A Universal Placement Technique of Compressed Instructions for Efficient Parallel Decompression

Xiaoke Qin, *Student Member, IEEE*, and Prabhat Mishra, *Senior Member, IEEE*

***Abstract*—Instruction compression is important in embedded system design since it reduces the code size (memory requirement) and thereby improves the overall area, power, and performance. 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 combines the advantages of both approaches by introducing a novel bitstream placement method. Our contribution in this paper is a novel compressed bitstream placement technique to support parallel decompression without sacrificing the compression efficiency. The proposed technique enables splitting 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 simultaneously work to produce the effect of high decode bandwidth. We prove that our approach is a close approximation of the optimal placement scheme. Our experimental results demonstrate that our approach can improve the decode bandwidth up to four times with minor impact (less than 3%) on the compression efficiency.**

***Index Terms*—Code compression, decompression, embedded systems, memory.**

I. 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 the 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 size of application binaries via compression. The compressed binaries are loaded into the main memory and then decoded by a decompression hardware before its execution in a processor. The *compression ratio* (CR) is widely used as a metric for measuring the efficiency of code compression. It is defined as the ratio between the compressed program size (CS) and the original program size (OS), i.e., $\mathrm{CR} = \mathrm{CS}/\mathrm{OS}$. Therefore, *a smaller CR implies a better compression technique.* There are two major challenges in code compression: 1) how to compress the code as much as possible and 2) how to efficiently decompress the code without affecting the processor performance.

Manuscript received November 29, 2008; revised February 7, 2009 and April 2, 2009. Current version published July 17, 2009. This work was supported in part by the National Science Foundation under CAREER Award 0746261. This paper was recommended by Associate Editor G. E. Martin.

The authors are with the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611-6120 USA (e-mail: xqin@cise.ufl.edu; prabhat@cise.ufl.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2009.2021730

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 the code compression efficiency by using state-of-the-art coding methods, such as Huffman coding [1] and arithmetic coding [2], [3]. Theoretically, they can decrease the CR to its lower bound governed by the intrinsic entropy of the code, although their decode bandwidth is usually limited to 6–8 bits/cycle. These sophisticated methods are suitable when the decompression unit is placed between the main memory and the cache (precache). However, recent research [4] suggests that it is more profitable to place the decompression unit between the cache and the processor (postcache). This way, the cache retains the data that are still in a compressed form, increasing the cache hits and therefore achieving potential performance gain. Unfortunately, this postcache decompression unit demands a higher decode bandwidth than what the first category of techniques can offer. This leads to the second category of research that focuses on a 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 [5], [6] 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 that 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 simultaneously work to decode single/adjacent instruction(s). Second, our methodology allows designers to use any existing compression algorithm, including variable-length encodings with little or no impact on the compression efficiency. We also prove that our approach can be viewed as an approximation of the optimal placement scheme, which minimizes the stalls during decompression. We have applied our approach using several compression algorithms to compress applications from various domains compiled for a wide variety of architectures, including TI TMS320C6x, PowerPC, SPARC, and MIPS. Our experimental results show that our approach can improve the decode bandwidth by up to four times with minor impact (less than 3%) on the compression efficiency.

The rest of this paper is organized as follows: Section II introduces the related work that addresses code compression for embedded systems. Section III describes our code compression and bitstream placement methods. Section IV presents our experimental results. Finally, Section V concludes this paper.

II. 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(s) as a symbol and use common coding methods to compress the application programs. Wolfe and Chanin [1] first proposed the Huffman-coding-based code compression approach. A line address table is used to handle the addressing of branches within the compressed code. Bonny and Henkel [7]–[9] further improved the code density by compressing the lookup tables used in Huffman coding. Lin *et al.* [10] used Lempel–Ziv–Welch (LZW)-based compression by applying it to variable-sized blocks of very long instruction word (VLIW) codes. Liao *et al.* [11] explored dictionary-based compression techniques. Das *et al.* [12] proposed a code compression method for variable length instruction set processors. Lekatsas and Wolf [2] constructed SAMC using arithmetic coding-based compression. These approaches significantly reduce the code size, but their decode (decompression) bandwidth is limited.

To speed up the decode process, Prakash *et al.* [13] and Ros and Sutton [14] improved the dictionary-based techniques by considering bit changes. Seong and Mishra [15], [16] further improved these approaches by using bitmask-based code compression. Lekatsas *et al.* [17] proposed a dictionary-based decompression technique, which enables decoding one instruction per cycle. These techniques enable fast decompression, but they achieve inferior compression efficiency compared with those based on well-established coding theory. Instead of treating each instruction as a symbol, some researchers observed that the numbers of different opcodes and operands are quite smaller than that of entire instructions. Therefore, dividing an instruction into different parts may lead to more effective compression. Nam *et al.* [18] and Lekatsas and Wolf [19] divided the instructions into several fields and employed different dictionaries to encode them. Bonny and Henkel [20] separately reencoded different fields of instructions to improve the CR. CodePack [21] divided each MIPS instruction at the center, applied two prefix dictionary to each of them, and then combined the encoding results together to create the finial result. However, in their compressed code, all these fields are stored one after another (in a serial fashion). As a result, the decode bandwidth cannot benefit very much from such an instruction division.

Parallel decoding is a common choice to increase the decoding bandwidth. There are many existing efforts in this field, which can broadly be classified into two categories. The approaches in the first category perform parallel decoding of patterns compressed with fixed-length encoding. Since all the instructions/data are compressed into fixed-length codes, such as variable-to-fixed coding [5], [6] and LZW-based coding [10], the boundaries between codes in the compressed stream are also fixed. Therefore, during decompression, multiple decoders can directly be applied on each code in parallel. Unfortunately, the compression efficiency is usually sacrificed when a fixed-length coding is used [22]. The efforts in the second category enable the parallel decoding of patterns compressed with variable-length coding. Based on the granularity (size of each compressed pattern), the second category can further be divided into two subcategories: block-level (coarse-grained) parallelism and code-level (fine-grained) parallelism. In block-level parallelism, the compressed data are organized into blocks, which are identified by block header or index tables. Decoders are concurrently applied to each block to achieve coarse-grained parallelism. This approach is widely used in multimedia decompression.[1] For example, Iwata *et al.* [23] employed parallel decoding at the slice level (collection of macroblocks) with prescanning for MPEG-2 bitstream decompression. In [24], a macroblock-level parallelization is developed to achieve higher MPEG-2 decoding speed. Roitzsch [25] improved the parallel H.264 decoding performance by applying slice decoding time prediction at the encoder stage. However, it is not practical to directly apply block-level parallelism for decoding compressed instructions. The reason is that if we split the branch blocks for parallel instruction decompression, the resultant blocks are much smaller (10–30 B) compared with the typical macroblocks in MPEG-2 or H.264 (several hundreds of bytes). As a result, common block identification techniques like block header, byte alignment, or index table will introduce a significant overhead in both instruction compression and decompression, which result in unacceptable compression efficiency and reduced decompression performance (for each decoder).

When code-level parallelism is employed, the decompression unit attempts to concurrently decode several successive codes (compressed data) instead of blocks. Since variable-length coding is used, parallel decoding is difficult because we do not know where the next code starts. Nikara *et al.* [26] addressed this problem by using speculative parallel decoders. Since the code boundary is unknown, they need one speculative decoder for each possible position in the input buffer. As a result, their method will require at least 32 decoders to achieve two or four times speed up for a 32-bit binary, whereas our approach only needs two or four decoders. Clearly, their approach will introduce a significant area/power overhead and may not be applicable in many scenarios.

III. Efficient Placement of Compressed Binaries

This paper is motivated by previous variable-length coding approaches based on instruction partitioning [18], [19], [21] to enable the parallel compression of the same instruction. The only obstacle preventing us from simultaneously decoding all fields of the same instruction is that the beginning of each compressed field is unknown unless we decompress all previous fields.

One intuitive way to solve this problem, as shown in Fig. 1, is to separate the entire code into two parts, compress each of them separately, and then place them separately. Using such a placement, the different parts of the same instruction can simultaneously be decoded using two pointers. However, if one part of the code (part B) is more effectively compressed than the other part (part A), then the remaining unused space for

[1]A typical video stream consists of several frames, where each *frame* is a collection of slices, each *slice* is a set of *macroblocks*, and each macroblock consists of several *blocks*.

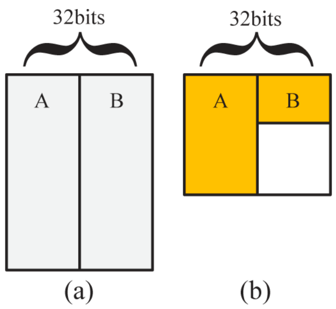


Fig. 1. Intuitive placement for parallel decompression. (a) Uncompressed code. (b) Compressed code.

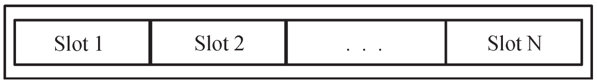


Fig. 2. Storage block structure.

part B will be wasted. Therefore, the overall CR will remarkably be hampered. Furthermore, the identification of branch targets will also be a problem due to 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 this paper is to enable parallel decompression for binaries compressed with variable-length encoding methods.

The basic idea of our approach to address this problem is to develop an efficient bitstream placement method. Our method enables the compression algorithm to automatically make maximum use of the space. 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. This way, we exploit the benefits of instruction division in both compression efficiency and decode bandwidth.

A. Overview of Our Approach

We use the following two definitions to explain our method.

Definition 1: Branch block is "the instructions between two consecutive possible branch targets" [10]. Note that there may be multiple branch instructions within a branch block, but only one branch target at the beginning of each branch block. Therefore, a branch block is a collection of basic blocks with no branch targets between them.

Definition 2: Input Storage block is a block of memory space that is used as the basic input unit of our compression algorithm. Informally, an input storage block contains one or more consecutive instructions in a branch block. Fig. 2 illustrates the structure of an input storage block. We divide it into several slots. Each of them contains adjacent bits of an instruction. All slots within an input storage block have the same size.

Our placement technique is applied to each branch block in the application. Fig. 3 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 [Fig. 3(a)], we first break every input storage block, which contains one or more instructions, into

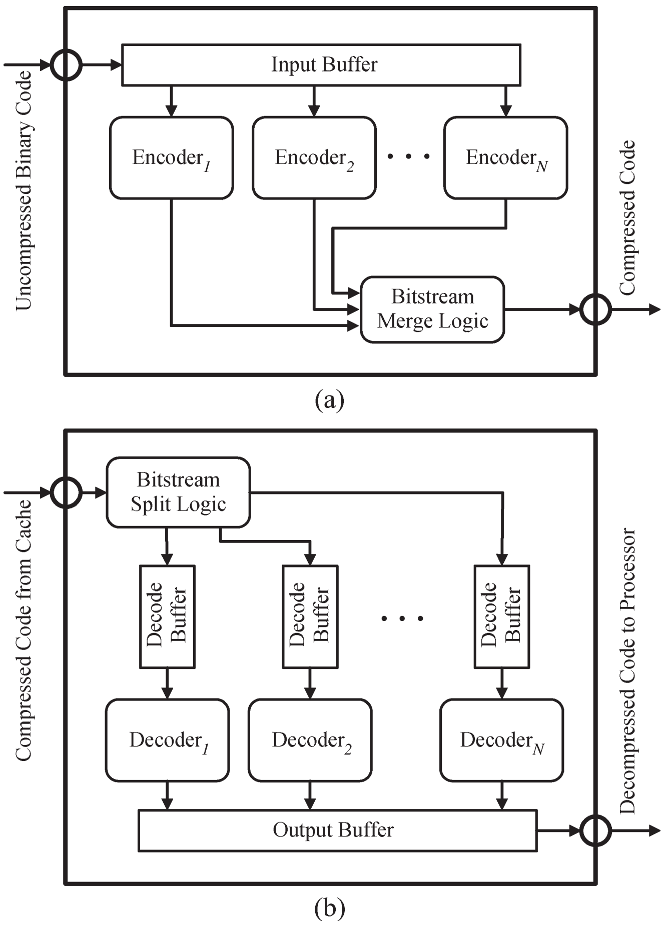


Fig. 3. Proposed instruction compression framework. (a) Compression technique. (b) Decompression mechanism.

several fields and then apply specific encoders to each of them. The resultant compressed streams are combined together by a bitstream merge logic based on a carefully designed bitstream placement algorithm (BPA). 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 solely on the binary code and the intermediate results produced during the encoding process.

During decompression [Fig. 3(b)], 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 BPA. 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.

From the viewpoint of overall performance, the compression algorithm affects the CR and the decompression speed in an obvious way. The bitstream placement actually governs whether multiple decoders are capable of working in parallel. In previous works, researchers tend to use a very simple placement technique, such as appending the compressed code for each symbol one after the other. When variable-length coding is used, the symbols must be decoded in order. In this paper, we demonstrate how a novel bitstream placement enables parallel decoding and boosts the overall decode performance. In the remainder of this section, we describe the four important stages in our framework: compression, bitstream merge, bitstream split, and decompression using an illustrative example. Then, we perform a theoretical analysis of our placement technique and prove that our approach is close to optimal in most practical scenarios. In the following discussion, we use the term *symbol* to refer to a sequence of uncompressed bits used as input to the encoder and *code* to refer to the compression result (of a *symbol*) produced by the encoder.

B. Compression Algorithm

Our bitstream placement method is compatible with a wide variety of fixed-to-variable compression algorithms. The only requirement for the compression algorithm is that the upper bound of the length of code required to produce an uncompressed symbol should be close to the length of an uncompressed symbol. Therefore, our placement method can be used to accelerate "stateless" algorithm like Huffman coding [1], as well as more complex algorithms like arithmetic coding [2], which may maintain internal states across the decompression of successive output symbols.

Before we apply a compression algorithm to separate streams, we need to split the original binary into multiple streams. As stated above, this will enable us to use parallel decoders during decompression. Many previous works [18], [19], [21], [27], [28] suggest 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 use this division pattern, if not stated otherwise.

In the following discussion, we use Huffman coding as the compression algorithm in each encoder [Encoder$_1$ $-$ Encoder$_N$ in Fig. 3(a)]. We use Huffman coding here as an example because it is a pure dictionary-based algorithm with no internal state. It will be much easier to explain how our placement method works based on such a relatively simple compression algorithm. As mentioned earlier, any compression technique with a fixed upper bound of code length can be used in our framework. We also use bitmask-based compression and arithmetic coding to demonstrate the applicability our placement method. The implementation details and experimental results are shown in Section IV.

Our implementation of Huffman coding is based on [1]. To improve its performance on instruction compression, we modify the original scheme by adding selective compression. Selective compression is a common choice in many compression techniques [16], [29]–[32]. Since the alphabet for instruction compression is usually very large, Huffman coding may produce many dictionary entries with a quite long compressed code. This is harmful for overall compression, 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

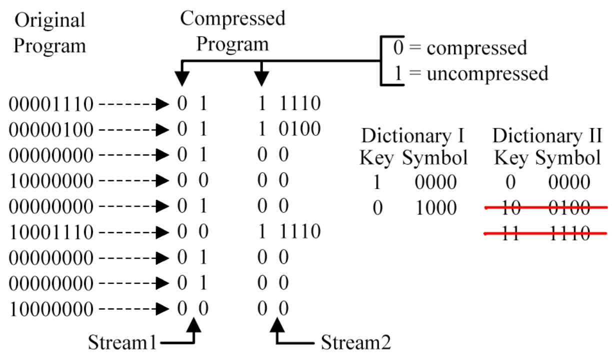


Fig. 4. Instruction compression using modified Huffman coding.

Stream1		Stream2	
Code	Value	Code	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

Fig. 5. Two compressed bitstreams from the example in Fig. 4.

Huffman coding table. Next, we remove any entry e that does not satisfy

$$(Length(Symbol_e) - Length(Code_e)) * Time_e > Size_e. \quad (1)$$

Here, $Symbol_e$ is the uncompressed symbol (one part of an instruction), $Code_e$ is the code of $Symbol_e$ created by Huffman coding, $Time_e$ is the total time for which $Symbol_e$ occurs in the uncompressed symbol sequence, and $Size_e$ is the space required to store this entry. For example, two unprofitable entries from Dictionary II (Fig. 4) are removed.

Once the unprofitable entries are removed, we use the remaining entries to build the dictionary. Fig. 4 shows an illustrative example of our compression algorithm. For the simplicity of illustration, we use 8-bit binaries instead of the 32-bit binaries 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 to 45 bits. The dictionary requires 15 bits. The CR for this example is 83.3%. The two compressed bitstreams (Stream1 and Stream2) are also shown in Fig. 5.

C. Bitstream Merge

The bitstream merge logic merges multiple compressed bitstreams into a single bitstream for storage. We first explain some basic models and terms that we will use in the following discussion. Next, we describe the working principle of our bitstream merge logic.

Definition 3: Output Storage block is a block of memory space that is used as the basic output unit of our merge logic (also the input unit of the split logic). An output storage block contains the placed compressed bitstreams. Like the input storage block, we also divide each output storage block into several slots like Fig. 2. Each slot contains adjacent bits extracted from the same compressed bitstream. For most of the time, all slots within an output storage block have the same size. However, we also allow some slots to have different sizes.

Definition 4: Sufficient decode length (SDL) is the maximum number of bits that should be pushed into the decompressor to produce the next uncompressed symbol. For example, the SDL will be the maximum length of all dictionary entries for Huffman coding. In case of arithmetic coding, the SDL is usually equal to $\lceil log_2(\text{the minimum allowed symbol probability})\rceil$. In our implementation, this number is equal to one plus the length of an uncompressed symbol, because selective compression is used.

Our bitstream merge logic performs two tasks to produce each output storage block filled with compressed bits from multiple bitstreams: 1) use the given BPA to determine the bitstream placement within the current storage block and 2) count the numbers of bits left in each buffer as if they finish decoding the 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**: Placement of Two Bitstreams
**Input**: Every Storage Block
**Output**: Placed Bitstreams

Set the size of Slot1 and Slot2 to $L_{SBlock}/2$;
**if** $!Ready(1)$[2] $and!Ready(2)$ **then**
    **if** $2 * SDL - Len(1) - Len(2) \leq L_{SBlock}$[3] **then**
        Set the size of Slot1 to $SDL - Len(1)$;[4]
        Set the size of Slot2 to $L_{SBlock} - (SDL - Len(1))$;
    **end**
    Assign Stream1 to Slot 1 and Stream2 to Slot 2;
**else if** $!Ready(1)$ and $Ready(2)$ **then**
    Assign Stream1 to Slot 1 and 2;
**else if** $Ready(1)$ and $!Ready(2)$ **then**
    Assign Stream2 to Slot 1 and 2;
**else if** $!Full(1)$[5] $and!Full(2)$ **then**
    Assign Stream1 to Slot 1 and Stream2 to Slot 2;
**else**
    No action;
**end**
```

Algorithm 1 is developed to support the 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. Fig. 6 illustrates our bitstream merge procedure using the example in Fig. 4. The size of the storage blocks and slots are 8 and 4 bits, respectively. In other words, each storage block has two slots. The SDL is 5. When the merge process begins [translates Fig. 6(a) to 6(b)], the merge logic gets A_1, A_2, and B'_1 and then assigns them to the first and second slots.[6] 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 storage 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, whereas Decoder$_1$ still has enough bits to decode in the next cycle. Therefore, it assigns both slots in the third output storage block from Stream2. 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.

D. 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.

1) Use the given BPA to determine the bitstream placement within the current compressed storage block, and then dispatch different slots to the corresponding decoder's buffer.
2) If all the decoders are ready to decode the next instruction, start decoding.
3) If the end of the current branch block is encountered, force all the decoders to start.

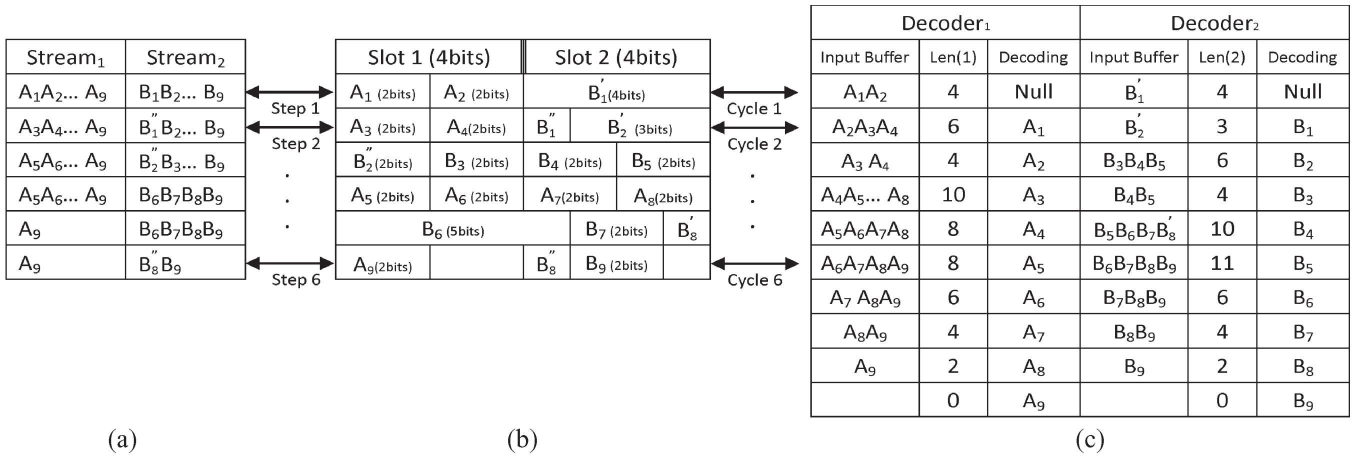
We use the example in Fig. 6 to illustrate the bitstream split logic. When the placed data in Fig. 6(b) is fed to the bitstream split logic [translates Fig. 6(b) to 6(c)], the length of the input buffers for both streams are less than SDL (i.e., 5). Therefore, the split logic determines the first slot, 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 buffer of Decoder$_1$, $Len(1)$ (i.e., 6), is greater than SDL, 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 the placed data are split.

E. Decompression Mechanism

Depending on the input compression algorithm, we employ the corresponding decompression mechanism with minor modification. For example, in case of Huffman coding, the design of our decoder is based on the Huffman decoder hardware proposed by Wolfe and Chanin [1]. The only additional operation is to check the first bit of an incoming code to determine whether it is compressed using Huffman coding. If it is, then 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 Fig. 3(b)] should be similar to the one given in [1]. Since each decoder

[2] $Ready(i)$ checks whether the ith decoder's buffer has at least SDL bits.
[3] L_{SBlock} is the size of each output storage block.
[4] $Len(i)$ returns the number of bits left in the ith decoder's buffer.
[5] $Full(i)$ checks whether the ith buffer has space to hold more slots.

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



Stream$_1$	Stream$_2$
$A_1A_2 \ldots A_9$	$B_1B_2 \ldots B_9$
$A_3A_4 \ldots A_9$	$B''_1B_2 \ldots B_9$
$A_5A_6 \ldots A_9$	$B''_2B_3 \ldots B_9$
$A_5A_6 \ldots A_9$	$B_6B_7B_8B_9$
A_9	$B_6B_7B_8B_9$
A_9	B''_8B_9

Slot 1 (4bits)	Slot 2 (4bits)
A_1 (2bits) A_2 (2bits)	B'_1(4bits)
A_3 (2bits) A_4(2bits)	B''_1 B'_2 (3bits)
B''_2(2bits) B_3 (2bits)	B_4 (2bits) B_5 (2bits)
A_5 (2bits) A_6 (2bits)	A_7(2bits) A_8(2bits)
B_6 (5bits)	B_7 (2bits) B'_8
A_9(2bits)	B''_8 B_9 (2bits)

Decoder$_1$			Decoder$_2$		
Input Buffer	Len(1)	Decoding	Input Buffer	Len(2)	Decoding
A_1A_2	4	Null	B'_1	4	Null
$A_2A_3A_4$	6	A_1	B'_2	3	B_1
A_3 A_4	4	A_2	$B_3B_4B_5$	6	B_2
$A_4A_5 \ldots A_8$	10	A_3	B_4B_5	4	B_3
$A_5A_6A_7A_8$	8	A_4	$B_5B_6B_7B'_8$	10	B_4
$A_6A_7A_8A_9$	8	A_5	$B_6B_7B_8B_9$	11	B_5
A_7 A_8A_9	6	A_6	$B_7B_8B_9$	6	B_6
A_8A_9	4	A_7	B_8B_9	4	B_7
A_9	2	A_8	B_9	2	B_8
	0	A_9		0	B_9

Fig. 6. Bitstream placement using two bitstreams in Fig. 5. (a) Unplaced data remaining in the input buffer of merge logic. (b) Bitstream placement result. (c) Data within Decoder$_1$ and Decoder$_2$ when the current storage block is decompressed.

can decode 8 bits/cycle, two parallel decoders can produce 16 bits per cycle. Decoders are allowed to begin decoding only when 1) all decoder buffers contain more bits than the SDL or 2) the bitstream split logic forces it to begin decoding. After combining the outputs of these parallel decoders together, we obtain the final decompression result.

F. Bitstream Placement for Four Streams

To further boost the output bandwidth, we have also developed a BPA that 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. Each output storage block has 64 bits. Normally, each slot within an output storage block contains 16 bits. Therefore, there are four slots in each storage block. The description of this algorithm is given in Algorithm 2. It is a direct extension of Algorithm 1. The goal is to provide each decoder with a 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 introducing more startup stalls. For example, we have to wait two cycles to decompress the first instruction using four decoders in the worst case. As a result, a high sustainable output bandwidth using too many parallel decoders may not be feasible if its startup stall time is comparable with the execution time of the branch block itself. For example, based on our simulation results on SimpleScalar, while a branch block usually contains 30–50 instructions, only 10.5 instructions are executed before a jump instruction on an average. Therefore, it is not profitable to allow startup stalls to exceed two cycles. However, there are many applications, such as field-programmable gate array (FPGA) configuration bitstream compression [33], where the startup stalls are not so critical. If our framework is used to accelerate decompression in such cases, then it is profitable to use more streams (more than four) to achieve a higher decompression bandwidth.

```
Algorithm 2: Placement of Four Bitstreams
Input: Every Storage Block
Output: Placed Bitstreams
Set the size of Slot 1, 2, 3 and 4 to $L_{SBlock}/4$;
$i_0$, $i_1$, $i_2$, $i_3$ used as stream indexes is any permutation of {1, 2, 3, 4}.
if $!Ready(i_0)$ and $Ready(i_1)$
  and $Ready(i_2)$ and $Ready(i_3)$ then
    Assign Stream $i_0$ to Slot 1, 2, 3 and 4;
else if $!Ready(i_0)$ and $!Ready(i_1)$
  and $Ready(i_2)$ and $Ready(i_3)$ then
    Assign Stream $i_0$ to Slot 1 and 2;
    Assign Stream $i_1$ to Slot 3 and 4;
else if $!Ready(i_0)$ and $!Ready(i_1)$
  and $!Ready(i_2)$ and $Ready(i_3)$ then
    Set the size of Slot $i_0, i_1, i_2$ to $\lfloor L_{SBlock}/3 \rfloor$;
    Assign all bits left in the storage block to $i_3$;
    Assign Stream $i$ to Slot $i$ $(i = 1, 2, 3, 4)$;
else if $!Ready(1)$ and $!Ready(2)$
  and $!Ready(3)$ and $!Ready(4)$ then
    if $4 * SDL - \sum_{i=1}^{4} Buf(i) \leq L_{SBlock}$ then
      Set the size of Slot $i$ to $SDL - Len(i)$ $(i = 1, 2, 3)$;
      Assign all bits left in the storage block to Slot 4;
    end
    Assign Stream $i$ to Slot $i$ $(i = 1, 2, 3, 4)$;
  else if $!Full(1)$ and $!Full(2)$
    and $!Full(3)$ and $!Full(4)$ then
      Assign Stream $i$ to Slot $i$ $(i = 1, 2, 3, 4)$;
  else
    No action;
end
```

G. Analysis of Our BPAs

In this section, we first formalize the bitstream placement problem. Then, we introduce the definition of optimal bitstream placement and illustrate how to construct the theoretically optimal placement. Finally, we prove that our placement methods (BPA1 and BPA2) are very close approximations of the optimal placement scheme. We show that BPA1 and BPA2 will introduce at most one and two stall cycles, respectively.

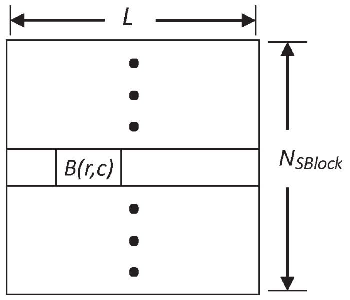


Fig. 7. Storage space structure.

In the postcache decompression scenario, the most important criteria for a placement algorithm is to ensure that it will not cause additional processor stalls. Since multiple decoders are used in parallel, all of them must get enough bits to decode before the complete instruction can be composed and sent to the processor. Therefore, if some decoder cannot get enough bits from the cache to produce the next symbol in a certain cycle, then the processor must stall for at least one cycle. Ideally, an optimal placement algorithm should minimize the total number of stalls required during decompression. However, it may not be easy to directly perform such a minimization. Instead, we will construct a placement algorithm and then prove that it achieves the smallest number of stalls.

To formalize the problem, we first introduce some notations. Since our method is applied to each branch block, we only need to consider the decompression of one branch block. Each branch block is split and compressed into bitstreams and then placed within storage space (as defined below).

Definition 5: Compressed Bitstream is a series of compressed codes that can be fed to a decoder to generate a series of uncompressed symbols, each of which is a part of successive instructions. We denote the set of N_S compressed bitstreams as $S = \{S_i | i \in I\}$, where $I = \{1, \ldots, N_S\}$ is the stream index set. Since they are split from the same branch block, each stream has N_{code} codes. The length of the kth code in stream S_i is $b_i(k)$, $1 \leq k \leq N_{\text{code}}$. In the following discussion, we use "stream" as "compressed bitstream" unless stated otherwise.

Definition 6: Storage Space is a collection of output storage blocks within which we can place all the N_s bitstreams. Formally, a storage space B is a $N_{\text{SBlock}} \times L$ matrix of bits (Fig. 7), which can be expressed as

$$B = \{(r, c) | 1 \leq r \leq N_{\text{SBlock}}, 1 \leq c \leq L\}$$

where L is the number of bits in an output storage block, and N_{SBlock} is the number of output storage blocks used to place these N_S streams.

For example, in BPA1 with Huffman coding, we have two streams $S = \{S_1, S_2\}$, so $N_S = 2$. Here, $b_i(k)$ is the length of a Huffman code representing the original 16-bit symbol. N_{code} is equal to the number of instructions within the branch block, and $L = 32$. In case of BPA2, we have four streams: $S = \{S_1, S_2, S_3, S_4\}$, N_{code} is equal to half of the total number of instructions within the branch block, and $L = 64$, because each output storage block in BPA2 has 64 bits.

Definition 7: The *Placement* of stream set S within storage space B is a function

$$P : B \rightarrow I$$

such that the bit (r, c) is assigned to stream $S_{P(r,c)}$. Note that the selection of P does not need to ensure that such a placement can be split during decompression without using any other information as our approach. Therefore, our proof holds for any possible bitstream placement.

For simplicity, we make some assumptions about the decompression process.

1) For all streams, the decode process of a single code requires one cycle. We perform our discussion based on this assumption to guarantee that our results are valid in the worst case. If the decoding needs more cycles, which happens in many published instruction compression algorithms, then the stalls caused by improper placement can easily be hidden.
2) $(N_S - 1) * SDL \leq L$. Most published instruction compression algorithms satisfy this assumption. They tend not to use too many bits to encode a symbol because such a code will be harmful to both the overall CR and the decode speed.

The remainder of this section is organized as follows: We first define what is the optimal placement of the compressed bitstream. Then, we show that optimal placement exists by constructing a placement P_{OPT} and proving that P_{OPT} is optimal. Finally, we prove that the performance of our practical placement algorithms (BPA1 and BPA2) are very close to this optimal placement with an additional stall of one and two cycles, respectively. It should be noted that P_{OPT} is not a practical placement algorithm, but it is a general theoretical model for analyzing the performance of BPA1 and BPA2.

Intuitively, the kth input storage block (recall that an input storage block contains one or more consecutive instructions) can only be decompressed when all the corresponding compressed codes are sent to the decompressor. If this is not satisfied in the kth cycle, then some stalls are required.

Definition 8: Required Stalls $RS(P, k)$ is the number of stalls required to decompress the kth input storage block on the decompressor side when the compressed streams are placed using P.

By definition, $RS(P, k)$ is equal to the minimum number t that satisfies

$$\forall i \in I : \sum_{j=1}^{k+t} M_i(P, j) \geq \sum_{j=1}^{k} b_i(j) \qquad (2)$$

where $M_i(P, j)$ represents the number of bits assigned to S_i according to placement P in row j of B, and B is the $N_{\text{SBlock}} \times L$ bit matrix in Fig. 7. The left-hand side (LHS) of (2) is the number of bits assigned to stream i in the first $k + t$ rows of B. The right-hand side (RHS) is the total size of the first k codes in stream S_i. LHS must be greater or equal to RHS for any i so that in cycle $k + t$, the kth input storage block can be decompressed.

Since there may be many branch instructions within the branch block, it is usually not profitable to eliminate stalls during the execution of the last part of the branch block by unnecessarily stalling the previous instructions. In other words, the optimal placement should enable the decompressor to output instructions as soon as possible. Formally, we define the optimal placement as follows.

Definition 9: A placement P_0 is *optimal* such that it has a minimum number of required stalls for all the input storage blocks, i.e., $RS(P_0, k)$ is the minimum among any placement P for $1 \le k \le N_{\text{code}}$.

An optimal placement P_{OPT} can be constructed as shown in Algorithm 3. To show its optimality, we use the following Lemma and Theorem 1.

```
**Algorithm 3**: Construction of $P_{\text{OPT}}$
**Input**: $S$, $B$
$t_0 = 0$;
**for** $k = 1$ **to** $N_{\text{code}}$ **do**
    **if**
        $\sum_{i \in I} \left( \sum_{j=1}^{k} b_i(j) - \sum_{j=1}^{k-1+t_{k-1}} M_i(P_{\text{OPT}}, j) \right) > L$    (3)
    **then**
        $t_k = t_{k-1} + 1$;
    **else**
        $t_k = t_{k-1}$;
    **end**
    Choose[7] $P_{\text{OPT}}(r, c)$ from $I$, where
    $k + t_{k-1} \le r \le k + t_k$, $1 \le c \le L$, so that
        $\forall i \in I : \sum_{j=1}^{k+t_k} M_i(P_{\text{OPT}}, j) \ge \sum_{j=1}^{k} b_i(j)$    (4)
    holds;
**end**
```

Lemma: $RS(P, k)$ is bounded by

$$RS(P,k) \ge \max\left\{0, \max_{1 \le l \le k}\left\{\left\lceil \sum_{i \in I}\sum_{j=1}^{l} b_i(j)/L \right\rceil - l\right\}\right\}.$$

Proof: Based on the definition of $RS(P, k)$

$$\forall i \in I : \sum_{j=1}^{k+RS(P,k)} M_i(P, j) \ge \sum_{j=1}^{k} b_i(j).$$

If we sum both sides over I, then we will get

$$(k + RS(P,k))\, L = \sum_{i \in I} \sum_{j=1}^{k+RS(P,k)} M_i(P, j) \ge \sum_{i \in I}\sum_{j=1}^{k} b_i(j)$$

which simply states that all the bits required to decompress the first k input storage blocks must be delivered to the decompressor before the kth input storage block is produced. Therefore

$$RS(P,k) \ge \left\lceil \sum_{i \in I}\sum_{j=1}^{k} b_i(j)/L \right\rceil - k.$$

Since we must decompress one instruction after another, all stalls required to decompress the previous instructions will be cumulated for the following instructions. Therefore

$$RS(P,k) \ge \max\left(RS(P,k-1), \left\lceil \sum_{i \in I}\sum_{j=1}^{k} b_i(j)/L \right\rceil - k\right).$$

If we expand the recursive expression for k times, we have

$$RS(P,k) \ge \max\left\{0, \max_{1 \le l \le k}\left\{\left\lceil \sum_{i \in I}\sum_{j=1}^{l} b_i(j)/L \right\rceil - l\right\}\right\}.$$

■

We denote this lower bound as $MS(k)$

$$MS(k) = \max\left\{0, \max_{1 \le l \le k}\left\{\left\lceil \sum_{i \in I}\sum_{j=1}^{l} b_i(j)/L \right\rceil - l\right\}\right\}.$$

Theorem 1: Any placement P_0 that satisfies

$$\forall i \in I : \sum_{j=1}^{k+MS(k)} M_i(P_0, j) \ge \sum_{j=1}^{k} b_i(j) \tag{5}$$

for all k from 1 to N_{code} is optimal.

Proof: If P_0 satisfies (5) for any k with $1 \le k \le N_{\text{code}}$, then P_0 is guaranteed to generate at most $MS(k)$ stalls when the first k input storage blocks are decompressed. However, $MS(k)$ is also the lower bound of $RS(P, k)$ for any placement P because of the lemma. Therefore, P_0 generates the smallest number of required stalls compared to any placement. Therefore, P_0 is optimal by our definition. ■

Theorem 2: P_{OPT} is an optimal placement.

Proof: We just need to show that P_{OPT} satisfies (5) for all k from 1 up to N_{code}. First, we prove that t_k defined in Algorithm 3 is equal to $MS(k)$ for all k from 1 up to N_{code}. We prove it by induction. In the basis step, t_1 is obviously equal to $MS(1)$ based on the definition. For any number $k \le N_{\text{code}}$, suppose $t_j = MS(j)$ for any j with $1 \le j < k$. Since $MS(k)$ is monotonically increasing with k, there are two cases.

1) $MS(k) > MS(k-1)$. $MS(k)$ must equal $MS(k-1)+1$ because $2L \ge \sum_{j=1}^{k} b_i(j)$. However, we also have $t_k = t_{k-1} + 1$. Otherwise, $t_k = t_{k-1} = RS(P_{\text{OPT}}, k) < MS(k)$, which violates the lemma. Therefore, $MS(k) = t_k$ in this case.
2) $MS(k) = MS(k-1)$. We prove $t_k = t_{k-1}$ by contradiction. If $t_k \ne t_{k-1}$, then (3) must be true. Therefore

$$L + \sum_{i \in I} \sum_{j=1}^{k-1+t_{k-1}} M_i(P_{\text{OPT}}, j) < \sum_{i \in I}\sum_{j=1}^{k} b_i(j).$$

[7]The "choose" statement is always possible. Since $(N_s - 1) * SDL \le L$, $2L \ge \sum_{j=1}^{k} b_i(j)$, for any $i_0 \in I, 1 \le j \le N_{\text{code}}$. This guarantees that two output storage blocks will be enough to place all the bits required to produce any one input storage block.

Or

$$\sum_{i\in I}\sum_{j=1}^{k} b_i(j) > L + L(k-1+t_{k-1}) = L(k+t_{k-1}).$$

Notice that $t_{k-1} = MS(k-1) = MS(k)$, and we have

$$\sum_{i\in I}\sum_{j=1}^{k} b_i(j) > L\left(k + MS(k)\right).$$

Therefore

$$MS(k) < \left\lceil \sum_{i\in I}\sum_{j=1}^{k} b_i(j)/L \right\rceil - k$$

which contradicts the definition of $MS(k)$.

Since we have proved that $t_k = MS(k)$ for all k from 1 up to N_{code}, P_{OPT} automatically satisfies (5) based on our construction. By Theorem 1, P_{OPT} is optimal. ■

Now, we use P_{OPT} as a reference model to evaluate our placement methods BPA1 and BPA2 proposed in the previous sections. First, we define an extended decoder.

Definition 10: Extended decoder is a decoder with a decode buffer (see Fig. 3), which starts decoding when there are at least SDL bits available within its decode buffer. The extended decoder behaves like a "delayed" version of the original decoder. It requires SDL bits to produce the first output symbol. Then, it consumes exactly the same number of bits to produce the kth output symbol as the original decoder consumes to produce the $k-1$th symbol. Formally, let $b'_i(k)$ be the number of bits consumed by the ith extended decoder to produce the kth output symbol. Then, we have

$$b'_i(k) = \begin{cases} SDL, & k=1,\\ b_i(k-1), & \text{otherwise.}\end{cases}$$

Compared with the construction of P_{OPT}, our BPA1 and BPA2 can be viewed as two special cases of P_{OPT} when $N_S = 2$ and 4 with two constraints: 1) they do not have an infinite decode buffer and 2) they use extended decoders. For the first constraint, we observed that for most real branch blocks, a buffer with a length of 50 bits will rarely be fully filled. For the second constraint, the number of stalls required by P_{OPT} using extended decoders is

$$\begin{aligned}
&\max\left\{0, \max_{1\le l\le k}\left\{\left\lceil \sum_{i\in I}\sum_{j=1}^{l} b'_i(j)/L\right\rceil - l\right\}\right\}\\
&= \max\left\{0, \max_{2\le l\le k}\left\{\left\lceil N_S * SDL/L + \sum_{i\in I}\sum_{j=1}^{l-1} b_i(j)/L\right\rceil - l\right\}\right\}\\
&\le \max\left\{0, \lfloor N_S * SDL/L\rfloor + \max_{2\le l\le k}\left\{\left\lceil \sum_{i\in I}\sum_{j=1}^{l-1} b_i(j)/L\right\rceil - (l-1)\right\}\right\}\\
&= \max\left\{0, \lfloor N_S * SDL/L\rfloor + \max_{1\le l\le k-1}\left\{\left\lceil \sum_{i\in I}\sum_{j=1}^{l} b_i(j)/L\right\rceil - l\right\}\right\}\\
&\le MS(k-1) + \lfloor N_S * SDL/L\rfloor\\
&\le MS(k) + \lfloor N_S * SDL/L\rfloor.
\end{aligned}$$

Therefore, our placement method together with the decompression mechanism will cause at most $\lfloor N_S * SDL/L\rfloor$ more stalls compared with P_{OPT}. For BPA1 and BPA2, this bound is equivalent to one and two cycles, respectively.

IV. EXPERIMENTS

The instruction compression and parallel decompression experiments of our framework are carried out using different application benchmarks compiled using a wide variety of target architectures. Our placement method is used with different compression algorithms. We implemented the decompression method using Verilog HDL to evaluate the hardware overhead introduced by our parallel decompression mechanism.

A. 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 CS includes the size of the compressed instructions and the dictionary and all the other data required by our decompression unit.

We have evaluated the relationship between the division position and the CR on different target architectures. Fig. 8 demonstrates the relationship between the division position and the CR on different target architectures. The result is interesting because the center is the best position for most architectures. Only TI TMS320C6x has a slightly biased best position after the seventeenth bit.

The optimal partition position for the TI TMS320C6x instruction set is slightly different, because the position after the seventeenth bit is the common boundary between the opcode and the source/destination operand in this instruction set. For all the other instruction sets like MIPS or PowerPC, this boundary is usually between the fifteenth and sixteenth bits. When a statistical compression scheme like Huffman coding is employed,

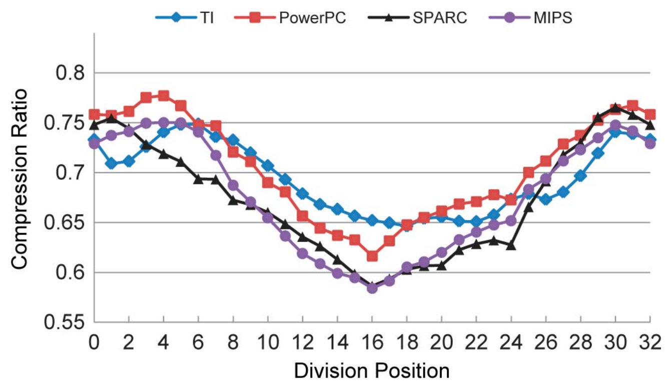


Fig. 8. CR with different division positions.

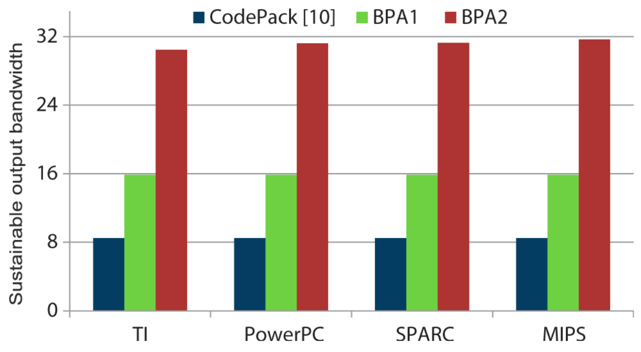


Fig. 9. Decode bandwidth of different techniques.

the number of repeated symbols has a positive impact on the CR. If we partition based on the common boundary between opcode and source/destination operands, it is very likely that the number of repeated symbols will increase, because the same opcode or operand will create the same symbol. This explains the relation between the CR and partition position in our experiments. The fact that the TI TMS320C6x data set has the best partition position after the seventeenth bit just reflects the position of the common boundary in its instruction set.

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 B 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 B. 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 has been no work on bitstream placement for enabling the parallel decompression of variable-length coding. Therefore, we compare our work (BPA1 and BPA2) with CodePack [21], which uses a conventional bitstream placement method. Here, BPA1 is our BPA in Algorithm 1, which enables two decoders to work in parallel, and BPA2 represents Algorithm 2 in Section III-F, which supports four parallel decoders.

B. Decompression Performance With Huffman Coding

Fig. 9 shows the efficiency of our different BPAs. Here, "decode bandwidth" means the sustainable output bits per cycle after the initial stalls. The number shown in the figure is the average decode bandwidth over all benchmarks. It is important to

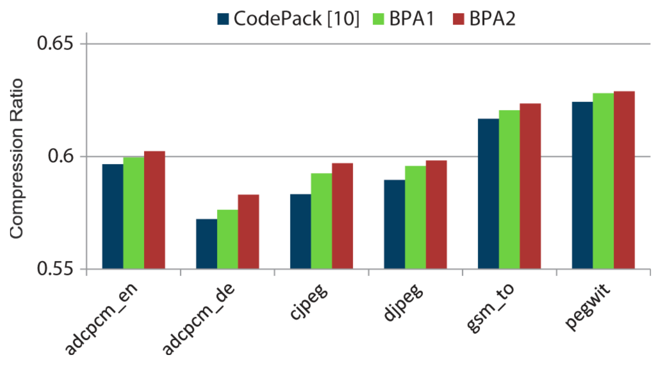


Fig. 10. CR for different benchmarks.

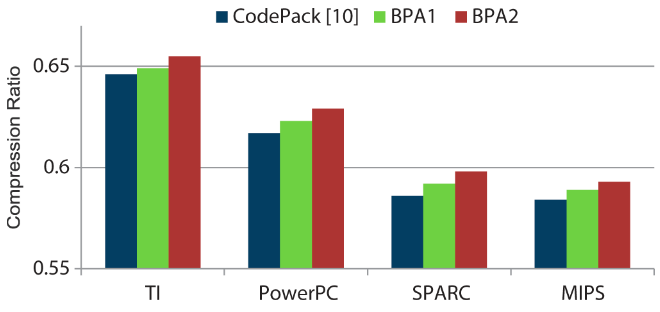


Fig. 11. CR on different architectures.

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 by up to four times. As discussed earlier, it is not profitable to use more than four decoders since it will introduce more startup stalls.

We have studied the impact of bitstream placement on compression efficiency. Fig. 10 compares the CRs between the three techniques using various benchmarks compiled for the MIPS architecture. The results show that our implementation of bitstream placement has less than 3% penalty on compression efficiency. This result is consistent across different benchmarks and target architectures, as demonstrated in Fig. 11, which compares the average CR of all benchmarks on different architectures.

C. Performance With Other Compression Algorithms

We also studied the impact of using our bitstream placement technique on a wide variety of compression algorithms, including arithmetic coding, bitmask-based compression, canonical Huffman coding [9], and Instruction-Set Architecture (ISA) dependent compression [34]. Our implementation of these algorithms is based on [2], [9], [16], and [34], respectively. The only modification we made is that we applied them to separated bitstreams instead of the original instruction stream. For bitmask-based compression, we also changed the dictionary size to 64 per stream. The reason is that after the stream split, the original optimal dictionary size of 4096 can be split into two dictionaries of size 64, since $64 * 64 = 4069$. Figs. 12 and 13 show the CRs using the benchmark programs compiled for the MIPS architecture. The results show that the penalty on CR is quite small.

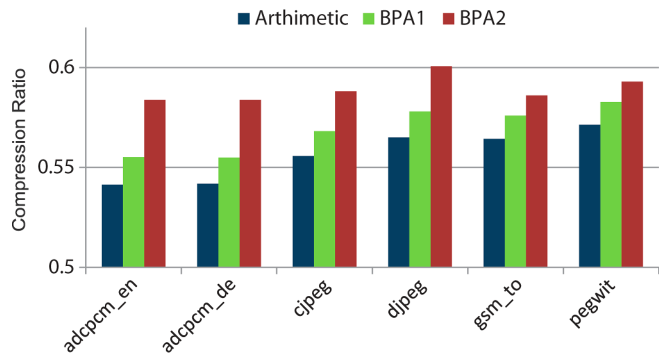


Fig. 12. CR using arithmetic coding.

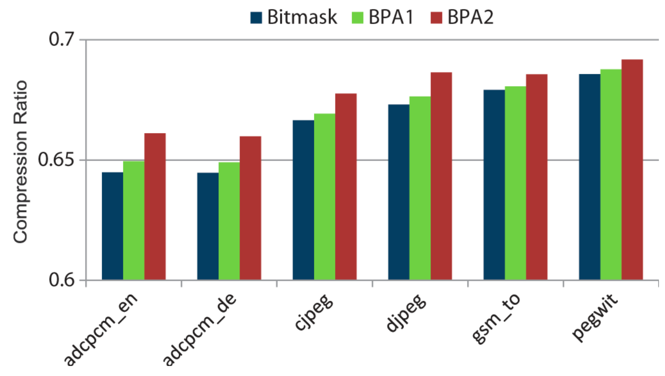


Fig. 13. CR using bitmask-based compression.

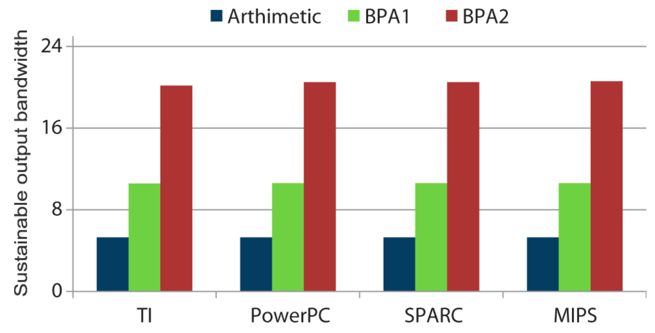


Fig. 14. Decompression bandwidth using arithmetic coding.

Fig. 14 presents the average decompression bandwidth of arithmetic coding using different BPAs. Although arithmetic coding is believed to have the best theoretical compression performance, the decompression is quite slow. The decompression of each 16-bit symbol may need three cycles in the worst case [2]. Our results suggest that we can speed up the decompression of arithmetic coding to four times using parallel decoders. Even for compression algorithms with a high decoding bandwidth, like bitmask-based compression, our approach can still be employed to accelerate its decompression performance. Fig. 15 illustrates the experimental results using bitmask-based compression. With four parallel decoders, the proposed approach can push the decompression bandwidth to the bound determined by the CR and the bandwidth between the instruction cache and the decoder. Note that such output bandwidth will be impossible without using parallel decoders, because each decoder needs at least one cycle to output a 32-bit symbol.

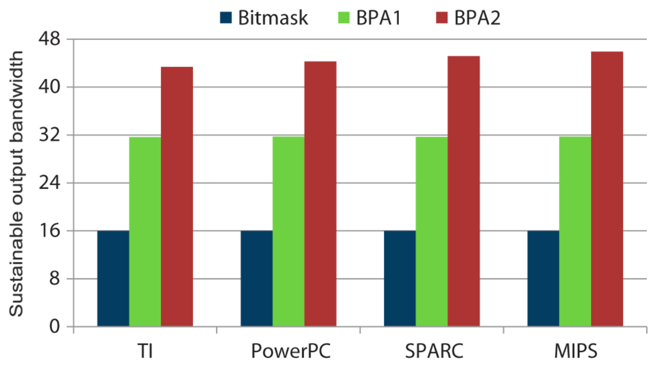


Fig. 15. Decompression bandwidth using bitmask-based compression.

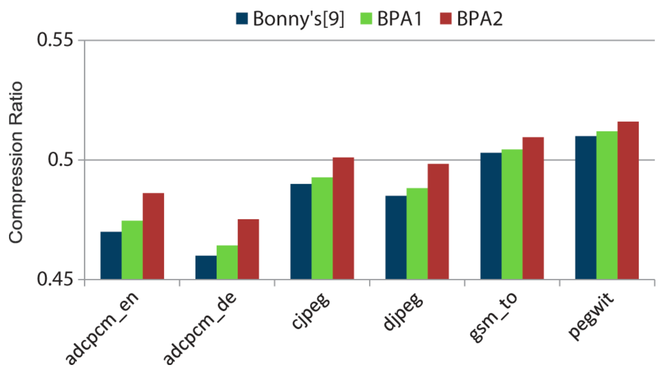


Fig. 16. CR comparison for MIPS using canonical Huffman coding [9].

TABLE I
COMPARISON USING DIFFERENT PLACEMENT ALGORITHMS

	CodePack [21]	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

Canonical Huffman coding with lookup table compression [9] is a more recent compression technique with both good CR and low decode overhead. Since this approach satisfies all the requirements of our parallel decompression technique, we have applied our approach on top of their framework. Fig. 16 shows the compression penalty for employing our approach. It can be seen that the CR penalty introduced by using parallel decoders is small (less than 3%) compared with the results reported in [9]. For the decompression bandwidth, we observed a similar increase (two to four times), as demonstrated in Section IV-B, because the behavior of their decoder is very similar to the selective Huffman decoder used in our previous experiments.

We have also applied our technique with ISA-dependent dictionary-based compression like [34], and the results show a similar decompression bandwidth improvement.

D. Decompression Overhead

We have implemented the Huffman decompression unit using Verilog HDL. The decompression hardware is synthesized using Synopsys Design Compiler [35] and TSMC 0.18 cell library. Table I shows the reported results for area, power, and critical path length.

TABLE II
DECOMPRESSION HARDWARE OVERHEAD USING 180- AND 65-nm TECHNOLOGY

		Area/μm^2	Power/mW		
			Dynamic	Leakage	Total
BPA1	180nm	137529	9.5	0.3	9.8
	65nm	16503	3.5	1.5	5.0
BPA2	180nm	253586	14.1	0.5	14.6
	65nm	30430	5.2	2.2	7.4

It can be seen that "BPA1" (uses 2 16-bit decoders) and CodePack have a similar area/power consumption. However, "BPA2" (uses 4 16-bit decoders) requires almost double the area/power compared with "BPA1" to achieve a 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 with the typical reduction in overall area and energy requirements [4] due to instruction compression.

We have also computed the decompression overhead using the 65-nm library. The results are shown in Table II. As expected, the leakage power has significantly increased (four to five times) compared with the 180-nm technology. Fortunately, the area is decreased by eight times at the same time. Thus, the dynamic power consumption has decreased. As a result, there is no significant difference in overall power consumption.

However, at the rate of four to five times, the leakage power will eventually dominate in the future. In other words, the overall power consumption can increase in 45 nm and beyond. In spite of the increase in power consumption, the overall energy saving due to compression will still be significantly higher than the energy consumed by decompression hardware. To quantitatively show this, let us assume that the power consumed by the decompression engine and the entire system are P_d and P_s, respectively, using current technology. Since code compression improves both cache instruction hits and communication bandwidth, it will improve the overall performance [4]. Suppose that the overall task execution time is reduced by $r\%$, i.e., from T_0 to $(1-r\%)T_0$. The ratio between the total energy saving for this task and the energy consumed by the decompression hardware is

$$\frac{P_s \times T_0 - (P_d + P_s) \times (1-r\%)T_0}{P_d \times (1-r\%)T_0} = \frac{r\%}{1-r\%}\frac{P_s}{P_d} - 1.$$

P_s is usually several orders of magnitude larger than P_d, and the value of $r\%$ is typically 30%–50% [4]. Therefore, based on the above ratio, the energy consumed by the decompression hardware is negligible compared with the overall energy saving due to code compression.

When a new technology is employed, the power consumptions of both the system and the decompression hardware will increase, but P_s/P_d should roughly be constant because the same technology is used for both of them. Since the cache access pattern of the task will not change, $r\%$ will not change either. Hence, even if new technology is used, the relation between the overall energy saving and the energy consumed by the decompression unit will still hold.

TABLE III
SPLIT LOGIC IMPACT ON OF DECOMPRESSION

	Area/μm^2		Critical path length/ns	
	Total	Split logic	Total	Split logic
BPA1	137529	8651 (6.3%)	5.76	0.82
BPA2	253586	11451 (4.5%)	5.94	0.89

We have also studied the impact of bitstream split logic[8] on the area and performance of our decompression unit. The results are shown in Table III. It can be seen that the bitstream split logic occupies a relatively small (4%–6%) area compared with the overall decompression unit. For the critical path, the overhead caused by the split logic is less than 1 ns.

V. CONCLUSION

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. Instruction compression techniques address the problem by reducing the program size. Existing research has explored two directions: efficient compression with slow decompression, or fast decompression at the cost of the compression efficiency. In this paper, we have studied the relationship between bitstream placement, CR, and decompression speed. We developed a novel bitstream placement that enables parallel decoding. This efficient placement of bitstreams allows the use of multiple decoders to decode different parts of the same/adjacent instruction(s) to improve the decode bandwidth. We proved that our approach can be viewed as an approximation of the theoretical optimal placement scheme, which minimizes the number of stalls during decompression. Moreover, the proposed placement method is not restricted to a specific compression algorithm. It can be employed to speed up almost any compression algorithm with negligible penalty in CR. We applied our technique using several typical compression algorithms to demonstrate the usefulness of our approach. Our experimental results using various embedded benchmarks compiled for different architectures demonstrated that our approach improved the decompression bandwidth by up to four times with less than 3% penalty in compression efficiency.

Currently, our technique models the decompressor as a black box that consumes incoming bitstream. In the future, we plan to investigate whether it is possible to make a finer analysis of the bits available within the decode buffer and to predict the number of bits that will be consumed in the next cycle. We also plan to apply our technique in other domains, which usually demand both efficient compression and fast decompression, such as test data compression for manufacturing testing and FPGA bitstream compression.

ACKNOWLEDGMENT

The authors would like to thank the reviewers for their constructive comments and suggestions.

[8]Since the bitstream merge step is done offline during compression, it does not impact the area/performance of the decompression unit.

References

[1] A. Wolfe and A. Chanin, "Executing compressed programs on an embedded RISC architecture," in *Proc. Annu. Int. Symp. Microarchitecture*, 1992, pp. 81–91.

[2] H. Lekatsas and W. Wolf, "SAMC: A code compression algorithm for embedded processors," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 18, no. 12, pp. 1689–1701, Dec. 1999.

[3] H. Lekatsas, J. Henkel, and W. Wolf, "Approximate arithmetic coding for bus transition reduction in low power designs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 13, no. 6, pp. 696–707, Jun. 2005.

[4] H. Lekatsas, J. Henkel, and W. Wolf, "Code compression for low power embedded system design," in *Proc. Des. Autom. Conf.*, 2000, pp. 294–299.

[5] Y. Xie, W. Wolf, and H. Lekatsas, "A code decompression architecture for VLIW processors," in *Proc. Annu. Int. Symp. Microarchitecture*, Dec. 1–5, 2001, pp. 66–75.

[6] Y. Xie, W. Wolf, and H. Lekatsas, "Code compression for VLIW processors using variable-to-fixed coding," in *Proc. Int. Symp. Syst. Synthesis*, Oct. 2–4, 2002, pp. 138–143.

[7] T. Bonny and J. Henkel, "Using Lin–Kernighan algorithm for look-up table compression to improve code density," in *Proc. 16th ACM Great Lakes Symp. VLSI*, 2006, pp. 259–265.

[8] T. Bonny and J. Henkel, "Efficient code density through look-up table compression," in *Proc. Conf. Des., Autom. Test Eur.*, 2007, pp. 809–814.

[9] T. Bonny and J. Henkel, "Efficient code compression for embedded processors," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 12, pp. 1696–1707, Dec. 2008.

[10] C. H. Lin, Y. Xie, and W. Wolf, "LZW-based code compression for VLIW embedded systems," in *Proc. Conf. Des., Autom. Test Eur.*, 2004, vol. 3, pp. 76–81.

[11] S. Liao, S. Devadas, and K. Keutzer, "Code density optimization for embedded DSP processors using datacompression techniques," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 17, no. 7, pp. 601–608, Jul. 1998.

[12] D. Das, R. Kumar, and P. P. Chakrabarti, "Dictionary based code compression for variable length instruction encodings," in *Proc. Int. Conf. VLSI Des.*, 2005, pp. 545–550.

[13] J. Prakash, C. Sandeep, P. Shankar, and Y. Srikant, "A simple and fast scheme for code compression for VLIW processors," in *Proc. Data Compression Conf.*, 2003, p. 444.

[14] M. Ros and P. Sutton, "A hamming distance based VLIW/EPIC code compression technique," in *Proc. Int. Conf. Compilers, Architecture, Synthesis Embedded Syst.*, 2004, pp. 132–139.

[15] S. Seong and P. Mishra, "A bitmask-based code compression technique for embedded systems," in *Proc. Int. Conf. Comput.-Aided Des.*, Nov. 5–9, 2006, pp. 251–254.

[16] S. Seong and P. Mishra, "Bitmask-based code compression for embedded systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 4, pp. 673–685, Apr. 2008.

[17] H. Lekatsas, J. Henkel, and V. Jakkula, "Design of an one-cycle decompression hardware for performance increase in embedded systems," in *Proc. Des. Autom. Conf.*, 2002, pp. 34–39.

[18] S. J. Nam, I. C. Park, and C. M. Kyung, "Improving dictionary-based code compression in VLIW architectures," *IEICE Trans. Fundam. Electron., Commun. Comput. Sci.*, vol. E82-A, no. 11, pp. 2318–2324, Nov. 1999.

[19] H. Lekatsas and W. Wolf, "Code compression for embedded systems," in *Proc. Des. Autom. Conf.*, 1998, pp. 516–521.

[20] T. Bonny and J. Henkel, "Instruction re-encoding facilitating dense embedded code," in *Proc. Conf. Des., Autom. Test Eur.*, 2008, pp. 770–775.

[21] C. R. Lefurgy, "Efficient execution of compressed programs," Ph.D. dissertation, Univ. Michigan, Ann Arbor, MI, 2000.

[22] Y. Xie, W. Wolf, and H. Lekatsas, "Code compression using variable-to-fixed coding based on arithmetic coding," in *Proc. Conf. Data Compression*, 2003, p. 382.

[23] E. Iwata and K. Olukotun, "Exploiting coarse grain parallelism in the MPEG-2 algorithm," Stanford Univ., Stanford, CA, Tech. Rep. CSL-TR-98-771, 1998.

[24] K. Li, H. Chen, Y. Chen, D. W. Clark, P. Cook, S. Damianakis, G. Essl, A. Finkelstein, T. Funkhouser, T. Housel, A. Klein, Z. Liu, E. Praun, J. P. Singh, B. Shedd, J. Pal, G. Tzanetakis, and J. Zheng, "Building and using a scalable display wall system," *IEEE Comput. Graph. Appl.*, vol. 20, no. 4, pp. 29–37, Jul./Aug. 2000.

[25] M. Roitzsch, "Slice-balancing h.264 video encoding for improved scalability of multicore decoding," in *Proc. Int. Conf. Embedded Softw.*, 2007, pp. 269–278.

[26] J. Nikara, S. Vassiliadis, J. Takala, and P. Liuha, "Multiple-symbol parallel decoding for variable length codes," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 12, no. 7, pp. 676–685, Jul. 2004.

[27] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain, "Code compression based on operand factorization," in *Proc. Annu. Int. Symp. Microarchitecture*, 1998, pp. 194–201.

[28] B. Gorjiara, M. Reshadi, and D. Gajski, "Merged dictionary code compression for FPGA implementation of custom microcoded pes," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 1, no. 2, pp. 1–21, Jun. 2008.

[29] J. Lee, W. Hong, and S. Kim, "Design and evaluation of a selective compressed memory system," in *Proc. Int. Conf. Comput. Des.*, 1999, pp. 184–191.

[30] Y. Xie, W. Wolf, and H. Lekatsas, "Profile-driven selective code compression," in *Proc. Conf. Des., Autom. Test Eur.*, 2003, pp. 462–467.

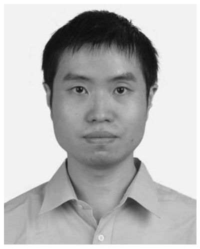
[31] O. Ozturk, H. Saputra, M. Kandemir, and I. Kolcu, "Access pattern-based code compression for memory-constrained embedded systems," in *Proc. Conf. Des., Autom. Test Eur.*, 2005, vol. 2, pp. 882–887.

[32] R. Kumar and D. Das, "Code compression for performance enhancement of variable-length embedded processors," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 1–36, Apr. 2008.

[33] J. H. Pan, T. Mitra, and W. F. Wong, "Configuration bitstream compression for dynamically reconfigurable FPGAs," in *Proc. Int. Conf. Comput.-Aided Des.*, 2004, pp. 766–773.

[34] K. Lin and C. P. Chung, "Code compression techniques using operand field remapping," *Proc. Inst. Elect. Eng.—Comput. Digital Tech.*, vol. 149, no. 1, pp. 25–31, Jan. 2002.

[35] [Online]. Available: http://www.synopsys.com/



Xiaoke Qin (S'08) received the B.S. and M.S. degrees from Tsinghua University, Beijing, China, in 2004 and 2007, respectively. He is currently working toward the Ph.D. degree in the Department of Computer and Information Science and Engineering, University of Florida, Gainesville.

His research interests are in the area of code compression, model checking, and system verification.

Prabhat Mishra (S'00–M'04–SM'08) received the B.E. degree in computer science from Jadavpur University, Calcutta, India, in 1994, the M.Tech. degree in computer science from the Indian Institute of Technology, Kharagpur, India, in 1996, and the Ph.D. degree in computer science from the University of California, Irvine, in 2004.

He spent several years with various semiconductor and design automation companies including Texas Instruments Incorporated, Synopsys, Intel, and Freescale (Motorola). He is currently an Assistant Professor in the Department of Computer and Information Science and Engineering, University of Florida, Gainesville. His research interests include the design automation of embedded systems, reconfigurable architectures, and functional verification. He is the coauthor of the book *Functional Verification of Programmable Embedded Architectures* (Kluwer, 2005). He is also the coeditor of the book *Processor Description Languages* (Morgan Kaufmann, 2008).

Dr. Mishra is a Professional Member of the Association for Computing Machinery (ACM). He currently serves as the Program Chair of the IEEE High Level Design Validation and Test workshop, Information Director of the ACM Transactions on Design Automation of Electronic Systems, Guest Editor of the Springer International Journal of Parallel Programming (IJPP), as a program/organizing committee member of several ACM and IEEE conferences, and as a reviewer of many premier journals, conferences, and workshops. His research has been recognized by various awards, including the CODES+ISSS Best Paper Award in 2003, the EDAA Outstanding Dissertation Award in 2005, and an NSF CAREER Award in 2008.