Novel Code Compression Techniques for Embedded Digital Systems Using Separated Dictionaries

[1]Mrs.Savitha.T, *Assistant Professor, Dept.of Electronics & Communication Engineering, Swami Vivekananda institute of Technology, Secunderabad*

[2]Mr.Parameshwar.B, *Assistant Professor, Dept.of Electronics & Communication Engineering, Swami Vivekananda institute of Technology, Secunderabad*

Abstract— *Engineers must consider performance, power consumption, and cost when designing embedded digital systems; furthermore, memory is a key factor in such systems. Code compression is a technique used in embedded systems to reduce the memory usage. BitMask-based code compression is a modified version of dictionary-based code compression. The basic purpose of BitMask is to record mismatched values and their positions to compress a greater number of instructions; it can be used exclusively or incorporated with the reference instructions to decode the codewords. In this paper, we applied a small separated dictionary, and variable mask numbers were used with the BitMask algorithm to reduce the codeword length of high frequency instructions. In addition, a novel dictionary selection algorithm was proposed to increase the instruction match rates. The fully separated dictionary method was used to improve the performance of the decompression engine without affecting the compression ratio (CR) (the compressed code size divided by original code size). Based on the experimental results, the proposed method can achieve a 7.5% improvement in the CR with nearly no hardware overhead.*

Keywords— Computer architecture, dictionary-based code compression (DCC), embedded systems, separated dictionaries..

I. Introduction

EMBEDDED systems have become an essential part of everyday life, and are widely used worldwide. Embedded systems must be cost effective, and memory occupies a substantial portion of the entire system. To reduce the system cost,Wolfe and Chanin [1] first proposed code compression for compressing the program size in the early 1990s to conserve the memory usage. In recent decades, the research in code compression has been conducted to reduce the code size and power consumption, as well as to improve the performance.

The compression ratio (CR) is a metric used to evaluate memory compression efficiency, which is defined as follows:

$$CR \equiv \frac{\text{Compressed Program Size + Decoding Table Size}}{\text{Original Program Size.}} \qquad (1)$$

Although the area occupied by integrated circuits has been reduced by recent technical advances, code compression techniques remain crucial for embedded systems.

The complexity and performance requirements for embedded programs grow rapidly, which results in additional memory usage and power consumption. For all the existing code compression techniques, all binary instructions are compressed offline and decompressed as required during execution. Thus, reducing the code size and providing a simple decompression engine are both challenges when applying code compression to embedded systems.

Dictionary-based code compression (DCC) [2] is commonly used in embedded systems, because it can achieve an efficient CR, possess a relatively simple decoding hardware, and provide a higher decompression bandwidth than the code compression by applying lossless data compression methods.

Thus, it is suitable for architectures with high-bandwidth instruction-fetch requirements, such as the very long instruction word (VLIW) processors. Although several existing code compression algorithms have exhibited favorable compression performance, no single compression algorithm has efficiently worked for all kinds of benchmarks. 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 [3], [4], 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.

This architecture has a better chance to parallel decompress instructions than existing single dictionary decoders. The remainder of this paper is organized as follows. Section II presents a review of related studies on code compression. Section III describes the BitMask-based compression approaches [3]–[5]. Section IV describes the proposed codeword-length-constrained BCC (CLCBCC) algorithm, mixed-bit saving dictionary selection (MBSDS), and fully separated dictionary. Section V presents the experimental results of the benchmarks for ARM Cortex-A9 and Texas Instruments (TI) C62×× and C64×× VLIW processors [6].Finally, the conclusion is drawn in Section VI.

II. RELATED WORK

Numerous lossless data compression algorithms have been applied to code compression for embedded systems. Wolfe and Chanin [1] were the first to use Huffman coding on Microprocessor without Interlocked Pipeline Stages processors and implemented a pre-cache structure with modified cache architecture. A line address table maps the compressed block addresses to actual memory addresses when the cache misses and branch instructions are encountered. Later research in code compression drew on this research and continued to use memory addresses for compression. Based on the same concept, Lekatsas and Wolf [7] applied arithmetic coding with Markov model to Reduced instruction set computing (RISC) processors.

All of these methods targeted RISC processors. Larin and Conte [8] applied Huffman coding to VLIW processors. Xie *et al.* [9] used Tunstall coding and arithmetic coding to perform variable-to-fixed compression on VLIW processors. Based on the branch blocks, Lin *et al.* [10] proposed a Lempel Ziv Welch-based code compression for VLIW processors. Lin *et al.* [11] proposed selective code compression, which maintained frequently executed small blocks uncompressed to trade CR for power and performance.

Bonny and Henkel [12] used Lempel Ziv Storer Szymanski (an optimized version of LZ77) compression algorithm in conjunction with a filled buffer technique and extended blocks to compress VLIW instructions. Then, they used Huffman coding to recompress the extended blocks.

Qin and Mishra [13] used bounded Huffman coding to compress instructions and proposed a bitstream placement algorithm to replace the compressed instructions such that all instructions were simultaneously parallel decompressed.

Bonny and Henkel [14] used extended blocks and divided each block into two parts: 1) left-uncompressed instructions and 2) compressed instructions using the Burrows–Wheeler algorithm. During the decompression phase, the decompression engine first sends the left-uncompressed instructions to the processor, while decompressing the compressed instructions. It waits until the left-uncompressed instructions are executed. Although this method sacrifices some CR, it improves the performance of a decompression engine.

Lefurgy *et al.* [2] proposed the first DCC algorithm, which replaced frequently executed instructions as dictionary indices. Gorjiara *et al.* [15] used DCC with a multi-dictionary for a no instruction set computer (NISC) architecture. Ros and Sutton [16] proposed improved DCC methods by considering Hamming distances and mismatches. Based on the DCC, Thuresson and Stenstrom [5] combined dynamic instruction stream editing and BitMask methods to compress instruction sequences. Seong and Mishra [3], [4] used several bits as a mask for increasing the instruction coverage rate, and they proposed a novel dictionary selection method to improve the CR. Qin *et al.* [17] combined the BCC and run-length coding with an improved dictionary-selection method for field-programmable gate array bit streams. Murthy and Mishra [18] used a map with a multi-dictionary for the NISC architecture.

Bonny and Henkel [19] used dictionary-based and canonical Huffman coding to reencode the codewords compressed by Huffman coding in embedded processors. Both instructions and lookup tables (LUTs) are compressed to achieve an optimal CR. Based on the

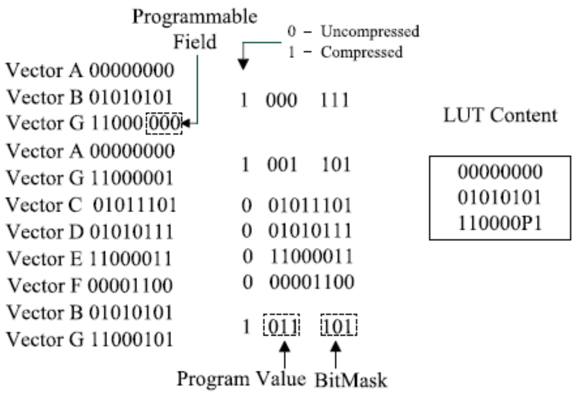

Fig. 1. Thuresson and Stenstrom’s [16] BitMask-based method.

same method, Ranjith *et al.* [20] applied the code compression in a delta-sigma control-system processor to reduce the memory cost and optimize power consumption in the processor. Based on the BCC, Chen *et al.* [21] used dictionary-entry replacement algorithm to reduce the power consumption of the systems. Azevedo Dias *et al.* [22] used Huffman coding to compress two adjacent instruction sequences and then used the same method to compress single instructions, which called compressed code using Huffman-based multilevel dictionary. They also design a one instruction per cycle decompression engine. Recent research in code compression has focused on two directions: 1) applying existing compression methods to various architectures for optimization and 2) combining several approaches to improve the performance, including CR.

Seong and Mishra [3], [4] and Wang and Lin [23] observed that no single compression algorithm operated efficiently for all the benchmarks. Thus, this paper integrates several approaches to form a new algorithm with smaller hardware overhead. New dictionary architecture is used to improve the decompression engine performance.

III. PROPOSED ALGORITHMS

In this section, the proposed algorithms are described. A separate dictionary was used to reduce the codeword length of high-frequency instructions. Variable mask numbers were used to eliminate the encoding redundancy. The combination of these methods is called as the CLCBCC. A modified version of a MBSDS algorithm from [23] was used to select an improved instruction combination for the dictionary. Compared with [23], a fully separated dictionary architecture is proposed to reduce the access latency of the dictionary. Experimental results, including benchmarks and various processor architectures, are presented in Section V.

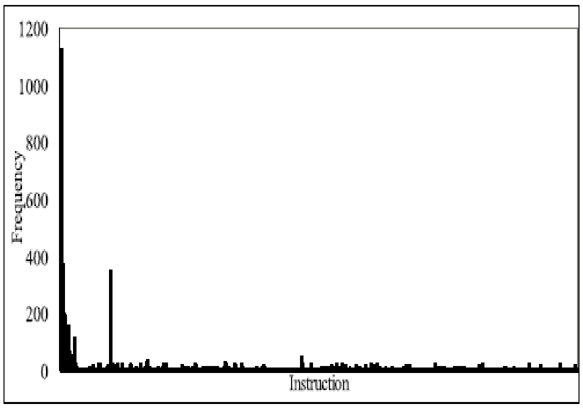

Fig. 2. *fft*: frequency distribution of 512 dictionary entries.

A. Separated Dictionaries

In certain cases, such as in low code density architecture [15], which contains a high number of unique instructions or because of algorithmic characteristics, a large LUT is required to compress the programs. A large LUT has several disadvantages: it requires a large chip area, additional power consumption, a long LUT latency, and a long codeword length.

Thus, it is desirable to minimize the dictionary size. The static frequency distribution of the instructions was analyzed from the set of benchmarks [23] on an TI C62××; the results demonstrated that only a small set of instructions consistently exhibited extremely high frequencies. Figs. 3 and 4 show the frequency distribution of dictionary entries from two benchmarks: 1) *fft*, a smaller benchmark with 512 entries and 2) *susan*, a larger benchmark with 1024 entries. Both distributions were generated using the frequency-based dictionary selection (FDS) algorithm. The frequency distributions were similar for all the benchmarks. Compressing these high-frequency instructions with the same codeword length as other low-frequency instructions would result in inefficient compression. To overcome this problem, these high-frequency instructions are separated into another small dictionary to obtain shorter codeword lengths. Two LUTs are used for the BitMask approach.

A large LUT is used to compress single instructions, and a small LUT is used to compress the extremely high-frequency instructions. The small LUT was modifiable for storing either single instructions or instruction sequences [23]. The specific dictionary architecture for the CLCBCC is shown in Fig. 5.

B. Variable Mask Numbers

Seong and Mishra [4] surveyed the size and combination of the masks and they concluded that a 4-bit fixed (4f) and a 1-bit sliding (1s) mask achieves an optimal CR. However, Wang and Lin [23] determined that using a 4-bit fixed and a 2-bit fixed masks in addition to a single 4-bit fixed mask achieves better results for the benchmarks. Although the maximum mask overhead was 13 bits (4 bits for 4-bit mask, 3 bits to record the position of the 4-bit fixed mask, 2 bits for 2-bit mask, and 4 bits to record the position of the 2-bit fixed mask), it was determined that ~50% of the instructions were compressed using only the 4-bit fixed mask in the benchmarks.

Thus, in this paper, the 4f mask and the 4f-2f masks are combined, and 1 bit is used to identify whether the codeword uses one or two masks. The encoding format is shown in Fig. 6, which contains four situations, such as uncompressed, matched with small dictionary, matched with large dictionary, and matched using a variable number of masks.

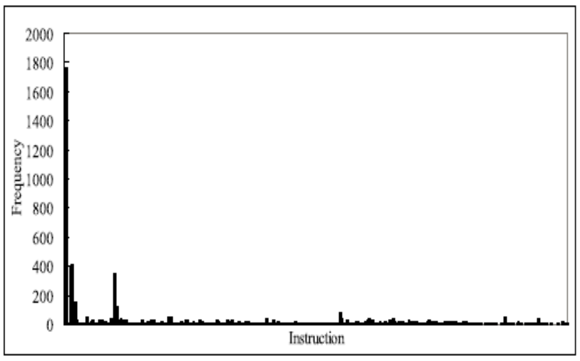

Fig. 4. *susan*: frequency distribution of 1024 dictionary entries.

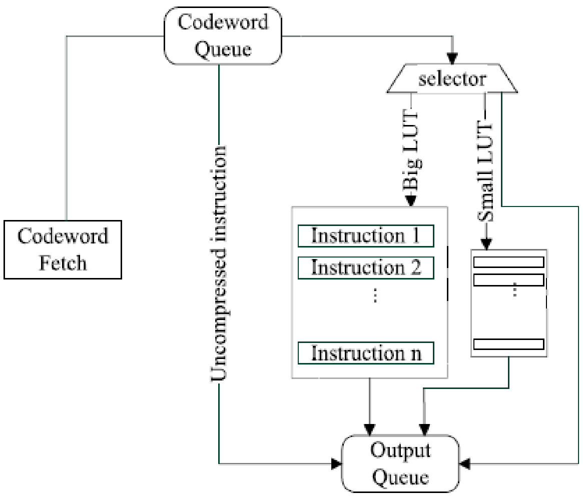

Fig. 5. Specific architecture for the CLCBCC.

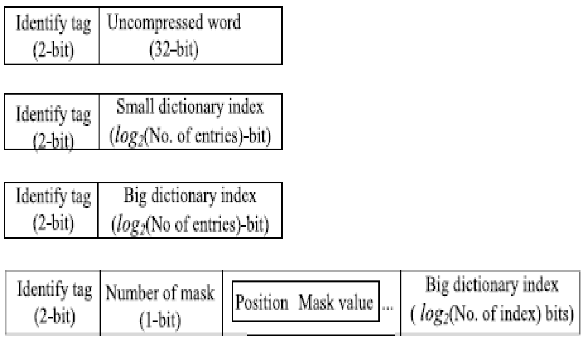
Identify tag (2-bit)
Uncompressed word (32-bit)
Identify tag (2-bit)
Small dictionary index (log_2(No. of entries)-bit)
Identify tag (2-bit)
Big dictionary index (log_2(No of entries)-bit)
Identify tag (2-bit)
Number of mask (1-bit)
Position
Mask value
...
Big dictionary index (log_2(No. of index) bits)

Fig. 6. Encoding format for our approach.

C. 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.

Case 1: A high-frequency node mostly connects to high-frequency nodes.

Case 2: A high-frequency node mostly connects to low-frequency nodes.

Case 3: A low-frequency node mostly connects to high-frequency nodes.

Case 4: A low-frequency node mostly connects to low-frequency nodes.

Case 5: A low-frequency node with few connections.

Cases 1, 2, and 4 are better choices for the CR improvement, and Case 2 nodes can achieve the most savings. Because the high-frequency nodes are usually selected into the dictionary, the benefits for nodes in Case 3 are limited. Thus, they are unsuitable for the dictionary. The nodes in Case 5 are never selected in the algorithm because their low frequency and few connections result in low savings.

An MBSDS algorithm is proposed in this paper, as shown in Algorithm 1. This algorithm is an improved version of the algorithm proposed in [23]. 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 neighbouring nodes of the most profitable node can be covered by the most profitable node, the node saving of each neighbouring node should subtract the edge saving from the edge with the most profitable node. Furthermore, all the edges of the neighbouring nodes are removed. These steps are repeated until the dictionary is full.

Algorithm 1 MBSDS

```
Inputs:
1. 32-bit unique instruction vectors
2. Dictionary size
3. Mask types
Output: Optimized dictionary
Begin
Step 1: Transform every unique instruction to a graph, G = (V, E).
If two nodes can be matched by using BitMask, use directional edges
to connect them.
Step 2: Allocate bit savings to the nodes and edges.
  ∀ node i,
  Node saving ($N_i$) = (original instruction size − compressed codeword size)
    × frequency of the instruction − 32 bits overhead.
  ∀ edge between nodes i and k,
  Edge saving ($W_{ik}$) = (original instruction size - compressed codeword size)
    × frequency of the matched instruction.
  Total bit saving ($S_i$) = $N_i + \sum_{k=1}^{n} W_{ik}$
Step 3: Calculate the total bit saving distribution of all nodes.
Step 4: Select the most profitable node $i^* = \arg\max f(S_i)$.
Step 5: Remove the most profitable node i from G and insert it into the
dictionary.
Step 6: ∀ nodes connect to i, construct the neighboring node set Nb(i).
Step 7: Delete all edges with at least one end in Nb(i)
Step 8: Update node savings of all the nodes in Nb(i), $S_i = N_k - W_{ik}$.
Step 9: Repeat Steps 3 - 8 until the dictionary is full.
Step 10: Return dictionary.
End
```

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.

IV. EXPERIMENTAL RESULTS

In this section, experimental benchmarking results for using the ARM Cortex-A9 and TIs C6× series VLIW processor are presented. The benchmarks were obtained from MediaBench [25] and Mibench [26], Figure 4: (a) K means clustering Image (b) Optic disc (c) Optic cup and are used to access various kinds of embedded applications.

The benchmarks were compiled using GNU Compiler Collection and Code Composer Studio for ARM Cortex-A9 and C6×, respectively. All instructions are extracted from the text sections of the compiled binaries on the ARM Cortex-A9 and C6× architecture.

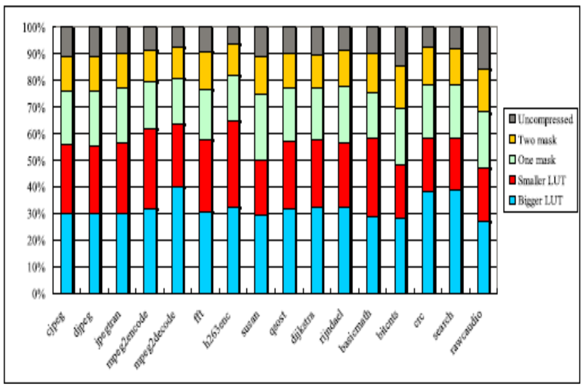

Fig. 11. Probability distribution of all codeword types using CLCBCC with FDS on ARM target.

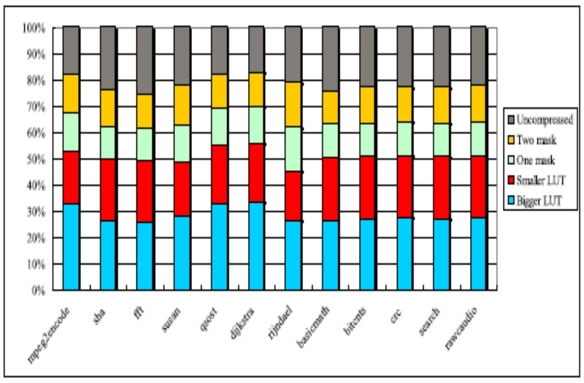

Fig. 12. Probability distribution of all codeword types using CLCBCC with FDS on TI C62×× target.

A. Compression Ratio

Figs. 11–13 show the probability distribution of all five possible codeword types when using the proposed approach with FDS on the ARM and TI C6× architectures. Beginning with the longest, the sequence of the lengths of codewords was uncompressed, decompressed using two masks (4f, 2f), one mask (4f), the large LUT, and the small LUT. LUT with a size of 2048, and a small LUT with a size of 16, and codeword lengths of 34, 27, 21, 13, and 6, the introduction of a separated LUT clearly reduces the codeword sizes with 20%–30% of the instructions in nearly all the benchmarks.

In Figs. 14–16, the CRs of three different code compression algorithms were compared: 1) DCC; 2) BCC with fixed mask numbers (4f, 1s); and 3) CLCBCC using a 2f and 4f masks [23], respectively. Furthermore, the saving rates of Thumb-2 [27] for benchmarks on the ARM Cortex-A9 processor are included in Fig. 14, as well. An FDS algorithm was used to select LUT entries in all these approaches. For CLCBCC, a small LUT was first constructed, and then the remaining instructions were used to construct the large LUT. The results demonstrate that using a small LUT for storing high-frequency instructions can result in a CR improvement of 6% for the CLCBCC compare with BCC. Although using the debug configuration for the C6× series on the Code Composer Studio achieved a lower CR and greater improvements [23], but the release configuration was used to simulate the experiment in this paper.

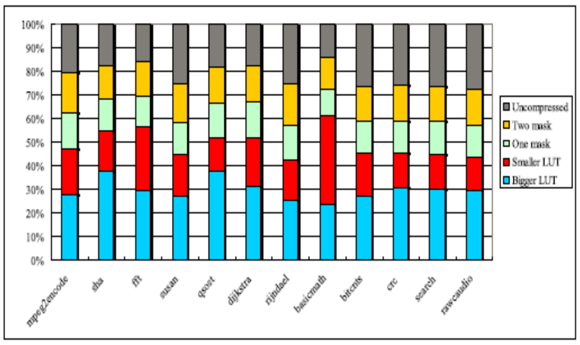

Fig. 13. Probability distribution of all codeword types using CLCBCC with FDS on TI C64×× target.

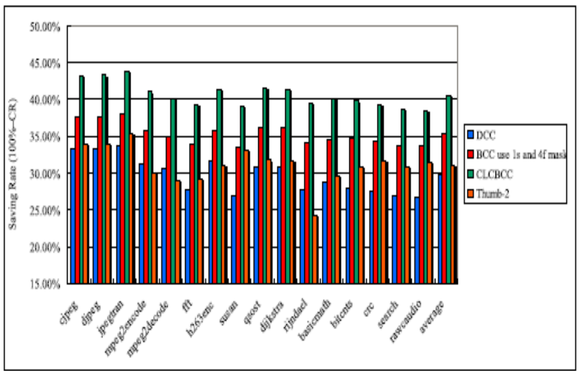

Fig. 14. Comparison of CR for benchmarks on ARM Cortex-A9 processor.

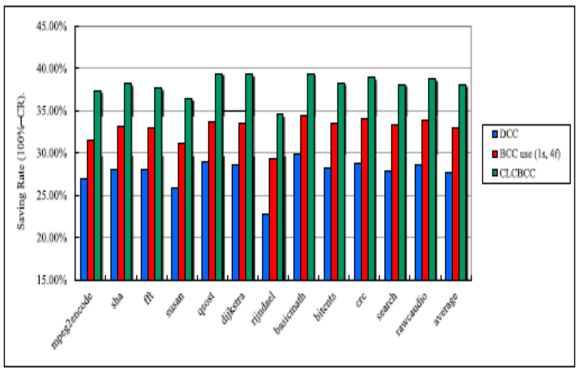

Fig. 15. Comparison of CR for benchmarks on C62×× processor.

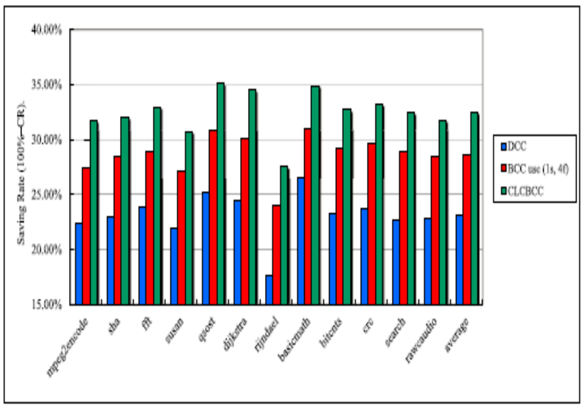

Fig. 16. Comparison of CR for benchmarks on C64×× processor.

B. Different Selection Algorithms

In Figs. 17–19, the CRs of four different dictionary selection algorithms are compared: 1) FDS; 2) BSDS [4]; 3) decoding aware dictionary selection (DADS) [17]; and 4) the proposed MBSDS, which is a modification from [23]. Figs. 17–19 compare these algorithms using the proposed CLCBCC and BCC with fixed mask numbers (4f, 1s).

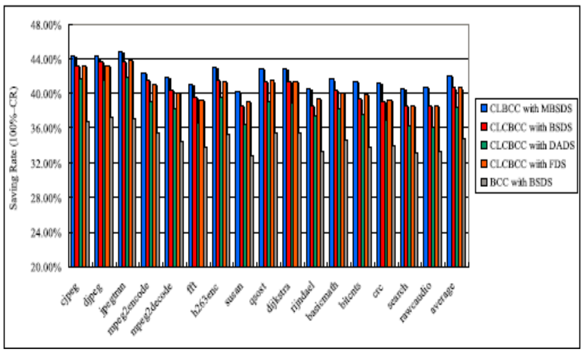

Fig. 17. Comparison of dictionary selection algorithms using CLCBCC and BCC approach on ARM target.

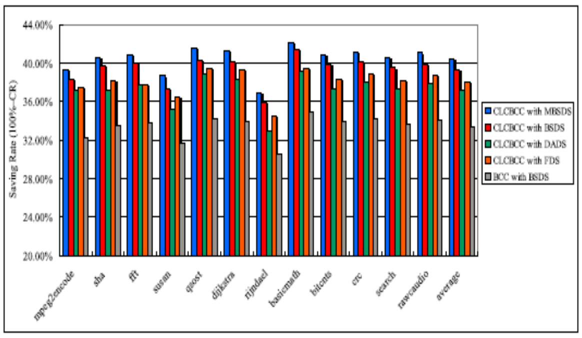

Fig. 18. Comparison of dictionary selection algorithms using CLCBCC and BCC approach on C62×× target.

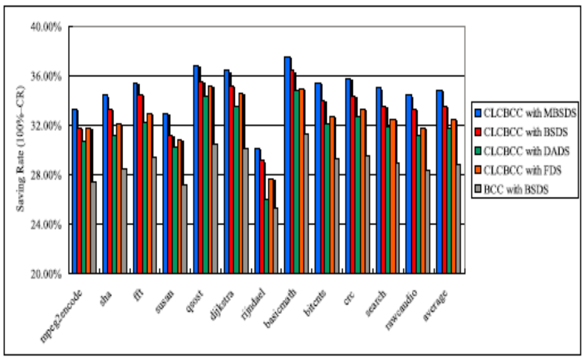

Fig. 19. Comparison of dictionary selection algorithms using CLCBCC and BCC approach on C64×× target.

Compared with other selection algorithms, the proposed MBSDS outperforms the FDS and DADS for all the benchmarks. Because FDS only took frequencies rather than BitMask matches, into the consideration DADS ensures higher instruction coverage [17].

However, it also selects numerous unnecessary Case 3 instructions and several incorrect instructions, which results in a much larger LUT. Seong and Mishra [4] claimed that a threshold value between 5 and 15 are good for BSDS, thus 10 was set as the threshold value for BSDS in the simulations. The results showed that using the same threshold value can be unstable in different benchmarks, which is a disadvantage of BSDS. If there are n unique instructions, the time complexity for both BSDS (with a given threshold value) and MBSDS is $O(n2)$. No studies have yet proposed an optimal method to obtain a superior threshold value. Thus, the threshold value can only be obtained by trial-and-error. The BSDS must be performed several times to determine the superior threshold value. Discovering the threshold value is time-consuming when the number of unique instructions is large. The proposed algorithm thus offers another advantage: 1) it does not require a threshold to decide whether a node can be selected and 2) it can avoid the special case described in Section IV-C. In other words, MBSDS not only improves the compression efficiency or CR, but also improves the performance of the algorithm.

After searching for new algorithms to achieve a better dictionary selection result, it was concluded that none of the algorithm can produce a substantial improvement for the benchmarks. This phenomenon is explained by Amdahl's law. The key point of a dictionary-selection algorithm for the BitMask method is to increase the match rate, to match a greater number of instructions for dictionary selection.

The rate of uncompressed instructions is based on the Instruction set architecture, compiler, and the limited dictionary size. On the ARM target, the rate of uncompressed instructions typically occupies a small part (7%–16%, as shown in Fig. 11) of every benchmark, and the instructions that can be matched by the dictionary selection algorithm occupy only a small portion of the initially uncompressed instructions (~6% on average). Thus, the benefit from a dictionary selection algorithm is constrained by the portion of initially uncompressed instructions, and consequently, regardless of how efficiently the algorithm increases the match rate, improvements remained marginal. In other words, the dictionary selection algorithm only operates efficiently for small benchmarks with a high-uncompressed instruction rate. This is explained by

$$\text{DSCR} \equiv \frac{\text{No. of Matched Inst. from Uncompressed Inst. Set}}{\text{Original Program Size}} \times (32\text{-len(BitMask Encode Codeword)}). \quad (2)$$

The dictionary selection CR (DSCR) saving rate is equal to the number of instructions that are matched from the set of uncompressed instructions. 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. Unless an algorithm that both increases the match rate and reduces LUT usage substantially in larger benchmarks exists, achieving betterCR savings using the dictionary-selection algorithm in several smaller benchmarks with a high uncompressed instruction rate is far more practical. This is the reason why BSDS functions better in [4], but not in the experiments in this paper.

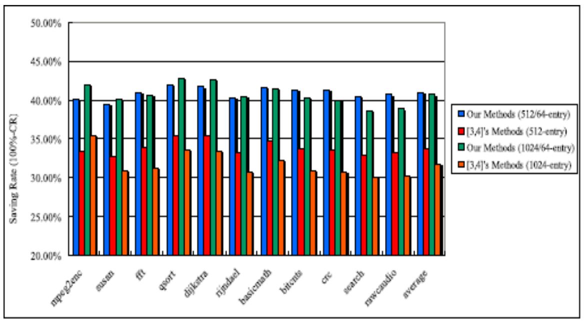

Fig. 20. CR comparison on ARM target with constant LUT size.

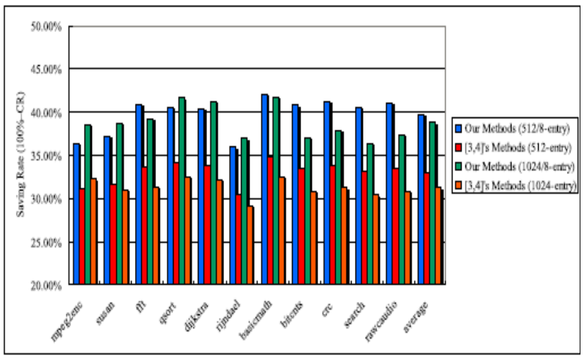

Fig. 21. CR comparison on C62×× target with constant LUT size.

However, focusing on a method for reducing only the uncompressed instruction rate is misleading. In trivial cases, when the algorithm creates an LUT that contains all the unique instructions in which the uncompressed instruction rate is reduced to 0%, but the LUT overhead and the encoding length may cause the CR to be over 100%. Reducing the uncompressed instruction rate is meaningful when considering both the overheads of LUTs and codewords. Thus, as shown in Figs. 12 and 13, when the most suitable LUT size is chosen for each benchmark following the dictionary selection algorithms, several C6× targets contained high uncompressed instruction rates (in excess of 20%). The results indicated that, compared with BCC with BSDS, the proposed CLCBCC with MBSDS improves CR by 7% on average, and >7.5% in the most favorable results (in large benchmarks) on ARM Cortex and C6× target, and a 6% CR improvement on C64×× on average. In other words, the proposed methods are helpful for large programs to save additional memory usage. Figs. 20–22 compare the CR of the proposed CLCBCC with MBSDS against the BCC with BSDS given a constant LUT size for all benchmarks. When a LUT with 512 entries is used for both algorithms, the proposed algorithm exhibits improvements of 7%, 7.5%, and 7% over BCC with BSDS [3], [4] on ARM, C62×× and C64×× targets, respectively.

When an LUT with 1024 entries was used, the improvements are 9%, 9%, and 8%, respectively.

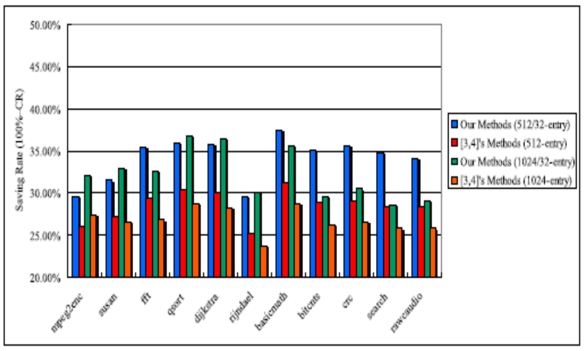

Fig. 22. CR comparison on C64×× target with constant LUT size.

TABLE I

COMPARISON OF DIFFERENT DECODERS

	FBD	CLCBD
Critical path (ns)	2.99	3.06
Area (mm^2)	3.14	3.17
Power (mW)	202.7	205

TABLE II

COMPARISON OF LUT ARCHITECTURE

	DD	FSDD
Critical path (ns)	3.91	1.45
Area (mm^2)	6.26	6.31
Power (mW)	405.9	383.3

No complex hardware technique was involved in improving decompression engine performance. Thus, the proposed FSDD method introduces nearly no hardware overhead, and does not affect the CR while improving execution time and power consumption.

V. CONCLUSION

An improved BCC algorithm is proposed 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. Although the tag overhead to identify the codeword type is increased by 1 bit, the proposed method improves CR by over 7.5% with a slight hardware overhead. 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. Multicore architecture has been a trend in modern embedded products. However, multicore systems require higher communication bandwidths either between the processors and the cache or between the cache and the memory, than singlecore systems. 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 [23] with 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] A. Wolfe and A. Chanin, "Executing compressed programs on an embedded RISC architecture," in *Proc. 25th Annu. Int. Symp. Microarchitecture*, Dec. 1992, pp. 81–91.

[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] *TMS320C62x DSP CPU and Instruction Set Reference Guide*, Texas Instruments, Dallas, TX, USA, Jul. 2006.

[7] 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.

[8] 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.