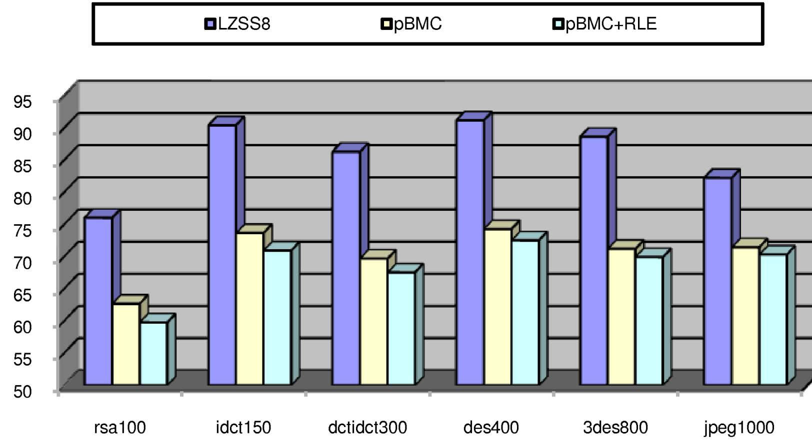

Figure 3-8. Comparison of compression ratio with LZSS-8 on Pan et al. benchmarks

LZSS: Figure 3-7 shows the comparison of compression ratio obtained by applying LZSS and two variants of decoding aware bitmask compression: a) pBMC: decode aware bitmask compression with efficient dictionary selection, and b) pBMC + RLE: pBMC combined with run length encoding. Figure 3-7 shows that pBMC + RLE technique achieves best compression ratio over all the other compression techniques. The pBMC + RLE technique compresses on an average 12% better than LZSS technique for benchmarks in Koch et al. [9]. The approach proposed in Seong et al. [13] fails to compress any of the benchmark below 50%. This is partly because the parameters selected does not yield better compression ratio and also because these benchmarks have a substantial amount of words repeating consecutively. The bitmask based compression proposed by Seong et al. [13] fails to capitalize this observation. Our decode friendly compression technique chooses efficient parameters to significantly compress these bitstreams combined with run length encoding of such repetitive words.

Figure 3-8 shows the compression ratio for Pan et al. [1] benchmarks. Our approach compresses these benchmarks with better compression ratio (20% better) than LZSS technique. The LZSS compression technique fails to compress these benchmarks substantially because these benchmarks are much larger and harder to compress than Koch et al. [5] benchmarks. The LZSS technique uses smaller window size and

smaller word length that inhibits exploiting matching patterns. This results in an overall unacceptable compression ratio. Another observation made is that run length encoding improves the compression ratio by only around 3 to 4% unlike the huge improvement over Koch et al. [5] benchmarks. This is because these benchmark do not have considerable repetitive patterns to provide significant improvement in compression ratio.

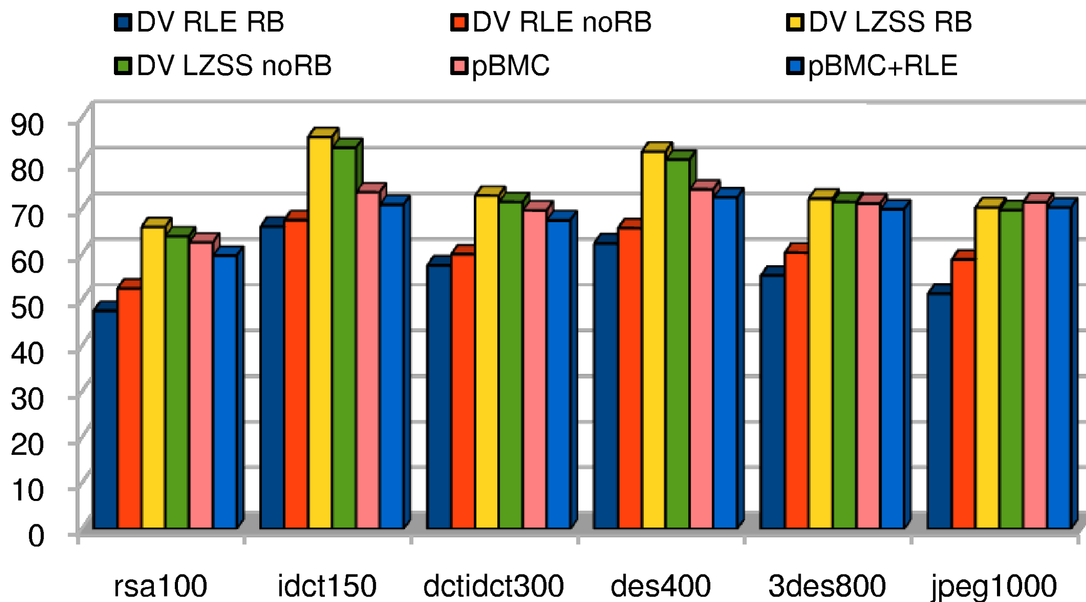

Figure 3-9. Comparison of compression ratio with difference vector compression technique on Pan et al. benchmarks

Difference vector: Figure 3-9 shows the compression ratio of our compression technique compared to difference vector [1] applied on single IP cores. The CLB frame's difference vector is encoded using Huffman based RLE with readback (DV RLE RB), without readback (DV RLE noRB), different vector encoded with LZSS with readback (DV LZS RB), and without readback (DV LZSS noRB). The compression technique proposed by Pan et al. [1] uses format specific characteristics of Virtex FPGA family. The technique parses all the CLB frames and rearranges the frames such that the difference between the frames are minimal. To get the best compression ratio these difference vectors are then encoded using variable length Huffman based run length encoding. From our implementation and the study conducted in [5], such complex encoding requires complex

Table 3-2. Operating speed and look up table usage of decoders

Type	Speed (MHz)	LUT Usage
Variable length bitmask decoder	130	445
Decode aware bitmask decoder	195	241
LZSS-8	198	83
LZSS-16	200	120

hardware to handle variable length Huffman codes and operates at very low speed. Our compression technique achieves around 5 to 10% closer to compression ratio achieved by best difference vector algorithm. By considering the decompression overhead imposed by Huffman based decoder, the compression ratio efficiency can be easily compensated by faster decompression time.

3.3.3 Decompression Efficiency

The decompression efficiency can be defined as the total number of cycles on the decoder output side to the total number of cycles needed to decompress uncompressed code. Lesser the number of cycles higher the performance because with less data being transferred a constant output is produced at a sustainable rate. The final configuration time is defined by the product of cycle time and the frequency at which the decoder operates. We synthesized variable length bitmask based decoder, fixed length bitmask based decoder and LZSS (8 bit symbols and 16 bit symbols) based decoder on Xilinx Virtex II family XC2v40 device FG356 package using ISE 9.2.04i to measure the decompression efficiency.

Fixed length vs. variable length bitmask decoder: We observed that both fixed length bitmask based and LZSS decoder can operate at a much higher frequencies compared to Huffman based or variable length bitmask based decompression engine. Converting variable length encoded words to fixed length has multiple advantages: i) high operational speed, and ii) scope for parallelizing the decoding process based on the current knowledge of at least 8 compressed words. Table 3-2 lists all the operating

Table 3-3. Decompression cycles for fixed length decoder

Benchmark	Decompression Cycles	Raw Cycles
des	255628	511256
RC5	331752	663504
fft	255628	511256
simpleFIR	255631	511262
ReCoLink	255632	511264
crossbar	255630	511260
ReCoNode	331752	663504

speeds of the three decoders. Our approach achieves almost the same operational speed as that of LZSS based accelerator. The results from the previous section shows that the data is better compressed in our approach. The decoder has less data to fetch and more data to output. Table 3-3 lists the number of cycles which is required to decode with and without compression. From the table we can conclude that our approach takes roughly half the number of cycles to that of the uncompressed decoder. An important thing to note is that uncompressed configuration process requires the configuration hardware to run at memory's slower operational speed. Further run length encoding of the compressed streams allows the decoder to accumulate the input bits for later decoding, while transmitting the data instantaneously for configuration.

Look up table usage: We evaluate the overhead with which decode aware compression achieves better compression and better decompression efficiency. We use number of look up table (LUT) on FPGA to measure the amount of resources utilized by each technique. Table 3-2 lists all the decoders and column 3 lists the number of LUT's used. The fixed length decoder requires lesser number of LUT than variable length bitmask based decoder. The fixed length decompression engine can be further improved by another 10% to 20% by using optimized one bit decrementer proposed in [15].

Decompression Time: Lastly we analyze the actual decompression time required to decode a FFT benchmark for Spartan III family. A cycle accurate simulator which simulates the decompression is used to estimate the decompression time. We have

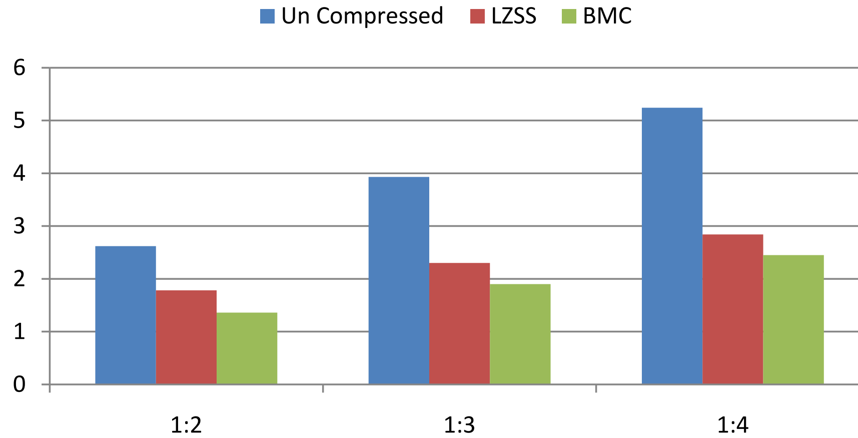

Figure 3-10. Comparison of decompression time for FFT benchmark

Table 3-4. Decompression time in milliseconds for FFT benchmark

	(Memory : FPGA) cycles					
	1 : 2		1 : 3		1 : 4	
FIFO Size	LZSS	BMC	LZSS	BMC	LZSS	BMC
1	1.78	1.36	2.3	1.9	2.84	2.45
4	1.76	1.34	2.27	1.89	2.82	2.44
8	1.74	1.34	2.25	1.88	2.8	2.43
16	1.72	1.33	2.23	1.88	2.78	2.43
32	1.7	1.33	2.22	1.88	2.78	2.43
64	1.69	1.33	2.2	1.87	2.77	2.42
Optimal	1.15	1.11	1.72	1.67	2.30	2.22
No Compression	2.62	2.62	3.93	3.93	5.24	5.24

simulated memory operating at different speed (2, 3 and 4 times slower) than FPGA operating speed. FPGA is simulated to operate at 100 MHz. For an uncompressed word, FPGA should operate at memory speed thus increasing the configuration time. In an optimal scenario the decompression time should be the product of compression ratio and uncompressed configuration time. Table 3-4 lists the required decompression time with different input buffer sizes. We observed that the buffer size does not affect the configuration time significantly. Figure 3-10 shows the improvement in decompression time over LZSS [5] technique by at least 15 to 20%. Our technique produces better compression ratio demonstrating better decompression efficiency closer to optimal decompression time.

CHAPTER 4
CONTROL WORD COMPRESSION

In this chapter we apply the proposed decoding aware bitmask compression technique to compress no instruction set computer (NISC) [16] control words and analyze its performance. No instruction set architecture is a reconfigurable processor architecture, which promises application a faster execution performance by selecting custom data path [17]. The application written in a high level language (currently supports programs written in ANSI C) is directly converted to control words bypassing the abstraction of instruction. This facilitates application to select custom datapath thus improving the system performance. The downside of this architecture is that the control words stored takes 4 to 5 times more space than regular instructions. In this chapter, we analyze the application of the proposed decoding aware compression technique to compress these control words. Two enhancements are proposed to improve compression efficiency, decompression overhead and resources usage. This chapter proposes a novel technique to smartly encode least frequently changing bits and to use multiple dictionaries to improve the compression efficiency by 15 to 20% over the best control word compression technique proposed by Gorjiara et al. [10].

4.1 Architecture of No Instruction Set Computer

No instruction set computer promises applications faster performance guarantees by analyzing datapath behavior and eliminating abstraction of instruction set to choose a custom datapath. By controlling selection of the optimal datapath NISC is capable of meeting application performance requirements. The datapath represented as control words forms the input to NISC processor. These control words tend to be at least 4 to 5 times wider than the regular instructions thus bloating the code size of the application. One of the promising approach is to reduce these control words by compressing them. Figure 4-1 shows compressed control word execution on a generic NISC architecture. The compressed control word is read from control memory (CMem) and decoded to obtain

the original control word and transmitted to the controller for execution. The branch and jump instructions are handled using target look up table (TLUT). These TLUT stores the offsets within the compressed code to resume decompression after a jump instruction. We now propose the enhanced version of the decode aware bitmask based compression technique.

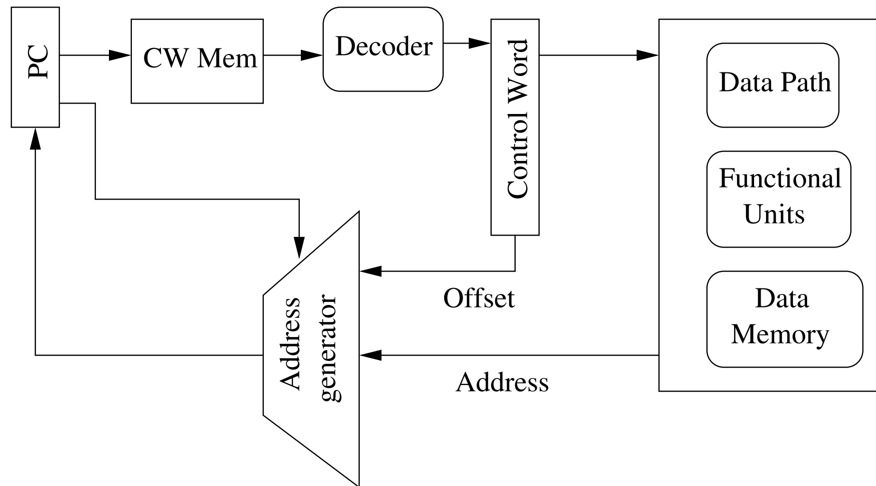

Figure 4-1. No instruction set computer architecture and decoder placement

4.2 Control Word Compression

No instruction set computer control words are usually 3 to 4 times wider than normal instructions. This is partly because the number of functional units and the data path usually runs closer to 100 or even more. This results in much wider control words. The direct application of any dictionary based or decoding aware compression technique on such control words fails to reduce the code size significantly. An interesting solution to obtain redundancy is proposed in [6], [10], [18] and [19], which splits the input control words to multiple slices and compresses them using multiple dictionaries. The application of this multiple dictionary approach on our proposed decoding aware bitmask compression technique (Section 3.1) with fixed word length improves the compression ratio better than the compression technique proposed by Gorjiara et al. [10]. The main disadvantage of existing algorithms is that they use variable length dictionary size, which depends on

control words pattern, whereas our proposed technique uses a fixed dictionary size. This limits the block RAM (BRAM) resources required to store potentially huge dictionaries to a very small and finite size.

No instruction set computer control words have don't care bits. This signifies that the bits indicating these functional units can be either enabled or disabled without affecting the output. Gorjiara et al. [10] discuss an interesting technique to resolve these don't care bits to obtain minimal dictionary size. Later we describe an extension of this algorithm to select an efficient dictionary that is bitmask friendly. The selected dictionary results in maximum bitmask matches with a smaller dictionary. In the next section we describe our compression algorithm applied on control words.

4.2.1 Bitmask Based Compression using Multiple Dictionaries

The input control words usually run close to 100 bits wide or even more, as discussed in the previous section. To achieve more redundancy and to reduce code size, the control words are split into two or more slices depending on the width of the control word. Each slice is then compressed using the Algorithm 2 described in Section 3.1. To achieve further code reduction two techniques are proposed in the following two sub sections. These techniques show an improvement in compression ratio without adding any significant overhead on the decoder.

Algorithm 6: Multi-dictionary compression

Input: Input control words with don't cares I
number of slices n
threshold bits that can change t
Output: Compressed control words C
W = strip_constant_bits(I)
$S[]$ = slice_and_remove_less_frequent_bits(W, n, t)
forall s *in* $S[]$ **do**
    $S[i]$ = bitmask_aware_dont_care_resolve(s)
    $C[i]$ = rle_bitmask_compress($S[i]$)
end
Generate verliog decoder code based on the parameters and skip map.
return C

The Algorithm 6 lists the major steps in compressing NISC control words. Initially all the constant bits are removed to get reduced control words along with an initial skip map (skip map represents the bits that can be skipped to compress and are hardcoded). In the next step the input is split into required slices. The less frequent bits are then removed from each slice using Algorithm 8. In each slice, don't care values are resolved using Algorithm 7. The resultant slices are compressed in the next step using the Algorithm 2 described in Section 3.1. It must be noted that the word length passed to decoding aware compression algorithm is fixed and determined by the control words width. This does not hamper the decompression efficiency because the compressed control words reside in FPGA memory unlike in the case of configuration bitstream compression where the compressed bitstreams reside on an external memory. Finally a Verilog decoder is generated based on the compression parameters.

Algorithm 7: Bitmask Aware Don't Care Resolution

Input: Unique input control words $C = \{c_i, f_i\}$
number and type of bitmasks b, B $= \{s_i, t_i\}$
Output: merged control words M
forall *u in C* **do**
  forall *v in C* **do**
    if *bit_conflictu, v cannot be bitmasked using B* **then**
      add (u,v) with $c_{uv} = f_u$ and (v,u) with $c_{vu} = f_v$
  end
end
colors = wp_color_graph(G)
sort_on_frequencies(G)
forall *clr in colors* **do**
  M = merge all the nodes with same color *clr*
  Retain the bits of most frequent words while merging
end

4.2.2 Bitmask Aware Don't Care Resolution

In a generic NISC processor implementation not all functional units are involved in a given datapath, such functional units can be either enabled or disabled. The compiler [20] inserts don't care bits in such control words. Any compression algorithm to get maximum

compression can utilize these don't care values efficiently. One such algorithm presented in [10] creates a conflict graph with nodes representing unique control words and edges between them represents that these words cannot be merged (conflict). Applying minimal k colors to these nodes results in k merged words. It is well known fact that vertex coloring is a NP Hard problem. Hence a heuristic based algorithm proposed by Welsh and Powell [21] is used to color the vertices to obtain optimal merged dictionary. This algorithm is well suited in reducing the dictionary size with exact matches. The dictionary chosen by this algorithm might not yield better bitmask coverage.

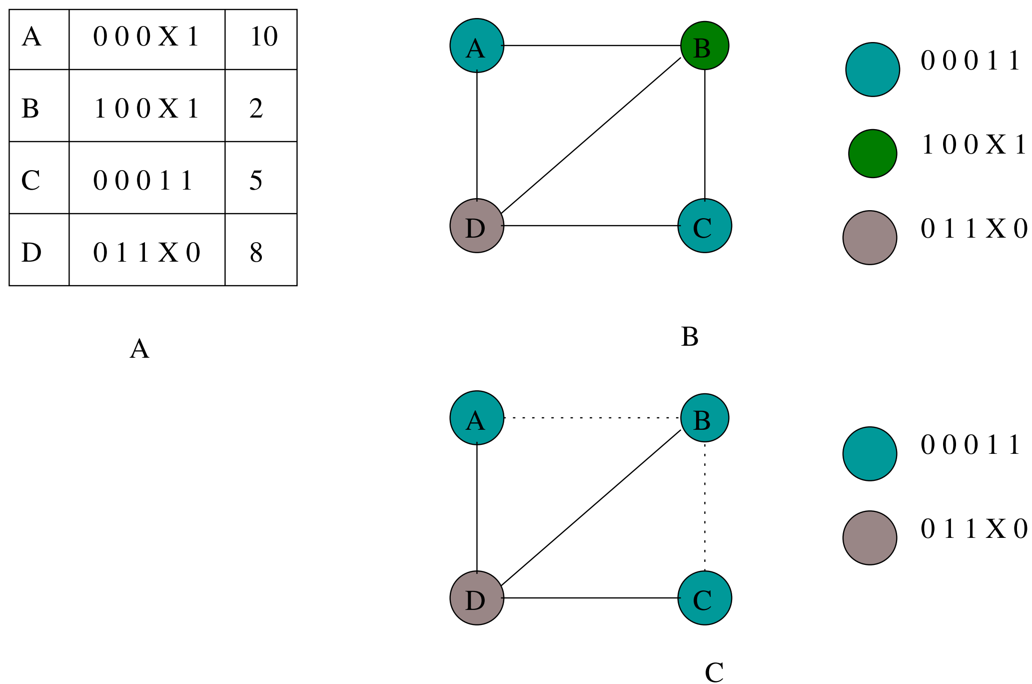

Figure 4-2. Bitmask aware don't care resolution. A) Input B) Welsh Powell - graph coloring C) Bitmask Aware Graph Coloring

An intuitive approach is to consider the fact that the dictionary entries will be used for bitmask based matching during compression. Algorithm 7 describes the steps involved in choosing such a dictionary. The algorithm allows certain bits that can be bitmasked while creating a conflict graph. This reduces the dictionary size drastically. The algorithm allows certain bits than can be bitmasked to avoid them to be represented as edges in the conflict graph, thus allowing the graph to be colored with less number of colors.

This results in smaller dictionary size with smaller dictionary index bits thus reducing the final compressed code size. It may be noted that while merging the vertices if the bits are already set then bits originating from the most frequent words will be retained. This promises reduced size as they result in more direct matches. Results indicate that dictionary chosen using the proposed algorithm produces 3 to 4% better compression ratio without any additional overhead on decompression.

Algorithm 8: Removal of Constant and Less Frequently Changing Bits

Input: Control Words with don't cares D
Threshold t number of bits
Output: Skip Map S
$S = \phi$
forall *w in D* **do**
  forall *b_i, i^{th} bit in w* **do**
    count_ones
    count_zeros
  end
end
create a skip_map of 0, 1 or taken with count $<$ threshold t.
forall *w in D* **do**
  if *w has a conflict with skip_map* **then**
    count the number of bits w conflicts with skip_map.
    if $conflict > 1$ **then**
      remove most conflict from previously calculated skip_map.
end
return S

Figure 4-2 describes an example don't care resolution of NISC control words and a merging iteration. The input words and their frequencies are provided to the Algorithm 7 as shown in Figure 4-2(A). There are four inputs A, B, C and D. Figure 4-2(B) represents the conflict graph constructed by the original don't care resolution algorithm [10]. The algorithm chooses three colors which represents the merged dictionary entries. The proposed bitmask aware graph creation algorithm skips the edges which can be bitmasked as shown in Figure 4-2(C). This example uses one 1-bit bitmask to store differences. The final colors indicate the merged dictionary entries. While merging the colored nodes, bits from high frequency words are retained.

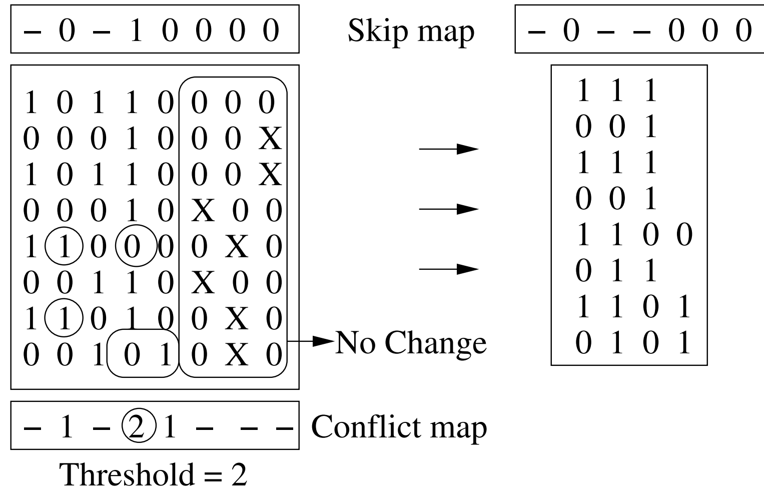

Figure 4-3. Removal of constant and least frequent bits in control words

4.2.3 Smart Encoding of Least Frequently Changing Bits

Upon closer analysis of the control word sequence reveals that some bits are constant or changes less frequently throughout the code segment. Removal of such bits improves compression efficiency and does not affect matches provided by rest of the bits. The less frequent bits are encoded by using yet another unused bitmask value as a marker (01 in case of a 2-bit bitmask). A threshold number determines the number of times that a bit can change in the given location throughout the code segment. It is found that 10 to 15 is a good threshold for the benchmarks used in our experiments. Algorithm 8 lists the steps in eliminating the constant bits and less frequently changing bits. Initially the algorithm calculates the number of ones and zeros in each bit position. In the next step only those bit positions with count 0 or less than threshold t are considered to be the initial skip map. In the case of less frequent bit positions each of the bit positions should not be allowed to change in the same control word, as this leads to multiple bit changes to be encoded in this word. To avoid this condition, the last step of the algorithm updates the skip map by constructing a conflict map for each word. The bit position which causes the most conflicts are eliminated thus leaving the new skip map covering one and only one bit positions in any given word.

Figure 4-3 describes an example control word sequence to demonstrate bit reduction. Each control word is scanned for number of ones and zeros in each bit position. The last three bit positions do not change throughout the input thus they are removed from the input, storing these same bits in a skip map. Columns with bit changes less than threshold (2 in this example) i.e. column 2, 4 and 5 have bits changing less frequently. In the final step conflict map is created (listed at the bottom part of the figure) representing the number of collisions. The bit positions with collisions 0 or 1 are considered for skipping, the remaining columns (column 4) are excluded from the initial skip map. The skip map and the bits which needs to be encoded are shown on the right side. It can be noted that there is a significant reduction in code size. The decompression section discusses in detail how these less frequent bits are reassembled.

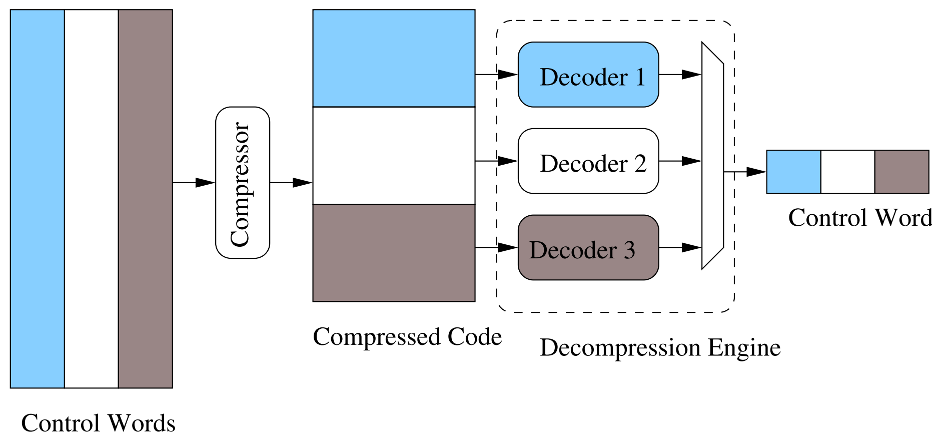

Figure 4-4. Control word compression methodology

The complete flow of uncompressed, compression and decompressed control words is shown in Figure 4-4. The input file containing the control words is passed to the compressor. The compressor reduces the control word size by applying the algorithms described in the previous section and outputs the compressed file in the order of slices. Later each decoder fetches compressed words from different location of the memory. These compressed words are then decoded using the dictionary stored on block RAM (BRAM). The decompressed code is then assembled to form the original control word.

4.3 Decompression of Control Words

This section analyzes the modification required to the decompression engine proposed for configuration bitstream compression in the previous chapter. This section also discusses the branch target lookup table required to handle branch control words.

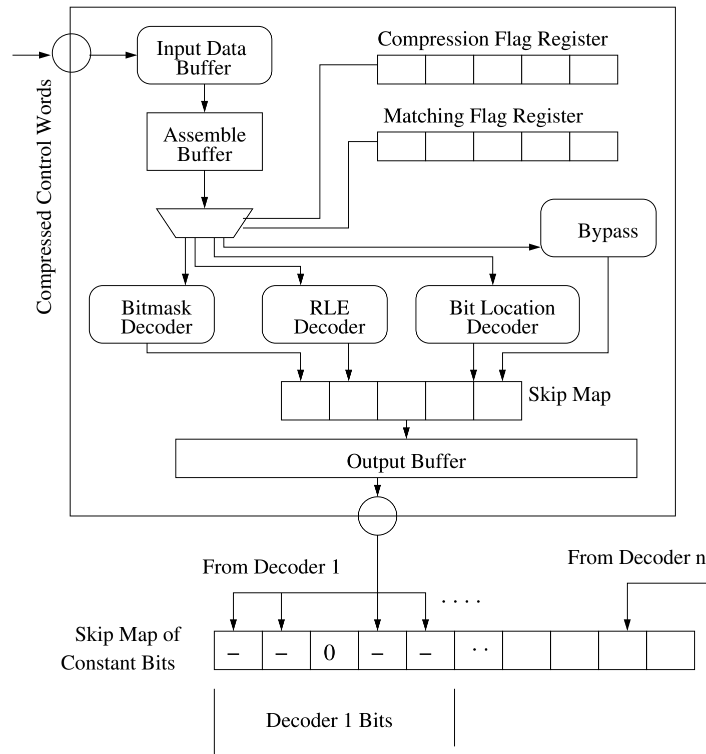

Figure 4-5. Multi-dictionary based decompression engine

4.3.1 Decompression Engine

Figure 4-5 describes the structure and components of the NISC control word decompression engine. The decompression hardware comprises of multiple decoding units for each slice of compressed control word. Each decompression engine contains input buffer to store the incoming data from memory. The data from input buffer is then assembled for further processing. Based on the type of compressed word, control is passed to the corresponding decode unit. Each decoding engine has a skip map register to insert extra bits that were removed during less frequently occurring bit optimization. A separate unit to toggle these bits handles the insertion of these difference bits. This unit reads the

offset within the skip map register to toggle the bit and places it in the output buffer. All outputs from decoding engine are then directed to the skip map for constant bits which holds the completely skipped bits (bits that never change).

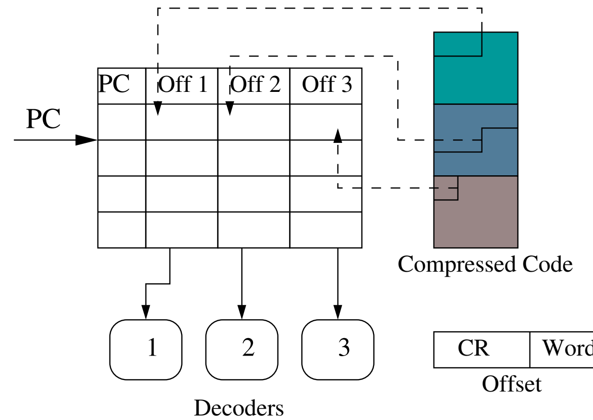

Figure 4-6. Branch target look up table for compressed control words

4.3.2 Branch Target Look Up Table

In any program branch control words produces program counter to jump to a different location to load a new control word. The decoder should handle such jumps within a program. A look up table based branch relocation approach is used in which static jump location targets are stored in a table [13]. The proposed technique uses multiple dictionaries and multiple decode units to handle decompression of all the slices. The table is redesigned to store offset of all the slices along with the new target location. Figure 4-6 describes the branch look up table design. The look up table is indexed based on new PC and returns multiple offsets to be used by individual decoders. Each offset stores the compress register (CR) offset within its compressed word. The decoder reads the new compress register from this offset. The offset also contains the word number from which the decoding resumes. The limitation of this implementation is that decoder cannot handle dynamic jumps which is also currently not supported by NISC architecture [20].

4.4 Experiments

The effectiveness of the proposed compression technique is measured using MiBench benchmarks [10]. The metrics evaluated are compression ratio, decompression speed,

resources used by decompression engine (LUT and BRAMs). It is found that the proposed compression technique is found to reduce the code size further by 15 to 20% over the compression technique proposed by Gorjiara et al. [10]. Decompression speed of the decoding units capable of operating at 130 MHz in the range of NISC processor operating range. BRAM used is fixed (1 or 2) for all the benchmarks.

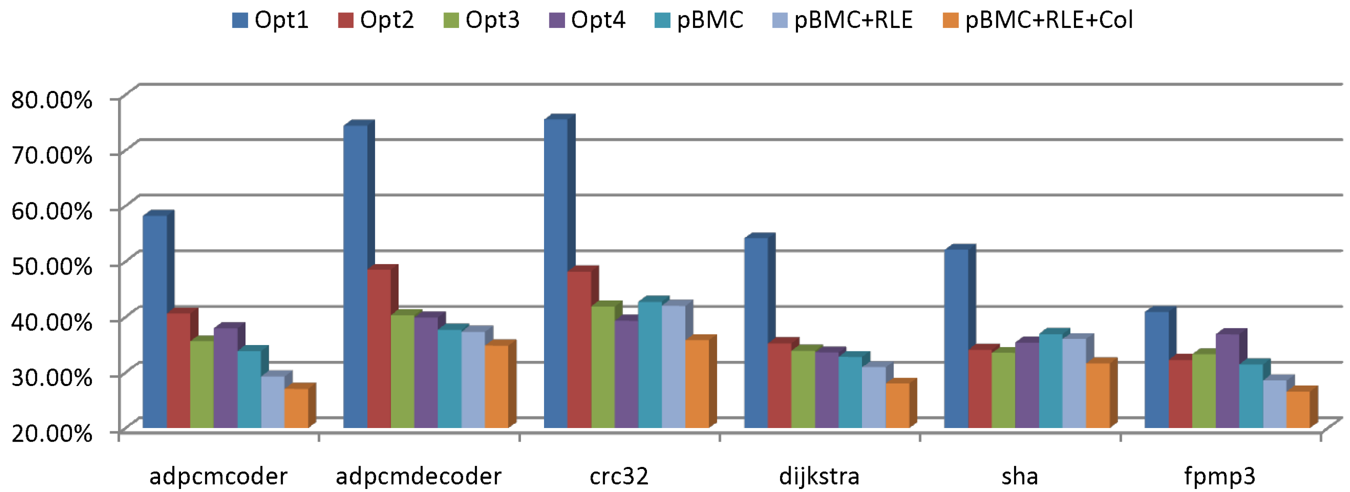

Figure 4-7. Comparison of compression ratio with dictionary based compression technique on MiBench benchmark

Figure 4-7 shows the comparison of compression ratio of different benchmarks provided in MiBench [22]. These benchmark consists of numerous applications from security algorithms, network and telecom domain. Each benchmark is compiled in release mode using NISC compiler [20] with optimization level set to 0. The proposed compression technique with 3 slice option is found to compress all the benchmarks with at least 15 to 20% better compression relative to three dictionary option of multi dictionary algorithm proposed by Gorjiara et al. [10].

CHAPTER 5
OPTIMAL REPRESENTATION OF BITMASKS

In a bitmask based compression each bitmask is represented as s_i, t_i, l_i, which denotes the size, type and offset within the word. A n-bit bitmask remembers n consecutive bit differences between a matched word and a dictionary entry. To store n bit differences a naive approach is to store all the n bits. But a careful and closer analysis reveals that, to encode the same n bits we only need $n - 1$ bits. The following section discusses on how this can be achieved.

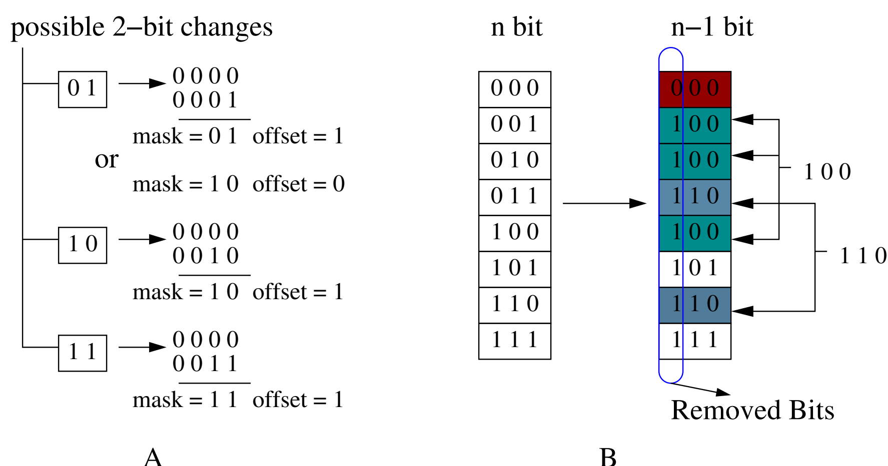

Figure 5-1. The n-1 encoding of n bit bitmask. A) Equivalence of 2-bit bitmask to 1-bit bitmask B) 3-bit bitmask transformed to 2-bit bitmask

5.1 Optimal Encoding of n Bits

To encode a single bit difference we do not need any bits to indicate the difference. As the presence of offset bits indicates that there is a one bit difference and the difference bit encoded will be always 1. Hence we can remove this bit from bitmask encoding. Now consider a 2-bit bitmask, there are four possibilities { 00, 01, 10, 11 }. In these possibilities the first pattern (00) never occurs as this indicates that there are no differences. The second and third bitmasks are both equivalent except that offset of these differ by one. Hence both can be represented using 10 bitmask. Thus there are only two bitmasks $(10, 11)$ that needs to be encoded. Hence a single bit is sufficient to represent these 2-bit bitmasks. In general a n bit bitmask can theoretically cover 2^n differences. Out

of these the first pattern will never be used which leaves $2^n - 1$ patterns to be encoded. Out of these patterns there are $2^{n-1} - 1$ have most significant bit set to 0 i.e. the first half of truth table. These bitmasks can be rotated such that it starts with 1 as shown in Figure 5-1(B). The rotation of the bitmask leaves the offset to be shifted suitably. Figure 5-1(A) shows all possible differences that can be encoded using a 2-bit bitmask. It can be noted that bitmask difference 01 is equivalent to bitmask difference 10. The only difference is that the offset gets changed from 1 to 0 as discussed earlier (the offset is relative from the least significant bit position). Thus in conclusion we need $n - 1$ bits to store n differences.

5.2 Proof for n-1 Bit Representation

Definition 1. *Let two words w_1 and w_2 have n bit consecutive differences then $f(n)$ be the function which represents the number of bit changes that n bits can record. Let $o(n)$ be the function which represents offset of the bit changes recorded from the least significant bit.*

Note that $f(n) = 2^n$, out of these 2^n bit changes there are 2^{n-1} bit changes that have most significant bit (MSB) set to 0 and 2^{n-1} bit changes have MSB set to 1.

Lemma 1. *Let G be the set that represents the bit changes with MSB set to 1, and H be the set that represents the bit changes with MSB set to 0. Then $G \equiv H$.*

Proof. Let $G = \{g_1, g_2, ..., g_m\}$, $H = \{h1, h2, ..., h_m\}$, where $g_1, g_2, ..., g_m$ are bit changes with MSB set to 1, $h_1, h_2, ..., h_m$ are bit changes with MSB set to 0, $m = 2^{n-1}$, and let i be a bit change element from set H. Then in m possible bit changes with MSB set to 0 for any i^{th} bit change element, let $r(i)$ be the number of bit rotations required such that i^{th} bit change has 1 in its MSB set then the new offset for this bit change be $o'(i) = o(i) - r(i)$. Since the number of rotation required is always less than n ($r(i) < n$), and the previous offset is at least n $o(n) \geq n$ the new offset $o'(i)$ is always greater than 0. Thus all the elements in set H can be transformed to bit change element with MSB set to 1. Thus both sets H and G are equivalent, which proves the lemma. □

Theorem 1. *Let n be the number of consecutive bit changes to encode between two words w_1 and w_2. Then $n-1$ bits are sufficient to encode n bit changes.*

Proof. A n bit change can encode possibly $f(n) = 2^n$ bit changes. Out of these 2^{n-1} bit changes have MSB set to 0. These bit changes can be converted to a bit change with MSB set to 1 (see lemma 1). Thus we have only 2^{n-1} or $f(n-1)$ to encode which requires $n-1$ bits to encode these changes, which completes the proof. □

5.3 Experiments

The application of this optimal representation improves the compression efficiency in cases where bitstreams contains data such that most of the words are encoded using one or more bitmasks. Figure 5-2 shows the comparison of the optimized representation of the bitmask applied on benchmarks used in configuration bitstream compression [9]. It is found that on an average there is an improvement of around 1 to 3% on overall compression efficiency. An advantage of this technique is that the improvement is achieved without adding any extra logic or overhead on decompression.

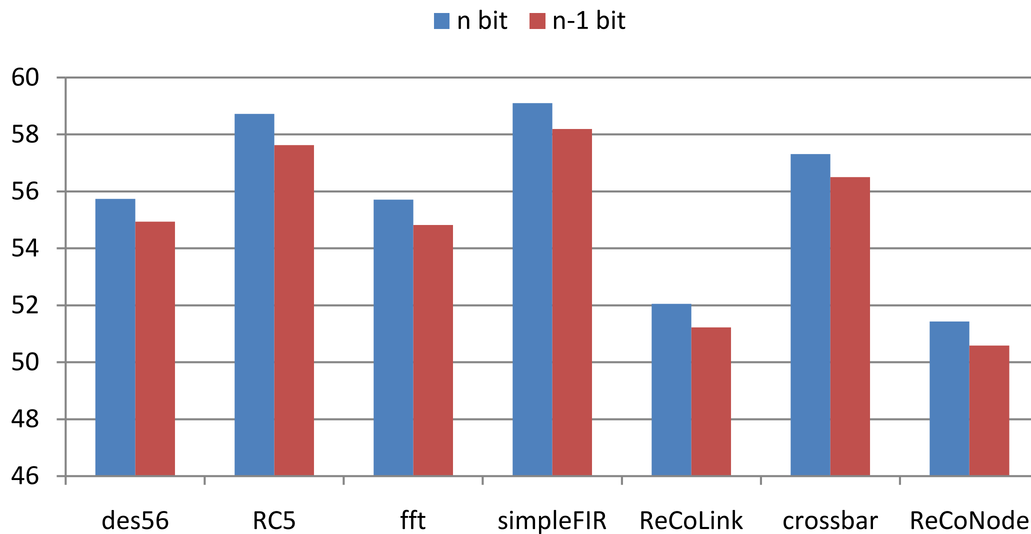

Figure 5-2. Comparison of compression ratio with and without using n-1 bit encoding scheme

CHAPTER 6
CONCLUSION AND FUTURE WORK

6.1 Conclusion

The existing compression algorithms can be classified into two categories. The first set of algorithms significantly reduce bitstream size but does not consider decompression overhead. The second set of compression techniques are efficient in decompression but compromises compression efficiency. Our study proposed a decoding-aware compression technique that tries to obtain the best possible compression efficiency as well as decompression performance. The proposed compression technique analyzes the effect of parameters on decompression overhead and selects compression parameters that are decode friendly. The proposed technique also exploits run length encoding of consecutive repetitive patterns to further improve the compression ratio and decompression efficiency. To enable decode friendly compression our study proposed a strategic rearrangement algorithm to reorganize variable length compressed bits to obtain fixed length compressed bitstreams. The fixed length encoding of the compressed words enabled the decompression engine to operate at FPGA's maximum operational frequency. A novel dictionary selection algorithm is developed that produces dictionary, covering maximum words using both minimal dictionary size and minimum number of bitmasks. Our experimental results demonstrated that the proposed technique improved compression ratio by 10 to 15% while the decompression engine capable of operating at 200 MHz. The configuration time is reduced by 15 to 20% compared to the best known decompression accelerator [5].

We also studied the application of decode aware compression technique to compress NISC control words and to reduce BRAM usage on FPGA. We presented a novel technique for smart encoding of constant and least frequently changing bits to further reduce the control word size. The control word size is efficiently reduced by splitting the wider control words to smaller slices and compressing them using multiple dictionaries. The proposed technique improved compression ratio by 15 to 20% over existing control

word compression techniques [10]. Finally our study also developed a novel encoding scheme to efficiently encode n bit changes in a bitstream data using only $n-1$ bits. This technique improved the compression ratio by 1 to 3% without adding any decompression overhead.

6.2 Future Research Directions

Memory and communication bandwidth has been a major bottleneck in most of the system design. As a result, there is an increasing performance gap between FPGA and memory. The decode aware compression presented in our study is a promising direction to bridge this gap by reducing the data size and by accelerating the decompression process. Our study explored only few problems in reconfigurable systems where decode aware compression can improve the system performance. The proposed techniques in our study can be further explored in the following directions:

1. Bitmask-based compression technique allows better compression and faster decompression engine. Binary tries work on longest prefix and bit differences, drastically reducing the bits required to encode. An interesting approach is to combine these two techniques to encode very hard to compress audio and video data. Such a combination would provide faster decoding and better lossless data compression.

2. The proposed technique can be applied in compressing data sent over heterogenous network elements. The decode aware decompression can bridge the gap between various network components operating at different bandwidth and frequency range.

3. Further studies can be conducted to eliminate the threshold parameter that is used to limit the exploration of word length. The input data pattern can be automatically analyzed to choose the parameters for compression. This potentially bring the compression ratio and decompression overhead closer to optimal efficiencies.

4. The current application of optimal representation of n bit difference can be further explored on systems that store bit differences. The systems that requires large number of bitmasks to encode data will be benefited by the proposed optimal encoding scheme. Some of the systems which we identified are in the area of efficient database storage and differential data backup based systems.

REFERENCES

[1] J. H. Pan, T. Mitra, and W. F. Wong, "Configuration bitstream compression for dynamically reconfigurable FPGAs," in *ICCAD '04: Proceedings of the 2004 Institute of Electrical and Electronics Engineers (IEEE)/Association for Computing Machinery (ACM) International Conference on Computer-Aided Design*, 2004, pp. 766–773.

[2] D. A. Huffman, "A method for the construction of minimum-redundancy codes," in *Proceedings Institute of Radio Engineers*, September 1952, pp. 1098–1101.

[3] S. Hauck and W. D. Wilson, "Runlength Compression Techniques for FPGA Configurations," in *FCCM '99: Proceedings of the seventh annual Institute of Electrical and Electronics Engineers (IEEE) symposium on Field-programmable Custom Computing Machines*, 1999, p. 286.

[4] A. Dandalis and V. K. Prasanna, "Configuration compression for FPGA-based embedded systems," in *FPGA '01: Proceedings of the 2001 Association for Computing Machinery (ACM)/Special Interest Group on Design Automation (SIGDA) ninth international symposium on Field Programmable Gate Arrays*, 2001, pp. 173–182.

[5] D. Koch, C. Beckhoff, and J. Teich., "Bitstream decompression for high speed FPGA configuration from slow memories," in *ICFPT '07: International Conference on Field-Programmable Technology*, 2007, pp. 161–168.

[6] S. W. Seong and P. Mishra, "A bitmask-based code compression technique for embedded systems," in *ICCAD '06: Proceedings of the 2006 Institute of Electrical and Electronics Engineers (IEEE)/ Association for Computing Machinery (ACM) International Conference on Computer-Aided Design*, 2006, pp. 251–254.

[7] Xilinx, Inc., "Virtex-II platform FPGA user guide, version 2.4." San Jose, CA, USA: Xilinx, Inc., 2003. [Online]. Available: http://www.xilinx.com

[8] Opencore, "IP Core repository." Stockholm, Sweden: Opencore.org, 1999. [Online]. Available: http://www.opencores.org

[9] D. Koch, C. Beckhoff, and J. Teich, "FPGA bitstream compression benchmark." Erlangen, Germany: Dept. of Computer Science 12, University of Erlangen-Nuremberg, 2007. [Online]. Available: http://www.reconets.de/bitstreamcompression/

[10] B. Gorjiara and D. Gajski, "FPGA-friendly code compression for horizontal microcoded custom IPs," in *FPGA '07: Proceedings of the 2007 Association for Computing Machinery (ACM)/Special Interest Group on Design Automation (SIGDA) 15th international symposium on Field Programmable Gate Arrays*, 2007, pp. 108–115.

[11] Xilinx, Inc., "Virtex series configuration architecture user guide, version 1.7." San Jose, CA, USA: Xilinx, Inc., 2004. [Online]. Available: http://www.xilinx.com

[12] A. Khu, "Xilinx FPGA Configuration Data Compression and Decompression." San Jose, CA, USA: Xilinx, Inc., 2001. [Online]. Available: http://www.xilinx.com

[13] S. Seong and P. Mishra, "Bitmask-Based Code Compression for Embedded Systems," in *TCAD '08: Institute of Electrical and Electronics Engineers (IEEE) Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2008, pp. 673–685.

[14] Y. Xie, W. Wolf, and H. Lekatsas, "Code compression for VLIW processors using variable-to-fixed coding," in *ISSS '02: Proceedings of the 15th International Symposium on System Synthesis*, 2002, pp. 138–143.

[15] S. Bi, W. Wang, and A. A. Khalili, "Multiplexer-based binary incrementer/decrementers," in *3rd International Institute of Electrical and Electronics Engineers North-East Workshop on Circuits and Systems (IEEE-NEWCAS) Conference*, June 2005, pp. 219–222.

[16] M. Reshadi, D. Gajski, and B. Gorjiara, "No Instruction Set Computer (NISC) Technology." Irvine, CA, USA: Center for Embedded Computer Systems, University of California Irvine, 2007. [Online]. Available: http://www.ics.uci.edu/~nisc/

[17] M. Reshadi and D. Gajski, "A cycle-accurate compilation algorithm for custom pipelined datapaths," in *CODES+ISSS '05: Proceedings of the 3rd Institute of Electrical and Electronics Engineers (IEEE)/Association for Computing Machinery (ACM)/International Federation for Information Processing (IFIP) international conference on Hardware/Software Codesign and System Synthesis*, 2005, pp. 21–26.

[18] S. J. Nam, I. C. Park, and C. M. Kyung, "Improving Dictionary-Based Code Compression in VLIW Architectures (Special Section on VLSI Design and CAD Algorithms)," in *Institute of Electronics, Information and Communication Engineers (IEICE) transactions on fundamentals of electronics, communications and computer sciences*, vol. 82, no. 11, November 1999, pp. 2318–2324.

[19] IBM, "CodePack: PowerPC Code Compression Utility User's Manual, version 3.0." Armonk, NY, USA: International Business Machines (IBM) Corporation, 1998. [Online]. Available: http://www.ibm.com

[20] M. Reshadi, "No-Instruction-Set-Computer (NISC) technology modeling and compilation," in *PhD thesis*. Irvine, CA, USA: University of California Irvine, 2007.

[21] T. Jensen and B. Toft, *Graph coloring problems*, ser. Discrete Mathematics and Optimization. New York: Wiley-Interscience, 1995.

[22] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 Institute of Electrical and Electronics Engineers (IEEE) International Workshop*, 2001, pp. 3–14.

BIOGRAPHICAL SKETCH

Chetan Murthy received his B.E. degree from the Department of Information Science and Engineering, in the People's Education Society Institute of Technology (PESIT) affiliated with Visvesraiah Technological University, India, in 2004. In 2004, he joined Huawei Technologies India Private Ltd., Bangalore, India. As a software engineer, he worked on developing and maintaining core kernel modules (memory, database, message, inter node communication, inspect, software patch, memory based zip, memory file system and flash based file system) for distributed systems supporting realtime capabilities running on PowerPC, MIPS, ARM and X86 architectures. Since summer of 2008, he has been working on compression algorithms for FPGA configuration bitstreams and no instruction set computer (NISC) control words at Embedded Systems Laboratory, University of Florida.