Bitmask-based Control Word Compression for NISC Architectures

Chetan Murthy and Prabhat Mishra
Computer & Information Science & Engineering
University of Florida, Gainesville, FL 32611, USA
{cmurthy, prabhat}@cise.ufl.edu

ABSTRACT

Implementing a custom hardware is not always feasible due to cost and time considerations. No instruction set computer (NISC) architecture is one of the promising direction to design a custom datapath for each application using its execution characteristics. A major challenge with NISC control word is that they tend to be at least 4 to 5 times larger than regular instruction size, thereby imposing higher memory requirement. A promising approach is to compress these control words to reduce the code size of the application. This article proposes an efficient bitmask-based compression technique to drastically reduce the control word size while keeping the decompression overhead minimal. The main contributions of our approach are: i) efficient don't care resolution for maximum bitmask coverage using limited dictionary entries, ii) run length encoding to significantly reduce repetitive control words, and iii) smart encoding of constant and less frequently changing bits. Our experimental results demonstrate that our approach improves compression efficiency by an average of 20% over the best known control word compression, giving a compression ratio of 25% to 35%.

Categories and Subject Descriptors: C.3 [Special-Purpose and Application-based Systems]: Microprocessor applications

General Terms: Design, Algorithms

Keywords: No-Instruction-Set Computer, Compression.

1. INTRODUCTION

It is not always efficient to run an application on a generic processor, whereas implementing a custom hardware is not always feasible due to cost and time considerations. One of the promising direction is to design a custom datapath for each application using its execution characteristics. The abstraction of instruction set in generic processors limits from choosing such custom data path. No instruction set architecture (NISC) promises faster performance guarantees by analyzing the datapath behavior and eliminating abstraction of instruction set to choose a custom datapath thus controlling the selection of optimal datapath to meet applications performance requirements. These datapath or control word (CW) tend to be at least 4 to 5 times wider than regular instructions thus increasing the code size of applications. One promising approach is to reduce these control words by compressing them. Figure 1 shows the compressed control word execution flow on a NISC architecture. The compressed control word is read from control word memory (CW Mem) and decoded to obtain original control word and sent to controller for execution.

Compression ratio is the metric commonly used to measure effectiveness of a compression technique (Equation 1). Clearly, smaller the compression ratio better the compression efficiency.

$$Compression\ Ratio = \frac{Compressed\ Size}{Uncompresssed\ Size} \quad (1)$$

Widely used code compression techniques utilize a dictionary to store most frequently occurring instructions. The instructions are replaced with smaller dictionary indices. The application of such algorithm on NISC control words is found not to result in significant reduction in code size. This is because the control words replacing the original instructions might not retain the same repeating pattern. Gorjiara et al. [3] described an interesting approach in which the control words are split to obtain redundancies and then compressed using multiple dictionaries. The main disadvantage of this technique is that the complete binary is stored in dictionary. This leads to large compressed code (less compression) and variable length dictionary size that requires variable number of block RAMs (BRAM) on which these dictionaries are stored. Seong et al. [9] presented a promising approach to achieve better compression by using limited dictionary entries and recording bit changes to match most of the instructions with dictionary. The direct application of this algorithm is found not to reduce the control word size significantly. It is a major challenge to develop an efficient compression technique which significantly reduces control word size and has minimal decompression overhead.

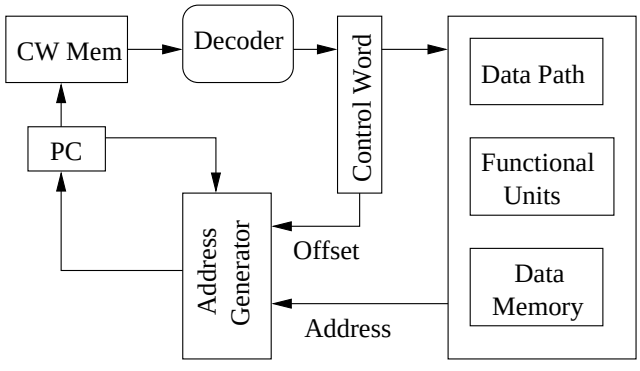

Figure 1: NISC architecture and decoder placement

In this paper, we propose an efficient compression technique that takes advantage of both NISC control word compression [3] and bitmask-based compression [9] techniques. This paper makes four important contributions: i) an efficient control word compression technique to improve compression ratio by splitting control words and compressing them using multiple dictionaries, ii) a bitmask aware don't care resolution to decrease dictionary size and improve

GLSVLSI'09, May 10–12, 2009, Boston, Massachusetts, USA.

dictionary coverage, iii) smart encoding of constant and less frequently changing bits to further reduce the control word size, and iv) run length encoding of repetitive sequences to both improve compression ratio and decrease decompression overhead by providing the uncompressed words instantaneously.

The rest of the paper is organized as follows. Section 2 discusses the existing code compression techniques. Section 3 describes NISC architecture and existing bitmask based compression. Section 4 describes our compression technique followed by a discussion on multiple dictionary based decompression in Section 5. Section 6 presents our experimental results. Finally Section 7 concludes the paper.

2. RELATED WORK

The first code-compression technique for embedded processors was proposed by Wolfe and Chanin [1]. Their technique uses Huffman coding, and the compressed program is stored in the main memory. The decompression unit is placed between the main memory and the instruction cache. They used a Line Address Table (LAT) to map original code addresses to compressed block addresses. Lekatsas et al. [4] proposed a dictionary based decompression prototype that is capable of decoding one instruction per cycle. The idea of using dictionary to store the frequently occurring instruction sequences has been explored by various researchers [2], [8]. The techniques discussed so far target reduced instruction set computer (RISC) processors. There has been a significant amount of research in the area of code compression for very long instruction word (VLIW) and no instruction set computer (NISC) processors. The technique proposed by Ishiura and Yamaguchi [6] splits a VLIW instruction into multiple fields, and each field is compressed by using a dictionary-based scheme.

Gorjiara et al. [3] applies similar approach in splitting the control words into different fields and compressing them using multiple dictionaries. Their approach can lead to unacceptable compression since it stores the complete binary in the dictionary. Moreover, it will require variable number of block RAMs (BRAM) to store variable-length dictionaries for different applications. Our approach outperforms [3] on both fronts by achieving 20% better compression (on average) using a fixed-length dictionary. Seong et al. [9] proposes a bitmask-based compression to improve compression ratio by creating matching patterns using bitmasks. However, the direct application of their algorithm is not beneficial due to lack of redundancy in longer control words. Moreover, the existing approach does not handle the presence of don't cares in input control words. As a result, it will sacrifice on compression efficiency by randomly replacing don't cares by 0's or 1's. The previous two methods ([3] and [9]) are closest to our approach. Section 6 presents experimental results to show how our approach improves compression efficiency compared to these approaches, without introducing any additional decompression overhead.

3. BACKGROUND AND MOTIVATION

3.1 No Instruction Set Computer (NISC)

NISC technology is based on horizontal microcoded architecture. In this technology, first a custom datapath is generated for an application, and then the datapath is synthesized and laid out properly to meet timing and physical constraints. The final step is to compile the program on the generated datapath. If the application is changed after synthesis, it is simply recompiled on the existing datapath. This feature significantly improves the productivity of the designer by avoiding repetition of timing closure phase. NISC relies on a sophisticated compiler [7] to compile a program described in a high-level language to binary that directly drives the control signals of components in the datapath. The values of control signals generated for each cycle are called a control word. The control words (CW) are stored in control word memory (CW Mem, shown in Figure 1) in programmable IPs, while they are synthesized to lookup-table logic in hardwired dedicated IPs.

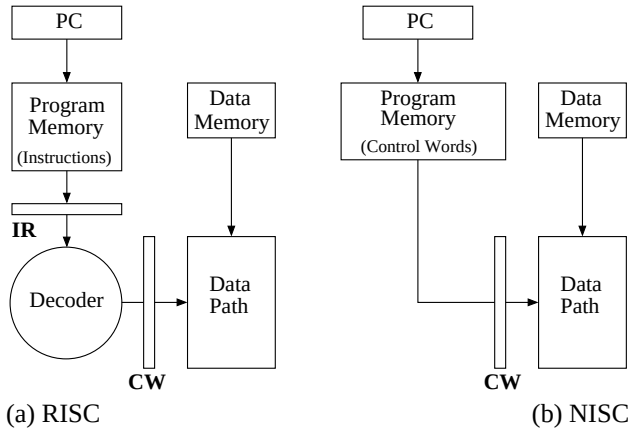

Figure 2: RISC and NISC Architectures

Figure 2 shows both RISC and NISC architectures. In RISC architectures, instructions are fetched from the program memory and stored in the instruction register (IR). The decoder decodes it to generate the required control word. In contrast, the NISC architecture stores the control word in the program memory thus eliminating the instruction decoder and hardware scheduler as shown in Figure 2(b). However, the code size is very large compared to RISC architectures due to two factors: control words are wider than instructions, and the number of NISC control words can be more than the number of RISC instructions. *On MiBench benchmarks, NISC implementation runs 5.54 times faster than RISC-based MicroBlaze, while its code size is four times larger [3].* The large code size increases the size of control memory in programmable IPs, and the area of control logic in dedicated IPs. The goal of our code compression technique is to reduce the control word size of NISC processors while maintaining the performance benefits.

3.2 Bitmask-based Compression

Dictionary-based compression techniques provide compression efficiency as well as fast decompression mechanism. The basic idea is to take advantage of commonly occurring instruction sequences by using a dictionary. The repeating occurrences are replaced with index of the dictionary that contains the original data. Figure 3 shows an example of dictionary based code compression using a 2 entry dictionary. Most frequently occurring words are stored in a dictionary and are replaced with the dictionary index (1 bit) during compression. The compressed program consists of both indices and uncompressed instructions. Figure 4(a) and (b) show the encoding scheme for a generic dictionary based compression technique. In this example, the dictionary based compression achieves a compression ratio of 97.5%.

Seong et al. [9] improve the standard dictionary based compression techniques by considering mismatches. The basic idea is to find the instruction sequences that are different in few consecutive bit positions and store that information in the compressed program. Compression ratio will depend on how many bit changes (and length of each consecutive change) are considered during compression. Figure 3 shows the same example compressed using one

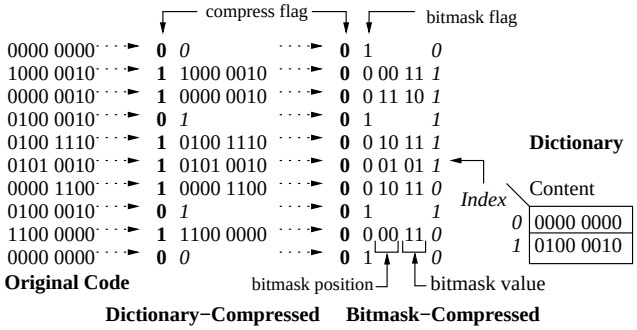

Figure 3: Dictionary and bitmask based compression

bitmask allowing 2 consecutive bit changes starting at even locations. In this example we are able to compress all the mismatched words using smaller number of bits and achieve compression ratio of 87.5%. Figure 4(b), (c), and (d) show the encoding format used by these techniques for a w-bit program. In general, the compression ratio depends mainly on the instruction width, dictionary size and the number of bitmasks used. A smaller instruction size results in more direct matches whereas increasing the number of words to compress. A larger dictionary size can match more words replacing them with dictionary index but at the cost of increased dictionary index bits. More number of bitmasks results in more compressed words at the same time requiring more bits to encode the bitmask information. Our approach splits the wider control words to achieve better redundancy by employing multiple dictionaries.

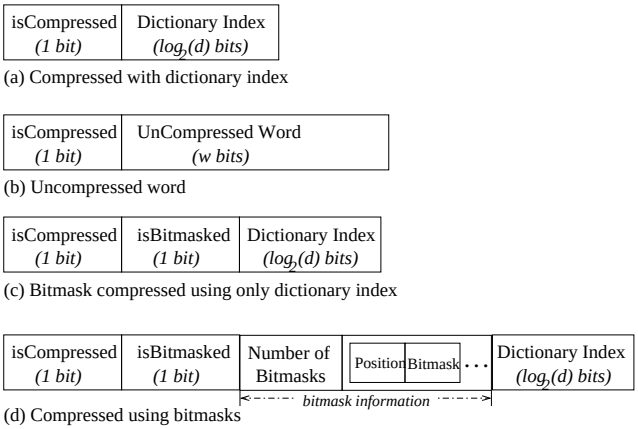

Figure 4: Encoding formats of existing compression techniques

4. CONTROL WORD COMPRESSION

The existing bitmask-based compression is promising but there are various challenges discussed in the previous section that needs to be addressed. NISC control words are usually 3 to 4 times wider than normal instructions. 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. Then don't care bits are resolved using vertex coloring. The resultant control words are then scanned for less frequent and constant bits. These infrequent bits are then encoded as a skip map. Then each slice is compressed using bitmask based algorithm described in article [9] by selecting profitable parameters. The parameters are selected based on the length of each slice.

Algorithm 1 lists the major steps in compressing NISC control words. Initially in step 1, 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 from compression, and are hardcoded). In step 2, the input is split into required slices. The less frequently changing bits are then removed from each slice using Algorithm 3. For each slice, don't care values are resolved using Algorithm 2 in step 3.1. The resultant slices are compressed in step 3.2, using a combination of run length encoding (RLE) and bitmask based compression [9]. Step 4 returns the compressed control words.

Algorithm 1: Multi-dictionary compression

Input: i) control words with don't cares, I
ii) number of slices n
iii) threshold bits that can change t
Output: Compressed control words C
1. W = remove_constant_bits (I)
2. $S[]$ = slice_and_remove_less_frequent_bits (W, n, t)
3. **forall** s *in* $S[]$ **do**
   3.1 $S[i]$ = bitmask_aware_dont_care_resolve (s)
   3.2 $C[i]$ = RLE_bitmask_compress ($S[i]$)
end
4. Return C

The complete compression and decompression methodology of control words is shown in Figure 5. In this example the input control word is split into three slices. The input file containing the control words is passed to the compressor. The compressor reduces the control word size by applying the Algorithm 1 and produces the compressed file in the order of slices. Later each decoder fetches compressed words from different locations in the memory. These compressed words are then decoded using the dictionary stored on block RAM (BRAM). The decompressed control word is then assembled to form the original control word.

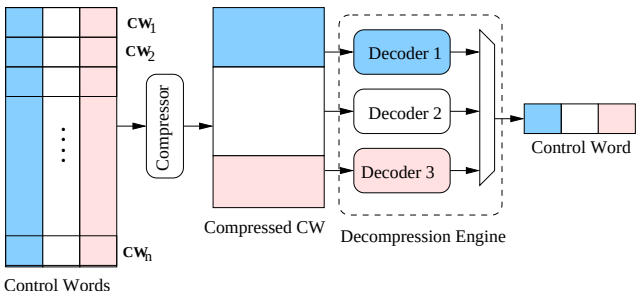

Figure 5: Compressed control word compression

4.1 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 [7] inserts don't care bits in such control words. To obtain maximum compression any compression algorithm can utilize these don't care values efficiently. One such algorithm presented in [3] creates a conflict graph with nodes representing unique control words and edges between them represent that these words cannot be merged. Application of minimal k colors to these vertices 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 [5] is applied to color the vertices to obtain merged dictionary. This al-

gorithm is well suited in reducing the dictionary size with exact matches. The dictionary chosen by this algorithm might not yield a better bitmask coverage.

An intuitive approach is to consider the fact that the dictionary entries will be used for bitmask based matching during compression. Algorithm 2 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 that 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 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 are retained. This promises reduced size as they result in more direct matches.

Algorithm 2: Bitmask Aware Don't Care Resolution

Input: i) Unique input control words $C = \{c_i, f_i\}$,
ii) 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_conflict (u,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
Return M

Figure 6 describes an example don't care resolution of NISC control words and a merging iteration. The input words and their frequencies are as shown in Figure 6 (a). There are four inputs A, B, C and D. Figure 6 (b) represents the conflict graph constructed by the don't care resolution algorithm in [3]. The algorithm chooses three colors which represents the merged dictionary entries. Our approach skips the edges which can be bitmasked as shown in Figure 6 (c). This example uses one 1-bit bitmask to store differences. The final colors indicate the merged dictionary entries. The traditional approach will require three dictionary entries, whereas our approach requires only two dictionary entries that results in savings of one bit in dictionary index.

4.2 Encoding Less 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 frequently changing bits are encoded by using an 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 3 lists the steps in eliminating the constant bits and less frequently changing bits. Initially the Algorithm 3 calculates the number of ones and zeros in each bit position. In the next step only those bit positions that change less than threshold t are considered to be the initial skip map. For any given control word if there are more than one bit position that change, it is not profitable to encode all these bit changes. To avoid this condition, the last step of the algorithm updates the initial skip map by constructing a conflict map for each control word. The bit position which causes the least conflict is retained for skipping.

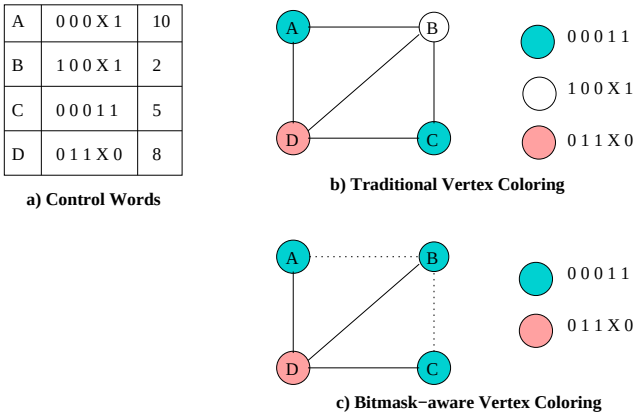

Figure 6: Bitmask aware don't care resolution

Figure 7 shows 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 bits in a skip map. Columns with bit changes less than threshold (2 in this example) i.e. column 2, 4 and 5 have less frequent bits changes. 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 of the figure. It can be noted that there is a significant reduction in control word size for compression. The decompression section discusses how these less frequently changing bits are reassembled.

Algorithm 3: Removal of Less Frequently Changing Bits

Input: i) Control Words with don't cares D,
ii) 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 bit from previously
      calculated skip_map.
end
return S

4.3 Run Length Encoding

Careful analysis of the control words pattern reveals that the in-

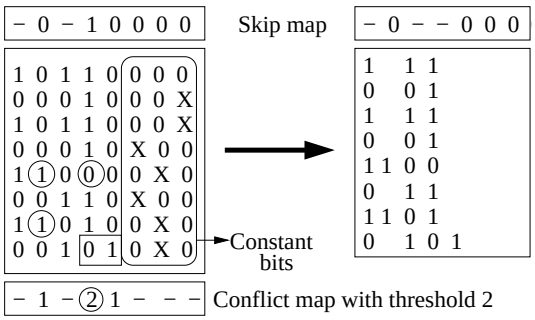

Figure 7: Removal of constant and less frequently changing bits

put control words contain consecutive repeating patterns of words. The bitmask based compression [9] encodes such patterns using same repeated compressed words. Instead we use a method in which repetition of such words are run length encoded (RLE). To represent such encoding no extra bits are needed. An interesting observation leads to the conclusion 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 on all the compressed words otherwise.

Another advantage of such run length encoding is that it alleviates the decompression overhead by providing the decompressed control word instantaneously to the configuration hardware in the same cycle. Figure 8 illustrates the bitmask-based RLE. The input contains, control word "0000 0000" repeating 5 times. In normal bitmask-based compression these control words will be compressed with repeated compressed words, whereas our approach replaces such repetitions using a bitmask of "00". The number of repetition is encoded as bitmask offset and dictionary bits combined together. In this example, the bitmask offset is "10" and dictionary index is '0'. Therefore, the number of repetition will be "100" i.e., 4 (in addition to the first occurrence). The compressed words are run length encoded only if the savings made by RLE word encoding is greater than the actual encoding. In other words, if there are r repetition of compressed words and cost of representing each control 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.

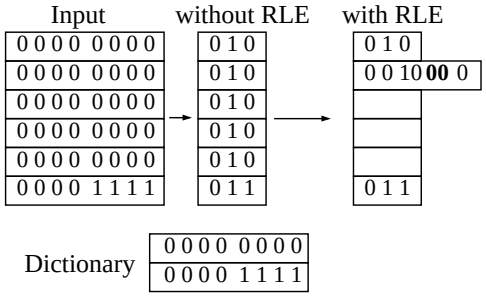

Figure 8: An illustrative example of RLE with bitmask

5. DECOMPRESSION MECHANISM

This section analyzes the modification required for the decompression engine proposed in [9]. Figure 9 describes the structure of the NISC control word decompression engine. The decompression hardware consists of multiple decoding units for each slice of compressed control words. Each decoder contains input buffer to store the incoming data from memory. 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 changing bit reduction. 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). The output from constant bit register will be connected to controller for execution.

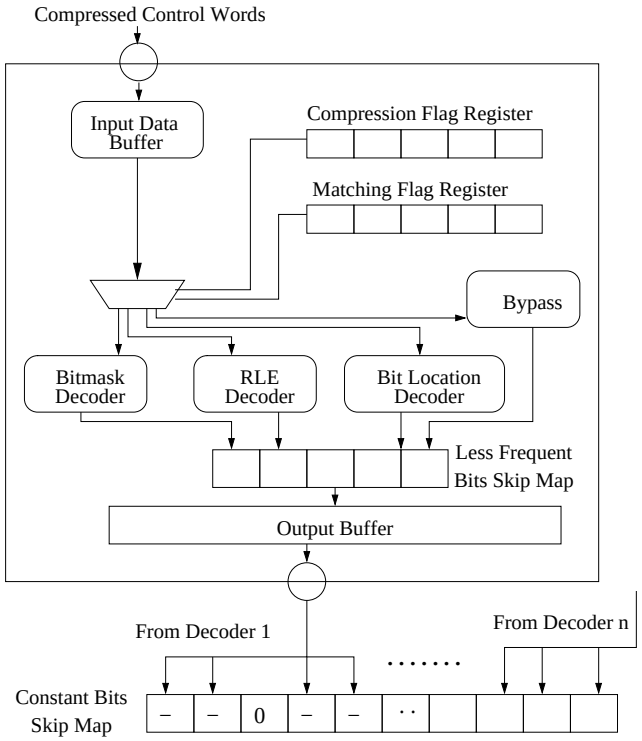

Figure 9: Multi-dictionary based decompression engine

6. EXPERIMENTS

The effectiveness of our proposed compression technique is measured using MiBench benchmarks [3]. In particular we use adpcmcoder, adpcmdecoder, crc32, dijkstra, sha, and fpmp3 programs. These application are commonly used embedded software in mobile, network, security and telecom domains. These benchmarks are also used in the existing control word compression technique [3]. We evaluate the compression ratio and resources used by decompression engine (BRAMs) against the compression approach proposed by Gorjiara et al. [3] and original bitmask based compression [9].

6.1 Compression Performance

The benchmarks are compiled in release mode using NISC compiler [7]. The profitable parameters selected for bitmask based compression are determined by the width of the control word. For example a control word between 16 and 32 bits, dictionary size of 16, two bitmasks of each 2-bit sliding is selected. For a control

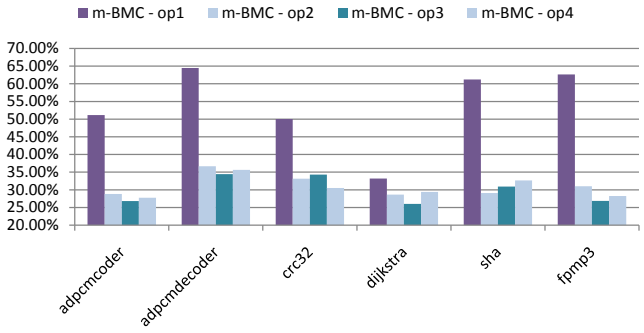

Figure 10: Our approach using multiple dictionaries

word less than 16 bits, dictionary size of 8, single bitmask of 2-bit sliding is selected for compression.

Figure 10 compares the compression ratios of our approach (m-BMC) using single (m-BMC-op1) and multiple (two: m-BMC-op2, three: m-BMC-op3, four: m-BMC-op4) dictionaries. We split the input control words equally into the number of dictionaries used. For each control word slice we select the profitable parameters mentioned above for bitmask based compression. The three dictionary option clearly outperforms the other combination except in the case of *sha* and *crc32* program, where two and four dictionary options respectively result in better compression ratio. Overall we find that using three dictionary option is better for most of the benchmarks.

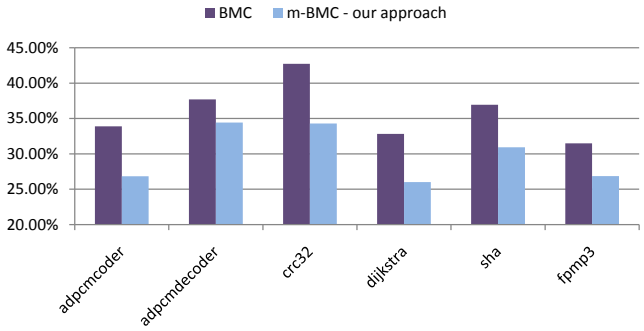

Figure 11: Bitmask-based compression versus our approach

Figure 11 compares our approach (using three dictionaries) with the existing bitmask based compression technique (BMC [9]). Our bitmask-based RLE approach combined with constant and less frequently changing bits outperforms the existing bitmask based compression method [9] by an average of 20 to 30%.

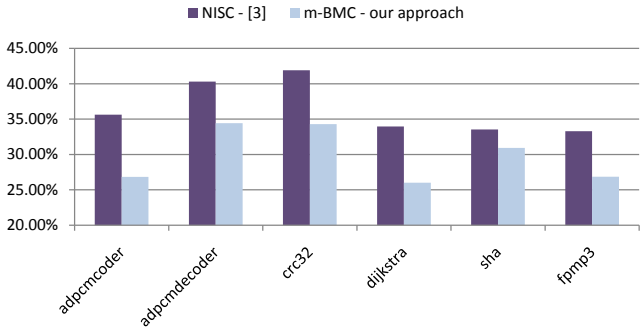

Figure 12: Our approach versus existing NISC compression

Figure 12 compares the compression ratios between the existing multi-dictionary compression technique proposed by Gorjiara et al. [3] and our approach. Both approaches use three dictionaries. On an average, our approach outperforms over NISC compression technique [3] by 15 to 20%. It is important to note that our approach uses up to six times less number of BRAMs to store the dictionary compared to [3].

6.2 Decompression Overhead

Our approach automatically generates the Verilog based decompression engine with selected compression parameters. The decompression engine can operate at the speed of the decoding units capable of operating at 100 MHz in the range of NISC processor operating range. The dictionary size used in all the benchmarks are small and limited. In our approach, the BRAM used to store these dictionaries are fixed requiring 1 or 2 BRAMs, whereas the existing method [3] uses up to six BRAMs.

7. CONCLUSIONS

This paper presented a bitmask based compression technique to reduce the size of NISC control words by splitting and compressing them using multiple dictionaries. We designed a bitmask aware don't care resolution that produces dictionary having large bitmask coverage with minimal and restricted dictionary size. We developed an efficient RLE technique that encodes the consecutive repetitive patterns to improve both compression efficiency and decompression performance. This paper also developed an efficient way of encoding constant and less frequently changing bits to significantly reduce the control word size. Our approach improved compression efficiency by 20 to 30 % over the best known compression technique [3] with no additional decompression overhead.

8. ACKNOWLEDGMENTS

This work was partially supported by NSF CAREER award 0746261. We would like to thank Dr. Bita Gorjiara and Dr. Mehrdad Reshadi for their insightful comments and suggestions about NISC technology and control word compression.

9. REFERENCES

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

[2] C. Lefurgy et al. Improving code density using compression techniques. *MICRO*, 194–203, 1997.

[3] B. Gorjiara and D. Gajski. FPGA-friendly code compression for horizontal microcoded custom IPs. *FPGA*, 108–115, 2007.

[4] H. Lekatsas and J. Henkel and V. Jakkula. Design of an one-cycle decompression hardware for performance increase in embedded systems. *DAC*, 34–39, 2002.

[5] T. Jensen and B. Toft. *Graph coloring problems*. Discrete Mathematics and Optimization. Wiley-Interscience, 1995.

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

[7] M. Reshadi. No-Instruction-Set-Computer (NISC) technology modeling and compilation. *PhD thesis*, UC Irvine, 2007.

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

[9] S. Seong and P. Mishra. Bitmask-based code compression for embedded systems. *IEEE Trans. CAD*, 27(4): 673–685, 2008.