ARTICLE IN PRESS

INTEGRATION, the VLSI journal ∎ (∎∎∎∎) ∎∎∎–∎∎∎

Contents lists available at SciVerse ScienceDirect

INTEGRATION, the VLSI journal

journal homepage: www.elsevier.com/locate/vlsi

Bitmask aware compression of NISC control words ☆

Kanad Basu *, Chetan Murthy, Prabhat Mishra

Department of Computer and Information Science and Engineering, University of Florida, United States

ARTICLE INFO

Article history:
Received 15 November 2010
Received in revised form
15 February 2012
Accepted 16 February 2012

Keywords:
No-instruction-set compiler
Compression

ABSTRACT

It is not always feasible to implement an application specific custom hardware due to cost and time considerations. No instruction set compiler (NISC) architecture is one of the promising directions to design a custom datapath for each application using its execution characteristics. A major challenge with NISC control words is that they tend to be at least 4–5 times larger than regular instruction size, thereby imposing higher memory requirement. A possible solution to counter this is to compress these control words to reduce the code size of the application. This paper proposes an efficient bitmask-based compression technique to drastically reduce the control word size while keeping the decompression overhead in an acceptable range. The main contributions of our approach are (i) smart encoding of constant and less frequently changing bits, (ii) efficient do not care resolution for maximum bitmask coverage using limited dictionary entries, (iii) run length encoding to significantly reduce repetitive control words and (iv) design of an efficient decompression engine to reduce the performance penalty. 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 25–35%. In addition, our technique only requires 1–3 on-chip RAMs, thus making it suitable for FPGA implementation.

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

Desktop-based systems use general purpose processors to execute a wide variety of application programs. However, it is not always profitable to run an application on a generic processor. On the other hand the implementation of a custom hardware may not be feasible due to cost and time constraints. One of the promising directions is to automatically design a custom datapath for each application using its execution characteristics and the design constraints. The datapath is synthesized and then the application is compiled on it. Any future change in the application is accommodated by just recompiling the application on the datapath. No Instruction Set Compiler (NISC) [3] is one such technology that is used in statically scheduled Horizontal Microcoded Architectures (HMA). The abstraction of instruction set in generic processors limits from choosing such custom data path. NISC, which operates on HMAs, promises faster performance guarantees by skipping the abstraction phase and directly compiling the program to microcodes. It also controls the selection of optimal datapath to meet the application's performance requirements and hence provides a maximum utilization of the datapath resources. Therefore, parallel architectures can be built on top of these despite the complexity of the controller or the hardware scheduler. However, these datapath or control words (CW) tend to be at least 4–5 times wider than regular instructions thus increasing the code size of applications. One promising approach is to reduce these control words by compressing them.

Fig. 1 shows the compressed control word execution flow on a NISC architecture. It should be remembered that these control words are specific to a particular datapath. The compressed control word is read from control word memory (CW Mem) and decoded to obtain the original control word and sent to controller for execution. *Compression ratio* is the metric commonly used to measure effectiveness of a compression technique, as shown in Eq. (1). Clearly, smaller the compression ratio better the compression efficiency:

$$\textit{Compression Ratio} = \frac{\textit{Compressed Code Size}}{\textit{Uncompresssed Code Size}} \tag{1}$$

One may argue that introduction of compression to offset the code size increase can nullify the advantages of using NISC

☆ This is an extended version of the paper that appeared in ACM Great Lakes Symposium on VLSI (GLSVLSI), 2009 [1]. The GLSVLSI paper presented initial results on applying bitmask based compression on NISC control words. This paper has significant additional material, in particular: (i) The decompression mechanism is discussed in detail in Section 5, with focus on the overall decompression scheme as well as individual bitmask based decoders, (ii) Section 6.5 compares the decompression overhead (both memory and performance) of our method and that proposed by [2]; (iii) We have also analyzed the effectiveness of each of the steps in the overall compression in Sections 6.3 and 6.4; (iv) A mathematical framework has been developed in Section 4.5 to derive a condition when the application of RLE is helpful in compression.

* Corresponding author.
E-mail addresses: kanad@ufl.edu, kbasu@cise.ufl.edu (K. Basu), cmurthy@cise.ufl.edu (C. Murthy), prabhat@cise.ufl.edu (P. Mishra).

0167-9260/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.vlsi.2012.02.004

Please cite this article as: K. Basu, et al., Bitmask aware compression of NISC control words, INTEGRATION, the VLSI journal (2012), http://dx.doi.org/10.1016/j.vlsi.2012.02.004

ARTICLE IN PRESS

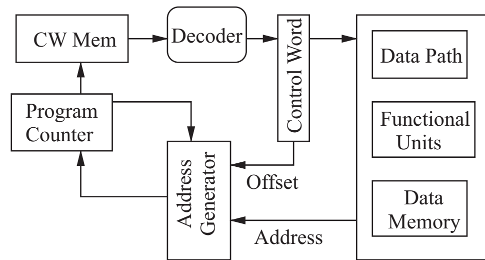

Fig. 1. NISC architecture and decoder placement.

technology. In other words, NISC improves the performance by removing the need for decoding control words, whereas use of compression re-introduces it to decode the compressed control words. It is a major challenge to employ an efficient compression technique to achieve best possible control word size reduction while employing a simple decoder to minimize the performance penalty. As the experimental results demonstrate, our method can provide significant code size improvement with moderate performance penalty.

Compression techniques may vary widely depending on the encoding approaches such as complex statistical coding or simple dictionary based compression. It is known that although statistical codings (such as Huffman coding) can provide a good compression, the decompression penalty is huge. A complex decompression architecture means longer running time for applications as well as additional power wastage. Hence, these methods may not be suitable in many systems with real time and other design constraints. Therefore, widely used code compression techniques utilize dictionary based compression. A dictionary is used to store the most frequently occurring instructions. The instructions are replaced with smaller dictionary indices. The application of such algorithm on NISC control words is not profitable. This is because the uncompressed control words are themselves long in length (typically around 90 bits [2]). Therefore, the probability that two words exactly match is $(\frac{1}{2})^{90}$, which is indeed a very low probability. As a result, control words replacing the original instructions might not retain the same repeating pattern. Gorjiara et al. [2] described an interesting approach in which the control words are split to obtain redundancies and then compressed using multiple dictionaries. The main disadvantage of their technique is that all the unique entries are stored in dictionary without considering any masking opportunity for mismatches. This leads to large compressed code (inefficient compression) and variable length dictionary size that requires variable number of on-chip block RAMs, in FPGA implementations where these dictionaries are stored.

In code compression domain, Seong and Mishra [4] 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. Thus, codes which mismatch the dictionary entries by small number of bits can also be compressed. Our approach is motivated by bitmask based compression (BMC) [4]. The direct application of BMC 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.

In this paper, we propose an efficient compression technique that takes advantage of both NISC control word compression [2] and bitmask-based compression [4] techniques. This paper makes five important contributions: (i) an efficient *NISC* control word compression technique to improve compression ratio by splitting control words and compressing them using multiple dictionaries, (ii) smart encoding of constant and less frequently changing bits to further reduce the control word size, (iii) a bitmask aware do not care resolution to improve dictionary coverage, (iv) run length encoding of repetitive sequences to both improve compression ratio and decrease decompression overhead by providing the uncompressed words instantaneously and (v) an efficient decompression engine to reduce the overall decompression overhead. Our algorithm is also easily implementable in FPGA, since it has reduced the on-chip BRAM requirements to 1–3, which was almost nine in the case of [2] for storing dictionaries, thus reducing the additional memory penalty to a large extent compared to [2].

The rest of the paper is organized as follows. Section 2 surveys 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 the decompression architecture in Section 5. Section 6 presents our experimental results. Finally, Section 7 concludes the paper.

2. Related work

Dictionary-based compression methods are widely used in many application domains [4]. In dictionary based compression, a dictionary stores the patterns with maximum frequency and the dictionary indices replace the actual code during compression. The first code-compression technique for embedded processors was proposed on similar lines by Wolfe and Chanin [5]. Their technique uses an amalgam of dictionary based compression and Huffman coding; 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. Their compression mechanism was further improved by IBM's CodePack [6]. A statistical method for code compression using arithmetic coding and Markov model was proposed by Lekatsas and Wolf [7]. Lekatsas et al. [8] also 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 Lefurgy et al. [9], Liao et al. [10] among others. Seong and Mishra [4] proposed a bitmask-based compression to improve compression ratio by creating matching patterns using bitmasks. A code compression scheme can be further classified into pre-cache and post-cache compression depending on the placement of the decompression unit. In a pre-cache scheme, the decompression engine is placed between the main memory and the cache, while in post-cache, it is placed between the cache and the processor. Pre-cache techniques have the advantage of low decompression overhead, since the decompression engine is fetched only when there is a cache miss. Post-cache schemes, on the other hand are useful since their cache sizes can be kept small, or an equal size cache can store larger amount of data.

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 [11] splits a VLIW instruction into multiple fields, and each field is compressed by using a dictionary-based scheme. However, instruction packing reduces the efficiency of these approaches, which was further improved by Ros and Sutton [12] by combining nearly identical instructions into a single entry dictionary. Mismatches were also considered by Prakash et al. [13]. Gorjiara et al. [2] applied similar approach as [11] by splitting the control words into different fields and compressing them using multiple dictionaries. Nam et al. [14] also used a dictionary based scheme to

Please cite this article as: K. Basu, et al., Bitmask aware compression of NISC control words, INTEGRATION, the VLSI journal (2012), http://dx.doi.org/10.1016/j.vlsi.2012.02.004

ARTICLE IN PRESS

compress VLIW instructions. Various techniques like Huffman based compression, Tunstall coding, LZW compression on VLIW instructions were used by Larin et al. [15], Xie et al. [16] and Lin et al. [17], respectively.

Recently, dictionary based compression techniques have been applied for NISC control words by Gorjiara et al. [2]. Their technique follows the same line of approach as [11]. They first split each control word into different fields and compress them using multiple dictionaries. However, their approach can lead to unacceptable compression since it stores all the unique binaries in the dictionary. Our method, on the other hand, selects the binaries needed for maximum compression and hence requires a smaller dictionary. Moreover, their method will require variable number of block RAMs (BRAM) to store variable-length dictionaries for different applications. Our approach outperforms [2] on both fronts by achieving 20% better compression (on average) and huge reduction in the number of RAMs required using a fixed-length dictionary.

The approach proposed by Seong and Mishra [4] is useful for NISC control word compression. 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 do not cares in input control words. As a result, it will sacrifice on compression efficiency by randomly replacing do not cares by 0's or 1's. These two methods [2,4] are closest to our approach. Section 6 presents experimental results to show how our method improves compression efficiency compared to these approaches, with minimal impact on 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 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 [3] 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 (CWs) are stored in the control word memory (CW Mem, shown in Fig. 1) in programmable IPs, while they are synthesized to lookup-table logic in hardwired dedicated IPs.

The NISC toolset [18] can be used by generating or specifying an architecture description and then running the application on the datapath obtained. The datapath can have components like registers, register files, multiplexers and functional units. Each control signal in the datapath is represented by a field in the control word. The bits of the control word may be '0', '1' or 'X', where 'X' represents a do not care. A major problem in compressing the control words is to efficiently assign values to the do not cares to enable efficient compression.

Fig. 2 shows CISC, RISC and NISC architectures. In CISC architecture, microcoded instructions are used. The Program Counter indexes the program memory; the data from the program memory is then indexed by the micro Program Counter (mPC) to index the micro-Program Memory (mPM). The primary concern with this architecture is that the instructions are too complex, consisting of a combination of many control words (microcodes). 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 directly stores the control word in the program memory thus eliminating the instruction decoder and hardware scheduler as shown in Fig. 2(c). 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. As can be seen in [2], when the code size and cycle number on running the MiBench benchmarks on the two processors, GNISC and Xilinx MicroBlaze [19] are compared, NISC implementation runs 5.54 times faster than RISC-based MicroBlaze, while its code size is four times larger [2]. The large code size increases the size of required control memory in programmable IPs, and the area of control logic in dedicated IPs. The goal of control word compression technique is to reduce the control word size of NISC processors while maintaining the performance benefits.

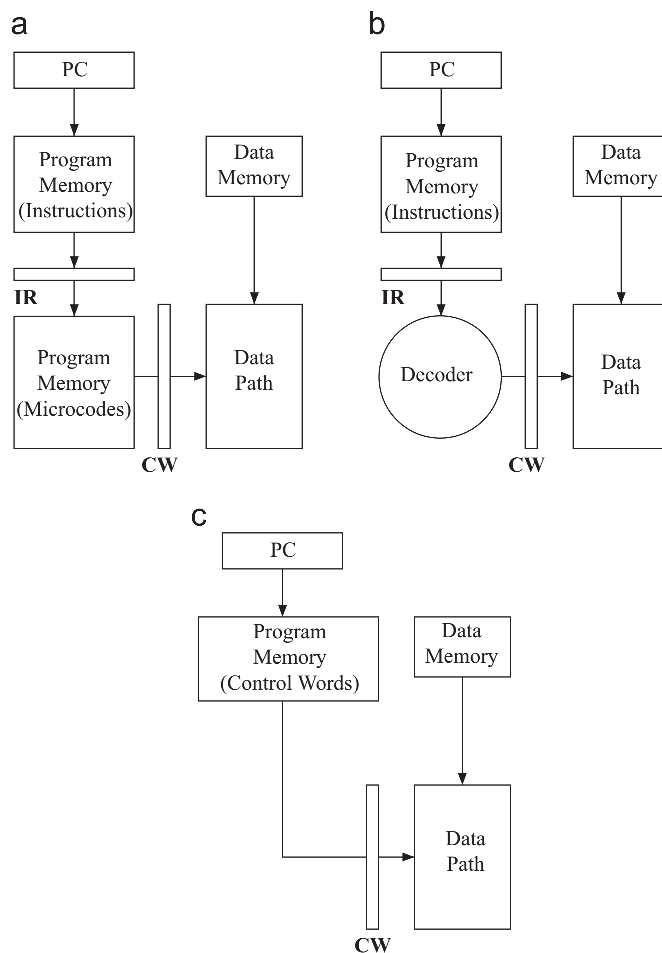

Fig. 2. CISC, RISC and NISC Architectures. (a) CISC, (b) RISC, and (c) NISC.

3.2. Bitmask-based compression

Bitmask based encoding improves the compression efficiency of traditional dictionary based compression. Fig. 3 shows an example of traditional dictionary based compression using a 2 entry dictionary. This example has been used by [4]. Here the dictionary size is of 2 instruction sequences, that is 16 bits. The two most frequently occurring sequences are stored in the dictionary and are replaced with the dictionary index (1 bit)

Please cite this article as: K. Basu, et al., Bitmask aware compression of NISC control words, INTEGRATION, the VLSI journal (2012), http://dx.doi.org/10.1016/j.vlsi.2012.02.004

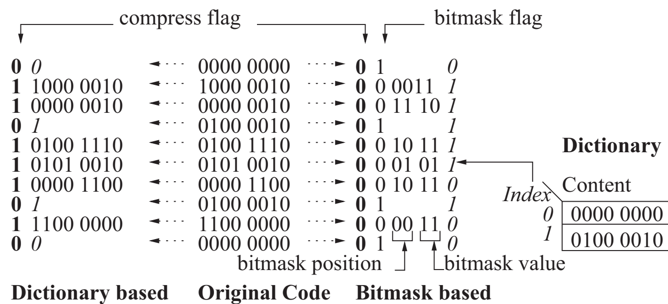

Fig. 3. Dictionary and bitmask-based compression, similar to [4].

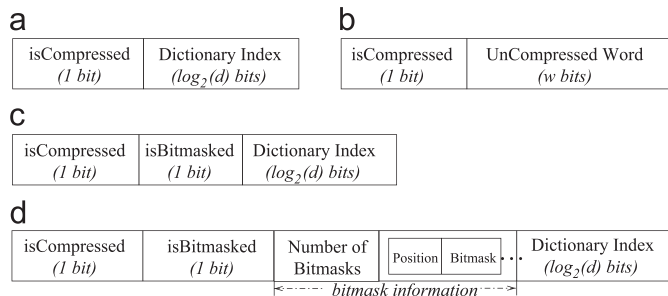

Fig. 4. Encoding formats of existing compression techniques. (a) Compressed with dictionary index, (b) uncompressed word, (c) bitmask compressed using dictionary index, and (d) compressed using bitmasks.

during compression. The compressed program consists of both indices and uncompressed instructions. The first bit represents whether a string has been compressed or not. A '0' represents that it is compressed while a '1' represents an uncompressed sequence. If the sequence is compressed, it is followed by the dictionary index corresponding to the matching pattern. Otherwise, the entire uncompressed sequence follows the first '1' bit. Fig. 4(a) and (b) show the encoding formats for traditional dictionary based compression. In this example, the dictionary based compression achieves a compression ratio of 97.5%.

Seong and Mishra [4] improved 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. Fig. 3 shows the same example compressed using one bitmask allowing two consecutive bit changes starting at even bit positions. As in the case of dictionary based compression, the first bit signifies whether an instruction is compressed or not. If not compressed, the entire uncompressed instruction follows it. If compressed, there may be two cases. The instruction may be compressed by direct matching with dictionary or it might have been compressed using bitmask (that is, there were mismatches present). A '1' in the second bit signifies the first case, while a '0' in the second bit signifies the second case. In the former case, the matching dictionary entry follows the '0'. When compressed by bitmask, the '1' is followed by bits to signify the bitmask position (2 bits in this case, since there are 4 even bit positions in a 8-bit string) and then the bitmask (2-bit bitmask in this case). The index to the nearest matching dictionary entry follows at the end. By nearest matching, we mean the dictionary entry which can compress the string with minimal number of bitmasks. Fig. 4(b)–(d) show the encoding format used by these techniques. In the example in Fig. 3 we are able to compress all the mismatched words using smaller number of bits and achieve compression ratio of 87.5%.

4. Control word compression using bitmasks

The existing bitmask-based compression is promising but there are various challenges discussed in the previous section that needs to be addressed. Fig. 5 shows the overview of our approach. NISC control words are usually 3–4 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. The control words are scanned for less frequent and constant bits. The constant bits are those that do not change with sequences. As a result, compressing them becomes easy, since all we have to do is to remember those bits once. The infrequent bits are then encoded as a skip map. A skip-map is one which keeps account of never changing bit positions or those which rarely change. A bit position which remains constant for all control words need not be taken into account when compressing the control words. Instead, that bit position can be just encoded once and remembered for all control words. The control words contain do not cares along with '0's and '1's. The do not cares can assume any value, which makes their existence important for compression purpose. It is important to assign values to do not cares carefully that make them suitable for compression with higher compression efficiency. We resolve the do not care bits using vertex coloring. Then each slice is compressed using bitmask based algorithm by selecting profitable parameters.

Algorithm 1 lists the major steps in compressing NISC control words. Initially in step 1, the input is split into required slices as discussed in Section 4.1. For each slice, the constant and less frequently changing bits are removed to get reduced control

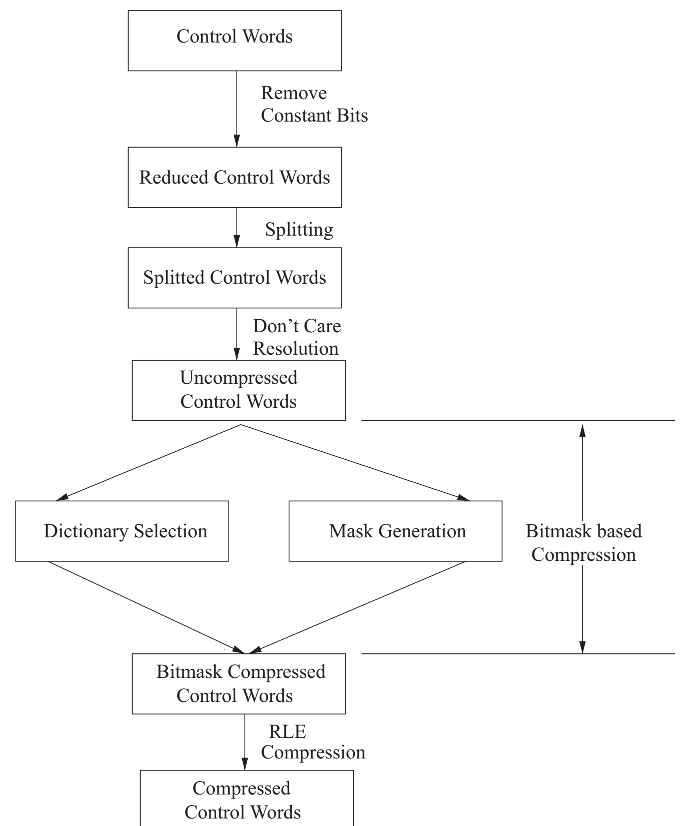

Fig. 5. Compression of NISC control words.

Please cite this article as: K. Basu, et al., Bitmask aware compression of NISC control words, INTEGRATION, the VLSI journal (2012), http://dx.doi.org/10.1016/j.vlsi.2012.02.004

ARTICLE IN PRESS

words along with an initial skip map (Section 4.2).[1] For each slice, do not care values are resolved using Algorithm 3 as described in Section 4.3. The resultant slices are compressed using a combination of bitmask based compression [4] (Section 4.4) and Run Length Encoding (RLE) (Section 4.5). Finally, the compressed control words are returned. The remainder of this section describes each of these steps in detail.

Algorithm 1. NISC control word compression.

Input: (i) control words with do not cares, I
(ii) number of slices n
(iii) threshold bits that can change t
Output: Compressed control words C
1. $S[]$=slice_the_control_words(I)
2. **forall** s in $S[]$ **do**
  2.1 $W[i]$=remove_constant_and_less_frequent_bits(s)
  2.2 $S[i]$=bitmask_aware_dont_care_resolve ($W[i]$)
  2.3 $C[i]$=bitmask_RLE_compress ($S[i]$)
end
3. Return C

4.1. NISC word slices

As discussed earlier, each NISC control word is almost 90-bit wide. As a result, it may not have the redundancy required for profitable bitmask based compression. In order to introduce more redundancy and matching among the control words, they are split into slices. Since each slice is smaller than the original control word, it is easier to match with a dictionary entry. Even if two slices are slightly different, they can be matched using bitmasks. Splitting of control words is illustrated in Fig. 6. In this example, the input control word is split into three slices. The input containing the control word slices is passed to the compressor. The compressor reduces the control word size by applying the Algorithm 1 and produces the compressed file. For example, compressor 1 encodes the first slice of every control word. 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 from the slices to form the original control word.

4.2. Encoding less frequently changing bits

A detailed analysis of the control word sequence reveals that some bits are constant or change less frequently throughout the code segment. Removal of such bits improves compression efficiency and does not affect matches provided by rest of the bits. Those bits can be remembered only once and need not be repeatedly encoded for each and every sequence. 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 2 lists the steps in eliminating the constant bits and less frequently changing bits. Initially the number of ones and zeros in each bit position is calculated. In the next step only those bit positions that change less than threshold t are considered in the initial skip map. A *skip map* maintains the positions and corresponding values that remain constant or almost constant for a set of binary instructions.

[1] Skip map represents the bits that can be skipped from compression, and are hardcoded.

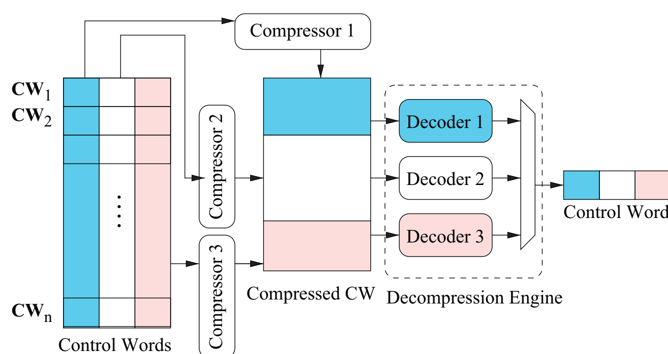

Fig. 6. Slicing of NISC control words for compression.

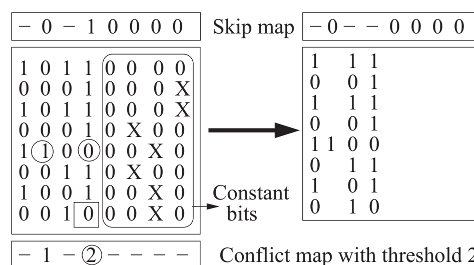

Fig. 7. Removal of constant and less frequently changing bits.

For any given control word if there are more than one bit position changes, it is not profitable to encode all these bit changes. Taking all these changes into account will not only complicate the encoding process, but will also lead to decompression overhead. 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.

Algorithm 2. Removal of less frequently changing bits.

Input: (i) Control Words with do not cares D,
(ii) Threshold t number of bits
Output: Skip Map S
$S=\phi$
forall w in D **do**
  forall b_i, *ith 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

Fig. 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 four 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 has less frequent bits changes. In the final step conflict map is created (listed at

Please cite this article as: K. Basu, et al., Bitmask aware compression of NISC control words, INTEGRATION, the VLSI journal (2012), http://dx.doi.org/10.1016/j.vlsi.2012.02.004

ARTICLE IN PRESS

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 need to be encoded are shown on the right side of the figure. In this example, we have converted do not cares into the constant bits with ‘0’s. The code words with conflicts are taken care of using a conflict-map as shown in Fig. 7. For example, in the second column, the constant-bit is 0, while the fifth code word from the top has a 1 in that position. The particular bit and its position are remembered in the conflict-map. During decompression, when the skip-map is considered, the conflict map takes care of such conflict bit positions in the code words. It should be noted that there is a significant reduction in control word size for compression.

4.3. Bitmask-aware do not 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 [3] inserts do not care bits in such control words. These do not care values can take either a value of ‘0’ or ‘1’. To obtain maximum compression, a compression algorithm can utilize these do not care values efficiently, that is, they should be assigned values to assist in maximal compression. One such algorithm presented in [2] creates a conflict graph with nodes representing unique control words and edges between them representing 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 [20] is applied to color the vertices to obtain a dictionary. This algorithm is well suited in reducing the dictionary size with exact matches. However, the dictionary chosen by this algorithm might not yield a better bitmask coverage, since the dictionary entries were selected based on direct match and do not take the benefit of matching using bitmask.

An intuitive approach is to consider the fact that the dictionary entries will be used for bitmask based matching during compression. Algorithm 3 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. These bits that can be bitmasked are not used to represent edges in the conflict graph, thus allowing the graph to be colored with fewer colors. This results in dictionary size with smaller dictionary index bits and hence reduces the final compressed code size. It may be noted that if the bits are already set, merging the vertices retains the bits originating from the most frequent words. This promises reduced code size as they result in more direct matches.

Fig. 8 describes an example of do not care resolution of NISC control words and a merging iteration. The input words and their frequencies are as shown in Fig. 8(a). There are four inputs A, B, C and D. Fig. 8(b) shows the scenario when we try to compress them using traditional vertex coloring scheme. As can be seen, three different colors are needed, which results in three dictionary entries. On the other hand, as shown in Fig. 8(c), our method utilizes the bitmask based compression mechanism in vertex coloring. As a result, we could reduce the number of dictionary entries to 2.

Algorithm 3. Bitmask aware do not 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*

4.4. Bitmask based compression

After the reduction of control words by removal of constant bits, we are left with the reduced portion of the control words that need to be compressed. These sequences of control words are then compressed using bitmask based compression as we discussed in Section 3.2. 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 but

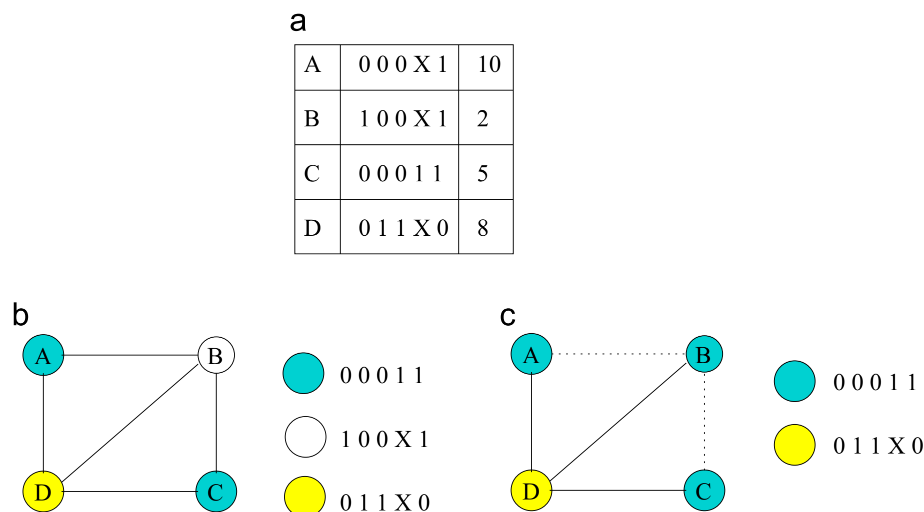

Fig. 8. Bitmask aware do not care resolution. (a) control words, (b) traditional vertex coloring, and (c) Bitmask-aware vertex coloring. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Please cite this article as: K. Basu, et al., Bitmask aware compression of NISC control words, INTEGRATION, the VLSI journal (2012), http://dx.doi.org/10.1016/j.vlsi.2012.02.004

ARTICLE IN PRESS

increases 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. A larger number of bitmasks results in more compressed words at the same time requiring more bits to encode the bitmask information. Increasing or decreasing any of the aforementioned parameters might result in poor compression efficiency. Decreasing any of these will result in smaller number of words that can be compressed. On the other hand, increasing any of these might lead to an increase in the length of the compressed words, to the extent that the final size of the compressed sequence of instructions may exceed the original uncompressed code size, thus nullifying the advantages of compression. Our approach splits the wider control words to achieve better redundancy by employing multiple dictionaries.

The number of dictionary entries and the number and type of bitmasks that are used to compress are decided first. For the Mediabench benchmarks, the length of each control words was around 90 bits, which when divided into three slices, resulted in each slice having 30 bits. For compressing these, we use eight dictionary entries per slice and at most two bitmasks (double bits) to compress them. The masks can be fixed or sliding. The lemma presented in [21] shows that our usage of 2-bit masks is appropriate in this case. The dictionary selection algorithm follows in line with that of [4]. We utilize a bit-saving dictionary selection method where each node, representing each data set, stores a bit-saving representing its frequency. Edge between any two nodes stores the bit-saving due to bitmask. The bit-saving distribution of each nods is calculated, and the most profitable nodes are included in the dictionary. This algorithm helps in providing with a beneficial dictionary.

4.5. Run length encoding

Careful analysis of the control words reveals that they contain consecutive repeating patterns. These control words will all match to the same dictionary entry and hence, the bitmask based compression [4] encodes such patterns using same repeated compressed words. Instead we use a method in which such repetitions are compressed using run length encoding (*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. For example, if we are using 2-bit masks, a mask of 00 simply signifies that the corresponding bits have never changed, thus nullifying the utility of such masks. Similar conditions can be obtained for higher length masks as well. Using this as a special marker, these repetition of control words can be encoded. Thus, a single compressed word can take care of all the compressed control words.

Fig. 9 illustrates the bitmask-based RLE. The input contains control word "0000 0000" repeating five 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 is greater than the actual encoding. Compressing using bitmasks without RLE requires 15 bits, however, our approach using RLE requires 10 bits, thus providing a saving of 5 bits.

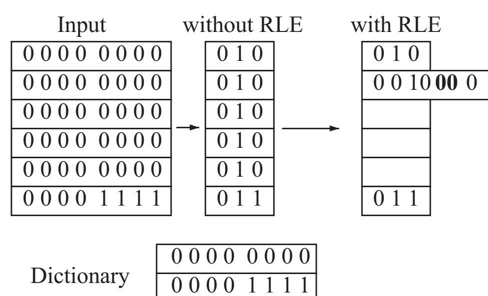

Fig. 9. An illustrative example of RLE with bitmask.

Application of RLE for compression might not always be beneficial. For example, when there are only a few repetitions, it is profitable not to employ RLE, since more bits may be needed to compress using RLE than ordinary bitmask based compression. It is obvious that RLE is only useful when the final length of the compressed code is less than the compressed code length without the application of *RLE*. We can mathematically derive the condition when application of RLE would be beneficial. Let there are $(n+1)$ repeating patterns in the compressed code, each consisting of x bits. The initial compressed data before application of RLE is $n \times x$.[2] Application of RLE should decrease the length of this compressed data, and hence, the final data length should be less than $n \times x$. In bitmask based compression scheme, let b denote the number of bits comprising the mask. For a 2-bit mask, $b=2$; let d be the number of bits to represent the dictionary, for a 64 entry dictionary, $d=6$; and let y be the number of bits used to represent the bitmask position numbers—for example, if each of the control words are of length 32 bits and there are 2-bit masks only on even-bit positions, then $y=4$. Therefore, using *RLE* compression, the bits required to compress a number of consecutive words using one RLE word is $l=y+d$. The number of repetitions n can be represented as

$$n = m \times (2^l - 1) + r \tag{i}$$

where m and r are non-negative integers. The maximum number of words that can be compressed using one RLE string is 2^l-1. Clearly, if possible, this case requires application of RLE m times for the first part of right hand side of Eq. (i), i.e., $m \times 2^l-1$. The number of bits used to represent 2^l-1 strings in *RLE* is $2+l+b$. Therefore, it is important to note that each application of RLE needs to satisfy the following inequality, where the right hand side represents the cost of using RLE and the left hand side the cost when RLE is not used.

$$(2^l - 1) \times x > 2 + l + b$$

For the last part of Eq. (i), i.e., r, RLE would be profitable if $r \times x > 2+l+b$.

5. Decompression mechanism

This section describes the decompression architecture used in NISC control words decompression. The basic decoding unit follows the same structure as [4]. This is shown in Fig. 10. As can be seen, the Lookup Table (LUT) provides the compressed portion of the splitted data which is then processed for decompression. Three different memory reads are necessary to read in the three slices of compressed data from three locations. Since the dictionary contents are fixed, a standard ASIC or FPGA implementation of a memory with multiple read ports (such as [22]) is required. The decompression unit first checks whether the data is compressed or not. If it is uncompressed, the data is then decompressed just by removing the first bit without any dictionary look up. If the input is compressed, there might be two

[2] It is to be noted that we intentionally dropped one pattern of length x, since, when RLE is applied, the first pattern remains unchanged.

Please cite this article as: K. Basu, et al., Bitmask aware compression of NISC control words, INTEGRATION, the VLSI journal (2012), http://dx.doi.org/10.1016/j.vlsi.2012.02.004

ARTICLE IN PRESS

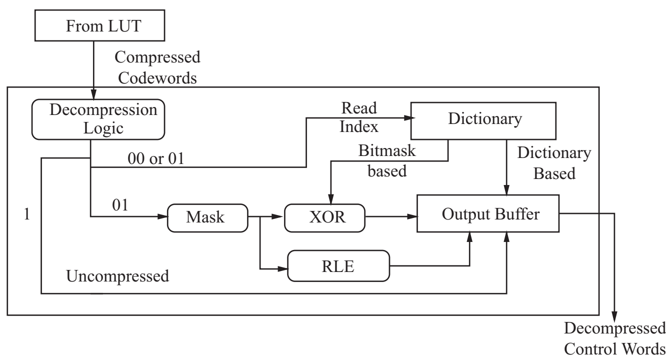

Fig. 10. Decompression of bitmask compressed data.

cases. It is either compressed using simple dictionary or using bitmask. Both cases are dealt separately as shown in Fig. 10. It is to be noted that for bitmask based compression, a parallel dictionary look up follows along with the mask generation. This parallel lookup provides a faster decompression, thus reducing the overall performance penalty. The special case of RLE decoder is shown in the figure, where the mask '00' indicates the presence of RLE. When handling a RLE compressed date, the decompression engine finds it being bitmask compressed. Upon looking at the *mask* value, 00, it infers that the code has been compressed using RLE. Therefore, a totally different path is utilized for decompression as shown by *RLE* in Fig. 10. In the RLE unit, it captures the number of repetitions and puts those uncompressed data in the output buffer. For example, if the compressed word "00 10 000" as in Fig. 9, is input to the decompression engine, it will look at the previous uncompressed word, that is, "00000000" and repeat it four more times.

Now, we analyze the modification required for the decompression engine proposed in [4]. The decompression hardware consists of multiple decoding units for each slice of compressed control words. Fig. 11 shows one such decoding unit. 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. In case of PC changes, once the original branch address is obtained the new control word locations (three locations for three slices) locations are obtained using branch-target table (same as *LAT* described in [5]). Such a table is also used in existing code compression techniques [4] that maintains a mapping between original address and address at compressed binary.

Our decompression architecture is different from that used by [2]. When [2] compresses the control words, they keep all the different types of entries in the dictionary, so that each control word can be matched with a dictionary entry. The primary problem of this approach is that, the dictionary size remains unknown (depending on the variation in each pattern), and hence the number of BRAMs required in FPGA implementation becomes variable. As a result, [2] have reported as high as nine BRAMs for decompression of Mediabench benchmarks. On the other hand, in our approach, the number of dictionary entries is fixed. Therefore, whatever be the input pattern, there will be the same number of bits reserved for creating the dictionary entry. As a result, the number of BRAMs are always 1–3 in our case, which is significantly less than the numbers reported by [2].

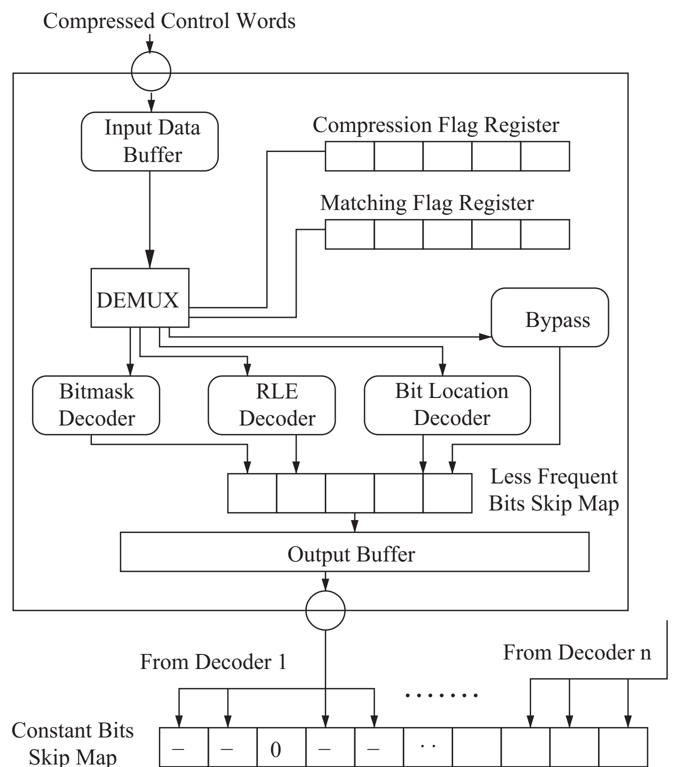

Fig. 11. Multi-dictionary based decompression engine.

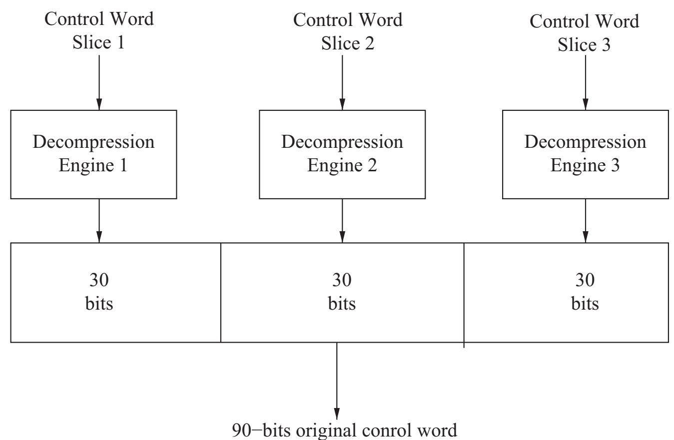

Fig. 12. Reconstruction from decompressed control words.

After all the three slices of control words are decompressed, they are appended using the mechanism shown in Fig. 12.

6. Experiments

The effectiveness of our proposed compression technique is measured using MediaBench benchmarks [2]. In particular we use adpcmcoder, adpcmdecoder, crc32, dijkstra, sha, and fpmp3 programs. These applications are commonly used embedded software in mobile, networking, security and telecom domains. These benchmarks are also used in the existing control word compression technique [2]. We evaluate the compression and decompression performance against the compression approach proposed by Gorjiara et al. [2] and original BMC [4]. We have explored the suitability of the different parameters used in different steps of our algorithm to find out the variation in

Please cite this article as: K. Basu, et al., Bitmask aware compression of NISC control words, INTEGRATION, the VLSI journal (2012), http://dx.doi.org/10.1016/j.vlsi.2012.02.004

ARTICLE IN PRESS

compression performance. Finally, we compare the decompression penalty in terms of both memory requirement and performance overhead of our approach with that of Gorijara et al. [2].

6.1. Compression performance

The benchmarks are compiled in release mode using NISC compiler [3]. The profitable parameters selected for bitmask based compression are determined by the width of the control word. For example a reduced control word between 16 and 32 bits, dictionary size of 16, two bitmasks of each 2-bit sliding is selected. For a control word less than 16 bits, dictionary size of 8, single bitmask of 2-bit sliding is selected for compression.

Fig. 13 compares our approach (using three dictionaries) with the existing bitmask based compression technique (BMC [4]). Here, we have plotted the compression ratio obtained for different Mediabench benchmarks using both the approaches. As discussed earlier, a lower compression ratio indicates a better compression performance. Our bitmask-based RLE approach (m-BMC) combined with constant and less frequently changing bits outperforms the existing bitmask based compression method [4] by an average of 20–30%. It is obvious that since BMC [4] did not use control word slicing, they had to compress larger set of control words, with less redundancy, and hence inferior compression is obtained. Also, they did not use effective techniques like bitmask aware do not care resolution or skip-maps.

Fig. 14 compares the compression ratio between the existing multi-dictionary compression technique (GNISC-opt) proposed by Gorjiara et al. [2] and our approach. Both approaches use three dictionaries. On an average, our approach outperforms NISC compression technique [2] by 15–20%.

6.2. Comparison with multiple dictionaries

Fig. 15 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 based on the number of dictionaries used. For each control word slice we select the profitable parameters mentioned in Section 6.1. The three dictionary option clearly outperforms the other combinations except in the case of *sha* and *crc32* benchmarks, 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. The best performance is obtained in the benchmark *dijkstra*, where we are able to reduce the final control word size to almost $\frac{1}{4}$th of the original size.

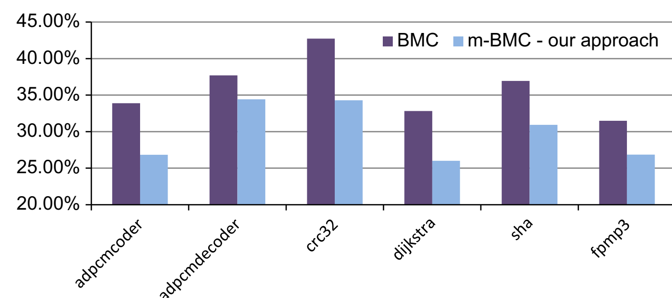

Fig. 13. Bitmask-based compression versus our approach.

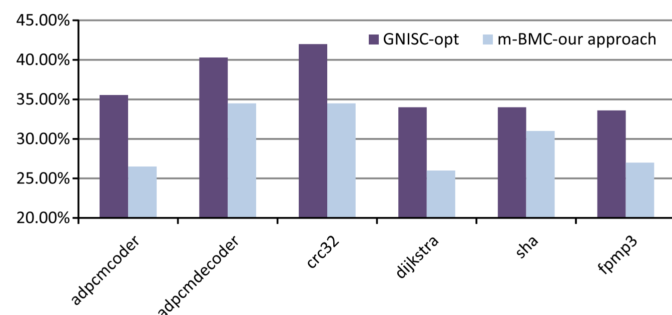

Fig. 14. Our approach versus existing NISC compression.

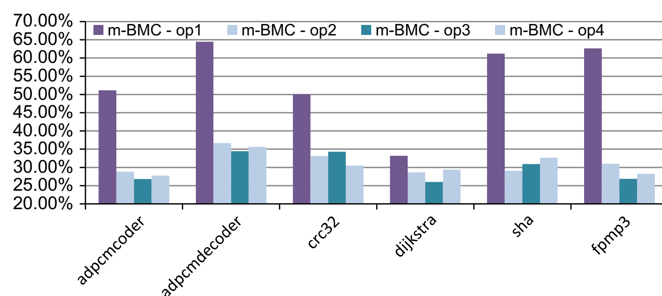

Fig. 15. Our approach using multiple dictionaries.

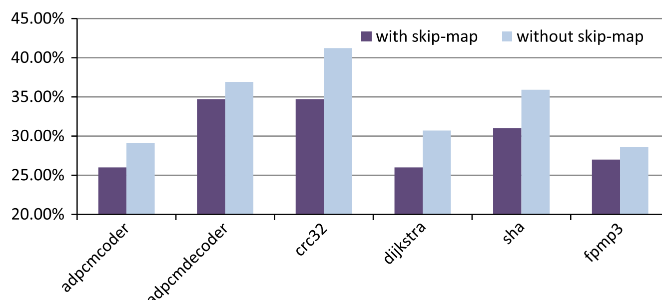

Fig. 16. Effect of skip-maps on compression ratio.

With lower number of dictionaries, it becomes difficult to find redundancies in the control words. On the other hand, with higher number of dictionaries, the number of sliced control words to be compressed increases, leading to larger number of compressed slices, and hence a direct improvement in the compression ratio.

6.3. Effect of skip-map on compression performance

Fig. 16 compares the compression performance of the proposed algorithm against the scenario where skip-map is not used. As can be seen clearly, introduction of skip-map provides a better compression performance. There are a couple of reasons behind this. First, if skip map is not used, the constant bits are being encoded over and over again for each of the control words, which adds to the compressed control word length. Also, since the constant bits are not removed, the length of the uncompressed control words increases. As a result, length of the slices also increases, thus reducing the redundancy for bitmask based compression.

6.4. Effect of do not care resolution on compression efficiency

In our algorithm, we have used a bitmask-aware do not care resolution technique as discussed in Section 4.3. In this section, we try to show the effectiveness of our do not care resolution algorithm by comparing with an equivalent compression algorithm where all the do not care bits are replaced with all '0's or all '1's. The results are shown in Fig. 17. It can be seen that our method

Please cite this article as: K. Basu, et al., Bitmask aware compression of NISC control words, INTEGRATION, the VLSI journal (2012), http://dx.doi.org/10.1016/j.vlsi.2012.02.004

ARTICLE IN PRESS

provides a better compression compared to using all ‘0’s or all ‘1’s. This is because we have carefully resolved the do not care bits so that they create maximum advantage to the bitmask based compression algorithm. However, in some cases, like *fpmp*3, the difference is not significant due to two reasons. First, even with the bitmask aware do not care resolution, those bits are converted to values ‘0’ or ‘1’. Also, none of those bits are those are not the positions where bitmasks are used.

6.5. Decompression overhead

Our approach automatically generates the Verilog based decompression engine with selected compression parameters. Since the decompression format and algorithm are fixed for all the benchmarks, we have developed generic Verilog modules to generate the decompression engine. These modules comprise of input parameters such as the number of slices and the dictionary entries. For each benchmark, these modules are linked with their respective parameters. The dictionary size used in all the benchmarks are small and limited. We used the *lsi*_10*k* target library and 180 nm technology to synthesize using Synopsys Design Compiler. The operating frequency obtained is 125 MHz.

FPGA implementation of our approach demands storing the dictionaries in memory blocks or RAMs. Logical memories are formed in FPGAs using one or more block RAMs depending on their parameters. It should be noted that to reduce the cost of FPGA implementation and to enhance the packaging capability, the BRAM requirement should be as small as possible. FPGA implementation of our approach shows that the BRAMs used to store these dictionaries are fixed requiring 1 or 3 BRAMs for the benchmarks, whereas the existing method (GNISC-opt) [2] uses up to nine BRAMs (18 Kbits per BRAM), thus improving the RAM utilization. Fig. 18 compares the number of on-chip BRAMs required in our approach and the one proposed by [2] (best results are shown for each of the benchmarks).

Addition of the decompression unit adds delay to the overall execution performance. In this case, the compressed control words from the LUT have to be decompressed first before processing it. The decompression unit may introduce execution delay in two scenarios. First, one extra cycle is required for every conditional branch without delay slot filled. This penalty is also incurred by the approach used in [2]. Additionally, in our case, each uncompressed instruction requires a delay of one clock cycle to fetch the entire instruction. The performance overhead is measured as

$$\text{Performance Overhead} = \frac{\text{Number of Extra Cycles}}{\text{Number of Actual Cycles}} \quad (2)$$

Our decompression engine incurs about 10% additional performance overhead compared to the approach proposed by Gorijara et al. [2]. In summary, compared to GNISC-opt [2], our approach can significantly reduce memory requirements by enabling both 20–30% code size reduction of control words and 3–7 times reduction in BRAMs for storing dictionaries with minor performance penalty (decompression overhead).

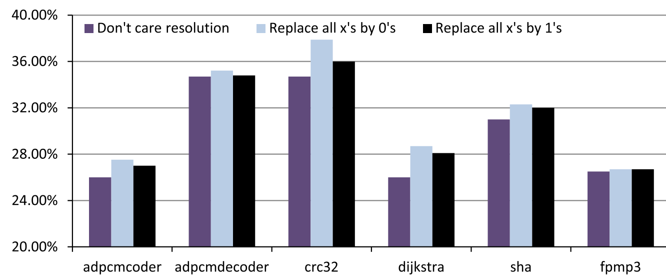

Fig. 17. Effect of bitmask-aware do not care resolution.

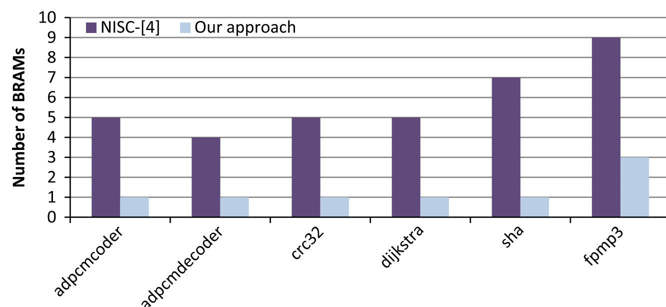

Fig. 18. Comparison of the number of 18 Kbits BRAMs required to store the dictionary entries.

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 do not 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. An efficient way of encoding constant and less frequently changing bits was also developed to significantly reduce the control word size. A suitable decompression architecture was proposed to reduce the overall decompression penalty. Our approach improved compression efficiency by 20–30% over the best known compression technique [2], with significantly less memory overhead and minor performance penalty.

Acknowledgments

This work was partially supported by NSF Grant CNS-0915376. 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.

References

[1] C. Murthy, P. Mishra, Bitmask-based control word compression for NISC architecture, in: Proceedings of Great Lakes Symposium on VLSI, 2009, pp. 321–326.
[2] B. Gorjiara, M. Reshadi, D. Gajski, Merged dictionary code compression for FPGA implementation of custom microcoded PES, ACM Transactions Reconfigurable Technology Systems 1 (2008) 1–21.
[3] M. Reshadi, No-Instruction-Set-Computer (NISC) Technology Modeling and Compilation, Ph.D. Thesis, University of California Irvine, Irvine, CA, USA, 2007.
[4] S.W. Seong, P. Mishra, An efficient code compression technique using application-aware bitmask and dictionary selection methods, in: Proceedings of Design Automation and Test in Europe, 2007, pp. 582–587.
[5] A. Wolfe, A. Channin, Executing compressed programs on an embedded RISC architecture, in: Proceedings of International Symposium on Microarchitecture MICRO, 1992, pp. 81–91.
[6] IBM, CodePack: PowerPC Code Compression Utility User's Manual, version 3.0, International Business Machines (IBM) Corporation, Armonk, NY, USA, 1998.
[7] H. Lekatsas, W. Wolf, SAMC: a code compression algorithm for embedded processors, IEEE Transactions on CAD 18 (1999) 1689–1701.
[8] H. Lekatsas, J. Henkel, V. Jakkula, Design of an one-cycle decompression hardware for performance increase in embedded systems, in: Proceedings of Design Automation Conference, 2002, pp. 34–39.

Please cite this article as: K. Basu, et al., Bitmask aware compression of NISC control words, INTEGRATION, the VLSI journal (2012), http://dx.doi.org/10.1016/j.vlsi.2012.02.004

ARTICLE IN PRESS

[9] C. Lefurgy, P. Bird, I.-C. Chen, T. Mudge, Improving code density using compression techniques, in: Proceedings of International Symposium on Microarchitecture MICRO, 1997, pp. 194–203.
[10] S. Liao, S. Devadas, K. Keutzer, Code density optimization for embedded DSP processors using data compression techniques, in: Proceedings of Advance Research in VLSI, 1995, pp. 393–399.
[11] N. Ishiura, M. Yamaguchi, Instruction code compression for application specific VLIW processors based of automatic field partitioning, in: Proceedings of International Workshop on Synthesis and System Integration of Mixed Information Technologies, 1997, pp. 105–109.
[12] M. Ros, P. Sutton, A hamming distance based VLIW/EPIC code compression technique, in: CASES '04: Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, ACM, 2004, pp. 132–139.
[13] J. Prakash, C. Sandeep, P. Shankar, Y. N. Srikant, A simple and fast scheme for code compression for VLIW processors, in: Proceedings of Data Compression Conference, 2003, p. 444.
[14] S. Nam, I. Park, C. Kyung, et al., Improving dictionary-based code compression in VLIW architectures, IEICE Transactions on Fundamentals of Electronics, Communications and Computer 82 (1999) 2318–2324.
[15] S.Y. Larin, T.M. Conte, Compiler-driven cached code compression schemes for embedded ILP processors, in: Proceedings: 32nd Annual International Symposium on Microarchitecture, Haifa, Israel, IEEE Computer Society Press, 1999, pp. 82–92.
[16] Y. Xie, W. Wolf, H. Lekatsas, Code compression for VLIW processors using variable-to-fixed coding, in: ISSS '02: Proceedings of the 15th International Symposium on System Synthesis, ACM, New York, NY, USA, 2002, pp. 138–143.
[17] C.H. Lin, Y. Xie, W. Wolf, Lzw-based code compression for VLIW embedded systems, in: DATE '04: Proceedings of the Conference on Design, Automation and Test in Europe, IEEE Computer Society, Washington, DC, USA, 2004, pp. 76–81.
[18] M. Reshadi, D. Gajski, B. Gorjiara, No Instruction Set Computer (NISC) Technology, Center for Embedded Computer Systems, University of California Irvine, Irvine, CA, USA, 2007.
[19] Xilinx, Inc., Virtex Series Configuration Architecture User Guide, Version 1.7, Xilinx, Inc., San Jose, CA, USA, 2004.
[20] T. Jensen, B. Toft, Graph Coloring Problems, Discrete Mathematics and Optimization, Wiley-Interscience, New York, 1995.
[21] K. Basu, P. Mishra, Test data compression using efficient bitmask and dictionary selection methods, IEEE Transactions on VLSI 18 (2010) 1277–1286.
[22] C.E. LaForest, J.G. Steffan, Efficient multi-ported memories for FPGAs, in: Proceedings of International Conference on Field Programable Gate Arrays, 2010, pp. 41–50.

Kanad Basu received the B.E. degree from the Deparment of Electronics and Telecommunication Engineering, Jadavpur University, India. He is currently a graduate student at the Embedded Systems Lab in the Department of Computer and Information Science and Engineering, University of Florida. His research interests include functional and structural testing, design for test and post-silicon validation. He received the Best Paper Award at the International Conference on VLSI Design, 2011.

Chetan Murthyreceived the B.E. degree with the Department of Information Science and Engineering, People's Education Society Institute of Technology, Visvesraiah Technological University, India, in 2004, and the M.S. degree from the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, in 2008. He then joined Huawei Technologies India Private Ltd., Bangalore, India. Since Spring 2009, he has been working as a Packet Forwarding Engineer with Juniper Networks, Inc., Sunnyvale, CA.

Prabhat Mishra received the B.E. degree from Jadavpur University, India, the M.Tech. degree from the Indian Institute of Technology, Kharagpur, and the Ph.D. degree from the University of California, Irvine—all in computer science. He is currently an Associate Professor with the Department of Computer and Information Science and Engineering, University of Florida. His research interests include design automation of embedded systems, energy-aware computing and hardware verification. He has published four books, nine book chapters and more than 80 research papers in premier journals and conferences. His research has been recognized by several awards including an NSF CAREER Award in 2008, two best paper awards (VLSI Design 2011 and CODES+ISSS 2003), and 2004 EDAA Outstanding Dissertation Award from the European Design Automation Association. Dr. Mishra currently serves as an Associate Editor of ACM Transactions on Design Automation of Electronic Systems, IEEE Design & Test of Computers, Journal of Electronic Testing, Guest Editor of IEEE Transactions of Computers, and as a program/organizing committee member of several ACM and IEEE conferences.

Please cite this article as: K. Basu, et al., Bitmask aware compression of NISC control words, INTEGRATION, the VLSI journal (2012), http://dx.doi.org/10.1016/j.vlsi.2012.02.004