Configuration Compression for FPGA-based Embedded Systems*

Andreas Dandalis
Electrical Engineering - Systems
University of Southern California
3740 McClintock Avenue, EEB 234
Los Angeles, CA 90089-2562
dandalis@usc.edu

Viktor K. Prasanna
Electrical Engineering - Systems
University of Southern California
3740 McClintock Avenue, EEB 200
Los Angeles, CA 90089-2562
prasanna@ganges.usc.edu

ABSTRACT

FPGAs are a promising technology for developing high-performance embedded systems. The density and performance of FPGAs have drastically improved over the past few years. Consequently, the size of the configuration bit-streams has also increased considerably. As a result, the cost-effectiveness of FPGA-based embedded systems is significantly affected by the memory required for storing various FPGA configurations. This paper proposes a novel compression technique that reduces the memory required for storing FPGA configurations and results in high decompression efficiency. Decompression efficiency corresponds to the decompression hardware cost as well as the decompression rate. The proposed technique is applicable to *any* SRAM-based FPGA device since configuration bit-streams are processed as raw data. The required decompression hardware is simple and is independent of the individual semantics of configuration bit-streams or specific features of the on-chip configuration mechanism. Moreover, the time to configure the device is not affected by our compression technique. Using our technique, we demonstrate up to 41% savings in memory for configuration bit-streams of several real-world applications.

1. INTRODUCTION

The enormous growth of embedded applications has made embedded systems an essential component in products that emerge in almost every aspect of our life: digital TVs, game consoles, network routers, cellular base-stations, digital communication devices, printers, digital copiers, multifunctional equipment, house appliances, etc. For example, from 200 million units shipped in the year 1997, the DSP embedded systems market has been forecast to grow to 1,200 million units in the year 2001 [13]. The goal of embedded systems is to perform a set of specific tasks to improve the functionality of larger systems. As a result, they are usually not visible to the end-user since they are *embedded* in larger systems. Embedded systems usually consist of a processing unit, memory to store data and programs, and an I/O interface to communicate with other components of the larger system. Their complexity depends on the complexity of the tasks they perform. The main characteristics of an embedded system are raw computational power and cost-effectiveness. The cost-effectiveness of an embedded system includes characteristics such as product lifetime, overall price, and power consumption, among others.

The unique combination of hardware-like performance with software-like flexibility make FPGAs a highly promising solution for embedded systems. Typical FPGA-based embedded systems have FPGA devices as their processing unit, memory to store data and FPGA configurations, and an I/O interface to transmit and receive data. FPGA-based embedded systems can sustain high processing rates while providing a high degree of flexibility required in dynamically changing environments. FPGAs can be reconfigured on demand to support multiple algorithms and standards. Thus, the degree of system flexibility strongly depends on the amount of configuration data that can be stored in the field. However, the size of the configuration bit-stream has increased considerably over the past few years. For example, the size of the configuration bit-stream of the VIRTEX series FPGAs, range from 0.6 Mbits to 16 Mbits [14]. As a result, storing configuration bit-streams in an FPGA-based embedded system becomes a critical problem drastically affecting the cost-effectiveness of the system.

In this paper, we propose a novel compression technique to reduce the memory requirements for storing configuration bit-streams in FPGA-based embedded systems. By compressing configuration bit-streams, significant savings in memory requirements can be achieved. The configuration compression occurs off-line. At runtime, decompression occurs and the decompressed data is fed to the on-chip configuration mechanism to configure the device. The major performance requirements of the compression problem are the decompression hardware cost and the decompression rate. The above requirements distinguish our compression problem from conventional software-based applications. We are not aware of any prior work that addresses the configura-

*This work is supported by the DARPA Adaptive Computing Systems program under contract no. DABT63-99-1-0004 monitored by Fort Huachuca and in part by the National Science Foundation under grant no. CCR-9900613.

FPGA 2001, February 11-13, 2001, Monterey, CA, USA.

tion compression problem of FPGA-based embedded systems with respect to the cost and speed requirements.

Our compression technique is applicable to any SRAM-based FPGA device since it does not depend on specific features of the configuration mechanism. The configuration bit-streams are processed as raw data without considering individual semantics. As a result, both complete and partial configuration schemes can be supported. The required decompression hardware is simple and independent of the configuration format or characteristics of the configuration mechanism. In addition, the achieved compression ratio is independent of the decompression hardware and depends only on the entropy of the configuration bit-stream. Finally, the time to configure an FPGA depends only on the data rate of the on-chip configuration mechanism, the speed of the memory that stores the configuration data, and the size of the configuration bit-stream. The decompression process does not add any overhead to the configuration time.

The proposed compression technique is based on the principles of dictionary-based compression algorithms. Even though statistical methods can achieve higher compression ratios, we propose a dictionary-based approach because statistical methods lead to high decompression hardware cost. The dictionary corresponds to configuration data that is stored in the memory. In our scheme, the dictionary is derived based on the well-known LZW compression algorithm [11]. However, a major deviation from LZW-based algorithms is the calculation of the compression ratio. Our compression technique proposes a novel way of constructing the dictionary to significantly improve the compression ratio. In addition, our technique delivers the decompressed data in order. On the contrary, in conventional LZW-based algorithms, the decompressed data is delivered in reverse order. By using a stack, the original data is reconstructed. The latter significantly affects the decompression rate of a hardware implementation.

Using our technique, we demonstrated $11-41\%$ savings in memory for configuration bit-streams of several real-world applications. The configuration bit-streams corresponded to cryptographic and digital signal processing algorithms. Our target architecture was VIRTEX series FPGAs [14]. The size of the configuration bit-streams ranged from 1.7 Mbits to 6.1 Mbits.

An overview of the configuration of SRAM-based FPGAs is given in Section 2. In Section 3, various aspects of compression techniques and the constraints imposed by embedded systems are presented. Our novel compression technique is described in Section 4. Experimental results are demonstrated in Section 5 and related work is described in Section 6. Finally, in Section 7, possible extensions to our work are described.

2. FPGA CONFIGURATION

An FPGA configuration determines the functionality of the FPGA device. An FPGA device is configured by loading a configuration bit-stream into its internal configuration memory. An internal controller manages the configuration memory as well as the configuration data transfer via the I/O interface. Throughout this paper, we refer to both the configuration memory and its controller as the configuration mechanism. Based on the technology of the internal configuration memory, FPGAs can be permanently configured once or can be reconfigured in the field. For example, Anti-Fuse technology allows one-time programmability while SRAM technology allows reprogrammability.

In this paper, we focus on SRAM-based FPGAs. In SRAM-based FPGAs, the contents of the internal configuration memory are reset after power-up. As a result, the internal configuration memory cannot be used for storing configuration data permanently. Using partial configuration, only a part of the contents of the internal configuration memory is modified. As a result, the configuration time can be significantly reduced compared with the configuration time required for a complete reconfiguration. Moreover, partial configuration can occur at runtime without interrupting the computations that an FPGA performs. SRAM-based FPGAs require external devices to initiate and control the configuration process. Usually, the configuration data is stored in an external memory and an external controller supervises the configuration process.

The time required to configure an FPGA depends on the size of the configuration bit-stream, the clock rate and the operation mode of the configuration mechanism, and the throughput of the external memory that stores the configuration bit-stream. Typical sizes of configuration bit-streams range from 0.6 Mbits to 16 Mbits [1, 2, 14] depending on the density of the device. The clock rate of the configuration mechanism determines the rate at which the configuration data is delivered to the FPGA device. The configuration data can be transferred to the configuration mechanism serially or in parallel. Parallel modes of configuration result in faster configuration time. Typical values of data rates can be as high as 480 Mbits/sec [1, 2, 14]. Thus, the external memory that stores the configuration bit-stream should be able to sustain the data rate of the configuration mechanism. Otherwise, the memory becomes a performance bottleneck and the time to configure the device increases. The latter could be critical for applications where an FPGA is configured on-demand based on run-time parameters.

Configuration bit-streams consist of data to be stored in the internal configuration memory as well as instructions to the configuration mechanism. The data configures the FPGA architecture, that is, the configurable logic blocks, the interconnection network, the I/O pins, etc. The instructions control the functionality of the configuration mechanism. Typically, instructions are used for initializing the configuration mechanism, synchronizing clock rates, and determining the memory addresses at which the data will be written. The format of a configuration bit-stream depends on the characteristics of the configuration mechanism as well as the characteristics of the FPGA architecture. As a result, the bit-stream format varies among different vendors or, even among different FPGA families of the same vendor.

3. COMPRESSION TECHNIQUES: APPLICABILITY & IMPLEMENTATION COST

Data compression has been extensively studied in the past. Numerous compression algorithms have been proposed to reduce the size of data to be stored or transmitted over a network. The effectiveness of a compression technique is characterized by the achieved compression ratio, that is, the ratio of the size of the compressed data to the size of the original data. However, depending on the application, metrics such as processing rate, implementation cost, and adaptability may become critical performance issues. In this section,

we will discuss compression techniques and the requirements to be met for compressing FPGA configurations in FPGA-based embedded systems.

In general, a compression technique can be either lossless or lossy. Lossless compression techniques reconstruct the exact original data after decompression. Lossless techniques are used in applications where any loss of information after decompression is critical. On the contrary, lossy compression techniques eliminate certain information of the original data after decompression. Lossy techniques are primarily used in image, video, and audio applications. For configuration compression, the configuration bit-stream should be reconstructed without loss of any information and thus, a lossless compression technique should be used. Otherwise, the functionality of the FPGA may be altered or, even worse, the FPGA may be damaged.

Lossless compression techniques are based on statistical methods or dictionary-based schemes. For any given data, statistical methods can result in better compression ratios than any dictionary-based scheme [11]. Using statistical methods, a symbol in the original data is encoded with a number of bits proportional to the probability of its occurrence. By encoding the most frequently-occurring symbols with fewer bits than their binary representation requires, the data is compressed. The compression ratio depends on the entropy of the original data as well as the accuracy of the model that is utilized to derive the statistical information of the given data. However, the complexity of the decompression hardware can significantly increase the cost of such an approach. In the context of embedded systems, dedicated decompression hardware (e.g., CAM memory) is required to align codewords of different lengths as well as determine the output of a codeword.

In dictionary-based compression schemes, single codewords encode variable-length strings of symbols [11]. The codewords form an *index* to a phrase dictionary. Decompression occurs by parsing the dictionary with respect to its index. Compression is achieved if the codewords require smaller number of bits than the strings of symbols that they replace. Contrary to statistical methods, dictionary-based schemes require significantly simpler decompression hardware. Only memory read operations are required during decompression and high decompression rates can be achieved. The latter suggests that, in the context of FPGA-based embedded systems, a dictionary-based scheme would result in fairly low implementation cost.

In Figure 1, a typical architecture of FPGA-based embedded systems is shown. These systems consist of an FPGA device(s), memory to store data and FPGA configurations, a configuration controller to supervise the configuration process, and an I/O interface to send and receive data. The configurations are compressed off-line by a general-purpose computer and the compressed data is stored in the embedded system. Besides the memory requirements for the compressed data, additional memory may be required during decompression. For example, in *LZ*-based algorithms [11], the dictionary can be reconstructed on the fly based on the index. As a result, in software-based applications, only the index is stored or transmitted. Thus, only the index is considered in the calculation of the compression ratio. However, in the context of embedded systems, the memory requirements to store the dictionary should also be considered.

At runtime, decompression occurs and the original configuration bit-stream is delivered to the FPGA configuration mechanism. As a result, the decompression hardware cost and the decompression rate become major requirements of the compression problem. The decompression hardware cost may affect the cost of the system. In addition, if the decompression rate can not sustain the data rate of the configuration mechanism, the time to configure the FPGA will increase.

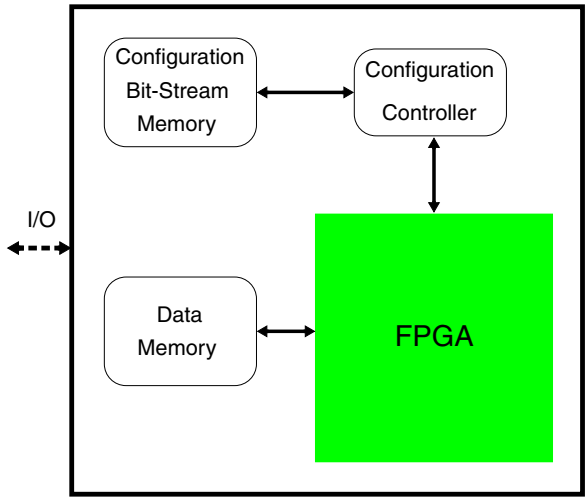

Figure 1: FPGA-based embedded system architecture

4. OUR COMPRESSION TECHNIQUE

Our compression technique is based on the principles of dictionary-based compression algorithms. Even though statistical methods can achieve higher compression ratios [11], we propose a dictionary-based approach because dictionary-based schemes lead to simpler and faster decompression hardware. In our approach, the dictionary corresponds to configuration data and the index corresponds to the way the dictionary is read in order to reconstruct a configuration bit-stream. In Figure 2, an overview of our configuration compression technique is shown. The input configuration bit-stream is read sequentially in the reverse order. Then, the dictionary and the index are derived based on the principles of the well-known *LZW* compression algorithm [11]. In general, finding a dictionary that results in optimal compression has exponential complexity [11]. By deleting non-referenced nodes and by merging common prefix strings, a compact representation of the dictionary is achieved. Finally, a heuristic is applied that further enhances the dictionary representation and leads to savings in memory. The original configuration bit-stream can be reconstructed by parsing the dictionary with respect to the index in reverse-order. The achieved compression ratio is the ratio of the total memory requirements (i.e., dictionary and index) to the size of the bit-stream. In the following, we describe in detail our compression technique as well as the decompression method.

In [6], we have demonstrated preliminary configuration compression results using a dictionary-based approach. However, the approach that is proposed in this paper is significantly different than the one proposed in [6]. In [6], the

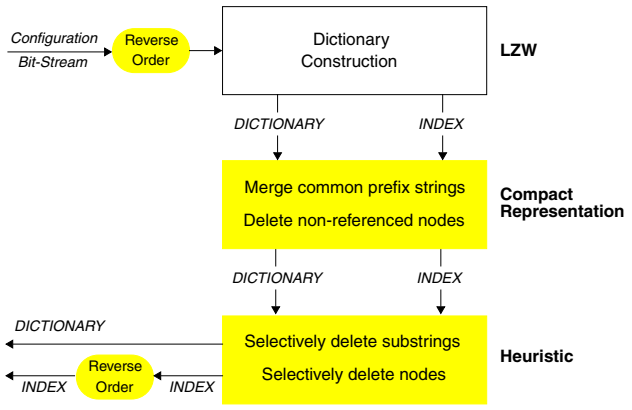

Figure 2: Our configuration compression technique

decompressed strings are delivered in the reverse order. In addition, the heuristic proposed in [6] only deletes all the leaf nodes from all the suffix trees without considering how the index size is affected. However, in this paper, a different approach is proposed. Our goal is to delete strings and individual nodes in a bottom-up approach considering the overall savings in memory, that is, both the dictionary and the index memory. As a result, by applying our technique to the configuration bit-streams in [6], the memory requirements for storing the dictionary and the index can be further improved by $6-13$ % (see Section 5).

4.1 Basic LZW Algorithm

The LZW algorithm is an adaptive dictionary encoder, that is, the coding technique of LZW is based on the input data already encoded. The input to the algorithm is a sequence of binary symbols. A symbol can be a single bit or a data word. Symbols are processed sequentially. By combining consecutive symbols, strings are formed. In our case, the input is the configuration bit-stream. Moreover, the bit-length of the symbol determines the way the bit-stream is processed (e.g., bit-by-bit, byte-by-byte). The main idea of LZW is to replace the longest possible string of symbols with a reference to an existing dictionary entry. As a result, the derived index consists of pointers to the dictionary.

Initially, the dictionary is preloaded with entries for all the symbols of the input alphabet (Algorithm 1). For example, if the symbol is a byte, the dictionary is preloaded with entries for $0-255$. One symbol s is read at a time. A temporary string S is utilized during compression. If the string Ss is not found in the dictionary, the code for S is added to the index and Ss becomes a new entry to the dictionary. The dictionary contains all the previously seen strings. There is no restriction on the size of the dictionary, so more and more phrases are generated as encoding proceeds. If the string Ss is found in the dictionary, a new symbol is read. The procedure terminates when all the input data has been read.

In software-based applications, only the index is considered in the calculation of the compression ratio. The main advantage of LZW (and any LZ-based algorithm) is that the dictionary can be reconstructed based on the index. As a result, only the index is stored in a secondary storage media or transmitted. The dictionary is reconstructed on-line and the extra memory required is provided by the "host".

Algorithm 1: The LZW algorithm [10]

Input: An input stream of symbols IN.
Output: The dictionary and the index.

```
dictionary ← input alphabet symbols
S = NULL
repeat
        s ← read a symbol from IN
        if Ss exists in the dictionary
          S ← Ss
        else
          output the code for S
          add Ss to the dictionary
          S ← s
        end
until (all input data is read)
```

However, in embedded systems, no secondary storage media is available and the extra required memory has to be considered in the calculation of the compression ratio. Also, note that the dictionary includes phrases that are not referenced by its index. This happens because, as compression proceeds, LZW keeps all the strings that are seen for the first time. This is performed regardless of whether these strings will be referenced or not. This is not a problem in software-based applications since the size of the dictionary is not considered in the calculation of the compression ratio.

4.2 Compact Dictionary Construction

In our approach, we propose a compact memory representation for the dictionary. In general, the dictionary is a forest of suffix trees (i.e., one tree for each symbol of the input alphabet). Each string in a tree is stored in the memory as a singly-linked list. The root of a tree is the head of all the lists in that tree. Every entry in the memory consists of a symbol and an address to a prefix string and every string is associated with an entry. A string is read by traversing the corresponding list from the address of its associated memory entry to the head of the list. Furthermore, dictionary entries that are not referenced in the index are deleted and not stored in the memory.

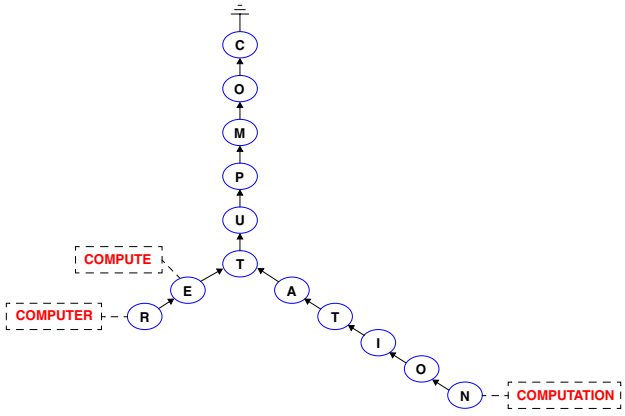

Figure 3: An illustrative example of our dictionary representation

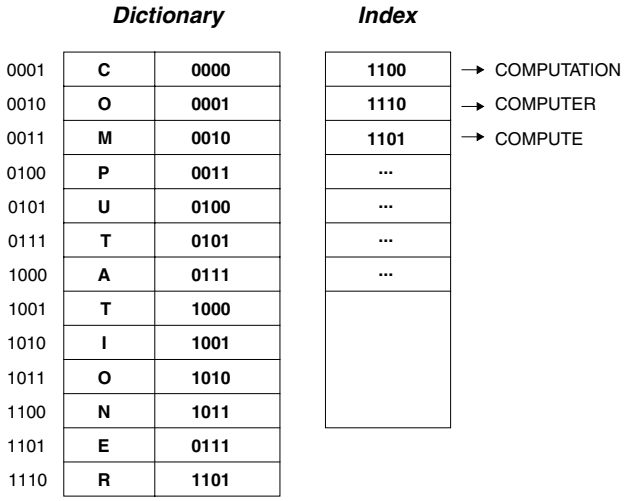

Figure 4: An illustrative example of memory organization for the dictionary and the index

Finally, common prefix strings are merged as one string. An example of our dictionary representation is shown in Figure 3. For illustrative purposes, we consider letters as symbols. The root of the tree is the symbol "C". Each one of the strings "$COMPUTE$", "$COMPUTER$", and "$COMPUTATION$" is associated with a node. Since the string "$COMPUT$" is a common prefix string, it is only represented once in the memory. In Figure 4, the memory organization for storing the dictionary and the index of the above example is shown. The memory requirements for the dictionary are $n_{dictionary} \times (data_{symbol} + \lceil \log_2 n_{dictionary} \rceil)$ bits, where $n_{dictionary}$ is the number of memory entries of the dictionary and $data_{symbol}$ is the number of bits required to represent a symbol. Similarly, the memory requirements for the index are $n_{index} \times \lceil \log_2 n_{dictionary} \rceil$ bits, where n_{index} is the number of memory entries of the index.

From the above example, we notice that during decompression, the decompressed strings are delivered in reverse order. In fact, in software-based implementations [11], a stack is used to deliver each decompressed string in the right order. However, in the considered embedded environment, additional hardware is required to implement the stack. In addition, the size of the stack should be as large as the length of the longest string in the dictionary. Moreover, the time overhead to reverse the order of the decompressed strings would affect the time to configure the FPGA. In our scheme, to avoid the use of a stack, we derive the dictionary after reversing the order of the configuration bit-stream. During decompression, the configuration bit-stream is reconstructed by parsing the index in the reverse order. In this way, the decompressed strings are delivered in order and the exact original bit-stream is reconstructed. We have performed several experiments to examine the impact of compressing a reverse-ordered configuration bit-stream instead of the original one. Our experiments suggest that the memory requirements for both the dictionary and the index are very close to each other in both cases (i.e., variation less than $\pm 1\%$).

4.3 Enhancement of the Dictionary Representation

After deriving the dictionary and its index, we reduce the memory requirements of the dictionary by selectively decomposing strings in the dictionary. In the following, a prefix string corresponds to a path from any node up to the tree root. Similarly, a suffix string corresponds to a path from a leaf node up to any node. Finally, a substring corresponds to a path between any two arbitrary nodes.

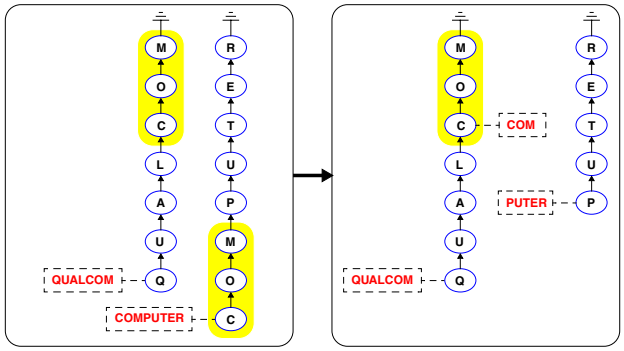

Figure 5: An illustrative example of enhancing the dictionary representation

The main idea is to replace frequently-occurring substrings by a new or an existing substring. As a result, while memory savings can be achieved for the dictionary, additional codewords are also introduced leading to index expansion. For example, consider the prefix strings "$COMPUTER$" and "$QUALCOM$" (see Figure 5). Again, for illustrative purposes, we consider letters as symbols. Since "COM" is a common substring, by storing it in the memory only once, the dictionary size can be reduced. However, one additional codeword is required for "$COMPUTER$" since it is decomposed in two substrings (i.e., "COM" and "$PUTER$"). In general, the problem of decomposing substrings that can result in maximum savings in memory has exponential complexity.

In the following, a 2-phase greedy heuristic is described that selectively decomposes substrings to achieve overall memory savings. A bottom-up approach is used that prunes the suffix trees starting from the leaf nodes and replaces deleted suffix strings by new (or existing) prefix strings. We concentrate only on suffix strings that include nodes pointed at by only one suffix string. Otherwise, the suffix string extends over large number of prefix strings resulting in lower possibility for potential savings in memory. Using our heuristic, $80 - 85\%$ of the nodes in all suffix trees were examined for the bit-streams considered in our experiments (see Section 5).

In the first phase, we delete suffix strings that can lead to potential savings in memory (see Algorithm 2). Initially, we identify repeated suffix strings that appear across all the suffix trees of the dictionary. As mentioned earlier, the number of suffix trees in the dictionary equals the number of symbols of the input alphabet. For each distinct suffix string s_i, the potential savings in memory $cost(s_i)$ are computed. The $cost(s_i)$ depends on the potential savings in dictionary memory and the potential index expansion assuming that s_i is deleted from all the suffix trees. Only suffix strings s_i with non-negative $cost(s_i)$ are deleted. By reducing the dictionary size, the number of bits that is required to address the dictionary (i.e., $\lceil \log_2 n_{dictionary} \rceil$) can decrease too. As a result, the word-length of both the dictionary and index

Algorithm 2: Our Heuristic: Phase 1.

Input: A dictionary D_{in} and an index I_{in}.
Output: Enhanced dictionary D_{temp} and index I_{temp}.

$STRINGS$={suffix strings in D_{in} containing nodes that are pointed at by only one suffix string}
U={s_i: $s_i \epsilon STRINGS \wedge (if\ i \neq j \Rightarrow s_i \neq s_j)$}
U_l={s_i: $s_i \epsilon U \wedge length(s_i)$=l}
/* $L = \max length(s_i)$
/* $data_{dictionary}$: word-length for the dictionary memory
/* $data_{index}$: word-length for the index memory
/* n_i: node of s_i with the highest distance from a leaf node
/* t_i: # of $x \epsilon STRINGS : x = s_i$
/* c_i: # of times n_i is referenced by the index
if $\exists$ prefix string $x \epsilon D_{in} : x = s_i$
  $a_i = 0$
else
  $a_i = 1$
end
$cost(s_i) = (t_i - a_i) * (data_{dictionary}) - c_i * data_{index}$
$S_{delete} = NULL$
for $l = 1..L$
  $S_{temp} = NULL$
  $\forall s_i : s_i \epsilon \{U_l \cup U\}$
    if $cost(s_i) \geq 0$
      $S_{delete} = S_{delete} \cup \{s_i\}$
    else
      $S_{temp} = S_{temp} \cup U_l \cup \{x \epsilon STRINGS : s_i \sqsupset x\}$
    end
  $U = U - S_{temp}$
end
delete $\{x \epsilon STRINGS : x = y \wedge y \epsilon S_{delete}\}$
S_{new} = {new prefix strings that replace the deleted substrings}
$D_{temp} = D_{in}$ − {deleted substrings} $\cup S_{new}$
I_{temp} = {restore I_{in} due to deleted substrings}

Algorithm 3: Our Heuristic: Phase 2.

Input: D_{temp} and I_{temp} from Algorirthm 2.
Output: Enhanced dictionary D_{enh} and index I_{enh}.

$N = \{n_i : n_i \epsilon s_i \wedge s_i \epsilon \{D_{temp} \cap STRINGS\}\}$
/* $STRINGS$ is the same set of strings as in Algorithm 2
/* n_i: dictionary node
$cost(n_i)$ = # of times n_i is referenced by the index
$depth(n_i)$ = distance from a leaf node
sort N in terms of $depth(n_i)$ /* ascending order
sort n_i of same *depth* in terms of $cost(n_i)$ /* ascending order
$N_m = NULL$
$n_{temp} = \mid D_{temp} \mid - 2^{\lceil \log_2 \mid D_{temp} \mid \rceil - 1}$ /* $\mid * \mid$= # of nodes in $*$
while $\mid N \mid \geq n_{temp}$
  repeat
    mark consecutive nodes in N
    with respect to sorting
    N_m = {marked nodes}
  until (# of marked nodes $- \sum cost(n_i) - \alpha == n_{temp}$)
/* $\sum cost(n_i)$: summation of costs of the marked nodes
/* α: # of nodes required to replace suffix strings that
/* will be deleted if marked nodes are deleted
  if (deletion of marked nodes results in overall savings)
    $N = N - N_m$
    $\mid D_{temp} \mid \leftarrow 2^{\lceil \log_2 \mid D_{temp} \mid \rceil - 1}$
    $n_{temp} \leftarrow 2^{\lceil \log_2 \mid D_{temp} \mid \rceil - 1}$
  else
    BREAK
  end
end
delete {marked nodes}
S_{new} ={new prefix strings that replace the deleted suffix strings}
$D_{enh} = D_{temp}$ − {marked nodes} $\cup S_{new}$
I_{enh} = {restore I_{temp} due to deleted substrings}

memories can decrease resulting in further savings in memory.

In the second phase, we selectively delete individual nodes of the suffix trees in order to decrease the number of bits required to address the dictionary (see Algorithm 3). The deletion of nodes results in index expansion. However, the memory requirements due to the increase of index size can be potentially amortized by the decrease of the word-length of both the dictionary and the index memories. The goal is to reduce the dictionary size while introducing minimum number of new codewords. Initially, nodes n_i of the same distance across all the suffix trees are sorted with respect to the number of codeword splits $cost(n_i)$ (i.e., number of new codewords introduced if the node will be deleted). Then, starting from the leaf nodes, we mark individual nodes according to their $cost(n_i)$. A marked node is eligible to be deleted. Nodes with smaller number of codeword splits are marked first. We continue to mark nodes until we achieve a 1 bit savings in addressing the dictionary. If the index expansion results in increasing the total memory requirements, the marked nodes are not deleted and the procedure is terminated. Otherwise, the marked nodes are deleted and the procedure is repeated.

4.4 Configuration Decompression

Decompression occurs at power-up or at runtime. The original configuration bit-stream is reconstructed by parsing the dictionary with respect to the index. As shown in Figure 6(b), the contents of the index (i.e., codewords) are read sequentially. A codeword corresponds to an address to the dictionary memory. For each codeword, all the symbols of the associated string are read from the dictionary memory and then the next codeword is read. A comparator is used to decide if the output data of the dictionary memory corresponds to a root node, that is, all the symbols of a string have been read. Depending on the output of the comparator, a new codeword is read or the last-read pointer is used to address the dictionary memory.

In Figure 6, both a typical scheme and our compression-based scheme for storing and reading the configuration bit-stream are shown. Typically, the configuration bit-stream is stored in memory. It is important to deliver the bit-stream sequentially otherwise the configuration mechanism will not be initialized correctly and the configuration process will fail. Depending on the configuration mode, data is delivered serially or in parallel. In our scheme, the only hardware overhead introduced is a comparator and a multiplexer. The output of the decompression process is

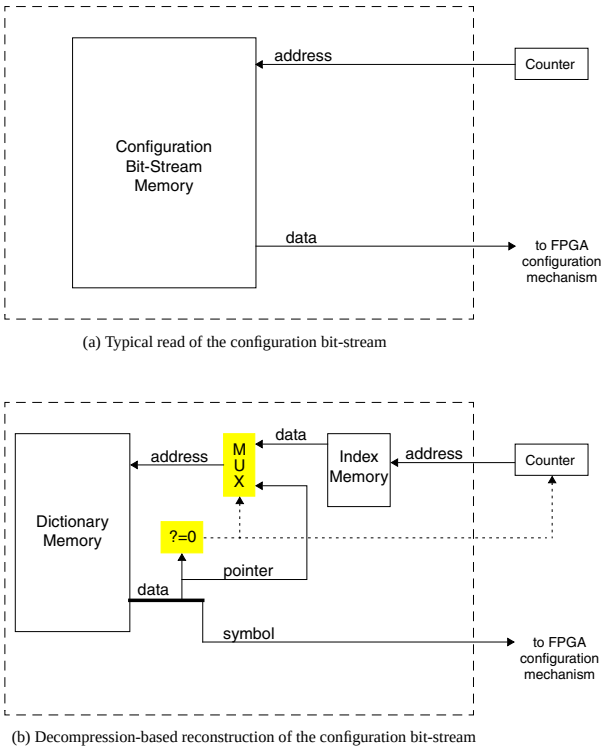

Figure 6: Our configuration decompression approach

identical to the data delivered by the conventional scheme. Moreover, the data rate for delivering the configuration data is the same for both the schemes and depends only on the memory bandwidth. The decompression process does not add any time overhead to the configuration time.

5. EXPERIMENTS & COMPRESSION RESULTS

Our configuration compression technique was applied to configuration bit-streams of several real-world applications. The target architecture was the VIRTEX series FPGAs [14]. For mapping onto the VIRTEX devices, we used the Foundation Series v2.1i software development tool. Each application was mapped onto the smallest VIRTEX device that met the area requirements of the corresponding implementation. The size of the configuration bit-streams ranged from 1.7 Mbits to 6.1 Mbits. In Table 1, the configuration bit-stream sizes for each implementation are shown.

The considered configuration bit-streams corresponded to implementations of cryptographic and signal processing algorithms. The cryptographic algorithms were the final candidates of the Advanced Encryption Standard (AES): $MARS$, $RC6$, $Rijndael$, $Serpent$, and $Twofish$. Their implementations included a key-scheduling unit, a control unit, and one round of the cryptographic core that was used iteratively. Implementation details of the AES algorithms can be found in [5]. We have also implemented digital signal processing algorithms using the logic cores provided with the Foundation 2.1i software tool [14]. A $1024-$ and a $512-$point complex FFT were implemented that were able to perform $IFFT$ too. In addition, four $256-$tap FIR filters were mapped onto the same device. In the latter implementation, all filters can process data concurrently. Finally, a $1024-$tap FIR filter was also implemented.

The configuration bit-streams were processed $byte-by-byte$ during compression, that is, the symbol for the dictionary entries was chosen to be an 8-bit word. As a result, the decompressed data is delivered as 8-bit words and, thus, parallel modes of configuration can be supported. Note that the maximum number of bits used in parallel modes of configuration is typically 8 bits [1, 2, 14]. If the configuration mode requires less than 8 bits (e.g., serial mode), an $8-to-n$ bit converter can be used, where n is the number of bits required by the configuration mode. In this work, for each configuration bit-stream, we do not attempt to find the optimal bit-length for the symbol that leads to the best compression results.

The compression results are shown in Tables 1 and 2. The results are organized with respect to the optimization stages of our technique (see Figure 2). The results shown for LZW correspond to the construction of the dictionary and the index using the LZW algorithm. The only difference compared to Figure 2 is that the LZW results include the optimization of merging common prefix strings in the dictionary. Hence, the results shown for $Compact$ correspond to the deletion of the non-referenced nodes in the dictionary. Finally, the results shown for $Heuristic$ correspond to the optimizations performed by our heuristic and are also the overall results of our compression technique.

In Table 1, the achieved compression ratios are shown. The compression ratio is the ratio of the total memory requirements (i.e., memory to store dictionary and index) to the bit-stream size. In addition, in Table 1, lower bounds on the compression ratios are shown. For our compression technique, the lower bound for each bit-stream corresponds to the entropy of the bit-stream with respect to the LZW compression algorithm. As mentioned in Section 3, the compression ratio is affected by the entropy of the data to be compressed [11]. We have calculated the lower bound by dividing the index size derived using LZW by the bit-stream size. Therefore, the lower bound corresponded to the compression ratio that can be achieved by LZW for software-based applications (assuming $8-bit$ symbols).

In Table 2, the compression results are shown in terms of the memory requirements. The memory requirements for the dictionary are $n_{dictionary} \times (8 + \lceil \log_2 n_{dictionary} \rceil)$ bits, where $n_{dictionary}$ is the number of memory entries of the dictionary. Similarly, the memory requirements for the index are $n_{index} \times \lceil \log_2 n_{dictionary} \rceil$ bits, where n_{index} is the number of memory entries of the index and $\lceil \log_2 n_{dictionary} \rceil$ is the number of bits required to address the dictionary.

LZW In software-based applications, only the index is considered in the calculation of the compression ratio. In addition, statistical encoding schemes are utilized for further compressing the index. As a result, in typical LZW applications, superior compression ratios (i.e., $10-20$ %) have been achieved by using commercially available software programs (e.g., *compress*, *gzip*). However, such commercial programs are not applicable to our compression problem. As discussed earlier, in the context of embedded environments, both the dictionary and the index are considered in the calculation of the compression ratio. The size of the derived dictionaries was comparable to the size of the original bit-streams. Therefore, negative compression occurred,

Table 1: Compression ratios for the configuration bit-streams

Implementation	Bit-stream size (bits)	Compression ratio			
		LZW	Compact	Heuristic	Lower Bound
MARS	3608000	179 %	96 %	82 %	73 %
RC6	2546080	119 %	69 %	59 %	48 %
Rijndael	3608000	198 %	104 %	89 %	81 %
Serpent	2546080	165 %	95 %	79 %	67 %
Twofish	6127712	186 %	103 %	86 %	76 %
FFT-256	1751810	140 %	85 %	68 %	56 %
FFT-1024	1751840	159 %	89 %	72 %	64 %
4 x FIR-256	1751840	180 %	97 %	80 %	73 %
FIR-1024	1751840	177 %	96 %	79 %	71 %

Table 2: Dictionary and Index memory requirements.

Implementation	Dictionary memory requirements & word-length (bits)						Index memory requirements & word-length (bits)					
	LZW		Compact		Heuristic		LZW		Compact		Heuristic	
MARS	3827070	26	1116912	24	172032	21	2644920	18	2351040	16	2968618	13
RC6	1811575	25	667575	23	172032	21	1227536	17	1083120	15	1516596	13
Rijndael	4231448	26	1149840	24	172032	21	2924874	18	2599888	16	3227981	13
Serpent	2511275	25	826152	24	172032	21	1703332	17	1603136	16	2017720	13
Twofish	6746558	26	1919550	25	360448	22	4666104	18	4406876	17	5273846	14
FFT-256	1479408	24	564144	23	81920	20	982192	16	920805	15	1107396	12
FFT-1024	1657900	25	574034	23	81920	20	1123037	17	990915	15	1181964	12
4 x FIR-256	1883900	25	575897	23	81920	20	1276717	17	1126515	15	1330044	12
FIR-1024	1849725	25	580612	23	81920	20	1253478	17	1106010	15	1303416	12

that is, the memory requirements for the dictionary and the index were greater than the bit-stream size.

Compact By deleting the non-referenced nodes in the dictionary, the number of the dictionary entries was reduced by a factor of $2.4 - 3.4$. As a result, the number of bits required to address the dictionaries was also reduced by 1 to 2 bits affecting the word-length of both the dictionary and the index memories accordingly. Compared with the LZW results, the memory requirements for the dictionaries were reduced by a factor of $2.5 - 3.7$. In addition, the memory requirements for the indices were also reduced by $6 - 13$ % even though the number of codewords remained the same. Overall, the compression ratios achieved at this optimization stage were $69 - 104$ %.

Heuristic Finally, the overall savings in memory were further improved by our heuristic. The goal of our heuristic was to reduce the size of the dictionary at the expense of the index expansion. Indeed, compared to the *Compact* results, the dictionary entries were reduced by a factor of $2.9 - 6.2$ while the number of codewords was increased by $35 - 50$ %. The number of bits required to address the dictionary was reduced by 2 to 3 bits affecting the word-length of both the dictionary and the index memories accordingly. As a result, even though the number of codewords was increased, the total memory requirements were reduced. Compared with the *Compact* results, the memory requirements of the dictionaries were further reduced by a factor of $3.2 - 7.1$ while the memory requirement of the indices were increased by $18 - 40$ %. Overall, the compression ratios achieved at this optimization stage were $59 - 89$ %. Our heuristic improved the compression ratios provided by the *Compact* results by $14 - 20$ %.

Considering the compression ratios achieved by LZW and the lower bounds on them, our compression technique performs well. The improvements over the LZW results were significant. On the average, our technique reduced the dictionary memory requirements by 94.5 % while the index memory requirements were increased by 11.5 %. As a result, our compression results were close to the lower bounds. On the average, our compression ratios were higher than the lower bounds by 14.5 %. Overall, our compression technique reduced the memory requirements of the configuration bit-streams by $0.35 - 1.04$ Mbits. The savings in memory corresponded to $11 - 41$ % of the original bit-streams.

6. RELATED WORK

Various lossless compression techniques have been extensively studied in the literature. The majority of these techniques have been developed for software-based applications. However, in embedded systems, the implementation cost becomes a very significant issue. In [9, 10], dictionary-based compression techniques were utilized for code minimization in embedded processors.

In [10], a fixed-size dictionary was used for compressing programs. The technique was applied to several instruction sets (i.e., PowerPC, ARM, i386). The size of the programs

was in the order of hundreds of bits. No detailed information was provided regarding the algorithm used to build the dictionary. The authors mainly focused on tuning the dictionary parameters to achieve better compression results based on the specific set of programs. However, such an approach is unlikely to achieve the same results for FPGA configurations where the bit-stream is a data file and not an instruction-based program. In addition, Huffman encoding was used for compressing the codewords. As a result, dedicated hardware resources were needed for decompressing the codewords.

In [9], the dictionary was built by solving a set-covering problem. The dictionary representation was based on the External Pointer Macro compression model. According to this model, a phrase is called by pointing to a memory address and reading as many consecutive memory addresses as the phrase length. A heuristic was developed to merge dictionary entries by subsuming a dictionary entry i by another entry j if $i \sqsupset j$. While this heuristic results in a minimal-size dictionary (i.e., minimal number of entries), it also results in entries of maximal-length. This happens because the goal was to substitute maximal number of entries by keeping the ones that "cover" maximal number of them. Hence, the longest phrases were kept in the dictionary. Moreover, the size of the considered programs was 0.5-10 Kbits and the achieved compression ratios (i.e. size of the compressed program as fraction of the original program) were approximately 85-95 %. Since the technique in [9] was developed for code size minimization, it is not fair to make any compression ratio comparisons with our results.

Work related to FPGA configuration compression has been reported in [7, 8]. In [7], the proposed technique took advantage of the characteristics of the configuration mechanism of the Xilinx XC6200 architecture. Therefore, the technique is applicable only to that architecture. In [8], runlength compression techniques for configurations have been described. Again, the techniques were developed specifically for the Xilinx XC6200 architecture. Addresses were compressed using runlength encoding while data was compressed using LZ compression (sliding-window method [11]). Dedicated on-chip hardware was required for both methods. A set of configuration bit-streams ($2 - 88$ Kbits) were used to fine-tune the parameters of the proposed methods. A $16-$bit size window was used in the LZ implementation. While this window size led to good results for these bit-streams, it is impractical for larger configuration bit-streams. Moreover, a fine-tuned scheme for larger configuration bit-streams would lead to larger size windows. As stated in [8], larger size windows impose a fairly high hardware penalty with respect to the buffer size as well as the supporting hardware.

7. CONCLUSIONS

In this paper, a novel configuration compression technique was proposed. Our goal was to reduce the memory required to store configurations in FPGA-based embedded systems and achieve high decompression efficiency. Decompression efficiency corresponds to the decompression hardware cost as well as the decompression rate. Although data compression has been extensively studied in the past, we are not aware of any prior work that addresses configuration compression for FPGA-based embedded systems with respect to the cost and speed requirements. Our compression technique is applicable to any SRAM-based FPGA device since it does not depend on specific features of the configuration mechanism. The configuration bit-streams are processed as raw data without considering individual semantics. Hence, both complete and partial configuration schemes can be supported. The required decompression hardware is simple and does not depend on the individual semantics of configuration bit-streams or specific features of the configuration mechanism. Moreover, the decompression process does not affect the time to configure the device. Using our technique, we have demonstrated $11 - 41$ % savings in memory for various configuration bit-streams of real-world applications. Considering the lower bounds derived for the compression ratios, the achieved compression ratios were higher than the lower bounds by 14.5 % on the average.

Future work includes the enhancement of our technique by incorporating a $unified$-dictionary, that is, deriving a single dictionary for a set of configurations. The latter will result in simplified memory organization since the word-length of the index memory will be the same across different algorithms. Possible solutions could be to process all configuration bit-streams as one entity or to process each configuration bit-stream individually and merge their dictionaries later. For updating the embedded system with a new configuration, only the index is derived based on the $unified$ dictionary.

In addition, we plan to develop a $skeleton$-based approach for our compression technique. A $skeleton$ is the "intersection" of a set of configuration bit-streams. By removing the data redundancy of the $skeleton$ in the bit-streams, savings in memory can be achieved. The original configurations are reconstructed based on the $skeleton$. Given a set of configurations, we plan to address the problem of deriving a $skeleton$ to maximize the savings in memory and/or to minimize the configuration time by using partial reconfiguration.

Related problems are also addressed by the USC MAARCII project (http://maarcII.usc.edu). This project is developing novel mapping techniques to exploit dynamic reconfiguration and facilitate run-time mapping using configurable computing devices and architectures [3, 4, 12].

8. REFERENCES

[1] Altera PLD Devices, http://www.altera.com/html/products/about.html

[2] Atmel FPGA, http://www.atmel.com/atmel/products/prod3.htm

[3] K. Bondalapati and V. K. Prasanna, "Loop Pipelining and Optimization for Run Time Reconfiguration", Reconfigurable Architectures Workshop, May 2000.

[4] A. Dandalis, "Dynamic Logic Synthesis for Reconfigurable Devices", PhD Thesis, Dept. of Electrical Engineering, University of Southern California. Under Preparation.

[5] A. Dandalis, V. K. Prasanna, and J. D. P. Rolim, "A Comparative Study of Performance of AES Final Candidates Using FPGAs", Workshop on Cryptographic Hardware and Embedded Systems, August 2000.

[6] A. Dandalis, V. K. Prasanna, and J. D. P. Rolim, "An Adaptive Cryptographic Engine for IPSec Architectures", IEEE Symposium on Field-Programming Custom Computing Machines, April 2000.

[7] S. Hauck, Z. Li, and E. J. Schwabe, "Configuration Compression for the Xilinx XC6200 FPGA", IEEE

Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 18, No. 8, pp. 1107-1113, August, 1999.

[8] S. Hauck, W. D. Wilson, "Runlength Compression Techniques for FPGA Configurations", IEEE Symposium on Field-Programmable Custom Computing Machines, April 1999.

[9] S. Laio, S. Devadas, and K. Keutzer, "A Text-Compression-Based Method for Code Size Minimization in Embedded Systems", ACM Transactions on Design Automation of Electronic Systems, Vol. 4, No. 1, pp. 12-38, January 1999.

[10] C. Lefurgy, P. Bird, I-C. Cheng, and T. Mudge, "Improving Code Density Using Compression Techniques", 29th Annual IEEE/ACM Symposium on Microarchitecture, December 1997.

[11] M. Nelson, J-L. Gaily, "The Data Compression Book", M&T Books, New York, 1996.

[12] R. Sidhu, S. Wadhwa, A. Mei, and V. K. Prasanna, "A Self-Reconfigurable Gate Array Architecture", International Conference on Field Programmable Logic and Applications, September 2000.

[13] World Semiconductor Trade Statistics Organization, http://www.wsts.org

[14] Xilinx Virtex Series FPGAs, http://www.xilinx.com/products/virtex.htm