Decoding-Aware Compression of FPGA Bitstreams

Xiaoke Qin, *Member, IEEE*, Chetan Muthry, and Prabhat Mishra, *Senior Member, IEEE*

***Abstract*—Bitstream compression is important in reconfigurable system design since it reduces the bitstream size and the memory requirement. It also improves the communication bandwidth and thereby decreases the reconfiguration time. Existing research in this field has explored two directions: efficient compression with slow decompression or fast decompression at the cost of compression efficiency. This paper proposes a novel decode-aware compression technique to improve both compression and decompression efficiencies. The three major contributions of this paper are: 1) smart placement of compressed bitstreams that can significantly decrease the overhead of decompression engine; 2) selection of profitable parameters for bitstream compression; and 3) efficient combination of bitmask-based compression and run length encoding of repetitive patterns. Our proposed technique outperforms the existing compression approaches by 15%, while our decompression hardware for variable-length coding is capable of operating at the speed closest to the best known field-programmable gate array-based decoder for fixed-length coding.**

***Index Terms*—Bitmask-based compression, bitstream compression, decompression hardware, field-programmable gate array (FPGA).**

I. Introduction

FIELD-PROGRAMMABLE GATE ARRAYS (FPGAs) are widely used in reconfigurable systems. Since the configuration information for FPGA has to be stored in internal or external memory as bitstreams, the limited memory size, and access bandwidth become the key factors in determining the different functionalities that a system can be configured and how quickly the configuration can be performed. While it is quite costly to employ memory with more capacity and access bandwidth, bitstream compression technique alleviates the memory constraint by reducing the size of the bitstreams. With compressed bitstreams, more configuration information can be stored using the same memory. The access delay is also reduced, because less bits need to be transferred through the memory interface. To measure the efficiency of bitstream compression, *compression ratio* (CR) is widely used as a metric. It is defined as the ratio between the compressed bitstream size (CS) and the original bitstream size (OS) i.e., $\mathrm{CR} = \mathrm{CS}/\mathrm{OS}$. Therefore, *a smaller compression ratio implies a better compression technique*. There are two major challenges in bitstream compression: 1) how to compress the bitstream as much as possible and 2) how to efficiently decompress the bitstream without affecting the reconfiguration time.

We can classify the existing bitstream compression techniques into two categories. The techniques in the first category have good compression ratio due to complex and variable-length coding (VLC) [1]–[3]. However, they also need expensive decompression hardware, which may not be acceptable for practical implementation. The other category of compression approaches accelerate decompression using simple or fixed-length coding (FLC) [4] and therefore have very efficient decompression hardware. The only concern is that their compression ratios are usually compromised.

Among various compression techniques that has been proposed in recent years, application of bitmask-based compression [5] seems to be attractive for bitstream compression, because of its good compression ratio and relatively simple decompression scheme. However, the original algorithm is proposed for instruction compression and not suitable for FPGA bitstream compression. Moreover, the use of variable-length coding is challenging for the design of decompression hardware because it requires expensive buffering circuitry as described in Section III. Hence, it is a major challenge to develop an efficient compression technique that can significantly reduce the bitstream size without sacrificing the decompression performance.

Our approach combines the advantages of previous compression techniques with good compression ratio and those with fast decompression. This paper makes three important contributions. First, it performs smart placement of compressed bitstreams to enable fast decompression of variable-length coding. Next, it selects bitmask-based compression parameters suitable for bitstream compression. Finally, it efficiently combines run length encoding and bitmask-based compression to obtain better compression and faster decompression.

The rest of this paper is organized as follows. Section II surveys the existing bitstream compression techniques used in FPGA configuration bitstreams. Section III discusses challenges of applying bitmask-based coding for FPGA bitstreams. Section IV describes our bitmask-based bitstream compression technique and associated placement of compressed bitstreams. Section V presents the experimental results. Finally, Section VI concludes this paper.

II. Related Work

The existing bitstream compression techniques can be classified into two categories based on whether they need special hardware support during decompression. Some approaches require special hardware features to access the configuration memory, like wildcard register, partial reconfiguration, or frame readback, which are provided only by certain FPGAs. For example, the wildcard compression scheme [6] is developed for the Xilinx XC6200 series FPGA, which support wildcard registers. Using these registers, the same logic configuration

Manuscript received June 16, 2009; revised August 30, 2009. First published December 08, 2009; current version published February 24, 2011. This work was supported in part by NSF Grant CNS-0915376.

The authors are with the Department of Computer and Information Science and Engineering, the University of Florida, Gainesville, FL 32611-6120 USA (e-mail: xqin@cise.ufl.edu; cmurthy@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/TVLSI.2009.2035704

can be written to multiple cells by a single operation. Pan *et al.* [1] used frame reordering and active frame readback to achieve better redundancy. The difference between consecutive frames (difference vector) is encoded using either Huffman-based run length encoding or LZSS-based compression. Such sophisticated encoding schemes can produce excellent compression. However, they did not address the decompression overhead in [1], which is a major bottleneck in reconfigurable systems.

In contrast, many bitstream compression techniques only access the configuration memory linearly during decompression, and therefore can be applied to virtually all FPGAs. The basic idea behind most of these techniques is to divide the entire bitstream into many small words, then compress them with common algorithms such as Huffman coding [7], arithmetic coding [8], or dictionary-based compression. Among them, LZSS-based algorithms have received special interest because the compressed stream can be decoded efficiently without complex hardware. For instance, Xilinx [9] introduced a bitstream compression algorithm based on LZ77 which is integrated in the System ACE controller. Huebner *et al.* [10] proposed an LZSS-based technique for Xilinx Virtex XCV2000E FPGA. The decompression engine is designed carefully to achieve fast decompression. Stefan *et al.* [11] observed that simpler algorithms like LZSS successfully maintains decompression overhead in an acceptable range but compromises on compression efficiency. On the other hand, compression techniques using complex algorithms can achieve significant compression but incurs considerable hardware overhead during decompression. Unfortunately, the authors did not model the buffering circuitry of the decompression engine in their work. Hence the hardware overhead presented for some variable-length coding techniques may be inaccurate.

To increase the decompression throughput of complex compression algorithms, parallel decompression can be used. Nikara *et al.* [12] improved the throughput employing speculative parallel decoders. Qin *et al.* [13] introduced a placement technique of compressed bitstreams to enable parallel decompression. However, since the structure of each decoder and buffering circuitry are not changed, the area overhead is also multiplied. Most importantly, this approach does not reduce the speed overhead introduced by the buffering circuitry for VLC bitstream. In contrast, our proposed approach will significantly improve the maximum operating frequency by effectively addressing the buffering circuitry problem.

III. Background and Motivation

In this section, we briefly analyze the decompression hardware complexity of common variable-length compression techniques. This analysis forms the basis of our approach. In the following discussion, we use the term **symbol** to refer to a sequence of uncompressed bits and **code** to refer to the compression result (of a **symbol**) produced by the compression algorithm. While compression efficiency is straightforward and widely used criteria to evaluate compression techniques, the complexity of decompression hardware determines whether an algorithm with promising compression ratio can be applied to commercial FPGAs. Interestingly, our study shows that the complexity of the decompression algorithm is not the only determining factor of the hardware complexity. When variable-length coding is employed, the hardware complexity is also determined by the complex buffering circuitry, which is overlooked by previous bitstream compression approaches.

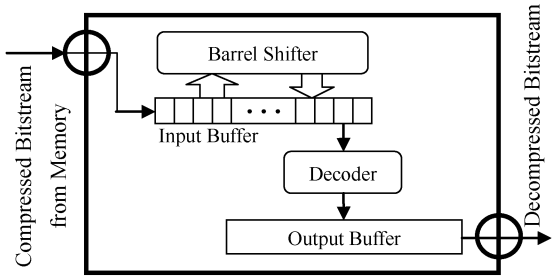


Fig. 1. Decompression engine structure.

The structure of a general decompression engine is shown in Fig. 1. An input buffer is required to store compressed bits fetched from the memory. In each cycle, several bits from the front of the input buffer is consumed by the decode logic to produce next output symbol. At the same time, new data from memory is shifted to fill the input buffer. The input buffer also has to be shifted so that bits in the next code is aligned to the beginning of the input buffer. From the hardware perspective, the area of the design is determined by the decode logic and the buffering circuitry, including the shift logic. The maximum operating speed is governed by the length of the critical path, which comprises of the identification of the code length and the shift operation.

While different coding schemes has quite different decoding hardware, their buffering circuitries also vary significantly. For fixed-length coding [4], [14], [15], the append and shift operations can be easily performed with reasonable $O(n)$ hardware cost for an n-bit buffer, because all codes has the same length and only one shift distance is necessary. On the other hand, buffering circuitries for variable-length coding are much more complex.

Since each code has different length, we have to use barrel shifter to align the input buffer in each cycle. Theoretically, a barrel shifter operating on a n-bit buffer needs $n\log(n)$ multiplexers (MUXes) organized into $\log(n)$ layers. When implemented in modern FPGAs (Xilinx Virtex II or Virtex 4), the barrel shifter for an input buffer with typical size of 32–64 bits consumes 200–400 four-input lookup tables (LUTs), which is similar to the total area of a typical bitmask or Huffman decoder. Moreover, barrel shifter will increase the latency remarkably by introducing several layers of combinational logic in the critical path. Therefore, the buffering circuitry is a major bottleneck in a decompression engine for variable-length coding both in terms of area and performance. Its importance is overlooked by previous works on VLC based bitstream compression, like [1] and [11].

Our work in this direction is motivated by this remarkable difference of the buffering circuitry complexity between the fixed-length coding and the variable-length coding. The basic idea of our approach is to split a single VLC bitstream into multiple FLC streams after compression, then reconstruct the original VLC bitstream from FLC bitstreams during decompression. Since FLC streams are buffered separately, it might be possible to simplify the complex buffering circuitry for VLC bitstream in terms of area and critical path length.

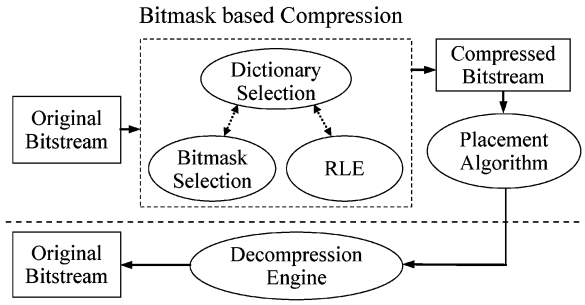


Fig. 2. Decode-aware bitstream compression framework.

IV. Decode-Aware Bitstream Compression

Fig. 2 shows our decode-aware bitstream compression framework. On the compression side, FPGA configuration bitstream is analyzed for selection of profitable dictionary entries and bitmask patterns. The compressed bitstream is then generated using bitmask-based compression and run length encoding (RLE). Next, our decode-aware placement algorithm is employed to place the compressed bitstream in the memory for efficient decompression. During run-time, the compressed bitstream is transmitted from the memory to the decompression engine, and the original configuration bitstream is produced by decompression.

Algorithm 1 outlines four important steps in our decode-aware compression framework (shown in Fig. 2): 1) bitmask selection; 2) dictionary selection; 3) integrated RLE compression; and 4) decode-aware placement. The input bitstream is first divided into a sequence of symbols with length of w. Then bitmask patterns and dictionary entries used for bitmask-based compression are selected as described in Section IV-A and Section IV-B. Next, the symbol sequence is compressed using bitmask and RLE. We use the same algorithm in [5] to perform the bitmask-based compression. The RLE compression in our algorithm is discussed in Section IV-C. Finally, we place the compressed bitstream into a decode friendly layout within the memory using placement algorithm in Section IV-C.

Algorithm 1: Decode-Aware Bitstream Compression

Input: Input bitstream

Output: Compressed bitstream placed in memory

Step 1: Divide input bitstream into symbol sequence SL.

Step 2: Perform bitmask pattern selection.

Step 3: Perform dictionary selection.

Step 4: Compress symbol SL into code sequence CL using bitmask and RLE.

Step 5: Perform decode aware placement of CL.

Since memory and communication bus are designed in multiple of bytes (8 bits), storing dictionaries or transmitting data other than multiple of byte size is not efficient. Thus, we restrict the symbol length to be multiples of eight in our current implementation. Since the dictionary for bitstream compression is smaller compared to the size of the bitstream itself, we use

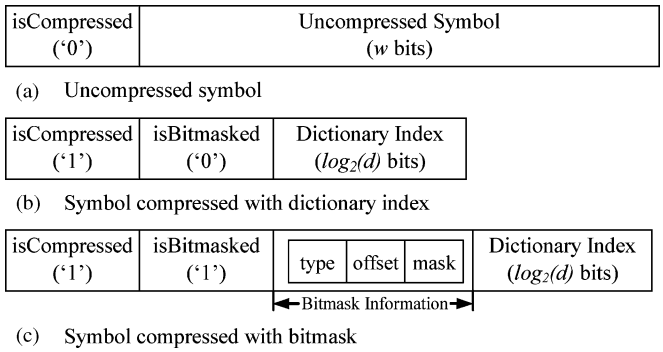


Fig. 3. Encoding formats in bitmask-based compression. (a) Uncompressed symbol. (b) Symbol compressed with dictionary index. (c) Symbol compressed with bitmask.

$d = 2^i$ to fully utilize the bits for dictionary indexing, where i is the number of indexing bits.

A. Bitmask Selection

Our bitmask-based compression is similar to [5], where three types of encoding formats are used. Fig. 3 shows the formats in these cases: no compression, compression using dictionary, and compression using bitmask. The selection of bitmask plays an important role in bitmask-based compression. Generally, there are two types of bitmask patterns. One is "fixed" bitmask, which can only be applied on fixed positions in a symbol. The other one is "sliding" bitmask, which can be applied at any position. For example, a 2-bit fixed bitmask ("2f" bitmask) is restricted to be used on even locations, but a 2-bit sliding bitmask ("2s" bitmask) can be used anywhere. Clearly, fixed bitmasks require less bits to encode its location, but they can only match bit changes at fixed positions. On the other hand, sliding bitmasks are more flexible, but consume more bits to encode. In other words, only a few number of bitmask patterns or their combinations are profitable for compression. Similar to [5], in our study of bitstream compression, we only use profitable bitmask patterns $(1s, 2s, 2f, 3s, 3f, 4s, 4f)$.

B. Dictionary Selection

Our dictionary selection algorithm is motivated by the bit-savings based dictionary selection technique proposed by Seong *et al.* [5]. The symbol space is represented as a graph $G = (V, E)$, where node $n \in V$ represents a symbol and edge $(n, n') \in E$ indicates that symbols corresponding to n and n' can represent each other by some bitmask. The goal of dictionary selection is to find the subset $D \subseteq V$, such that the bit-saving due to compression is maximized when D is used as dictionary.

Algorithm 2 shows our dictionary selection algorithm. Compared to the dictionary selection approach proposed in [5] for instruction compression, we made an important optimization at Step 5). In the original algorithm [5], any node adjacent to the most profitable node is removed, if its profit is less than certain threshold. This mechanism is designed to reduce the dictionary size. However, if the threshold is not chosen properly, some high frequency symbols may be incorrectly removed. Since the dictionary size in bitstream compression is usually negligible compared with the size of the bitstream, it is not beneficial to reduce the dictionary size by scarifying the compression

ratio. Therefore, our algorithm used new heuristics in Step 5), which carefully removes edges instead of nodes. Experimental results in Section V-A show that our approach is more suitable for bitstream compression, because we ensure better dictionary coverage.

Algorithm 2: Decode-Aware Dictionary Selection

Input: Input symbol sequence SL, Parameters $\{w, d\}$

Output: Dictionary D

Step 1: Construct graph $\mathbf{G} = (V, E)$ from SL.

Step 2: **Calculate bit savings** $S_{TS}(n)$ **of all** $n \in V$.

Step 3: Select the most profitable node N.

Step 4: Remove N from $\mathbf{G}$ and insert into D.

Step 5: **For each node** $v \in adj(N)$, **if the edge between the adjacent nodes and** N **is duplicated then remove that edge**.

Step 6: Repeat Steps 2 to 5 until D is full or $\mathbf{G}$ is empty.

return D

C. Run Length Encoding of Compressed Words

The configuration bitstream usually contains consecutive repeating bit sequences. Although the bitmask-based compression [5] encodes such patterns using same repeated compressed words, it is suggested in [2] and [4] that run length encoding (RLE) of these sequences may yield a better compression result. Interestingly, to represent such encoding no extra bits are needed. Note that bitmask value 0 is never used, because this value means that it is an exact match and would have encoded using zero bitmasks. Using this as a special marker, these repetitions can be encoded without changing the code format of bitmask-based compression.

Fig. 4 illustrates the bitmask-based RLE. The input contains word "00000000" repeating five times. In normal bitmask-based compression these words will be compressed with repeated compressed words, whereas our approach replaces such repetitions using a bitmask of "00". In this example, the first occurrence will be encoded as usual, whereas the remaining 4 repetitions will be encoded using RLE. The number of repetition is encoded as bitmask offset and dictionary bits combined together. In this example, the bitmask offset is "10" and dictionary index is "0". Therefore, the number of repetition will be "100" (i.e., 4).

The compressed words are run length encoded only if the RLE yields a shorter code length than the original bitmask encoding. In other words, if there are r repetitions of code with length l and the number of bits required to encode them using RLE is l' bits, RLE is used only if $r * l > l'$ bits. Since RLE is performed independently, the bit savings calculation during dictionary selection (see Section IV-B) should be modified accordingly to model the effect of RLE.

D. Decode-Aware Placement of Compressed Bitstreams

The placement algorithm places all bitmask codes in the memory so that they can be decompressed using the efficient decompression hardware in Section IV-E. The basic idea is to split the original single VLC bitstream into multiple FLC bitstreams for storage. During decompression, these FLC bitstreams are buffered separately, then used to reconstruct the original bitstream by bitmask decoding. Since the buffering circuitry for FLC bitstreams is much simpler than that of VLC bitstreams, the overall decompression performance will be improved even when multiple FLC buffers are used. In the rest of this section, we first define "power-two n-bit stream," which we will use in the following discussion. Then we describe how we split the original compressed bitstream into multiple FLC bitstreams and how to place these FLC bitstreams into the memory in such a way that the original bitstream can be reconstructed during decompression.

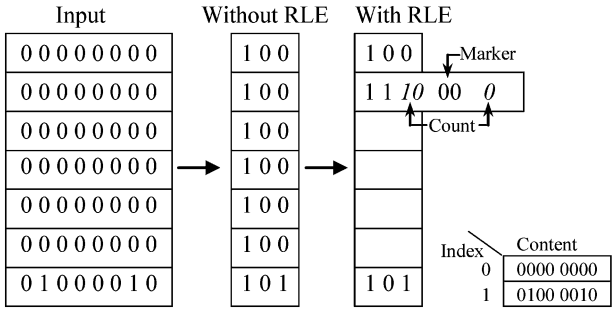


Fig. 4. RLE-based compression.

Definition 1: **Power-Two n**-bit Stream ("**PT-n** stream" for short) is FLC stream of n-bit codes, where n is a power of two such as 2^0, 2^1, 2^2, and so on. Since each code in a PT-n stream has the same length of n, the shift distance is fixed when a PT-n stream is buffered.

Algorithm 3 describes the process to split the original single VLC bitstream into multiple power-two streams. It is developed based on the binary representation of the code length. Once power-two streams are constructed from the original bitstream, we use Algorithm 4 to place all power-two streams within the memory in such a way that we can reconstruct the original bitstream on decompression side. The basic idea is to exactly simulate the decompression process and always assign the memory line to the power-two stream which is required to decode the next code by the decompression engine.

Algorithm 3: Construction of power-two streams

Input: Compressed Code Sequence CL, Memory Bandwidth b

Output: Power-two Stream List

Initialize empty PT-1 streams CS and BS;

Initialize empty PT-n streams $PT-1, PT-2, \ldots, PT-b$;

foreach *Code* e *in* CL **do**

Append "isCompressed" flag of e to CS;

Append "isBitmasked" flag of e (if any) to BS;

if e *is NOT compressed* **then** $L \leftarrow L_u - 1$[1];

[1] L_u is the code length for unmatched symbols.

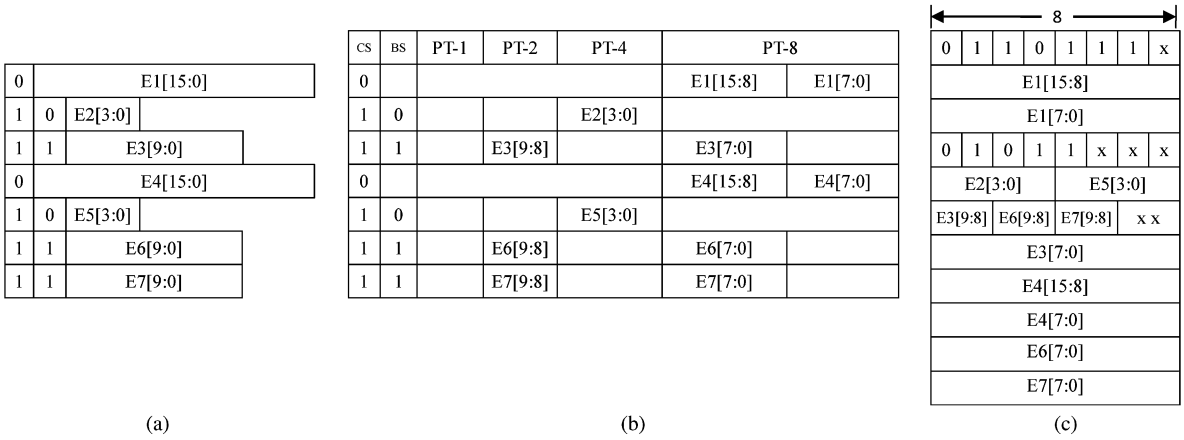


Fig. 5. Decoding aware placement of bitmask codes. (a) Bitmask codes. (b) Constructed power-two streams. (c) Power-two streams placement result. "x" implies unused space.

else if e *is bitmasked* **then** $L \leftarrow L_b - 2$[2];

else $L \leftarrow L_d - 2$[3];

for $j = 0$ **to** $\log_2 b - 1$ **s.t.** $L\&2^j \neq 0$ **do**

Append $e[L-1 : L-2^j]$ to stream $PT - 2^j$;

$L \leftarrow L - 2^j$;

end

if $L \geq b$ **then** Append $e[L-1:0]$ to stream $PT - b$;

end

return $CS, BS, PT-1, PT-2, PT-4, \ldots, PT-b$

We use Fig. 5 to illustrate the application of Algorithm 3 and 4. Fig. 5(a) shows a sample output of the bitmask-based compression algorithm. In this example, the symbol length is 16. The dictionary has 16 entry, thus 4 bits are required for dictionary index. Hence, the code length is $1 + 16 = 17$ for unmatched symbols, $2 + 4 = 6$ for fully matched symbols. For symbols matched using 2-bit bitmask, we need 2 bits for the bitmask pattern and 4 bits to encode the 15 different positions. Therefore, the code length is $2 + 4 + 2 + 4 = 12$.

First, we take Fig. 5(a) as input to produce the power-two streams [see Fig. 5(b)] using Algorithm 3. Initially, there are five nonempty power-two streams: CS, BS, PT-2, PT-4, and PT-8, where CS and BS are the streams for "isCompressed" and "isBitmasked" flags. PT-1 is empty because all codes in this example have even length without flag bits. In the first iteration of Algorithm 3, since the memory bandwidth is 8, all bits in E1 is assigned to stream PT-8, because its length is 16 without the flag bit. In the second iteration, since the length of E2 without flags is 4, it is appended to stream PT-4. Similarly, E3[9:8] is assigned to PT-2, while the remaining bits of E3 are assigned to PT-8, because its length without flags is 10. This process repeats until all the codes are correctly split into the power-two streams.

[2] L_b is the code length for symbols matched with bitmasks.

[3] L_d is the code length for fully matched symbols.

Next, Algorithm 4 takes power-two streams in Fig. 5(b) as input and produces the memory placement in Fig. 5(c). The first line of the memory is assigned to stream CS because the $CScnt$ which simulates the counter within the decompression engine is zero. Then the second and third memory lines are assigned to PT-8 as E1 is not compressed. The fourth line is assigned to BS, because E2 is compressed and $BScnt = 0$. When it comes to the fifth line, since E2 is contained in PT-4, we assign this line to PT-4. Note that this memory line also contains E5, which is the next code in PT-4. Similarly, next two memory lines will be assigned to PT-2 and PT-8, because E3 is contained in both streams. This process repeats until all power-two streams are placed.

The maximum number of power-two streams is determined by the memory bandwidth. Since the length of any code can be written as $\sum_{i=0}^{\log_2 b - 1} a_i * 2^i + a' * 2^b$, where a_i and a' are integers, and b is memory bandwidth, it is easy to see that any bitmask code stream can be split into $\log_2 b + 3$ power-two streams: $\mathrm{CS}, \mathrm{BS}, \mathrm{PT}-1, \mathrm{PT}-2, \ldots, \mathrm{PT}-b/2$ and $\mathrm{PT}-b$. Since the total length of some power-two streams may not be a multiple of the memory bandwidth b, the above placement technique may waste some space in memory. For example, all bits marked "x" in Fig. 5(c) are not used to place any compressed information and therefore wasted. More precisely, the following theorem shows the upper bound of wasted space.

Theorem 1: The total number of unused bits N_w is less than $(\log_2 b + 2) * b$, where b is the memory bandwidth.

Proof: The $\mathrm{PT}-b$ stream will not waste any space because its total length must be a multiple of b. Therefore, in the worst case, we have $\log_2 b + 2$ power-two streams whose length are not multiple of b. In Algorithm 4, the space in a memory line is wasted, only if this line is used to place the last several bits of a power-two stream. As a result, we have at most $\log_2 b + 2$ memory lines that are not fully filled. Notice that none of these lines is completely empty, the total number of unused bits $N_w < (\log_2 b + 2) * b$. ■

Considering the fact that common size of FPGA configuration bitstreams are hundreds of kilobytes to several megabytes, $(2 + \log_2 b) * b$ bits overhead (40 bits with 8-bit memory) virtually has no impact on the overall compression ratio.

Algorithm 4: Placement of power-two streams

Input: Compressed Code Sequence CL
PT Streams CS, BS, PT-1, PT-2, ... , PT-b
Empty memory M
Output: Memory with Placed Bitstreams
Reset all the counters:
$CScnt, BScnt, cnt[0], cnt[1], ..., cnt[\log_2 b - 1]$;
foreach e *in* CL **do**
  if $CScnt = 0$ **then**
    Assign next line in M to CS;
    $CScnt \leftarrow b$;
  end
  if e *is NOT compressed* **then**
    Assign next $(L_u - 1)/b$ lines in M to PT-b;
  else
    if $BScnt = 0$ **then**
      Assign next line in M to BS;
      $BScnt \leftarrow b$;
    end
    if e *is bitmasked* **then** $L \leftarrow L_b - 2$ **else** $L \leftarrow L_d - 2$;
    for *j=0* **to** $\log_2 b - 1$ **s.t.** $L\&2^j \neq 0$ **do**
      if $cnt[j] = 0$ **then**
        Assign next line in M to PT-2^j;
        $cnt[j] \leftarrow b/2^j$;
      end
      $cnt[j] \leftarrow cnt[j] - 1$;
    end
    if $L \geq b$ **then**
      Assign next $\lfloor l/b \rfloor$ lines in M to PT-b;
    end
    $BScnt \leftarrow BScnt - 1$;
  end
  $CScnt \leftarrow CScnt - 1$;
end
return M

E. Decompression Engine

The decompression engine is a hardware component used to decode the compressed configuration bitstream and feed the uncompressed bitstream to the configuration unit in FPGAs. As discussed in Section III, a decompression engine usually has two parts: the buffering circuitry is used to buffer and align codes fetched from the memory, while decoders perform decompression operation to generate original symbols. Since the decoders are well studied in previous literatures, we implement our bitmask and RLE decoder based on designs proposed by Seong *et al.* [5] and Koch *et al.* [4] respectively. In the rest of this section, we will mainly focus on our novel buffering circuitry for the decompression engine and our decompression algorithm.

The structure of our decompression engine for 8-bit memory is shown in Fig. 6. An "Assemble Buffer with a Left Shifter Array" (ABLSA) is employed to replace the original "Buffer with a Barrel Shifter" (BBS) buffering circuitry in Fig. 1. The basic working principle of ABLSA is to use an array of left shift registers to buffer the power-two bitstreams separately. Since the code length in bitmask-based compression is uniquely determined by the first two bits of a code (isCompressed and isBitmasked flags), we can easily obtain the length of a code by checking of front bits of stream CS and BS. Then, the shift register (or PT streams) that hold bits of the code is identified based on the binary representation of the code length. Finally, the original code is assembled in the assemble buffer and fed to the bitmask or RLE decoders. When some shifter becomes empty,

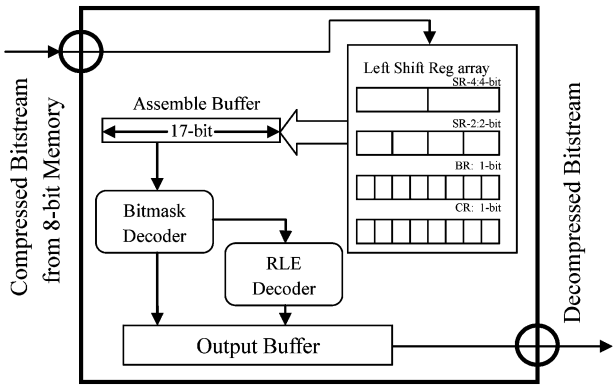


Fig. 6. Decompression engine.

it is guaranteed to be loaded correctly by our decompression algorithm.

Before delving into our algorithm, we first describe the general structure of ABLSA. Let the maximum code length and the memory bandwidth be l and b. Since stream $\mathrm{PT}-b$ is sent to the assemble buffer directly, there are at most $\log_2(b)+2$ power-two streams to be buffered. Therefore, ABLSA contains $\log_2(b)+2$ shift registers, including two 1-bit shift registers CR, BR used to buffer CS and BS, respectively, and other $\log_2(b)$ shift registers $\mathrm{SR}-1, \mathrm{SR}-2, \ldots, \mathrm{SR}-b/2$ used to buffer streams PT-1, PT-2, PT-4, ..., and $\mathrm{PT}-b/2$ respectively. Each of them has the capacity of b bits, same as the memory bandwidth. The size of the assemble buffer AB is l, because AB only holds one code at a time. Fig. 6 shows a specific case of our decompression engine for $b = 8$. There are two 1-bit shift registers, a 2-bit shift register, and a 4-bit shift register. They are used to buffer stream CS, BS, PT-2 and PT-4, respectively. Each of these shifter has the same capacity of 8 bits. Note that we omit shift register SR-1, because PT-1 is empty in this case. Also we do not have SR-8 since 8-bits from stream PT-8 will be directly transferred to the assemble buffer.

Our decompression mechanism can be viewed as the reverse of the placement procedure in Algorithm 4. We determine the code type by checking CR and BR, then assemble the code using bits buffered in different shift registers. If any shift register is empty, they are reloaded using incoming memory lines. Since the decompression algorithm and the placement algorithm use the exact same control flow to map power-two streams with memory lines, it is guaranteed that the original bitstream can be reconstructed in the same order.

We use the example in Fig. 5 to illustrate the bitstream split logic, which consumes Fig. 5(c) and produces Fig. 5(b). When the placed data in Fig. 5(c) is fed to the decompression engine in Fig. 6, $CScnt = 0$ at the beginning, so the first line "0110111x" is loaded into CR. Next, since $CR[7] = 0$, the code to be assembled is not compressed, and its length is $L_u - 1 = 16$. Thus, we assemble it using the next two memory lines, which contains E1[15:8] and E1[7:0], then shift CR for 1 bit. For the second code, since current $CR[7] = 1$ and $BScnt = 0$, we first load BR with the fourth memory line "01011xxx". After that, we have $BR[7] = 0$, which indicates that the next code to be assembled is compressed with fully matching, and its length is $L_d - 2 = 6$. Therefore, we load SR-4 with the next line, which contains E2[3:0] and E5[3:0]. E2[3:0] is then sent to the

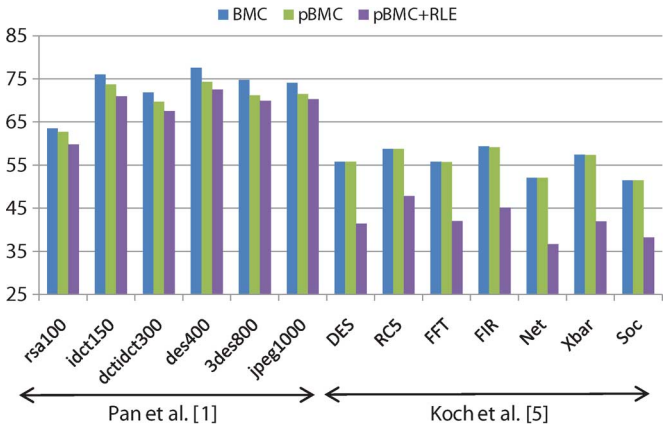


Fig. 7. Comparison of compression ratio with bitmask-based compression.

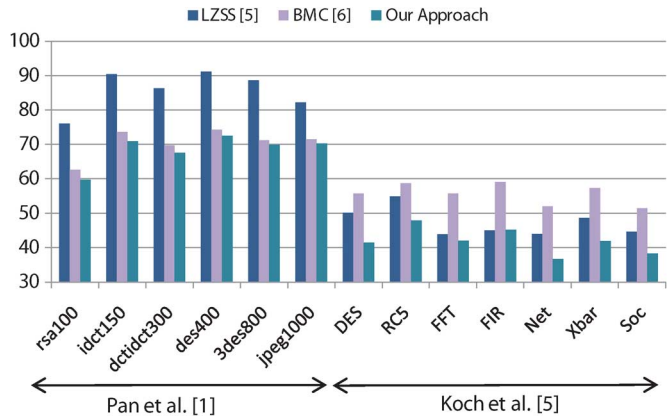


Fig. 8. Comparison of compression ratio with LZSS and BMC.

assemble buffer for decoding. This process repeats until all data placed in memory are decompressed.

Now we show that ABLSA actually requires less area and shorter critical path length than BBS when implemented in FPGA. Recall that the BBS buffering circuitry has to shift the input buffer in each cycle so that bits in the next code is aligned to the beginning of the input buffer. The maximum shift distance is equal to the maximum code length l, because the front part of the buffer was occupied by the previous code. Since the input buffer should be able to hold at least one code plus a memory line, its minimum size is $b + l$. Therefore, it consumes $\lceil \log_2(l) \rceil (b + l)$ MUXes for the barrel shifter and $b + l$ flip-flops (FFs) for the buffer registers.

For ABLSA, we have $\log_2(b)+2$ left shift registers and a l-bit assemble buffer. As a result, the ABLSA consumes $(\log_2(b) + 2)b + l$ MUXes and FFs. Clearly, ABLSA has less combinational logic than BBS (roughly 50% less MUXes for $b = 8$ and $l = 17$). Since commercial FPGAs usually has same number of MUXes and FFs in a slice [16], [17], this guarantees that the total area of ABLSA is smaller than BBS even if the FF count for ABLSA is higher. In case of critical path length, ABLSA performs fixed distance shift on each buffer. Since these shift operations are accomplished in parallel, it adds only one layer of logic on the critical path. In contrast, the barrel shifter in BBS has $\lceil \log_2(l) \rceil$ layers of cascaded combinational logic. Therefore, BBS has a slower maximum operating speed. These conclusions are also supported by our experimental results in Section V-B.

V. Experiments

We used two sets of hard to compress IP core bitstreams chosen from image processing and encryption domain (derived from Koch *et al.* [4] and Pan *et al.* [1]) to compare compression and decompression efficiencies. We used Xilinx Virtex-II family IP core benchmarks to analyze the results in this article. The same results are found applicable to other families and vendors too. We compared our approach with existing best known distance vector (DV)-based bitstream compression technique proposed by Pan *et al.* [1] and best known parameterized LZSS based decompression accelerator proposed by Koch *et al.* [4]. In our experiments, Pan *et al.* [1] benchmarks are compressed with 32 bit symbols, 512 entry dictionary entries and two sliding 2- and 3-bit bitmasks for storing bitmask differences. Koch *et al.* [4] benchmarks are compressed using 16 bit symbols, with 16 entry dictionary and a 2-bit sliding bitmask.

A. Compression Efficiency

We first compare our improved bitmask-based compression technique with the original approach proposed in [5]. To avoid the bias caused by parameter selection, we use the same bitmask parameters for both of them. Three different compression techniques are compared for compression efficiency: 1) bitmask-based compression (BMC) [5]; 2) BMC with our dictionary selection technique (pBMC); and 3) BMC with our dictionary selection technique and run length encoding (pBMC+RLE). Fig. 7 shows the compression results on Pan *et al.* [1] and Koch *et al.* [4] benchmarks.

It can be seen that our dictionary selection algorithm outperforms the original technique [5] on Pan *et al.* [1] benchmarks. The dictionary generated by our algorithm improves the compression ratio by 4% to 5%. The reason is that Pan *et al.* [1] benchmark requires large dictionaries for better compression ratio (size up to 1 K entries). Since in our approach we do not have to find the threshold value manually for each bitstream, our algorithm adaptively finds the most suitable dictionary entries for each bitstream. On the other hand, our method has the same performance as [5] on Koch *et al.* [4] benchmarks, which only require a small dictionary.

The experimental results also illustrate the improvement of compression ratio due to the run length encoding used in our technique. The column pBMC+RLE in Fig. 7 shows improvement on all the benchmarks. On an average we found 5% to 7% reduction over pure bitmask-based compression for Pan *et al.* [1] benchmarks and 15% improvement on Koch *et al.* [4] benchmarks. This is due to the fact that FPGA configuration bitstreams usually have many repetitive patterns. Our RLE encoding technique adaptively compresses these patterns without compromising the effectiveness of bitmask-based compression technique.

However, RLE alone cannot compress all bitstreams effectively. To show the advantages of bitmask-based compression, we compared BMC [5] and our approach with LZSS [4], which also employs RLE. The results are given in Fig. 8. It can be observed that pure LZSS is quite effective (10% better than BMC) on Koch *et al.* [4] benchmarks, because these benchmarks have large amount of repetitive patterns, which are suitable for run length encoding. Nevertheless, LZSS is not able to reduce the bitstream size significantly on Pan *et al.* [1] benchmarks, which

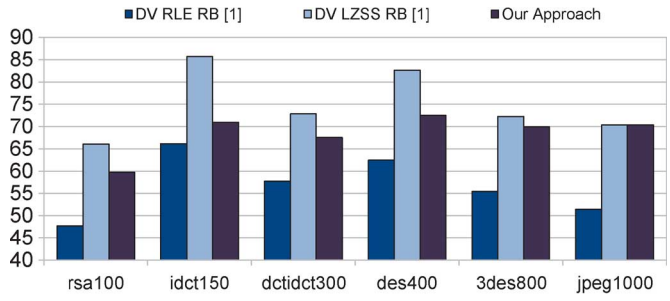


Fig. 9. Comparison with difference vector compression.

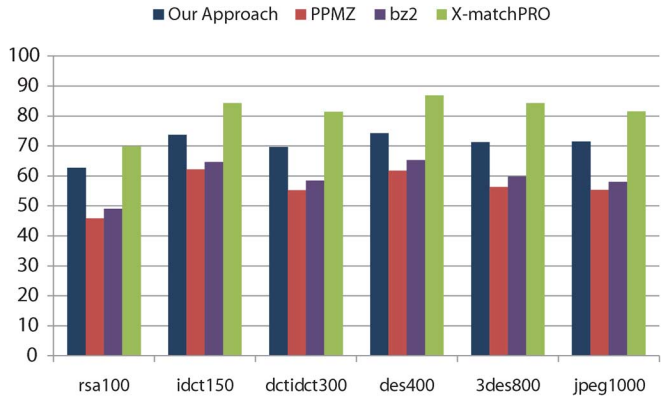


Fig. 10. Comparison with other compression techniques.

have much more random bits that cannot be effectively represented by RLE. BMC, on the other hand, behaves in an opposite way on the two benchmarks. It is designed to encode random bit changes but not suitable for long repetitive patterns. Our approach takes the advantage of both RLE and BMC by adaptively switching between them. As a result, our approach has better compression ratio over all the benchmarks as shown in Fig. 8. We achieve 15% improvement in compression ratio over LZSS and 14% over BMC on an average.

Next, we compare our technique with difference vector approach [1]. The difference vector is encoded using Huffman based RLE with frame readback (DV RLE RB), and LZSS with frame readback (DV LZSS RB). Fig. 9 shows the results. As expected, the Huffman based method achieves the best compression (10% to 15% better than our approach) using shorter variable length encodings. Based on our experiments and the study conducted in [4], such encoding requires complex and large hardware to handle variable-length Huffman codes and has slower operating speed. Thus Huffman-based decoder despite of its good compression ratio is not suitable for real-time decompression. Compared with DV LZSS RB approach, which is suitable for practical implementation, our method has better (10% to 15%) compression performance.

We also compare our approach with general compression algorithm PPMZ,[4] bz2,[5] and X-Match PRO [18]. Fig. 10 shows the results. Our approach outperforms X-Match PRO. Since software compressors like PPMZ and bz2 have complex compression algorithm and almost unlimited resources, it is excepted to generate near optimal results. Interestingly, our approach can achieve comparable results (within 10%–15%).

B. Decompression Efficiency

We measured the decompression efficiency using the time required to reconfigure a compressed bitstream, the resource usage and maximum operating frequency of the decompression engine. The reconfiguration time is calculated using the product of number of cycles required to decode the compressed bitstream and operating clock speed. We have synthesized decompression units for variable-length bitmask-based compression, difference vector-based compression (DV RLE RB), LZSS (8 bit symbols[6]), and our proposed approach on Xilinx Virtex II family XC2v40 device FG356 package using ISE 9.2.04i to measure the decompression efficiency. The results are given in Table I.

TABLE I
OPERATING SPEED AND LOOKUP TABLE USAGE OF DECODERS

Type	Speed (MHz)	Slice Usage
Bitmask decoder [5]	130	250
LZSS [4]	198	45
Our Approach	195	130

We observed that our approach can operate at a much higher frequency and occupies only 60% area compared to original bitmask-based decompression engine. Since our approach has the identical bitmask decoding circuit of the original one, the improvement is due to our ABLSA as we expected. Compared with LZSS, our approach achieves almost the same operating speed as that of LZSS-based accelerator. Although our implementation require more area than LZSS, we have achieved 15%–20% better compression which means we can decompress more configuration information during the same amount of time.

VI. CONCLUSION

The existing compression algorithms either provide good compression with slow decompression or fast decompression at the cost of compression efficiency. In this paper, we proposed a decoding-aware compression technique that tries to obtain both best possible compression and fast decompression performance. The proposed compression technique analyzes the effect of parameters on compression ratio and chooses the optimal ones automatically. We also exploit run length encoding of consecutive repetitive patterns efficiently combined with bitmask-based compression to further improve both compression ratio and decompression efficiency. To reduce the hardware overhead during decompression, we proposed a smart placement algorithm which enables the compressed variable-length coding bitstream to be stored and buffered in the form of multiple fixed-length coding bitstreams. Since the buffering circuitry for fixed-length coding streams can be implemented efficiently, the area and configuration delay of our decompression engine are reduced significantly. Our experimental results demonstrated that our technique improves the compression ratio by 10% to 15% while the decompression engine is capable of operating at 200 MHz in Virtex II FPGAs. The configuration time is reduced by 15% to 20% compared to the best known decompression accelerator [4].

[4][Online]. Available: http://www.cbloom.com/src/ppmz.html

[5][Online]. Available: http://www.bzip.org/

[6]The decompressors usually emit an 8-bit symbol per clock cycle, but decompression hardware may work with different datapath widths or even multicycle paths.

Currently, our placement technique is designed for bitmask-based compression. In the future, we plan to investigate more placement algorithms that are compatible with other compression techniques such as Huffman coding and Arithmetic coding. We also plan to use our technique in other bitstream related applications like manufacturing test data compression.

REFERENCES

[1] 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.

[2] S. Hauck and W. D. Wilson, "Runlength compression techniques for FPGA configurations," in *Proc. IEEE Symp. Field-Program. Custom Comput. Mach.*, 1999, pp. 286–287.

[3] A. Dandalis and V. K. Prasanna, "Configuration compression for FPGA-based embedded systems," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 13, no. 12, pp. 1394–1398, Dec. 2005.

[4] D. Koch, C. Beckhoff, and J. Teich, "Bitstream decompression for high speed FPGA configuration from slow memories," in *Proc. Int. Conf. Field-Program. Technol.*, 2007, pp. 161–168.

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

[6] S. Hauck, Z. Li, and E. Schwabe, "Configuration compression for the Xilinx XC6200 FPGA," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 18, no. 8, pp. 1107–1113, Aug. 1999.

[7] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[8] A. Moffat, R. Neal, and I. H. Witten, "Arithmetic coding revisited," in *Proc. Data Compression Conf.*, 1995, pp. 202–211.

[9] A. Khu, "Xilinx FPGA configuration data compression and decompression," WP152 ed. Xilinx, San Jose, CA, 2001.

[10] M. Huebner, M. Ullmann, F. Weissel, and J. Becker, "Real-time configuration code decompression for dynamic FPGA self-reconfiguration," in *Proc. Int. Parallel Distrib. Process. Symp.*, 2004, pp. 138–143.

[11] R. Stefan and S. Cotofana, "Bitstream compression techniques for Virtex 4 FPGAs," in *Proc. Int. Conf. Field Program. Logic Appl.*, 2008, pp. 323–328.

[12] 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.

[13] X. Qin and P. Mishra, "A universal placement technique of compressed instructions for efficient parallel decompression," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 28, no. 8, pp. 1224–1236, Aug. 2009.

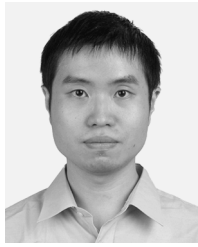
[14] Y. Xie, W. Wolf, and H. Lekatsas, "Code compression for VLIW processors using variable-to-fixed coding," in *Proc. Int. Symp. Syst. Synth.*, 2002, vol. 2, no. 04, pp. 138–143.

[15] L. Li, K. Chakrabarty, and N. A. Touba, "Test data compression using dictionaries with selective entries and fixed-length indices," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 8, no. 4, pp. 470–490, 2003.

[16] *"Virtex-II Pro User Guide,"* Xilinx Inc., San Jose, CA, Nov. 2007.

[17] *"Virtex-4 FPGA User Guide,"* Xilinx Inc., San Jose, CA, Dec. 2008.

[18] J. L. Nunez, C. Feregrino, S. Jones, and S. Bateman, "X-MatchPRO: A ProASIC-based 200 Mbytes/s full-duplex lossless data compressor," in *Proc. Int. Conf. Field-Program. Logic Appl.*, 2001, pp. 613–617.



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

His research interests include the areas of code compression, model checking, and system verification.

Chetan Murthy received the B.E. degree with the Department of Information Science and Engineering, People's Education Society Institute of Technology, Visvesraiah Technological University, India, in 2004, and the M.S. degree from the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, in 2008.

He then joined Huawei Technologies India Private Ltd., Bangalore, India. Since Spring 2009, he has been working as a Packet Forwarding Engineer with Juniper Networks, Inc., Sunnyvale, CA.

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

He is currently an Assistant Professor with the Department of Computer and Information Science and Engineering, University of Florida, Gainesville. His research interests include design automation of embedded systems, reconfigurable architectures, and functional verification. Dr. Mishra currently serves as Program Chair of IEEE HLDVT 2009, Information Director of ACM TODAES, and as a Program/Organizing Committee Member of several ACM and IEEE conferences.

Dr. Mishra was a recipient of various awards including an NSF Career Award in 2008, the EDAA Outstanding Dissertation Award in 2005, and the CODES+ISSS Best Paper Award in 2003.