

Configuration Compression for the Xilinx XC6200 FPGA

Scott Hauck, Zhiyuan Li

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

Eric Schwabe

School of CTI
DePaul University
234 S. Wabash Ave. Chicago, IL 60604 USA
eschwabe@cti.depaul.edu

Abstract

One of the major overheads in reconfigurable computing is the time it takes to reconfigure the devices in the system. This overhead limits the speedups possible in this exciting new paradigm. In this paper we explore one technique for reducing this overhead: the compression of configuration datastreams. We develop an algorithm, targeted to the decompression hardware imbedded in the Xilinx XC6200 series FPGA architecture, that can radically reduce the amount of data needed to transfer during reconfiguration. This results in an overall reduction of about a factor of 4 in total bandwidth required for reconfiguration.

Configuration Compression

FPGAs are often used as powerful custom hardware for applications that require high-speed computation [Hauck98b]. One of the major benefits provided by FPGAs is the ability to reconfigure during execution. For past 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.

In most current systems the FPGA must sit idle while it is being reconfigured, wasting cycles that could otherwise be performing useful work. For example, applications on the DISC and DISC II system have spent 25% [Withlin96] to more than 70% [Wirthlin95, Villasenor96] of their execution time performing reconfiguration. Thus, it is obvious that a reduction in the number of cycles wasted to reconfiguration can generate significant improvements in performance. Previously, we have presented methods for overlapping reconfiguration with computation via configuration prefetching [Hauck98a]. In this paper, we consider another method that can reduce the reconfiguration overhead: the compression of configuration streams such that the total number of write operations to load a configuration can be reduced. Our method is based on the special decompression hardware provided by the Xilinx XC6200 FPGA, which can be adapted for use on other FPGA architectures.

Xilinx XC6200 Field Programmable Gate Arrays

The target for our compression work is the Xilinx XC6200 series FPGAs [Xilinx97]. This FPGA has special hardware called Wildcard Registers that can be viewed as decompressors for configuration data. However, to the best of our knowledge there is no compression algorithm that can efficiently use this hardware. In this paper we present an algorithm for configuration compression that targets this special hardware. We will first briefly describe the features of Xilinx XC6200 series FPGA.

The XC6200 FPGA is an SRAM based high-performance Sea-Of-Gates FPGA optimized for datapath designs. All user registers and SRAM control store memory are mapped into a host processor's address space, thus making it easy to configure and access the chip's state. The XC6200 can handle 8-bit, 16-bit and 32-bit data writes. A simplified block diagram of the XC6216 is shown in Figure 1.

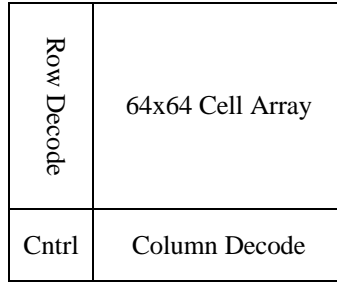


Figure 1. XC6216 simplified block diagram.

The XC6200 provides several types of programming control registers: (1) Device Configuration Register, which controls global device functions and modes. (2) Device Identification Register, which controls when the computation starts. Usually the ID Registers are written in the final step of configuration. (3) Map Register, which can map all the possible cell outputs from a column onto the external data bus. By correctly setting the map register, the state register can be easily accessed without complicated mask operations. (4) Mask Register, which can control which bits on the data bus are valid and which bits are ignored. (5) Wildcard Register, which allows some cell configuration memories within the same row or column of cells to be written simultaneously. Since the Wildcard Registers are the primary architectural feature used by our algorithm, we will give more details for the Wildcard Registers.

There are two Wildcard Registers, Row Wildcard Register and 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. Let us consider the effect of the Row Wildcard Register on row address translation (the Column Wildcard Register has the same effect on column address translation). A logic one bit in the Row Wildcard Register indicates that the corresponding bit of the row address is a wildcard, which means the address decoder matches rows whose addresses have either a “1” or a “0” on the wildcard bits. Thus, the number of cells that will be configured at the same time is 2^n if there are n logic one bits in the Wildcard Register. For example, suppose the Row Wildcard Register is set as “010001” and the address to the row address decoder is set as “110010”. In this case the row decoder selects rows 100010, 100011, 110010, and 110011. If these locations share the same computation, and thus would need to be configured with the same value, all four could be configured with a single write operation. Thus, by using Wildcard Registers faster configuration could be achieved.

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 such that several cells with the same function get configured simultaneously. The Wildcard Registers can inform the address decoder which bits in the address can be Wildcarded and which bits cannot. Theoretically, up to 4096 cells can be configured by only 3 writes (two Wildcard Registers writes and one address write) if we assume all 4096 cells share the same function. With this “decompressor” hardware available, there is the potential to achieve significant reductions in the required configuration bandwidth. The key is how to find an algorithm that can efficiently use this decompression hardware. In this paper we will present one such algorithm.

Configuration Compression Algorithm

Given a normal configuration file, our algorithm will generate a new configuration file that performs the same configuration with fewer writes by using the Wildcard Registers. Our algorithm contains two stages. In the first stage of the algorithm, we assume that writes to the wildcard registers are free, and thus 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. 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.

Before discussing the details for this algorithm, we first describe the format of the configuration file we use. The standard Xilinx XC6200 configuration file (.cal file) consists of a series of configuration address-data pairs. The address is 16 bits long, and the data elements are 8 to 32 bits (depending upon the mask register setting). Since most of the benchmark files we have obtained contain 8 bit data values, and since 8 bit writes can emulate 32 bit writes (but not vice-versa without the danger of overwriting other locations), we only consider 8-bit data here. Note that

this happens to also be one of the more difficult cases for a compression algorithm, since with 8-bit writes it is not possible to write both the row and column wildcard registers simultaneously, thus increasing the wildcard overhead. However, our algorithm can easily be simplified to handle 16 or 32 bit writes.

Two points must be made about the .cal files. First, a .cal file contains data to configure the entire chip, including both the logic array and the configuration registers. However, the Wildcard registers only operate on the logic array memory addresses, meaning that it is not possible to compress the configuration register writes. Thus, these register writes represent a fixed overhead for our algorithm. We will ignore these writes during the discussion that follows, although our algorithm does maintain all control register writes from the source file, and our results include these fixed overheads. Second, the XC6200 is partially reconfigurable, meaning that a .cal file may contain writes to only a portion of the logic array. Thus, there are regions of the array that are not modified by the input configuration. Since these locations may contain data from previous configurations that must be maintained, we treat all locations not written by an input .cal file as “Don’t Touches”. That is, we do not allow our algorithm to reconfigure these locations, thus restricting the amount of compression possible.

The First Stage of the Compression Algorithm

In the first stage of the algorithm, we assume that both wildcard registers can be written during the same cycle as data is written to the logic array’s configuration. Thus, we ignore the overhead of wildcard writes in order to simplify the compression problem. However, we will show that even this simplified version of the problem is NP-hard by transforming 2-level logic minimization into this compression problem. Although this will demonstrate that an optimal algorithm is unlikely for this problem, it will also point the way towards an efficient solution via standard logic minimization techniques.



Figure 2. Example of the transformation of 2-level logic minimization into the simplified configuration compression problem. The Karnaugh Map (left) of the circuit is transformed into the configuration to be compressed (right). “DT” indicates don’t touches in the configuration.

In the standard two-level logic minimization problem, the goal is to find the minimum number of cubes that cover the ON set of a function, while covering none of the OFF set. In the configuration compression problem we seek to find the fewest wildcard-augmented writes that will set the memory to the proper state.

We formally define the two problems in question as follows:

TWO-LEVEL-LOGIC: Given a Boolean function X , specified as an ON-set and an OFF-set with a total of n terms, and a value j , is there a set of at most j cubes that covers precisely those minterms in the ON-set of the given function?

WILDCARD-WRITES: Given a configuration Y with n total addresses and a value k , is there a sequence of at most k wildcard writes that implements the given configuration?

TWO-LEVEL-LOGIC is known to be NP-complete, by a reduction from the MINIMUM-DNF problem [Garey79]. In the following, we establish the NP-completeness of WILDCARD-WRITES.

Theorem: WILDCARD-WRITES is NP-complete.

Proof: First, we observe that WILDCARD-WRITES is in NP, since if we are given a configuration Y , a value k , and a sequence of wildcard writes, it is easy to verify in polynomial time that the sequence contains at most k wildcard writes and implements the configuration Y .

To show that WILDCARD-WRITES is NP-hard, we proceed by reduction from the TWO-LEVEL-LOGIC problem.

Let an instance (X, j) of TWO-LEVEL-LOGIC with n terms be given. Each of the n terms is in either the ON or the OFF set of the function X . We will construct a corresponding instance (Y, k) of WILDCARD-WRITES as follows: For each address in the configuration Y , set the address to “1” if the corresponding term is in the ON set of X , and to “Don't Touch” if the corresponding term is in the OFF set of X . Set $k = j$. (For an example of the construction of Y from X , see Figure 2.)

Now, we observe that there is a one-to-one correspondence between a cube that covers a collection of minterms in the ON set of X and a wildcard write that sets the values of the corresponding addresses in Y to “1”. It follows from this observation that the ON set of X can be precisely covered by at most j cubes if and only if Y can be implemented with at most j (equivalently, k) wildcard writes. Since it is clear the construction of (Y, k) from (X, j) can be performed in polynomial time, it follows that TWO-LEVEL-LOGIC is polynomial-time reducible to WILDCARD-WRITES.

From the reducibility of TWO-LEVEL-LOGIC to WILDCARD-WRITES, it follows that WILDCARD-WRITES is NP-hard. Since we have already established that it is in NP, it follows that WILDCARD-WRITES is NP-complete. \square

Since even the simpler decision version of the problem of finding the smallest set of wildcard writes that implements a particular configuration is NP-complete, we are unlikely to find an efficient (polynomial-time) algorithm to construct the smallest such set of writes. Thus we focus our attention in this paper on heuristic techniques instead.

Because of the similarity of the two problems, we should be able to use standard logic minimization techniques to find the wildcards for the configuration problem. For the example in Figure 3, 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 cells in the Karnaugh map. Since we have Wildcard Registers, we can compress the 4 configuration memory addresses in the cube into one address “--10”, where “-” means wildcard. Before configuring these 4 cells, we first set the Row Wildcard Register to “11”(which means the row address following is read as “--”) and the Column Wildcard Register to “00”. The row address decoder then automatically decompresses the address, configuring all four cells at the same time.

	00	01	10	11
00	1	1	2	DT
01	1	1	2	DT
10	1	3	2	3
11	3	3	2	DT

Figure 3. Example for demonstrating the potential for configuration compression.

Even though this configuration problem can be viewed as the logic minimization problem, there is a difference between these two problems. In logic minimization the logic is static, which means all “1” terms are written in the Karnaugh map at the same time, and the sum of the product terms (cubes) exactly covers the logic for each output. However, in configuration compression the configuration is done dynamically, which means that later writes can overwrite previous values. Thus, we can consider the values of the cells that have not yet been written into the FPGA as Don't Cares. With these Don't Cares, we may be able to use fewer product terms (cubes) to cover the cells which need to be written to the FPGA, reducing the number of writes in the configuration. For example, in Figure 3, suppose data “1” is written before data “3”. We can find a single cube to cover all the “1”s, instead of 2, if we consider the cells with data “3” as Don't Cares (Figure 4a). This means we need just one address write to configure all “1”s. Of course, all cells covered by the cube shaded in Figure 4a are configured with data “1”, including those cells which actually require the data “3”. However, since the XC6200 FPGA is a reconfigurable device, those cells currently with the wrong data can be rewritten with the correct configuration later, as shown in Figure 4b.

	00	01	10	11
00	1	1		
01	1	1		
10	1	X		
11	X	X		

(a)

	00	01	10	11
00	1	1		
01	1	1		
10	1	3		3
11	3	3		

(b)

Figure 4. Example of the use of Don't Cares in configuration compression.

From the example above, we can see that the order in which specific values are written has an impact on the total number of writes needed. If we ignore Wildcard Register writes, the total number of writes needed to complete the configuration in Figure 3 is 4 for the case that the “1”s are written before the “3”s. However, for the case that the “3”s are written before the “1”s, the total number of writes will be 5. This is because we can write all “1”s in one cycle if the “3”s are Don't Cares, while the “3”s will take two writes regardless of whether the “1”s are written before or after the “3”s. Thus, we not only have to consider how to most efficiently write each value into the memory, but also what order these writes should occur to best compress the data. We can certainly find an optimal sequence for a specific configuration by doing exhaustive search, but the runtimes would be significant. Thus, heuristic algorithms are required not just for finding Wildcarded addresses, but also to determine the order of Wildcard writes. Before we present these heuristics, we first introduce the logic minimization technique we used for our configuration algorithm.

Wildcarded Address Creation via Logic Minimization

The logic minimization problem is a well-known NP-complete problem, and there exist sophisticated heuristic algorithms to find near optimal solutions. The Espresso algorithm [Brayton84] is widely used for single-output logic optimization, and it is claimed that optimal solutions will be produced in most cases. We use Espresso as a major portion of our configuration compression algorithm. The input required by Espresso is an encoded truth table, as shown in Figure 5(left). Each line consists of a minterm index encoded in binary, followed by either a “1” (for members of the On set) or a “-” (for members of the Don't Care set). The corresponding minimized truth table is shown in Figure 5(right).

1000	1	00--	1
0001	1	-000	1
0010	1		
0011	1		
0000	-		

Figure 5. Espresso input (left), and the resulting output (right).

The configuration memory addresses in the .cal file can be viewed as the minterms for the Espresso input file. Assume for example that we decide that the “3”s are the next values to write to the memory, and that the “1”s have already been written, though the “2”s have not. We can use Espresso to find the proper Wildcarded writes by assigning all addresses with the value to be written assigned to the On set, all Don't Touch and already written values assigned to the Off set, and all values not yet written assigned to the Don't Care set. Thus, the “3” addresses would be passed to Espresso with a “1”, and the “2” addresses would be passed with a “-”. The results of Espresso will be a set of cubes that correspond to Wildcarded writes. These writes contain all of the addresses that need to be set to the value to be written, as well as locations that will be written in future writes, yet will contain none of the Don't Touch nor already written addresses.

Now we present the first stage of our algorithm:

1. Read the input .cal 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 .cal 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.

This algorithm uses the Espresso-based techniques discussed earlier, with a greedy choice of the order in which to write the different values. We greedily pick the group with the most addresses in it because this group should benefit the most from having as many Don't Cares as possible, since the values may be scattered throughout the array. An example of this is shown in Figure 6. If we choose to write the "5"s first, the total number of writes (excluding Wildcard Register writes) is 5, while it only requires 3 writes if the "6"s are written first. This greedy method has been as efficient as other more complete heuristic methods we have implemented.

	00	01	10	11
00	6	6	6	6
01	6	6	6	6
10	6	6	6	5
11	6	6	5	6

Figure 6. An example to illustrate the reason for selecting bigger groups.

Since a single cube may not cover all the addresses in the currently picked group, we pick the cube that covers the most addresses since it provides the greatest compression factor. When this group is picked again (in order to cover the rest of the addresses) we will put Don't Cares for those configuration memory addresses "occupied" by the same function data. Thus, later cubes are still allowed to cover these earlier addresses, since writing the same value twice does not cause any problems.

One additional optimization we have added to the algorithm is to perform a preprocessing to determine if any of the groups will never benefit from any Don't Cares, and thus can be scheduled last. For each group, we run Espresso twice. In the first run, all locations that will be configured, except for the members of the group, are assigned to the Don't Care set. In the second run, these nodes instead form the Off set. In both cases the group members are assigned to the On set. If the number of cubes found in both runs are identical, it is clear that the Don't Cares do not help in reducing the number of writes for this value. Thus, this group is always scheduled last.

One final concern for the first stage of our algorithm is the XC6216 column Wildcard restriction. Because of the electrical properties of the memory write logic, the architecture restricts the number of Wildcards in the column address to at most 4 bits. To handle this, we examine the cube picked in step 7 and see if it meets this restriction. If there are too many Wildcards in the column bits, we iteratively pick one Wildcard to remove until the restriction is met. To pick the Wildcard to remove, we determine how many addresses have a "0" in a given Wildcard bit, and how many have a "1". The Wildcard removed is the one with the most addresses with a specific value ("1" or "0"), and that value replaces the Wildcard.

Once the first stage of the algorithm is completed, we have a list of address data pairs, with Wildcards in most of the addresses, which will produce the desired configuration. However, while this series of writes assumes that the Wildcard Registers can be set in the same cycle with the configuration memory write, it actually takes 3 cycles to perform this operation: Row Wildcard Register write, Column Wildcard Register write, and configuration memory write. Thus, the Wildcard writes will triple the total number of writes. In stage two of the algorithm we use techniques for sharing Wildcard Register writes between multiple configuration memory writes, significantly reducing this overhead.

Write 1 Addresses	Write 2 Addresses	Original Writes	Reduced Writes
(000000, 000100)	(100000, 100100)	(0-0-00, 00--00)	(0-0000, 00--00)
(010000, 000100)	(100000, 100100)	(1-0000, 1---00)	(1-0000, 10--00)
(010000, 001000)	(110000, 101000)		

Figure 7. An example of Wildcard reduction. The addresses to be configured are shown at left. At center is the set of writes given by the first stage, which requires unique row and column Wildcards. The reduced version at right can share both row and column Wildcards by removing some Wildcard bits.

The Second Stage of the Compression Algorithm

The objective of this stage is to reorder the sequence of writes created in the first stage in order to share Wildcard Register writes between configuration memory writes. Also, since Espresso will find the largest cube that covers the required configuration addresses, there may be some wildcard bits that can be changed into “0” or “1” while still covering all required memory addresses. Performing such reductions may increase the number of compatible Wildcard Register values, again increasing Wildcard Register value sharing. We call this second transformation “wildcard reduction”. Figure 7 gives an example of two consecutive writes that cannot share any Wildcard Register values after the first stage, yet after wildcard reduction both wildcards can be shared. The number of writes needed for writing the 6 configuration memory addresses is down to 4, 2 less than that without wildcard sharing.

Before we continue the discussion, we first need to define some terms:

Required Addresses Set: The set of addresses that become occupied because of this write (the addresses this write is used to set).

Maximum Address: The Wildcarded address found by Espresso.

Minimum Address: The Wildcarded address with the minimum number of Wildcards that still covers the Required Address Set.

Intersect(Addr1, Addr2): The set of addresses covered by both addresses Addr1 and Addr2.

And(Wild1, Wild2): The bitwise AND of two Wildcard Register values. Retains a Wildcard bit only when it appears in both values.

Or(Wild1, Wild2): The bitwise OR of two Wildcard Register values. Contains a Wildcard bit when either source Wildcard value has a Wildcard at that bit.

Superset(Wild1, Wild2): True if every Wildcard bit in Wild2 is also in Wild1.

In the second stage we reorder the sequence of writes found in stage one, and apply wildcard reduction selectively, in order to find a new order with a much lower Wildcard Register write overhead. 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 then create a new order, and apply wildcard reduction.

In the first stage, the sequence we created is not necessarily the only order in which the sequence of writes can correctly be applied. For example, the writes in Figure 7 can be reversed without altering the resulting configuration since neither write overwrites relevant data from the other. Of course there are some writes that are not swappable, so we must determine which writes must be kept in sequence, and which can be reordered. Once we have this information, we can reorder the writes to increase Wildcard Register value sharing. The following condition gives one situation in which writes can be reordered, and forms the basis for our partial order generation algorithm. Note that in the paragraphs that follow, we assume that write A preceded write B in the original order.

Condition 1: If $\text{Intersect}(\text{Maximum Address}(A), \text{Required Addresses Set}(B)) = \{ \}$, then A and B can be reordered.

In order to create a partial order, we investigate each (not necessarily consecutive) pair of nodes in the original order. If condition 1 does not hold for this pair of nodes, an edge is inserted into the partial order group, requiring that the earlier write must occur before the later write. Once all pairs have been considered, we have created a partial order for the entire set of writes. 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. All nodes that become schedulable once a given node is removed from the partial order are called the children of that node.

At any given point in the scheduling process the partial order graph determines which nodes are candidates to be scheduled. Now, we must develop an algorithm for choosing the best candidate node to schedule. We use the following rules as our scheduling heuristics. The rules are applied in order, with ties at an earlier rule broken by the rules that follow. Thus, losers at any rule are eliminated, and only the winners are compared with the following rules.

1. Candidate can share both row and column wildcards with the preceding writes.
2. A child of the candidate can share both wildcards with a different current candidate.
3. Candidate can share either the row or column wildcard with the preceding writes.
4. Candidate with the greatest number of other candidates and children that can share both row and column wildcards with it.
5. Candidate with the greatest number of other candidates and children that can share either the row or column wildcard with it.
6. Candidate with the greatest number of children.

Rules 1 and 3 measure the immediate impact of scheduling the candidate on the number of wildcard writes. Rule 2 adds some lookahead, scheduling a candidate early in order to allow its children to share wildcards with another current candidate. Rules 4 – 6 attempt to increase the number of good candidates, hoping that the greater flexibility will result in lower Wildcard overheads.

In order to implement the rules given above, we must determine when two writes can share a row or column Wildcard. To do this, we use the following condition:

Condition 2: If (Maximum Wildcard of A And Maximum Wildcard of B) is the *Superset* of (Minimum Wildcard of A Or Minimum Wildcard of B), then A and B can share the wildcard.

The intuition behind this condition is that if A and B can share a Wildcard, then the Maximum Wildcard of A must be the Superset of the Minimum Wildcard of B, and the Maximum Wildcard of B must be the Superset of the Minimum Wildcard of A. Otherwise, they cannot share the wildcard. Notice that the wildcard sharing is not transitive. That is, if A and B can share a wildcard, and B and C can share a Wildcard, it is not always true that A and C can share a wildcard. For example, B might have all bits as Wildcards, while A and C each have only one Wildcarded position, and the position is different for A and C.

The non-transitivity of the Wildcards is an important consideration. If we apply the scheduling rules discussed earlier pairwise, we may schedule three writes in series which we expect to share all Wildcards, when in fact we require new Wildcard writes before the third write. To deal with this, when a node is scheduled we generate a new Minimum Wildcard and Maximum Wildcard for the schedule so far. These Wildcard bounds must represent all possible values in the Wildcard registers at this point in the schedule. This process is captured by the following rules:

1. If the scheduled candidate cannot share the current Wildcard:
Minimum Wildcard(schedule) = Minimum Wildcard(candidate)
Maximum Wildcard(schedule) = Maximum Wildcard(candidate).
2. If the scheduled candidate can share the current Wildcard:
Minimum Wildcard(schedule) = Or(Minimum Wildcard(schedule), Minimum Wildcard(candidate))
Maximum Wildcard(schedule) = And(Maximum Wildcard(schedule), Maximum Wildcard(candidate)).

These rules maintain the Minimum and Maximum Wildcards in order to more accurately determine which candidate can share a Wildcard with the preceding writes. Thus, whenever we apply the rules for determining which candidate to choose, we always use the schedule's Minimum and Maximum Wildcards to determine whether a candidate can share a Wildcard.

Benchmark	Input size	Control writes	Config. writes	Wildcard writes	Total Writes	Ratio	CPU time(ms)
counter	199	40	53	13	106	53.2%	1.3E3
parity	208	16	9	3	28	13.5%	3.0E2
adder4	214	40	43	14	97	45.3%	4.5E3
zero32	238	42	12	3	57	23.9%	4.0E2
adder32	384	31	28	14	73	19.0%	1.7E3
smear	696	44	224	37	305	43.8%	4.5E4
adder4rm	908	46	473	45	564	62.1%	8.3E4
gray	1201	44	530	74	648	52.2%	2.6E5
top	1367	70	812	87	969	70.8%	1.3E6
demo	2233	31	423	91	545	24.4%	2.8E6
ccitt	2684	31	346	84	461	17.2%	2.2E6
tally	3366	42	211	42	295	8.7%	4.5E6
t	5819	31	834	192	1057	18.2%	1.1E7
correlator	11011	38	1663	225	1926	17.4%	5.0E7
Geometric Mean:						27.7%	

Table 1. The results of the compression algorithm on the benchmark circuits.

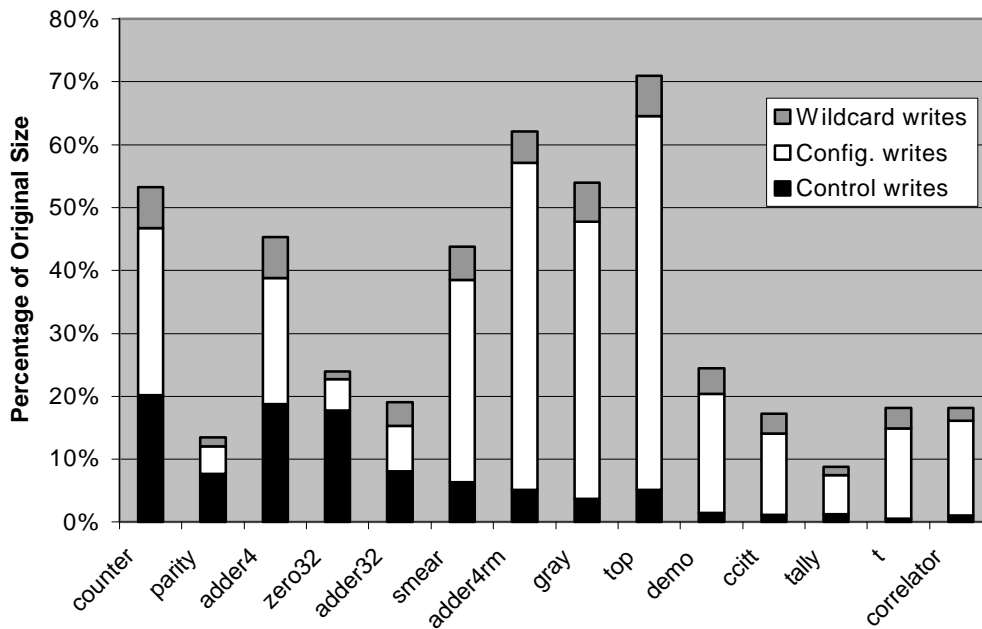


Figure 8. Graph of compressed file size as a portion of the original file. Bar height represents the total resulting file size, and is broken into components. The gray portion is the writes to the wildcard register, white is the actual writes to the array, and the black represents the fixed, non-compressible portions of the files.

Experimental Results

The algorithm described above was implemented in C++ on Sun Sparc20, and was run on a set of benchmarks collected from current XC6200 users.

The results are shown in Table 1 (as well as graphically in Figure 8). The size of the initial circuit is given in the “Input size” column in term of the number of configuration writes in the original .cal files. This size includes all writes required to configure the FPGA, including both compressible writes to the logic array, as well as non-compressible control register writes. The “Control Writes” column represents the number of non-compressible writes, and is a fixed overhead for both the original and compressed file. The size of the compressed file is contained in the “Total Writes” column, which includes control writes, writes to the logic array (“Config. Writes”), and writes to the Wildcard registers (“Wildcard Writes”). The ratio column is the ratio of the compressed file size to the original file size. The CPU time for compressing each benchmark is represented in the last column. As can be seen, the algorithm achieves a factor of 1.4 to 11 compression ratio, with an average compression factor of almost 4. This represents a significant reduction in the bandwidth requirements for reconfiguration in reconfigurable systems.

Conclusions

One of the primary problems in reconfigurable computing is the time and bandwidth overheads due to reconfiguration. This can overwhelm the performance benefits of reconfigurable computing, and reduce the potential application domains. Thus, reducing this overhead is an important consideration for these systems.

In this paper we have presented what we believe is the first general-purpose compression algorithm for reconfigurable computing configurations. By using the fixed decompression hardware in the Xilinx XC6200 it achieves almost a fourfold reduction in bandwidth requirements, which will result in a significant reduction in reconfiguration times. By combining this technique with other configuration management strategies, including configuration prefetching [Hauck98a], we believe that the reconfiguration overhead can be virtually eliminated from many reconfigurable computing systems.

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

- [Brayton84] R. K. Brayton, G. D. Hachtel, C. T. McMullen and A. L. Sangiovanni-Vincentelli, “Logic Minimization Algorithms for VLSI Synthesis”, *Kluwer Academic Publishers*, 1984.
- [Garey79] M. R. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.
- [Hauck98a] S. Hauck, “Configuration Prefetch for Single Context Reconfigurable Coprocessors”, *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65-74, 1998.
- [Hauck98b] S. Hauck, “The Roles of FPGAs in Reprogrammable Systems”, *Proceedings of the IEEE*, Vol. 86, No. 4, pp. 615-639, April 1998.
- [Villasenor96] J. Villasenor, B. Schoner, K.-N. Chia, C. Zapata, H. J. Kim, C. Jones, S. Lansing, B. Mangione-Smith, “Configurable Computing Solutions for Automatic Target Recognition”, *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 70-79, 1996.
- [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.