Don’t Care Discovery for FPGA Configuration Compression

Zhiyuan Li
Motorola Labs
1303 E. Algonquin Rd.,
Schaumburg, IL 60196 USA
azl086@motorola.com

Scott Hauck
Department of Electrical Engineering,
University of Washington,
Seattle, WA 98195-352500
hauck@ee.washington.edu

Abstract

One of the major overheads in reconfigurable computing is the time it takes to reconfigure the devices in the system. The configuration compression algorithm presented in our previous work [1] is one efficient technique for reducing this overhead. In this paper, we develop an algorithm for finding Don’t Care bits in configurations to improve the compatibility of the configuration data. With the help of the Don’t Cares, higher configuration compression ratios can be achieved by using our modified configuration compression algorithm. This achieves a compression ratio of a factor of 7, where our previous algorithm only achieved a factor of 4.

1. Configuration Compression

FPGAs are often used as powerful hardware for applications that require high speed computation. One major benefit provided by FPGAs is the ability to reconfigure during execution. For systems in which reconfiguration was done infrequently, the time to reconfigure the FPGA was of little concern. However, as more and more applications involve run-time reconfiguration, fast reconfiguration of FPGAs becomes an important issue [2]. In most systems an FPGA must sit idle while it is being reconfigured, wasting cycles that could otherwise be used to perform useful work. For example, applications on the DISC and DISC II system spend 25% [3] to 71% [4] of their execution time performing reconfiguration. Thus, a reduction in the amount of cycles wasted to reconfiguration can significantly improve performance. Previously, we have presented a technique for reducing the overhead by compressing the configuration datastreams [1]. In this paper, we will present a technique for finding possible Don’t Cares in configuration data such that higher compression ratios can be achieved.

2. Don’t Cares Overview

There are two types of Don’t Cares for each configuration: *True Don’t Cares* and *Partial Don’t Cares*. In practice, most applications do not utilize all function units or routing resources. Therefore, unused function units and routing resources will not affect computation’s results, and the corresponding configuration bits for these locations can be considered as True Don’t Cares. For other locations, not all configuration bits are important to computation, and some bits can be turned into Don’t Cares without causing incorrectness. We call these bits Partial Don’t Cares, since only part of a data word is a Don’t Care. By exploring Don’t Cares and then assigning each Don’t Care bit a certain value, we can increase regularities within each application.

As an example, each cell of a Xilinx 6200 FPGA can route signals in four different directions. However, in most cases, only one or two directions are actually used for the computation, so the configuration for unused directions can be treated as Partial Don’t Cares. Although none of these locations is a True Don’t Care, regularities for different locations may increase by turning used bits into Don’t Cares, and thus fewer writes to cover the necessary configuration can be found. Suppose there are only two locations specified in a configuration, with address 1 containing data “00101000” and address 2 containing data “00100000”. Obviously, two separate configuration writes are required. However, assume that we can modify the value in address 1 to “0010-000”, where “-” means Don’t Care. Without considering the overhead of the Wildcard Register write, one write is now sufficient to complete the configuration of both locations.

3. The Backtracing Algorithm

Obviously, exploring Don’t Cares requires detailed information about configuration bit-streams. Additionally, an algorithm that uses this information to discover all configuration bits that sufficiently contribute to correct computation must be developed. Furthermore, all output locations must be specified: from the user’s point of view, these locations contain the information that the user really needs.

The outputs of these locations are computed by logic operations on the inputs to them, meaning that the locations providing these inputs could affect the results of the outputs. This identifies fields within newly specified locations that are critical to the computation results. Our algorithm backtraces the inputs to these fields and gets another set of important fields. This backtracing process is repeated until all important fields for the computation are traversed. Notice that these traversed fields normally represent a subset of the given configuration. This is because some configuration bits specified in the configuration file become Don't Cares, meaning that we can assign arbitrary values to them.

Given a configuration file, our goal is to discover the Don't Cares. Once this stage is complete, with minor modifications, existing configuration compression algorithm can be applied to find a compressed version of a configuration. The algorithm starts from circuit output, backtracing all configuration bits that contribute to the correct computation. This determines all programming bits necessary for correct computation, meaning that all other bits don't matter and can thus be considered as Don't Cares.

During backtracing we seek all portions of a circuit that help produce a given signal. Once these regions are found for each circuit output, we have identified all locations that must be configured with a specified value. Thus, all other locations can be treated as Don't Cares. For example, in Figure 1, the only output of the circuit is "O". We backtrace this signal, discovering that it is computed by a register. This means that its clock circuitry and its input "A" are important. Backtracing A will show that the function block of this cell is important, requiring B and C to be backtraced. Eventually, we will reach the registers that start this computation. Using this recursive backtracing process, we will identify the entire circuity shown. For this example all other configuration data is irrelevant to proper circuit function, and can be considered as Don't Care. Thus, all Northward and Westward routings, the logic blocks of cells 1 and 2, and the register in cell 3 can be configured arbitrarily. It is this flexibility that helps boosts compression ratios significantly.

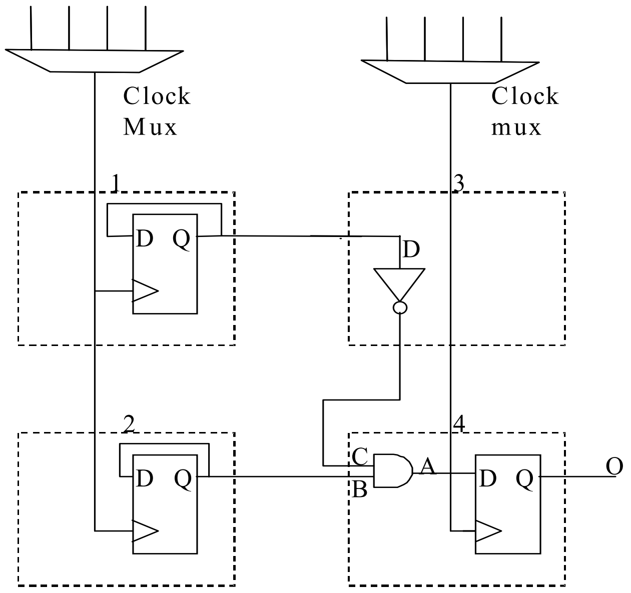

Figure 1: Sample circuit for backtracing.

One major concern for any lossy approach is whether the correctness of computation can be held for circuitry generated by decompressed data. Converting Don't Cares into certain values, the given configuration can be changed to a different configuration, since new values can be assigned to the newly discovered Don't Care bits. However, the resulting computation of the two configurations will be identical. From the user's point of view, if the outputs of both configurations produce the same result, we can safely say that both configurations meet the user's needs. Since the backtracing starting from the outputs for a given configuration covers all important fields necessary to the outputs, the computation correctness is maintained.

One final concern is that the new configuration will overwrite locations that may be used by other configurations. Since the locations traversed during backtracing contain information for the correct computation, those locations must be specified by the original configuration or by initialization (Reset) values. In either case, if the given configuration does not overwrite any locations that are used by other computations, the new configuration also will not, since it is a subset of the given one.

Two types of base devices (FPGAs) are considered in this paper: the Xilinx 6200 series FPGAs and the Xilinx Virtex FPGAs. A first generation partial run-time reconfigurable FPGA, the Xilinx 6200 series provides a special hardware, called wildcard registers, that allows multiple locations to be configured simultaneously. The wildcard compression algorithm [1] we developed not only efficiently compresses configuration bit-streams for all members of the Xilinx 6200 family, but also works for devices with similar features. The Xilinx Virtex FPGAs are one of the most widely used reconfigurable devices, with millions of gates. The architecture of the Virtex family points us in interesting directions for future reconfigurable devices. Compression approaches for Virtex FPGA [4, 9] can significantly reduce the size of configuration bit-streams for these devices.

Since Xilinx does not disclose the information necessary for discovering the Don't Cares in Virtex applications, we will focus mainly on algorithm design for the Xilinx 6200 architecture. However, we will still demonstrate the potential impact of Don't Cares for Virtex compression.

4. Don't Care Discovery for the Xilinx 6200 FPGAs

In order to do the backtracing, we need information about output locations. One set of our benchmarks is compiled by XACT6000 tools, which produce a symbol table file (.sym file) that specifies the locations of all circuit inputs and outputs. For another set of benchmarks that is not created by XACT6000 tools, we had created the symbol files that consist of output information. Before we discuss our Don't Care discovery technique, we first present our compression algorithm.

4.1. Xilinx XC6200 Field Programmable Gate Arrays

The XC6200 provides several types of programming control registers, including the Wildcard Registers, which allow some cell configuration memories within the same row or column of cells to be written simultaneously. There are two Wildcard Registers, the Row Wildcard Register and the Column Wildcard Register, which are associated with the row address decoder and the column address decoder, respectively. Each register has one bit for each bit in the row address or the column address. The Wildcard Registers can be viewed as "masks" for the row and column address decoder. The Wildcard Registers and the address decoder can be viewed as a configuration decompressor. Given a compressed configuration file, which has Wildcard Register writes followed by address writes, the address is decompressed so that several cells with the same function get configured simultaneously. With this hardware available, there is the potential to achieve significant reductions in the required configuration bandwidth. The key is to find an algorithm that can efficiently use this decompression hardware. An overview of our previous compression algorithm [1] is presented below.

4.2. Configuration Compression Algorithm

Our configuration compression algorithm contains two stages. In the first stage of the algorithm we seek to find the minimum number of writes necessary to configure the array for a given configuration. This will create a series of writes with arbitrary wildcards, meaning that these wildcard writes may add a significant overhead. This is because a single wildcarded write may require two writes to the wildcard registers and then one write to the configuration memory. The second stage of the algorithm attempts to reduce this wildcarding overhead by sharing the same wildcard in a series of writes, thus reducing the number of times the wildcard registers must be changed. In our previous paper, we have proved that the configuration compression problem is NP-complete. This problem is quite similar to another NP-complete problem—2-level logic minimization. The intuition behind this similarity is that if we can find the minimum number of cubes that cover the required set of minterms for a logic minimization problem, then we can find the minimum number of wildcards that covers the FPGA locations that correspond to those minterms. For the example in Figure 2, normal configuration will need 4 writes to configure all cells with the function "2". However, by using logic minimization techniques we can find a single cube that covers the corresponding minterms. We then can compress the 4 configuration memory addresses in the cube into one address “- -10”, where “-“ means wildcard. Instead of configuring the cells with 4 writes, 2 writes are sufficient (one for the wildcard register write, one for the address write).

	00	01	10	11
00	1	1	2	5
01	1	1	2	5
10	1	3	2	3
11	3	3	2	5

Figure 2: Example for demonstrating the potential for configuration compression

Since the XC6200 FPGA is a reconfigurable device, later writes can overwrite the previous value for a location. Thus, by considering the values of the cells that have not yet been written into the FPGA as Don’t Cares, we may be able to find a smaller number of cubes to cover the cells which need to be written to the FPGA, reducing the number of writes in the configuration. For the example in Figure 2, suppose value “1” is written before value “3”. By considering the cells with value “3” as Don’t Cares, we find a single cube “0---” to cover all the “1”s, instead of 2 separate writes.

In the first stage of the configuration compression algorithm, the logic minimization tool Espresso [6] is used. The basic steps of the first stage algorithm are:

1. Read the input configuration file and group together all configuration memory addresses with the same value. Mark all address locations as unoccupied.
2. Sort the groups in decreasing order of the number of addresses to be written in that group.
3. Pick the first group, and write the addresses in the group to the Espresso input file as part of the On set.
4. Write all other addresses marked unoccupied to the Espresso input file as part of the Don’t Care set.
5. Write all addresses marked occupied, yet with the same value as the first group, to the Espresso input file as part of the Don’t Care set.
6. Run Espresso.
7. Pick the cube from the Espresso output that covers the most unoccupied addresses in the first group, and add the cube to the compressed configuration file. Mark all covered addresses as occupied, and remove them from the group.
8. If the cube did not cover all of the addresses in the group, reinsert the group into the sorted list.
9. If any addresses remain to be compressed, go to step 2.

Once this stage of the algorithm is complete, a series of writes is created. Since wildcards are contained in most of the addresses, before writing an address the Wildcard Registers must be set. The Wildcard writes represent a significant overhead. In stage two of the algorithm, we reduce this overhead by reordering writes, creating Wildcard Register sharing between multiple configuration memory writes. In order to reduce the overhead, we reorder the sequence of writes found in stage one such that the address writes that have potential wildcard sharing are placed next to each other. In order to do this, we convert the totally ordered sequence of writes from the first stage into a partial order that captures only those ordering constraints necessary to maintain correctness. We have rules for creating the partial order graph (for details, refer to [1]). Each node represents an address write, and an edge from node A to node B means that B must be scheduled later than A. Only those nodes without any incoming edges can be scheduled first. After a node is scheduled, that node and any edges connected to it are removed, potentially allowing other nodes to be scheduled. The heuristic that remove almost all wildcard overhead are presented in [1].

4.3. Don’t Care Discovery

A key technique for Don’t Care discovery is to backtrace all components that produce outputs. There are three major components in the array: cells, switches and IOBs. There are 4096 cells arranged in a 64 × 64 array, and each cell has 3 separate 8-bit configuration bytes. One of these bytes controls the neighbor routing multiplexers, and two others control the functionality. Switches are located at the boundary of blocks of 4 × 4 cells, and they are labeled according to the signal travel direction. Each of the east and west switches has one configuration byte controlling neighbor routing, length 4 wire routing, and length 16 wire routing. Each north and south switch has multiple configuration bytes that controls: neighbor routing, length 4 and length 16 routing, and global signals including clock and clear lines. Each IOB consists of multiple configuration bytes controlling routing and some circuit control signals. A configuration can be viewed as the configurations of the multiplexers in cells, switches, and IOBs. If any multiplexer in a specified unit (cells, switches and IOBs) is not used for the computation, then the corresponding configuration bits for that multiplexer are considered Don’t Cares. We now present details on how to find Don’t Cares for cells, switches and IOBs.

Figure 3 shows the basic XC6200 cell in detail, with the function unit at left and cell routings at right. Input multiplexers select outputs from neighbors or from length 4 wires to connect to X1, X2, and X3. The Y2 and Y3 multiplexers provide for conditional inversion of the inputs. The CS multiplexer selects a combinatorial or sequential output. The RP multiplexer controls the contents of the register to be “protected”. If the register is configured as “protected”, then only the processor interface can write it.

Two configuration bytes control the multiplexers for the function unit. Don’t Care discovery depends on the functionality of the cell. For example, if the CS multiplexer selects the sequential output and the RP multiplexer configures the register as protected (feeds the register output back into its input), then all X and Y multiplexers can be set as Don’t Cares because the user interface is the only source that can change the F output. If either the Y2 or Y3 multiplexer selects the output of the register, then the corresponding X multiplex can be set to Don’t Care. The X1 multiplexer can be set to Don’t Care if Y2 and Y3 both select the same signal. For any of the four neighbor routing multiplexers not used for computation or routing, the bits for controlling the multiplexer can be considered Don’t Cares.

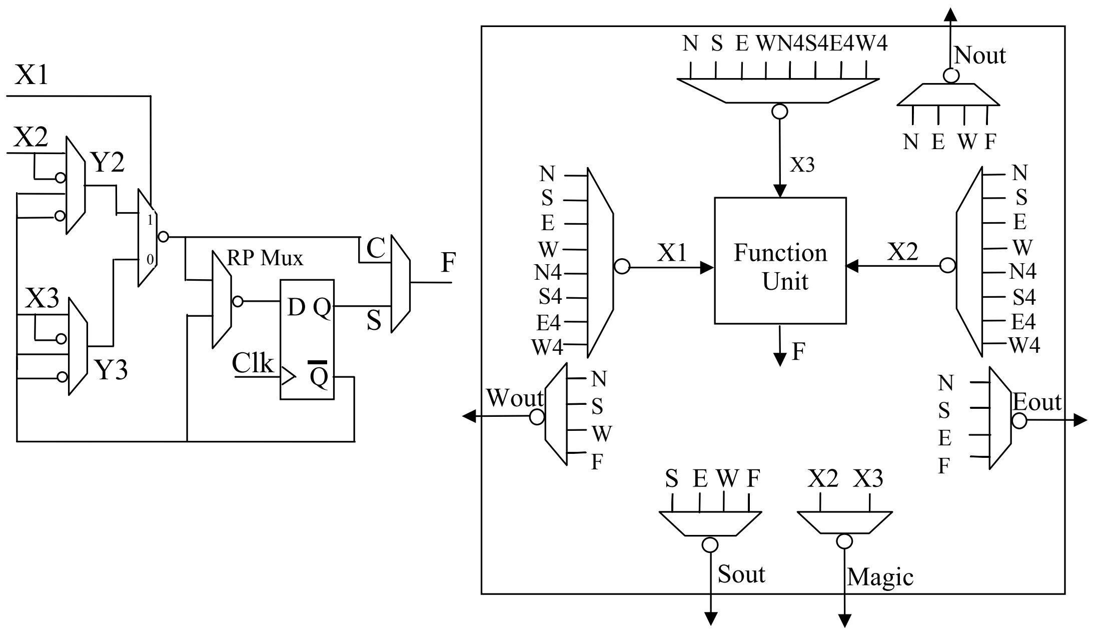

Figure 3: Xilinx 6200 function unit and cell routing.

Figure 4 shows the North switch at 4 × 4 block boundaries. Two multiplexers control neighbor routing and length 4 routing to the North, and there is an additional length 16 multiplexer at each 16 × 16 boundary. South, East and West switches have structures that are similar to the North switches.

Generally, if any of the multiplexers is not used, then the configuration bits for that multiplexer can be set to Don't Cares. However, the configuration bits for the Nout multiplexer (Figure 4 center) cannot be set to Don't Cares if the N4out multiplexer selects NCout, since the same programming bits control the upper and lower four input multiplexers. If NCout and Nout select different inputs, both inputs must be backtraced.

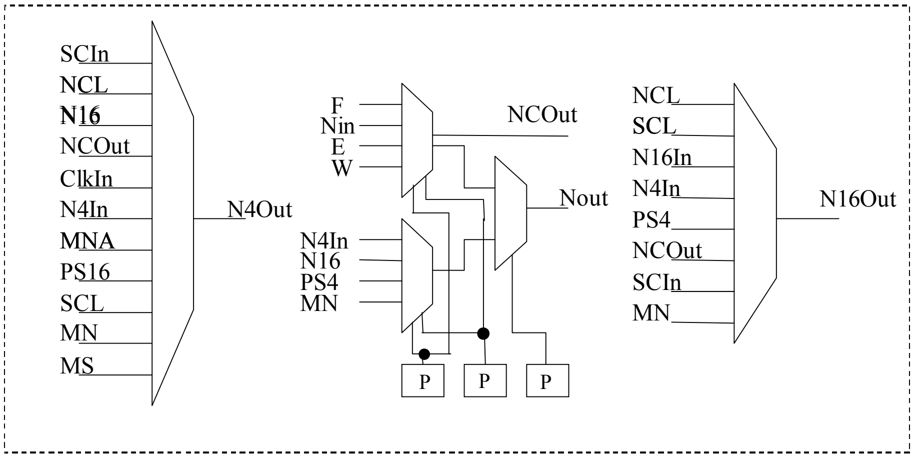

Figure 4: The Xilinx 6200 North switch at 4 × 4 block boundaries.

Each North switch contains an additional Clock multiplexer. This multiplexer is traversed only if a cell in the same column within the 4 × 4 block is configured as a register. Each South switch at the 16 × 16 boundary contains a Clear multiplexer. This multiplexer is traversed only if any cell at the same column within the 16 × 16 block is configured as a register.

Our algorithm does not attempt to find Don't Cares in IOBs for two reasons. First, there are only 64 IOBs at each side of the array, meaning that we will not benefit significantly from finding Don't Cares. Second, the architecture of IOB involves many circuit-control signals that cannot be turned to Don't Cares. However our algorithm does traverse identified IOBs to backtrace other units. Thus, our algorithm is conservative, since it may not discover Don't Cares in IOBs, but will always produce valid output.

We now present the basic steps of our Don't Care discovery algorithm..

1. Read the input .cal file and mark a unit as "touched" if any part of it is specified in the .cal file. Mark all configuration bits as Don't Cares.

2. Read the .sym file and put all output units (IOBs and registers used as outputs) into a queue.

3. Remove a unit from the queue. If it has already been backtraced, ignore it. Otherwise, mark its configuration bits as "no longer Don't Care", and insert its important inputs into the queue. Mark the unit as "touched".

4. If the queue is not empty, goto Step 3.

5. Produce a new target configuration where:

   5.1. All locations that were not marked as touched are considered as Don't Touch.

   5.2. All bits that were marked as "no longer Don't Care" are assigned their values from the .cal file.

   5.3. All other bits are Don't Cares.

A location that is marked “Don’t touch” is one that isn’t configured in the source configuration file, and thus may have useful information due to partial reconfiguration. It therefore should generally be left unchanged.

4.4. The Modification of the Configuration Compression Algorithm

Once the Don’t Care discovery algorithm is complete, we have a list of address data pairs, with Don’t Care bits contained in many of the data values. In order to take advantage of these Don’t Cares we need to make some modifications to our configuration compression algorithm.

In our original configuration compression algorithm, locations with the same data value are placed in the same group. This is because the addresses with the same value represent an On set in the corresponding logic minimization problem. However, by discovering the Don’t Care bits, each On set can be represented by a set of locations that not necessarily consist of the same value. After modifying the Don’t Cares to “1” or “0”, the locations with different values in the given configuration can be placed into the same group since these locations are compatible. Note that it is possible that an address can now fit into multiple groups instead of fitting just one group in our original compression algorithm because of the Don’t Cares. Therefore the flexibility for our configuration compression algorithm is increased. For example, suppose that after the discovery of the Don’t Care bits address A contains data “00-000-0”. Assume there are 3 groups, where group 1 has value “00000000”, group 2 has value “00000010” and group 3 has value “00100000”. Address A is compatible with the value of each of the 3 groups and is placed into all 3 groups. Writing any value representing the 3 groups into address A properly configures it. This is because any of the 3 values can create the necessary configuration for the computation. Even though address A may be overwritten by values from the other two groups, the necessary configuration for computation for that location is maintained. Our original algorithm can take advantage of this feature to find fewer cubes covering the necessary configuration.

In our original configuration compression algorithm the data associated with an address has a fixed value, so the locations are grouped by their values. However, after running the Don’t Care discovery algorithm, a location with Don’t Cares can be placed into multiple groups dependent on their compatibility. Thus we need to develop an algorithm to group the locations such that the addresses (locations) in each group are compatible. An address (location) can appear in as many as 2^n groups, where n is the number of Don’t Care bits contained in its data value. Notice that compatibility is not transitive. That is, if A and B are compatible, and B and C are compatible, it is not always true that A and C are compatible. For example, assume A, B and C have values “000100-0”, “0-0-0000” and “0100-000” respectively. A and B are compatible, and B and C are compatible, but A and C are not compatible. This non-transitivity property is an important consideration, making grouping decisions complex.

For 8-bit data, the simplest method for grouping is to create 256 groups, with the values 0 to 255. For each address data pair, place it into every group with a compatible value. However, this technique has exponential time complexity, and if we want to extend this technique to a 32-bit data bus the number of groups needed is 2^{32}. It is obvious that a heuristic method is needed. We present our heuristic grouping algorithm as following:

1. Once Don’t Care discovery is complete, put those addresses with Don’t Care data bits into a list. For those addresses without Don’t Care Data bits, group them according to their data values.
2. Search the list, removing those addresses that can be fit into any of the current groups, and put them into all compatible groups.
3. Repeat until the list is empty:

    3.1 Pick a location from the list with the fewest Don’t Care bits.

    3.2 The value for the group is equal to the value for the picked location, but with all Don’t Care bits converted to “0” or “1”. These bits are converted iteratively, converting to the value that has the most compatible other locations.

    3.3 Add all locations compatible to this value to the group. If they are on the unassigned list, remove them.

We also need to make modifications to other steps of the configuration compression algorithm as well. To make it clear, we present the modified algorithm:

1. Apply the Don't Care Discovery algorithm to find Don't Cares. Group the address data pair by using our grouping algorithm. Mark the address locations specified in given .cal file as unoccupied. Mark the address locations not specified in the .cal file, but used in the backtrace, as occupied.
2. Sort the groups in decreasing order of the number of addresses unoccupied in that group.
3. Pick the first group, and write the addresses in the group to the Espresso input file as part of the On set.
4. Write all other addresses marked unoccupied to the Espresso input file as part of the Don't Care set.
5. Write all addresses marked occupied, yet with a value compatible with the group, to the Espresso input file as part of the Don't Care set.
6. Run Espresso.
7. Pick the cube from the Espresso output that covers the most unoccupied addresses in the first group, and add the cube to the compressed configuration file. Mark all covered addresses as occupied.
8. If the cube did not cover all of the addresses in the group, reinsert the group into the sorted list.
9. If any addresses remain unoccupied, go to step 2.

This algorithm has several classes of locations: configured, initialized, and untouched. Configured locations are those whose value is set in the input .cal file, and our algorithm will generate a write to set these values. Untouched locations, which are not found in either the backtrace or the .cal file, can be viewed as either Don't Touch, if these unused cells may be used for other functions, or Don't Care, if the cells will be left unused. Initialized locations are locations that are not set in the .cal file, but are discovered to be important during backtracing. Thus the initialization value must be used. Our algorithm handles these locations as potential group members, but which are already set as occupied. Thus, compatible values can overwrite these locations to achieve better compression, but the algorithm is not required to write to these locations if it is not advantageous.

Experimental Results

The results are shown in Table 1. The size of the initial circuit is given in the "Input size" column. This size includes all writes required to configure the FPGA, including both compressible writes to the array, as well as non-compressible control register writes. The "Ctrl" column represents the number of non-compressible writes, and is a fixed overhead for both the original and compressed file. The results of the compressed version by our original algorithm are shown in the column "Original Compression". The results of the compressed version by our new algorithm are shown in the column "New algorithm", with unspecified locations considered as Don't Touch or Don't Care (which is appropriate depends on the details of the use of these configurations).

The number of writes to configure the logic array is shown in the column "Cnfg", the number of wildcard register writes is shown in "Wcrd", the "Ratio1" is the ratio of the total number of writes (the summation of "Ctrl", "Cnfg" and "Wcrd") to the size of the input configurations. Notice that the "Ctrl" writes represent a fixed startup cost that often can be ignored during Run-Time reconfiguration. Thus, to reflect the compression ratio without this initial startup cost, we use "Ratio2", which equals to ("Cnfg" + "Wcrd")/("Input size" – "Ctrl"), to represent the compression ratio for the compressible part of the circuits. In last two rows, the total number of writes and compression ratios of all benchmarks are calculated for two cases, with and without counting the "Ctrl" writes. As can be seen, the use of Don't Care discovery as pre-processing can improve the average compression factor from 4 to 7.

Bench-mark	Input size	Ctrl	Original compression algorithm				New algorithm (Don't Touch)				New algorithm (Don't Care)			
			Cnfg	Wcrd	Ratio1	Ratio2	Cnfg	Wcrd	Ratio1	Ratio2	Cnfg	Wcrd	Ratio1	Ratio2
Counter	199	40	53	13	53.2%	41.5%	29	5	37.2%	21.4%	22	4	33.2%	16.4%
parity	208	16	9	3	13.5%	6.3%	6	2	11.5%	4.2%	6	2	11.5%	4.2%
Add4	214	40	43	14	45.3%	32.7%	24	7	33.2%	17.8%	16	6	29.0%	12.6%
zero32	238	42	12	3	23.9%	7.7%	8	3	22.3%	5.6%	6	3	21.4%	4.5%
adder32	384	31	28	14	19.0%	11.9%	20	13	16.7%	9.3%	20	13	16.7%	9.3%
Smear	696	44	224	37	43.8%	40.0%	150	36	33.0%	28.5%	121	32	28.3%	23.5%
Add4rm	908	46	473	45	62.1%	60.1%	279	78	44.3%	41.4%	203	65	34.6%	31.1%
Gray	1201	44	530	74	53.9%	52.2%	378	53	39.5%	37.3%	311	44	33.2%	30.4%
Top	1367	70	812	87	70.8%	69.3%	531	65	48.7%	46.0%	419	57	39.9%	36.7%
demo	2233	31	423	91	24.4%	23.3%	281	77	17.4%	16.3%	241	66	15.1%	13.9%
ccitt	2684	31	346	84	17.2%	16.2%	235	55	12.0%	11.0%	204	50	10.6%	9.6%
t	5819	31	834	192	18.2%	17.7%	567	176	13.3%	12.8%	492	162	11.8%	11.3%
correlator	11011	38	1663	225	17.4%	17.2%	1159	187	12.6%	12.3%	1004	176	11.0%	10.8%
Totals:														
w/ctrl	27162			6836	(25.2%)			4928	(18.1%)			4249	(15.6%)	
w/o ctrl	26658			6332	(23.8%)			4424	(16.6%)			3745	(14.0%)	

Table 1: The results of the compression algorithms.

5. Virtex Compression with Don't Cares

Although Xilinx does not disclose the information necessary to discover Don't Cares in the Virtex applications, we can still evaluate the potential impact of the Don't Cares for Virtex compression. In order to make an estimate, we randomly turn some bits of the data stream into Don't Cares and bound the impact of Don't Cares on the Readback algorithm in [4].

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

Figure 5 demonstrates the potential effect of Don't Cares over the benchmarks listed in Table 3.2. The X-axis is the percentage of the Don't Cares we randomly create and the Y-axis is normalized over the results without considering Don't Cares. From XC6200 FPGA applications, we discovered that typically 30% of the bits in a configuration are Don't Cares. As can be seen in Figure 5, by using upper-bound approach a factor of 1.3 improvement can be achieved on applications containing 30% Don't Cares, while a factor of 2 improvement can be achieved using the lower-bound approach.

Conclusions

One of the primary problems in reconfigurable computing is the time overhead due to reconfiguration. Reducing this overhead is an important consideration for current systems. In our previous section, we presented a general-purpose compression algorithm for reconfigurable computing configurations by using the decompression hardware in the Xilinx XC6200 FPGA. In this section, we have presented a Don't Care discovery algorithm to increase the size of the Don't Care set. By combining this technique with our modified version of the compression algorithm, compressed file sizes are about 14% of the original file sizes. This represents a compression ratio of a factor of 7, where our original algorithm only achieved a factor of 4 compression on these benchmarks. In addition, we have evaluated the expected impact of Don't Cares on Virtex devices.

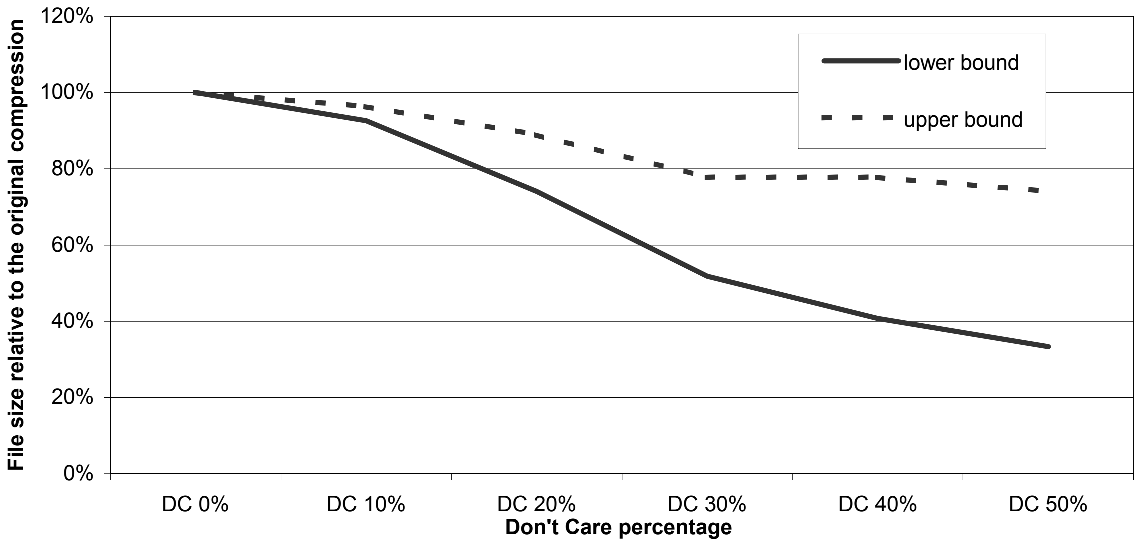

Figure 5: The effect of Don't Cares on benchmarks in Table 3.2 for Virtex compression.

References

[1] S. Hauck, Z. Li, E. Schwabe, "Configuration Compression for the Xilinx XC6200 FPGA", pp 123-130, *IEEE Symposium on FPGAs for Custom Computing Machines, 1998.*

[2] S. Hauck, "The Roles of FPGAs in Reprogrammable Systems", *Proceedings of the IEE*E, Vol. 86, No. 4, pp. 615-638, April, 1998.

[3] S. Hauck, "Configuration Prefetch for Single Context Reconfigurable Coprocessors", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65-74, 1998.

[4] Z. Li, S. Hauck, "Configuration Compression for Virtex FPGA", *IEEE Symposium on FPGAs for Custom Computing Machines,* pp. 113-124, 2001.

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

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

[7] R. K. Brayton, G. D. Hachtel, C. T. McMullen and A. L. Sangiovanni-Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis", *Kluwer Academic Publisher*s, 1984.

[8] Xilinx, Inc., "XC6200 Field Programmable Gate Array Product Description", April 1997.

[9] Andreas Dandalis, Viktor Prasanna, "Configuration Compression for FPGA-based Embedded Systems", *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp 172-181, 2001.