Lossless Compression Decoders for Bitstreams and Software Binaries Based on High-Level Synthesis

Jian Yan, Junqi Yuan, Philip H. W. Leong, *Senior Member, IEEE*,
Wayne Luk, *Fellow, IEEE*, and Lingli Wang, *Member, IEEE*

***Abstract*— As the density of field-programmable gate arrays continues to increase, the size of configuration bitstreams grows accordingly. Compression techniques can reduce memory size and save external memory bandwidth. To accelerate the configuration process and reduce the software startup time, four open-source lossless compression decoders developed using high-level synthesis techniques are presented. Moreover, in order to balance the objectives of compression ratio, decompression throughput, and hardware resource overhead, various improvements and optimizations are proposed. Full bitstreams and software binaries have been collected as a benchmark, and 33 partial bitstreams have also been developed and integrated into the benchmark. Evaluations of the synthesizable compression decoders are demonstrated on a Xilinx ZC706 board, showing higher decompression throughput than those of the existing lossless compression decoders using our benchmark. The proposed decoders can reduce software startup time by up to 31.23% in embedded systems and 69.83% reduction of reconfiguration time for partial reconfigurable systems.**

***Index Terms*— Compression decoder, field programmable gate array (FPGA) bitstream, high-level synthesis (HLS), lossless compression, software binary.**

I. Introduction

DURING the last decade, the progress in silicon technology has led to an enormous growth of resources on current static random access memory (SRAM)-based field-programmable gate arrays (FPGAs). As the amount of resources increases, the size of configuration memory needed to store the bitstreams grows accordingly. Bitstreams for high-end products from the leading FPGA vendors have broken through the 300 megabits barrier [1], [2]. Storing the bitstreams in FPGA-based systems becomes a critical problem since it needs large external memory that could increase the overall system cost [3]–[7]. In high-radiation environments, continuing growth of bitstreams has become a dramatic problem as nonvolatile memory required to store the bitstreams must be radiation hardened. Such memory has lower density and is much more expensive than conventional memory, resulting in very high system costs.

In addition to the hardware aspects, there is also an increase in the size of software binaries or executable programs that are executed in FPGA-based embedded systems. With the trend toward more complex software systems with respect to code size and the number of processors, the capacity of the external nonvolatile memory needed to store the software binaries grows accordingly. Furthermore, the required throughput for reading software binaries needs to improve in order to meet the limited startup time of some systems [8].

To solve the above problems, bitstream and software binary compression techniques are developed to reduce the external memory usage and the required external memory bandwidth. There have been several approaches investigating the effectiveness of compressing bitstreams or software binaries. However, there are still three limitations of existing methods. First, there is no prior work that simultaneously addresses compression ratio, hardware resource overhead, and decompression throughput. Some compression techniques with good compression ratio and few hardware resources have poor decompression throughput and hence are not competitive. Second, benchmarks are not available for full bitstreams, partial bitstreams, and software binaries, and do not represent the statistical characteristics for a wide range of applications. Third, there are substantial differences between implementation results and simulation results.

To address these challenges, four lossless compression decoders developed using high-level synthesis (HLS) techniques are presented. At the decompression level, various improvements and optimizations are applied to the compression decoders in order to balance the objectives of compression ratio, decompression throughput, and hardware resource overhead simultaneously. At the benchmark level, 28 full bitstreams and 25 software binaries have been collected. Thirty-three partial bitstreams have been developed and integrated into the benchmark. Moreover, since no specific bitstream organizations and software binary data structures are required, these decoders are applicable to other modern FPGA devices and software binaries. Implementations of the proposed compression decoders can run at 200 MHz. Evaluations of the synthesizable compression decoders are demonstrated on a Xilinx ZC706 board. The decompression throughput of the proposed compression decoders is superior to that of the existing lossless compression decoders on the benchmark by making use of different kinds of optimizations.

Manuscript received October 28, 2016; revised March 5, 2017; accepted May 26, 2017. Date of publication June 22, 2017; date of current version September 25, 2017. This work was supported in part by the National Natural Science Foundation of China under Grant 61131001; in part by the Australian Research Council through Linkage Project under Grant LP130101034; in part by the European Union Horizon 2020 Research and Innovation Programme under Grant 671653; and in part by U.K. EPSRC (EP/L00058X/1, EP/L016796/1, EP/N031768/1, and EP/P010040/1). *(Corresponding author: Lingli Wang.)*

J. Yan, J. Yuan, and L. Wang are with the State Key Laboratory of ASIC and System, Fudan University, Shanghai 201203, China (e-mail: 13110720061@fudan.edu.cn; 15110720075@fudan.edu.cn; llwang@fudan.edu.cn).

P. H. W. Leong is with the School of Electrical and Information Engineering, The University of Sydney, Sydney, NSW 2006, Australia (e-mail: philip.leong@sydney.edu.au).

W. Luk is with the Department of Computing, Imperial College, London SW7 2BZ, U.K. (e-mail: wl@doc.ic.ac.uk).

Digital Object Identifier 10.1109/TVLSI.2017.2713527

In summary, the following contributions are made in this paper.

1) Four open-source lossless compression decoders are developed using HLS. Various improvements and optimizations are applied to these decoders in order to balance the objectives of compression ratio, decompression throughput, and hardware resource overhead. Implementations of the synthesizable lossless compression decoders can run at 200 MHz.
2) A benchmark including full bitstreams, partial bitstreams, and software binaries is provided. Comparison results between the proposed compression decoders and the existing compression decoders on the benchmark are presented in respect of maximum frequency, as well as compression ratio, decompression throughput, and hardware resource overhead.
3) Evaluations of the synthesizable compression decoders are demonstrated on a Xilinx ZC706 board, showing higher decompression throughput than those of the existing compression decoders on the benchmark. Moreover, the proposed method can reduce software startup time by up to 31.23% in embedded systems and 69.83% reduction of reconfiguration time for partial reconfigurable systems.

The rest of this paper is organized as follows. Section II discusses related work. Section III gives overall design considerations. Section IV presents design and implementation of lossless compression decoders in detail. Experimental results and analysis are presented in Section V. Finally, Section VI concludes this paper.

II. Related Work

Existing compression techniques for bitstreams and software binaries can be classified into two categories based on whether they are device dependent during decompression.

A. Device-Dependent Approaches

Some methods require special hardware features during decompression, such as wildcard registers or configuration mechanism, which are provided only in certain FPGAs.

Wildcard registers allow configuration memory within the same row or column to be written simultaneously. Some approaches that take advantage of the characteristics targeting Xilinx XC6200 FPGAs are presented in [9] and [10]. By using these registers, faster configuration could be achieved. In addition, a subsequent approach [11] from the same authors takes advantage of these registers based on run-length encoding (RLE) for the same family of FPGAs. The use of "don't cares" for bitstream compression has been proposed in [12] and [13]. However, the method requires specific information regarding the bitstream format and the internal structure of the FPGA. This information is confidential in modern FPGAs, preventing us from exploiting such techniques.

Some studies [13]–[16] investigate the ability to compress bitstreams using the configuration mechanism. These studies show that bitstreams can be compressed efficiently by utilizing interframe or intraframe regularity. In addition, some researchers [13], [14] use frame reordering and run-time frame readback to achieve better redundancy, which are combined with RLE, Huffman encoding, Lempel–Ziv–Storer–Szymanski (LZSS) encoding, or computing the XOR difference between frames. These complex compression technologies can produce excellent compression results. However, not only do these methods require the knowledge of the bitstream format and the internal structure of the FPGA, but also they do not address the problem of decompression throughput and decompression hardware resource overhead.

B. Device-Independent Approaches

Altera incorporates a hardware compression decoder in products, such as Stratix II FPGAs [17]. This decompression feature allows FPGAs to receive a compressed configuration bitstream and decompress it at run time, reducing storage requirements and configuration time. The Xilinx bitstream generation tool (BitGen) includes an option called "-g compress," which uses multiple-frame write sequences to minimize the size of bitstreams. This option is appropriate for the compression of full and partial bitstreams. Both Xilinx and Altera have built-in hardware compression decoders in the external configuration devices, such as the System ACE MPM [18], Platform Flash PROM [19], and Enhanced Configuration [20]. Using these devices, designers do not require knowledge of the bitstream format and the internal structure of the FPGA. In fact, they can save the total solution cost including storage memory, board space, configuration speed, and source of supply. However, both Xilinx and Altera omit details on decompression throughput and hardware resource overhead.

There are also other approaches in the field of bitstream compression and decompression, which involve RLE methods, statistical methods, dictionary-based methods, bitmask-based methods, and other variations [3]–[7], [21]–[26]. Since configuration bitstreams are processed as raw data, these proposed techniques are applicable to other SRAM-based FPGA device, and do not depend on specific features of the configuration mechanisms. A modified Lempel–Ziv–Welch (LZW) dictionary-based compression method has been proposed in [3] and [4]. Although the decompression hardware is simple, the memory requirement is high. The work presented in [5] and [6] tends to be more exhaustive in terms of compression techniques. Koch *et al.* [5] and [6] discuss different bitstream compression techniques, and also compare their own compression methods with state-of-the-art software programs like GZIP applied to different bitstreams. But the decompression throughput is not competitive. Qin *et al.* [7] compare their compression techniques with previous methods, and the proposed compression decoder is designed carefully to achieve less hardware resource overhead and higher frequency. Unfortunately, they do not address the decompression throughput. A detailed implementation of LZSS compression decoder has been shown in [21]. The decoder can decompress the compressed partial bitstreams at run-time, but the authors do not report the decompression throughput. The LZSS approach is also used to compress portable partial bitstreams in [22]. Reference [23] shows high compression efficiency of partial bitstreams using the XOR operation and RLE. However, BRAM is used to store partial

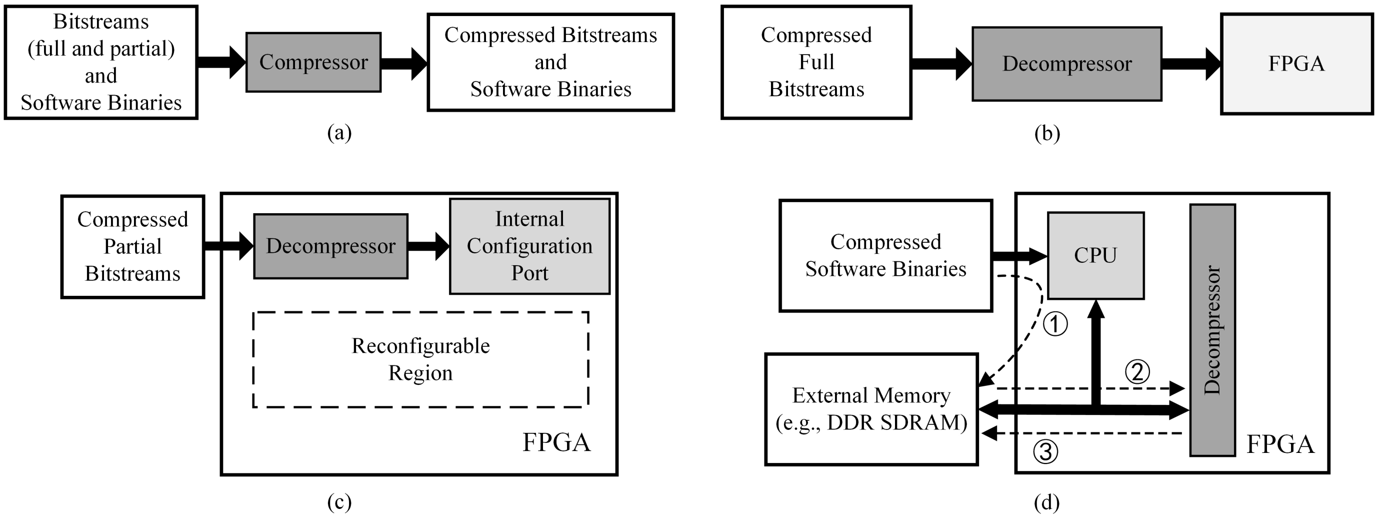


Fig. 1. Process of compression and decompression. (a) Compression stage (compilation time, offline). (b) Decompression stage of full bitstreams (run time, off-chip). (c) Decompression stage of partial bitstreams (run time, on-chip). (d) Decompression stage of software binaries (run time, on-chip).

bitstreams before configuration. Larger partial bitstream files need more BRAMs, and hence, this method is not suitable for large partial bitstream files. A wide range of compression algorithms have been evaluated in [24] based on eight benchmark circuits. Although the proposed compression decoders can run higher than 200 MHz, the decompression throughput is compromised. Duhem *et al.* [25] propose an optimized RLE method to compress partial bitstreams and use the corresponding compression decoder to accelerate configuration process. However, the throughput is low. A compression decoder based on LZ77 and bitmask schemes has been proposed in [26], but the hardware resource overhead is high and the decompression throughput is low. Not only do these methods consider the compression ratio of each algorithm, but also the frequency of compression decoders and hardware resource overhead.

The organization of software binaries is very different from FPGA bitstreams. Some methods are also appropriate for software binaries. The LZSS compression technique is utilized for software binaries minimization in [6]. Bitmask-based compression approaches are proposed to compress software binaries in [27] and [28]. However, the decompression throughput of the proposed decoders in this paper is superior to these lossless compression decoders.

Although compression techniques for bitstreams have been proposed, all the above compression decoders are designed using hand-written register-transfer level (RTL) descriptions. In this paper, four lossless compression decoders are developed using HLS. In order to balance the objectives of compression ratio, decompression throughput, and hardware resource overhead, various improvements and optimizations are applied to these decoders. Furthermore, an extended benchmark is provided, including full bitstreams, partial bitstreams, and software binaries.

III. Design Considerations

HLS is introduced in general, and then the process of compression and decompression is discussed.

A. High-Level Synthesis

High-level synthesis tools perform automated translation of high-level language (e.g., C, C++, and SystemC) inputs to RTL implementations. HLS tools perform scheduling of operations to finite state machine states and binding of operations to functional units. They also generate I/O interfaces to connect with memory or other interface protocols. HLS tools include pragmas to guide optimization of area and performance. Efficient use of pragmas, together with code reorganizations to make pragmas effective, is the key design and the optimization technique for HLS. In a word, HLS bridges hardware and software domains, which can improve both productivities for hardware designers and system performance for software designers [29], [30].

In this paper, Xilinx Vivado HLS is used to develop and verify the compression decoders at the C-level. Multiple implementations based on the source code are created, using optimization directives and exploring the design space, which increase the possibility of finding an optimal implementation.

B. Process of Compression and Decompression

The goal of compression is to minimize the size of bitstreams and software binaries. The compression of bitstreams and software binaries is similar to that of data compression, taking advantage of regularity and repetitions existing in the data stream. The process of compression and decompression is shown in Fig. 1. The original software binaries or full and partial bitstreams are compressed at the compilation time as shown in Fig. 1(a). The compressed full bitstream is transferred to the off-chip decompressor, and then the decompressed full bitstream is transmitted to the FPGA configuration memory through the configuration interface as shown in Fig. 1(b). For partial reconfiguration, the compressed partial bitstream stored in external memory is transferred to the on-chip decompressor, which is directly connected to the internal configuration port. Then, the function implemented

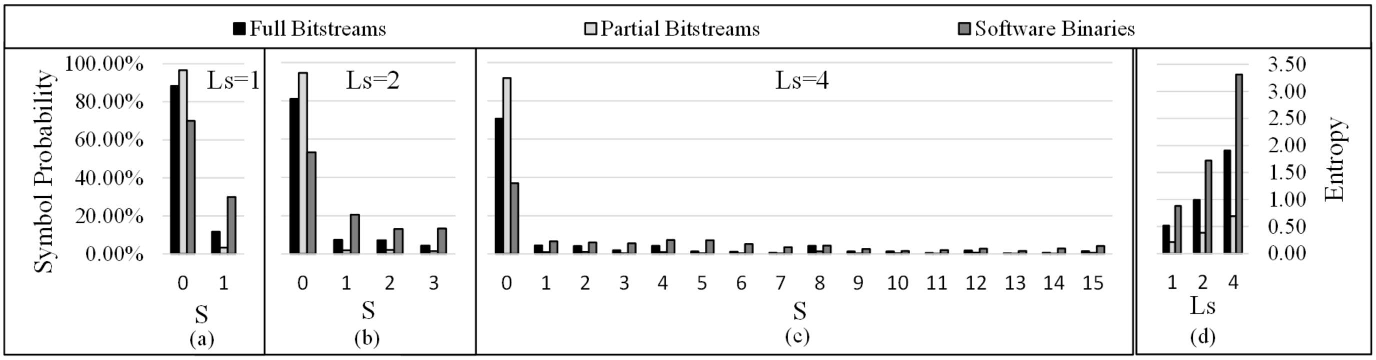


Fig. 2. Characteristics of the benchmark. (a)–(c) Symbol probability of symbol (S) value. (d) Entropy of Ls.

in the reconfigurable region is modified by the decompressed partial bitstream as shown in Fig. 1(c). In embedded systems, the CPU reads the compressed software binary stored in the external nonvolatile memory and transfers it to the external memory ①, such as double data rate synchronous dynamic random access memory (DDR SDRAM). Then, the compressed software binary is transmitted to the on-chip decompressor ②. Finally, the decompressed software binary is return to the external memory ③ as shown in Fig. 1(d).

There are two problems which must be resolved for the process of compression and decompression. First, an efficient compression encoder must be provided. The more efficient the compression encoder is, the more memory it can save. Second, since the decompression is performed at run-time, both decompression throughput and hardware resource overhead should be carefully considered. The resource overhead of the compression decoder should be as small as possible. The throughput of the decoder has to be higher than that of the configuration interface; otherwise, the compression decoder will affect the configuration rates. In addition, any information loss in bitstreams or software binaries may generate undesired outputs, or even worse it may damage devices. Any lossless compression technique has to satisfy the following condition: the outputs of the decompressor must be the same as the original inputs of the compressor. Lossless compression techniques are well-studied with a lot of efficient methods. In this paper, four lossless compression techniques are selected, based on RLE [31], LZ77 [32], LZSS [33], and LZG [34]. The details of designs and implementations are presented in the following section.

IV. Design and Implementation of Lossless Compression Decoders

In this section, the design and implementation metrics are first introduced. Then, analysis of the benchmark and selections of reference codes is presented. Finally, design flow and optimization methodology are discussed in more detail.

A. Design and Implementation Metrics

In this paper, compression techniques are compared and analyzed using the following metrics.

1) The compression ratio is widely used as a metric in [6] and [7]. It is defined as the ratio between the compressed data size and the original data size

$$\text{Compression Ratio} = \frac{\text{Compressed Data Size}}{\text{Original Data Size}}. \quad (1)$$

A smaller compression ratio means a better compression technique and saving more memory of external nonvolatile memory.

2) The decompression throughput is defined as the ratio between the decompressed data size (the same as the original data size) and the decompression time

$$\text{Decompression Throughput} = \frac{\text{Decompressed Data Size}}{\text{Decompression Time}}. \quad (2)$$

In order to speed up the hardware configuration and reduce software startup time, higher decompression throughput is required.

3) The decompression efficiency is the ratio of the decompression throughput to the frequency and the data width

$$\text{Decompression Efficiency} = \frac{\text{Decompression Throughput}}{\text{Frequency} \cdot \text{Data Width}}. \quad (3)$$

To measure the efficiency of the decompression technique, the decompression efficiency is used as a metric.

4) Hardware resource overhead. Since decompression is performed at run-time, the hardware resource overhead of the compression decoder should be as small as possible.

Considering the above metrics, the required compression technology should have better compression ratio, higher decompression throughput, higher decompression efficiency, and less hardware resource overhead.

B. Analysis of the Benchmark

The following characteristics of the benchmark are shown in Fig. 2.

1) The probability of symbol (S) “0” is much higher than that of other symbols, especially for the partial bitstreams as shown in Fig. 2(a)–(c).

2) The probability of each symbol decreases with the increase of the symbol length (Ls) as shown in Fig. 2(a)–(c).
3) The average entropy of the 33 partial bitstreams is the lowest. The average entropy of the 25 software binaries is the highest as shown in Fig. 2(d).

Only one symbol consistently exhibits extremely high frequency for each category of the benchmark. There are more repetitive patterns in the 33 partial bitstreams and more random symbols in the 25 software binaries. These characteristics imply that partial bitstreams have a better compression ratio than full bitstreams and software binaries. Details of the compression ratio and the benchmark are presented in Section V.

C. Reference Codes Selection

As discussed in Section III, the compression technologies for bitstreams and software binaries have to be lossless. In the selection of reference codes, although only software compression decoders are the inputs to HLS, it is critical that software compression encoders are used consistently in order to generate the input data for the compression decoders. In addition, in order to perform HLS, the reference codes cover software compression decoders. Furthermore, the reference codes should not include embedded assembly or platform-dependent optimizations, which are not supported by Xilinx Vivado HLS.

To summarize, the following characteristics should be considered in the selection of reference codes.

1) Software implementations of both lossless compression and decompression methods are available.
2) The software compression decoders are able to be extracted from the reference codes.
3) There is no embedded assembly or platform-dependent optimizations.

Based on the above characteristics, RLE [31], LZ77 [32], LZSS [33], and LZG [34] are selected.

In addition to the above considerations, RLE is simple and very useful for data that contain many repeating patterns. Three similar dictionary-based methods are selected for a good compression ratio and simple decompression schemes. LZ77 and LZSS are widely used for bitstream compression [5], [6], [13], [18], [21], [24], [26]. Both methods can achieve a good compression ratio, but the decompression throughput is not competitive. Compared with LZ77 and LZSS, the decompression method of LZG is simpler and faster with that of comparable compression ratio. Experimental results including the benchmark are presented in Section V.

1) RLE [31]: If a data item *D* occurs *N* consecutive times in the input stream, replace the *N* occurrences with the single pair *ND*. Actually, there is also a flag item *F* which is required during the RLE decompression. Thus, the compressed data stream is *FND*. RLE decompression is also straightforward. When a flag item *F* is read, the repetition count *N* and the actual data *D* are immediately read, and the data item *D* is written *N* times on the output stream.

2) LZ77 [32]: LZ77 is a dictionary-based text compression scheme. The scheme works by defining a sliding window or a fixed-size dictionary to hold data from an input stream, and then referring to the sliding window when compressing the remainder of the input stream. If a pattern in the input stream is already in the sliding window, this pattern is replaced with a length–distance pair. As compression progresses, the sliding window is updated by shifting in more data from the input stream, subsequently forcing earlier entries out. Decompression is the inverse of the compression process. The same sliding window is used to hold uncompressed data. When a length–distance pair to the sliding window is encountered, the decompressor simply copies the specified number of data from the dictionary, shifts these data into the same sliding window, and then continues processing the rest of the compressed input stream.

3) LZSS [33]: LZSS is an efficient variant of LZ77. It holds the look-ahead buffer in a circular queue and holds the search buffer (the dictionary) in a binary search tree. Since the buffer size is 4096 B, the position can be encoded in 12 b. The scheme also represents the match length in 4 b, and thus, the position–length pair is just two bytes long. If the longest match is no more than two characters, then the scheme sends just one character without encoding, and continues the process with the next symbol. It also needs one extra bit each time to tell the decoder whether it is sending a position–length pair or an unencoded character.

4) LZG [34]: Liblzg is a minimal implementation of an LZ77 class compression library. It implements an algorithm that is a variation of the LZ77 algorithm, with the primary focus of providing a very simple and fast decompression method. It contains four unique marker symbols, which are used to separate literal data from the various forms of length–distance pair encodings.

D. Design Flow and Optimization Methodology

Fig. 3 shows a brief overview of the design flow and optimization methodology. The gray parts are the optimization methodology. The functional correctness of the compression decoder extracted from the reference code is verified with a C testbench. After all nonsynthesizable constructs are eliminated or converted for synthesizability, the initial hardware design of the compression decoder is generated by HLS with the default optimization. The initial design prepares for further optimizations. These optimizations will be discussed later in more detail. In order to validate the optimized design, C/RTL cosimulation is performed within Vivado HLS. During C/RTL cosimulation, the same C testbench used in C simulation is reused and the synthesized function is replaced by the RTL design. Once the C/RTL cosimulation is completed, the design is packaged as an IP. Detailed explanations of the design flow are presented in [35]. Optimizations are shown in the following.

1) I/O Optimization: In the C-level design, all input and output operations are performed in negligible time via a function call. In contrast, for an RTL design, these same input and output operations involves transferring data through I/O ports. Although Vivado HLS supports various I/O protocols, AXI4-Stream Interfaces without side channels are used.

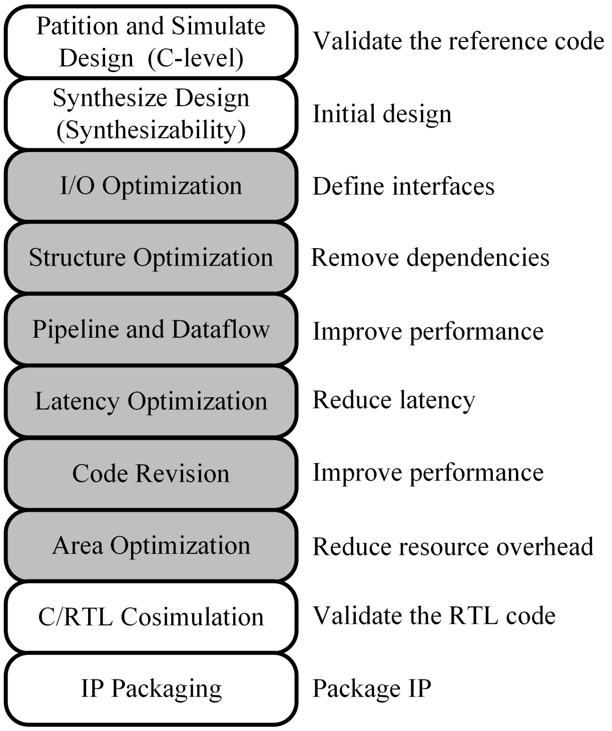
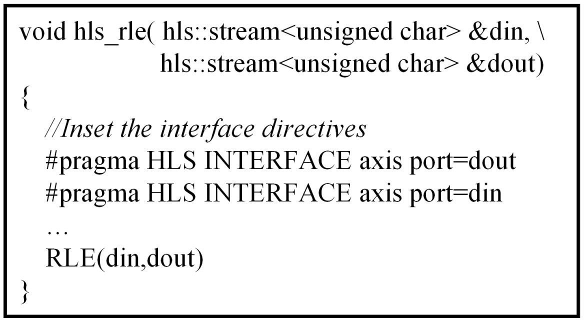


Fig. 3. Design flow and optimization methodology.



```
void hls_rle( hls::stream<unsigned char> &din, \
              hls::stream<unsigned char> &dout)
{
   //Inset the interface directives
   #pragma HLS INTERFACE axis port=dout
   #pragma HLS INTERFACE axis port=din
   …
   RLE(din,dout)
}
```

Fig. 4. I/O optimization of the RLE decoder.

These interfaces can be applied to any input or output arguments, and can achieve high performance. In addition, since the compression decoder accesses data in a streaming manner, these interfaces will not be the bottleneck. Furthermore, AXI4-Stream Interfaces are industrial standard interfaces and easy to use during the system design. Fig. 4 shows the I/O optimization of the RLE decoder. Other decoders are nearly the same. Fig. 5 shows the corresponding interfaces of the RLE decoder. The width of the I/O can be modified easily by changing the arguments.

2) Structure Optimization: C/C++ code can contain dependencies that prevent a function or a loop from being pipelined. In these cases, code improvements are needed to remove the dependencies. A shift register function is synthesized as a slide window of the dictionary-based compression method. Fig. 6 shows the data dependence in the L1 loop. There is the dependence between two shifting operations. Shifter writing Sreg.shift(tmp,0) cannot be executed until shifter reading tmp = Sreg.read(offset-1) has been executed. After structure optimization, Fig. 7 shows no data dependence in the L2 and the L3 loops. Even though data outputting dout.write(tmp), shifter reading, and writing tmp1 = Sreg.shift(tmp,offset-2), as well as assignment tmp = tmp1 appear serially in the code, these operations can be pipelined. Therefore, the data dependence between two operations can be removed by structure optimization. Although this optimization increases the hardware resource overhead, it can achieve higher performance.

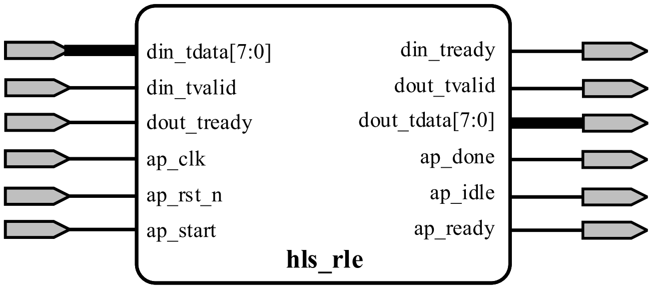
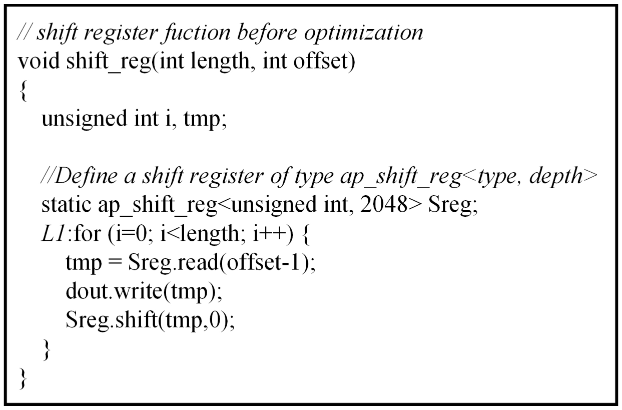


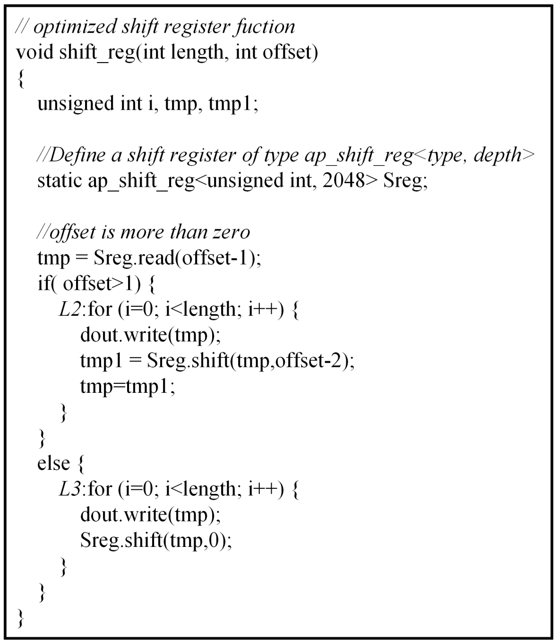
Fig. 5. Interfaces of the RLE decoder.



```
// shift register fuction before optimization
void shift_reg(int length, int offset)
{
   unsigned int i, tmp;

   //Define a shift register of type ap_shift_reg<type, depth>
   static ap_shift_reg<unsigned int, 2048> Sreg;
   L1:for (i=0; i<length; i++) {
      tmp = Sreg.read(offset-1);
      dout.write(tmp);
      Sreg.shift(tmp,0);
   }
}
```

Fig. 6. Shift register function before optimization.



```
// optimized shift register fuction
void shift_reg(int length, int offset)
{
   unsigned int i, tmp, tmp1;

   //Define a shift register of type ap_shift_reg<type, depth>
   static ap_shift_reg<unsigned int, 2048> Sreg;

   //offset is more than zero
   tmp = Sreg.read(offset-1);
   if( offset>1) {
      L2:for (i=0; i<length; i++) {
         dout.write(tmp);
         tmp1 = Sreg.shift(tmp,offset-2);
         tmp=tmp1;
      }
   }
   else {
      L3:for (i=0; i<length; i++) {
         dout.write(tmp);
         Sreg.shift(tmp,0);
      }
   }
}
```

Fig. 7. Structure optimization of the shift register function.

```
//bad shift register fuction
void shift_reg(int length, int offset)
{
   unsigned int i, tmp;

   //Define a shift register of type ap_shift_reg<type, depth>
   static ap_shift_reg<unsigned int, 2048> Sreg;

   //L1 loops can be pipelined
   #pragma HLS PIPELINE
   L1:for (i=0; i<=0x7FFFFFFF; i++) {
      if(i<length) {
         tmp = Sreg.read(offset-1);
         dout.write(tmp);
         Sreg.shift(tmp,0);
      }
   }
}
```

Fig. 8. Pipeline optimization of the shift register function.

3) Pipeline and Dataflow Optimizations: In order to improve the performance of the compression decoders, more and more concurrent and parallel operations are required. The pipeline optimization can be applied to functions and loops. The dataflow optimization can be applied at the task level that contains the functions and loops.

There is a tradeoff between area and performance when loops are pipelined. L1 loops can be pipelined after code conversion shown in Fig. 8. But it can be seen that bad pipeline optimization may result in large hardware resource overhead and long latency. Thus, it might make no sense to improve the performance.

On the other hand, applying dataflow optimization can pipeline sequential tasks and improve performance. Compared with the RLE decoder shown in Fig. 4, we add another two tasks shown in Fig. 9(a). The three tasks run serially and require six clock cycles without dataflow optimization shown in Fig. 9(a). During dataflow optimization, Vivado HLS schedules each task to execute as soon as possible and allows executions of the three tasks to overlap with only four clock cycles, increasing the overall throughput of the design shown in Fig. 9(b). Because the three tasks share the same interfaces, they cannot be executed concurrently within two clock cycles.

4) Latency Optimization: There is no need to apply latency optimization when the loops or function are pipelined. However, if the loops or functions are not pipelined shown in Fig. 7, the overall throughput will be limited by the latency, because the task does not start reading until the task has completed. In this case, the maximum latency is needed to be constrained.

5) Code Revision: In order to improve the decompression throughput further, the reference code revision is required. For example, when a data item occurs three consecutive times, the RLE method compresses the input stream with the three characteristics discussed in Section IV-C. The corresponding RLE decompression process is shown in Fig. 10(a). The decoder reads three times (flag item, repetition count, and actual data), and then, the actual data are written three times on the output stream. The RLE encoder does not compress the input stream until the data item occurs more than three consecutive times after the code revision. Thus, this optimization does not affect the compression ratio. The RLE decoder alternates reading input stream and writing output stream presented in Fig. 10(b). Combining with the dataflow optimization, the decompression throughput is improved, whereas the compression ratio is not affected by the code revision shown in Fig. 10(c).

6) Area Optimization: Different optimizations have different effects on the hardware resource overhead. In order to increase the throughput of the design, part of the codes is pipelined. Thus, this optimization increases the hardware resource overhead. Width improvement of I/O interfaces is also applied to these decoders. Wider I/O interfaces mean higher performance and larger hardware resource overhead. In order to achieve higher performance and less hardware resource overhead, arbitrary precision data types are used and the widths of variables can be arbitrary. In standard C/C++ data types, the widths of variables are 8, 16, 32, and so on. Thus, using the standard C/C++ data types results in unnecessary hardware resource overhead. Furthermore, it has a negative effect on the timing of the overall design. Compared with the shift register function shown in Fig. 6, the area optimization of the shift register function is illustrated in Fig. 11. As the depth of the shifter is 2048, the 11-b unsigned integer data type is used to reduce hardware resource overhead compared to using the 32-b signed integer data type for the variable offset.

V. Experimental Results and Analysis

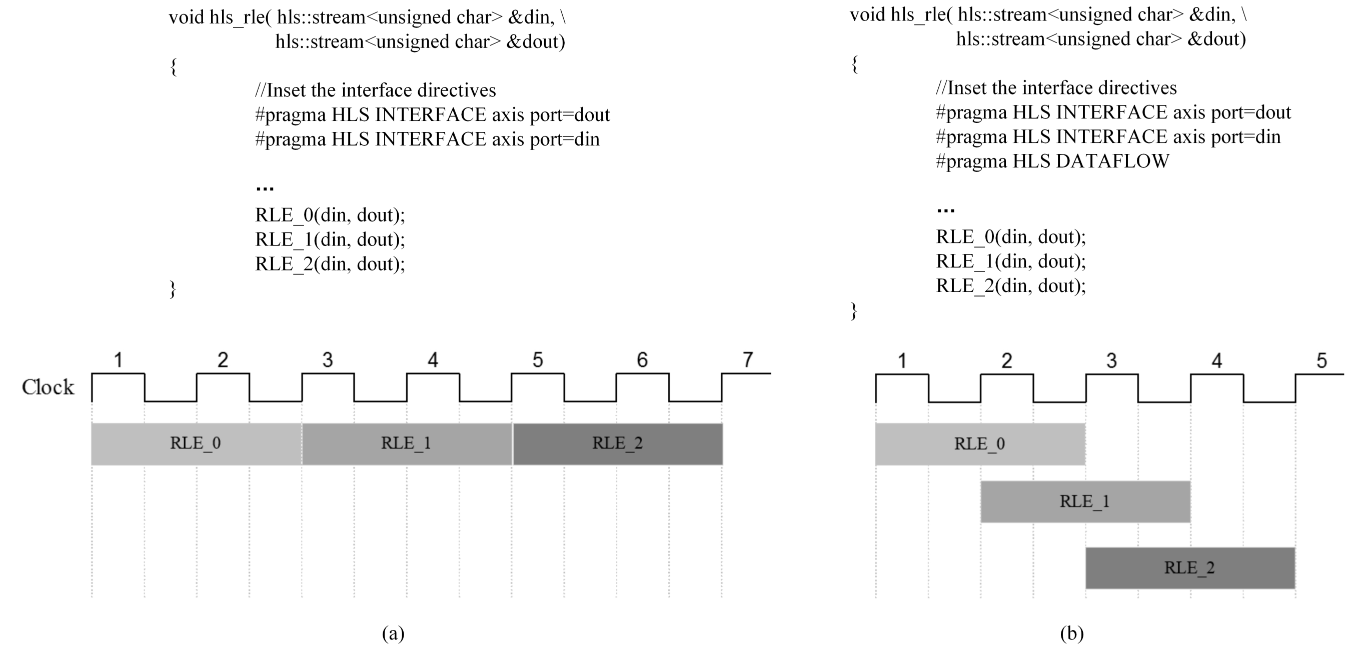
A benchmark including full bitstreams, partial bitstreams, and software binaries is presented. Then, the general evaluation system is introduced with four metrics. Finally, evaluations of the embedded system and the partial reconfigurable system are discussed.

A. Benchmark

In order to investigate different compression decoders, a suitable benchmark is required. The benchmark should include full bitstreams, partial bitstreams, and software binaries.

1) Full Bitstreams: Bitstreams that represent the statistical characteristics for a wide range of applications are collected. These applications are cryptography applications [Data Encryption Standard (DES) and RC5], signal processing applications (fast Fourier transform and finite-impulse response), system applications (system-on-chip), and communication applications (Xbar and Net). Both Altera and Xilinx bitstreams are provided. These full bitstreams have a high logic utilization of the available resources. More details can be found in [5] and [36].

2) Partial Bitstreams: Table I summarizes the partial bitstream benchmark. Table I lists the source of the modules, the register utilization, the LUT utilization, the slice distribution, and the bitstream size for different modules. These implementations are generated using the default configuration of place and route methodology, with no attempt to increase



```
void hls_rle( hls::stream<unsigned char> &din, \
            hls::stream<unsigned char> &dout)
{
        //Inset the interface directives
        #pragma HLS INTERFACE axis port=dout
        #pragma HLS INTERFACE axis port=din

        ...
        RLE_0(din, dout);
        RLE_1(din, dout);
        RLE_2(din, dout);
}
```

```
void hls_rle( hls::stream<unsigned char> &din, \
            hls::stream<unsigned char> &dout)
{
        //Inset the interface directives
        #pragma HLS INTERFACE axis port=dout
        #pragma HLS INTERFACE axis port=din
        #pragma HLS DATAFLOW

        ...
        RLE_0(din, dout);
        RLE_1(din, dout);
        RLE_2(din, dout);
}
```

Fig. 9. Dataflow optimization of the RLE decoder. (a) Without dataflow optimization. (b) With dataflow optimization.

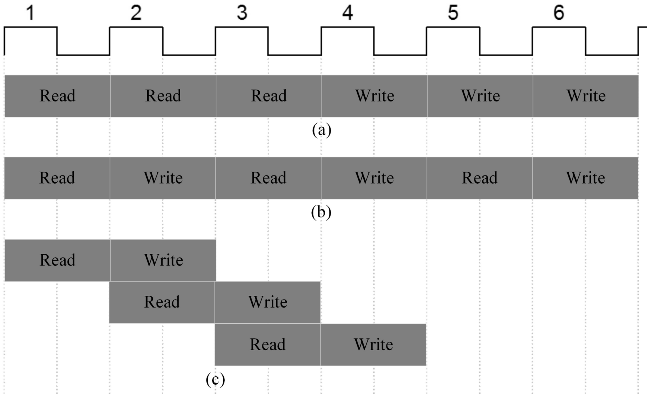


Fig. 10. Code revision of the RLE decoder. (a) RLE decompression process before code revision. (b) RLE decompression process after code revision. (c) Code revision with dataflow optimization.

```
#include "ap_int.h"

// shift register fuction before optimization
void shift_reg(int length, ap_uint<11> offset)
{
   unsigned int i, tmp;

   //Define a shift register of type ap_shift_reg<type, depth>
   static ap_shift_reg<unsigned int, 2048> Sreg;
   L1:for (i=0; i<length; i++) {
      tmp = Sreg.read(offset-1);
      dout.write(tmp);
      Sreg.shift(tmp,0);
   }
}
```

Fig. 11. Area optimization of the shift register function.

the structure regularity for partial bitstreams by manual optimizations. Bitstreams of blank are automatically generated by the bitstream generation tool. Other modules are chosen from different applications.

1) Signal processing applications include a float point adder or subtractor (add/sub), a coordinate rotational digital computer, a Discrete Cosine Transform module [37], and a divider.
2) Cryptography applications contain an Advanced Encryption Standard (AES) module [38], an AES decryption module [38], and a DES module [39]. These modules are all from OpenCores.
3) Communication applications include a cyclic redundancy check (CRC) module [40], a first-input, first-output (FIFO) module, and a Loopback module.

All partial bitstreams are generated by Xilinx Vivado 2015.4 with three different options. "Default" means the default option of bitstream generator. The "Xilinx Compression" option uses multiple-frame write sequences to minimize the size of bitstreams. This is the same as the "-g compress" option in the previous Xilinx BitGen tool. It is device independent and appropriate for both full and partial bitstreams. Whether the slice distribution is high or low, the compressed size of the partial bitstreams is similar. The bitstream generator adds an extra CRC value after each frame by using the "CRC per frame" option. Thus, the size of bitstream is larger than that of the "default" option. In order to evaluate the influences of bitstream generation options on the compression technologies, the three types of partial bitstreams are used.

3) Software Binaries: In order to evaluate the efficiency of compression technologies for software binaries in embedded systems, 25 software binaries are collected. These software binaries are Lightweight IP application examples, and can run on both MicroBlzae and ARM processor-based systems with

TABLE I
PARTIAL BITSTREAM BENCHMARK

Module	source	Slice Utilization		Slice Distribution	Place and Route Methodology	Bitstream Size (Bytes)		
		Register	LUT			Default	Xilinx Compression	CRC per Frame
Add/Sub(fp)	Xilinx	12.45%	14.50%	29.13%	Auto	418,496	141,112	436,444
AES	OpenCores	17.61%	41.50%	66.75%			142,568	
AESI	OpenCores	41.84%	61.56%	98.50%			142,568	
Blank	Xilinx	0.00%	2.19%	5.13%			87,120	
CORDIC	Xilinx	9.00%	17.81%	27.63%			142,748	
CRC	OpenCores	1.56%	4.53%	6.38%			117,356	
DCT	OpenCores	13.08%	47.31%	63.13%			142,568	
DES	OpenCores	6.19%	12.47%	22.63%			132,316	
Div	Xilinx	14.08%	12.44%	34.25%			140,788	
FIFO	Xilinx	2.91%	4.84%	7.00%			116,956	
Loopback	By hand	1.05%	0.16%	2.50%			82,952	

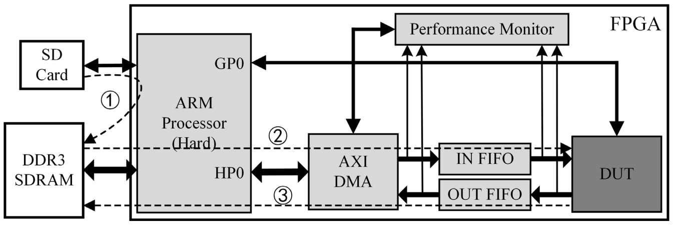


Fig. 12. Architecture of the evaluation system.

different cache size. The size of these software binaries varies from 300 KB to 2 MB. One of these software binaries is the initialization of the hard ARM processor. More detailed information about the software binaries can be found in [41].

B. General Evaluation System

1) Evaluation System Architecture: To evaluate the four metrics of compression techniques, the Xilinx ZC706 board is adopted as the evaluation system. Four compression decoders are developed with Xilinx Vivado HLS 2015.4. Logic synthesis and implementation of each design are performed with Xilinx Vivado 2015.4. Fig. 12 shows the architecture of the evaluation system, where the HP0 port, the AXI direct memory access (DMA) controller, IN FIFO, and OUT FIFO, as well as the Design Under Test (DUT) all run at 200 MHz. The AXI DMA controller, the DUT, and the Performance Monitor are also connected to the GP0 port, which also runs at 200 MHz. Each compression decoder is mapped to the DUT block. The compressed data and golden data are stored in the external SD card before system power-ON. After the system is power-ON, the ARM processor reads the data stored in the SD card and transfers the data to the external DDR3 SDRAM ①. Then, the AXI DMA controller reads the compressed data from the DDR3 SDRAM and transmits it to the compression decoder ②, the decompressed data returns back to the DDR3 SDRAM after decompression ③. Finally, the Performance Monitor reports the result after checking the consistency between the decompressed data and the golden data.

2) Compression Ratio: Fig. 13 shows the compression ratio of the benchmark. The first subgraph shows the average compression ratio of 28 full bitstreams. The second and third subgraphs present the average compression ratio of 33 partial bitstreams and 25 software binaries. The last subgraph demonstrates the average compression ratio of the whole benchmark. Lower compression ratio means saving more memory and being more efficient. Thus, using these compression methods, 23.77%–58.68% (1%–76.23% to 1%–41.32%) memory can be saved for the benchmark. The 8-b compression methods are more efficient than the 32-b compression methods. Because the partial bitstreams have a large amount of repetitive patterns, all the compression methods are applicable to the partial bitstreams and the compression ratio of the partial bitstreams is lower than 25%. Nevertheless, there are more random symbols in the software binaries, making RLE series unsuitable as they almost have no effect. LZG series is more efficient than RLE series, especially for software binaries. Compared with the state-of-the-art compression methods, such as GZIP (version 1.2.4) [42] and 7-zip (version 16.02) [44], the proposed compression technologies are less efficient, but the proposed decompressed methods have higher decompression throughput and less hardware resource overhead.

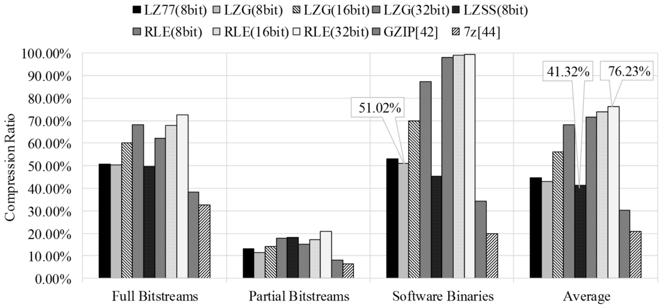


Fig. 13. Compression ratio of the benchmark.

Compression ratios for 28 full bitstreams are presented in Fig. 14. Besides the bitstreams for different applications, we also compare the average ratio of all the bitstreams. Compared with the previous methods [6], the proposed compression technologies LZ77(8 b), LZG(8 b), and LZSS(8 b) achieve better compression ratio on most of applications.

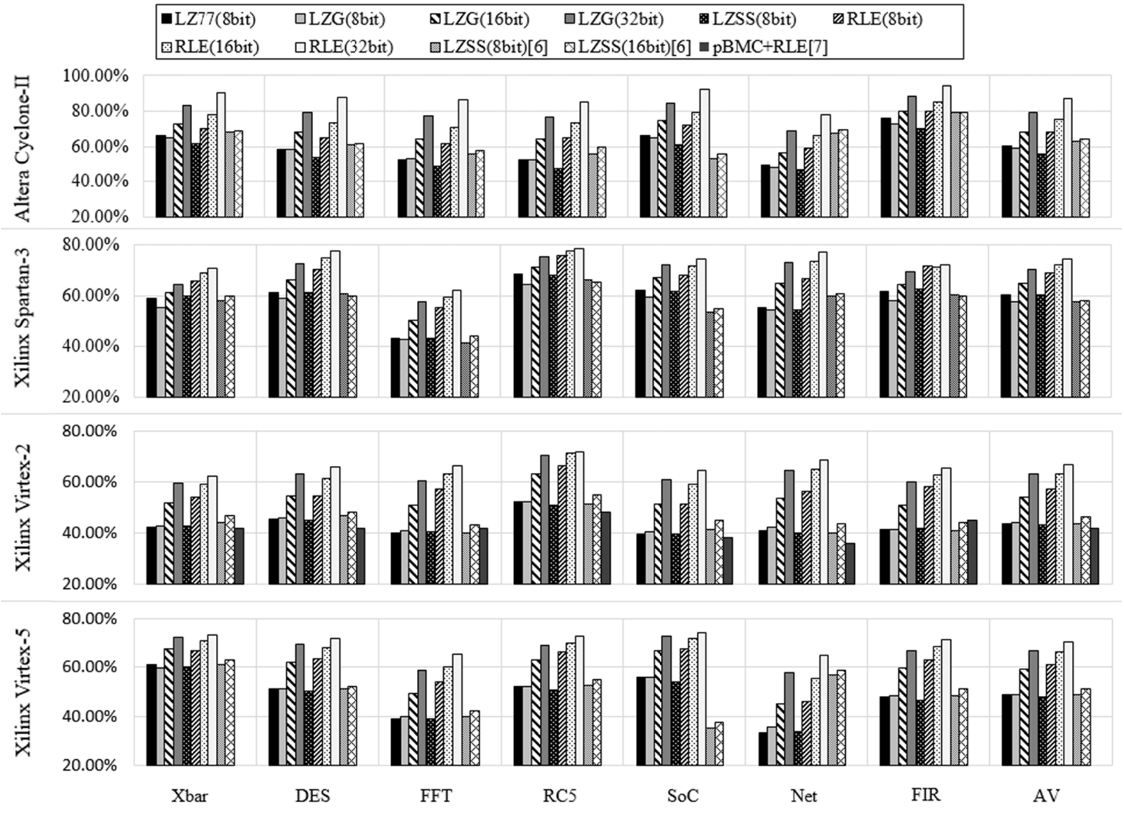


Fig. 14. Compression ratio of both Xilinx and Altera bitstreams.

Others are less efficient than the approaches [6]. The position–length pair in LZSS(8 b) [6] is one byte long, while the position–length pair is two bytes long in this paper. More repetitive patterns can be matched with the longer position–length pair, and thus, the compression ratio of proposed LZSS(8 b) is better than that of LZSS(8 b) [6] except for SoC applications. However, more hardware resource are needed for the longer position–length pair. A bitmask-based method is designed to encode random bit changes, and RLE is appropriate for the large amount of repetitive patterns. pBMC + RLE [7] takes advantage of using bitmask and RLE to compress the Xilinx Virtex-2 bitstreams. As a result, pBMC + RLE [7] has a better compression ratio on some applications. Although some compression approaches have exhibited favorable compression ratio, no single compression method is efficient for the whole benchmark.

Furthermore, the proposed four compression techniques with different input and output widths outperform the Xilinx compression method by 14%–22% (30.17%–16.19 to 30.17%–8.04%) for the 11 "Default" partial bitstreams shown in Fig. 15. Because of many repetitive patterns in the partial bitstreams, the proposed four compression techniques are more efficient than Xilinx compression method for partial bitstreams.

3) Decompression Throughput: The decompression throughput of the benchmark is illustrated in Table II. With the increasing data width of the compression decoder, the decompression throughput increases accordingly. Compared with LZG(8 b), LZG_x1(32 b) with 32-b input–output achieves about 2.7 times higher decompression performance. On the other hand, applying dataflow optimization can pipeline sequential tasks and improve performance. Based on the design of LZG_x1(32 b),LZG_x3(32 b) adds extra two tasks and performs dataflow optimization. LZG_x2(32 b) contains two tasks and also applies dataflow optimization. Although using this optimization method increases hardware resource overhead, LZG_x3(32 b) achieves 23% higher decompression throughput than LZG_x1(32 b). This optimization also applies to the RLE decoder.

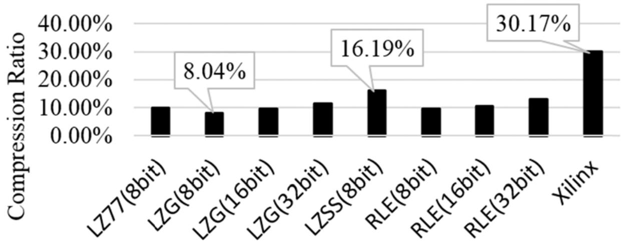


Fig. 15. Compression ratio of partial bitstreams.

4) Hardware Resource Overhead: Table II lists the hardware resource overhead of each compression decoder. With the increasing data width of the compression decoder, the hardware resource overhead increases accordingly. Compared with RLE(8 b), RLE_x1(32 b) with 32-b input–output costs about 3 times registers and LUTs. LZG_x1(32 b) costs 2.03 times registers using area optimization method, but it costs 3.4 times LUTs compared with LZG(8bit). The shift register in LZG

TABLE II
IMPLEMENTATION RESULTS OF COMPRESSION DECODERS

Type	LZ77 (8bit)	LZG (8bit)	LZG (16bit)	LZG_x1 (32bit)	LZG_x2 (32bit)	LZG_x3 (32bit)	LZSS (8bit)	RLE (8bit)	RLE (16bit)	RLE_x1 (32bit)	RLE_x2 (32bit)	RLE_x3 (32bit)
Register	399	93	153	189	342	487	311	33	57	105	141	177
LUT	1197	688	1252	2354	2608	2899	365	41	79	121	198	276
Compression Ratio	44.54%	43.08%	56.03%	68.13%			41.32%	71.51%	74.06%	76.23%		
Average Decompression Throughput (MB/s)	102.2	136.7	259.6	499.8	580.3	615.7	76.1	133.5	261.6	508.3	606.8	652.3
Decompression Efficiency (Normalized)	62.67%	83.85%	79.59%	76.62%	88.97%	94.39%	46.68%	81.89%	80.21%	77.93%	93.03%	100.00%

decoders is implemented by LUTs. According to the wider of the input and output, the resource overhead of the LUT increases more than that of the register.

5) Decompression Efficiency: The decompression efficiency of the compression decoders is shown in Table II. With the increasing data width of the compression decoder, the decompression efficiency decreases. However, decompression efficiency increases with the dataflow optimization. The decompression efficiency of the RLE_x3(32 b) is superior to that of other decoders. The reasons behind these results come from the highest decompression throughput and dataflow optimization.

The implementation results of the three similar dictionary-based methods are shown in Table II. LZSS(8 b) has better average compression ratio than LZ77(8 b) and LZG(8 b). Since the decompression method of LZG is simpler, LZG(8 b) achieves higher decompression throughput than that of LZSS(8 b) and LZ77(8 b). LZG(8 b) costs the medium hardware resource with the least number of registers and medium number of LUTs. Thus, LZG(8 b) is more efficient than LZ77(8 b) and LZSS(8 b).

6) Impact of FIFO Depth on the Decompression Throughput: In order to estimate the impact of FIFO depth on the decompression throughput, we perform measurements for different depths of the IN_FIFO and OUT_FIFO as shown in Fig. 12. The depth of both IN_FIFO and OUT_FIFO varies from 16 to 512. Experimental results show that FIFO depth has a little impact (less than 2%) on the performance of the DMA controller. However, the FIFO depth does not have an impact on the decompression throughput. Larger FIFOs cannot achieve an improvement in decompression throughput.

7) Comparison and Analysis: A comparison of different compression decoders is shown in Table III. Although the compression ratio of RLE_x3(32 b) is worse than that of the existing compression decoders, RLE_x3(32 b) can run at higher frequency and achieve higher decompression throughput with less hardware resource overhead. But RLE_x3(32 b)is not appropriate for software binaries with the large amount of random symbols. Nevertheless, LZG(8 b) is efficient for software binaries with medium hardware resource and good compression ratio.

In addition to the comparison given above, the decompression throughput of RLE_x3(32 b) and LZG_x3(32 b) is superior to that of the existing compression decoders on the benchmark. The reasons behind these results come from different kinds of optimizations.

C. Embedded System

In order to evaluate the startup time of the embedded system, the same architecture shown in Fig. 12 is used. The SD card controller runs at 50 MHz with the maximum throughput of 25 MB/s. The ARM processor runs at 666.66 MHz and has a 32-b timer. The timer runs at 333.33 MHz and is used to measure the startup time. The compressed software binaries are stored in the external SD card before system power-ON. In this system, the software binaries are so large that they have to run in the DDR3 SDRAM. The startup time ratio is shown in the following:

$$\text{Startup Time Ratio} = \frac{\text{Reading Time} + \text{Decompression Time}}{\text{Original Startup Time}}. \tag{4}$$

The original startup time only includes reading uncompressed software binaries from the SD card and writing them to the DDR3 SDRAM. However, when using compression methods illustrated in Fig. 12, the startup time not only contains the reading compressed software binaries from the SD card ①, but also includes the decompression time between the DDR3 SDRAM and the compression decoders ② and ③. The reading time of the compressed/uncompressed software binaries is linear to the size of compressed/uncompressed software binaries. The boot loader runs on the on-chip memory with instruction cache and data cache enable. The decompression time not only contains the communication time between the DDR3 SDRAM and the compression decoder, but also includes the initialization of the DMA. Fig. 16 presents the average startup time ratio (4) for different compression decoders over the 25 software binaries. As the decompression throughput of the hardware compression decoders is higher than the maximum throughput of the SD card controller, the hardware decompression time is less than 25% of the original startup time. Due to the compression of the software binaries, the reading time of the compressed software binaries is less than that of the original time. However, since the RLE compression methods are not efficient for software binaries, the total startup time is more than original startup time.

TABLE III
COMPARISON OF DIFFERENT COMPRESSION DECODERS

	LZG (8bit)	RLE_x3 (32bit)	LZG_x3 (32bit)	LZSS (8bit) [5][6]	LZSS (16bit) [5][6]	pBMC +RLE [7]	LZSS [21]	FDIC [24]	LZSS [24]	LZ77+ Bitmask [26]	Bitmask [27]	Bitmask [28]	GZIP [43]
Slice Usage	medium	low	high	low	low	low	low	low	low	high	low	N/A	highest
BRAM		0											10.5
Fmax(MHz)	205	381	205	198	200	195	75	395	277	109	130	326.8	165
Compression Ratio	good	not good	general	good	good	good	good	good	good	good	good	good	best
Average Decompression Throughput (MB/s)	136.7	652.2	615.7	198*	400*	N/A	N/A	424.6*	115.8*	211.5	N/A	N/A	495

* This is the maximum decompression throughput.

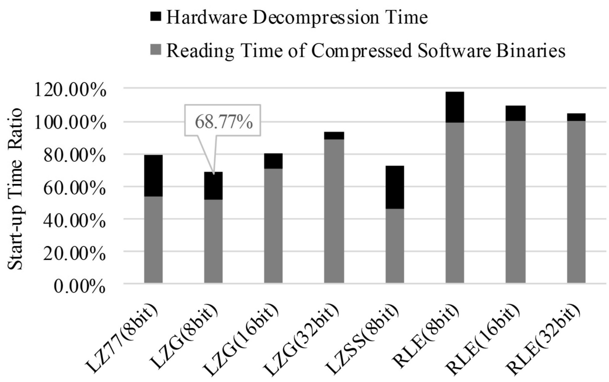


Fig. 16. Startup time ratio of different compression decoders.

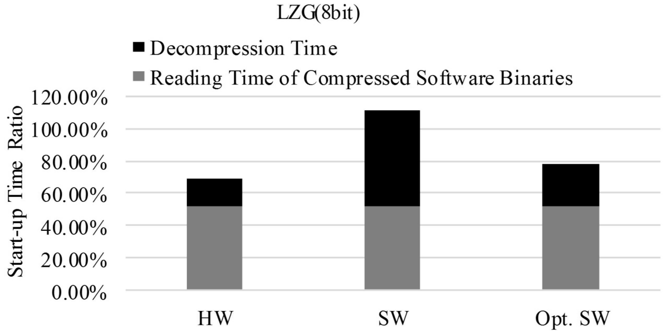
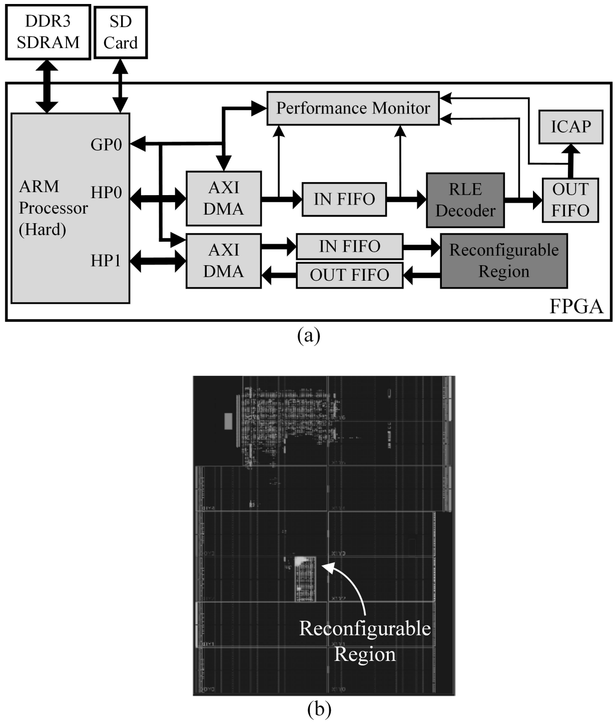


Fig. 17. Comparison of startup time ratio between hardware and software.

The results shown in Fig. 16 reveal that LZG(8 b) is the best method to reduce the software startup time. Because LZG(8 b) has a good compression ratio and high decompression throughput, it can reduce 31.23% (1%–68.77%) original startup time and save 48.98% (1%–51.02%) external memory.

In the evaluation system shown in Fig. 12, the compressed software binaries can also be decompressed by the software compression decoder running on the hard ARM processor. Fig. 17 illustrates the comparison of the average startup time ratio between the hardware LZG(8 b) decoder and the software LZG(8 b) decoder for the 25 software binaries. Compared with the default software (SW) compression decoder, the optimized software (Opt. SW) compression decoder can reduce about 30% startup time. However, hardware (HW) compression decoder also can achieve speed-up compared with the optimized software compression decoder.



(a)

(b)

Fig. 18. (a) Architecture of the partial reconfigurable system. (b) FPGA floorplan.

D. Partial Reconfigurable System

Fig. 18 presents the architecture of the partial reconfigurable system and the FPGA floorplan. Compared with the architecture of the general evaluation system shown in Fig. 12, the partial reconfigurable system adds another HP port and includes a reconfigurable region. The RLE_x3(32 b) decoder is applied to the partial reconfigurable system because of the highest decompression throughput and less hardware resource overhead. The Internal Configuration Access Port (ICAP) runs at 100 MHz. Overclocking of the ICAP is not recommended by the device vendor. To run safely, the speed limit of the ICAP is 100 MHz. Therefore, the maximum throughput of

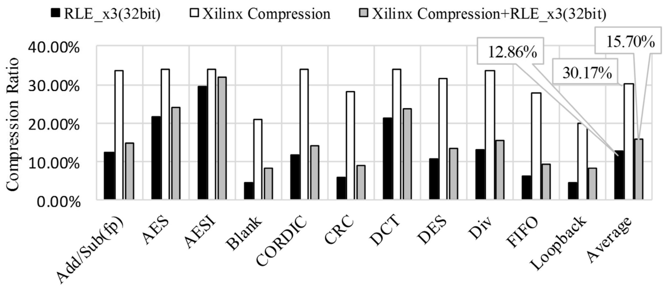


Fig. 19. Compression ratio of partial bitstreams.

the ICAP is 400 MB/s with 32-b mode. As discussed in Section V-B, the FIFO depth does not have impact on the decompression throughput. The RLE decoder runs at 142.8 MHz, and the maximum throughput of the RLE decoder is theoretically higher than 400 MB/s. Thus, the performance of the ICAP is not affected by the RLE decoder and FIFO depth. Fig. 19 shows the compression ratio of partial bitstreams. The average compression ratio of the RLE_x3(32 b) for the "Default" partial bitstreams is 12.86% and an 87.14% (1%–12.86%) memory saving is achieved. At the same time, 86% memory bandwidth can be saved. Combining with the "Xilinx Compression" method and RLE_x3(32 b), not only can 84.3% (1%–15.7%) memory and 83% memory bandwidth be saved, but also 69.83% (1%–30.17%) reconfiguration time can be reduced by using multiple-frame write sequences.

VI. Conclusion

Compression techniques for bitstreams and software binaries have been extensively studied, but the existing compression decoders are designed using hand-written RTL descriptions and provide low decompression throughput. In this paper, four lossless compression decoders are developed using HLS. A benchmark including 28 full bitstreams, 33 partial bitstreams, and 25 software binaries is provided. Various improvements and optimizations are applied to the compression decoders in order to balance the objectives of compression ratio, decompression throughput, and hardware resource overhead simultaneously. Evaluations of the synthesizable compression decoders are demonstrated on a Xilinx ZC706 board running at 200 MHz, showing higher decompression throughput than that of the existing compression decoders on the benchmark. Moreover, the proposed decoders LZG(8 b) can reduce software startup time by up to 31.23% in embedded systems, and RLE(32 b) can reduce 69.83% reduction of the reconfiguration time for partial reconfigurable systems.

In addition, since no specific bitstream organizations or data structures are required, LZG(8 b) is applicable to other full bitstreams and software binaries. RLE(32 b) is efficient for partial bitstreams with low hardware resource usage and high decompression throughput. Although the proposed design method was demonstrated on a Xilinx ZYNQ FPGA, it can be applied to other FPGAs. The synthesizable lossless compression decoders and the benchmark can be downloaded from the website [45].

Currently, we are exploring the combination of these compression decoders for a wide range of applications. Moreover, we plan to include error checking and error reporting logic within the compression decoders to improve reliability.

References

[1] *UltraScale Architecture Configuration User Guide, UG570 (v1.6)*, Xilinx Inc., San Jose, CA, USA, Dec. 2015.

[2] *Stratix V Device Datasheet*, Altera Inc., San Jose, CA, USA, Jun. 2016.

[3] A. Dandalis and V. K. Prasanna, "Configuration compression for FPGA-based embedded systems," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2001, pp. 173–182.

[4] 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.

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

[6] D. Koch, C. Beckhoff, and J. Teich, "Hardware decompression techniques for FPGA-based embedded systems," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 2, Jun. 2009, Art. no. 9.

[7] X. Qin, C. Muthry, and P. Mishra, "Decoding-aware compression of FPGA bitstreams," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 19, no. 3, pp. 411–419, Mar. 2011.

[8] J. Meyer, J. Noguera, M. Hüebner, R. Stewar, and J. Becker, "Embedded systems start-up under timing constraints on modern FPGAs," in *Proc. 21st Int. Conf. Field-Program. Logic Appl.*, Sep. 2011, pp. 103–109.

[9] S. Hauck, Z. Li, and E. Schwabe, "Configuration compression for the Xilinx XC6200 FPGA," in *Proc. IEEE Symp. FPGAs Custom Comput. Mach.*, Apr. 1998, pp. 138–146.

[10] 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.

[11] S. Hauck and W. D. Wilson, "Runlength compression techniques for FPGA configurations," in *Proc. 7th IEEE Symp. Field-Program. Custom Comput. Mach.*, Apr. 1999, pp. 286–287.

[12] Z. Li and S. Hauck, "Don't care discovery for FPGA configuration compression," in *Proc. ACM/SIGDA 7th Int. Symp. Field-Program. Gate Arrays*, Feb. 1999, pp. 91–98.

[13] Z. Li and S. Hauck, "Configuration compression for virtex FPGAs," in *Proc. 9th IEEE Symp. Field-Program. Custom Comput. Mach.*, Mar. 2001, pp. 147–159.

[14] J. Pan, T. Mitra, and W.-F. Wong, "Configuration bitstream compression for dynamically reconfigurable FPGAs," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Nov. 2004, pp. 766–773.

[15] F. Farshadjam, M. Fathy, and M. Dehghan, "A new approach for configuration compression in Virtex based RTR systems," in *Proc. Conf. Elect. Comput. Eng.*, May 2004, pp. 1093–1096.

[16] M. Martina, G. Masera, A. Molino, F. Vacca, L. Sterpone, and M. Violante, "A new approach to compress the configuration information of programmable devices," in *Proc. Conf. Design, Autom. Test Eur., Designers' Forum*, May 2006, pp. 48–51.

[17] *Stratix II Device Handbook*, Altera Inc., San Jose, CA, USA, May 2007, vol. 1.

[18] A. Khu, "Xilinx FPGA configuration data compression and decompression," Xilinx Inc., San Jose, CA, USA, Tech. Rep. WP152 (v1.0), Sep. 2001.

[19] *Platform Flash PROM User Guide, UG161 (v1.5)*, Xilinx Inc., San Jose, CA, USA, Oct. 2009.

[20] *Enhanced Configuration (EPC) Devices Datasheet*, Altera Inc., San Jose, CA, USA, May 2016.

[21] M. Huebner, M. Ullmann, F. Weissel, and J. Becker, "Real-time configuration code decompression for dynamic FPGA self-reconfiguration," in *Proc. 18th Int. Parallel Distrib. Process. Symp.*, Apr. 2004, p. 138.

[22] C. Beckhoff, D. Koch, and J. Torresen, "Portable module relocation and bitstream compression for Xilinx FPGAs," in *Proc. 24th Int. Conf. Field Programm. Logic Appl. (FPL)*, Sep. 2014, pp. 1–8.

[23] S. Bayar and A. Yurdakul, "A dynamically reconfigurable communication architecture for multicore embedded systems," *J. Syst. Archit.*, vol. 58, nos. 3–4, pp. 140–159, Mar. 2012.

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

[25] F. Duhem, F. Müller, and P. Lorenzini, "Reconfiguration time overhead on field programmable gate arrays: Reduction and cost model," *IET Comput. Digit. Tech.*, vol. 6, no. 2, pp. 105–113, Mar. 2012.

[26] Y. Gao, H. Ye, J. Wang, and J. Lai, "FPGA bitstream compression and decompression based on LZ77 algorithm and BMC technique," in *Proc. IEEE 11th Int. Conf. ASIC (ASICON)*, Nov. 2015, pp. 1–4.

[27] S.-W. 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.

[28] W. J. Wang and C. H. Lin, "Code compression for embedded systems using separated dictionaries," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 1, pp. 266–275, Jan. 2016.

[29] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," *J. Design Autom. Embedded Syst.*, vol. 16, no. 3, pp. 31–51, Sep. 2012.

[30] S. Windh *et al.*, "High-Level language tools for reconfigurable computing," *Proc. IEEE*, vol. 103, no. 3, pp. 390–408, Mar. 2015.

[31] *Run Length Encoding/Decoding Reference Software*, accessed on 2013. [Online]. Available: https://sourceforge.net/projects/nikkhokkho/files/RLE64/

[32] *LZ77 Reference Software*, accessed on 2004. [Online]. Available: https://sourceforge.net/projects/compressions/

[33] *LZSS Reference Software*, accessed on 2004. [Online]. Available: https://oku.edu.mie-u.ac.jp/~okumura/compression/lzss.c

[34] *LIBLZG Reference Software*, accessed on 2014. [Online]. Available: http://liblzg.bitsnbites.eu/

[35] *High-Level Synthesis User Guide, UG902 (v2015.4)*, Xilinx Inc., San Jose, CA, USA, Nov. 2015.

[36] *ReCoNets*, accessed on 2007. [Online]. Available: http://www.reconets.de/bitstreamcompression/

[37] *Pipelined DCT/IDCT Project*, accessed on 2010. [Online]. Available: http://www.opencores.org

[38] *AES Project*, accessed on 2002. [Online]. Available: http://www.opencore.org

[39] *DES Project*, accessed on 2001. [Online]. Available: http://www.opencore.org

[40] *CRC Project*, accessed on 2005. [Online]. Available: http://www.opencore.org

[41] A. Sarangi, S. MacMahon, and U. Cherukupaly, "lightweight IP application examples," Xilinx Appl. Note, XAPP1026 (v5.1), Xilinx Inc., San Jose, CA, USA, Nov. 2014.

[42] GZIP, accessed on 2003. [Online]. Available: http://www.gzip.org/

[43] *GZIP/ZLIB/Deflate Data Compression Core*, CAST Inc., Woodcliff Lake, NJ, USA, accessed on 2017. [Online]. Available: http://www.cast-inc.com/ip-cores/data/zipaccel-d/cast-zipaccel-d-x.pdf

[44] 7-zip, accessed on 2016. [Online]. Available: http://www.7-zip.org/

[45] *Synthesizable Lossless Compression Decoders*, accessed on 2016. [Online]. Available: https://github.com/jian1/lossless-compression-decoders

Jian Yan received the B.S. degree from the School of Communication and Information Engineering, Shanghai University, Shanghai, China, in 2011, and the M.S. degree from the School of Microelectronics, Fudan University, Shanghai, in 2013, where he is currently pursuing the Ph.D. degree.

His current research interests include the code compression, computer architecture, computing resource virtualization, and the development of partially reconfigurable systems.

Junqi Yuan received the B.S. degree from the School of Information Science and Technology, Fudan University, Shanghai, China, in 2013.

His current research interests include field-programmable gate array logic architecture and computer-aided design.

Philip H. W. Leong (SM'02) received the B.Sc., B.E., and Ph.D. degrees from The University of Sydney, Sydney, NSW, Australia.

In 1993, he joined ST Microelectronics, Milan, Italy, as a Consultant, where he was involved in advanced flash memory-based integrated circuit design. From 1997 to 2009, he was with The Chinese University of Hong Kong, Hong Kong. He is currently a Professor of Computer Systems with the School of Electrical and Information Engineering, The University of Sydney, a Visiting Professor with Imperial College, London, U.K., and a Chief Technology Advisor of ClusterTech, Hong Kong.

Dr. Leong was a recipient of the 2005 FPT conference Best Paper Award and the 2007 and 2008 FPL conference Stamatis Vassiliadis Outstanding Paper Awards.

Wayne Luk (F'09) received the M.A., M.Sc., and D.Phil. degrees in engineering and computing science from Oxford University, Oxford, U.K.

He founded and leads the Computer Systems Section, Custom Computing Group, Department of Computing, Imperial College London, U.K. He was a Visiting Professor at Stanford University, Stanford, CA, USA, and at Queen's University Belfast, Belfast, U.K. He is currently a Professor of Computer Engineering with Imperial College, London, U.K. He has authored or edited six books and four special journal issues.

Dr. Luk is a member of the Program Committee of many international conferences such as FCCM, FPGA, and DATE. He is a fellow of the BCS. He had ten papers that received awards from the ASAP, FPL, FPT, SAMOS, SPL, and ERSA conferences. He received the Research Excellence Award from Imperial College in 2006. He is the founding Editor-in-Chief of the *ACM Transactions on Reconfigurable Technology and Systems*.

Lingli Wang (M'99) received the M.S. degree in electrical engineering from Zhejiang University, Hangzhou, China, in 1997, and the Ph.D. degree in electrical engineering from Edinburgh Napier University, Edinburgh, U.K., in 2001.

He was with the Altera European Technology Centre, Holmers Farm Way, High Wycombe, Buckinghamshire, U.K. for four years. In 2005, he joined Fudan University, Shanghai, China, where he is currently a Full Professor with the State Key Laboratory of ASIC and System, School of Microelectronics. His current research interests include logic synthesis, reconfigurable computing, and quantum computing.