International Journal of Research
Available at https://edupediapublications.org/journals
e-ISSN: 2348-6848
p-ISSN: 2348-795X
Volume 04 Issue 14
November 2017

A Novel Algorithm to Improve the Performance of the Decoder Without Affecting the Compression Ratio

T. Mounika & R. Guru Prasad
[1] *M.Tech Student, Dept of ECE, ALITS, Affiliated to JNTUA, AP, India .*
[2]*Assistant Professor, Dept of ECE, ALITS, Affiliated to JNTUA, AP, India*

ABSTRACT: *A larger memory can accommodate more and large applications but increases cost, area, as well as energy requirements. Code-compression techniques address this issue by reducing the code size of application programs. It is a major challenge to develop an efficient code-compression technique that can generate substantial reduction in code size without affecting the overall system performance. In this paper, various steps in the code compression process were combined into a new algorithm to improve the compression performance (including the CR) with a smaller hardware overhead. Based on the Bitmask code compression (BCC) algorithm, a small separated dictionary is proposed to restrict the codeword length of high-frequency instructions, and a novel dictionary selection algorithm is proposed to achieve more satisfactory instruction selection, which in turn may reduce the average CR. Furthermore, the fully separated dictionary architecture is proposed to improve the performance of the dictionary-based decompression engine.*

KEYWORDS: Bitmasks, code compression, decompression, embedded systems, memory, compression ratio, code density, run length encoding.

I. INTRODUCTION

MEMORY is one of the key driving factors in embedded-system design because a larger memory indicates an increased chip area, more power dissipation, and higher cost. As a result, memory imposes constraints on the size of the application programs. Code-compression techniques address the problem by reducing the program size. Fig. 1 shows the traditional code-compression and decompression flow where the compression is done offline (prior to execution) and the compressed program is loaded into the memory. Compression ratio (CR), widely accepted as a primary metric for measuring the efficiency of code compression, is defined as

$$\text{CR} = \frac{\text{Compressed program size}}{\text{Original program size}}.$$

Dictionary-based code-compression techniques are popular because they provide both good CR and fast

decompression mechanism. The basic idea is to take advantage of commonly occurring instruction sequences by using a dictionary. Recently proposed techniques improve the dictionary-based compression by considering mismatches. The basic idea is to create instruction matches by remembering a few bit positions. The efficiencies of these techniques are limited by the number of bit changes used during compression. It is obvious that if more bit changes are allowed, more matching sequences will be generated. However, the cost of storing the information for more bit positions offsets the advantage of generating more repeating instruction sequences. Studies have shown that it is not profitable to consider more than three bit changes when 32-b vectors are used for compression. There are various complex compression algorithms that can generate major reduction in code size. However, such compression scheme requires a complex decompression mechanism and thereby reduces overall system performance. It is a major challenge to develop an efficient code-compression technique that can generate Substantial code-size reduction without introducing any decompression penalty (and thereby reducing performance).

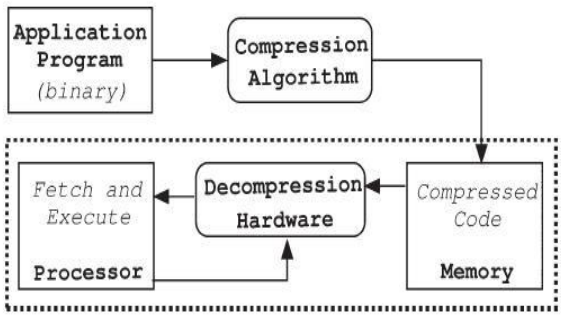


Fig. 1. Code-compression methodology.

Various challenges in bitmask-based compression by developing efficient techniques for application-specific bitmask selection and bitmask-aware dictionary selection to further improve the CR. In dictionary-based schemes,

entire sequences of common instructions are selected and replaced by a single new codeword which is then used as an index to a dictionary that contains the original sequence of instructions. In both cases, lookup tables (LUTs) are used to store the original instructions. The encoded instructions serve as indices to those tables. One of the major problems is that the tables can become large in size, thus diminishing the advantages that could be obtained by compressing the code.

II. RELATED WORK

The first code-compression technique for embedded processors was proposed by Wolfe and Chanin. 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 and Wolf proposed a statistical method for code compression using arithmetic coding and Markov model. Lekatsas *et al.* 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 Fig. 2 shows an example of the standard dictionary-based code compression.

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 explicitly parallel instruction computing (EPIC) processors. The technique proposed by Ishiura and Yamaguchi splits a VLIW instruction into multiple fields, and each field is compressed by using a dictionary-based scheme. Nam *et al.* also use dictionary-based scheme to compress fixed-format VLIW instructions. Various researchers have developed code-compression techniques for VLIW architectures with flexible instruction formats. Larin and Conte applied Huffman coding for code compression. used Tunstall coding to perform variable-to-fixed compression. Lin *et al.* proposed a Lempel–Ziv–Welch (LZW)-based code compression for VLIW processors using a variable-sized-block method. Ros and Sutton have used a post compilation register reassignment technique to generate compression-friendly code. Das *et al.* applied code compression on variable-length instruction-set processors.

III. BIT COMPRESSION ALGORITHM

Field-programmable gate array are widely used in reconfigurable systems. Since the configuration information for FPGA has to be stored in internal or external memory as bit streams, the limited memory size, and access bandwidth become the key factors in determining the different functionalities that a system can be configured and how quickly the configuration can be performed. While it is quite costly to employ memory with more capacity and access bandwidth, bit stream compression technique alleviates the memory constraint by reducing the size of the bit streams. With compressed bit streams, more configuration information can be stored using the same memory. The access delay is also reduced, because less bits need to be transferred through the memory interface. To measure the efficiency of bit stream compression, compression ratio (CR) is widely used as a metric. It is defined as the ratio between the compressed bit stream size (CS) and the original bit stream size (OS). Bit stream compression is important in reconfigurable system design since it reduces the bit stream size and the memory requirement. It also improves the communication bandwidth and thereby decreases the reconfiguration time. Existing research in this field has explored two directions: efficient compression with slow decompression or fast decompression at the cost of compression efficiency. This paper proposes a novel decode-aware compression technique to improve both compression and decompression efficiencies. The three major contributions of this paper are: 1) smart placement of compressed bitstreams that can significantly decrease the overhead of decompression engine; 2) selection of profitable parameters for bitstream compression; and 3) efficient combination of bitmask-based compression and run length encoding of repetitive patterns.

BITMASK TECHNIQUE:

Bitmask-based compression is an enhancement on the dictionary-based compression scheme, which helps us to get more matching patterns. In dictionary-based compression, each vector is compressed only if it completely matches with a dictionary entry.

Three possible cases in bitmask are

The vectors that match directly are compressed with 3 bits. The first bit represents whether it is compressed (using 0) or not (using 1). The second bit indicates whether it is compressed using bitmask (using 0) or not (using 1). The last bit indicates the dictionary index.

International Journal of Research
Available at https://edupediapublications.org/journals
e-ISSN: 2348-6848
p-ISSN: 2348-795X
Volume 04 Issue 14
November 2017

Data that are compressed using bitmask requires 7 bits. The first two bits, as before, represent if the data is compressed, and whether the data is compressed using bitmasks. The next two bits indicate the bitmask position and followed by two bits that indicate the bitmask pattern.

The data which is different for more than 1 bit is left uncompressed.

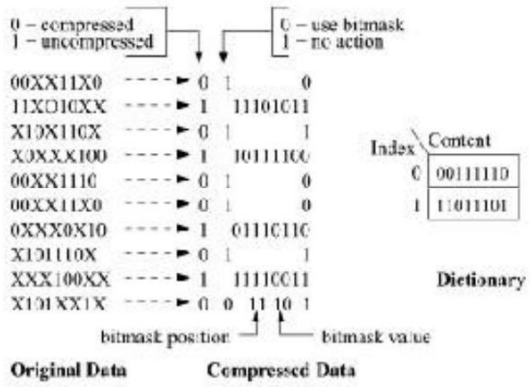
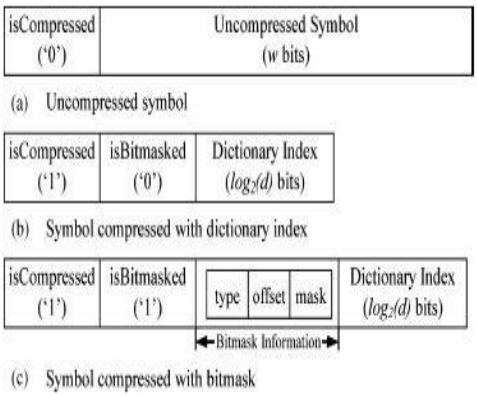


Fig 2 Bit mask Technique



COMPRESSION METHOD

ALGORITHM

Our compression method is based on the technique introduced in [Bird96][Chen97a]. A dictionary compression algorithm is applied after the compiler has generated the program. We search the program object modules to find common sequences of instructions to place in the dictionary.

Our algorithm has 3 parts:
1. Building the dictionary
2. Replacing instruction sequences with code words
3. Encoding code words

IV. OUR ALGORITHMS

DICTIONARY-BASED CODE COMPRESSION

This section describes the existing dictionary-based approaches and analyzes their limitations. First, we describe the standard dictionary-based approach. Next, we describe the existing techniques that improve the standard approach by considering mismatches (hamming distance). Finally, we perform a detailed cost–benefit analysis of the recent approaches in terms of how many repeating patterns they can generate from the mismatches. This analysis forms the basis of our technique to maximize the repeating patterns using bitmasks.

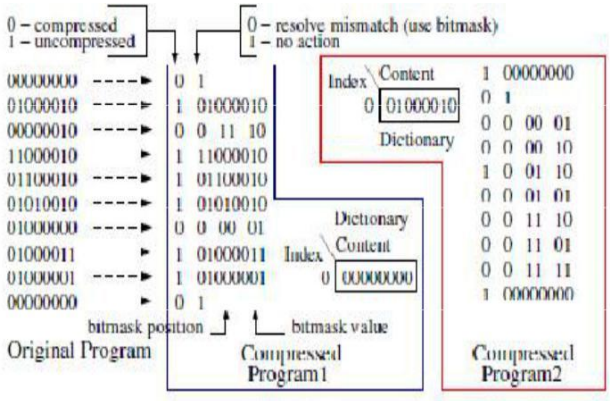


Fig 3 Different Dictionary Selection Methods

DICTIONARY-BASED APPROACH

Dictionary-based code-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 a code word that points to the index of the dictionary that contains the pattern. The compressed program consists of both code words and uncompressed instructions. Fig. 2 shows an example of dictionary based code compression using a simple program binary. The binary consists of ten 8-b patterns, i.e., a total of 80 b. The dictionary has two 8-b entries. The compressed program requires 62 b, and the dictionary requires 16 b. In this case, the CR is 97.5% [using (1)]. This example shows a variable-length encoding. As a result, there are several factors that may need to be included in the computation of

International Journal of Research
Available at https://edupediapublications.org/journals
e-ISSN: 2348-6848
p-ISSN: 2348-795X
Volume 04 Issue 14
November 2017

the CR, such as byte alignments for branch targets and the address-mapping table.

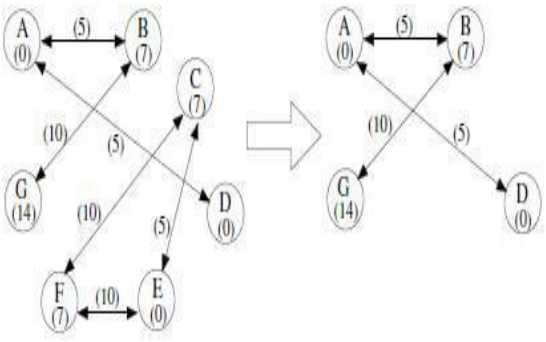


Fig 4 Bit-Saving Dictionary Selection Method

MIXED BIT SAVING ALGORITHM

FDS cannot achieve an optimal CR in BCC, because it cannot guarantee that the matched rate of high-frequency instructions is maximized. The proposed dictionary selection algorithm is based on the Graph representation. The instructions are transformed into nodes, and an edge between two nodes indicates that these two instructions have been matched to each other using the BitMask approach. In general, the nodes are classified into five cases according to the frequency and connection pattern.

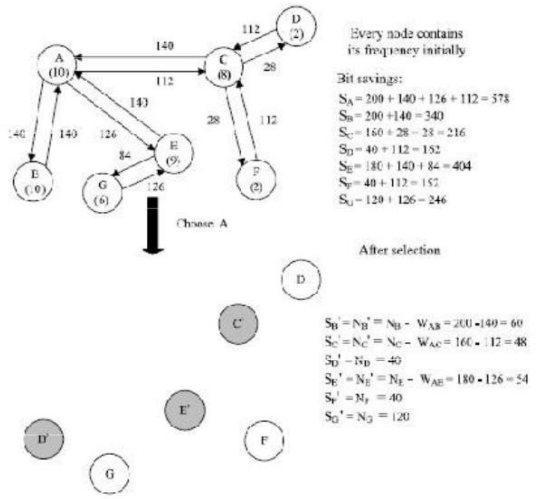


Fig 5 MBSDS

The new algorithm first transforms every unique instruction into a single node. Two directional edges between two nodes indicate that these two instructions were matched to each other using the Bitmask compression approach. The proposed algorithm then calculates the bit saving of all nodes, and inserts the most profitable node into the dictionary. The most profitable node is then removed from the graph. Since all the neighboring nodes of the most profitable node can be covered by the most profitable node, the node saving of each neighboring node should subtract the edge saving from the edge with the most profitable node.

Furthermore, all the edges of the neighboring nodes are removed. These steps are repeated until the dictionary is full. The most profitable node achieves the savings from the combination of its own node saving and the edge savings of other nodes. However, connected instructions cannot be easily inserted into the dictionary. Whether these connected instructions should be selected into the dictionary in the following rounds is solely determined by their frequency values.

This method offers an advantage. When only the edges connected with the most profitable node are removed after the most profitable node is selected and inserted into the dictionary, the algorithm is likely to choose one of the neighbors of the most profitable node as the new profitable node.

This, however, would likely result in many incorrect or Case 3 nodes being selected and inserted into the dictionary. All symbols in this example are 32-bit wide, the dictionary contained 1024 entries, only one 2-bit mask was used, and the overhead of the identification tag is 2-bit. After each symbol was transferred into the nodes, every node contained its frequency value. When two nodes were matched to each other using the Bitmask, the algorithm will create two-directional edges to connect them; the direction pointed to the instruction, and the weight corresponding to the actual edge saving when the connected node was compressed by the matched node.

Suppose a threshold value of 10 is used, Nodes A and B were selected and inserted into the dictionary, and Nodes C and E were deleted from the graph. The rest of candidate nodes D and F, were then selected in the following round. Nodes C and E (if G was not selected) were more efficient than B, D, or F. As this result demonstrated, an unsatisfactory threshold value reduced the efficiency of the BSDS algorithm compared with the FDS algorithm. The proposed dictionary selection algorithm is compared with some prior arts.

International Journal of Research
Available at https://edupediapublications.org/journals
e-ISSN: 2348-6848
p-ISSN: 2348-795X
Volume 04 Issue 14
November 2017

DECOMPRESSION ENGINE

The proposed decompression engine was implemented [23] using the Verilog hardware description language and synthesized using a Synopsys' Design Compiler [24] and a TSMC 0.13-μm cell library. The decompression engine, the logic diagram of which is shown in Fig. 9 consisted of a control unit, a demultiplexer, shift buffers, LUTs, and the BitMask unit. The control unit controls other units and assigns tasks to other units according to the control signals. The input queue initializes itself, collects compressed instructions from the storage space, and shifts the contents of the buffer after the decoding process is completed. The output queue stores the decompressed instructions and delivered them to the processor or cache. The large LUT and small LUT store the original binary instructions and synthesized using a flip-flop logic.

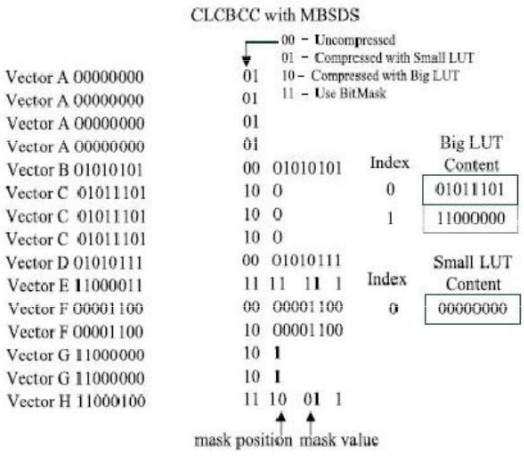


Fig 6 CLCBCC with MBSDS.

The small LUT stores high-frequency instructions enabling them to be quickly decoded with a shorter codeword length. The BitMask unit executes the shifting of masks and XOR operations based on the instructions from the large LUT to obtain the original instructions. The BitMask unit also accesses the dictionary and executes shift operations in parallel during decompression. The proposed decompression engine has a decompression bandwidth of 32 bits/cycle. As mentioned in Section IV-A, even though the instructions with an extremely high execution frequency were placed in the small LUT to improve the compression efficiency, a longer LUT latency in the large LUT(s) constrained the performance of the decompression engine. Certain researchers have restricted the LUT size in their research to avoid this problem. However, restricting the LUT size is not an optimal solution in all the cases.

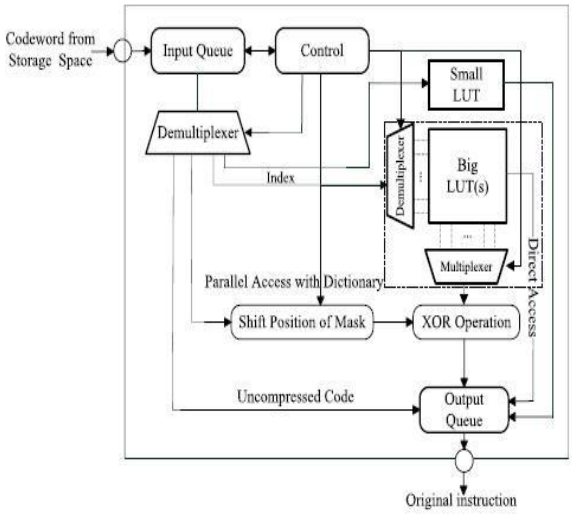


Fig 7 Logic diagram of decompression engine.

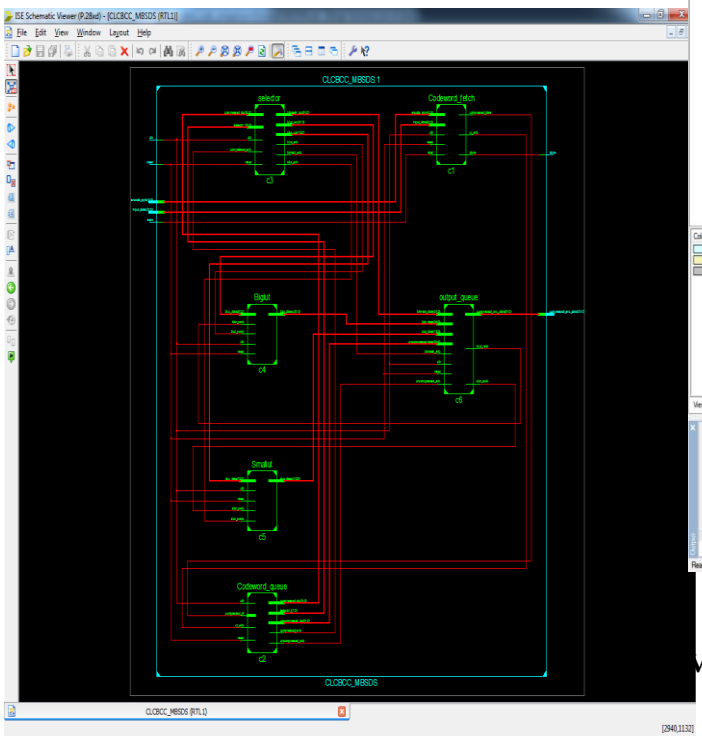
This number is multiplied by the number of saved bits by the BitMask method, and then it is divided by the original program size. The algorithm only achieves a more efficient DSCR when the program size is small and the number of uncompressed instructions is high. For a large benchmark containing 40 000 instructions, and for which 2048 entries were used to develop the large LUT, the uncompressed instruction rate, created using the CLCBCC with FDS, was 10%. Within the uncompressed instructions, 60% were matched by a more satisfactory dictionary selection algorithm (40% of instructions were not matched because of their Hamming distance mismatch with a limited number of LUT entries using the BitMask method).

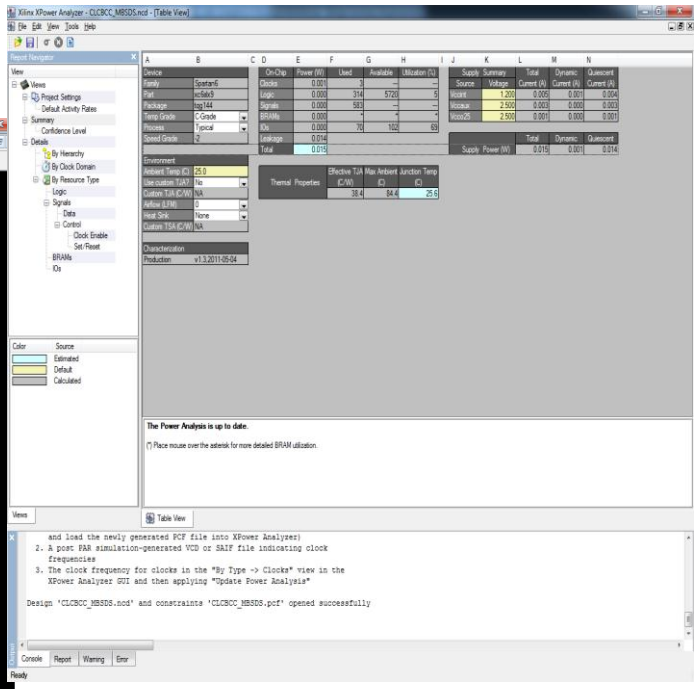
A total of 50% of the instructions were matched using one mask, and 50% using two masks. According to Amdahl's law, the CR saving is ideally 10% multiplied by 12/32 (assuming every instructions can be matched using only one mask) equals to 3.75%. In practice, the algorithm only achieved CR savings of 1.59%. This indicates that no dictionary selection algorithm can produce a substantial improvement in larger benchmarks.

International Journal of Research
Available at https://edupediapublications.org/journals
e-ISSN: 2348-6848
p-ISSN: 2348-795X
Volume 04 Issue 14
November 2017

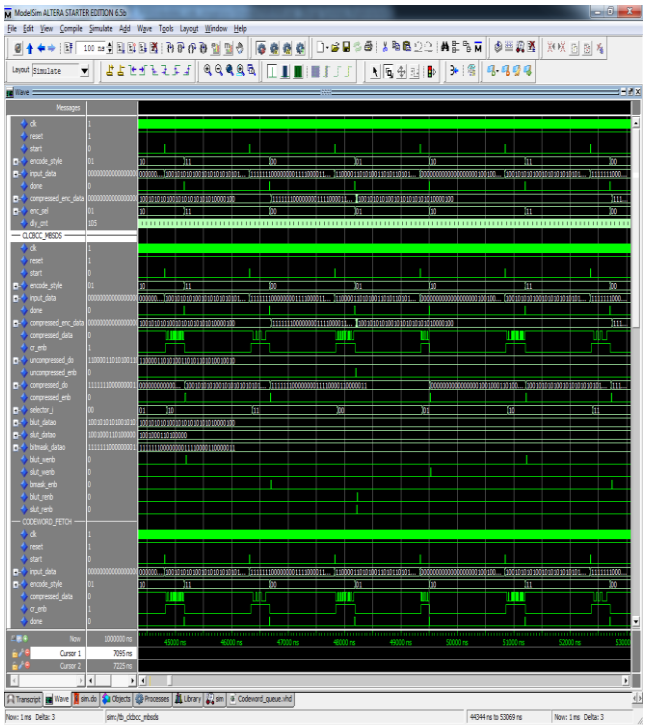
SIMULATION RESULTS

RTL view





Simulation wave form



Power report

V. CONCLUSION

In this paper the encoding format was modified to enable the decompression engine to support multi-LUT access and use variable mask numbers to operate with the referenced instructions. A new dictionary selection algorithm was also proposed to improve the CR. The fully separated dictionary architecture was used to improve the performance of the decoder, and this architecture is better suitable to decompress instruction in parallel to increase the decompression bandwidth per cycle. The design of a decompression engine is a new challenge for multicore systems. In the future studies, the design and implementation of a general multilevel separated dictionary decompression engine fully separated LUTs method and a parallel decompression engine will be investigated, for applying code compression to architectures with high bandwidth requirements, such as multicore architectures. Not only the CR, but also performance, power consumption, and communication bandwidth between the memory and the caches should be analyzed.

REFERENCES

1. Wei Jhih Wang and Chang Hong Lin ,Code Compression for Embedded Systems Using Separated Dictionaries", in IEEE Transactions on very large scale integration (VLSI) systems 1063-8210 © 2015 IEEE

2. C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge, "Improving code density using compression techniques," in *Proc. 30th Annu. ACM/IEEE Int. Symp. MICRO*, Dec. 1997, pp. 194–203.
3. S.-W. Seong and P. Mishra, "A bitmask-based code compression technique for embedded systems," in *Proc. IEEE/ACM ICCAD*, Nov. 2006, pp. 251–254.
4. S.-W. Seong and P. Mishra, "An efficient code compression technique using application-aware bitmask and dictionary selection methods," in *Proc. DATE*, 2007, pp. 1–6.
5. M. Thuresson and P. Stenstrom, "Evaluation of extended dictionarybased static code compression schemes," in *Proc. 2nd Conf. Comput.Frontiers*, 2005, pp. 77–86.
6. H. Lekatsas and W. Wolf, "SAMC: A code compression algorithm for embedded processors," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 18, no. 12, pp. 1689–1701, Dec. 1999.
7. S. Y. Larin and T. M. Conte, "Compiler-driven cached code compression schemes for embedded ILP processors," in *Proc. 32nd Annu. Int. Symp. Microarchitecture*, Nov. 1999, pp. 82–91.
8. C. H. Lin, Y. Xie, and W. Wolf, "Code compression for VLIWembedded systems using a self-generating table," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 15, no. 10, pp. 1160–1171, Oct. 2007.
9. C.-W. Lin, C. H. Lin, and W. J. Wang, "A Power-aware codecompression design for RISC/VLIW architecture," *J. Zhejiang Univ.-Sci. C (Comput. Electron.)*, vol. 12, no. 8, pp. 629–637, Aug. 2011.
10. T. Bonny and J. Henkel, "FBT: Filled buffer technique to reduce code size for VLIW processors," in *Proc. IEEE/ACM Int. Conf. CAD (ICCAD)*, Nov. 2008, pp. 549–554.
11. M. Ros and P. Sutton, "A hamming distance based VLIW/EPIC code compression technique," in *Proc. Compilers, Arch., Synth. Embed.Syst.*, 2004, pp. 132–139.
12. J. Ranjith, N. J. R. Muniraj, and G. Renganayahi, "VLSI implementation of single bit control system processor with efficient code density," in *Proc. IEEE Int. Conf. Commun. Control Comput. Technol. (ICCCCT)*, Oct. 2010, pp. 103–108.
13. W. R. Azevedo Dias, E. D. Moreno, and I. Nattan Palmeira, "A new code compression algorithm and its decompressor in FPGA-based hardware," in *Proc 26th Symp. Integr Circuits Syst.Design.*