M.Tech.(Computer Science) Dissertation Series

Partial Reconfiguration in Floorplanning of FPGAs with Heterogeneous Resources

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

Dr. Susmita Sur-kolay

By

Megha Sangtani

Roll No : CS0609

Indian Statistical Institute

203, B.T. Road

Kolkata-700108

Indian Statistical Institute

Certificate of Approval

This is to certify that the thesis entitled **Partial Reconfiguration in Floorplanning of FPGAs with Heterogeneous Resources** submitted by Megha Sangtani towards partial fulfillment for the degree of M. Tech. in Computer Science at Indian Statistical Institute, Kolkata, embodies the work done under my supervision.

Dr. Susmita Sur-Kolay
ACMU, ISI, Kolkata

Date:

Acknowledgment

It is with great reverence that I wish to express my deep sense of gratitude to my guide Dr. Susmita Sur-Kolay, Associate 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.

I express my sincere thanks to Ms. 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. In the last, but not the least, I want to thank my almighty GOD.

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. 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 unified global floorplan topology generation method 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 wirelength of the floorplan is minimized.

Contents

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 switchboxes present at every crosspoint of horizontal and vertical channels. An FPGA is programmed using a configuration file which contains the place and route information to implement the design. To reprogram an FPGA, i.e, 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. Heterogeneity in resources makes the mapping of logic on FPGA board more difficult, which requires additional steps in the mapping process.

1.1 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't 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 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 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

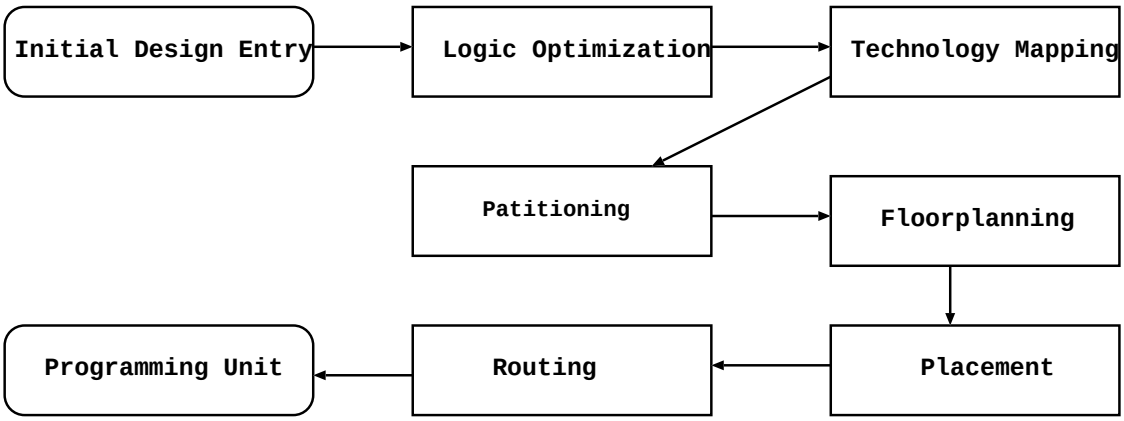


Figure 1.1: FPGA Design Cycle

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.

1.3 Scope of this thesis

In this thesis, an deterministic floorplan topology generation method is proposed for partial reconfiguration of a set of designs at different instances which minimizes the partial reconfiguration overhead, while optimizing the performance of the designs. The rest of the thesis is organized as follows.

In chapter 2, the floorplanning problem in the context of partial reconfiguration is introduced. Chapter 3 describes the method proposed. The method is demonstrated with an example in Chapter 4. The experimental results on a set of benchmarks are given in Chapter

5. 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

Modern FPGAs are heterogeneous with different kinds of resources like CLBs, RAMs, Multipliers (MUL) etc., while earlier FPGAs used to have only CLBs. Fig. 2.1 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 Figure 2.1, 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.

2.2 Problem Formulation

First, the basic terminology is given below.

Definition 2 *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.*

Definition 3 *Resource Requirement Vector [?]: For a module 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.*

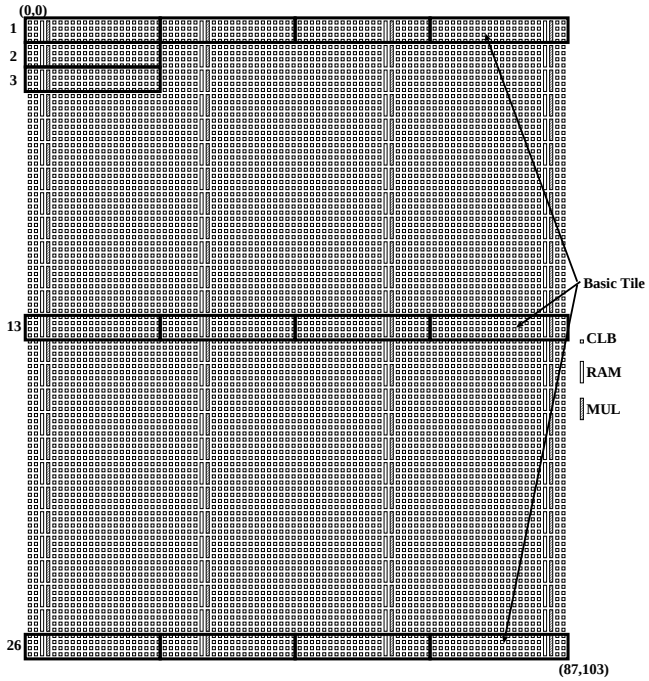


Figure 2.1: Spartan-3 XC3S5000 FPGA Architecture

2.2.1 Floorplanning Problem for heterogeneous FPGAs

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 $m_i \in M$, and (c) the netlist 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 [7, 8]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. This bounding box cost has also been

extensively used as a FPGA placement metric [6]. The problem formulation as stated above is a generalization of that given in [9, 10] and as such is *NP*-hard. Like most of the prior works on FPGA heterogeneous floorplanning [9, 11], we also consider HPWL as the objective function.

2.2.2 Partial Reconfiguration Problem

Definition 4 *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.*

The floorplanning problem for partial reconfiguration is defined as follows: In a given schedule, let there be k instances $I_1, I_2, \cdots I_k$. Let $s_1, s_2, \cdots s_m$ be m common modules that remain active in all instances.. For each $I_i, 1 \leq i \leq k$, let there be n_i modules $d_{i1}, d_{i2}, \cdots d_{in_i}$. The connectivity of the modules in each instance is also given. The objective is to give a floorplan of all modules across all instances such that

(i) the resource requirement of each module is satisfied in each instance,
(ii) the location and shape of each static module is same across all instances,
(iii) the half-perimeter wirelength (HPWL) of netlist for all instances is minimized.

As in any floorplanning problem, we consider sum of HPWL of each instance as the objective function to be minimized in the context of partial reconfiguration.

2.3 Existing Approaches

There are only a few floorplanning approaches for FPGA. Most of them use probabilistic techniques like simulated annealing with sequence pair representation of the floorplan. 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 and if the initial floorplan is not good then it may take large number of

iterations to get the optimal floorplan. Singhal and Bozorgzadeh have introduced a new multi layer sequence pair representation based floorplanner in their paper [3] which maximizes the overlap of common components of multiple designs thereby reducing reconfiguration overhead and guarantees a feasible floorplan with minimum area packing. The main drawback of this method is the long execution time because of probabilistic nature of simulated annealing. The commercial tool like Xilinx' Planahead [15] requires manual placement of the common modules beforehand, and then the rest of the modules are placed. This method is also based on simulated annealing.

To overcome the long execution time of simulated annealing based approach and yet arrive at a global floorplan such that the partial reconfiguration is minimized and performance of floorplan of each instance in optimized, a fast deterministic method is proposed. As in case of [3], this method places the common modules with same shape across all instances at a specific position on the chip so that they need not be reconfigured again and again, thereby reducing reconfiguration overhead. Also the remaining modules are placed in such a way that the wirelength (HPWL) is minimized globally. Unlike the method in [3], where sequence-pair is used as floorplan representation under simulated annealing based moves, the proposed method uses slicing tree [1] representation and node sizing for topology generation. Fekete et. al have proposed an alogorithm for optimal free space management and routing conscious dynamic placement for reconfigurable devices in their paper [12]. They find an optimal feasible communication - conscious placement which minimizes the total weighted mahattan distance between new module and already placed modules.

Chapter 3

Proposed Method

The goal of the proposed method is to design a fast yet effective unified floorplan topology such that static modules occupy same position at every instance and still the performance is optimizes. Our method consists of three phases: (i) creating a linear arrangement of modules for each instance with the fixed position of static modules, (ii) slicing tree topology generation for floorplan and, (iii) realization of the floorplan using actual resource requirements of the modules.

In the first phase, we obtain a linear arrangement of modules to minimize the sum of the total wirelengths such that heavily connected modules come closer to each other.

In the second phase a list of global slicing tree topologies is generated for each instance with the positions of static modules fixed at the bottom left and top right corners of the floorplan.

In the third phase, we group the set of slicing trees, obtained in the second phase, on the basis of shapes of static modules. For each of the slicing tree in each group, a rectangular region is assigned to every module, which respects the cut direction and the actual resource requirement of the modules.

Finally, one floorplan from each instance is chosen such that the total wirelength is minimum.

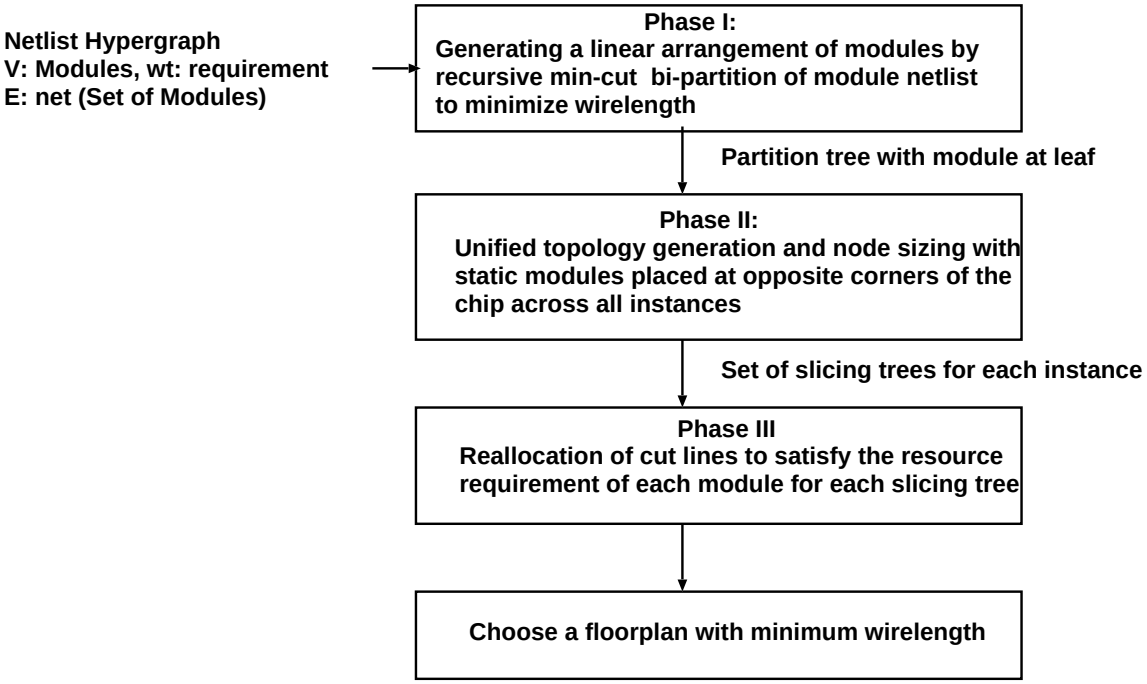


Figure 3.1: Flow of the Proposed Method

3.1 Preliminaries

For modules with homogeneous resource requirement, the area of a module could be considered for generation of shapes, which can be later used for node sizing in traditional topology generation, when floorplans are represented as slicing trees [1]. For heterogeneous resource requirements, where each resource type have specific location on the board, shapes can not be generated from the resource requirement vector. Thus, *basic tile*, an uniform entity is defined to compute the resource requirement of each module, which could be easily adapted for generation of shapes during node sizing.

Definition 5 *Basic Tile: A Basic FPGA Tile* $A = (a_{clb}, a_{ram}, a_{mul})$ *is a* 3 *tuple vector composing of the minimum number of CLBs, RAMs, MULs that constitute a basic unit which can be repeated horizontally and vertically to cover all the rows and columns of a given FPGA architecture.*

The given architecture is thus composed of, say, $T_w \times T_h$ basic tiles arranged in h rows and w columns. In Fig. 3.2, The basic tile $A = (80, 1, 1)$ consists of 20×4 CLBs , 1 RAM and 1

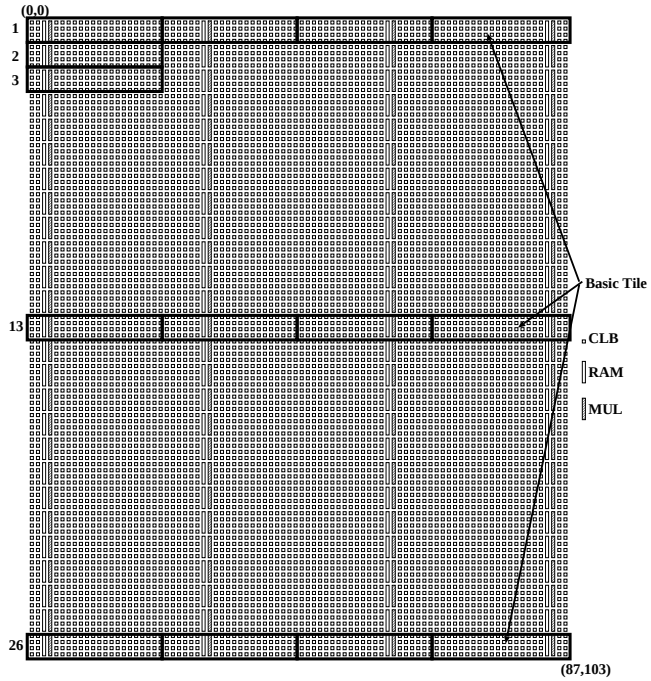


Figure 3.2: Spartan-3 XC3S5000 FPGA Architecture, tessellated with a basic tile, indicated by a rectangle of 4 rows and 20 columns of CLBs and 1 pair of RAM-MUL blocks

MUL. The entire architecture (Spartan-3) in Fig. 3.2 can be covered by 26 rows and 4 columns of basic tile A.

3.2 PHASE I- Linear arrangement of Modules

To minimize the wire length of the feasible floorplan we obtain a linear arrangement of modules such that heavily connected modules come closer to each other. Finding an optimal linear arrangement of modules of a netlist is NP-hard. Thus we use a min-cut bi-partitioning heuristic recursively till there is a single module in each partition. The recursive bi-partitioning generates a binary tree at every step of recursion, which is called a *decomposition* or *partition* tree. The left to right order of the modules at the leaves is considered to be a good linear arrangement [5]. The partition tree generated in this phase is the baseline of slicing tree generation in the next phase.

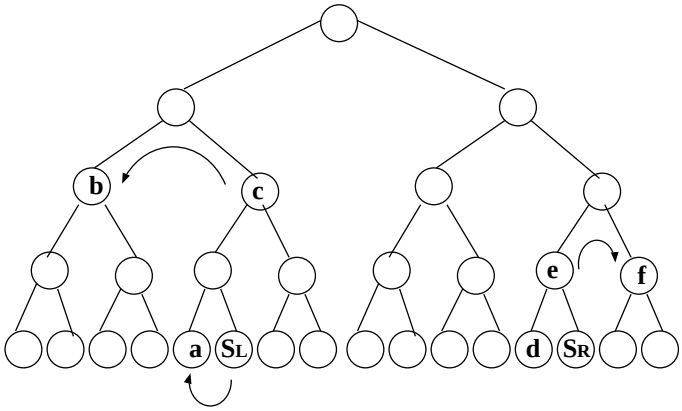
We use state-of-the-art bi-partitioning tool $hMetis$ [13] for partitioning hypergraphs. The netlist of modules is best represented by an hypergraph $H = (V, E)$. Each vertex $v \in V$

corresponds to a module m_i, $i = 1, 2, \cdots n$. An hyperedge $e = \{v_1, v_2, \cdots v_p\} \in E$ corresponds to a signal net connecting the set of modules $\{v_1, v_2, \cdots v_p\}$. If there are more than one such sets, then the total number of such set is considered to be the weight of that hyperedge e. The minimum number of tiles required by a module m is computed from the size of the *basic tile* A and the resource requirement vector R_m. This is given as the weight of the vertex corresponding to module m. The tool $hMetis$ produces a balanced min-cut partitioning of this hypergraph.

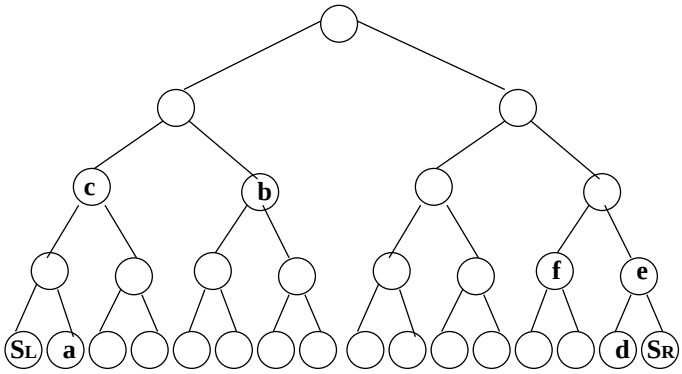
Static modules should have the same shape and location across all instances. It is beneficial to place all the static modules to the extreme corners of the floorplan, so that, we get the largest continuous space in the middle of the chip to place the rest of the dynamic modules contiguously. We generate the slicing tree of the floorplan in the second phase from the partition tree obtained in this phase. The objective of placing the static modules to the corners of a floorplan led to the following observation.

Observation 1 *In a slicing tree representation of a floorplan, the modules at left most and right most leaves of the tree always correspond to the two opposite corners of the floorplan.*

From Observation 1, if we can place the static modules at extreme ends of the partition tree, the static modules will definitely be on the opposite corners on the floorplan. To generate a partition tree with this constraint we do the following. First we extract the static modules and the corresponding netlist from the given schedule. then we bi-partition the static modules into two groups S_L and S_R and call each of them a *super module*. Now we have two static super modules and some dynamic modules along with the netlist for each instance. This netlist of modules is bi-partitioned recursively until each partition contains at most one module/super module per partition. In the first level of recursive bi-partitioning, we force S_L and S_R to be in different partitions, so that, they can be pushed to extreme left and right positions during the further recursive partitioning. As swapping of left and right partition in a bi-partition do not affect the min-cut, two partitions can interchange their position without affecting the cut. During recursive bi-partitioning the left and right partitions are swapped in such a way that partitions having static super modules are always pushed to the extreme left and extreme right of the linear arrangement of the modules obtained at leaf level. The swapping of partitions with static super modules to the leftmost and the rightmost leaf of the partition tree is shown



Before Swapping



After Swapping

Figure 3.3: Swapping of static super modules to extreme ends of the partition tree; the arrow shows the partitions to be exchanged

in Fig. 3.3.

Thus we get one partition tree for each instance where the static modules are at the extreme left and right leaves.

3.3 PHASE II- Unified floorplan topology generation

In this step a set of sliceable floorplan topologies is generated for each instance by appropriate horizontal and vertical node sizing starting from a set of possible shapes (in terms of tiles) of each module.

3.3.1 Generation of Module Shapes

A list$D = \{(w_1, h_1), (w_2, h_2), \cdots, (w_t, h_t)\}$ of irredundant shapes of a module m, is a list of t possible shapes of m, where (w_i, h_i) denotes the width and height of i^{th} shape of m in terms of basic tiles. D is said to be irredundant if each individual w_i and h_i are distinct.

By making individual w_i and h_i distinct, a shape with smaller height is chosen from two implementations with same width. Thus the inferior shape is always gets eliminated. A set of possible irredundant rectangular shapes for m_i is generated by factorizing T_{m_i}. As we are considering only rectangular shapes there may not be many choices such that $width \times height = T_{m_i}$. A few more shapes are generated by factorizing all integers from T_{m_i} to the smallest composite integer which is greater than T_{m_i}.

3.3.2 Node Sizing

A subtree rooted at an internal node p corresponds to a sub floorplan. The sub floorplan at p is generated by joining $(w_i, h_i) \in D_l$ and $(w_j, h_j) \in D_r$ vertically or horizontally, where D_l and D_r are i^{th} shape of left child (left sub floorplan) and j^{th} shape of right child (right sub floorplan) of p. If p is the parent of leaves then the left and right sub floorplans are the shapes of the modules themselves.

Vertical Cut : We use the vertical node sizing algorithm of [1] to generate a sub floorplan with vertical cut. Let $D_l = \{(w_{l_1}, h_{l_1}), (w_{l_2}, h_{l_2}), \cdots, (w_{l_s}, h_{l_s})\}$, with $|D_l| = s$ and $D_r = \{(w_{r_1}, h_{r_1}), (w_{r_2}, h_{r_2}), \cdots, (w_{r_t}, h_{r_t})\}$, with $|D_r| = t$, be the set of possible irredundant shapes of the left sub floorplan/module and the right sub floorplan/module respectively, of a node in partition tree. D_l is sorted such that, $w_{l_1} < w_{l_2} < \cdots < w_{l_s}$ and $h_{l_1} > h_{l_2} > \cdots > h_{l_s}$.

D_r is also sorted as above. If (w_{l_i}, h_{l_i}) and (w_{r_j}, h_{r_j}) are merged vertically, the resultant floorplan size becomes $(w_{v_k}, h_{v_k}) = (w_{l_i} + w_{r_j}, max(h_{l_i}, h_{r_j}))$. The number of resultant irredundant shapes is at most $s + t - 1$ [1].

Horizontal cut: To merge sub floorplans using horizontal cut, we use the same irredundant lists D_l and D_r as described above. The lists are sorted in increasing order of height and decreasing order of width i.e. $h_{l_1} < h_{l_2} < \cdots < h_{l_s}$ and $w_{l_1} > w_{l_2} > \cdots > w_{l_s}$

Merging (w_{l_i}, h_{l_i}) and (w_{r_j}, h_{r_j}) by a horizontal cut the resultant size of the floorplan be-

comes $(w_{v_k}, h_{v_k}) = (max(w_{l_i}, w_{r_j}), h_{l_i} + h_{r_j})$. As in case of vertical cut the number of resultant irredundant shapes is at most $s + t - 1$.

3.3.3 Generation of Slicing Trees

A set of slicing tree is generated for the decomposition or partition tree obtained in PHASE - I by appropriate shape generation and node sizing as described above. Each leaf node of the tree corresponding to a module contains a list of possible shapes, i.e., (width, height) pair in terms of tiles. For all instances, the corresponding partition trees are traversed simultaneously bottom-up, level by level, generating a set of irredundant sub-floorplans by combining the available shapes of its left and right children with vertical or horizontal cut. Whenever a static super module is combined with its neighbouring dynamic module by a particular cut, the shape generated at parent must also be irredundant. To generate irredundant shapes at the parent node, a particular shape of static super modules may be thrown out in some of the instances when combined with its neighboring dynamic modules by a cut. We discard such shapes of static modules from all the instances when a particular shape is eliminated from any of the instances so that the list of shapes of static modules remain same for all the instances. If at any level, we end up with an empty list of shapes for any of the static super modules then we can directly report that floorplanning is not possible on the FPGA board for the linear arrangement of modules/super modules obtained in the first phase, and may iterate with another linear arrangement.

At the end of this phase we get a set of slicing trees for each of the instances with static super modules at two opposite corners of the floorplan. The list of shapes (width, height) generated at root may not fit the target FPGA chip when the shapes are considered in terms of tiles. The target chip is 4×26 tiles. We consider only those slicing trees with root shape of width between 3 and 6 as there is a high possibility of getting a feasible floorplan on this target board in the third phase.

3.4 PHASE III - Realization of Slicing Trees on the chip

For every slicing tree generated in the previous step, now we assign coordinate position to each module. We allocate a rectangular region which satisfies the CLB requirements.

3.4.1 Allocation of Rectangular Region to a Module

Each slicing tree is traversed top down and a rectangular region $({x_{min}}^p, {y_{min}}^p, {x_{max}}^p, {y_{max}}^p)$ is assigned to every node p using the cut direction and the number of CLBs required at p. Let the region allocated to some node p contains r_{clb} rows and c_{clb} columns of CLBs. If the CLB requirements at node p, its left child l and its right child r are p_{clb},l_{clb} and r_{clb} respectively. If the p represents the vertical cut, the number of clb columns allocated to p is p_{col} and number of clb rows allocated to p is p_{row} then the rectangular box assignment for l and r is done by the following equations : -

Let the rectangular assigned to l and r are $({x_{min}}^l, {y_{min}}^l, {x_{max}}^l, {y_{max}}^l)$ and $({x_{min}}^r, {y_{min}}^r, {x_{max}}^r, {y_{max}}^r)$ respectively. Let the clb rows and columns allocated by l and r are l_{row}, l_{col} and r_{row}, r_{col}.

For a horizontal cut at p,

$l_{col} = l_{clb}/p_{row}$,

$r_{col} = r_{clb}/p_{row}$,

${x_{min}}^l = {x_{min}}^p$, ${y_{min}}^l = {y_{min}}^p$,

${x_{max}}^l = {x_{min}}^p + l_{col} - 1$, ${y_{max}}^l = {y_{max}}^p$,

${x_{min}}^r = {x_{max}}^p - r_{col} + 1$, ${y_{min}}^r = {y_{min}}^p$,

${x_{max}}^r = {x_{min}}^p$, ${y_{max}}^r = {y_{max}}^p$.

For a vertical cut at p,

$l_{row} = l_{clb}/p_{col}$,

$r_{row} = r_{clb}/p_{col}$,

${x_{min}}^l = {x_{min}}^p$, ${y_{min}}^l = {y_{min}}^p$,

${x_{max}}^l = {x_{max}}^p$, ${y_{max}}^l = {y_{min}}^p + l_{row} - 1$,

${x_{min}}^r = {x_{min}}^p$, ${y_{min}}^r = {y_{max}}^p - r_{row} + 1$,

${x_{max}}^r = {x_{max}}^p$, ${y_{max}}^r = {y_{min}}^p$.

As a convention, the vertical cut line is positioned by counting the columns from left to

right for the left child and right to left for the right child. Silmilarly for horizontal cut, it is positioned by counting the rows from bottom to top for the left child and top to bottom for the right child. This may generate a overlapping rectangular region in the middle of the two rectangles assigned to the left and right child. We allocate the CLBs required by a module to the non overlapping region of the rectangles assigned to the corresponding module. The remaining CLB requirement of each module, called *deficit*, has to be allocated either to the overlapping rectangle or to the neighboring rectangles. By positioning the cutlines as described above, free rectangular region might get generated too in the middle. Thus, three types of rectangular region is created by positioning of cut lines. (i) non overlapping part of rectangle assigned to a module either with no free CLBs within it or with some free CLBs in it (ii) overlapping rectangle, where conflicts for CLB requirements of more than one module needs to be resolved, (iii) free rectangles created at the middle due to the convention followed to assign rectangles to a module. The deficit (if any) of each module is satisfied during post processing described in Section 3.4.4

3.4.2 Pruning the Trees

While allocating the rectangular regions to modules in different instances,their RAM/MUL requirements are not considered. So we cannot say anything about whether RAM/MUL requirement of modules will be satisfied within rectangular region allocated.

Definition 6 *Major Violation : If a module has RAM/MUL requirement and has been assigned the rectangular box such that no RAM/MUL column going through it, then the module is said to have the major violation.*

We discard all the floorplans from each instance in which any module has major violation. The intuition behind such floorplans is, if any module has been assigned such a rectangular box through no RAM?MUL column is going through and it has RAM/MUL requirements, then it would be difficult to borrow the RAM/MULT from neighboring modules if they have excess of them and then adjusting the shape of the affected modules. This may make the shapes of the modules very bad so we avoid such floorplans.

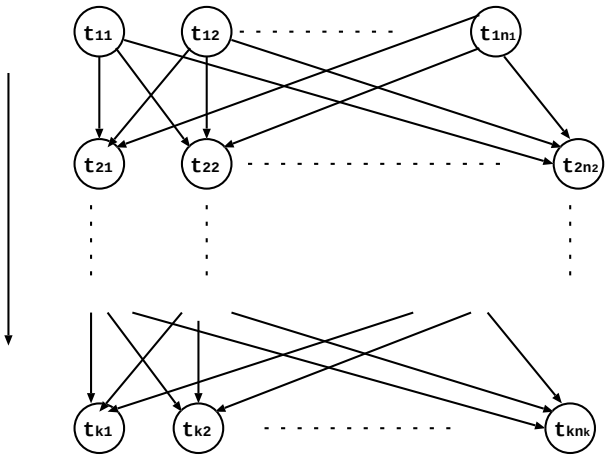


Figure 3.4: Graph generated for selecting set of trees in a group

3.4.3 Grouping the floorplans

After getting the set of floorplans for each instance, the question arises which set of the trees to be considered together for global floorplan so that the objective of the partial reconfiguration can be achieved, i.e. the same shapes for the static modules across all instances.

For this purpose, we calculate the aspect ratios of static modules/supermodule in each floorplan for each instance(after rectangular region allocation). We group the floorplans from each instance on the basis of nearly equal aspect ratios of the static module/super module such that a group contains at least one floorplan from each instance of the design. If there is more than one floorplan for an instance in the group, we need to select a single floorplan for that instance. For this, we define the following for the slicing topology corresponding to each floorplan.

Definition 7 *Hamming distance between two slicing trees: Let a and b be the strings representing the level order traversal of nodes from the root till one level above the leaves of the two slicing trees respectively, with horizontal cut represented as* 0 *and vertical cut represented as* 1*. Let $l = min\{length(a), length(b)\}$ and the length of the longer string be truncated till l from right. Then the hamming distance between these trees is the number of ones in $aXORb$.*

This basically measure the closeness among two slicing trees in terms of slicing topology. In the context of partial reconfiguration, a schedule implies the ordering of the instances on

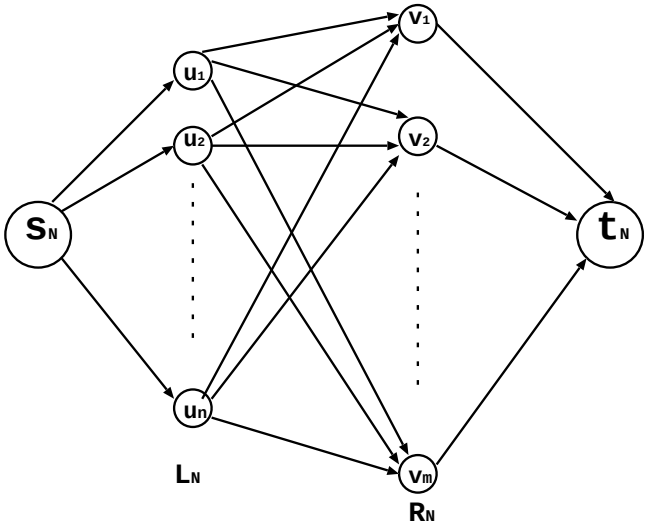


Figure 3.5: Network flow graph for reallocation of resources

the time line. To have same shapes of static from one instance to the consecutive one, the change in slicing tree must be minimum. Let, $T_1 < T_2 < \cdots T_k$ be the k trees of k instances in a given schedule, where $T_i < T_j$ indicates that i^{th} instance is executed prior to j^{th} instance in the schedule. So, T_1 and T_k are the trees from the first and the last instances in the schedule. We formulate a method for selecting slicing trees consecutively from each instance as follows. Let $G = (V, E)$ be a directed graph with $v \in V$ corresponding to the trees in a group. There exists a weighted edge $e \in E$, between $u, v \in V$, if $uandv$ corresponds to the slicing trees in consecutive instances. The weight is the hamming distance between u and v. If there are k instances, we find a minimum weighted k-length path starting from the node corresponding to T_1 to T_k. The floorplans corresponding to the trees in the minimum weighted path is selected as the final floorplans for a group.

3.4.4 Postprocessing for reallocation of resources

The set of floorplans chosen for each instance in a group have static modules with nearly equal aspect ratios but not exactly the same shape. We consider all pair of shapes, taking one from the list of S_L and the other from S_R and use this shape pair for static modules in all instances and reallocate CLBs of those dynamic modules that are either generating overlap with the shapes of static modules or they have some deficiency or excess CLBs within its rectangular

region. To allocate the deficit of a module, if exists, we use a minimum cost maximum flow (MCMF) formulation for each floorplan in the group.

For this, a network flow graph $N = (V_N, E_N)$ is generated. Here, N is a bipartite graph with a source node s_N and a sink node t_N. Let $V_N = L_N \cup R_N$. Each $v \in L_N$ corresponds to a module that is deficit of CLBs. Each $v \in R_N$ corresponds to the three types of rectangles with only free CLBs, described in Section 3.4.1. Let $E_N = E_s \cup E_{uv} \cup E_t$. For each $v \in L_N$ there exists an edge $e \in E_s, e = (s_N, v)$ with capcity as deficit of CLBs in module corresponding to v and cost as 1. For each $v \in R_N$ there exists an edge $e \in E_t, e = (v, t_N)$ with capacity as free CLBs in the rectangle corresponding to v and cost as 1. For the floorplan, a rectangular dual graph RD **??** is generated from the adjacency relationship of rectangles. For each $u \in L_N$, and for each $v \in R_N$, there exists an edge $e \in E_{uv}$ with capacity equal to the free CLBs in rectangle corresponding to v and cost is the length of the shortest path in RD from the vertex in RD corresponding to u to the vertex in RD corresponding to v. Figure 3.5 shows one such network flow graph. By solving MCMF, if the amount of flow is equal to the total deficit of CLBs, then these CLBs corresponding to each $v \in L_N$ is allocated by its neighboring rectangles. For each edge $e = (u, v) \in E_{uv}$ having a positive flow f and cost c implies that module corresponding to u borrows f CLBs from the rectangle corresponding to v following the c length path in RD from the vertex in RD corresponding to u to the vertex in RD corresponding to v. This results in rectilinear shape of a module. If MCMF does not give a solution for any one of the floorplan in a group, this group is rejected as a candidate solution for the partial reconfiguration problem.

Finally, the RAM/MULs of each module are allocated by another MCMF formulation described in [4]. This gives the final floorplans for each instance in partial reconfiguration problem. We choose a group with feasible floorplans of all instances with minimum sum of HPWL over all instances.

3.5 Time Complexity

Let k be the number of instances. Let h be the maximum number of signal nets in any instance. The min-cut bipartitoner $hMetis$ takes linear time in number of hyperedges i.e. signal nets in our case. We run $hMetis$ for all instances to get the linear arrangement in phase I. So, the

total time spent in first phase is $O(kh)$.

Let q be the maximum number of shapes generated for any module over all instances. Let n be the maximum number of modules in any instance. Then the slicing tree generation for any instance takes atmost $O(qn^2)$ time [4]. We generate slicing trees for k instances, so the time taken for phase II is $O(kqn^2)$.

Time taken in rectangular region allocation is $O(kn)$, while in pruning of floorplans takes $O(k)$ time. Grouping of floorplans takes $O(klogk)$. Rectangular dual graph for all floorplans is generated in $O(kn^2)$ and MCMF for floorplans in a group is solved in $O(kn^3)$. In this way total time taken in phase III is $O(kn^3)$.

Thus, total time complexity of the proposed method is $O(k(h + n^3))$.

Chapter 4

An Example

The method proposed in previous chapter is illustrated through an example in this chapter. We consider a synthetic benchmark that fits on a Spartan 3 Xilinf FPGA chip XC3S5000. This benchmark has two instances 0 and 1 which have 26 and 20 modules respectively. It has 4 static modules numbered 0,1,2,3. Instance 0 has static modules 0,1,2,3 and dynamic modules numbered from 4 to 25 while instance 1 has dynamic modules numbered from 26 to 41 with same static modules. Instance 0 requires 8141 CLBs, 84 RAMs and 84 Multipliers while instance 1 requires 8226 CLBs, 83 RAMs and 83 Multipliers. After calculating the requirements of these instances in terms of basic tiles, the instance 0 requires $max\{\frac{8141}{80}, 84, 84\} = 102$ tiles , while instance 1 requires $max\{\frac{8226}{80}, 83, 83\} = 103$ tiles.

We partition the static modules in two groups with min-cut bi-partitioner $hMetis$. As a result of this we get modules numbered 0 and 1 in one group and modules numbered 2 and 3 in other group. We treat them as two super modules. So, S_R contains modules 0 and 1 while S_L contains modules 2 and 3. We partition the dynamic modules of each instance by the balanced min-cut bipartitioner *hmetis* recursively by fixing the static modules to their respective partitions as described in section 3.2. The left to right order of the leaves of the partition tree gives the linear arrangement of modules as shown in Figure 4.1.

First, we generate a set of shapes (width, height) for each of the module using the requirements in terms of basic tiles. Here both the static super modules require 5 tiles so the shapes generated are $1 \times 5, 5 \times 1$. Few other shapes are also generated to have more trees, so other

irredundant shapes used for static modules are $2 \times 3, 3 \times 2$. We build up the slicing trees for both the instances as described in section 3.3. One such slicing tree for each instances is shown in Figure 4.1. The internal nodes represents the cut used to join the child subtree at parent node. The shapes, i.e. (width, height) pair, of the module/super module/internal node, are shown beside the corresponding node in the slicing tree. The shape at root for both the trees is 4×28. Since the width is equal to width of the target chip, these slicing trees are possible feasible floorplans.

The realization of the two floorplans corresponding to the slicing trees of Figure 4.1, after rectangular region allocation to each module, are shown in Figure 4.2 and 4.3. The rectangles with module numbers shown in the figures, show the non overlapping regions, which may or may not have free spaces and the rectangular strips without any module numbers shows the overlapping or free regions. The shapes of S_L and S_R in instance 0 are 19×19 and 21×18, while in instance 1, these are 21×18 and 19×20.

We choose the common shape for S_L and S_R to be 21×18 and 21×18 respectively. We draw the rectangular dual graphs for both the instances, shown in Figure 4.4 and 4.5, where numbered vertices corresponds to the modules with same number and vertices labeled with albhabets corresponds to the free or overlapped region. From these graphs we draw a network flow graph described in 3.4.4, for both the floorplans shown. After solving MCMF in section 3.4.4, we get the floorplans with all the CLB requirements satisfied by generation of rectilinear shapes as shown in Figure 4.6 and 4.7.

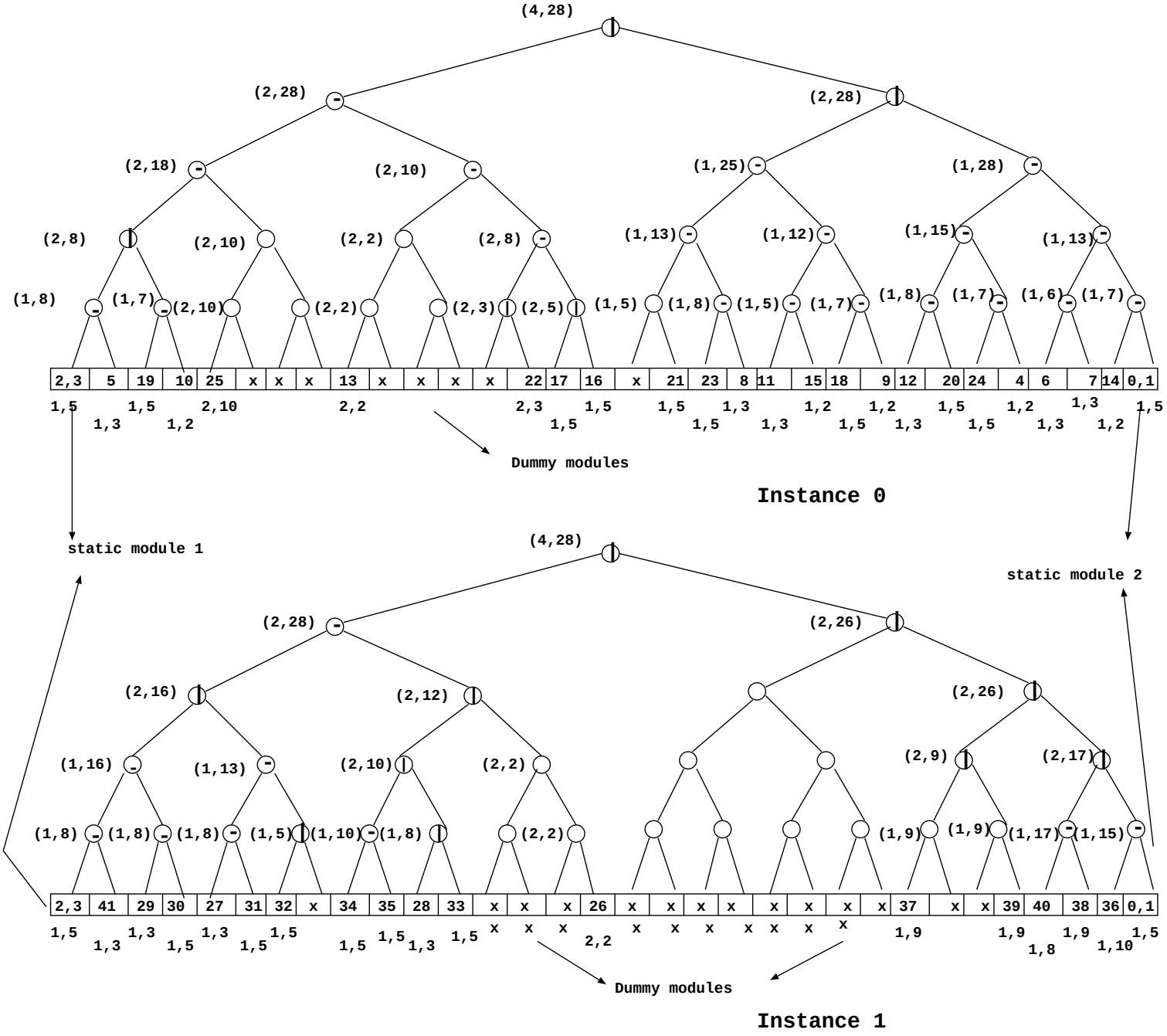


Figure 4.1: Slicing trees of two instances; '|' and '–' represent vertical and horizontal cut; shapes of module are given as (width, height)

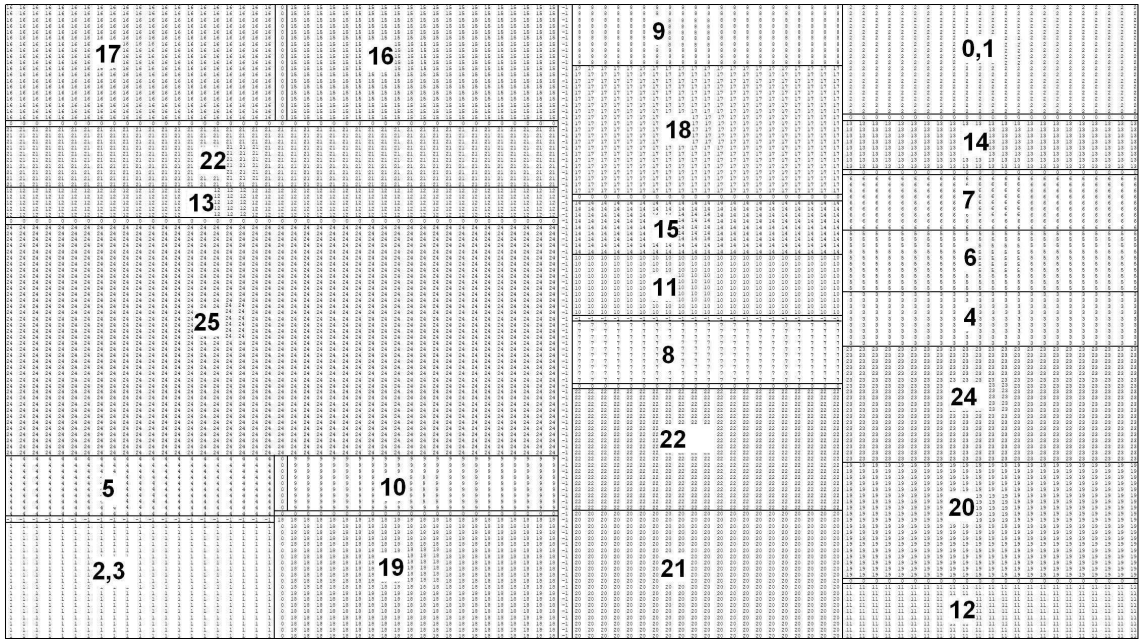


Figure 4.2: Floorplan for instance 0 after rectangular region allocation

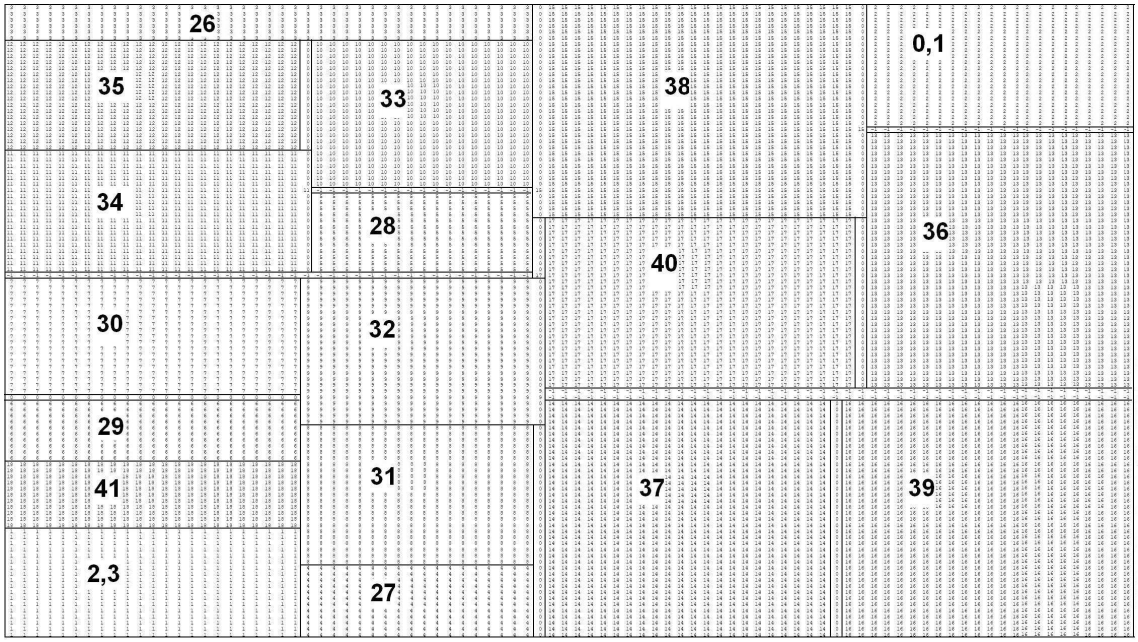


Figure 4.3: Floorplan for instance 1 after rectangular region allocation

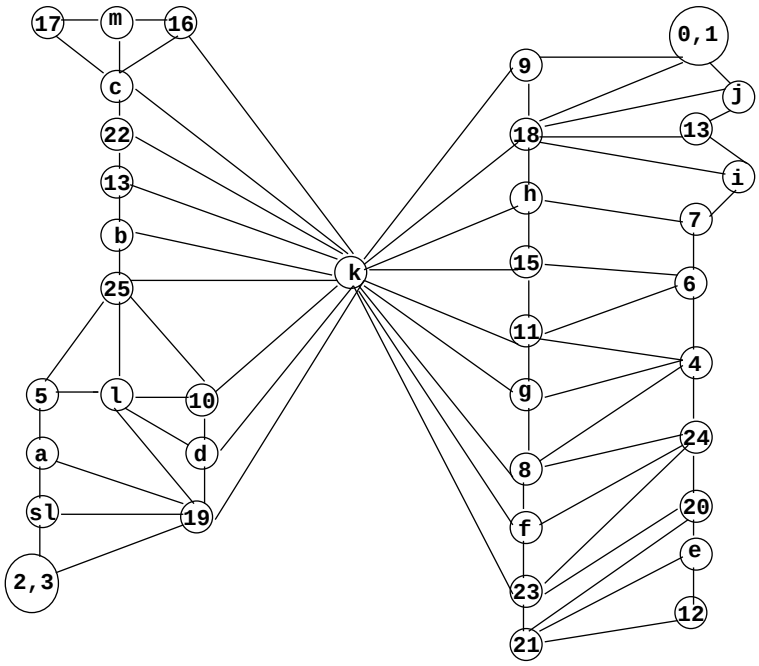


Figure 4.4: Rectangular dual graph for instance 0; albhabets show free or overlapped region

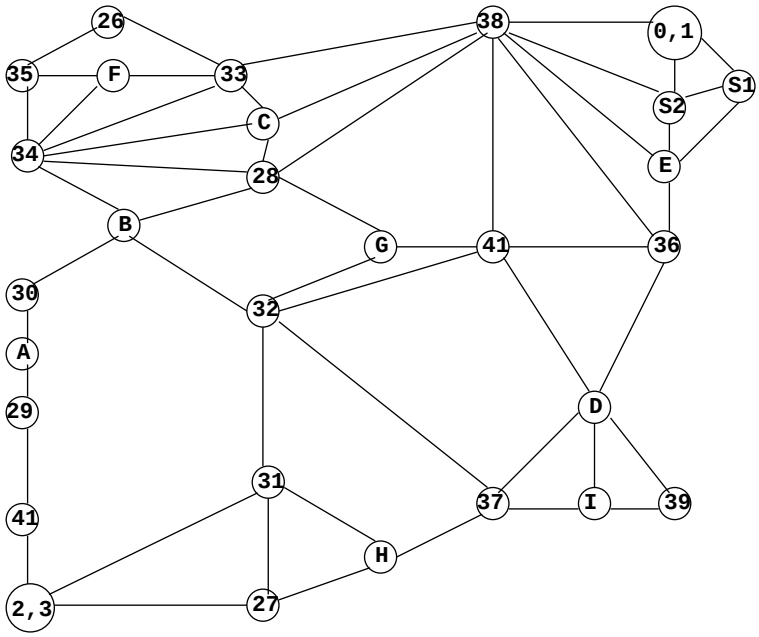


Figure 4.5: Rectangular dual graph for instance 1; albhabets show free or overlapped region

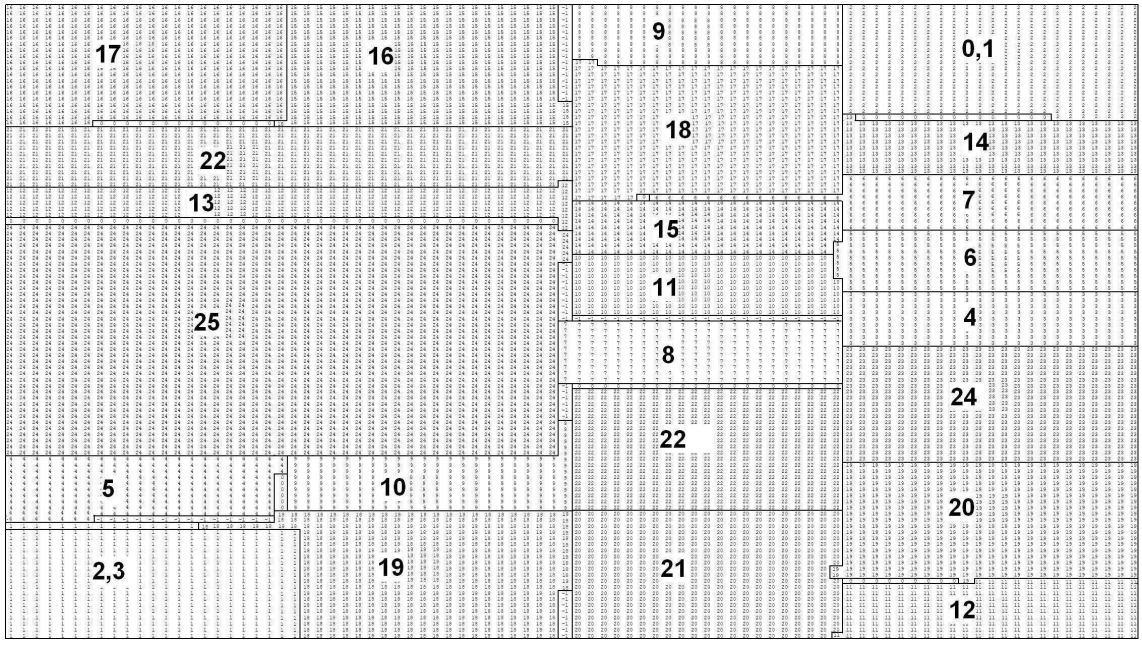


Figure 4.6: Rectilinear shape generation for instance 0

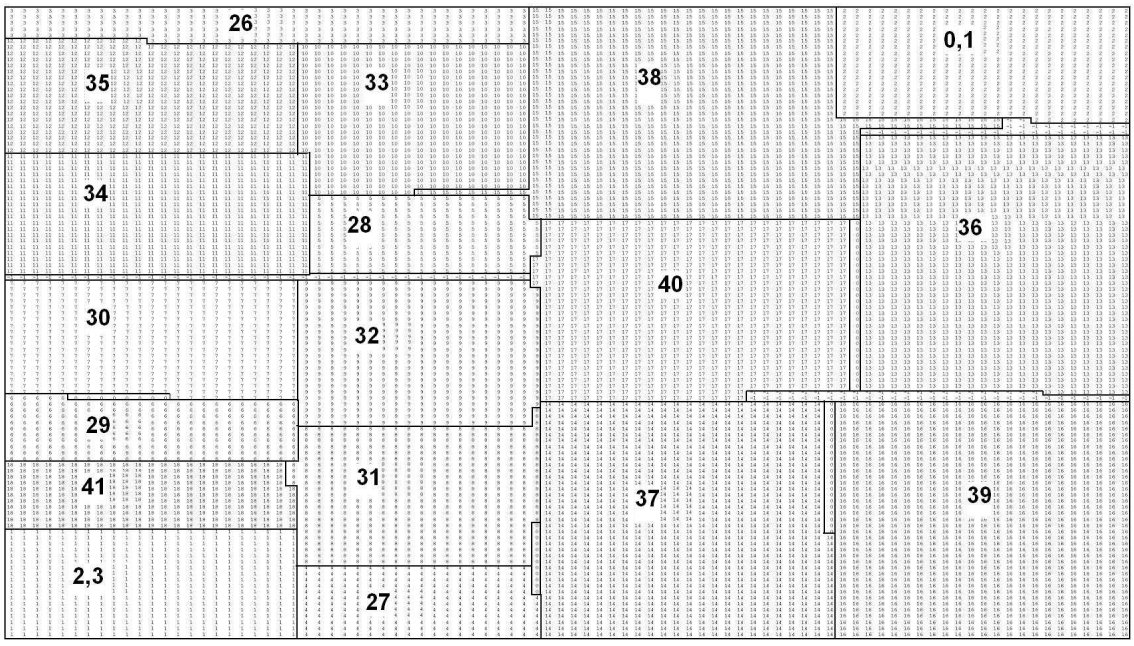


Figure 4.7: Rectilinear shape generation for instance 1

Chapter 5

Experimental Results

The method described is implementred using C on linux platform with hMetis[13] and LEDA [14] library on Intel core 2 duo CPU. The method has been tested for 10 different synthetic benchmarks. Table 5.1 shows the number of instances, number of static modules, maximum number of modules and signal nets for any instance in each benchmark used for the experiment.

Table 5.1 Characteristics of 10 benchmark used for experiments

benchmark	**no of instances**	**no of static modules**	**max # modules**	**max # signal nets**
*bench*1	5	4	31	660
*bench*2	5	2	31	527
*bench*3	6	3	33	510
*bench*4	6	4	29	486
*bench*5	6	2	31	450
*bench*6	7	3	30	510
*bench*7	8	3	34	500
*bench*8	9	5	30	420
*bench*9	10	3	29	544
*bench*10	10	5	31	680

As described in the method, we first partition the modules in each instance till there is one module in each partition. Table 5.2 shows the number of modules and their CLB, RAM and MUL requirement along with the number of signal nets in each instance of each benchmark. with a comparison of the partitioning time (i) when static modules are fixed to specific partition and (ii) otherwise.

Table 5.2 Comparison between the partitioning time with and without fixing the partitions of static modules: C:CLB, R: RAM, M:MUL requirement

benchmark	**instance no** (C,R,M)	**# modules**	**# nets**	**partitioning time(in sec.)**	
				(i)	**(ii)**
*bench*1	0(8141, 84, 84)	24	260	0.176	0.092
	1(8226, 83, 83)	18	360	0.124	0.096
	2(7858, 83, 83)	23	500	0.196	0.124
	3(7854, 89, 89)	16	252	0.104	0.064
	4(8171, 78, 78)	31	660	0.348	0.152
*bench*2	0(7938, 78, 78)	18	144	0.092	.0.056
	1(8078, 74, 74)	29	319	0.196	0.104
	2(8011, 80, 80)	26	234	0.156	0.092
	3(8126, 95, 95)	31	527	0.316	0.160
	4(7519, 87, 87)	29	145	0.132	0.072
*bench*3	0(8143, 85, 85)	25	494	0.260	.0.184
	1(8228, 82, 82)	19	400	0.152	0.124
	2(7841, 83, 83)	21	374	0.148	0.124
	3(7844, 87, 87)	15	208	0.100	0.060
	4(8173, 79, 79)	33	510	0.284	0.152
	5(7761, 74, 74)	19	280	0.156	0.096
*bench*4	0(7753, 67, 67)	23	250	0.128	.0.092
	1(8087, 91, 91)	19	294	0.116	0.084
	2(7932, 89, 89)	21	391	0.196	0.104
	3(7832, 74, 74)	25	480	0.220	0.116
	4(7627, 78, 78)	29	217	0.156	0.076
	5(7627, 78, 78)	12	84	0.036	0.024

*bench*5	0(7916, 77, 77)	17	272	0.108	.0.072
	1(8092, 75, 75)	30	450	0.216	0.124
	2(8031, 83, 83)	26	416	0.232	0.152
	3(8126, 95, 95)	31	310	0.224	0.112
	4(7529, 88, 88)	30	150	0.140	0.064
	5(7675, 69, 69)	19	361	0.136	0.100
*bench*6	0(7911, 76, 76)	16	255	0.116	.0.060
	1(8095, 76, 76)	29	480	0.236	0.132
	2(8027, 81, 81)	26	243	0.180	0.088
	3(8112, 93, 93)	29	510	0.240	0.144
	4(7529, 88, 88)	30	372	0.192	0.108
	5(7656, 69, 69)	18	361	0.128	0.092
	6(7778, 84, 84)	13	196	0.076	0.048
*bench*7	0(8164, 87, 87)	27	206	0.160	.0.112
	1(8246, 81, 81)	19	200	0.100	0.072
	2(7856, 83, 83)	22	299	0.132	0.092
	3(7854, 90, 90)	16	272	0.128	0.064
	4(8180, 81, 81)	34	385	0.216	0.116
	5(7762, 73, 73)	18	190	0.108	0.056
	6(7719, 86, 86)	17	342	0.164	0.096
	7(8147, 91, 91)	24	500	0.200	0.116
*bench*8	0(8171, 83, 83)	25	392	0.192	.0.120
	1(8253, 83, 83)	20	414	0.148	0.100
	2(7825, 83, 83)	21	120	0.100	0.052
	3(7840, 88, 88)	14	306	0.096	0.060
	4(8135, 78, 78)	30	264	0.176	0.088
	5(7747, 72, 72)	18	420	0.152	0.116
	6(7742, 88, 88)	18	420	0.180	0.108
	7(8145, 91, 91)	23	312	0.144	0.088
	8(7949, 80, 80)	23	234	0.160	0.100
*bench*9	0(7899, 74, 74)	16	306	0.116	.0.072
	1(8080, 74, 74)	28	174	0.132	0.072
	2(8028, 81, 81)	26	81	0.096	0.044
	3(8119, 94, 94)	29	300	0.184	0.104
	4(7528, 84, 84)	29	300	0.148	0.112

	5(7664, 69, 69)	18	209	0.108	0.052
	6(7825, 84, 84)	14	120	0.072	0.040
	7(8124, 80, 80)	18	361	0.144	0.100
	8(7746, 80, 80)	31	544	0.284	0.156
	9(7768, 81, 81)	23	192	0.116	0.080
*bench*10	0(8132, 83, 83)	23	286	0.148	.0.100
	1(8231, 81, 81)	18	294	0.120	0.072
	2(7830, 79, 79)	20	115	0.092	0.044
	3(7839, 87, 87)	14	323	0.128	0.068
	4(8185, 78, 78)	31	680	0.304	0.172
	5(7770, 76, 76)	19	220	0.124	0.080
	6(7744, 87, 87)	18	336	0.110	0.088
	7(8151, 91, 91)	23	520	0.228	0.132
	8(7980, 81, 81)	24	378	0.172	0.104
	9(7980, 89, 89)	29	192	0.132	0.076

The details of the slicing trees obtained for each instance of a benchmark by the method described in Section 3.3 is shown in Table 5.3. The columns 3 to 6 show the number of slicing trees, range of aspect ratio of static modules for feasible floorplans and the number of floorplans that are discarded because of major violation.

Table 5.3 Number of slicing trees, aspect ratio range of static modules, trees with major violation

benchmark	**instance no**	**no of slicing trees**	**aspect ratio range in feasible floorplans**		**no of floorplans with Major violation**
			SL	**SR**	
*bench*1	0	21	$0.51 - 1.80$	$1.21 - 6.50$	3
	1	21	$1.27 - 6.37$	$0.55 - 1.62$	2
	2	20	$0.42 - 5.11$	$0.90 - 6.50$	2
	3	20	$0.75 - 17.60$	$0.48 - 1.86$	1

	4	20	$1.27 - 6.37$	$0.65 - 2.61$	3
*bench*2	0	20	$1.14 - 8.80$	$1.81 - 15.75$	.2
	1	22	$1.14 - 8.20$	$1.30 - 5.83$	3
	2	21	$1.72 - 8.40$	$2.55 - 8.40$	3
	3	22	$1.72 - 8.80$	$1.30 - 8.40$	2
	4	20	$0.76 - 4.57$	$1.14 - 4.71$	3
*bench*3	0	22	$0.68 - 2.91$	$0.87 - 8.40$	2
	1	21	$0.32 - 2.53$	$1.50 - 6.00$	2
	2	20	$0.95 - 6.37$	$0.87 - 8.80$	1
	3	20	$0.59 - 6.62$	$0.30 - 2.10$	1
	4	22	$1.27 - 6.37$	$0.87 - 3.25$	2
	5	20	$0.86 - 4.10$	$2.10 - 8.40$	3
*bench*4	0	21	$1.10 - 17.60$	$1.15 - 8.28$	3
	1	22	$1.68 - 6.62$	$1.15 - 6.37$	1
	2	21	$0.90 - 17.60$	$2.21 - 8.85$	1
	3	21	$0.41 - 2.06$	$0.68 - 2.53$	3
	4	21	$0.48 - 4.40$	$0.86 - 4.20$	3
	5	19	$0.57 - 6.62$	$0.51 - 1.80$	2
*bench*5	0	20	$2.10 - 8.40$	$1.81 - 29.33$	3
	1	21	$0.87 - 3.50$	$1.14 - 3.87$	3
	2	20	$2.20 - 9.40$	$1.00 - 3.50$	3
	3	21	$1.72 - 8.40$	$1.14 - 6.33$	2
	4	21	$0.72 - 2.55$	$1.30 - 4.85$	3
	5	20	$1.30 - 6.33$	$2.20 - 6.00$	2
*bench*6	0	20	$1.38 - 29.33$	$0.90 - 6.62$	2
	1	22	$0.93 - 3.50$	$1.33 - 6.62$	3
	2	21	$0.57 - 14.00$	$0.50 - 1.86$	3
	3	21	$0.68 - 2.30$	$0.62 - 2.28$	2
	4	22	$0.82 - 3.50$	$0.65 - 2.61$	3
	5	20	$6.83 - 29.33$	$0.57 - 2.28$	2
	6	19	$0.68 - 6.83$	$0.57 - 17.60$	1
*bench*7	0	20	$0.68 - 2.69$	$1.72 - 13.25$	3
	1	20	$0.51 - 1.62$	$0.33 - 8.4$	2
	2	21	$0.82 - 5.33$	$3.25 - 29.33$	0
	3	20	$0.75 - 3.00$	$6.16 - 29.33$	2

	4	19	$1.27 - 6.37$	$1.72 - 8.40$	3
	5	18	$1.05 - 6.37$	$1.81 - 29.33$	2
	6	20	$0.68 - 4.20$	$1.30 - 6.16$	3
	7	21	$0.95 - 6.37$	$0.63 - 15.75$	3
*bench*8	0	21	$1.80 - 8.28$	$0.54 - 2.00$	3
	1	20	$5.22 - 17.60$	$0.70 - 11.00$	3
	2	21	$4.30 - 17.60$	$0.75 - 5.27$	2
	3	20	$0.86 - 6.62$	$1.17 - 4.58$	3
	4	21	$1.62 - 6.37$	$0.70 - 11.00$	2
	5	19	$0.59 - 5.11$	$0.85 - 11.00$	2
	6	20	$2.61 - 11.60$	$0.39 - 2.80$	3
	7	21	$0.51 - 5.11$	$0.70 - 6.30$	2
	8	21	$4.20 - 17.60$	$0.85 - 5.63$	2
*bench*9	0	18	$1.05 - 3.45$	$0.47 - 2.20$	1
	1	21	$0.95 - 6.25$	$0.38 - 2.66$	2
	2	19	$0.65 - 2.91$	$1.14 - 8.60$	3
	3	22	$0.75 - 1.62$	$3.87 - 14.50$	2
	4	20	$0.75 - 2.91$	$0.47 - 2.30$	3
	5	20	$0.68 - 2.28$	$0.50 - 1.81$	2
	6	21	$2.21 - 5.11$	$0.57 - 3.50$	2
	7	20	$0.45 - 6.25$	$2.2 - 29.33$	0
	8	20	$1.15 - 8.14$	$1.00 - 3.37$	2
	9	19	$0.78 - 3.08$	$0.57 - 29.33$	2
*bench*10	0	20	$0.50 - 3.76$	$0.90 - 6.50$	2
	1	19	$0.92 - 4.41$	$0.51 - 2.28$	3
	2	20	$0.67 - 11.00$	$0.82 - 6.50$	2
	3	20	$1.27 - 4.58$	$1.33 - 7.00$	–
	4	21	$0.75 - 6.40$	$0.65 - 2.61$	2
	5	17	$0.67 - 11.00$	$2.28 - 17.60$	1
	6	17	$0.67 - 5.09$	$0.43 - 1.86$	1
	7	21	$0.67 - 6.40$	$2.28 - 17.60$	2
	8	21	$0.82 - 6.40$	$0.82 - 6.50$	3
	9	23	$0.43 - 1.84$	$0.65 - 2.61$	2

We calculate the wirelengths (HPWL) for all floorplans in each group after phase three either considering (i) the terminals at the center of the module or (ii) the terminals at the periphery of the module. The HPWL thus obtained for each tree is shown in Table 5.4.

Table 5.4 Wirelengths for different floorplans for each benchmark

benchmark	**instance no**	**Tree**	**Wirelength**	
			centre to centre	**boundary to boundary**
*bench*1	0	T_{01}	35515	45188
	1	T_{11}	40737	59138
		T_{12}	44453	58722
	2	T_{21}	65646	88755
		T_{22}	65004	86402
	3	T_{31}	28792	44188
		T_{32}	32184	42728
		T_{33}	29914	41951
	4	T_{41}	91028	113993
*bench*2	0	T_{01}	18458	25142
		T_{02}	18408	24379
	1	T_{11}	44129	54079
	2	T_{21}	30328	38202
	3	T_{31}	73635	92601
		T_{32}	73733	91635
	4	T_{41}	19649	24137
*bench*3	0	T_{01}	60969	81614
		T_{02}	63307	81870
	1	T_{11}	46771	64440
		T_{12}	48879	64115
	2	T_{21}	41099	61593
		T_{22}	44847	59131
		T_{23}	45596	59352
	3	T_{31}	23218	34066
		T_{32}	23852	34258

		T_{33}	24290	34177
	4	T_{41}	67229	87126
		T_{42}	68677	86463
	5	T_{51}	36097	46386
*bench*4	0	T_{01}	32725	42059
	1	T_{11}	37934	51198
		T_{12}	37125	49764
		T_{13}	37597	50478
	2	T_{21}	48079	69011
		T_{22}	50442	65531
		T_{23}	48144	65732
	3	T_{31}	62747	80276
	4	T_{41}	30318	37342
	5	T_{51}	9694	14305
		T_{52}	10285	14058
*bench*5	0	T_{01}	35595	44853
	1	T_{11}	62944	76657
	2	T_{21}	57887	71338
	3	T_{31}	39421	52579
		T_{32}	40231	50181
	4	T_{41}	19408	23996
	5	T_{51}	47534	62750
		T_{52}	48243	61594
*bench*6	0	T_{01}	32060	45951
		T_{02}	35337	45270
	1	T_{11}	65929	81783
	2	T_{21}	34594	42045
	3	T_{31}	66429	86260
		T_{32}	68763	85701
	4	T_{41}	51576	62824
	5	T_{51}	41377	64751
		T_{52}	49940	63295
	6	T_{61}	25652	34799
		T_{62}	26159	34324

		T_{63}	26703	34444
*bench*7	0	T_{01}	37523	48905
	1	T_{11}	22654	33453
		T_{12}	25545	32964
	2	T_{21}	29336	48312
		T_{22}	34916	47738
		T_{23}	36095	48038
		T_{24}	35188	48167
	3	T_{31}	31161	44694
		T_{32}	33415	44970
	4	T_{41}	51152	64457
	5	T_{51}	18714	30929
		T_{52}	21960	31139
	6	T_{61}	42210	55645
	7	T_{71}	59661	82217
*bench*8	0	T_{01}	53395	68901
	1	T_{11}	55649	72491
	2	T_{21}	15888	21554
		T_{22}	16567	21268
	3	T_{31}	37418	53711
	4	T_{41}	33037	45567
		T_{42}	35523	44687
	5	T_{51}	53453	75047
		T_{52}	56455	73941
	6	T_{61}	54469	72621
	7	T_{71}	39094	56243
		T_{72}	40431	54552
	8	T_{81}	28334	41194
		T_{82}	31463	40476
*bench*9	0	T_{01}	33967	50422
		T_{02}	35840	48668
		T_{03}	36523	49222
	1	T_{11}	21418	28784
		T_{12}	21829	27988
	2	T_{21}	10038	13029

	3	T_{31}	36422	49298
		T_{32}	37649	48397
	4	T_{41}	38911	49313
	5	T_{51}	25783	33751
		T_{52}	24544	33380
	6	T_{61}	15022	20344
		T_{62}	15121	20843
	7	T_{71}	44152	63347
		T_{72}	45670	63056
		T_{73}	47510	62306
		T_{74}	45975	62716
	8	T_{81}	69127	92185
		T_{82}	72623	91076
	9	T_{91}	21931	31453
		T_{92}	23801	30822

Chapter 6

Concluding Remarks and Future Work

In this thesis we have proposed a fast deterministic floorplanning method in the context of partial reconfiguration on FPGA with heterogeneous resources consisting of CLBs, RAMs and Mutlipliers. 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. Our method generates global slicing topology such that the exact physical location of each static module along with the shape, remains same across all instances. We choose the set of floorplans with minimum total semi perimeter wirelength over all instances. We show with an example how our method works for two instances.

We plan to test our method for a set of realistic benchmarks to show that it is faster than the simulated annealing based methods. As a future work, the constraint imposed on the positions of the static modules will be relaxed to incorporate the general positions of the static modules.

Bibliography

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

[2] L. J. Stockmeyer, "Optimal Orientations of Cells in Slicing Floorplan Designs", Information and Control, vol. 57, no. 2/3, pp. 91 - 101, 1983.

[3] L. Singhal, E. Bozorgzadeh, "Multi-layer Floorplanning on a Sequence of Reconfigurable Designs", in Proc. International Conference on Field Programmable Logic and Applications, pp. 605 - 612, 2006.

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

[5] G. Even, J. S. Naor, S. Rao, B. Schieber, "Divide-and-conquer Approximation Algorithms Via Spreading Metrics", Journal of the ACM (JACM), vol. 47, no. 4, pp. 585 - 616, 2000.

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

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

[8] 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.

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

[10] 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.

[11] L. Cheng, 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.

[12] A. Ahmadinia, C. Bobda, S. P. Fekete, J. Teich, J. C. van der Veen, "Optimal Free-Space Management and Routing-Conscious Dynamic Placement for Reconfigurable Devices", IEEE Trans. Comput., vol. 56, no. 5, pp. 673- 680, 2007.

[13] http://www-users.cs.umn.edu/ karypis/metis/hmetis

[14] http://www.algorithmic-solutions.com

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