Partial Reconfiguration of Field Programmable Gate Array Devices Using Xilinx Architecture

A dissertation submitted in partial fulfillment of the requirements for the
Master of Technology in Computer Science degree in
Indian Statistical Institute, Kolkata
in the year of 2007.

By

Ayan Roy Chowdhury
(MTC0506)

Under The Supervision of

Dr. Susmita Sur-Kolay

Indian Statistical Institute
203, Barrackpore Trunk Road
Kolkata - 700035

Certificate of Approval

This is to certify that the thesis entitled "Partial Reconfiguration of Field Programmable Gate Array Devices Using Xilinx Architecture" submitted by Ayan Roy Chowdhury towards partial fulfillment for the degree of M.Tech in Computer Science at Indian Statistical Institute, Kolkata, embodies the work done under my supervision.

Dated:

Signed:
Dr. Susmita Sur-Kolay
Supervisor

Countersigned:
External Examiner

Acknowledgement

In the course of my dissertation work I came across some people who have contributed directly or indirectly in imparting knowledge to me. This project would be incomplete without me expressing my hearty gratitude to them.

I wish to convey my regards and sincere gratitude to **Dr. Susmita Sur-Kolay** of Advanced Computing and Microelectronics Unit, Indian Statistical Institute, Kolkata for her excellent guidance and invaluable suggestions throughout my work

I am also indebted to **Prof. Bhargab Bikram Bhattacharya** of the same unit for this continual support, valuable guidance and inspirational motivations without which this project would not have been possible.

I am thankful to **Ms Pritha Banerjee,** for providing constant support and guidance throughout the tenure of my dissertation.

Last but not the least, I am also grateful to my classmates who have encouraged and helped me to carry out the project.

Ayan Roy Chowdhury
20th July, 2007

Chapter 1

1.1 Introduction

Field Programmable Gate Arrays (FPGA) are specific integrated circuits that can be programmed by users easily. The FPGA contains versatile functions, configurable interconnects and input/output interface to adapt to the user specification. It has a bunch of simple, configurable logic blocks arranged in an array with interspersed switches that can rearrange the interconnections between the logic blocks. Each logic block is individually programmed to perform a logic function (such as AND, OR, XOR, etc.) and then the switches are programmed to connect the blocks so that the complete logic functions are implemented. FPGA allow rapid prototyping using custom logic structures, and are very popular for limited production products. Modern FPGA are extremely dense, with complexity of several millions of gates which enable the emulation of complex hardwares such as parallel microprocessors, mixture of processor and signal processing chips etc.

One key advantage of FPGA is their ability to be reprogrammed, in order to create a completely different hardware by modifying the logic gate array. Now-a-days FPGA not only exists as simple component, but also as macro-blocks in system-on-chip designs. In the case of communication systems, the

configurable logic may be dynamically changed to adapt to improved communication protocol. In the case of very low power systems, the configurable logic may handle several different tasks in series, rather than embedding all corresponding hardware that never works in parallel.

1.2 Motivation

As we have mentioned that in several cases we have a series of different tasks those never runs in parallel. In these cases what is usually done is reconfiguring the system totally whenever one task is finished and the next task is triggered.

Configuration of logic for an FPGA is done by using dedicated CAD tools. The subsequent steps in the flow consist of partitioning the circuit, floor-planning on the board followed by placement and routing. The entire flow is handled by CAD tools meant for FPGAs. Now the main advantage of using FPGA in these kinds of applications is that we are utilizing the non parallel nature of the tasks in the process of reconfiguring the same hardware. This saves a huge cost in terms of hardware resource requirements. But one of the problems of this method is the reconfiguration time needed for transition between two tasks. Because for the subsequent task we have to load the new design to CAD tool, configure logic blocks by dumping the design to board followed by the floorplan, place and route stages.

To come up with a possible solution to this problem instead of going for a total reconfiguration of the FPGA device, the concept of partial reconfiguration is creeping in. This is possible because most of the applications are such that we will have some part of the designs similar between different tasks. So in consecutive stages we don't need to configure the entire device if by some means we can keep the common part from the earlier instance unaffected. We have to configure rest portion of the device, not the entire one. This is the essence of Partial Reconfiguration.

1.3 Scopes & Benefits

Partial Reconfiguration in FPGA devices has a very wide scope of practical purpose applications where some online application or communication is active. Partial reconfiguration offers countless benefits across multiple industries. It can be an important component to any design or application – allowing designers more capabilities and resources than meets the eye.

Partial reconfiguration is the ability to reconfigure selected areas of an FPGA anytime after its initial configuration. We can do this while the design is operational and the device is active (known as active partial reconfiguration) or when the device is inactive in shutdown mode (known as static partial reconfiguration).

By taking advantage of partial reconfiguration, we gain the ability to:

- Adapt hardware algorithms
- Share hardware between various applications
- Increase resource utilization
- Provide continuous hardware servicing
- Upgrade hardware remotely

Using partial reconfiguration, we can dramatically increase the functionality of a single FPGA, allowing for fewer, smaller devices than would otherwise be needed. Important applications for this technology include reconfigurable communication and cryptographic systems.

A portion of the design is being reconfigured, as the rest of the system can continue to operate, there is no loss of performance or functionality with unaffected portions of a design – no down time. It also allows for multiple applications on a single FPGA.

We will highlight a few of the benefits of using partial reconfiguration.

- The ability to change hardware – FPGA can be updated at any time, locally or remotely. Partial reconfiguration allows us to easily support, service, and update hardware in the field.

- Hardware sharing – Because partial reconfiguration allows us to run multiple applications on a single FPGA, hardware sharing is

realized. Benefits include reduced device count, reduced power consumption, smaller boards, and overall lower costs.

- Shorter reconfiguration times – Configuration time is directly proportional to the size of the configuration bitstream. Partial reconfiguration allows us to make small modifications without having to reconfigure the entire device. By changing only portions of the bitstream – as opposed to reconfiguring the entire device – the total reconfiguration time is shorter.

1.4 Applications

Partial reconfiguration is useful in a variety of applications across many industries. The aerospace and defense industries have certainly taken advantage of its capabilities. Partially reconfigurable devices have benefited the *Joint Tactical Radio System* (JTRS) Program by a significant amount.

Partial reconfiguration is the cornerstone for power-efficient, cost-effective *Software-Defined Radios* (SDRs). Through the JTRS Program, SDRs are becoming a reality for the defense industries as an effective and necessary tool for communication. SDRs satisfy the JTRS standard by having both a software-reprogrammable operating environment and the ability to support multiple channels and networks simultaneously.

With partial reconfiguration, the ability to implement an SDR modem using shared resources can be realized. A shared resources model enabled by partial reconfiguration of an FPGA to support multiple waveforms can be supported by the SCA as mandated by JTRS. FPGA implementations of SDR, with partial reconfiguration, results in effective use of resources, lower power consumption, and extensive cost savings.

Another example is in mitigation and recovery from single-event upsets (SEU). In-orbit, space-based, and extra-terrestrial applications have a high probability of experiencing SEUs. By performing partial reconfiguration, in conjunction with *Readback*, a system can detect and repair SEUs in the configuration memory without disrupting its operations or completely reconfiguring the FPGA. By the term Readback we mean, the process of reading the internal configuration memory data to verify that current configuration data is correct or not.

In the modern days FPGAs are not only consisting of mere *Configurable Logic Blocks* (CLBs) or even RAM or Multiplier, but beside these there are integrated processor cores, DSP chips and other useful hardware on the same board. So the application area of FPGA is also widening up. As a matter of fact the need for reducing configuration time and cost and increasing efficiency, partial reconfiguration is the method that all FPGA designers need to concentrate.

Chapter 2

2.1 Some Earlier Approaches to Partial Reconfiguration

Partial Reconfiguration as a research topic is fairly new in the field of VLSI physical design. Basically the idea has a few variations as far as the target is concerned. Some of the approaches involve finding a satisfactory schedule out of a series of tasks that would facilitate the implementation of partial reconfiguration. Other approach deals with the case when we do not have the flexibility to schedule the tasks. This is true for all online communication based applications. In these cases we need to maintain the order of the tasks and can not do the scheduling according to our own. We need to place the part which will not be reconfigured in such a way that it doesn't affect the performance of the device by much in either of the tasks.

We will basically concentrate on the second class of the problem that is mentioned above. There are not too many researches done in this topic till date, either in the industries or in the institutes. As far as the industry research is concerned Xilinx Inc has come out with a CAD tool for its FPGA floorplanning, placement and routing that supports partial reconfiguration to some extent. We will discuss in details about their approach towards this problem. Other than industries there

are a few works that deals with this particular problem. We will cite a few of them in the subsequent section.

2.1.1 Partial Reconfiguration in PlanAhead – A Xilinx Approach [Ref.12]

Xilinx offers Partial Reconfiguration option in its CAD tool named ***PlanAhead*** which works with Xilinx-ISE software. This tool is compatible to whole Virtex family and Spartan-3 family of Xilinx FPGAs.

Instead of resetting the device and performing a complete reconfiguration, new data is loaded to reconfigure a specific area of a device, while the rest of the device is still in operation. For current FPGA devices, data is loaded on a column-basis, with the smallest load unit being a configuration bitstream "frame," which varies in size based on the target device.

PlanAhead supports two kinds of partial reconfiguration. Active partial reconfiguration is done when the device is active. Except during some interdesign communication, certain areas of the device can be reconfigured while other areas remain operational and unaffected by the reprogramming. In contrast, static Partial Reconfiguration is done before the device is fully active or when the device is inactive. This can be accomplished by de-asserting the chip select (CS) during configuration, for example, to load in special data. For Partial Reconfiguration to

take place, the rest of the device is in shutdown mode and is brought up again once the configuration is completed.

Active partial reconfiguration is mainly done in two different ways.

- Module-based
- Difference-based

- Module Based Partial Reconfiguration – This is also done in two ways depending upon whether any communication is needed between the modules.

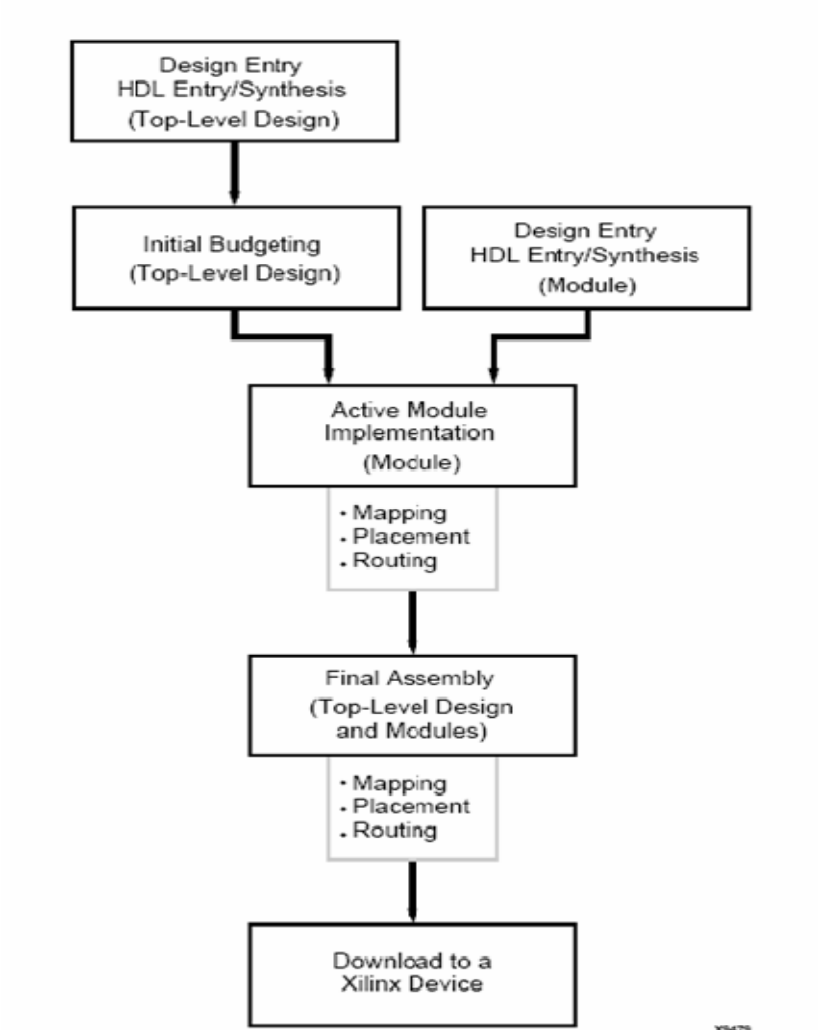


Fig. Module based Design flow overview

If the modules are independent that means a reconfigurable (dynamic) module doesn't interact with any other module (static/dynamic) then conventional Modular design flow is used. If a communication is needed between reconfigurable (dynamic) module and any other module (static/dynamic) then BUS MACRO is used at the boundary of two such modules.

- Bus Macro is a pre-routed hard macro so doesn't change from instance to instance.
- Bus Macro provides a fixed BUS of Inter Design Communication.
- Each time Partial Reconfiguration is performed, the Bus Macro is used to establish unchanging routing channels between modules.
- The HDL code should ensure that any reconfigurable module signal that is used to communicate with another module does so only by first passing through a bus macro.

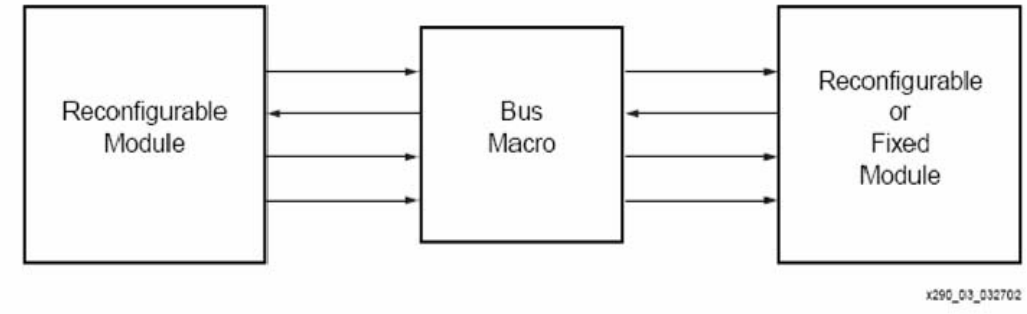


Fig. Bus Macro used for Intermodule signal

- Difference-based Partial Reconfiguration – This is done by changing the design by a small amount (such as changing I/O standards, LUT equations and block RAM content).

  Two supported ways to make such design changes: at the front end or the back end.

  - Front-End:
    - This involves changes in HDL or Schematic.
    - The design must be re-sythesized and re-implemented to create a new 'Place And Route' NCD file.

  - Back-End:
    - Changes can be made directly in the NCD file.
    - Section of a design can be modified using FPGA editor tool.
    - BitGen switches then can produce custom bitstreams that only modify small sections of the device.

The main drawback that this tool has is the restrictions in terms of a set of design rules to be followed while placing the modules in the board. Reconfigurable modules must have the following properties:

1. The reconfigurable module height is always the full height of the device.
2. The Reconfigurable module width ranges from a minimum of four slices to a maximum of the full-device width, in four-slice increments.

3. Horizontal placement must always be on a four-slice boundary; the leftmost placement being x = 0, 4, 8, …
4. All logic resources encompassed by the width of the module are considered part of the reconfigurable module's bitstream "frame." This includes slices, TBUFs, block RAMs, multipliers, IOBs, and most importantly, all routing resources.
5. Clocking logic (BUFGMUX, CLKIOBs) is always separate from the reconfigurable module. Clocks have separate bitstream frames.
6. IOBs immediately above the top edge and below the bottom edge of a reconfigurable module are part of the specific reconfigurable module's resources.
7. If a reconfigurable module occupies either the leftmost or rightmost slice column, all IOBs on the specific edge are part of the specific reconfigurable modules resources.
8. To help minimize problems related to design complexity, the number of reconfigurable modules should be minimized (ideally, just a single reconfigurable module, if possible).

This is said, the number of slice columns divided by four is the only real limit to the number of defined reconfigurable module regions.

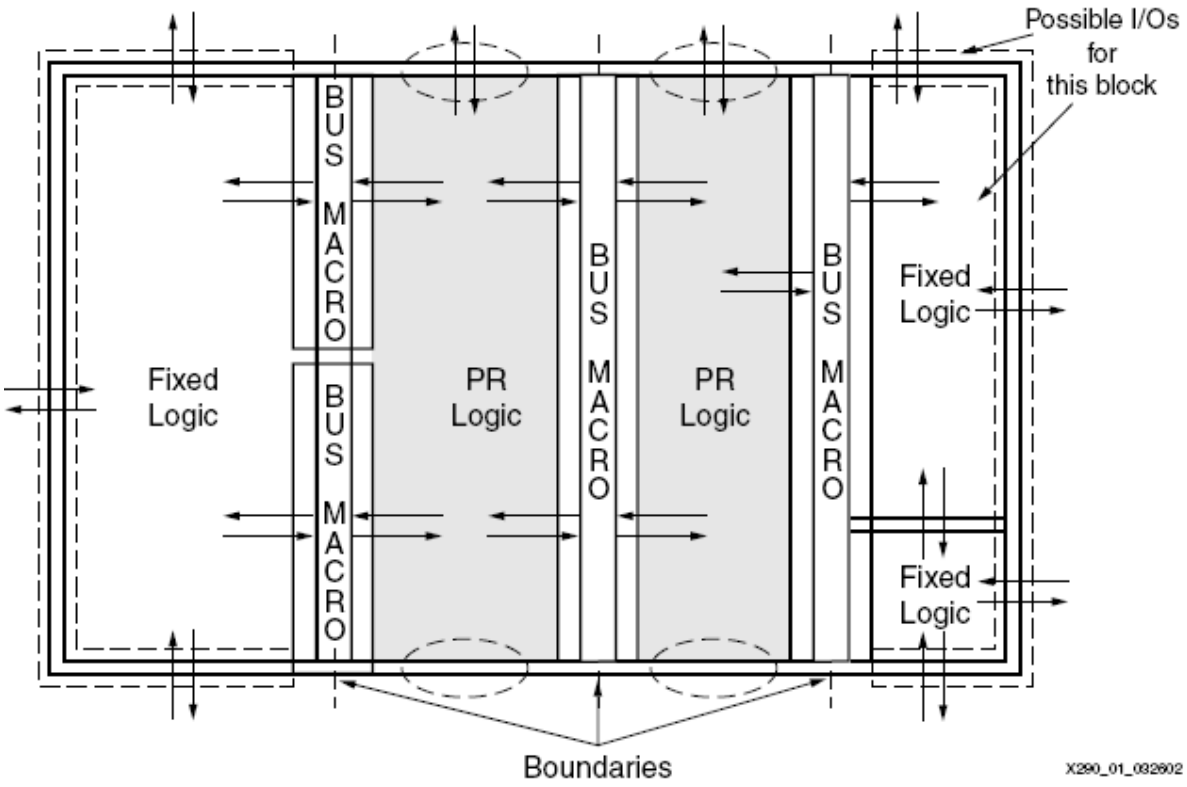


Fig. Design Layout with Two Reconfigurable Modules

2.1.2 Physically Aware HW-SW partitioning approach [Ref.3]

A physically aware hardware-software (HW-SW) scheme is presented here for minimizing application execution time under HW resource constraints, where the HW is a reconfigurable architecture with partial dynamic reconfiguration capability. Such architectures impose strict placement constraints that lead to implementation infeasibility of even optimal scheduling formulations that ignore the nature of these constraints. An exact and a heuristic formulation are proposed that simultaneously partition, schedule, and do linear placement of tasks on such

architectures. With the exact formulation, it is proved that the critical nature of placement constraints. We demonstrate that our heuristic generates high-quality schedules by comparing the results with the exact formulation for small tests and a popular, but placement unaware scheduling heuristic for larger tests.

This work makes several contributions:

- It demonstrates that existing approaches that do not consider physical task layout can result in unrealizable (infeasible) designs.
- It outlines an exact approach that incorporates physical layout.
- It presents a KLFM heuristic (Kernighan-Lin / Fiduccia Matheyses) incorporating detailed linear placement that generates good results on a large set of benchmarks.
- It shows applicability of our work to heterogeneous architectures. Modern FPGAs have heterogeneous architectures containing columns of dedicated resources like embedded multipliers, embedded memory blocks. Usage of such specialized resources usually leads to more area-efficient and faster implementations.

There is a task graph with n tasks, where each task has multiple possible implementations. Each HW implementation of a task occupies a certain number of columns. We have one available SW processor, and a HW resource constraint of m HW columns for application mapping. The objective was to find an optimal schedule where each task is bound to HW or SW, the

task implementation is fixed, and, for HW tasks, the physical task location is determined.

To understand the problem space and determine optimality, an Integer Linear Program is formulated. Then they used the concept of KLFM algorithm for scheduling the tasks in the task graph. The next step in the approach is the Earliest Starting Time (EST) computation. The goal was to find the earliest time slot when the task can be scheduled, subject to the various constraints. They proceeded first searching for the earliest instant when we can have a feasible task placement, i.e. enough adjacent columns are available for the task. Once they obtained a feasible placement, they tried to satisfy the other constraints. If the reconfiguration controller was available at the instant the space becomes available, then the reconfiguration component of the task can proceed immediately. Otherwise, the reconfiguration component of the task has to wait till the reconfiguration controller becomes free. Once the reconfiguration component is scheduled, it is checked if the execution component can be immediately scheduled subject to dependency constraints. The EST computation thus embeds the placement issues and resource constraints related to reconfiguration.

1.3 Direct & Merge Dynamic Reconfiguration [Ref.4]

Two methods for implementing modular reconfiguration in Virtex FPGAs are compared and contrasted. The first method is

the Direct Dynamic Reconfiguration which offers simplicity and fast reconfiguration times, but limits the geometry and connectivity of the system. The second method, developed recently is Merge Dynamic Reconfiguration which enables modules to be allocated arbitrary areas of the FPGA, bridging the gap between theory and reality and unlocking the latent potential of dynamic reconfiguration. The cost of this advancement is increased reconfiguration time.

In the direct dynamic reconfiguration process, reconfigurable modules are composed from complete frames of configuration memory. This implies that a module occupies the full height of the device, including the I/O at the top and bottom of the reconfiguration region. The module may be a variable number of CLB columns in width, and all logic and routing within the reconfiguration region are dedicated to the module. Using this scheme, a module may be replaced very simply by writing over the existing configuration for the frames that coincide with the module area, using a partial bitstream.

This is exactly the same method that is adopted by PlanAhead for the module based active partial reconfiguration. Hence this approach comprises with same drawbacks those have already been mentioned in earlier section. In order to come out of those loopholes another approach called Merge Dynamic Reconfiguration has been proposed.

The merge dynamic reconfiguration method was created in order to circumvent the limitations of direct reconfiguration, and exploit the glitchless reconfiguration property of Virtex FPGAs. A statically routed signal can pass through a reconfigured region unperturbed provided the configuration bits associated with the route persist in the new configuration. However, as the module designs are placed and routed independently from the static part of the design, the resources allocated to a static route could also be used in one or more module implementations. This is avoided through the use of reserved routing – within a module region, certain routing resources are always reserved for static routing and modules must avoid using any of these resources, even if unused by the static design. Routing congestion and delay are reduced by routing through module regions, and module regions can be contiguous. The second major innovation in merge reconfiguration is in the way the partial bitstream is loaded. Rather than writing the bitstream directly to the configuration memory, the current configuration is read back from the device and modified with information from the partial bitstream before being written back.

Chapter 3

3.1 Problem Definition

The focus of the current work was a case where we have a series of tasks to be executed on an FPGA device at different time instances. We assume that designs for individual tasks are such that some part of the design is common for all the tasks. In terms of the modules we can say that some modules remain static for all the time instances. But their position may not be the same if we do the floorplan for individual time instances. This would lead to total reconfiguration of the device at every time instance leading to high reconfiguration time and cost. But there is a scope of partial reconfiguration here which would not alter the position of those static modules, yet gives a satisfactory floorplan in terms of total wire length, net delay in every time instance. The time instances of the individual tasks are fixed beforehand which says that we can not reshuffle the tasks. So we need to deal with all the instance designs and find some positions for the static modules and rest of the space is occupied by other modules specific to instances (we call it as dynamic modules). So the problem sounds similar to a 3-D floorplan problem where the third dimension is the temporal axis.

3.2 Problem Statement

- Suppose there are **n** time instances, denoted by – $\mathbf{I_1, I_2, \ldots, I_n}$
- **K** modules are present in all the n-instances – they are called ***static modules*** $\{\mathbf{S_1, S_2, \ldots S_k}\}$.
- All the instances have some modules other than static ones – called ***dynamic modules*** $\{\mathbf{M_{11}, M_{12}, \ldots, M_{mn}}\}$**.**
- Static modules do not change their positions on the floor.
- Dynamic modules are to be replaced on the fly in each subsequent instances.

Goal is to achieve the following:

- There is a feasible floorplan for every instance. All the modules are accommodated on the board.
- Guarantees total hardware availability (CLB, BRAM, MULT) for all the modules in each of the instances.
- Maintaining module integrity, i.e. all the modules are contiguous.
- Guarantees total routability.
- Minimizes total net length for all instances.
- Minimizes delay for all instances.
- Minimizes reconfiguration cost.

Chapter 4

4.1 Approach

As of any physical design problem here also the first phase is partitioning phase where we partition the circuits of all the instances. After that comes floorplanning of a particular instance and mapping rest of the instance modules with that floorplan. In this section we will give an overview of the algorithm where as in the next section we will present the detail one.

I . Partitioning phase :

In this phase we take the netlist (hypernets) of all the instances as inputs and generate slicing tree for each instance by recursively bi-partitioning the circuits of all instances together.

During the bi-partitioning we move a module from one partition to another if it helps to improve the cut-cost of the bipartition. Moreover it should also satisfy the balance criteria to avoid all the modules coming into one partition.

The move of a dynamic module is simple and specific to an instance but that of a static module involves all the instances i.e. the module is moved from one partition to the other in all the instances.

So we obtain finally the partition tree for all the instances with static modules taking similar places for all the instances.

II . Instance Mapping Phase :

In this phase we first determine the instance which demands maximum resource requirements (in terms of CLB, RAM and MUL). We call it as *guiding instance*.

Next we map the nodes of partition trees for other instances with that of guiding instance. Static modules are holding similar places in all the trees.

III . Floorplan Generation Phase :

Now we use a fast floorplan generation algorithm (Ref 8) to allocate the position of the modules on the board. Floorplanning in FPGA is different from that in ASIC as we have CLB, RAM & Multiplier placed in FPGA. So traditional ASIC approaches do not suffice here. So we have used an existing floorplanning algorithm for FPGA to do the job [Ref 8].

At first the floorplan for the guiding instance is generated. Now without altering the positions of the static modules we place the dynamic modules in subsequent instances.

The dynamic modules of the other instances are accommodated in the space where the dynamic modules of the guiding instance resided.

Since the floorplan of guiding instance (demanding maximum resource) is already generated, other instances will demand lesser resources and must be well accommodated in the board-space used by guiding instance.

4.2 Detail Algorithm

I . Partitioning Algorithm :

Step 1 :

Mark all the modules with a Static or Dynamic tag to according to their natures.

Step 2 :

Take only the dynamic modules and put those into two partitions arbitrarily keeping in mind that the balanced criteria satisfy.

One of the method of doing this is

- Sort the dynamic modules of an instance in descending order of CLB requirement.
- Put the first one in list-1.
- Put next two modules in list-2 and next two in list-1.
- Repeat last step until all the modules are put into either of the lists.

Repeat it for all the instances.

Step 3 :

Put the static modules arbitrarily into the two partitions such that the balance condition is not violated. We can use the previous algorithm again for initial partition of static modules. The only difference being that we will put those in list-1 & list-2 of all the instances simultaneously.

Steps 4 :

For all instances, calculate the K-L gain of all the modules present. Maintain the gain-bucket structure proposed by Fiduccia-Matheyses [Ref.1,2] for every instance.

According to their gains put the modules in common gain-list. Module from left partition of an instance goes to left common gain list and that from right partition goes to right common gainlist.

For static modules among different values of gains obtained from different instances calculate the average of those and put it in the common gain list in the same way.

Step 5 :

Sort the common list by their gain values. Construct the gain bucket structure proposed by Fiduccia-Matheyses partitioning algorithm[Ref. 1,2].

Pick up the module from the bucket with highest gain and move it to the other partition.

Check the balance condition of that particular instance in which the moved chosen cell belong to. If it is a static module, check all the instances.

If balance condition satisfied, *Accept* the move and *Lock* the module. Otherwise Reject it. Check the next available module in terms of gain and repeat the above procedures.

Update the gains of the modules after the accepted move. Also update the gain buckets of the corresponding instance if the moduleis dynamic otherwise update that of all the instances. Update the gain bucket of the common list as well.

Step 6 :

Repeat step 5 until all the modules are locked or no more module can be chosen for the move due to balance criteria.

Step 7 :

Among all the moves taken fix that many number of moves from the beginning for which the partial sum of the moves becomes maximum.

Step 8 :

Repeat steps 3-7 recursively until all the leaf nodes of the slicing tree contain single module.

So we obtain finally the partition tree for all the instances with static modules taking similar places for all the instances.

II . Module Mapping Algorithm :

Step 1:

Get the slicing trees for all instances from the previous phase. Determine the maximum resource utilizing instance and set its slicing tree as *Guiding Tree*.

Step 2 :

Calculate the number of tiles required by each modules of each instance. Tiles are nothing but an assumed building block consisting of a fixed number of CLB, RAM, MULT.

Step 3:

In the guiding tree start from root node and match the root nodes of other instances. Then go top down fashion and match either node by node or node by subtree or subtree by node.

Step 4 :

Put a tag for all the nodes of the other slicing trees about the correspondence with the guiding tree obtained from the previous step.

III . Floorplan Generating Algorithm :

Step 1:

Use the floorplanning algorithm for FPGAs [Ref.8] to generate the floorplan of the guiding instance. In brief, the algorithm states that

- For each module, generate
  - a set of *irredundant (wi,hi)* pair (in terms of basic tiles)
- Post-order traversal of partition tree
  - Nodes are merged with respect to horizontal cut or vertical cut.
    - Vertically : V-list (wv,hv)
    - Horizontally: H-list (wh,hh)
  - Form combined list which is an irredundant implementations from V-list and H-list
    - M-list (wm,hm) :
- A set of irredundant implementations formed at root – Slicing tree
- Realization of slicing tree on FPGA
  - Allocation of Rectangular Region by calculating the co-ordinates of rectangular blocks.
  - Allocation of RAM/MUL Formulated as a Minimum Weighted Bi-partite Matching (MWBM)

Step 2:

Starting from root in a top down manner for other instance trees fix up the positions of static modules as they were in guiding instance floorplan.

Step 3:

For the rest of the floorplan get the correspondence tag obtained from previous phase and calculate the co-ordinates of rectangular regions again in top down manner.

Hence get the floorplan of other instance as well.

Chapter 5

5.1 Experimental Results

For the problem of partial reconfiguration there is no standard existing benchmark available. So we have developed an example circuit and tested it with the mentioned partitioning algorithm and with the partitioning tool hMetis which follows Fiduccia-Matheyses algorithm in a slightly modified form.

The format of our input netlist file would be like below

```
file format: test*.reconfig
================================================

<FPGA architecture>

.num_modules <Total no of modules over all the instances> <Number of
static modules>

.module <Module_id> <Module type : s (static) d (dynamic)> <CLB
requirement> <RAM requirement> < MUL requirement>
{repeat above line for all modules}

.num_instance <Total number of time instances>

-------------------------------------------------------------------------------

.instance <instance_id>

.module_list <list of module ids of this time instance>

.hnet :<hnet_wt>:<hnet member modules>
{repeat above line for all hnets for this particular instance}

-------------------------------------------------------------------------------
{repeat above block for the number of time instances present}
```

Example input netlist file :

Test1.reconfig

```
.num_modules 32 3
.module 0 d 400 5 5
.module 1 s 400 5 5
.module 2 s 400 5 5
.module 3 d 400 5 5
.module 4 d 400 5 5
.module 5 d 400 5 5
.module 6 d 400 5 5
.module 7 d 400 5 5
.module 8 d 400 5 5
.module 9 d 400 5 5
.module 10 d 400 5 5
.module 11 d 400 5 5
.module 12 d 400 5 5
.module 13 d 400 5 5
.module 14 d 400 5 5
.module 15 d 400 5 5
.module 16 d 480 6 6
.module 17 d 480 6 6
.module 18 d 480 6 6
.module 19 s 480 6 6
.module 20 d 800 10 10
.module 21 d 480 6 6
.module 22 d 400 5 5
.module 23 d 400 5 5
.module 24 d 800 10 10
.module 25 d 400 5 5
.module 26 d 400 5 5
.module 27 d 800 10 10
.module 28 d 960 12 12
.module 29 d 400 5 5
.module 30 d 400 5 5
.module 31 d 800 10 10
```

Contd..

```
.num_instance 2

.instance 0
.module_list 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
.hnet:1:13 4
.hnet:1:13 4 15 18 19 12
.hnet:1:15 18 19 12
.hnet:1:15 18
.hnet:1:19 12
.hnet:1:11 2
.hnet:1:13 15 11
.hnet:1:6 14
.hnet:1:4 18 12 2 6 14
.hnet:1:9 10
.hnet:1:7 1
.hnet:1:9 7
.hnet:1:8 0
.hnet:1:10 1 0
.hnet:1:5 3
.hnet:1:9 7 8 5
.hnet:1:17 16
.hnet:1:1 8 5 17

.instance 1
.module_list 20 1 2 21 22 23 24 25 26 27 28 29 30 31 19
.hnet:1:20 23
.hnet:1:20 23 21 19 24
.hnet:1:20 21 19 24
.hnet:1:20 21
.hnet:1:19 24
.hnet:1:22 2
.hnet:1:20 22
.hnet:1:24 25
.hnet:1:23 21 24 2 25
.hnet:1:26 27
.hnet:1:27 1
.hnet:1:27 1 31
.hnet:1:29 30
.hnet:1:26 27 30 31
.hnet:1:1 31 30 28
.hnet:1:29 30
```

After partitioning Instance-0 with hMetis that uses modified version of Fiduccia Matheyses algorithm, we get this partition.

Partitions with hMetis :

```
9
19
1
11
1
16
7
4
7
9
19
14
11
16
12
2
2
12
14
6
```

The above result will lead to the following set of leaf nodes in a partition tree.

B-(2,4)-(15,16)-B-7-B-19-(6,8)-B-(0,9)-B-(3,12)-(14,17)-B-(11,18)-B-(5,13)-B-B-(1,10)

For our calculation purpose we may eliminate blank partition and expand those which have more than one modules.

2-4-15-16-7-19-6-8-0-9-3-12-14-17-11-18-5-13-1-10

After partitioning with our partitioner we get the output in a tree manner where the nodes are the set of modules.

Partitions with our partitioner.

Set of nodes at each node of partition tree are shown below.

```
0:  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
-------------------------------------------------------------
1:  1 5 3 4 17 18 11 10 12 6
2:  2 8 14 9 15 7 16 0 13 19
-------------------------------------------------------------
3:  1 5 10 11 18
4:  3 6 12 4 17
5:  2 7 15 9 19
6:  8 14 16 0 13
-------------------------------------------------------------
7:  1 18
8:  10 11 5
9:  3 17
10: 4 6 12
11: 2 7 9
12: 15 19
13: 8 14 0
14: 16 13
-------------------------------------------------------------
15: 18
16: 1
17: 10 11
18: 5
19: 3
20: 17
21: 4 12
22: 6
23: 2 9
24: 7
25: 15
26: 19
27: 8 0
28: 14
29: 16
30:13
-------------------------------------------------------------
```

So one of the possible leaf node set will be as below

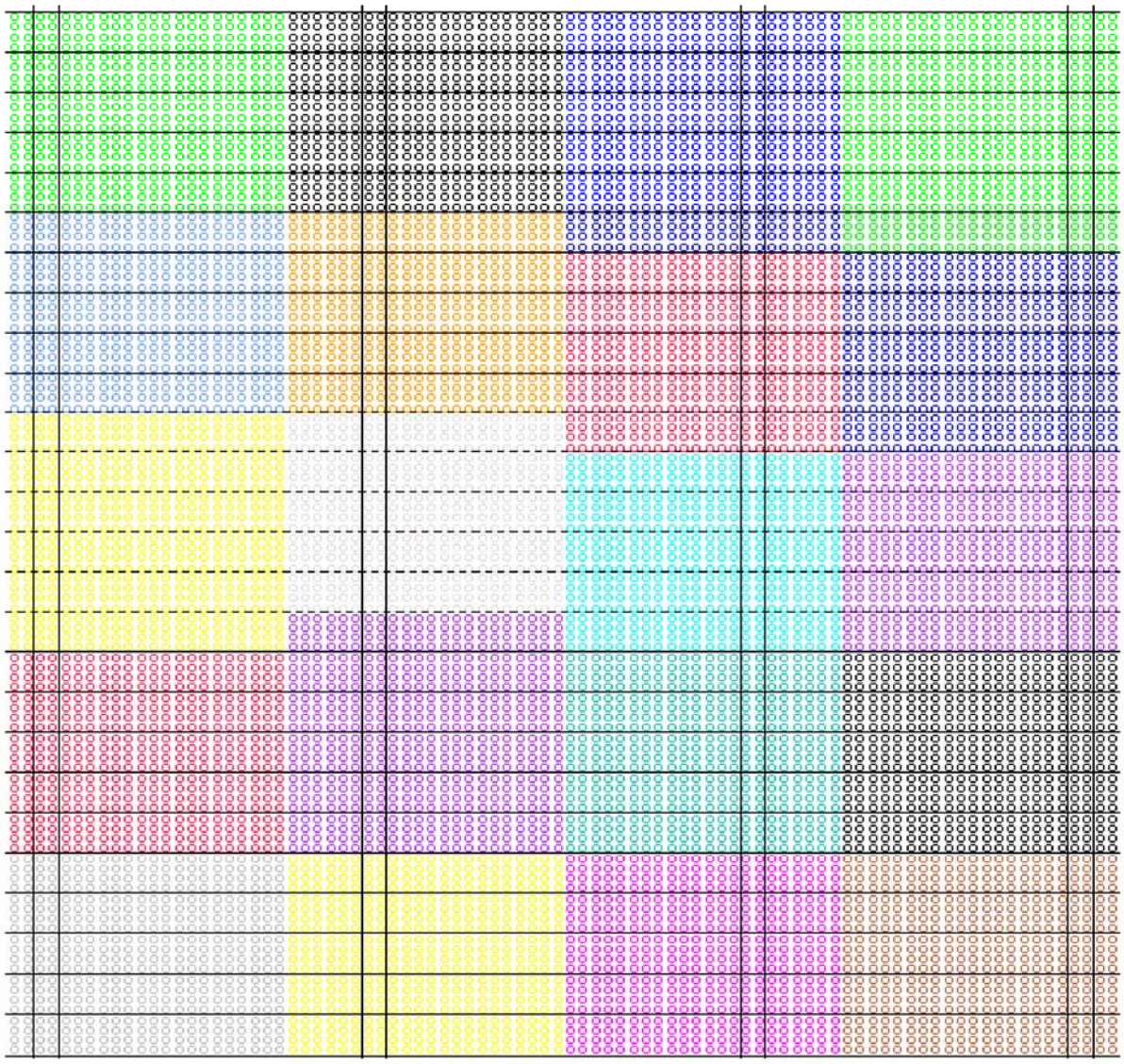
18-1-10-11-5-3-17-4-12-6-2-9-7-15-19-8-0-14-16-13

Now similar partitions are done in Instance-1 as well. After calculating the CLB requirement of both the instance, we found

#CLB in Instance-0 : 7920
#CLB in Instance-1 : 7440

So the Guiding Instance will be Instance-0 as it requires maximum no of CLBs.

So we run the floorplanning algorithm on the partition tree of Instance-0 and finally we get a bunch of floorplans out of which we select the below as the best in terms of integrity of modules.



Chapter 6

6.1 Implementation Details

While implementing the mentioned algorithm we used Xilinx Spartan3 device architecture.

The partitioning phase is written in scripting language – Perl.

To compare with a standard partitioner we used hMetis tool which is a modified version of Fiduccia Matheyses algorithm,

The floorplanner that we used was written in C language.

6.1.1 Data Structure

The main data structure that has been used looks like following. Most of the input information as well as the generated data are stored in this structure. The structure basically a hash defined in Perl language.

```
@module[ ] {
                Inst_id => <-1 for static module, otherwise it is
                           equal to the instance number where
                           this module is present>
                Module_type => <static or dynamic>
                Clb_req => <CLB requirement of module>
                RAM_req => <RAM requirement of module>
                MULT_req => <MULT requirement of module>
                @pin_counter => <pin count in different instances>
                @hnet_member => <this module present in these
                                 hnets>
```

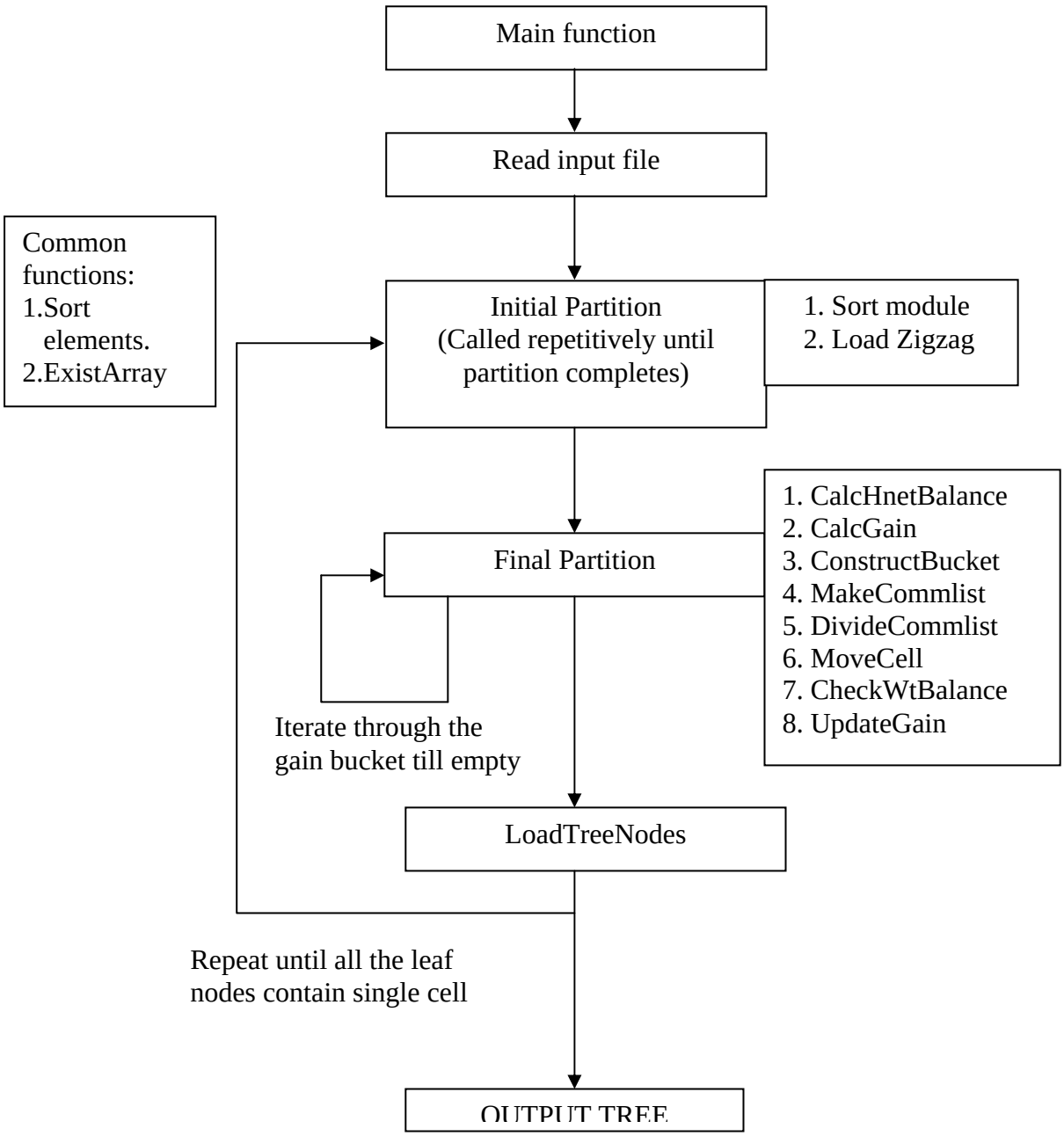
```
                    Partition_side => <side in which this module belong
                                      after partition>
}


@instance[ ] {
            Hnet_size => <no of hnets in this instance>
            %Hnet0 =>  {
                              Hnet_weight => <wt of hnet>
                              @Hnet_edge => <hnet modules>
                              l_index => <no of hnet modules in left
                                         partition>
                              r_index => <no of hnet modules in right
                                         partition>
                              }
                    .
                    .
            %Hnet<n-1> => <field keys are same for all>
            @Module_list => <list of modules present in this instance>
            %treenode => {
                        <node_id> => <set of modules in that
                                     particular node>
                        }
            %gain => {
                        <module_id> => <gain of the module>
                        }
            @Arr1 => <list 1 for using in the time of partitioning>
            @Arr2 => <list 2 for using in the time of partitioning>
            %bucket => {
                        <gain_value> => <list of modules having that
                                        gain value>
                        }


%commhash = {

            %hash1 => {one the list of common gain bucket}
            %hash2 => {other list of common gain bucket}
            }
```

6.1.2 Flow of Function Calls for the Partitioner



Chapter 7

7.1 Conclusion

The problem of Partial Reconfiguration is a tricky one to handle. The focus of the dissertation was firstly to judge what the present situation is, in terms of checking existing CAD tools in the domain. If any tools presently support this methodology or not, even if it supports to what extent. After that survey work the attempt was to build up a new algorithm which would do away with the drawbacks that existing CAD tools or algorithms have and to provide more flexibility to the user.

The work that has been done is successfully partitioning the circuits and offering fairly good result as compared to others. Although the implementation may be a bit tricky the algorithm is simple enough and uses some ideas of Fiduccia Matheyses algorithm. But it has been extended to this 3D floorplanning problem successfully. Again the floorplan generating module that has been used is quite elegant and gives fairly good results.

References

[1] C. M. Fiduccia, R. M. Mattheyes, ”A Linear-time heuristic for improving network partitions”, DAC, 1982.

[2] B Kernighan, S Lin, ”An efficient heuristic procedure for partitioning graphs”, The Bell System Technical Journal, V-29, 1970.

[3] S Banerjee, E Bozorgzadeh, N Dutt, ”HW-SW partitioning for architectures with partial dynamic reconfiguration”, Technical Report CECS-TR-05-02, UC Irvine.

[4] P. Sedcole, B. Blodget, T. Becker, J. Anderson and P. Lysaght, “Modular dynamic reconfiguration in Virtex FPGAs”, IEE Proc.-Comput. Digit. Tech., Vol. 153, No. 3, May 2006

[5] Xilinx Inc. ‘Virtex-4 user guide’, UG 290, v1.2, 2005

[6] Cindy Kao, Xilinx, Inc “Benefits of Partial Reconfiguration”, 2005.

[7] Nij Dorairaj, Xilinx, Inc. “PlanAhead Software as a Platform for Partial Reconfiguration”, 2005.

[8] Pritha Banerjee, Susmita Sur-Kolay & Arijit Bishnu, “Floorplanning in Modern FPGAs", in proc International Conference on VLSI Design 2007.

[9] Xilinx Inc : “Development System Reference Guide”.

[10] Xilinx Inc : “PlanAhead Methodology Guide”.

[11] Love Singhal and Elaheh Bozorgzadeh, “Multi-layer Floorplanning on a Sequence of Reconfigurable Designs”,

[12] Xilinx Inc : “Two Flows for Partial Reconfiguration: Module Based or Difference Based” , 2004.

[13] “Basics of FPGA”, http://www.wikipedia.org/.

[14] George Karypis and Vipin Kumar, “hMETIS - A Hypergraph Partitioning Package Version 1.5.3 Manual”, 1998.