NANYANG TECHNOLOGICAL UNIVERSITY

NANYANG TECHNOLOGICAL UNIVERSITY

MAPPING DATA FLOW GRAPHS ON COARSE-GRAINED FPGA OVERLAYS

by

HSIEH, MU-HUA
(G1501670L)

A Dissertation Submitted in partial fulfillment of
the requirements for the degree of
Master of Embedded Systems

Supervised by

Assoc. Prof. Douglas L. Maskell

July 2016

Contents

List of Figures

List of Tables

Abstract

FPGA based accelerator design is a complex process, requiring low-level hardware device expertise and specialist knowledge of both hardware and software systems, resulting in major design productivity issues. High level synthesis (HLS) has been proposed to address the design productivity issue and has helped to simplify accelerator design by raising the level of programming abstraction from RTL to high level languages, such as C/C++/OpenCL. Even though HLS tools have improved in efficiency, allowing designers to focus on high level functionality instead of low-level implementation details, the prohibitive compilation times (specifically the place and route times in the backend flow) have largely been ignored and are now a major productivity bottleneck that prevents designers from using mainstream design and debug methodologies based on rapid compilation. One solution that has been explored extensively by researchers is to implement a coarse-grained reconfigurable architecture on top a commercial FPGA device, referred to as a coarse-grained overlay. This allows the coarse-grained elements and structure, specifically the FU and interconnect to be changed at runtime as per the application requirements. Applications can be written at a higher level of abstraction with compilation to the overlay being several orders of magnitude faster than for the fine grained FPGA on which the overlay is implemented. This report presents a placement and routing (PAR) tool for coarse-grained island-style overlays based on the algorithm implemented in widely used and accepted Versatile Place and Route (VPR) tool. We start with understanding the placement and routing (PAR) algorithms in detail, develop a python based PAR tool which is customized for island-style FPGA overlays, and implements Fault Tolerance in it. We aim to adapt the algorithms to support different interconnect architectures as a future work.

Chapter 1

Introduction

1.1 Motivation

Field Programmable Gate Arrays (FPGAs), which allow the implementation to be modified post-deployment [1], are now more commonly used for rapid-prototyping of application specific accelerators in heterogeneous computing platforms. Some of the key advantages of FPGAs over other available platforms include reprogrammability compared to ASICs, lower power consumption than multicore processors and GPUs, real-time execution, and most importantly, the high spatial parallelism which can be used to significantly accelerate compute-intensive algorithms. For more than a decade, researchers have shown that FPGAs can accelerate a wide variety of applications, in some cases by several orders of magnitude compared to state-of-the-art GPPs [2, 3, 4, 5, 6].

FPGA based accelerator design is a complex process, requiring low-level hardware device expertise and specialist knowledge of both hardware and software systems, resulting in major design productivity issues. High level synthesis (HLS) [7, 8, 9, 10, 11] has been proposed to address the design productivity issue and has helped to simplify accelerator design by raising the level of programming abstraction from RTL to high level languages, such as C/C++/OpenCL. These tools allow the functionality of an accelerator to be described at a higher level to reduce developer effort, enable design

portability, enable rapid design space exploration, thus improving productivity, verifiability, and flexibility. Even though HLS tools have improved in efficiency, allowing designers to focus on high level functionality instead of low-level implementation details, the prohibitive compilation times (specifically the place and route times in the backend flow) have largely been ignored and are now a major productivity bottleneck that prevents designers from using mainstream design and debug methodologies based on rapid compilation.

One solution that has been explored extensively by researchers is to implement a coarse-grained reconfigurable architecture on top a commercial FPGA device, referred to as a coarse-grained overlay [12, 13, 14, 15, 16, 17, 18, 19, 20]. This allows the coarse-grained elements and structure, specifically the FU and interconnect to be changed at runtime as per the application requirements. Applications can be written at a higher level of abstraction with compilation to the overlay being several orders of magnitude faster than for the fine grained FPGA on which the overlay is implemented.

This report presents a placement and routing (PAR) tool for coarse-grained island-style overlays based on the algorithm implemented in widely used and accepted Versatile Place and Route (VPR) tool. We start with understanding the placement and routing (PAR) algorithms in detail, develop a python based PAR tool which is customized for island-style FPGA overlays, and implements Fault Tolerance in it.

1.2 Contribution

We start with understanding the algorithm of Versatile Placement and Routing (VPR) tool, which is widely accepted in industry and academia, then developed a python-based PAR tool for coarse-grained FPGA overlays, which uses APIs from Python graph library for implementation. The ultimate goal is to adapt the python-based PAR tool to support different FPGA architectures as the future work.

Our main contributions can be summarized as below:

- Understanding of placement and routing algorithms used in VPR
- Python based implementation of the VPR algorithms
- Fault tolerance implementation in placement algorithm

1.3 Organization

The remainder of the dissertation is organized as follows: Chapter 2 presents the background information of placement and routing algorithms and the algorithms used in VPR tool. Chapter 3 shows the placement steps of data flow graph (DFG) nodes on an island-style coarse-grained FPGA overlay, the routing steps of DFG edges on the FPGA overlay, and the implementation of fault tolerance in our python based place and route tool. Chapter 4 presents the conclusion and discusses the future work.

Chapter 2

Background

2.1 Placement

In case of fine-grained architectures, generally applications are described in hardware description language (HDL) such as Verilog/VHDL. The process of generating configuration data from HDL description can be divided into four major steps:

- Synthesis
- Technology Mapping
- Placement
- Routing

Synthesis step transforms the HDL to a hierarchical network of basic building blocks. Given a set of library cells, technology mapping is generally defined as mapping the network to the library cells. In case of FPGAs, this library is composed of k-LUTs, flip-flops, basic arithmetic circuits like adders, and advanced hard blocks. Therefore, the technology mapping for FPGAs consists of transforming the Boolean network into a set of nodes. Placement is the process of determining which logic blocks should be placed where. In other words, which specific logic blocks on FPGA should be used for a particular instance of a logic block of given network. Routing is the process of finding routes so that all logic blocks used in placement stage are properly connected. In the next sections, we introduce existing methods for placement.

2.1.1 Simulated Annealing Method

According to Fig. 2.1, simulated annealing algorithm is a heuristic-based probabilistic technique to approximate global minimum optimum for the objective function in a large search space. Basically, it mimics the annealing process to gradually cool the melting metal to produce solid metal with high quality. It interprets slow cooling process as a slow decrease in the probability of accepting worse solutions when exploring the solution space. It is the property of heuristics as it allows exploring more searches for global optimal solution. Therefore, if the cooling schedule is perfect enough, the simulated annealing process can ultimately converge to a global optimal solution.

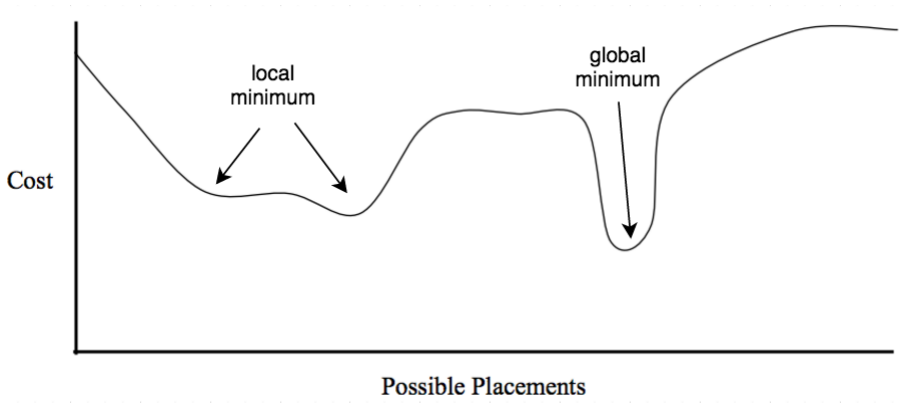


Figure 2.1: Global and Local minimum value of Simulated Annealing algorithm

Basically, it calculates the delta value by deducting cost of old placement from new one, then uses random number generator to generate number between 0 and 1. And the new placement is acceptable if random number less than the exponential on the delta between new and old cost of placement divided by temperature. Otherwise, it will rip up to the original placement. The above process is executed iteratively by slowly decreasing the temperature. The well-known overall algorithm is shown in Fig. 2.2.

Fig. 2.3 shows the relationship between probabilities to accept the new placement and delta in different temperatures. It begins with very high temperature, which is able to accept even bad swap. Then the temperature slowly goes to a very low value. It is obvious that the curve becomes steeper as the temperature turns low

```
P = InitialPlacement ();
T = InitialTemperature ();

while (ExitCriterion () == False) {
    while (InnerLoopCriterion () == False) {  /* "Inner Loop" */
        P_new = PerturbPlacementViaMove (P);
        ΔCost = Cost (P_new) – Cost (P);
        r = random (0,1);
        if (r < e^(-ΔCost/T)) {
            P = P_new;      /* Move Accepted */
        }
    } /* End "Inner Loop" */
    T = UpdateTemp (T);
}
```

Figure 2.2: Simulated Annealing algorithm

which means it only accepts the certain swap in lower temperature to ensure the good quality of swap.

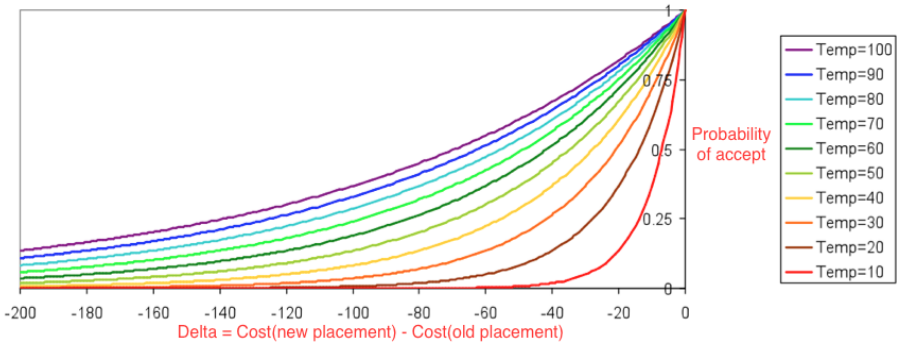


Figure 2.3: Distributed Graph of Probability in Simulated Annealing Algorithm

Just like the procedure to cool the real melted metal, the cooling schedule is important. If it is cooled too fast, it will force a greedy solution. In practice simulated annealing is easy to program, very versatile, and widely used.

The advantages for simulated annealing algorithm are:

- Cost function can be modified based on the wire length or timing
- Can reach globally optimal if given enough processing time

The disadvantages:

- The overall procedure takes longer time for large circuits

2.1.2 Analytical Method

The best algorithm for FPGA placement is simulated-annealing algorithm, which has been discussed above. However, it is not scalable as it consumes long time to converge. The basic idea of analytical placement is used to place the blocks based on appropriate net length estimation and then working towards disjointness. Then the wire-length is approximated by Half-Perimeter Wirelength (HPWL) of the smallest bounding box, which contains all terminals of a net inside. The limit for HPWL model is that the cost is not accurate if a net with more than two terminals. It also means that HPWL cannot be efficiently minimized. To avoid this problem, a quadratic wire length objective function can be applied. However, it will sacrifice the short nets and over-emphasize the longer nets. An analytical method named StarPlace model is proposed by [21] to solve this drawback. The method uses start net to model each block, which has a connection to the center of the gravity of the net. The length of each edge is the quadratic distance from block to center of the gravity. It seeks the minimum sum of the square roots of all distance between different blocks that connect to the same net instead of the sum of all quadratic distances. Thus, it can avoid over-emphasizing the cost of long net when performing optimization.

2.2 Routing

Routing is an important part as interconnection mainly decides the area of the FPGA, and interconnection delay is usually greater than the delay on logic blocks. As a result, an ideal routing algorithm manages to reduce the wiring area and critical path of circuit. The ultimate target is improving the performance of the circuit. Basically, routing is an NP complete problem. It can be divided into two phases:

- Global routing: the main purpose of global routing is balancing routing channels and preventing any overuserd routing resources. If the net with more than one

sink, then it will be decomposed into k-1 steps to be routed. In fact, there are various ways to reach sink(s), and the router chooses the path with the least congested channels by tracking the usage of every routing resource.

- Detailed routing: it assigns each specific wiring segment to each interconnection. The detailed router specifies the exact wiring segment to be routed to the logic blocks. In order to achieve this, it constructs a directed graph which consists of available routing resources such as wires, connection boxes and switch boxes on the FPGA. The directed graph is based on Dijkstra's algorithm which helps us find out the shortest path between two nodes.

Therefore, performance can be achieved by avoiding congested channels and minimizeing wiring area and length.

The routing algorithm can be divided into serial and parallel method. For traditional serial method, there are Boolean based method and geometric based method. Nowadays, the geometric based one turns to be more powerful. Routing represents the final step in the CAD system. It allocates the FPGA's routing resource to interconnect the placed logic cells. It guarantees that all interconnections are connected. Besides, it should maximize the performance of time-critical connections. FPGA routing typically generates the Routing-Resource (RR) graph. Its target is finding a feasible routing to ensure all signals will not share the same nodes in routing network. In addition to the above mention, it is necessary to balance the tradeoff between optimizing delay for critical nets and finding the feasible routing for all nets.

2.2.1 Geometric Routing Algorithm and Rip-up and Re-route

The geometric routing algorithm is based on rip-up and re-route approach. It can easily describe the target architecture, but the disadvantage is that it is difficult to achieve routing solution. It consists of two steps which are checking if there any routing resource violations or timing violations such as wire delay and net delay. There are three path search methods: 1. Directed search method 2. Breadth first search method and 3. Depth first search method. For the breadth first search method, it minimizes the total wire length and the cost function focuses on the distance.

Whereas the cost function focuses on congestion and distance for depth first search function. The well-known Pathfinder algorithm adapts the above 2 methods into 2 steps which are initial routing and re-routing. The cost function Eq. 2.1. is:

$$\mathbf{Cost(n)} = \mathbf{b(n)} * \mathbf{h(n)} * \mathbf{p(n)} \tag{2.1}$$

where b(n) is the base cost, h(n) is the historical congestion and p(n) is the present congestion penalty.

2.2.2 Maze Routing Algorithm

The maze router adapts Lee Algorithm initially. The router is based on a wavefront expanding method that manages to find the shortest distance between two points and avoid overused routing resources. Basically, this algorithm iteratively routes, rips up and re-routes to eliminate the congested channels. One of the main advantages guarantees that the path can be found if it exists between two points. Besides, the path is surely the shortest available one. For the principal disadvantage of maze routing, it does not take the subsequent nets into account which doing the routing, which means that the different orderings of nets will deeply affect the performance of algorithm.

It can be decomposed into three phases. The first phase is called propagation phase. It will label the un-occupied routing resources with the distance from original resource to the current. The expansion will continue until propagating wave reaches the destination Sink. Therefore, it is unroutable if there are no un-occupied routing resources to expand and it has not reached the destination cell.

The second phase is called Retracing. It performs a backtracking from Sink node to Source node with numbers labeled decrementing. And the third phase is clearing the numbered routing resources in previous step except those on the chosen path. Those chosen routing resources will become blocking in the next propagation stage for seeking other targets.

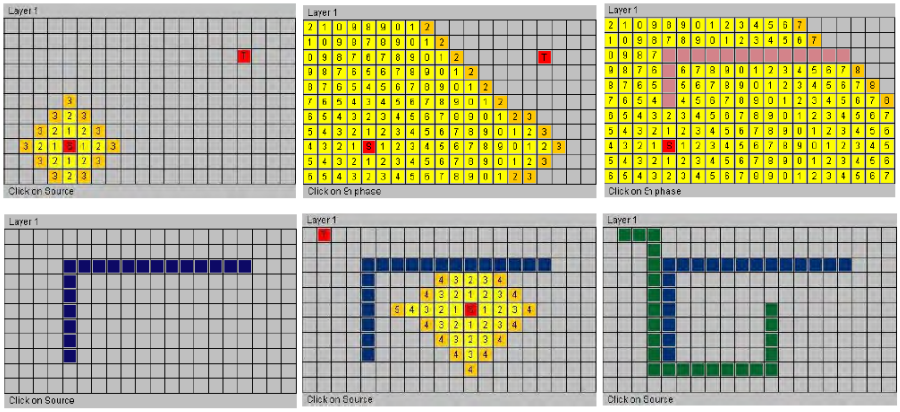


Figure 2.4: Maze expanding flow

2.2.3 A* Search Routing

The maze routing is a special case of A* routing which allows tuning the search path from a breadth-first search (BFS) into a short depth-first search (DFS). The BFS is an exhaustive algorithm that considers all possible paths and manages to find the optimized one while it can be slow to process; in contrast, the DFS may not get the minimum cost but it is faster.

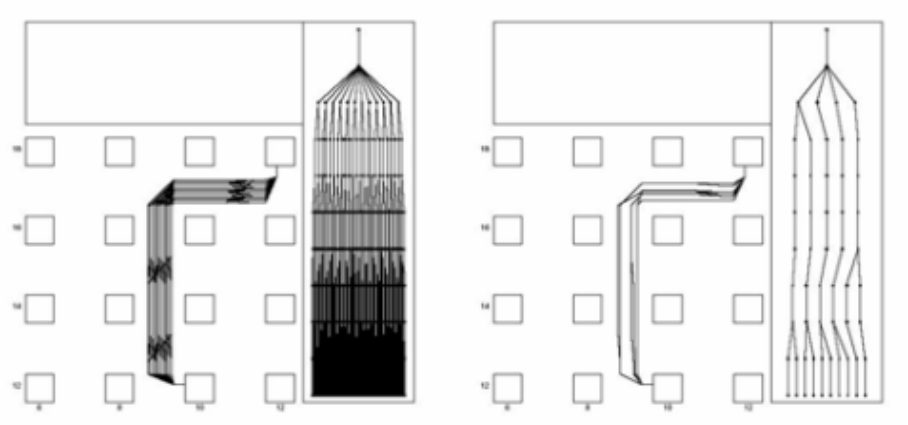


Figure 2.5: A* search algorithm

In general, routing resources in FPGA and the interconnections inside can be represented by a graph e.g. G = (V,E) where V represents the routing nodes and E represents the interconnections between wires or switches. In addition, each node has

a corresponding cost, c_i, which represents the current occupancy. For the successfully routing, each node cannot be occupied more than one net. A* algorithm considers a function Eq. 2.2. at each node V in the partial route from Source node to Sink node as below:

$$\mathbf{f_i} = \mathbf{g_i}(\mathbf{1}) \tag{2.2}$$

where g_i is the cost of the path from the source through V, d_i is the estimated cost of the path from V to destination. g_i is represented in maze routing algorithms as the total cost of the previous path f_{i-1} plus the cost of the next candidate node or Eq. 2.5.:

$$\mathbf{g_i} = \mathbf{f_{i-1}} + \mathbf{c_i} \tag{2.3}$$

where c_i is the node cost and represents the current usage of the node and it is used to record the nodes occupied by previous routing iteration; f_{i-1} is the total cost of previous path. As the BSF consumes more time, a sub-optimal but much faster approach is performed by using a scaling factor, alpha, which is between 0 and 1.

$$\mathbf{f_i} = (\mathbf{1} - \mathbf{alpha}) * (\mathbf{f_{i-1}} + \mathbf{c_i}) + \mathbf{a} * \mathbf{d_i} \tag{2.4}$$

2.2.4 The Pathfinder

The pathfinder algorithm is based on the maze router, but it speeds up the algorithm by routing every node on a no restricted environment and allowing to overuse routing resources. The cost function implemented by the pathfinder is shown below:

$$\mathbf{f_i} = (\mathbf{1} + \mathbf{h_n} * \mathbf{h_{fac}}) * (\mathbf{1} + \mathbf{p_n} * \mathbf{p_{fac}}) + \mathbf{b_{n,n+1}} \tag{2.5}$$

where b is the penalty of bend wires, p_n is the cost to use a particular wire, h_n is the historical value to track the wires used during previous iterations, and f_{fac} and p_{fac} are the weighting factors. It will continue ripping up and re-routing the nets until there is no overuse of routing resources.

2.3 Versatile Place and Route (VPR)

Mapping a circuit into an FPGA architecture is called CAD process. The CAD process consists of several steps:

- Logic optimization: it performs several layer of minimization for Boolean equations to optimize crtical delay and area.
- Mapping: it transforms from Boolean equations into a circuit of FPGA logic blocks. Besides, it optimizes the number of CLB required (area optimization) and the critical path time (delay optimization).
- Placement: it places the elements to the specific location on the FPGA, and manages to minimize the length of interconnection.
- Routing: it connects those specific logic blocks which are placed at previous step with available routing resources.

VPR is a versatile placement and routing tool for array-based FPGAs. Especially for the research, it is frequent used to evaluate and experiment the utility of new architecture. And benchmark the circuits by mapping, placing and routing onto the FPGA architecture, and then evaluate the performance and quality such as area and path delay. Therefore, it is extremely necessary to obtain a CAD tool with flexibility to support such a variety of FPGA architectures. In addition, VPR has a versatile routers in FPGA CAD tool as it allows to describe a variety of FPGA architecture targets to speed up the comparisons within different architectures. It outperforms than other current FPGA place and route tools in minimizing routing area. Although the algorithms used are the well-know ones, it enhances the performance to improve the quality.

As Fig. 2.6 shown below, it outlines the VPR CAD flow. The input file consists of a netlist file and a text file, which describes the FPGA architecture such as the number of input and output pins on logic blocks, the sides of logic block that each input and output is accessible, the number of I/O pads that fit into row or column of the FPGA, and the dimensions of the logic block array. In addition, the relative widths of vertical and horizontal channels can be specified if global routing is used. Finally, the type of

switch box, the Fc value for logic block inputs/outputs and I/O pads can be specified as well if combined global an detailed routing is performed. VPR can place the circuit or load a pre-existing placement file. The router is based on the modified version of Pathfinder algorithm, and it provides two different objectives-oriented methods to routing the placement in previous stage.

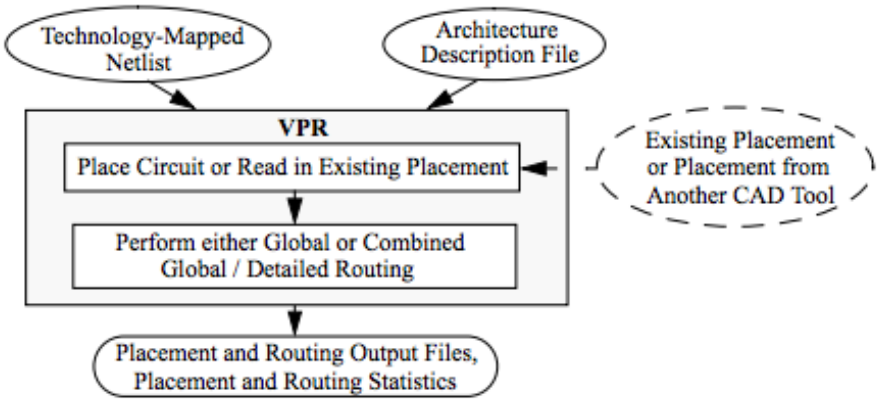


Figure 2.6: CAD flow

- VPR routability-driven router

For the routability-driven router, its primary goal is successful routing with the minimum tracks used. To achieve this, it incorporates with a modified routing cost function as shown in :

$$\mathbf{cost_n} = \mathbf{b_n} * \mathbf{h_n} * \mathbf{p_n} + \mathbf{bend_{n,m}} \tag{2.6}$$

b_n : the base cost, the range is usually between 0.95 or 1 for most routing resources and 0 for sinks. The reason for sink is 0 is used to prevent the router from continuing searching for possible connections within expanding wavefront step if the sink is already reached. $bend_{n,m}$: the penalty for bending the wire. (It is only used in global routing part.) p_n : the present congestion. It is the congestion penalty which is the difference between the number of a wires that can be used on that channel and the number of nets using the channel. It is re-calculated within every iteration to avoid over-subscribing any channel. Its value is given by:

$$\mathbf{p_n = 1 + max(0, [1 + occupancy_n - capacity_n] * p_{fac})} \tag{2.7}$$

p_{fac} : a weighting factor. It equals to 0.5 in the first iteration and multiply 1.5 to 2 in the subsequent iterations. h_n : the historical congestion penalty. It tracks the previous cost of routing resources and to avoid using it in the subsequent iteration.

$$\mathbf{(h_n)^i = (h_n)^{i-1} + max(0, [1 + occupancy_n - capacity_n] * h_{fac})} \tag{2.8}$$

h_{fac} : a constant weighting factor between 0.2 to 1. It is similar to p_{fac} while it is computed for historical congestion penalty.

- VPR's timing-driven router

The goal of timing-driven router is reducing delay time on the circuit. To achieve this, it adds an Elmore delay model to cost function. Therefore, it gives preference to the solutions with less delay. It starts with setting up an upper bound of delay, and then it routes the farthest nets. The imposed ordering on routing produces suboptimal track counts, and faster results. A common feature for both routers is that global route divides the k-terminal nets into k-1 steps, and it iterates k-1 times wavefront-expanding to connect each terminal. The standard maze router empties the current wavefront, while VPR router adds all the routing segments so far into heap, and it continues expanding the wavefront with a cost of 0. Therefore, it will reach the rest of terminal much more fast than if entire wavefront has to expand from scratch. After successful placement and routing, the output files contain placement, routing result and useful statistics to evaluate which kinds of architecture on FPGA is more suitable. It can support island-style and row-based FPGAs now.

2.4 Python-graph Library

Python-graph is a library (containing a set of APIs) for working with graphs in Python. It provides suitable data structure for representing graphs and an implementation of important algorithms. It is a pretty useful development environment

for developers interested in exploring graph algorithms for data flow graph (DFG) analysis. The most important feature which we have used is *digraph* class and the APIs are listed in Fig. 2.7.

Instance Methods	
boolean	__eq__(self, other) Return whether this graph is equal to another one.
	__init__(self) Initialize a digraph.
boolean	__ne__(self, other) Return whether this graph is not equal to another one.
	add_edge(self, edge, wt=1, label='', attrs=[]) Add an directed edge to the graph connecting two nodes.
	add_node(self, node, attrs=None) Add given node to the graph.
	del_edge(self, edge) Remove an directed edge from the graph.
	del_node(self, node) Remove a node from the graph.
list	edges(self) Return all edges in the graph.
boolean	has_edge(self, edge) Return whether an edge exists.
boolean	has_node(self, node) Return whether the requested node exists.
list	incidents(self, node) Return all nodes that are incident to the given node.
list	neighbors(self, node) Return all nodes that are directly accessible from given node.
number	node_order(self, node) Return the order of the given node.
list	nodes(self) Return node list.

Figure 2.7: Useful APIs for DFG processing

Using the set of APIs provided by this library, we developed a set of python modules containing implementation of frequently used graph scheduling algorithms. We implemented As Soon As Possible (ASAP), As Late As Possible (ALAP) and List Scheduling algorithms, described later in this report.

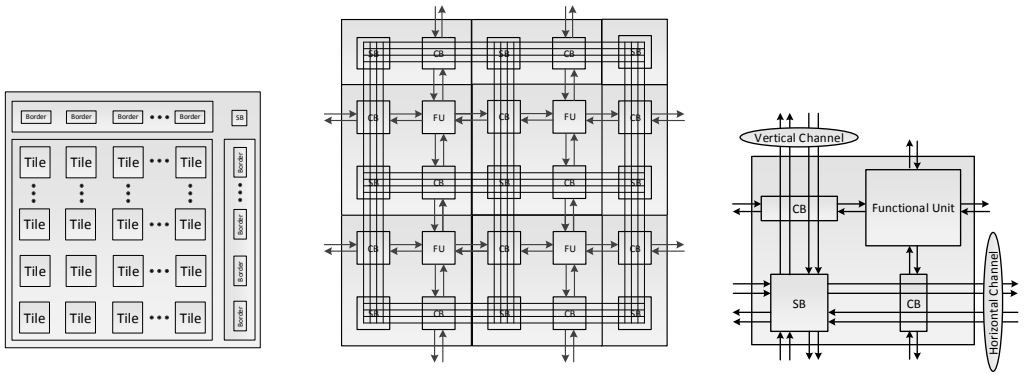
Chapter 3

Placement and Routing of DFG nodes on Island-style Overlay

In this chapter, we firstly introduce an island-style coarse-grained overlay architecture (published previously in [17]) and then describe the placement of input DFG on to the overlay using placement algorithm used in VPR tool. Then we also explain how to route the DFG edges on the interconnection architecture and the implementation of routing algorithm used in the VPR tool. Finally, We describe our python implementation of the placement and routing algorithm.

3.1 Island-style Overlay Architecture

The island-style overlay instantiates the tiles where instantiates routing resources and borders where instantiates one switch box (SB) and one connection box (CB), forming the boundary at the top and right arrays, as shown in Fig. 3.1(a). The I/O, which is contained around the periphery of the overlay fabric, can be connected to a FIFO port. Fig. 3.1(b) shows an 2×2 overlay architecture having 4 tiles, two north borders, two east borders and one northern east corner. It consists 4 FUs, 9 SBs and 8 CBs which means that an $N \times N$ overlay has N^2 FUs, $(N+1)^2$SBs and $N^2 + 2 * N$ CBs. Within each tile, it contains a functional unit (FU) and some routing resources, as shown in Fig. 3.1(c).



(a) Overlay block diagram. (b) Architecture of a 2×2 overlay. (c) Tile architecture.

Figure 3.1: Overlay architecture.

The interconnection resources used in routing DFG edges are explained below.

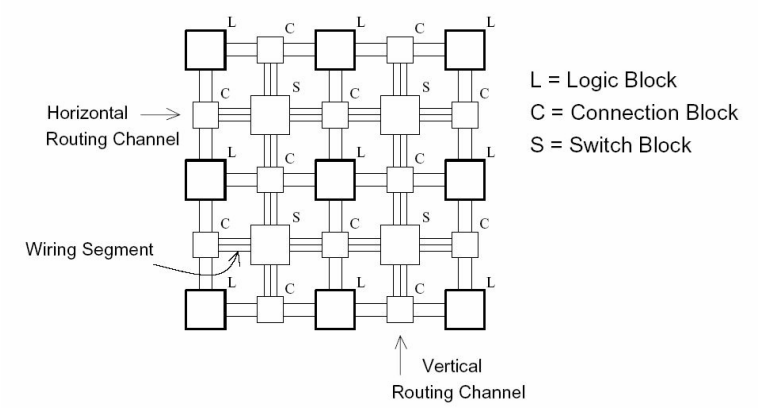


Figure 3.2: High level architecture showing interconnect resources.

As the Fig. 3.2. shown above, the main elements used in routing process are switch boxes, connection boxes and channels.

3.1.1 Switch Boxes

For the routing interconnection between different logic blocks, switch box is an important element as it is responsible for connecting with another direction (horizontal/vertical) track in other channels. It increases the flexibility and routability of FPGA. The flexibility Fs indicates for a segment that enters the switch block, the

maximum number of segments that it can connect to in the switch block. The different architecture of switch boxes can influence the results of routing. Fig. 3.3 shows the common topologies used for the switch box.

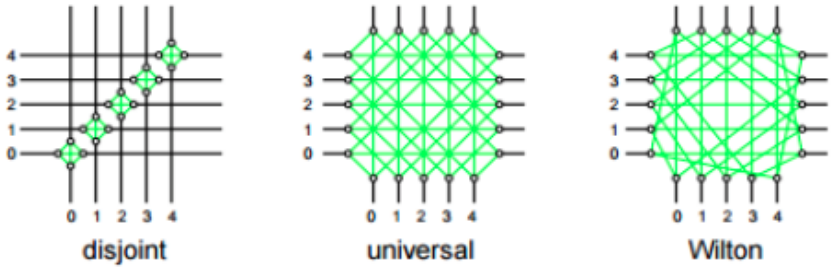


Figure 3.3: Switch Box Topologies

3.1.2 Connection Boxes

The connection box can connect the channels with all possible neighboring logic blocks. Fc represents the number of tracks in each channel to which each logic block input and output pin can connect. There are two different architectures of connection box which define the pattern of switches to connect to the tracks as shown in Fig. 3.4.

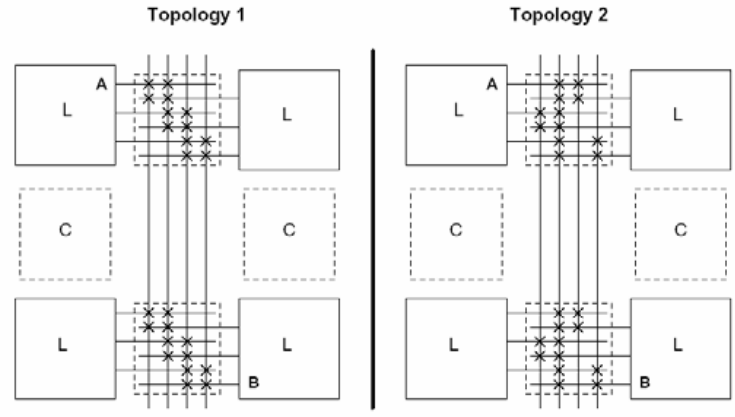


Figure 3.4: Connection Box

3.1.3 Channels

Channels contain tracks, which consist of horizontal and vertical directions. The main responsibility is connecting different pins of logic blocks after HDL is mapped on the overlay and placement is complete. W represents the width of channel or the number of tracks in each channel. Channel segments indicate that it can have various lengths or one size, which can only span one logic block. Besides, channels can be uni-directional and bi-directional. In our example, we assume that the segments can only span one logic block, and the channels are uni-directional.

3.2 Automated Mapping Tool

In this section, we base on an automated mapping tool (published previously in [17]), which allows mapping high level description (HLD) of computing kernels to the overlay.

Table 3.1: Compute Kernel Code Descriptions

(a) C description

```
#include<math.h>
#define SIZE 1000

int kernel(int x){
  int temp = 16*x;
  return (x*(x*(temp*x-20)x+5));
}

int main(void){
  int i;
  int in[SIZE];
  int out[SIZE];
  for (i=0; i<SIZE; i++){
    out[i] = kernel(in[i]);
  }
        return 0;
}
```

(b) DFG description

```
digraph kernel {
N8 [ntype="operation", label="add_Imm_5_N8"];
N9 [ntype="outvar", label="O0_N9"];
N1 [ntype="invar", label="I0_N1"];
N2 [ntype="operation", label="mul_N2"];
N3 [ntype="operation", label="mul_N3"];
N4 [ntype="operation", label="mul_Imm_16_N4"];
N5 [ntype="operation", label="mul_N5"];
N6 [ntype="operation", label="mul_N6"];
N7 [ntype="operation", label="sub_Imm_20_N7"];
N8 -> N2;
N1 -> N5;
N1 -> N6;
N1 -> N2;
N1 -> N3;
N1 -> N4;
N2 -> N9;
N3 -> N6;
N4 -> N5;
N5 -> N7;
N6 -> N8;
N7 -> N3;
}
```

3.2.1 Data Flow Graph (DFG) Generation

It starts with a C language description; the tool transforms C to a Data Flow Graph description in Table 3.1.

3.2.2 DFG to VPR Compatible Netlist Conversion

To be compatible with the VPR requiring input file, we use a netlist generator, which generates a VPR compatible netlist as shown in Table 3.2. Then we make use of VPR tool to place logic block onto the overlay and route the interconnections between them.

Table 3.2: PAR input File

Netlist description

```
.input N1
pinlist: N1

.output out:N7
pinlist: N7

.fu N2
pinlist: N1 N6 open open N7 open open open open
subblock: N2_blk 0 1 open open 4 open open open open

.fu N3
pinlist: N1 N5 open open N3 open open open open
subblock: N3_blk 0 1 open open 4 open open open open

.fu N4
pinlist: N1 open open open N4 open open open open
subblock: N4_blk 0 open open open 4 open open open open

.fu N5
pinlist: N1 N4 open open N5 open open open open
subblock: N5_blk 0 1 open open 4 open open open open

.fu N6
pinlist: N1 N3 open open N6 open open open open
subblock: N6_blk 0 1 open open 4 open open open open
```

3.2.3 Placement and Routing onto the Overlay

Mapping DFG nodes onto homogeneous function units and DFG edges to the overlay's routing resources to connect the mapped function units. At this level, a netlist is able

to have more than 100 nodes and make the problem much easier than that of fine-grained FPGA. The Fig. 3.6. displays a placement of DFG on a 5x5 Architecture.

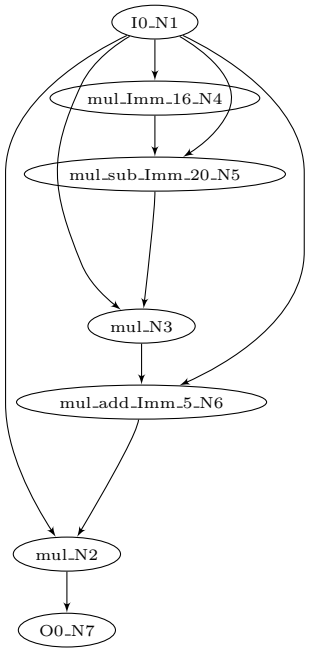


Figure 3.5: Data Flow Graph (DFG)

3.3 Detailed Description of the Placement

In VPR tool, it use simulated annealing algorithm mentioned in Chapter 2. It mimics the cooling process of the melting metal at a high temperature and gradually lowing temperature, which means the energy and defects are reduced. In this case, a high temperature is initially selected. It is a heuristic-based iteration for minimizing the costs. The cost function used here is linear congestion cost function, which mainly focuses on wire length and penalizes placements which requires more routing area on the overlay that has narrow channel.

$$\mathbf{Cost} = \sum_{\mathbf{n=1}}^{\mathbf{N_{nets}}} \mathbf{q(n)} \left[\frac{\mathbf{bb_x(n)}}{\mathbf{C_{av,x}(n)}} + \frac{\mathbf{bb_y(n)}}{\mathbf{C_{av,x}(n)}} \right] \tag{3.1}$$

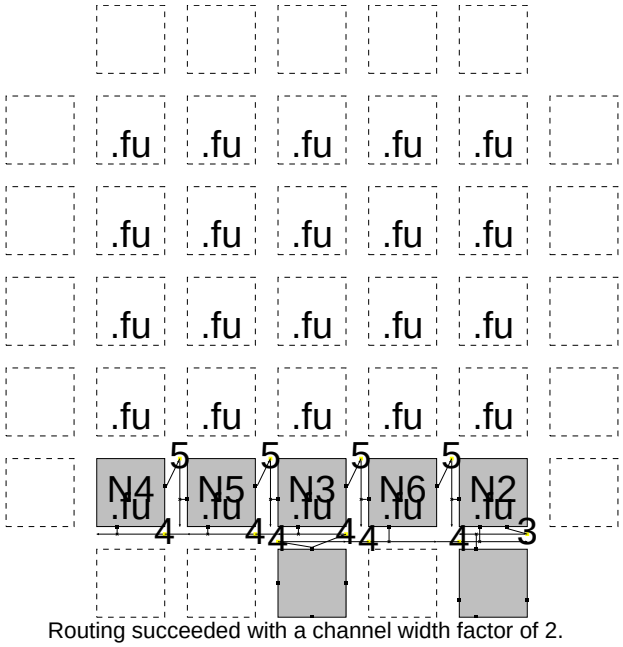


Figure 3.6: DFG mapped onto the Overlay after Placement and Routing.

Table 3.3: Net q(n) factor.

pin number	q(n)	pin number	q(n)
1∼3	1.0000	15	1.6899
4	1.0828	20	1.8924
5	1.1536	25	2.0743
6	1.2206	30	2.2334
7	1.2823	35	2.3895
8	1.3385	40	2.5356
9	1.3991	45	2.6625
10	1.4493	50	2.7933

$q(n)$: the factor depends on the number of terminals that a net has to connect. It becomes bigger than 1 for a net with more than 3 terminals. The $q(n)$ used here is , as suggested in Fig. 3.3.

$C_{av,x}(n)$ and $C_{av,y}(n)$: the constant average channel capacities (in tracks) in x and y direction over the bounding box. In our case, we assume that all channels have the same density of the tracks, so the $C_{av,x}(n)$ and $C_{av,y}(n)$ are the constant

value. Therefore, the complexity of linear congestion function reduces and becomes a bounding box based function. The total cost of the placement is the summation of the costs of each net. The cost for each net is the x-direction and y-direction distance of bounding box divided by $C_{av,x}(n)$ and $C_{av,y}(n)$ respectively The bounding box is defined as the maximum range of the x and y coordinates of the net spanning. For example, there are two coordinates (a, b) and (c, d) which are source and target of the net, then the equation of the bb_x and bb_y are shown in Eq. 3.2 and Eq. 3.3.

$$\mathbf{bb_x = x_2 - x_1 + 1} \tag{3.2}$$

$$\mathbf{bb_y = y_2 - y_1 + 1} \tag{3.3}$$

As doing the simulated annealing, a fixed number of swap are executed within certain temperature. The number of the swap is calculated as the following Eq. 3.4.

$$\mathbf{move\_lim = 10 * (N_{blocks})^{1.33}} \tag{3.4}$$

where Nblocks is the total number of logic blocks and I/O pads on a circuit.

3.3.1 Flow of the Placement

The placement flow in VPR starts with a random placement. The initial temperature is defined as 20 times of the standard deviation of cost that randomly swaps the Nblocks to ensure that it can accept any swap in the beginning of the simulated annealing, where Nblocks is the total number of the logic blocks and I/O pads on the FPGA.

After initially random placement, the nodes are placed on a $3x3$ overlay as shown on Fig. 3.7 and the cost is calculated by the linear congestion function, which is adding the cost of each net. For example, N1 is the input node in the DFG which is placed initially on the I/O block located at (0,2). N7 is the output node in the DFG which is placed initially on the I/O block located at (2,0). The initial placement shows 5 compute blocks, 2 I/O blocks and 6 nets.

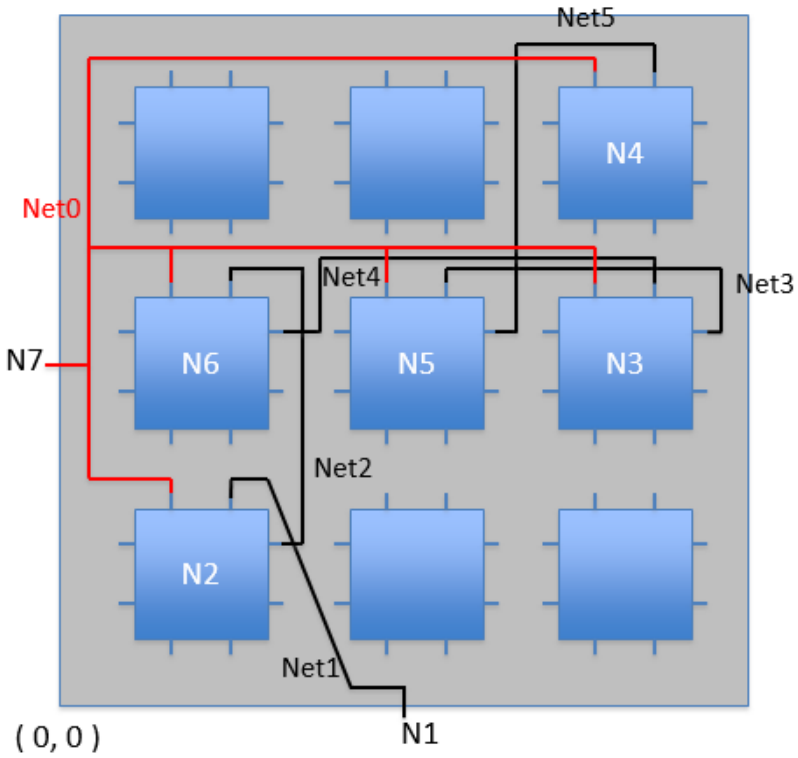


Figure 3.7: Initial Placement Connections.

The calculation of the cost of initial placement is shown as below: The initial placement shows 5 compute blocks, 2 I/O blocks and 6 nets. We use the initial temperature as 0.4 and evaluate a fixed number of moves at this temperature.

$$\left[\frac{\mathbf{bb_x(n)}}{\mathbf{C_{av,x}(n)}} + \frac{\mathbf{bb_y(n)}}{\mathbf{C_{av,x}(n)}}\right] \tag{3.5}$$

Net_0 bb_coor [(1,1), (3,3)] Net_1 bb_coor [(1,1), (2,1)] Net_2 bb_coor [(1,1), (1,2)] Net_3 bb_coor [(2,2), (3,2)] Net_4 bb_coor [(1,2), (3,2)] Net_5 bb_coor [(2,2), (3,3)]

The cost of the initial placement is calculated: 0.073236 + 0.03 + 0.03 + 0.03 + 0.04 + 0.04 = 0.2432

The iteration of swap starts with temperature = 0.4 and it executes 133 times

before reducing the temperature. Each swap randomly selects and moves a logic block to another random location on the FPGA. After that, the new cost is calculated on linear congestion function for this new placement, and then compare to the previous cost. The new cost will be accepted if it is less than the previous one, and this happens frequently especially when temperature is higher. However, if the new cost is larger than the previous one, the Eq. 3.6 is used to decide whether accepting the swap or not. As the equation shown, it is oppositely related to the temperature, which means that higher temperature have higher probability to accept even the cost increases. Therefore, it is the approach to avoid getting stuck at a local minimum. When it comes to the lower temperature, the percentage of accepting swap is obviously lower.

$$\mathbf{if}(\mathbf{random}(\mathbf{0},\mathbf{1}) < \mathbf{e}^{-\mathbf{delta}*\mathbf{cost}/\mathbf{T}}) \tag{3.6}$$

The placement is optimized after hundreds and thousands of iterations . Fig. 3.3 is the example to demonstrate the few random swap at temperature = 0.4.

In the first iteration (iteration 0), the N4 block is selected and swapped to random location from (3,3) to (2,3). The new random location is empty. As Net0 and Net5 connect N4, the bounding box of these two nets is changed after N4 moves to new location and new cost as well. Therefore, it will affect the total cost of the placement. If the net change is less than 0, the swap is accepted absolutely. However, the swap is assessed by the probability factor calculation and compared to a random generated number, which is between 0 and 1 as shown below. In the second iteration (iteration 1), N4 is selected again and the new location is randomly selected as (2,1), which is empty too. After updating the bounding box, the delta of two costs is negative. Then the swap is accepted again. In the third iteration (iteration 2), N5 is selected and the new randomly selected location is (3,2), which is occupied by N3. It means that nets to be considered are those connect to either N3 or N5. Thus, Net0, Net3, Net4 and Net5 are updated and as well as their bounding box. The delta after swapping is positive, so the probability factor is used to assess the move. And it is accepted in this case. The above example just simply demonstrates the few iterations of the placement process by using the simulated annealing algorithm. The temperature

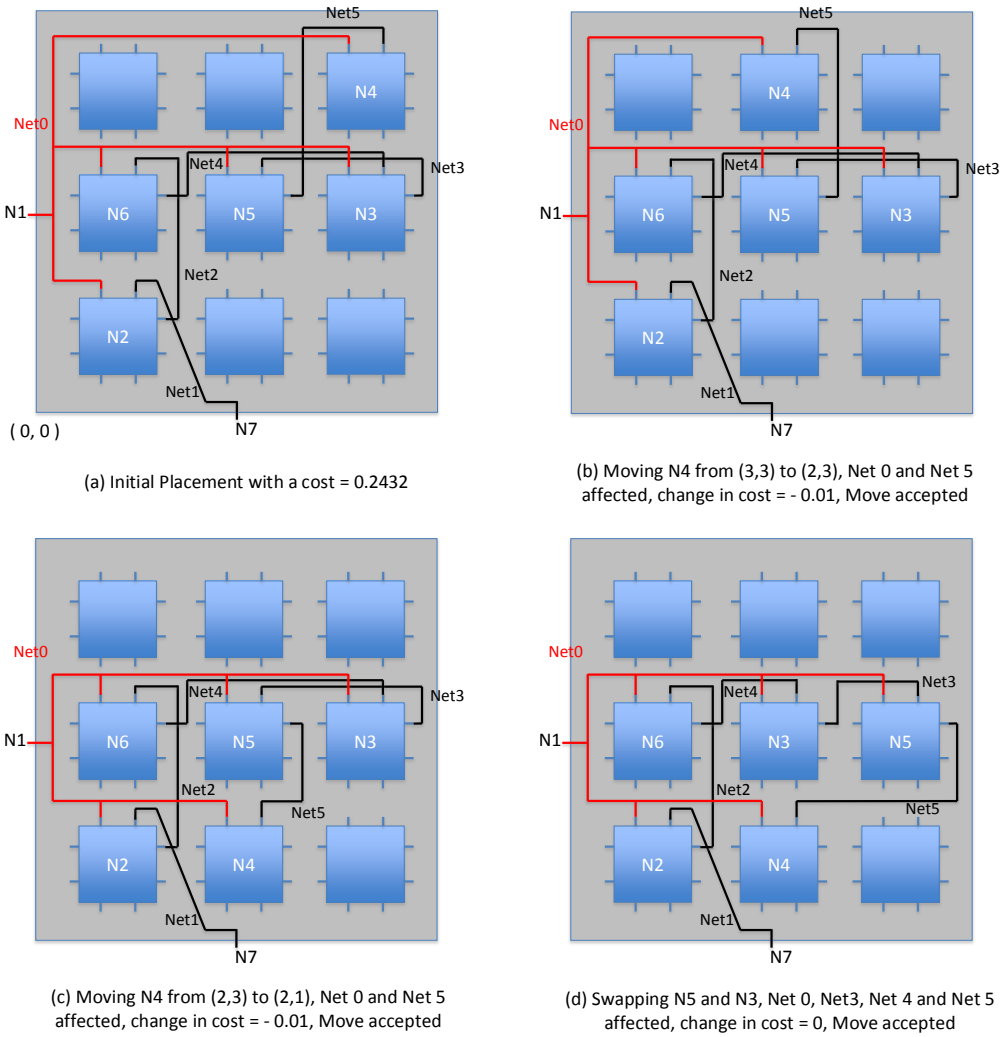


Figure 3.8: Placement Iterations at a Given Temperature

will be updated after the number of iteration reaches the move_lim. And the overall process terminates when the T ¡= 0.005 × Cost/Nnets.

The Table 3.4 shows the manual trace of Moves.

Table 3.4: Evaluation of Moves at a Given Temperature

Cost Calculation

```
Iteration 0 :
Swap N4(3,3) <-> empty (2,3).
Nets to be updated: Net 0, Net 5
Net 0 bbCoor = [ ( 1,1), (3,3) ] Temp Cost=1.2206*(3-1+1)/100
+1.2206*(3-1+1)/100=0.073236
Delta=Temp Cost-N0Cost =0.073236-0.073236=0
Net 5 bbCoor = [ ( 2,2), (2,3) ]
Temp Cost=1*(2-2+1)/100 +1*(3-2+1)/100=0.03
Delta=Temp Cost-N5Cost = 0.03-0.04 = -0.01  DeltaTotal=0+(-0.01)=-0.01
DeltaTotal < 0 -> Accept

Iteration 1:
Swap N4 (2,3) <-> empty (2,1)
Nets to be updated: Net0, Net5
Net0 bbCoor [ ( 1,1), (3,2) ]
TempCost=1.2206*(3-1+1)/100 +1.2206*(2-1+1)/100=0.06103
Delta=TempCost-N0Cost = 0.06103-0.073236 = -0.012206
Net5 bbCoor [ ( 2,1), (2,2) ]
TempCost = 1*(2-2+1)/100 +1*(2-1+1)/100=0.03
Delta = TempCost-N5Cost = 0.03-0.03=0
DeltaTotal=-0.012206
DeltaTotal <0 => Accept

Iteration 2:
Swap N5(2,2) <-> N3(3,2)
Nets to be updated: Net0, Net3, Net4, Net5
For Net0, Net3, the cost will be the equivalent as bbCoor is same.

Net4 bbCoor [ ( 1,2), (2,2) ]
TempCost = 1*(2-1+1)/100+1*(2-2+1)/100=0.03
Delta=TempCost-N4Cost =0.03-0.04=-0.01
Net5 bbCoor [ ( 2,1), (3,2) ]
TempCost=1*(3-2+1)/100 +1*(2-1+1)/100=0.04
Delta=TempCost-N5Cost =0.04-0.03=0.01
DeltaTotal = -0.01+0.01 = 0
prob_fac = exp(-0/0.4) = 1 > 0.5 -> Accept
```

3.4 Detailed Description of the Routing

The DFG is mapped onto homogeneous functional units and needs to be connected using the interconnect architecture. The main elements of the interconnect architecture are channels, connection boxes and switch boxes. For each logic block, it consists of 4 RECEIVER pins, 4 DRIVER pins and one Global RECEIVER pin. For each channel, it consists of two unidirectional tracks, which mean the width of channel is 2. Switch box flexibility F_s = 3, which means any track can connect to 3 neighbor channels except those corner cases. Connection box flexibility F_c = 1. This is an

ideal case that greatly simplifies the difficulty of routing task.

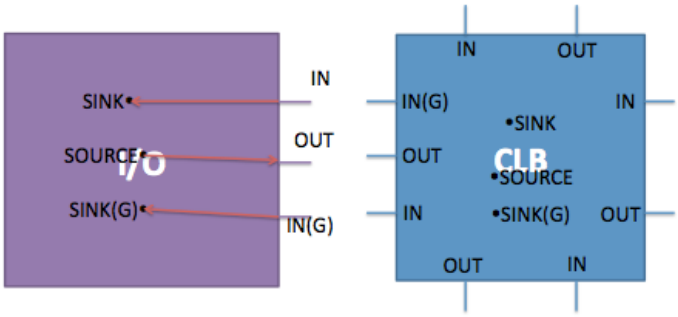


Figure 3.9: IO block and Configurable Logic Block

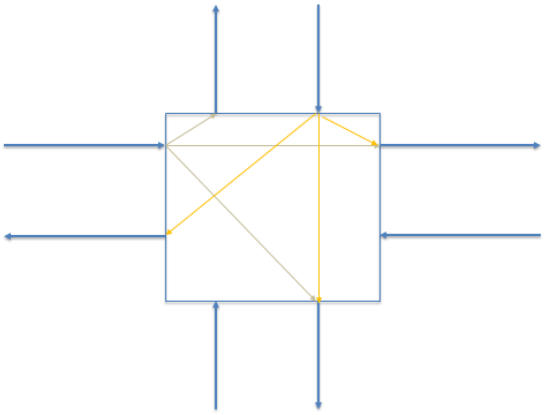


Figure 3.10: Switch Box Uni-Directional Routing

Routing is used to connect the available routing resources with the CLBs and I/Os which are distributed on the FPGA after placement process, transiting signals from where they are generated to the blocks where going to use. An ideal routing algorithm should lower the wire routing area and the critical path of nets to improve the performance of circuit. Generally, there are two steps in the routing process that are global and detailed routing. The purpose of global routing is balancing the congestion on channels. It begins with routing each net according to the lowest

cost regardless of the congestion caused. And the algorithm will balance out the congestion after iteratively re-routing the nets. After global routing, the detailed routing aims to build a directed graph from the routing resources to represent the available interconnection between tracks, input pins, output pins, Switch boxes and logic blocks on the FPGA. There is another special case that the routing process can be complete within one step detailed routing that is the routing directed graph is constructed initially, and each node (e.g. Sink, Source, track_x and track_y) on the graph has own cost function. The routing process is searching sink node(s) from source node(s) based on the cost.

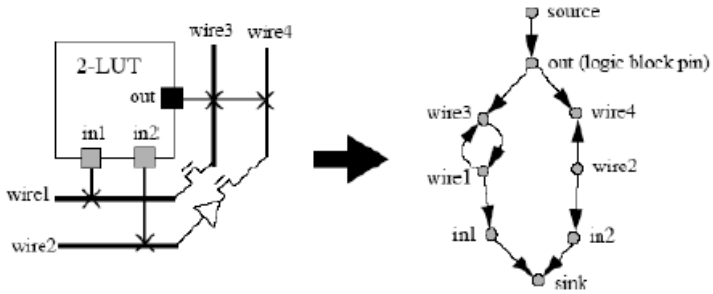


Figure 3.11: Routing Resource Graph

The algorithm used to find the shortest path between two nodes is based on Dijkstra's algorithm (ie. Maze router). And the VPR tool implements the Pathfinder Negotiated Congestion algorithm. The pathfinder algorithm initially routes each net by expanding wavefront technique to find out the shortest path regardless of any overuse of wiring segments or logic block pins, which is based on the concept of maze router. The cost function for each routing resource is calculated on the current overuse of routing resources and any prior overuse of routing resources that occurred in previous routing iterations. Therefore, the cost of over-used resources will gradually increase in the subsequent iterations and the algorithm will find an alternative route to avoid those unfavorable resources. Subsequently, it iteratively rips up and re-routes all nets until there is no overuse of routing resources exist. Besides, it solves the drawback that occurs in maze router that is performance is dependent on net ordering. The VPR router expands wavefront in an efficient way,

which adds the routing resources routed so far into the wavefront firstly and continue expanding normally with a cost of 0. For example, the maze router is called k-1 times to execute the overall expanding for a net with k terminals. In the first invocation, the wavefront expands from the Source node until it reaches any one of the Sink nodes. Normally, the router will empty the wavefront and start the next iteration with the empty wavefront. However, the maze router will start from the wavefront in the previous iteration. As a result, the net will be routed much faster.

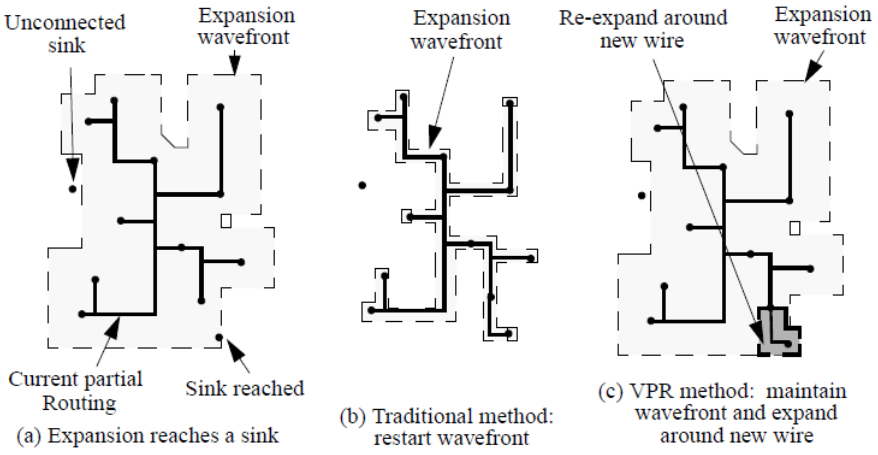


Figure 3.12: Wavefront graph

The Fig. 3.5 shows the important structs used in routing process. We simply explain the terminology which will be used in following contexts.

3.4.1 Initialization of the Routing

The routing process in VPR tool starts with creating the routing graph. After the routing graph is constructed, the graph is shown in Fig. 3.13. It consists of CLBs, I/Os, switch boxes, and channels. The number represents the index of each rr node on the routing graph. The index can be tracked as the reference node.

Table 3.5: Routing resource struct

Variable	Meaning
t_rr_type type	{ SOURCE, SINK, IPIN, OPIN, CHANX, CHANY}
short num_edges	Number of edges exiting this node.That is, the number of nodes to which it connects.
int *edges	Array of indices of the neighbours of this node.
int prev_node	Index of the previous node used to reach this one; used to generate the traceback.,If there is no predecessor, prev_node = NO_PREVIOUS.
short target_flag	Is this node a target (sink) for the current routing Number of times this node must be reached to fully route.

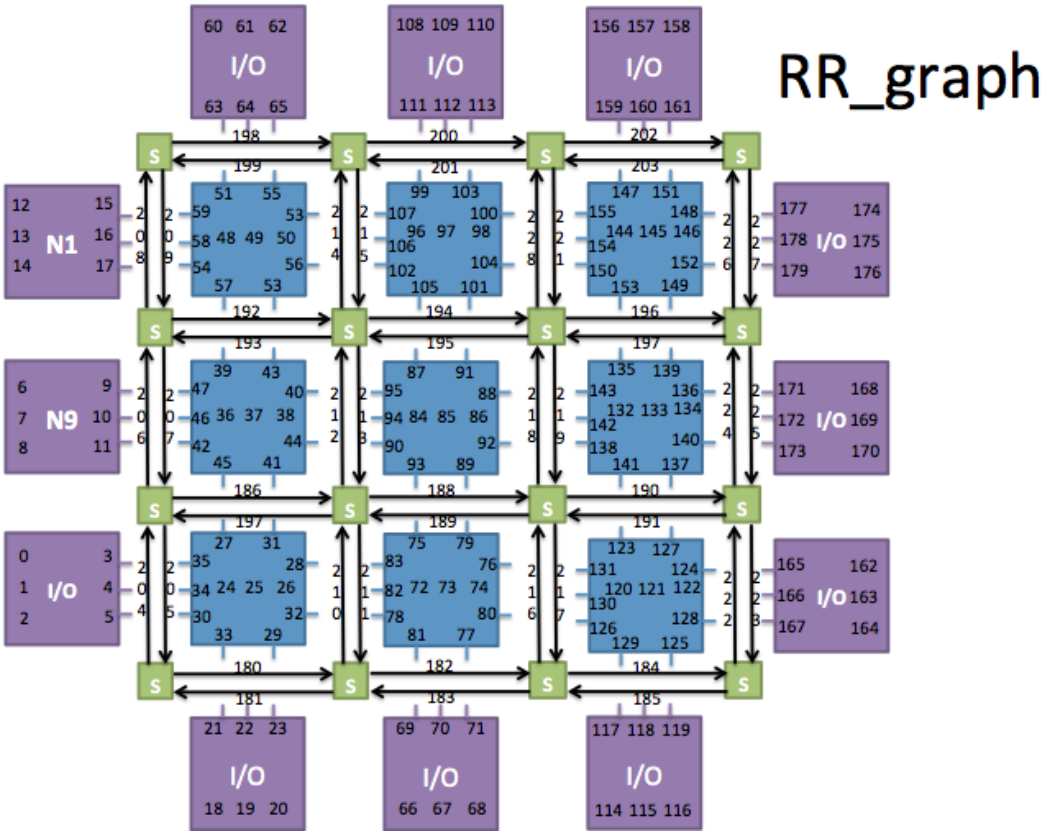


Figure 3.13: RR graph after initialization

3.4.2 Cost Functions in Routing

The Pathfinder Negotiated Congestion algorithm uses of the current routing as well as the routing history in previous routing when calculating the cost functions. Each node has an unique cost. The cost of node is indicated below:

- pres_cost: The present congestion cost term for this node. Occ represents the number of nets, which occupy this node. Capacity is the maximum utilization allowed for each node to be occupied. (It is updated in each iteration after each net.) If Occ >Capacity, Eq. 3.8 is used. Otherwise, Eq. 3.7 is uesd.

$$\mathbf{pres\_cost} = \mathbf{1} \tag{3.7}$$

$$\mathbf{pres\_cost} = \mathbf{1}. + (\mathbf{occ} + \mathbf{1} - \mathbf{capacity}) * \mathbf{pres\_fac} \tag{3.8}$$

- acc_cost: Accumulated cost. (It only updates after all nets have been routed.)

$$\mathbf{acc\_cost} += (\mathbf{occ} - \mathbf{capacity}) * \mathbf{acc\_fac}; \tag{3.9}$$

* pres_fac: The sharing penalty; it is multiplied by a factor in each routing iteration.

* acc_fac: Historical congestion cost multiplier. If the occ >capacity then the acc_cost is increased according the delta between occ and capacity, which is multiplied by the acc_fac.

- Congestion cost of using this node

$$\mathbf{Congestion\_cost} = \mathbf{base\_cost} * \mathbf{acc\_cost} * \mathbf{pres\_cost} \tag{3.10}$$

base_cost: The basic cost of using an rr_node.

- Total path cost: The total cost of the path up to and including this node plus the expected cost to the target.

$$\mathbf{path\_cost} = \mathbf{new\_back\_pcost} + \mathbf{astar\_fac} * \mathbf{expected\_cost\_to\_target}() \quad (3.11)$$

Table 3.6: Determines the expected cost to reach the target from current

```
if(rr_type == CHANX || rr_type == CHANY)
{
          ...Some calculation ...
          return (cong_cost);
}
else if(rr\_type == IPIN)
{
          return base_cost;
}
else
{
        return 0;
}
```

$$\mathbf{new\_back\_pcost} = \mathbf{old\_back\_pcost} + \mathbf{congestion_cost}(\mathbf{new\_node}) \quad (3.12)$$

where new backward_path_pcost is the summation of old backward_path_cost in previous step and congestion cost of new nodes.

3.4.3 Flow of the Routing

After initializing the routing graph, the next step is creating the minimum spanning tree (MST) for each net prior to running the algorithm. It contains the Source node and all Sinks node of the net, which is a spanning tree of a connected, undirected graph. Taking N1-net for example, Source node is N1_blk pin1 and the Sink nodes are N2_blk pin0, N3_blk pin1, N4_blk pin0, N5_blk pin0 and N6_blk pin0. In the first invocation, it finds out the nearest sink from source and records them in minimum spanning tree. Then it finds out the nearest distance between the minimum spanning tree and the sink, which is not recorded yet. After finding the nearest distance, it records the pair of two nodes into minimum tree. It iterates the invocations until all sinks are stored in the tree.

After creating the mst tree, it starts a loop of 50 iterations, which routes each

net by using directed search algorithm. As it begins with allowing to route overused resources. Therefore, it is necessary to check if the routing is feasible or not. As our implemented cost function, the cost of overused node will be gradually updated after several iterations. And if a successful routing cannot be found within 50 iterations, the routing will be failed.

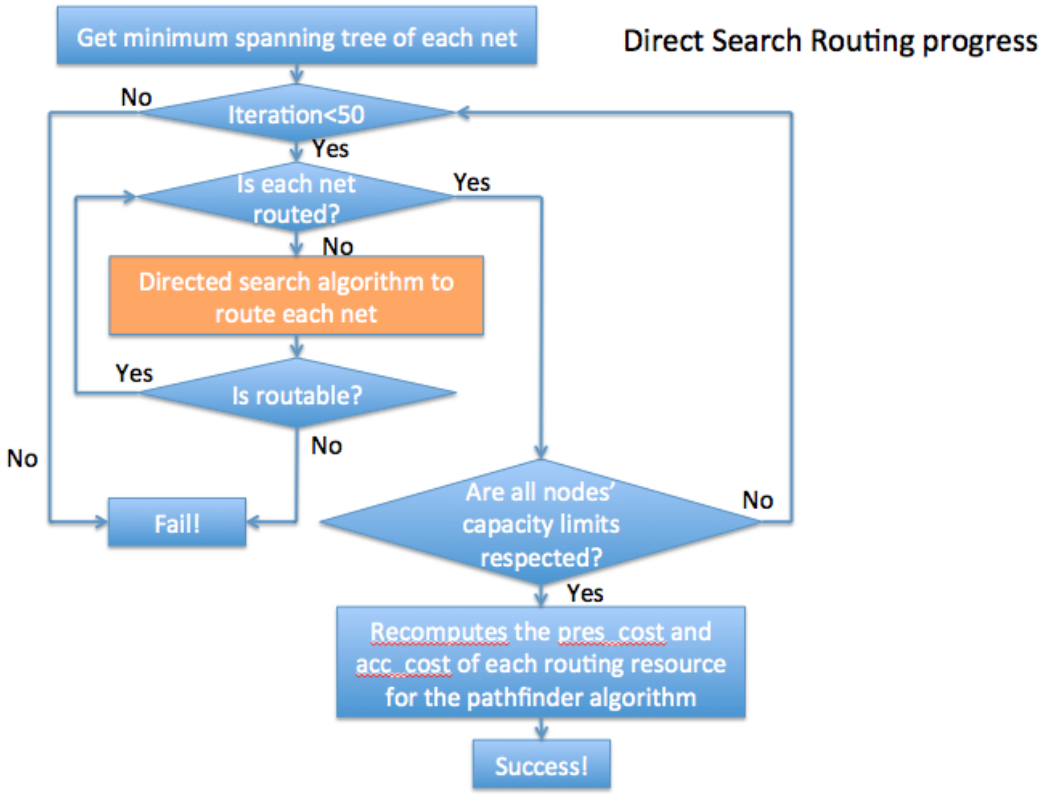


Figure 3.14: Direct Search Routing progress

3.4.4 Directed Search Algorithm in the Routing

There are two data structures used in the Directed search algorithm.

The Directed search algorithm is about to route each net by expanding the wavefront from Source node to Sink node(s). As we mentioned before, it uses the maze router with an efficient fanout method. Basically, it iterates number of Sink-1 times to reach all Sink nodes. It restarts the expanding from the previous wavefront, which

Table 3.7: Trace and Binary Tree struct

Data Structure	Description
Trace	Store the traceback (routing) of each net.
Binary heap	Store the potential node to be routed in order of the cost, so the head of the heap is the node with the minimum cost.

includes all wiring segments routed so far by adding all of these nodes into the binary heap. After it completely routes an individual net, it updates these connected nodes into the trace, frees the heap and updates the present congestion cost before next iteration.

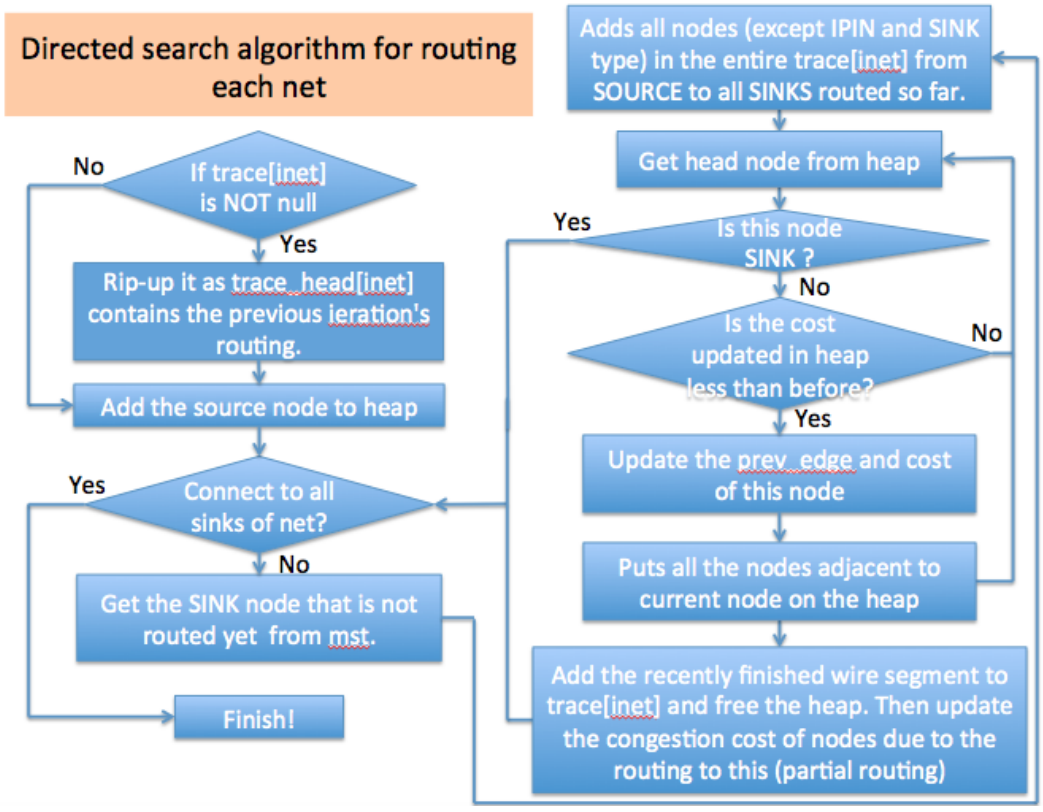


Figure 3.15: Directed search algorithm for routing each net

3.4.5 Routing Demonstration

ATake Net_2 for example, the Source node is N8_blk and the Sink node is N2_blk. Fig. 3.16 to Fig. 3.31 demonstrate the process of updating the cost function step by step when expanding wavefront from Source node to Sink node. Starting with calculating the cost of Source node, and it iteratively calculates neighboring nodes which are adjacent to the nodes which are calculated in previous step until reaching the Sink node.

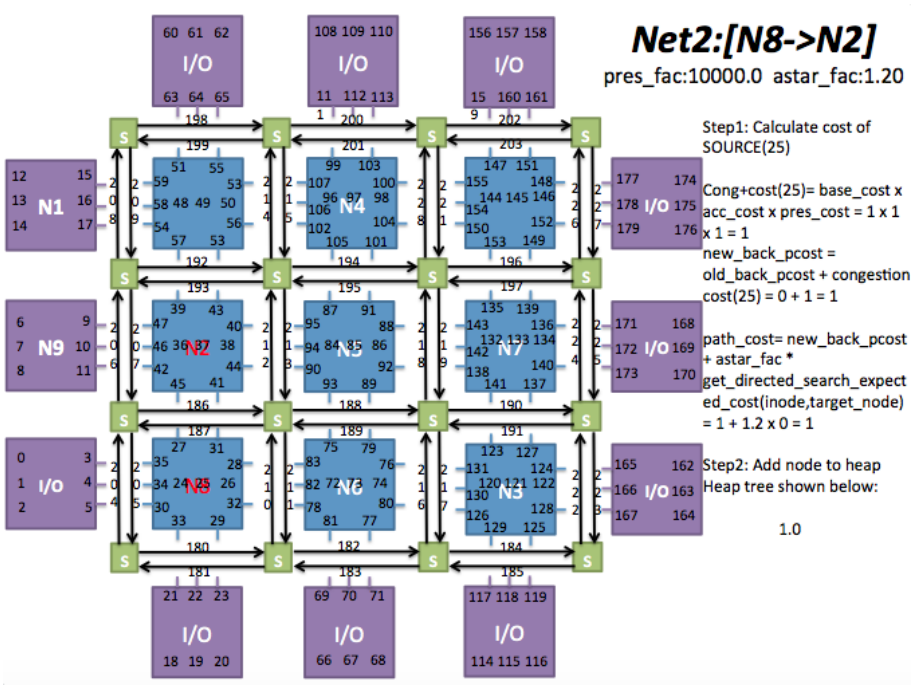


Figure 3.16: Routing process step 1

3.5 Fault Tolerance

After implement the placement and routing algorithms into Python. We implemented the Fault tolerance property in the placement step. Fault tolerance is a methodology that ensures that a system can continue working properly even failures occurred in the components. The severity of failure is proportional to system quality compared to

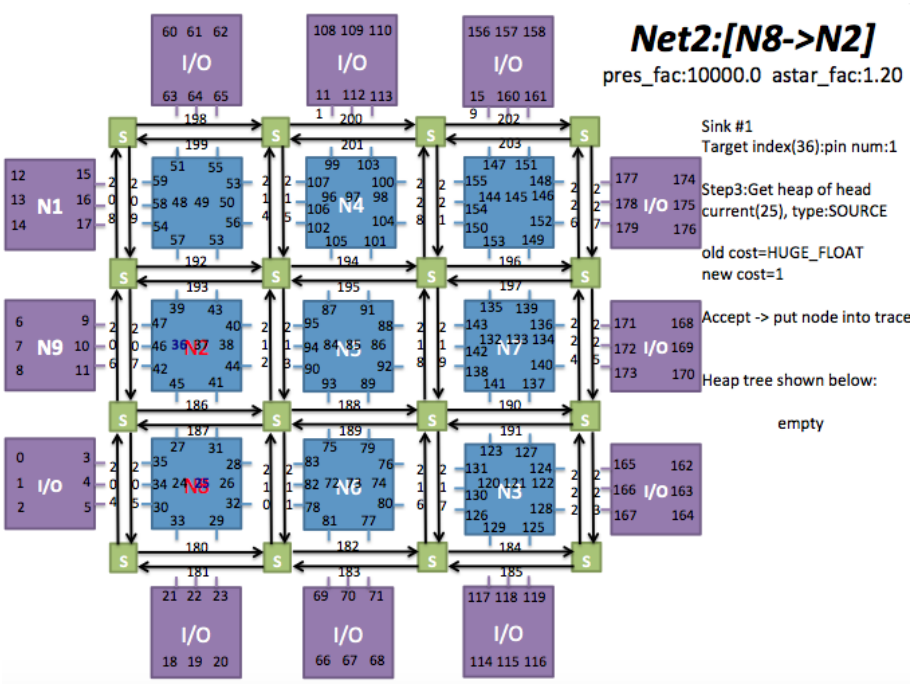


Figure 3.17: Routing process step 2

the normal system that a small failure may breakdown whole system. Its advantage enables a system keep operating, possibly in a degraded level, but can still execute. As the structure of FPGA is homogeneous, our implementation enables developer do the placement but avoid some selected logic blocks to be placed. As a result, the FPGA can be used and tested even there are some resources broken.

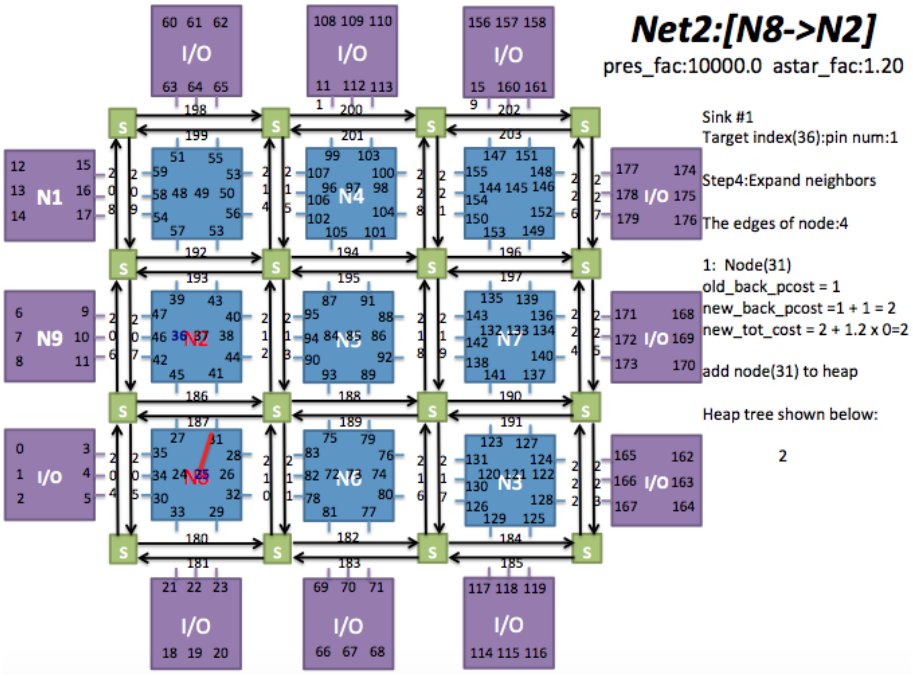


Figure 3.18: Routing process step 3

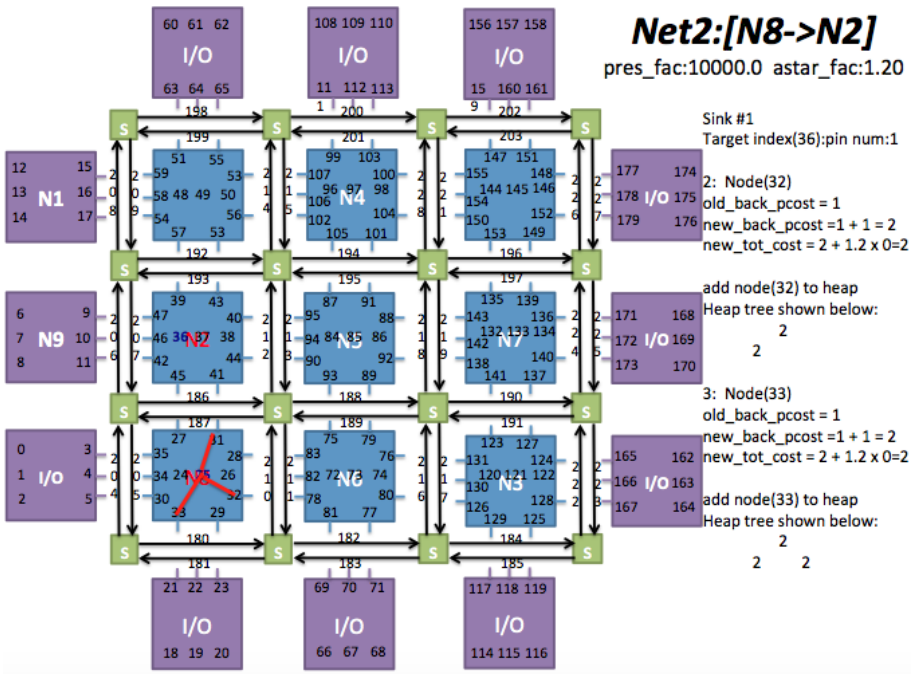


Figure 3.19: Routing process step 4

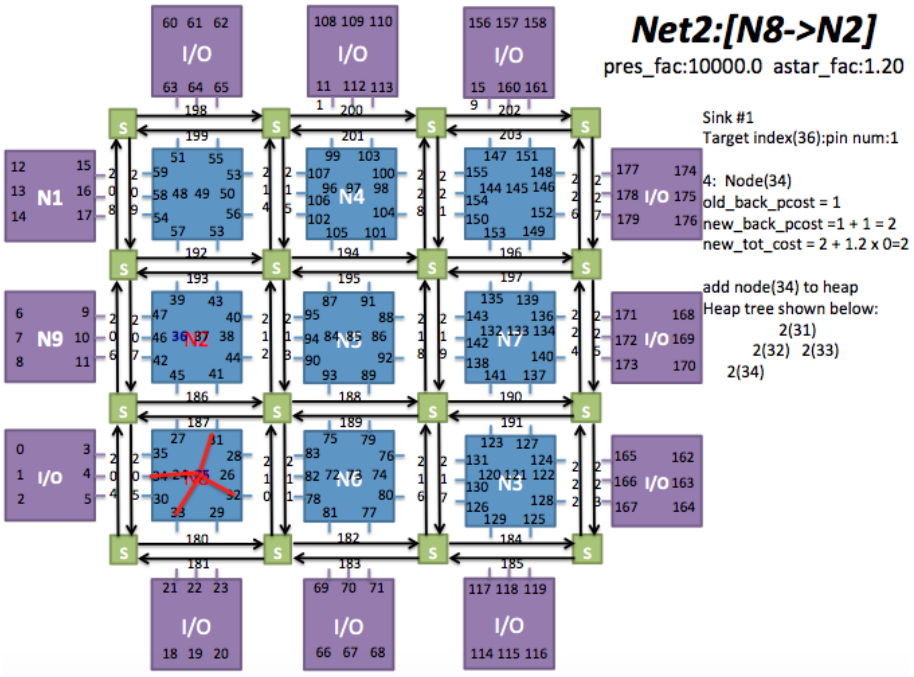


Figure 3.20: Routing process step 5

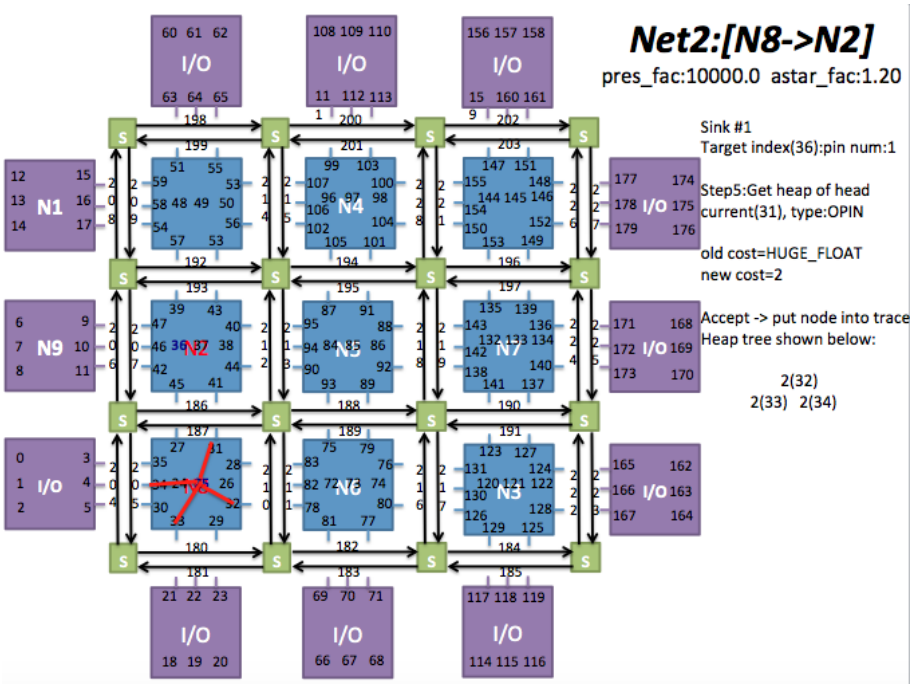


Figure 3.21: Routing process step 6

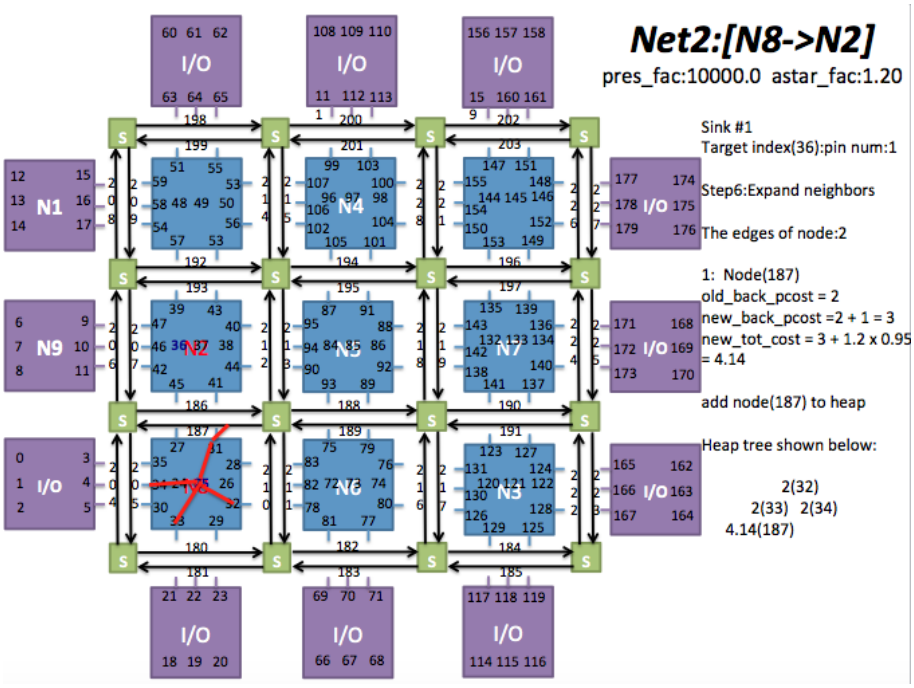


Figure 3.22: Routing process step 7

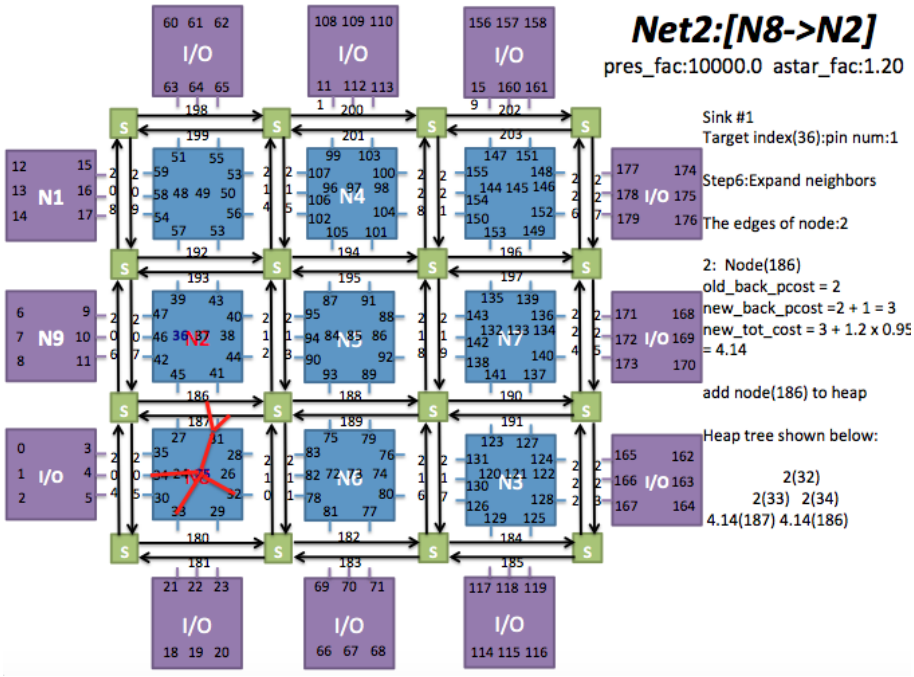


Figure 3.23: Routing process step 8

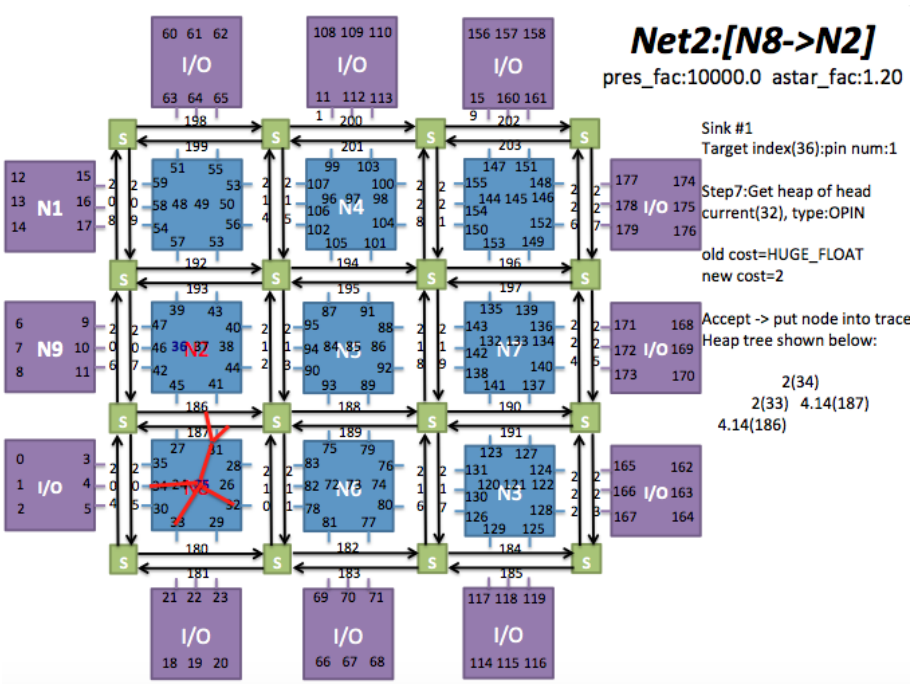


Figure 3.24: Routing process step 9

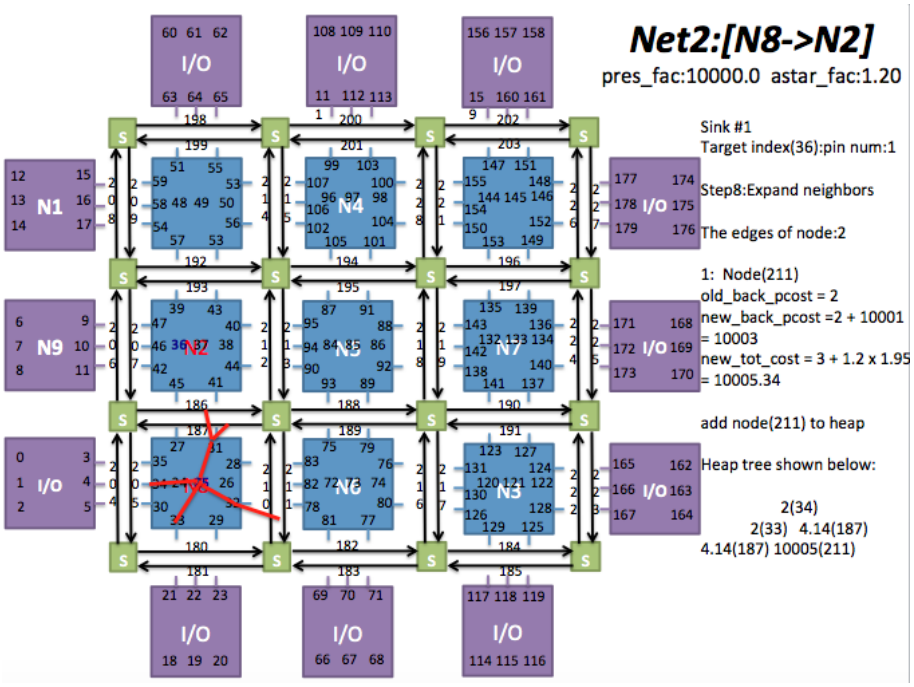


Figure 3.25: Routing process step 10

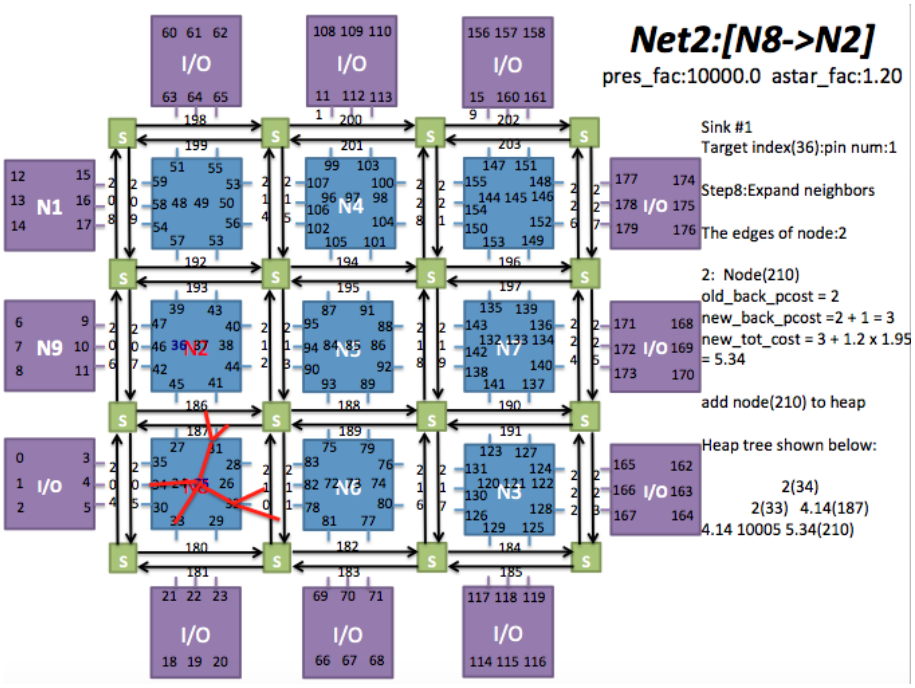


Figure 3.26: Routing process step 11

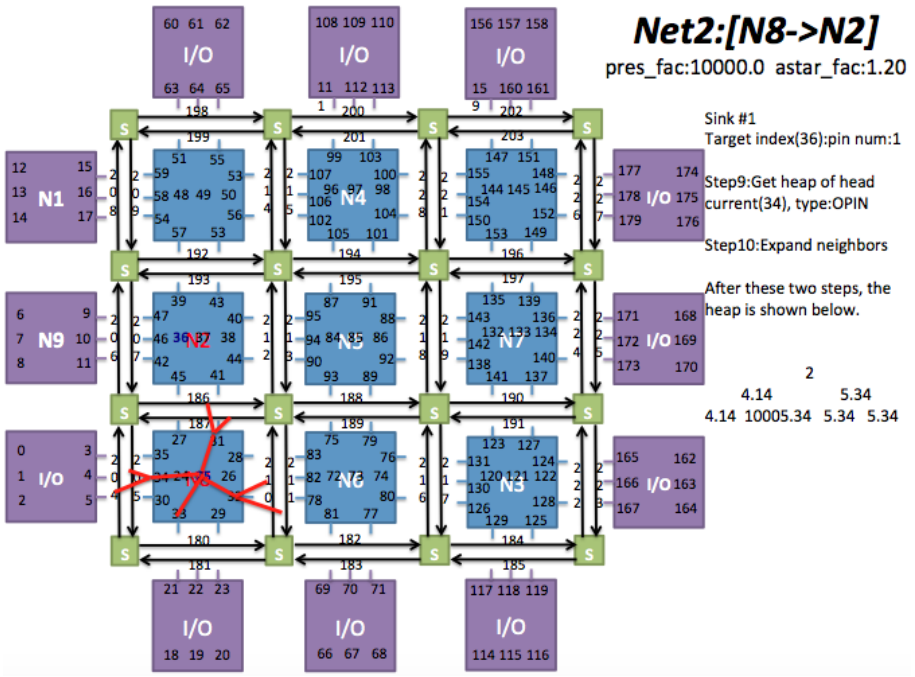


Figure 3.27: Routing process step 12

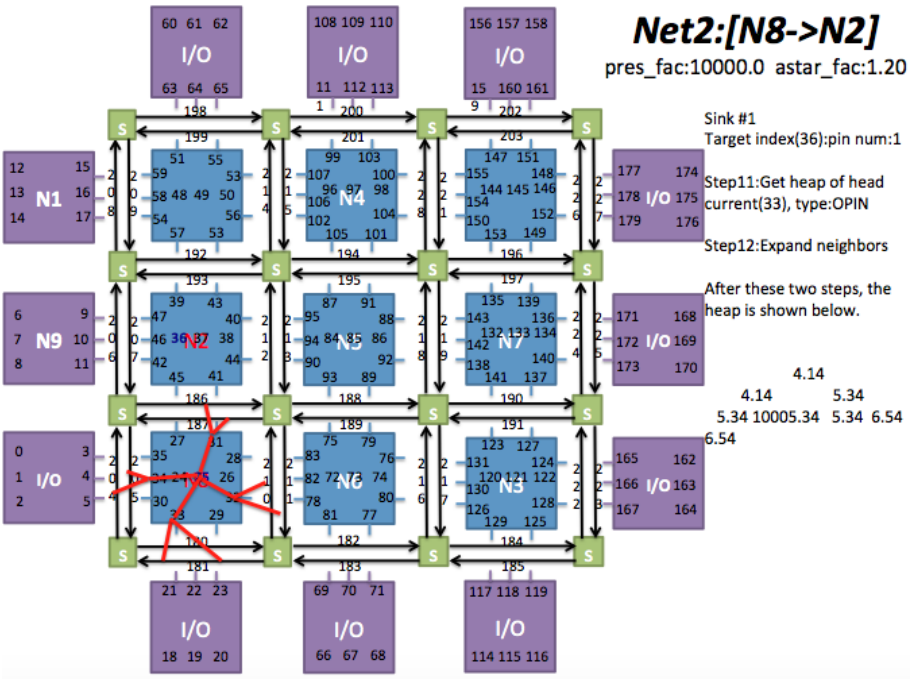


Figure 3.28: Routing process step 13

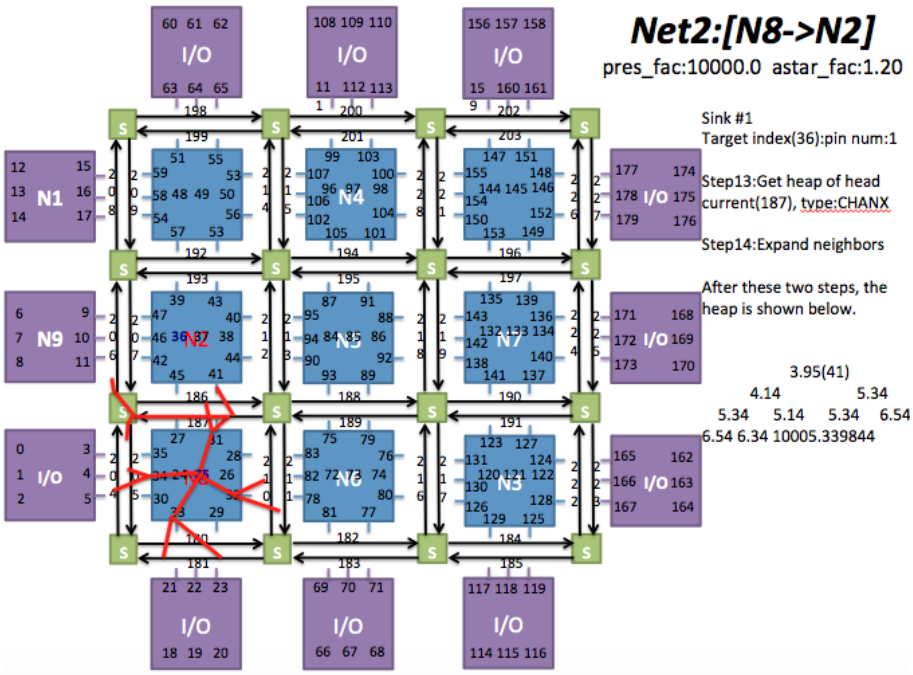


Figure 3.29: Routing process step 14

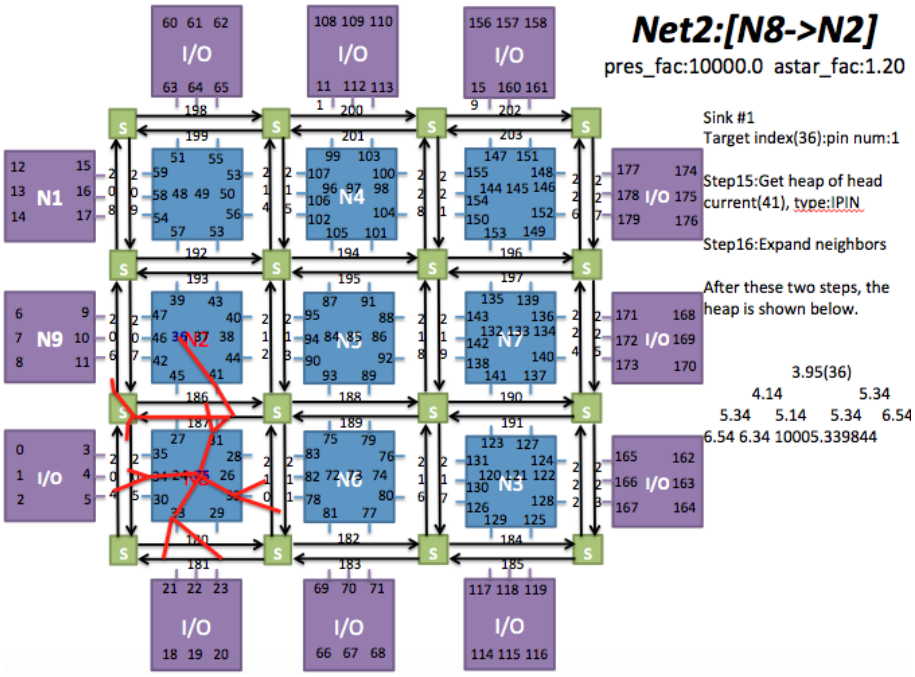


Figure 3.30: Routing process step 15

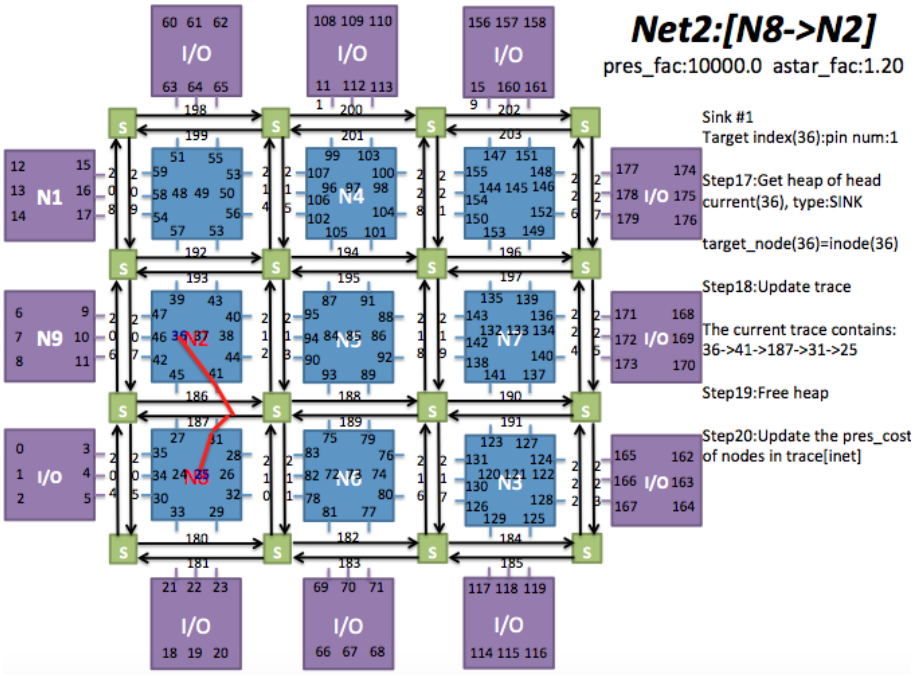


Figure 3.31: Routing process step 16

Chapter 4

Conclusions and Future Work

4.1 Conclusions

This report discussed the placement and routing of high level description of application (in data-flow graph format) on coarse-grained FPGA overlays. The algorithms used in Versatile Placement and Routing (VPR) tool were first discussed. This work included developing an understanding of the placement and routing algorithm. We developed an understanding of the underlying FPGA architecture as well as the various stages in the design process where placement and routing plays an important role. We then develop python implementation of placement and routing algorithms. We plan to release it publicly for others to use in research community.

4.2 Future work

The future work in this project mainly involves adapting the algorithms for different interconnect architecture since the current implementation only supports island-style architectures. As a next step, we aim to work on scalability analysis and runtime optimization of our implemenation. Long term goal is to build a Python-based toolchain to implement novel placement and routing algorithms which would make it easier to further extend our research to alternative architectures.

Appendix A

Python Implementation of Placement Algorithm

Table A.1: Python Function for Initial Placement

```
#Places the blocks into randomly selected coordinates
def doInitialPlacement():
	index = 0
	while blocksPlaced() == -1: #check if all blocks have been placed already
		flag = 0
		#print index
		block = block_list[index]
		print "Block being placed is ", block.name
		if block.getType() == "CLB":
			#possible coordinates for a CLB
			rand_x, rand_y = random.randrange(1,4,1), random.randrange(1,4,1)
			while [rand_x, rand_y] in FT_clb:
				rand_x, rand_y = random.randrange(1,4,1), random.randrange(1,4,1)
		else:
			#location_list_io = [[0,0],[0,1], [0,2],[0,3],[0,4],[4,0],[4,1], [4,2],[4,3],[4,4]]
			location_list_io = [[0,1], [0,2],[0,3],[4,1], [4,2],[4,3],[1,0],[2,0],[3,0],[1,4],[2,4],[3,4]]
			 #list of possible locations for io blocks
			random_index = randrange(0,len(location_list_io))
			rand_x, rand_y = location_list_io[random_index]
			while [rand_x, rand_y] in FT_io:
				rand_x, rand_y = random.randrange(1,4,1), random.randrange(1,4,1)

		print "The random coordinates are ", rand_x,rand_y
		# CLheck random coordinates in list of all coordinates
		for block1 in block_list:
			print block1.name, block1.getLocation(),

			if block1.getLocation()[0] == rand_x and block1.getLocation()[1] == rand_y:
				flag = 1

		if flag == 1:
			print "Location generated", rand_x,rand_y, "is already filled"
			print "generate another random location"
			continue #generate another random location
		else: #place block
			block.x = rand_x
			block.y = rand_y
			print block.name, " is placed at", block.x,block.y
			print "###################"
			index += 1
```

Table A.2: Python Functions developed for Placement process

```
#checks if all blocks have been placed.
def blocksPlaced():
	for block in block_list:
		if block.isPlaced() == -1:
			return -1
	return 1

#return block object on providing the name of the block
def getBlock(name):
	for block in block_list: #block_list is the list of block objects generated at the start
		if block.name == name: #if query block is in the block list, then return the block object, need to put exception handle
			return block

#Return net object on providing the name of the net
def getNet(name):
	for net in net_list:  #net_list is the list of net objects generated at the start
		if net.name == name: #if query net is in the net list, then return the net object, need to put exception handle
			return net

def printBB():
	print "Bouding box coordinates of all the nets are"
	for net in net_list:
		print net.name,": (",net.minx,",",net.miny,"),(",net.maxx,",",net.maxy,")"

#assess if a swap is accepted or note
def assessSwap(delta, t):
	if delta <= 0:
		return 1
	else:
		randno = random.random()
		prob_fact = math.exp(-delta/t)
		if prob_fact > randno:
			return 1
		else:
			return -1

# Criteria for Simulated Annealing to Stop
def exitCrit(temp,cost):
	if temp <0.005*cost/len(net_list) : #num of nets = len(net_list)
		return 1
	else:
		return 0

#Calculates total cost of a given configuration
def costFunction():
	total_cost = 0
	for net in net_list:
		print "\n[Bounding box for net",net.name,"is :",net.getBBCoord(),"]"
		print "bbx,bby = ",net.bbx,",",net.bby
		cost = net.netCost()
		total_cost+=cost
	return total_cost
```

Table A.3: Python Function for Try Swap

```
# Global declaration: the coordinates to be skipped
FT_clb = [[3,3],[2,2]]
FT_io = []

#this function is used to attempt to make a swap in the location of a randomly selected block
# Return 1 if the move is accepted, otherwise 0 return 0 if rejected
def try_swap(t):
	global current_cost
	to_block = None
	from_block = random.choice(block_list)

	#find out nets affected by swap
	to_nets = []
	from_nets = from_block.getNets()
	from_nets_name = []
	for net in from_nets:
		from_nets_name.append(net.name)

	#initial cost
	#old_cost = current_cost
	# initial from coordinates
	_from_x = from_block.x
	_from_y = from_block.y

	#initial to coordinates
	if from_block.type == "CLB":
		_to_x, _to_y = random.randrange(1,4,1), random.randrange(1,4,1)
		while [_to_x, _to_y] in FT_clb:
			_to_x, _to_y = random.randrange(1,4,1), random.randrange(1,4,1)
	else:
		#selects a random value from the possible locations for io
		location_list_io = [[0,1], [0,2],[0,3],[4,1], [4,2],[4,3],[1,0],[2,0],[3,0],[1,4],[2,4],[3,4]]
		random_index = randrange(0,len(location_list_io))
		_to_x, _to_y = location_list_io[random_index]
		while [_to_x, _to_y] in FT_io:
			_to_x, _to_y = random.randrange(1,4,1), random.randrange(1,4,1)

	for block in block_list:
		#print block.name
		#print block.getLocation()[0],",",block.getLocation()[1]
		if (block.getLocation()[0] == _to_x )& (block.getLocation()[1] == _to_y) :
			to_block = block
			to_nets = to_block.getNets()

	#store list of nets to update - includes nets connected to 'to' block and 'from' block
	nets_to_update = list(set(to_nets + from_nets))

	#Print the current values of net cost and bounding box for nets in net_to_update
	for net in nets_to_update:
		cost = net.netCost()
		print "Net cost = ", cost

	old_net_cost = 0
	for net in nets_to_update:
		#Find the net cost of nets to be updated before swapping
		old_net_cost += net.netCost()

	print "old net cost of nets_to_update = ", old_net_cost
	# Perform swap
	if to_block == None: #move to empty location
		from_block.x = _to_x
		from_block.y = _to_y
	else: #swap to and from blocks
		to_block.x,to_block.y = from_block.x, from_block.y
		from_block.x, from_block.y = _to_x, _to_y

	new_net_cost = 0
	#Blocks have been swapped => nets attached to these blocks have different Bounding box coordinates
	#Find net cost of nets to be updated after swapping
	for net in nets_to_update:
		new_net_cost += net.netCost()
	print "new net cost of nets_to_update = ",new_net_cost #print new net costs

	delta = new_net_cost - old_net_cost
	keep_switch = assessSwap(delta,t) #assess the swap
	if keep_switch == 1: #The move is accepted
		current_cost += delta
		return 1;
	else: # The move is rejected, revert block locations to old valuese
		from_block.x,from_block.y = _from_x,_from_y
		if to_block != None:
			to_block.x, to_block.y = _to_x, _to_y
		return 0;
```

Table A.4: Python code for Placement

```
##########################################################################
# Start Simulated Annealing #
num_blocks = len(graph.nodes())
inner_num = pow(num_blocks, 1.3333)
#move_lim = (int) (inner_num*pow(num_blocks,1.3333))
move_lim = int(10*inner_num)
print "Move lim = ", move_lim
list_all_nodes = list_compute_nodes + list_output_nodes + list_input_nodes

doInitialPlacement() #do initial placement

print " The nets in the design are listed"
for net in net_list:
       print net.name, net.num_target,
       net.printTargetList()
       print " "

initial_cost = costFunction() #get initial cost
printPlacement() #print initial placement
printBB() #print bounding box
temp = intialTemp() #set initial temperature
current_cost = initial_cost
total_iter = 0

line = "T                Av. Cost         Accept. rat.     Tot. Moves\n"
outfile.write(line);

#Start of while loop
while exitCrit(temp,current_cost) == 0:
       print "Number of iterations = ", move_lim
       print "Start of while loop!"
       print "temperature = ", temp
       success_sum = 0
       avg_cost = 0

       #start of inner iteration
       for inner_iter in range(move_lim):
              print "The cost of the configuration in this iteration is: ",current_cost
              if try_swap(temp) == 1:
                     success_sum += 1
                     avg_cost += current_cost
       #Total iterations is incremented
       total_iter += move_lim
       success_rat = float(success_sum/float(move_lim))

       print "The average cost of the configuration in this iteration is: ",avg_cost, success_sum
       if success_sum != 0 :
              avg_cost = avg_cost/ success_sum

    line = "%f  %f       %f       %d\n" % (temp, avg_cost, success_rat, total_iter)
       outfile.write(line)
       print "The average cost of the configuration in this iteration is: ",avg_cost
       #update temperature
       oldt = temp
       temp = tempSchedule(temp,success_rat) # Temperature is updated
       print " The new temperature is", temp

print "Final placement cost is ", current_cost
print "The final placement is "
printPlacement()
```

Table A.5: Python Class for Block

```
#Data Structure to store information about a block
class Block:
	pins = []
	nets = []   #nets[0] - net connected to pin 0, nets[1] - net connected to pin 1, x,y - location

	#constructor
	def __init__(self,name,blk_type):
		self.name = name  #name is the node name in the dfg
		self.type = blk_type  #blk_type can be CLB/INPAD/OUTPAD
		self.nets = []
		self.x = self.y = -1  #initializing with -1, -1
		if(self.type == 'CLB'):
			self.pins = [-1 -1 -1 -1 -1 -1 -1 -1]
		else:
			self.pins = []


	#Add nets to the blocks
	def addNet(self, net):
		self.nets.append(net)


	#sets the coordinates of the block
	def setLocation(self, x, y):
		self.x = x
		self.y = y


	#returns the coordinates of the block
	def getLocation(self):
		return self.x,self.y

	#checks if block is placed
	def isPlaced(self):
		if self.x > -1 and self.y > -1:
			return 1
		else:
			return -1

	#returns all details of the block
	def details(self):
		print self.name, self.type, self.getLocation(),

	#returns the type of block
	def getType(self):
		return self.type

	#returns the nets connecting to self
	def getNets(self):
		return self.nets
```

Table A.6: Python Class for Net

```
#Data Structure to store information about a block
# Net has 1 source and can have >=1 destination(target)
class Net:
	num_target = 0
	ncost = 0.0

	#initialises net with net name and source block
	def __init__ (self, name, src, src_blk):
		self.name = name
		self.source = src
		self.source_blk = src_blk
		self.target_list = []

	#adds another destination to a net
	def addConn(self,dest,dest_blk):
		self.num_target += 1
		self.target_block = dest_blk
		if self.target_block != None:
			self.target_list.append(self.target_block)

	#bbx = [ left bottom right top ]
	#sets q factor depending on number of terminals
	def getQfactor(self):
		if len(self.target_list)<=11:
			self.qfactor = cross_count[len(self.target_list)]
		else:
			self.qfactor = 1.5455

		return self.qfactor

	#returns the coordinates of the bounding box for the net
	def getBBCoord(self):
		maxx = self.source_blk.getLocation()[0]
		minx = max(maxx,1)
		maxy = self.source_blk.getLocation()[1]
		miny = max(maxy,1)
		#check coordinates of each target in target list to set min and max of bb
		for target in self.target_list:
			if target.getLocation()[0] < minx:
				minx = max(target.getLocation()[0],1)
			if target.getLocation()[0] > maxx:
				maxx = target.getLocation()[0]
			if target.getLocation()[1] <miny:
				miny = max(target.getLocation()[1],1)
			if target.getLocation()[1] > maxy:
				maxy = target.getLocation()[1]
		#Bounding box coordinates for net is found
		self.maxx = maxx
		self.minx = minx
		self.miny = miny
		self.maxy = maxy
		self.bbx = self.maxx - self.minx + 1
		self.bby = self.maxy - self.miny + 1
		mincoord = self.minx,self.miny
		maxcoord = self.maxx,self.maxy

		return mincoord,maxcoord

	#returns the bouding box
	def findBB(self):
		return self.bbx, self.bby

	#Returns cost of a net
	def netCost(self):
		self.bbx, self.bby = self.findBB()
		#Cavx(n) and Cavy(n) are the average channel capacities
		cavx = cavy = 0.01
		cost = self.getQfactor() * (self.bbx + self.bby)*cavx
		return cost

	#prints list of target nodes for the net
	def printTargetList(self):
		for target in self.target_list:
			if target.name != None:
				target.details(),
```

Appendix B

Python Implementation of Routing Algorithm

Table B.1: Python Functions for Directed Search

```
def directed_search_expand_trace_segment(trace, target, astar_fac, rem_conn_to_sink):
	if rem_conn_to_sink == 0:
		#usual case
		for item in trace:
			item_type = type(item)
			item_info = getNodeinRRNodeInfo(item)
			if item_type == Block:
				sink = item_info.isSink
			if item_type == Ipin or sink == 1:
				total_cost = astar_fac * get_directed_search_expected_cost(item, target)
				node_to_heap(item,total_cost)

def directed_search_expand_neighbour(heap, net,current,target,astar_fac):
	bb_min_coord = []
	bb_max_coord = []
	bb_min_coord, bb_max_coord = net.getBBCoord()
	#Puts all the nodes adjacent to the current node on the heap
	for node_neighbour in current.neighbours():
		#Check if node neighbour is outside the bounding box
		if node_neighbour.x > bb_max_coord.x or node_neighbour.y > bb_max_coord.y or node_neighbour.x < bb_min_coord.x or
		      node_neighbour.y < bb_min_coord.y:
			continue
		#Prune away IPINs that lead to blocks other than the target one.
		if type(node_neighbour) == Ipin and node_neighbour.blk != target:
			continue

		new_back_pcost = 0
		#new_back_pcost = old_back_pcost + get_rr_cong_cost()
		if bend_cost!= 0:
			if (current.type == "CHANY" and node_neighbour.type == "CHANX") or (current.type == "CHANX" and node_neighbour.
			      type == "CHANY"):
				new_back_pcost += bend_cost
		#Calculate expected cost of the neighbour node to reach the target node
		new_tot_cost = new_back_pcost +astar_fac*get_directed_search_expected_cost(node_neighbour, target_node)
		node_info = getNodeinRRNodeInfo(neighbour_node)
		node_info.total_cost = new_tot_cost
		node_to_heap(node_neighbour,new_tot_cost)
```

Table B.2: Python Functions for Routing

```
def directed_search_route_net(net,mst_net):
#Start the Directed Search Algorithm to route a particular net
        num_sinks = len(net.target_list)
        heap = BinHeap()
        target = mst_net[0][1]
        directed_search_add_source_to_heap(net,target, astar_fac)

        #do some more stuff.............
#Maze router is invoked num_sinks times to complete all the connections
        for i in range(num_sinks):
                #Since heap is emptied after a sink is found....
                #in the first iteration the heap head contains the source node
                target_node = mst_net[i][1]
                directed_search_expand_trace_segment()
                #In the first iteration, the source node is the head of the heap
                current = getHeapHead(heap) # current node is head of the heap

                if current == None:
                        print "\n␣Infeasible␣routing"
                inode = current

#Expanding the wavefront from source node till target node is reached. Nodes are retrieved from the
#heap. The head of the heap contains the neighbour with minimum cost........
                while inode != target_node:
                        rrnode_info = getNodeinRRNodeInfo(current.node)
                        old_tcost = rrnode_info.total_cost
                        new_tcost = current.cost
                        #old_back_cost = rrnode_info.backward_path_cost
                        #new_back_cost = current.backward_path_cost
                        if old_tcost >new_tcost:# and old_back_cost > new_back_cost:
                                directed_search_expand_neighbour(heap,net,current,target_node,astar_fac)
                        old_cur = heap.heapList[1]
                        current = heap.getHeapHead()

                        if current == None:
                                reset_path_costs()
                                return -1
                #End of while loop
                #After a sink is found..
                updateTraceback()
                pathfinderUpdateOneCost()
                empty_heap()
                reset_path_costs()
                        #get head from the Heap



def isFeasibleRouting():
        print ""
        return -1


def doRouting():

        iter = 0
        num_nets = len(net_list)
        pres_fac = first_iter_pres_fac
        #Do NUM_ITERATIONS iterations
        while iter <= NUM_ITERATIONS:
                #Reset RRNodeInfo[] items to default values at the start of every iteration
                # The values keep updating until routing of all nets is attempted
                del rr_node_info_list[:]

                for inet in range(0,num_nets):
                        #First, get the MST of the net - holds info about the source and sinks.
                        mst_net = buildMST(net)
                        directed_search_route_net(net,pres_fac,0,mst_net)
                        #if the net is not routable return to ...
                        if isRoutable(nets[inet]) == 0:
                                return -1

                if isFeasibleRouting() == 1:
                        #the routing is complete
                        return 1
                else :
                        if iter == 0:
                                pres_fac = initial_pres_fac
                                pathfinderUpdateCost(pres_fac, 0)
                        else:
                                pres_fac *= pres_fac_mult
                                pathfinderUpdateCost(pres_fac,acc_fac)
```

Bibliography

[1] O.T. Albaharna, P. Y K Cheung, and T.J. Clarke. On the viability of FPGA-based integrated coprocessors. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 206–215, 1996.

[2] Russell Tessier, Kenneth Pocek, and Andre DeHon. Reconfigurable computing architectures. *Proceedings of the IEEE*, 103(3):332–354, 2015.

[3] Stephen M Trimberger. Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology. *Proceedings of the IEEE*, 103(3):318–331, 2015.

[4] Andre DeHon. Fundamental underpinnings of reconfigurable computing architectures. *Proceedings of the IEEE*, 103(3):355–378, 2015.

[5] A. George, H. Lam, and G. Stitt. Novo-g: At the forefront of scalable reconfigurable supercomputing. *Computing in Science Engineering*, 13(1):82–86, 2011.

[6] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Survey*, 34:171–210, June 2002.

[7] Stephen Neuendorffer and Fernando Martinez-Vallina. Building zynq accelerators with vivado high level synthesis. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 1–2, 2013.

[8] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: high-level synthesis for FPGA-based Processor/Accelerator systems. In *Proceedings of the*

International Symposium on Field Programmable Gate Arrays (FPGA), pages 33–36, 2011.

[9] Yun Liang, Kyle Rupnow, Yinan Li, and et. al. High-level synthesis: productivity, performance, and software constraints. *Journal of Electrical and Computer Engineering*, 2012(649057):1–14, January 2012.

[10] Walid Najjar and Jason Villarreal. FPGA code accelerators - the compiler perspective. In *Proceedings of the Design Automation Conference*, 2013.

[11] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing modular hardware accelerators in c with ROCCC 2.0. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 127–134, 2010.

[12] C. Plessl and M. Platzner. Zippy - a coarse-grained reconfigurable array with support for hardware virtualization. In *Proceedings of the International Conference on Application-Specific Systems, Architecture Processors (ASAP)*, pages 213–218, 2005.

[13] Neil W. Bergmann, Sunil K. Shukla, and Jrgen Becker. QUKU: a dual-layer reconfigurable architecture. *ACM Transactions on Embedded Computing Systems (TECS)*, 12:63:1–63:26, March 2013.

[14] J. Coole and G. Stitt. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 13–22, October 2010.

[15] Davor Capalija and Tarek S. Abdelrahman. A high-performance overlay architecture for pipelined execution of data flow graphs. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2013.

[16] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. Dyser: Unifying

functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32(5):38–51, 2012.

[17] A. K. Jain, S. A. Fahmy, and D. L. Maskell. Efficient overlay architecture based on DSP blocks. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 2015.

[18] A. K. Jain, X. Li, S. A. Fahmy, and D. L. Maskell. Adapting the DySER architecture with DSP blocks as an Overlay for the Xilinx Zynq. In *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART)*, 2015.

[19] Cheng Liu, C.L. Yu, and H.K.-H. So. A soft coarse-grained reconfigurable array based high-level synthesis methodology: Promoting design productivity and exploring extreme FPGA frequency. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 228–228, 2013.

[20] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. A fully pipelined and dynamically composable architecture of cgra. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, pages 9–16, 2014.

[21] G. Grewal M. Xu and S. Areibi. Starplace: A new analytic method for fpga placement. *Integration, the VLSI Journal*, 43(1):1–33, January 2011.