THE ENTROPY OF FPGA RECONFIGURATION

Usama Malik and Oliver Diessel

School of Computer Science and Engineering,
University of New South Wales, Sydney, Australia &
Embedded, Real-time, and Operating Systems (ERTOS) Program,
National ICT Australia†
{umalik,odiessel}@cse.unsw.edu.au

ABSTRACT

In line with Shannon's ideas, we define the entropy of FPGA reconfiguration to be the amount of information needed to configure a given circuit onto a given device. We propose using entropy as a gauge of the maximum configuration compression that can be achieved and determine the entropy of a set of 24 benchmark circuits for the Virtex device family. We demonstrate that simple off-the-shelf compression techniques such as Golomb encoding and hierarchical vector compression achieve compression results that are within 1-10% of the theoretical bound. We present an enhanced configuration memory system based on the hierarchical vector compression technique that accelerates reconfiguration in proportion to the amount of compression achieved. The proposed system demands little additional chip area and can be clocked at the same rate as the Virtex configuration clock.

1. INTRODUCTION

Several researchers have shown that FPGA configuration data corresponding to typical circuits can be compressed by 20-95% (e.g. [1, 2]). However, it is not clear how the performance of various compression techniques can be compared against each other. Indeed, what are the limits of configuration compression? Moreover, what parameters of circuits and devices impact upon the performance of these techniques? To address these issues, this paper proposes an objective measure of how much a given configuration bitstream can be compressed and suggests simple methods to achieve reasonable compression for a set of benchmark circuits on Xilinx Virtex devices. Section 3 defines *entropy of reconfiguration* to be the entropy of the configuration bitstream that is required to configure a given input circuit. The entropies of the benchmark circuits are calculated, thereby providing an estimate of the possible reduction in their configuration data sizes. In the light of this analysis, Section 4 studies two simple compression techniques: Golomb encoding and hierarchical vector compression. It is shown that these methods perform within 1-10% of the compression bound. Vector compression is chosen for hardware implementation due to its simplicity. A scalable hardware decompressor is presented and analysed in Section 5.

†National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

2. BACKGROUND

This paper employs basic results of information theory to define an objective measure of configuration compression performance. We are not aware of any previous work along these lines. In addition to this, the analysis of this paper shows that simple off-the-shelf techniques provide reasonable compression in practice. These algorithms run in linear time and do not use complex procedures to re-order input data as is the case for the methods presented in [1, 2]. In this paper, we argue that the compression performance achieved by the suggested simpler methods is also superior. Finally, our decompressor is readily implemented in hardware and scales well with configuration memory and port sizes.

The work presented in this paper extends the analysis reported in [3, 4] and focuses on the amount of non-null data in a given circuit configuration. We first note that the *null* configuration for Virtex devices does not just consist of zeros. Let there be a *null* configuration, ϕ, represented as a bit vector of size n bits. Let there be a circuit configuration C also of size n bits. Let k be the number of bits in C that differ from the corresponding bit in ϕ. A new bit vector, ϕ', of size n is constructed by clearing all bits in ϕ that remain unchanged in C while setting the rest. In other words, ϕ' represents the positions in ϕ where the bits need to be flipped to produce C.

In order to understand the characteristics of a typical ϕ', we considered a set of benchmark circuits on Virtex devices. These circuits were collected from the Xilinx core generator, from *opencores.org* and from [5]. In all experiments, we only considered the frames that configure the columns of CLBs because these constitute a majority in the overall

bitstream. The total amount of CLB configuration data for the device under consideration will be referred to as n in the subsequent discussion. Table 1 lists the relevant parameters of the circuits (the number of 4-input LUTs and the number of nets in the un-routed netlist). A '-' in the XCV200 column means that the corresponding circuit could not be mapped onto that device. A '*' in the *Circuit* column means that the circuit could not be mapped to smaller devices because there were not enough IO blocks. For all circuits, the Xilinx ISE5.2 tools were required to optimise area use. Configuration data was generated for each instance of circuit mapping and compared with the corresponding null configuration at the bit level using Xilinx JBits2.8. The ϕ' vector corresponding to each configuration was generated. This allowed us to determine the number of non-null bits, k, for each configuration. The results for three devices, XCV200, XCV400 and XCV1000 can be found in Table 1, which shows that the value of k changes little from one device to another. We also observe that $k << n$ in each instance and increases with the circuit size.

3. ENTROPY OF RECONFIGURATION

Our interest lies in finding the minimum amount of configuration data needed to configure a given circuit on a given device. Considering a circuit configuration as a bit string, we are interested in finding the length of the shortest string representing that configuration, i.e. its Kolmogorov complexity. However, finding the Kolmogorov complexity of an arbitrary string is NP hard. We therefore take the approach commonly used in the field of text compression. If we can model the data source, i.e. can determine the probabilities of various symbols it outputs, then we can easily determine its entropy, which gives us a bound on compressibility. This is what we show in the subsequent sections.

Definition: Let us recall the definition of *entropy* (also called *Shannon's entropy*). Let $\mathbb{X}$ be a discrete random variable on a finite set. Let the probability distribution function of $\mathbb{X}$ be $p(x) = Pr(\mathbb{X} = x)$. The *entropy*, $H(\mathbb{X})$, can be defined as: $H(\mathbb{X}) = -\sum_{x \in \mathbb{X}} p(x) log_2(p(x))$[6]. The entropy of a *memoryless* information source determines the minimum channel capacity that is needed for a reliable transmission of the source. In other words, entropy provides an estimate of the *average* minimum number of bits that are needed to encode a string of symbols produced by the source. Encoding a message with less than $H(\mathbb{X})$ bits per symbol will result in a loss of information (or the communication will be unreliable).

Consider an FPGA that is in an unknown configuration state and a new circuit that is to be configured onto the device. We define *entropy of reconfiguration*, H_r, to be the entropy of the data source that generates the configuration bitstream required to configure the input circuit onto the target FPGA. The interpretation of H_r is that it defines the minimum number of bits/symbol needed to configure the required circuit and therefore provides an estimate of the maximum compression possible for the configuration. Application of this method presupposes that FPGA configurations can be modelled as strings of randomly generated symbols without significant error. We are therefore charged with finding suitable symbol sets and evaluating a representative set of configurations to determine the validity of randomness assumption. Assuming we can do so, we are then able to assess the performance of given compression heuristics and obtain lower bounds on the delay involved in configuring the circuit.

Modelling Configuration Data Sources: We aim to define a suitable symbol set over ϕ' and to assign probability distributions to these. The most striking feature of the ϕ' vectors is their sparsity, i.e. long runs of zeros. Given this observation, we consider the runlengths of zeros as our symbol set. Let X be a random variable that specifies this run-length where $X \in \{0, 1, 2, ..., n-1\}$. In other words, $X = i$ means that the output symbol contains i zeros followed by a one. In the following discussion, *a run of length i bits* means i zeros followed by a one. The problem of finding a probability distribution function for our model data source can thus be formulated as finding a probability distribution of X.

To find a probability distribution function for our benchmark ϕ' configurations, we considered the frequency with which runs of various lengths occur in the test data. Let $f(i)$ be the number of times a run of length i bits occurs in a given ϕ'. Without loss of generality let us assume that the first and the last bits in ϕ' are zeros. With this assumption, the total number of runlengths in ϕ' is $k+1$. Thus, the probability that a run of length i bits occurs in ϕ' is given by $\frac{f(i)}{k+1}$. We examined our benchmark ϕ' for various devices. We illustrate the results by considering the ϕ' for *fpu* on an XCV400. We found that $P(X = 0)$ was approximately 0.25. The remaining run-lengths were distributed as illustrated in Figure 1. The other ϕ' configurations in the benchmark exhibited a similar trend.

Measuring Entropy of Reconfiguration: The entropy of reconfiguration for each benchmark circuit, represented as a ϕ' vector, was calculated with runlengths of zeros as the symbol set. Results corresponding to circuits mapped onto various devices are recorded in Table 1 under the columns headed H_r. The minimum bitstream size for a circuit is estimated by $k \times H_r$. Thus, the estimated minimum number of bits needed to encode the *fpu* ϕ' for an XCV400 is $155,387 \times 4.66 = 724,103$. This is 31.4% of the size of the complete CLB configuration for an XCV400. In other words, the best compression possible for this circuit configuration is 68.6% (Table 1 column *Shann. % red.*). We round the result due to uncertainty in the results as indicated.

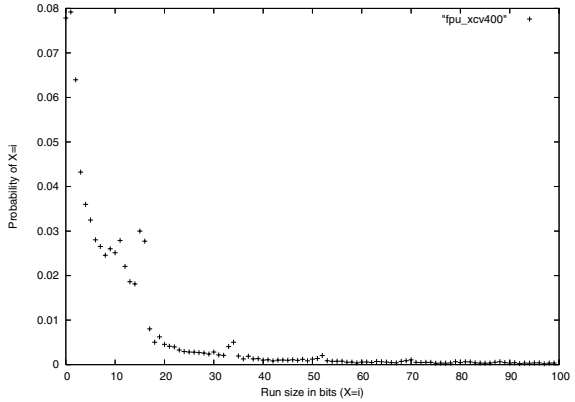

Fig. 1. The relationship between $P(X = i), i > 0$ in fpu_{xcv400}.

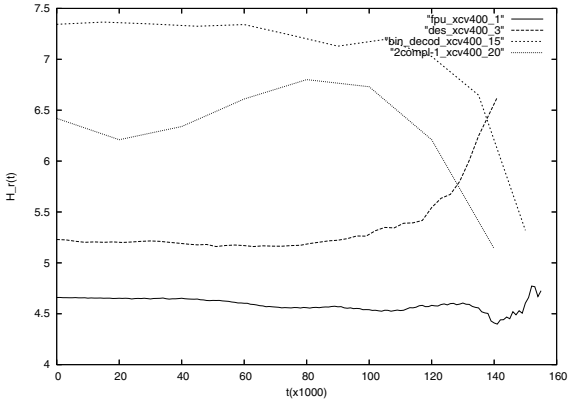

Fig. 2. H_r^t as a function of the number of symbols dropped.

We have found it difficult to convincingly demonstrate the randomness of runlength symbols in any configuration using straightforward approaches. Sophisticated techniques such as Fourier analysis may validate the assertion, but we have not yet attempted this. In this paper we report upon a simple method we have applied to the data to gain some confidence that the data is sufficiently random for first order estimates of compressibility.

The motivation behind the method is the fact that the entropy of a random process is independent of the number of symbols already produced. By verifying that the calculated entropy of successively shorter tails of our benchmark configurations does not change significantly, we gain some confidence that runlengths (set bits) are randomly distributed throughout the data.

We recalculated the entropies H_r^t of all configurations having skipped the leading t symbols in the ϕ' bitstreams. The results for 4 circuits that were mapped to an XCV400 and are representative of the range in complexity and size present in the benchmark set appear plotted in Figure 2. For these plots we calculated H_r^t at increments of $t = 1000$. Since the number of symbols $k + 1$ per configuration varies substantially for these circuits, we scaled the plot for *2compl-1* by a factor of 20, for *bin_decod* we scaled the plot by a factor of 15 and for *des* by a factor of 3.

The results for all plots with $t < k/2$ are relatively constant, which is encouraging. As t is increased further, the number of symbols left in the tail becomes too small to accurately measure the probabilities of individual symbol occurrences. It seems reasonable that circuit flattening resulting from synthesis and place and route tools should result in a relatively random use of resources and that this ought to produce a corresponding randomness in the setting of switches as given by ϕ'.

4. TECHNIQUES FOR COMPRESSING CONFIGURATION DATA

This section discusses configuration compression in the light of the model outlined in the previous section. We assume that the decompressor has a knowledge of the null bitstream and it can reproduce a configuration given its ϕ'. The task of the compression algorithm is to compress ϕ'. We evaluate the performance of two simple compression schemes: Golomb encoding and hierarchical vector compression and show that they perform reasonable compression of the benchmark configuration data.

Golomb encoding: Golomb encoding is a variable-to-variable encoding that can be considered to be a variant of runlength encoding. Let us suppose that we would like to encode the runlengths of zeros in an input ϕ'. The runlengths can be of size $0, 1, 2, ...n$. Golomb encoding divides the runlengths into groups of size m. The parameter m is the optimisation parameter. The j_{th} group of runlengths is assigned a unique *group prefix* by concatenating j ones followed by a zero. Within each group, the m runlengths are identified using a binary code (also called the *tail*). Thus, each runlength can be uniquely identified by concatenating its group pre-fix with its tail.

Our ϕ' configurations were compressed using Golomb encoding for various m ranging from 2 to 512. Results for an XCV400 are recorded in Table 1 under the column heading *Glb. % rd.* This table presents the percentage reduction in the amount of configuration data for various values of m compared to n. The m we found best is listed in the next column. Comparing these figures with the Shannon values shows that Golomb encoding performs reasonably well assuming the optimisation parameter is adaptive. Similar results were found for all device sizes.

Hierarchical vector compression: Another technique for compressing sparse bit vectors is hierarchical vector compression (discussed in [7]). Let us refer to the uncompressed

Circuits	#4-inpt LUTs	#Un-rtd nets	XCV200 (n=1,161,216)		XCV400 (n=2,304,000)		XCV1000 (n=5,750,784)		Compression results (XCV400)						
			k (bits)	H_r	k (bits)	H_r	k (bits)	H_r	Shan. %rd.	Glb. %rd.	m	VA. %rd.	b	Virt. %rd.	Arch. %rd.
encoder	127	456	4,302	5.48	4,394	5.36	4,320	5.28	99	98	256	98	8	76	97
uart	93	467	5,321	5.39	5,129	5.10	5,536	5.15	99	98	256	98	8	64	97
asyn-fifo	22	584	5,441	6.00	5,885	5.69	5,913	5.69	99	97	256	97	8	45	96
*add-sub	49	344	-	-	5,997	6.59	6,155	5.84	98	97	256	97	8	46	96
*2compl1	N/A	N/A	-	-	7,806	6.50	9,212	6.18	98	97	256	96	8	40	95
spi	150	796	7,983	5.60	7,956	5.63	8,041	4.93	98	97	256	97	8	60	96
fir-srg	216	726	8,534	4.93	8,503	4.92	8,169	4.72	98	96	256	97	8	78	96
dfir	179	782	7,981	5.30	8,535	5.09	8,710	4.91	98	96	256	97	8	60	96
cic3r32	152	736	9,061	5.00	9,092	4.88	8,478	4.79	98	96	128	97	8	67	96
ccmul	262	905	9,956	5.67	9,956	5.66	10,215	5.55	98	96	128	96	8	63	95
*bin.decod	288	1,249	-	-	10,670	7.33	10,648	6.66	97	96	128	95	4	21	94
*2compl2	129	388	-	-	11,154	6.75	12,738	6.61	97	95	128	94	4	24	94
ammod	271	990	11,546	5.21	11,653	5.24	12,032	5.27	97	95	128	95	8	43	95
bfproc	418	1,347	14,753	5.04	14,859	5.16	15,497	5.34	97	94	128	94	4	25	94
costLUT	547	2,574	16,424	5.54	16,752	5.76	16,093	5.13	96	94	128	94	4	47	93
gpio	507	3,022	30,762	5.35	30,924	5.56	32,226	5.92	93	89	128	90	4	26	90
iir	894	2,907	34,830	4.81	33,648	4.68	33,506	4.67	93	88	32	90	4	48	90
des	132	5,060	-	-	48,118	5.23	49,827	5.88	89	85	32	86	4	10	86
cordic	1,112	4,745	48,759	4.71	49,364	4.62	50,202	4.70	90	85	32	87	4	38	87
rsa	1,114	5,039	49,179	4.78	50,121	4.88	51,283	5.10	89	84	32	86	4	26	86
dct	1,064	5,327	52,916	4.84	52,999	5.46	53,959	5.08	89	84	32	86	4	20	86
blue-th	2,711	11,152	-	-	100,996	4.90	101,776	5.39	78	73	16	74	4	0	74
vfft1024	3,101	11,405	-	-	113,695	4.53	114,648	4.75	78	70	16	73	4	3	73
fpu	3,914	13,522	-	-	155,387	4.66	155,354	5.01	69	63	16	62	4	0	62

Table 1. Predicted and observed reductions in each ϕ' configuration.

vector as Level-0 (l_0). We also refer to the l_0 vector as the *vector address* (VA) of the set bits in ϕ'. The l_0 vector can be compressed as follows. Partition the l_0 vector into equal size *blocks* each of size b_0 bits. Next, drop the blocks that only contain zeros. To reconstruct the original vector we need to know where to insert the zeros. Therefore, create a new vector (l_1) whose length, $|l_1|$, is equal to the number of blocks at l_0. The leftmost bit of this new vector corresponds to the leftmost block of the l_0 vector, and so on. A bit in the l_1 vector is set if there is a bit set in the corresponding block. Compress the resulting l_1 vector by applying the above procedure recursively using a block size of b_1. This procedure can be repeated until a reasonable compression is achieved. The j number of levels compressed to and the associated block sizes, $b_0, b_1..b_{j-1}$, are the optimisation parameters in this procedure.

Hierarchical vector compression was applied to each ϕ' of the benchmark configurations for various sized devices. The block size, b, was varied from 2 to 128 and was applied at all levels. Between two and five levels of compression in total were considered. Using this approach, the best results were obtained for all circuits when three levels of compression were used with $b = 4$ or $b = 8$. Table 1 contains the results for XCV400 configurations (the results for the other devices were similar). The column under the heading *VA. % rd.* lists the percentage reduction compared to n while the optimal values of b are listed in the next column. We observe that vector compression also performs reasonably well. The column under the heading *Virt. %rd.* lists the the amount of non-null frame data as a percentage of n for an XV400. This would be the reduction in configuration data achieved if one used Virtex' frame oriented partial reconfiguration to only load non-null frame assuming the device is in its default configuration state. A second observation is that its optimisation parameters, j and b, vary less as compared to the optimisation parameter, m, of Golomb encoding. We therefore choose vector compression as our preferred method for hardware implementation.

Comparison with other approaches: Given the probability distributions of our benchmark circuits, it is clear that Huffman encoding will be sub-optimal since the symbol frequencies do not decrease in decreasing powers of two. Dictionary based methods will also perform sub-optimally since many bits are wasted when fixed length indices are used to access infrequent patterns in the dictionary. Another caveat with these methods is the hardware complexity of the corresponding decompressors.

The results published in [1] and in [2] are the most relevant to our work. The work in [1] reported best compression when LZSS was used in conjunction with frame re-ordering (30-95% reduction in configuration data for a variety of circuits on a variety of Virtex devices). The method published in [2] exploited intra-configuration regularities by first constructing bit vectors needed to transform a given frame into

another. The runs of zeros in these vectors were compressed using Huffman encoding. For a set of benchmark circuits on variously sized Virtex devices, the authors use both their method and the LZSS based methods and show that their technique acheives better compression. However, the overall reduction in bitsreams ranges from 25% to 80%. Both papers are not clear on how the circuit utilisation was measured. A direct comparison on compression performance is therefore difficult. We also comment that any method that relies upon removing inter-frame regularities mostly compresses null data (as per our analysis). This is likely to increase the overall entropy because the non-null bits are almost randomly located and writing one frame on top of another will *add* null bits into the *next* frame to clear the set bits of the previous frames.

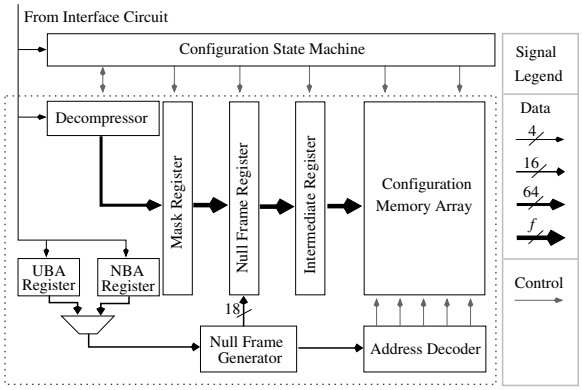

Fig. 3. The proposed memory architecture.

5. DECOMPRESSING CONFIGURATIONS IN HARDWARE

We enhance the current Virtex configuration architecture [8] to incorporate a *vector decompressor*. The new memory internally generates null frames while the input ϕ' configuration is being decompressed. The generated null frames are then modified based on the uncompressed ϕ' data before writing to the target frame registers. The new design supports partial reconfiguration in the same manner as the current Virtex. Vector compression, with $b = 4, j = 3$, is therefore performed individually on a frame by frame basis (as opposed to considering the entire configuration as a single vector). The user supplies a block of non-null, or *user*, frames, and a list of addresses where null frames need to be added to clear the remains of any previous circuit.

Memory design: In the enhanced Virtex, a *decompression system* is added to receive ϕ' vectors (Figure 3). This system outputs uncompressed data into a *mask register*. An f-bit *null frame* register is added whose inputs come from a *null frame generator*, where f is the number of bits in the frame. The null frame generator takes in a frame address and outputs the 18 bits that are written to each CLB at that address. This data is broadcast to and stored in the null frame register. Once the mask register is full, i.e. it contains a frame worth of ϕ', the contents of the null frame are written into the *intermediate register* based on the contents of the mask register. A bit in the null frame register is written to the intermediate register as is if the corresponding bit in the null frame is cleared, otherwise it is complemented.

The memory operates in a pipelined fashion. As decompressing a frame can take several cycles, we use the null frame generator in the background to generate null frames for the null blocks. The null blocks address is stored in the null block address (NBA) register. When a ϕ' vector for a frame is ready, the null frame generator is switched to generate a null frame for the user block address (UBA) register instead of the null block address. The required user frame is then written to in the intermediate register as explained above. The process iterates until all user frames have been loaded. When there are many more null than user frames to be written, configuration may need to continue after decompression has finished to complete circuit loading. Typical configurations need many more cycles on average to load user frame data, and therefore provide ample time for all null frames to be loaded in the background.

Decompressor design: The configuration port size in the new memory is set to 4 bits so that it matches the block size. As three levels of compression are used, four bits of the Level-3 vector span 256 bits at Level-0. The decompressor therefore operates in units of 256 bits (Figure 4). These units are sequentially selected using a *control shift register* which is initialised by asserting the topmost bit. Note that there can be a final *partial* block in the mask register because Virtex frames are not always a multiple of 256 (e.g. XCV400 has an 800-bit frame).

An input frame is decoded by traversing the various compressed levels in parallel. The Level-i vector is latched into the i_{th} vector-address-decoder, VAD_i, one 4-bit compressed VA word at a time. Each decoder produces a sequence of 4-bit output vectors that respectively indicate which bits in the VA word, from most to least significant, are set. The l_0 4-bit word is directed to the VAD register which is of size 4×16 bits. In turn, the VAD_1 output vectors to indicate which 4-bit block of the VAD register the next 4 bits from the interface circuit are written to. VAD_2 output vectors control which block of 16 bits the VAD_1 vectors refer to, and the VAD_3 vectors determine which 64-bit block of the mask register the VAD register contents are written. The output of VAD_3 is lateched into the mask control (MC) register and togther with the contents of the control shift register forms the final control signal for the mask register. Once 256-bits of a unit are updated, the control shift register is signalled and the next unit is selected. The cycle repeats until all units

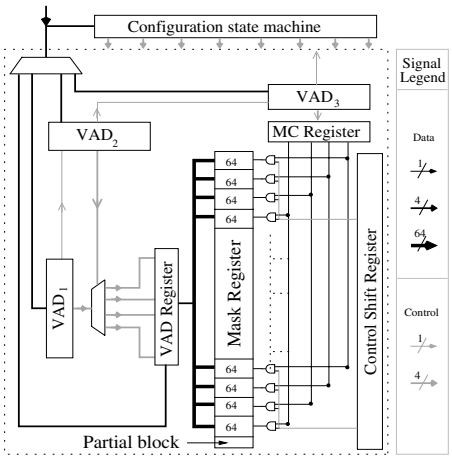

Fig. 4. The design of the decompressor.

in a frame are updated. The internal details of a VAD can be found in [9].

Design analysis: As discussed in the previous section, our solution strategy is to perform decompression on a frame-by-frame basis. This method is likely to increase the size of the compressed bitstream as the frame sizes cannot always be an integral multiple of b^{j+1}. Due to this factor, additional data needs to be inserted at higher levels to make for even sized blocks. We compensated for this in the results shown for an XCV400 device under the $Arch.\%red$ column of Table 1 These results show that the enhanced configuration architecture achieves comparable results to the previously analysed methods.

In terms of area, the main component that is added to the existing Virtex is the decompressor. This system contains three vector address decoders and a 64-bit configuration bus. Each VAD contains four flip-flops and eight two input gates and thus requires a small area. This circuit can easily be clocked at 66MHz which is the configuration speed of the current Virtex (please see [9] for more details). The existing Virtex already contains a 32-bit configuration bus. Thus, our design adds a 32-bit bus that spans the height of the chip and a small number of gates for the decoding system.

The design we presented assumes a 4-bit configuration port. This system can be readily scaled to wider port sizes. Assume that the port size is increased from 4 to $4p$ where p is a strictly positive integer. We scale the system by implementing p decompressors each with its own mask register. Each p is assigned a 4-bit portion of the port and operates independently of the other decompressors. The p decompressors share a single 64-bit bus to reach their assigned mask register, A priority decoder can be used to resolve any conflicts between various decompressors that are attempting to access the bus simultaneously. This can result in a loss of throughput as some decompressors must be stalled. A simulation of the scaled memory system showed that the proposed architecture reduced reconfiguration time to within 15% of a linearly scaled improvement for p upto 16.

6. CONCLUSIONS & FUTURE WORK

This paper has developed the idea of the *entropy of reconfiguration* allowing us to determine the quality of configuration compression. Practical methods and device enhancements for achieving compression bounds on Virtex were described. In an effort to generalise the results for non-Virtex devices, the VPR placement and routing tools are currently being used to examine a wide range of hypothetical fine-grained island-style FPGAs.

The results of this paper show that circuits, when mapped onto a device using CAD tools, introduce *randomness* at the configuration data level. Circuits that do not exhibit this property can certainly be constructed. This is likely to be the case when circuits are hand-mapped to achieve regularity, or configuration data corresponding to a sub-circuit is simply copied to various locations on the device. Configuration compression for this broader class of circuits is the subject of future research.

7. REFERENCES

[1] Z. Li and S. Hauck, "Configuration compression for Virtex FPGAs," in *IEEE Symposium on FCCM*, 2001, pp. 111–119.

[2] J. Pan, T. Mitra, and W. Wong, "Configuration bitstream compression for dynamically reconfigurable FPGAs," in *International Conference on CAD*, 2004, pp. 766–773.

[3] U. Malik and O. Diessel, "On the placement and granularity of FPGA configurations," in *International Conference on FPT*, 2004, pp. 161–168.

[4] U. Malik and O. Diessel, "A configuration memory architecture for fast run-time-reconfiguration of FPGAs," in *International Conference on FPL*, 2005, pp. 636–639.

[5] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*. Springer, 2001.

[6] C. Shannon, "A mathematical theory of communication," *Bell Systems Technical Journal*, pp. 379–423, 1948.

[7] Y. Choueka, F. Fraenkel, S. Klein, and E.Segal, "Improved hierarchical bit-vector compression in document retrieval systems," in *Annual ACM Conference on Research and Development in Information Retrieval*, 1986, pp. 88–96.

[8] D. Schultz, S. Young, and L. Hung, "Method and structure for reading, modifying and writing selected configuration memory cells of an FPGA," U.S. Patent 6 255 848, 2001.

[9] U. Malik and O. Diessel, "A configuration memory architecture for fast FPGA reconfiguration," *Technical Report UNSW-CSE-TR0509*, 2005.