

Bitstream Decompression for High Speed FPGA Configuration from Slow Memories

Dirk Koch, Christian Beckhoff, Jürgen Teich

Department of Computer Science 12, University of Erlangen-Nuremberg, Germany
{dirk.koch, teich}@cs.fau.de, christianbeckhoff@gmx.de

Abstract—In this paper, we present hardware decompression accelerators for bridging the gap between high speed FPGA configuration interfaces and slow configuration memories. We discuss different compression algorithms suitable for a decompression on FPGAs as well as on CPLDs with respect to the achievable compression ratio, throughput, and hardware overhead. This leads to various decompressor implementations with one capable to decompress at high data rates of up to 400 megabytes per second while only requiring slightly more than a hundred look-up tables. Furthermore, we present a sophisticated configuration bitstream benchmark.

I. INTRODUCTION

The progress in silicon technology has led to an enormous growth of resources found on current FPGAs. And with larger FPGAs, the reconfiguration data has increased significantly over the last decade and has just passed the 10 megabyte barrier [1], [2] for the high end products of the leading FPGA vendors. In order to keep the configuration times within acceptable margins, most recent FPGA families offer high speed configuration with data rates of up to several hundreds of megabytes per second. Such high data rates cannot be delivered by today's low cost non-volatile memories. For instance, typical NAND-flash devices offer read data rates of just a few tens of megabytes per second. Consequently, configuration bitstream storage for high speed configuration is costly. Note that some systems require multiple different configurations, making the bitstream storage even more expensive. For example, if a system supports in-field updates, such a system will normally provide a fallback configuration bitstream for the case of an update failure. In such environments, bitstream compression may help to reduce the memory usage and the required memory bandwidth, thus saving cost.

Especially for systems using runtime reconfiguration, the configuration speed is an important issue dictating if two or more modules are suitable to share the same FPGA resource area over time. If, for example, two different modules of a video processing system have an accumulated execution time smaller than the time between two video frames, these two modules may be executed time-multiplexed on an FPGA. But this is only feasible if also the reconfiguration can be performed within the given time budget. For typical multimedia applications, this implies reconfiguration times in the range of a few milliseconds.

We propose to use bitstream decompression for accelerating the configuration speed by enhancing the throughput that is typically constrained by slow configuration storage devices.

For instance, when bitstream compression reduces the size of a bitstream to 50%, the configuration speed may be doubled.

When bitstream compression should save cost and enhance the configuration speed, the decompression has to be performed in hardware. Bitstreams may contain fractions of data that are selectable to any value, thus allowing higher compression ratios [3]. Unfortunately, all bitstream formats of today's FPGA are confidential, thus preventing us to exploit such techniques. Consequently, our proposed compression algorithm variations work lossless. For hardware decompression, we observe the following objectives:

- *Compression ratio*: In order to allow memory savings and configuration throughput enhancements, we require significant compression ratios.
- *Throughput*: In order to speed up the reconfiguration with the help of hardware decompression, the decompressor has to emit a high continuous data rate. As discussed in Section II, this is not only a question of the achievable clock frequency but also a question of possible burst throughputs on both sides of the decompression module.
- *Resource overhead*: In order to get an overall benefit due to reconfiguration, the decompression hardware should be as small as possible.

Contribution In this paper, we analyze existing compression algorithms with respect to their suitability for hardware accelerated bitstream decompression. We are the first discussing various modifications to standard algorithms in order to fulfill the objectives of *compression ratio*, *throughput*, and *resource overhead* at the same time and with high quality. Here, high quality means implementations that achieve compression ratios that can compete with state of the art software programs, such as *gzip*, implementations that allow throughputs of several hundreds of megabytes, and implementations with small resource requirements. We discuss issues for FPGA implementations as well as for CPLD designs, and some of our presented implementations perform well also for bitstreams of different FPGA families. Furthermore, we developed a benchmark including several dense and therefore hard compressible configuration bitstreams that can be accessed freely on our website [4].

The paper is structured as follows: In Section II, we present general aspects and a motivating example why bitstream decompression algorithms must be carefully designed in order to reduce configuration time. Next, in Section III, we introduce our configuration bitstream benchmark that is used in Section IV to rate our implementations.

II. ALGORITHMS FOR BITSTREAM COMPRESSION

Bistream compression is the process of encoding FPGA configurations using fewer bits than the original unencoded bitstream. We define the *compression ratio* η as

$$\eta = \frac{\text{size}(\text{compressed bitstream})}{\text{size}(\text{original bitstream})}$$

The main goal of bitstream compression is to hide a low bandwidth of a configuration memory through hardware efficient decompression. All of our presented accelerators for bitstream compression are capable to emit one word per clock cycle. Let $|w_{FPGA}|$ be the word width of the configuration interface and $|w_{deco}|$ be the decompressor output data width, then a decompression accelerator has to operate with at least $\frac{|w_{FPGA}|}{|w_{deco}|}$ times the clock frequency of the configuration interface in order to allow full configuration speed.

In order to accelerate the configuration process, it is not only important to put focus on the *best* compression ratio, it is much more important to consider the configuration memory *bandwidth*. Let d_{FPGA} be the configuration interface data rate, and d_{MEM} be the peak data rate of the configuration memory, then the configuration data rate is:

$$d_{conf} = \begin{cases} d_{FPGA} & \text{if } d_{MEM} \cdot \frac{1}{\eta} \geq d_{FPGA} \quad \text{full speed} \\ d_{MEM} \cdot \frac{1}{\eta} & \text{else} \quad \text{limited by memory bandwidth} \end{cases}$$

Note that for any lossless compression algorithm, it is possible to construct cases where the compression ratio is larger than one [5], thus, degrading the configuration time. For instance, let us assume a 100 kB configuration bitstream where 10 kB of the bitstream contains worst case compression data that may be compressed by a ratio of $\eta_1 = 4$, while the 90 kB remaining bitstream data may be compressed at $\eta_2 = 0.1$. Then, the average compression ratio is $10\% \cdot \eta_1 + 90\% \cdot \eta_2 = 49\%$. If we further assume a configuration interface data rate of the FPGA of $d_{FPGA} = 100$ MB/s and a configuration memory data rate to the decompressor of $d_{MEM} = 50$ MB/s, it seems to be possible to configure the FPGA in the shortest possible time of $\frac{100 \text{ kB}}{d_{FPGA}} = 1$ ms instead of $\frac{100 \text{ kB}}{d_{MEM}} = 2$ ms when not using decompression.

However, this holds true only if a decompression accelerator reads at a continuous data rate from the configuration memory. With respect to our example, the 10% containing worst case configuration data produce a configuration memory burst that idles the configuration interface, while in the remaining 90% of the bitstream, the maximal configuration speed is limited by the configuration interface bandwidth. In the last case, we are not taking full benefit of the high compression ratio and we waste bandwidth at the configuration memory interface. As shown in the experimental results (see Section IV), these bursts cannot be hidden significantly by a FIFO buffer between the configuration memory and the decompression accelerator. When taking the bandwidth of a configuration memory into account, the configuration time is $\frac{10 \text{ kB} \cdot \eta_1}{d_{MEM}} + \frac{90 \text{ kB}}{d_{FPGA}} = 1,7$ ms. This means that the configuration time is reduced by just 15% as compared to an uncompressed configuration process. Despite the good average compression ratio of below 50%, we are far away from the optimal configuration time of 1 ms.

If we design the bitstream compression algorithm such to limit the worst case compression ratio at the cost of a reduced best case compression ratio, we can speed up the configuration time significantly. For instance, limiting the compression ratio of the 10% of the bitstream containing worst case configuration data to, lets say $\eta'_1 = 2$ while having a compression ratio of $\eta'_2 = 0.4$ for the remaining 90%, results in an average compression ratio of $10\% \cdot \eta'_1 + 90\% \cdot \eta'_2 = 56\%$. If we now compute the configuration time for the adjusted algorithm, we achieve a much better configuration time of just $\frac{10 \text{ kB} \cdot \eta'_1}{d_{MEM}} + \frac{90 \text{ kB}}{d_{FPGA}} = 1,3$ ms.

These numerical examples point out that looking only at the overall compression ratio will not necessarily enhance the configuration speed. To the best of our knowledge, this problem has neither been identified nor tackled in related publications.

In the following, we will examine configuration ratio issues together with implementation aspects for variations of well known compression algorithms, such as run length encoding, Lempel Ziv, and Huffman encoding.

A. Run Length Encoding

Run length encoding [6] is a simple technique to compress a sequence of identical words belonging to a data stream. Thus, run length encoding is a good candidate to compress configuration bitstreams that are sparsely populated with binary 1 values and that offer lots of consecutive identical words. The recurrences will be encoded by tuples (w, l) with w being the next word and l being the run length denoting the number of times this word is repeated in the original data. The data width of the next word is $|w|$ bit and the amount of bits used to encode the run length is $|l|$ bit. Run lengths larger than $2^{|l|}$ will be encoded by multiple tuples that allow to recombine the original input data. When w and l are encoded with a fixed amount of bits, the compression ratio η_{RLE} is in the following range ($\hat{\eta}_{RLE}$ denotes the worst case compression ratio):

$$\frac{|w| + |l|}{|w| \cdot 2^{|l|}} \leq \eta_{RLE} \leq \frac{|w| + |l|}{|w|} = \hat{\eta}_{RLE}. \quad (1)$$

The cost to encode a run length tuple is always $|w| + |l|$ bits. These bits code in the best case $|w| \cdot 2^{|l|}$ bits and in the worst case just $|w|$ bits. Note for the last case that the compressed data will be larger than the original input data.

The word width $|w|$ and the amount to code the run length should be determined with care, because when random data is compressed using run length encoding, the compressed data will be larger than the original input data. This is illustrated in Figure 1 on the basis of a probabilistic state machine. The probability that the next word in a random data stream is identical to the present one is $\tilde{p}(1) = \frac{1}{2^{|w|}}$. The probability for longer recurrences i decreases exponentially: $\tilde{p}(i) = \frac{1}{2^{|w| \cdot i}}$. The compression ratio for an infinite random data stream is therefore:

$$\tilde{\eta}_{RLE} = \sum_{i=1}^{2^{|l|}} \frac{|w| + |l|}{|w| \cdot i} \cdot \frac{1}{2^{|w| \cdot i}}. \quad (2)$$

Even for relatively small word widths, the probability that just two consecutive words are identical is about zero (e.g., $< 1\%$ for bytes). Therefore, $\tilde{\eta}_{RLE}$ converges against the worst case compression ratio $\hat{\eta}_{RLE} = \frac{|w|+|l|}{|w|}$ for increasing $|w|$.

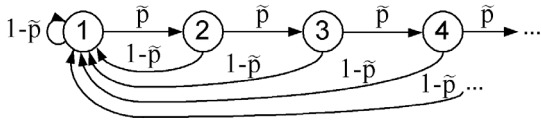


Fig. 1. Probabilistic state machine for the compression of random data with run length encoding. The states represent the run lengths and the probability $\tilde{p} = \frac{1}{2^{|w|}}$ denotes the chance that the next word is identical.

The values of $\tilde{\eta}_{RLE}$ and $\hat{\eta}_{RLE}$ indicate the behavior when the bitstream contains fractions of high entropy data. This can happen especially for initialization data of internal RAM blocks. As the relative amount of RAM initialization data is not negligible¹, we demand that all compression algorithms must have an $\tilde{\eta}$ close to 1 and an $\hat{\eta} \leq 2$. Consequently, the run length must not be encoded with more than the half amount of bits chosen for word width: $|l| \leq \frac{1}{2}|w|$. As a starting point for parameterizing the run length algorithm, we analyzed a histogram of the probability distribution of the run lengths for different word widths $|w|$ of the complete benchmark corpus.

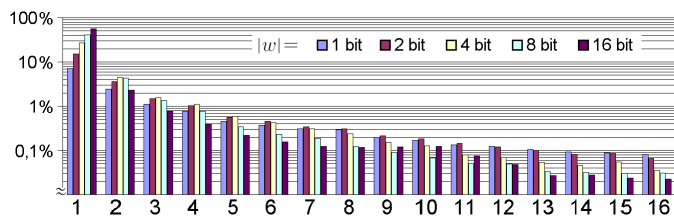


Fig. 2. Normalized histogram of the first 16 run lengths found inside all benchmark bitstreams (see Section III). The columns denote the percentage of run length occurrences against different word widths with respect to the complete amount of words.

The histogram in Figure 2 reveals that higher rates of consecutive sequences exist only for small word widths. For instance, for a word width of $|w| = 16$, the probability is 55% that the run length is 1, which means that most pairs of consecutive words differ. Furthermore, the histogram points out the potential for data compression through run length encoding. If we add up, for example, the probabilities of the first ten run lengths ($\sum_{i=1}^{10} p(i)$), we see that for all word widths except for the width $|w| = 16$, the sum of the probabilities is below 50%. Thus, for these word widths, the majority of run lengths is larger than ten, pointing out a feasibility for run length compression. However, when using the described encoding, the run length 1 will lead to a relatively large portion of data σ in the compressed data stream that is larger than the original one: $\sigma = \frac{|w|+|l|}{|w|} \cdot p(1)$. For instance, if we set the word width to $|w| = 4$ bit, we find according to Figure 2 that 27%

¹According to the data sheets [1], [2] for the chosen FPGAs in the benchmark section, the relative amount of RAM preload data is in average 14% of the overall bitstream size.

of two consecutive words differ. If we further set the length field to $|l| = 2$ bit, we get $\sigma = \frac{6}{4} \cdot 27\% = 40.5\%$. Note that the final compression ratio is always worse than σ .

In order to overcome these issues, we propose two variations. The first one works on a bit width of a single bit, while the second one uses a flag to distinguish between a recurrent sequence or differing consecutive words.

1) *Toggle RLE*: If the word width is set to $|w| = 1$ bit, the run length value specifies the distance between two bit flips. If we further define a fixed start bit value, the complete bitstream data is compressible by encoding only run lengths while omitting the next word w . By reserving some codes for specifying run lengths without toggling at the end, it is possible to compress recurrent sequences of any length.

In a former implementation of such a decompressor [7], we have demonstrated good compression ratios in combination with tiny hardware decompression modules. In this paper, we present an improved version that achieves better compression ratios. We examined different prefix free encodings in order to balance between the compression ratio and the hardware to decode the prefix free code words specifying the run lengths. In addition, the encodings ensure a bounded compression ratio for random data as well as for worst case examples. The toggle RLE hardware decompressor requires only a counter, a control state machine, and a decoder for the run length values. Therefore, this technique is suitable for FPGA as well as for CPLD implementations. One drawback of this technique is the sequential output of the bitstream data. Even for high speed designs running at 200 MHz, the throughput is limited to just 25 MB/s.

2) *Flag RLE*: In order to enhance the throughput, we examined a variation of the above presented traditional run length encoding scheme that uses a flag to distinguish between the case that two consecutive words differ and the case of a recurrent sequence of identical words (run length equal one or larger than one). The corresponding encodings are $(0', w)$ and $(1', w, l)$. Thus, the worst case compression ratio is $\hat{\eta}_{RLE} = \frac{|w|+1}{|w|}$. As shown in Section IV, the flag RLE has an inferior compression ratio compared to the toggle RLE approach, but it has a $|w|$ times higher throughput, because the toggle RLE decompressor emits only one bit per cycle. The flag RLE algorithm is also suitable for FPGA and CPLD implementations, but care has to be taken of the decompressor input interface. In the case of a typical 8-bit input interface, we found the most hardware efficient implementation when using an extra register for the flags in order to keep the alignment of the incoming compressed data constant. This extra register will store the flags for a block of eight code words and it is loaded once at the beginning of a new block.

B. LZSS

The LZ algorithms (named after their inventors A. Lempel and J. Ziv) are string substitution schemes. A well known string substitution schemes is LZ77 [8]. String substitution schemes replace uncompressed data by references to already compressed data. During compression, as shown in Figure 3,

a search buffer keeps track of the last n already compressed words and a look-ahead buffer allows accessing the next m words of the uncompressed data. During the compression phase, the search buffer is scanned for matches of all prefixes of the look-ahead buffer. The longest prefix of the look-ahead buffer found in the search buffer is then replaced by its starting *offset* and its *length*. If no prefix was found, we emit the first word (*nextWord*) of the look-ahead buffer.

The LZSS approach [9], as an improvement of LZ77, uses a flag to distinguish between these two cases, similar to the methodology presented for the flag RLE in Section II-A.

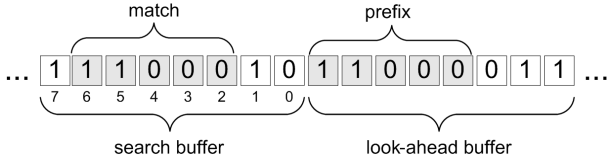


Fig. 3. LZSS compression. A search buffer holds already compressed data and a look-ahead buffer contains the next uncompressed sequence. The example shows a five bit prefix of the look-ahead buffer that is also found in the search buffer at position six, which is encoded as ('1', 6, 5).

Let o , l , and n denote the *offset*, the *length*, and the *nextWord* and $|o|$, $|l|$, and $|n|$ the number of bits to code these values. Then, during LZSS compression, either the code word ('1', o, l) or ('0', n) is emitted. The compression rate becomes minimal if the longest possible prefix of size $l_{max} = 2^{|l|}$ is matched. In this case, $l_{max} \cdot |n|$ bits are encoded with $1 + |o| + |l|$ bits. If no match was found, $|n|$ bits are encoded by $1 + |n|$ bits. This leads to the following lower and upper bound for the compression rate η of LZSS:

$$\frac{1 + |o| + |l|}{l \cdot |n|} \leq \eta_{LZSS} \leq \frac{1 + |n|}{|n|} = \hat{\eta}_{LZSS} \quad (3)$$

The upper bound is equal to the upper bound of the flag RLE algorithm (see Section II-A.2). Thus, LZSS does not lead to a drastic downgrading of the compression ratio if fractions of the configuration bitstream contain data with a high entropy.

1) *LZSS Hardware Decompression*: During decompression, offset and length refer to already emitted data. Thus, the decompressor must keep track of the last n emitted words, with n being the search buffer size of the compression program. For the LZSS decompression, we enhanced an approach for an LZ77 hardware decompressor proposed in [10] (shown in Figure 4) with a state machine that decodes LZSS encodings. During decompression, the emitted data is fed into a shift register that represents the mentioned search buffer. If there is no match in the shift register, *nextWord* will be emitted. In case of a match, the offset field (o) specifies the position of the shift register containing the first word of a l word long sequence.

As CPLDs contain only a relatively small amount of memory bits, LZ decompressors are inapplicable for CPLD implementation, because of the required shift register. For the implementation of the shift register on FPGAs, the following four options exist.

- 1) Look-up table flip-flops
- 2) Frame Data Input Register (FDRI) in Xilinx FPGAs [11],
- 3) Dedicated RAM-Blocks
- 4) Dedicated shift register primitives (SLRC16E) in Xilinx FPGAs [12]

The first solution consumes an unfavorable amount of hardware resources. The second one is an interesting alternative when implementing hardware decompression on the FPGA itself as a fixed part of the chip. Accessing the FDRI leads to a significant hardware overhead and the access is pretty slow. However, [13] and [14] present compression techniques that are based on the FDRI, but both papers omit a discussion of a possible hardware implementation of a decompression module. The third variant uses dedicated on-chip RAM resources that allow implementing a long shift register that was used in [15] for LZSS configuration data decompression in hardware. But huge shift registers require more bits in the compressed data in order to specify the offset o , thus leading to lower compression ratios especially for short matching sequences. By exploring various search buffer sizes and encoding schemes, we found no significant benefit of using huge shift registers, because configuration data for densely packed FPGA designs is highly irregular. Therefore, dedicated RAMs are oversized.

Instead, we examined the last option to implement the shift register with dedicated shift register primitives that are available on most Xilinx FPGA families [2]. We discovered that even small search buffers, as little as 32 words, lead to good compression results, as being revealed in our experimental results (to be found in Section IV). In our approach, the shift register has been implemented by using *SLRC16E* primitives on Xilinx FPGAs. This primitive allows to use a 4-bit look-up table alternatively as a 16 bit shift register with random access to the individual register contents. An l deep shift register of width n can be created by cascading $n \cdot \lceil \frac{l}{16} \rceil$ *SLRC16E* primitives.

For the sake of completeness, we want to mention an implementation [16] for decompressing configuration data in hardware that has some relationship with the LZ78 algorithm that is based on a dictionary rather than a shift register. However, the work reports only moderate compression ratios and omit details on the resource consumption.

2) *LZSS Parameterization*: Experimental results confirmed our assumption, that among the run lengths (shown in Figure 2), there is no uniform distribution among the prefix lengths found during LZSS compression. This observation lets us conclude that it is not suitable to code all LZSS prefix run

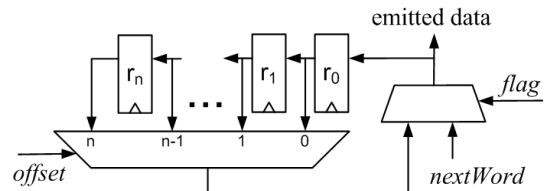


Fig. 4. LZ77 HW decompressor proposed in [10].

lengths with a fixed code word size. If we use prefix free encodings with adjusted word widths such as Huffman codes for the length encoding, we will require an unlikely amount of hardware resources, because of the different word widths that must be aligned to the input data width, as demonstrated in Section II-C. Therefore, we encoded only a subset containing $2^{|l'|}$ run lengths of all $2^{|l|}$ possible run lengths using less bits ($|l'| < |l|$). We explored manually how many bits are required to encode the length ($|l'|$) and which subset of run lengths leads to the best compression ratios. Note that the search space to find the optimal prefix run lengths for just one encoding of l' is $\binom{2^{|l|}}{2^{|l'|}}$. For our best encodings for run length prefixes, we found that at least 50% of the subset contains the first 10 smallest prefix run lengths.

Limiting the number of encoded prefix lengths will result in more code words in the compressed bitstream, because for an prefix length l that is not encoded, the biggest encoded prefix length smaller than l will be used instead. In this case, only a smaller prefix length is encoded and a new search for a prefix is executed. However, encoding more prefix lengths leads only to marginal better compression rates. In addition, longer prefix lengths will not necessarily reduce the configuration time, because they represent high compression ratios that cannot be exploited as the configuration interface has a limited input data rate.

The best parameter set found in the above mentioned exploration for our LZSS derivation was one bit for the flag, $|l| = 3$ bit, $|o| = 5$ bit, and $|n| = 8$ bit. This results in a nine bit long code word which does not match a typical 8-bit interface. To simplify the interfacing, we used an extra register for the flags that stores the flags for a block of 8 consecutive code words. Then the remaining compressed code word is 8 bit wide, thus, ideal for connecting a memory.

Recent FPGAs offer high speed configuration interfaces that are up to 32-bit wide and that operate with at least 100 MHz [1], [2]. In order to match these interfaces, we could extend the word width to 32 bits. This would influence the compression as follows: With an increased word size, long prefix lengths become more rarely. Therefore, the subset of encoded prefix lengths has to be adapted. In addition, the worst case compression rate improves with higher word widths. However, it is also possible to enhance the throughput by increasing the clock frequency of the decompressor in combination with a serial-to-parallel converter at the output. Our LZSS decompressor consists of a few simple basis elements and the whole design can be easily pipelined. Consequently, we can easily build decompression modules that run at least at 200 MHz (e.g., when implemented on a Xilinx Virtex-II device). This speed in combination with a word width of 16 bit allows us to emit data by a decompression module at a rate of 400 MB/s. Thus, we can utilize the full configuration interface bandwidth of today's FPGAs.

C. Huffman Compression

As an entropy encoding scheme, Huffman encoding [17] assigns short code words to frequently occurring words and

longer code words to infrequently occurring words. Huffman encoding is based on a Huffman tree, which in turn is created upon a probability distribution of words. Creating an individual Huffman tree for each bitstream file is called *dynamic Huffman encoding*. Such a Huffman tree is part of the compressed data and has to be reconstructed before decompression. Such decompressors turn out to be costly in terms of hardware usage [18].

Applying the same Huffman tree for all bitstreams is called *static Huffman encoding*. The static Huffman tree is based on the probability distribution of all words in our benchmark corpus and will never change. The compression rate for the overall corpus is hence still optimal, whereas we could achieve better compression rates for single bitstream files using dynamic Huffman encoding. We found that static Huffman compression performs well on our corpus as can be seen in Figure 5. The higher the variation in the probability distribution, the better the compression ratio achievable with Huffman encoding.

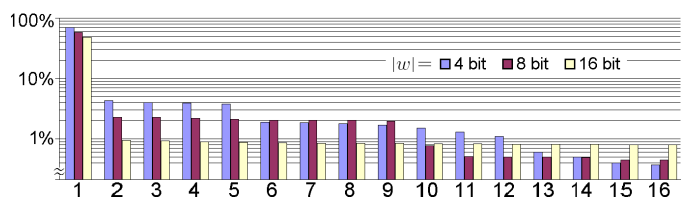


Fig. 5. Normalized probability histogram of the occurrences for the first 16 most frequently appearing words found inside all benchmark bitstreams.

Let $|h_{min}|$ and $|h_{max}|$ denote the smallest and the largest Huffman code word widths found in our static Huffman tree and $|w|$ the word width used for compression. Then, the compression ratio for Huffman encoding is:

$$\frac{|h_{min}|}{|w|} \leq \eta_{Huff} \leq \frac{|h_{max}|}{|w|} = \hat{\eta}_{Huff} \quad (4)$$

As we demand that the decompression ratio is bounded in such way that fractions of high entropy data will not completely cut of the throughput, we have to restrict the Huffman code. If we want to ensure that the worst case compression ratio $\hat{\eta}_{Huff}$ will be not larger than e.g. 2, we have to limit the largest code word width to $|h_{max}| \leq 2 \cdot |w|$. We can further check the compression ratio for random data that is the average code word length over all Huffman code words $h(i)$:

$$\tilde{\eta}_{Huff} = \frac{1}{|w|} \cdot \frac{1}{2^{|w|}} \sum_{i=1}^{2^{|w|}} h(i) \quad (5)$$

In the case $\hat{\eta}_{Huff}$ is larger than 2 or $\tilde{\eta}_{Huff}$ is to far away from 1, we have to reduce the depth of the Huffman tree. Reducing the Huffman tree does not essentially reduce the maximum configuration speed. As can be seen in the histogram in Figure 5, there is one code word with a probability of more than 50%, thus it will be encoded in a classical Huffman tree with one bit. However, if we assume that the configuration memory interface can deliver 50% of the maximum configuration data

rate, then we can spend up to 4 bits to encode this word before the configuration speed drops down for a Huffman tree with $|w| = 8$ bit wide output words. In other words, before this point, the maximum throughput of the configuration interface will limit the configuration speed. By applying this technique, we will reduce the average compression ratio on the one side. But on the other side, we will enhance the configuration speed massively, because longer code word sizes can now be reduced.

1) *Huffman Hardware Decompression*: After computing a static Huffman tree that fulfills our requirements, we can automatically generate the hardware decoder module. An example of the data path of such an implementation is shown in Figure 6. When in the example a new word has to be decoded, the *start* input becomes one and according to the values in the so-called *alignment register* (flip-flops r_0, \dots, r_2) exactly one output of the decoder tree becomes active. As the words of the original bitstream data is encoded into code words of different lengths $h(i)$, the alignment register is used to arrange the incoming data d such that the Huffman tree decoder gets its incoming code word always at exactly the same position. The outputs of the Huffman tree decoder indicates the decoded word of the alphabet $\{00, 01, 10, 11\}$ that can be emitted in the following. Next, the alignment register is shifted right by the length $|h(i)|$ of the encoded word. For instance, if the output for word 01 becomes active, the alignment register is shifted by two, which is selected by the multiplexer input $c2$.

A control state machine (not shown in Figure 6) keeps track about the fill level of the alignment register. If the alignment register has space for a new input value due to the shift operation after decoding a word $h(i)$, the alignment register is loaded with the next value from the input. The bottom multiplexers shift the input word to the most right free position in the alignment register. For instance, let us assume that r_3, \dots, r_5 are empty (not containing a code word) and that $r_0=0$ and $r_1=1$, thus, storing the encoded word for the output word 01. Then, upon $start=1$, the output 01 of the Huffman tree encoder will become active. In the following, r_0 and r_1 will be consumed and the last valid fragment of the compressed bitstream stored in r_2 will be shifted into r_0 . Then we will have five free entries in the alignment register and a new input word is stored in the registers r_1, \dots, r_4 which is selected by the multiplexer input $f5$.

The example points out that we require more logic for the input data alignment than for the Huffman tree decoder itself. Note that we require just one 4-bit look-up table to decode one of the output words, because in this example, no word requires more than the inputs of *start* and r_0, \dots, r_2 for decoding. The size of the alignment register $|r|$ depends on the input data width $|d|$ and the Huffman tree depth which is equal to h_{max} . If in the case that the alignment register is missing one bit to decode the next word, it must be possible to load the alignment register with a new input word, thus, the total required alignment register width is $|r| \geq |d| + h_{max} - 1$.

The logic overhead for the multiplexers is enormous. For each occurrence of a Huffman code word width, we have

to spend one input for the top multiplexers as well as one input for the bottom ones adjusting the input data. In addition, the structure contains long combinatorial paths and is hard to pipeline, because only when a decode phase has determined the next output word, we know from which position the next decode phase has to start.

However, as shown in Section IV, static Huffman encoding reaches the best compression ratios.

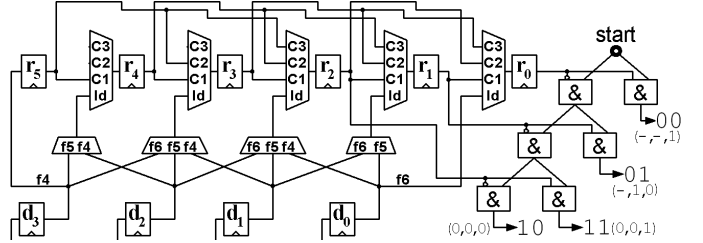


Fig. 6. Huffman decompressor. In the right a Huffman tree decoder is shown that decodes a new output word of the alphabet $\{00, 01, 10, 11\}$ ($W = 2$) upon $start=1$. The values in brackets denote the according encodings. The multiplexers and the *alignment register* r adjust the incoming compressed bitstream data d according to the width of a particular code word $h(i)$.

III. BENCHMARK

All compression algorithms exploit some statistical characteristics within the uncompressed data like non-uniform probability distributions of code words or some regularities found in the input data. In the case of FPGA configuration bitstreams, the exploited characteristics stem from the configuration facility itself, because FPGAs are designed as general purpose devices for implementing any kind of digital hardware. Therefore, only a subset of the logic and routing resources is used to implement a specific module and the used logic has an encoded counterpart inside the configuration bitstream.

However, if we want to build a suitable benchmark for investigating bitstream compression algorithms, we have to generate bitstreams that represent the statistical characteristics for a wide range of different FPGA designs. Most FPGA designs will have a high logic utilization, because unused resources produce significant overheads like monetary cost, power consumption, or configuration time. Therefore, we generated benchmark bitstreams having a logic utilization of at least 90% of the available look-up table resources². Only the *SoC* [19] bitstreams have lower utilization ratios. These bitstreams stem from a system-on-a-chip design where parts of the FPGA resources are reserved for runtime reconfiguration.

In order to detect influences on the compression ratio that are related to the implemented algorithm of a module and not only to the logic utilization, we chose examples from different application domains:

- In the *crypto core* section, we chose a DES module [20] and an RC5 core [21]. In these modules, we found a

²The FIR filter design for the Virtex V is slightly below the 90% border. Here, the routing resources limit higher logic utilization.

TABLE I

BITSTREAM BENCHMARK. THE TABLE LISTS THE FPGA, THE UTILIZATION FOR LUTS AND RAM-BLOCKS, AND THE ENTROPY FOR 8-BIT WORDS.

Module	Altera Cyclone-II	Xilinx Spartan-III	Xilinx Virtex-II	Xilinx Virtex-V
DES	EPC2C5 / 97% / 0% / 2,5	XC3S1000 / 97% / 0% / 3,3	XC2V1000 / 95% / 0% / 2,3	XC5VLX30 / 90% / 0% / 2,5
RC5	EPC2C15 / 97% / 44% / 2,2	XC3S1000 / 99% / 100% / 4,0	XC2V1500 / 92% / 54% / 2,6	XC5VLX30 / 93% / 65% / 2,7
FFT	EPC2C5 / 95% / 0% / 2,2	XC3S200 / 91% / 0% / 2,5	XC2V1000 / 99% / 0% / 2,3	XC5VLX30 / 98% / 0% / 2,1
FIR	EPC2C20 / 99% / 0% / 3,4	XC3S1000 / 91% / 0% / 3,8	XC2V1000 / 91% / 0% / 2,4	XC5VLX30 / 83% / 0% / 2,4
Net	EPC2C8 / 96% / 21% / 2,8	XC3S400 / 90% / 81% / 3,2	XC2V1000 / 92% / 37% / 1,9	XC5VLX30 / 94% / 81% / 2,6
Xbar	EPC2C8 / 95% / 0% / 2,8	XC3S1000 / 96% / 0% / 3,5	XC2V1000 / 96% / 0% / 2,3	XC5VLX30 / 94% / 0% / 3,0
SoC	EPC215 / 68% / 22% / 2,2	XC3S1000 / 74% / 87% / 2,8	XC2V1500 / 70% / 43% / 1,9	XC5VLX30 / 55% / 62% / 1,6

huge amount of simple Boolean functions, bit shuffling operations, and shift registers.

- In the *signal processing* section, we chose an FFT module [20] and an FIR filter. These modules consist of several multiply-accumulate operations and lots of distributed registers. For the FIR case, all implementations utilized 100% of the dedicated multipliers offered by the different FPGA architectures.
- In the *communication* section, we chose a network router [19] (named *NET*) and a crossbar (named *Xbar*). While the router consists of a huge amount of communicating state machines, the crossbar consists mostly of wide multiplexers.

Table I summarizes the synthesis results for all examined FPGA families. The table lists the device, the logic utilization, and the memory utilization for the different test modules. We found that the synthesis tools significantly influence the amount of module complexity that can be placed on an FPGA device. All bitstreams for Altera devices have been synthesized with the Quartus-II 6.2 tool chain and respectively with the ISE 8.2 software for the Xilinx devices.

IV. EXPERIMENTAL RESULTS

Figure 7 lists the measured compression ratios for all benchmark bitstreams and some decompressor variations. As a reference, the figure lists the compression ratios achieved with gzip (V 1.3.5 --best) and for two vendor-specific dedicated configuration memory devices [22], [23] that both perform bitstream decompression. The results for our techniques reveal the best compression ratios for the Huffman decompressor that in some cases outperforms gzip. Unfortunately, as shown in Table II, our Huffman decompressors consume a significant amount of hardware resources and achieve only moderate clock frequencies.

With two exceptions, one of our run length decompressors ranges always better than the dedicated configuration devices that both allow a maximal output data rate of up to 40 MB/s which seems to be limited by the internal flash memory. However, except the Huffman case and the CPLD implementation, all of our implementations allow high clock frequencies with up to 200 MHz, and in each cycle, a word may be emitted.

The T-RLE* compression algorithm is based on a prefix free encoding for the run length which has not been synthesized because of its ineptness (see Section II-C.1). The output word width of the toggle run length accelerators is 1 bit

while all other decompressors have an 8-bit output. Only the LZSS16 module has a 16-bit output that allows, due to the high clock speed, a configuration data rate of up to 400 MB/s.

TABLE II

SYNTHESIS RESULTS OF DIFFERENT DECOMPRESSION ACCELERATORS.

The T-RLE_{CPLD} toggle RLE implementation was synthesized for an Altera EPM7064-10 CPLD, while the other results are for a Xilinx XC2V40-5.

	T-RLE _{FPGA}	T-RLE _{CPLD}	F-RLE	LZSS8	LZSS16
LUTs / MCs	111	57	85	83	120
F _{max} [MHz]	165	67	193	198	200

The following synthesis results for the static Huffman decompressors are achieved for devices having the slowest available speed-grade.

Huffman	Cyclone	Spartan-III	Virtex-II	Virtex-IV
LUTs	4451	4223	5286	4114
F _{max} [MHz]	50	60	45	66

As discussed in Section II, the maximum clock frequency and the achieved compression ratio will not automatically deliver the minimal reconfiguration time. Exemplary, we measured the impact of the memory bandwidth for a fixed compression ratio to the configuration time for one FFT benchmark bitstream ($m=131$ kB for a Xilinx Spartan-III FPGA) that is decompressed by the LZSS8 algorithm to $\eta = 41\%$. We used an RTL simulation in order to measure the exact decompression time. Furthermore, in order to examine the impact of a FIFO between the memory and the decompressor, we performed these measurements for different FIFO sizes. While the decompressor was working at full system clock speed of 100 MHz (d_{FPGA}), we restricted the accesses to the memory to λ times of the system clock by allowing a read operation only every $\frac{1}{\lambda}$ cycle. The complete data path is 8 bit wide, thus, under optimal conditions the configuration time to configure a bitstream of size m is $t_{conf} = \frac{m \cdot \eta}{d_{FPGA} \cdot \lambda}$, when the configuration interface is not limiting the time to $\frac{m}{d_{FPGA}}$. For the case that the memory can deliver data at full system clock speed ($\lambda = 1$), the configuration time for the $m=131$ kB bitstream is 1.13 ms and the compression brings no benefit with respect to the reconfiguration time.

Table III lists the configuration times for slower memories, the optimal time based on the memory data rate and the compression ratio, and the time a reconfiguration would take without compression. The results are for a benchmark bitstream that is close to the average found over all bitstreams and all accelerators. As can be seen, we cannot reach the optimal reconfiguration times, but still we can enhance the configu-

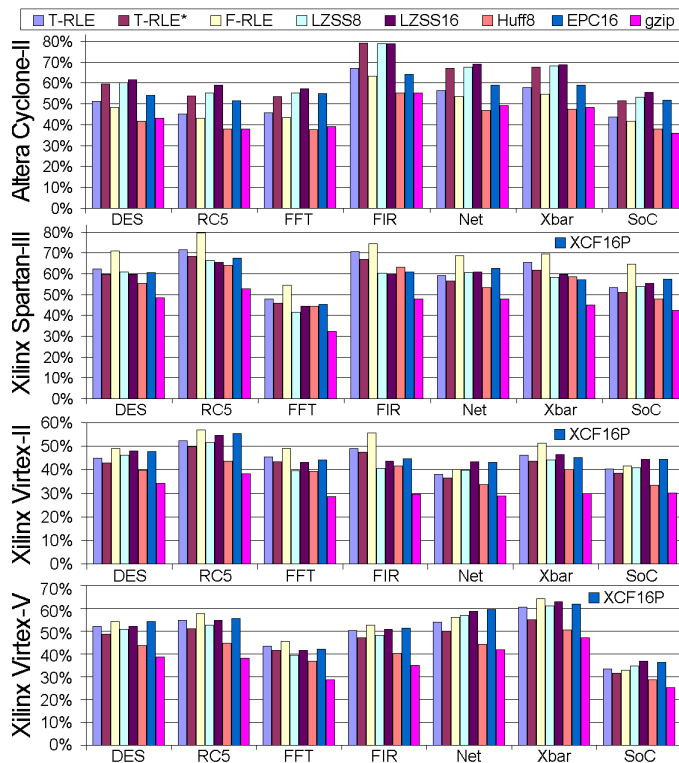


Fig. 7. Achieved compression ratios for the benchmark bitstreams in Table I.

ration speed significantly by balancing compression ratio and data rates between the configuration memory and configuration interface. Furthermore, the table points out that the FIFO size does not have much impact on the final configuration time. Experiments with larger FIFOs have not demonstrated a significant further improvement of the configuration speed. Consequently, an extra input FIFO can be omitted.

V. CONCLUSIONS AND FURTHER WORK

In this paper, we have proposed modifications of traditional compression algorithms such as run length compression, Lempel-Ziv, and Huffman encoding. The modifications allow to configure today's high-end FPGAs at full configuration speed of 400 MB/s from memories that deliver just the half bandwidth. This speed is sufficient to exchange a 1 million gate equivalent FPGA design within a millisecond. The achieved compression ratios can compete with state of the art software solutions and the required hardware for the

TABLE III

SUMMARIZED CONFIGURATION TIMES FOR DIFFERENT FIFO SIZES AND DIFFERENT MEMORY BANDWIDTHS $\lambda = \frac{d_{MEM}}{d_{FPGA}}$ FOR THE SPARTAN-III FFT BENCHMARK BITSTREAM. THE TABLE LISTS THE TIME IN MS.

	Fifo size in byte						optimal	no compr.
	1	4	8	16	32	64		
$\lambda = \frac{1}{2}$	1.78	1.76	1.74	1.72	1.70	1.69	1.13	2.62
$\lambda = \frac{1}{3}$	2.3	2.27	2.25	2.23	2.22	2.2	1.61	3.93
$\lambda = \frac{1}{4}$	2.84	2.82	2.80	2.78	2.78	2.77	2.15	5.24

decompression accelerators consume less than one percent of the logic resources of such a 1 million gate equivalent FPGA.

Future work will continue to integrate the high speed decompressors into SoC designs by the use of DMA mechanisms.

VI. ACKNOWLEDGMENTS

This work is supported in part by the German Science Foundation (DFG) Priority Programm SPP 1148 Rekonfigurierbare Rechensysteme under grant Te163/10-2.

REFERENCES

- [1] Altera Inc., *Altera Devices*, jun 2007, www.altera.com/products/devices/dev-index.jsp.
- [2] Xilinx Inc., *Xilinx : Silicon Devices*, jun 2007, www.xilinx.com/products/silicon_solutions/.
- [3] Zhiyuan Li and Scott Hauck, "Don't Care Discovery for FPGA Configuration Compression," in *FPGA '99: Proceedings of the 1999 ACM/SIGDA 7th International Symposium on Field Programmable Gate Arrays*. New York, NY, USA: ACM Press, 1999, pp. 91–98.
- [4] Dept. of Computer Science 12, *Bitstream Compression Benchmark*, <http://www.reconets.de/bitstreamcompression/>.
- [5] Volker Heun, *Grundlegende Algorithmen*. Vieweg, 2003.
- [6] David Salomon, *Data Compression - The Complete Reference*. Springer, 2004.
- [7] Dirk Koch and Jürgen Teich, "Platform-Independent Methodology for Partial Reconfiguration," in *Proceedings of the first conference on Computing Frontiers*. ACM Press, 2004, pp. 398–403.
- [8] Jacob Ziv and Abraham Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.
- [9] James A. Storer and Thomas G. Szymanski, "Data Compression via Textual Substitution," *J. ACM*, vol. 29, no. 4, pp. 928–951, 1982.
- [10] Scott Hauck and William D. Wilson, "Runlength Compression Techniques for FPGA Configurations," *Field-Programmable Custom Computing Machines*, pp. 286–287, 1999.
- [11] Xilinx Inc., *Virtex-II Platform FPGA User Guide*, 2005.
- [12] Xilinx Inc., *Xilinx Libraries Guide ISE 6.3i*, 1998.
- [13] Ju Hwa Pan and T. Mitra and Weng-Fai Wong, "Configuration Bitstream Compression for Dynamically Reconfigurable FPGAs," in *IC-CAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, Washington, DC, USA, 2004, pp. 766–773.
- [14] Zhiyuan Li and Scott Hauck, "Configuration Compression for Virtex FPGAs," in *FCCM '01: Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 147–159.
- [15] Michael Hübner, Michael Ullman, Florian Weissel and Jürgen Becker, "Real-time Configuration Code Decompression for Dynamic FPGA Self-Reconfiguration," *RAW04, Santa F, New Mexico, USA*, April 2004.
- [16] A. Dandalis and V. K. Prasanna, "Configuration Compression for FPGA-based Embedded Systems," in *FPGA '01: Proceedings of the international symposium on field programmable gate arrays*, New York, NY, USA, 2001, pp. 173–182.
- [17] David A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, September 1952.
- [18] Gregor Wetekam and Bernd Lutz, "Hardware-Implementierung einer 3D-Huffman-Decodierung für dynamische Volumendaten," *Hardware for Visual Computing Workshop*, 2005.
- [19] T. Streichert, D. Koch, C. Haubelt, and J. Teich, "Modeling and Design of Fault-Tolerant and Self-Adaptive Reconfigurable Networked Embedded Systems," *EURASIP Journal on Embedded Systems*, vol. 2006, p. 15, Apr. 2006.
- [20] <http://www.opencores.org>.
- [21] D. Koch, M. Körber, and J. Teich, "Searching RC5-Keys with Distributed Reconfigurable Computing," in *Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 2006)*. Las Vegas, USA: CSREA Press, June 2006, pp. 42–48.
- [22] Xilinx Inc., *Platform Flash In-System Programmable Configuration PROMs*, 2007.
- [23] Altera Inc., *Enhanced Configuration Devices (EPC4, EPC8 & EPC16) Data Sheet*, 2007.