Efficient Placement of Compressed Code for Parallel Decompression

Xiaoke Qin and Prabhat Mishra

Department of Computer and Information Science and Engineering
University of Florida, Gainesville FL 32611-6120, USA
{*xqin, prabhat*}*@cise.ufl.edu*

Abstract

Code compression is important in embedded systems design since it reduces the code size (memory requirement) and thereby improves overall area, power and performance. Existing researches in this field have explored two directions: efficient compression with slow decompression, or fast decompression at the cost of compression efficiency. This paper combines the advantages of both approaches by introducing a novel bitstream placement method. The contribution of this paper is a novel code placement technique to enable parallel decompression without sacrificing the compression efficiency. The proposed technique splits a single bitstream (instruction binary) fetched from memory into multiple bitstreams, which are then fed into different decoders. As a result, multiple slow-decoders can work simultaneously to produce the effect of high decode bandwidth. Our experimental results demonstrate that our approach can improve decode bandwidth up to four times with minor impact (less than 1%) on compression efficiency.

1 Introduction

Memory is one of the most constrained resources in an embedded system, because a larger memory implies increased area (cost) and higher power/energy requirements. Due to dramatic complexity growth of embedded applications, it is necessary to use larger memories in today's embedded systems to store application binaries. Code compression techniques address this problem by reducing the storage requirement of applications by compressing the application binaries. The compressed binaries are loaded into the main memory, then decoded by a decompression hardware before it's execution in a processor. Compression ratio is widely used as a metric of the efficiency of code compression. It is defined as the ratio (CR) between the compressed program size (CS) and the original program size (OS) i.e., CR = CS / OS. Therefore, a smaller compression ratio implies a better compression technique. There are two major challenges in code compression: i) how to compress the code as much as possible; and ii) how to efficiently decompress the code without affecting the processor performance.

The research in this area can be divided into two categories based on whether it primarily addresses the compression or decompression challenges. The first category tries to improve code compression efficiency using the state-of-the-art coding methods such as Huffman coding [1] and arithmetic coding [2]. Theoretically, they can decrease the compression ratio to its lower bound governed by the intrinsic entropy of code, although their decode bandwidth usually is limited to 6-8 bits per cycle. These sophisticated methods are suitable when the decompression unit is placed between the main memory and cache (pre-cache). However, recent research [3] suggests that it is more profitable to place the decompression unit between the cache and the processor (post-cache). In this way the cache retains data still in a compressed form, increasing cache hits, therefore achieving potential performance gain. Unfortunately, this post-cache decompression unit actually demands much more decode bandwidth than what the first category of techniques can offer. This leads to the second category of research that focuses on higher decompression bandwidth by using relatively simple coding methods to ensure fast decoding. However, the efficiency of the compression result is compromised. The variable-to-fixed coding techniques [12] are suitable for parallel decompression but it sacrifices the compression efficiency due to fixed encoding.

In this paper, we combine the advantages of both approaches by developing a novel bitstream placement technique which enables parallel decompression without sacrificing the compression efficiency. This paper makes two important contributions. First, it is capable of increasing the decode bandwidth by using multiple decoders to work simultaneously to decode a single/adjacent instruction(s). Second, our methodology allows designers to use any existing compression algorithms including variable-length encodings with little or no impact on compression efficiency.

The rest of the paper is organized as follows. Section 2 introduces related work addressing code compression for embedded systems. Section 3 describes our code compression and bitstream placement methods. Section 4 presents our experimental results. Finally, Section 5 concludes the paper.

2 Related Work

A great deal of work has been done in the area of code compression for embedded systems. The basic idea is to take one or more instruction as a symbol and use common coding methods to compress the code. Wolfe and Chanin [1] first proposed the Huffman-coding based code compression approach. A Line Address Table (LAT) is used to handle the addressing of branching within compressed code. Lin et al. [4] uses LZW-based code compression by applying it to variable-sized blocks of VLIW codes. Liao [5] explored dictionary-based compression techniques. Lekatsas et al. [2] constructed SAMC using arithmetic coding based compression. These approaches significantly re-

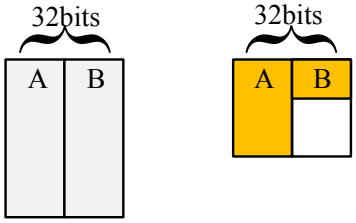
duces the code size but their decode (decompression) bandwidth is limited.

To speed up the decode process, Prakash et al. [6] and Ros et al. [7] improved conventional dictionary based techniques by considering bit changes of a 16-bit or 32-bit vectors. Seong et al. [8] further improved these approaches using bitmask based code compression. These techniques enable fast decompression but they achieve inferior compression efficiency compared to those based on well established coding theory.

Instead of treating each instruction as a single symbol, some researchers observed that the number of different opcodes and operands are quite smaller than that of entire instructions. Therefore, a division of a single instruction into different parts may lead to more effective compression. Nam et al. [9] and Lekatsas et al. [10] broke instructions into several fields then employed different dictionary to encode them. CodePack [11] divided each MIPS instruction at the center, applied two prefix-dictionary to each of them, then combined the encoding results together to create the finial result. However, in their compressed code, all these fields are simply stored one after another (in a serial fashion). The variable-to-fixed coding technique [12] is suitable for parallel decompression but it sacrifices the compression efficiency due to fixed encoding. The variable size encodings (fixed-to-variable and variable-to-variable) can achieve the best possible compression. However, it is impossible to use multiple decoders to decode each part of the same instruction simultaneously, when variable length coding is used. The reason is that the beginning of next field is unknown until the decode of the current field ends. As a result, the decode bandwidth cannot benefit very much from such an instruction division. Our approach allows variable length encoding for efficient compression and proposes a novel placement of compressed code to enable parallel decompression.

3 Efficient Placement of Compressed Binaries

Our work is motivated by previous variable length coding approaches based on instruction partitioning [9, 10, 11] to enable parallel compression of the same instruction. The only obstacle preventing us from decoding all fields of the same instruction simultaneously is that the beginning of each compressed field is unknown unless we decompress all previous fields.



(a) Uncompressed code (b) Compressed code

Figure 1. Intuitive placement for parallel decompression.

One intuitive way to solve this problem, as shown in Figure 1, is to separate the entire code into two parts, compress each of them separately, then place them separately. Using such a placement, the different parts of the same instruction can be decoded simultaneously using two pointers. However, if one part of the code (part B) is more effectively compressed than the other one (part A), the remaining unused space for part B will be wasted. Therefore, the overall compression ratio will be hampered remarkably. Furthermore, the identification of branch targets will also be a problem due to the unequal compression. As mentioned earlier, fixed length encoding methods are suitable for parallel decompression but it sacrifices the compression efficiency due to fixed encoding. The focus of our research is to enable parallel decompression for binaries compressed with variable length encoding methods.

The basic idea of our approach to handle this problem is to develop an efficient bitstream placement method. Our method enables the compression algorithm to make maximum usage of the space automatically. At the same time, the decompression mechanism will be able to determine which part of the newly fetched 32 bits should be sent to which decoder. In this way, we exploit the benefits of instruction division in both compression efficiency and decode bandwidth.

3.1 Overview of Our Approach

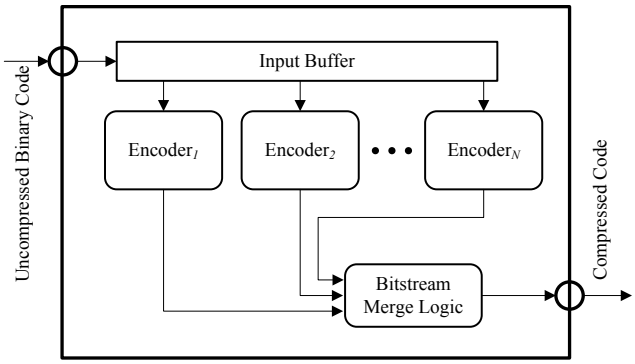
In our approach, we use branch blocks[1] [4] as the basic unit of compression. In other words, our placement technique is applied to each branch blocks in the application. Figure 2 shows the block diagram of our proposed compression framework. It consists of four main stages: compression (encode), bitstream merge, bitstream split and decompression (decode).

During compression (Figure 2a), we first break every input storage block (containing one or more instructions) into several fields, then apply specific encoders to each one of them. The resultant compressed streams are combined together by a bitstream merge logic based on a carefully designed bitstream placement algorithm. Note that the bitstream placement cannot rely on any information invisible to the decompression unit. In other words, the bitstream merge logic should merge streams based on only the binary code itself and the intermediate results produced during the encoding process.

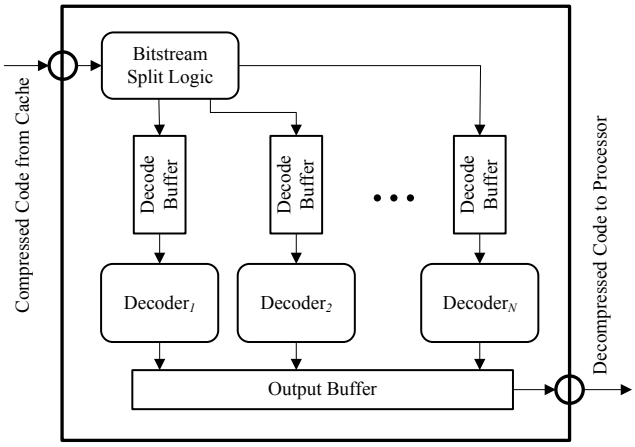
During decompression (Figure 2b), the scenario is exactly the opposite of compression. Every word fetched from the cache is first split into several parts, each of which belongs to a compressed bitstream produced by some encoder. Then the split logic dispatches them to the buffers of correct decoders, according to the bitstream placement algorithm. These decoders decode each bitstream and generate the uncompressed instruction fields. After combining these fields together, we obtain the final decompression result, which should be identical to the corresponding original input storage block (containing one or more instructions).

From the viewpoint of overall performance, the compression algorithm affects the compression ratio and decompression speed in an obvious way. Nevertheless, the bitstream placement actually governs whether multiple decoders are capable to work in parallel. In previous works, researchers tend to use a very simple placement technique: they appended the compressed code for each symbol one after the other. When variable length coding is used, symbols must be decoded in order. In this paper,

[1] The instructions between two consecutive branch targets.



(a) Compression technique



(b) Decompression mechanism

Figure 2. Proposed code compression framework.

we demonstrate how a novel bitstream placement enables parallel decoding and boosts the overall decode performance. The remainder of this section describes the four important stages in our framework: compression, bitstream merge, bitstream split and decompression.

3.2 Compression Algorithm

In our current implementation, we use Huffman coding as the compression algorithm of each single encoder (Encoder$_1$ - Encoder$_N$ in Figure 2 (a)), because Huffman coding is optimal for a symbol-by-symbol coding with a known input probability distribution. To improve its performance on code compression, we modify the basic Huffman coding method [1] in two ways: i) instruction division and ii) selective compression. As mentioned earlier, any compression technique can be used in our framework.

Similar to previous works [9, 10, 11], we believe that compressing different parts of a single instruction separately is profitable, because the number of distinct opcodes and operands is far less than the number of different instructions. We have observed that for most applications it is profitable to divide the instruction at the center. In the rest of this paper, we will use this division pattern, if not stated otherwise.

Selective compression is a common choice in many compression techniques [8]. Since the alphabet for binary code compression is usually very large, Huffman coding may produce many dictionary entries with quite long keywords. This is harmful to the overall compression ratio, because the size of the dictionary entry must also be taken into account. Instead of using bounded Huffman coding, we address this problem using selective compression. First, we create the conventional Huffman coding table. Then we remove any entry e which does not satisfy Equation 1.

$$(Length(Symbol_e) - Length(Key_e)) * Time_e > Size_e, \quad (1)$$

Here, $Symbol_e$ is the uncompressed symbol (one part of an instruction), Key_e is the key of $Symbol_e$ created by Huffman coding, $Time_e$ is the total time for which $Symbol_e$ occurs in the uncompressed code, and $Size_e$ is the space required to store this entry. For example, two unprofitable entries from Dictionary II (Figure 3) are removed.

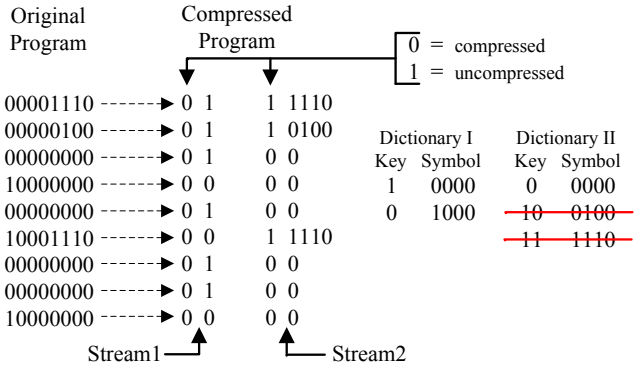


Figure 3. Code compression using modified Huffman coding

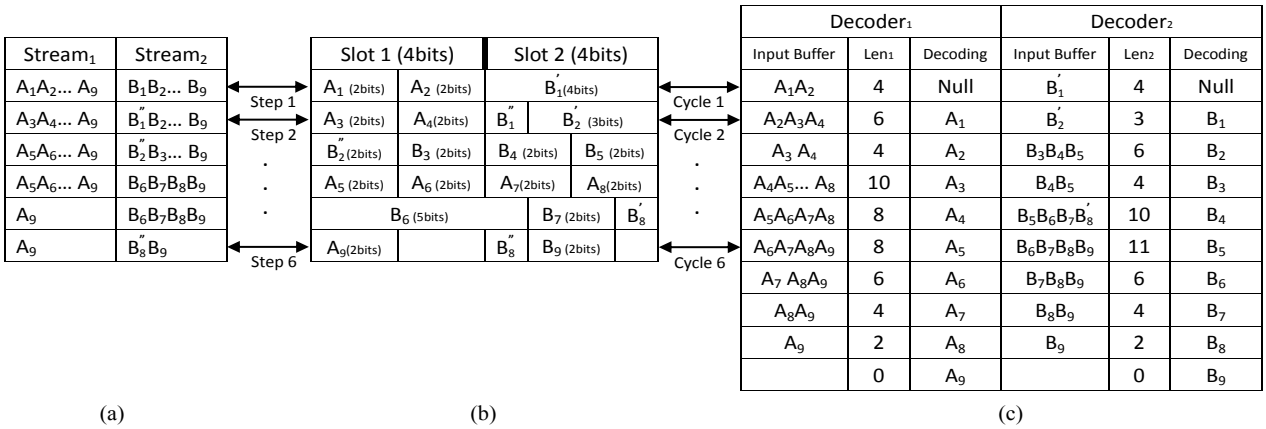
Once the unprofitable entries are removed, we use remaining entries as the dictionary for both compression and decompression. Figure 3 shows an illustrative example of our compression algorithm. For the simplicity of illustration, we use 8-bit binaries instead of 32 bits used in real applications. We divide each instruction in half and use two dictionaries, one for each part. The final compressed program is reduced from 72 bits to 45 bits. The dictionary requires 15 bits. The compression ratio for this example is 83.3%. The two compressed bitstreams (Stream1 and Stream2) are also shown in Figure 4.

Stream1		Stream2	
Symbol	Value	Symbol	Value
A_1	01	B_1	11110
A_2	01	B_2	10100
A_3	01	B_3	00
A_4	00	B_4	00
A_5	01	B_5	00
A_6	00	B_6	11110
A_7	01	B_7	00
A_8	01	B_8	00
A_9	00	B_9	00

Figure 4. Two compressed bitstreams from the code compression example in Figure 3

3.3 Bitstream Merge

The bitstream merge logic merges multiple compressed bitstreams into a single bitstream for storage. We first explain



(a) (b) (c)

Figure 5. Bitstream placement using two bitstreams in Figure 4. (a) Unplaced data remaining in the input buffer of merge logic, (b) Bitstream placement result, (c) Data within Decoder$_1$ and Decoder$_2$ when current storage block is decompressed[2].

some basic models and terms which we will use in the following discussion. Next, we describe the working principle of our bitstream merge logic.

Definition 1: **Storage block** is a block of memory space, which is used as the basic input and output unit of our merge and split logic. Informally, a storage block contains one or more consecutive instructions in a branch block. Figure 6 illustrates the structure of a storage block. We divide it into several slots. Each of them contains adjacent bits extracted from the same compressed bitstream. In our current implementation, all slots within a storage block have the same size.

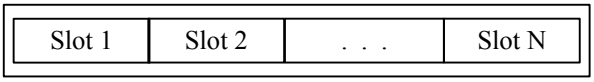


Figure 6. Storage block structure

Definition 2: **Sufficient decode length (SDL)** is the minimum number of bits required to ensure that at least one compressed symbol is in the decode buffer. In our implementation, this number equals one plus the length of an uncompressed instruction field.

Our bitstream merge logic performs two tasks to produce each output storage block filled with compressed bits from multiple bitstreams: i) use the given **bitstream placement algorithm** (BPA) to determine the bitstream placement within current storage block; ii) count the numbers of bits left in each buffer as if they finish decoding current storage block. We pad extra bits after the code at the end of the stream to align on a storage block boundary.

Algorithm 1 is developed to support parallel decompression of two bitstreams. The goal is to guarantee that each decoder has enough bits to decode in the next cycle after they receive the current storage block. Figure 5 illustrates our bitstream merge procedure using previous code compression example in Figure 3. The size of storage blocks and slots are 8 bits and 4 bits respectively. In other words, each storage block has two slots. The SDL is 5. When the merge process begins (translates Figure 5a to Figure 5b), the merge logic gets A_1, A_2 and B'_1, then assigns them to the first and second slots[2]. Similarly, A_3, A_4, B''_1 and B'_2 are placed in the second iteration (step 2). When it comes to the third output block, the merge logic finds that after Decoder$_2$ receives and processes the first two slots, there are only 3 bits left in its buffer, while Decoder$_1$ still has enough bits to decode in the next cycle. So it assigns both slots in the third output block from Stream$_2$. This process repeats until both input (compressed) bitstreams are placed. The "Full()" checks are necessary to prevent the overflow of decoders' input buffers. Our merge logic automatically adjusts the number of slots assigned to each bitstream, depending on whether they are effectively compressed.

Algorithm 1: Placement of Two Bitstreams
Input: Every Storage Block
Output: Placed Bitstreams

if *this is the first block* **then**
  Assign Stream 1 to Slot 1 and Stream 2 to Slot 2
else
  if *!Ready(1) and !Ready(2)* **then**
    Assign Stream 1 to Slot 1 and Stream 2 to Slot 2
  else if *!Ready(1) and Ready(2)* **then**
    Assign Stream 1 to Slot 1 and 2
  else if *Ready(1) and !Ready(2)* **then**
    Assign Stream 2 to Slot 1 and 2
  else if *!Full(1) and !Full(2)* **then**
    Assign Stream 1 to Slot 1 and Stream 2 to Slot 2
end

$Ready(i)$ checks whether the i^{th} decoder's buffer contains at least SDL bits.
$Full(i)$ checks whether corresponding buffer has enough space to hold more slots.

3.4 Bitstream Split

The bitstream split logic uses the reverse procedure of the bitstream merge logic. The bitstream split logic divides the single compressed bitstream into multiple streams using the following guidelines:

- Use the given BPA to determine the bitstream placement within current compressed storage block, then dispatch dif-

[2]We use $'$ and $''$ to indicate the first and second parts of the same compressed instruction in case it does not fit in the same storage block.

ferent slots to the corresponding decoder's buffer.
- If all the decoders are ready to decode the next instruction, start the decoding.
- If the end of current branch block is encountered, force all decoders to start.

We use the example in Figure 5 to illustrate the bitstream split logic. When the placed data in Figure 5b is fed to the bitstream split logic (translates Figure 5b to Figure 5c), the length of the input buffers for both streams are less than SDL. So the split logic determines the first and the second slot must belong to Stream1 and Stream2 respectively in the first two cycles. At the end of the second cycle, the number of bits in the $Decoder_1$ buffer, Len_1 (i.e., 6), is greater than SDL (i.e., 5), but Len_2 (i.e., 3) is smaller than SDL. This indicates that both slots must be assigned to the second bitstream in the next cycle. Therefore, the split logic dispatches both slots to the input buffer of Decoder_2. This process repeats until all placed data are split.

3.5 Decompression Mechanism

The design of our decoder is based on the Huffman decoder hardware proposed by Wolfe et al. [1]. The only additional operation is to check the first bit of an incoming code, in order to determine whether it is compressed using Huffman coding or not. If it is, decode it using the Huffman decoder; otherwise send the rest of the code directly to the output buffer. Therefore, the decode bandwidth of each single decoder (Decoder_1 to Decoder_N in Figure 2 (b)) should be similar to the one given in [1]. Since each decoder can decode 8 bits per cycle, two parallel decoders can produce 16 bits per cycle. Decoders are allowed to begin decoding only when i) all decoders' decoder buffers contains more bits than SDL; or ii) bitstream split logic forces it to begin decoding. After combining the outputs of these parallel decoders together, we obtain the final decompression result.

3.6 Bitstream Placement for Four Streams

In order to further boost the output bandwidth, we have also developed a bitstream placement algorithm which enables four Huffman decoders to work in parallel. During compression, we take every two adjacent instructions as a single input storage block. Four compressed bitstreams are generated by high 16 bits and low 16 bits of all odd instructions, as well as high 16 bits and low 16 bits of all even instructions. We also change the slot size within each output storage block to 8 bits, so that there are 4 slots in each storage block. We omit the complete description of this algorithm here due to the lack of space. However, the basic idea remains the same and it is a direct extension of Algorithm 1. The goal is to provide each decoder with sufficient number of bits so that none of them are idle at any point. Since each decoder can decode 8 bits per cycle, four parallel decoders can produce 32 bits per cycle.

Although we can still employ more decoders, the overall increase of output bandwidth will slow down by more start up stalls. For example, we have to wait 2 cycles to decompress the first instruction using four decoders in the worst case. As a result, high sustainable output bandwidth using too many parallel decoders may not be feasible, if its start up stall time is comparable with the execution time of the code block itself.

4 Experiments

The code compression and parallel decompression experiments of our framework are carried out using different application benchmarks compiled using a wide variety of target architectures.

4.1 Experimental Setup

We used benchmarks from MediaBench and MiBench benchmark suites: adpcm_en, adpcm_de, cjpeg, djpeg, gsm_to, gsm_un, mpeg2enc, mpeg2dec and pegwit. These benchmarks are compiled for four target architectures: TI TMS320C6x, PowerPC, SPARC and MIPS. The TI Code Composer Studio is used to generate the binary for TI TMS320C6x. GCC is used to generate the binary for the rest of them. Our computation of compressed program size includes the size of the compressed code as well as the dictionary and all other data required by our decompression unit.

We have evaluated the relationship between the division position and the compression ratio on different target architectures. We have observed that for most architectures, the middle of each instruction is usually the best partition position. We have also analyzed the impact of dictionary size on compression efficiency using different benchmarks and architectures. Although larger dictionaries produce better compression, our approach produces reasonable compression using only 4096 bytes for all the architectures. Based on these observations, we divide each 32-bit instruction from the middle to create two bitstreams. The maximum dictionary size is set to 4096 bytes. The output bandwidth of the Huffman decoder is computed as 8 bits per cycle [1] in our experiments. To the best of our knowledge, there have been no work on bitstream placement for enabling parallel decompression of variable length coding. So we compare our work (BPA1 and BPA2) with CodePack [11], which uses a conventional bitstream placement method. Here, **BPA1** is our bitstream placement algorithm in Algorithm 1, which enables two decoders to work in parallel, and **BPA2** represents our bitstream placement algorithm in Section 3.6, which supports four parallel decoders.

4.2 Results

Figure 7 shows the efficiency of our different bitstream placement algorithms. Here, "decode bandwidth" means the sustainable output bits per cycle after initial stalls. The number shown in the figure is the average decode bandwidth over all benchmarks. It is important to note that the decode bandwidth for each benchmark also shows the same trend. As expected, the sustainable decode bandwidth increases as the number of decoder grows. Our bitstream placement approach improves the decode bandwidth up to four times. As discussed earlier, it is not profitable to use more than four decoders since it will introduce more start up stalls.

We have studied the impact of bitstream placement on compression efficiency. Figure 8 compares the compression ratios between the three techniques on various benchmarks on MIPS

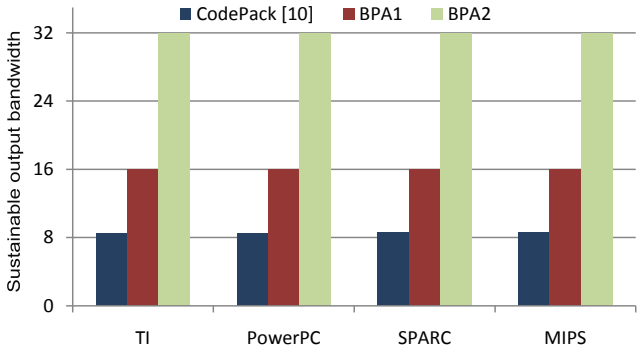


Figure 7. Decode bandwidth of different techniques

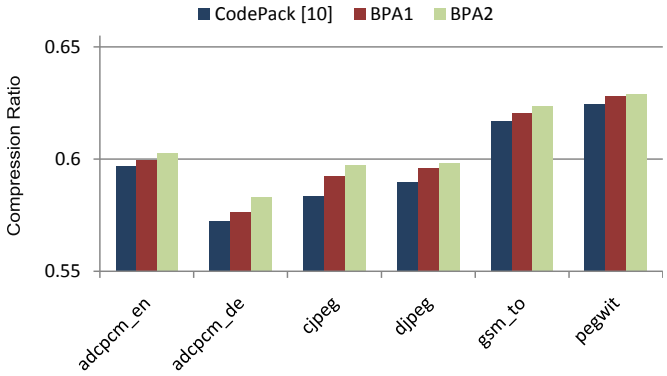


Figure 8. Compression ratio for different benchmarks

architecture. The results show that our implementation of bitstream placement has less than 1% penalty on compression efficiency. This result is consistent across different benchmarks and target architectures as demonstrated in Figure 9 which compares the average compression ratio of all benchmarks on different architectures.

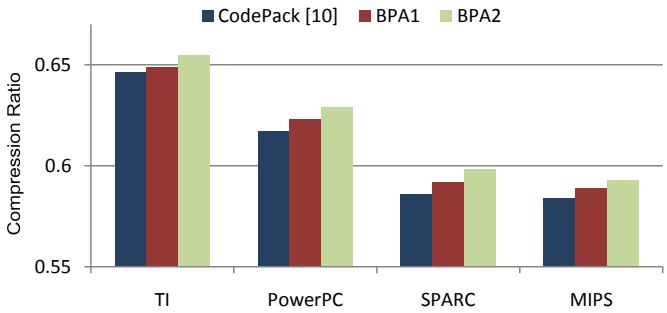


Figure 9. Compression ratio on different architectures

We have implemented the decompression unit using Verilog HDL. The decompression hardware is synthesized using Synopsis Design Compiler and TSMC 0.18 cell library. Table 1 shows the reported results for area, power, and critical path length. It can be seen that "BPA1" (uses 2 16-bit decoders) and CodePack have similar area/power consumption. On the other hand, "BPA2" (uses 4 16-bit decoders) requires almost double the area/power compared to "BPA1" to achieve higher decode bandwidth, because it has two more parallel decoders. The decompression overhead in area and power is negligible (100 to 1000 times smaller) compared to typical reduction in overall area and energy requirements due to code compression.

Table 1. Comparison using different placement algorithms

	CodePack [11]	BPA1	BPA2
Area/μm^2	122263	137529	253586
Power/mW	7.5	9.8	14.6
Critical path length/ns	6.91	5.76	5.94

5 Conclusions

Memory is one of the key driving factors in embedded system design since 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. Existing researches have explored two directions: efficient compression with slow decompression, or fast decompression at the cost of the compression efficiency. This paper combines the advantages of both approaches by introducing a novel bitstream placement technique for parallel decompression. We addressed four challenges to enable parallel decompression using efficient bitstream placement: instruction compression, bitstream merge, bitstream split and decompression. Efficient placement of bitstreams allows the use of multiple decoders to decode different parts of the same/adjacent instruction(s) to enable the increase of decode bandwidth. Our experimental results using different benchmarks and architectures demonstrated that our approach improved the decompression bandwidth up to four times with less than 1% penalty in compression efficiency.

References

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

[2] H. Lekatsas and Wayne Wolf, "SAMC : A code compression algorithm for embedded processors," *IEEE TCAD*, 18(12), 1999.

[3] H. Lekatsas, J. Henkel and W. Wolf, "Code compression for low-power embedded system design," *DAC*, 294–299, 2000.

[4] C. Lin, Y. Xie, and W. Wolf, "LZW-based code compression for VLIW embedded systems," *DATE*, 76–81, 2004.

[5] S. Liao, S. Devadas, and K. Keutzer, "Code density optimization for embedded DSP processors using data compression techniques," *IEEE TCAD*, 17(7), 601–608, 1998.

[6] J. Prakash et al., "A simple and fast scheme for code compression for VLIW processors," *DCC*, pp 444, 2003.

[7] M. Ros and P. Sutton, "A hamming distance based VLIW/EPIC code compression technique," *CASES*, 132–139, 2004.

[8] S. Seong and P. Mishra, "Bitmask-based code compression for embedded systems," *IEEE TCAD*, 27(4), 673–685, April 2008.

[9] S. Nam et al., "Improving dictionary-based code compression in VLIW architectures," *FECCS*, E82-A(11), 2318–2324, 1999.

[10] H. Lekatsas and W. Wolf, "Code compression for embedded systems," *DAC*, 516–521, 1998.

[11] C. Lefurgy, *Efficient Execution of Compressed Programs*, Ph.D. Thesis, University of Michigan, 2000.

[12] Y. Xie et al., "Code compression for embedded VLIW processors using variable-to-fixed coding," *IEEE TVLSI*, 14(5), 2006.