NANYANG TECHNOLOGICAL UNIVERSITY

PLACEMENT AND ROUTING TOOL FOR COARSE GRAINED FPGA OVERLAYS

by

MUTHUSWAMI LAKSHMINARAYANAN SUSHEEL
(U1222684A)

A Report Submitted in partial fulfillment of
the requirements for the degree of
Bachelor of Computer Engineering

Supervised by

Assoc. Prof. Douglas L. Maskell

March 2016

Contents

List of Figures

List of Tables

Abstract

With the advancements in technology, parallel processing architectures such as multi-core processors, digital signal processors (DSPs), graphics processing units (GPUs), massively parallel processor arrays (MPPAs) and field programmable gate array (FPGA) based accelerators are gaining popularity for accelerated execution of compute kernels. Research efforts have shown strength of FPGA accelerators in a wide range of application domains where compute kernels can be implemented as high performance fully parallel and pipelined designs. Despite these advantages, FPGAs have not yet been ready for mainstream computing. One reason is that design productivity remains a major challenge, restricting the effective use of FPGA accelerators to niche disciplines involving highly skilled hardware engineers. Coarse-grained FPGA overlay architectures have been shown to be effective when paired with general purpose processors, offering software-like programmability, fast compilation, application portability and improved design productivity. These architectures enable general purpose hardware accelerators, allowing hardware design at a higher level of abstraction. This report presents a placement and routing (PAR) tool for coarse grained island-style overlays based on the algorithms used in widely accepted VPR placement and routing tool. We start with understanding the PAR algorithms in detail and develop a python based PAR tool customized for coarse-grained island-style overlays.

Acknowledgment

First and foremost, I would like to thank Assoc Prof Dr Douglas Leslie Maskell for his guidance, enthusiastic support and strong encouragement without which my project would not be a success.

Moreover I would also like to thank Abhishek Kumar Jain for his professional guidance, continuous support, effective suggestions, constructive criticism and timely help. I am thankful for his technical advice, subject matter tips and continuous encouragement during the course of the project. I am also thankful to him for carefully guiding me, reading and commenting on countless revisions of this report.

I would also like to thank Hsieh, Mu-Hua for her contribution to the project and working on it together in a team which has been greatly beneficial for the completion of the work.

Finally, I am indebted to my family for their prayers and encouragement. I thank them for their understanding and their efforts to support me in my Final Year Project.

Chapter 1

Introduction

1.1 Motivation

Silicon technology will continue to provide an exponential increase in the availability of raw transistors. Effectively translating this resource into application performance, however, is an open challenge that conventional processor designs will not be able to meet. On the other hand, Field Programmable Gate Array (FPGA) devices provide a sea of high performance computing blocks for implementing kernels as high performance fully parallel and pipelined designs. For more than a decade, researchers have shown that FPGAs can accelerate a wide variety of software, in some cases by several orders of magnitude compared to state-of-the-art general purpose processors. The most fundamental difference is that general-purpose processors provide functionality to execute a list of instructions sequentially, whereas FPGA architectures can implement compute kernels by mapping compute operations on configurable blocks.

While the performance benefits of FPGAs over processor based systems have been well established [1, 2, 3, 4], such platforms have not seen wide use beyond specialist application domains such as digital signal processing and communications. Poor design productivity has been a key limiting factor, restricting their effective use to experts in hardware design [5]. Even as High Level Synthesis (HLS) tools improve in efficiency [6, 7], prohibitive compilation time (specifically place and route time) still limits productivity and mainstream adoption of reconfigurable platforms.

Despite numerous efforts in reducing reconfiguration times and improving CAD tool support for dynamic reconfiguration of FPGA fabric [8, 9, 10, 11, 12], It still prevents designers from using FPGA as a rapidly reconfigurable hardware accelerator. Thus, the requirement to rapidly change the hardware fabric, that is to perform a hardware context switch, has led to the development of coarse grained overlay architectures which allow for fast compilation and software like programmability.

Coarse-grained FPGA overlay architectures [13, 14, 15, 16, 17, 18, 19, 20, 21] have been shown to be effective when paired with general purpose processors, offering software-like programmability, fast compilation, application portability and improved design productivity. These architectures enable general purpose hardware accelerators, allowing hardware design at a higher level of abstraction. In our work, we aim to develop a placement and routing (PAR) tool for coarse grained island-style overlays.

1.2 Contribution

We start with understanding the algorithms used in widely accepted VPR placement and routing tool and develop a python based PAR tool customized for coarse-grained overlays. APIs from Python graph library have been used for implementing the tool. We aim to adapt the algorithms to support different interconnect architectures as a future work. The main contributions can be summarized as follows:

- Understanding of placement and routing algorithms used in VPR tool
- Python based implementation of the algorithms

1.3 Organization

The remainder of the report is organized as follows: Chapter 2 presents background information on overlay architectures including placement and routing. Chapter 3 studies current state of the art overlays and techniques for placement and routing. Chapter 4 shows the steps involved in the placement of a data flow graph (DFG) nodes on an island-style coarse grained overlay. Chapter 5 shows the steps involved in the routing of a data flow graph (DFG) edges on the overlay. We conclude in chapter 6 and discuss future work.

Chapter 2

Background

2.1 Coarse Grained FPGA Overlays

Overlay architectures consist of a regular arrangement of coarse grained routing and compute resources. The key attraction of overlay architectures is software-like programmability through mapping from high level descriptions, application portability across devices, design reuse, fast compilation by avoiding the complex FPGA implementation flow, and hence, improved design productivity. Another main advantage is rapid reconfiguration since the overlay architectures have smaller configuration data size due to the coarse granularity. Accelerators can be described at a higher level of abstraction and compiling it for overlays is several orders of magnitude faster than for the fine grained FPGAs. Researchers have proposed fine [22], [23] and coarse grained [13], [14], [15], [16], [20], [24] overlay architectures to abstract FPGA fabric resources.

As shown in Fig. 2.1, coarse grained FPGA overlay architecture is a two-dimensional array of reconfigurable tiles, implemented on top of a commercial FPGA device. Coarse grained tiles contains programmable processing elements (PEs) interconnected using programmable interconnect (PI) and the functions of the PE and the PI are controlled by configuration data. The overlay overcomes the need for a full cycle through the vendor implementation tools, instead presenting a much simpler problem of placing arithmetic operations on an array of processing elements and routing data via an interconnect network.

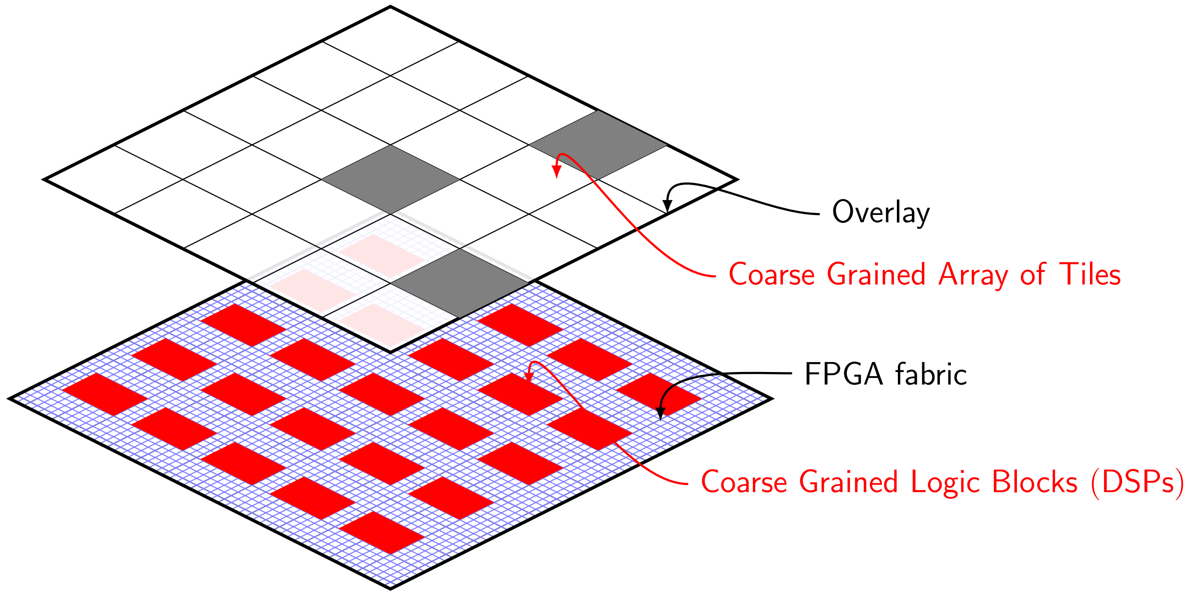

Overlay
Coarse Grained Array of Tiles
FPGA fabric
Coarse Grained Logic Blocks (DSPs)

Figure 2.1: FPGA Overlay Architecture

Researchers have shown the effective use of coarse grained overlay architectures by pairing them with host processors as a coprocessor [21, 25] or as a part of the processor's pipeline [26]. Fig. 2.2 shows the integration of DySER [26, 27] overlay into the pipeline of a processor.

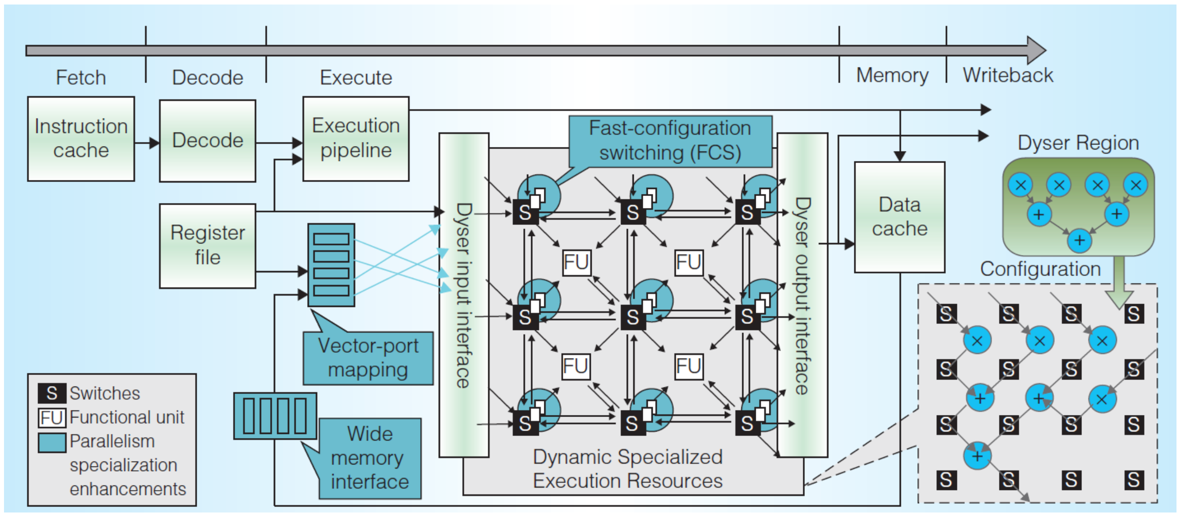


Figure 2.2: DySER Interfacing with Host Processor [17]

In DySER overlay, the functional unit (FU) provides resources for the mathematical and logical operations, and synchronization logic. It receives its input values from the four neighboring switches and outputs its result to the switch in the south-east direction. The switches allow datapaths to be dynamically specialized. They form a on chip network that creates paths from inputs to the functional units, between functional units, and from functional units to outputs. Fig. 2.3 shows the mapping of kernels on DySER architecture.

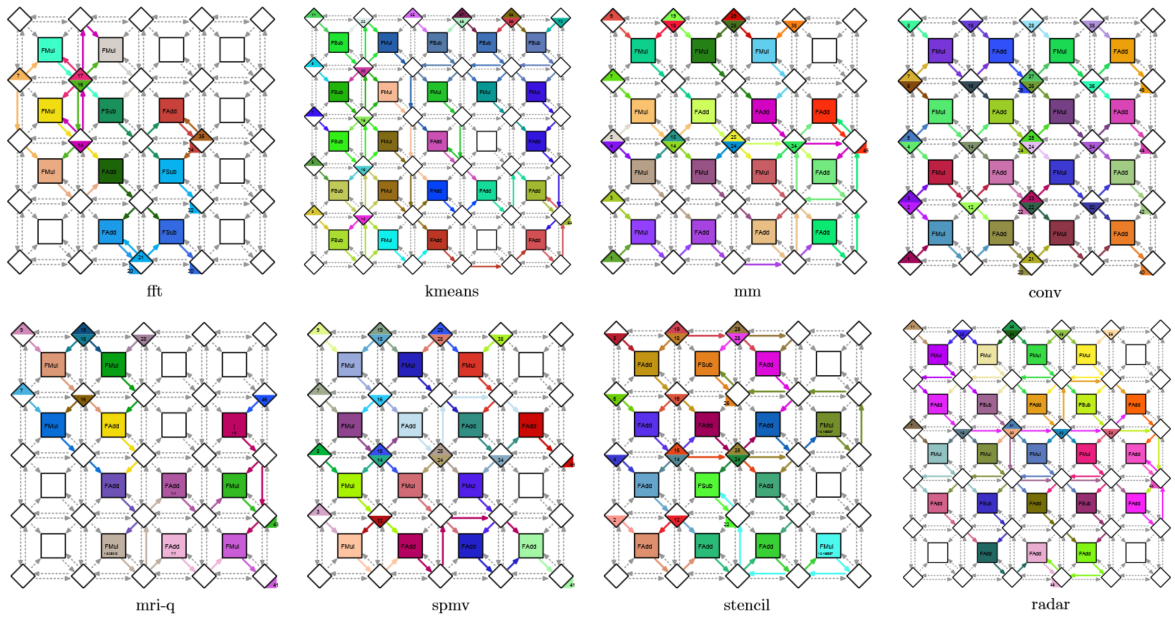


Figure 2.3: Mapping of Kernels on DySER Architecture.

One example of pairing the overlay (Intermediate Fabric (IF) Overlay [15]) with a high performance ARM processor via an Advanced eXtensible Interface (AXI) interface in a commercial computing platform (the Xilinx Zynq[28]) is shown in Fig. 2.4. Zynq platform partition the hardware into a Processing system (PS), containing one or more processors along with peripherals, bus and memory interfaces, and the Programmable Logic (PL) where custom hardware including an overlay can be implemented. The Xilinx-Zynq consists of a dual-core ARM Cortex A9 processor equipped with a double-precision Floating Point Unit (FPU), commonly used peripherals and reconfigurable fabric.

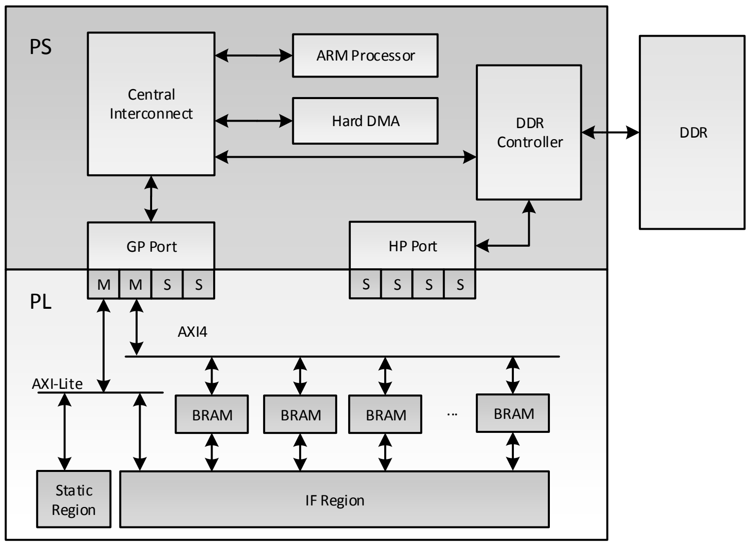


Figure 2.4: Intermediate Fabric (IF) Interfacing with Host Processor [25]

When paired as a coprocessor, run-time management, including overlay configuration loading, data communication, can be carried out using an operating system (Linux) [21] and also using a commercial hypervisor [29]. Firstly, user needs to identify a kernel, to be implemented on top of overlay. Then DFG can be extracted after compiling this code using compiler front-end. After that a place and route tool can be used to map the DFG on top of overlay. After generating configurations based on the placement and routing, kernel code can be transformed in the code containing overlay APIs.

An overlay provides a leaner mechanism for hardware task management at runtime as there is no need to prepare distinct bitstreams in advance using vendor-specific compilation (synthesis, map, place and route) tools. Instead, the behaviour of the overlay can be modified using software defined overlay configurations. The possible configuration space and configuration data size is much smaller than for direct FPGA implementation of kernels because of the coarser granularity of the overlay. In the next section, we provide an overview of placement and routing for fine-grained and coarse-grained architectures.

2.2 Placement and Routing

In this section, we will discuss about placement and routing for fine grained and coarse grained architectures with the help of examples. 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.

To give an example, Fig. 2.5 shows HDL description of a compute kernel for accumulating four 16-bit numbers. Fig. 2.6 shows the mapping of the description onto a fine-grained FPGA architecture.

```verilog
module kernel(a,b,c,d,out);
input[15:0] a,b,c,d;
output[15:0] out;

assign out = a + b + c + d;

endmodule
```

Figure 2.5: Verilog Code for accumulating four 16-bit numbers.

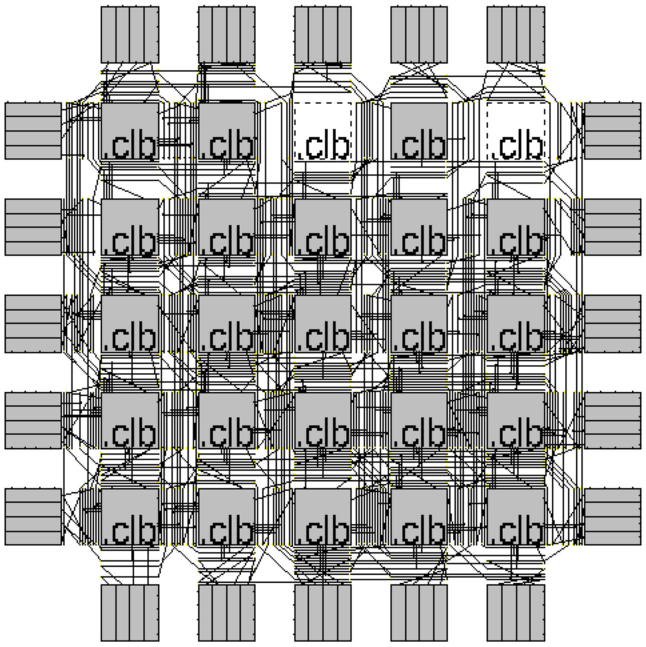


Figure 2.6: Placement of Routing on Fine-grained architecture

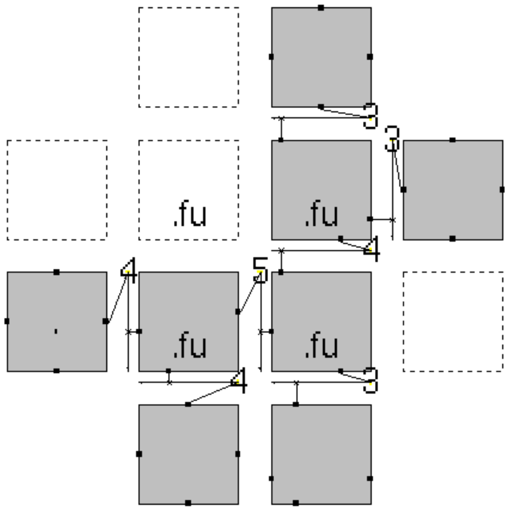


Figure 2.7: Placement of Routing on Coarse-grained architecture

Fig. 2.7 shows the mapping on a coarse grained architecture where each track is 16-bit wide and each functional unit (FU) is a 16-bit arithmetic operator.

Chapter 3

Literature Survey

In the area of coarse grain overlay architectures, the compute routing logic can either perform the same operation over the time, or can loop over a short list of instructions or can execute a fully fledged instruction stream. Based on this variety, researchers have proposed both spatially configured and time multiplexed overlays that are mapped to the fine grained fabrics of modern FPGAs. In spatially configured overlays, the compute logic and routing of the overlay are unchanged while a compute kernel is executing while in time multiplexed overlays, the compute logic and routing of the overlay change on a cycle by cycle basis while a compute kernel is executing [20, 30, 31]. In this report, we focus on the work done by other researchers in the area of placement and routing on spatially configured overlays.

An island-style interconnect based overlay architecture (spatially configured), referred to as an intermediate fabric (IF) [15], [32] was proposed to support near-instantaneous placement and routing (shown in Fig. 3.1). Standard VPR [33] algorithms were used for placement and routing of compute kernels. It consists of 192 heterogeneous functional units comprising 64 multipliers, 64 subtracters, 63 adders, one square root unit, and five delay elements with a 16-bit datapath and supported the fully parallel, pipelined implementation of compute kernels.

Unlike a physical device, whose architecture must support many applications, IFs have been specialized for particular domains or even individual applications. Such specialization hides the complexity of fine-grained Commercial Off-the-shelf (COTS)

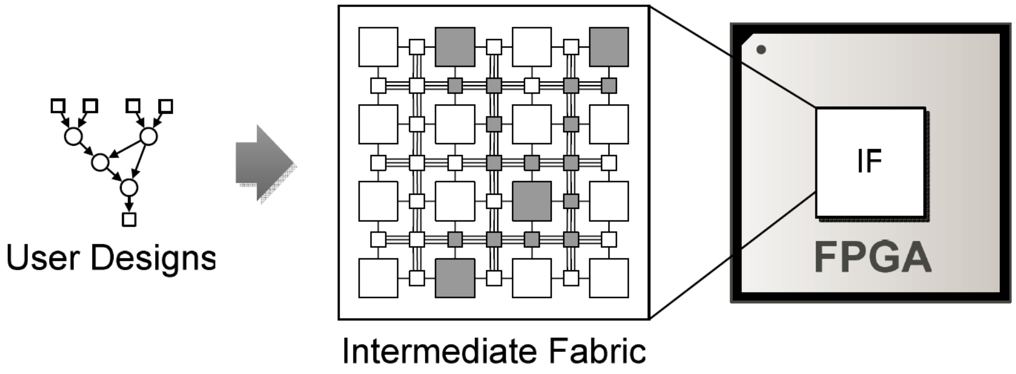


Figure 3.1: Intermediate Fabrics as Island-style Overlay [15].

devices, thus enabling fast place and route (700x speedup over vendor tools) at the cost of significant area (34% - 44%) and performance (7%) overhead when implemented on an Altera Stratix III FPGA [32]. However, the IF only achieved an F_{max} of 125 MHz resulting in low throughput for the application benchmarks tested. Area overhead comes into picture mainly because of virtual interconnect logic which comprised of multiplexers based routing. Based on the above mentioned work on IFs, an end-to-end tool flow was presented for FPGA-accelerated scientific computing [34].

Another spatially configured overlay based on nearest neighbor interconnect (shown in Fig. 3.2) was proposed in [16]. This overlay executes a given DFG by mapping the graph nodes to the FUs and by configuring the routing logic to establish inter-FU connections that reflect the graph edges [16]. Multiple instances of the DFGs are then executed in a pipelined fashion on the overlay to achieve high performance. It consisted of a 24×16 overlay with a nearest-neighbor-connected mesh of 214 routing cells and 170 heterogeneous functional units (FU) comprising 51 multipliers, 103 adders and 16 shift units. When implemented on an Altera Stratix IV FPGA, the overlay consumed 75% of the total device ALMs, with the routing network consuming 90% of the ALM resource used. An F_{max} of 355 MHz and a peak throughput of 60 GOPS was reported. A placer and router was developed by customizing VPlace [35] and PathFinder [36], respectively.

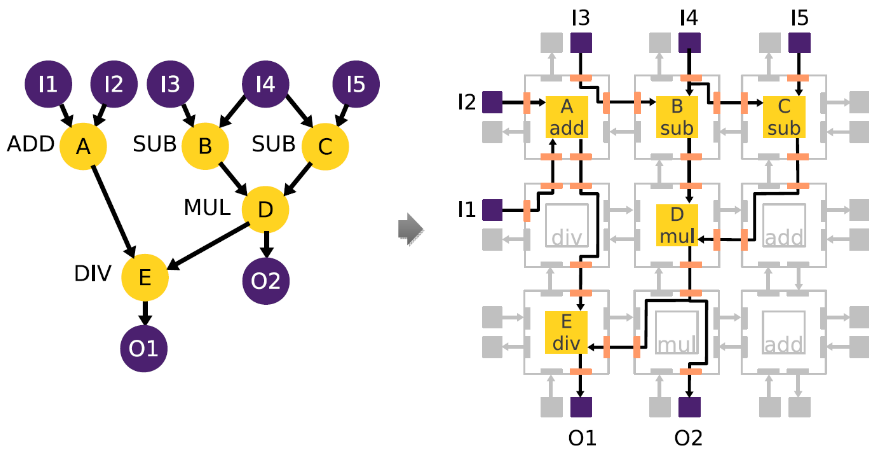


Figure 3.2: Nearest-neighbor connected Mesh of Functional units [16].

DySER [17, 27] was proposed as a coarse grained overlay architecture for improving the performance of general purpose processors. It was originally designed as a heterogeneous array of 64 functional units interconnected with a circuit-switched mesh network and implemented on ASIC. The DySER architecture was then improved and prototyped, along with the OpenSPARC T1 RTL, on a Xilinx XC5VLX110T FPGA [26]. However, due to excessive LUT consumption, it was only possible to fit a 2x2 32-bit DySER, a 4x4 8-bit DySER or an 8x8 2-bit DySER on the FPGA. An adapted version of a 6x6 16-bit DySER was implemented on a Xilinx Zynq-7020 [19]. The larger DySER array was achieved by using a DSP block as the compute logic, thus better targeting the architecture to the FPGA.

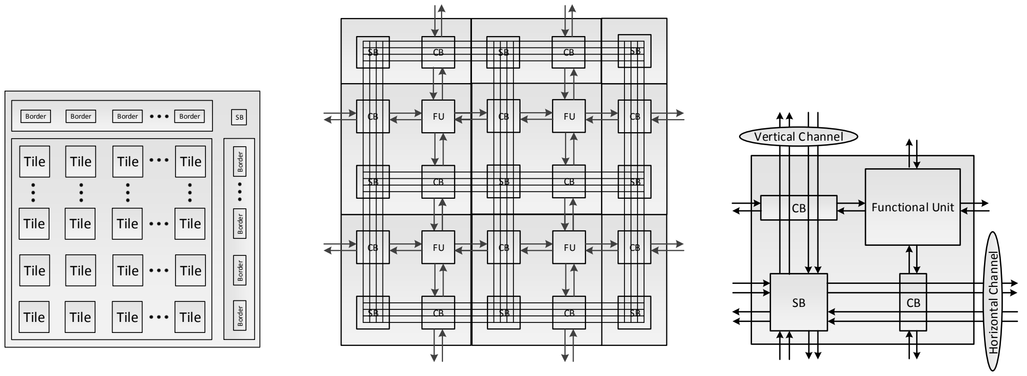
An overlay architecture with the FU based on the DSP blocks found in Xilinx FPGAs was recently proposed [18]. This overlay combines multiple operations in a compute kernel and maps them to the DSP block, resulting in a significant reduction in the number of processing nodes required. An F_{max} of 370 MHz with throughputs better than that achieved by directly implementing the benchmarks onto the fabric using Xilinx Vivado HLS were reported. In the next chapters, we use this overlay architecture as a base platform to discuss about the placement and routing of data flow graphs.

Chapter 4

Placement of DFG nodes on Island-style Overlay

In this chapter, we first describe an island-style coarse-grained overlay architecture (published previously in [18]) and then describe the placement of input DFG on to the overlay using placement algorithm used in VPR tool. We also describe our python implementation of the placement algorithm.

4.1 Island-style Overlay Architecture



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

Figure 4.1: Overlay architecture.

The overlay instantiates the tiles and borders, where each tile instantiates virtual routing resources and a functional unit (FU) and each border instantiates one switch box (SB) and one connection box (CB), forming the boundary at the top and right of the array, as shown in Fig. 4.1(a). This results in an overlay architecture which contains I/O around the periphery of the overlay fabric. This I/O can be connected to a FIFO or BRAM I/O data port. Fig. 4.1(b) shows the architecture of a 2×2 overlay having four tiles, east boundary (two east borders), north boundary (two north borders) and a switch box at the north east corner. It shows that a 2×2 overlay would consist of 4 FUs, 9 SBs and 8 CBs. Extrapolating, an $N \times N$ overlay would incorporate N^2 FUs, $(N+1)^2$ SBs and N^2+2*N CBs. Each tile contains a functional unit (FU) and virtual routing resources, as shown in more detail in Fig. 4.1(c).

4.2 Automated Mapping Tool

In this section, we describe an automated mapping tool (published previously in [18]) which allows to map high level description of compute kernels onto the overlay. The mapping process comprises DFG extraction from high level compute kernels, mapping of the DFG nodes onto the DSP48E1 primitives, VPR compatible FU netlist generation, the placement and routing of the FU netlist onto the overlay, latency balancing and finally, the configuration generation.

4.2.1 Data Flow Graph (DFG) Generation

Starting with a C description of the compute kernel, the tool transforms this to a DFG description, as shown in Table 4.1. Fig. 4.2(a) shows the nodes and edges in an example DFG. In the next step, the DFG description is parsed and translated into a technology-mapped DFG. For example, we can use multiply-subtract and multiply-add to collapse N5-N7 and N6-N8 in Fig. 4.2(a) into N5 and N6 of Fig. 4.2(c), respectively. This results in the mapped DFG shown in Fig. 4.2(b).

Table 4.1: Compute Kernel Code Descriptions

(a) C description

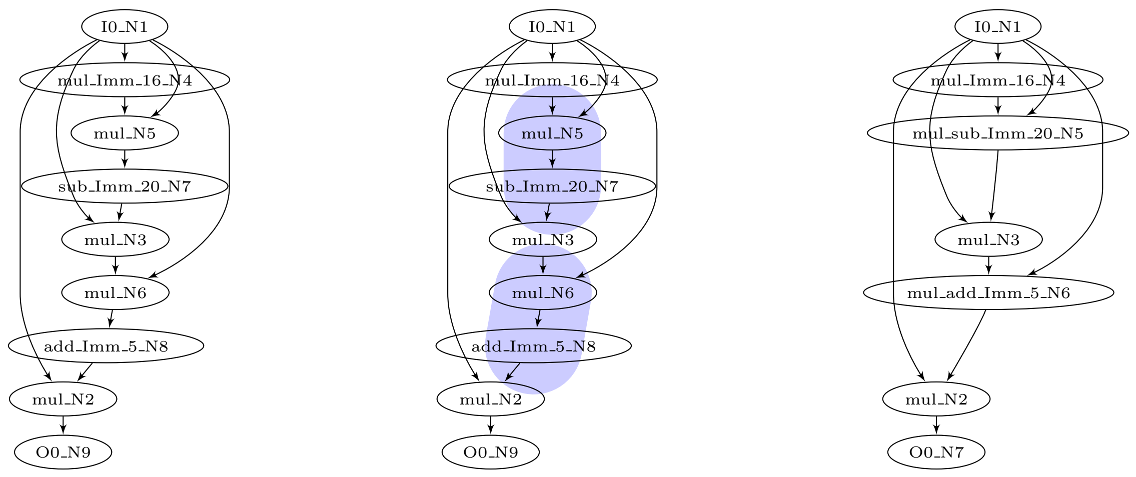
```c
#include<math.h>
#define SIZE 1000

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;
}
```



(a) Input DFG (b) Node-merging (c) Mapped DFG

Figure 4.2: DSP48E1 aware mapping.

4.2.2 DFG to VPR Compatible Netlist Conversion

We use a netlist generator which takes the mapped DFG and generates a VPR-compatible netlist of FUs, as shown in Table 4.2. We now make use of algorithms used in VPR tool to place nodes onto the overlay and route signals between them. Rather than map logic functions to LUTs and single-bit wires to 1-bit channels, we are mapping nodes in the graph to FUs, and 16-bit wires onto 16-bit channels.

Table 4.2: 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
```

4.2.3 Placement and Routing onto the Overlay

The place and route algorithm maps DFG nodes onto homogeneous FUs and DFG edges to the overlay's routing pats to connect mapped FUs. At this level of granularity, a netlist can have 100's of nodes, making the problem much smaller than that of fine-grained FPGA placement and routing which deals with netlists of millions of nodes. Placement of DFG (shown in Fig. 4.3) onto a 5×5 overlay is shown in Fig. 4.4.

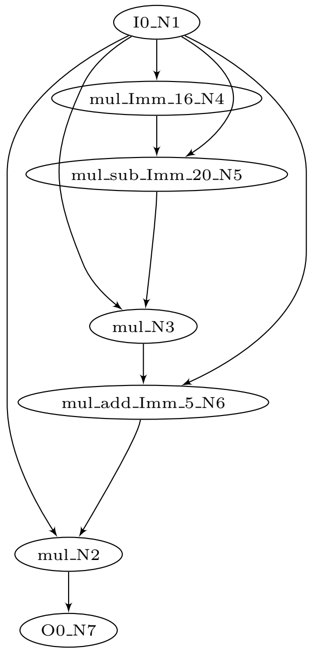
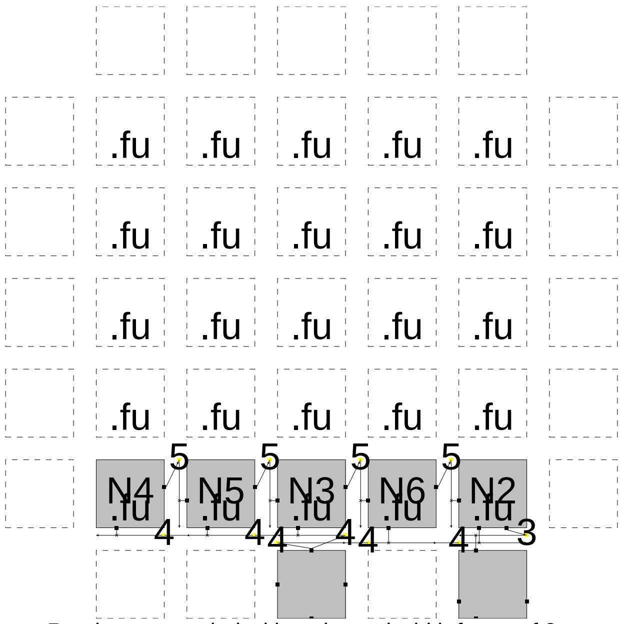


Figure 4.3: Data Flow Graph (DFG)



Routing succeeded with a channel width factor of 2.

Figure 4.4: DFG mapped onto the Overlay after Placement and Routing.

4.3 Detailed description of Placement process

Placement algorithm used in VPR tool uses simulated annealing which is a method for solving optimization problems. It models the heating of a metal at high temperatures and gradually lowering temperature thereby minimizing energy and reducing defects. In the case of FPGA placement, a high temperature is first selected. In VPR, the annealing schedule automatically adjusts to different cost functions and circuit sizes. It is a heuristic-based search for minimizing the value of a cost function. The cost function takes real values over a set of states and is shown in Eq. 4.1.

$$\mathbf{Cost} = \sum_{\mathbf{n=1}}^{\mathbf{N_{nets}}} \mathbf{q(n)} \left[\frac{\mathbf{bb_x(n)}}{\mathbf{C_{av,x}(n)}} + \frac{\mathbf{bb_y(n)}}{\mathbf{C_{av,x}(n)}} \right] \tag{4.1}$$

The $q(n)$ factor has different values depending on the number of terminals a net has. The value of $q(n)$ becomes greater than one for a net with more than 3 terminals. $C_{av,x}(n)$ and $C_{av,y}(n)$ are the average channel capacities (in tracks) in the x and y directions over the bounding box of the given net. The cost function used here is linear congestion cost function and focuses only on wire length and penalizes placements which require more routing in areas of the overlay that have narrower channels. In case of overlay, we assume all channels to have the same capacity (in other words equal density of tracks in all channels), hence we use a constant value, 0.01, for both $C_{av,x}(n)$ and $C_{av,y}(n)$. Hence the linear congestion cost function reduces to a bounding box cost function. The total cost of a placement is the summation of the costs of each net involved. Bounding box of a net is calculated by considering the maximum range of its x and y co-ordinates. bb_x and bb_y denote the horizontal and vertical spans of the bounding box of a net. If (x_1, y_1) and (x_2, y_2) are the bounding box coordinate of a net, then bb_x and bb_y can be calculated using the equation shown in Eq. 4.2 and Eq. 4.3.

$$\mathbf{bb_x = x_2 - x_1 + 1} \tag{4.2}$$

$$\mathbf{bb_y = y_2 - y_1 + 1} \tag{4.3}$$

4.3.1 Initial Placement of DFG onto the Overlay

The algorithm starts with a random initial placement. The initial temperature is defined thorough the execution of N_{blocks} moves, where N_{blocks} = number of I/O nodes + number of compute nodes.

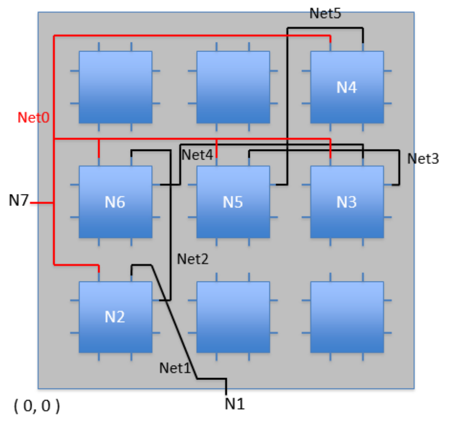


Figure 4.5: Initial Placement Connections

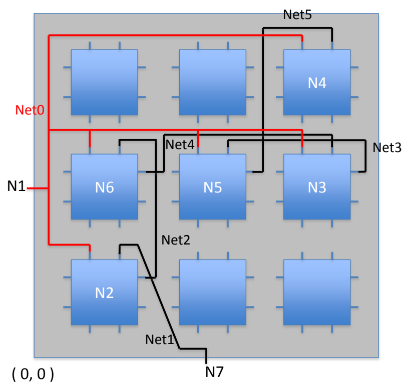
Initially, the blocks are placed randomly and the cost of the placement is calculated by adding the cost of each net. For example, the nodes in the DFG (shown in Fig. 4.3) are placed on a 3×3 overlay as shown in Fig. 4.5. $N1$ is the input node in the DFG which is placed initially on the I/O block located 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. The cost of the initial placement is calculated as 0.2432. We use the initial temperature as 0.4 and evaluate a fixed number of moves at this temperature.

4.3.2 Evaluation of Moves at a given Temperature

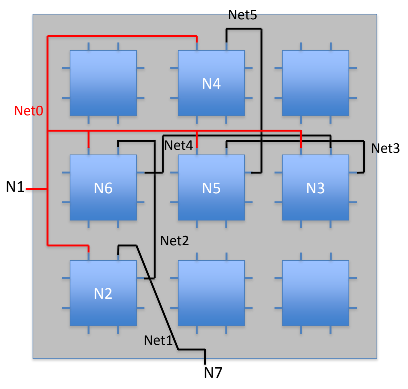
At a given temperature, a fixed number of moves are made. The number of moves (move_lim) made is calculated according to the Eq. 4.4.

$$\mathbf{move\_lim = 10 * (N_{blocks})^{1.33}} \tag{4.4}$$

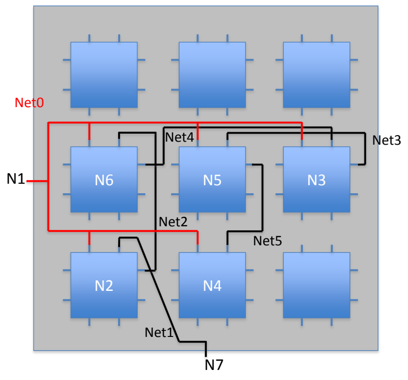
A logic block is selected at random and moved to another random location. If another logic block occupied that location then the two blocks are swapped. The cost is calculated for this new configuration and compared to the previous cost. In most cases, moves where the cost reduces are the ones which are accepted. However, at high temperatures even moves where the cost increases are accepted. In this way we do not get stuck at a local minima. At lower temperatures the percentage of moves where the cost increases that get accepted is far lower. The placement obtained after all the iterations are complete is the optimum placement given the parameters and the value of the objective function obtained is minimum. Fig. 4.6 shows few moves and their evaluation at a given temperature.



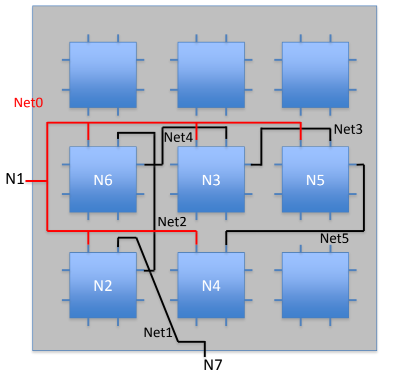
(a) Initial Placement with a cost = 0.2432



(b) Moving N4 from (3,3) to (2,3), Net 0 and Net 5 affected, change in cost = - 0.01, Move accepted



(c) Moving N4 from (2,3) to (2,1), Net 0 and Net 5 affected, change in cost = - 0.01, Move accepted



(d) Swapping N5 and N3, Net 0, Net3, Net 4 and Net 5 affected, change in cost = 0, Move accepted

Figure 4.6: Placement Iterations at a Given Temperature

4.3.3 Detailed description of Each Move

Table 4.3 shows the manual trace of Moves.

Table 4.3: 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
```

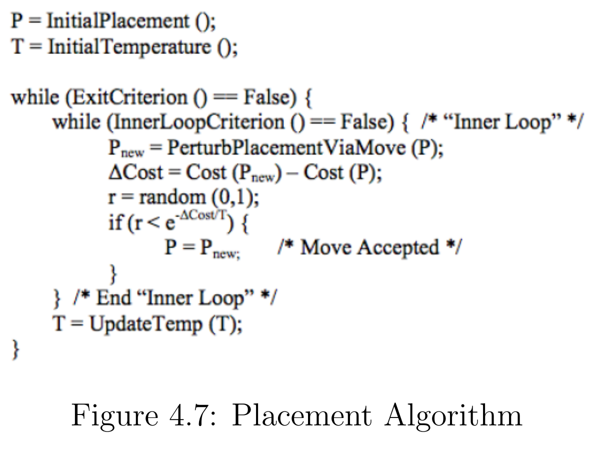
In the first iteration (Iteration 0), block N4 which is initially at (3,3) is moved to a randomly selected location - (2,3), which is initially empty. N4 has two nets connected to it - Net 0 and Net 5. Due to the movement of N4, the bounding box of these two nets changes and thus their cost. The new costs of these two nets are calculated using the new bounding box values. Change in cost for each net is calculated and then the total change. If Delta (net change) is negative, the move is accepted. Otherwise, the swap is assessed using the probability factor calculated as shown in Eq. 4.5. If this value is greater than that of a random number between 0 and 1, then the move is accepted. Otherwise it is rejected.

$$\mathbf{prob\_fac} = \mathbf{e^{-delta/Temp}} \tag{4.5}$$

In the next iteration (Iteration 1), N4 is again selected randomly and the destination randomly selected is (2,1), which is empty in this case too. The bounding boxes for the nets are calculated again and the value of the delta is calculated. It is negative again, so the move is accepted.

In iteration 2, N5 is selected randomly. The random destination is (3,2) which happens to be occupied by N3. Thus, the nets to consider in this case are the ones which are connected to either N5 and N3. Net0, Net3, Net 4 and Net 5 are the nets whose bounding box value is updated. The change in cost of these nets is found to be 0. The move is then accepted after assessing the swap.

The above iterations show us an example of the process of placement by using placement algorithm shown in Fig. 4.7. At a given temperature, multiple iterations of moves are executed. The temperature is then updated according to the annealing schedule selected. Alpha is the scaling factor for the updated temperature. Once the temperature is updated the iterative process is repeated again. The anneal is terminated when $T \leq 0.005 * Cost/Nnets$.



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

while (ExitCriterion () == False) {
    while (InnerLoopCriterion () == False) {  /* “Inner Loop” */
        $P_{new}$ = PerturbPlacementViaMove (P);
        $\Delta$Cost = Cost ($P_{new}$) – Cost (P);
        r = random (0,1);
        if (r < $e^{-\Delta Cost/T}$) {
            P = $P_{new;}$        /* Move Accepted */
        }
    }  /* End “Inner Loop” */
    T = UpdateTemp (T);
}
```

Figure 4.7: Placement Algorithm

Chapter 5

Routing of DFG edges on Island-style Overlay

The next step after placement of DFG nodes on coarse-grained overlay is routing of DFG edges. In this chapter, we first explain the interconnect architecture and then we describe the implementation of the algorithms for routing used in the VPR tool.

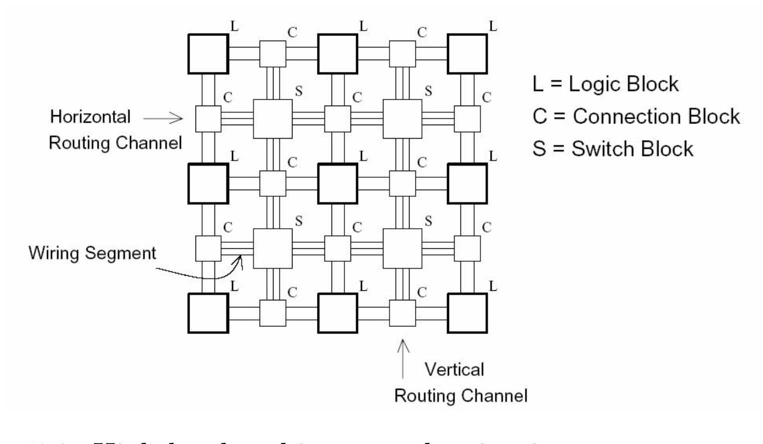


Figure 5.1: High level architecture showing interconnect resources

5.1 Generic Interconnect Architecture

The DFG is mapped onto homogeneous functional units (in Fig. 5.1 shows as logic block) and needs to be connected using the interconnect architecture. The main elements of the interconnect architecture are channels, connection boxes and switch boxes.

5.1.1 Channels

Channels in turn consist of tracks. Tracks travel in horizontal and vertical directions. These are responsible for the connection of the pins of logic blocks which are mapped onto the overlay. The width of the channels is represented by W which is equal to the number of tracks present in the channel. Channel segments can generally have variable length or they may have the size of one CLB span only. In our overlays, we assume that the segments span one Logic block. In addition, channels can be uni-directional or bi-directional. Uni-directional channels can connect structures only in one direction and not the opposite direction. In our case, we use uni-directional channels. Channels are connected to Logic block pins via connection boxes and to other channels through switch boxes.

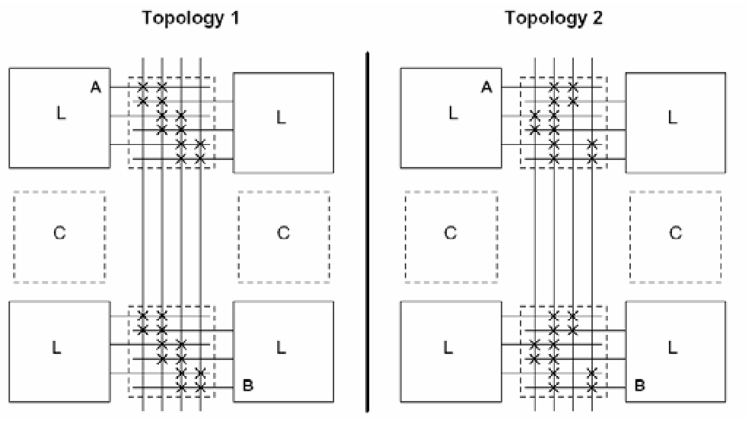


Figure 5.2: Connection Box

5.1.2 Connection Boxes

Connection boxes connect the channel wires with the I/O pins of the logic blocks. The number of tracks in each channel to which each logic block input and output pin can connect is called connection box flexibility, f_c. The topology of a connection box decides the pattern of switches that connects logic block I/O to the tracks as shown in Fig. 5.2.

5.1.3 Switch Boxes

Switch boxes are responsible for connecting channels with one another. They allow the wires of a channel to switch directions between horizontal and vertical directions. The flexibility f_s indicates for a segment that enters the switch block, the maximum number of segments that it can connect to in the switch block. The topology of the switch block defines the connections and influences the routing results of the interconnect matrix. Fig. 5.3 shows the common topologies used for the switch box.

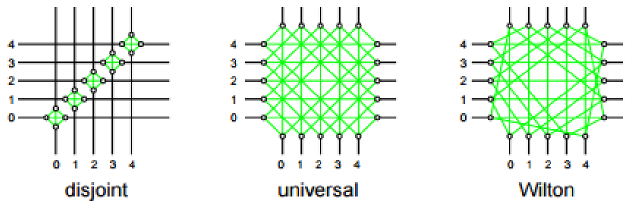


Figure 5.3: Switch Box Topologies

5.1.4 Interconnect architecture used for the overlay

Each logic block has 4 RECEIVER pins, 4 DRIVER pins and one global pin. In our coarse-grained overlay implementation, we have channel width of 2 meaning two tracks per channel. Also, the channels are unidirectional. Switch box flexibility f_s is equal to 3 and $f_c = 1$. A pin of a logic block can thus connect to all the tracks of a channel. In our case, switch box architecture allows a track to connect to 3 other tracks from neighbouring channels. Coarse-grained overlays have greatly simplified the task of routing. The replacement of 16-bit wires with 1-bit wires has reduced the number of nodes and resources needed from millions to a few hundreds. Thus we aim to exploit this main advantage.

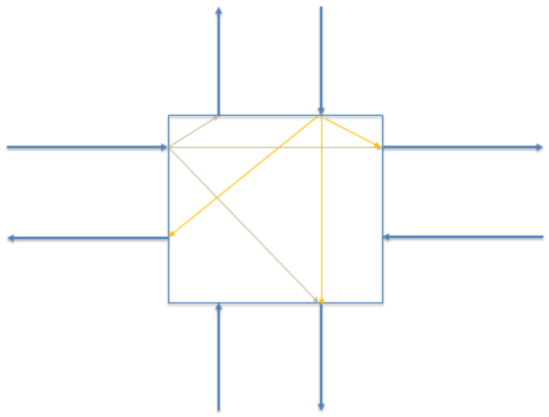


Figure 5.4: Switch Box Uni-Directional Routing

5.2 Routing Algorithm

Routing involves using the physical resources of the overlay to implement net connections between logic blocks. Firstly, all the resources are mapped onto a routing resource graph. Algorithms are then implemented on this graph to achieve routing. There are general two stages in solving the routing problem: Global and Detailed Routing. Global routing performs a coarse route with the objective of balancing congestion across the channels while connecting the nets. Initially, each net connection is routed according to lowest cost. However,as the iterations progress, the congestion is balanced out.

Given a global routing, detailed Routing selects individual tracks for connecting nets. All possible detailed routes are considered by means of searching on the routing resource graph. A directed graph is constructed using resources to represent connections betweens tracks, input pins, switch boxes and output pins. In other cases, a single step detailed router performs the entire process of routing. In this case, a

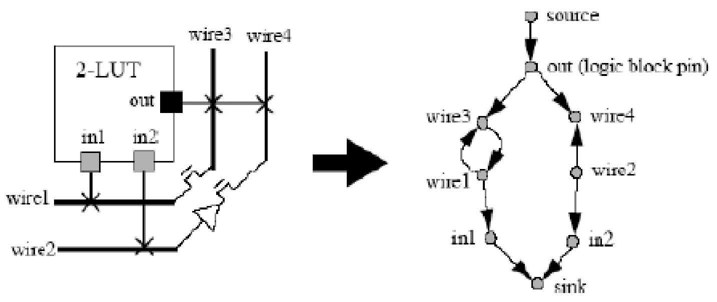


Figure 5.5: Routing Resource Graph

routing resource graph is constructed. Each node on this graph is assigned a cost function and search is performed on the graph to connect source to target.

The Algorithms used are based on Djikstra's algorithm to find shortest path between two nodes. The algorithm implemented in VPR is the Pathfinder Negotiated Congestion Algorithm. The Pathfinder algorithm is based on the Maze router which involves an expanding wavefront technique to find the shortest path while avoiding used resources. Pathfinder allows re-use of resources. However, the cost of over-used resources is gradually increased in subsequent iterations making them unfavourable for use. This combats Maze Router's drawback of the performance being net ordering dependent as a path found can block the routing of subsequent nets.

5.2.1 Pathfinder Algorithm for Routing

Before the routing process starts, some data structures need to be initialized. A minimum spanning tree (MST) is created for each net prior to running the algorithm. This MST contains the source and all the targets of the net. They are inserted in order of minimum distance.

A binary heap is also required for the directed search algorithm. The neighbours of a given node are inserted into the heap along with their cost. The head of the heap always contains the minimum cost neighbour to visit next.

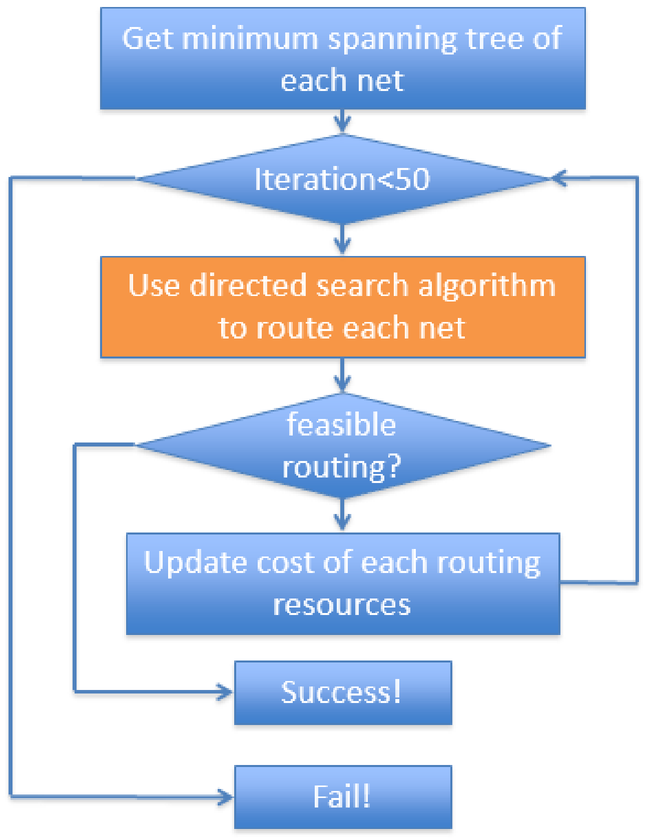

Get minimum spanning tree of each net
Iteration<50
Use directed search algorithm to route each net
feasible routing?
Update cost of each routing resources
Success!
Fail!

Figure 5.6: Algorithm for Routing

5.2.1.1 Cost Function

The Pathfinder negotiated congestion algorithm used in VPR incorporates details about current routing as well as history of congestion in previous iterations into the cost function for routing resources. Each node in the routing resource graph is assigned a cost. The cost function that is updated in each iteration when a portion of a net's routing is complete is shows in Eq. 5.1.

$$\mathbf{RRnode\_pres\_cost} = \mathbf{1} + (\mathbf{occ} + \mathbf{1} - \mathbf{capacity}) * \mathbf{pres\_fac} \tag{5.1}$$

where *RRnode_pres_cost* is the present cost of a node in the routing resource graph, *occ* is the number of nets which utilize this node, capacity is the maximum utilization allowed and $pres\_fac$ is the present sharing penalty factor. $pres\_fac$ is multiplied with factor every iteration to increase the sharing penalty. The cost that is updated for each iteration of routing all the nets is shown in Eq. 5.2 and Eq. 5.3 when $occ \geq capacity$:

$$\mathbf{RRnode\_acc\_cost} = \mathbf{RRnode\_acc\_cost} + (\mathbf{occ} - \mathbf{capacity}) * \mathbf{acc\_factor} \quad (5.2)$$

$$\mathbf{RRnode\_pres\_cost} = \mathbf{1} + (\mathbf{occ} + \mathbf{1} - \mathbf{capacity}) * \mathbf{pres\_fac} \quad (5.3)$$

Else, *RRnode_acc_cost* does not change and *RRnode_pres_cost* can be calculated as shown in Eq. 5.4

$$\mathbf{RRnode\_pres\_cost} = \mathbf{1} + \mathbf{pres\_fac} \quad (5.4)$$

where *RRnode_acc_cost* is the accumulated cost of a resource, acc_factor is the historical congestion cost multiplier. If the occupancy of a node is greater than its capacity then the accumulated cost is increased according to acc_factor. If not, then the accumulated cost is not updated as it is within the capacity of the resource.

5.2.1.2 Description of Algorithm

A loop of 50 iterations is then run where each iteration uses the directed search algorithm to route each net. Since, overuse is allowed, some resources such as tracks of channels might be used by more than one net. In such cases, the routing is not feasible. The overused resources are assigned a higher cost for the next iteration and the loop continues. After all the iterations complete, if a successful routing is not found, then the routing as failed.

The directed search route algorithm attempts to route a net by iteratively connecting its source to its sinks, if there are multiple sinks. It uses a Maze Router

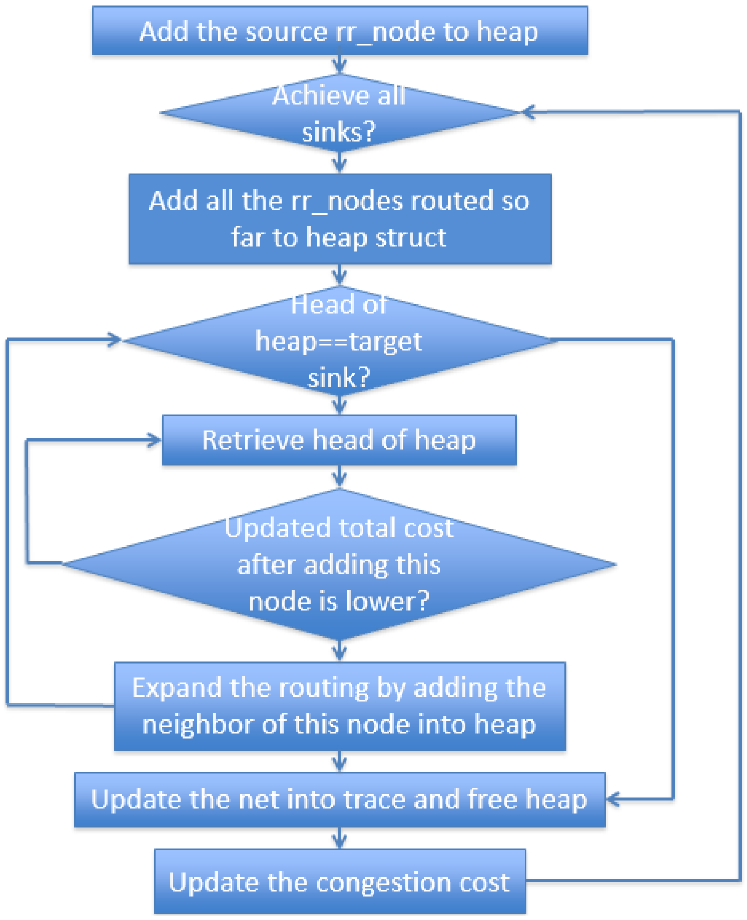

Add the source rr_node to heap
Achieve all sinks?
Add all the rr_nodes routed so far to heap struct
Head of heap==target sink?
Retrieve head of heap
Updated total cost after adding this node is lower?
Expand the routing by adding the neighbor of this node into heap
Update the net into trace and free heap
Update the congestion cost

Figure 5.7: Directed Search Route Net

(Djikstra Algorithm) to route a net. It iterates number of sinks times till all are connected. Till the sink is reached in the head of the heap, the program loops. In the first iteration, the source of the net is added into the heap, along with a cost calculated based on the congestion cost and the expected cost to reach the target node.In each iteration, the head of the heap becomes the current node. When a node is added into the heap, its total cost is calculated using the expected cost to reach the target node. If the new cost of the current node from the heap is less than the cost which is stored in the node data structure previously, the cost and path details stored in the node data structure are updated to the current node values and the neighbours of the current node are added into the heap along with their cost. In case of multi-fan out nets, the algorithm restarts with the entire first wire segment included as part of the source. This makes the program more efficient as a new wavefront does not have to be created from start. The routing resource nodes routed so far are added to the heap structure after their total cost is calculated. For each sink iteration, code loops till the head of the heap is the target sink. It then updates the net into the trace, frees the heap and updates congestion cost for the next iteration.

The result of the routing algorithm after the process is finished is a complete trace of the resources used for the connection of all logic blocks placed in the overlay. The resources used to connect the Source to the Sink nodes for each net are listed one by one along with their coordinates. This gives us the path to route each net and achieves the objective of the algorithm.

Chapter 6

Conclusions and Future Work

This chapter concludes and summarizes this report. Furthermore, in this chapter we discuss future research directions in detail.

6.1 Conclusions

This report discussed the placement and routing of high level description of application (in data-flow graph format) on coarse-grained FPGA overlays.

Course-grained FPGA overlays offer many-fold advantages such as software-like programmability through mapping from high level descriptions, portability, reuse and fast compilation. Faster compilation and rapid reconfiguration due to smaller configuration data size lead to significantly improved design productivity and efficiency. It helps reduce a complex problem that must be solved with vendor implementation tools into a simpler one of placing functions on an array of processing elements and routing data.

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.

6.2 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 tool-chain 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

```python
#Places the blocks into randomly selected coordinates
def doInitialPlacement():
   index = 0
   while blocksPlaced() == -1: #check if all blocks have been placed already
      flag = 0
      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)

      else:
         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]

      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

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

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

  print "From block:",from_block.name, "- nets:", from_nets_name
  _from_x = from_block.x # initial from coordinates
  _from_y = from_block.y

  if from_block.type == "CLB":  #initial to coordinates
    _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]

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

  print "From location is ", from_block.x,from_block.y
  print "To Location is ",_to_x,_to_y
  if to_block == None:
    print "To location is empty"
  else:
    print "To location is occupied by ", to_block.name
    print "Nets of ",to_block.name,":",
    for net in to_block.getNets():
      print net.name,

  #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 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:
    print "\n[Bounding box for net",net.name,"is :",net.getBBCoord(),"]"
    print "bbx,bby = ",net.bbx,",",net.bby
    cost = net.netCost()

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

  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;
    #print "The move is accepted: Delta = ", delta,"New Cost = ", current_cost
  else: # The move is rejected
    #print "The move is rejected"
    from_block.x,from_block.y = _from_x,_from_y  #revert block locations to old valuese
    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

```python
#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] Katherine Compton and Scott Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

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

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

[4] 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.

[5] Greg Stitt. Are field-programmable gate arrays ready for the mainstream? *IEEE Micro*, 31(6):58–63, 2011.

[6] 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.

[7] 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.

[8] K. Vipin and Suhaib A. Fahmy. Architecture-aware reconfiguration-centric floorplanning for partial reconfiguration. In *Proceedings of the International Symposium on Applied Reconfigurable Computing (ARC)*, pages 13–25, 2012.

[9] K. Vipin and Suhaib A. Fahmy. A high speed open source controller for FPGA partial reconfiguration. In *Proceedings of International Conference on Field Programmable Technology (FPT)*, pages 61–66, 2012.

[10] K. Vipin and Suhaib A. Fahmy. Automated partitioning for partial reconfiguration design of adaptive systems. In *Proceedings of IEEE International Symposium on Parallel Distributed Processing, Workshops (IPDPSW) – Reconfigurable Architectures Workshop (RAW)*, 2013.

[11] K. Vipin and Suhaib A. Fahmy. Automated partial reconfiguration design for adaptive systems with CoPR for Zynq. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2014.

[12] K. Vipin and Suhaib A. Fahmy. ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq. *IEEE Embedded Systems Letters*, January 2014.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] A. Brant and G.G.F. Lemieux. ZUMA: an open FPGA overlay architecture. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–96, 2012.

[23] M. Hubner, P. Figuli, R. Girardey, D. Soudris, K. Siozios, and J. Becker. A heterogeneous multicore system on chip with run-time reconfigurable virtual FPGA

architecture. In *IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*, 2011.

[24] Karel Heyse, Tom Davidson, Elias Vansteenkiste, Karel Bruneel, and Dirk Stroobandt. Efficient implementation of virtual coarse grained reconfigurable arrays on FPGAS. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2013.

[25] A. K. Jain, K. D. Pham, J. Cui, S. A. Fahmy, and D. L. Maskell. Virtualized execution and management of hardware tasks on a hybrid ARM-FPGA platform. *Journal of Signal Processing Systems*, 77(1–2):61–76, Oct. 2014.

[26] Jesse Benson, Ryan Cofell, Chris Frericks, Chen-Han Ho, Venkatraman Govindaraju, Tony Nowatzki, and Karthikeyan Sankaralingam. Design, integration and implementation of the dyser hardware accelerator into opensparc. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2012.

[27] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 503–514, 2011.

[28] Xilinx Ltd. Zynq-7000 technical reference manual. `http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`, 2013.

[29] Khoa Dang Pham, Abhishek Kumar Jain, Jin Cui, Suhaib A Fahmy, and Douglas L Maskell. Microkernel hypervisor for a hybrid ARM-FPGA platform. In *Proceedings of the International Conference on Application-Specific Systems, Architecture Processors (ASAP)*, 2013.

[30] Alexander Brant. Coarse and fine grain programmable overlay architectures for FPGAs. Master's thesis, University of British Columbia, 2013.

[31] K. Paul, C. Dash, and M.S. Moghaddam. reMORPH: a runtime reconfigurable architecture. In *Euromicro Conference on Digital System Design*, 2012.

[32] G. Stitt and J. Coole. Intermediate fabrics: Virtual architectures for near-instant FPGA compilation. *IEEE Embedded Systems Letters*, 3(3):81–84, September 2011.

[33] Vaughn Betz and Jonathan Rose. VPR: A new packing, placement and routing tool for fpga research. In *Field-Programmable Logic and Applications*, pages 213–222, 1997.

[34] G. Stitt, A. George, H. Lam, C. Reardon, M. Smith, B. Holland, V. Aggarwal, Gongyu Wang, J. Coole, and S. Koehler. An end-to-end tool flow for FPGA-Accelerated scientific computing. *IEEE Design and Test of Computers*, 28(4):68–77, August 2011.

[35] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. Timing-driven placement for fpgas. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 203–213, 2000.

[36] Larry McMurchie and Carl Ebeling. Pathfinder: a negotiation-based performance-driven router for fpgas. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, 1995.