M.Tech.(Computer Science) Dissertation Series

Multi-layer floorplanning for partial reconfiguration of FPGA devices

A dissertation submitted in partial fulfillment of the requirements for the M.Tech.(Computer Science) degree of the Indian Statistical Institute

Under the supervision of

Prof. Susmita Sur-Kolay

By

Chiranjit Chakraborty

Roll No : CS0703

Indian Statistical Institute

203, B.T. Road

Kolkata-700108

Indian Statistical Institute

Certificate of Approval

This is to certify that the thesis entitled **Multi-layer floorplanning for partial reconfiguration of FPGA devices** submitted by Chiranjit Chakraborty towards partial fulfillment for the degree of M. Tech. in Computer Science at Indian Statistical Institute, Kolkata, embodies the work done under my supervision.

The thesis is a faithfully record of bonafide research work carried out by Chiranjit Chakraborty under my supervision and guidance. It is further certified that no part of this thesis has been submitted to any other university or institute for the award of any degree or diploma.

Prof. Susmita Sur-Kolay
ACMU, ISI, Kolkata

Date:

Countersigned
(External Examiner)
Date:

Acknowledgment

It is with great reverence that I wish to express my deep sense of gratitude to my guide Dr. Susmita Sur-Kolay, Professor, Advanced Computing and Microelectronics Unit, Indian Statistical Institute, Kolkata, for her valuable guidance, suggestions and encouragement to carry out the work and the meticulous care with which she helped me throughout preparation of the report. I have benefited a great deal because of her deep insight. She not only suggested the problem but also helped me in all aspects including the preparation of this manuscript. This work has been possible only because of her continuous suggestions, inspiration, motivation and full freedom given to me to incorporate my ideas.

I express my sincere thanks to Pritha Banerjee (Research Scholar), Advanced Computing and Microelectronics Unit, for her valuable inspiration and help throughout this work. I am also very thankful to all the teachers for the valuable feedback. I also express my thanks to my friends and my classmates for their help.

I am short of adequate words in expressing my heartiest thanks to my parents who are the constant of encouragement to me. It is the loving care and understanding of them, which has placed me at the present level of academic career.

Abstract

Modern Field Programmable Gate Arrays (FPGA) with heterogeneous resources with millions of gates, have been widely used for prototyping large design nowadays. However, large designs might not fit in one FPGA chip. Since, all the modules of a given application might not be active at the same time, the FPGA resources may remain unutilized during the execution of the application. In such applications partial reconfigurability of FPGA helps, where a part of the FPGA chip remains active and inactive part of FPGA could be replaced by another set of modules. Given a schedule of instances with each instance having a set of active modules and their connectivity, a global floorplanning method is essential to reduce the partial reconfiguration overhead while optimizing the performance of the design. This can be done by fixing the position and shapes of common modules across all instances at the same location, while the rest of the temporary modules can be swapped in and out of the board. This is called reconfiguration. Modern FPGAs have different types of resources like CLBs, RAMs and Multipliers. This heterogeneity in resources makes floorplanning in FPGA difficult, especially when the design to be implemented is large. In this dissertation we propose a simulated annealing based multi-layer floorplanning to obtain the fixed positions for the common modules across all instances such that resource requirement of rest of the modules are still satisfied and the total cost of the floorplan is minimized.

Contents

1 Introduction **1**

1.1 Motivation of Partial Reconfiguration . . . 1

1.2 Scope and benefits of Partial Reconfiguration . . . 2

1.3 Applications of Partially Reconfigurable FPGAs . . . 3

1.4 FPGA Physical Design Cycle . . . 4

1.5 Scope of this thesis . . . 5

2 Partial Reconfiguration on FPGA **6**

2.1 Target Architecture . . . 7

2.2 Existing Approaches . . . 7

3 Proposed Algorithm Design **9**

3.1 Definition . . . 9

3.2 Problem Definition . . . 10

3.3 Sequence pair Representation of floorplan . . . 11

3.4 Proposed Algorithm . . . 12

3.4.1 Phase I: Initialization of data structure . . . 12

3.4.2 Phase II: Generation of initial Floorplan . . . 14

3.4.3 Phase III: Improvement by probabilistic algorithm . . . 19

3.5 Remark . . . 21

3.6 Some achievements . . . 22

4 Implementation details **23**

4.1 Different Moves design . . . 23

4.1.1 Whitespace allocation move . . . 26

4.1.2 Block matching for multi-layer floorplanning . . . 27

4.2 Cost calculation . . . 28

4.2.1 Area . . . 28

4.2.2 Aspect ratio 29
4.2.3 Wirelength 29
4.2.4 Resources scattered 30
4.3 Data Structure Design 30
4.3.1 Basic Data Structure 30
4.3.2 Primary Data Structure 31

5 Results **33**
5.1 An example 33
5.2 Result Comparison 37
5.3 Result Interpretation 41

6 Concluding Remarks and Future Work **42**

Chapter 1

Introduction

Field Programmable Gate Arrays (FPGAs) are programmable integrated circuits. It consists of array of *Configurable Logic Blocks* (CLBs) with wires of different lengths layed out in horizontal and vertical channels. CLBs consist of Look-up tables and flip flops which can be programmed to implement a design. A given digital design is implemented as a netlist of CLBs and the connectivity is implemented by connecting required wires on the vertical and horizontal channels with the help of switch boxes present at every cross point of horizontal and vertical channels. An FPGA is programmed using a configuration file which contains the place and route information to implement the design. For reprogramming an FPGA in order to implement a different application, all that is required is downloading of a new or different configuration file to the FPGA chip. Applications are often mapped to FPGAs using a four step process: design entry, technology mapping, physical placement and routing. Then a configuration file is downloaded to program the FPGA. Recent advancements in fabrication technology and device architecture have resulted in tremendous growth in FPGAs, both in terms of density and performance. Earlier FPGAs used to have only CLBs for mapping the logic but modern FPGAs have other resources also on the board like RAMs, Multipliers, DSPs, microprocessor cores along with array of CLBs [15] [16]. Heterogeneity in resources makes the mapping of logic on FPGA board more difficult, which requires additional steps in the mapping process.

1.1 Motivation of Partial Reconfiguration

The obvious benefit of FPGA is that the functionality on it can be changed and updated at some time in the future. The FPGA can be completely reprogrammed with new logic. For many users, this still is not enough. If one wants to change the logic within a part of an FPGA without disrupting the entire system, it can be done by partially reconfiguring the application on a device. Partial reconfiguration is a design process, which allows a limited, predefined portion of an FPGA to be reconfigured while the remainder of the device continues to operate. Using partial reconfiguration, the functionality of a single FPGA can be increased, allowing for fewer, smaller

devices than would otherwise be needed. Partial reconfiguration is useful for systems with multiple functions that can time-share the same FPGA device resources. In such systems, one section of the FPGA continues to operate, while other sections of the FPGA are disabled and reconfigured to provide new functionality. This is analogous to the situation where a microprocessor manages context switching between software processes. In the case of partial reconfiguration of an FPGA, however, it is the hardware, not the software that is being switched.

1.2 Scope and benefits of Partial Reconfiguration

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

As we can understand now that the partial reconfiguration is the ability to reconfigure selected areas of an FPGA anytime after its initial configuration. We can do this when 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 can gain the ability to:

1. adapt hardware algorithms;
2. share hardware between various applications;
3. increase resource utilization;
4. provide continuous hardware servicing;
5. 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 i.e. no down time. It also allows for multiple application in a single FPGA.

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

a. **The ability to change the 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.

b. **Hardware sharing -** As partial reconfiguration allows us to run multiple applications on a single FPGA, hardware sharing is possible to realize. Benefits include reduced device count, reduced power consumption, smaller boards, and overall lower costs.

c. **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.

1.3 Applications of Partially Reconfigurable FPGAs

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)* [14] Program by a significant amount.

Partial reconfiguration is the cornerstone for power-efficient, cost effective *software-defined radios (SDRs)* [14]. 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

Partial reconfiguration can also be used in many other applications. Another example is in mitigation and recovery from *single-event upsets (SEU)* [14]. 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. (Readback is the process of reading the internal configuration memory data to verify that current configuration data is correct.)

In the modern days FPGAs are not only consisting of *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 the FPGA is also widening up. As a matter of fact the need for reducing configuration time and cost with increasing efficiency, partial reconfiguration is the method that all FPGA designers need to concentrate.

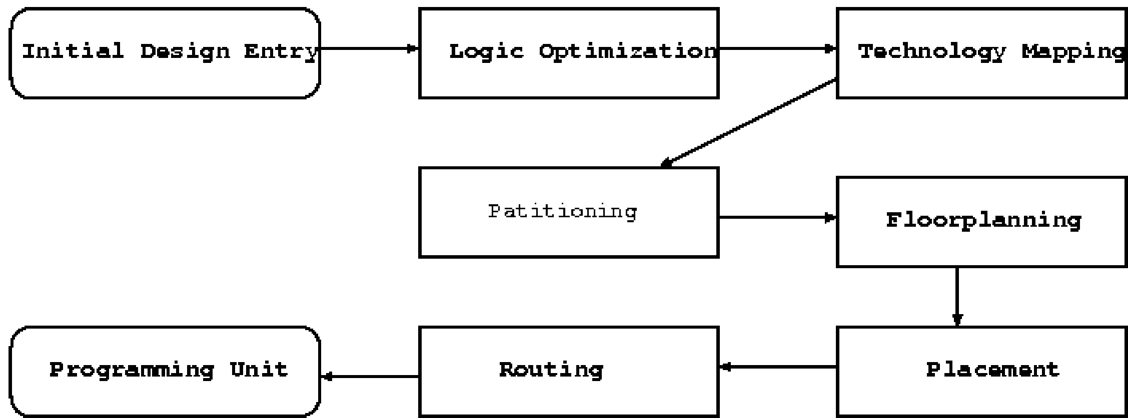

Figure 1.1: FPGA Design Cycle

1.4 FPGA Physical Design Cycle

First of all, the design to be programmed on FPGA is defined in terms of design equations in some high level language like VHDL. These design equations are mapped on to the resources available on the FPGA board as a netlist of logic blocks in technology mapping phase. The design is partitioned in components based on their connectivity and then the location of the components are determined on the FPGA board in placement phase. After placement, routing of the wires, which are connectivity between the components, is done by determining switch boxes through which the wire should go. Finally, the design is programmed on the FPGA board.

Earlier Floorplanning on FPGA was generally ignored in the physical design cycle [17] as the design were comparatively smaller and the resources on FPGA were homogeneous, namely CLBs. With the advent of technology, modern FPGAs are capable of implementing large designs with millions of gates. This has made floorplanning an important step in the design flow. As shown in Figure 1.1 the floorplanning is done prior to the placement and route phase in the FPGA physical design cycle. In the context of partial dynamic reconfiguration too, floorplanning has become an essential step. Formally, it is the process of determining the location of the modules on the chip such that no two modules overlap and there is enough space left to complete the interconnections. The input for floorplanning is a set of modules and a netlist describing the connectivity of the modules. At this stage the estimate for the required areas for different modules are available, but their exact dimension can vary in a range. As the result of floorplanning, we get a floorplan, which describes the exact location and size of each module on the chip.

1.5 Scope of this thesis

In this thesis, a probabilistic approach to the floorplanning of the partial reconfiguration of FPGA is proposed. This proposed design of the partial reconfiguration is called as multi-layer design. Different resources of FPGA chip is present in different layers. So, this name is given.

In Chapter 2, the floorplanning problem in the context of partial reconfiguration and the previous approaches are discussed. Chapter 3 describes the method proposed with exact problem definition. The detail implementation and data structure design is discussed in Chapter 4. Besides, some special functions design of the algorithm will be discussed in this chapter in details. The experimental results on a set of input data are given in Chapter 5. The method is also demonstrated with an example in this chapter. Concluding remarks and future work appear in Chapter 6.

Chapter 2

Partial Reconfiguration on FPGA

Recent Field Programmable Gate Array (FPGA) architectures like Xilinx' Virtex series allow partial dynamic reconfiguration. This means, inactive parts of a design implemented on FPGA hardware could be replaced by other designs while the remaining part of FPGA is still executing some other operations. So, partial reconfiguration helps executing a large application to be executed in the same piece of hardware by swapping in and out parts of the design, even if it does not fit completely on the same chip. Alternatively, a set of independent application can run on the same piece of FPGA hardware utilizing the FPGA resources effectively. This definitely, incurs an additional partial reconfiguration overhead each time a new part is swapped in and out of the FPGA hardware. Hence an appropriate scheduling of task/application/design is necessary to reduce the partial reconfiguration overhead such that common tasks/designs need not be swapped in and out again and again.

Moreover, at any instance of time, the tasks should be mapped onto the FPGA such that new tasks in the schedule can be fitted onto the board contiguously. It may be possible that, some tasks are already mapped on the board in such a way that, even though there are enough resources that satisfy the requirements of scheduled tasks at that instance, they are not contiguous. As Modern FPGAs are heterogeneous in nature with preplaced blocks like RAM, Multipliers along with sea of CLBs, the mapping of new tasks at any instance becomes more complex. The whole chip may have to be reconfigured which defeats the whole purpose of partial reconfigurability feature of FPGA. Even if it is possible to map all the active tasks at any instance of time on to the FPGA satisfying its resource requirement contiguously, does this mapping meet the required performance specification? In order to obtain a globally optimized floorplan of active tasks at different instance of time that also minimizes partial reconfiguration overhead, the floorplanning problem is defined in the context of partial reconfiguration.

2.1 Target Architecture

Partial reconfiguration is a design process, which allows a limited, predefined portion of an FPGA to be reconfigured while the remainder of the device continues to operate. Using partial reconfiguration, the functionality of a single FPGA can be increased, allowing for fewer, smaller devices than would otherwise be needed. Partial reconfiguration is useful for systems with multiple functions that can time-share the same FPGA device resources. In such systems, one section of the FPGA continuously work to operate, while other sections of the FPGA are disabled and reconfigured to provide new functionality. This is analogous to the situation where a microprocessor manages context switching between software processes. In the figure 2.1 it shows a Xilinx Spatran-3 FPGA where the CLBs are arranged in columns interleaved with columns of RAM-MUL pair at certain intervals. Each small square represents a CLB. A pair of shaded rectangular block spanning 4 rows of CLBs represents a pair of RAM and MUL. We use this architecture, though the described method is applicable on other architectures as well.

Definition 1 *Let W and H be the width and height of a target FPGA architecture, where the units are the width and height of a CLB respectively. A coordinate system $(0, 0, W, H)$ with top-left corner at $(0, 0)$ and bottom-right corner at (W, H) is assumed for the given chip.*

In the above figure , it is $(0, 0, 87, 103)$. Each resource on the architecture is identified by its coordinate position (x, y), where $0 \leq x \leq W$ and $0 \leq y \leq H$. Henceforth, the term target FPGA architecture and target chip will be used synonymously. So the problem of partial reconfiguration is to assign the co-ordinates for the position of all the required resources so that the common portion of the chip need not to be reconfigured. Here in the architecture diagram mainly CLBs, RAMS and Multipliers are shown. Our proposed algorithm can handle any kind of resources to be designed together.

2.2 Existing Approaches

There are only a few floorplanning approaches for FPGA. Singhal and Bozorgzadeh have introduced a new multi layer floorplanner in their paper [3]. They have introduced a new multi layer sequence pair representation based floorplanner in their paper which maximizes the overlap of common components of multiple designs. Thereby it is reducing reconfiguration overhead and guarantees a feasible floorplan with minimum area packing. They start with some initial floorplan topology and perform some perturbations like complement the cut lines or swapping of modules to get a new floorplan and this floorplan is accepted only if it is better than the previous floorplan, otherwise it is rejected with a probability which depends on the number of iterations done so far. In this way the initial floorplan plays an important role in this method. If the initial floorplan is not good then it may take large number of iterations to get the optimal floorplan.

Later, Singhal and Bozorgzadeh modified their work [4]. By the term multi layer in their previous work

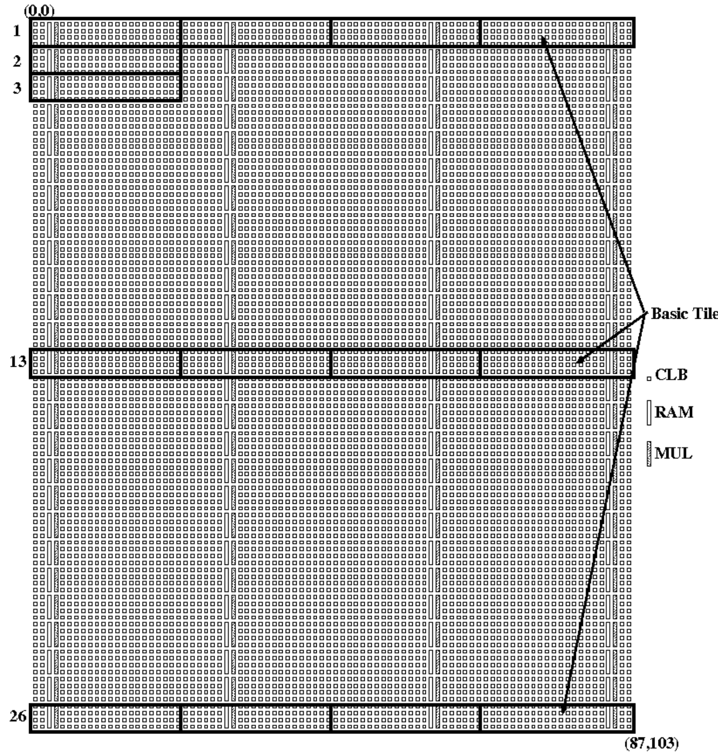

Figure 2.1: Spartan-3 XC3S5000 FPGA Architecture

[3], they don't mean that different resources should be in different layers. Here by multi layer they meant to say that their design can handle different design layer or instances. They introduced a multi-box floorplanner later [4] where different resources are really in different layers. But that design is for fully configuration. Our design is actually extension of that work in partial reconfiguration. Besides, we have used some approaches in initial design to make the running time little bit faster.

Last year Pritha Banerjee, Megha Sangtani and Susmita Sur-Kolay proposed a Floorplanning technique for Partial Reconfiguration in FPGAs [5]. In their work they have proposed a fast deterministic floorplanning method in the context of partial reconfiguration on FPGA with heterogeneous resources consisting of CLBs, RAMs and Multipliers. They have used a tile concept and whole problem is actually covering the floor with the tiles, as shown in Figure 2.1. To reduce the configuration overhead the static modules are placed in a fixed position in bottom left and top right corners of the board, while remaining contiguous space is used for placing the dynamic modules of the instances. Their method generates global slicing topology such that the exact physical location of each static module along with the shape, remains same across all instances. They have chosen the set of floorplans with minimum total semi perimeter wirelength over all instances. But the main problem is, the static modules are always placed in a fixed place. Plus their work is mainly to handle three kind of resources. But their proposed method is deterministic. So we have used their approach also to some extent to reduce the runtime.

Chapter 3

Proposed Algorithm Design

The partial dynamic reconfigurability of FPGAs is characterized by their ability to reconfigure subsets of their logic and routing resources at run time. This intrinsic dynamic reconfiguration results in accommodation of complex and dense designs. So, for designing the FPGA chip, we have to take care such that the advantages of the partial reconfiguration are taken properly. The goal of the proposed method is to design an efficient floorplan such that static modules occupy same position at every instance and still the performance is optimizes. Firstly, we will start with some definitions. Then the exact problem, that we are going to work in, will be explained. After that we will discuss the data representation and the details of the algorithms step by step.

3.1 Definition

Modern FPGAs are heterogeneous with different kinds of resources like CLBs, RAMs, Multipliers (MUL) etc., while earlier FPGAs used to have only CLBs.

The obvious benefit of FPGA is that the functionality on it can be changed and updated at some time in the future. The FPGA can be completely reprogrammed with new logic. For many users, this still is not enough. If one wants to change the logic within a part of an FPGA without disrupting the entire system, it can be done by partially reconfiguring the application on a device. Partial reconfiguration, as already mentioned, is a design process, which allows a limited, predefined portion of an FPGA to be reconfigured while the remainder of the device continues to operate. So before describing the algorithm of designing such kind of devices, we need to know some definitions.

Modules and Signal nets: Let $M = \{m_1, m_2, \ldots, m_n\}$ be a set of n distinct modules. Let $S = \{s_1, s_2, \ldots, s_q\}$ be a set of q signal nets. Each signal net $s_i \in S$ is associated with a set of distinct modules $M_{s_i} = \{m_j \mid m_j \in M\}$, and the set S is called a *netlist*. If $M_{s_i} = M_{s_j}$, then the two distinct signal nets s_i and s_j connect the same set of modules.

Resource Requirement Vector: For a mTodule m, a 3-tuple vector $R_m = (m_{clb}, m_{ram}, m_{mul})$ represents the number of CLBs, RAMs and MULs required by module m.

Instance: One FPGA chip can represent different logics, at different instant of time. Each of those designs is called Instance.

Static and Dynamic modules: Given a schedule of instances, modules which are common and remains active in all instances are called static modules. The rest of the modules which are swapped in and out of an instance, are called dynamic modules.

3.2 Problem Definition

Given a *target architecture* $(0, 0, W, H)$ with its resource locations, a design consisting of (a) a set of soft (flexible in shape) modules M, (b) the resource requirement vectors R_{m_i} for each $m_i \in M$, (c) different modules m_{I_i} present in each instance $I_i \in I$ of the FPGA design and (d) the connectivity information of the modules in the form of sequence pair S,
find a floorplan by assigning a connected region $(x_{min}, y_{min}, x_{max}, y_{max})$ to each module on the target architecture such that
(i) $0 \leq x_{min} \leq x_{max} \leq W$ and $0 \leq y_{min} \leq y_{max} \leq H$,
(ii) region for no two modules overlap with each other,
(iii) for each module m_i, the resources in its region satisfies R_{m_i}
(iv) a certain cost function is optimized.

A floorplan is said to be *feasible* if it satisfies all three conditions (i), (ii) and (iii). The cost function to be optimized is typically the wirelength [10, 11]for which the popular metric HPWL (half-perimeter wirelength), i.e.,the sum of the semi-perimeter of the bounding boxes for each net, is used. In the absence of information at this stage, the net terminals on a soft module are assumed to be at the center of the module. Otherwise, The pin location of each interconnect is fixed at the boundary of the block. This bounding box cost has also been extensively used as a FPGA placement metric [9]. The problem formulation as stated above is a generalization of that given in [8, 12] and as such is *NP*-hard. Like most of the prior works on FPGA heterogeneous floorplanning [8, 13], we also consider HPWL as the objective function. In our work, we have used some other factors as well for calculating the cost. We have discussed those things in details in the chapter 4.

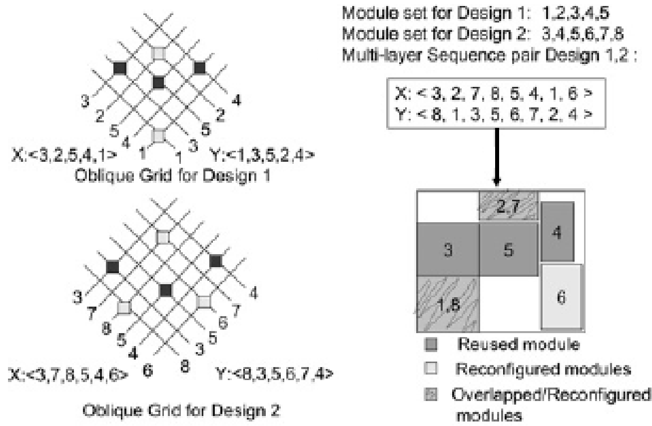

Figure 3.1: Sequence Pair

3.3 Sequence pair Representation of floorplan

A sequence pair is a pair of sequences of n elements representing two permutations of n modules. The two permutations capture geometric relations between each pair of modules. Every two modules constrain each other in either vertical or horizontal direction. The following relationship holds for sequence pairs

xArray = ($\langle .... a, ..., b, ...\rangle$, $\langle .... a, ..., b, ...\rangle$)

$\Longrightarrow a$ is to the left of b

yArray = ($\langle .... a, ..., b, ...\rangle$, $\langle .... b, ..., a, ...\rangle$)

$\Longrightarrow a$ is above b

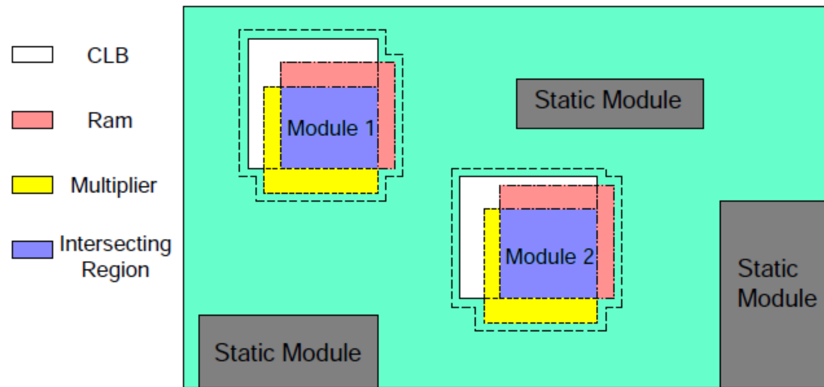

Figure 3.2: Reconfigurable static and dynamic module

3.4 Proposed Algorithm

The proposed work has different steps which are going to be explained step by step. We have used the multi-layer concept. In the multi-layer concept, different resources (eg. multiplier, CLB, RAM etc) can be in different layers. (as shown in the Figure 3.2) So we can consider that the layers are present one by one above another layer. So the bounding box of one resource can have the intersection with the same of another resource, as they are in different layers. Now the algorithm of the design is explained below. For simplicity, the whole algorithm is subdivided into three parts and explained step by step.

Input: Number of Modules, Number of Instances, Number of Resources, sizes of resources, floor height, floor width, resource requirement vectors per module, Indices of the modules per instance, xAarry, yArray.
Output: The positions of resources for each module in each instance. If a particular module is not present in particular instance, that is represented properly. The common modules (for all its resources) should be present in same location for all the instances.

Algorithm Steps:

Step: 1 The required design is given as the file input to the algorithm. In the first step the data is read from the input file and stored in the proper data structure. (Data structure design will be discussed later in details)
$noMod$ = Number of Modules
$noInst$ = Number of Instances
$noRes$ = Number of Resources The different modules required in different instances are stored.

Step: 2 For the given requirement, if it is really possible to give a design or not, is checked.

(a) Total area required for each instance is calculated.
Then checked if for all instances i ($1 \leq i \leq noInst$), the area required is less than the total floor area or not.
If not it sends the error message, else continue.

3.4.1 Phase I: Initialization of data structure

Step: 3 The common modules are determined and stored separately. For doing that, following steps are taken.

(a) Firstly the common modules of the first two instances are determined. Then iteratively the other instances are considered one by one and common modules of all the instances are found.

Step: 4 Required relative position of the given modules in different instances are read from another input file. This file is actually generated randomly. But for making the design efficient, this file is written in such a

way, that the aspect ratio of the bounding box of over all design is possible to keep near 1:1. The position is given in the form of Sequence Pair. It's referred as multiple sequence pair.

(a) From the multiple sequence pair, individual sequence pairs for all the instances are calculated. So the relative positions of all the modules in different instances are determined.

Step: 5 From the sequence pair, a matrix named position matrix is constructed. It's a n x n matrix, where n is the number of modules.

The matrix $Position[i][j] =$

1, if 'i'th module is to the left of 'j'th module
-1, if 'i'th module is to the right of 'j'th module
2, if 'i'th module is to the above of 'j'th module
-2, if 'i'th module is to the below of 'j'th module

So, $Position[i][j] = -Position[j][i]$
At each step of the design, it's checked if the Position matrix is changed or not. Here the design is considered to be in 4th quadrant. If it's in the 1st quadrant, then just the above and below relation should be interchanged.

Step: 6 Initial design is then done. Mainly the data structure construction and initialization of the bottom left and top right corner co-ordinates to -1 are done here.

```
for(i=0;i<noInst;i++)Figure
      for(j=1;j<=noMod;j++)
            for(k=0;k<noRes;k++)
                  Design[i][j].BlockPlacement[k].topRightx <--- -1;
                  Design[i][j].BlockPlacement[k].topRighty <--- -1;
                  Design[i][j].BlockPlacement[k].bottomLeftx <--- -1;
                  Design[i][j].BlockPlacement[k].bottomLefty <--- -1;
            end for
      end for
end for
```

Design is the main data structure for the design. *BlockPlacement* is the data structure for storing the two end co-ordinates of the blocks. Data structure design is explained in chapter 4 in details, the above pseudo code will be much clearer then.

Step: 7 The multiplier and rams are present in the form of strips. The distance between 2 strips of rams and multiplier are given as the input design requirement. The positions of the strips are then calculated.

3.4.2 Phase II: Generation of initial Floorplan

Placing the common modules

Step: 8 After that a feasible starting design is taken to start the design. To improve the efficiency of the algorithm, different random moves are used in the initial design also. At the very first, the locations of the common modules are determined.

(a) The data structure for storing the common modules are stored in a separate array.

(b) The common modules are sorted according to the position in the sequence pair Xarray.

(c) In the sorted list the modules which will come later will be always right or below the existing modules.

(d) All the common or static modules should be present together to reduce the cost. For starting the design choose any position in the floor. For example, we can take a position which is almost at the middle of the floor to be designed.

(e) For placing the first module, set the bottom left co-ordinate as the position chosen in the previous step for all the resources i, $1 \le i \le noRes$.

(f) Calculate the *width*s and *height*s of all the resources i, $1 \le i \le noRes$. Few things we need to take into mind in time of calculating the *width*s and *height*s.

   i. The heights and widths should be as equal as possible. The design is efficient if the blocks are almost square.

   ii. For some resources, heights are more compared to width for each unit. For example, for 1 unit ram, height is 4 column, width 1 column. So, for such resources, height should be multiple of 4.

   iii. For some resources, like ram and multiplier, the elements are present in the form of strips. So the elements are not present continuously. So the width should be increased accordingly.

(g) From the *width*s and *height*s of all the resources i, $1 \le i \le noRes$, the top right co-ordinates of all the resources are calculated just by adding the *width* and *height* values.

(h) One separate data structure is maintained to keep track of the the portion of the floor already been designed.

(i) If the design goes beyond the floor area, shift the starting position toward the above or left direction and re-adjust the co-ordinates.

(j) After one module is designed, consider the next module. Check the position of the new module with respect to the already designed modules consulting the *Position* matrix.

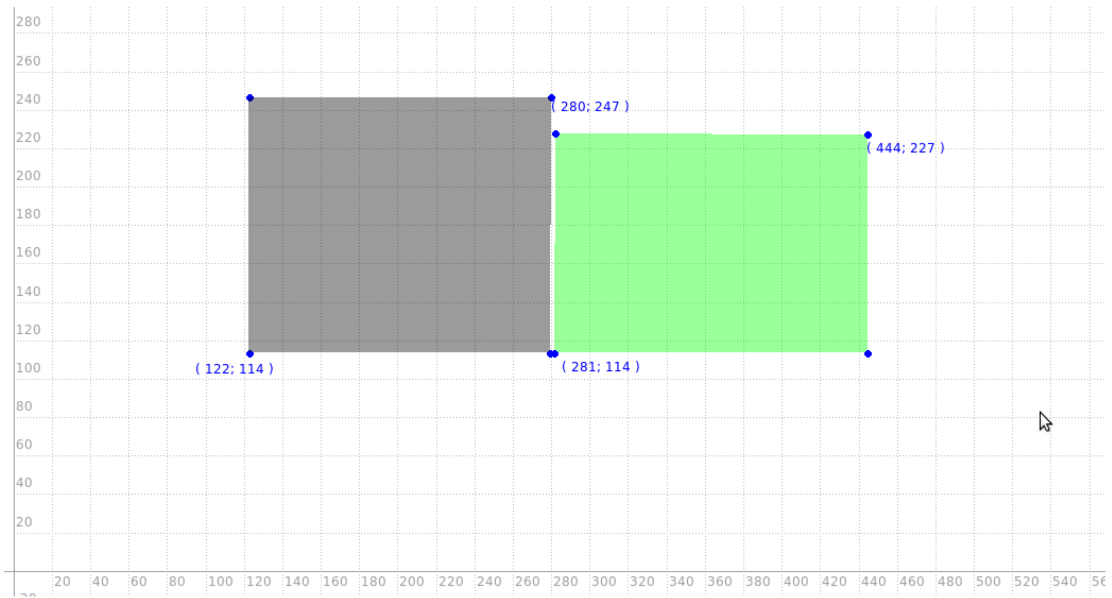

Figure 3.3: New module to the right of the existing module

(k) Similarly calculate the *width* and *height* of the current module. It's done in the similar manner satisfying some conditions (which was used to design the 1st module).

(l) If the new module is to the right of the existing modules (Figure 3.3), adjust the co-ordinates of the new module in the following manner,

```
for(i=0;i<noInst;i++)
      for(k=0;k<noRes;k++)
              Design[i][common[l]].BlockPlacement[k].topRightx
                            <--- status[k].topRightx + resourceWidth[k];
/* the width of the module is added to the top right x co-ordinate of
status to find its x span */

              Design[i][common[l]].BlockPlacement[k].topRighty
                            <--- status[k].bottomLefty + resourceHeight[k];
/* the height of the module is added to the bottom left y co-ordinate of
status to find its y span */

              Design[i][common[l]].BlockPlacement[k].bottomLeftx
                            <--- status[k].topRightx+1;
              Design[i][common[l]].BlockPlacement[k].bottomLefty
```

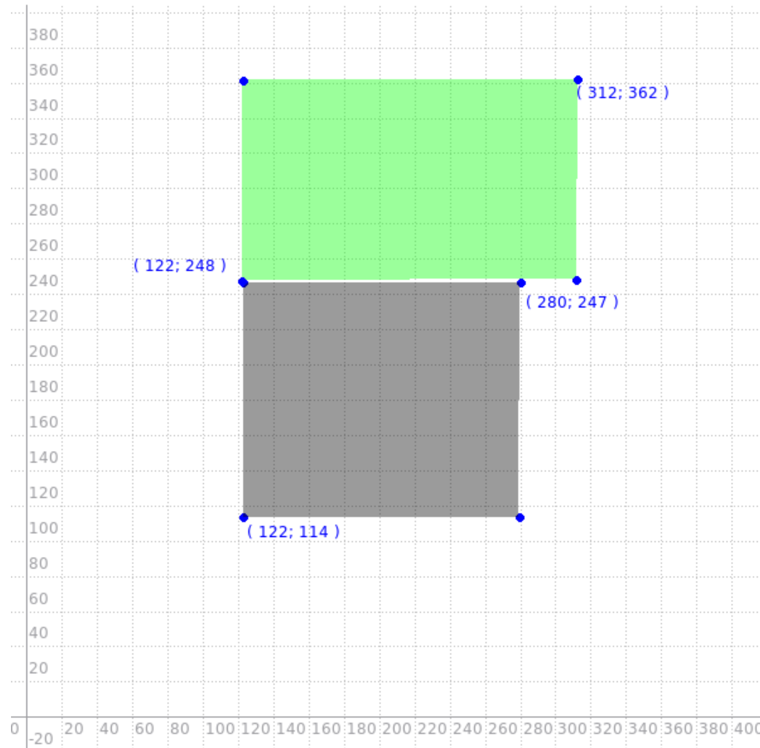

Figure 3.4: New module to the top of the existing module

```
                                   <--- status[k].bottomLefty;
/* the bottom left x co-ordinate is found by increasing the top right x co-ordinate
 of status by just 1. The bottom left y co-ordinate will be same as status */


      end for
end for
```

for all l, $1 \leq l \leq noCommon$, $noCommon$ = number of common modules.
Here the array common is storing the indices of the modules which are common in all the instances. *status* array is storing the bounding box already been designed.

(m) If the new module is to the below of the existing modules (if the design is in the 1st quadrant, then top of the existing module), adjust the co-ordinates (Figure 3.4) of the new module in the following manner,

```
for(i=0;i<noInst;i++)
      for(k=0;k<noRes;k++)
```

```
                Design[i][common[l]].BlockPlacement[k].topRightx
                              <--- status[k].bottomLeftx + resourceWidth[k];
/* the width of the module is added to the bottom left x co-ordinate of
status to find its x span */

                Design[i][common[l]].BlockPlacement[k].topRighty
                              <--- status[k].topRighty + resourceHeight[k];
/* the height of the module is added to the top right y co-ordinate of
status to find its y span */

                Design[i][common[l]].BlockPlacement[k].bottomLeftx
                              <--- status[k].bottomLeftx;
                Design[i][common[l]].BlockPlacement[k].bottomLefty
                              <--- status[k].topRighty+1;
/* the bottom left y co-ordinate is found by increasing the top right y co-ordinate
 of status by just 1. The bottom left x co-ordinate will be same as status */
      end for
end for
```

for all l, $1 \leq l \leq noCommon$, $noCommon$ = number of common modules.

(n) Throughout the procedure, the local data structure is maintained to keep track of the the portion of the floor already been designed.

(o) Repeat the steps (j), (k), (l), (m) and (n) till all the common module is designed.

(p) If the design goes beyond the floor area, shift the starting position toward the above or left direction and re-adjust the co-ordinates. If the shifting is not done here, the co-ordinates will be automatically re-adjusted after the moves which will be discussed below.

Placing the remaining modules

Step: 9 After the common module is designed, all the other modules will be designed for all the instances. Now, for each instance i ($1 \leq i \leq noInst$) do the following steps

(a) Now consider the per instant sequence pair. Again consider the Xarray modules in the sorted order. In the sorted list the modules which will come later will be always right or below the existing modules.

(b) Again another local data structure is maintained to keep track of the the portion of the floor already been designed. Initialize the local status data structure to be the origin.

(c) Consider the 1st module. If it's not one of the static modules, start designing the module.

(d) If the first module is one of the common module, then change the local status data structure to the status maintained in the common design. And then go to step (i). Not only for the 1st module, if at any stage common module is encountered, the local status is changed accordingly.
else consider the origin as the starting co-ordinate.

(e) For placing the first module, set the bottom left co-ordinate as the position chosen in the previous step for all the resources i, $1 \leq i \leq noRes$.

(f) Calculate the *width*s and *height*s of all the resources i, $1 \leq i \leq noRes$. We need to take care of some conditions as mentioned in $8(f)i$, $8(f)ii$ and $8(f)iii$ in time of calculating the *width*s and *height*s.

(g) From the *width*s and *height*s of all the resources i, $1 \leq i \leq noRes$, the top right co-ordinates of all the resources are calculated just by adding the *width* and *height* values.

(h) After the design is done, local status data structure is modified accordingly.

(i) For designing the next module, the relative position of the module with respect to the already designed modules are determined consulting the *Position* matrix.

(j) From the position matrix, consider the modules which are to the left of the current module. Similarly, consider the modules which are to the above of the current module.

(k) From the position of those existing modules, the starting candidate co-ordinates of the current module are found. That means, the x and y co-ordinate values are found, which satisfy the *Position* matrix values. These are the minimum x and y co-ordinate values to satisfy the conditions.

(l) After finding the candidate co-ordinates, different position around the location, which is given by the local status data structure, is considered.

(m) Now randomly choose any one from the positions which satisfies the candidate co-ordinate positions.

(n) From the chosen position, set the bottom left co-ordinates of the current module.

(o) Similarly calculate the *width* and *height* of the current module. It's done in the similar manner satisfying some conditions (which was used to design the 1st module).

(p) If the new module is to the right of the existing modules, adjust the co-ordinates of the new module in the following manner,

```
for(k=0;k<noRes;k++)

      Design[instIndex-1][modIndex].BlockPlacement[k].bottomLeftx
        <--- maxTempX+1;

      Design[instIndex-1][modIndex].BlockPlacement[k].bottomLefty
```

```
        <--- localStatus[k].bottomLefty + 1;
end for

for(k=0;k<noRes;k++)

      Design[instIndex-1][modIndex].BlockPlacement[k].topRightx
       <--- Design[instIndex-1][modIndex].BlockPlacement[k].bottomLeftx
       + resourceWidth[k];

      Design[instIndex-1][modIndex].BlockPlacement[k].topRighty
       <--- Design[instIndex-1][modIndex].BlockPlacement[k].bottomLefty
       + resourceHeight[k];
end for
```

Here *instIndex* is denoting the instance whose design is now considered. Similarly, *modIndex* denoting the module which is now being designed.

(q) If the new module is in any other position, the above code segment will not be changed. Just the bottom left co-ordinates will be changed. Basically, we follow the same algorithm as described in the section 8(l) and 8(m).

(r) Throughout the procedure, the local data structure is maintained to keep track of the the portion of the floor already been designed.

(s) Repeat the steps (i) to (r) till all the remaining modules are designed for this particular instance.

(t) At each step of the designing the modules, the position of the module is re-adjusted, to make the design much more compacted.

Step: 10 Using the step (9), all the modules in all the instances are designed. But, it's just the initial design. Now the design can be improved using the following steps.

3.4.3 Phase III: Improvement by probabilistic algorithm

Step: 11 The cost of the current design is calculated. (Details of the cost calculation is discussed later.)

Step: 12 Any module in a particular instant can be moved or not, is checked. For valid the following things need to be considered:

(i) After the movement, the resource bounding boxes should not have any intersection with the bounding boxes of same resources of other modules in the instance.

(ii) The relative positions of the modules should not be changed. That means, the position matrix formed from the sequence pair should not be modified after the movement.

For checking the first condition, following steps are taken.

(a) The end co-ordinates of the module is stored in temporary variable.

(b) The co-ordinates are calculated for the module after shifting the module for all the resources for a certain amount.

(c) The shifted position is then checked if it has any intersection with all the other modules in the instances for all the resources.
If there is no intersection, return "YES", else send "NO".

For checking the second condition, just the position matrix elements corresponding to that module is calculated and checked if it's different from the existing entries in the matrix.
If it's different, return "NO", else send "YES".

Step: 13 For the movements of the different blocks, the static blocks are considered first.

Step: 14 For each of the common module, it's checked if it can be shifted for all the instances. If it's "YES", for all the instances, then move the module up to that amount in all the instances. Else, the module is not moved.

Step: 15 For doing the movement, just the co-ordinate values are changed.

```
for(k=0;k<noRes;k++)

    Design[instance-1][module].BlockPlacement[k].topRightx -= smallAmount;
    Design[instance-1][module].BlockPlacement[k].topRighty;
    Design[instance-1][module].BlockPlacement[k].bottomLeftx -= smallAmount;
    Design[instance-1][module].BlockPlacement[k].bottomLefty;

end for
```

Step: 16 At each step cost is calculated to check if the movement is improving the cost or not. If it's not improving the cost, then that move is not taken.

Step: 17 Now any particular move is considered randomly with the proper probability distribution. (Details of different kind of moves will be discussed later.)

Step: 18 After the move is chosen, it's checked if that particular type of move can be taken or not.

```
If it can be done, then
        it's checked, if it's improving the cost or not.

        If the cost is improved,
                the move is taken.
        else
                the particular move is taken with certain probability.
                // It's done to avoid the effect of Hill climbing operation.

else
        discard that move
```

Step: 19 When the terminating condition is satisfied, (the terminating condition can be of different kind, for example certain number of moves are taken, certain number of moves are considered, no improvement in cost is found in certain number of moves last considered etc) no further move is taken.

Step: 20 If the design is within the required floor, return the message *success*. If it's not in the required floor even after reaching the terminating condition, send the message that *more place is required for the design.*

Step: 21 Whether it's success or not, it will always give the result for the position of the modules. If it's not success, it suggests the amount of extra space required. The resulting positions of the different modules of all the instances are the final result. The output is written in the output file with particular format.

3.5 Remark

The proposed design of the FPGA for partial reconfiguration is called as multi-layer design. Different resources of FPGA chip is present in different layers. So, this name is given. Here the concept of simulated annealing is used. But here different steps are not completely random. For example, the shifting moves are considered first. The shape changing moves are given less priority etc. These are done so that the design converges to the required design as fast as possible.

The algorithm will always give a correct design, as at each step the validity is explicitly checked. But whether the result is optimum or not, that's not possible to decide as it's after all a NP-hard problem. But the area required will be minimum compared to the previous deterministic algorithm as it's a multilayer design. The results of the algorithm is discussed in chapter 5 with some inputs. Besides, it has some other benefits what we have discussed below.

3.6 Some achievements

There are certain constrains which we have become successful to remove. The following design features of our algorithm can be written here.

1. It can handle any number of resources.
2. The position of the static modules is not fixed, it can be anywhere in the floor.
3. If the initial sequence pairs is not given, the algorithm will give the design for which the cost is optimal from this algorithm.
4. All the static modules can be placed together to improve the efficiency of the design.
5. The overall bounding boxes of the module need not to be rectangular, it can be rectilinear.
6. Different resources of the modules are in different layers. So the bounding boxes of two different resources can intersect each other. In this way, much compact design is possible.

Chapter 4

Implementation details

The efficiency of an algorithm is very much dependent on the implementation of the algorithm. Using the efficient data structure, the time complexity of an algorithm can be reduced to a significant extent. Our proposed algorithm is already discussed in previous chapter. The implementation of the algorithm specially cost calculation, different kinds of moves design and the data structure will be discussed here. We have already mentioned about the use of different kind of moves and cost calculation in our proposed algorithm. Now the detail discussion will be given here.

4.1 Different Moves design

This section explains different moves made to improve the design. Every traditional simulated annealing framework in a floorplanner has two kinds of moves. One kind of move is made on blocks like orientation and aspect ratio moves. By the general term block, we meant to say both resource and module. The other kind of move is made on data representation sequence pair like swapping positions of two modules. Our design for partial reconfiguration has some additional moves to handle reconfiguration aspect of the floorplanning and multiple designs. The moves on the blocks in the multi-layer floorplanning are of the following types:

1. Changing the orientation of the blocks (module wise),
2. Changing the aspect ratio of the blocks (both resource and module wise),
3. Changing the whitespace along the border of the blocks (resource wise),

The first two moves are standard floorplanning moves. These two moves can only be made if the blocks in the design have soft shapes. The orientation and shape of the blocks are changed during the simulated annealing iterations of the design as described above. But the moves are taken only if the blocks allow such moves. Otherwise, these two moves are not performed on the blocks. The third move on the blocks is the whitespace move.

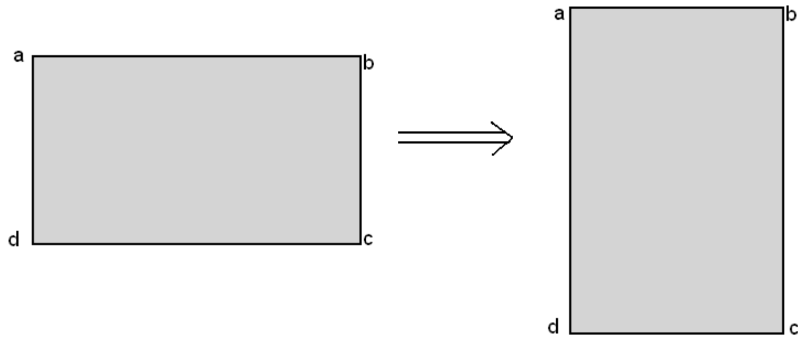

Figure 4.1: Changing the orientation of the blocks

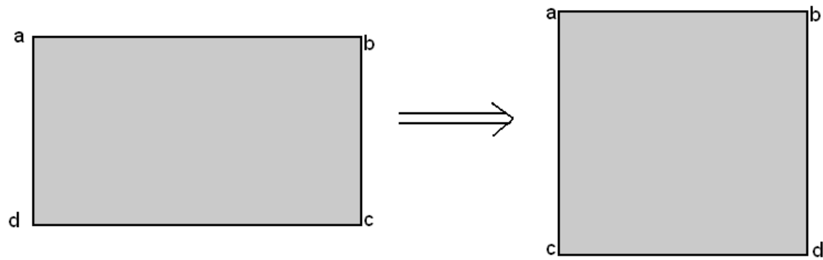

Figure 4.2: Changing the aspect ratio of the blocks

The moves on the data representation which in our case is multi-layer sequence pair, are similar to general sequence pair. But it can handle some additional moves for reconfiguration. These moves are applied to the multi-layer sequence pair to generate different floorplans for designs. The following moves are applied to any of the instance chosen at random.

1. **Compaction Move**: swapping two random modules of a layer corresponding to two ordinary nodes in the sequence,

2. **Shifting move**: moving a block in the left/right or top/bottom direction,

3. **Matching move**.

4. **Intra module shift**.

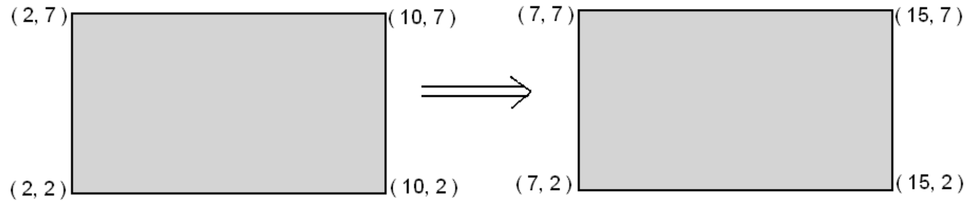

Figure 4.3: Shifting Move

The first two moves are standard moves in the simulated annealing based floorplanner. In the second move, a block is moved in the left or right direction by changing its relative position to other blocks in the multi-layer sequence pair, depending on the displacement slack available to the block. The sequence pair is modified by shifting the position of the targeted block in either or both sequence(s) of the multi-layer sequence pair to move the block in either left, right, top or bottom position. Shifting move is one of the most important moves and its contribution in compaction of the design is the most.

Another compaction move swaps the relative location of the module with another module by swapping the positions of the two modules in both the sequences of the sequence pair. Note that the compaction moves do not change the module placement directly. They first make changes to the sequence pair and then the changed sequence pair creates changes in the module placement. The direct changes to the modules placement cannot be preserved over the subsequent simulated annealing iterations. In that case, any change in the sequence pair in the next iteration will change the floorplan too drastically to preserve the previous moves about position of the blocks.

The matching move refers to the mapping of modules in one design to the modules in other designs. The module that are mapped to each other are the static module and are not reconfigured. The static modules are placed in same position in the initial design. The detail of the initial design is discussed above. In the subsequent stages, the static modules are moved together in all the instances. Details of different kind of matching is discussed later.
In intra-module shift, different resources in a same module are shifted to increase the intersection of different modules as shown in Figure 4.4.
In the next subsection, the concept of whitespace allocation move is discussed.

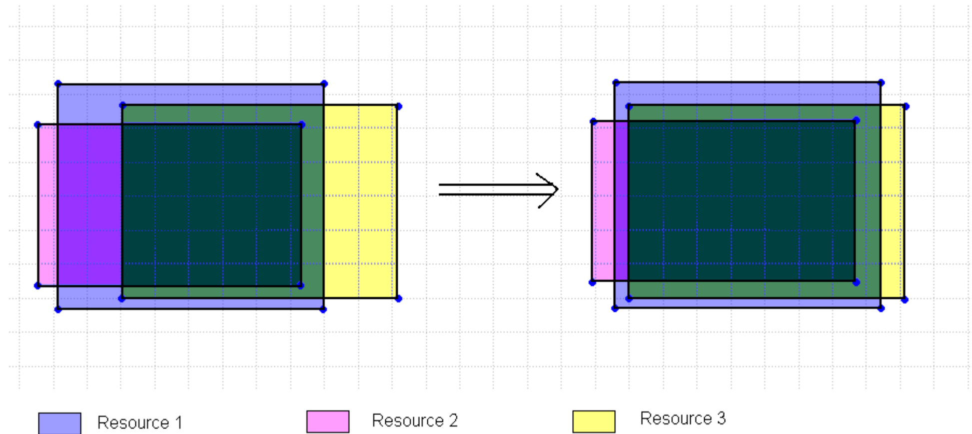

Figure 4.4: Intra-module shift

4.1.1 Whitespace allocation move

The Parquet tool uses sequence pair representation to make random moves of placement. In sequence pair representation, two permutations (orderings) of the blocks are maintained. The two permutations capture geometric relations between each pair of blocks. In sequence pair, all neighboring blocks are placed adjacent to each other. This representation automatically does compaction of the floorplan, as no whitespace is added and blocks are placed as close to each other as possible. This feature is good for finding minimum area floorplans. But, it is not desirable for finding congestion and modules-wire overlap free floorplans. Some extra whitespace is required in such floorplans for wires to pass through.

In order to add whitespace in the sequence pair representation, we add four new offsets to each block. The four new integral offsets, n, s, w and e, represent the whitespace region on all four sides of blocks. The total width of the block is therefore increased by w + e and total height is increased by n + s. The original block is placed inside this expanded block and the remaining space is occupied as whitespace. For this work, we only consider rectangular blocks for FPGA placements, although this technique can easily be extended for any rectilinear shape by keeping four or more offsets.

The sequence pair representation stores only the relative placement of blocks and is not affected by the size of the modules. The only change required for using the offsets is that while finding placement from sequence pairs, the sizes of the blocks are computed by adding the current offsets. That's why the size of each resources of each module is increased by certain amount.

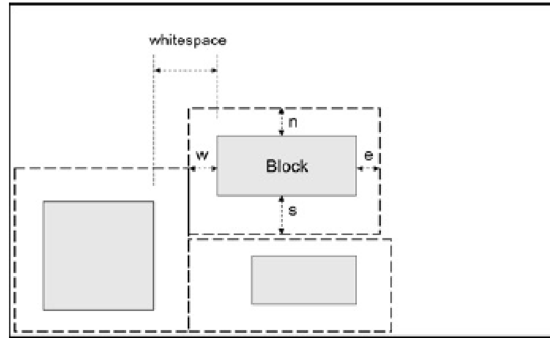

Figure 4.5: Whitespace allocation through offsets

4.1.2 Block matching for multi-layer floorplanning

It is possible that all the designs to be reconfigured containing one or more common blocks. Since, they are reused, these blocks will have same locations in all the designs. Implementing such case using multilayer sequence pair can be done by representing these blocks using a single block in the multi-layer sequence pair. The common block will have either left-right or top-bottom relationship with each block of each design. This will ensure that the common block obtains same coordinates in the floorplans of all the designs and is never reconfigured.

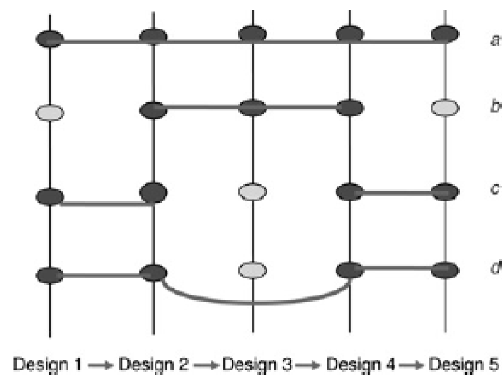

Figure 4.6: Different kind of matching in five instances

In the diagram different kind of moves are shown. Mainly the first kind of matching is handled here. Because that's the main concept of partial reconfiguration. But the design will be efficient if the other matchings are

also handled.So the other kind of matchings are also taken care of by some extent in our design. But if in some instance there are very number of modules, the design may not be efficient to handle the all kind of matching. So obviously there is a tradeoff in the design.

4.2 Cost calculation

The cost function of a floorplanner represents the objectives that the floorplanner has to achieve while finding a feasible floorplan of the design. The simulated annealing engine in the floorplanner tries to reduce the total cost of the system. The simulated annealing engine stores the current best floorplan solution. It then makes a move to the data representation (sequence pair) to generate a new floorplan. The change in the cost of each factor (such as area, and wirelength) is computed compared to the currently stored solution. The total change in cost function is compared to the stored solution to take the decision. The cost function is linear combination of different components. If the change in the cost is negative, the new solution is accepted. Otherwise, the new solution is accepted with certain probability based on the probability randomly chosen.

The total cost of our floorplan is

$$\text{Cost} = \alpha \times \text{Area} + \beta \times \text{Aspect Ratio} + \lambda \times \text{Wire Length} + \mu \times \text{Resources scattered}$$

The changes in costs of each variable are normalized to have a single range of values of each variable. The constants α, β, λ and μ are the scaling factors and represent respective weights of area, aspect ratio, wirelength and resources scattered . Their sum is equal to 1. In our experiments, the values of α, β, λ and μ are 0.3, 0.25, 0.25 and 0.2 respectively. Now, different components are discussed below.

4.2.1 Area

Area represents the normalized value of total change in area occupied by the k designs of new design compared to the stored solution. The new area of the floorplan is found by using the minimum bounding box which encompasses all the designs. It is then subtracted with the area of the stored solution. The difference is then divided by the area of the stored solution. This gives the percentage change in the area by the new floorplan compared to the current stored solution. The percentage change is then multiplied with a normalization constant to set this number comparable to other factors. The change in the area cost is negative if the area of the new floorplan is less than the current floorplan. Hence, by reducing Area cost, the tool tends to reduce the total area and increase the overlap in the designs.

4.2.2 Aspect ratio

The variable AR refers to the change in aspect ratio of the device, that is, the ratio of width and height of the new solution compared to the current stored solution. It is normalized to the values comparable to other factors in the total cost function.

$$\text{AR-cost} = ((\text{newAR - requiredAR})^2 \text{ - } (\text{currentAR - requiredAR})^2) + 0.0001 \times (\text{newAR - requiredAR})$$

The above equation [3] gives the function (AR-cost) to compute the change in cost of the aspect ratio. The aspect ratio function is computed as a penalty function. Similarly, the squared value of the difference between the currently stored aspect ratio and the required aspect ratio is computed. The required aspect ratio is found by finding the square root of the area required. That means, the required aspect ratio is that of a square. The two squared values are then subtracted from each other. In order to further add the penalty, the above computed difference is added to the difference between the current aspect ratio and required aspect ratio, multiplied by a normalizing constant (0.0001). This cost function calculates the penalty of the current floorplan in failing the area constraints and compares it with the penalty of the previously found good floorplan. If the current floorplan has a lower penalty than the previous floorplan, it has a higher probability of being chosen. The above computed function is then normalized with another constant to make it comparable with the other cost functions. This function makes sure that the designs fit in the device dimensions.

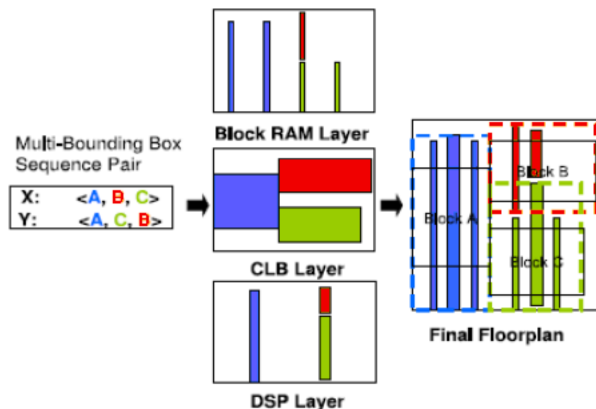

Figure 4.7: Multi Layer Heterogeneous Floorplanning

4.2.3 Wirelength

WL is the normalized value of the change in the total wirelength in each design. The total wirelength is measured by adding the wirelength of each interconnect in each design. The half-perimeter bounding box is used for each

interconnect. The new total wirelength is then subtracted with the total wirelength of currently stored solution. The difference is then divided by the wirelength of the stored solution to give the percentage change in the wirelength. The percentage change in wirelength is then normalized with a constant and added to the total cost function with the scaling weight. The scaling weights are used for tuning the relative importance of various cost functions. The wirelength minimization objective also helps in reducing the congestion and improving the clock period of the design.

4.2.4 Resources scattered

As we have discussed, different resources are present in different layers. So the design will be efficient if the intersection among the resources in different layers are maximum. To measure the intersection, the center of all the rectangular resources are found first. Then, for all the pairs, the distances are found and added. Then this sum is divided by the total numbers of pairs to find the representative distance. This new representative distance is then subtracted with the representative distance of currently stored solution. The difference is then divided by the distance of the stored solution to give the percentage change in the resources scattered. This percentage change is then added to the total cost function with the scaling weight.

4.3 Data Structure Design

For implementation of the algorithm, we have defined many structures. This is not the only data structure one can use to implement our algorithm. Even one can try to design more efficient data structure

4.3.1 Basic Data Structure

Let's start with the basic structures one by one and see how it's used.

```
typedef struct resource
{
        int height;
        int width;
}resource;
```

The above structure is maintained for each block used in the algorithm. The blocks are rectangle, used to represent different resources for each module in each instance.

```
typedef struct SP
{
        int xArray[MAX];
```

```
        int yArray[MAX];
        int ModNo_pINst;
}SP;
```

This data structures are used to store the sequence pairs sequence. xArray[MAX] and yArray[MAX] are used to store xArray and yArray as it's mentioned in the previous chapter. This data structure is maintained for each instance. From the given input sequence pairs, these sequence pairs are constructed for each instance. So, obviously, the next structure will be,

```
SP InstancePosition[MAX]; // maintained for each instance
```

So, each element in the above array stores SP for each instance. $ModNo_pINst$ stores the number of modules present in that instance.

4.3.2 Primary Data Structure

Now let's look at the data structure to store the whole design for all the modules with all the required resources in all the instances.

```
typedef struct Block
{
        int topRightx;
        int topRighty;
        int bottomLeftx;
        int bottomLefty;
}Block;



typedef struct Module
{
        Block BlockPlacement[MAX];
}Module;
```

The above data structure is the most important data structure used in this algorithm. *Block* structure is used to specify the co-ordinates of the bottom left and top right corners as the name suggests. $Module$ data structure is used to store information of each modules. Each module consist of several resources. So for each resource, the *Block* structure is maintained. That's why, here each element of the *BlockPlacement* array stores the information about each resource in that module. The *Block* data structure is also maintained to keep store the *status* data structure as mentioned in time of algorithm description the the previous chapter.

So obviously, the next structure will be,

```
Module **Design;
Block *status;
```

Design is actually a dynamic two dimensional array. It stores the whole design of the FPGA.

```
for(i=0;i<noInst;i++)
{
        for(j=1;j<=noMod;j++)
        {
                for(k=0;k<noRes;k++)
                {
                        cout << endl;
                        cout << Design[i][j].BlockPlacement[k].topRightx<<" ";
                        cout << Design[i][j].BlockPlacement[k].topRighty<<" ";
                        cout << endl;
                        cout << Design[i][j].BlockPlacement[k].bottomLeftx<<" ";
                        cout << Design[i][j].BlockPlacement[k].bottomLefty<<" ";
                        cout << endl;
                }
        }
}
```

$noMod$ = Number of Modules
$noInst$ = Number of Instances
$noRes$ = Number of Resources

The above code segment will actually print the design data. So from the above segment, the data structure *Design* can be understood.

Chapter 5

Results

The method described is implemented using C++ on linux platform. Before checking the algorithm with different inputs, let's start with an example. From this example we can get the idea how the algorithm is working.

5.1 An example

Let's see the design for a circuit where number of instances are 4. There are total 10 modules and 3 resources. The resource requirement for different modules are given below:

```
Module 1 = 10 20 480       Module 2 = 23 27 498       Module 3 = 23 26 489
Module 4 = 12 14 486       Module 5 = 24 18 490       Module 6 = 12 23 474
Module 7 = 20 11 498       Module 8 = 23 27 490       Module 9 = 11 25 499
Module 10 = 10 17 492
```

Design need to be done in 200×200 plane. The size of resource 1, 2 and 3 are respectively 4×1 , 4×1 and 1×1. Here each dimension in sizes are in terms of number of columns or rows.

Different instances are consist of different modules. The modules in different instances are given below:

```
Instance 1 = 1 2 3 7 9     Instance 2 = 2 4 7 9 10
Instance 3 = 4 5 6 7 8 9   Instance 4 = 3 4 7 9
```

The sequence pairs xArray and yArray are given as follows:

```
xArray = 6 2 3 4 5 1 7 8 9 10
yArray = 3 5 7 1 10 8 9 2 6 4
```

Results on Example

		Instance 1		Instance 2		Instance 3		Instance 4	
		Bottom left corner	Top right corner	Bottom left corner	Top right corner	Bottom left corner	Top right corner	Bottom left corner	Top right corner
Module 1	Resource 1	(58, 1)	(89, 17)						
	Resource 2	(58, 1)	(89, 17)						
	Resource 3	(58, 1)	(72, 14)	Not present		Not present		Not present	
Module 2	Resource 1	(0, 0)	(25, 12)	(0, 0)	(25, 12)				
	Resource 2	(0, 0)	(28, 12)	(0, 0)	(28, 12)				
	Resource 3	(0, 0)	(24, 22)	(0, 0)	(24, 22)	Not present		Not present	
Module 3	Resource 1	(29, 1)	(54, 13)					(0, 0)	(25, 12)
	Resource 2	(29, 1)	(57, 13)					(0, 0)	(28, 12)
	Resource 3	(29, 1)	(52, 23)	Not present		Not present		(0, 0)	(23, 22)
Module 4	Resource 1			(29, 1)	(51, 9)	(26, 1)	(48, 9)	(29, 1)	(51, 9)
	Resource 2			(29, 1)	(54, 9)	(26, 1)	(51, 9)	(29, 1)	(54, 9)
	Resource 3	Not present		(29, 1)	(52, 23)	(26, 1)	(49, 23)	(29, 1)	(52, 23)
Module 5	Resource 1					(52, 1)	(77, 13)		
	Resource 2					(52, 1)	(71, 13)		
	Resource 3	Not present		Not present		(52, 1)	(75, 23)	Not present	
Module 6	Resource 1					(0, 0)	(22, 8)		
	Resource 2					(0, 0)	(25, 12)		
	Resource 3	Not present		Not present		(0, 0)	(22, 22)	Not present	
Module 7	Resource 1	(0, 25)	(22, 37)	(0, 25)	(22, 37)	(0, 25)	(22, 37)	(0, 25)	(22, 37)
	Resource 2	(0, 25)	(19, 33)	(0, 25)	(19, 33)	(0, 25)	(19, 33)	(0, 25)	(19, 33)
	Resource 3	(0, 25)	(24, 47)	(0, 25)	(24, 47)	(0, 25)	(24, 47)	(0, 25)	(24, 47)
Module 8	Resource 1					(49, 26)	(74, 38)		
	Resource 2					(49, 26)	(77, 38)		
	Resource 3	Not present		Not present		(49, 26)	(72, 48)	Not present	
Module 9	Resource 1	(23, 25)	(41, 33)	(23, 25)	(41, 33)	(23, 25)	(41, 33)	(23, 25)	(41, 33)
	Resource 2	(20, 25)	(47, 37)	(20, 25)	(47, 37)	(20, 25)	(47, 37)	(20, 25)	(47, 37)
	Resource 3	(25, 25)	(48, 47)	(25, 25)	(48, 47)	(25, 25)	(48, 47)	(25, 25)	(48, 47)
Module 10	Resource 1			(49, 26)	(68, 34)				
	Resource 2			(49, 26)	(77, 34)				
	Resource 3	Not present		(49, 26)	(72, 48)	Not present		Not present	

Figure 5.1: Table

In our work, in some cases we have considered another resource (resource 4). Its size is 1×1.

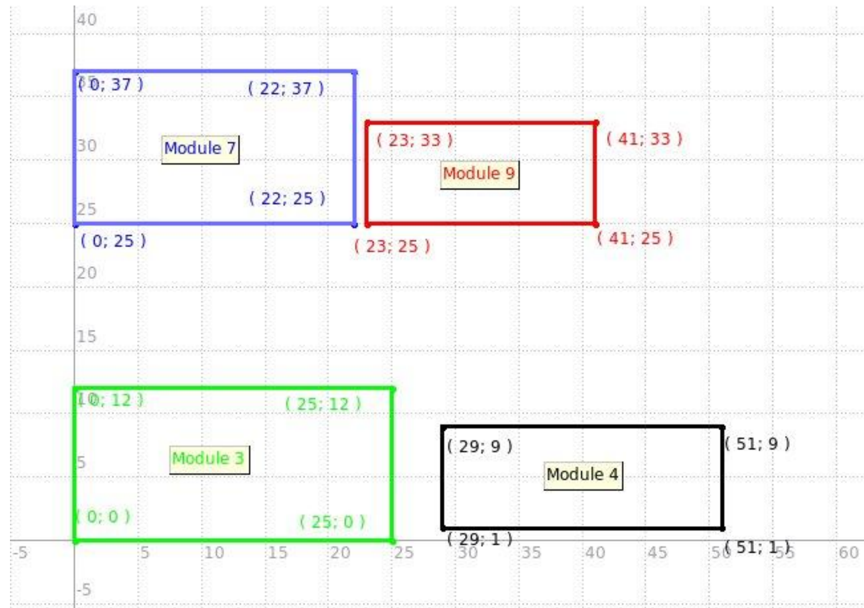

Figure 5.2: Instance 4 bounding box layout for resource 0

The output of the algorithm for this particular design is tabulated here. But it's sometime hard to understand how it's working with the numerical values. So, the bounding boxes for different modules in instances 4 are drawn below. As there are total 3 layers, the positions are drawn in 3 diagrams layer wise.

There are some observations what we can note from the tables and diagrams. Module 7 and 9 are the common modules in all the instances. So the position of the modules are same in all the 4 instances.

Module 7 and 9 are placed side by side. Other modules are placed depending upon the position required by the sequence pairs.

As the design suggests, the resource 2 is taking actually more places. Here the bounding box of resource 2 is almost square. Because the size of this resource is 1×1, that means square. So placement of many such resources are possible to be made almost square. In the design phases, it will try to change the aspect ratio. But, if the aspect ratios are such that the shapes are far from being square, it adds more to the cost. So for efficient design, the blocks are tried to make square. But if the targeted floor is of a shape far from a square, then the penalty in changing the aspect ratio, can be modified accordingly.

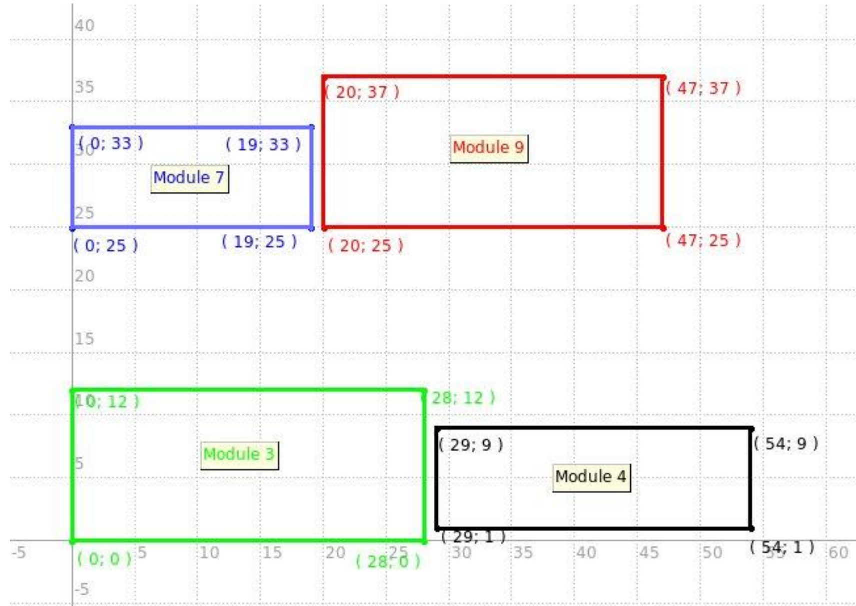

Figure 5.3: Instance 4 bounding box layout for resource 1

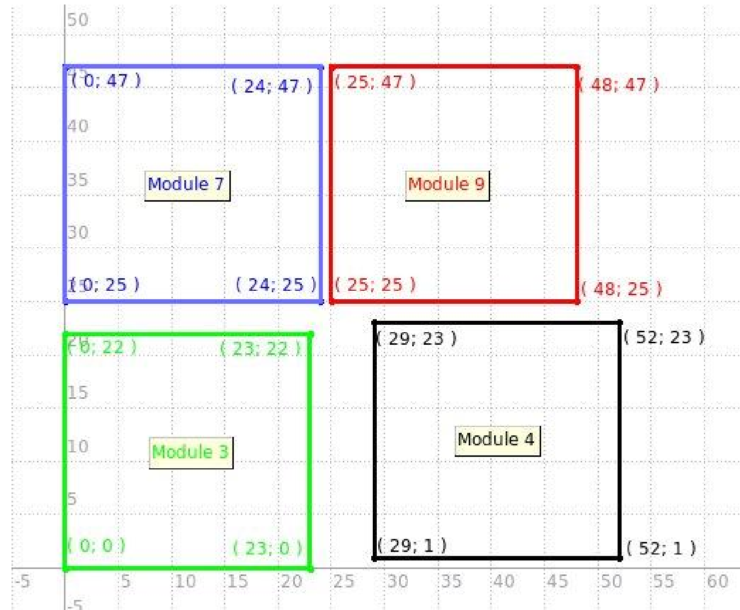

Figure 5.4: Instance 4 bounding box layout for resource 2

5.2 Result Comparison

Now let's see how the design works for different circuits. The result is written according to the runtime in a system with Athlon 64 $\times$ 2 Dual-core processor and 1 GB ram.

Input	#Module	#Instances	#Resources	Module No < Resource Requirement Vector per >	Instance No <Modules Present>	Sequence Pairs	Time (Sec)
ipfile1	10	4	3	1 <10 20 480> 2 <23 27 498> 3 <23 26 489> 4 <12 14 486> 5 <24 18 490> 6 <12 23 474> 7 <20 11 498> 8 <23 27 490> 9 <11 25 499> 10 <10 17 492>	1 <1 2 3 7 9> 2 <2 4 7 9 10> 3 <4 5 6 7 8 9> 4 <3 4 7 9>	**x** <6 2 3 4 5 1 7 8 9 10> **y** <3 5 7 1 10 8 9 2 6 4>	0.253
ipfile2	12	4	3	1 <10 20 480> 2 <23 26 498> 3 <22 27 489> 4 <12 14 487> 5 <24 19 491> 6 <13 22 474> 7 <20 12 497> 8 <23 27 490> 9 <11 25 498> 10 <10 17 492> 11 <15 18 502> 12 <13 14 497>	1 <1 2 3 7 9 12> 2 <2 4 7 9 10> 3 <4 5 6 7 8 9> 4 <3 4 7 9 11 12>	**x** <6 2 3 4 5 1 7 8 9 10 11 12> **y** <3 5 7 11 1 10 8 9 2 12 6 4>	0.288
ipfile3	10	4	3	1 <10 20 4801> 2 <23 27 4982> 3 <23 26 4891> 4 <12 14 4863> 5 <24 18 4904> 6 <12 23 4745> 7 <20 11 4986> 8 <23 27 4900> 9 <11 25 4997> 10 <10 17 4925>	1 <1 2 3 7 9> 2 <2 4 7 9 10> 3 <4 5 6 7 8 9> 4 <3 4 7 9>	**x** <6 2 3 4 5 1 7 8 9 10> **y** <3 5 7 1 10 8 9 2 6 4>	0.112
ipfile4	11	4	3	1 <10 20 480> 2 <23 27 498> 3 <22 24 489> 4 <12 14 487> 5 <23 18 490> 6 <12 23 474> 7 <21 11 478> 8 <13 27 490> 9 <21 25 499> 10 <10 17 492> 11 <13 18 504>	1 <1 2 3 7 9 11> 2 <2 4 7 9 10 11> 3 <4 5 6 7 8 9 11> 4 <3 4 7 9 11>	**x** <6 2 3 4 5 1 7 8 9 10 11> **y** <3 5 7 1 10 8 11 9 2 6 4>	0.416

Input	#Module	#Instances	#Resources	Module No < Resource Requirement Vector per >	Instance No <Modules Present>	Sequence Pairs	Time (Sec)
ipfile5	10	5	3	1 <10 20 480> 2 <23 27 498> 3 <23 26 489> 4 <12 14 486> 5 <24 18 490> 6 <12 23 474> 7 <20 11 498> 8 <23 27 490> 9 <11 25 499> 10 <10 17 492>	1 <1 2 3 7 9> 2 <2 4 7 9 10> 3 <4 5 6 7 8 9> 4 <3 4 7 9> 5 <2 3 7 8 9>	**x** <6 2 3 4 5 1 7 8 9 10> **y** <3 5 7 1 10 8 9 2 6 4>	0.336
ipfile6	30	7	3	1 <10 20 480> 2 <23 27 498> 3 <23 26 489> 4 <12 14 487> 5 <24 18 490> 6 <12 23 474> 7 <21 11 498> 8 <23 27 490> 9 <11 25 499> 10 <10 17 492> 11 <13 15 489> 12 <12 23 564> 13 <23 27 490> 14 <12 22 494> 15 <14 19 492> 16 <13 15 449> 17 <22 23 514> 18 <24 27 498> 19 <22 22 479> 20 <15 15 486> 21 <23 18 450> 22 <22 23 474> 23 <14 14 486> 24 <22 27 490> 25 <16 23 474> 26 <21 21 498> 27 <22 27 490> 28 <23 25 499> 29 <12 17 492> 30 <13 24 497>	1 <1 2 3 7 9 12 17 13 30> 2 <2 4 7 9 10 22 17 30> 3 <4 5 6 7 8 9 23 16 17 30> 4 <3 4 7 9 12 17 23 30> 5 <1 4 6 7 8 9 12 17 30> 6 <2 4 5 7 9 11 17 27 30> 7 <4 7 9 11 17 25 29 30>	**x** <6 2 3 4 5 1 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30> **y** <3 5 21 7 1 22 16 18 10 8 9 20 2 6 4 13 12 15 23 24 29 19 30 17 11 14 26 25 27 28>	1.251

ipfile7	30	7	4	1 <10 20 480 234> 2 <23 27 498 235> 3 <23 26 489 332> 4 <12 14 486 342> 5 <24 18 490 256> 6 <12 23 474 256> 7 <20 11 498 278> 8 <23 27 490 321> 9 <11 25 499 329> 10 <10 17 492 298> 11 <12 15 489 310> 12 <12 23 564 238> 13 <23 27 490 199> 14 <12 22 499 400> 15 <14 19 492 345> 16 <13 15 449 378> 17 <22 23 514 389> 18 <24 27 498 321> 19 <22 22 479 390> 20 <15 14 486 366> 21 <23 18 450 369> 22 <22 23 474 311> 23 <12 14 486 399> 24 <22 28 490 254> 25 <16 23 474 239> 26 <20 21 498 345> 27 <22 27 490 342> 28 <21 25 499 389> 29 <10 17 492 317> 30 <13 24 497 288>	1 <1 2 3 7 9 12 17 13 30> 2 <2 4 7 9 10 22 17 30> 3 <4 5 6 7 8 9 23 16 17 30> 4 <3 4 7 9 12 17 23 30> 5 <1 4 6 7 8 9 12 17 30> 6 <2 4 5 7 9 11 17 27 30> 7 <4 7 9 11 17 25 29 30>	**x** <6 2 3 4 5 1 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30> **y** <3 5 21 7 1 22 16 18 10 8 9 20 2 6 4 13 12 15 23 24 29 19 30 17 11 14 26 25 27 28>	1.545
ipfile8	30	8	3	1 <10 20 480> 2 <23 27 498> 3 <23 26 489> 4 <12 14 486> 5 <24 18 490> 6 <12 23 474> 7 <20 11 498> 8 <23 27 490> 9 <11 25 499> 10 <10 17 492> 11 <12 15 489> 12 <12 23 564> 13 <23 27 490> 14 <12 22 499> 15 <14 19 492> 16 <13 15 449> 17 <22 23 514> 18 <23 27 498> 19 <22 22 479> 20 <15 14 486> 21 <23 18 450> 22 <22 23 474> 23 <12 14 486> 24 <22 28 490> 25 <16 23 474> 26 <20 21 498> 27 <22 27 490> 28 <21 25 499> 29 <10 17 492> 30 <13 24 497>	1 <1 2 3 7 9 12 17 24 30> 2 <2 4 7 9 10 24 17 30> 3 <4 5 6 7 8 9 24 16 17 30> 4 <3 4 7 9 12 17 23 24 30> 5 <1 4 6 7 8 9 12 17 24 30> 6 <2 4 5 7 9 11 17 24 30> 7 <4 7 9 11 17 24 29 30> 8 <3 4 5 7 9 17 24 30>	**x** <6 2 3 4 5 1 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30> **y** <3 5 21 7 1 22 16 18 10 8 9 20 2 6 4 13 12 15 23 24 29 19 30 17 11 14 26 25 27 28>	1.493

Figure 5.5: Comparison table

Input	#Module	#Instances	#Resources	HPWL per instance	Cost Function Value	Time (Sec)
ipfile1	10	4	3	2234 3456 3234 402	0.841	0.253
ipfile2	12	4	3	5234 3352 3434 4024	0.721	0.288
ipfile3	10	4	3	2334 3346 3154 522	0.692	0.112
ipfile4	11	4	3	5432 3575 3539 4127	0.823	0.416
ipfile5	10	5	3	5907 3247 3274 793 3455	0.713	0.336
ipfile6	30	7	3	12361 22173 12372 22382 23714 21371 21731	0.821	1.251
ipfile7	30	7	4	28371 12389 21938 23914 32381 32432 22312	0.811	1.545
ipfile8	30	8	3	13183 23184 31212 41834 22472 37282 13825 12816	0.763	1.493

Figure 5.6: Performance table

5.3 Result Interpretation

1. The input files are chosen in such a way, that we can compare the time required if any one of #Modules, #Resources, #Instances or #Common Modules are increased.

2. If the number of resources are increased (in $ipfile7$ compared to $ipfile6$), the running time is increased significantly.

3. Similarly, if the number of common modules is increased (in $ipfile8$ compared to $ipfile6$), the running time is increased significantly.

4. If the sizes of any particular resources are increased ($ipfile3$), the running time is decreased significantly. As the size is increased, number of moves required is reduced. So, time is decreased.

5. The overall time required is not more compared to previous deterministic algorithm [5]. Besides, the area required is also low as the design is multi-layer.

6. This algorithm is faster comapred to previous probabilistic algorithm [3][4].

7. It can handle any number of resources which are not done in previous deterministic algorithm [5].

Chapter 6

Concluding Remarks and Future Work

In this we propose a simulated annealing based multi-layer floorplanning to obtain the fixed positions for the common modules across all instances such that resource requirement of rest of the modules are still satisfied and the total cost of the floorplan is minimized. In general simulated annealing based algorithm is little bit slow. So I have introduced some sort of determinism in the algorithm to reduce the run time as discussed before. So, there is a scope of improvement so that time taken is further reduced.

We plan to test our method for a set of realistic benchmarks to show that it is faster than the previous simulated annealing based methods and more efficient than the previous deterministic algorithm.

There is no standard partitioner like *hmetis* for multi-layer design. So the partitioner used in our work is not much efficient. There is lots of scope to improve it. Then the overall design will also give the result in much faster way.

Here the bounding box of a module can be rectilinear, but the individual resource bounding box is rectangle. So, in the next phase, the individual bounding box can be designed as rectilinear so that the design is much more efficient. We have done some work on L shaped block. But the generalization part is yet to be done.

Bibliography

[1] M. Sarrafzadeh, C.K. Wong, "An Introduction to VLSI Physical design", Mc graw Hill, 1996.

[2] Naveed A. Sherwani, "Algorithms For VLSI Physical Design Automation", Kluwer Academic Publishers, 1994.

[3] L. Singhal, E. Bozorgzadeh, "Multi-layer Floorplanning on a Sequence of Reconfigurable Designs", IEEE Int. Conf. on Field Programmable Logic and Applications (FPL), Madrid, Spain, 2006.

[4] L. Singhal and E. Bozorgzadeh, "Heterogeneous floorplanner for fpga", IEEE FCCM, Apr. 2007.

[5] Pritha Banerjee, Megha Sangtani and Susmita Sur-Kolay, "Floorplanning for Partial Reconfiguration in FPGAs", in proc. of the 22nd International Conference on VLSI Design (VLSID 2009), IEEE CS Press, January 2009, New Delhi, India.

[6] P. Banerjee, S. Sur-Kolay, A. Bishnu, "Floorplanning in Modern FPGAs", in Proc. International Conference on VLSI Design, pp. 893 - 898, 2007.

[7] Jin Xu, Pei-ning Guo, Chung-Kuan Cheng, "Sequence-pair approach for rectilinear module placement", in Proc. International Symposium on Physical Design 1998, Monterey, CA, ETATS-UNIS (06/04/1998)

[8] Y. Feng, D. P. Mehta, "Heterogeneous Floorplanning for FPGAs" in Proc. International Conference on VLSI Design, pp. 257-262, 2006.

[9] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research", in 7th International Workshop on Field Programmable Logic and Applications, pp. 213-222, 1997.

[10] A.B. Kahng, "Classical Floorplanning Harmful?", ISPD 2000, pp. 207-213.

[11] J.A. Roy, S. N. Adya, D. A. Papa, I. L. Markov, "Min-Cut Floorplacement", IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems, vol. 25, No. 7, pp. 1313-1326, Jul. 2006.

[12] Y. Feng, D. P. Mehta, H. Yang, "Constrained Floorplanning using Network Flows", IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 23, no. 4, pp. 572-580, 2004.

[13] L. Cheng and M. D. F. Wong, ”Floorplan Design for Multimillion Gate FPGAs”, in IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems, vol. 25, no. 12, pp. 2795-2805, Dec. 2006.

[14] http://www.xilinx.com

[15] Ian Kuon, Russell Tessier and Jonathan Rose, ”FPGA Architecture: Survey and Challenges”, Foundations and Trends in Electronic Design Automation, Vol. 2, No. 2, pp 135-253, 2008

[16] Deming Chen , Jason Cong and Peichen Pan , ”FPGA Design Automation: A Survey”, Foundations and Trends in Electronic Design Automation, Vol. 1, Issue 3, pp 139-169, 2006

[17] Vaughn Betz, Jonathan Rose, Alexander Marquardt, ”Architecture and CAD for Deep-Submicron FPGAs”, Springer, 1999