Silicon Compaction/Defragmentation for Partial Runtime Reconfiguration

Kolin Paul
Department of Computer Science
IIT Delhi
Hauz Khas
New Delhi 110016, India
email: kolin@cse.iitd.ac.in

Joel Porquet
Universit Pierre et Marie Curie
4, Place Jussieu
75252 Paris Cedex 05
France
email: joel.porquet@laposte.net

Abstract

The effective use of Run Time Reconfiguration (RTR) in modern FPGAs opens up new avenues to design area and power efficient high performance architectures. However the current design flow for exploiting RTR in designs, leads to the problem of silicon Defragmentation. We propose a silicon compaction/defragmentation technique which works on already placed and routed modules to generate partial bitstreams (programming files) for the device. We have outlined a method which generates these partial bitstreams very fast taking into account the size and position of the "free" silicon when the device is in operation. The other advantage of this method is that the changes in the basic FPGA fabric needed to implement this defragmentation strategy are (almost) trivial.

1 Introduction and Motivation

Since the mid and late 90's FPGAs have really come into their own mainly because of increased gate densities in the chip themselves and also the speed at which circuits can operate. Concurrent with this development has been the realization of the concept of dynamic reconfiguration enunciated very lucidly by Lysaght[16]. The run time reconfiguration(RTR) characteristic of the new breed of FPGAs presents us with unique challenges and opportunities that will (in the near future) have wide ramifications in both the way we design processors as well as in the development of CAD tools to support the design process. With chips containing billions of transistors becoming the norm in the future, effective ways of utilizing this silicon space is an active research area. One way is to build bigger processors — but that means we invest a lot of effort to develop ways to make the chip not turn into a furnace. Rather than doing this, if we build a small core processor and leave most of the silicon "free", then we can build custom circuits on this "free" area. Conceptually, this is feasible if we can effectively harness the "free" silicon by building different circuits which are separated temporally. The other perspective is to use very small silicon area (solves the power issues) and use that area to dynamically change "behavior" when the application is executing. We already see that today — in processors where different applications can run on the same processor. We endeavor to take this concept to encompass functional partitioning within an application. We feel that future processors will have a portion of their silicon area "free" to be (re)configured at runtime and this is going to be the predominant design space of processors. The challenges that we faced when we incorporated Virtual Memory(VM) and MMU in the design space of processors was a watershed. The incorporation of "free" silicon to be used for RTR in the design of processors is the second watershed that we face today and the challenges and opportunities are similar to the previous watershed. Just as VM offers the user (the system programmer) an unlimited (virtually) amount of (virtual) memory (implemented in a finite amount of "real" memory), the processor designer will have an "infinite" amount of "virtual" silicon to design and build high performance applications on a limited amount of "real" silicon.

The advantages of RTR are often offset by the problems of limited runtime reconfigurability in current devices supporting RTR. As we will see in detail in a later section, the principal limitation is that the "reconfigurable" regions have to be placed and routed well in advance of their being used. The reason for this is that placement and routing algorithms (P&R) are very time consuming and hence it is not feasible to do P&R files at runtime. In fact, the floor planning has to be done for all the dynamically swappable modules during the time of designing and generating the "bit" file(s) necessary for the programming the device. This in essence, means that we have constrained the RTR paradigm to be available only at "compile" time. This indicates that there are cases where we cannot effectively use the available silicon because of the same issues that operating systems designers faced in solving the memory compaction problem [18]. Researchers in the general area of reconfigurable computing have also reported the analogous problem which we term as "silicon compaction'[9, 19, 4]. This paper reports a method which would allow the "fast" generation of partial bit files while working with an already placed and routed design. *The primary motivation of this paper is to provide a solution to mitigate the problem of silicon compaction (or equivalently defragmentation) to enable development of applications using RTR in the design space of embedded high performance processors.*

10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)
0-7695-2978-X/07 $25.00

2 Review and Background

Dynamic reconfiguration of hardware or adaptive computing systems have been the topic of study in academic circles for some time now. Lysaght et. al.[16] were probably the first to recognize the potential of these important feature with respect to FPGAs and provide a taxonomy for different forms of runtime reconfiguration. Recently, things have started moving in industry where people have realized that this design paradigm can be gainfully employed to build high performance applications. A lot of network applications have been built using FPGAs[11]. Following Horta et al[10], systems can either be Compile time Reconfigurable(CTR) or Run Time Reconfigurable(RTR)[2]. CTR systems are statically compiled and donot change behavior during the life time of the application — typical examples are SPLASH[7] and PAM[15]. RTR systems change the structure of the hardware circuit at runtime either by full reconfiguration[13, 6] or by partial reconfiguration[12, 17, 5]. The ability to partially reconfigure the silicon while a portion of the silicon is "running" enables us to really exploit the maximum in terms of "reuse" and is more interesting. Partial runtime reconfiguration allows an FPGA to implement multiple functions which are separated temporally on the same silicon area — and the configuration can be changed while the application is running. There are two basic flows to implement partial RTR using Xilinx based devices.

- Difference Based
- Module Based

Using tools like the FPGA Editor sections of the placed and routed design can be modified. Bitstream producing tools like BitGen with appropriate switches then can produce custom bitstreams that only modify small sections of the device. Switching the configuration of a module from one implementation to another is very quick, as the bitstream differences are smaller than the changes to an entire device bitstream. These bitstreams can be loaded quickly and easily due to their size and software support. *The experimental results presented by researchers [8] show that if the differences are small, the gain is significant.* The reconfiguration overheads are very low. This method is, of course, suitable only if the changes are small and very local. If we have to make changes which are large, then it is more practical to follow the modular based approach to implement the partial runtime reconfiguration. ***One crucial observation that we noticed in this approach is that if the changes are made on the placed and routed design, the generation of bitstream as well as the reconfiguration overheads are low.***

The design flow in the case of Module Based Partial Reconfiguration is slightly different. We will walk through the design of the synthetic pedagogical example. Let us assume that the application can be functionally (and temporally) partitioned into 5 distinct regions (A,B,C,D and FL). The four regions A,B,C and D are candidates which are be swapped in dynamically. FL refers to logic which does not change during runtime.A general floor-plan is constructed which is referred to as the Initial budgeting phase where we set up the area constraints (Figure 1). The regions A, B, C and D are mapped with reference to the region FL. Any communication between the modules is done using Bus Macros[1].

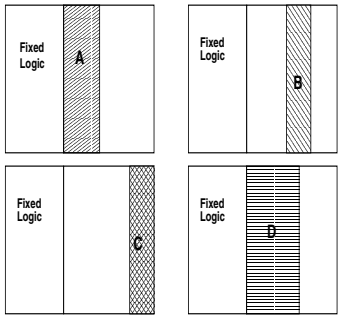

Figure 1. Floor-plan of Application

In the next phase, the individual runtime swappable modules are implemented. This is known as the active implementation phase. Here, these modules (A,B,C and D) are placed and routed according to the area constraints defined by the floor-plan in the earlier phase. In the next phase, the final bitstreams are generated. This phase is termed as the final assembly. Clearly the position of the individual runtime modules are fixed in the second phase of this design flow. The partial bit streams that are generated correspond to the floor-plan defined in the first phase. This design flow constrains the partial bit streams to be loaded in exactly the same place defined in the initial budgeting phase. This "static nature" of the "dynamically loadable modules" leads to **silicon fragmentation**. We illustrate with the following use case scenario of the application.

The application starts with the regions FL and A in place — this implies that A is being used. During this time the partial bit streams are prefetched and loaded into the regions anticipating (temporal scheduling) that they would be required subsequently. This reduces the reconfiguration overheads and the time required for the context switch. After some time, module A and C have finished their jobs and are ready to be swapped out — in effect the silicon area occupied them is ready for reconfiguration (Figure 2). At this point in the execution of the application, we notice two crucial things:

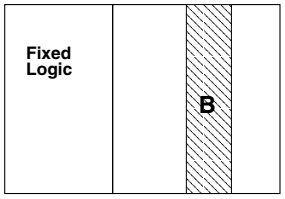

Figure 2. Floor-plan of Application (at Runtime)

- There is free silicon (regions previously occupied by A and C).
- The free silicon is fragmented (module B is still operational) and thus regions A and C are not available as a contiguous region.

This is the classic memory fragmentation/compaction problem. Clearly we cannot schedule the module D until module B has finished operation. In our example, module D requires an area which

is greater that either of A or C but less than the combined area of A and C. We cannot however create a new partial bit stream with D being split into regions A and C. This is because of two reasons:

- It is well known that place and route techniques are very time consuming and also expensive in terms of space requirements. Hence we do not have the luxury to do a execute a Place & Route algorithm at runtime.
- Routing signals between the regions A and C through a module in operation (B) is not feasible. Current partial reconfiguration fabrics require that we use entire columns while floor-planning for different modules.

The second point is actually an engineering limitation in current devices and is essentially related to the manner in which we program the device. This can be easily overcome if we allow the reconfiguration regions to be rectangular without requiring that the height of the module be fixed to the number of rows of active components (CLBs) in the device. In fact, the latest version of the PlanAhead tool offered by Xilinx allows rectangular runtime programmable regions to be defined([14]). However the first issue is a more complex one and is crucial for effective utilization of silicon as well as ensuring that high performance applications using RTR can be built. The rest of the paper provides a detailed algorithm to do a fast generation of partial bit streams with area constraints being determined dynamically at runtime.

3 Silicon Compaction Strategy for RTR

The difficulty in reusing fragmented silicon for loading modules at runtime is because of the "non-relocatable" nature of hardware circuits when compared to software modules. This is especially true when we do design keeping in mind that we need to use our available real estate in the most efficient manner possible. This implies that we impose very hard timing as well as area constraints and the resulting optimization problem (in P&R) is very expensive. Most of the algorithms used in P&R for FPGA like fabrics also use the same concepts of using very "good" but expensive algorithms to optimally utilize the resources. The use of the modern FPGA fabrics allows us to reuse real estate area at run time which in turn, opens up new vistas in the design of high performance area (power) efficient architectures. Unfortunately the static component in the design flow which compels us to do an offline place and route does not allow us to fully utilize the potential promised by these fabrics. In this paper, we propose to use a well known feature of the FPGAs to generate the programming files (bit streams) taking into consideration the area constraints which are available to us during runtime. We will discuss the proposed method informally in the following paragraph to build up the case for the formal algorithm.

With reference to the running example, we have seen that although we do have adequate silicon resources available (area freed by module A + area freed by module C is greater than the area required by module D), we cannot load module D because of fragmentation. We propose that we work with the placed and routed graph corresponding to the module D. (The generation of the bit stream from the P&R graph is of polynomial order (time) complexity). In general, D is a (strongly) connected graph as shown

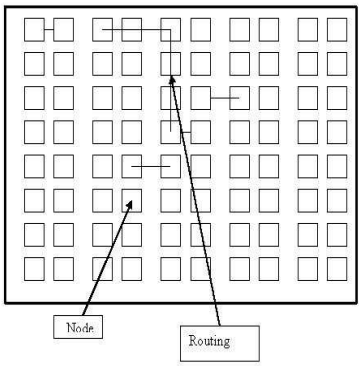

Figure 3. Placed & Routed Graph

below. In this, the nodes in the graph represent the basic logic elements (LUTs / CLBs / Flip Flops etc) and the edges represent the connectivity (routing). The size of the graph (number of nodes) is directly proportional to the area occupied by the module in the device. We propose a bipartite partitioning of the graph into two regions **X** and **Y** subject to certain constraints. The first constraint is that that the number of nodes in X and Y are such that they fit in the regions A and C respectively. This is principally because we want to physically displace the portions of the module from the initially intended place to positions which are determined during a later time. ***Clearly this is possible in the case of FPGAs because all the regions have uniform resources as well as predictable delays though both the logic elements as well the routing elements. This important property of FPGAs is the basis for our algorithm for runtime silicon defragmentation.*** The second issue is to be able to route signals from regions X and Y. Clearly this is only possible if we reserve some silicon for performing this routing. This is illustrated in the left half of Figure 4.

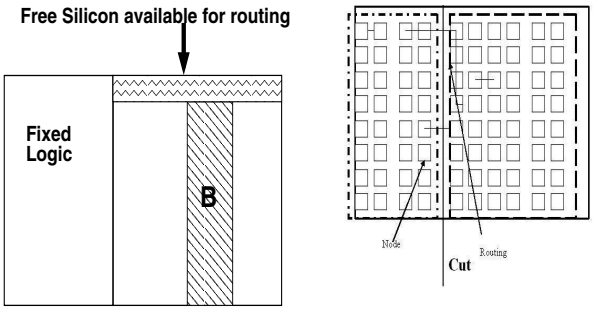

Figure 4. Routing Area for Modules and P & R partitioned graph

We would like to run a Min-Cut (K) algorithm here to determine where we should have the partition so that the signals going from region X to region Y are less than or equal to K. This is illustrated in the right half of Figure 4. This becomes our second constraint. The third and most crucial aspect that we need to keep in mind is the timing issue. Clearly we do not want to suffer from a degradation of performance because of relocating

0-7695-2978-X/07 $25.00

portions of the circuit in different locations. However we do know the upper bound of the delay (D) that will be incurred when we are placing the partitioned circuit in regions X and Y. In our example, this value is equal to the delay in the routing elements placed in a region whose width is the same as B. We also know that all nets in our module will not have the same critical path delay. Therefore

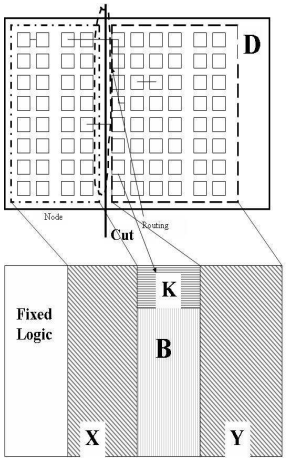

Figure 5. Floor-plan of D following the partitioning

our third constraint is to select a suitable cut where we have a slack in the combinational delay between the wires crossing the cut which is greater than upper bound D mentioned above. The floor-plan of the circuit under these three constraints would appear similar to Figure 5.

We also use a heuristic which doesnot look at the entire P&R graph but only considers a region around the center of the graph. (We assume that topologically the graph can be laid out in the same manner as the actual device. In such a case the graph will have a length as well as width commensurate with the device properties). This is indicated in Figure 6 and is motivated by the consideration that in essence, we are trying to break up the module (D) represented by the P&R graph into two regions which satisfy certain area constraints (to fit in regions X & Y). We do not want to tamper with the P&R in these regions — we simply want to be able to find a partition satisfying the three constraints which enables us to translate horizontally a portion of the module (say to the right of the cut) to be placed in a different region.

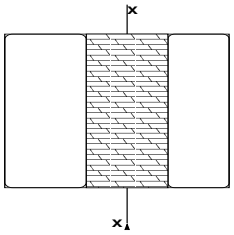

Figure 6. Floor-plan of D following the Partitioning

4 Experimental Setup

Versatile Place and Route [3] is an opensource FPGA placement and routing tool. It produces two files which respectively contain the placement of all the blocks and the routing of all the nets. We use these files to build our P&R Graph at runtime. We modified **VPR** in order to add the slack time of each net in the routing file. The slack times are calculated as follows :

- **VPR** finds the delays of all possible paths by computing the delays through each CLB and net which are part of each path.
- The path with the highest delay becomes the critical path and its delay becomes the cycle time.
- Finally, by scanning all the paths from end to start, **VPR** can compute the slack times of each net by comparison with the cycle time.

Our algorithm tries to break up a module into two regions R1 and R2. If S is the width of the module (in terms of CLBs) then clearly, the addition of the widths of R1 and R2 must be equal or greater than S. Therefore the window of the module in which the mincut can be performed is the union between the widths of R1 and R2. Obviously, if the addition of the widths of R1 and R2 is equal to S, running the bipartitioning algorithm would be useless since no cell would be able to be moved. Figure 7 displays an example of a 6 CLBs width module, with a region R1 of 5 CLBs width and a region R2 of 4 CLBs width.

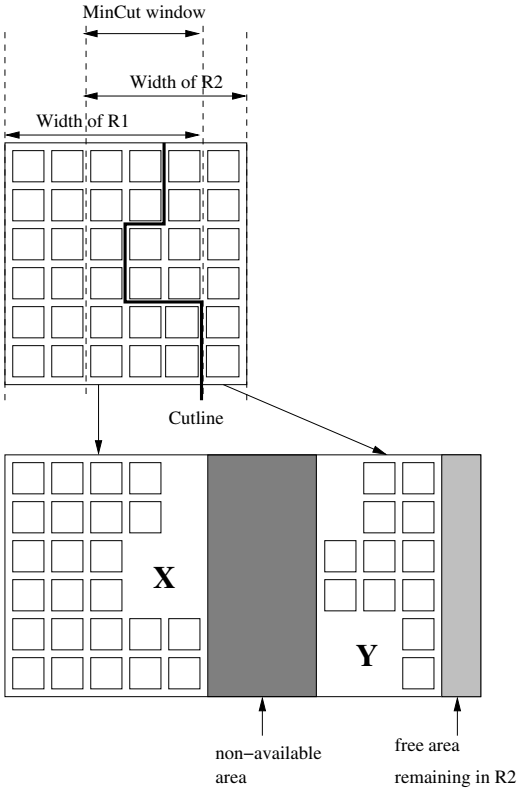

Figure 7. MinCut window

Furthermore, all the nodes in the window cannot be moved. In this example, only the nodes which are along the vertical segments of the cutline can be moved. Figure 8 shows two bipartitioning situations : the first one is not allowed but the second one is.

0-7695-2978-X/07 $25.00

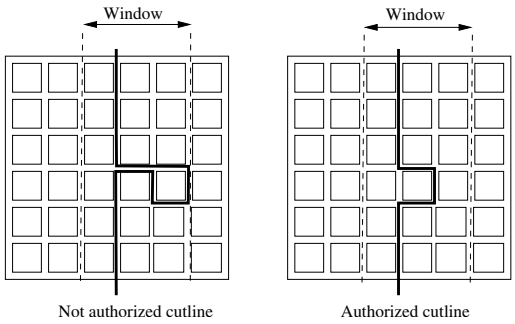

Figure 8. Two bipartitioning situations

Therefore a node can be in 3 states : fixed if it is not in the window ; locked if it is in the window but unmoveable ; mobile if it is in the window and moveable.

Furthermore, as we have mentioned earlier, the routing resources between regions R1 and R2 are not infinite. Let us assume that K tracks are available for the nets which are crossing the cut. That would means that only K distinct nets would be able to cross the cut. However, in an FPGA, a source CLB can reach multiple CLB targets with distinct nets sharing the same equipotential. Therefore two or more nets with the same equipotential and which are crossing the cut, require only one track. These nets should be rerouted in R1 to merge in one net, then cross the cut, and finally be rerouted in R2 to reach their targets. Consequently, the size of the cut only indicates the number of different equipotentials of the cut.

Also when bipartitioning the module into parts X and Y, we use two upper bounds which we call `x_upper_bound` and `y_upper_bound`. `x_upper_bound` points out the greatest x coord of the CLB(s) belonging to X which are at the left extremum. In the same way, `y_upper_bound points` out the lowest coord of the CLB(s) belonging to Y which are at the right extremum. Then they give the real size of the two regions, after splitting. Thus one of our aims is to minimize the difference between both these bounds in order to reduce the size of the regions. These upper bounds can move during the execution of the bipartitioning algorithm, since CLBs are moved between X and Y to find the mincut.

Since the routing of the nets doesn't generate shortest paths only, sometimes we can observe **fake crossing nets**. A fake crossing net is a net with both source and target belonging to the same part, but which crosses the upper bound of this part. In this case, the net should be rerouted to stay in its own part. Figure 9 shows an example in which the net belonging to X is not a fake net (the routing resources are still available after splitting) but the net belonging to Y is a fake crossing one and thus needs to be rerouted.

5 Bipartitioning Algorithm and Results

Now that we have an intuitive feel of the method to perform a defragmentation of the available silicon area during runtime, we are in a position to state the compaction/defragmentation algorithm formally.

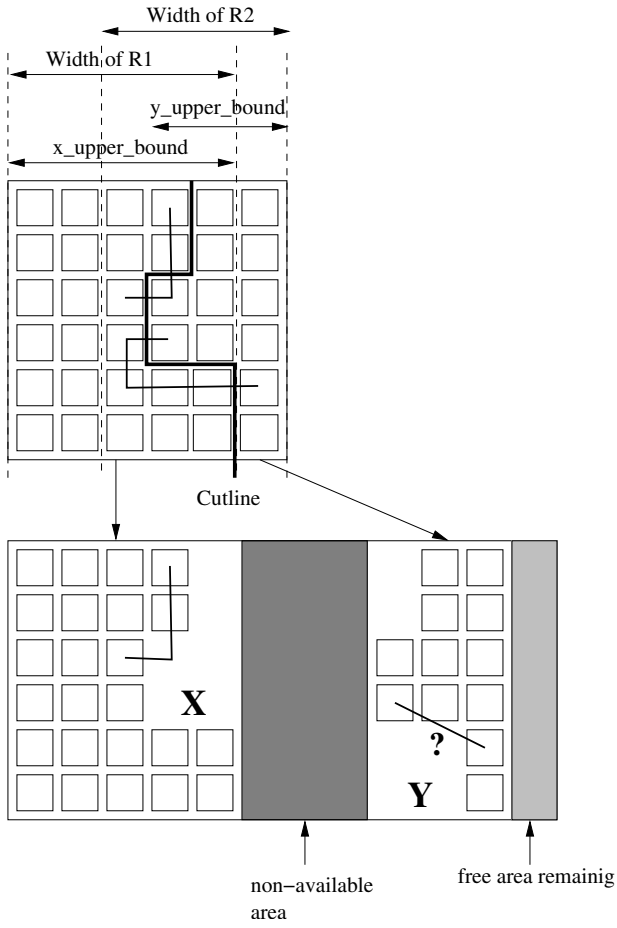

Figure 9. Fake crossing net

Algorithm: BiPartitionCut

Step 1 Determine new area constraints

Step 2 Determine the number of available routing resources (K)

Step 3 Determine the delay that will be incurred due to routing the signals of the "relocatable" module across an executing module(width of the region B).

Step 4 Determine the region to explore for the min-cut(K).

Step 5 Solve min-cut(K) subject to the following constraints:

- array constraints determined in Step One.
- available slack in all lines which are in the min-cut(K) is greater than the delay determined in Step 3.

This entire algorithm is easily encoded in a small embedded processor (like Microblaze) in the fixed logic (FL) region. In order to split a module in two parts, we decide to use a clone of `Feduccia and Mattheyses` mincut algorithm. Let us see how we have transformed it for our needs.

0-7695-2978-X/07 $25.00

5.1 Usual FM algorithm

In the general case, FM algorithm is run on a undirected graph. Each node is tagged with a weight value : a good balance means that we have more or less the same weight in both sides.

Algorithm: FM_Original Initialization

- Generate an initial bipartition with a good balance
- Compute the gain for each cell (ie, if the opposite node is in the other side, gain is incremented; else, it is decremented)
- Initialize the buckets structure (ie, node are sorted by their gain value in order to find the cell with highest gain easily)

Main loop

- Unlock all the cells
- $i = 0$
- Local loop
  - Select the cell with the maximum gain which respects all the constraints (ie, which maintains a good balance if it is moved)
  - Move the cell and lock it
  - Update gains of connected cells
  - Save the new cutsize in history H_i
  - $i = i + 1$
- While some cells are unlocked
- Find k such that $H_k = \sum_{t=1}^{k} H_t$ is maximized
- All moves after k are replaced in their initial part

While the cutsize is improved

In our case, we do not have any weight associated to the cells, so there is no balance to respect. The gain value of a cell is computed as follows : for each of its nets, if the opposite cell is in the same part, gain is decremented. On the contrary, gain is incremented if the opposite cell is in the other part. We also include the fake crossing nets in this computation : the gain is better if the movement of the cell reduces the number of fake crossing nets.

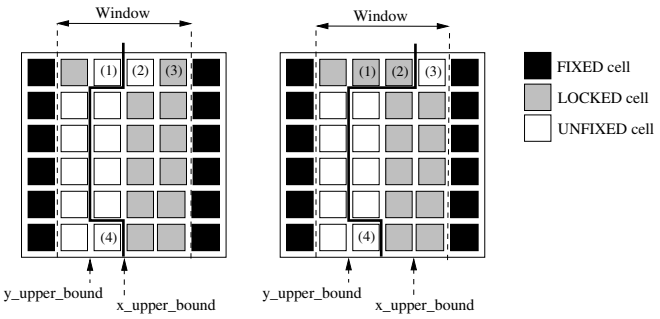

Figure 10. Move cell (2) from Y part to X part and lock it - State of (1) is now locked while state of (3) is now unfixed - The `x_upper_bound` has changed so the gains of the cells which belong to the corresponding column are updated (ie, gain value of (4) is updated because now its potential movement would not change the number of fake crossing nets of X part)

Algorithm: FM_Modified Initialization

- Set the fixed state for cells not in the window
- Generate an initial bipartition : we shift a vertical cutline in the window to find the best position (ie, the one which gives the best cutsize)

Main loop

- Set the unfixed state for moveable cells in the window and set the locked state for other cells in the window
- Compute the fake crossing nets for both parts
- Compute the current cutsize
- Compute the gain value for all unfixed cells and store them in the buckets structure
- $i = 0$
- Local loop
  - Select the cell with the maximum gain : if several cells have the same gain, take the one which movement gives the best upper bounds difference (ie, the size of both parts are minimized)
  - Move the cell, lock it and update the cutsize
  - Update the upper bounds and compute the fake crossing nets for both parts
  - Update gains of connected cells
  - Change the state of cells situated aside (ie, if the cell is moved from Y to X - right part to left part - then the cell on the right is set as unfixed, but only if it is not already in the fixed state. Its gain is computed, and the cell on the left side is set as locked) - See figure 10
  - If the upper bounds have changed, compute the new gain of the cells which belong to the corresponding columns
  - Save the new cutsize in history H_i
  - $i = i + 1$
- While some cells are unfixed
- Find k such that $H_k = \sum_{t=1}^{k} H_t$ is maximized
- All moves after k are replaced in their initial part

While the cutsize is improved

The results of running this algorithm on MCNC benchmarks is shown in Tables 1-2. The first table lists the number of equipotentials present in each of the circuits chosen in the benchmark. This in turn reflects on the routing complexity that we encountered in the circuits while performing the bipartitioning. The second table shows that it is possible to "partition" successfull all the circuits at

the designated positions using our algorithm. The second and third columns of this table denote the sizes of the partitions R1 and R2 and the areas (X&Y)where we want to relocate the second portion of the chosen circuit. For each of the circuits in the benchmarks, it is observed that the number of equipotentials cut is approximately 10% of the total number. This implies that it has been possible to reroute successfully when moving the portion of the circuit to a new region. The table also indicates the actual number of nets cut to create this bi partition. The next column indicates that the "dynamic partitions" have been created within available slack.

6 Conclusion and Future Research

The effective use of Run Time Reconfiguration in modern FPGAs opens up new avenues to design area and power efficient high performance architectures. However the current design flow for exploiting RTR in designs, leads to the problem of silicon defragmentation. We propose a silicon compaction / defragmentation technique which works on already placed and routed modules to generate partial bitstreams (programming files) for the device. We have outlined a method for which generates these partial bitstreams very fast taking into account the size and position of the "free" silicon when the device is in operation. The other advantage of this method is that the changes in the basic FPGA fabric needed to implement this defragmentation strategy are almost trivial. The limitation of the proposed method is that the floorplan of the "dynamic" module must respect the interface with the fixed logic (FL) as also other dynamic modules that are defined in the initial floorplan. We are also in the process of integrating the changes in the algorithm which are necessary when we have the module to be "relocated" communicating with both the FL region as well as another executing module. We are looking at the problem of relaxing this constraint. The other major assumption in this work that the fabric is regular is being looked into as modern FPGAs have embedded multipliers and memory structures which require special handling.

Table 1. MCNC Circuits

Circuit	CLBs width	#CLBs	#Nets	#Equipotentials
count	7	47	142	82
adder	7	49	130	82
5xp1	8	57	208	64
C499	10	74	312	115
C1355	10	74	312	115
b9	11	117	257	158
C432	12	124	420	160
9sym	12	144	490	153
C1908	13	145	534	178
C880	14	174	656	234
C3540	21	431	1597	481
C6288	23	527	2055	559
C5315	38	620	2268	798
C7552	40	735	2559	945
dalu	34	1131	3051	1206
alu4	40	1544	5408	1536

References

[1] Dynamic and Partial Reconfiguration. http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf.

[2] B L Hutchings and M J Wirthlin. Implementation Approaches for Reconfigurable Logic Applications. In *Field Programmable Logic and Applications (FPL'1995)*, August 1995.

[3] V. Betz and J. Rose. VPR: A new packing, placement and routing tool for FPGA research. In W. Luk, P. Y. Cheung, and M. Glesner, editors, *Field-Programmable Logic and Applications*, pages 213–222. Springer-Verlag, Berlin, 1997.

[4] K. Compton, Z. Li, J. Cooley, S. Knol, and S. Hauck. Configuration relocation and defragmentation for run-time reconfigurable computing. *IEEE Trans. Very Large Scale Integr. Syst.*, 10(3):209–220, 2002.

[5] D E Taylor, J S Turner, and J W Lockwood. Dynamic Hardware Plugins (DHP): Exploiting Reconfigurable Hardware forHigh performance reconfigurable Routers. In *IEEE OPENARCH 2001: 4th IEEE Conference on Open Architectures and Network Programming*, 2001.

[6] D Ross, O Vellacott, and M Turner. An FPGA Based Hardware Accelerator for Image Processing. In *More FPGAs: Proceedings of the 1993 International Workshop on Field programmable Logic and Applications*, 1993.

[7] D T Hoang. Searching Genetic Databases on Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines, IEEE Computer Society Press*, 1993.

[8] H. Dhand, N.Goel, M.Agarwal, and K.Paul. Partial and Dynamic Reconfiguration in Xilinx FPGAs : A Quantitative Study . In *Proc. 9rd VLSI Design & Test Symposium*. Bangalore, India, August 2005.

[9] O. Diessel and H. A. ElGindy. Run-time compaction of fpga designs. In *FPL '97: Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, pages 131–140, London, UK, 1997. Springer-Verlag.

[10] E L Horta, J W Lockwood, and D Parlour . Dynamic Hardware Plugins in an FPGA with Partial Runtime Reconfiguration. In *Design Automation Conference*, 2002.

[11] S. Hauck. The Roles of FPGAs in reprogrammable Systems. In *Proceedings of the IEEE*, volume 86, pages 615–638, April 1998.

[12] J D Hadley and B L Hutchings. Designing a Partially Reconfigured System. In *Field programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, Proc Spie 2607*, pages 210–220, 1995.

[13] J M Ditmar. A Dynamically Reconfigurable FPGA-Based Content Addressible Memory for IP Characterization. In *Masters Thesis, KTII Royal Institute of Technology, Stckholm, Sweden*, 2000.

[14] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgeford. Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration on XILINX FPGAS . *16th International Conference on Field Programmable Logic and Applications , Madrid, Spain*, 2006.

0-7695-2978-X/07 $25.00

Table 2. MCNC Circuits Results-I

Circuit	R1,R2 width	X,Y width	Extra Area	#Equipot-entials cut	Nets Cut	Slack	X,Y FC Nets
count	(3,6)	(3,6)	28.57	7	23	4.44E-16	(0,0)
	(5,5)	(4,3)	0.00	5	20	0	(0,3)
	(6,3)	(4,3)	0.00	5	20	0	(0,3)
adder	(3,6)	(3,6)	28.57	4	7	4.44E-16	(0,0)
	(5,5)	(4,3)	0.00	5	7	4.44E-16	(0,3)
	(6,3)	(4,3)	0.00	6	8	4.44E-16	(0,3)
5xp1	(3,6)	(3,6)	28.57	12	84	0	(0,0)
	(5,5)	(4,3)	28.57	9	39	2.01E-09	(0,3)
	(6,3)	(4,3)	28.57	11	21	2.01E-09	(0,3)
C499	(3,6)	(3,6)	0.0	21	56	0	(0,0)
	(5,5)	(4,3)	25.00	18	20	0	(0,3)
	(6,3)	(4,3)	0.00	19	20	0	(0,3)
C1355	(3,6)	(3,6)	0.0	26	59	0	(0,0)
	(5,5)	(4,3)	10.00	28	38	1.77E-09	(0,3)
	(6,3)	(4,3)	10.00	16	18	1.77E-09	(0,3)
b9	(3,6)	(3,6)	20.00	12	46	0	(0,0)
	(5,5)	(4,3)	0.00	14	20	5.25E-09	(0,3)
	(6,3)	(4,3)	20.00	10	20	5.25E-09	(0,3)
C432	(3,6)	(3,6)	9.09	28	46	0	(0,0)
	(5,5)	(4,3)	9.09	21	20	0	(0,3)
	(6,3)	(4,3)	9.09	26	20	0	(0,3)
9sym	(3,6)	(3,6)	0.0	20	73	0	(0,0)
	(5,5)	(4,3)	16.67	20	165	0	(0,3)
	(6,3)	(4,3)	0.00	21	153	0	(0,3)
C1908	(3,6)	(3,6)	0.00	43	106	0	(0,0)
	(5,5)	(4,3)	8.33	37	20	0	(0,3)
	(6,3)	(4,3)	16.67	26	20	4.77E-09	(0,3)
C880	(3,6)	(3,6)	0.00	26	85	0	(0,0)
	(5,5)	(4,3)	7.69	42	20	1.19E-09	(0,3)
	(6,3)	(4,3)	30.77	21	20	1.77E-09	(0,3)
C3540	(3,6)	(3,6)	4.76	45	112	1.78E-15	(0,0)
	(5,5)	(4,3)	0.00	71	172	5.32E-15	(0,3)
	(6,3)	(4,3)	9.52	66	333	5.32E-15	(0,3)
C6288	(3,6)	(3,6)	8.70	34	248	4.44E-16	(0,0)
	(5,5)	(4,3)	8.70	34	262	0	(0,3)
	(6,3)	(4,3)	4.35	35	216	0	(0,3)
C5315	(3,6)	(3,6)	10.53	47	121	3.16E-09	(0,0)
	(5,5)	(4,3)	5.26	52	140	3.16E-09	(0,3)
	(6,3)	(4,3)	2.63	56	155	5.33E-15	(0,3)
C7552	(3,6)	(3,6)	0.00	22	140	0	(0,0)
	(5,5)	(4,3)	0.00	22	140	0	(0,3)
	(6,3)	(4,3)	0.00	56	190	0	(0,3)
dalu	(3,6)	(3,6)	2.94	53	229	4.44E-16	(0,0)
	(5,5)	(4,3)	8.82	61	403	7.08E-09	(0,3)
	(6,3)	(4,3)	8.82	62	338	9.04E-09	(0,3)
alu4	(3,6)	(3,6)	2.50	84	647	6.18E-09	(0,0)
	(5,5)	(4,3)	7.50	114	1395	2.38E-09	(0,3)
	(6,3)	(4,3)	0.00	108	550	1.12E-08	(0,3)

[15] P Bertin, H Touati, and E Lagnese. PAM Programming Environments. In *IEEE Workshop on FPGAs for Custom Computing Machines, IEEE Computer Society Press*, 1994.

[16] Patrick Lysaght, Hugh Dick, Gordon McGregor, David McConnel, and Jon Stockwood . Prototyping Environment for Dynamically Reconfigurable Logic. In *Field Progammable Logic* , 1995.

[17] S McMillan and S Guiccione. Partial Runtime Reconfiguration using JRTR. In *Field programmable Logic (FPL 2000)*, pages 352–360, 2000.

[18] A. Silberschatz and P. B. Galvin. *Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[19] C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Transactions on Computers*, 53(11):1393–1407, 2004.

0-7695-2978-X/07 $25.00