A new compression based approach for reconfiguration overhead reduction in Virtex based RTR systems

Farshid Farshadjam a, Mehdi Dehghan b, Mahmood Fathy a, Majid Ahmadi c,*

a Department of Computer Engineering, Iran University of Science and Technology, Tehran, Iran
b Department of Computer Engineering, Amirkabir University of Technology, Tehran, Iran
c Department of Electrical and Computer Engineering, University of Windsor, Ont., Canada N9B 3P4

Received 7 February 2004; accepted 12 September 2005
Available online 20 March 2006

Abstract

The advantage of RTR systems usually comes with some costs. The required time to map some areas of a program to an FPGA is considerable and affects the performance of RTR systems. Several methods have been developed to speed up the configuration process in these systems. Configuration compression can reduce the total number of write operations to load a configuration and it has been proven to be an efficient technique for dealing with the configuration overhead. In this paper, we have developed a new approach for reconfiguration overhead reduction in Virtex Based RTR Systems by using a compression technique based on Lempel-Ziv (LZ) algorithm. Since the order of the sequence of configuration frames affects the compression rate, we have proposed an algorithm based on Genetic Algorithm for finding the optimal configuration sequence of frames. The proposed algorithm will be applied to the input configuration file in a batch (offline) manner, and its time complexity is tolerable considering the overhead reduction obtained by having the optimal sequence of frames in run-time configuration decompression. Also, corresponding to our approach, a hardware model has been designed for configuration decompression.

© 2006 Elsevier Ltd. All rights reserved.

Keywords: Run-time reconfiguration systems; Configuration overhead reduction; Compression techniques; Genetic algorithms

1. Introduction

FPGAs are usually used as flexible hardware for the execution of applications that require high-speed computation in RTR (Run-Time Reconfiguration) systems. In such systems, the areas of a program that can be accelerated through the use of reconfigurable hardware (FPGAs) are frequently too numerous or complex to be loaded simultaneously onto the available hardware. For such cases, it is beneficial to be able to swap different configurations in and out of the reconfigurable hardware whenever needed during program execution, as shown in Fig. 1. This concept is known as run-time reconfiguration [1].

* Corresponding author. Tel.: +1 519 253 3000; fax: +1 519 971 3695.
E-mail address: ahmadi@uwindsor.ca (M. Ahmadi).
Therefore, the FPGAs must be reconfigured during the run-time of an application in RTR systems. Since run-time reconfigurable systems involve reconfiguration during program execution, the reconfiguration must be done as efficiently and quickly as possible. This is done for the benefit gained by hardware acceleration not to be overshadowed by the overhead of the reconfiguration. Stalling execution of either the host processor or the reconfigurable hardware because of configuration is clearly undesirable. In the DISC II system, from 25% [2] to 71% [3] of the execution time is spent in reconfiguration, while in the UCLA ATR work this figure can rise to over 98.5% [4]. Having reduced the delays caused by reconfiguration, performance could be greatly improved. Therefore, fast configuration is a key factor in research for run-time reconfigurable systems.

In order to reduce the configuration overhead, some methodologies such as configuration pre-fetching, configuration caching and configuration compression have been developed [5]. Configuration compression decreases the total number of write operations to load a configuration by reducing the size of the configuration file. This tactic has proved as an efficient technique to deal with the configuration overhead [6].

Unfortunately, most of the proposed compression methods are dependent on architecture of previous FPGAs and cannot be efficiently applied to the new FPGAs such as Xilinx Virtex series [7]. Dandalis and Prasanna [8] introduced a LZ-based approach to any SRAM-based FPGA such as Virtex series. However, they didn’t take the individual features within the configuration bitstream into consideration, resulting in a non-optimal compression of the bitstream. In addition, Hauck and Li [9] proposed a compression approach for Virtex series by taking advantage of the similarities within the bitstream by using the LZSS (Lempel-Ziv-Storer-Szymanski) compression approach. However, they only considered maximum similarities within the bitstream and did not analyze the effect of readback process on configuration overhead. Therefore, their algorithm does not generate the optimal sequence of configuration frames.

In this paper, we have developed a new compression approach for reconfiguration overhead reduction in RTR systems that it works efficiently on Xilinx Virtex devices. The major objectives of our work are as follows:

1. To propose a new compression method called LZFF (Lempel-Ziv-Farshid-Farshadjam) based on Lempel-Ziv (LZ) algorithm for reducing the configuration information. The compression ratio of this method is higher than LZSS and is very suitable for this application.
2. To design a hardware model corresponding to our approach that speeds up the decompression process and reduces the configuration overhead.
3. To develop a Genetic Algorithm approach for finding the optimal configuration sequence of frames to reduce the reconfiguration overhead. This sequence must keep a maximum similarity within the bitstream with the minimum number of readbacks and minimum Block Select RAMs usage.

2. Xilinx Virtex architecture

Virtex is one of the latest products of Xilinx Corporation with a huge configuration capacity (sea of Gates) [7]. Each Virtex device contains configurable logic blocks (CLBs), input-output blocks (IOBs), block RAMs, clock resources, programmable routing, and configuration circuitry. These logic functions are configurable.
through the configuration bitstream. Configuration bitstreams contain a mix of commands and data. They can be read and written through one of the configuration interfaces on the device. An overview of virtex architecture is shown in Fig. 2.

The configuration for the Virtex device is performed through the Frame Data Input Register (FDR). The FDR is a shift register into which configuration frames are loaded one by one before being transferred to the configuration memory. In addition, readback feature of Virtex allows the previous data frame to be read back to the FDR.

The Virtex configuration memory can be visualized as a rectangular array of bits as shown in Fig. 3. The bits are grouped into vertical frames, which are one-bit wide and extend from the top portion of the array to the bottom portion. A frame is the atomic unit of configuration - it is the smallest portion of the configuration memory that can be written to or read from. For each frame, the first 18 bits control the two IOBs at the top portion of the frame; then, 18 bits are allocated for each CLB row; finally, the next 18 bits control the two IOBs at the bottom portion of the CLB frame. The frame then contains enough “pad” bits to make it an integral multiple of 32 bits.

In this paper, we have used two concepts of intra-frame and inter-frame similarity repeatedly. These concepts will be explained in detail in the following sections. Due to the similarity of components in a circuit that is mapped into Virtex, we can expect some similarities between different configuration frames. We call this kind of similarity inter-frame similarity. Also, the similarity within each frame may be as important as the similarity between the frames. This intra-frame similarity exists in circuits that contain similar structures between the mapped CLB rows (see Fig. 3).

3. Overview of our approach

As mentioned before, in an RTR system several parts of an application can be accelerated through the use of reconfigurable hardware (FPGAs) whereas other areas must be executed on the microprocessor. During the
compilation of such applications, the configuration data must be generated for the parts of application that need to be accelerated. But the overhead of reconfiguration of these parts of application during the run time or execution overshadows the benefit gained by hardware acceleration. In our approach by compressing the generated configuration data in the compilation process (which is an offline process), the overhead of reconfiguration can be reduced in the execution process of an application. Then during the execution of the application, the compressed configuration data will be transferred into Virtex and decompressed by our proposed decompression hardware. Fig. 4 illustrates the general concepts of the proposed approach.

As we use a dictionary based compression algorithm, at first we must find a dictionary for each frame. Then each frame must be compressed based on its dictionary frame. Therefore, our approach in Fig. 4 consists of three phases (Fig. 5).

**First Phase:** Build $I_{FSG}$ (Inter-Frame Similarity Graph)

Create $vcf[n]$ array; /* The elements of this array are $n$ frames of the input configuration file. */

FOR $i = 1$ to $n$ DO
  Bit Conversion($vcf[i]$); /* Bit Conversion function converts the bitstream of configuration frames into a symbol stream */

FOR $i = 1$ to $n$ DO
  FOR $j = 1$ to $n$ DO
    $\text{Compression rate}[i][j] = \text{frame compression}(vcf[i],vcf[j]);$
    /* Frame compression function computes the compression ratio of $vcf[i]$ frame based on $vcf[j]$ frame as its dictionary by using the LZFF compression algorithm */

Build $I_{FSG}(V,E)$ Graph;
/* Building a complete directed weighted graph with $V$ nodes and $E$ edges named $I_{FSG}$(Inter-Frame Similarity Graph). $V$ is the set of configuration frames($vcf[1]...vcf[n]$) and $E$ is set of directed weighted edges that represent compression flow computed according the compression rate array (Fig. 8)*/

**Second Phase:** Search for the best dictionary frame of each configuration frame

$\text{Configuration Tree(root, V, E)} \leftarrow \text{GA}(I_{FSG}(V,E));$
/* GA function applies the proposed Genetic Algorithm on the $I_{FSG}(V,E)$ Graph to create the optimal $\text{Configuration Tree(root, V, E)}$. $E$ is a subset of $E$*/

**Third Phase:** Compress each frame based on its dictionary frame by LZFF and determine the order of compressed configuration frames for Virtex configuration

FOR $i = root$ to last node of $V$ set of configuration tree DO
  • Pre-order ($i$);
  • Determine how the dictionary of frame $i$ must be provided during the decompression of the frames;
  • Compress frame $i$ based on its dictionary frame by the LZFF compression;
  /* final compressed sequence of input configuration frames is determined. */

Fig. 5. The algorithm of our approach.
As shown in Fig. 5, in the first phase we convert the bitstream of configuration frames into a symbol stream. Since the size of the symbol affects the compression ratio, this conversion is carried out in our proposed compression algorithm LZFF which is based on the LZSS algorithm. Because of the inherent similarities in the configuration data, a dictionary based compression algorithm will be most suitable. The LZFF utilizes the similarity of frames very well and its compression rate is better than LZSS. Therefore, we can apply the LZFF compression algorithm to the configuration frames.

By configuring similar frames consecutively, higher compression ratios by LZFF can be achieved. So it is necessary to find the most similar frame among the remaining for each frame. The most similar frame is used as a dictionary frame by LZFF in the compilation process. During the execution process, each frame must be configured after its dictionary frame.

In order to find the most similar frame to each frame, each frame will be used as a fixed dictionary in the compilation process, and LZFF will be applied to each of the remaining frames. A compression ratio value is calculated by applying LZFF to each frame based on that dictionary frame. The most similar frame will result in the lowest value. By repeating this process, a complete directed weighted graph can be built. We call this the Inter-Frame Similarity Graph (I-FSG(V,E)), in which each node corresponds to a configuration frame and the weight of each edge denotes the inter-frame similarity between a dictionary frame and a destination frame. Lower weight of each edge denotes higher similarity. The source node of a directed weighted edge represents a dictionary frame for destination nodes.

In the second phase we search the Inter-Frame Similarity Graph to find the optimal configuration sequence of the configuration frames. Since lower weight of each edge in the Inter-Frame Similarity Graph denotes higher similarity and higher compression ratio, we must search for a subset of the weighted edges in that graph such that every node can be reached while the sum of the weights is minimized. This means that a directed minimum spanning tree must be built.

In this tree, each node which has one child or more is a dictionary frame. In order to determine the final configuration sequence, the configuration tree must be traversed. Since the dictionary of each frame must be configured first, only the pre-order traversal works correctly. However, in pre-order traversal only the left child of the dictionary frame can be configured after it. Therefore, dictionary frame is required to be read back in the extended FDR for the other children. However, readback operation imposes some overheads and delays. In addition, it is required to keep frames in Block Select RAMs to read back. It is possible to get two or more configuration trees which have equal compression rates but with different number of readback operations and different Block Select RAMs usage. In order to achieve the best possible configuration tree, we used a Genetic Algorithm for optimal decision making. In other words, Genetic Algorithm attempts to find a configuration tree from the set of configuration frames with the maximum compression ratio and minimum number of penal parameters such as readback operations and Block Select RAMs usage.

In the third phase and after Genetic Algorithm has determined the optimal configuration tree, the optimal configuration sequence of frames can be obtained by pre-order traversal starting from the root of the tree. Therefore, the dictionary of each frame is specified and the final configuration sequence of frames is compressed by LZFF. In other words, each frame of this sequence is compressed based on its dictionary frame and we can take advantage of both inter-frame and intra-frame similarity. Also, based on hardware variations that we have proposed, it must be determined which frame should be read back and which should be loaded from the top portion of the extended FDR to the bottom portion of it. The details of our approach have been described thoroughly in the next three sections.

4. First phase

4.1. The analysis of similarity and compression algorithm

Usually in a hardware circuit, there exist many similar gates and hardware resources (such as AND, OR, DECODER, ADDER and so on). Therefore, some similarities are expected to exist between configuration frames. We call this kind of similarity inter-frame similarity. On the other hand, there are usually some similarities within a single frame of configuration that we call intra-frame similarity. This similarity can be found in circuits that contain similar structures between rows. Therefore, inter-frame and intra-frame similarities are
individual features of the Virtex configuration bitstream. Regarding these similarities, we have proposed a compression technique in our approach that takes advantage of the similarity between data very well and fits this application efficiently.

Since each bit of the configuration information has a key role in describing a part of hardware, only a lossless compression technique should be applied. Well-known lossless compression techniques including Huffman, Arithmetic, and Lempel-Ziv (LZ) coding were considered by Hauck and Li [10]. They evaluated the performance and compression ratio of these techniques on a sample set of Virtex configuration files [9]. The results of their simulation proved the compression ratio of LZSS (a version of Lempel-Ziv) to be better than the others.

The problem with Huffman and Arithmetic coding lies in the fact that they are probability based compression techniques. In other words, transferring and keeping a significant amount of probability values in these methods will cause additional overhead and increases the hardware and time of decompression. Also, arithmetic coding can not take advantage of the existent similarity in the Virtex configuration bitstream because it is a compression algorithm that performs well on a stream of unrelated symbols.

However, the LZ compression is an algorithm that represents repeated symbols efficiently. This dictionary based compression algorithm maintains a group of symbols that can be used to code recurring patterns in the stream. If the algorithm spots a sub-stream of the input symbols that have been stored as parts of the dictionary, the sub-stream can be represented in a shorter code word. The related symbols caused by the similarities in the configuration bitstream make LZ algorithm an effective compression approach. In addition, the shift based FDR in the Virtex can be used as a dictionary during decompression process. It is obvious that additional hardware must be added to allow reading of specific frame locations during the decompression, which will be explained later on. Considering the inherent advantage of the dictionary based compression algorithms, in this paper we will propose an LZ based compression algorithm (LZFF) that uses the similarity between data very well.

4.2. The bitstream conversion

In dictionary based compression algorithms, the size of the basic symbols could affect the compression ratio. For the long symbols the potential intra-symbol similarities will likely be overwhelmed. On the other hand, in spite of the fact that very short symbols (such as one bit) retain all the similarities, the coding overhead will be significantly increased. For this reason we must convert the configuration bitstream into a symbol stream and finely tune the size of the symbols to achieve the higher compression ratio.

As mentioned above, some bits are added to make the configuration bitstream an integral multiple of 32 bits. Therefore much of the similarities will be missed if the symbol length is set to 32-bit or any powers of 2. As shown in Fig. 1, each CLB row within a frame is controlled by an 18-bit value and the similarities discussed above exist in the 18-bit fragments rather than 32-bit ones. In order to preserve those similarities, the original 32-bit configuration bitstream must be broken. Hauck and Li demonstrated the simulation results of 6-bit symbols are better than any other sizes [9].

4.3. The proposed compression technique (LZFF)

The dictionary based compression algorithms such as Lempel-Ziv usually use the similarity between data splendidly. This type of compression algorithms is of great use in our approach because for the compression of each configuration frame we can use the previous data frames as a dictionary (during the compilation process). Then during the execution process in RTR system, each compressed configuration frame must be configured immediately after its dictionary, because it can only be decompressed based on its dictionary frame. In this way, we can take advantage of the configuration frames’ similarities and reduce the size of configuration frames.

There are many variations of the LZ compression algorithms, such as LZ78 and LZW that maintain occurred patterns as a dictionary. In other words, they build a static dictionary of occurred patterns. However, retaining such a dictionary requires excessive amount of hardware resources and is not usable in our approach. Other variations of LZ compression algorithms such as LZ77 utilize a dynamic dictionary. In such algorithms the dictionary is a simple shift based buffer. The LZ77 tracks the last n symbols of data previously seen, where
is the size of dictionary with a serial input. However, this method will generate a code word with three fields when no matching is found.

There is a variation of the LZ77, called LZSS, which has a higher compression ratio and only uses a sliding window buffer as its dictionary which is most suitable for use with the FDR of the Virtex. LZSS institutes limitations on what can or cannot be encoded. The length variable must be larger than a predetermined minimum. If the match is found and the length is smaller than the minimum, the current symbol is written. However, when the length of the match is larger than or equal to the minimum, the codeword containing the index pointer of the first symbol in the dictionary and length of the match is output. In addition, LZSS will need to add a flag bit to indicate whether the bits are a compressed LZ-packet or an uncompressed symbol. Hauck and Li have used the LZSS for configuration compression of Virtex [9]. Whereas, in our approach we have used our proposed compression method (we call it LZFF) that is better than LZSS and works on this application efficiently.

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

As it can be seen, this coding algorithm generates variable-sized codes because the sizes of appended bits will be changed corresponding to significant bits after ‘1’ which make decompression impossible. To overcome this problem, we fixed a number of bits that are responsible for determining the number of significant 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 computed as follows:

\[ n = 2^k + r \]  

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

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

According to Eqs. (1) and (2) formulas, in the above example \( n, k \) and \( r \) will be 12, 3 and 4 respectively. Therefore \( m \) is equal to \((3 + 1)\) and we require 4 bits for showing the number of 11 significant bits after ‘1’ in 12 bits code. Consequently, we can replace the “000000010100” field with “01000100” and reduce it by 4 bits.

The proposed compression technique has some advantages in comparison with LZSS. First, in this technique the size of the codes is not fixed and is determined according to the requirements. 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 sizes. For example, some dictionary sizes such as 512, . . . , 65536 require 9, . . . , 16 bits in the LZSS codes respectively, but the codes of our compression technique only require 4 bits for the appended bits.

4.4. The proposed hardware model

In this work we have proposed a hardware model corresponding to our approach. As mentioned in the previous section, the LZFF only requires a simple shift based buffer and the FDR in Virtex fits this scheme per-
fectly. Therefore, in the proposed hardware model we have used the FDR as a dictionary. This model contains three important parts.

The first part is the decompression hardware which has been illustrated by a control box in Fig. 6. This controller checks the input configuration frames for the LZFF codes and uncompressed symbols. If it detects an LZFF code, it recovers the corresponding configuration symbols from the dictionary to this code and feeds it to the bottom portion of the FDR. In any case, the configuration symbols will be shifted upwards in the shift based FDR and transferred to the specified frame once the FDR is filled.

As mentioned above, during the decompression of a new frame, the content of its dictionary frame in the FDR must be shifted upwards to obtain that new frame. But the FDR can only contain one configuration frame and consequently, the symbols of the dictionary frame in top of the FDR will be lost and inter-frame similarity will be missed symbol by symbol. In order to take advantage of the inter-frame similarity, the dictionary frame should be saved in a buffer. Therefore, the FDR must be modified to keep the dictionary frame. Of course, the size of the modified FDR must be balanced against the potential hardware cost. In our research, we allow the modified FDR to contain 2 frames of data. This will not significantly increase the hardware overhead yet will utilize the similarities in the configuration stream. Therefore, the second part of our proposed hardware model is the expansion of the FDR to the structure shown in Fig. 6.

As it can be seen in Fig. 6, the bottom portion of the extended FDR can transfer data to the configuration memory. Therefore, during the decompression, the compressed bitstream is decoded and then transferred to the bottom portion of the extended FDR. The incoming data will be shifted upwards in the extended FDR. The configuration data will be transferred to the specified frame once the bottom portion of the extended FDR is filled with a new frame.

Also, for increasing the speed of decompression process and reducing some of the configuration overheads specifically for the Virtex, a reloading path has been proposed for our hardware model. This part of the proposed hardware model will be explained in detail in the second phase (Section 5) of our approach.

### 4.5. Building the inter-frame similarity graph

The goal of configuration compression is to take advantage of both inter-frame and intra-frame similarities. In the configuration stream, some of the frames are very similar, and by configuring them consecutively, higher compression ratios can be achieved. On the other hand, based on our proposed hardware model the key factor of higher compression ratio in the LZFF is the similarity between the frame in the extended
FDR and the new incoming frame. Therefore, in order to achieve a higher compression ratio, we will seek to place a certain frame in the extended FDR such that it be the most similar frame to the incoming frame.

We illustrate the above discussion with an example. As it can be seen, Fig. 7 shows four configuration frames so that the frames (b), (c) and (d) are more similar to (a) than to each other. For this reason, in order to take advantage of inter-frame similarity it is better to select frame (a) as the dictionary frame for the others. Therefore, these three frames must be configured after frame (a) immediately. But this is impossible because only one frame can be configured at a time. Therefore, the frame (a) must be loaded in the bottom of the extended FDR from the FPGA array for configuration of any one of the frames (b), (c), (d). To deal with this problem we can use the readback feature of Virtex. In current Virtex devices, the data stored in the Block Select RAMs can be transferred to logic in a very short amount of time. This feature could be used for increasing the inter-frame similarity because it allows the frame with greatest similarity to the new frame to be read back to the bottom portion of extended FDR and reused as a dictionary.

By providing the fast readback from only the Block Select RAMs, we can use the Block RAMs as caches during reconfiguration to hold commonly requested frames without significant hardware costs. Therefore, in the above example, with the fast readback feature, we can temporarily store frame (a) in the Block Select RAMs, read it back to the extended FDR, and use it as a dictionary when other frames are configured. While without readback, the inter-frame similarities between (a) and two of frames (b), (c), (d) will be missed.

Based on our proposed hardware, in order to fully take advantage of inter-frame similarity, we must determine which frame to be loaded in the bottom portion of the extended FDR so that it can maximize the compression of the incoming frame. In other words, we must determine the dictionary of each frame.

---

**Fig. 7.** An example of inter-frame similarity.

**Fig. 8.** An inter-frame similarity graph.
obtain such information, the amount of similarity between frames must be specified. For this reason each frame must be used as a fixed dictionary and then our compression technique (LZFF) can be applied to each of the other frames. Note that the compression is performed without changing the contents of the dictionary. The lower output code represents high similarity and consequently high compression rate.

When this process is completed, a complete directed weighted graph can be built (Fig. 8) which we call the Inter-Frame Similarity Graph (I-FSG($V, E$)). In this graph each node of the $V$ set corresponds to a configuration frame and the source node of each directed weighted edge of $E$ set represents a dictionary frame for destination nodes. The weight of each edge denotes the inter-frame similarity (compression flow) between a dictionary frame and a destination frame so that the lower weight of an edge denotes higher similarity. In this way the dictionary of each frame can be easily specified based on the similarity values in the Inter-Frame Similarity Graph.

5. Second phase

5.1. Finding the optimal configuration tree

Given an Inter-Frame Similarity Graph, in order to achieve high compression ratio, the most similar frames must be configured successively. In other words, a sequence of most similar frames must be determined such that each frame is the dictionary of the next frame in sequence. Since lower weight of each edge denotes higher similarity or higher compression ratio, we must seek to find a subset of the weighted edges in the Inter-Frame Similarity Graph such that every node can be reached and the aggregate of the weights is minimized. But the frames of this subset cannot form a sequence of frames because as shown in Fig. 7, sometimes a frame is the dictionary of some frames and thus a sequence of configuration frames cannot be obtained.

For solving this problem, Hauck and Li [9] first constructed a configuration tree by applying the directed minimum spanning tree algorithm on the Inter-Frame Similarity Graph, and then performed pre-order traversal of configuration tree for determining the sequence of configuration frames. Since the dictionary of each frame must be configured before its children, only the pre-order traversal works correctly. In the following paragraph, we show the disadvantage of Hauck and Li’s algorithm and explain our algorithm in details.

Consider the two configuration trees in Fig. 9. Suppose that the similarity of these trees is equal. However, according to Hauck and Li’s algorithm, tree (b), in Fig. 9, requires 7 readback operation and tree (a) only requires 3 readback operations. Therefore, configuration tree in Fig. 9(a) is better than configuration tree in Fig. 9(b). But in Hauck algorithm these trees are considered the same.

For a practical configuration tree with many frames, the delay and configuration overhead for the Block Select RAM readback will go beyond the acceptable limit. Also, each stored frame on Block Select RAMs

![Fig. 9. Two trees with equal similarity but different read back operations.](image-url)
must remain for the readback of its last child, while some applications may restrict the use of the Block Select RAM. In spite of these effective parameters, Hauck and Li only considered maximum similarity in determining the configuration tree. Consequently, the final configuration sequence would not be optimum.

To overcome this problem, we must seek to find a configuration tree with the maximum similarity and the minimum amount of readbacks and Block Select RAMs usage. But this process is very difficult and time-consuming for big Inter-Frame Similarity Graphs and is not possible to be solved algorithmically. Consequently, an optimal configuration tree cannot be obtained in polynomial time. This fact leads to an NP-complete problem and today many researchers are looking for ways to solve such problems. The Genetic Algorithm is one of the methods used to solve such problems and it generates a near-optimal solution for NP-complete problems.

Therefore, we can apply the Genetic Algorithm on the Inter-Frame Similarity Graph for finding an optimal configuration tree with the maximum similarity and the minimum number of readback operation and Block Select RAMs usage. In this process, the various configuration trees are generated by the Genetic Algorithm and evaluated based on the desired features (similarity, readback, and Block Select RAMs). The evaluation of each configuration tree is performed by a fitness function. The fitness function calculates the amount of similarity, the number of readback operation, and the number of Block Select RAMs usage for each configuration tree. Then the optimal configuration tree can be obtained based on the fitness computations.

Before elaborating further on our proposed Genetic Algorithm, we will explain two optimization techniques in our approach, which must be applied to each configuration tree before pre-order traversal. The fitness function also calculates the number of readback operation and the amount of Block Select RAMs usage for each configuration tree after these optimizations. These optimizations are performed to delete some delays and configuration overheads for the readback operation and the constraint on usage of the Block Select RAMs. The following two sections describe these optimization methods.

5.2. Reduction of the number of the readback operation

During the pre-order traversal in Hauck, Li’s algorithm, a frame with multiple children needs to be stored in the Block Select RAMs for the future readback. For example, consider the configuration tree shown in Fig. 10. The pre-order traversal of this tree is A, B, C. Therefore, at first, frame A is loaded to the bottom portion of the extended FDR and then configured. Also, it is stored in the Block Select RAM for readback and used as the dictionary of frame C. Then frame B is decompressed based on its dictionary frame (frame A) and configured (Fig. 11).

As it can be seen in the above figure, since frame A does not exist in the bottom portion of the extended FDR, it is necessary to read back for the decompression of frame C (Fig. 12). Although the configuration of this simple tree needs a readback process and a Block Select RAM usage, these parameters will be increased for a practical configuration tree with many frames.

As it can be seen above, in [9] a copy of each dictionary frame of the configuration tree that has more than one child must be stored into an empty slot of the Block Select RAMs to be read back for the decompression of its children. But in our approach, only some of the dictionary frames are required to be stored into the Block Select RAM. In other words, we can sometimes prepare the dictionary frame without any readback operation. We explain our technique by an example.

Consider the configuration tree shown in Fig. 10 again. The pre-order traversal of this tree is A, B, C. Therefore, at first, frame A is loaded into the bottom portion of the extended FDR and then configured. When decompressing its first child in the pre-order traversal (frame B), this dictionary frame will be shifted upwards in the extended FDR and placed on the top portion of it (Fig. 11). Also, frame A is the dictionary of

Fig. 10. A simple configuration tree.
the next frame in the pre-order traversal (frame C). But based on our hardware model, for decompression of the frame C, its dictionary frame (frame A) must be in the bottom portion of the extended FDR. For this reason, we can load frame A from the top portion of the extended FDR to the bottom portion (Fig. 13).

However, in order to take advantage of this technique, we must modify our hardware model to the structure shown in Fig. 13. This is the second part of our hardware model in our approach. By adding this reloading path to the hardware model, the decompression of the frame C can be done without any readback. This will utilize the similarities in the configuration stream without significantly increasing the hardware overhead. In this way, in a practical configuration tree, many readback operations will be deleted and the speed of decompression will be increased. Also, the configuration overhead will be reduced.

By analyzing the configuration trees, we found that a dictionary frame can use the reloading path for each of its leaf children once to decompress its next child in the pre-order traversal. However, this is true under one
condition, that the child leaves of a dictionary frame are traversed first. We demonstrate the above discussion by an example.

Fig. 14 shows two configuration trees. The pre-order of configuration tree (a) is A, B, D, E, and C. Therefore, frame A is configured first and then its first child (frame B). However, corresponding to the pre-order sequence, the frame C cannot be configured and the frame D must be configured now. Therefore, based on our hardware model, the frame A can not remain in the top portion of the extended FDR and must be stored into an empty slot of Block Select RAM to be read back for the decompression of the frame C.

Now we exchange the subtrees of frame A (Fig. 14(b)). The pre-order of this tree is A, C, B, D, and E. Similar to tree (a), frame A is configured first and then frame C is decompressed. Based on our hardware model, during the decompression of frame C, frame A will be shifted upwards in the extended FDR and placed on the top portion of it. Regarding the pre-order sequence of tree (b) frame B must be traversed now. However, this frame is also the next child of frame A. Therefore, by using the reloading path, we can prepare the dictionary of frame B (frame A). In this way, it’s not necessary to store and readback frame A.

Therefore, in order to take advantage of the reloading path and eliminate the additional readback operation, we should readjust the configuration tree for any node with child leaves so that its children are configured first.

5.3. Reduction of the usage of the block select RAMs

Another concern in our algorithm is the storage requirement for those frames that are read back. In other words, the Block Select RAMs in Virtex are critical resources so that some applications may restrict their use.
By analyzing the inter-frame similarity graphs, we found that although a large number of frames need to be read back, they are not required to be held in the Block Selected RAMs all the time, and they can share the same memory slot without any conflict.

It is obvious that the memory required by a node depends on the memory required by each of its children. One important observation is that the memory required by the largest sub-tree can overlap with the memory required by other sub-trees. In addition, since the last child of a node to be configured can use the memory slot released by its parent, the memory required by configuring all sub-trees can be equal to that of configuring the largest sub-tree. Since the pre-order traversal will scan the left sub-trees before the right sub-trees, we should readjust the configuration tree such that for each node the sub-tree, which requires the most memory, should be set as the rightmost sub-tree. Therefore, by combination of these two techniques (optimizations), we can reduce the number of readback operations and required the Block Select RAM usage for a configuration tree during the configuration process. In this way, some overheads and delays for readback operations will be deleted and the Block Select RAMs are more accessible for applications.

5.4. Problem definition and formulation

As mentioned before, an Inter-Frame Similarity Graph is a complete directed weighted graph. In this graph, each node corresponds to a configuration frame and the weight of each edge denotes the inter-frame similarity (compression rate) between a dictionary frame and a destination frame.

This graph can be modeled as $G=(V,E)$, while $V$ is the set of all nodes, representing configuration frames, $E$ is the set of directed weighted edges representing compression flow. For example, directed weighted edge $i \rightarrow w j$ indicates that frame $j$ is compressed based on frame $i$ by compression rate $w$. In other words, frame $i$ is the dictionary of frame $j$.

Each compression flow is bidirectional, i.e. the existence of a compression flow $e=(u,v)$ from node $u$ to node $v$ implies the existence of another compression flow $e'=(v,u)$ for any $u,v \in V$ as each two frames can be dictionary of each other with different compression rate. One non-negative integer value function is associated with each compression flow $e(e \in E)$, Compression Rate $CR(e):E \rightarrow N$, where $N$ is the set of positive natural numbers. The compression rate function, $CR(e)$, indicates the rate of compression of destination frame based on dictionary frame. Because of the asymmetric nature of the dictionary based compression methods such as LZFF, it is often the case that $CR(e) \neq CR(e')$.

A configuration tree $T(R)$ is a sub-graph of $G$ that spans the root frame $R \in V$. We set each frame as the root of configuration tree that has minimum $CR(e)$. Let $C(T(R))$ be the compression flow set of configuration tree $T(R)$. Let $N_r(T(R))$ be the number of the readback operation of the configuration tree $T(R)$ based on our proposed hardware model. Also let $MR(T(R))$ be the number of Block Select RAMs usage. The total similarity or total compression rate of the tree $T(R)$ is defined as the sum of the similarities or compression rates in that tree and can be given by:

$$C(T(R)) = \sum_{e \in E(T(R))} CR(e). \quad (3)$$

Finally, finding a configuration tree with the maximum similarity and the minimum number of readback operations and the Block Select RAMs usage is defined as the minimization of $C(T(R))$ subject to the minimization of $N_r(T(R))$ and $MR(T(R))$.

5.5. The proposed genetic algorithm

Genetic Algorithms are search algorithms based on the mechanisms of genetic adaptation in biological systems [12]. The algorithm maintains a population of individuals (chromosomes) where each individual corresponds to a specific solution to the given problem. The quality of an individual is judged by its fitness measure. A simple Genetic Algorithm works according to the following steps:

1. Generating an initial population of individuals.
2. Evaluating the fitness of every individual.
3. Producing a new generation of population based on the fitness of individuals in the current population (by using genetic operators like crossover and mutation).
4. Evaluating the fitness of every individual in the new generation.
5. Continuing to execute steps 3, 4 until a given termination criterion is gratified.

The Genetic Algorithm starts with an initial population of random individuals and simulates the process of evolution. After a number of generations, highly fit individuals will emerge corresponding to good solutions to the given problem. One of the most difficult tasks in designing a Genetic Algorithm lies in the representation (encoding) of individuals. This genetic encoding (representation) of an individual is called a genotype, and the corresponding physical appearance of an individual is called a phenotype. Note that fitness is defined in terms of phenotype. In the following section the components of the Genetic Algorithm are introduced.

5.5.1. Genotype

The strings used for the encoding of individuals must represent information about the solution and should be able to encode all possible solutions uniformly. The representation of individuals is divided in two categories: direct or indirect [11]. With the direct representation, the strings can be read directly, while with the indirect representation, a decoding process is needed to expand the strings into meaningful information for evaluation. Since the desired target in this phase is a configuration tree, Genotype must represent a tree. There are various encoding methods for the representation of trees [11]. They can be classified broadly into three categories: edge, node, and edge-node encoding.

In edge encoding [13], a string is used to represent the edges of the spanning tree that contains binary and integer numbers. For example, Dengiz et al. [14] used an integer string to represent a tree in communication networks, where each integer arbitrarily represents an edge. Edge encoding has proved to be a poor representation for Minimum Spanning Tree (MST) due to the low probability of obtaining a tree [15].

In node-based encoding the nodes are represented in the encoding. A famous node-based representation method for trees is the Prüfer number encoding [16]. Prüfer encoding is an indirect encoding method where a tree with \( n \) nodes is represented with \( n - 2 \) digits, where each digit is an integer between one and \( n \). However, it has a very limited locality, due to the complete change in the tree caused by the slightest fluctuation of any one digit in the Prüfer number [15]. Another node-based representing method for trees is determinant encoding which is an indirect encoding strategy proposed by Abuali et al. [17] to overcome the bottlenecks of Prüfer encoding. In this method a tree with \( n \) nodes is represented with \( n - 1 \) digits, while each digit is an integer between one and \( n \). The decoding algorithm treats each allele of gene as its index in the chromosome and the index represents its direct connecting node. The first gene is decoded as index 2, second as index 3, and so on.

Edge and node encoding use information about both nodes and links. An example in this category is Link and Node Biased (LNB) encoding [15], which uses a bias value for each node and link in calculating the cost of the network. This representation has some limitations including long encoding and lack of information about the degree on nodes. Abuali et al. [17] comparing Prüfer, determinant, and LNB encoding methods, realized that determinant encoding provided better performance. In this paper, we examine the impact of two encoding methods that are appropriate for the DCMST problem: Prüfer and determinant encoding.

5.5.1.1. Initial population. Genetic Algorithm is started with a set of solutions (represented by chromosomes) called the initial population. Two parameters must be decided for initialization: the initial population size and the procedure to initialize the population. In spite of the former opinion about population size, recent studies have shown that very big population size usually does not improve the performance of Genetic Algorithm in terms of the speed of convergence [11]. Also, there are two procedures for generation of the initial population: random initialization and heuristic initialization. In this paper we use the random procedure for both experimental encoding methods. For each gene, we randomly generate an integer from a range of one to the number of nodes. Since the chromosomes coded by Prüfer are all legal spanning trees, there is no need to repair illegal chromosomes. However, determinant encoding generates illegal trees that need to be repaired. In the following section we explain our repair function.
5.5.1.2. Repair function. Although determinant encoding is an indirect encoding strategy, the decoding algorithm is very simple. However, it generates some chromosomes that are illegal and makes the decoding algorithm impossible. Therefore, chromosomes in this encoding must be repaired to correct the modeling of solutions. A determinant code may be illegal due to three reasons: missing root, self-loop or cycles.

- If we label the root of a tree by “1”, Missing root occurs when in a chromosome there exist no genes that have value “1”. Because, the index starts from two and therefore, the code will not span the root. For example, in this code (3 6 3 4 5) there is no root and we have these connections (3-2), (6-3), (3-4), (4-5), (5-6).
- Self-loop occurs when the value of a gene is equal to its correspondent node index. For example, in determinant coding, a chromosome (4 3 5 11) cannot construct a six-node spanning tree because the value of the gene in the third index is three, which causes an illegal self-loop connection (3-3).
- Cycle occurs when a subset of connections constructs a loop. For example, in this code (4 2 3), there is a cycle (4-2), (2-3), (3-4) in which one of the connections is unnecessary. The algorithm illustrated below describes a strategy to solve all the three situations resulting in illegal spanning trees.

Given a determinant encoding string DC with length of \( N - 1 \), where \( N \) represents number of nodes. Assume DC\((x)\) represents the allele of the \( x \)th index in chromosome DC, where \( x \) starts from two to \( N \). The cost for a nine-node spanning tree problem is shown in Table 1. Also Fig. 15 shows the chromosome coded by the determinant encoding method, which contains three kinds of problems: missing node 1, self-loop, and cycles. The algorithm presented below illustrates the repairing process for the three problems with the above characteristics.

1. Repair the missing root
   Scan the determinant code, DC, to ensure that the root is in the code. This means: \( \exists x \text{DC}(x) = 1 \).
   If the root is not already in the code, check the cost table and pick node \( x \) (\( x \neq 1 \)), where 1 has the lowest connection cost to node \( x \) and Set DC\((x)\) = 1. If there are some nodes with equal costs, select one of them at

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>*</td>
<td>35</td>
<td>35</td>
<td>20</td>
<td>70</td>
<td>20</td>
<td>60</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>60</td>
<td>*</td>
<td>40</td>
<td>70</td>
<td>80</td>
<td>60</td>
<td>80</td>
<td>45</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>70</td>
<td>50</td>
<td>*</td>
<td>40</td>
<td>50</td>
<td>45</td>
<td>75</td>
<td>35</td>
<td>80</td>
</tr>
<tr>
<td>4</td>
<td>60</td>
<td>40</td>
<td>35</td>
<td>*</td>
<td>25</td>
<td>80</td>
<td>75</td>
<td>90</td>
<td>70</td>
</tr>
<tr>
<td>5</td>
<td>35</td>
<td>30</td>
<td>50</td>
<td>80</td>
<td>*</td>
<td>55</td>
<td>60</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td>6</td>
<td>45</td>
<td>90</td>
<td>60</td>
<td>50</td>
<td>55</td>
<td>*</td>
<td>45</td>
<td>40</td>
<td>35</td>
</tr>
<tr>
<td>7</td>
<td>25</td>
<td>50</td>
<td>75</td>
<td>75</td>
<td>40</td>
<td>55</td>
<td>*</td>
<td>90</td>
<td>60</td>
</tr>
<tr>
<td>8</td>
<td>90</td>
<td>55</td>
<td>80</td>
<td>45</td>
<td>90</td>
<td>90</td>
<td>65</td>
<td>*</td>
<td>25</td>
</tr>
<tr>
<td>9</td>
<td>85</td>
<td>40</td>
<td>60</td>
<td>65</td>
<td>75</td>
<td>35</td>
<td>90</td>
<td>70</td>
<td>*</td>
</tr>
</tbody>
</table>

Index: 2 3 4 5 6 7 8 9

(a) Genotype

Fig. 15. Determinant code problems.
random. For example, there are no roots in Fig. 15. Therefore, we must find the lowest cost node connected to node 1. Suppose that node 1 has the lowest connection cost to nodes 2, 3, and 6. Since they all have the same cost, one of them is chosen randomly, say, index 6. Replace the allele in index 6 with 1 (DC(6) = 1). The result is shown in Fig. 16.

2. Repair self-loop
For a given encoding string DC, identify $x$, where $x = DC(x)$. Then, for each $x$, repeat the following process to repair the self-loops. Check the cost of connecting each node $i (i \neq x)$, with node $x$. Select node $j$, which has the lowest connection cost with node $x$ and set $DC(x) = j$. If there are some nodes with equal cost, choose one at random.

For example, there are two self-loops in Fig. 15: the third and the seventh index (4-4 and 7-7). Select a node with the lowest cost connection to nodes 4 and 7. Suppose again that node 1 is paired with node 4 and node 6 is paired with node 7. The result is presented in Fig. 17.

3. Number of Components (NC)
In this section we are sure that the code includes the root and there are no self-loops. Now the code must be checked to represent a tree. If the code represents a tree, it will be only one component. Otherwise there are two or more components in the code. The component that contains the root will be the tree component and will have no cycle. But the other components will contain only one cycle. This is true because any given vertex has an in-degree of one and each component will only have one cycle at most. Therefore, in this section, we keep track of each node in the code and assign it to a component by using a grouping algorithm. If the number of components (NC) is equal to one, then the determinant code represents a tree, and no repair is required. Otherwise, the determinant code must be repaired in step 4. In the grouping algorithm, the nodes in the same component are grouped in the same set and numbered so that one is assigned to the root component and two, three... to the other components. Therefore, the algorithm constructs an array (Array A in Fig. 18). This array contains information on the sets where each node belongs to. For the sets numbered by two, three... in this array, repair function is used to delete the cycles and connect the components to the tree component. Fig. 18 shows the result of the grouping algorithm.

![Fig. 16. Repairing missing root.](image1)

![Fig. 17. Repairing self-loops.](image2)

![Fig. 18. Grouping the nodes in the determinant code.](image3)
4. Repair cycle
As mentioned in the previous step, we consider NC to be the number of components in the graph represented by the determinant code. Let \( T \) be the tree component. Let \( C_1, C_2, \ldots, C_{NC-1} \) be the other various components. While the number of components (NC) is greater than one, repeat the following steps:
(a) Select a component at random, \( C_K \).
(b) Find the connections that make a cycle in \( C_K \).
(c) Find the connection \( j \to i \) that has the most cost among the connections which make a cycle in \( C_K \).
(d) Delete this connection.
(e) Find the lowest cost connection \( p \to i \) where \( p \) is one of the vertices in the \( T \) component.
(f) Connect the \( T \) and \( C_K \) components by adding the connection \( p \to i \). This means that \( DC(i) = p \).
(g) Let \( NC = NC - 1 \).
For example in Fig. 18, there are only two components: \( T \) and \( C_1 \). So we must select \( C_1 \) component. Having checked the cost of the links of this component, we find out that link \((3, 5)\) has the most cost among the others. According to the above algorithm, we delete this link. Now we must find a node in the \( T \) component that has the lowest cost to node 5. Based on the cost table, node 4 is found to be lower in cost. Thus, we replace the allele in index 5 by node 4 (\( DC(5) = 4 \)). The updated table and chromosome are represented in Fig. 19.

5.5.1.3. Fitness function. The fitness function will interpret the chromosome in terms of the physical representation (phenotype) and evaluate its fitness based on certain characteristics desired in the solution. The definition of the fitness function is very critical because it must accurately measure the desirability of the features described by the chromosome. The Genetic Algorithm maintains a population of individuals: \( T = \{ T_0, T_1, \ldots, T_{N-1} \} \) where \( T_i \) is a configuration tree and \( N \) is the population size. Therefore, we define the fitness function for \( T(R) \) as:

\[
F(T(R)) = C(T(R)) + (N_r(T(R)) + MR(T(R))) \times N, \tag{4}
\]

where \( C(T(R)) \) denotes the similarity or the compression rate of the configuration tree. A function is used to penalize the configuration tree by adding a negative value (a function of the number of readback operations and the number of Block Select RAMs usage) to the \( C(T(R)) \) value.

5.5.1.4. Selection methods. The following is a comparison of some selection methods and their suitability to our application. First, the Roulette Wheel Selection will have some problems when the fitness differs profoundly. For example, if the best chromosome fitness is 90% of the entire roulette wheel then the other chromosomes will have very few chances to be selected. Second, the Rank Selection can lead to slower
convergence, because the best chromosomes do not differ so much from the other ones. Therefore, we used the tournament selection because this method showed good results compared to the other methods [12].

5.5.1.5. Replacement strategy. There are many methods for the selection of the population. There are advantages and disadvantages to each method [18–20]. Some researchers prefer to use the enlarged sampling approach since it reduces the possibility of duplicating chromosomes entering the population during the selection process [12]. Typically, there are two enlarged sampling strategies: \((\mu + \lambda)\) and \((\mu, \lambda)\). In strategy \((\mu + \lambda)\), \(\mu\) 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 \((1 < \mu < \lambda)\) solutions from out of \(\lambda\) offspring solutions. In this paper we have used \((\mu + \lambda)\) method.

5.5.1.6. Reproduction operators: crossover. The role of crossover operator is to recombine information of two good parent solutions, which is expected to bring about even better offspring solutions. The form of the crossover operator depends on how the problem is coded. Crossover methods such as one-point, two-point, and uniform crossover are extensively used in Genetic Algorithm models. Booker [21] compared one and two-point crossover and realized that the two-point crossover is better for non-order problems. Sywerda [22] compared one-point, two-point, and uniform crossover operator on six different types of problems using a steady-state GENETIC ALGORITHM with simple mutation and binary encoding. He found uniform crossover to be more effective than two-point crossover and two-point crossover consistently better than one-point crossover. Poon and Carter [23] compared the performance of ten crossover operators over six applications and determined that order and union crossover operators performed uniformly well in all applications while the position-based and intersection-crossover operators were the worst performers. In this paper, we evaluate the effectiveness of one-point, two-point, and uniform crossover methods.

5.5.1.7. Reproduction operators: mutation. The Mutation operation is of great importance on which the performance of Genetic Algorithm depends. When a new offspring is produced, the mutation operation is performed according to the probability of mutation. Various mutation methods have been examined including inversion, insertion, displacement, reciprocal exchange, and heuristic mutation [12]. Two mutation methods have been used here, namely, insert and exchange mutation.

5.5.1.8. 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 [11]. Fitness convergence occurs when almost all the chromosomes in the population have the same fitness value. In this work, in order to optimize the performance of proposed Genetic Algorithm, we used a heuristic halting criterion that has been described in simulation result section.

6. Third phase

Our experiments show that the aforementioned Genetic Algorithm generates a configuration tree with maximum inter-frame similarity and minimum number of readback operation and number of Block Select RAM usage. Now we can first obtain the final sequence of configuration frames by pre-order traversal of optimal configuration tree that was generated by Genetic Algorithm and then compress it. But as discussed in second phase, some frames cannot be placed after their dictionary in the final sequence. Therefore, we must simultaneously perform the compression process and decision on readback operation during the traversing of each frame of the configuration tree.

As a result, during the traversing of each frame, we compress it based on its parent (its dictionary frame) by LZFF and take advantage of both inter-frame and intra-frame similarities. Also, based on our proposed hardware model, we determine how the dictionary of each frame must be provided during its decompression. In other words, unless it is specified that a dictionary frame can remain in the top portion of the extended FDR for decompression of the next child in the final sequence of configuration, it can not use the reloading path of our hardware model. Otherwise it must be stored in Block Select RAM to be read back for decompression of the next child.
7. Experimental results

We implemented our algorithm in C++ on a personal computer Pentium III 800 MHZ, 128 MB RAM. Because benchmarks of Xilinx were unavailable, all experiments are performed on various simulated configuration files. The simulation of a configuration file is begun with the Inter-Frame Similarity Graph building stage in our algorithm (Fig. 5). In other words, we generated a complete directed weighted graph with \( n \) nodes and random weights, corresponding to each configuration file with \( n \) frames. In this graph the random generated weights represent and simulate the compression ratio of a destination frame based on a dictionary frame by LZFF compression technique.

Having compared our algorithm with Hauck and Li, we noticed some advantages:

- We proposed a new compression method called LZFF, the compression ratio of which is higher than LZSS and is very useful for this application.
- We designed a hardware model corresponding to our approach that speeds up the decompression process which makes two optimizations. These optimizations are performed to delete some delay and configuration overhead for readback operations and constraint on the usage of the Block Select RAMs. Therefore we calculated the number of readback operations and required Block Select RAM for 4 configuration trees by optimizations of our algorithm and Hauck and Li’s algorithm. The results of these calculations are shown in Table 2. As can be seen, by increasing the nodes of configuration trees the variations are increased.

We developed a Genetic Algorithm approach for finding the optimal configuration sequence of frames to reduce the reconfiguration overhead, while Hauck and Li’s algorithm only focused on maximum similarity (Fig. 9). However, the sequence generated by our approach keeps maximum similarity within bitstream with minimum number of readbacks and minimum Block Select RAMs usage.

We implemented our algorithm in C++ on a personal computer Pentium III 800 MHZ, 128 MB RAM. All experiments are performed on various simulated configuration files. The simulation of a configuration file is begun with the Inter-Frame Similarity Graph building stage in our algorithm (Fig. 5). In other words, we generated a complete directed weighted graph with \( n \) nodes and random weights, corresponding to each configuration file with \( n \) frames. In this Graph the random generated weights represent and simulate the compression ratio of a destination frame based on a dictionary frame by the LZFF compression technique.

In order to evaluate the performance of our proposed Genetic Algorithm in finding the optimal configuration tree, we identified four experimental variables—encoding, crossover, mutation, and frames of the configuration file. We considered two encoding methods (determinant and Prüfer), three crossover methods (one-point, two-point, and uniform), two mutation methods (insert and exchange), and four configuration files of varying frames (20, 40, 60, 80). Hence, an experimental design with 48 cells (2 \( \times \) 3 \( \times \) 2 \( \times \) 4) was used to represent the combinations of all the factors. For each cell, ten data sets were generated and their average has been reported. In total, there were 480 data points for the experiment, 48 cells with ten data points in each cell.

The results of the experiments are shown in Table 3. The values in the cells of this table report the configuration overhead and processing time. Each configuration overhead is generated by applying corresponding combination of Genetic Algorithm factors on the corresponding configuration file with \( n \) frames. Also, the

<table>
<thead>
<tr>
<th>Configuration tree with 20 nodes</th>
<th>Configuration tree with 40 nodes</th>
<th>Configuration tree with 60 nodes</th>
<th>Configuration tree with 80 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of readback operations</td>
<td>Number of block select RAM usages</td>
<td>Number of readback operations</td>
<td>Number of block select RAM usages</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---------------------------------</td>
<td>---------------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>Hauck and Li’s algorithm</td>
<td>11 2</td>
<td>23 3</td>
<td>33 3</td>
</tr>
<tr>
<td>Our algorithm</td>
<td>4 1</td>
<td>7 2</td>
<td>12 2</td>
</tr>
</tbody>
</table>

Table 2
The comparison of number of readback operations and block select RAM usages
processing time represents the required time for generation of a configuration overhead on a Pentium III 800 MHZ. In this table the columns of the table represent the frames of the configuration file and the rows show the various combinations of Genetic Algorithm factors.

The highlighted cells represent the best solutions. As can be seen, most of the best solutions belong to the combination of determinant encoding, uniform crossover, and exchange mutation (E1C3M1). Therefore, in this research we selected E1C3M1 combination as the best combination of experimental factors. Also, one of the most important results of our experiment is that in this application the determinant encoding is better than Prüfer encoding. In other words, the generated configuration overhead and the consumption time of the determinant encoding are far less than the Prüfer method.

### 7.1. Control parameters

The population size, crossover rate, mutation rate, and halting criterion are important control parameters for GA. Based on selected factors (E1C3M1) we have tuned the proposed Genetic Algorithm. In the following sections we explain parameter settings.

#### 7.2. Population size

The population size indicates how many chromosomes exist in population (in one generation). If there are too few chromosomes, GA has few possibilities to perform crossover and only a small part of the search space is explored. On the other hand, if there are too many chromosomes, GA slows down. Research shows that after some limit (which depends mainly on encoding and the problem) it is not useful to increase population size, because it does not make the problem solving process faster. Therefore, satisfactory results can be obtained with a much smaller population size [7].

In our algorithm we have evaluated the effect of various population sizes such as 25, 50, 75, and 100 over different configuration files with 20, 40, 60, 80, and 150 frames. The population with 100 chromosomes causes the lowest configuration overhead in comparison with other population sizes. As a result, the population size was fixed at 100.

#### 7.3. Crossover rate

Crossover rate indicates how often crossover might be carried out. If there is no crossover the offspring is the exact copy of the parents. If there is a crossover, offspring is made from parts of parents’ chromosome. If
crossover probability is 100%, then all offsprings are made by crossover. If it is 0%, a whole new generation is made from the exact copies of chromosomes from old population (but this does not mean that the new generation is the same!). Crossover is made in hope that the new chromosomes will have good parts of the old chromosomes and maybe the new chromosomes will be better. However it is good to leave some parts of the population to survive to the next generation.

In our algorithm we have evaluated the effect of various crossover rate such as 25%, 50%, 80%, and 100% over different configuration files with 20, 40, 60, 80, and 150 frames. The crossover rate 100% causes the lowest configuration overhead in comparison with other crossover rates. As a result, the crossover rate was fixed at 100% ($p_c = 1.0$).

7.4. Mutation rate

Mutation rate indicates how often the parts of the chromosomes will be mutated. If there is no mutation, offspring is taken after crossover (or copy) without any change. If mutation is performed, a part of chromosome will be changed. If mutation rate is 100%, the whole chromosome will be changed, if it is 0%, there will be no change. Mutation is made to prevent falling GA into local extreme, but it should not occur very often, because then GA will in fact change to random search. In our algorithm we have evaluated the effect of various mutation rates such as 5%, 25%, 50%, 75%, and 100% over different configuration files with 20, 40, 60, and 80 frames. A mutation rate of 100% causes a lowest configuration overhead in comparison with other mutation rates. But based on our experiments, it became apparent that by increasing the mutation rate, the number of iterations and time of convergence of GA will be increased intensely. So it is better to select a trade-off between configuration overhead and number of iterations. By analyzing the results, we find out that despite the slight increase of the number of iterations, a mutation rate of 25% reduces the configuration overhead efficiently. As a result, the mutation rate was fixed at 25% ($p_m = 0.25$).

7.5. Halting criterion

For a configuration file with many frames, it is time-consuming to obtain the optimal configuration tree with maximum similarity and minimum number of readback operations and Block Select RAMs usages. In other words, setting the fitness convergence as the convergence criterion for the proposed Genetic Algorithm would lead to a large increase in convergence time for large configuration files. This could be overcome by setting an appropriate convergence criterion.

Based on our experiments and by analyzing the process of convergence of the proposed Genetic Algorithm, we found that variations of configuration overheads are very trivial when the variance tends to zero. For example Fig. 20 shows the variance of convergence of the proposed Genetic Algorithm for a configuration file with 20 frames. As it can be seen, in iteration 32 the variance declines severely and drops to 5, whereas it reaches zero in iteration 46. On the other hand, the configuration overhead only changes 6 units between iterations 32 and 46 ($210_{32} - 204_{46} = 6$). Therefore by setting the tending the variance to zero as the convergence criterion, the proposed Genetic Algorithm will converged without a significant change in the final configuration overhead.

We also repeated our analysis for configuration files with 40, 60, and 80 frames. In this way, Fig. 21 shows the variance of convergence of the proposed Genetic Algorithm for a configuration file with 40 frames. As it can be seen, in iteration 57 the variance declines severely and reaches to 7, whereas it reaches to zero in iteration 96. On the other hand, the configuration overhead only changes 19 units between iterations 57 and 96 ($478_{57} - 459_{96} = 19$).

Also Fig. 22 shows the variance of convergence of the proposed Genetic Algorithm for a configuration file with 60 frames. As it can be seen, in iteration 80 the variance declines severely and reaches 9, whereas it reaches zero in iteration 94. On the other hand, the configuration overhead only changes 10 units between iterations 80 and 94 ($687_{80} - 677_{94} = 10$).

Finally Fig. 23 shows the variance of convergence of the proposed Genetic Algorithm for a configuration file with 80 frames. As can be seen, in iteration 95 the variance declines severely and reaches 9, whereas it
Fig. 20. Variance of convergence of proposed Genetic Algorithm for a configuration file with 20 frames.

Fig. 21. Variance of convergence of proposed Genetic Algorithm for a configuration file with 40 frames.
Fig. 22. Variance of convergence of proposed Genetic Algorithm for a configuration file with 60 frames.

Fig. 23. Variance of convergence of proposed Genetic Algorithm for a configuration file with 80 frames.
reaches zero in iteration 118. On the other hand, the configuration overhead only changes 33 units between iterations 95 and 118 (1218\textsubscript{95} − 1185\textsubscript{118} = 33).

Regarding the above results and based on other repeated experiments we set a least lower bound for variance as the convergence criterion for the proposed algorithm. In this way we selected 10 as the least lower bound.

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

References

Farshid Farshadjam received his B.Sc. in Computer Engineering from Islamic Azad University, Southern-Tehran (IAU-ST), Iran in 1998, and his M.Sc. from Iran University of Science and Technology (IUST), Tehran, Iran in 2003. He is a faculty member of Computer Engineering and Information Technology at Islamic Azad University, Garmsar (IAU-G). Prior to joining IAU-G in 2004, he was a lecturer at IAU-ST working in the area of Wireless and embedded Systems. His research interests are in computer system architectures, wireless networks, Run-Time Reconfigurable systems (RTR), and evolutionary algorithms.

Mehdi Dehghan received his B.Sc. in Computer Engineering from Iran University of Science and Technology (IUST), Tehran, Iran in 1992, and his M.Sc. and Ph.D. from Amirkabir University of Technology (AUT), Tehran, Iran in 1995, and 2001, respectively. He is an assistant professor of Computer Engineering and Information Technology at Amirkabir University of Technology (AUT). Prior to joining AUT in 2004, he was a research scientist at Iran Telecommunication Research Center (ITRC) working in the area of Quality of Service provisioning and Network Management. His research interests are in high speed networks, wireless networks, fault tolerant computing, and distributed systems.

Mahmood Fathy has received his B.Sc. in Electronic Engineering from Iran University of Science and Technology (IUST), Tehran, Iran in 1983, and his M.Sc. in microprocessor Engineering from Bradford university (UK) in 1986, and his Ph.D. from University of Manchester (UMIST, UK) in 1991. At present, he is an associate professor in computer Engineering Department, Iran university of science and technology. He Has published more than 130 papers in the area of image processing in traffic, computer architecture for real-time image processing, and computer networks. His current research interests are: QoS in wireless ad hoc networks including video transmission, image processing applications and hardware in intelligent transportation systems including RTR systems, and computer system architecture.

Majid Ahmadi has received his B.Sc. degree in Electrical Engineering from Arya Mehr University in Tehran, Iran and the D.I.C. and Ph.D. degree from Imperial College of Science and Technology, London, England, in 1970 and 1977 respectively. He is currently a University Professor in Electrical and Computer Engineering at the University of Windsor. He is also the Director of the Research Centre for Integrated Microsystems. His areas of interest are Design Stability and Implementation of 1-D and 2-D Digital Filters, Pattern Recognition and Machine Vision, Neural Network Architectures, Computer Arithmetics and VLSI Implementation. He has published over 300 papers in these areas. He is Fellow of the IEEE and IEE.