

Configuration Compression for Virtex FPGAs

Zhiyuan Li
Motorola Labs,
Motorola Inc.
Schaumburg, IL 60196 USA
azl086@motorola.com

Scott Hauck
Department of Electrical Engineering
University of Washington
Seattle, WA 98195 USA
hauck@ee.washington.edu

Abstract

Although run-time reconfigurable systems have been shown to achieve very high performance, the speedups over traditional microprocessor systems are limited by the cost of configuration of the hardware. Current reconfigurable systems suffer from a significant overhead due to the time it takes to reconfigure their hardware. In order to deal with this overhead, and increase the compute power of reconfigurable systems, it is important to develop hardware and software systems to reduce or eliminate this delay. In this paper, we explore the idea of configuration compression and develop algorithms for reconfigurable systems. These algorithms, targeted to Xilinx Virtex series FPGAs with minimum modification of hardware, can significantly reduce the amount of data needed to transfer during configuration. In this work we have extensively investigated current compression techniques, including Huffman coding, Arithmetic coding and LZ coding. We have also developed different algorithms targeting different hardware structures. Our readback algorithm allows certain frames to be reused as a dictionary. In addition, we have developed frame reordering techniques that better uses the regularities by shuffling the sequence of the configuration. We have also developed a wildcard approach that can be used for true partial reconfiguration. The simulation results demonstrate that a factor of 4 compression ratio can be achieved.

Introduction

FPGAs are often used as powerful hardware for applications that require high-speed computation [Compton02]. One major benefit provided by FPGAs is the ability to reconfigure during execution. However, the advantages of run-time reconfiguration do not come without a cost. By requiring multiple reconfigurations to complete a computation, the time it takes to reconfigure the FPGA becomes a significant concern. The serial-shift configuration approach, as its name indicated, transfers all programming bits into the FPGA in a serial fashion. Recent devices have moved to cutting-edge technology, resulting in FPGAs with over one million gates. The configuration's size for such devices is over one megabyte [Xilinx00]. It could take milliseconds to seconds to transfer such a large configuration using the serial-shift approach.

Many techniques have studied to reduce the configuration overhead. These include configuration prefetching [Hauck98], configuration caching [Li00], and configuration compression. Configuration compression that can reduce the total number of write operations to load

a configuration [Hauck99, Li99]. Unfortunately, most of the previous compression techniques cannot be applied to the new generation FPGA such as Xilinx Virtex series [Xilinx00] with millions of gates. A LZ-based approach [Dandalis01] is applicable to any SRAM-based FPGA. However, without considering the individual features within the configuration bitstream this approach does not compress the bitstream efficiently. In this paper, we propose compression approaches that work efficiently on Xilinx Virtex devices.

The goal of configuration compression for reconfigurable systems is to minimize the amount of configuration data that must be transferred. Configuration compression is performed at compile-time. Once compressed, the bit-streams are stored in off-chip memory. During reconfiguration at run-time, the compressed bit-stream is transferred onto the reconfigurable device and then decompressed. The processes of compression and decompression are shown in Figure 1.

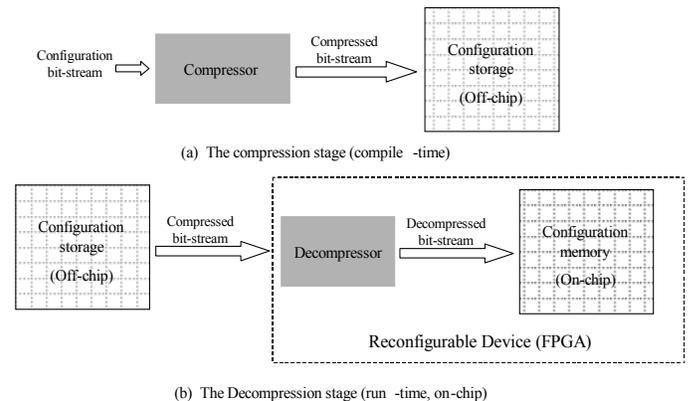


Figure 1: The original configuration data is compressed at compile-time (a). When reconfigurations occur, the compressed data is transferred to the decompressor on the reconfigurable device (b).

As can be seen in Figure 1, two issues must be resolved for configuration compression. First, an efficient compression algorithm must be developed. Second, since decompression is performed on-chip, building a decompressor should not result in significant hardware overhead.

Furthermore, any configuration compression technique must satisfy the following two conditions: (1) the circuitry generated from the decompressed bit-stream must not cause any damage to reconfigurable devices, and (2) the circuitry generated must result in the same outputs as those produced by circuitry generated from the original

configuration data. Consequently, most configuration compression research does not involve lossy techniques since any information loss in a configuration bit-stream may generate undesired circuitry on reconfigurable devices, and, even worse, may severely damage the chips.

Lossless compression techniques satisfy the above conditions naturally, because the decompressed data is exactly the same as the original configuration data. Lossless data compression is a well-studied field, with a variety of very efficient coding algorithms. However, applying these algorithms directly may not significantly reduce the size of the configuration bit-stream, because a number of differences exist between configuration compression and general data compression.

Configuration Compression Vs. Data Compression

The fundamental strategy of compression is to discover regularities in the original input and then design algorithms to take advantage of these regularities. Since different datatypes possess different types of regularities, a compression algorithm that works well for a certain data input may not be as efficient as it is for other inputs. For example, Lempel-Ziv compression does not compress image inputs as effectively as it does text. Therefore, in order to better discover regularities within a certain datatype, a specific algorithm must be developed. Existing lossless algorithms may not be able to compress configuration data effectively, because those algorithms cannot exploit potential specific regularities within configuration bit-streams.

Since decompression is performed on a specific architecture of a specific device can have a significant impact on compression algorithm design. Lossless data compression algorithms do not consider this architecture factor, causing the following problems:

- (1) Significant hardware overhead can result from building the decompressor on-chip. For example, a dictionary-based approach, Lempel-Ziv-Welch coding requires a significant amount of hardware to maintain a large lookup table during decompression.
- (2) The decompression speed at run-time may offset the effectiveness of the compression. For example, in Huffman compression, each code word is decompressed by scanning through the Huffman tree. It is very hard to pipeline the decompression process, and therefore it could take multiple cycles to produce a symbol. As the result, the time saved from transferring compressed data is overwhelmed by slow decompression.
- (3) Certain special on-chip hardware that can be used as decompressor may be wasted. For example, wildcard registers on the Xilinx 6200 series FPGAs can be used as decompressors. Unfortunately, no existing algorithm has been developed to take advantage of this special feature.

Realizing the unique features required for configuration compression, we have focused on exploring regularity and developing proper compression techniques for various devices. However, any technique will be limited if it can merely apply to one device. Therefore, our goal is to investigate the characteristics of different configuration architecture domains, and develop efficient compression algorithms for a given domain. In order to find the best approach to reduce the size of the configuration file, we will consider general-purpose compression techniques such as Huffman, Arithmetic and Lempel-Ziv coding, as well as a wildcarded approach.

Xilinx Virtex FPGAs

Each Virtex [Xilinx00] 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 bit-stream. Configuration bit-streams that contain a mix of commands and data can be read and written through one of the configuration interfaces on the device. A simplified block diagram of a Virtex FPGA is shown in Figure 1.

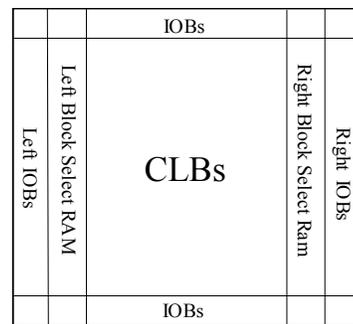


Figure 2: Virtex architecture.

The Virtex configuration memory can be visualized as a rectangular array of bits. The bits are grouped into vertical frames that are one bit wide and extend from the top of the array to the bottom. A *frame* is the atomic unit of configuration, meaning that it is the smallest portion of the configuration memory that can be written to or read from. Frames are grouped together into larger units, called *columns*. In Virtex devices, there are several different types of columns, including one center column, two IOB columns, multiple block RAM columns, and multiple CLB columns. As shown in Figure 3, each frame sits vertically, with IOBs on the top and bottom. For each frame, the first 18 bits control the two IOBs on the top of the frame, then 18 bits are allocated for each CLB row, and another 18 bits control the two IOBs at the bottom of the frame. The frame then contains enough “pad” bits to make it an integral multiple of 32 bits.

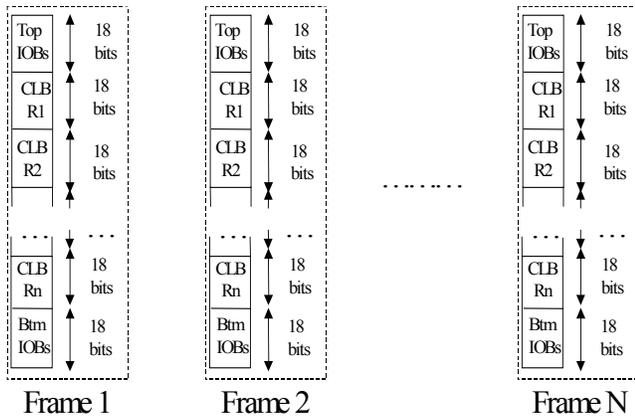


Figure 3: Virtex frame organization.

The configuration for the Virtex device is done through the *Frame Data Input Register (FDR)*. The FDR is essentially a shift register into which data is loaded prior to transfer to configuration memory. Specifically, given the starting address of the consecutive frames to be configured, configuration data for each frame is loaded into the FDR and then transferred to the frames in order. The FDR allows multiple frames to be configured with identical information, requiring only a few cycles for each additional frame, thus accelerating the configuration. However, if even one bit of the configuration data for the current frame differs from the previous frame, the entire frame must be reloaded.

Algorithms Overview

As we mentioned, well-known techniques -- including Huffman [Huffman52], Arithmetic [Witten87] and LZ [Ziv77] coding -- are very efficient for general-purpose compression, such as text compression. However, without considering features of the bit-stream, applying these techniques directly will not necessarily reduce the size of the configuration file. Given the frame organization described above, it is likely that traditional compression will either miss or destroy the regularities contained in the configuration files. For example, the commercial tool gzip achieves a compression factor of 1.85 in our benchmark set, much less than is achievable.

In this work, we will consider general-purpose compression approaches including Huffman, Arithmetic and Lempel-Ziv coding because of their proven effectiveness. In addition, we will extend the wildcard approach used for Xilinx 6200 [Hauck99] bit-stream compression. Before we discuss the details of our compression algorithms, we will first analyze the potential regularities in the configuration files.

Regularity Analysis

Current Virtex devices load whole frames of data at a time. Because of the similarity of resources in the array, we can expect some regularity between different frames of data. We call this similarity *inter-frame* regularity. In

order to take advantage of this regularity, the frames containing the same or similar configuration data should be loaded consecutively. For example, an LZ77 compression algorithm uses recently loaded data as a fixed-sized dictionary for subsequent writes, and by loading similar frames consecutively, the size of the configuration files can be greatly reduced. The current Virtex frame numbering scheme, where consecutive frames of a column are loaded in sequence, can be a poor choice for compression. After analyzing multiple configuration files, we discovered that the N th frame of all columns are more likely to contain similar configuration data since they control identical resources. Therefore, if we clustered together all of the N th frames of the columns in the architecture, we can achieve a better compression ratio. Of course, changing the order of the frames will incur an additional overhead by providing the frame address, but the compression of frame data may more than compensate for this overhead. Note that Huffman and Arithmetic coding are probability-based compression approach, meaning that the sequence that the configuration data is written will not affect the compression ratio.

Regularity within frames may be as important as regularity between frames. This *intra-frame* regularity exists in circuits that contain similar structures between rows. To exploit this regularity we will modify the current FDR with different frame buffer structures and develop the corresponding compression algorithms. For Lempel-Ziv compression, the shift-based FDR fits the algorithm naturally. However, extending the size of the FDR structure to a larger window can provide even greater compression ratios, though this must be balanced against potential hardware overheads. For the wildcarded approach, the structure of the Wildcard Registers used in Xilinx 6200 can be applied to the FDR of Xilinx Virtex FPGAs to allow multiple locations within the FDR to be written at the same time.

Symbol Length

Even though the configuration bit-stream is packed with 32-bit words for the Virtex devices, much of the regularity will be missed if the symbol length is set to 32-bit or other powers of two. As was shown in Figure 3, each CLB row within a frame is controlled by an 18-bit value, and the regularities we discussed above exist in the 18-bit fragments rather than 32-bit ones. In order to preserve those regularities we will break the 32-bit original configuration bit-stream. In addition to regularity, two other factors are considered to determine the length of the basic symbol. First, for Lempel-Ziv, Arithmetic and Huffman coding, the length of the symbol could affect the compression ratio. If the symbol is too long, the potential intra-symbol similarities will likely be overwhelmed. On the other hand, very short symbols, though retaining all the similarities, will significantly increase coding overhead. Second, since decompression is done at run-time, the

potential hardware cost should be considered. For example, both Huffman and Arithmetic coding are probability-based approaches which require that the probabilities of symbols be known during decompression. Retaining long symbols and their probabilities on-chip could consume significant hardware resources. In addition, transferring the probability values to the chip could also represent an additional configuration overhead.

As discussed above, using 18-bit symbols will retain the regularities in the configuration bit-stream. However, for Huffman and Arithmetic coding, the probabilities of 2^{18} symbols need to be transferred and then retained on-chip to correctly decompress the bit-stream. Clearly, this is not possible to implement and will increase configuration overhead. Therefore, we choose to use 6-bit or 9-bit symbols for Huffman, Arithmetic and Lempel-Ziv compressions. Using 6-bit or 9-bit symbols will preserve the potential regularities in the bit-streams and limit additional overheads.

Notice that the number of bits in the 32-bit words packed in each frame may not necessarily be a multiple of six or nine. Therefore, if we simply take the bit-streams and break them into 6-bit or 9-bit symbols, we will likely destroy *inter-frame* regularity. To avoid this, during the compression stage we will attach the necessary pad bits to each frame to make it a multiple of six or nine. This represents a pre-processing step for each of the compression algorithms.

Huffman coding

The goal of Huffman coding is to provide shorter codes to symbols with higher frequency. Huffman coding assigns an output code to each symbol, with the output codes being as short as one bit or considerably longer than the original symbols, depending on their probabilities. The optimal number of bits to be used for each symbol is $\log_2(1/p)$, where p is the probability of a given symbol. The probabilities of symbols are sorted, and a prefix binary tree is built based on the sorted probabilities, with the highest probability symbol at the top and the lowest at the bottom. Scanning the tree will produce the Huffman code. Figure 4 shows a set of symbols (a) and its corresponding Huffman tree (b). Given a string "XILINX" the resultant Huffman code is 1110110010111, using 13 bits.

Huffman compression for Virtex devices consists of two simple steps:

1. Convert the input bit-stream into a symbol stream.
2. Perform Huffman coding over the symbol stream.

The problem with this scheme lies in the fact that the Huffman codes must be an integral number of bits long. For example, if the probability of a symbol is $1/3$, the optimum number of bits to code that symbol is around 1.6 . Since Huffman coding requires an integral number of bits

to the code, assigning a 2-bit symbol leads to a longer compressed code than is theoretically possible.

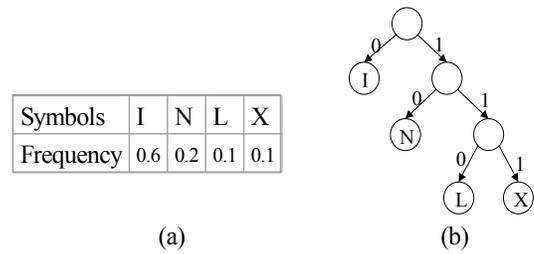


Figure 4: An example of Huffman coding. A set of 4 symbols and their frequencies are shown in (a). The corresponding Huffman tree is shown in (b).

Another factor that needs to be considered is decompression speed. Since each code word is decompressed by scanning through the Huffman tree, it is very hard to pipeline the decompression process. Therefore it could take multiple cycles to produce a symbol. Also, it is difficult to parallelize the decoding process, because Huffman is a variable-length code.

Arithmetic Coding

Unlike Huffman coding, which replaces each input symbol by a code word, Arithmetic coding takes a series of input symbols and replaces it with a single output number. The symbols contained in the stream may not be coded to an integral number of bits. For example, a stream of five symbols can be coded in 8 bits, with 1.6-bit average per symbol. Like Huffman coding, Arithmetic coding is a statistical compression scheme. Once the probabilities of symbols are known, the individual symbols are assigned to an interval along a probability line, and the algorithm works by keeping track of a high and low number that bracket the interval of the possible output number. Each input symbol narrows the interval. As the interval becomes smaller, the number of bits needed to specify it grows. The size of the final interval determines the number of bits needed to specify a stream. Since the size of the final interval is the product of the probabilities of the input stream, the number of bits generated by Arithmetic coding is equal to the entropy. Figure 5 shows the process of Arithmetic coding for string "XILINX" over the same symbol set used for Huffman coding. The generated code is 11110011011, two bits shorter than the Huffman code.

Note that the basic idea described above is difficult to implement, because the shrinking interval requires the use of high precision arithmetic. In practice, mechanisms for fixed precision arithmetic have been widely used.

The Arithmetic compression for Virtex devices consists of two steps:

1. Convert the input bit-stream into a symbol stream.
2. Perform the fixed-precision Arithmetic coding over the symbol stream.

The problem with this algorithm is that Arithmetic coding considers the symbols to be mutually unrelated (independent). However, the regularities existing in the configuration bit-stream may cause certain symbols to be related to each other. Therefore, this approach may not be able to yield the best solution for configuration compression. One solution to this problem is to combine multiple symbols together and discover the probabilities of the combined symbols. However, this will cause additional overhead by transferring and retaining a significant amount of probability values.

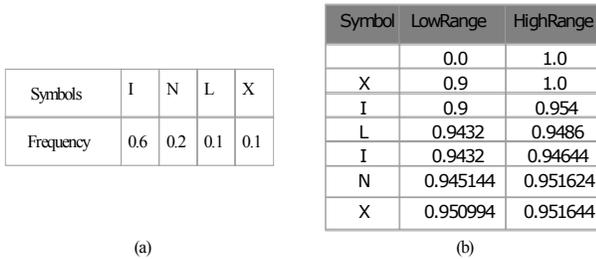


Figure 5: An example of Arithmetic coding. The same symbol set used for the Huffman coding is shown in (a). The coding process for string “XILINX” is shown in (b). The final interval, represented by the last row in (b), determines the number of bits needed.

Lempel-Ziv-Based (LZ) Compression

Recall that Arithmetic coding is a compression algorithm that performs better on a stream of unrelated symbols. LZ compression is an algorithm that more effectively represents groups of symbols that occur frequently. 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 that has been stored as part of the dictionary, the sub-stream can be represented in a shorter code word. The related symbols caused by the regularities in the configuration bit-stream make LZ algorithms an effective compression approach.

There are variations of LZ compression, including LZ77 [Ziv77], LZ78 [Ziv78] and LZW [Welch84]. In general, LZ78 and LZW will achieve better compression than LZ77 over a finite data stream. A lookup table is used to maintain occurred patterns for LZ78 and LZW. However, the excessive amount of hardware resources required to retain the table for LZ78 and LZW during decompression prevent us from considering those schemes for configuration compression. The “sliding window” compression of LZ77 requires only a buffer, and the shift-based FDR fits the scheme naturally, though hardware must be added to allow reading of specific frame locations during execution.

The LZ77 compression algorithm tracks the last n symbols of data previously seen, where n is the size of the sliding window buffer. When an incoming string is found to match part of the buffer, a triple of values corresponding

to the matching position, the matching length, and the symbol that follows the match is output. For example, in Figure 6, we find that the incoming string 3011 is in buffer position 3 with match length 4, and the next symbol is 0. So the algorithm will output codeword (3, 4, 0).

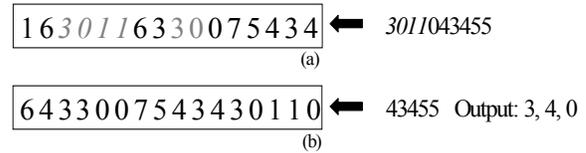


Figure 6: The LZ77 sliding window compression example. Two matches found are shown in gray. LZ77 selects the longer match “3011”, and the resultant codeword is (3, 4, 0). (a) shows the sliding window buffer and the input string before encoding. (b) shows the buffer and input string after encoding.

Standard LZ77 compression containing the three fields will reach entropy over an infinite data stream. However, for a finite data stream, this format is not very efficient in practice. For the case when no matching is found, rather than outputting the symbol, the algorithm will produce a codeword containing three fields, wasting bits and worsening the compression ratio. An extension of LZ77, called LZSS [Storer82], will improve coding efficiency. A threshold is given and if the matching length is shorter than the threshold, only the current symbol will be output. When 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, to achieve correct decompression, a flag bit is required for each code word to distinguish the two cases.

As mentioned above, the FDR in Virtex devices can be used as the sliding window buffer, and LZSS can take advantage of the intra-frame regularity naturally. However, since the current FDR can contain only one frame of configuration data, using it as the sliding window buffer will not take full advantage of inter-frame regularities. Thus, we modify the FDR to the structure shown in Figure 7. As can be seen in Figure 7, the bottom portion of the modified FDR, which has same size as the original FDR, can transfer data to the configuration memory. During decompression the compressed bit-stream is decoded and then fed to the bottom of the modified FDR. Incoming data will be shifted upwards in the modified FDR. Configuration data will be transferred to the specified frame once the bottom portion of the modified FDR is filled with newly input data. In addition, configuration data that is written to the array can be reloaded to the bottom portion of the modified FDR. This lets a previous frame be reused as part of the dictionary, and the inter-frame regularity is better utilized. Specifically, before loading a new frame, we could first read a currently loaded frame from the FPGA array back to the frame buffer, and then load the new frame. By picking a currently loaded frame that most resembles the

new frame, we may be able to exploit similarities to compress this new frame.

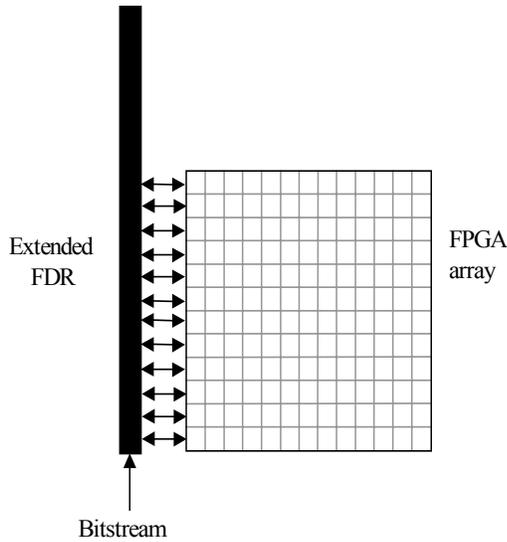


Figure 7: The hardware model for LZ77 compression.

While this technique will be slow due to delays in sending data from the FPGA array back to the FDR, there may be ways to accelerate this with moderate hardware costs. In current Virtex devices, the data stored in the Block Select RAMs can be transferred to logic very quickly. We can exploit this feature by slightly modifying the current hardware to allow the values stored in the Block Selected RAMs to be quickly read back to the modified FDR. By providing the fast readback from only the Block Select RAMs, we efficiently use the Block RAMs as caches during reconfiguration to hold commonly requested frames without significant hardware costs. Also, the size of the modified FDR must be balanced against the potential hardware cost. In our research, we allow the modified FDR to contain two frames of data. This will not significantly increase hardware overhead, yet it will utilize the regularities in the configuration stream [richmond01].

Finding regularities in a configuration file is a major goal. LZ compression performs well only in the case where common strings are found between the sliding window buffer and incoming data. This requires quite a large buffer to find enough matches for general data compression. However, for configuration compression, the hardware costs will restrict the size of the sliding window buffer. Thus, performing LZ compression directly over the datastream will not render the desired result. In order to make compression work efficiently for a relatively small buffer, we need to carefully exploit the data stream, finding regularities and intelligently rearranging the sequence of frames to maximize matches. In the following sections, we discuss algorithms that apply LZSS compression, targeting the hardware model described above. These algorithms are all realistic but

require different amounts of hardware resources and thus provide different compression ratios.

The Readback Algorithm

The goal of configuration compression is to take advantage of both inter-frame and intra-frame regularities. In the configuration stream, some of the frames are very similar. By configuring them consecutively, higher compression ratios can be achieved. An FPGA’s readback feature allows us to read back the frame that most resembles the new frame into the modified FDR and thus reuse it as a dictionary. This increases the number of matches for LZSS, permitting us to fully use regularities within the bit-stream. For example, in Figure 8, four frames are to be configured, and frames (b), (c) and (d) are more like (a) than like each other. Without readback, inter-frame regularities between (c), (d) and (a) will be missed. However, with the fast readback feature, we can temporarily store frame (a) in the Block Select RAMs, reading it back to the modified FDR and using it as a dictionary when other frames are configured. This fast readback will significantly increase the utilization of inter-frame regularities with negligible overhead. Since the modified FDR is larger than the size of the frame, LZSS will be able to use intra-frame regularities naturally.

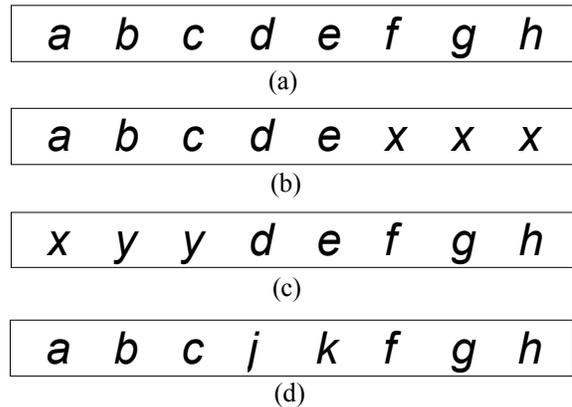


Figure 8: Example to illustrate the benefit of readback. (b), (c), and (d) resemble to (a). By reusing (a) as a dictionary, better compression can be achieved.

Discovering *inter-frame* regularities represents an issue that will influence the effectiveness of compression. Based on the hardware model we proposed above, the similarity between the frame in the modified FDR and the new incoming frame is the key factor for compression. More specifically, we seek to place a certain frame in the modified FDR so that it will most aid the compression of the incoming frame. In order to obtain such information, each frame is used as a fixed dictionary in a preprocessing stage, and LZSS is applied to all other frame, which are called *beneficiary* frames. Note that LZSS is performed without moving the sliding window buffer, meaning that the dictionary will not be changed. This approach excludes potential intra-frame regularities within each

beneficiary frame, providing only the inter-frame regularity information. The output code length represents the necessary writes for each beneficiary frame based on the dictionary, and shorter codes will be found if the beneficiary frame resembles the dictionary.

Once this process is over, a complete directed graph can be built, with each node standing for a frame. The source node of a directed weighted edge represents a dictionary frame, and the destination node represents a beneficiary frame. The weight of each edge denotes the inter-frame regularity between a dictionary frame and a beneficiary frame. One optimization performed is to delete the edges that present no inter-frame regularity between any two frames. Figure 9(a) shows an example of the inter-frame regularity graph.

Given an inter-frame regularity graph, our algorithm seeks an optimal configuration sequence that maximizes the inter-frame regularities. Specifically, we seek a subset of the edges in the inter-frame regularity graph such that every node can be reached and the aggregate edge weight is minimized. Solving this problem is equivalent to solving the *directed minimum spanning tree* problem, where every node has one and only one incoming edge, except for the root node. Figure 9(b) shows the corresponding optimal configuration sequence graph of Figure 9(a). In the configuration sequence graph, a frame with multiple children needs to be stored in Block Select RAMs for future readback. For example, in Figure 9(b), a copy of frame A will be stored in Block Select RAMs and read back to the modified FDR to act as a dictionary.

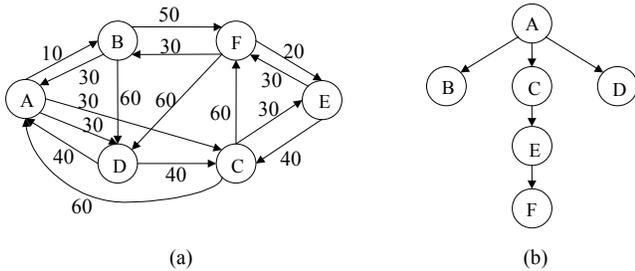


Figure 9: Seeking optimal configuration sequence. An inter-frame regularity graph is shown in (a). The corresponding optimal configuration sequence graph is shown in (b).

Now we present our Readback algorithm:

- 1 Convert the input bit-stream into a symbol stream.
- 2 For each frame, use it as a fixed dictionary and perform LZSS on every other frame.
- 3 Build an inter-frame regularity graph using the values computed in step 2.
- 4 Apply the *standard directed minimum spanning tree algorithm* [Chu65] on the inter-frame regularity graph to create the configuration sequence graph.

5. Perform pre-order traversal starting from the root. For each node that is being traversed:
 - 5.1. If it has multiple children, a copy of it will be stored in an empty slot of the Block Select RAMs.
 - 5.2. If its parent node is not in the modified FDR, read the parent back from the Block Select RAMs.
 - 5.3. Perform LZSS compression.
 - 5.4. If it is the final child traversed of the parent node, release the memory slot taken by the parent.

Step 2 measures inter-frame regularities between frames. Results are used to build the inter-frame regularity graph and the corresponding configuration sequence graph in Steps 3 and 4 respectively. The Pre-order traversal performed in Step 5 uses the parent frame of the currently loading frame as a dictionary for LZSS compression. Note that a copy of the currently loading frame will be stored in the Block Select RAMs if it has multiple children in the configuration sequence graph. Also, additional overhead from setting configuration registers will occur if frames to be configured are not contiguous.

One final concern for our Readback algorithm is the storage requirement for the reused frames. Analyzing configuration sequence graphs, we found that although a large number of frames need to be read back, they are not all required to be held in the Block Selected RAM at the same time, and they can share the same memory slot without conflict. For example, in Figure 10, both frame A and frame B need to be read back. Suppose the left subtree was configured first; then frame A will occupy a slot in the Block Selected RAMs for future readback. Once the configuration of the left tree is complete, the memory slot taken by frame A can be reused by frame B during configuration of the right sub-tree.

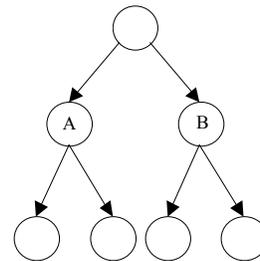


Figure 10: An example of memory sharing.

We have developed an algorithm using a bottom-up approach that accurately calculates the memory slots necessary. By combining it with our Readback algorithm, usage of the Block Select RAMs can be minimized. The details of the algorithm are as follows:

1. For each node in the configuration sequence graph, assign 0 to the variable V and number of children to C .
2. Put each node whose children are all leaves into a queue.
3. While the queue is not empty:
 - 3.1. Remove a node from the queue.
 - 3.2. If it has one child, $V = V_{child}$, else $V = \max(\text{largest } V_{child}, (\text{second largest } V_{child} + 1))$.
 - 3.3. For its parent node, $C = C - 1$. If $C = 0$, put the parent node into the queue.

Figure 11 shows an example of our Memory Requirement Calculation algorithm. At left is the original configuration sequence graph. At right shows the calculation of the memory requirement using a bottom-up approach. The number inside each node represents the number of memory slots necessary for configuring its sub-trees. As can be seen, only two memory slots are required for this 14-node tree. 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 for 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 equal that of configuring the largest sub-tree. Since the pre-order traverse will scan the left sub-trees before the right ones, we should readjust the configuration sequence graph to set each node in the sub-tree that requires the most memory as the rightmost sub-tree. In order to apply the memory minimization to our compression, we modify Step 4 of our readback algorithm as follows:

4. Apply the standard directed minimum spanning tree algorithm on the inter-frame regularity graph to create the configuration sequence graph. Perform the Memory Calculation algorithm, and the largest sub-tree for each node is set as the rightmost sub-tree.

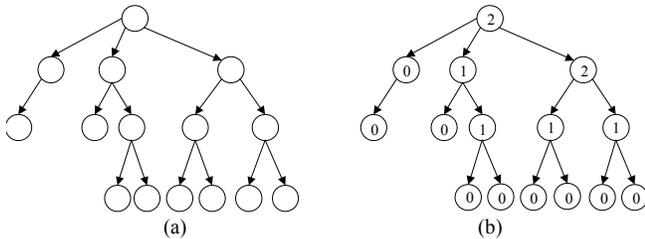


Figure 11: An example to illustrate our Memory Requirement Calculation algorithm. A configuration sequence graph is shown in (a), and the corresponding memory requirement calculation procedure is shown in (b).

Active Frame Reordering Algorithm

The Readback algorithm allows frames to be read back to the modified FDR to achieve effective compression. However, the delay and hardware alterations required for the Block Selected RAM readback may not be acceptable. Some applications may restrict the use of the Block Select RAMs. In order to take advantage of the regularities within the configuration bit-stream, we have developed a frame reordering algorithm that does not require frame readback.

As can be seen in our readback algorithm, frame reordering enhances compression by utilizing inter-frame regularities. This idea can still be applied to applications without the readback feature. In our readback algorithm, once the inter-frame regularity graph is built, a corresponding configuration sequence graph can be generated, and traversing the configuration sequence graph in pre-order can guarantee the maximum utilization of the regularities discovered. However, without frame readback, traversing the configuration sequence graph might not necessarily be the optimal solution, since parent nodes cannot be reused as a dictionary. Our Active Frame Reordering algorithm uses a greedy approach to generate a configuration sequence that allows each frame to be used as a dictionary only once. It still takes the inter-frame regularity graph as input. However, instead of using the directed MST approach to create a configuration sequence, a spanning chain will be generated using a greedy approach. The details of the algorithm are as follows:

1. Convert the input bit-stream into a symbol stream.
2. For each frame, use it as fixed dictionary and perform LZSS on every other frame.
3. Build an inter-frame regularity graph using the values that resulted from Step 2.
4. Put the two frames connected by the minimum weight edge into a set. Let H be the head and T be the tail of this edge.
5. While not all frames are in the set:
 - 5.1. For all incoming edges to H and outgoing edges from T , find the shortest one that connects to a frame not in the set. Put that frame into the set. The frame is set to H if the edge found is an incoming edge to H ; otherwise set the frame to T .
6. Perform LZSS compression on the chain discovered in Step 5.

The basic idea of the algorithm is to grow a spanning chain from the two ends. Step 5 finds a frame not in the chain with the shortest edge either coming into an end or going out from the other. This greedy process is repeated until all frames are put in the spanning chain. For

example, in Figure 9, the order of the frames to be put into the chain discovered by our algorithm is ABDFEC (the configuration sequence will be DABFEC). The cost of the sequence is 160, slightly larger than the optimal spanning chain (150). Starting from one end of the discovered spanning chain, LZSS can be performed to generate a compressed bit-stream.

Fixed Frame Reordering Algorithm

One simple algorithm is to reorder the frames such that the N th frame of each column is consecutive. Performing LZSS over the sequence generated by the simple reordering takes advantage of the regularities within applications. The overhead of setting the configuration registers can be eliminated using this fixed frame order, since the order that frames appear is fixed for all configurations.

Wildcarded Compression for Virtex

Since the Wildcard Compression [Hauck99] achieves good results for the Xilinx 6200 FPGAs, we would like to apply it to Virtex FPGAs. For Virtex configurations, multiple rows within a frame can contain the same configuration data. Instead of configuring them one by one, the wildcarded approach allows these rows to be configured simultaneously. To apply the wildcarded approach to Virtex, an address register and a wildcard register will be added as an augmented structure to the FDR. They will allow specified rows within the FDR to be configured.

For circuits with repetitive structures, multiple frames could be very similar, yet not completely identical. By allowing the FDR to be addressable, we take advantage of this inter-frame regularity. Instead of loading the whole frame, we can load only the differences between frames. For example, in Figure 12, two frames need to be configured, and the second frame has only three values different from the first one. In this case, only the configuration data for the 3 different values needs to be loaded. In addition, if the three different rows can be covered by a wildcard, one write is enough to configure the whole second frame. This structure will also support true partial reconfiguration. More specifically, for each frame to be reconfigured, rather than loading the entire frame, we can simply load the difference from the current configuration. Note that adding the Address Register and Wildcard Register represents additional hardware cost. Moreover, extra bits for the address and wildcard need to be transferred for every write.

The Wildcard algorithm consists of 2 stages. In the first stage we reorder similar frames so they will be configured consecutively. This creates a sequence in which the number of writes necessary for configuring each frame is greatly reduced. In the second stage, we find the wildcards covering the writes for each frame and thus further reduce the configuration overhead. The first stage

takes advantage of inter-frame regularities while the second stage focuses on intra-frame regularities.

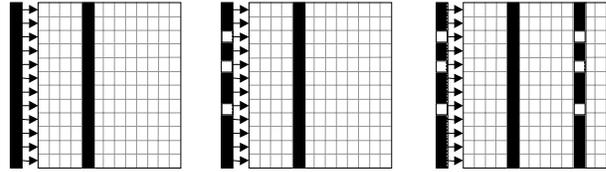


Figure 12: An example of inter-frame compression using addressable FDR.

In the first stage, we discover the number of non-matching rows between each pair of frames; the result indicates the extent of similarity between the frames. An undirected graph is built to keep track of the regularities, and a near optimal sequence needs to be discovered. Since each frame is configured exactly once, finding the sequence based on the regularity graph is equivalent to solving the traveling salesman problem. An existing algorithm is an approximation with a ratio bound of two for the traveling-salesman problem with triangle inequality [Lawler85]. Given a complete undirected graph $G = (V, E)$ that has a nonnegative integer cost $c(u, v)$ associated with each edge $(u, v) \in E$, cost function c satisfies the triangle inequality if for all vertices $u, v, w \in V$, $c(u, w) \leq c(u, v) + c(v, w)$. Since the differences between frames satisfy the triangle inequality, we can apply the approximation algorithm on our compression algorithm. The details of our Wildcard algorithm are as follows:

1. Convert the input bit-stream into an 18-bit symbol stream.
2. For each pair of frames, identify the different 18-bit symbols between them.
3. Build a regularity graph using the results from Step 2.
4. Perform the Approx-TSP-Tour algorithm [Lawler85] to determine the order of frames to be configured.
5. For each frame configuration, use the Wildcard algorithm to find the wildcards to cover the differences.

Results

All algorithms are implemented in C++ on a Sun Sparc Ultra 5 workstation and were run on a set of benchmarks collected from Virtex users. Detailed information about the benchmarks is shown in Table 1.

Figure 13 shows simulation results for compression approaches using 6-bit symbols; the wildcard approach uses 18-bit symbols. The left 10 benchmarks are automatically mapped and use more than 50% of the chip area. The ‘‘Geo. Mean’’ column is the geometric mean of the 10 benchmarks. The three right-most benchmarks are

either hand mapped or use only a small percentage of the chip area, and are included to demonstrate how hand mapping or low utilization affects compression. Figure 14 demonstrates simulation results for 9-bit symbols (The Wildcard algorithm is not shown, since it uses only 18-bit symbols.).

Table 1: Information for Virtex benchmarks.

Bench mark	Source	Device	Chip Utilization	Mapping
Mt1mem0	Rapid	400	>80%	Auto
Mt1mem1	Rapid	400	>80%	Auto
Mars	USC	600	Unknown	Auto
RC6	USC	400	Unknown	Auto
Serpent	USC	400	Unknown	Auto
Rijndael	USC	600	Unknown	Auto
Design1	HP	1000	>70%	Auto
Pex	North eastern	1000	93%	Auto
Glidergun	Xilinx	800	>80%	Hand
Random	Xilinx	800	>80%	Hand
U1pc	Xilinx	100	1%	Auto
U50pc	Xilinx	100	50%	Auto
U93	Xilinx	100	>90%	Auto

As can be seen in the figures, the readback algorithm performs better than other algorithms for both 6-bit and 9-bit cases for most of the benchmarks. This is because the Readback algorithm takes full advantage of inter-frame regularities within the configuration bit-stream by reusing

certain frames as dictionaries. Though they cannot fully utilize inter-frame regularities, the reordering techniques still provide fairly good results without using the Block Select RAMs as a cache. The Active Reordering algorithm performs better than the Fixed Reordering algorithm since active reordering can better use inter-frame regularities by actively shuffling the sequence of frames, while fixed reordering can utilize only the regularities given by the fixed sequence.

Surprisingly, although the wildcard approach can exploit both inter-frame and intra-frame regularities, it still yields worse compression ratios than the active reordering scheme for most of the benchmarks. There are several reasons for this. First, the wildcard approach requires address and wildcard specifications for each write, adding significant overhead to the bit-stream. The additional overhead overwhelms the benefits provided by the regularities within the applications. Second, the wildcard approach requires a comparison between the same rows of the given frames to discover inter-frame regularities. Consequently, the similarity the wildcard approach can discover is aligned in rows, and any unaligned similarities that benefit the LZ-based approaches will not help wildcards. For example in Figure 15, the Wildcard algorithm cannot discover the inter-frame regularity between frame A and frame B. However, the regularity can be exploited for LZ-based approaches. Third, the Wildcard approach requires that enough rows covered by a wildcard share the same configuration value to achieve better compression. However, even the XCV1000, which is a relatively large device, has only 64 rows and it is not likely to find enough rows covered by a wildcard that have the same configuration value. For many cases, each wildcard contains only one row, and the address/wildcard overhead is still applied.

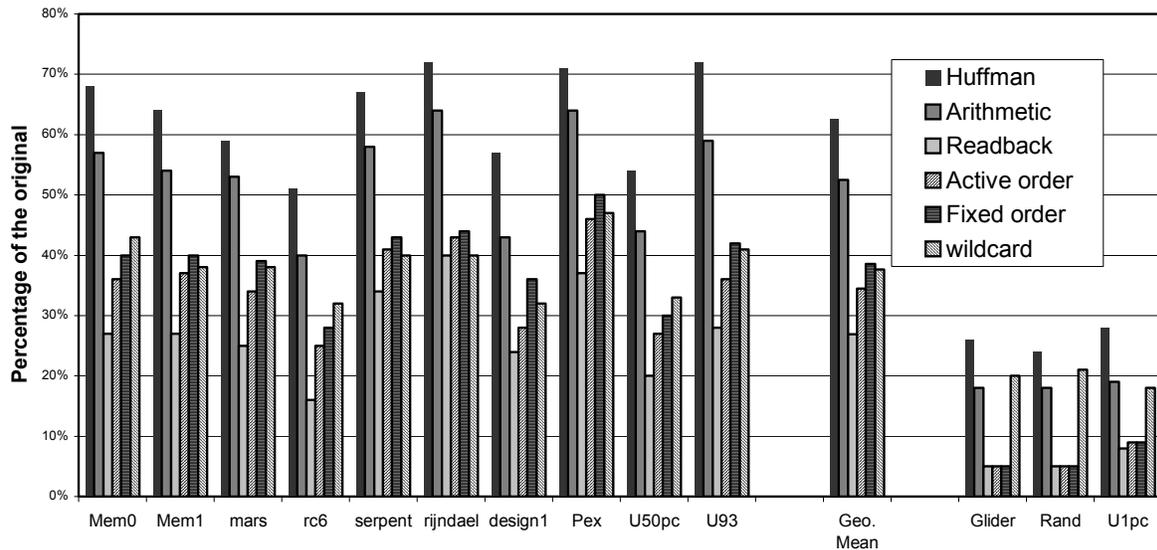


Figure 13: The simulation results for 6-bit symbol.

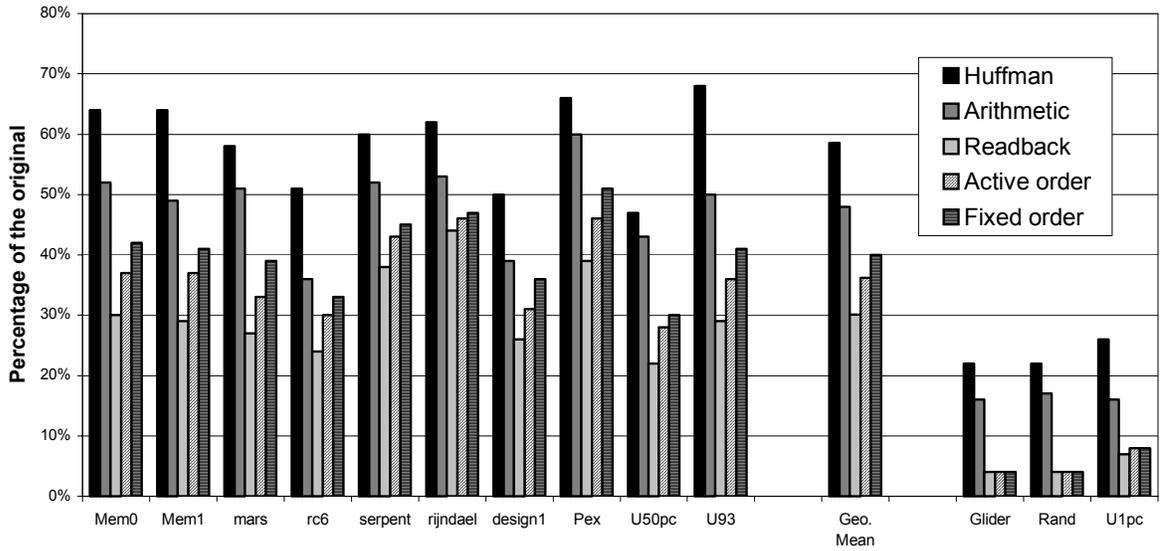


Figure 14: The simulation results for 9-bit symbol.

The probability-based Huffman and Arithmetic coding techniques perform significantly worse than other techniques, since they do not consider regularities within the bit-stream. The Huffman approach did worse than the Arithmetic approach, simply because of its inefficient coding method. Adding the fact that these two approaches require significant hardware for the decompressor, we will not consider using them for configuration compression.

	Frame A values	Frame B values
Row 1	1	2
Row 2	2	3
Row 3	3	4
Row 4	4	5
Row 5	5	6

Figure 15: Unaligned regularity between frames. The wildcard approach will miss this regularity, while the LZ-based approaches will take advantage of them.

Don't Cares

In an FPGA configuration it is possible to find configuration bits whose value is unimportant to the functioning of the circuit. For example, the configuration of unused logic blocks usually is unimportant. Our previous research [Li99] shows that with the help of these Don't Care bits within the configuration datastream, higher compression ratios can be achieved. Although Xilinx does not disclose the information necessary to discover Don't Cares in the Virtex applications, we can still evaluate the potential impact of the Don't Cares for Virtex compression. In order to make an estimate, we randomly turn some bits of the data stream into Don't Cares and bound the impact of Don't Cares on our Readback algorithm.

In practice, the discovered Don't Care bits need to be turned to '0' or '1' to produce a valid configuration bit-stream. The way that the bits are turned affects the frame sequence and thus the compression ratio. Finding the optimal way to turn the bits may take exponential time. We have used a simple greedy approach to turn these bits to create an upper-bound for our Readback algorithm. The configuration sequence graph is built taking into account the Don't Cares. We greedily turn the Don't Care bits into '0' or '1' to find the best matches. Note that once a bit is turned, it can no longer be used as a Don't Care. To discover the lower-bound, we do not turn the Don't Care bits; thus, they can be used again to discover better matches.

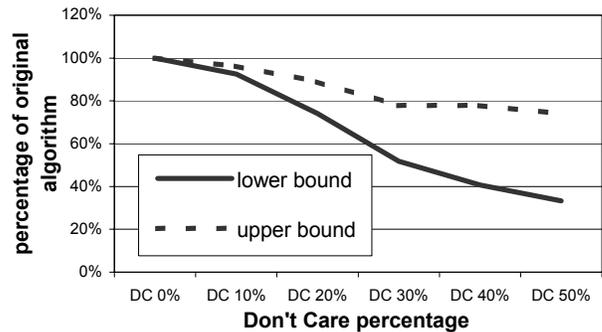


Figure 16: Estimate Result for Virtex bitstream with Don't Cares.

Figure 16 demonstrates the potential effect of Don't Cares over the benchmarks listed in Table 3.2. The X-axis is the percentage of the don't cares we randomly create and the Y-axis is the normalization over the results without considering Don't Cares. As can be seen in Figure 16, by using upper-bound approach a factor of 1.3 improvement can be achieved on applications containing 30% Don't Cares, while a factor of 2 improvement can be achieved using the lower-bound approach.

Hardware Costs

Since decompression must be performed on-chip, hardware costs for building decompressors must be evaluated to determine whether our compression algorithms are viable techniques. In this work we focus on the hardware implementation of the LZ decompressor because its compression algorithms outperform other approaches. We have implemented this LZ decompressor, and demonstrated the hardware cost is minimal. The overall increase in area is less than 1% for Virtex 1000 or larger devices [Richmond01].

Conclusions

One of the major problems in reconfigurable computing is the time and bandwidth overheads due to reconfiguration. This can overwhelm the performance benefits of reconfigurable computing, and reduce the potential application domains. Thus, reducing this overhead is an important consideration for these systems. In this paper we have researched current compression techniques, including Huffman coding, Arithmetic coding and LZ coding for the Virtex FPGA. We have also developed different algorithms targeting different hardware structures. Our Readback algorithm allows certain frames to be reused as a dictionary and sufficiently utilizes the regularities within the configuration bit-stream. Our Frame Reordering algorithms exploit regularities by shuffling the sequence of the configuration. The simulation results demonstrate that a factor of four compression ratio can be achieved. We believe the configuration compression algorithms we developed can be extended to any similar reconfigurable devices without significant modifications.

References

- [Chu65] Y. J. Chu and T. H. Liu. On the shortest arborescence of a directed graph. *Science Sinica*, v.14, pp.1396-1400, 1965.
- [Compton02] K. Compton, S. Hauck, Reconfigurable Computing: A Survey of Systems and Software, *ACM Computing Surveys*, Vol. 34, No. 2. pp. 171-210. June 2002.
- [Dandalis01] Andreas Dandalis, Viktor Prasanna. Configuration Compression for FPGA-based Embedded Systems. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2001.
- [Hauck98] S. Hauck. Configuration Prefetch for Single Context Reconfigurable Coprocessors. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65-74, 1998.
- [Hauck99] S. Hauck, Z. Li, E. Schwabe. Configuration Compression for Xilinx 6200 FPGA, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 18, No. 8, pp. 1107-1113, August, 1999.
- [Huffman52] D. A. Huffman. A Method for the Construction of Minimum Redundancy Codes. *Proceedings of the Institute of Radio Engineers* 40, pp 1098—1101, 1952.
- [Lawler85] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. The Traveling Salesman Problem. *John Wiley and Sons*, New York, 1985.
- [Li99] Z. Li, S. Hauck, Don't Care Discovery for FPGA Configuration Compression. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 91-100, 1999.
- [Li00] Z.Li, K. Compton, Scott Hauck. Configuration Caching Management Techniques for Reconfigurable Computing". *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 87-96, 2000.
- [Richmond01] Melany Richmond. A Lemple-Ziv based Configuration Management Architecture for Reconfigurable Computing. Master's Thesis, University of Washington, Dept. of EE, 2001.
- [Storer82] J.A. Storer, T. G. Szymanski. Data Compression via Textual Substitution. *Journal of the ACM*, 29:928-951, 1982.
- [Welch84] T. Welch. A technique for high-performance data compression. *IEEE Computer*, pp 8-19, June 1984.
- [Witten87] I. H. Witten, R. M. Neal, J. G. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, vol. 30, pp. 520-540, 1987.
- [Xilinx00] Xilinx, Inc. Virtex II Configuration Architecture Advanced Users' Guide. March, 2000.

[Ziv77] J. Ziv, and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, pp 337-343, May 1977.

[Ziv78] J. Ziv, and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, pages 530-536, September 1978.