IEEE Symposium on FPGAs for Custom Computing Machines, 2001.

Configuration Compression for Virtex FPGAs

Zhiyuan Li
Department of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208-3118 USA
zl@ece.nwu.edu

Scott Hauck
Department of Electrical Engineering
University of Washington
Seattle, WA 98195-2500
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 researched the current compression techniques, including the Huffman coding, the 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 and sufficiently utilize the regularities within the configuration bitstream. In addition, we have developed frame reordering techniques that better uses the regularities by shuffling the sequence of the configuration. We have also developed the wildcard approach that can be used for true partial reconfiguration. The simulation results demonstrate that a factor of 4 compression ratio can be achieved.

1. Introduction

FPGAs are often used as powerful hardware for applications that require high-speed computation. One major benefit provided by FPGAs is the ability to reconfigure during execution. For many applications, the systems need to be reconfigured frequently during run-time to exploit the full potential of using the reconfigurable hardware. By reducing the reconfiguration overhead, the performance of the system is improved. Many researches have studied techniques to reduce the configuration overhead. Some of these techniques include configuration prefetching [Hauck98a], configuration caching [Li00], and configuration compression. Configuration compression that can reduce the total number of write operations to load a configuration has been proved as an efficient technique to deal with the configuration overhead [Hauck98b, Hauck99, Li99]. Unfortunately, much of the previous compression researches 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.

Configuration compression, which is similar to data compression, takes advantage of regularity and repetitions existing in the data stream. Lossless data compression is a well-studied field with several very efficient coding techniques developed. 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.

2. Xilinx Virtex Series FPGA

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 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.

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 2, each frame sits vertically, with IOBs on the top and the 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.

The configuration for the Virtex device is done through the Frame Data Input Register (FDR). The FDR is essentially a shift register into which the data is loaded prior to transfer to configuration memory. More specifically, given the starting address of the consecutive frames to be configured the 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 is different from the previous frame the entire frame must be reloaded. In the sections that follow, we will present configuration compression techniques with minimum hardware modification that can take advantage of these regularities within the configuration datastream.

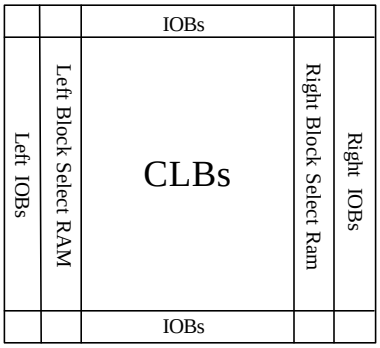

Figure 1. Virtex architecture.

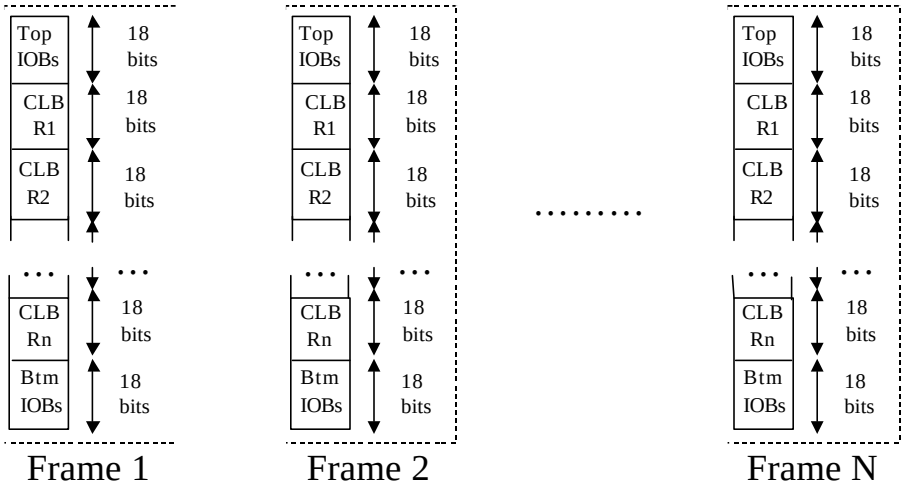

Figure 2. CLB frame organization.

3. Configuration Compression Algorithms

As we mentioned above, 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 the feature of the bitstream, applying these techniques directly will not necessarily reduce the size of the configuration file. Given the frame organization describe above, it is likely traditional compression will either miss or destroy the regularities contained in the configuration

files. For example, commercial tool gzip achieves a factor of 1.85 on our benchmark set, which is 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 our wildcard approach used for Xilinx 6200 bitstream compression. Before we discuss the details of our compression algorithms, we will first analyze the potential regularities in the configuration files.

3.1. 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 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 *Nth* frame of the columns are more likely contain similar configuration data since it controls identical resources. Therefore, if we clustered together all of the *Nth* 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 the frame data may more than compensate for this overhead. Note that Huffman and Arithmetic coding are probability based compression approach, meaning that the sequence of the configuration data is written will not affect the compression ratio.

Regularity within the frames may be as important as regularity between the 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 the potential hardware overheads. For our wildcarded approach, the structure of the wildcard registers used in Xilinx 6200 can be applied to the FDR to allow multiple locations within the FDR to be written at the same time.

3.2. Symbol Length

Even though the configuration bitstream 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 powers of 2. As shown in Figure 2, 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 original 32-bit configuration bitstream. Except for the 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 the coding overhead. Second, since the decompression is done at run-time, the potential hardware cost should be considered. For example, both Huffman and Arithmetic coding are probability based approaches and require that the probabilities of symbols be known during the decompression. For long symbols, retaining these symbols with their probabilities on chip could use up significant hardware resources. In addition, transferring the probability values to the chip could also represent additional configuration overhead.

As discussed above, using 18-bit symbols will retain the regularities in the configuration bitstream. However, for Huffman and Arithmetic coding, the probabilities of 2^{18} symbols need to be transferred and then kept on chip to correctly decompress the bitstream. Clearly, this is not possible to implement and will increase the 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 bitstreams and limit the additional overheads.

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

In the following sections, we will discuss the compression algorithms we developed based upon the above factors. Also, we will evaluate the performance of each algorithm on a set of representative benchmarks.

3.3. Huffman coding

The goal of the Huffman coding is to provide shorter codes to symbols with higher frequency. The Huffman coding assigns an output code to each symbol, with the output codes being as short as 1 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 probability symbol at the bottom. Scanning the tree will produce the Huffman code. The Huffman compression for Virtex simply consists of 2 steps:

1. Convert the input bitstream into a symbol stream.
2. Perform the 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 the Huffman coding requires an integral number of bits to the code, assigning the symbol to 2-bit leads to a longer compressed code than is theoretically possible.

Another factor that needs to be considered is the decompression speed. Since 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. Also, it is difficult to parallelize the decoding process because Huffman is a variable length code.

3.4. Arithmetic Coding

Unlike the Huffman coding, which replaces an input symbol by a code word, the Arithmetic coding completely bypasses that idea. Instead, it 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 5 symbols can be coded in 8 bits, with 1.6-bit average per symbol. Like Huffman coding, the 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 symbol of the input narrows the interval and 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 the arithmetic coding is equal to the entropy. 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 bitstream into a symbol stream.
2. Perform the fixed precision Arithmetic coding over the symbol stream.

The problem with this algorithm is the fact that Arithmetic coding considers the symbols to be mutually unrelated. However, the regularities existing in the configuration bitstream 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 accurate probabilities of combined symbols. However, this will cause additional overheads by transferring and keeping a significant amount of probability values. Another way to improve the performance is to calculate the probabilities of combined symbol by simply multiplying the probabilities of individual symbols. This dynamic approach will increase the precision of the interval without considering

the correlation between the symbols. However, performing additional multiplications at the decompression end will slow down the decompression speed.

3.5. Lempel-Ziv Based Compression

Recall that the Arithmetic coding is a compression algorithm that performs better on a stream of unrelated symbols. The LZ compression is an algorithm that represents groups of symbols that occur frequently more 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 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 bitstream make LZ algorithms an effective compression approach.

There are variations of LZ compression including LZ77, LZ78 and LZW. In general, LZ78 and LZW will achieve better compression than LZ77 over a finite data stream. A look-up table is used to maintain occurred patterns for LZ78 and LZW. However, the excessive amount of hardware resources required by retaining the table for LZ78 and LZW during the decompression will restrict 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 with part of the buffer, a triple of values corresponding to the matching position, the matching length, and the following symbol after the match is output. For example in Figure 3, we find that the incoming string 3011 is in buffer position 2 with match length 4, and the next symbol is 0. So the algorithm will output codeword (3, 4, 0).

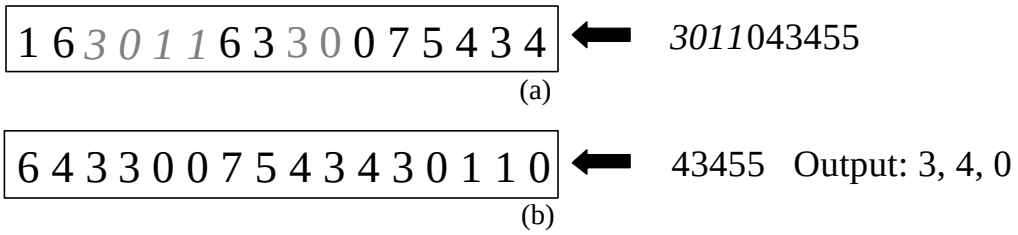

Figure 3. The LZ77 sliding window compression example. Two matches found are illustrated in color gray. The LZ77 selects the longer match "3011" and the resulted codeword is (3, 4, 0). (a) shows the sliding window buffer and the input string before encoding. (b) shows the buffer and input string after encoding.

Standard LZ77 compression containing the 3 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 that no matching is found, rather than outputing the symbol, the algorithm will produce a codeword containing 3 fields, wasting bits and worsening the compression ratio. An extension of LZ77 called LZSS will improve the coding efficiency. A threshold is given and if the matching length is shorter than the threshold, only the current symbol will be outputted. For the case that the matching length is longer than the threshold, the output codeword will consist of the index pointer and the length of the matching. In addition, in order to achieve correct decompression a flag bit is required for each code word to distinguish the two cases.

As we mentioned above, the FDR in the 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 only contain one frame of configuration data, using it as the sliding window buffer will not fully take advantage of *inter-frame* regularities. Thus, we modify the FDR to the structure shown in Figure 4. As can be seen in Figure 4, the bottom portion of the modified FDR, which has size of the original FDR, can transfer data to the configuration memory. During the decompression the compressed bitstream is decoded and then fed to the bottom of modified FDR. The data coming in will be shifted upwards in the modified FDR. The configuration data will be transferred to the specified frame once the bottom portion of the modified FDR is filled with newly input data.

In addition, the configuration data that is written to the array can be reloaded to the bottom portion of the modified FDR. This allows a previous frame to be reused as part of the dictionary and the *inter-frame* regularity is better utilized. More 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 with the greatest similarity to the new frame, we may be able to exploit the similarities to compress this new frame. While this technique will be slow due to the 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 in nearly no time. Therefore 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 can 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 2 frames of data. This will not significantly increase the hardware overhead yet will utilize the regularities in the configuration stream.

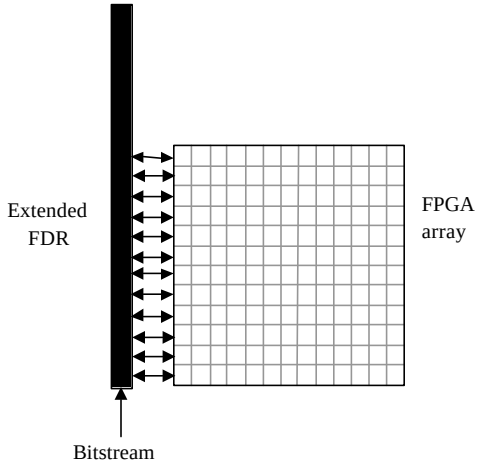

Figure 4. The hardware model for LZ77 compression.

Given a configuration file, finding the regularities is a major goal. The LZ compression only performs well in the case that common strings are found between the sliding window buffer and the incoming data. This requires quite a large buffer for finding enough matchings 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 desired result. In order to make compression work efficiently for a relatively small buffer, we need to carefully exploit the datastream, finding regularities and intelligently rearrange the sequence of the frames to maximize matchings. In this work, we have developed algorithms that apply LZSS compression targeting the hardware model described above. These algorithms are all realistic but require different amount of hardware resources and thus provide different compression ratios.

3.5.1. 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, and by configuring them consecutively, higher compression ratios can be achieved. The readback feature allows the frame with greatest similarity to the new frame to be read back to the modified FDR and reused as a dictionary, increasing the number of matchings for LZSS. This allows us to fully use regularities within the datastream. For example in Figure 5, 4 frames are to be configured and frames (b), (c) and (d) are more similar to (a) than to each other. Without readback the *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 dictionary when other frames are configured. The fast readback will significantly increase the utilization of the *inter-frame* regularities with negligible overhead. Since the

modified FDR is larger than the size of the frame, the LZSS will be able to use *intra-frame* regularities naturally. However, discovering *inter-frame* regularities represents an issue that will affect the effectiveness of the 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 such that it will greatly aid the compression of the incoming frame. In order to obtain such information, each frame will be used as a fixed dictionary in a preprocessing stage, and LZSS will be applied to each other frame, which are called *beneficiary* frames. Note that the LZSS is performed without moving the sliding window buffer, meaning that the dictionary will not be changed. This will exclude the potential *intra-frame* regularities within each *beneficiary* frame, providing the accurate *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 is more similar to the dictionary.

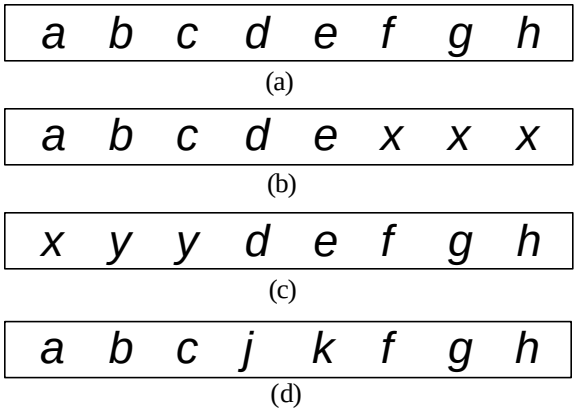

Figure 5. Example to illustrate the benefit of readback.

Once this process is complete, 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 is performed to delete the edges that present no *inter-frame* regularity between any two frames. Figure 6 (a) shows an example of the *inter-frame* regularity graph.

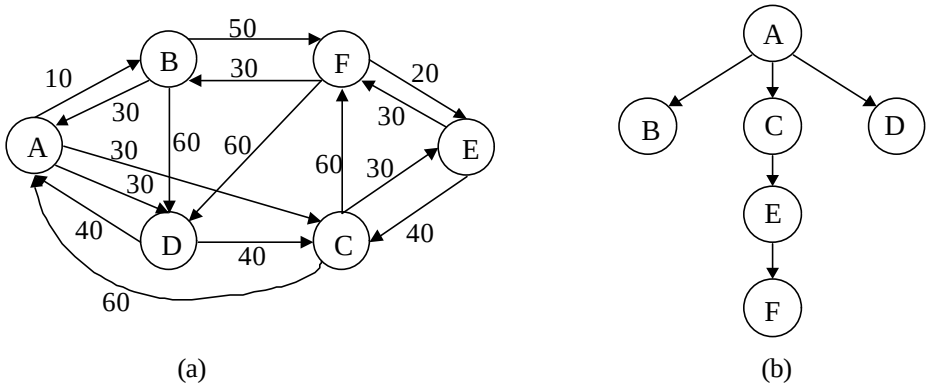

Figure 6. (a). An *inter-frame* regularity graph. (b). The corresponding optimal configuration sequence graph.

Given an *inter-frame* regularity graph, our algorithm seeks to find an optimal configuration sequence such that the *inter-frame* regularities are maximized. Specifically, we seek to find a subset of the edges in the *inter-frame* regularity graph such that every node can be reached and the aggregate of the subset 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 6 (b) shows the

corresponding optimal configuration sequence graph of Figure 6 (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 6 (b), a copy of the frame A will be stored in Block Select RAMs and read back to the modified FDR to act as a dictionary.

Now we present our readback algorithm:

1. Convert the input bitstream into symbol stream.
2. For each frame, use it as fixed dictionary and perform the LZSS on each 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 traverse 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 into 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 the 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 investigates the *inter-frame* regularities between frames. The results are used to build the *inter-frame* regularity graph and the corresponding configuration sequence graph in steps 3 and 4 respectively. Pre-order traversal performed in step 5 uses the parent frame of the currently loading frame as dictionary for the 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 of setting configuration registers will occur if frames to be configured are not continuous.

One final concern for our readback algorithm is the storage requirement for those reused frames. By analyzing the configuration sequence 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 RAM all the time, and they can share the same memory slot without conflict. For example in Figure 7, both frame A and frame B need to be read back. Suppose the left sub-tree needs to be 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. We have developed an algorithm using bottom-up approach that accurately calculates the memory slots necessary. By combining it with our readback algorithm the usage of the Block Select RAMs can be minimized. The details of the algorithm are as follows:

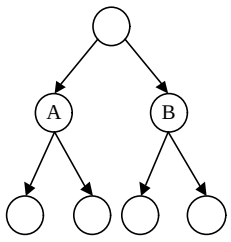

Figure 7. An example of memory sharing.

1. For each node in the configuration sequence graph, assign 0 to the variable V and number of children to C.
2. For each node that all its children are leaves, put it 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 8 illustrates an example of our memory requirement calculation algorithm. The left is an original configuration sequence graph. The 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 2 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 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 traverse will scan the left sub-trees before the right sub-trees, we should readjust the configuration sequence graph such that for each node the sub-tree that requires the most memory should be set 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.

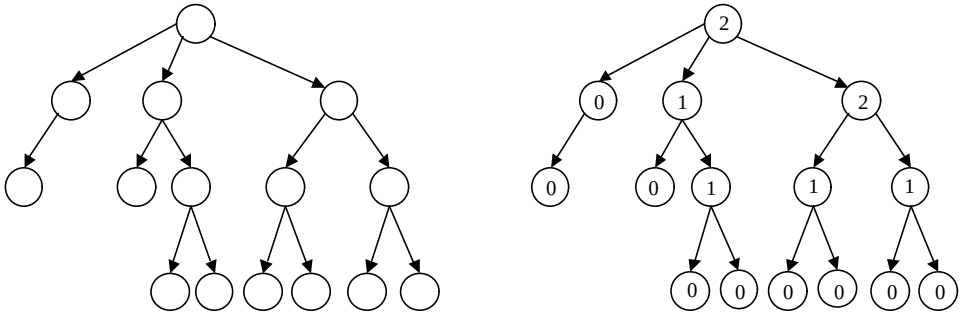

Figure 8. An example to illustrate our memory requirement calculation algorithm.

3.5.2. 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 datastream, we have developed a frame reordering algorithm that does not require the frame readback feature.

As can be seen in our readback algorithm, frame reordering will enhance the compression by utilizing the *inter-frame* regularities. This idea can still be applied on 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 the frame read back, traversing the configuration sequence graph might not necessarily be the optimal solution since the 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 dictionary only once. The active frame reordering algorithm 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 bitstream into symbol stream.
2. For each frame, use it as fixed dictionary, perform the LZSS on each other frame.
3. Build an *inter-frame* regularity graph using the values resulted in 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 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 a incoming edge to H, otherwise set the frame to T.

6. Perform the LZSS compression on the chain discovered in 5.

The basic idea of the algorithm is to grow a spanning chain from the two ends. The step 5 finds a frame not in the chain with the shortest edge either coming in to 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 6, 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 compressed datastream.

3.5.3 Fixed Frame Reordering Algorithm

One simple algorithm is to reorder the frames such that the *Nth* frame of each column to be configured in continued sequence. Performing the LZSS over the sequence generated by the simple reordering will take advantage of the regularities within these applications and the overhead of setting the configuration registers can be eliminated using this fixed frame order.

3.6. Wildcarded Compression

Our previous research using wildcard registers achieves good compression for the Xilinx 6200 FPGA [Hauck98b]. As mentioned above, multiple rows within a frame can contain the same configuration data. Instead of configuring them one by one, the wildcarded approach could allow 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 of the FDR. The address register and the wildcard register will allow specified rows within the FDR to be configured.

For circuits with repetitive structures, multiple frames could be very similar, yet not be completely identical. By allowing the FDR to be addressable, we can take advantage of this *inter-frame* regularity. Instead of loading the whole frame, we can merely load the differences between frames. For example in Figure 9, two frames need to be configured and the second frame only has 3 different rows from the first one. In this case, only the configuration data for the 3 different rows needs to be loaded. In addition, if the 3 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 address register and wildcard register represents additional hardware cost. Moreover, extra bits for the address and wildcard are need to be transferred for every write.

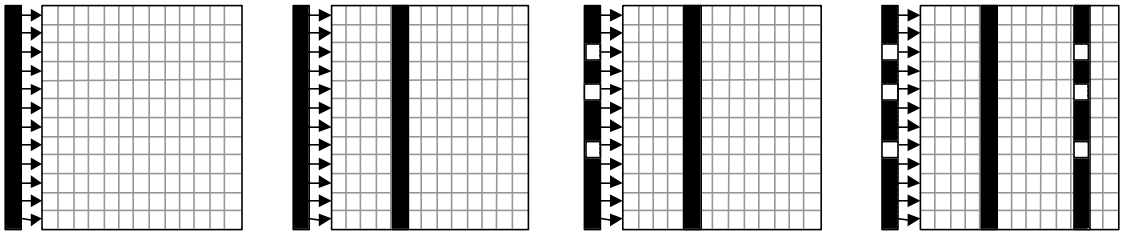

Figure 9. An example of inter-frame compression using addressable FDR.

The wildcarded algorithm consists of 2 stages. In the first stage we seek to reorder frames with great similarity to be configured consecutively. This will create a sequence in which the number of writes necessary for configuring each frame is greatly reduced. In the second stage we will seek to find the wildcards covering the writes for each frame and thus further reduce the configuration overhead. The first stage will take advantage the *inter-frame* regularities while the second stage will focus on *intra-frame* regularities. In the first stage, we will discover the number of different rows between each pair of the frames and the result indicates the extent of similarity between the frame. 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 2 for the traveling-salesman problem with triangle inequality. 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 wildcarded algorithm are as follows:

1. Convert the input bitstream into 18-bit symbol stream.
2. For each pair of frames, find out the different 18-bit symbols between them.
3. Build a regularity graph using the results of the step 2.
4. Perform Approx-TSP-Tour algorithm to determine the order of the frames to be configured.
5. For each frame configuration, using the wildcard algorithm [Hauck98b] to find the wildcards to cover the differences.

4. Don't Cares

The previous research [Li99] shows that with the help of the don't cares bits within the configuration data stream, higher compression ration can be achieved. Although Xilinx does not disclose the information necessary for discovering the don't cares in the Virtex applications, we would still evaluate the potential r the Virtex compression. In order to make an estimate evaluation, we will 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 bitstream. The way that the bits are turned will affect the frame sequence and thus the compression ratio. Finding the optimal way to turn the bits takes exponential time. We have used a simple greedy approach to turn these bits to result an upper bound for our readback algorithm. The configuration sequence graph is built with the consideration of the don't cares. In step 5, we will greedily turn the don't care bits into '0' or matchings. Note that once a bit is turned, it cannot be used as don't care anymore. In order to discover the lower bound, we will not turn the don't care bits and they can be used again to discover better matchings.

5. Simulation 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. The detail information about the benchmarks can be seen in Table 1.

Benchmark	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	Northeastern	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

Table 1. Information for the Benchmarks.

Figure 10 shows the simulation results of the compression approaches using 6-bit symbols, with the exception that the wildcard approach uses 18-bit symbols. The left 10 benchmarks are automatic mapped and use more than 50% of the chip area. The "Geo. Mean" column is the geometric mean of the 10 benchmarks. The right 3 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 affect compression. Figure 11 demonstrates the simulation results of 9-bit symbols. (The wildcard algorithm is not shown since it only uses 18-bit symbols). 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 fully utilizes the *inter-frame* regularities within the configuration datastream by reusing certain frames as dictionaries. The reordering techniques, though they cannot fully utilize the *inter-frame* regularities, 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 the frames, while fixed reordering can only utilize the regularities given by the fixed sequence.

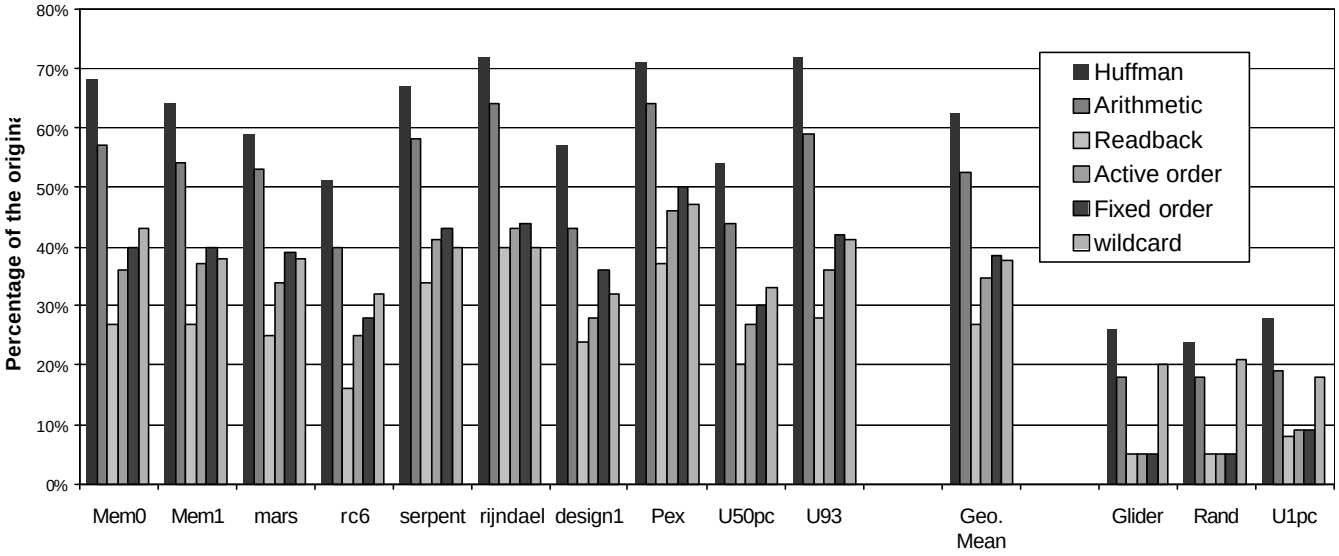

Figure 10. The simulation results for 6-bit symbol.

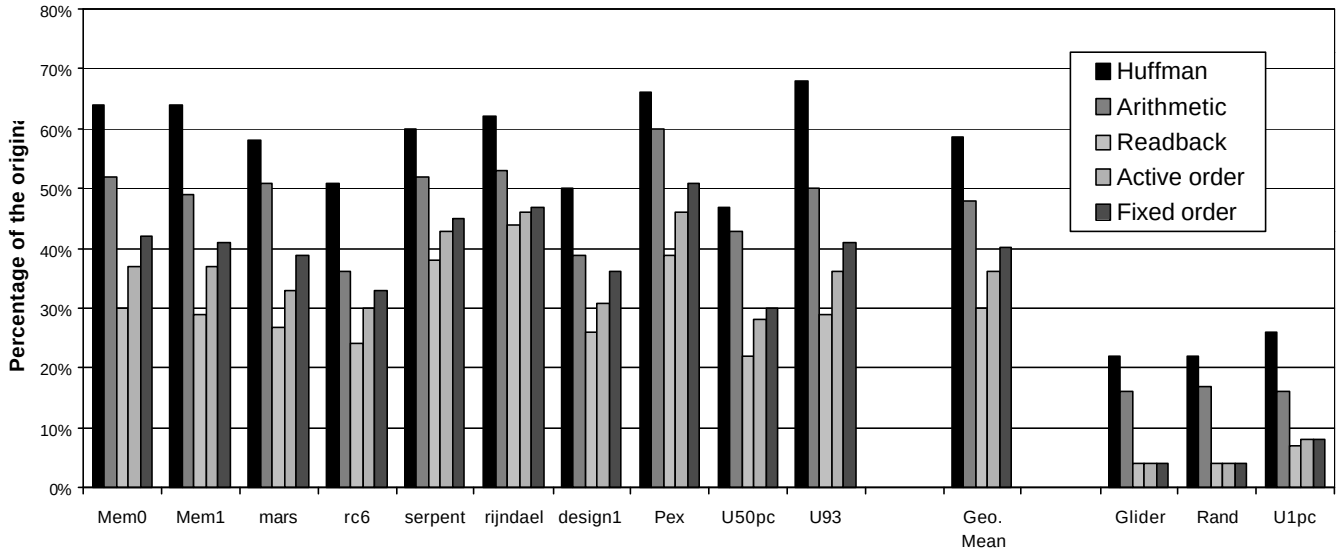

Figure 11. The simulation results for 9-bit symbol.

Surprisingly, although the wildcard approach that can exploit and utilize both *inter-frame* and *intra-frame* regularities, it still yields worse compression ratio than the active reordering scheme for most of the benchmarks. There are several reasons for the worse performance of the wildcard approach: 1) The wildcard approach requires address and wildcard specification for each write, adding significant overheads to the bitstream. The additional overhead overwhelm the benefits provided by the regularities within the applications. 2) The wildcard approach requires the comparison between the same rows of given frames to discover the *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 the wildcard approach. For example in Figure 12, the wildcard approach cannot discover the inter-frame regularity between Fame A and Frame B. However, the regularity can be utilized for the LZ-based approaches. 3) The wildcard approach requires enough rows covered by a wildcard share the same configuration value to achieve better compression. However, even for the XCV1000, which is a relatively large device, there are only 64 rows and it is not likely to find enough rows covered by a wildcard which have the same configuration value. For many cases, each wildcard only contains one row and the address/wildcard overhead is still applied.

	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 12. Unaligned regularity between frames. The wildcard approach will miss this regularity that benefits the LZ-based approaches.

The probability based Huffman and Arithmetic coding techniques perform significantly worse than other techniques since they do not consider the regularities within the bitstream. The Huffman approach did worse than the Arithmetic approach simply because of its inefficient coding method. Adding that these 2 approaches require significant amount of hardware for decompression, we will not consider use them for configuration compression.

By comparing the results in Figure 10 and 11, we found the LZ based approaches perform better on 6-bit symbols than 9-bit for most of the benchmarks. By analyzing the bitstream, we found that the regularities discovered within the bitstream may not result in very long matchings. Increasing the symbol size will shorten the matchings and increase the length of the codewords for single symbols. The Huffman and Arithmetic approaches perform better on 9-bit symbols because 9-bit symbols distribute the probability better.

Most of the benchmarks we tested use a significant amount chip resources, since this represents the most common case. The only exception is "u1pc", which uses about 1% of the chip area. As can be seen in the figures, the compression ratio is very high. The other two benchmarks that have very high compression ratio use more than 80% of the chip area, but were hand placed. After analyzing the two benchmarks, we found the two handcrafted circuits have extremely strong *intra-frame* regularities. Specifically, most rows within each frame are identical and very long matchings can be found.

Figure 12 demonstrates the potential effect of the don't cares over the benchmarks that are the larger circuits automatic placed and routed (left benchmarks in Figure 10). The X axis is the percentage of the don't cares we randomly create and the Y axis is the normalization over the results (Figure 10) without considering the don't cares. As can be seen in Figure 12, by using our upper bound algorithm a factor of 1.3 improvement can be achieved on applications containing 30% don't cares, while a factor of 2 improvement is achieved using the lower bound approach. We believe that better heuristics will fill this gap.

As we mentioned previously, the symbol length will affect the regularities within the bitstream. Figure 13 shows the results for different symbol lengths. As can be seen clearly, the regularities for Virtex applications are carried as multiples of 3. The symbol lengths of powers of 2 used in traditional

compression will either miss or destroy the regularities within the data stream and result in a much worse compression.

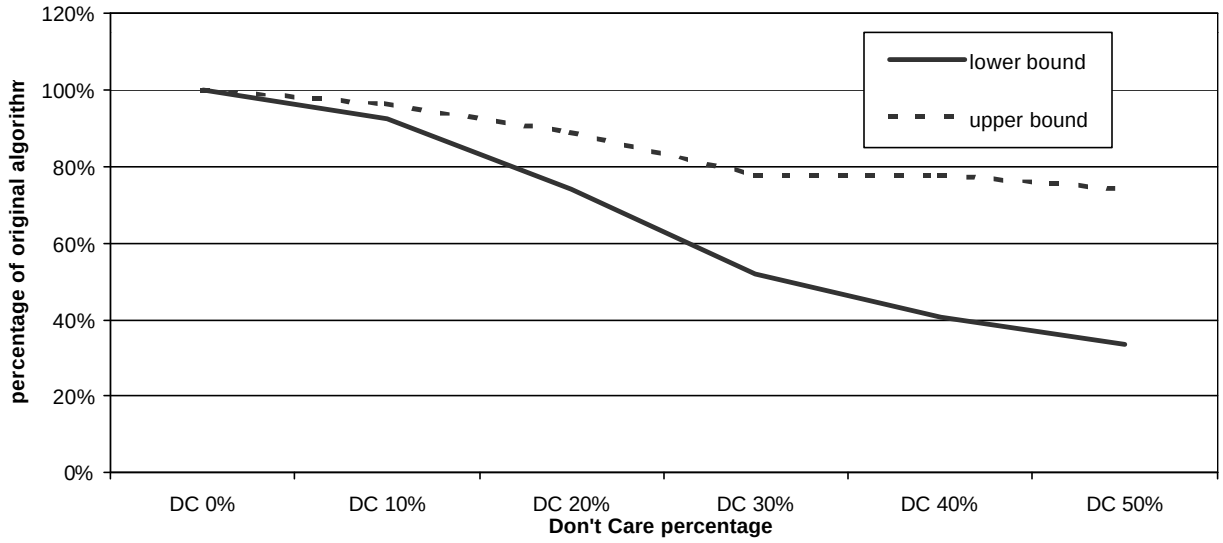

Figure 12. The effect of Don't Cares.

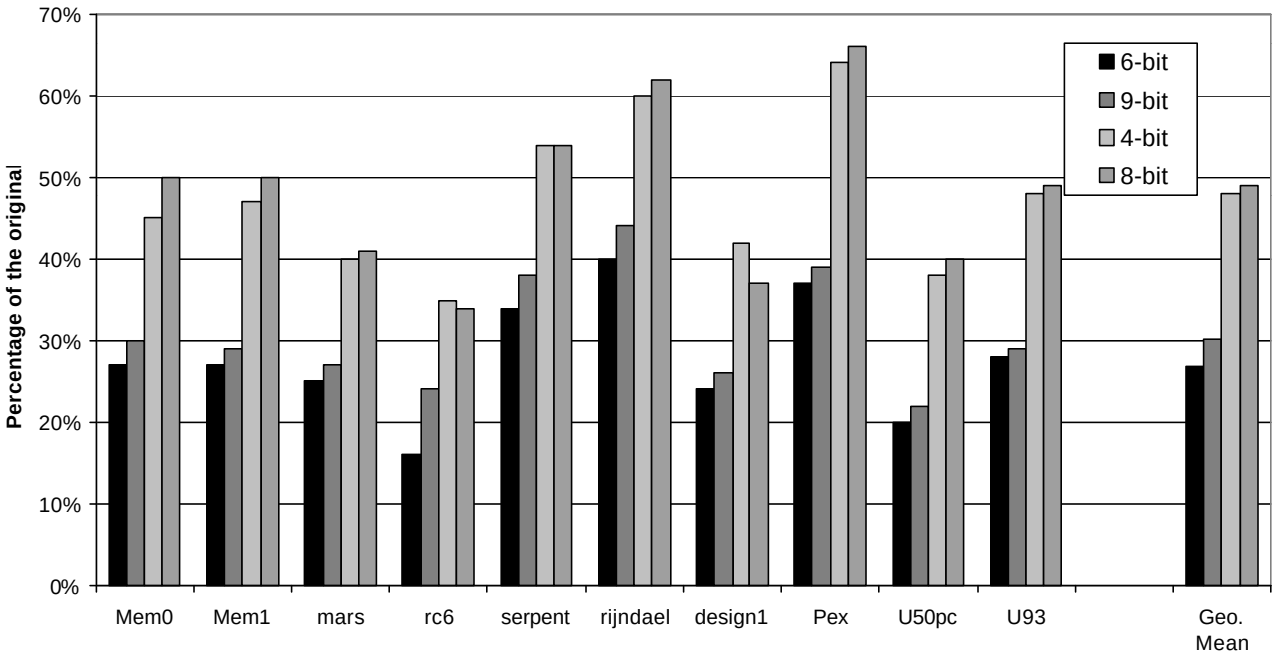

Figure 13. The effect of the symbol length.

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

IEEE Symposium on FPGAs for Custom Computing Machines, pp 276-277, 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.

[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

[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., "Configuration Architecture of Virtex Field Programmable Arrays Product

[Ziv77] J. Ziv, A. Lempel, "A Universal Algorithm for Sequential data Compression *Information Theory IT—23*, pp 337-343, May 1977.