Submitted to *IEEE Symposium on FPGAs for Custom Computing Machines*, 1999.

Runlength Compression Techniques for FPGA Configurations

Scott Hauck, William D. Wilson
Department of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208-3118 USA
{hauck, wdw510}@ece.nwu.edu

Abstract

The time it takes to reconfigure FPGAs can be a significant overhead for reconfigurable computing. In this paper we develop new compression algorithms for FPGA configurations that can significantly reduce this overhead. By using runlength and other compression techniques, files can be compressed by a factor of 3.6 times. Bus transfer mechanisms and decompression hardware are also discussed. This results in a single compression methodology which achieves higher compression ratios than existing algorithms in an off-line version, as well as a somewhat lower quality compression approach which is suitable for on-line use in dynamic circuit generation and other mapping-time critical situations.

Configuration Compression

Reconfigurable computing is an exciting new area that harnesses the programmable power of FPGAs. In the past, FPGAs were used in applications that required them to be configured only once or a few times. The infrequency in which the FPGAs were programmed meant that these applications were not limited by the device's slow configuration time [Hauck98a]. However, as reconfigurable computing is becoming more popular, the configuration overhead is becoming a true burden to the useful computation time. For example, applications on the DISC and DISC II systems have spent 25% [Wirthlin96] to 71% [Wirthlin95] of their execution time performing reconfiguration.

Reconfigurable computing demands an efficient configuration method. In order for reconfigurable computing to be effective, there must be a method to quickly configure the device with a minimal amount of data transfer. However, the amount of information needed to configure an entire FPGA can be quite large. Sending this large amount of information to the FPGA can be quite time consuming, in addition to power consuming.

A logical solution would be to compress the data stream sent to the FPGA. This would reduce the amount of external storage needed to hold the configuration, reduce the amount of time needed to send the configuration information to the device, and reduce the amount of communication through the power-hungry off-chip I/O of the FPGA. Once the configuration information arrives to the decompression hardware in the FPGA, it can be written to the configuration memory at a faster rate than would have been possible through the slow I/O of the device.

In previous work [Hauck98b, Li99] we developed a technique using the wildcard feature of the Xilinx XC6200 series FPGA [Xilinx97]. While this algorithm provided good compression results, it also requires a very complex compression algorithm, and may not achieve the best possible compression results.

In this paper we explore the configuration information of the Xilinx XC6200 series. Based on the nature of the data, several compression techniques will be proposed. Using these compression techniques, algorithms and support hardware structures are developed to compress the address/data pairs sent to the device. Next, these compression strategies are performed on a group of Xilinx XC6216 configurations to determine their performance. Finally, conclusions will be drawn on the capabilities of these techniques.

Configuration Information

The Xilinx XC6200 FPGA is an SRAM based, high performance Sea-Of-Gates FPGA optimized for reconfigurable computing. All configuration resources are accessed by addressing the SRAM through a standard memory interface. The Xilinx XC6200 series are partially reconfigurable devices. The configuration file consists of a set of address/data pairs. Since the device is partially reconfigurable, the target addresses written to may not be contiguous. Therefore, if the data is compressed the addresses must be compressed as well.

The configuration data falls into four major areas: cell function, routing, input/output buffers, and control. The addresses which configure the cell function, routing, and the input/output buffers will normally never be written to more than once in a configuration. The control data may be written to multiple times in a configuration, but this

data represents a very small fraction of the total configuration. In addition, the addresses that are accessed usually fall in sequential order, or into a series with a certain offset. The compression technique chosen for the addresses should take advantage of the consistent offsets, in addition to not requiring the repetition of identical data.

FPGAs configurations often exhibit a great deal of regularity, which is reflected in the configuration data. Upon examining a XC6200 configuration file, it is clear that many structures are duplicated by the repetitive sequences of data. For this reason, the compression technique chosen for the data should take advantage of the repetitive sequences.

Ordering Restrictions

The ability of the XC6200 series to perform partial reconfiguration allows for almost arbitrary reordering of the configuration data. Reordering the data may facilitate some additional compression techniques. However, a few restrictions do apply. These restrictions logically divide the data into three sections. These sections will be programmed to the device in order.

The first section contains part of the control configuration that must be written before any of the cell and routing configuration. This configuration data defines how the remaining cell, routing, and IOB configuration data will be interpreted. This data can be reordered.

The second section contains cell, routing, and IOB configuration data which is not bound to any control configuration data. There are no restrictions as to how this data can be reordered.

The third section contains control information that is bound to cell, routing, and IOB configuration data. This data cannot be reordered without affecting the result of the configuration. In addition, this section will contain control information that must be written at the very end of the entire configuration.

Note that although the address/data pairs can be reordered, each pair must be kept together, since if the addresses and data moved independently the data will be assigned to the wrong location in the chip. This can lead to conflicting optimization goals, since an ordering that may maximize address compression may hurt data compression and vice-versa.

Compression Considerations

The data compression for FPGA configurations must normally be lossless. Although the use of don't cares, based on research by Li and Hauck [Li99], may allow for lossy compression, this is an area of future work and will not be investigated in this paper. The chosen compression strategy must be able to completely recover the exact data that was compressed.

The compression technique chosen must allow for online decompression. Although compression will normally occur offline, where the entire configuration sequence is available, the entire compressed configuration sequence will not be available upon decompression. If off-line decompression were implemented, it would greatly increase the configuration time, in addition to requiring significant amounts of on chip memory. The chosen compression strategy must be able to decompress the data as it is received with a limited amount of special-purpose hardware. Finally, the compression technique may reorder the data, but it must stay within the guidelines previously described.

Run-Length Compression

A variation of Run-Length encoding perfectly meets the requirements for the address compression. A series of addresses with a common offset can be compressed into a codeword of the form: base, offset, length. Base is the base address, offset is the offset between addresses, and length is the number of addresses beyond the base with the given offset. For example, the following sequence of addresses: 100, 103, 106, 109, 112 can be compressed into the codeword: base = 100, offset = 3, and length = 4. This compression technique does not require repetitive data, and will take advantage of the sequences of addresses sharing a common offset.

The configuration data sometimes repeats data values many times. For this reason, we will attempt to compress the data streams with Run-Length encoding as well, although the compression may not be as great as that achieved with the addresses.

Lempel-Ziv Compression

Lempel-Ziv takes advantage of repetitive series of data. A series of data that is repeated can be replaced with a single compressed codeword. This codeword will tell when the series previously occurred, how long it is, and will

give a new piece of data that is not part of the series. The codeword will consist of the form: pointer, length, and lastSymbol. The pointer will represent where the start of the series previously occurred within a window of time [Ranganathan93].

For example, assume the window size is eight data items, and the window currently contains: C, B, A, D, A, F, A, L. These would be the last 8 values decompressed, with the most recent in the first position (the C in position 1). If the next four pieces of data are A, B, M, then we can send the codeword: pointer = 3, length = 2, and lastSymbol = M. This indicates that we start with the value decompressed 3 cycles ago (pointer = 3), take 2 values (A & B), and append M (lastSymbol = M). The new window, after this codeword is received, will be: M, B, A, C, B, A, D, A.

We have made a variation to the basic Lempel-Ziv algorithm that will allow for the length to exceed the window size. Therefore if a particular piece of data or a particular series of data is repeated many times, this hardware will be able to handle it. For example, assume the same window size and contents as before: C, B, A, D, A, F, A, L. If the next 13 pieces of data are: B, C, B, C, B, C, B, C, B, C, B, C, D, then we can send the codeword: pointer = 2, length = 12, and lastSymbol = D to represent this entire series. Since the number of characters copied (length = 12) exceeds the pointer value, some of the characters duplicated will be recycled and duplicated multiple times.

This will meet the requirements for the data compression. This cannot be used for the address compression since the address stream has almost no repetition.

Compression Strategies for Address/Data Pairs

There are several options as to how to compress the address and data pairs.

1) Basic Run-Length

This strategy begins by ordering the address data pairs within series one and two (described under Ordering Restrictions above) in numerical order by address. The three address series and three data series are compressed using Run-Length compression. This software compression requires a very minimal amount of time to run, and could even be done on-line.

2) Lempel-Ziv

This strategy begins by ordering the address data pairs within series one and two (described under Ordering Restrictions above) in numerical order by address. The three address series are then compressed using Run-Length compression and the three data series are compressed using Lempel-Ziv compression. Although this software compression is slightly slower than the Run-Length/Run-Length strategy, it is still very fast.

3) Run-Length with Reordering

This strategy uses a more intensive compression algorithm. It attempts to reorder the address data pairs in a more optimal manner. The algorithm performs the following reordering:

1) Sort the address/data list such that all alike data occurs together.
   NOTE: This logically separates the addresses into lists that belong to a certain data value.

2) Within each list of identical data values, order the addresses according to the following scheme:

   a) find the longest possible address series which can be compressed into a single runlength statement.

   b) create a codeword from that series and remove those addresses from the list.

   c) repeat a) and b) until no more addresses remain uncompressed.

   NOTE: At this point, each address list corresponding to a certain data value has been formed into codewords.

3) Attempt to order the data series groups such that address lists within them may cross boundaries (i.e. the last runlength of one series and the first of another can be fused into a single runlength statement).

4) Compress the data lists according to a Run-Length variation in which all codewords have a zero offset (i.e. the data must be identical, and no bits are wasted on sending the increment).

This algorithm runs at least an order of magnitude slower than the previous two strategies.

Bus Transactions

In order to implement these compression algorithms, we must determine a method for sending address and data codewords across the configuration bus to the FPGA. In order to make a fair comparison with other approaches we must guarantee that these transactions use only as many wires as the standard communication method.

Multiplexing of the Address Bus

Since the addresses must be compressed and sent in addition to the data, our compression strategies use both the address and data buses to send the compressed codewords. In other words, once the decompression sequence is initiated, there is no longer a concept of address, but instead all available address and data bits are used to send the data of the compressed codewords (be it compressed address or data). To initiate the decompression sequence one would either write the number of address/data pairs to the compression hardware, signaling it to take control of the address and data bus, or turn on and off compression by writing to a specific known address. Upon completion of the decompression, the decompression hardware would relinquish control of the two buses.

The XC6200 series has the ability to be configured with an 8 bit, 16 bit, or 32 bit data bus. The compression strategies will be tested on 8 bit configuration data, since we have benchmarks readily available with 8 bit configuration data. In addition, the benchmarks are designed for the Xilinx XC6216, whose address bus is 16 bits. Therefore, between the address and data bus there are a total of 24 available bits to send the compressed codewords.

Variation in Codeword Parameter Sizes

There is a great amount of freedom in the choice of the codeword parameter sizes. Within a codeword different parameters are not required to be represented by an identical number of bits. Depending on the number of bits devoted to each parameter, the codeword size may change as well. However, to simplify the hardware the codeword sizes will be restricted to a few values, namely 12 bits, 16 bits, and 24 bits. Independent of the compression technique and codeword size, the codewords will be packed to use all available bits for every transaction. By limiting the codewords to a few key sizes, the hardware which must unpack the codewords will be simplified.

Run-Length Bus Formats

In Run-Length, the size of the base is fixed by the width of the items being sent (addresses are 16 bit, data is 8 bit). However, the size of the length and offset can be varied within the limits of the codeword length.

Since the base must be 16 bits for address Run-Length codewords, address codewords must be 24 bits, thereby completely filling one bus transaction. This leaves 8 available bits for offset and length. In our experiments, we try several variations in an attempt to find the optimal combination.

Since the base must be 8 bits for data Run-Length codewords, there are several codeword size options. In fact, the data can take advantage of all three combinations, 12, 16, or 24 bits. In our experiments, we try several variations of assigning the remaining bits to the offset and length within each codeword size.

Lempel-Ziv Bus Formats

In Lempel-Ziv the lastSymbol size is fixed for the same reason the base size was fixed in Run-Length. This represents a piece of data, and therefore must be 8 bits. The pointer must represent the size of the data buffer in the decompression hardware. In addition, the length should be at least as big as the buffer, so that it can utilize the entire buffer. Due to the variation we added, the length can actually be longer than the buffer.

However, the question still arises as to what size to make the buffer. Since the lastSymbol is 8 bits, the Lempel-Ziv codeword can take all three codeword sizes. These codeword sizes, in addition to the pointer and length variations (with the length greater than or equal to the pointer), will be explored in the experiment section.

Codeword Buffering or Dynamic Wait State Generation

Generally, internal accesses can occur at a much faster rate than external accesses. This is due to the slow off-chip I/O in which external accesses must traverse. However, even with this faster rate it is likely that the decompression hardware will not completely service a codeword before another codeword arrives. This leaves two options: either buffer the codewords, or provide an external wait state signal to notify external devices that the decompression hardware is not ready to receive an additional codeword.

If the choice is made to buffer the codewords, the question becomes how big to make the buffer. Given a reasonable size buffer it will always be possible to overflow the buffer with codewords that offer tremendous compression. Therefore, there must exist a method to handle overflow. Since the software performing the compression is aware of the hardware's configuration, it can insert the equivalent of "nop" writes until it knows that the hardware will be able to handle an additional codeword.

If the choice is made to provide an external wait state signal, overflow is no longer a concern. However, this will provide further restrictions on what external hardware is needed to program the device. There remains a need for a small buffer, such that the decompression hardware will have a steady supply of codewords, and never stall. This will allow for faster configuration.

Distinguishing Address and Data Transactions

The compression algorithms contained in this paper require two different types of communications: Address and Data. Since the same bus signals are used in both address and data transactions, there must be some mechanism to distinguish between these communications. While it is possible to include a single signal which specifies the transaction type, this will waste bandwidth. Instead, we can use the following rule: if fewer data values have been received (in encoded form) than address values, the next transaction is a data value. Otherwise, the next transaction is an address value. Since the number of values encoded in each transaction is easy to determine (since the length is explicitly encoded in the transaction), accumulators can record the number of each type of value received. A simple comparison between these accumulated values can thus determine the type of transaction received.

Hardware Support

Along with compression algorithms and communication protocols, the compression of configuration streams also requires that their be fast, efficient, and small decompression hardware built into the FPGA architecture itself. In this section we discuss the hardware constructs necessary to support each of the compression methods discussed in this paper.

Run-Length Hardware

The Run-Length hardware is shown in Figure 1. It consists of a register to hold the current address to output; a down counter to count the length; an adder, to add the offsets to the previous value; and a mux to choose between a previous valued added to the offset and the new base value.

The mux chooses the output of the base register when the down-counter equals zero. When a new code word arrives, the base value is written into the address register at the same time that the length is written into the down-counter. The down-counter then counts down until zero, while the address register captures its previous value plus the offset. This continues until the down-counter reaches zero.

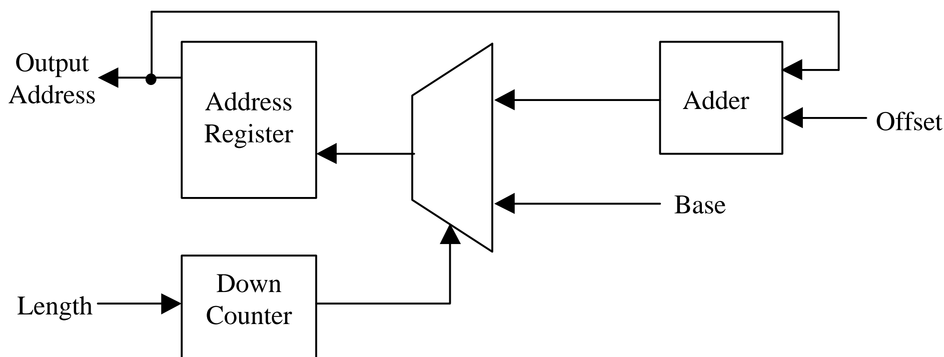

Figure 1: Run-Length hardware support.

Lempel-Ziv Hardware

The Lempel-Ziv hardware is shown in Figure 2. It consists of a set of registers that buffers the current window, one big mux whose select line is the pointer, and a down counter to count the length. If the down-counter is zero, the source of D_0 is the LastSymbol. Otherwise, the source of the down-counter is the output of the mux which selects among the registers in the buffer. The number of registers is up to the designer, and is shown as *n* in this diagram.

When a codeword is processed the length is initially written into the down-counter. As the down-counter counts down, the data registers shift the data to the right, bringing in new data from the register selected by the pointer. When the down-counter reaches zero, the lastSymbol is written into D_0.

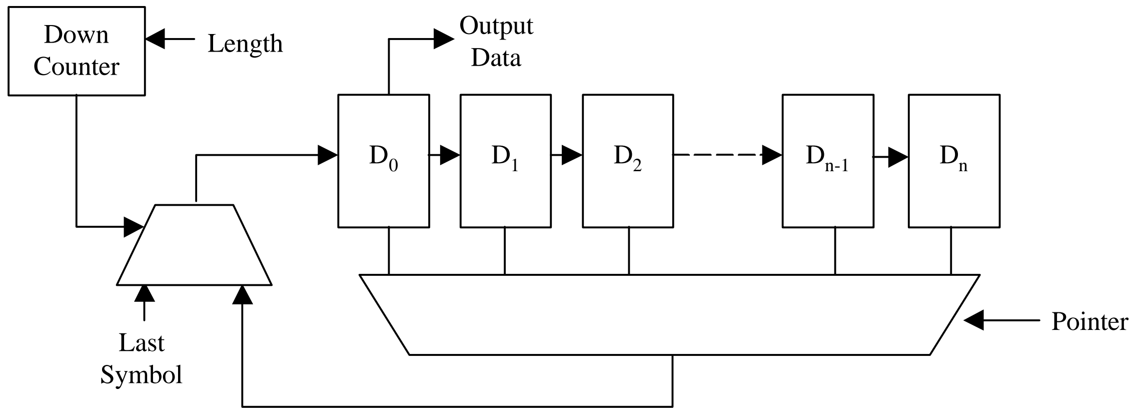

Figure 2: Lempel - Ziv hardware support.

Codeword Unpacking

The hardware that must unpack the codewords will largely depend on the codeword sizes chosen. The address codeword will always be 24 bits. However, if the data codeword is chosen to be 12 bits, the hardware will appear as in Figure 3.

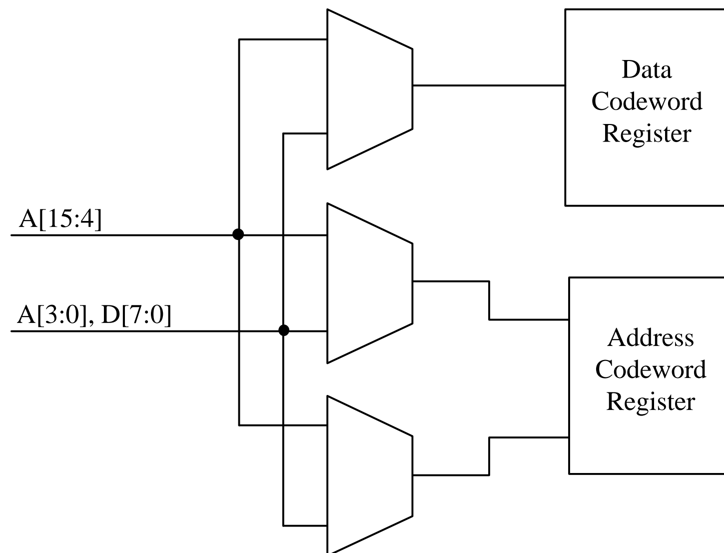

Figure 4: Unpacking hardware support.

Control Hardware will control the multiplexers, the write enables of the data codeword register, and the write enables of the upper and lower half of the address codeword register. Control hardware will monitor the lengths in the address and data decompression hardware, and will expect the next codeword to be provided for the decompression hardware with the shortest length.

Experiments

The algorithms described above were implemented in C++, and were run on a set of benchmarks collected from current XC6200 users. These benchmarks include files whose layouts and routing were optimized carefully at a low-level by Gordon Brebner and Virtual Computer Corp., as well as files whose routing was determined by the Xilinx XACT6000 software's routing tool.

The benchmarks were compressed using the three compression strategies, Basic Run-Length, Lempel-Ziv, and Reordering Run-Length. Within each type of strategy the address codeword parameter sizes, and the data codeword and codeword parameter sizes were varied to find the best parameter settings.

Parameter Setting Experiments

For each of the compression algorithms presented in this paper the overall set of address/data pairs are ordered (numerically by address for Lempel-Ziv and basic Runlength, by data value & some address information in Reordered Runlength), and then the address and data codewords are generated independently. These compression methods allow for the codeword and parameter sizing variations to be studied in isolation between the address and data values. Therefore, each compression technique will be isolated below to determine the optimal parameter and codeword sizing.

In the results tables that follow the two leftmost columns list the benchmark name and initial length, indicating the number of words needed to achieve that configuration with no compression. Along the top of the table are the parameter settings tested in a given run.

For Runlength the parameters include: "Length Bits", the number of bits used to represent the number of values represented in a single codeword; "Offset Bits", which is the number of bits used to represent the value change between successive items; "Codeword Length", which is the total number of bits required for the entire codeword (including 8 or 16 bits for the base data or address respectively).

For Lempel-Ziv compression the parameters include: "Pointer Bits", the number of bits used to index into the Lempel Ziv history buffer; "Length Bits", the number of bits used to hold the number of values copied from the buffer; "Codeword Length", the total number of bits required for the entire codeword (including 8 bits for the "next character" data value). Note that for Lempel - Ziv the "Pointer Bits" also dictates the length of the history buffer (N pointer bits requires 2^N registers in the history buffer). Thus, using a large number of pointer bits will significantly increase the hardware costs of this technique.

In the column beneath each parameter setting is the number of bus cycles required to transfer the benchmark's information within that compression algorithm and settings. For example, if a given parameter setting is using 16 bit codewords (2/3 of a bus transaction), and 9 codewords are needed to transfer a benchmark, then 6 bus transactions are reported. All totals are rounded up to the nearest whole transaction amount. The best value for a given benchmark amongst all parameter settings is highlighted in gray, as is the best overall parameter settings.

	Length Bits	7	6	5	4	3	2	1
	Offset Bits	1	2	3	4	5	6	7
	Codeword Length	24	24	24	24	24	24	24
CalFile	Length							
counter	198	71	56	56	57	63	67	84
parity	208	7	7	7	13	26	46	72
adder4	213	67	52	53	55	61	64	86
zero32	238	22	22	22	28	41	61	88
adder32	384	12	12	12	24	48	84	132
smear	695	198	143	135	150	169	205	270
adder4rm	907	440	342	331	328	342	339	390
gray	1200	480	392	371	355	370	424	502
top	1366	761	634	597	565	564	575	625
demo	2233	349	332	337	371	448	601	818
ccitt	2684	290	274	295	344	461	666	956
t	5819	1022	967	972	1065	1249	1608	2165
correlator	11001	3986	2734	2513	2548	2769	3371	4234
Sum	27146	7705	5967	5701	5903	6611	8111	10422

Table 1. Effects of parameter selection on the address transactions in Basic Runlength and Lemple-Ziv compression.

In these tables we have attempted to test most reasonable combinations of parameter settings. For the addresses, we always require a codeword of 24 bits (since the address is 16 bits, leaving no space for other compression information in 16 bit codewords), and thus the Length and Offset bits will share 8 bits. For data we can use 12, 16, or 24 bit codewords, and since the data item is only 8 bits, there are 4, 8, or 16 bits respectively available for other compression information.

In Table 1 we give the results of runlength compression on addresses when the files are reordered (within the reordering restrictions) to have the addresses numerically increasing. This is the address compression method used for both basic Runlength and Lempel - Ziv compression (remember that Lempel - Ziv only works on data values, since it requires exact duplication of values, which will not normally happen in the address streams). This reordering for runlength compression should give the highest address compression possible, though the compression of the data items may suffer. As can be seen, giving 5 bits to length and 3 bits to offset gives the best overall results, although (6,2) and (4,4) also perform well.

	Length Bits	8	7	6	5	4	3	2	1	3	2	1
	Offset Bits	8	1	2	3	4	5	6	7	1	2	3
	Codeword Length	24	16	16	16	16	16	16	16	12	12	12
CalFile	Length											
counter	198	50	43	42	42	42	46	52	60	37	42	50
parity	208	19	15	14	14	18	26	36	52	21	28	40
adder4	213	50	39	38	38	39	43	48	62	34	39	51
zero32	238	29	23	22	22	26	34	45	62	27	36	48
adder32	384	136	172	161	162	154	130	136	128	135	132	141
smear	695	272	234	226	224	221	224	228	239	183	190	205
adder4rm	907	379	344	339	332	327	320	315	316	264	273	292
gray	1200	578	512	498	489	478	461	454	452	388	391	405
top	1366	714	648	638	632	609	593	556	536	489	490	504
demo	2233	345	278	272	276	302	341	443	591	277	353	473
ccitt	2684	330	268	263	282	314	362	492	684	297	398	550
t	5819	994	792	745	764	808	935	1180	1553	769	938	1231
correlator	11001	5889	5506	5399	5188	5019	4743	4693	4556	4162	4150	4098
Sum	27146	9785	8874	8657	8465	8357	8258	8678	9291	7083	7460	8088

Table 2. Effects of parameter selection on the data transactions in Basic Runlength compression.

Table 2 reports the data compression results for Basic Runlength. Similar to address compression, it can be seen that the best results occur when more bits are given to the length (dictating the maximum number of data items that can be combined together) than offset. Also, although giving the most number of bits to both Length and Offset will compress the most values together into a single codeword, this requires more bits per codeword, and can in fact hurt overall compression. In fact, using only 12 bits per codeword by assigning 3 bits to length and 1 to offset gives the best results overall.

Table 3 contains the results of Lempel - Ziv compression on the data values. Recall that because of the extensions we made to the algorithm, it is beneficial if the length of the duplication (length bits) is at least as long as the buffer length (pointer bits). Also, large sizes of pointer bits increases the history buffer size, radically increasing hardware complexity. As can be seen, having pointer bits and length bits both equal to 8 gives the best results, although pointer bits and length bits of 4 each is close in quality. However, the 256 registers necessary for the 8-bit pointers imposes a fairly high hardware penalty, both in buffer size as well as support hardware. For this reason, we believe the (4,4) case is a more realistic balance between hardware complexity and compression performance.

Table 4 presents the address compression results when the address/data pairs are reordered to maximize data compression. Although some efforts are taken within this algorithm to also improve address compression, it cannot achieve nearly the same quality of address compression as our previous algorithm. For example, the best results for this algorithm (where length bits and offset bits are both 4) gives an overall size of 7,463 bus transactions, where the previous approaches result in only 5,701 bus transactions for the addresses. However, as we will see, this

degradation in address compression is more than balanced by the improvement in data compression. One other important consideration is the variability of the optimum amongst parameter settings. For example, in only one case does the (4,4) setting give the best results for any specific benchmark, but for the overall suite it yields the best results.

	Pointer Bits	8	7	6	4	3	2	2	1
	Length Bits	8	9	10	4	5	6	2	3
	Codeword Length	24	24	24	16	16	16	12	12
CalFile	Length								
counter	198	51	54	55	42	40	42	40	35
parity	208	19	19	19	18	14	14	25	19
adder4	213	48	50	50	36	38	38	36	33
zero32	238	25	25	25	22	18	19	30	25
adder32	384	23	23	24	29	17	16	52	34
smear	695	168	176	185	136	153	166	142	152
adder4rm	907	269	284	297	233	244	268	219	222
gray	1200	362	402	440	353	380	415	325	342
top	1366	438	471	502	408	469	530	408	435
demo	2233	166	188	243	224	210	231	296	242
ccitt	2684	153	219	234	222	200	212	326	256
t	5819	353	492	731	602	598	602	762	718
correlator	11001	1804	1945	2069	1761	2347	4380	3406	3895
Sum	27146	3879	4348	4874	4086	4728	6933	6067	6408

Table 3. Effects of parameter selection on the data transactions in Lempel - Ziv compression.

	Length Bits	7	6	5	4	3	2	1
	Offset Bits	1	2	3	4	5	6	7
	Codeword Length	24	24	24	24	24	24	24
CalFile	Length							
counter	198	72	57	57	58	64	71	88
parity	208	8	8	8	14	27	46	72
adder4	213	71	56	57	59	65	67	87
zero32	238	22	22	22	28	41	61	88
adder32	384	26	26	26	29	55	87	135
smear	695	354	261	214	210	222	243	283
adder4rm	907	567	436	422	419	422	417	456
gray	1200	767	648	631	592	588	587	616
top	1366	1034	895	845	810	787	772	789
demo	2233	522	494	498	527	588	694	902
ccitt	2684	457	430	441	484	597	751	1044
t	5819	1799	1541	1534	1568	1654	1840	2298
correlator	11001	9620	7775	3849	2665	2621	3368	4480
Sum	27146	15319	12649	8604	7463	7731	9004	11338

Table 4. Effects of parameter selection on the address transactions in Reordering Runlength compression.

Table 5 shows the compression results for the data values within reordering runlength compression. As expected, the data values compress much better in this algorithm than the other approaches, since the data items have been perfectly aligned for runlength encoding (barring those alignments not allowed by the reordering restrictions). Note

that since identical data values are aligned together, all of the runlength transactions should represent multiple instances of a single data value. Thus, there is no need for an offset within these runlengths, and thus 0 bits are used for offset. In this approach using 8 bits for the length provides the best overall results. However, note again that the optimum for any given run is just as likely to use 4 bits of length as 8, though the overall optimum is 8 bits.

	Length Bits	16	8	4
	Offset Bits	0	0	0
	Codeword Length	24	16	12
CalFile	Length			
counter	198	46	31	25
parity	208	20	14	14
adder4	213	46	31	25
zero32	238	32	22	20
adder32	384	14	10	13
smear	695	102	68	61
adder4rm	907	105	70	67
gray	1200	119	80	79
top	1366	188	126	118
demo	2233	46	34	83
ccitt	2684	45	35	95
t	5819	65	55	193
correlator	11001	198	144	389
Sum	27146	1026	720	1182

Table 5. Effects of parameter selection on the data transactions in Reordering Runlength compression.

Overall Algorithm Comparisons

From the data given in the previous section, we can now construct the best parameter settings for Basic Runlength, Lempel - Ziv, and Reordering Runlength algorithms. These results are given in the left portion of Table 6, which lists the number of bus transactions required for address and data compression. Also listed is the overall compression ratio, which is the ratio of original file size to compressed file size. Given these results it is clear that Reordering Runlength is the preferred approach, since it achieves better compression results than the other approaches while requiring much simpler hardware decompressors than Lempel - Ziv. Note also that the hardware that supports Reordering Runlength is identical to Basic Runlength, it is just the compression algorithm that differs. This is useful, since the Basic Runlength compression algorithm is extremely simple, and could even be executed on-line to support dynamic circuit creation and other advanced techniques, while Reordering Runlength would be available for higher quality when the compression can be performed off-line.

The data for Reordering Runlength also suggests a modification to the basic algorithm. Recall that for both address and data compression in Reordering Runlength, the best compression results for a given benchmark often used different parameter settings than the best overall compression approach for the entire suite. It is actually quite easy to take advantage of this feature. It would be simple to provide runlength hardware which allows the user to determine on a file by file basis what the best allocation of bus signals to length and offset values. When a configuration was being sent to the chip, it would first set the bus format when it is turning on the compression features of the chip. This is similar to the setting of bus width and other masking data which must be done for the Xilinx XC6200 series currently. Then, each file can be decompressed with the optimal parameter settings, resulting in potentially higher compression results. This algorithm is shown in the "Adaptive Reorder" section of Table 6.

For comparison with the algorithms proposed here, we present the results of our previous Wildcard-based compression algorithm [Hauck98b]. This algorithm makes use of the decompression hardware already built into the XC6200 series architecture, and can achieve good compression results. These numbers are given in the rightmost column of Table 6. As can be seen, the Reordering Runlength approach beats the Wildcard algorithm by about 2%, while the Adaptive Runlength achieves about 9% better results. Just as important, the Runlength hardware also

supports an extremely fast, on-line compression algorithm (Basic Runlength), which still achieves good compression results. This on-line option may be very useful for techniques in partial run-time reconfiguration and dynamic circuit creation, where the time to create the circuit may be a critical concern. In the Wildcard approach the complexity of efficiently using the Wildcard hardware makes it unlikely that an efficient on-line compression algorithm could be produced. By having a single hardware structure which can provide better compression results than the existing algorithm, as well as an option for on-line compression, a more flexible compression approach can be achieved.

		Basic Runlength			Lempel - Ziv			Reorder Runlength			Adaptive Reorder			Wild
CalFile	Length	Addr	Data	Comp	Addr	Data	Comp	Addr	Data	Comp	Addr	Data	Comp	Comp
counter	198	56	37	2.13	56	42	2.02	58	31	2.22	57	25	2.41	1.88
parity	208	7	21	7.43	7	18	8.32	14	14	7.43	8	14	9.45	7.43
adder4	213	53	34	2.45	53	36	2.39	59	31	2.37	56	25	2.63	2.21
zero32	238	22	27	4.86	22	22	5.41	28	22	4.76	22	20	5.67	4.18
adder32	384	12	135	2.61	12	29	9.37	29	10	9.85	26	10	10.67	5.26
smear	695	135	183	2.19	135	136	2.56	210	68	2.50	210	61	2.56	2.28
adder4rm	907	331	264	1.52	331	233	1.61	419	70	1.85	417	67	1.87	1.61
gray	1200	371	388	1.58	371	353	1.66	592	80	1.79	587	79	1.80	1.85
top	1366	597	489	1.26	597	408	1.36	810	126	1.46	772	118	1.53	1.41
demo	2233	337	277	3.64	337	224	3.98	527	34	3.98	494	34	4.23	4.1
ccitt	2684	295	297	4.53	295	222	5.19	484	35	5.17	430	35	5.77	5.82
t	5819	972	769	3.34	972	602	3.70	1568	55	3.59	1534	55	3.66	5.51
correlator	11001	2513	4162	1.65	2513	1761	2.57	2665	144	3.92	2621	144	3.98	5.86
Sum	27146	5701	7083		5701	4086		7463	720		7234	687		
Geometric Mean				2.64			3.20			3.34			3.60	3.28

Table 6. Overall comparison of complete compression algorithms.

Conclusions

In this paper we have presented algorithms, communication methodologies, and hardware support for accelerating reconfiguration via compressing datastreams. The algorithms include techniques for harnessing runlength encoding and Lempel - Ziv approaches to the unique features of FPGA configurations. The bus formats and parameter settings provide efficient communications of these items, allowing for relatively simple hardware, embedded in the FPGA architecture, to perform the decompression. This results in a compression algorithm superior to the previously existing Wildcard approach. Our Adaptive Runlength algorithm provides significant compression results, reducing configuration size (and thus bandwidth requirements) by a factor of 3.60. Faster on-line algorithms can also use this hardware to achieve a compression ratio of 2.64. Such an on-line algorithm can be used for dynamic circuit creation and other situations where configuration compile time is a significant concern, including many applications of reconfigurable computing. Combined, this provides a complete and efficient compression suite for FPGA configuration management.

Acknowledgments

Thanks to Gordon Brebner for providing CAL files for use as benchmarks. This research was funded in part by DARPA contract DABT63-97-C-0035 and NSF grants CDA-9703228 and MIP-9616572.

References

[Hauck98a] S. Hauck, "The Roles of FPGAs in Reprogrammable Systems", *Proceedings of the IEEE*, Vol. 86, No. 4, pp. 615-639, April 1998.

[Hauck98b] S. Hauck, Z. Li, E. J. Schwabe, "Configuration Compression for the Xilinx XC6200 FPGA", *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 138-146, 1998.

[Li99] Z. Li, S. Hauck, "Don't Care Discovery for FPGA Configuration Compression", to appear in *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, 1999.

[Ranganathan93] N. Ranganathan, S. Henriques, “High-Speed VLSI Designs for Lempel-Ziv-Based Data Compression.” *Trans. on Circuits & Systems- Analog and Digital DSP*, Vol. 40, No. 2, Feb. 1993

[Wirthlin95] M. J. Wirthlin, B. L. Hutchings, “A Dynamic Instruction Set Computer”, *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 99-107, 1995.

[Wirthlin96] M. J. Wirthlin, B. L. Hutchings, “Sequencing Run-Time Reconfigured Hardware with Software”, *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 122-128, 1996.

[Xilinx97] Xilinx, Inc., “XC6200 Field Programmable Gate Arrays Product Description”, April 1997.