Configuration Compression for FPGA-based Embedded Systems[1]

Andreas Dandalis[2] and Viktor K. Prasanna[3]

Abstract

Field Programmable Gate Arrays (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 the decompression rate scales with the speed of the memory used for storing the FPGA configuration. 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.

I. 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, multi-functional equipment, home appliances, etc. 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 Field Programmable Gate Arrays (FPGAs) a highly promising solution for embedded systems [8], [11], [12], [13], [18], [22], [23], [25]. Such systems will significantly reduce design risks and increase the production

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

[2]A. Dandalis is with the Intel Corporation, Hillsboro, USA, andreas.dandalis@intel.com.

[3]V. K. Prasanna is with the University of Southern California, Los Angeles, USA, prasanna@usc.edu.

volume of the chip. Additionally, the requirements of applications in the information-based networked world are changing so rapidly that reconfigurability is possibly the only way to overcome severe time-to-market requirements.

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. Moreover, by incorporating Run-Time Reconfiguration (RTR), applications consisting of multiple configurations can be executed [3], [6], [7], [17], [20], [21]. 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-II series FPGAs, range from 0.4 Mbits to 43 Mbits [24].

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, increased configuration data density can be achieved resulting in smaller size memories and/or increased system flexibility. Smaller size memories and increased system flexibility are essential to enhance the cost-effectiveness and lifetime of the embedded systems under consideration. Moreover, increased flexibility can enable the usage of smaller FPGA devices in environments that require on-demand computations. A smaller FPGA device that is reconfigured on-demand is more cost-effective compared with a larger FPGA device that is configured once for all the possible tasks that can be executed during an application.

Using our compression technique, 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 configuration 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 and can support both complete and partial configuration schemes. The proposed technique process configuration bit-streams as raw data without considering any individual semantics. Such semantics depend on the target FPGA device architecture and the placement/routing tool used for application mapping. On the other hand, application-

specific computations can lead to hardware structural regularities. Such regularities are explored by our technique as repeated strings at the bit-stream level.

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. In addition, a *unified*-dictionary approach is proposed for compressing sets of configurations. Even though statistical methods can achieve higher compression ratios, we propose a dictionary-based approach because statistical methods lead to high decompression hardware cost. Using our technique, we demonstrate $11-41\%$ 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 the VIRTEX series FPGAs [24]. The latter choice has been made for demonstration purposes only. Single as well as sets of configuration bit-streams were compressed using our technique. 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. In Section 4, related work is described. Our novel compression technique is described in Section 5 and experimental results are demonstrated in Section 6. Finally, in Section 7, possible extensions to our work are described.

II. 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 0.4 Mbits to 44 Mbits [1], [2], [24] 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 for FPGAs can be as high as 480 Mbits/sec (i.e., 8 bits $\times$ 60 MHz) [1], [2], [24] while some programmable logic cores support even higher data rates. 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. Such an increase 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.

III. 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. 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 [19]. 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 [19]. 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. Therefore, in the context of FPGA-based embedded systems, a dictionary-based scheme would result in fairly low implementation cost.

In Figure III, 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

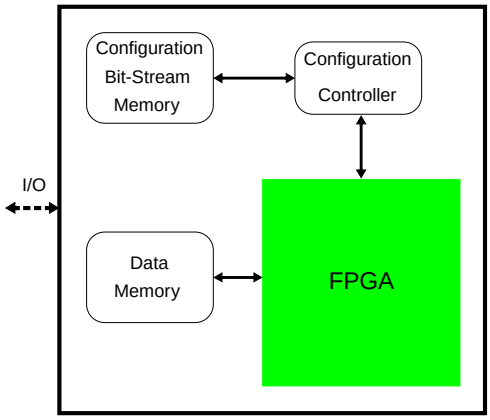


Fig. 1: FPGA-based embedded system architecture

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. In this work, embedded systems that also include a microprocessor can be considered under the assumption that the microprocessor is not used for compression/decompression (e.g., due to high utilization by other tasks). Besides the memory requirements for the compressed data, additional memory may be required during decompression (e.g., storing temporary data). However, in the context of embedded systems, the memory requirements to store temporary data 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.

Our compression technique can lead to smaller memory requirements for storing FPGA data and thus, reducing the cost of the configuration memory (e.g., configuration-specific system). At the same time, the power requirements can be reduced since memories of smaller size can be used. In addition, the tight coupling of configuration memory and FPGA devices can result in systems with superior flexibility (e.g., system with fixed-size configuration memory). Such flexibility can enable the usage of smaller FPGA devices in environments that require on-demand computations.

IV. Related Work

Work related to FPGA configuration compression has been reported in [9], [10], [16]. In [9], 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 [10], runlength compression techniques for configurations have been described. Again, the techniques took advantage of specific characteristics of the Xilinx XC6200 architecture. Addresses were compressed using runlength encoding while data was compressed using LZ compression (sliding-window method [19]). 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. However, as stated in [10], larger size windows impose a fairly high hardware penalty with respect to the buffer size as well as the supporting hardware. In [16], dictionary-based techniques were developed to reduce the time required to transfer configuration data to VIRTEX series FPGAs. A compressed version of the configuration bit-stream is fed to the configuration circuitry of the FPGA and decompression takes place inside the FPGA. A modified VIRTEX configuration mechanism was proposed to support decompression. High compression ratios were reported. However, the time overhead for decompressing the configuration data was not clearly identified.

In [14], [15], dictionary-based compression techniques were utilized for code minimization in embedded processors. However, code minimization takes advantage of the semantics of programs for Instruction Set Architecture (ISA) based processors and is unlikely to achieve similar results for FPGA configuration bit-streams (i.e., raw data). For example, programs can have jumps that require decompression to be performed not in a sequential manner while configuration bit-streams should be decompressed sequentially. In [15], a fixed-size dictionary was used for compressing programs. 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 a program for ISA-based processors. In addition, Huffman encoding was used for compressing the codewords. As a result, dedicated hardware resources were needed for decompressing the codewords. In [14], the dictionary was built by solving a set-covering problem. The underlying compression model was developed with respect to the semantics of programs for ISA-based processors (i.e., control-flow and operational instructions).

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 [14] was developed for code size minimization, it is not fair to make any compression ratio comparisons with our results.

V. Our Compression Technique

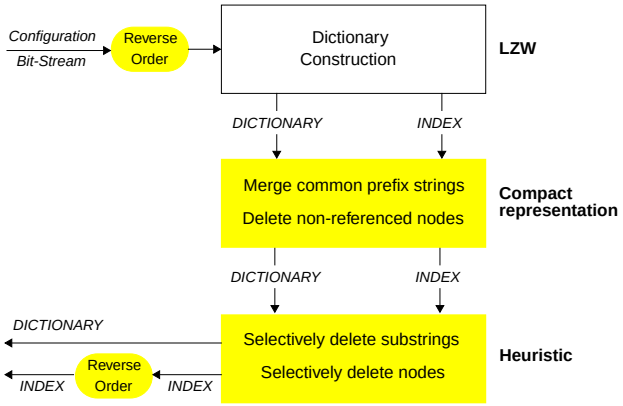


Fig. 2: Our configuration compression technique

Our compression technique is based on the principles of dictionary-based compression algorithms. Compression occurs off-line while decompression occurs on-line to reconstruct the original configuration bit-stream. Even though statistical methods can achieve higher compression ratios [19], 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 [19]. In general, finding a dictionary that results in optimal compression has exponential complexity [19]. 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.

A. 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 (Algorithm 1). 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.

Algorithm 1: The LZW algorithm [20]

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)

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”. 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.

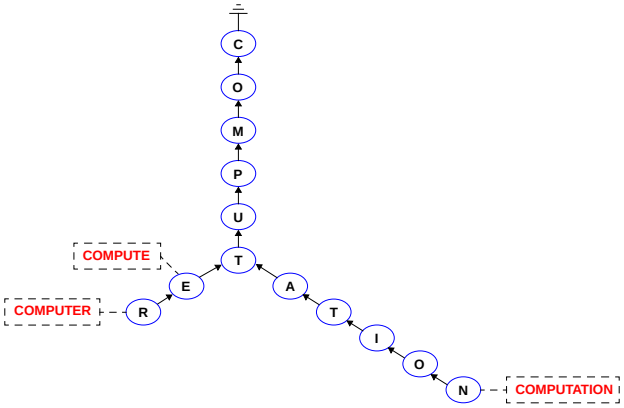


Fig. 3: An illustrative example of our dictionary representation

B. 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. 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 contents of the dictionary entries are shown in ascending order of their memory address. For each dictionary entry, the corresponding symbol and the address to a prefix string are shown. The shown index entries correspond to a dictionary memory address. 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 [19], a stack is used to deliver each decompressed

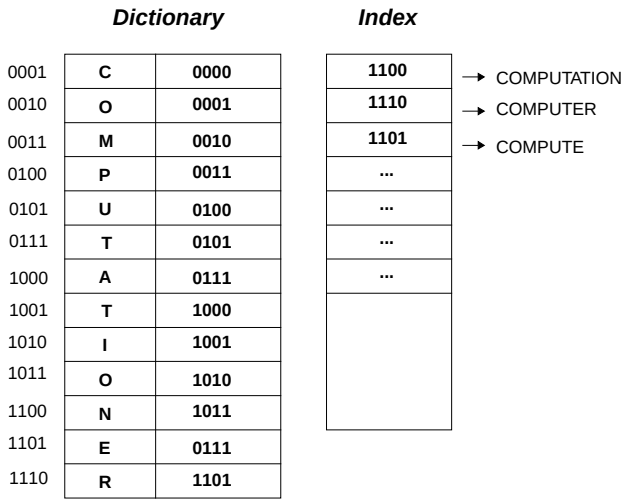


Fig. 4: An illustrative example of memory organization for the dictionary and the index

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 $\pm 1\%$).

C. 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.

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

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 suffix strings}
$D_{temp} = D_{in}$− {deleted substrings} $\cup S_{new}$
I_{temp} = {restore I_{in} due to deleted substrings}

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

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
$N_m = NULL$
$n_{temp} =\mid D_{temp} \mid -2^{\lceil \log_2 |D_{temp}| \rceil - 1}$ /* $\mid * \mid =$ # 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)
      $N = N - N_m$
      $\mid D_{temp} \mid \leftarrow 2^{\lceil \log_2 |D_{temp}| \rceil - 1}$
      $n_{temp} \leftarrow 2^{\lceil \log_2 |D_{temp}| \rceil - 1}$
    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$\}$

our heuristic, $80 - 85\%$ of the nodes in all suffix trees were examined for the bit-streams considered in our experiments (see Section VI).

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

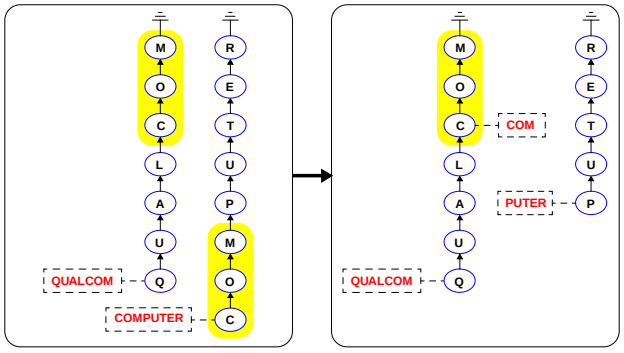


Fig. 5: An illustrative example of enhancing the dictionary representation

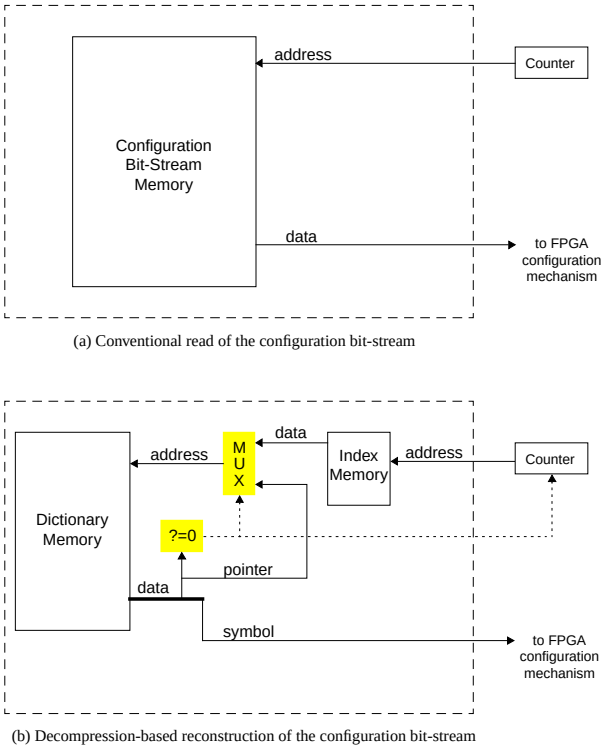
(i.e., $\lceil \log_2 n_{dictionary} \rceil$) can decrease too. As a result, the word-length of both the dictionary and index memories can decrease resulting in further savings in memory.

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.

D. 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.

As a result, the decompression rate scales with the speed of the memory used for storing the dictionary.



(a) Conventional read of the configuration bit-stream

(b) Decompression-based reconstruction of the configuration bit-stream

Fig. 6: Our configuration decompression approach

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 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. Moreover, the hardware cost is minimal compared with the conventional scheme.

VI. 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 [24]. VIRTEX FPGAs have been chosen for demonstration purposes only. For mapping onto the VIRTEX devices, we used the Foundation

TABLE I: Compression ratios for single configurations

Configuration	*Bit-stream size (bits)*	*Compression ratio*			
		LZW	*Compact*	*Heuristic*	*LZW 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	1751840	140 %	85 %	68 %	56 %
FFT-1024	1751840	159 %	89 %	72 %	64 %
4 $\times$ FIR-256	1751840	180 %	97 %	80 %	73 %
FIR-1024	1751840	177 %	96 %	79 %	71 %

TABLE II: Dictionary and Index memory requirements for single configurations

Configuration	*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	2796586	13
RC6	1811575	25	667575	23	172032	21	1227536	17	1083120	15	1344564	13
Rijndael	4231448	26	1149840	24	172032	21	2924874	18	2599888	16	3055949	13
Serpent	2511275	25	826152	24	172032	21	1703332	17	1603136	16	1845688	13
Twofish	6746558	26	1919550	25	360448	22	4666104	18	4406876	17	4913398	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 $\times$ FIR-256	1883900	25	575897	23	81920	20	1276717	17	1126515	17	1330044	12
FIR-1024	1849725	25	580612	23	81920	20	1253478	17	1106010	15	1303416	12

Series v2.1i software development tool. Each application was mapped onto the smallest VIRTEX device that met the area requirements of the corresponding implementation. We have purposely used the smallest possible array in order to achieve high hardware utilization and avoid trivial cases such as large sequences of zeros. Our results indicated that such cases were indeed avoided and no ”magic” words were found. The derived suffix trees were very flat structures of small height. All bit-streams used corresponded to complete configurations. However, this is not a limitation of our approach since it can also handle partial configuration bit-streams. The size of the configuration bit-streams ranged from 1.7 Mbits to 6.1 Mbits. In Table I, 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 [24].

TABLE III: Compression ratios for sets of configurations

Configurations	*Bit-streams size (bits)*	*Compression ratio*			
		LZW	*Compact*	*Heuristic*	*Baseline*
MARS, Rijndael	2 × 3608000	181 %	97 %	85 %	85.50 %
RC6, Serpent	2 × 2546080	136 %	76 %	68 %	69.00 %
FFT-256, FFT-1024, 4 × FIR-256, FIR-1024	4 × 1751840	142 %	84 %	71 %	74.75 %

TABLE IV: Dictionary and Index memory requirements for sets of configurations

Configurations	*Dictionary memory requirements & word-length (bits)*						*Index memory requirements & word-length (bits)*					
	LZW		*Compact*		*Heuristic*		*LZW*		*Compact*		*Heuristic*	
MARS, Rijndael	7676856	27	2193250	25	360448	22	5397406	19	4829258	17	5772508	14
RC6, Serpent	4092062	26	1376568	24	360448	22	2828394	18	2514128	16	3095974	14
FFT-256, FFT-1024, 4 × FIR-256, FIR-1024	5889468	26	2063875	25	360448	22	4072788	18	3846522	17	4648700	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 this 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], [24]. 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. Note also that our heuristic can be applied for any symbol length. However, 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. Our goal is to optimize a given dictionary structure regardless the symbol length.

A. Single Configurations

The compression results for single configurations are shown in Tables I and II. 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 with 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 I, 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 I, 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 [19]. The critical metric is the entropy of the LZW model that allows comparing the performance of our heuristic with respect to the model used to derive the dictionary. As a result, the IID model is misleading for this case since it considers all symbols as independent data while LZW model is based on strings of symbols. 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 II, 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, 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. Given a fixed-size configuration memory, memory savings enhance the flexibility of a system since more bit-streams can fit in the configuration memory. On the other hand, in the case of designing an application-specific system with specific configurations, the savings in memory are related to the cost of the system. The absolute value for savings in memory will determine the smallest memory size that can be used. Inevitably, the memory size availability also affect the overall cost savings.

B. Sets of Configurations

Our technique can be extended to compress a set of configurations by incorporating a *unified*-dictionary approach. The proposed approach differs from our configuration compression technique (see Figure 2) only with respect to the way the dictionary is constructed. Instead of constructing multiple dictionaries by processing the configuration bit-streams independently, the bit-streams are processed in a sequence

by sharing the same dictionary. The LZW algorithm (see Algorithm 1) is applied to each configuration bit-stream without initializing the dictionary. Every time LZW is called, it uses the dictionary that was derived by the preceding call. The derived indices are grouped in one index for facilitating the processing through the remaining stages of our compression technique (see Figure 2).

The goal of the *unified*-dictionary approach is to construct a single dictionary for multiple configurations in order that the world-length of the index memory will be the same across different configurations. As a result, a simple memory organization will be required for decompression, which is identical to the one shown in Figure 6(b). On the contrary, if the configuration bit-streams are processed independently (*baseline*[4]), a more complex memory organization will be required that consists of multiple memory modules of various word-lengths. Furthermore, if the dictionaries obtained by the *baseline* approach are grouped to form a single dictionary, the compression ratio would increase due to the increase in the number of bits required to address the dictionary entries.

In Tables III and IV, the achieved compression ratios and the dictionary and index memory requirements are shown. Configurations corresponding to the same FPGA device are grouped together since it is uncommon for FPGA systems to utilize different FPGA devices as computing nodes (except FPGA devices that are used for control). Clearly, besides resulting in simple memory organization, the proposed approach achieves better compression ratios than the *baseline* approach. This happens because the increase of the number of bits required to address the dictionary entries is amortized by the decrease of the number of index entries. The number of index entries decreases due to the fact that, after the first call to LZW, the dictionary is not initialized with the alphabet symbols but, it already contains some strings. Therefore, for larger number of configuration bit-streams, a larger decrease in the number of index entries is expected. Compared with the *baseline* approach, for $\{$*MARS, Rijndael*$\}$, $\{$*RC6, Serpent*$\}$, and $\{$*FFT-256, FFT-1024, 4* $\times$ *FIR-256*$\}$, the number of index entries decreases by 8.41%, 9.89%, and 19.06% respectively. Before applying our heuristic (see Figure 2), the number of dictionary entries is decreased by $7 - 17\%$ compared with the *baseline* approach. This happens because common entries among different dictionaries are replaced by a single entry in the *unified* dictionary. However, after applying our heuristic, the number of dictionary entries is the same for both the approaches.

[4]For comparison purposes, in the remainder of this section, the solution to processing the configuration bit-streams independently is referred as *baseline*.

VII. 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. As a result, 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 and the decompression rate scales with the speed of the memory used for storing the dictionary. 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. In addition, a *unified*-dictionary approach was proposed for compressing sets of configurations. Such an approach achieves better compression ratios than compressing the configurations independently while leading to a simple memory organization that does not require multiple memory modules of different word-length.

Future work includes the development of a $skeleton$-based approach for our compression technique. A $skeleton$ corresponds to the correlation among a set of configuration bit-streams. By removing the data redundancy of the $skeleton$ in the bit-streams, savings in memory can be achieved. Given a set of configurations, we plan to address the problem of deriving a $skeleton$ in order to reduce the size of individual indices. Related problems are addressed in [4].

The work reported here is part of 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.

REFERENCES

[1] Altera PLD Devices, `http://www.altera.com`

[2] Atmel FPGA, `http://www.atmel.com`

[3] K. Bondalapati, P. Diniz, P. Duncan, J. Granacki, M. Hall, R. Jain, and H. Ziegler, *DEFACTO: A Design Environment for Adaptive Computing Technology*, Reconfigurable Architectures Workshop, April 1999.

[4] A. Dandalis, *Dynamic Logic Synthesis for Reconfigurable Devices*, PhD Thesis, Dept. of Electrical Engineering, University of Southern California, December 2001.

[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] O. Diessel and H. ElGindy, *On dynamic task scheduling for FPGA-based systems*, International Journal of Foundations of Computer Science, Special Issue on Scheduling: Theory and Applications, 12(5), pp. 645 - 669, October 2001.

[7] J.G. Eldredge and B.L. Hutchings, *Run-Time Reconfiguration: A Method for Enhancing the Functional Density of SRAM-Based FPGAs*, Journal of VLSI Signal Processing, Vol. 12(1), pp. 67-86, January 1996.

[8] R. Hartenstein, J. Becker, M. Herz, and U. Nageldinger, *An Embedded Accelerator for Real World Computing*, IFIP International Conference on Very Large Scale Integration, August 1997.

[9] 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.

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

[11] R. D. Hudson, D. I. Lehn, and P. Athanas, *A Run-Time Reconfigurable Engine for Image Interpolation*, IEEE Symposium on Field-Programmable Custom Computing Machines, April 1998.

[12] H. Kim, A. K. Somani, and A. Tyagi, *A Reconfigurable Multi-function Computing Cache Architecture*, IEEE Transactions on Very Large Scale Integration Systems, Volume 9(4), pp. 509-523, August 2001.

[13] M. Klimesh, V. Stanton, and D. Watola, *Hardware Implementation of a Lossless Image Compression Algorithm Using a Field Programmable Gate Array*, The Telecommunications and Mission Operations Progress Report, Jet Propulsion Laboratory, California Institute of Technology, February 2001.

[14] 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.

[15] 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.

[16] Z. Li and S. Hauck, *Configuration Compression for Virtex FPGAs*, IEEE Symposium on Field-Programmable Custom Computing Machines, April 2001.

[17] R. Maestre, F.J. Kurdahi, N. Bagerzadeh, H. Singh, R. Hermida, and M. Fernandez, *Kernel Scheduling in Reconfigurable Computing*, Design, Automation and Test in Europe Conference, March 1999.

[18] J. T. McHenry, P. W. Dowd, F. A. Pellegrino, T. M. Carrozzi, and W. B. Cocks, *An FPGA-based coprocessor for ATM firewalls*, IEEE Symposium on Field-Programmable Custom Computing Machines, April 1997.

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

[20] S. Ogrenci-Memik, E. Bozorgzadeh, R. Kastner, and M. Sarrafzadeh, *A Super-Scheduler for Embedded Reconfigurable Systems*, International Conference on Computer-Aided Design, November 2001.

[21] N. Shirazi, W. Luk, and P.Y.K. Cheung, *Framework and Tools for Run-Time Reconfigurable Designs*, IEE Proceedings Computers and Digital Techniques, Vol. 147, No. 3, pp. 147 - 152, May 2000.

[22] S. Swanchara, S. Harper, and P. Athanas, *A Stream-Based Configurable Computing Radio Testbed*, IEEE Symposium on Field-Programmable Custom Computing Machines, April 1998.

[23] J. Villasenor and W. H. Mangione-Smith, *Configurable Computing*, Scientific American, pp. 66-71, June 1997.

[24] Xilinx FPGA Devices, `http://www.xilinx.com`

[25] Xilinx Success Products Stories, `http://www.xilinx.com/company/success/csprod.htm`