Design and simulation of a new approach for reconfiguration overhead reduction in RTR systems

Mahmood Fathy¹, Mehdi Dehghan¹,², Farshid Farshadjam¹

1- Department of Computer Engineering, Iran University of Science and Technology, Tehran, Iran
2- Iran Telecom Research Center, Tehran, Iran

Abstract
The advantage of RTR systems usually comes with some costs. Necessary time for mapping some areas of a program to FPGA is considerable and affects the performance of RTR systems. Configuration compression can reduce the total number of write operations to load a configuration and it has been proved as an efficient technique to deal with the configuration overhead. In this paper, we have developed a new approach for reconfiguration overhead reduction in Xilinx Virtex FPGAs by using compression technique. Since the order of sequence of configuration frames affects the compression rate, we have used genetic algorithm for finding an optimal configuration sequence of frames.

1. Introduction
FPGAs are usually used as flexible hardware for execution of applications that require high-speed computation in RTR systems. They must be reconfigured frequently during run-time of an application [1]. But this process imposes some overhead to RTR systems. So, by reducing the reconfiguration overhead, the performance of the system is improved. For this reason, some tactics such as configuration Pre-fetching, configuration caching and configuration compression have been developed [2]. Configuration compression has shown good results among other techniques and reduces required information for configuration.

The major objectives of our work are:
1) configuration compression for reducing the configuration information.
2) development a GA approach for finding an optimal configuration sequence of frames to reduce the reconfiguration overhead.

Our algorithm targeted to Xilinx Virtex series FPGAs.

2. Xilinx Virtex Architecture
Virtex is one of the latest products of Xilinx Corporation and its configuration capacity (sea of gates) is very high [3]. Virtex configuration memory can be visualized as a rectangular array of bits. The bits are grouped into vertical frames, which are one-bit wide and extend from the top of the array to the bottom. A frame is the atomic unit of configuration. Virtex configuration is done by Frame Data Register (FDR). FDR is a shift register into which data is loaded prior to transfer to the configuration memory. Also read-back feature of virtex allows the previous data frame to be read back to FDR (see fig.1).

![Figure 1. Virtex architecture.](image)

3. Configuration Compression
Since the process of configuration of FPGA series is different, compression technique must be based on FPGA architecture. On the other hand, the role of each bit of configuration information in describing a part of hardware dictates compression technique to be loss-less. Well-known techniques including Huffman and arithmetic coding are very efficient for general-purpose compression such as text compression. The problem with these schemes lies in the fact that their decompression process is slow. Also, those will cause additional overheads by transferring and keeping a significant amount of probability values.

Usually in a hardware circuit, there are many similar gates (such as AND, OR, DECODER, ADDER and so on), so configuration frames of them would be similar. We can take advantage of the similarities of configuration frames in the compression phase. Lempel-Ziv is a dictionary-based compression technique that uses regularities between frames very well [4].

There are variations of LZ compression algorithms. Some of them required the excessive amount of hardware resources for retaining the table of similar patterns (dictionary) during decompression process. However, the LZSS compression algorithm only uses a sliding window buffer as its dictionary and the
shift based FDR fits the scheme naturally. This algorithm tracks the last \( n \) symbols of data previously seen, where \( n \) is the size of dictionary with serial input. If the matching length (of the sequences of configuration frames and the dictionary entries) is shorter than a given threshold, only the current symbol will be outputted. In the case that the matching length is longer than the threshold, the output codeword will consist of the index pointer and the length of the matching. In addition, in order to achieve correct decompression a flag bit is required for each code word to distinguish these two cases.

3.1. Hardware Variations

As shown in figure 2(a), the original FDR in Virtex devices can be used as the sliding window buffer and our compression algorithm can take advantage of the intra-frame regularity naturally. However, since the current FDR can only contain one frame of configuration data, using it as the sliding window buffer will not fully take advantage of inter-frame regularities. Thus, we proposed to modify the FDR to the structure shown in figure 2(b). As it can be seen in figure 2(a), the bottom portion of the modified FDR, which has the same size of the original FDR, can transfer data to the configuration memory. During decompression phase, the compressed bits stream is decoded and then fed to the bottom of modified FDR. The incoming data will be shifted upwards in the modified FDR. The configuration data will be transferred to the specified frame once the bottom portion of the modified FDR is filled with newly input data.

![Figure 2. (a) Original FDR, (b) Modified FDR](image)

3.2. Proposed Compression Technique

The output codeword in LZSS will consist of the index pointer and the length of the matching. Since the size of dictionary in LZSS is fixed and only its content is changeable, the size of output codeword is also fixed. For example, suppose a dictionary with 4096-symbol size. Therefore, we require 12 bits as index pointer to a dictionary position and 12 bits as the length of matched sequence. However, all of these bits in the fields of LZSS are not used in several cases. Suppose that in the above example, the length of matching is 20 symbols. Therefore, its LZSS code for matching length is “000000010100”. As it is specified in this field, only 5 bits (“10100”) are significant and other bits are unimportant. Also, there are unimportant zeros in the index pointer part when using intra-frame regularity (Because usually the beginning of dictionary will be addressed). Therefore, some of LZSS output codeword in this kind of work contain unimportant zeros. To deal with this problem, we must change the size of the fields correspond to the requirement and eliminate unimportant zeros.

Consideration of the above example shows that significant bits in the length of matching field begin with ‘1’ and is followed by “0100” bit-stream. So, there are only 4 significant bits after ‘1’. If we specify the number of significant bits after ‘1’ and append them to the significant bits, the above code (12 bits) will change as “10010100”. On the other hand, the value of the first three bits (“100”) in decimal shows the number of significant bits after ‘1’.

But decompression process of this code is impossible. Because it must be specified that how many bits are responsible for determining the number of bits after ‘1’. For this reason, the size of appended bits must be fixed to the maximum range in the code. In general, an \( n \) bit code is compute as follows:

\[
 n = 2^k + r \tag{1}
\]

So, we require \( m \) bits for appended bits of an \( n \) bit code:

\[
 m = \begin{cases} 
 k & \text{if } r = 0 \\ 
 k + 1 & \text{else} 
\end{cases} \tag{2}
\]

Therefore, in the above example \( m \) is equal to 4 and we require 4 bits for showing the number of 11 bits after ‘1’ in 12 bits code. Consequently, we can replace the “000000010100” field with “010010100” and reduce it by 3 bits.

The proposed compression technique has some advantages in comparison with LZSS. First, in this technique the size of codes is not fixed and determined corresponding to need. Second, similar to LZSS, it only requires one flag bit for separating
symbols and codes. Third, the size of appended bits is fixed for a large range of dictionary size. For example, some dictionary sizes such as 256, 512, ..., 65536 require 8, 9, ..., 16 bits in LZSS codes respectively, but codes of our compression technique only require 4 bits for appended bits.

4. Proposed Compression Algorithm
To take advantage of both inter-frame and intra-frame regularities in order to achieve efficient compression rate, we need to carefully find regularities and then intelligently rearrange the sequence of the frames based on maximum matching. On the other hand, some of the frames in a configuration file are very similar; thus we can achieve higher compression ratio by configuring them consecutively. If a frame in configuration sequence cannot be configured after its dictionary, the read-back feature allows the frame with the greatest similarity to the new frame to be read back to the bottom of modified FDR and reused as a dictionary.

Based on our proposed hardware, we must determine a certain frame to load in the bottom of modified FDR so that it greatly assists in the compression of the incoming frame. In order to obtain such information, each frame will be used as a fixed dictionary and then our compression technique will be applied to each other frame. Note that the compression is performed without changing the contents of dictionary. The lower output code represents high regularity and consequently high compression rate.

When this process is completed, a complete directed graph can be built, so that each node is correspond to a configuration frame and the weight of each edge denotes the inter-frame regularity between a dictionary frame and a destination frame. Lower weight of each edge denotes higher regularity. The source node of a directed weighted edge represents a dictionary frame for destination nodes. After creating regularity graph, we must seek to find an optimal sequence of configuration frames so that the inter-frame regularities are maximized. Hauck and Li used directed minimum spanning tree algorithm so that maximum regularity could be achieved [5].

In order to determine final configuration sequence, configuration tree must be traversed. Since dictionary of each frame must be configured first, only the pre-order traversal works correctly. In the following sections, we show the disadvantages of Hauck, Li’s algorithm and describe our algorithm in details.

4.1. Reduction of the number of Read-back
During the pre-order traversal in Hauck, Li’s algorithm, a frame with multiple children needs to be stored in Block Select RAMs for future read-back. For example, consider configuration tree in figure 3.

![Figure 3. A simple configuration tree.](image_url)

Pre-order traverse of this tree is ABC. Therefore, at first, frame A is loaded to the bottom of modified FDR and then configured. Also, it’s stored in Block Select RAM for read-back and used as dictionary of frame C. Then frame B is decoded by frame A and configured. Since frame A doesn’t exist in bottom of modified FDR, it’s necessary to be read back for decoding of frame C. Therefore, configuration of this simple tree needs a read-back process and a Block Select RAM usage. But we have proposed a technique for eliminating of these overheads.

In the process of configuration of a frame, the incoming data and dictionary frame will be shifted upwards in the modified FDR. If existing frame in the top of modified FDR will be the next required dictionary frame, we can reload it to the bottom of modified FDR. Therefore, in above example, during decoding of frame B, frame A will be shifted upwards in the modified FDR. But, frame A is the dictionary of the next frame in pre-order traversal. For this reason, by reloading the frame A from the top of modified FDR to the bottom, decoding of frame C can be done without any overhead. Of course, during pre-order traversal, this process must be applied to all of the leaf children of a node in configuration tree (See table I).

4.2. Reduction of the use of Block Select RAMs
As the numbers of frames are increased, numbers of required Block Select RAM for read-back are increased too. But each dictionary frame isn’t necessary to occupy the Block Select RAMs in all the time, so they can share the same memory slot without any conflict. It is obvious that in pre-order traversal, memory slot released by a node is only usable for rightmost sub-tree of that node, and other sub-trees of that node must use separate memory slot. Therefore, we should re-adjust the configuration graph so that for each node, the sub-tree which requires the most block of memory should be set as the rightmost sub-tree.

4.3. Determining the final configuration sequence
In spite of delay and configuration overhead that read-back and using Block Select RAMs impose to

Int’l Symposium on Telecommunications ( IST2003 ), 16-18 August, 2003 Isfahan-Iran
configuration, Hauck and Li only considered maximizing similarity in determining the configuration tree. Consequently, final configuration sequence will not be efficient. Therefore, we seek to find the minimum number of read-backs and Block Select RAMs usage. This problem is similar to Degree Constraint Minimum Spanning Tree (DCMST). The goal of DCMST is to find a minimum cost tree with constraint on the number of children of each node. The DCMST problem is NP-hard and traditional heuristics have had only limited success in solving small to midsize problems. But genetic algorithms (GAs) have shown good results in solving NP-hard problems [6]. Therefore, we can have used GA for finding optimal configuration tree with minimal overhead.

5. GA design
Since GAs consist of various components and parameters that can be modified, it is important to understand the impact of these factors on GA performance [7].

5.1. Problem Representation
The strings used for problem representation must have information on the all of the relevant parameters of the problem space and should be able to uniformly represent all possible solutions. There are several methods for representation of trees in GA, which can be classified broadly into three categories: edge, node, and edge-node encoding. In this work, we examine the impact of two node-based encoding methods that are appropriate for the DCMST problem: Prüfer and determinant encoding.

5.2. Population Initialization
There are two parameters to be decided for initialization: the initial population size and the procedure of population initialization. Recent studies have shown that satisfactory results can be obtained with a much smaller population size. Also there are two ways to generate the initial population: random initialization and heuristic initialization. we used the random method in our research.

5.3. Fitness Function
The fitness function will interpret the chromosome in terms of the physical representation (phenotype) and evaluate its fitness based on certain characteristics that are desired in the solution. The function should have an efficient computation time, since it is used a large number of times to evaluate each solution.

\[ f(t) = c(t) + \left( N_r(t) + MR(t) \right) \times n \]  

which \( c(t) \) denotes the regularity or compression rate of configuration tree. A function is used to penalize configuration sequence by adding a negative value (a function of number of read-back operations and number of Block Select RAMs usages) to the \( c(t) \) value. \( N_r(t) \) and \( MR(t) \) denote the number of read-backs and the number of required Block Select RAMs for pre-order traversal of the tree, respectively.

5.4. Selection Methods
Typically, there are two methods to select the population: \((\mu + \lambda)\) and \((\mu, \lambda)\). In \((\mu + \lambda)\) strategy, \(\mu\) parents and \(\lambda\) offspring compete for survival and the \(\mu\) best solutions are selected for the next generation. In \((\mu, \lambda)\) strategy, we select the \(\mu\) best \((1 < \mu < \lambda)\) solution from out of the \(\lambda\) offspring solutions. We used the stochastic \((\mu + \lambda)\) method.

5.5. Reproduction Operators
We have compared three crossover methods (one-point, two-point, uniform crossover) and also two mutation methods (insert and exchange mutation).

5.6. Halting Criterion
An important control parameter is the halting criterion. There are several halting criteria to choose from, including the number of generations, the acceptable computing time, and the fitness convergence. Fitness convergence occurs when almost all the chromosomes in the population have the same fitness value. In this work, fitness convergence is selected as the halting criterion.

6. Experimental Results
The results of the experiments are shown in table II. The columns of the table represent the configuration tree size and the rows show the various combinations of GA factors. These factors are E1 (Determinant encoding), E2 (Prüfer encoding), C1 (one-point crossover), C2 (two-point crossover), C3 (uniform crossover), M1 (exchange mutation), and M2 (insert mutation). The values in each cell of the table report the average value for the ten data sets in that cell. The highlighted cells represent the best solutions. As table II shows, the results of combination of determinant encoding, uniform crossover, and exchange mutation are best among the others. Also, we considered various values for other control parameters and consequently the initial population size was fixed at 100, and the crossover and mutation rate were chose as 1.0, 0.25, respectively.

7. Conclusions
The main goal of developing reconfigurable systems is to execute the applications very fast. But these systems suffer from a significant overhead due to the time it takes to reconfigure the hardware. Therefore,
Table I. The comparison of the number of read-backs

<table>
<thead>
<tr>
<th>Various configuration tree</th>
<th>20 Nodes</th>
<th>40 Nodes</th>
<th>60 Nodes</th>
<th>80 Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of read-back in Hauck algorithm</td>
<td>11</td>
<td>23</td>
<td>33</td>
<td>45</td>
</tr>
<tr>
<td>Number of read-back in our algorithm</td>
<td>4</td>
<td>7</td>
<td>12</td>
<td>18</td>
</tr>
</tbody>
</table>

Table II. The combination of GA factors

<table>
<thead>
<tr>
<th></th>
<th>20 Nodes</th>
<th>40 Nodes</th>
<th>60 Nodes</th>
<th>80 Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost</td>
<td>CPU(sec)</td>
<td>Cost</td>
<td>CPU(sec)</td>
</tr>
<tr>
<td>E1C1M1</td>
<td>227.2</td>
<td>0.513</td>
<td>624.8</td>
<td>2.274</td>
</tr>
<tr>
<td>E1C2M1</td>
<td>222.2</td>
<td>0.485</td>
<td>578.8</td>
<td>2.004</td>
</tr>
<tr>
<td>E1C3M1</td>
<td>216.3</td>
<td>0.495</td>
<td><strong>485.7</strong></td>
<td>1.8</td>
</tr>
<tr>
<td>E1C1M2</td>
<td>234.8</td>
<td>0.461</td>
<td>672.1</td>
<td>2.406</td>
</tr>
<tr>
<td>E1C2M2</td>
<td>225.7</td>
<td>0.473</td>
<td>615.2</td>
<td>1.971</td>
</tr>
<tr>
<td>E1C3M2</td>
<td>220.5</td>
<td><strong>0.411</strong></td>
<td>490.1</td>
<td>2.468</td>
</tr>
<tr>
<td>E2C1M1</td>
<td>393.6</td>
<td>0.905</td>
<td>1195.8</td>
<td>14.6</td>
</tr>
<tr>
<td>E2C2M1</td>
<td>409.8</td>
<td>0.842</td>
<td>1198.3</td>
<td>13.74</td>
</tr>
<tr>
<td>E2C3M1</td>
<td>378.4</td>
<td>1.049</td>
<td>1296.2</td>
<td>8.25</td>
</tr>
<tr>
<td>E2C1M2</td>
<td>450.5</td>
<td>0.623</td>
<td>1309.6</td>
<td>2.856</td>
</tr>
<tr>
<td>E2C2M2</td>
<td>424.5</td>
<td>0.83</td>
<td>1270.4</td>
<td>3.14</td>
</tr>
<tr>
<td>E2C3M2</td>
<td>400.7</td>
<td>0.837</td>
<td>1272.8</td>
<td>7.504</td>
</tr>
</tbody>
</table>

reduction of this overhead is an important consideration in these systems. In this paper, we have researched current compression techniques for Virtex FPGA and developed a new compression technique based on LZ coding. Our compression technique is a portion of our compression reduction algorithm. Our algorithm seeks to find an optimal configuration sequence with maximum regularity and minimum number of read-back and minimum Block Select RAMs usage by applying GA to configuration graph. The simulation results demonstrate the effective improvement compared with previous works.

8. References


Int’l Symposium on Telecommunications ( IST2003 ), 16-18 August, 2003 Isfahan-Iran

432