Architecture, Mapping Algorithms and Physical Design of Mesh-of-Functional-Units FPGA Overlays for Pipelined Execution of Data Flow Graphs

by

Davor Capalija

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2017 by Davor Capalija
Abstract

Architecture, Mapping Algorithms and Physical Design of
Mesh-of-Functional-Units FPGA Overlays
for Pipelined Execution of Data Flow Graphs

Davor Capalija
Doctor of Philosophy
Graduate Department of The Edward S. Rogers Sr. Department of Electrical and
Computer Engineering
University of Toronto
2017

The massively parallel fabric of FPGAs allows them to deliver high levels of performance. However, the widespread use of FPGAs by programmers has been hindered by their programmability wall: FPGAs have low levels of abstraction that require hardware expertise and FPGA CAD tools also have long compile times. In order to tackle this challenge, in this thesis we explore an approach that is based on overlays, which are pre-compiled FPGA circuits that are themselves programmable via a software-familiar model without the need for FPGA CAD tools.

We propose, design and evaluate a high-performance mesh-of-functional-units overlay architecture that projects the programming model of the pipelined execution of data flow graphs (DFGs). The overlay architecture consists of cells, with each one containing a functional unit (FU), routing logic with elastic pipelines, and FIFOs in every routing hop. This architecture realizes latency insensitive data-driven execution, facilitates high $f_{\text{MAX}}$ and scalability to large mesh sizes and balances pipeline latencies.

We design a DFG-to-overlay mapping algorithm that places, routes, and balances DFGs on the overlay for high-throughput operation, and uses internal seed-sweep to hide the seed variability from the programmer. We also propose a tile-based bottom-up CAD
flow that is based on the partitioning and floorplanning of the overlay into tiles. In addition to being able to compile overlays to FPGA fabric such that they achieve high $f_{\text{MAX}}$ when scaled to large mesh sizes, the flow also enables the parallel compilation and quick stitching of tiles from a pre-compiled library.

We prototype two overlays on a Stratix IV FPGA that has 212K ALMs: a 355 MHz 24x16 integer overlay and a 312 MHz 18x16 floating-point overlay. We map 16 DFGs and show that the two overlays deliver a throughput of up to 37 GOPS and 22 GFLOPS, respectively. The DFG mapping to the overlay is fast, taking no more than 7 seconds for the largest DFG. Furthermore, we use the tile-based bottom-up flow to compile five large overlays to the FPGA and achieve 37% higher $f_{\text{MAX}}$ than what the flat flow achieves (i.e., the default flow of the CAD tool) with only 8% higher resource use. Compared to the flat flow, which compiles an overlay in 4 hours, the bottom-up flow can stitch together an overlay from pre-compiled tiles in 35 minutes and can compile one from scratch in one hour if tiles are compiled in parallel on a cluster.

Finally, we assess the resource overhead of implementing a DFG on the overlay by comparing to compiling the DFG directly to FPGA fabric using a DFG-to-FPGA tool. The floating-point and integer overlays use respectively 4x and 9x more resources on average compared to the circuits compiled with the DFG-to-FPGA tool. This increase in resources enables an average reduction of 1500x in DFG mapping time across the DFG set, which is a significant decrease.

Thus, we demonstrate that it is possible to design overlays that project a software-familiar programming model and that also expose massive FPGA parallelism, deliver high-performance, scale with increasing FPGA resources and provide a considerable reduction in compile time. However, the many benefits of our overlay architecture do come with a modest resource overhead.
Acknowledgements

This thesis and my PhD studies would not have been possible without the great help and enduring support of many people.

I thank my PhD supervisor Professor Tarek Abdelrahman for patiently and persistently guiding me throughout my PhD. He has played a critical role both in my academic and my personal growth. He painstakingly mentored me in creating, shaping, and executing my ideas, while also teaching me how to take them through the rigor of the scientific process. He has also dedicated a great effort to develop my ability to communicate my work, both in written and verbal form, and instilled in me the significance of great communication. I thank him for believing in me. His commitment was unfaltering, even after I started working at Altera and pushed to finish the writing of the thesis alongside trying to make an impact on the FPGA industry.

I thank Professors Jonathan Rose and Stephen Brown for serving on my PhD committee. They provided invaluable feedback and direction for my research, which greatly improved this thesis. They have also invested themselves into helping me improve my ability to communicate, incorporate the feedback of others, put my ideas to the test, and develop intellectual humility. I also thank late Professor Greg Steffan for serving on my PhD committee in the beginning of my doctoral studies. He was a pioneer in research on overlay architectures and has inspired many to pursue this direction.

I thank Professors Paul Chow and Greg Stitt for serving on the committee of my final PhD exam. Their comments have greatly contributed to making this a better thesis.

I thank my peers at the ECE department who were my continuous source of support, encouragement, advice and help on academic as well as personal matters. While many could be mentioned here, I am grateful to having the friendship and support of Frank Franjo Plavec, Cedomir Segulja, Ivan Matosevic, Utku Aydonat, Adam Czajkowski, Tom Czajkowski, Xander Chin, and Charles Lo.

I am grateful for the encouragement and patience of my managers, my team, and
colleagues at Altera. They have sincerely appreciated my will to finish the writing of the thesis alongside working and they have put great effort in my growth as an engineer, leader and as a person. I thank Deshanand Singh, John Freeman, Peter Yiannacouras, Andrew Ling, Gordon Chiu, Doris Chen, Andrei Hagiescu, Michael Kinsner, Raluca Gheorghe, and Shane O’Connell.

I thank Professor Sinisa Srbljic for his mentorship during my undergraduate studies, his passion and drive in enabling the younger generations to make the world better, and for guiding me to pursue my graduate studies at the University of Toronto. His influence and help has had an immensely positive impact on my life.

My parents Jadranka and Milan, my sister Davorka, and my brother-in-law Miroslav have selflessly given me unconditional love and support in pursuit of my dreams. They have always been my role models of integrity, compassion, and love of life. They are my platform, which enables me to reach higher. Even though leaving to Canada meant being apart, they have always followed me, trusting me that I will do the right thing.

The bravery and love of my wife, soulmate and teammate Jasmina has been my fuel throughout my PhD, and no dream is ever out of reach when we dream together. I thank her for teaching me that dreams are made real with leadership, perseverance, and engineering. She was with me at every hard place on this journey; through all the rises and falls. It was her energy and resilience that pushed me forward. We grew up together, day-by-day, since meeting at UofT, as partners, researchers, and citizens of the world.

My parents-in-law Dragana and Zoran, my brother-in-law Voya and his wife Laura have really been there for me throughout my PhD, never withholding their love, empathy, and warmth. They did so much to make Canada home for me, I never felt alone.

To my friends and colleagues in Canada, Croatia, and elsewhere not mentioned by name, I thank you for helping me make this happen.

Lastly, I thank the Natural Sciences and Engineering Research Council of Canada (NSERC) for their grants that supported this work.
Contents

List of Tables x

List of Figures xii

1 Introduction 1
   1.1 Motivation ................................................................. 1
   1.2 FPGA Overlays .......................................................... 5
       1.2.1 Use Models of Overlays ................................... 6
   1.3 Research Goals ......................................................... 8
   1.4 Research Overview .................................................... 8
   1.5 Thesis Contributions .................................................. 13
   1.6 Thesis Organization ................................................... 14

2 Background 15
   2.1 Field-Programmable Gate Arrays (FPGAs) ............................. 16
       2.1.1 FPGA Architecture ........................................... 16
       2.1.2 CAD Flow ....................................................... 17
       2.1.3 Altera Stratix IV Architecture ............................... 19
       2.1.4 Placement and Routing Algorithms ........................... 21
       2.1.5 Hierarchical CAD Techniques for Physical Design .......... 26
   2.2 Data Flow Graphs (DFGs) ............................................. 28
       2.2.1 Pipelined execution of DFGs ................................. 29
   2.3 Overlay Architectures .................................................. 31
       2.3.1 Mesh of Functional Units (Mesh-of-FUs) Overlay Architectures .. 33
   2.4 Elastic Pipelines ....................................................... 35
   2.5 Latency Balancing of Pipelines ..................................... 38

3 Overlay Architecture Overview 44
   3.1 Design Challenges ..................................................... 47
   3.2 Summary of Approach ................................................ 49

4 Overlay Architecture Implementation 54
   4.1 Overlay Cell Architecture ......................................... 54
       4.1.1 FUs ............................................................... 55
       4.1.2 Routing Logic .................................................. 60
Routing-only Overlay Cells ........................................ 63
4.1.3 Data-Driven Pipeline Units (DDPUs) .......................... 65
2-Slot Elastic Buffers ........................................ 66
N-Slot Elastic FIFO ........................................ 69
Inelastic Register ........................................ 73
The Arrangement and Selection of DDPUs within the Overlay Cell Architecture ........................................ 73
4.1.4 Reconfigurable Join and Fork Synchronization Logic ........ 76
Reconfigurable Join Synchronization Logic .......................... 76
Reconfigurable Fork Synchronization Logic .......................... 79
4.2 Latency Balancing of the Overlay’s Pipeline Paths for Maximizing Throughput ...................................... 82
4.3 Overlay Reconfiguration, Clocking and Reset ................... 88
4.3.1 Reconfiguration Logic ...................................... 88
4.3.2 Overlay Clocking and Reset .................................. 91
4.4 Summary .................................................... 92

5 DFG-to-Overlay Mapping Tool .................................... 94
5.1 Overview .................................................... 94
5.2 Algorithm Inputs, Parameters, and Constraints .................. 98
5.2.1 DFG Description .......................................... 100
5.2.2 Overlay Functional Layout Description ....................... 102
5.3 Placer ...................................................... 103
5.3.1 Definitions ............................................... 104
5.3.2 Cost Function ............................................ 106
5.3.3 Move Generation ......................................... 106
5.3.4 Annealing Schedule ....................................... 109
5.4 Router ...................................................... 112
5.4.1 PathFinder Adaptation .................................... 112
5.4.2 Controlling PathFinder’s Routing Iteration Limit .......... 114
5.5 Latency Balance Checking ..................................... 115
5.6 Finding the Most Balanced Mapping of a DFG .................. 121
5.6.1 Algorithm Heuristics ..................................... 122
5.6.2 Finding the balancing D-FIFO depth of a post-map DFG .... 123
5.7 Balancing Node Insertion ..................................... 124
5.7.1 Motivation and Challenges ................................ 125
5.7.2 ILP Model and Heuristics ................................ 126

6 Overlay Physical Design and Compilation to FPGA ............ 130
6.1 Target FPGA Device and CAD Tool ............................ 133
6.2 Physical Regions ............................................ 134
6.3 Flat Flow .................................................. 134
6.4 Top-Down Flow ............................................. 135
6.5 Floorplanning Granularity: Cells vs. Tiles ....................... 136
6.6 Bottom-Up Flow ............................................ 139
# 7 Experimental Evaluation

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Benchmark DFGs</td>
<td>149</td>
</tr>
<tr>
<td>7.2 Evaluation Platform and Methodology</td>
<td>154</td>
</tr>
<tr>
<td>7.3 Overlay Prototypes</td>
<td>155</td>
</tr>
<tr>
<td>7.4 Evaluation Metrics</td>
<td>158</td>
</tr>
<tr>
<td>7.5 Performance and Scalability</td>
<td>160</td>
</tr>
<tr>
<td>7.5.1 $f_{\text{MAX}}$</td>
<td>160</td>
</tr>
<tr>
<td>7.5.2 Throughput</td>
<td>161</td>
</tr>
<tr>
<td>7.5.3 Latency</td>
<td>163</td>
</tr>
<tr>
<td>7.6 DFG-to-Overlay Mapping Tool</td>
<td>163</td>
</tr>
<tr>
<td>7.6.1 DFG-to-Overlay Compile time</td>
<td>164</td>
</tr>
<tr>
<td>7.6.2 Insertion of Balancing Nodes</td>
<td>166</td>
</tr>
<tr>
<td>7.6.3 Impact of the Overlay’s FIFO Depth on DFG Throughput</td>
<td>166</td>
</tr>
<tr>
<td>7.6.4 DFG Routability and Balancability</td>
<td>172</td>
</tr>
<tr>
<td>7.7 Resource Usage</td>
<td>174</td>
</tr>
<tr>
<td>7.8 Resource Overhead</td>
<td>175</td>
</tr>
<tr>
<td>7.8.1 Discussion of Alternative Approaches</td>
<td>176</td>
</tr>
<tr>
<td>7.8.2 Comparison of the DFG-to-overlay and DFG-to-FPGA Approaches</td>
<td>177</td>
</tr>
<tr>
<td>7.8.3 Analysis</td>
<td>182</td>
</tr>
<tr>
<td>7.8.4 Constraining DFG Mapping to Smaller Overlays</td>
<td>185</td>
</tr>
<tr>
<td>7.8.5 Summary</td>
<td>188</td>
</tr>
<tr>
<td>7.9 Evaluation of Overlay Physical Design Flows</td>
<td>190</td>
</tr>
<tr>
<td>7.9.1 Benchmark Overlays</td>
<td>190</td>
</tr>
<tr>
<td>7.9.2 Experiments</td>
<td>191</td>
</tr>
<tr>
<td>7.9.3 Flat Flow</td>
<td>192</td>
</tr>
<tr>
<td>7.9.4 Top-Down Flow</td>
<td>192</td>
</tr>
<tr>
<td>7.9.5 Bottom-Up Flow</td>
<td>194</td>
</tr>
<tr>
<td>7.9.6 Benefits of the Tile-Based Bottom-up Flow</td>
<td>196</td>
</tr>
<tr>
<td>7.9.7 Impact of Overlay-to-FPGA Functional Layout Matching</td>
<td>199</td>
</tr>
</tbody>
</table>
8 Related Work
8.1 Mesh-of-FUs Architectures ................................. 201
8.2 Place and Route Algorithms for FPGAs ..................... 207
8.3 Partitioning, Floorplanning and Bottom-up Compilation .... 209
8.4 Alternative FPGA Overlay Architectures ....................... 211
8.5 Elastic Pipelines ............................................. 213

9 Conclusions .................................................. 215
9.1 Research Summary ............................................ 215
9.2 Concluding Remarks ......................................... 219
9.3 Future Directions ............................................. 221

A DFG-to-FPGA Tool ........................................... 227

B Benchmark DFGs ................................................. 230

Bibliography ..................................................... 245
# List of Tables

4.1 The FU set: function, pipeline depth and \( f_{\text{MAX}} \). The FUs are generated from Altera’s megafunctions library [114, 115]. .................................................. 56
4.2 FU types: FPGA resource usage (Stratix IV). The prefix “f-” denotes floating point units. .......................................................... 57
4.3 FU types: run-time configurable operations. .............................................. 59
4.4 DDPU types: FPGA resource usage (Stratix IV). ....................................... 70

5.1 The temperature update schedule \( \gamma(\alpha) \) function; this schedule is the same as in [68, 124]. .............................................. 110

6.1 Stratix IV FPGA characteristics .............................................................. 133

7.1 DFG characteristics. The table also contains the number of balancing nodes inserted by the DFG-to-Overlay mapping algorithm, and the ratio of input nodes to compute nodes. ...................................................... 151
7.2 Overlay characteristics (CR=compute-routing cells, R=router-only cells). 158
7.3 Achieved throughput and latency of the DFGs on OV-i and OV-f. ........... 162
7.4 DFG-to-Overlay mapping tool parameters. ............................................ 164
7.5 DFG-to-overlay compile times to OV-f with 64-deep FIFOs, and OV-i with 32-deep FIFOs. .............................................................. 165
7.6 Resource usage of each overlay and resources used only by the FUs of each overlay. ................................................................. 174
7.7 DFGs mapped directly to FPGA fabric using the DFG-to-FPGA flow (DFG-to-HDL and Quartus II tools). ........................................... 180
7.8 DFGs mapped to the entire overlay mesh using the DFG-to-Overlay flow. 181
7.9 Comparison of DFG-to-Overlay and DFG-to-FPGA approaches (DFGs are mapped to the entire mesh of the overlay). ....................... 182
7.10 DFGs mapped to entire overlay mesh. It shows: the percentage of ALMs of the entire OV-f/i overlays, the usage of compute-routing and routing-only mode cells and the total usage of cells. Also shown is the ratio of routing-only and compute-routing cells. 184
7.11 DFGs mapped to the smaller overlays that correspond to regions of the base overlays (OV-f/OV-i). .......................................................... 186
7.12 Comparison of DFG-to-Overlay and DFG-to-FPGA approaches (DFGs are mapped to the smaller overlays). ................................. 187
7.13 DFGs mapped to the smaller overlays. The DFG resource usage on the smaller overlays in terms of ALMs and as a percentage of the entire base overlay (OV-f/OV-i), the usage of compute-routing and routing-only mode cells in the region and the total usage of overlay cells in the region. Also shown is the ratio of routing-only and compute-routing cells.

7.14 DFGs mapped to the entire base overlay and to the smaller overlays (i.e., rectangular regions of base overlays). The resource usage in ALMs, the usage of compute-routing mode cells, routing-only mode cells, and the total usage of cells are compared.

7.15 Breakdown of five custom overlays (12×15 mesh) across FU type.

7.16 Resource use of five overlays (12×15 mesh) and f_{MAX} achieved with flat, top-down and bottom-up flows. The bottom-up flow adds resource cost (see section Section 7.9.5).

7.17 Average compile time (hours:minutes) of the benchmark overlays (12×15 mesh) using best-of-5 seeds method across different flows.
List of Figures


1.2 Relative increase (since 1998 to 2010) of logic cell count of largest Altera FPGA (data points and plot from [43]).

2.1 A netlist for a simple circuit (adapted from [50]).

2.2 A region of a Stratix IV device showing its column-based layout. This region shows one DSP column, one M144K column, two M9K columns, and several LAB/MLAB columns.

2.3 Pseudo-code for a generic high-level simulated annealing-based placer [50, 61].

2.4 (a) The circuit for a Triptych FPGA cell and (b) its representation in PathFinder by the resource routing graph [61]. The top of the figure highlights the representation of a 2-to-1 mux with a fan-out of 3.

2.5 A simple DFG with 5 compute, 5 input, and 2 output nodes.

2.6 Multiple instances of a DFG from Figure 2.5 operating on independent data.

2.7 Pipelined execution of independent data on a DFG from Figure 2.5.

2.8 (a) EBs in a linear pipeline, (b) an example 2-slot EB.

2.9 Linear inelastic pipeline with a global stall signal.

2.10 (a) Pipeline for the DFG from Figure 2.5 before balancing, (b) transformed pipeline by insertion of balancing registers (shaded). The registers reflect the latencies of the FUs and of their connections.

2.11 DFG annotated with variables for the ILP formulation.

3.1 (a) Overlay architecture with 4-NN topology, an example 4×4 mesh with a custom functional layout, (b) high-level view of overlay cell architecture.

3.2 The DFG from Figure 2.5 mapped to a 3×3 overlay.

4.1 Overview of overlay cell architecture.

4.2 A DDPU wrapper for a 2-input FU.

4.3 A Multi-FU DDPU wrapper that contains two FU DDPU wrappers.

4.4 Routing connections of a regular overlay cell: (a) connections that bypass the FU, (b) connections that feed in and out of the FU.

4.5 Overlay cell’s data-paths and routing muxes (synchronization logic is not shown).
4.6 Fan-out cascading across multiple neighbouring overlay cells.  

4.7 (a) An example functional layout with routing-only cells (marked with X), (b) routing connections of a routing-only overlay cell.  

4.8 Six versions of a 2-slot elastic buffer implementation: (a)(b)(c) type 1 versions, (d)(e)(f) type 2 versions.  

4.9 N-slot elastic FIFO.  

4.10 Inelastic register.  

4.11 DDPU selection design constraints. Two issues that arise when D-REGs are used for all input and output port DDPU of the overlay cell: 1) combinational loop of the stall signal; 2) long paths of the stall signal.  

4.12 The selection and arrangement of DDPU types in the overlay cell architecture.  

4.13 (a) join synchronization logic between FU$_{IN1}$, FU$_{IN2}$ and FU DDPU wrapper, (b) fixed 2-input join, (c) configurable 1-input/2-input join, (d) configurable 2-input join with a constant lock-in.  

4.14 The design of a fork sender (FS) and a fork receiver (FR). An example overlay cell configuration with two forks.  

4.15 (a) An example DFG with two sub-DFGs highlighted (A and B), b) mapped to a 3x3 overlay.  

4.16 A post-map DFG for the sub-DFG A from Figure 4.15. It demonstrates dynamic latency balancing of two routed paths from $I_3$ and $I_4$ that join at node B.  

4.17 A post-map DFG for the sub-DFG B from Figure 4.15. Shown are the steady-state latencies of each pipeline unit and the latencies of paths joining at FUs.  

4.18 An example 3x3 overlay with overlay reconfiguration control, reconfiguration memory with 2 overlay bitstream, the details of overlay cell reconfiguration logic (RECONF), and per-cell reset synchronizer (RS).  

5.1 High-level pseudo-code of the DFG-to-overlay mapping algorithm.  

5.2 (a)(d) Two example DFGs, (b)(e) described with PyDFG API, and (c)(f) described in DOT language.  

5.3 (a) An example 4x4 overlay, (b) its functional layout represented with a 6x6 array.  

5.4 (a) A simple DFG with 9 nodes and 5 nets, (b) placed-and-routed on a 2x3 overlay.  

5.5 VPlace $R_{limit}$ update schedule [68, 124].  

5.6 Adapted $R_{limit}$ update schedule for DFG-to-overlay mapping.  

5.7 Balancing DFG created for the DFG B from Figure 4.15a and its post-map DFG in Figure 4.17.  

5.8 Balancing DFG created for the minimal mapping of an example DFG.  

6.1 Impact of region size on f$_{MAX}$ and logic utilization. An add/subtract (A) overlay cell is compiled to regions of increasing size (each data point averaged over 5 seeds).
6.2 Constraints for an overlay I/O register connected to the boundary overlay cell (2,1) of a 4×4 mesh overlay: (a) I/O register allowed anywhere in the large region surrounding the overlay; (b) I/O register allowed only within a fine-grained I/O region.

6.3 (a) 8×10 mesh overlay - functional layout with four 4×5 tiles, (b) compiled with top-down flow and cell-based floorplanning, (c) compiled with top-down flow and tile-based floorplanning, requires a smaller overlay region on Stratix IV – figures (b) and (c) have matching scale.

6.4 “Mirrored” inter-tile EBs inserted on an inter-tile path.

6.5 The bottom-up compilation of a 4×8 overlay by stitching two 4×4 tiles; tiles 1 and 2 are compiled independently using fine-grained tile I/O constraints.

6.6 (a) a template tile-based floorplan with 9 regions on a Stratix IV device, (b)(c) OV_A and OV_B (each has a 12×15 mesh with nine 4×5 tiles) compiled from-scratch to the template floorplan, (d) OV_C is compiled by stitching a mix of tiles from OV_A and OV_B.

6.7 An example of matching a functional layout of a 3x10 overlay to a section of Stratix IV FPGA with two DSP columns. The desired physical placement of three overlay DSP-based cells from DSP overlay columns is shown. Overlay cell codes: D=div, L=log, S=sqrt, A=addsub, C=abs-cmp and R=routing-only cell.

7.1 (a) OV-i 24×16 functional layout, (b) OV-i partitioning into 20 tiles, (c) OV-f 18×16 functional layout, (d) OV-f partitioning into 16 tiles.

7.2 The smallest balancing D-FIFO depths \(d_{\text{min}}\) for the benchmark DFGs mapped to OV-i and OV-f instances.

7.3 Impact of varying the D-FIFO depth of OV-f overlay instances on throughput of N-Body DFGs.

7.4 Impact of varying the D-FIFO depth of OV-f overlay instances on throughput of MatMul, MatMulAndAdd and BlackScholes DFGs.

7.5 Impact of varying the D-FIFO depth of OV-i overlay instances on throughput of RGB2YIQ and SAD DFGs.

7.6 Impact of varying the D-FIFO depth of OV-i overlay instances on throughput of GaussianBlur, MatMul and MatMulAndAdd DFGs.

7.7 Percentage of placements that are routable and their breakdown into balanced routable placements and imbalanced routable placements.

7.8 Flat flow \(f_{\text{MAX}}\) (data points best-of-5 seeds).

7.9 Top-down flow \(f_{\text{MAX}}\) (data points best-of-5 seeds).

7.10 Bottom-up flow \(f_{\text{MAX}}\) (data points per-tile best-of-5 seeds).

7.11 Benefit of inter-tile EBs: bottom-up compile of OV_2 with and without inter-tile EBs (data points per-tile best-of-5 seeds).

7.12 Comparison of cell-based and tile-based floorplanning using top-down flow on OV_2: (a) floorplan size, (b) \(f_{\text{MAX}}\) (data points best-of-5 seeds).
7.13 Flat flow $f_{\text{MAX}}$ comparison for an overlay with layout matching and an overlay w/o layout matching. The two overlays have the same set of cells (data points averaged over 5 seeds).

B.1 N-Body DFG ............................................................. 231
B.2 N-Body-2x DFG .......................................................... 232
B.3 N-Body-3x DFG .......................................................... 233
B.4 BlackScholes DFG ....................................................... 234
B.5 MatMul DFG (SP FP and int32 variants have the same topology) .... 235
B.6 MatMulandAdd DFG (SP FP and int32 variants have the same topology) 236
B.7 RGB2YIQ DFG ............................................................ 237
B.8 RGB2YIQ-2x DFG ....................................................... 238
B.9 RGB2YIQ-4x DFG ....................................................... 239
B.10 RGB2YIQ-5x DFG ...................................................... 240
B.11 SAD DFG ................................................................. 241
B.12 SAD-2x DFG ............................................................ 242
B.13 GaussianBlur DFG ..................................................... 243
B.14 GaussianBlur-2x DFG ................................................ 244
Chapter 1

Introduction

1.1 Motivation

The 35-year period spanning from the early 1970’s to the mid-2000’s was marked by one of the most impressive trends in the history of computer architecture, namely the exponential scaling of the performance of single-threaded microprocessor architecture (Figure 1.1). The exponential increase in on-die transistor counts and transistor speeds in each new generation of technology was oriented towards the continuous enhancement of three key single-thread microarchitectural techniques: increasing the clock frequency, improving the dynamic extraction of instruction-level parallelism, and constructing deeper and larger cache hierarchies [1–3].

The ability to exponentially increase both transistor count and transistor switching speed while keeping the overall power consumption of a device near constant was enabled by Dennard’s scaling, which is the ability to scale down a transistor’s supply voltage and capacitance in order to reduce the power consumption of subsequent generations [2, 3, 5]. The scaling of supply voltage significantly slowed down in early 2000’s. As a result, the doubling of each generation’s transistor count prevents an increase of the clock frequency if overall power consumption is to remain constant. The manageable power consumption
Chapter 1. Introduction

Figure 1.1: The end of single thread performance scaling. Dotted line extrapolations by C. Moore [4]. Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten [3].

of a given device plateaued at about 200 Watts due to thermal limits, i.e., the inability to dissipate more heat without damaging the device, as well as the costs associated with both the increased power consumption and the required cooling systems [6]. The overarching result of these technological shifts was the end of frequency scaling, and with it came the end of the exponential increase of single-thread performance.

Taking the fixed power budget as a fundamental design constraint, the key to obtaining better performance is to improve the energy efficiency of the architecture (performance per unit of power). This insight prompted a shift in computer architecture design towards heterogeneous architectures in which traditional large cores optimized for single-thread performance are coupled with a programmable accelerator. This accelerator often consists of a many-core processor that incorporates a multitude of smaller and specialized cores. One example of an accelerator is a general-purpose graphics processing unit (GPU), which incorporates hundreds of small streaming cores that have been specialized for the high-throughput execution of data-parallel kernels [7].

The emergence of heterogeneous architectures triggered the introduction of new programming models such as CUDA [8], OpenCL [9], and OpenACC [10], which are used for
writing parallel programs for these architectures. These models enable the programmer to partition a program into serial sections that execute on the CPU, and *kernels*, which are compute-intensive parallel code sections, that execute on the accelerator.

Similar to a GPU, a field-programmable gate array (FPGA) is a massively parallel device that can be geared towards the acceleration of compute-intensive parallel code sections (i.e., kernels). The differentiating feature of an FPGA is its flexibility to form a *custom architecture* by implementing a custom circuit in its fine-grained reconfigurable fabric. A circuit on an FPGA can be customized to the application’s kernels in multiple ways. The FPGA’s fabric can be configured to implement function-units matching the operators of the kernels [11]. Additionally, the bit-widths of the circuit’s data-path can be customized to match the data types in the kernels [12]. Similarly, function units can be connected in a topology that directly represents the data-flow of the kernels [13]. Finally, the FPGA’s memory blocks can be organized in an arbitrary memory hierarchy [14]. This high degree of application-driven customization is one of the key factors that give FPGAs an energy-efficiency advantage over fixed-architecture accelerators across a wide range of application domains [15–20].

Traditionally, FPGAs have been used in a variety of specialized market segments, such as broadcast, communications, manufacturing, and the automotive industry [21]. The lack of software-programmer-friendly and higher-level abstraction tools had previously limited the use of FPGAs. They predominately had to be “programmed” by hardware experts who had the necessary knowledge of low-level hardware design, hardware description languages (HDL), FPGA CAD tools, and timing analysis.

Nonetheless, recent years have seen increasing interest in the use of FPGAs as general purpose programmable accelerators alongside CPUs, GPUs and digital signal processing (DSP) processors [16, 20, 22, 23]. PCIe-based accelerator cards with FPGAs are commercially available from a range of vendors [24–28]. Furthermore, the introduction of a platform that tightly couples a Xeon CPU and an FPGA [29], and Amazon’s offering
of an EC2 F1 instance [30] within its cloud computing platform both reflect the current trend towards more mainstream FPGA usage.

Alongside the FPGA-based platforms, high-level synthesis (HLS) tools that translate C++ or a C-based language such as OpenCL to HDL have become available from Altera [31], Xilinx [32], Maxeler [33], Calypto [34], as well as through the academic open-source LegUp project [35]. While these tools raise the level of abstraction at which a program is written, they still suffer from very long compile times and do not entirely insulate the software programmer from the hardware intricacies of FPGA design. The HLS-generated HDL must still go through the CAD tool, which entails a lengthy process of synthesis, place, route, and timing analysis. This process can take multiple hours or even several days for a large circuit.

This programming challenge is a major obstacle for software programmers who are accustomed to short compile times and fast develop-test-debug cycles. The long FPGA compile times are further exacerbated when, after a lengthy multi-hour compile, the circuit does not fit on the FPGA and the programmer is left with no circuit to test. Making a circuit fit often requires non-intuitive code changes (from a software perspective), such as reducing the number of floating-point operations in the code or reducing the number of compute kernels. Furthermore, small code changes may cause a drop in maximum clock frequency ($f_{\text{MAX}}$) due to CAD noise\(^1\) that is inherent to FPGA CAD tools. The concepts of no-fit and $f_{\text{MAX}}$ drop due to CAD noise are non-existing phenomena in software-programmed architectures that have a fixed clock frequency and also allow for programs to contain a practically “unlimited” number of operations.

Collectively, the challenges described above are referred to as the FPGA programmability wall [38]. Thus, there is a growing need for tools that can increase the speed of

\(^1\)The term CAD noise refers to the variations in circuit quality (e.g., $f_{\text{MAX}}$ and resource usage) that arise because CAD algorithms are heuristics for solving NP-complete problems and they often rely on randomization based on a seed value. When the variations are a result of seed changes, the CAD noise is often called the seed noise. In certain CAD algorithms, noise may also arise due to small perturbations such as small circuit modifications (e.g., changing the width a single signal from 64 to 63 bits) or small design constraint changes (e.g., changing the target operating frequency from 401 to 400 MHz) [36, 37].
FPGA development cycles and insulate software programmers from the hardware intricacies of FPGA CAD tools. The development of these tools is critical if FPGAs are to become widely adopted as general purpose accelerators and be able to compete with software-programmed GPUs and DSP processors. In this thesis, we explore an approach that is based on overlays.

### 1.2 FPGA Overlays

An overlay is a pre-compiled FPGA circuit with a programmable architecture [39–41]. In effect, FPGA users can target this programmable architecture and use its associated tools to implement their custom circuits instead of using hardware design and FPGA CAD tools.

There are several potential advantages of overlays that make them a candidate for tackling the programmability wall of FPGAs. An overlay’s architecture aims to project a higher-level programming model that abstracts away the low-level model of an FPGA and thus insulates the software programmer from the intricacies of hardware design. Since overlays are pre-compiled, programmers would not need to use FPGA CAD tools, they would only need to use the software tools that map the application’s kernels to an overlay architecture. Depending on an overlay’s higher-level programming model, it may be possible to design these software tools to be much faster than the FPGA CAD while offering a familiar CPU-like development flow. Also, the building blocks of an overlay architecture could be designed to be coarser than the fine-grained LUTs of an FPGA. As a result, the overlay would require less configuration bits than the FPGA, which would enable faster reconfiguration time.

Nonetheless, overlays unavoidably have a resource overhead compared to a hard-wired circuit that implements the same functionality. This is because FPGA resources are spent to form the programmability of the overlay’s architecture.
However, it remains an open research question whether it is feasible to design an overlay architecture that realizes the above potential advantages while also delivering performance that is comparable to that of a hard-wired circuit. Applications that achieve a high-performance on FPGAs are able to leverage FPGA’s unique combination of massive parallelism, flexibility, and exponential increase in resources with each subsequent FPGA generation (Figure 1.2). Specifically, it is unclear whether it is possible to design a high-performance overlay architecture that preserves these unique advantages of the FPGA architecture. That is, it is unclear whether an overlay architecture can be designed such that: 1) it exposes the massive parallelism of the FPGA fabric; 2) it has the flexibility to be customized for the application’s kernels (or a broader application domain); and 3) it can scale with increase in FPGA size. Finally, the extent of resource overhead of such an architecture and the impact of this overhead on performance are also not known. This work attempts to provide answers to some of these questions.

1.2.1 Use Models of Overlays

There are several use models of overlays that may facilitate the adoption of FPGAs as accelerators. In general, overlays have the potential to be an enabling technology in use models where lowering the entry threshold for FPGA use is critical or where cross-platform compatibility is required (this can be achieved by implementing the same overlay on different platforms).

In the first use model, the overlay-based implementation is deployed in the initial phase of the product’s lifetime. The reason for this can be to allow for frequent updates to functionality (enabled by the overlay’s fast develop-test-debug cycle). It can also be because the application’s kernels are in their beta stage and testing in the field is the primary objective. In cases where the overlay-based implementation does not end up meeting the performance requirements of the end-product, the kernels can be “hardened” – once their functionality matures – by compiling them directly to the FPGA
Figure 1.2: Relative increase (since 1998 to 2010) of logic cell count of largest Altera FPGA (data points and plot from [43]).

using an HLS- or HDL-based solution. A hypothetical example of this use model is the deployment of a beta search feature of the Bing search engine, which already uses FPGAs to accelerate its search features [42].

In the second use model, the implementation of the kernels on the overlay meets the performance requirements of the end-product and the overlay-based implementation is deployed as the actual solution. Note that, while an HLS- or HDL-based solution may be capable of achieving better performance, an overlay-based solution delivers good enough performance for the end-product. Hence, an HLS- or HDL-based solution may not be necessary.

The third use of overlays is as a prototyping platform. If an application is initially developed using the overlay, it can be modified and tested more frequently on the actual accelerator platform using its real I/O interfaces.

In all of the above use models of overlays, there is a direct relationship between the performance of the underlying overlay architecture and the usefulness and the applicability of the overlay technology. Specifically, the higher the performance of the underlying overlay architecture the more useful and applicable the overlay technology will be.
1.3 Research Goals

The overarching goals of this work are to explore the feasibility of overlays as a candidate for tackling the FPGA programmability wall and to validate their potential advantages. We specifically focus on the challenge of the design of a high-performance overlay. That is, we focus on the open question of which overlay architecture and which programming model are most suitable for exposing the FPGA architecture’s key advantages of massive parallelism, customizability, and generational scaling. We endeavor to address this research challenge through three components.

The first component of our research is to define an overlay architecture that enables high-performance both through high $f_{\text{MAX}}$ and high throughput (in terms of operations per cycle), and also projects a software-like programming model that can express parallel code sections. The second component is to design an efficient FPGA CAD flow for the compilation of the overlay architecture to the FPGA fabric. The flow should enable the overlay architecture to scale to a large size (i.e., large number of operations) while maintaining a high $f_{\text{MAX}}$: a large overlay instance should achieve the same high $f_{\text{MAX}}$ as a small overlay instance. Furthermore, the flow should also enable the quick compilation of custom overlay instances and enable the creation of a library of reusable overlay components. Finally, the third component is to develop an algorithm that can map parallel code sections to the overlay. The algorithm should be fast, achieve high throughput on the overlay, and isolate the programmer from the hardware intricacies of the FPGA design process.

1.4 Research Overview

In this thesis, we describe the design, implementation and evaluation of a mesh-of-functional-units (mesh-of-FUs) overlay architecture for the pipelined execution of data flow graphs (DFGs), which are graphs whose nodes are machine operations and whose
edges represent the flow of data between pairs of operations. A mesh-of-FUs is a regular array of FUs that are interconnected through reconfigurable routing logic. Specifically, our overlay architecture is a four-nearest-neighbor connected mesh of overlay cells, each consisting of a functional unit (FU) that is surrounded by reconfigurable routing logic. The 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. Multiple instances of the DFG are then executed in a pipelined fashion on the overlay to achieve high throughput.

We assume that the input model to the overlay is DFGs irrespective of how they are generated. In this work, we extract DFGs manually from parallel code sections. In the remainder of the thesis, in the context of the overlay and mapping algorithm, the term “target applications” refers to the DFGs that were extracted from parallel code sections.

The mesh-of-FUs overlay architecture has the potential to exploit the three key FPGA advantages described above. First, it naturally exposes the FPGA’s massive parallelism through an array of FUs that can execute in parallel. Second, it can also be customized for certain application domains; for instance, a mesh can contain different FUs (e.g., mul, div, exp, sqrt), and an overlay instance can be customized through the choice and arrangement of FUs within the mesh. Finally, mesh-of-FUs has the potential to keep scale with FPGAs by increasing the mesh size to match the size of each new generation of FPGAs.

We chose DFGs as our programming model because they offer a balance between their usability as a common parallel programming abstraction and their inherent potential for pipelined high-throughput execution. DFGs can be used as an abstraction for expressing parallel programs, such as streaming and digital signal processing (DSP) [44, 45]. In addition, the pipelined execution of DFGs can be extracted from modern parallel languages. For example, OpenCL executes multiple instances of a kernel independently with no sharing of data; thus, instances of the body of a kernel can be executed in a
pipelined fashion. Similarly, DFGs can be extracted from the bodies of parallel loops and executed in pipelined fashion [46].

Our overlay architecture uses data-driven execution: an FU executes an operation only when its inputs are ready and when its output is free. We achieve data-driven execution through an elastic pipeline design [47, 48], and we also leverage elastic pipelines to design self-contained overlay cells. As a result, the overlay is modular with an entirely distributed pipeline flow control free of global signals. Such architecture enables scaling to a large number of FUs and maintains high $f_{\text{MAX}}$ when compiled with our tile-based bottom flow (described below). Furthermore, we tailor the design of these elastic pipelines for FPGA fabric and mix them with inelastic pipelines within the overlay cell to make sure that resource overhead and $f_{\text{MAX}}$ are balanced.

We design overlay cells with deep elastic FIFOs (e.g., capacity of 32) at their inputs that provide a latency margin at each routing hop. This facilitates the latency balancing mechanism of the overlay pipelines by allowing the latency of shorter paths (lower latency) in the DFG to be “stretched”, and thus balance the latency of longer (higher latency) paths.

We design a DFG-to-overlay mapping tool that uses a set of fast algorithms to find the most latency balanced DFG mapping and to maximize the throughput of pipelined DFG execution. This tool uses simulated annealing-based placement and PathFinder-based routing [49]. Furthermore, it uses an integer linear programming (ILP) based latency balance-checking algorithm that tests whether a given DFG mapping on the overlay is balanced, i.e., if the latency margins of given mapping are sufficient to “stretch” shorter DFG paths to achieve latency balance with longer paths. We leverage the fast runtime of the tool and perform internal seed-sweeping (transparently to the user) in order to generate multiple DFG-to-overlay mappings, and, once these mappings have been generated, we can then select the one that is the most latency balanced. Finally, since the overlay is pre-compiled and its $f_{\text{MAX}}$ is fixed (does not depend on the DFG), the tool
does need to perform timing analysis, which is a key factor in insulating the programmer from the hardware design.

We show that, when the mesh-of-FUs architecture is scaled to span the size of a large FPGA and then compiled with a flat flow (i.e., the default flow of FPGA CAD), mesh-of-FUs suffer a large drop in $f_{\text{MAX}}$. This occurs despite the regular array-based topology and distributed flow control of the overlay’s pipelines, as a flat flow cannot exploit the regular and distributed structure of the overlay circuit. The flat flow compiles the entire overlay circuit from scratch. Thus, it also suffers from additional shortcomings: it does not allow for the quick compilation of new overlays or the quick modification of existing ones, and it cannot reuse pre-compiled overlay components to build new overlays.

We address the above shortcomings with a \textit{tile-based bottom-up} CAD flow. This flow utilizes hierarchical physical design techniques (partitioning and floorplanning) to divide the overlay into \textit{tiles}, which are groups of adjacent overlay cells. The tiles are then compiled into a coarse-grain rectangular floorplan that allows for the tight placement of differently sized FUs within a tile for resource efficiency. Additional elastic buffers are inserted on inter-tile paths to ensure that they are not a bottleneck for the $f_{\text{MAX}}$ of the overlay.

The tile-based bottom-up flow enables \textit{independent} compilation of each tile into its designated region within the floorplan. A large overlay can then be compiled with high $f_{\text{MAX}}$ by “stitching” a set of pre-compiled high-$f_{\text{MAX}}$ tiles. The use of rectangular coarse-grained floorplans allows for overlays of the same mesh size to be compiled to the same \textit{template floorplan}. This facilitates the design of a \textit{tile library}, which is a collection of pre-compiled tiles that can be reused to quickly stitch together new overlays. Furthermore, independent tile compilation enables the parallel compilation of tiles, presenting an opportunity for reducing the overlay compilation time.

We implement two prototype overlay instances on a Stratix IV FPGA: a $24 \times 16$ integer-FU overlay that operates at 355 MHz, and an $18 \times 16$ floating-point-FU overlay.
that operates at 312 MHz. The high $f_{\text{MAX}}$ of these large overlays is attained by matching
the heterogeneous layout of the FUs within the overlay to the layout of blocks within an
FPGA and by compiling the overlays using a bottom-up tile-based methodology.

We use the DFG-to-overlay mapping tool to map 16 DFGs (extracted from 7 parallel
code sections) to the two overlays; the mapping tool achieves latency-balanced mapping
for all DFGs. We show that the two overlays achieve an operation throughput of up to
35 GOPS (billions of integer operations per second) and 22 GFLOPS (billions of floating
point operations per second), respectively. We also show that mapping the DFGs to the
overlays takes no more than a few seconds.

The resource overhead involved in using overlays to execute DFGs is measured by
comparing their resource usage to the direct DFG compilation to FPGA fabric (i.e., the
hard-wired DFG circuit). For the floating-point DFGs, the overlay-based implementation
incurs overhead between 3.8x and 4.6x over the direct approach, while the overhead
incurred by the integer DFGs is between 7.5x and 10.8x.

This overhead in consumed resources is expended in order to achieve significant re-
ductions in compile time. For the floating point DFGs, the DFG-to-Overlay tool is three
orders of magnitude faster (900x to 2800x) than the DFG-to-FPGA compilation flow, and
it is two to three orders of magnitude faster (200x to 2000x) for the integer DFGs. We
conclude that our proposed overlay architecture is capable of delivering high-performance
execution of data flow graphs with modest resource overhead given the reduction in com-
pile time.

Furthermore, we conduct a comparative evaluation of the tile-based bottom-up flow
against the flat and top-down flows on 5 custom overlays. Our results show that: (1)
the bottom-up flow achieves 37% higher $f_{\text{MAX}}$ on average than the flat flow and slightly
higher $f_{\text{MAX}}$ than the top-down flow; (2) there is little $f_{\text{MAX}}$ degradation with the bottom-
flow (no more than 22%) as the overlay’s mesh is scaled from 1x1 to 12x15 overlay cells,
whereas $f_{\text{MAX}}$ degrades up to 45% and 26% with the flat and top-down flows; and (3) the
use of inter-tile buffers adds no more than 8% resource overhead.

Finally, we use the bottom-up flow to build a small library of 45 pre-compiled tiles that can be used to stitch together a new overlay in 35 minutes while achieving high $f_{\text{MAX}}$. In comparison, the same overlay would take 4 hours to compile using the flat flow. The bottom-up flow can also compile an overlay from scratch in an hour using multiple machines. In this process, the tiles are first compiled in parallel (one tile per machine) before being stitched together.

1.5 Thesis Contributions

This thesis makes the following contributions:

1. A high-performance mesh-of-FUs overlay architecture for the pipelined execution of data-flow-graphs. The architecture’s salient features are data-driven execution, FPGA-tailored elastic pipelines and self-contained overlay cells that work together to facilitate high $f_{\text{MAX}}$, and the latency balancing mechanism based on the latency margins of deep elastic FIFOs embedded in each routing hop.

2. A fast DFG-to-overlay mapping tool that finds the most latency-balanced DFG mapping to maximize the throughput of pipelined DFG execution. This tool’s first salient feature is a combination of placement, routing and latency-balance checking algorithms to find a DFG-to-Overlay mapping. The algorithm’s second salient feature is enabled by its fast run-time. It is an internal seed-sweep that generates multiple DFG mappings and the algorithm then picks the most balanced one.

3. A tile-based bottom-flow that utilizes circuit partitioning, floorplanning, and tile isolation via inter-tile elastic buffers to achieve high $f_{\text{MAX}}$ and scalability of large custom mesh-of-FUs overlays. The flow reduces the compilation time of the mesh-of-FUs through the independent compilation of tiles and fast tile stitching. It also facilitates tile reuse and the creation of a pre-compiled tile library.
4. Experimental evaluation of the overlay architecture and DFG-to-mapping algorithm using two large overlay prototypes: a 24x16 integer-FU overlay that achieves 355 MHz, and an 18x16 floating-point-FU overlay that achieves 312 MHz. We map 16 DFGs extracted from 7 parallel code sections to the two overlays and achieve throughput of 35 GOPS and 22 GFLOPs, respectively. We also show that mapping the DFGs to the overlays takes only a few seconds.

5. Experimental evaluation of the tile-based bottom-up flow on 5 custom overlays. This evaluation shows that the tile-based bottom-up flow achieves 37% higher f_{\text{MAX}} than the flat flow for the large overlays. We also show that bottom-up flow can compile an overlay in 35 minutes by reusing and stitching together pre-compiled tiles. This is significantly faster than the flat flow, which takes 4 hours to compile the same overlay. We project that the use of multiple machines can enable the bottom-up flow to compile an overlay from scratch in an hour.

1.6 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 provides necessary background material; Chapters 3 and 4 describe the overlay architecture; Chapter 5 presents the DFG-to-overlay mapping algorithm; Chapter 6 describes the physical design of the overlay architecture, including the tile-based bottom-up and top-down flows; Chapter 7 presents the experimental evaluation of the overlay architecture, DFG-to-mapping flow, and physical design flows; Chapter 8 reviews related work; and, finally, Chapter 9 provides concluding remarks and directions for future work.
Chapter 2

Background

This chapter provides background material related to the FPGA architecture, its CAD flows, and the CAD techniques used for the physical design of FPGA circuits. It also describes the salient features of the Stratix IV’s architecture, which is the target architecture of our experimental evaluation. The chapter then elaborates on the simulated annealing-based placement algorithm and the PathFinder routing algorithm used in the FPGA CAD. These algorithms are also central to our DFG-to-overlay mapping algorithm.

Furthermore, this chapter describes DFGs and their pipelined execution in hardware. Next, the mesh-of-FUs overlay architecture and other types of overlay architectures are introduced and their key characteristics are elaborated upon. Following this, the basics of elastic pipelines, which are the key building blocks of our overlay architecture, are introduced. The chapter closes by describing techniques and algorithms for pipeline balancing as they play an important role in our DFG-to-overlay mapping algorithm.
2.1 Field-Programmable Gate Arrays (FPGAs)

2.1.1 FPGA Architecture

An FPGA is a prefabricated programmable integrated circuit that is designed as an array of programmable logic blocks and routing switches. A digital logic circuit is implemented by configuring the logic blocks and routing switches using a bitstream [50, 51]. We begin by describing the generic features of an FPGA architecture commonly found in existing commercial architectures as well as research architectures proposed in academia [50–55]. Once the generic features of these architectures have been described, we move on to discuss the features specific to the Stratix IV architecture that we target in this work.

The main building block of an FPGA is a basic logic element (BLE), which traditionally has a 4-input look-up table (4-LUT) and a flip-flop (FF). A 4-LUT can implement any 4-input logic function by programming its SRAM bits, and the LUT can either feed or bypass the flip-flop via a programmable multiplexer. BLEs in modern FPGAs can contain multiple flip-flops per each LUT and can also contain LUTs with more than 4 inputs that are fracturable into smaller LUTs [56, 57]. For example, 6-LUTs in Xilinx Virtex-6 devices [56] can be fractured into two 5-LUTs that share some of the inputs. Furthermore, modern FPGAs have a hierarchical architecture in which several BLEs are grouped into logic clusters referred to as logic array blocks (LABs) in Altera devices [57] or slices and configurable logic blocks (CLBs) in Xilinx devices [56].

Most commercial FPGAs employ an island-style architecture in which LABs are arranged in a two dimensional mesh with rows and columns of routing channels distributed throughout the device [51, 53]. Each routing channel contains a number of routing wires (up to 185 in [58]). The wires in the channels are segmented and there are wire segments of different lengths that aim to provide the most suitable length for each connection [53]. A switch block connects wires in adjacent channels through programmable switches. A connection block also has programmable switches that connect the LAB’s inputs and
outputs to the wire segments surrounding it. BLEs can be connected via a series of wire segments by configuring the SRAM bits of programmable switches in the connection blocks and routing switches. The routing channels, switch blocks, and connection blocks are often collectively referred to as routing fabric, interconnect, or simply routing. The advantage of a hierarchical FPGA architecture is that the routing fabric internal to the LABs is faster and also consumes less silicon area than the routing fabric that connects different LABs together [50]. However, this requires the added expense of more complicated CAD algorithms which are necessary to identify related circuit elements and cluster them into LABs.

In addition to configurable logic and routing fabric, FPGAs also contain other programmable blocks such as I/O blocks, clock distribution networks, memory blocks, phase-locked loops (PLLs), and dedicated “hard” blocks. A typical hard block is a digital signal processing (DSP) block that has dedicated circuitry that implements multiply and accumulate operations. These blocks are usually 9 or 18 bits wide and can be combined to perform multiply-accumulate operations of larger bit-widths. Hard blocks have limited programmability but are faster and smaller (in terms of silicon area) than an equivalent implementation that uses only BLEs. For this reason, hard blocks reduce the cost and improve the performance of digital circuits that make heavy use of multiply-accumulate operations [59, 60].

### 2.1.2 CAD Flow

The FPGA CAD flow transforms a circuit described in a hardware description language (HDL)–such as Verilog or VHDL–into an FPGA bitstream [50, 51]. The flow commonly consists of eight main stages: RTL elaboration, logic synthesis, technology mapping, clustering, placement, routing, timing analysis, and bitstream generation.

The central data structure in the FPGA CAD is a netlist, which consists of a set of elements (e.g., logic gates, LUTs, FFs), a set of element pins (i.e., inputs and outputs
of elements), and the set of *nets* that connect the elements together via their pins. The connectivity within a netlist is represented as a directed graph in which the pins are nodes and the connections between pins are edges. A net is defined as a subset of edges that have the same source node (pin). The pins are also often called *terminals* [61]. Figure 2.1 shows a simple netlist comprised of 2-LUTs, input ports, a flip-flop, and an output port. There are several types of netlists which are described in Section 2.1.5.

The first step in the CAD flow is *RTL elaboration*. It takes HDL as input and creates a *technology-independent netlist* that consists of generic elements such as input and output ports, primitive logic gates (e.g., 2-input AND), flip-flops, complex functions (e.g., arithmetic operations), and memory structures [60]. *Logic synthesis*, which is the second step, is able to optimize the primitive logic gates within the netlist with the goal of reducing the circuit area and/or reducing the critical path delay [62, 63].

In the third step, *technology mapping*, the technology-independent netlist of primitive logic gates is transformed into a netlist comprised of the logic resources of the target FPGA architecture (*technology-mapped netlist*), such as LUTs, DSPs, and memory blocks. One of the mapping goals is to minimize LUT usage, i.e., to *pack* as many primitive gates into each LUT as possible [51, 64].
In the fourth step, clustering, BLEs are grouped into logic clusters (i.e., LABs or CLBs). Then, in the fifth step, placement, a specific physical location is determined within an FPGA for each element in the mapped netlist (i.e., LABs, DSPs, and memory blocks). Routing, which is the sixth step, establishes signal paths between the placed elements using wire segments, connection blocks, and switch blocks of the programmable routing. In other words, each net is realized as a set of wires that connect source and sink pins.

In the seventh step, timing analysis is performed to determine the timing characteristics of the resulting circuit, such as the maximum clock frequency \( f_{\text{MAX}} \). Bitstream generation is the final step of the CAD flow. In this step, the placed-and-routed circuit is taken as input and used to generate the bitstream that configures the SRAM bits of mapped programmable FPGA resources, including logic, routing fabric, DSP, and memory blocks.

Often the term synthesis is used to refer to the first three steps: RTL elaboration, logic synthesis, and technology mapping. The next three steps of clustering, placement, and routing are collectively referred to as either placement-and-routing, fitting, or implementation. In older literature, the term, “synthesis”, also refers to the entire HDL-to-bitstream CAD flow; however, the contemporary terms for the entire CAD flow are compilation (used by Altera) and implementation (used by Xilinx). We use “compilation” to refer to the entire flow.

### 2.1.3 Altera Stratix IV Architecture

The BLE in Stratix IV is referred to as an adaptive logic module (ALM) [57]. The Stratix IV architecture is an island-style FPGA and each logic cluster (i.e., LAB) contains 10 ALMs. A single ALM contains 8-inputs and can be configured to implement a number of 7-input logic functions (i.e., a subset of 7-LUT functionality), all 6-input logic functions (full 6-LUT functionality), or two independent functions consisting of smaller LUT sizes...
Figure 2.2: A region of a Stratix IV device showing its column-based layout. This region shows one DSP column, one M144K column, two M9K columns, and several LAB/MLAB columns.

(e.g., two independent full 4-LUTs) [54, 65]. The ALM also contains two flip-flops.

In addition, the ALM comprises two dedicated one-bit adder blocks to improve the performance of arithmetic operations. The carry-in of the first adder is driven by the carry-out of the preceding ALM, thereby establishing a carry chain. This enables the fast propagation of carry signals in multi-bit arithmetic operations.

There are three different memory block types in the Stratix IV FPGA: 640-bit memory logic array blocks (MLABs), 9-Kbit blocks (M9Ks), and 144-Kbit blocks (M144Ks). Every other LAB in a Stratix IV device can be configured as either an MLAB or as a regular logic LAB. An MLAB can be configured either as a 64×10 (depth×width) or a 32×20 simple dual-port SRAM block. M9K and M144K blocks also support a range of depth×width configurations [54, 65].

A DSP block can implement several multi-bit arithmetic operations including multiplication, multiply-add, and multiply-accumulate (MAC). Examples of available configurations of a single DSP block include four 18×18-bit integer multipliers and two 36×36-bit integer multipliers. However, there are many other configurations available aside from these two examples [65].

The blocks in a Stratix IV device (i.e., LAB, DSP, and memory blocks) are organized
in a column-based layout, with blocks of the same type being grouped into columns that span the height of the device. Figure 2.2 shows a region of a Stratix IV device. The majority of the columns are LAB columns while DSP and M9K/M144K columns are interspersed throughout the device. The layout of the columns in Altera FPGAs is often non-uniform and asymmetric. This is the case with our target Stratix IV device (EP4SGX530KH40C2), which has a total of 184 columns made up of 166 LAB columns, 4 DSP columns, and 14 columns of M9Ks and M144Ks. The non-uniformity is caused by the DSP columns being spaced unevenly due to a varying number of LAB columns being inserted between them. This non-uniformity is also observable in the layout of memory block columns (M9Ks, M144Ks). The layout of the blocks is left-to-right asymmetric; that is, it is not a mirror image with respect to the center column.

Stratix IV FPGAs feature three types of routing resources [65, 66]: local LAB routing, row routing, and column routing. There are three types of row routing, including direct link routing between adjacent LABs and other blocks, R4 routing which spans four LABs, and R20 routing which spans 20 LABs in the same row. Similarly, there are three types of column routing, including shared arithmetic and carry chains from LAB to LAB, C4 routing which spans four LABs in the same column, and C12 routing which spans 12 LABs in the same column.

2.1.4 Placement and Routing Algorithms

In the first part of this section, we elaborate on the simulated annealing (SA) based [67] placement algorithm in the context of the FPGA CAD. We adapt this algorithm for the placement of DFGs onto our overlay architecture as detailed in Section 5.3. The generic SA-based placer described here is the most widely used type of placer in the FPGA CAD [61], and it is used by Altera’s Quartus II [43] and the academic VPR tool [68].

In the FPGA flow, the role of the placement algorithm is to decide the physical location for each element of a circuit’s netlist; in other words, it determines which FPGA
Figure 2.3: Pseudo-code for a generic high-level simulated annealing-based placer [50, 61].

Simulated annealing mimics the annealing procedure by which strong metal alloys are created [61, 67]. The process consists of first using a high temperature (i.e., allowing circuit elements to move freely) to “melt” the system being optimized (in this case a circuit), then lowering the temperature gradually until the system “freezes” and no further changes occur, thus obtaining a high-quality circuit placement.

Simulated annealing mimics the annealing procedure by which strong metal alloys are created [61, 67]. The process consists of first using a high temperature (i.e., allowing circuit elements to move freely) to “melt” the system being optimized (in this case a circuit), then lowering the temperature gradually until the system “freezes” and no further changes occur, thus obtaining a high-quality circuit placement.

Simulated annealing mimics the annealing procedure by which strong metal alloys are created [61, 67]. The process consists of first using a high temperature (i.e., allowing circuit elements to move freely) to “melt” the system being optimized (in this case a circuit), then lowering the temperature gradually until the system “freezes” and no further changes occur, thus obtaining a high-quality circuit placement.

Figure 2.3 shows a high-level pseudo-code for a generic simulated annealing-based placer. First, the initial placement \( P \) is created by randomly assigning netlist elements to legal location on the FPGA. The placement is then improved gradually by a large
number of placement perturbations, or *moves*. A move consists of picking a random logic block from circuit’s netlist and selecting a random new location within the FPGA for it. If there is another block occupying the new location, then the two blocks are swapped.

The quality of the placement of a netlist’s blocks is evaluated using a cost function. For instance, a wirelength-based placer uses a cost function that estimates the placement’s the total wirelength by summing the half-perimeter of each net’s bounding box [50, 61].

The central part of SA is the decision criterion for accepting the moves. Moves that result in cost reduction are always accepted. However, a move can still be accepted despite resulting in a cost increase. The probability of accepting the move is computed as $\exp(-\Delta C/T)$, where $\Delta C$ is the cost function change cause by the move. *Temperature* ($T$) controls the probability of accepting the move: (a) at higher temperatures the move is more likely to be accepted, even if it results in a large cost increase; (b) at lower temperatures most moves that degrade the placement (increase the cost) are rejected. This characteristic of accepting moves that increase the cost enables the SA to escape local minima in the cost function.

The temperature is decreased according to the *annealing schedule*, which monitors the percentage of accepted moves and adjusts the rate of temperature change accordingly. The other parameters of the annealing schedule are as follows: the inner loop criterion (*InnerLoopCriterion*) controls the number of moves attempted at a given temperature, usually a function of netlist size; the exit criterion (*ExitCriterion*) determines the convergence of the algorithm, i.e., determines when the cost difference has become “minimal” and then terminates the algorithm; and the $R_{\text{limit}}$ parameter controls the range within which a block can be moved. In the beginning large perturbations are allowed, and the block can be moved anywhere within the chip. As the temperature lowers, the range is gradually reduced. As a result, the moves become local, thereby increasing the likelihood of being accepted.

We now describe PathFinder, a negotiated-congestion routing algorithm [70]. We
adapt this algorithm for the DFG routing on the overlay architecture as described in Section 5.4. The key feature of PathFinder is its architecture-independence, which stems from the use of a directed graph representation of the underlying FPGA architecture [61]. This graph is commonly called the resource routing graph. There is a node for each pin of the logic block and each wire of the FPGA. The edges represent potential connections (due to programmable switches) among the pins and wires. An example of a routing resource graph for a Triptych FPGA cell is shown in Figure 2.4.

Each node $n$ in the resource routing graph has a base cost ($b_n$) which typically represents the length of the corresponding wire. The source and sink nodes of a net that has been placed onto the FPGA form a subset of the resource routing graph nodes. The goal of the algorithm is to successfully route all nets by finding paths in the resource routing graph between each net’s source and sink nodes in a manner that minimizes the total cost of used routing resources. This is achieved by employing heuristics which incrementally build a net’s routing tree one sink at the time.

The key idea behind this heuristic is to use Dijkstra’s shortest path search algorithm in the routing resource graph starting from the net’s source until any of the sinks are found [70]. Once a sink has been found, back-tracing is used to construct the path from
the source to the found sink. This path then becomes the initial net routing tree. The
search for the next net sink is done by restarting the Dijkstra search. This time all
the nodes of the current routing tree are used as the sources for the search (initialized
with cost 0). As a result, the algorithm will attempt to reuse the routing hops from the
previously routed sinks to route paths to the remaining sinks.

Within one routing iteration this procedure is performed for each net. In a routing
iteration the algorithm deliberately allows for the overuse (or congestion) of routing
resource graph nodes (e.g., the same wire is used by two different nets). However, this
is not a legal routing and Pathfinder iteratively rips-up and re-routes every net in the
circuit until all congestion has been resolved. Congestion is modeled by assigning $h_n$
(historical congestion) to each node and increasing it after each iteration where node $n$
is overused. This gives the router congestion memory [50]. A node is also assigned a
“present-sharing” cost $p_n$ which is a function of the number of signals that use node $n$
during one routing iteration and the iteration number. The total cost of a node ($c_n$) is
computed as

$$c_n = (b_n + h_n)p_n \quad (2.1)$$

The key idea behind the historical and present congestion models is that by increasing
the cost of a congested node, it will be left to the net that needs it the most, as nets with
alternative paths will eventually find them [61]. In essence, the congestion is eventually
avoided by the “negotiation” for routing resources among nets.

The above is a description of a routability-driven version of the PathFinder. A timing-
driven version is obtained by extending the cost function shown in Equation 2.1 with delay
and the notion of the net’s timing criticality [49, 70].


2.1.5 Hierarchical CAD Techniques for Physical Design

*Physical design* refers to the later stages of the CAD flow, which commonly include clustering, placement, and routing. In this section, we provide a brief overview of the background and terminology of the hierarchical CAD techniques that can be used during the physical design phase of FPGA circuits [71, 72] in the context of the Altera’s Quartus II CAD tool. The main hierarchical techniques are *partitioning* and *floorplanning* which enable designers to divide a design into smaller pieces, implement them separately, and assemble the pieces together [73, 74]. With the exponential growth in FPGA capacity (Figure 1.2) these techniques are becoming increasingly recommended by FPGA vendors [71] in order to manage the complexity of the design size, enable incremental design, reduce design iteration time, and facilitate design reuse.

A *design partition* (or simply a *partition*) is a module in the design hierarchy (e.g., an HDL module) that has been designated by the designer to be synthesized or compiled separately. A partition is created by adding a *partition directive* to the CAD project [71]. A design is usually divided into a number of partitions that are each realized as a separate netlist. Several types of a netlist are used: (a) a *post-synthesis* netlist, which is obtained after technology-mapping; (b) a *post-placement* netlist, which is obtained after placement; and (c) a *post-placement-and-routing*, or *post-fit* netlist, which is obtained after full placement and routing. A partition can be reused across CAD projects\(^\text{1}\) by exporting it from one project and importing into another. Partitions can be *merged* into a single netlist at different stages of the physical design depending on the netlist types. For example, if post-synthesis netlists are used, then partitions are merged into a single post-synthesis netlist following the synthesis of each partition. Afterwards, the merged netlist is placed-and-routed. If post-fit netlists are used, then only the paths between the partitions need to be routed. The CAD tool will attempt to preserve the placement and

---

\(^1\)Quartus II software organizes and manages the elements of a design (such as netlists, hierarchy, constraints and settings) within in a project.
A physical region is a set of an FPGA’s contiguous physical resources, and it is commonly a rectangular section of the FPGA. The granularity of the physical regions in the Stratix IV architecture is at the LAB level. The basic rectangular physical region is $H$ LABs tall and $W$ LABs wide and thus has a total area of $H \times W$ LABs. Non-rectangular and non-contiguous regions can be created by merging several of these rectangular regions [75]. The placement of an HDL module (i.e., all of its netlist elements, such as LUTs, FFs, and DSPs) can be constrained to the resources within a physical region. This is achieved by adding a placement directive to the CAD project [61, 75]. Quartus II uses the term, LogicLock region, to refer to a physical region.

A floorplan is the division of an FPGA’s resources into a set of physical regions. Each physical region is assigned a module (or a partition) from the circuit to be implemented on the FPGA. A floorplan is designed by creating a set of physical regions and then using placement directives to assign the modules to them. Floorplanning is often an iterative process. First, the initial floorplan is created and the circuit is compiled. If the resources required by a given module exceed the size of the corresponding physical region, then the region size must be increased, which may require modifications to the adjacent regions (e.g., decreasing size, modifying shape, or merging) [71, 75]. The circuit is then re-compiled and the process is repeated until all of the modules fit within their corresponding regions.

A CAD flow, or a compilation flow, is a series of steps that begins with a circuit description in HDL and ends with an FPGA bitstream. However, one flow can differ from another because steps may be implemented differently, have different constraints, or be optimized for a different goal.

A flat flow is a compilation flow which does not have partitions or a floorplan and features a single Quartus II compile. It requires minimal effort from the designer and is the default flow of commercial FPGA CAD tools [71].
Partitioning enables the design of top-down and bottom-up flows. In a top-down flow, a design is divided into partitions that are placed-and-routed together in a single CAD project. A top-down flow may or may not use a floorplan as the selection of partitions and the design of the floorplan are up to the designer.

In a bottom-up flow, a design is divided into partitions and each partition is placed-and-routed independently in a separate project. The compiled partitions are then integrated into the whole design within a single top-level project. A bottom-up flow requires a floorplan. The designer chooses partitions, designs the floorplan, and manages all the projects.

In an incremental compile, only a subset of partitions is synthesized (or placed-and-routed) at a given time. The remaining partitions preserve their post-fit or post-synthesis netlists from a previous compilation (or are imported). Altera also uses the term “incremental compilation flow” to collectively refer to top-down, bottom-up, and incremental compiles (as defined above).

Commercial FPGA CAD tools offer limited automation of partitioning and floorplanning, and their application by FPGA designers is iterative and requires manual tuning [71, 72, 74, 76]. FPGA designers apply these techniques to preserve an unmodified module of a large design, reuse pre-synthesized or pre-placed-and-routed modules, and stitch together a large design from smaller modules in a team-based design fashion.

2.2 Data Flow Graphs (DFGs)

Our definition of a data-flow graph (DFG) is that of a directed acyclic graph that represents data dependencies among a group of compute operations. There are three types of nodes in a DFG: compute nodes, input nodes, and output nodes. A compute node represents a compute operation such as add, multiply, or square root. External data is fed into the DFG by the input nodes, and the data computed by the DFG is output by
A directed edge in the graph represents a flow of data (i.e., a data dependency) between a pair of nodes. An input node has no incoming edges, and an output node has no outgoing edges. A compute node has one or two incoming edges and at least one outgoing edge. There is no state associated with a compute node; a compute node performs its operation on the data that is fed to it by its incoming edges. Commonly, the number of incoming edges in a node is called \textit{in-degree}, or \textit{fan-in}, and the number of outgoing edges is called \textit{out-degree}, or \textit{fan-out}.

An example DFG is shown in Figure 2.5. In this example, input nodes $I_3$ and $I_4$ flow data to compute node $B$, which represents a multiply operation. The node $E$, which represents a divide operation, feeds the output node $O_1$. Node $D$ has a fan-out of two and feeds data to $E$ and $O_2$.

2.2.1 Pipelined execution of DFGs

A DFG commonly represents a body of a parallel loop or a body of a streaming kernel [77]. Each instance of such a DFG operates on independent data. Hence, multiple instances of the DFG can be executed in parallel as shown in Figure 2.6. We map the DFG to the overlay and then execute multiple DFG instances in pipelined fashion by feeding independent data into the DFG each cycle through its input nodes. After the
Figure 2.6: Multiple instances of a DFG from Figure 2.5 operating on independent data.

Figure 2.7: Pipelined execution of independent data on a DFG from Figure 2.5.
initial DFG latency, the DFG outputs data every cycle. Thus, performance is improved by maximizing the throughput at which instances of the DFG complete execution. An example of pipelined DFG execution is shown in Figure 2.7. In this thesis, the overlay architecture and DFG-to-overlay mapping tool are optimized to maximize the throughput of pipelined DFG execution, possibly at the expense of the latency of a single DFG execution.

DFGs are a common abstraction for expressing parallelism in several application domains, such as streaming [44] and digital signal processing (DSP) applications [45]. DFGs have been directly implemented in ASICs for pipelined DFG execution [78–81], and they have also been used to represent loops in high-level synthesis [77, 82]. Moreover, they are essential in recent commercial FPGA tools, such as Maxeler’s MaxCompiler, which takes a DFG description and synthesizes a pipelined DFG directly to the FPGA fabric [20].

DFGs that are suitable for pipelined execution can also be extracted from modern parallel programming languages. For example, OpenCL executes multiple instances of a kernel independently with no sharing of data. Thus, a DFG that is extracted from the body of a kernel can be executed in a pipelined fashion. Similarly, DFGs can be extracted from the bodies of parallel loops in an OpenMP program and executed in a pipelined fashion.

2.3 Overlay Architectures

We define an overlay as a pre-compiled FPGA circuit that realizes a programmable architecture, referred to as an overlay architecture. This architecture projects a different programming model than the underlying FPGA fabric. Since the overlay circuit is pre-compiled, only its bitstream needs to be downloaded to the FPGA. Thus, the programmer only needs to compile the application to the overlay (using an overlay tool flow) and then
load, or configure, the application into it. The programmer does not need to invoke FPGA
CAD tools, which are completely removed from the overlay tool flow. FPGA CAD tools
are only used to design and compile the overlay to the FPGA fabric, which can be done
by a hardware designer.

A canonical example of an overlay architecture is a soft processor that projects a
programming model defined by its instruction set architecture (ISA). This is an example
of a software-programmable overlay architecture that projects a programming model that
is higher than that of an FPGA. It hides all the hardware intricacies of the underlying
FPGA, such as LUTs, block memories, placement, routing, and timing analysis.

There are several broad classes of overlay architectures that project a high-level pro-
gramming model and that consist of building blocks that are coarser (e.g., they contain
32-bit operations or instructions) than the FPGA’s fine-grained LUT-level architecture.
Another potential advantage of the overlay’s coarseness is that the number of bits re-
quired to reconfigure the overlay is smaller then the number of bits required to reconfigure
an FPGA. As a result, the overlay can enable shorter reconfiguration times. Examples of
these overlay architectures include multi-processors and many-cores [83–85], mesh-of-soft-
processors [86], vector processors [87, 88], GPU-inspired processors [89, 90], VLIW-based
processors [91], networks-on-chip [92, 93], and mesh-of-functional-units (mesh-of-FUs)
overlays [39, 94–96].

In addition to all of the FPGA hardware details being hidden, the process of compiling
an application to a high-level overlay architecture often takes under a minute [39, 87, 88,
94], which is orders of magnitude faster than compiling an application directly to FPGA
fabric. This in turn radically reduces development time and allows fast develop-test-
debug iterations, thus bringing FPGA development closer to that of software-programmed
devices such as CPUs, DSPs, and GPUs.

Overlay architectures also have the potential to enable the binary portability of ap-
lications by implementing the same overlay on different FPGA devices that might differ
in terms of size, generation, features, or vendor.

Recently, several commercial overlay architectures have become available. Vector-Blox [97] offers a vector soft processor that is customized for matrix-based applications. Convey Computer [98] calls their overlays *personalities*. They act as x86 coprocessors, and they execute a set of custom instructions that are invoked by the CPU. Convey provides several general-purpose personalities as well as a personality development kit (PDK) that allows users to develop their own personalities. Mitrionics offers a Mitrion Virtual Processor [99], which is fine-grained massively parallel configurable soft-core processor. Their overlay is programmed using an implicitly parallel Mitrion-C programming language.

There are also overlay architectures with different goals than those of the high-level architectures mentioned above. One such example of this type of overlay architecture is an FPGA-like overlay that realizes “FPGA-on-an-FPGA” [100]. This approach aims to achieve FPGA bitstream compatibility among different vendors and parts as well as compatibility with open FPGA tool flows such as VPR [52].

### 2.3.1 Mesh of Functional Units (Mesh-of-FUs) Overlay Architectures

Mesh-of-FUs overlay architecture [39, 94–96] consists of an array of interconnected overlay cells that are each comprised of an FU and a programmable routing logic. Mesh-of-FUs promise high performance because they naturally expose the massively parallel FPGA fabric. Furthermore, they can be heterogeneous; this means that they can contain different types of FUs, which allows overlays to be customized for different application domains through the choice and arrangement of FUs in the overlay’s mesh.

Mesh-of-FUs are designed for pipelined DFG execution [39, 94–96]. The nodes of a DFG are mapped to FUs and edges between nodes are mapped to paths among the FUs. The paths are realized by configuring the overlay’s routing logic. Both the FUs and the
routing are pipelined, which allows for new DFG input data to be fed into the overlay every cycle, thus realizing pipelined execution.

Several types of interconnect topologies have been proposed for mesh-of-FUs overlays. Nearest neighbor interconnect [95, 96] consists of connections between neighbouring FUs. In addition, distant FUs can be connected using pass-through connections, thus effectively bypassing intermediate FUs. Some overlay interconnects mimic island-style FPGA interconnect [39]. These overlay interconnects feature connection boxes to connect FUs to virtual routing tracks and planar switch boxes that make connections between tracks. In addition, application-specific topology can be derived from the target application [94]. In this process, the topology of an application’s DFG is analyzed and then connections between FUs are created that closely match the DFG edges between corresponding operations, which consequently produces a DFG-specific interconnect topology. In all of the above interconnects, the overlay’s programmable interconnect is mux-based. The muxes are implemented using FPGA logic blocks, hence the interconnect requires the use of both FPGA logic and routing resources.

The execution of FUs can be scheduled statically or dynamically. A static schedule is generated offline and each FU is assigned a sequence of control signals that trigger FU execution in specific pre-determined clock cycles [94]. In dynamic scheduling, FU execution is triggered at run-time: an FU executes when all of its operands are ready and its output can receive the results [40]. Static scheduling requires that latencies of I/O interfaces be deterministic (or that they never stall) and that latencies along pipeline paths can always be balanced. In contrast, dynamic scheduling can deal with both non-deterministic I/O latencies (due to cache misses or memory contention) and also imbalanced pipeline paths by stalling the FU whose data is not available. However, dynamic scheduling requires more hardware support to implement the stalling mechanisms.

The mesh-of-FUs pipeline can be centrally controlled using a global enable signal [39, 95] or control can be distributed so that each FUs has its own control logic [40, 94, 96].
Although central control potentially requires fewer resources than distributed control, it suffers from several drawbacks: first, it may limit the scalability to large meshes; second, any I/O stall will cause a global stall of the entire mesh; and third, since there is no individual control of the FUs, correct execution can only be achieved if pipelines are balanced.

### 2.4 Elastic Pipelines

Synchronous elastic pipelines are built with elastic buffers (EBs) [47, 48, 101, 102] that replace regular (i.e., inelastic) registers between pipeline stages.

The basic EB is a 2-slot FIFO which can be implemented using transparent latches or flip-flops [47]. These 2-slot EBs are also referred to as relay stations [103, 104] or Ambric registers [105, 106] in the literature. The FPGA fabric dictates the use of a flip-flop-based implementation because flip-flops are the elementary storage block in FPGAs [102].

Two EBs synchronize the transfer (or flow) of data \( D \) via a pair of control signals, as shown in Figure 2.8. A forward-feeding data-presence valid signal \( V \) that indicates whether the EB has data or is empty (i.e., a pipeline bubble), and a back-propagating stall/accept \( A \) signal that indicates whether the EB is stalled (full) or if it can receive new data. In a given clock cycle, the sender EB will transfer data to the receiver EB if the sender has data at its output (valid is 1) and if the receiver can receive data (accept is 1). If the accept is 0, the sender stalls and data transfer is retried next cycle. This protocol is called synchronous elastic protocol or flow (SELF) [101].

When there are no stalls, a 2-slot EB behaves like a regular register, i.e., its occupancy is one data (the half-full state), its latency is one clock cycle, and data is forwarded through the EB every cycle (maximum throughput). When there is a stall, the EB can hold up to two pieces of data (the full state—occupancy of two) while waiting for the stall to end. In order to return to the maximum throughput state, the EB first returns
Figure 2.8: (a) EBs in a linear pipeline, (b) an example 2-slot EB.

to the half-full state by sending the first piece of data. Then, in the next cycle, the EB will both receive and send data, thus resuming maximum throughput operation.

The implementation of one such 2-slot EB is shown in Figure 2.8 [102, 107]. The main data register \((main)\) is used for the pipelining of data similar to a regular inelastic register. In addition, there is an auxiliary register \((aux)\) that also stores data that has been received from the previous EB. If a stall cycle occurs \((A_N=0)\), the data in the main register will not be sent (i.e., stored by the next EB) and will need to be resent in the next cycle. Since the stall will be delayed by one cycle before being back-propagated to the previous EB, the auxiliary register will store data sent by the previous EB. Without the auxiliary register, this data would either have to be dropped or it would overwrite data in the main register. In the next cycle the stall is propagated \((A_P=0)\) to the previous EB. No data in this cycle is received from the previous EB; that is, the EB is full and both of its registers hold valid data.
Several other implementations of 2-slot EBs are also possible, and all of them require two data registers (main and aux) and a mux which selects data from either of these registers [101, 102].

The above is intended to serve as a description of EBs with capacity of two data items, however general EBs are deep elastic FIFOs of arbitrary finite capacity (N slots). We present a detailed design of several variations of 2-slot EBs and also deeper N-slot EBs in Section 4.1.3.

One important characteristic of elastic pipelines is distributed pipeline control. The synchronization logic is local to EBs because all signals (data, valid, and accept signals) are registered in each EB and propagate one EB per cycle. No global control signals exist in the pipeline other than the global clock signal. In contrast, inelastic pipeline uses a global signal to stall the pipeline. Figure 2.9 shows a linear pipeline of inelastic registers. While inelastic pipeline does permit bubbles by pipelining the valid signal along with the data signal, all the registers are controlled with a single global stall signal. This will result in a large fan-out and long propagation delays for large pipelines, thus limiting the $f_{\text{MAX}}$ [102]. The $f_{\text{MAX}}$ impact is more pronounced on FPGAs due to larger interconnect delays (a combination of longer wires and programmable routing switches) [108, 109].

Another property of elastic pipelines is latency insensitivity [102, 103]. Elastic pipelines are able to tolerate the latency variability of computation and communication, which enables them to operate correctly independent of the latencies of the various pipeline units.
This property allows the insertion of EBs on pipeline paths without the need to modify the rest of the design. Inserting EBs on paths with long propagation delays can improve \( f_{\text{MAX}} \) \[103\] at the expense of increased latency.

While elastic pipelines are latency insensitive in terms of functionality, they are sensitive to latency in terms of throughput. Much like an inelastic pipeline, elastic pipelines have to be *latency balanced* in order to achieve high throughput. We describe latency balancing and techniques for achieving it in Section 2.5.

The downside of EBs is the cost of their implementation in the FPGA fabric. As mentioned above, the 2-slot EB requires an extra register and a mux in comparison to an inelastic register. Thus, *elasticizing* \[47\] an inelastic pipeline by replacing every register with a 2-slot EB can double the circuit’s register resource usage. We approach this challenge by designing several 2-slot EB variants that are tuned for FPGA fabric, and by using both inelastic registers and EBs. We further elaborate on this in Chapter 4 and discuss the overarching challenge of trading-off between resource usage, \( f_{\text{MAX}} \), and the throughput of the overlay architecture.

### 2.5 Latency Balancing of Pipelines

In a general pipeline, the units (i.e., stages) can receive data from multiple units (a join), and a unit can send data to multiple stages (a fork). This type of pipeline has multiple paths and can be represented as a directed graph. To ensure that a pipeline operates at the maximum throughput—i.e., all pipeline units move data every cycle—it is necessary that each join in the pipeline receives all its input data at the same time. In other words, the latencies of all distinct paths from the primary inputs of the pipeline to a join unit should be equal, or *balanced*) \[79, 110\]. It is also assumed that all primary inputs are fed into the pipeline at the same time, and, similarly, it is required that data arrives at the primary outputs at the same time. Hence, the above balancing requirement is generalized
Figure 2.10: (a) Pipeline for the DFG from Figure 2.5 before balancing, (b) transformed pipeline by insertion of balancing registers (shaded). The registers reflect the latencies of the FUs and of their connections.

to require that all paths through the pipeline (from primary inputs to primary outputs) have the same latency.

In the case of an elastic pipeline, two imbalanced paths joining at a unit will cause EBs on the shorter paths to fill up sooner. This in turn leads to stalls and reduces throughput. In contrast, in an inelastic pipeline, imbalanced paths would lead to incorrect execution, i.e., latency balance is a correctness requirement [77]. This is due to the fact that individual inelastic registers cannot be stalled on their own because the data in the entire pipeline moves and stalls in lockstep.

Figure 2.10a shows an inelastic pipeline for the DFG in Figure 2.5. In the imbalanced implementation there are no registers on the edges; thus, the latencies of paths stem from the latencies of the FUs. Since paths contain different FUs, and they themselves have different latencies, the DFG is imbalanced. For example, the paths from I_1 and I_2 to the \textit{div} FU have a latency of 2 clock cycles while the paths from I_3, I_4, and I_5 have a latency of 5 clock cycles. Additionally, the latency from input node I_3 to O_1 is 2 cycles longer than the latency between it and O_2.

Figure 2.10b shows the transformed pipeline that was balanced by inserting three
registers on the edge between the \textit{mul} and the \textit{div} FUs, and two registers on the edge inbound to \(O_2\). In an inelastic pipeline, a set of \(N\) registers inserted on a DFG edge can also be implemented as a single shift register of depth \(N\). These shift registers are also referred to as \textit{delay buffers} in the literature [79–81].

Often the goal in designing a balanced DFG pipeline is to minimize the resource cost of balancing, which can be done by determining a set of delay buffers and their insertion points such that the total length of inserted buffers is minimized. This task is often called \textit{buffer assignment} or \textit{buffer minimization} [79–81]. Minimizing delay buffers in balancing pipelined DFGs in the context of direct implementation in ASIC fabric has been addressed by several authors [79–81, 111]. Most commonly, however, this problem is solved by formulating an integer linear program (ILP).

We provide an overview of an ILP formulation that is based on the formulation proposed by Hu et al. [79] as well as similar formulations proposed in [80, 111]. Two special nodes are introduced into the DFG: the source node (\(S\)), and the sink node (\(T\)).
By definition, these nodes have a latency of 0. Edges are added between the source node and each input node and also between each output node and the sink node. Those edges also by definition have latencies of 0.

The balancing requirement can then be stated as follows: the accumulated latencies along all the distinct paths from the source node \( S \) to a particular DFG node must be equal. Figure 2.11 shows a DFG which is annotated with variables for the ILP formulation. Two sets of known variables denote the node and edge latencies of an imbalanced DFG: \( T_u \) denotes the latency of a DFG node \( u \) (commonly an FU latency) and \( w_{uv} \) is the latency of the edge between nodes \( u \) and \( v \). There is also a set of unknown edge variables \( b \) which are to be solved for in the ILP model: \( b_{uv} \) denotes the lengths (i.e., latency) of a delay buffer to be inserted on the edge between nodes \( u \) and \( v \). Given that the set of \( b \) variables balances the DFG, all paths from \( S \) to a node \( u \) should then have equal latency, which we denote as an unknown variable \( D_u \). For example, the balancing requirement for node \( A \) states that two paths from \( S \) to \( A \) must have equal latency \( D_A \):

\[
D_{I_1} + T_{I_1} + w_{I_1A} + b_{I_1A} = D_A \quad (2.2a)
\]
\[
D_{I_2} + T_{I_2} + w_{I_2A} + b_{I_2A} = D_A \quad (2.2b)
\]

The set of \( D_u \) variables allows for a concise formulation. In case of node \( E \) we only have to write two edge equations (2.4) each corresponding to one of its incoming edges. This is instead of writing six equations for all the six paths from \( S \) to \( E \).

\[
D_A + T_A + w_{AE} + b_{AE} = D_E \quad (2.3a)
\]
\[
D_D + T_D + w_{DE} + b_{DE} = D_E \quad (2.3b)
\]

In general, for each node \( u \) with \( i_n \) incoming edges, we write \( i_n \) edge equations for a total of \(|E|\) equations for the entire DFG.

The ILP formulation seeks to minimize the cost of delay buffers by minimizing the
sum of all $b_{uv}$ that are subject to the set of edge equations:

$$\text{minimize: } \sum_{(u,v) \in E} b_{uv} \quad (2.4a)$$

subject to: $D_u + T_u + w_{uv} + b_{uv} = D_v \quad (2.4b)$

where $u, v \in V$, $(u, v) \in E$, integer $D_u, b_{u,v} \geq 0 \quad (2.4c)$

Furthermore, since all inputs are fed to the pipeline at the same time, for each input node $u$ we set $D_u = 0$ (in the above example $D_{I_{1..5}} = 0$). Hence, this formulation has $|E|$ equations and $|E| + |V| - |I|$ unknown variables, where $I$ denotes the set of input nodes.

A general ILP formulation does not have a polynomial-time solution [79]. Several alternative approaches for solving buffer minimization have been proposed in an effort to reduce the algorithm’s run-time, including DFG decomposition [81] and buffer redistribution [111]. Boros et al. [80] show that the buffer minimization problem is polynomially solvable by reformulating it as a minimum-cost network flow problem with an $O(mn \log n)$ algorithm.

The above formulations minimize the total buffer length of direct DFG implementation in ASIC fabric under the following assumptions: first, the length of individual buffers is not limited; second, the total allowed buffer length (or total cost) is not limited; and third, the computed buffer assignment solution is always assumed to have a feasible hardware implementation. In other words, it is assumed that every DFG can be balanced in hardware regardless of the number/length/cost of buffers used, DFG topology, or buffer insertion points. Furthermore, the possible impact of a specific buffer assignment on $f_{\text{MAX}}$ or DFG routability in ASIC is not considered.

In contrast, our problem is different in several aspects: (1) in a pre-compiled overlay the capacities of individual FIFOs are fixed; (2) the total number of available FIFOs in a pre-compiled overlay is fixed; (3) finding a balanced DFG mapping on the overlay is complicated by the tasks of placement and routing of the DFG; and (4) a balanced DFG
mapping (optimal solution) may not exist, in which case an approximate (most balanced) mapping should be computed. Another difference is that we use elastic FIFOs, and these can have variable latency that is determined by their occupancy at run-time. This is in contrast to the fixed-latency delay buffers discussed above. Furthermore, there is a design trade-off between the capacity and number of FIFOs within the overlay (easier to balance DFGs) and the overlay’s $f_{\text{MAX}}$ and resource usage. Our solution to DFG-to-overlay mapping and a detailed discussion is presented in Chapter 5.
Chapter 3

Overlay Architecture Overview

In this thesis, we design a mesh-of-FUs overlay architecture for the pipelined execution of data flow graphs. The building blocks of the overlay are overlay cells, which are arranged in a rectangular mesh and connected by a four nearest neighbour (4-NN) topology, as shown in Figure 3.1. The overlay’s mesh size indicates the total number of overlay cells and is expressed as H×W, where H is the number of rows in the mesh, and W is the number of columns in the mesh.

Each overlay cell contains a functional unit (FU), reconfigurable routing logic, and reconfigurable synchronization logic. FUs and routing have word-wide data-paths (e.g., 32 bits). The overlays that we design in this work use a set of 32-bit integer FUs and/or a set of single-precision floating-point FUs.

An FU implements single or multiple machine operations such as multiply, add, square root, or divide. The functionality of a single-operation FU is fixed at run-time, which means that it always executes the same operation. An example is the abs FU that computes the absolute value of a number that is provided at its input. A multi-operation FU executes only one of its operations at run-time, and this operation is selected by configuring the FU via a special set of configuration bits (FU\textsubscript{op}). An example of a multi-operation FU is the addsub FU which implements add and subtract operations.
Furthermore, an overlay cell can also have multiple FUs, but only one of them can be active at run-time. In this case, additional configuration bits (FU\textsubscript{sel}) are used to select the active FU, and FU\textsubscript{op} is then used to select the active operation within the active FU.

Different cells in the overlay can have different types of FUs, and it is this feature that makes the overlay heterogeneous. A heterogeneous overlay can be characterized by its functional layout, which is defined by the arrangement of FU types in the overlay’s mesh. The functional layout of an example 4×4 overlay is shown in Figure 3.1. This example overlay uses eight different types of floating-point FUs. The FU types that we use are described in detail in Section 4.1.1.

Each cell has four input ports and four output ports that send and receive data to and from the four neighbouring cells. The cell’s routing and synchronization logic can be reconfigured at run-time to: (1) create connections between the cell’s input and output ports and the FU’s input and output ports; and (2) connect the cell’s input ports to the cell’s output ports, thus bypassing the FU. This allows for non-neighbouring cells to
be connected by a series of routing hops, i.e., by passing through intermediate cells, and bypassing their FUs. Much like reconfigurable FUs, the cell’s routing and synchronization logic are reconfigured at run-time using a set of configuration bits. Data flows into and out of the overlay through the input and output ports of the overlay boundary cells, which are termed overlay inputs and overlay outputs, and are shown in Figure 3.1.

A DFG is mapped to the overlay such that each DFG node is mapped to a distinct FU in the overlay, and each DFG edge is formed by configuring a routing path between the FUs that correspond to the source and sink nodes of that edge. The overlay is configured with a specific DFG by loading the overlay bitstream, which contains configuration bits for the FUs, as well as the routing logic and synchronization logic of each call. The overlay configuration is fixed during the execution of a DFG, which means that the selected operations executed by the FUs and the mapped routing paths are fixed. In order to execute a different DFG, the overlay must first be stopped and then reconfigured by loading a different overlay bitstream.

Figure 3.2 shows, at a high level, how the DFG from Figure 2.5 is mapped to a 3×3 overlay. Specifically, it highlights the mapped FUs that execute the operations of the DFG nodes and the routing paths that connect the mapped FUs. For example, the FUs
that execute the nodes $A$ and $E$ are connected together by bypassing the divider FU in the second row. The routing connections can fan-out to multiple output ports of a cell, which allows an FU to feed multiple other FUs; for example, the output of the $D$ multiplier fans-out to the input of the $E$ divider and to the overlay output $O_2$.

### 3.1 Design Challenges

In this work we address the design of a mesh-of-FUs architecture that contains 100’s of FUs, which enables it to exploit fine-grain pipelined parallelism on a large scale. In order to be able to map and route DFGs with various topologies, the architecture necessitates the use of non-linear pipelines in which pipeline units can fan-out (i.e., feed) to multiple other units and also each FU can join data from multiple pipeline units. Further, we focus on heterogeneous mesh-of-FUs because of their ability to be customized for a specific application or application domain, which can be made possible by designing overlays with custom functional layouts. However, there are several sets of challenges that arise in the design of such an architecture.

**Scheduling and pipeline control:** The first challenge is to determine the mechanism by which the execution of FUs is scheduled. It can be static or dynamic, and it can use central or distributed pipeline control. This scheduling and control mechanism affects all aspects of the overlay architecture. For example, static scheduling may require fewer logic resources and be able to achieve higher $f_{\text{MAX}}$ than dynamic scheduling, but it is also less flexible: it requires the latencies of the I/O interfaces to be deterministic (or they never stall), and it also requires that the latencies along pipeline paths can always be balanced. Dynamic scheduling has the potential to achieve higher throughput in those scenarios because it is more flexible: it can adapt to non-deterministic I/O latencies and also deal with imbalanced pipeline paths by stalling FUs whose data is not available.

A simple centralized control of the overlay’s pipelines could be implemented using a
Chapter 3. Overlay Architecture Overview

global enable signal. However, this approach suffers from several drawbacks: 1) it may limit the overlay's scalability to larger meshes; 2) any stall of the overlays I/O ports will cause a global stall of the entire mesh; and 3) without the ability to stall FUs separately, correct execution can only be achieved if the pipelines are balanced. On the other hand, distributed pipeline control without global signals would allow for better scalability and higher $f_{\text{MAX}}$ than central control. However, this may come at a higher resource cost.

**Latency balancing for maximizing throughput:** The second challenge is achieving maximum throughput of the pipelined execution of a DFG on an overlay, which means one DFG instance completes execution every cycle. Such execution can be achieved if each mapped FU executes every cycle without stalls, and this is ensured only if the mapped DFG is latency balanced, i.e., if for each mapped FU, all of its inputs arrive at the same time. In other words, the latencies of all distinct mapped paths from overlay input ports to that FU are equal.

However, the paths that converge at an FU may have different latencies for two reasons. First, the DFG itself can be imbalanced because there may be different number of compute nodes on different DFG paths, and also because the latencies of various FU types are different. Second, even if the DFG itself is balanced it can become imbalanced because the place-and-route algorithm may route DFG paths using different number of hops despite these DFG paths having the same number of compute nodes with the same FU latencies.

For example, in Figure 3.2 the mapped paths originating at input $I_4$ and joining at FU $D$ have a difference of four pipeline units. This results in an imbalanced DFG mapping despite the fact that the DFG is itself balanced.

**High $f_{\text{MAX}}$ and efficient use of FPGA resources:** The third challenge is achieving high $f_{\text{MAX}}$ while using the least amount of FPGA resources. Given the scheduling and pipeline control mechanisms what should be the target $f_{\text{MAX}}$ of the overlay? The target $f_{\text{MAX}}$ dictates the choice of pipeline depths of FUs and also the degree of pipelining.
Maintaining high $f_{\text{MAX}}$ for large overlay meshes: The fourth challenge is how to scale the architecture to more than 100 FUs. An overlay architecture with deeply pipelined FUs and routing paths is likely to achieve high $f_{\text{MAX}}$ for small meshes, but it is an open question as to how a high $f_{\text{MAX}}$ can be maintained as the overlay mesh increases in size. If high $f_{\text{MAX}}$ is not maintained then the resources invested for the deep pipelining of the FUs and routing are “wasted”. This results in diminishing returns in terms of performance per FPGA resources invested as mesh size increases.

Design of overlay’s functional layout for non-uniform heterogeneous FPGA layout: The fifth challenge relates to the design of overlay’s functional layout. This challenge stems from the asymmetric and non-uniform layout of heterogeneous FPGA resources. Furthermore, FPGA devices usually have far fewer DSP and block RAM columns than logic columns, hence, in order to achieve high $f_{\text{MAX}}$, overlay cells that have DSP-based FUs likely have to be placed in the regions close to the DSP columns. As a result, the design of the overlay’s functional layout is constrained by the layout of the FPGA’s columns.

Routability of large DFGs: Finally, the 4-NN topology may provide enough routing flexibility to route small DFGs with only a dozen nodes on small meshes. However, routability can become more challenging as the DFG size and mesh size are scaled up.

3.2 Summary of Approach

Dynamic data-driven scheduling and elastic pipelines: We address the first challenge by utilizing *data-driven execution*, which is a process wherein the execution of an FU is triggered *dynamically* by the availability of input data. For example, in Figure 3.2, the multiplier $D$ stalls if one or both of its inputs (i.e., outputs of $B$ and $C$) do not have data available. Similarly, the multiplier stalls if its output is not ready to accept data. We
implement the data-driven execution using synchronous elastic pipelines based on elastic buffers (EBs) and elastic protocol which uses two synchronization signals: data-presence (valid) and back-pressure or stall (accept). We design flip-flop-based 2-slot EBs and also deeper EBs that are implemented as general N-slot FIFOs.

The primary role of EBs within the overlay pipelines is to localize synchronization. Consequently, the overlay architecture features a distributed pipeline control: the transfer of data between pipeline units is synchronized locally and the synchronization signals propagate only as far as the neighbouring cell in each cycle. This, coupled with the cell-based design of the overlay, gives rise to a modular architecture: each overlay cell is self-contained and interfaces with neighbouring cells with four input elastic channels and four output channels that correspond to the cell’s input and output ports. Hence, no global or centralized control signals exist in the overlay other than the clock signal. As a result, this architecture has great potential for being able to scale to the size of modern FPGAs.

**Elastic FIFOs for latency balancing**: The role of the deeper N-slot FIFOs is to serve as the latency balancing mechanism of the overlay. In essence, they act as delay buffers that increase the latency of shorter paths (as described in Section 2.5).

There are three key differences between elastic FIFOs and inelastic delay buffers. First, the latency of an N-slot elastic FIFO is variable and is dependent on the FIFO’s occupancy (the number of data items the FIFO contains). Thus, we say that it has a latency margin. For example, if the overlay architecture has 32-slot FIFOs, then they can serve as delay buffers of any latency from 2 to 31 clock cycles (a detailed design of EBs is described in Section 4.1.3). Second, an elastic FIFO dynamically reaches a steady-state latency at run-time. During a period of stall cycles the FIFO fills up to a certain occupancy which then remains constant after the FIFO is unstalled. In other words, these FIFOs “stretch” the latency of the shorter paths at run-time during stalls to equalize it with the latency of longer paths. As a result, elastic FIFOs do not require additional
configuration logic to set each one to a required latency at the time of configuration. In comparison, an inelastic overlay containing statically-configured shift registers— which act as delay buffers— requires configuration logic and paths to each shift register in order to configure them to the required balancing latency. Third, unlike inelastic delay buffers, elastic FIFOs do not have to balance the pipeline to achieve correct operation. If an elastic pipeline is imbalanced, the elastic FIFOs will stall and the pipeline will operate below maximum throughput.

The FIFOs serve to address the latency imbalance that stems from both the imbalance of the DFG itself and imbalance that may arise due to different DFG edges being mapped with different number of routing hops. However, since the FIFOs have limited capacity, it is important to find a DFG mapping that ensures that the latency margins of the FIFOs are sufficient to overcome both sources of imbalance, thus achieving a latency-balanced state and maximum DFG throughput. We design a mapping algorithm that places a DFG, routes it, and then checks whether or not the resulting mapping is balanced. The algorithm iteratively performs DFG placement with different seeds (seed sweeping) until it finds a seed that produces a DFG placement that is both routable and balanced.

**Optimizing $f_{\text{MAX}}$ and resource usage:** The upper-bound of an overlay instance’s $f_{\text{MAX}}$ is the $f_{\text{MAX}}$ of the slowest FU within that overlay instance. Thus, the first step in designing an overlay architecture to achieve high $f_{\text{MAX}}$ is the selection of an FU library and the parametrization of FU pipeline depths. Our approach chooses a set of FUs and tunes their parameters such that most FUs can achieve an $f_{\text{MAX}}$ of at least 400 MHz (the overlay’s target $f_{\text{MAX}}$). This methodology facilitates a high upper-bound $f_{\text{MAX}}$ for most overlay instances (only those instances with slower FUs would have an upper-bound $f_{\text{MAX}}$ that is lower than 400 MHz).

We apply and evaluate this methodology using integer and floating-point FUs from Altera’s IP library and compile them to Altera’s Stratix IV architecture. The library enables the parametrization of FUs in terms of pipeline depth and data-path widths. We
analyzed the FU library and determined that most FUs can be compiled with a pipeline depth that enables the FU itself to achieve 400 MHz. This methodology is generic and can be applied to other FU libraries (by vendors such as Xilinx and Achronix or academic libraries such as FloPoCo [112] and VFLOAT [113]). Furthermore, FU libraries can be targeted to different FPGA families (Altera’s IP can be targeted to Stratix, Arria and Cyclone families) or even FPGA architectures by different vendors (FloPoCo and VFLOAT can be targeted to Altera and Xilinx FPGAs).

Furthermore, we optimize the $f_{\text{MAX}}$ and resource usage of the overlay architecture by determining the minimum amount of elastic buffers that are required within an overlay cell to enable a small 2x2 overlay to achieve the target $f_{\text{MAX}}$. We do this by tuning the granularity of elasticity; we determine what portion of the overlay cell’s pipeline should be elastic (EBs) and what portion should be inelastic (regular registers with inelastic synchronization). Specifically, our design uses entirely inelastic FUs which we surround with 2-slot EBs. We then optimize the design of these 2-slot EBs for FPGA fabric. Moreover, we determined that 32-slot and 64-slot EBs sufficiently function as a latency balancing mechanism and need to only be located at cell’s input ports. Finally, we employ regular registers at the cell’s output ports, thus saving resources.

**Routability:** We increase the routability of the overlay by adding routing-only overlay cells into the mesh. These cells do not have an FU and consume fewer resources than regular cells. DFG compute nodes are not mapped to routing-only cells; thus, routing-only cells reduce the density of mapped FUs in the mesh (i.e., increasing its porosity), in effect decreasing routing congestion. However, employing these cells within the overlay can increase its resource cost. The decision as to how many routing-only cells should be added to an overlay’s mesh is a trade-off between routability and resource cost.

**Scalability:** There is no global or centralized control in the overlay other than the clock signal. Thus, the overlay architecture has the potential for scaling to the size of modern FPGAs. However, in Chapter 6 we demonstrate that the $f_{\text{MAX}}$ of a large overlay
will be limited by the CAD tool if a flat (i.e., push-button) flow is used. We address this challenge by designing a tile-based bottom-up flow that utilizes hierarchical physical design techniques—specifically, partitioning and floorplanning—to exploit the regularity and modularity of the overlay circuit. The overlay is partitioned into tiles—groups of adjacent overlay cells—which are then compiled to a rectangular coarse-grain floorplan. Furthermore, we leverage the latency insensitivity [103] of the overlay architecture and insert special EBs on inter-tile paths to ensure these paths are not a bottleneck for $f_{\text{MAX}}$. This enables the independent compilation of tiles as well as the ability to compile them in parallel.

**Matching the overlay’s functional layout to FPGA layout:** We approach the challenge of asymmetric and non-uniform FPGA layout by designing overlay functional layouts that have DSP-based overlay cells arranged to *DSP overlay columns*, which form a subset of all overlay columns. Second, we position the DSP overlay columns within the overlay such that, when the overlay is placed-and-routed onto FPGA device, the logic resources of the DSP overlay columns are clustered around the FPGA DSP columns, i.e., the layouts are matched.
Chapter 4

Overlay Architecture

Implementation

4.1 Overlay Cell Architecture

An overview of the overlay cell architecture is shown in Figure 4.1. The cell’s input ports are labeled N$_{IN}$, W$_{IN}$, E$_{IN}$ and S$_{IN}$, and the output ports are labeled N$_{OUT}$, W$_{OUT}$, E$_{OUT}$ and S$_{OUT}$. The main components of the cell are: an FU, routing logic, data-driven pipeline units (DDPUs), and join/fork synchronization logic. The DDPUs are a set of FPGA-tailored EBs, elastic FIFOs and regular inelastic registers that communicate in an elastic manner. The overlay cell contains 11 DDPUs: four at the input ports, four at the output ports, two at the FU inputs and one at the FU output. The cell’s routing logic comprises muxes and wires that facilitate the run-time reconfigurable connectivity among input ports, output ports, and FU inputs and outputs. Join synchronization logic (JOIN) synchronizes the arrival of data into the FU and fork synchronization logic (FS and FR) synchronizes the forking (i.e., fan-out) of data from one DDPU to multiple other DDPU within the cell. The FU is augmented with a DDPU wrapper that enables the FU to synchronize itself with the surrounding DDPU in an elastic manner. We elaborate
on each of the cell’s components in detail in the subsections that follow.

### 4.1.1 FUs

We use pipelined FUs that can produce a result every cycle, which thus allows for maximum throughput. In order to keep the FUs resource-efficient we opt to use FUs with entirely inelastic pipelines. In this work, we use a set of integer FUs and a set of single-precision floating-point FUs. While integer FUs can be synthesized using a range of data-widths, we use a standard 32-bit data-width. Thus, both sets of FUs have the same 32-bit data-width. Each FU can also be synthesized with a range of pipeline depths. Since the FU’s pipeline depth affects the FU’s $f_{\text{MAX}}$, selecting pipeline depths is our key design decision with regards to FUs. We generate the HDL for the FUs by using Altera’s megafuntions library [114, 115]. As described earlier in the chapter, alternative FU libraries can also be employed; however, we limit our focus to Altera-based FUs because it allows us to present and analyze a concrete example of an FU library that is mapped to a target FPGA device. In addition, focusing on Altera-based FU libraries allows us to substantiate our methodology for FU pipeline depth selection and our choice of the
Table 4.1: The FU set: function, pipeline depth and $f_{\text{MAX}}$. The FUs are generated from Altera’s megafunctions library [114, 115].

An overlay instance with a heterogeneous functional layout has multiple types of FUs, and the overlay’s $f_{\text{MAX}}$ will be limited by the slowest FU type. Thus, in order to maximize the $f_{\text{MAX}}$ of each overlay instance, it is necessary to avoid the use of slow FUs, which can create a bottleneck for overlay’s $f_{\text{MAX}}$. We analyzed the FU set and determined that most FUs can be synthesized to achieve at least 400 MHz, with only four FUs achieving a lower $f_{\text{MAX}}$ (between 330-400 MHz). Our FU set is shown in Table 4.1, which also shows the chosen pipeline depth for each FU and the $f_{\text{MAX}}$ that is achievable for that particular pipeline depth. The $f_{\text{MAX}}$ for each FU is obtained by compiling the FU by itself\(^1\). Thus, the reported $f_{\text{MAX}}$ is the upper bound on the FU’s $f_{\text{MAX}}$.

Furthermore, the target FPGA fabric sets an upper bound on the $f_{\text{MAX}}$ of the overlay architecture via the inherent $f_{\text{MAX}}$ limits of its FPGA blocks. In the case of Stratix IV, the DSPs have an $f_{\text{MAX}}$ limit of 446 MHz (ALUTs and M9K blocks can achieve a higher $f_{\text{MAX}}$). Thus, DSP-based integer and floating point multipliers are limited to 446 MHz. Since multiplication is one of the most frequent operations in applications, it is

\(^1\)We surround the FU by wrapper registers to ensure that all the input and output paths to and from the FU are included in the timing analysis.
Table 4.2: FU types: FPGA resource usage (Stratix IV). The prefix “f-” denotes floating point units.

likely that most overlays will contain multiplier FUs. The DSP’s $f_{\text{MAX}}$ limit matches the average $f_{\text{MAX}}$ of the floating-point FUs (450 MHz). We consider this to be the “native” FPGA fabric speed of Stratix IV for compute applications. Thus, we design the overlay’s architecture—DDPUs, routing and synchronization— to target an $f_{\text{MAX}}$ between 400 and 446 MHz.

The resource usage of the FUs is shown in Table 4.2. Different FUs use a wide range of logic resources (ALMs); for example, $f$-log uses 6x more ALMs than to $f$-mult-dsp and 20x more than $f$-cmp. Furthermore, FUs also use a wide range of DSP elements. For example, $f$-addsub and $shift$ do not use any, $mult$ and $f$-mult-dsp use 4 DSPs while $f$-exp uses 22 DSPs. This variability in both the type and the amount of resources used by the FUs poses challenges for the functional layouts of overlays as well as for the physical design of these overlays in the FPGA fabric. We elaborate on our approach in Chapter 6.

An FU consists of a data-path pipeline that can be controlled by a single enable signal (i.e., inelastic pipeline), and an FU does not contain any other synchronization signals or synchronization logic. In order to interface such an FU with the surrounding DDPUs, we design a $DDPU\ wrapper$ for the FU, which is shown in Figure 4.2. The

---

2This excludes the trivial $f$-abs FU which can run at a single-ALUT speed of 800 MHz.
Chapter 4. Overlay Architecture Implementation

![FU DDPU wrapper](image)

**Figure 4.2:** A DDPU wrapper for a 2-input FU.

DDPU wrapper extends the FU’s data-path pipeline with elastic protocol signals (valid and accept) and valid bit registers. The DDPU wrapper is then directly connected to the join synchronization logic at its input and the DDPU FU_OUT at its output (the join logic that synchronizes data for the FU is described in Section 4.1.4).

For an FU with pipeline depth \(N\) we add \(N\) valid bit registers (\(v_1\) to \(v_N\)). Each valid register indicates whether the corresponding FU’s pipeline register contains valid data. The valid registers are connected to form a shift register. The input to the valid shift register is a signal \(V_{IN}\) which is generated by the join synchronization logic, while the output of the valid shift register (\(V_{OUT}\)) is fed to the FU output DDPU (FU_OUT). The accept signal \(A_{OUT}\) from the DDPU at the FU’s output (FU_OUT) serves as the enable signal for the FU’s data pipeline registers and the valid registers. In any cycle where \(A_{OUT}\) is set high, the data in the valid registers and the FU data pipeline shift in lockstep to the right. With this in mind, it is important to note that the wrapper does not add any elasticity to the FU (e.g., no EBs or FIFOs). The FU’s pipeline remains entirely inelastic with \(A_{OUT}\) serving as an enable signal. Furthermore, \(A_{OUT}\) is also directly back propagated to the join synchronization logic (\(A_{IN}=A_{OUT}\)).

As mentioned earlier, an overlay cell can have multiple FUs of which only one can be active at run-time. Figure 4.3 shows how this is implemented by combining the DDPU
Figure 4.3: A Multi-FU DDPU wrapper that contains two FU DDPU wrappers.

Table 4.3: FU types: run-time configurable operations.

<table>
<thead>
<tr>
<th>FU type</th>
<th>Data type</th>
<th>Run-time configurable operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>shift</td>
<td>int</td>
<td>$a \ll b$, (signed)$a \gg b$,</td>
</tr>
<tr>
<td>addsub</td>
<td>int</td>
<td>$a + b$, $a - b$, $a + b_{\text{const}}$, $a + 0$, $0 - a$</td>
</tr>
<tr>
<td>abs</td>
<td>int</td>
<td>$\text{abs}(a)$</td>
</tr>
<tr>
<td>mult</td>
<td>int</td>
<td>$a \times b$, $a \times b_{\text{const}}$</td>
</tr>
<tr>
<td>div</td>
<td>int</td>
<td>$a / b$</td>
</tr>
<tr>
<td>f-abs</td>
<td>fp sp</td>
<td>$\text{abs}(a)$</td>
</tr>
<tr>
<td>f-cmp</td>
<td>fp sp</td>
<td>$a &lt; b$, $a &gt; b$, $a &lt; 0$, $a &gt; 0$</td>
</tr>
<tr>
<td>f-addsub</td>
<td>fp sp</td>
<td>$a + b$, $a - b$, $a + b_{\text{const}}$, $a + 0$, $0 - a$</td>
</tr>
<tr>
<td>f-sqrt</td>
<td>fp sp</td>
<td>$\sqrt{a}$</td>
</tr>
<tr>
<td>f-mult-dsp</td>
<td>fp sp</td>
<td>$a \times b$, $a \times b_{\text{const}}$</td>
</tr>
<tr>
<td>f-mult-lut</td>
<td>fp sp</td>
<td>$a \times b$, $a \times b_{\text{const}}$</td>
</tr>
<tr>
<td>f-log</td>
<td>fp sp</td>
<td>$\log(a)$</td>
</tr>
<tr>
<td>f-div</td>
<td>fp sp</td>
<td>$a / b$, $1 / a$</td>
</tr>
<tr>
<td>f-exp</td>
<td>fp sp</td>
<td>$\exp(a)$</td>
</tr>
</tbody>
</table>

wrappers of two FUs into a multi-FU DDPU wrapper (a “parent” wrapper). A multi-FU DDPU wrapper has the same interface as the FU DDPU wrappers. Input data and valid signals ($D_{IN1}$, $D_{IN2}$, and $V_{IN}$) are forwarded to both DDPU wrappers. Similarly, $A_{OUT}$ is connected to both wrappers. The output data and valid signals of both FU wrappers are then fed to a mux that is controlled by the configuration bit $FU_{SEL}$, which steers the output of the selected FU to the parent wrapper’s output.

Table 4.3 shows the run-time reconfigurable operations implemented by each FU. These operations are sufficient to implement the set of DFGs that we used to evaluate the overlay architecture. In future work, this FU set can be augmented with other FU
types and operations. Some FUs in our set support only a single-operation, such as \( f\text{-abs} \), \( f\text{-log} \), and \( f\text{-exp} \), while others, such as \( addsub \) and \( f\text{-addsub} \) FUs, support multiple operations.

In regular addition and subtraction \((a + b)\) and \((a - b)\) operations both FU inputs change every cycle; thus, the FU executes the operation on a new pair of values every cycle. We extend a subset of the FUs with operations in which one of the FU inputs is constant. We do this in two ways. The first way is through the use of a \textit{loaded constant}, a mode in which the second FU input is only loaded at the start of DFG execution (i.e., initialized) and afterwards is held constant (e.g., \( a + b_{\text{const}} \) in the \textit{addsub} FU). Each loaded constant is routed to the FU from a dedicated DFG input in the same manner that regular DFG inputs are routed to FUs. Loaded constants require special consideration in the design of join synchronization logic (Section 4.1.4) as well as in the balancing algorithm (Chapter 5). The second way is by extending some FUs with a mode in which the second FU input is a \textit{built-in constant} that does not need to be loaded (e.g., \( a + 0 \) and \( 0 - a \)). In this mode, the FU is synchronized in the same manner as 1-input FUs.

### 4.1.2 Routing Logic

The routing logic of the overlay cell has been designed to maximize the routability of DFGs on a 4-NN mesh overlay topology. This is achieved by utilizing two types of connections that allow for full connectivity among the cell’s input ports and output ports as well as the FU’s inputs and outputs. Furthermore, the connectivity has been designed such that it maximizes the ability to fan-out data within the cell.

The first type of routing connections in an overlay cell are those that bypass the FU, as shown in Figure 4.4a. Each output port of a cell is connected to three of the cell’s inputs ports; it is not connected to the input port on the same “side” of the cell. This sort of “U-turn” connection would simply route data back to the sender cell, and would consume the FPGA’s resources without contributing to the overlay’s routability. For
example, the output port $S_{\text{OUT}}$ can receive data from either $W_{\text{IN}}$, $N_{\text{IN}}$, or $E_{\text{IN}}$ but not from $S_{\text{IN}}$.

The second type of routing connections feed data from the cell’s input ports into the FU, and from the FU to the output ports, as is illustrated in Figure 4.4b. Each FU input can receive data from any of the cell’s input ports. The FU’s output can feed data to all of the cell’s output ports.

Figure 4.5 shows how the cell’s routing connectivity is implemented with muxes and data-paths. Each input port DDPU fans-out data to five DDPUs: three at the output ports and two at the FU inputs. The figure shows all connections for the input port $N_{\text{IN}}$; it omits other connections for clarity. The FU output DDPU fans-out its data to four output ports that can each select one of four data-paths: the three input ports and the FU output. This is implemented using a 4-to-1 mux for each output port. The figure shows the data-path connections that feed the mux of the output port $S_{\text{OUT}}$. Each FU input can select one of four data-paths that correspond to the four inputs ports, which is also implemented using 4-to-1 muxes.

Thus far, we have described all of the connections that are implemented by the data-
paths and muxes within the overlay cell. We next describe how cell’s routing connections and fan-outs can be reconfigured at run-time in order to establish the specific routing paths necessary for the execution of a particular DFG. The routing paths within the cell are established by configuring the select bits of the six 4-to-1 muxes. For example, all five fan-out connections of the input port \( N_{IN} \) can be used at run-time if all of the five muxes it connects to are configured to select data from it. In general, if \( K \ (K \leq 5) \) muxes are configured to select data from the same input port, then the fan-out of that input port at run-time is \( K \).

Fan-outs larger than five are established by fan-out cascading across multiple neighbouring overlay cells. An example of this method is shown in Figure 4.6. A routing path enters the top left cell, wherein it has a fan-out of four, and feeds the FU. It then fans-out further into the neighbouring cells and feeds two more FUs. In the four cells shown in the figure, the total fan-out is 8.
Routing-only Overlay Cells

We design routing-only overlay cells which can be added to an overlay’s mesh in order to increase the routability of the overlay. A routing-only overlay cell does not have an FU or the surrounding DDPUs (FU\textsubscript{IN1}, FU\textsubscript{IN2} and FU\textsubscript{OUT}) and thus consumes fewer resources than any overlay cell with an FU. Routing-only cells can be placed at any location in the mesh. Figure 4.7a shows an example of an overlay mesh that contains routing-only cells. In order to ensure clarity, it should be noted that we refer to cells with an FU as compute-routing cells. Figure 4.7b shows the routing connections of a routing-only cell, which are equivalent to the FU-bypass routing connections of a compute-routing overlay cell.

Since DFG compute nodes are not mapped to routing-only cells, all of a routing-only cell’s input and output ports are available for routing hops. In contrast, mapping a DFG compute node to a compute-routing overlay cell with a 1-input FU (e.g., f-log) uses up one of the cell’s input ports and at least one of its output ports (this depends on how the node’s fan-out is cascaded by the router). Thus, at least 2 out of 8 ports (25%) in a cell with a mapped FU will be in use, while the remaining ports will be available for routing hops (via FU bypass connections).
Furthermore, in the case of a cell with a 2-input FU, mapping a DFG node to its FU will use up at least three of the cell’s ports (3/8 or 37.5%), and if the FU is configured to fan out to two of the cell’s output ports, then 4 ports in total will be used up (4/8 or 50%). At the extreme, a 2-input FU with a fan-out of 4 may use up all 4 outputs ports. Since no output ports are left available, this cell cannot be used for routing hops. It should be noted that the router may cascade the fan-out of 4 across neighbouring cells, thereby making routing hops available in the source cell (the router is discussed in Section 5.4).

The addition of routing-only cells to the overlay increases the amount of routing resources that are available for use as routing hops, i.e., for connecting FUs in non-adjacent cells. Routing-only cells also decrease the density of FUs within the overlay, which in turn decreases the density of mapped FUs in the overlay, which consequently decreases routing congestion thus improving routability.

However, the use of routing-only cells increases the percentage of the FPGA’s resources that are dedicated to the overlay’s routing resources. For example, routing-only overlay cells may increase the resource usage of mapping an easily routable DFG to the
overlay, because such a DFG may not require the routing-only cells to be routable. The decision as to how many routing-only cells should be added to the overlay is a trade-off between the routability of DFGs of various topologies and the resource usage of an overlay.

The same effect—decreasing the density of mapped FUs within the overlay—can be accomplished by having compute-routing overlay cells serve as routing-only cells. This can be done by marking a compute-routing cell within the overlay’s functional layout (an input passed to the DFG-to-overlay mapping tool) as a routing-only cell. Hence, the mapping tool will not map a DFG node to any such marked cell, effectively treating them as routing-only cells. With this approach, improved routability is achieved by deliberately not utilizing FUs that have been synthesized as a part of the overlay.

The latter approach is more flexible than using dedicated routing-only cells as the number and arrangement of marked routing-only cells within the functional layout can be adapted on a per-DFG basis. However, for harder-to-route DFGs, which may require a high number of routing-only cells, the marked routing-only cells approach is more expensive than the use of dedicated routing-only cells. This is because each compute-routing cell that is marked as as a routing-only cell “wastes” an FU, and thus a large amount of FPGA resources would be wasted. In contrast, since dedicated routing-only cells do not contain FUs, the cost of increasing the routability via dedicated routing-only cells is lower. Therefore, we opt to use the former approach.

### 4.1.3 Data-Driven Pipeline Units (DDPUs)

DDPUs are pipeline units that communicate and synchronize in an elastic manner (i.e., using the elastic protocol). We design three types of DDPUs: 2-slot elastic buffers, N-slot elastic FIFOs, and inelastic registers. The design of the overlay cell architecture uses a combination of these DDPU types to achieve target $f_{\text{MAX}}$, scalability to large meshes, and elasticity while using an acceptable amount of resources. In this section, we first
describe the design, characteristics, and role of each DDPU type, before moving on to
describe the arrangement of the various DDPU types within the overlay cell.

2-Slot Elastic Buffers

In ASIC fabric, 2-slot elastic buffers (EBs) are efficiently implemented using a master-
slave latch pair with no data-path overhead [47]. In contrast, the FPGA fabric necessi-
tates the use of flip-flops (FFs) to implement two data-path registers (e.g., each 32-bits)
and a data-path mux. This has two major implications: first, it incurs considerable
resource cost over a single 32-bit register; and second, there are several possible arrange-
ments of the EB’s data registers and muxes, which gives rise to an EB design space. As
a result, the design of the EBs and their arrangement within the overlay cell affects the
overlay’s functionality, $f_{\text{MAX}}$ and resource usage.

We design and explore the use of two types of 2-slot EBs (type-1 and type-2) which
differ from one another in the design of the EB’s data-path (the basic 2-slot EB design is
introduced in Section 2.4). In addition, we design three versions of each type of EB that
each differ in their functional behaviour, and thus differ in control logic implementation.
This is described in detail below.

Figure 4.8 shows all six versions of EB implementation: the three versions of the
type-1 EB are shown in Figures 4.8a to 4.8c, while the three versions of type-2 are shown
in Figures 4.8d to 4.8f.

In the type-1 EBs, the data-path mux is between the aux and main data registers,
whereas it is located after the two data registers in the type-2 EBs. The location of the
data-path mux not only affects the mapping of the EB to the FPGA logic resources, but
it also affects the timing delay between EBs connected in a pipeline as well as between
EBs and the other components of the cell, including the FU, synchronization logic and
FIFOs. This effect can be observed by connecting D-BUF1 and D-BUF2 (Figures 4.8c
and 4.8d) in a pipeline. The paths between the two EBs are registered at both ends,
Figure 4.8: Six versions of a 2-slot elastic buffer implementation: (a)(b)(c) type 1 versions, (d)(e)(f) type 2 versions.
and there is no combinational logic on the paths. We say that these two EBs have a mirrored structure. In contrast, if two D-BUF1 EBs are connected in a pipeline, then the paths between the two EBs pass through a mux which results in larger timing delays. We leverage the mirrored structure of D-BUF1 and D-BUF2 in our tile-based bottom-up flow to ensure that inter-tile paths are not a bottleneck for $f_{\text{MAX}}$. This is elaborated upon in Section 6.6.1.

The three versions of each EB type have different functional behaviour. For a steady stream of data without stalls, all versions have identical behaviour in that they have a latency of 1 clock cycle and use only their main registers to store data. In other words, when no stalls are present all EBs behave like regular registers; however, the differences in the behaviour of the three versions arise when there are bubbles and stalls in the pipeline.

The first version of type-1 is D-EB1 (Figure 4.8a). This version is functionally equivalent to a 2-slot FIFO, which means that when its output is stalled, it can fill up both of its data registers for any pattern of data and bubbles at its input. Additionally, D-EB1 will back-propagate a stall to the upstream units only when it is full (i.e., contains two data items).

We term the second version a semi-elastic buffer (D-SEB1 in Figure 4.8b). Its functionality is a subset of D-EB1, which results in it having a simpler control logic. In contrast to D-EB1, D-SEB1 will not fill up both of its registers when its output is stalled or when it receives a pattern of alternating data and bubbles at its input (the auxiliary slot will contain a bubble). However, much like D-EB1, an empty D-SEB1 with a stalled output does not back-propagate a stall to the upstream units. This ensures the forward progress of data in the upstream units allowing D-SEB1 to eventually receive valid data even while its output is stalled.

The third version is termed a stall buffer (D-BUF1 in Figure 4.8c), and this version’s functionality is a subset of D-SEB1 which results in it having the simplest control logic.
In contrast to D-SEB1, D-BUF1 back-propagates a stall even if it is empty. Thus, D-BUF1 does not ensure the progress of data in the upstream units that would allow it to receive valid data while its output is stalled (it will remain empty).

D-EB2, D-SEB2, and D-BUF2 have equivalent functionality to D-EB1, D-SEB1, and D-BUF1, respectively. The simpler control logic of D-SEB1/2 and D-BUF1/2 can help increase $f_{\text{MAX}}$ at the expense of full capacity in the above-described stall scenarios. Moreover, D-SEB1/2 are sufficient to implement any elastic pipeline as they ensure the forward progress of data in pipelines where pipeline paths join together (i.e., a join point). In contrast, an elastic pipeline cannot be built using only D-BUF1/2 as they do not ensure forward progress in pipelines with join points; therefore, they must be accompanied by D-EB1/2 or D-SEB1/2 units. The arrangement and choice of EBs within the overlay cell and the trade-off between resource usage and $f_{\text{MAX}}$ is elaborated upon in Section 4.1.3, while the design of the join synchronization logic and the requirements for the EBs used at pipeline join points is detailed in Section 4.1.4.

The FPGA resource usage of the EB versions is shown in Table 4.4. All EBs require 67 FFs, which thus makes 34 ALMs the minimum resource usage of an EB (each Stratix IV ALM has two FFs). The data mux in type-1 EBs can be implemented using a hard 2-to-1 synchronous load mux of the ALM, thereby avoiding the use of the ALM’s ALUTs. This frees up 32 ALUTs and provides an opportunity to pack the overlay cell’s logic that feeds the EBs into these ALUTs.

Since EBs are comprised of flip-flops and logic with a few inputs (2-to-1 muxes and 2-input gates), we expect that their mapping will be similar across LUT/flip-flop-based FPGA architectures.

**N-Slot Elastic FIFO**

The design of our N-slot elastic FIFO ($D\text{-FIFO}$ in Figure 4.9) is based on Altera’s single-clock FIFO (SCFIFO [116]). An N-slot D-FIFO has a capacity of $N$ data words. We use
Chapter 4. Overlay Architecture Implementation

<table>
<thead>
<tr>
<th>DDPU type</th>
<th>Data width</th>
<th>ALUTs</th>
<th>MLUTs</th>
<th>ALUTs (total)</th>
<th>FFs</th>
<th>ALMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-EB1</td>
<td>32</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>67</td>
<td>37</td>
</tr>
<tr>
<td>D-SEB1</td>
<td>32</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>67</td>
<td>37</td>
</tr>
<tr>
<td>D-BUF1</td>
<td>32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>67</td>
<td>37</td>
</tr>
<tr>
<td>D-EB2</td>
<td>32</td>
<td>35</td>
<td>0</td>
<td>35</td>
<td>67</td>
<td>37</td>
</tr>
<tr>
<td>D-SEB2</td>
<td>32</td>
<td>34</td>
<td>0</td>
<td>34</td>
<td>67</td>
<td>34</td>
</tr>
<tr>
<td>D-BUF2</td>
<td>32</td>
<td>33</td>
<td>0</td>
<td>33</td>
<td>67</td>
<td>34</td>
</tr>
<tr>
<td>D-REG</td>
<td>32</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>33</td>
<td>20</td>
</tr>
<tr>
<td>D-FIFO-4</td>
<td>32</td>
<td>17</td>
<td>32</td>
<td>49</td>
<td>49</td>
<td>38</td>
</tr>
<tr>
<td>D-FIFO-8</td>
<td>32</td>
<td>21</td>
<td>32</td>
<td>53</td>
<td>55</td>
<td>42</td>
</tr>
<tr>
<td>D-FIFO-16</td>
<td>32</td>
<td>27</td>
<td>32</td>
<td>59</td>
<td>61</td>
<td>44</td>
</tr>
<tr>
<td>D-FIFO-32</td>
<td>32</td>
<td>31</td>
<td>32</td>
<td>63</td>
<td>67</td>
<td>48</td>
</tr>
<tr>
<td>D-FIFO-64</td>
<td>32</td>
<td>35</td>
<td>64</td>
<td>99</td>
<td>73</td>
<td>61</td>
</tr>
<tr>
<td>D-FIFO-128</td>
<td>32</td>
<td>74</td>
<td>128</td>
<td>202</td>
<td>110</td>
<td>109</td>
</tr>
</tbody>
</table>

Table 4.4: DDPU types: FPGA resource usage (Stratix IV).

Figure 4.9: N-slot elastic FIFO.
the *show-ahead* mode of the SCFIFO, which uses a read-acknowledge (*rdack*) mechanism to output data. The FIFO provides the first available piece of data on its data-output bus (*Q*) without needing to first request it. The assertion of the *rdack* signal triggers the FIFO to provide the next available piece of data in the subsequent clock cycle. The FIFO’s *empty* signal is set low (i.e., not empty) in the same cycle as the data at output *Q* becomes valid.

We augment the show-ahead SCFIFO with logic that translates the FIFO’s control signals into elastic valid and accept/stall signals. The data is written into the FIFO (*wrreq*) when the it is not full and the data at its input is valid (*V_P*). Similarly, the read-acknowledge signal is asserted when the FIFO is not empty and the accept signal (*A_N*) is high.

The minimum latency of the show-ahead FIFO is 2 clock cycles; that is, it takes at least 2 clock cycles for data to become available at output *Q* after the write-request signal has been asserted (i.e., the signal to write data into the FIFO). Thus, the maximum throughput steady state\(^3\) of D-FIFO can be achieved at the minimum occupancy of 2. The occupancy of the D-FIFO is increased by one in any cycle wherein there is a stall at the FIFO output (no data leaves) but data enters it. If the D-FIFO’s output is unstalled at an occupancy of *K* data words, then the D-FIFO’s new maximum-throughput-steady-state latency will be *K* clock cycles.

The D-FIFO stalls its input when its occupancy reaches *N* data words (in a stalled-output state) and unstalls it again once its output has been unstalled and one data word has left the D-FIFO. Thus, the maximum occupancy (and thus maximum latency) at which the FIFO operates at maximum throughput is *N-1* data words.

The ability of the D-FIFO’s occupancy/latency to dynamically increase at run-time is the basis of the latency balancing mechanism of the overlay. The D-FIFOs on the shorter paths of a DFG that has been mapped to an overlay increase their occupancy
(and thus latency) during stalls in order to match the latency of the longer paths. We describe this mechanism in detail in Section 4.2.

We explore the use of D-FIFOs with capacities of up to 128 data words, and we implement them using MLABs (LUT-based SRAM). The 4-slot D-FIFO is the smallest capacity that can be implemented using MLABs. Table 4.4 shows the FPGA resource usage for D-FIFOs with capacities of 4 to 128 words. We can observe that D-FIFOs with capacities between 4 and 32 consume 32 MLUTs (implemented as 16 ALMs), and that their total amount of ALMs is similar. This is because each ALM natively implements a 32-word × 2-bit SRAM block, which means that D-FIFOs shallower than 32 words that are built using these SRAM blocks are underutilizing their depth. Furthermore, a 32-slot D-FIFO uses only 2.4x and 1.3x more ALMs than the 1-slot D-REG and 2-slot EBs, respectively. Hence, MLAB-based FIFOs provide an efficient approach for balancing the latencies of DFG paths that have been mapped to the overlay.

We use only MLAB-based FIFOs and we avoid the use of M9K blocks. In the Stratix IV architecture every other LAB can serve as an MLAB which makes them available anywhere on the FPGA device. Since D-FIFOs (indeed all DDPUs) are implemented using LABs/MLABs, DDPUs do not impose any constraints on the placement of overlay cells to an FPGA device. In contrast, the columns of M9K blocks are spread throughout the FPGA device, and this may complicate the placement of overlay cells consisting of M9K-based FIFOs. We leave this investigation for future work.

LUT-based SRAM blocks, such as Stratix IV MLABs, are also commonly available in FPGA devices from other vendors (Distributed RAM in Xilinx devices [117, 118] and LRAM in Achronix devices [119]). Thus, the efficient D-FIFO implementation that we have described for Stratix IV could also be achieved on other FPGA architectures.
Inelastic Register

The simplest DDPU is an inelastic pipeline register (\textit{D-REG}), which is shown in Figure 4.10. It holds only a single data word (i.e., a single slot) and thus sets the lower bound on the FPGA resources required to implement a single-slot pipeline unit. It uses 33 flip-flops and can be implemented with a minimum of 17 ALMs. Depending on the degree of register packing that was achieved in a particular compile the resource usage can vary slightly (on average 20 ALMs, as is shown in Table 4.4). Additionally, D-REG directly back-propagates the accept signal (i.e., without registering it). The latency of D-REG is 1 clock cycle.

The Arrangement and Selection of DDPU\textsuperscript{s} within the Overlay Cell Architecture

Figure 4.1 shows a generic overlay cell architecture that can contain up to 11 DDPU\textsuperscript{s} at the following locations: four at the input ports, four at the output ports, two at the FU inputs, and one at the FU output. Each of these DDPU\textsuperscript{s} can be implemented as a different DDPU type. An instance of the overlay cell architecture is designed by selecting DDPU types for each of 11 locations.

For example, we can design the cell with D-FIFO-32 units at the input ports and D-EB1 units at all other DDPU locations. Moreover, the overlay cell’s resource usage
can be reduced by omitting DDPUs at some locations. In one instance of a cell we could
omit the output port DDPUs; in this case, only the DDPUs at the input ports would
serve as pipeline units on the routing paths between adjacent overlay cells. Furthermore,
different overlays cells within an overlay could potentially use a different set of DDPU
types.

This flexibility gives rise to the design space of overlay cell architecture. We first
present the design constraints of DDPU selection within a cell, and we then move on to
describe an overlay cell design with a DDPU selection that offers a trade-off between high
\( f_{\text{MAX}} \), high elastic latency margin for latency balancing, and reasonable resource usage.

A low resource usage of the overlay cell could be achieved with a DDPU selection
wherein D-REGs are placed at the cell’s input and output ports; and then further mini-
mized by placing D-REGs only at the input ports with no DDPUs at the output ports.
However, we show below that such a design is not feasible, which thus imposes a design
constraint: D-REGs cannot be the only DDPU type that is used at the input and output
ports of each overlay cell.

Two issues arise in an overlay cell design that uses D-REGs at all of the cell’s ports
Figure 4.12: The selection and arrangement of DDPU types in the overlay cell architecture.

(demonstrated in Figure 4.11). The first issue is the presence of the combinational loops of the stall/accept signal (A). Combinational loops can cause instability and unreliability, and should therefore be avoided in synchronous design [120, 121]. Figure 4.11 shows an example combinational loop which occurs among four adjacent cells. Additionally, larger combinational loops that traverse more than four cells can also arise. The second issue that arises are the long combinational paths of stall/accept signal (A) that span the entire overlay.

The second design constraint concerns the DDPUs at the FU’s inputs (FUIN1 and FUIN2). The join synchronization logic (described in Section 4.1.4) stalls both FUIN1 and FUIN2 DDPUs until they both hold data (i.e., while any of the two is empty). For this reason, FUIN1 and FUIN2 must not back-propagate a stall when they are empty. This would block the progress of data in the upstream pipeline units and subsequently prevent FUIN1 and FUIN2 from receiving valid data during the stalled state. This limitation precludes the use of three DDPU types (D-REG, D-BUF1, and D-BUF2) for FUIN1 and FUIN2 since these back-propagate stalls when they are empty.

We evaluated various overlay cell designs and chose a design (shown in Figure 4.12) that offers a good trade-off between high fMAX, high elasticity for latency balancing and reasonable resource usage. After considering a number of different designs, we selected ...
D-SEB2 for FU\textsubscript{IN1} and FU\textsubscript{IN2}. We found that using EBs with full FIFO functionality (D-EB1 and D-EB2) was unnecessary. Instead, the simpler control logic of D-SEB2 allows for higher $f_{\text{MAX}}$ while addressing the second design constraint (D-SEB2 does not back-propagate stall when it is empty). We use D-BUF2 for FU\textsubscript{OUT}. D-SEB2 and D-BUF2 DDPUs pipeline all signals (data, valid and accept), and thus isolate the FU from the rest of the overlay cell’s logic. We also found that using D-SEB2 and D-BUF2 to isolate the FU results in slightly higher $f_{\text{MAX}}$ than using D-SEB1 and D-BUF1.

We use D-FIFO-32 (or D-FIFO-64) at each cell’s input port as it provides a high latency margin for each routing hop. Furthermore, D-FIFOs pipeline all signals, thus avoiding the issue of combinational loops and long stall signal paths (the first design constraint). We use D-REG at the output ports to pipeline the data and valid signals, opting not to pipeline the stall signal (e.g., by using D-BUF1/2). As a result, the overlay cell uses fewer resources. Since the stall signal is pipelined at each input port within the D-FIFO, the longest propagation delay of the stall signal occurs between the D-FIFOs of two adjacent overlay cells.

### 4.1.4 Reconfigurable Join and Fork Synchronization Logic

Join and fork synchronization logic [47, 48, 122] are used to synchronize non-linear elastic pipeline paths within the overlay cell. The join logic we design is run-time reconfigurable, which enables the run-time reconfigurable input modes of FUs. The fork logic is also run-time reconfigurable, and this enables the run-time reconfigurable routing connectivity within the cell. We describe each in turn below.

**Reconfigurable Join Synchronization Logic**

The elastic overlay pipeline allows the data to arrive at the FU\textsubscript{IN1} and FU\textsubscript{IN2} DDPUs at different times; that is, the overlay pipeline is latency insensitive. The join logic synchronizes the transfer of data from FU\textsubscript{IN1} and FU\textsubscript{IN2} to the FU by ensuring that the
FU begins execution only when valid data exists in both FU\textsubscript{IN1} and FU\textsubscript{IN2} and when the output of the FU is not stalled.

Figure 4.13a shows the location of 2-input join logic between the FU\textsubscript{IN1} and FU\textsubscript{IN2} DDPUs and a 2-input FU DDPU wrapper. Figure 4.13b shows the join logic implementation. Data signals (D\textsubscript{IN1} and D\textsubscript{IN2}) are directly forwarded from the DDPUs into the FU DDPU wrapper. Data entering the DDPU wrapper is valid only when data from both FU\textsubscript{IN1} and FU\textsubscript{IN2} is valid; that is, valid signal V\textsubscript{IN} is an AND of V\textsubscript{IN1} and V\textsubscript{IN2}. Similarly, FU\textsubscript{IN1} stalls (A\textsubscript{IN1} is 0) when either FU\textsubscript{IN2} does not have valid data, or when the FU DDPU wrapper is stalled (an AND of V\textsubscript{IN2} and A\textsubscript{IN}).

The join logic in Figure 4.13b is fixed, which means that it always synchronizes both FU\textsubscript{IN1} and FU\textsubscript{IN2} with the FU’s DDPU wrapper. An example of an FU using this type of join synchronization is the shift FU for which new data can be brought in every cycle through both FU\textsubscript{IN1} and FU\textsubscript{IN2}.

The join logic in Figure 4.13c is reconfigurable and can synchronize either both FU\textsubscript{IN1} and FU\textsubscript{IN2}, or only FU\textsubscript{IN1} with the FU’s DDPU wrapper. This configurability is required for the \textit{f-div} FU which implements a regular 2-input operation (a/b) but can also implement a 1-input operation with a built-in constant (1/a). In the case of the 1-input operation, the data from FU\textsubscript{IN2} is not used and synchronization with it is disabled by setting \textit{do-not-sync\textsubscript{IN2}} to 1. This type of configurability is also required in cells with two FUs where one is a 2-input FU and the other is a 1-input FU. An example of this type of configurability is a cell with \textit{addsub} and \textit{abs} FUs.

Figure 4.13d shows a different type of join configurability which allows for the permanent stalling of FU\textsubscript{IN2}. This functionality is required for operations in which one of the operands is a constant that needs to be loaded at run-time. The constant arrives at FU\textsubscript{IN2} via the overlay’s routing in the same manner as streamed data, however the constant remains “locked-in” during the execution. This is achieved by setting \textit{always-stall\textsubscript{IN2}} to 1, thus permanently stalling FU\textsubscript{IN2}. Once the constant arrives at FU\textsubscript{IN2}, the
Figure 4.13: (a) join synchronization logic between \( \text{FU}_{\text{IN1}} \), \( \text{FU}_{\text{IN2}} \) and \( \text{FU DDPU wrapper} \), (b) fixed 2-input join, (c) configurable 1-input/2-input join, (d) configurable 2-input join with a constant lock-in.
output data $D_{IN2}$ becomes valid and the valid signal $V_{IN2}$ becomes high. Since $FU_{IN2}$ is stalled, this state remains for the entire duration of the DFG’s pipelined execution. Hence, data in $FU_{IN2}$ is always available, and the FU will execute in any cycle where the streamed data in $D_{IN1}$ is also valid.

Both types of join configurability (1-input/2-input and constant “lock-in”) can be implemented at the same time in an overlay cell. For example, both types are implemented in overlay cells with the $f$-addsub FU, which contains operations that use built-in constants as well as those that use loaded constants (i.e., $a + 0$ and $a + b_{const}$). Note that 1-input-only FUs (such as $f$-exp, $f$-log) do not require any join synchronization logic.

**Reconfigurable Fork Synchronization Logic**

The overlay cell’s run-time reconfigurable routing enables the following fan-out capabilities: 1) each input port DDPU can fan-out (i.e., fork) to multiple output port DDPU and to $FU_{IN1}$ and $FU_{IN2}$; and 2) $FU_{OUT}$ can fork data to multiple output port DDPU (see Section 4.1.2 for a detailed description).

We refer to DDPU that fork data as *fork senders*, and to DDPU that receive the forked data as *fork receivers*. The overlay cell has 5 fork senders: 4 input ports and $FU_{OUT}$. It has 6 fork receivers: 4 output ports, $FU_{IN1}$, and $FU_{IN2}$. An overlay cell can be run-time configured to have up to 5 forks (one for each fork sender), each with a unique set of fork receivers.

The fork logic is required to synchronize the transfer of data between a fork sender and its receivers, as any of the fork’s receivers may stall in any cycle. Consequently, the sender must be stalled until all of the receivers are ready (i.e., unstalled) to receive the sender’s current data. Only then can the sender be unstalled and accept new data from its upstream unit.

We base our fork logic on *lazy fork* [102, 122] synchronization wherein all receivers accept valid data from the sender simultaneously, which occurs in the clock cycle when
all of them are ready. An alternative design could be based on eager fork, which allows receivers to accept data independently as they become ready. However, in either case, the fork sender is stalled until all of its receivers have accepted the current data. We opt for lazy fork because it uses simpler synchronization logic [102, 122].

We design a distributed fork logic wherein each fork sender and each fork receiver have their own synchronization logic. This design feature is denoted as fork receiver (FR) and fork sender (FS) blocks in Figure 4.1. The implementation details of the FS logic for input port N\textsubscript{IN} and the FR logic for output port S\textsubscript{OUT} are shown in Figure 4.14. The same figure also shows an example run-time configuration that establishes two forks: 1) the sender is N\textsubscript{IN} and the receivers are S\textsubscript{OUT}, W\textsubscript{OUT}, and FU\textsubscript{IN1}; and 2) the sender is FU\textsubscript{OUT} and the receivers are N\textsubscript{OUT} and E\textsubscript{OUT}.

The FS logic of input port N\textsubscript{IN} can be run-time configured to synchronize N\textsubscript{IN} with any subset (or all) of its 5 possible receivers (FU\textsubscript{IN1}, FU\textsubscript{IN2}, E\textsubscript{OUT}, S\textsubscript{OUT}, and W\textsubscript{OUT}), which is achieved by AND-ing the accept signals of the receivers. In Figure 4.14, N\textsubscript{IN} is synchronized with FU\textsubscript{IN1}, S\textsubscript{OUT}, and W\textsubscript{OUT} by setting the corresponding bits of the fork sender configuration mask to 1. It does not synchronize with the receivers FU\textsubscript{IN2} and E\textsubscript{OUT}, and thus the corresponding bits are set to 0.

The FR logic of output port S\textsubscript{OUT} can be run-time configured to synchronize S\textsubscript{OUT} with one of its 4 possible senders (FU\textsubscript{OUT}, N\textsubscript{IN}, W\textsubscript{IN}, and E\textsubscript{IN}). S\textsubscript{OUT} must also synchronize with the other receivers in the fork of the same sender; thus, it can synchronize with 5 other possible receivers. In Figure 4.14 S\textsubscript{OUT} is synchronized with W\textsubscript{OUT} and FU\textsubscript{IN1}, and the MUX steers the valid signal from the fork sender. The valid signal is then AND-ed with the accept signals of the other receivers in the fork. The fork receiver configuration mask comprises two control bits for the MUX and a five-bit mask that establishes synchronization with receivers of the same fork.

A 5×6 fork configuration matrix specifies the forks that will be run-time configured within a particular overlay cell. The configuration masks for each fork sender and each
Chapter 4. Overlay Architecture Implementation

An example run-time configuration with 2 forks:

Fork_1
\[
\begin{align*}
\text{Sender}_1 &= \{ N_{IN} \} \\
\text{Receivers}_1 &= \{ S_{OUT}, W_{OUT}, F_{UIN1} \}
\end{align*}
\]

Fork_2
\[
\begin{align*}
\text{Sender}_2 &= \{ F_{OUT} \} \\
\text{Receivers}_2 &= \{ N_{OUT}, E_{OUT} \}
\end{align*}
\]

Figure 4.14: The design of a fork sender (FS) and a fork receiver (FR). An example overlay cell configuration with two forks.
fork receiver are derived from this matrix. The matrix elements have binary values, and the matrix contains a row for each fork sender and a column for each fork receiver. Ones in a row denote the fork receivers of the corresponding fork sender.

### 4.2 Latency Balancing of the Overlay’s Pipeline Paths for Maximizing Throughput

The maximum throughput of pipelined execution of a DFG that has been mapped to the overlay is a steady state in which one DFG instance completes execution every cycle. In other words, the DFG’s output nodes—which are mapped to the overlay’s output ports—output data every cycle. Furthermore, each mapped FU executes every cycle when in the maximum throughput state.

Achieving maximum throughput requires that the latencies of all distinct paths from the DFG input nodes to an FU be equal, or balanced. However, even if the balancing condition is not satisfied, the DFG execution on the overlay will still be functionally correct because the overlay’s FUs can be stalled on a per-FU basis: each FU will stall during any cycle where its input data is not ready. Due to the stalls, the mapped DFG will execute at suboptimal throughput, and what throughput is achieved will depend on the degree of imbalance in the mapped DFG.

We also adopt an additional balancing condition (introduced in Section 2.5): all paths through a mapped DFG— from the input to output DFG nodes— must have the same latency (we term this mapped DFG latency). As a result, it is not necessary to feed data into DFG’s input nodes independently or to provide additional buffering of DFG input nodes outside of the overlay. In other words, data can be fed to all inputs of a balanced DFG simultaneously. Similarly, data can be read out of all DFG outputs simultaneously. The DFG outputs become ready after the DFG pipeline is filled up—this

---

4The basics of latency balancing are described in Section 2.5.
Figure 4.15: (a) An example DFG with two sub-DFGs highlighted (A and B), b) mapped to a 3x3 overlay.

The latency-balanced state of a DFG that has been mapped to the overlay is achieved dynamically at run-time. We demonstrate this mechanism by using the DFG in Figure 4.15a and its mapping to a 3x3 overlay, as shown in Figure 4.15b. Specifically, we study the mapping of sub-DFGs A and B. The latencies of the paths of a mapped DFG are examined through the use of a post-map DFG, which is a graph that represents the mapping (i.e., placement-and-routing) of a DFG to the overlay. The graph’s nodes represent two types of pipeline units: 1) the overlay’s FUs to which the DFGs nodes were mapped; and 2) the overlay’s DDPUs that were traversed by the routing paths connecting the FUs. The graph’s edges are the overlay’s routing connections that were used to connect those FUs and DDPUs.

Figure 4.16a shows the post-map DFG for sub-DFG A, and it contains the two paths from input nodes $I_3$ and $I_4$ that join at node B. The overlay from the example is synthesized with D-FIFOs of depth 10 (D-FIFO-10) at each cell’s input port. The arrangement of other DDPUs within an overlay cell is the same as shown in Figure 4.12. The left path
Figure 4.16: A post-map DFG for the sub-DFG A from Figure 4.15. It demonstrates dynamic latency balancing of two routed paths from $I_3$ and $I_4$ that join at node $B$. 
(from \(I_3\)) hops through cell (1,1) and traverses through a D-FIFO-10 and a D-REG. It then enters cell (1,2) through a D-FIFO-10, passes through a D-SEB2 and then feeds the \textit{addsub} FU. The right path (from \(I_4\)) enters cell (1,2), passes through a D-FIFO-10 and a D-SEB2 and then feeds the FU. The post-map DFG also contains the source node \((S)\) which connects to input nodes \(I_3\) and \(I_4\) to denote that data is fed into the inputs of the DFG simultaneously.

Figure 4.16b shows the occupancy of each pipeline unit after 3 clock cycles. In this state, the first 3 data items have been fed into both paths. The minimum latency of the right path is 3 clock cycles: 2 clock cycles of D-FIFO-10 and 1 clock cycle of D-SEB2. Thus, at this point in time the data is valid at the FU’s right input (D-SEB2). However, the join logic stalls the right path’s D-SEB2 because the left one does not have valid data. Figure 4.16c shows the pipeline’s state one cycle later when the occupancy of the right D-SEB2 increases to 2 due to the stall; D-SEB2 is now full and thus propagates the stall upstream to D-FIFO-10. Figure 4.16d shows the pipeline’s state after 6 cycles. Since the minimum latency of the left path is 6 cycles, the data is now also valid at the FU’s left input. At this point, the join logic will unstack both D-SEB2 units feeding the FU. The right D-FIFO-10 is still stalled because the D-SEB2 is still full in this clock cycle.

Figure 4.16e shows the pipeline’s state after 7 clock cycles. In this state, the \textit{addsub} FU has received the first data item into its first pipeline stage. Due to a stall in the previous cycle, the right D-FIFO-10 occupancy increases to 5. The right D-SEB2 was unstacked in the previous cycle, thus decreasing its occupancy to 1 and unstacking the upstream D-FIFO-10. Thus, in this cycle no stalls are present in the pipeline, and data is available at both of the FU’s inputs. Additionally, we can observe in Figure 4.16f that the pipeline remains in the same state in the following cycle. This is the case because a new data item has been inserted into each path and in the same cycle the FU has consumed one data item from each path. Therefore, the maximum-throughput steady-state of the pipeline has been achieved wherein all units receive and send data every cycle.
thereby maintaining their occupancy and, thus, latency.

This example demonstrates how the difference in minimum latencies of the right and the left path (3 vs. 6 clock cycles) results in stalls of the right path for 3 clock cycles. Consequently, the latency of the right D-FIFO-10 increases from its minimum latency of 2 clock cycles to 5 clock cycles thereby increasing (or “stretching”) the latency of the right path to 6 clock cycles. Additionally, this manifests the capability of D-FIFO-10 to use its latency margin in order to achieve the latency balance of the two paths.

Next, we demonstrate latency balancing on an example with more than two paths. Figure 4.17 shows the post-map DFG of sub-DFG $B$ from Figure 4.15. While the previous example showed the step-by-step process of reaching a steady state execution, this example only shows the steady state; specifically, each DDPU in the post-map DFG in the figure is annotated with its steady state latency. We also show the latencies of the paths joining at the FUs, which are indicated by the dashed lines in the figure.

Three D-FIFO-10 units ($F_1$, $F_2$, $F_3$ - grey highlight in the figure) use their latency margin to balance the DFG’s paths. As was shown in the previous example, D-FIFO-10 $F_1$ (fed by $I_4$) has a latency of 5 cycles in order to balance the longer left path that joins at addsub FU ($B$). D-FIFO-10 $F_1$ (fed by $I_5$) has a latency of 8 in order to balance the left path joining at addsub FU ($C$). It should be noted that the latency of 8 cycles is required because the latency of $F_1$ was increased to 5, as described earlier. If $F_1$ itself was not used for balancing the leftmost path and remained at the minimum latency of 2 cycles, a latency of 5 for $F_2$ would be sufficient.

D-FIFO-10 $F_3$ balances two paths that fork at D-FIFO-10 $F_2$ and join at addsub FU ($D$). D-FIFO-10 $F_3$ is on the left path whose minimum latency is shorter (9 cycles) than the minimum latency of the right path (15 cycles). Thus, the latency of $F_3$ increases from 2 to 8 to match the latency of the longer right path. The other D-FIFO-10 units in the DFG do not use their latency margins and maintain the minimum latency of 2 cycles.
Figure 4.17: A post-map DFG for the sub-DFG B from Figure 4.15. Shown are the steady-state latencies of each pipeline unit and the latencies of paths joining at FUs.
The latency margins of the D-FIFOs on the overlay's routing paths eliminate the need for the DFG-to-overlay mapping algorithm to map joining DFG paths with equal minimum latencies. Maximum throughput can be achieved with any mapping in which the difference of the minimum latencies of joining paths is within the latency margins of D-FIFOs on the shorter path.

An additional mechanism is used when the latency imbalance exceeds the latency margin that can be accumulated on shorter paths. This mechanism involves adding one or more balancing nodes— which are nodes that perform no operation and are mapped to D-FIFOs— to the shorter DFG paths. These nodes effectively increase the latency margin that can be accumulated on shorter paths, thus enabling them to balance longer paths. We elaborate upon this mechanism in Section 5.7.

### 4.3 Overlay Reconfiguration, Clocking and Reset

#### 4.3.1 Reconfiguration Logic

The reconfiguration logic uses a special set of shift registers to store the overlay’s configuration bits, which we term the overlay bitstream. The reconfiguration logic also uses its own clock and clock enable signals. The reconfiguration logic uses a low-speed clock (100 MHz) since it is used less frequently than the main overlay logic and is not critical for performance. The paths between the bitstream registers and the registers within overlay’s cells have no bearing on overlay’s timing constraints (i.e., they are false paths). because the bitstream loading and overlay execution are mutually exclusive operations. Thus, it can be guaranteed that the signals on all of the paths from the overlay bitstream registers to the cell’s registers will have been stabilized at the moment the overlay starts DFG execution.

Figure 4.18 shows the overlay reconfiguration architecture for a 3×3 overlay. The overlay bitstream is comprised of row bitstreams, with one for each overlay row. The
Figure 4.18: An example $3 \times 3$ overlay with overlay reconfiguration control, reconfiguration memory with 2 overlay bitstream, the details of overlay cell reconfiguration logic (RECONF), and per-cell reset synchronizer (RS).

configuration bits for each cell are stored in a 1-bit wide shift register with 74 entries$^5$. The shift registers of cells in the same row are chained, thus forming the row bitstream. For a $3 \times 3$ overlay, each row bitstream is 222 bits long ($3 \times 74$). The row bitstreams are stored in the reconfiguration memory which is 3 bits wide with one bit for each row bitstream. Thus, the entire overlay bitstream is 222 3-bit words. The reconfiguration control loads the overlay bitstream into the overlay by reading one word from the reconfiguration memory per cycle, splitting it into bits and shifting the bits into row shift registers in parallel. Consequently, it takes 222 cycles to reconfigure a $3 \times 3$ overlay. In this example, the reconfiguration memory can store two overlay bitstreams, and hence has 444 words. The reconfiguration shift registers do not require resetting because the new bitstream is simply shifted-in, effectively overwriting the old bitstream. The overlay’s reconfiguration architecture is similar to a serial FPGA reconfiguration architecture (used in some FPGAs) [61] in the sense that they both use shift registers.

$^5$Sufficient for prototype overlays we designed and also leaves room for extending the FU functionality. We provide a breakdown later in the section.
The process of reconfiguring the overlay is as follows:

1. The overlay’s logic is put into a reset state (detailed in the next section).

2. The new overlay bitstream is shifted into the overlay as described earlier.

3. The reconfiguration control waits for 5 cycles to ensure that the configuration bits of each cell have propagated to the cell’s logic.

4. The overlay’s logic is brought out of reset.

5. The reconfiguration control waits another 10 cycles to ensure that all overlay cells have come out of reset.

6. The overlay is now ready for execution and data can be fed into it.

The 74-bit configuration bitstream of each overlay cell has the following components:

- Each output port DDPU has 7 bits for the fork receiver configuration mask (as described in Section 4.1.4). Two select bits of the valid mux are also used for the data mux.

- Each input port DDPU has 5 bits for the fork sender configuration mask.

- Each FU input DDPU (FU_{IN1}, FU_{IN2}) has 7 bits (same as the output ports)

- The FU output DDPU (FU_{OUT}) has 4 bits for the fork sender mask.

- The FU configuration is 8 control bits: 4 bits for FU_{sel} and 4 bits for FU_{op}. These bits are also used to determine the do-not-sync and always-stall bits for the join synchronization logic (as described in Section 4.1.4). Currently, 4 bits for the FU configuration would be sufficient for all of the FUs that were used in the overlays that we designed; 8 bits leave room for FU function extensions.
4.3.2 Overlay Clocking and Reset

The main overlay logic features two global signals: a high-speed clock (e.g., 400 MHz) and a reset signal \( ov_{clk} \) and \( ov_{reset.n} \), respectively in Figure 4.18).

As a recommended design practice, we use a synchronized asynchronous reset that asserts reset asynchronously but deasserts it synchronously through the use of a reset synchronizer [120, 123]. However, such a reset mechanism still requires a global signal. This results in long propagation delays to the registers of an overlay that spans the size of a large Stratix IV. Thus, it is necessary to expend extra effort in the design of the reset architecture in order to meet the recovery and removal timing constraints between the reset signal and the high-speed clock. To tackle this challenge we do two things. First, we employ a distributed reset synchronizers technique [123]. As shown in Figure 4.18, each overlay cell contains a 3-stage reset synchronizer (RS). The synchronizer takes the global reset signal \( ov_{reset.n} \) and generates the synchronized \( cell_{reset.n} \) signal that is only used within that overlay cell. As a result, each \( cell_{reset.n} \) signal is localized to an overlay cell, limiting its propagation delay. Second, we reset only those cell’s registers that carry valid and accept signals. In contrast, the cell’s data registers are not reset. This reduces the fan-out of \( cell_{reset.n} \), and hence improves its propagation delay within the cell. We are able to do this since values in data registers are never used for computation unless the valid register denotes that the data is valid.

While the above scheme meets the recovery and removal timing constraints locally within each cell, the propagation delays of the global \( ov_{reset.n} \) to different overlay cells can still vary. This variability does not pose an issue for putting the overlay into the reset state as we set the global \( ov_{reset.n} \) to low before starting the overlay reconfiguration. Since reconfiguration takes dozens of cycles, it is guaranteed that the global reset will have propagated to all the cells within that time frame.

However, releasing the global reset will result in overlay cells coming out of reset at different times because per-cell reset synchronizers may take different number of cycles
to come out of reset, and the propagation delay of the global $ov\_reset\_n$ may be different for different cells.

As described in the previous section, the overlay is brought out of reset after overlay reconfiguration has finished. We do this by first releasing the global reset and then waiting for another 10 cycles. Following this procedure ensures that all of the overlay cells have come out of reset. Once this has been done, the overlay is ready for execution and data can be fed into it.

4.4 Summary

In concluding this chapter, it would be useful to summarize the capabilities of the overlay architecture and the benefits it provides.

- **Distributed pipeline control.** There are no global control signals in the overlay (other than the clock and reset signals), which facilitates its *scalability to large mesh size*.

- **Latency insensitivity.** Any routable placement of a DFG on an overlay will execute correctly, i.e., latency balance is not required for correctness. This, in turn, provides two benefits:
  
  - **Programmer productivity.** DFG testing is possible on an overlay that is readily available but may not be able to balance the DFG for high performance.
  
  - **Bitstream portability.** A DFG that has been mapped to a given overlay functional layout is portable across overlays with that same functional layout but with different underlying elastic pipeline implementations. For example, it allows for changes in the latencies and the implementation of FUs and DDPUs as well as for changes in DDPU selection within the cell.
• **Overlay cell-level modularity.** Overlay cells are self-contained. This facilitates the use of *hierarchical physical design* techniques which can improve $f_{\text{MAX}}$ of the overlay, enable *bottom-up compilation* of the overlay, and enable *reuse* either at the level of single cells or tiles (groups of adjacent cells).

• **DDPU-level architectural flexibility.** The overlay cell architecture is flexible in how it allows for the selection of various DDPU types for the 11 DDPU locations within the cell. This gives rise to an architectural design space that can *trade-off between high $f_{\text{MAX}}$, high latency margin, and resource usage.*

• **Elastic pipeline and fine-grained stallability.** The use of elastic pipelines and synchronization logic is fine-grained; that is, *each FU and each overlay’s I/O port* can be stalled independently. All of the overlay’s pipeline units have their own flow control and thus pipeline units with no routing paths configured between them do not stall each other. Fine-grained stallability enables the following capabilities:

  - **Dynamic latency balancing.** Enables the use of elastic FIFOs to balance the overlay dynamically via stalls at the FUs. Routing paths are balanced by dynamically “stretching” the latency of the shorter paths during stalls.

  - **Variable latency and throughput of I/O.** The independent stallability of FUs and I/O ports enables them to interface with memory subsystems whose latency and throughput can vary at run-time (i.e., they are not statically predictable). Examples of such memory subsystems that may prove fruitful for exploration in future work include caches, banked memory with arbitrated access, and I/O streaming.

  - **Independent execution of unrelated DFGs.** Multiple unrelated DFGs can execute independently on the overlay. If one DFG stalls, the others can still execute at maximum throughput. Furthermore, these DFGs can “overlap”, meaning that they can use the routing resources of the same cell.
Chapter 5

DFG-to-Overlay Mapping Tool

5.1 Overview

The DFG-to-overlay mapping algorithm maps a DFG onto the overlay in a spatial manner. That is, each compute node in a DFG is mapped to a distinct FU in the overlay, and each DFG edge is mapped to a distinct routing path in the overlay to connect the mapped FUs. The goal of the algorithm is to find a mapping that achieves the highest possible throughput of the pipelined execution of the input DFG on the target overlay.

In broad terms, the mapping algorithm has three main components. The first component is the placement, which chooses the location of each DFG node in the overlay’s mesh. The second component is the routing, which finds routing paths between mapped FUs in the overlay. The third component is the latency balancing, which is comprised of a combination of heuristics and exact tests that ensure that a placed-and-routed mapping of a DFG on the overlay is latency balanced.

While the DFG-to-overlay algorithm performs the placement and routing components much like the FPGA CAD, it differs in the following respects: (1) the $f_{\text{MAX}}$ of a pre-compiled overlay is fixed, which means that it does not depend on which FUs are used or which routing paths are configured. Thus, no timing analysis is performed during
DFG-to-overlay mapping; (2) DFGs have 100’s of nodes, making the problem much smaller than that of the FPGA place-and-route algorithm, which maps circuits with up to a million nodes; and (3) the DFG-to-overlay algorithm automatically finds a latency-balanced DFG mapping, whereas the FPGA place-and-route algorithm does not modify, balance, or otherwise optimize the latencies of paths in the input circuit (this task is done by an HDL designer or an HLS tool).

The high-level pseudo-code of the DFG-to-overlay mapping algorithm consists of four parts and is illustrated in Figure 5.1.

The first part is the insertion of balancing nodes into a DFG (described in detail in Section 5.7). This DFG transformation heuristic is performed by \texttt{EstimateBalancingNodes} and \texttt{InsertBalancingNodes} before a DFG is mapped to the overlay. This part of the algorithm first estimates whether the latency margins of the overlay’s D-FIFOs are sufficient to balance the DFG. If the heuristic estimates that they are insufficient, it inserts one or more balancing nodes into each imbalanced DFG path. The subsequent place-and-route part of the algorithm maps the balancing nodes to the overlay’s D-FIFOs. This effectively increases the latency margins of those DFG paths, which in turn increases the likelihood that a balanced mapping will be found by the algorithm, albeit at a cost of additional routing resources.

The algorithm’s second part maps the DFG to the overlay. This is performed by the \texttt{PlaceRouteCheckBalanced} procedure (shown at the bottom of Figure 5.1). The body of the innermost loop performs a place-and-route (PAR) pass, which is a two-step algorithm that consists of a simulated-annealing-based placer and a PathFinder-based router. The placer (SA\_Placer) is an adaptation of VPlace [68, 124], and it aims to minimize the total routing hop count of the DFG. Similarly, we adapt a routability-driven version of the PathFinder [49] in order to minimize the total hop count and to maximize DFG routability (PF\_Router). At the beginning of the algorithm, the PathFinder’s search for a routable mapping is limited to 20 iterations of its internal loop (routings
1: procedure DFGtoOverlayMap(dfg, ov_fu_layout, ov_fifo_depth, start_seed, max_seeds, 
PF_router_iter_limit, PF_router_iter_inc)
2:  if EstimateBalancingNodesRequired(dfg, ov_fifo_depth) then
3:  DFG = InsertBalancingNodes(dfg, ov_fifo_depth)
4:  end if
5:  [routable, balanced, dfg_map_set, num_maps] = 
6:  PlaceRouteCheckBalanced(dfg, start_seed, max_seeds, overlay_fifo_depth, ov_fu_layout, 
PF_router_iter_limit, PF_router_iter_inc)
7:  if routable then
8:    if balanced then
9:       mapped_dfg = dfg_map_set[num_maps-1]
10:      else
11:         mapped_dfg = FindMostBalancedMapping(dfg_map_set, ov_fifo_depth, num_maps)
12:    end if
13:  end if
14:  overlay_bitstream = GenerateOverlayBitstream(mapped_dfg, overlay_FU_layout)
15:  end if
16:  return [routable, balanced, overlay_bitstream]
17: end procedure

22: procedure PlaceRouteCheckBalanced(dfg, ov_fu_layout, ov_fifo_depth, start_seed, max_seeds, 
PF_router_iter_limit, PF_router_iter_inc)
23:  seed = start_seed
24:  balanced = false
25:  i = 0; num_maps = 0
26:  while not balanced and i < max_seeds do  \ balancing loop
27:    routable = false
28:    while not routable and i < max_seeds do  \ PAR loop
29:      placed_dfg = SA_Placer(seed, dfg)
30:      [routable, mapped_dfg] = PF_Router(placed_dfg, ov_fu_layout, PF_router_iter_limit)
31:      seed = NextSeed(seed)
32:      i += 1
33:      if not routable then
34:         PF_router_iter_limit += PF_router_iter_inc
35:      end if
36:    end while
37:    if routable then
38:       dfg_map_set[num_maps] = mapped_dfg
39:       num_maps = num_maps + 1
40:    balanced = CheckBalanced(dfg, mapped_dfg, ov_fifo_depth)
41:  end if
42: end while
43: return [routable, balanced, dfg_map_set, num_maps]
44: end procedure

Figure 5.1: High-level pseudo-code of the DFG-to-overlay mapping algorithm.
The PAR pass is executed iteratively until a routable placement is found. This is achieved by performing a seed sweep, meaning that each run of the placer uses a different seed. The set of seeds is generated as a sequence based on the start_seed parameter to ensure deterministic behaviour and repeatability.

The placement algorithm uses the seed to initialize the random number generator, which is in turn used to generate the initial random placement as well as random placement moves within the process of simulated annealing. This method of using the seed within the placement algorithm is analogous to the method that is used in certain simulated-annealing-based placers for FPGAs [50, 51, 61].

Hard-to-route DFGs may require more PathFinder iterations to successfully route. Thus, the algorithm incrementally increases the number of routing iterations attempted by the PathFinder. If the DFG placement that is generated based on the current seed is not routable within the routing iteration limit, the limit is increased and the algorithm will try to route the next placement—generated based on the subsequent seed—using more routing iterations.

Once a routable placement is found, it is checked for latency balance (CheckBalanced) using an ILP-based algorithm. If the DFG mapping is balanced, the algorithm terminates; if it is not balanced, then the seed sweep is resumed. If a balanced mapping is not found within 40 PAR passes (i.e., 40 different seeds), the algorithm terminates.

The third part of the algorithm is performed only if a balanced mapping is not found in the previous step. The function FindMostBalancedMapping takes as input a set of routable DFG mappings (dfg_map_set), which are each obtained by a different placer seed in part two. The function then uses heuristics to determine which of the DFG mappings is the most balanced (i.e., likely to achieve the highest throughput) and then returns it as a result.

Finally, the fourth part of the algorithm generates the overlay bitstream for the pre-
viously obtained DFG mapping. This part is skipped if no routable mappings are found and an error is reported instead.

The place-and-route steps in the inner loop of the PlaceRouteCheckBalanced algorithm are fast due to the relatively smaller problem size and simple place-and-route heuristics, as described above. This allows us to perform a seed-sweep, which may otherwise be prohibitively expensive. Our experimental evaluation shows that this strategy is effective and results in fast compilation times as well as balanced DFG mappings.

In summary, a high-throughput DFG mapping is achieved as a result of: (1) the latency margins of the D-FIFOs; (2) inserting balancing nodes into highly imbalanced DFGs; (3) routing hop-count minimization heuristics of the placer and the router; (4) seed sweeping; (5) the overlay’s latency insensitivity and its attendant ability to execute a DFG at a suboptimal throughput; and (6) heuristics for determining the most balanced DFG mapping in cases where optimal balancing is not found.

In the remainder of this chapter we detail each part of the algorithm. We begin by describing the algorithm’s inputs and constraints in Section 5.2 before moving on to describe the placer in Section 5.3 and the router in Section 5.4. The latency balance checking algorithm is discussed in Section 5.5 and the balancing node insertion is described in Section 5.7. The heuristics used to find the most balanced mapping in a set of DFG mappings is discussed in Section 5.6.

5.2 Algorithm Inputs, Parameters, and Constraints

The DFG-to-overlay tool takes two inputs:

- The DFG description (dfg, discussed in Section 5.2.1).
- The overlay specification:
  - The overlay’s functional layout (ov_fu_layout, discussed in Section 5.2.2).
The depth of the overlay’s D-FIFOs (\texttt{ov\_fifo\_depth}, the default is 32).

The latencies of the FUs and the arrangement of overlay cell’s DDPUs are relevant for the latency balancing aspect of the algorithm. In its current implementation, the algorithm uses a fixed DDPU arrangement, which is shown in Figure 4.12. Furthermore, the algorithm assumes that the D-FIFOs are only at the input ports of the overlay’s cells and that all of the D-FIFOs have the same depth. The other DDPUs (D-REG, D-SEB1/2, D-BUF1/2) have a steady state latency of 1 clock cycle and no latency margin for latency balancing.

This aspect of the tool could be generalized by providing the DDPU arrangement as an external template (as a part of the overlay specification). We leave this extension of the tool for future work.

The DFG-to-overlay tool has several user-controllable parameters for tuning the behaviour of the algorithm.

- Starting seed (\texttt{start\_seed}). The default is 1. The value of 0 has a special meaning, and is used to generate a random number as a starting seed.

- Maximum number of PAR seeds (\texttt{max\_seeds}). The default is 40.

- PathFinder’s initial router iteration limit (\texttt{pf\_router\_iter\_init\_limit}). The default is 20.

- PathFinder router iteration limit increment (\texttt{pf\_router\_iter\_inc}). The default is 2. The routing limit increment can be disabled by setting this value to 0.

- PathFinder router iteration limit cap (not shown in the pseudo-code). The default is 200.

The tool also has a number of internal parameters that are not user-controllable. These parameters are described in the remainder of the chapter. One possible direction for future work is exposing some of these parameters to the user.
5.2.1 DFG Description

The DFG-to-overlay algorithm takes as input DFGs that are described using the *DOT graph description language* [125]. DFGs can also be described at a higher level in Python using the PyDFG API that we developed as part of this thesis. In this case, the DOT language representation of a DFG can be generated by executing the Python program that describes the DFG via the PyDFG API. A constraint of the mapping algorithm is that the input DFG must be a directed acyclic graph (DAG). The mapping algorithm allows for two types of DFG input nodes: *regular input nodes*, which represent streaming data whose value can change every cycle, and *constant input nodes*, which represent constant data that is loaded once into the overlay and does not change afterwards.

We demonstrate the key features of both descriptions through the two DFG examples shown in Figures 5.2a and d. The PyDFG description and the DOT description for the first DFG ($DFG_1$) are shown in Figures 5.2b and c. The compute nodes in this DFG perform integer operations.

In PyDFG, a regular (i.e., streamed) input node of a DFG is specified with the `in_node(node)` function, where `node` is a unique string identifier of the node (i.e., a node name). A compute node is specified with the `compute_node(node, op_code, src1_node, src2_node)` function, where `op_code` specifies the compute operation, while `src1_node` and `src2_node` represent two DFG nodes that feed data into the compute node (`node`). An output node is specified with the `out_node(node, src_node)` function, where `src_node` is the node that feeds the output node.

In the DOT language, a DFG is specified by first listing the DFG nodes and their attributes. As shown in Figure 5.2c, each node has `type` and `op` attributes which specify the FU type and operation. Once the FU types and operations have been specified, the DFG’s edges are listed (the construct $x \rightarrow y$ denotes an edge between the nodes $x$ and $y$).

Figures 5.2e and f show the PyDFG and DOT descriptions for $DFG_2$. The compute
from dfg_gen_lib import *

def simple_dfg_1(dfg_name):
    init_new_DFG()
    i1 = in_node("i1")
    i2 = in_node("i2")
    i3 = in_node("i3")
    i4 = in_node("i4")
    i5 = in_node("i5")
    a = compute_node("a", INT_MUL, i1, i2)
    b = compute_node("b", INT_SUB, i3, i4)
    c = compute_node("c", INT_ADDSUB, i4, i5)
    d = compute_node("d", INT_MUL, b, c)
    e = compute_node("e", INT_DIV, a, d)
    o1 = out_node("o1", e)
    o2 = out_node("o2", d)
    write_DFG_to_file(dfg_name)

(b) DFG 1 described in PyDfg

digraph {
    i1 [type=IN, op=IN];
    i2 [type=IN, op=IN];
    i3 [type=IN, op=IN];
    i4 [type=IN, op=IN];
    i5 [type=IN, op=IN];
    a [type=MULT, op=MUL, left_src=i1];
    b [type=ADDSUB, op=SUB, left_src=i3];
    c [type=ADDSUB, op=SUB, left_src=i4];
    d [type=MULT, op=MUL, left_src=b];
    e [type=DIV, op=DIV, left_src=a];
    o1 [type=OUT, op=OUT];
    o2 [type=OUT, op=OUT];
    i1 -> a; i2 -> a;
    i3 -> b; i4 -> b;
    i4 -> c; i5 -> c;
    b -> d; c -> d;
    a -> e; d -> e;
    d -> o2;
    e -> o1;
}

(c) DFG 1 described in DOT

def simple_dfg_2(dfg_name):
    init_new_DFG()
    i3 = in_node("i3")
    i4 = in_node("i4")
    i5 = in_const_node("i5")
    b = compute_node("b", FP_SP_SUB, i3, i4)
    c = compute_node("c", FP_SP_ADDSUB, i4, i5)
    d = compute_node("d", FP_SP_MUL, b, c)
    e = compute_node("e", FP_SP_MUL, i1, i2)
    o1 = out_node("o1", e)
    o2 = out_node("o2", d)
    write_DFG_to_file(dfg_name)

(e) DFG 2 described in PyDfg

digraph {
    i3 [type=IN, op=IN];
    i4 [type=IN, op=IN];
    i5 [type=IN, op=CONSTR];
    b [type=FP_SP_ADD_SUB, op=SUB, left_src=i1];
    c [type=FP_SP_ADD_SUB, op=SUB, left_src=i3];
    d [type=FP_SP_MUL, op=MUL, left_src=b];
    e [type=FP_SP_DIV, op="i_DIV_A"]; o1 [type=OUT, op=OUT];
    o2 [type=OUT, op=OUT];
    i3 -> b; i4 -> b;
    i4 -> c; i5 -> c;
    b -> d; c -> d;
    d -> e;
    d -> o2;
    e -> o1;
}

(f) DFG 2 described in DOT

Figure 5.2: (a)(d) Two example DFGs, (b)(e) described with PyDFG API, and (c)(f) described in DOT language.
nodes in this DFG perform single-precision floating-point operations. In contrast to \( DFG_1 \) where all compute nodes perform operations on streamed data, some compute nodes in \( DFG_2 \) perform operations on loaded constants and built-in constants. For example, the compute node \( C \) subtracts a loaded constant (input \( I_5 \)) from the streamed input \( I_4 \). In PyDFG, the loaded constant input \( I_5 \) is specified with the \texttt{in\_const\_node()} function. The \texttt{compute\_node()} function for the node \( C \) uses the \texttt{FP\_SP\_SUB\_CONST} opcode to denote that the second operand is a loaded constant. Furthermore, node \( E \) performs operation \( 1/a \) in which the second operand is a built-in constant. This is specified using the \texttt{FP\_SP\_1\_DIV\_A} opcode.

### 5.2.2 Overlay Functional Layout Description

The overlay’s functional layout is represented as a two-dimensional array. Figure 5.3a shows an example \( 4 \times 4 \) overlay and Figure 5.3b shows its functional layout described using a \( 6 \times 6 \) array. Two additional rows (on the top and bottom), and two additional columns (on the left and right) in the functional layout array specify the locations of the overlay’s I/O ports. The inner \( 4 \times 4 \) array describes the functional layout of the overlay’s cells. The N/A symbols at the corners of the array denote empty locations (i.e., no cell or I/O exists), and instruct the mapping tool to ignore them.

The routing-only cells are denoted as \texttt{NO\_FU}. A cell with multiple FUs is denoted by listing all FU types and delineating them with the “/” symbol. For example, the code \texttt{FP\_SP\_ABS/FP\_SP\_CMP} at location \( (2,2) \) describes a cell with the \texttt{f-abs} and \texttt{f-cmp} FUs. Note that only one DFG node can be mapped to an overlay cell, even though a cell may contain multiple FUs. This is explained in more detail in Section 5.3.

Similarly, the code \texttt{IN/OUT} denotes that an array location contains both an overlay input port and an overlay output port. In contrast to an overlay cell, two DFG nodes—one input and one output—can be mapped to an I/O location.

In Figure 5.3, overlay I/O ports entirely surround the overlay cell array. However, it
is possible to omit some I/O locations by marking them with N/A. In addition, an I/O location can have only an input port (IN) or only an output port (OUT). In general, the arrangement of I/O locations around the overlay cell array can be arbitrary.

5.3 Placer

The placement algorithm takes three inputs: (1) a DFG, (2) the overlay’s functional layout, and (3) a seed to initialize the random number generator used by simulated annealing.

The placement of DFG compute nodes to overlay cells is a one-to-one mapping: each compute node is mapped to a single overlay cell (a single FU), and only a single compute node can be mapped to each cell. The same holds for a cell with multiple FUs—only one compute node can be mapped to it—since only one of the cell’s FUs can be active at run-time.

As described in the previous section, the overlay’s I/O locations (IN/OUT) are treated differently: up to two nodes can be mapped to each I/O location— one DFG input node

Figure 5.3: (a) An example 4×4 overlay, (b) its functional layout represented with a 6×6 array.
and one DFG output node.

The role of the placer is to map DFG nodes to the locations within the overlay’s functional layout such that the cost function, which is defined below, is minimized. We use a simulated-annealing-based placer whose fundamentals have been covered in Section 2.1.4. Below, we first define the key placer concepts in the context of the overlay architecture, and we then describe the salient features of our placer, including the cost function, the move generation, and the annealing schedule.

5.3.1 Definitions

Analogous to a net in a digital circuit, we define a DFG net as a subset of DFG edges that have the same source node. We utilize the concept of a net for the heuristics within the placer that facilitate the close placement of same-net DFG nodes. This in turn enables the router to better share routing resources among the same-net DFG edges, thus minimizing the use of the overlay’s routing resources.

We next define the concepts of a routing hop, routing tree, and a routing path in the context of the DFG-to-overlay mapping, and we use the example in Figure 5.4 to illustrate them. The example shows a simple DFG with nine nodes and five DFG nets. For simplicity, the output nodes from compute nodes C, D, and E are omitted. The figure also shows an example placement-and-routing of the DFG to a 2×3 overlay. Net net_1 has four edges and is the largest net, net_5 has two edges, while the other three nets have only one edge.

A routing hop is the crossing of an overlay cell boundary, i.e., the pairing of an output port from one cell and an input port from an adjacent cell. A special case of a routing hop is a crossing between an overlay I/O port and a cell on the overlay’s boundary. Both cases are shown in Figure 5.4b.

A routing tree of a placed-and-routed DFG net is a tree whose root and leaf nodes are the overlay’s I/O ports and FUs. The inner nodes of the routing tree represent routing
Chapter 5. DFG-to-Overlay Mapping Tool

A routing path is a path in the routing tree that runs from its root to one of its leaf nodes. Each edge of a net corresponds to a distinct routing path in the net’s routing tree. A leaf node in the routing tree can be a root of another net’s routing tree if the two nets share a node (i.e., if a source node of one net is a sink node of another net). No inner nodes (i.e., routing hops) are shared between any two nets.

Figure 5.4b shows the routing trees of all five example nets, we describe two of them in detail. The routing tree of the two-edge net\textsubscript{5} is rooted in the overlay input port at location (0,2) and its leaf nodes are FUs at locations (1,2) and (1,3). The inner nodes are two routing hops (0,2)→(1,2) and (1,2)→(1,3). The four-edge net\textsubscript{1} has the largest routing tree and demonstrates the routing hop sharing among its four edges (i.e., its four routing paths). Functional unit A fans-out to two routing hops that are each shared by
two routing paths, meaning that each of two routing hops has a fan-out of two. Thus, in total, four distinct routing paths are formed.

5.3.2 Cost Function

The routing hop count of a mapped DFG net is the number of routing hops in its routing tree. It represents the amount of the overlay’s routing resources that are used to establish a net. For example, the routing hop counts of nets net\(_2\), net\(_5\) and net\(_1\) are 1, 2, and 4, respectively. The routing hop count of an entire mapped DFG is computed as the sum of the routing hop counts across all of the DFG’s nets, and it reflects the total amount of the overlay’s routing resources that are used to implement a DFG.

The goal of our placer is to find a placement that minimizes the DFG’s routing hop count. Since the routing is performed only after the placement is already determined, the placer uses heuristics to estimate the routing hop count of each net’s routing tree. We adapt the bounding box metric \([50, 69]\) to estimate the net’s routing hop count. The bounding box \(bb(net_i)\) of a net’s routing tree is the minimal-area rectangular section \(b_y \times b_x\) within the overlay’s functional layout that contains all of the routing tree’s nodes. For example, the net net\(_2\) has a bounding box with \(2 \times 1\) locations and net\(_5\) with \(2 \times 2\) locations. The net net\(_1\) has the largest bounding box with \(2 \times 3\) locations.

The placer uses the cost function \(cost(net_i) = b_y - 1 + b_x - 1\) to estimate the routing hop count of a mapped net net\(_i\) with a \(b_y \times b_x\) bounding box. This metric is also known as the half-perimeter wire-length metric \([50, 68, 124]\). The cost of a given placement of an entire DFG is the sum of costs across all nets. This metric is commonly used for FPGA PAR, and, thus, we have opted to adopt it.

5.3.3 Move Generation

As introduced in Section 2.3, a simulated annealing placer makes a series of random moves (or perturbations) to gradually improve the placement of a circuit. A generic
SA-placer targeting a typical column-based FPGA architecture can perform a move by selecting a random netlist element (e.g., a CLB) and then selecting a new random device location \((x_{new}, y_{new})\). The new location can be selected by first randomly selecting a device column \(y_{new}\) that contains CLBs, and then selecting a random location within the column \(x_{new}\). This is possible because column-based FPGA devices typically have only a few block types (CLB, DSP, BRAM) that are organized into homogeneous columns (only one block type per column).

In contrast, our design of the overlay’s functional layouts allows for the entirely heterogeneous arrangement of any number of FU types, which means that any FU type can be located at any \((x, y)\) location within the functional layout\(^1\). Thus, the overlay’s functional layouts do not have the column-based regularity and a few block types that are common features of FPGA devices. Consequently, a new random location \((x_{new}, y_{new})\) in the layout will likely not have the required FU type to make a move. For example, for an overlay with an 18×16 mesh layout and only 3 \(f\text{-}exp\) FUs, a randomly selected \((x_{new}, y_{new})\) location has a chance of “hitting” an \(f\text{-}exp\) of only 1% (the probability of a legal move).

The move generation is further complicated by the need to make a move to a location within a given distance \(d\) of the old location. A commonly used distance metric is the Manhattan distance \((d = dx + dy)\), where \(dx\) and \(dy\) are offsets from the old location in the \(x\) and \(y\) dimensions. For a low-density FU such as \(f\text{-}exp\), there may be no other \(f\text{-}exp\) FUs even within a large distance range (e.g., a quarter of mesh width and height).

We address the above challenges by designing a placer that does not assume any form of layout regularity and does not rely on random offsets \((dx, dy)\) from the old location to select a new location. Instead, our placer pre-computes a legal move list \((LML)\) for each \((x, y)\) location in the layout. If \(F_T\) is the FU type at location \((x, y)\), then the \(LML(x,y)\)

\(^1\)We later describe that, due to physical design considerations, the overlay layouts can be designed to facilitate the close physical placement of DSP-based overlay cells to the FPGA’s DSP columns. However, from the placer’s perspective the overlay’s columns are entirely heterogeneous.
will contain the locations of all FUs of this type in the overlay’s layout; that is, the LML records the legal locations to which a move can be made from \((x, y)\). The locations stored in the LML\((x,y)\) are sorted by their distance to \((x, y)\).

Furthermore, we pre-compute a distance index \((DI)\) for each LML. The distance index has \(H + W - 1\) entries (one for each possible distance) with \(H + W - 2\) being the longest Manhattan distance in the overlay’s functional layout. An entry at index \(d\) in the DI contains the number of elements \((N_d)\) within the LML whose distance to \((x, y)\) is less or equal to \(d\). Since the LML is distance-sorted, the overlay layout locations that are within the distance range \(d\) are found in the LML’s indices from 0 to \(N_d - 1\).

The LML and DI for each layout location are pre-computed at the beginning of the placer. Next, an initial random placement is computed. After this initial set-up, the following steps are performed to find a new location \((x_{\text{new}}, y_{\text{new}})\) that is within the range \(d\) of the old location \((x, y)\):

1. Look up the distance index \((N_d)\) in DI\((x,y)\) at position \(d\).
2. Generate a random number \((i)\) in the range \([0, \ldots, N_d - 1]\).
3. Look up LML\((x,y)\) at position \(i\) to obtain the new location \((x_{\text{new}}, y_{\text{new}})\).

Consequently, finding a new legal random location is an \(O(1)\) operation. This fast run-time is at the expense of memory resources that are required to store LMLs and DIs. The space complexity of the DI data structure is \(O(H \times W \times (H + W - 1))\) because there are exactly \(H + W - 1\) entries for each of the \(H \times W\) layout locations.

The upper bound on the space complexity of the LML data structure is \(O((H \times W)^2)\) because each LML can have up to \(H \times W\) entries and there are \(H \times W\) LMLs in total. For example, assuming that each LML entry is 4 Bytes, the memory usage of the LML data structure for a \(32\times32\) overlay will be 4 MB \((4 \times 32^4)\), and the memory usage for a \(128\times128\) overlay \((16384\ \text{cells})\) will be 1 GB. Therefore, the required memory resources are well within the capabilities of currently available workstations.
In order to complete a move to a new location (or a swap if another DFG node is already at the new location), the cost function must be updated. We use incremental bounding box update [50] for the DFG nets affected by the move. The time complexity of this operation is on average $O(1)$ [50]. Thus, the entire move operation—finding a new legal location and then updating the cost function—has an $O(1)$ time-complexity.

5.3.4 Annealing Schedule

The annealing schedule controls the parameters of simulated annealing. We designed the annealing schedule for the DFG-to-overlay mapping tool by adapting the VPlace annealing schedule [50, 68, 124]. We started with the parameters of VPlace and then empirically tuned them for use in the DFG-to-overlay mapping. In particular, we found that the temperature update schedule (which controls how the temperature $T$ is varied throughout the annealing process) and the number of moves attempted at each temperature were suitable for the DFG-to-overlay mapping; thus, we elected to adopt them. In addition, we tuned the $R_{limit}$ (the allowed range for a random move) and the ExitCriterion (when the anneal should terminate) for the DFG-to-overlay mapping. Since a discussion of the background and use of these annealing parameters has already been provided in Section 2.1.4, the remainder of this section will be concerned with detailing the annealing schedule adaptation.

The main value that is monitored during the anneal is the ratio ($\alpha$) between the number of accepted moves and the total number of attempted moves at a given temperature $T$. At the start of the anneal when temperature is high, $\alpha$ tends to be close to 1 (almost all moves are accepted); however, $\alpha$ gradually converges to 0 as the temperature is lowered (no moves are accepted). Previous work [68, 126] has determined that it is beneficial to keep $\alpha$ close to 0.44 for as long as possible during the anneal. VPlace uses a temperature update schedule that attempts to increase the amount of time spent at temperatures for which ($\alpha$) is in the vicinity of 0.44. We found that such a schedule is also...
suitable in the context of DFG-to-overlay mapping, and we describe the key mechanism below.

A new lower temperature ($T_{\text{new}}$) is obtained by multiplying the old temperature ($T_{\text{old}}$) by the coefficient $\gamma$: $T_{\text{new}} = \gamma T_{\text{old}}$. The $\gamma$ coefficient is a function of $\alpha$ [68, 124], which is shown in Table 5.1.

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\gamma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha &gt; 0.96$</td>
<td>0.5</td>
</tr>
<tr>
<td>$0.8 \leq \alpha \leq 0.96$</td>
<td>0.9</td>
</tr>
<tr>
<td>$0.15 \leq \alpha \leq 0.8$</td>
<td>0.95</td>
</tr>
<tr>
<td>$\alpha \leq 0.15$</td>
<td>0.8</td>
</tr>
</tbody>
</table>

Table 5.1: The temperature update schedule $\gamma(\alpha)$ function; this schedule is the same as in [68, 124].

Betz and Rose [68, 124] also discovered that the annealer is not extremely sensitive to the exact values of the $\gamma(\alpha)$ function. The key reason for this is the appropriate form of the function: $\gamma$ is close to 1 (0.95 in this case) for $\alpha$ in the interval around 0.44 (in this case $[0.15 - 0.8]$), and significantly smaller outside of this interval, i.e., when $\alpha$ comes close to 0 and 1.

Our schedule for updating $R_{\text{limit}}$ differs from that of VPlace. We adopt the same form as the $R_{\text{limit}}$ update schedule of VPlace (shown in Figure 5.5), but we also tune the values to obtain the adapted schedule for the overlay (shown in Figure 5.6).

$$R_{\text{limit}}^{\text{init}} = \text{max FPGA dimension} \quad (5.1a)$$
$$R_{\text{limit}}^{\text{new}} = R_{\text{limit}}^{\text{old}}(1 - 0.44 + \alpha) \quad (5.1b)$$
$$1 \leq R_{\text{limit}} \leq \text{max FPGA dimension} \quad (5.1c)$$

Figure 5.5: VPlace $R_{\text{limit}}$ update schedule [68, 124].

The key difference is the threshold value of $\alpha$ at which the $R_{\text{limit}}$ begins to decrease. VPlace uses the threshold of 0.44, and thus, when $\alpha$ is less than 0.44, the $R_{\text{limit}}$ decreases. We empirically determined that a threshold value close to 0.2 is more suitable for the
\[ R_{\text{limit}}^{\text{init}} = \max \text{ Manhattan dist.} = H + W - 2 \]  \hspace{1cm} (5.2a)  
\[ R_{\text{limit}}^{\text{new}} = R_{\text{limit}}^{\text{old}} (1 - 0.2 + \alpha) \]  \hspace{1cm} (5.2b)  
\[ 4 \leq R_{\text{limit}} \leq \max \text{ Manhattan dist.} \]  \hspace{1cm} (5.2c)

Figure 5.6: Adapted \( R_{\text{limit}} \) update schedule for DFG-to-overlay mapping.

DFG-to-overlay mapping as it delays the decrease of the \( R_{\text{limit}} \) to a later stage of the annealing process (only when \( \alpha \) drops below 0.2). Furthermore, we set the lower bound on the \( R_{\text{limit}} \) to 4.

The reasons for adapting the schedule heuristics in Table 5.6 are as follows. Overlay layouts are more heterogeneous (more FU types) than FPGA layouts, and thus the density of individual FU types is lower than that of the CLBs in FPGA layouts. For example, the density of an \( f\text{-div} \) FU may be 1/9 (one \( f\text{-div} \) per 3\times{}3 tile of cells), and an \( f\text{-log} \) FU may have an even lower density (e.g., 1/20). As a result, there are often no legal moves in the immediate neighbourhood of a given FU. In order to allow the annealer to find legal moves in most cases, it is desirable to maintain a high \( R_{\text{limit}} \) value throughout a large temperature range and to not decrease the \( R_{\text{limit}} \) to 1.

Furthermore, a DFG may be I/O dominated, i.e., it may have a large fraction of I/O nodes. Since an overlay’s I/O ports are on its periphery, moving an I/O node from one side of the overlay to another requires a large \( R_{\text{limit}} \). A small \( R_{\text{limit}} \) only allows an I/O node to be shifted along the same side. Although the \( R_{\text{limit}} \) starts at the maximum Manhattan distance in the overlay, we empirically found that maintaining the maximum \( R_{\text{limit}} \) throughout a large temperature range allows the annealer to search through a larger design space of placements. This in turn helps to find routable mappings for I/O-dominated DFGs.

To terminate the anneal (\( \text{ExitCriterion} \)), we incorporate an extra condition in addition to the one used by VPlace. If either condition is met, the anneal is terminated \([68, 124]\). We monitor the ratio (\( \beta \)) between the number of moves that improve the placement
and the total moves attempted at a given temperature, and we stop the anneal when \( \beta \) is less or equal to a pre-set threshold at two consecutive temperatures. We empirically found that this threshold can be as low as 0. For example, in the case of a 100-node DFG, after approximately 100 temperatures there will be no moves that improve the placement.

Finally, we compute the initial temperature and the number of moves per temperature using the same methods as VPlace \([68, 124]\).

### 5.4 Router

The design goal of the routing algorithm is the use of heuristics that maximize the likelihood of successfully routing the placement of a DFG (as produced by the placer). In this section, we first describe our adaptation of the PathFinder routing algorithm, and we then discuss the heuristics that control how many routing iterations the routing algorithms performs.

#### 5.4.1 PathFinder Adaptation

We adapt the routability-driven version of the PathFinder \([49]\) routing algorithm. The algorithm uses a heuristic that suggests that a DFG placement will have a higher likelihood of being routed successfully if the total routing hop count of the entire DFG is minimized during the routing process.

The PathFinder algorithm minimizes the routing hop count of the DFG by minimizing the routing hop count of each net’s routing tree. This is achieved by employing heuristics that incrementally build the net’s routing tree one sink of a net at a time. A discussion on the background of the Pathfinder and routing hop count minimization heuristics has been presented in Section 2.1.4. We empirically found that these heuristics work well for minimizing the hop count in the overlay’s 4-NN routing topology, and we show several
examples of routing trees that demonstrate this approach in Figure 5.4.

The router first creates a routing-resource graph (RRG) for the overlay’s functional layout. Each routing hop—which consists of a pair of output-input ports from two adjacent cells—is a node in the graph. There are two special types of nodes in the RRG: source nodes, which are any nodes with an in-degree of 0; and sink nodes, which are any nodes with an out-degree of 0. For each FU, we create one source node to represent the FU output and one sink node to represent both of the FU’s inputs. Furthermore, we also create a source node for each overlay input port and a sink node for each overlay output port.

When a DFG net is routed in this RRG, the result is a routing tree that begins at one of the RRG sources, i.e., the routing tree root is always mapped to an RRG source. Similarly, the leaf nodes of the routing tree are mapped to a set of RRG sinks.

Each node \( n \) in the RRG is assigned a capacity \( (cap_n) \), which refers to the maximum number of nets that can use that node. All RRG nodes have a capacity of 1; FU sink nodes are an exception to this rule and have a capacity of 2. A post-routing processing step determines which one of the two nets using the FU sink node connects to FU\textsubscript{IN1} and which one connects to FU\textsubscript{IN2}. The routing algorithm also keeps track of node occupancy \( (occ_n) \), or how many nets are currently using that node.

We adopt the cost function which is stated formally as: 
\[
c_n = b_n h_n p_n \] [124],
where \( c_n \) is the overall cost of using an RRG node \( n \), \( b_n \) is the base cost of using the node, \( h_n \) is the node’s historical congestion penalty, and \( p_n \) is the node’s present congestion penalty. We treat all of the overlay’s routing resources as equal and set \( b_n \) to 1 for all RRG nodes. The functions that compute \( p_n \) and \( h_n \), as well as the range for the routing schedule values, are borrowed from previous work [124]. Specifically, we use 0.5 for the initial value of \( p_{fac} \), 1.5 for the \( p_{fac} \) multiplier, and 0.2 for \( h_{fac} \).
5.4.2 Controlling PathFinder’s Routing Iteration Limit

The process of routing all of the DFG’s nets is referred to as a *routing iteration*. If any of
the routing nodes are overused, or congested \((occ_n > cap_n)\), after one routing iteration,
node congestion penalties \((p_n\) and \(h_n)\) are adjusted and the routing iteration is re-run.
This is performed iteratively until a congestion-free routing is achieved.

If a congestion-free routing (i.e., a DFG is routable) is possible after only one routing
iteration, then all of the DFG’s nets will be routed using the shortest paths allowed by
the underlying DFG placement. The more routing iterations that are required, the longer
the routing paths that pass through the congested regions of the overlay will be.

In FPGA router implementations, it is common for the number of allowed routing
iterations to be limited to a pre-set value (e.g., 45) [50, 124]. If the circuit is not routed
within this limit, it is classified as unroutable.

Having established the concepts of routing iterations and routability, we proceed in
this section by discussing two approaches based on a fixed routing iteration limit, and
then by describing an adaptive approach that we use in this work. The first approach
is to use a low routing iteration limit (e.g., 25) in a fashion similar to FPGA routers.
The placements of some DFGs may always be routable within the limit, whereas only
a fraction may be routable within the limit for others. As a result, the disadvantage of
the low-limit approach is that some DFG placements are classified as unroutable even
though they could be routed with more iterations. On the other hand, the advantage
of this approach is that placements that are not at all routable are rejected relatively
quickly (within 25 iterations).

The second approach is to set the routing iteration limit to a large number (e.g.,
200). Unlike the low-limit approach, this approach can prevent hard-to-route DFGs (or
hard-to-route DFG placements in general) that may require many routing iterations from
being classified as unroutable. Thus, the advantage of the high-limit approach is that
any placement of a given DFG that is at all routable is likely to be routed. However,
for some DFGs, a large portion of their placements (generated by different seeds) may simply be unroutable. For example, if on average two thirds of the placements of a given DFG are not at all routable, then two thirds of router runs will execute all 200 routing iterations before classifying the DFG as unroutable. As a result, a considerable amount of algorithm run-time can be spent on unroutable DFG placements.

We propose an approach that combines the advantages of both of these approaches described above. It does so by adapting the routing iteration limit during the algorithm’s run-time. The initial routing iteration limit is set to a low value of 20. Anytime the algorithm fails to route a DFG placement within the current routing iteration limit, we increment the limit by a pre-set value (we use 2). The next DFG placement (using the next seed) is routed with the updated routing iteration limit. Since the algorithm uses up to 40 placements (i.e., seeds), the routing iteration limit can increase up to 100. On the one hand, this allows the router to adapt to harder-to-route DFGs that require a larger number of routing iterations; on the other hand, since the algorithm starts off with a low limit, it does not waste routing iterations when a large portion of DFG placements are not at all routable.

5.5 Latency Balance Checking

A balanced DFG mapping (i.e., placement-and-routing) can be achieved even if joining DFG paths are not mapped with equal minimum latencies (the sum of the minimal latencies of traversed DDPU units). A balanced mapping is achieved if the difference between the minimum latencies of joining paths falls within the latency margin provided by the elastic D-FIFOs on the shorter path.

Without D-FIFOs, a balanced DFG mapping would be harder to achieve because all joining DFG paths would have to be mapped with exactly equal minimum latencies. This may be possible if both the overlay’s functional layout and its routing topology are
customized and tuned to match the DFG itself. The use of D-FIFOs enables DFGs to be balanced on a generic routing topology such as 4-NN.

We develop an algorithm that performs an exact test to determine whether a post-map DFG (i.e., the placement-and-routing of the DFG on the target overlay) is latency balanced. The inputs to the algorithm are: a DFG, its post-map DFG, and the depth of the overlay’s D-FIFOs \(d_{OV}\). The output of the algorithm is a binary decision: either the post-map DFG is balanced, or it is not.

Our algorithm is an adaptation of the ILP-based latency balancing algorithm described in Section 2.5, which formulates an ILP program that computes the buffer depths that need to be inserted into each edge in order for the DFG to be balanced. In contrast, instead of inserting buffers of arbitrary depth, our ILP algorithm tests whether the mapped D-FIFOs—those that were traversed by the routing paths of the mapped DFG—provide enough latency margin to balance joining paths. In other words, our algorithm only tests whether the ILP model has a solution or not.

The key aspect of our algorithm is the formulation of a bounded ILP problem in which the ILP variables representing the unknown latencies of mapped D-FIFOs are bounded. Specifically, the unknown latency of a D-FIFO is bounded by its minimum and maximum possible latency. Our algorithm first creates a balancing DFG by annotating the nodes of the input DFG with known values (fixed latencies of FUs) and the edges with known values (traversed DDPUs with fixed latencies). The edges are also annotated with ILP variables that represent the unknown latencies of the traversed D-FIFO. These above known values and variables are obtained from the post-map DFG.

We demonstrate our formulation using the example post-map DFG shown in Figure 4.17. This post-map DFG is obtained by mapping the sub-DFG B from Figure 4.15a to the overlay in Figure 4.15b. The corresponding balancing DFG that is created by our balance checking algorithm is shown in Figure 5.7. Each compute node \(u\) is assigned a known value \(T_u\) that represents the latency of the overlay’s FU to which node \(u\) is mapped.
Figure 5.7: Balancing DFG created for the DFG B from Figure 4.15a and its post-map DFG in Figure 4.17.
to. Each edge \((u,v)\) is assigned a known value \(w_{uv}\) that represents the fixed latency component accumulated of the overlay’s routing path to which edge \((u,v)\) is mapped to. A fixed latency of 1 is contributed by each D-REG and each 2-slot EB DDPU on the routing path. This is because we are solving for the maximum-throughput steady-state in which these DDPU types have a latency of 1 clock cycle. For example, the edge \(w_{CD}\) has a fixed latency of 4. We obtain this number by traversing the corresponding routing path in the post-map DFG (Figure 4.17) and then adding a latency of 1 for each of two D-REGs, one D-BUF2, and one D-SEB2.

Furthermore, a DFG net \(net_u\) (with node \(u\) as source) whose routing tree traverses \(K_u\) D-FIFOs is associated with a set of unknown variables \(B_u = \{b_i^u \mid i \in \{1, \ldots, K_u\}\}\), where each \(b_i^u\) represents the latency of one D-FIFO. As illustrated in Figure 4.17, the routing tree of \(net_{I_4}\) traverses two D-FIFOs: \(F_1\) and \(F_4\). Thus, \(K_{I_4}\) equals 2 and the associated set of variables is \(B_{I_4} = \{b_1^{I_4}, b_2^{I_4}\}\).

For each edge \((u,v)\) of \(net_u\) we assign a set \(B_{uv}\) (a subset of \(B_u\), \(B_{uv} \subseteq B_u\)) that contains variables representing D-FIFOs that are traversed by the routing path corresponding to that edge. For example, edge \((I_4, C)\) is assigned a set with two variables \((B_{I_4C} = \{b_1^{I_4}, b_2^{I_4}\})\) since it traverses both \(F_1\) and \(F_4\), while \((I_4, B)\) is assigned a set with only \(b_1^{I_4}\) because it traverses only \(F_1\).

The total latency \(L_{uv}\) of an edge \((u,v)\) is a sum of its known fixed latency and unknown variables that represent the latencies of the D-FIFOs mapped to that edge:

\[
in \text{general, edge } (u,v) : L_{uv} = w_{uv} + \sum_{i \in B_{uv}} b_i^u \quad (5.3a)
\]

\[
e.g., \text{ edge } (I_4, C) : L_{I_4C} = w_{I_4C} + b_1^{I_4} + b_2^{I_4} \quad (5.3b)
\]

\[
e.g., \text{ edge } (I_4, B) : L_{I_4B} = w_{I_4B} + b_1^{I_4} \quad (5.3c)
\]
Chapter 5. DFG-to-Overlay Mapping Tool

$D_v$). Edge equations for the node $v$ then have the following form:

for each incoming edge $(u, v)$ of node $v$:

$$D_v = L_{uv} + T_u + D_u$$  \hspace{1cm} (5.4a)

e.g., for node $B$ and edge $(I_3, B)$:

$$D_B = L_{I_3B} + T_{I_3} + D_{I_3}$$  \hspace{1cm} (5.4b)

e.g., for node $B$ and edge $(I_4, B)$:

$$D_B = L_{I_4B} + T_{I_4} + D_{I_4}$$  \hspace{1cm} (5.4c)

Note that $L_{uv}$ variables are used here to achieve a more concise presentation of the edge equations. In the actual formulation of the model, we do not use $L_{uv}$ variables and write edge equations directly with $w_{uv}$ and $b_{i_u}$.

In our ILP model, D-FIFO latencies are bounded. The minimum possible latency of a D-FIFO in a maximum-throughput steady-state is 2, and the maximum possible latency is $d_{OV} - 1$. Thus, we set the two model constraints for each D-FIFO to bound the corresponding variable $b_{i_u}^i$:

$$\forall u \in V \text{ and } \forall i \in \{1, ..., K_u\} : 2 \leq b_{i_u}^i \leq d - 1$$  \hspace{1cm} (5.5a)

Subject to these bound constraints and edge equations, our ILP model seeks to find a solution that minimizes the sum of all $b_{i_u}^i$:

$$\text{minimize: } \sum_{u \in V} \sum_{1 \leq i \leq K_u} b_{i_u}^i$$  \hspace{1cm} (5.6a)

While this objective function minimizes latencies across all $b_{i_u}^i$, for the purpose of the latency balance testing we only care whether the ILP model has a solution or not. However, by minimizing the sum across all $b_{i_u}^i$, the model also computes the minimum steady-state latency through the DFG. In other words, the solution computes the minimum latency of 2 for the D-FIFOs that are on the longest of the paths that join at a node (i.e., the path for which the sum of fixed latencies and minimum D-FIFO latencies is the largest). The model computes a latency larger than 2 only for the D-FIFOs that are on
shorter paths that need to be “stretched” to balance the longest path joining at a given node. As a result, such formulation directly minimizes the latencies of paths through the DFG, and hence the overall DFG latency.

This model matches the steady-state latencies of the overlay’s D-FIFOs that will be achieved at run-time during pipelined DFG execution. Since data on a shorter path will arrive at a join node before data on a longer path, the shorter path will stall—thereby increasing the latency of its D-FIFOs—until the data on the longer path also arrives at the join node. Afterwards, there are no stalls and the maximum throughput is achieved. Hence, the latency of the longer path is not increased and the latency of its mapped D-FIFOs will remain at 2.

The unknown variables $D_u$ can be eliminated before solving the ILP model since they are linear combinations of $b^i_u$ variables. Eliminating these variables reduces the number of unknown variables by $|V|$. We eliminate these variables by first writing a direct form of edge equations for a node $v$, and then replacing each $D_u$ with any of its edge equation expressions. This process is done recursively until the DFG’s input nodes have been reached. In this example, the method for node $D$ is as follows:

write the direct form of the edge equations for node $D$:

$$L_{BD} + T_B + D_B = L_{CD} + T_C + D_C$$

(5.7a)

then replace $D_B$ and $D_C$ with edge expressions:

$$L_{BD} + T_B + L_{I_3} B + T_{I_3} + D_{I_3} = L_{CD} + T_C + L_{I_4} C + T_{I_4} + D_{I_4}$$

(5.7b)

since $D_{I_3} = 0, D_{I_4} = 0, T_{I_3} = 0, T_{I_4} = 0$, it simplifies to:

$$L_{BD} + T_B + L_{I_3} B = L_{CD} + T_C + L_{I_4} C$$

(5.7c)

The final ILP model has a variable for each traversed D-FIFO. If each DFG edge traverses $h$ D-FIFOs on average (equal to the number of traversed routing hops), then there are $h |E|$ model variables. There are $2h |E|$ constraints that bound the D-FIFO
latencies and there are $|E|$ edge equations. Hence, the model has $2h|E| + |E|$ constraints in total.

If the ILP model finds a solution that satisfies the edge equations and bounding constraints, it then returns integer values for each $b^i_u$. We can then also derive the values of $D_u$ variables and the steady-state DFG latency, which is the value of any $D_{O_k}$, or the latency from any of the DFG input nodes to any of the DFG output nodes ($O_k$).

We use an MILP solver $1p\_solve$ [127] to formulate and solve the balancing ILP model. The CheckBalanced procedure in Figure 5.1 first creates the balancing DFG before setting the ILP constraints using the $1p\_solve$ API. Finally, it invokes the $1p\_solve$ solver. If a solution exists, the CheckBalanced procedure computes the steady-state DFG latency and returns a positive result. Otherwise, a negative result is returned.

### 5.6 Finding the Most Balanced Mapping of a DFG

A key capability of the overlay architecture is its latency insensitivity: a mapping of a DFG to an overlay will result in correct execution as long as the placed DFG can be routed successfully. The latency-balancing aspect of the mapping algorithm is required only for high performance and not for functional correctness.

Even if a latency-balanced mapping (i.e., maximum throughput) cannot be found, a DFG can still be mapped with high throughput (e.g., 80% of the maximum) by finding a mapping that is close to being balanced.

The DFG-to-overlay mapping algorithm performs seed-sweeping and generates DFG placements until it finds one that is both routable and balanced. The balance checking of a post-map DFG (i.e., placed-and-routed DFG) is performed by an exact algorithm (CheckBalanced in Figure 5.1) that was described in the previous section. In the event that a balanced mapping is not found among $num\_seeds$ DFG placements, the algorithm will have generated $num\_mappings$ post-map DFGs (i.e., the routable subset
of all *num_seeds* DFG placements).

Each of the *num_mappings* (*M*) post-map DFGs achieves a certain level of throughput that is below maximum, i.e., they are not balanced. We develop a heuristic algorithm that estimates which post-DFG in the set is the most balanced and likely to achieve the highest throughput. A post-map DFG *g_i* that is not balanced for the target overlay’s D-FIFO depth (*ov_fifo_depth* or *d OV*) may be balanced if the overlay’s D-FIFO depth is larger than *d OV*. We introduce a balancing *D-FIFO depth* (*d_i*), which is the minimum D-FIFO depth that balances a post-map DFG *g_i*.

### 5.6.1 Algorithm Heuristics

The approach we use in the algorithm is to first compute the balancing D-FIFO depth for each post-map DFG in a generated set of post-map DFGs (*G* = \{*g_1*, ..., *g_M*\}). The result of this computation is a set of balancing D-FIFO depths, *D* = \{*d_1*, ..., *d_M*\}. Our heuristic then estimates that the most balanced post-map DFG will be the one with the smallest balancing *D-FIFO depth* in the set. In this section, we present one example that illustrates the motivation for using this heuristic, and we construct another example that demonstrates that our algorithm is indeed a heuristic and does not always yield an optimal solution.

Let us assume that an overlay has a D-FIFO depth of 32 and that four DFG mappings have been generated (*G* = \{*g_1*, *g_2*, *g_3*, *g_4*\}). Furthermore, let us assume that none of the mappings are balanced for a D-FIFO depth of 32, and that their balancing D-FIFO depths are *D* = \{101, 35, 80, 40\}. Our heuristic picks the mapping with the smallest balancing depth (*g_2*, *d_2* = 35). It is guaranteed that, if the overlay’s D-FIFO depth is 35, then *g_2* will execute at maximum throughput. Since the other post-map DFGs have a larger balancing depth, it is also guaranteed that they will not be balanced at a D-FIFO depth of 35. Thus, their throughput will be less than that of *g_2*.

In actuality, the D-FIFO depth of the overlay is 32 and *g_2* may not have the highest
throughput at that depth. It is very likely that it has higher throughput than \( g_1 \) and \( g_3 \) whose balancing depths are much larger than that of \( g_2 \) (101 and 80 vs. 35). However, \( g_4 \)'s balancing depth (40) is relatively close to 35, and it may thus have a higher throughput at the target D-FIFO depth of 32. This scenario is possible if \( g_4 \) has a smaller relative latency imbalance than \( g_2 \).

For example, assume that in \( g_4 \) a shorter path and a longer path are balanced at a latency of 300 cycles when the D-FIFO depth is equivalent to \( g_4 \)'s balancing depth (40). But, if the D-FIFO depth is 32 (8 slots smaller than the balancing depth), then the shorter path will have a latency of 292 cycles. Thus, stalls will occur in 8 out of 300 cycles resulting in a throughput that is 97% of the maximum.

Furthermore, assume that in \( g_2 \) a shorter path and a longer path are balanced at a latency of 50 cycles when the D-FIFO depth is the same as \( g_2 \)'s balancing D-FIFO depth (35). However, if the D-FIFO depth is 32 (3 slots smaller than the balancing depth), then the shorter path will have a latency of 47 cycles. This will lead to 3 stalls every 50 cycles and a throughput that is 94% of the maximum.

These examples demonstrate that, while our heuristic may not always select the optimal post-map DFG, it will select a post-map DFG that achieves a result that is close to the highest possible throughput within the post-map DFG set.

### 5.6.2 Finding the balancing D-FIFO depth of a post-map DFG

Finally, we describe the algorithm that finds the balancing D-FIFO depth \( d \) of a post-map DFG \( g \) (FindBalancingDepth). Our algorithm leverages the fact that \( d \) is the minimum D-FIFO depth that balances the post-map DFG. Consequently, a given D-FIFO depth \( (l) \) that is smaller than \( d \) will not balance the post-map DFG, only a larger depth will do that. A call to the CheckBalanced(DFG, G, L) function returns \texttt{false} if \( l \) does not balance the post-map DFG (thus \( l \) is smaller than \( d \)) or \texttt{true} if \( l \) balances it (thus \( l \) is equal or larger than \( d \)).
This property of \texttt{CheckBalanced(dfg, g, l)} enables the use of a binary search of the D-FIFO depths in the interval \([d_{OV} + 1, 1024]\) in order to find \(d\). We iteratively call \texttt{CheckBalanced(dfg, post\_map\_dfg, l)} where \(l\) is the interval median. If the result is \texttt{false}, the upper half of the interval is searched, otherwise the lower half is searched. It is guaranteed that \(d\) will be found in 10 invocations of the \texttt{CheckBalanced} function \((\log_2 1024)\). The \texttt{FindBalancingDepth} function is invoked for each post-map DFG. Since there can be up to 40 post-map DFGs (40 seeds), there will be no more than 400 invocations of \texttt{CheckBalanced} in total.

The number of invocations can be further reduced by progressively lowering the upper bound that is passed to the \texttt{FindBalancingDepth} function. The initial upper bound of the first \texttt{FindBalancingDepth} invocation (i.e., for the first post-map DFG) is 1024. We use \(d_{\text{best}} - 1\), where \(d_{\text{best}}\) is the smallest balancing depth found so far, as the upper bound for the remaining post-map DFGs because the goal of the algorithm is to iteratively improve \(d_{\text{best}}\). Hence, searching D-FIFO depths larger than \(d_{\text{best}}\) for subsequent post-map DFGs is unnecessary.

### 5.7 Balancing Node Insertion

There are two sources of the latency imbalance of a mapped DFG: 1) the input DFG itself may be latency imbalanced; and 2) the placement and routing on the overlay can cause DFG imbalance. The overlay’s elastic D-FIFOS provide latency margins that can neutralize latency imbalance to a certain extent, but in some cases the imbalance may be large enough that the latency margins accumulated on shorter DFG paths may not be sufficient to balance them with longer paths.

We begin this section by discussing both the challenges that arise when there are two sources of imbalance and the motivation for inserting special balancing nodes into the DFG. Once this has been outlined, we then present the details of the algorithm that
transforms the input DFG via the insertion of balancing nodes.

5.7.1 Motivation and Challenges

Let us consider an example input DFG with a pair of joining paths: 1) a shorter 2-node path whose FU-only latency is 40 cycles (counting only the latencies of the FUs), and 2) a longer 12-node path whose FU-only latency is 240 cycles. Furthermore, let us assume that the target overlay has D-FIFOs with a depth of 32, and that the longer path is mapped to the overlay with one routing hop for each DFG edge (i.e., in the shortest manner possible). Therefore, the minimum possible latency of the longer path on the overlay is 300 cycles: 240 cycles for the 12 FUs, 24 cycles for D-SEB2 and D-BUF2 surrounding these FUs, and 36 cycles as the minimum latency of 12 hops (2 cycles minimum for each D-FIFO and 1 cycle for each D-REG).

Since the goal of the place-and-route algorithm is to minimize hop counts across all paths, it may also minimally map the shorter 2-node paths by using 2 hops. In that case, the maximum possible latency of the shorter path is 108 cycles: 40 cycles for the FUs, 4 cycles for D-SEB2 and D-BUF2 surrounding the FUs, and 64 cycles for the 2 hops (31 cycle maximum for each D-FIFO-32 and 1 cycle for each D-REG). Hence, in this scenario the latency margin – and thus the maximum latency of the shorter path – is not sufficient to balance the longer path.

In order to balance the longer path, the shorter 2-node path would have to be mapped using at least 8 routing hops. This is $4 \times$ more hops than the minimum routing of 2 hops. To tackle this challenge we need a mechanism to deliberately lengthen (map with extra routing hops) the paths whose latency margins are insufficient.

Our approach is to insert special balancing nodes on such paths in the input DFG. The balancing nodes are mapped to NOPs and routing-only cells, thereby forcing the router to lengthen the path by routing extra hops that connect the added cells. In effect, this method indirectly instructs the place-and-route algorithm to map balancing modes
to D-FIFOs (one each hop), thus accumulating more latency margin.

With this approach, we combined the overall hop minimization by the place-and-route algorithm for DFG routability with the selective lengthening of short paths for latency balancing.

### 5.7.2 ILP Model and Heuristics

Our technique uses an ILP model and a heuristic to determine: 1) on which DFG paths balancing nodes should be inserted; and 2) how many nodes should be inserted.

The key idea of this technique is to determine the latency deficit of the shorter DFG paths relative to the longer paths that join them in the case when: 1) the shorter paths are mapped minimally— that is, each edge is mapped with a single routing hop, thus having one D-FIFO per edge; and 2) the longer paths are lengthened by a factor $J$, meaning that each edge is mapped with $J$ routing hops.

This hypothetical DFG mapping is unfavourable in terms of latency balancing: it conservatively assumes that the place-and-route algorithm will lengthen the longer paths while minimizing the shorter paths of a DFG. As a result, our approach will account for both the imbalance of an input DFG and the unfavourable placement-and-routing of the DFG.

In developing our method, we first create a post-map DFG that represents the minimal mapping of the DFG, which means that all paths are mapped with a minimum number of routing hops. Thus, every DFG edge has only one D-FIFO associated with it. Next, we construct an ILP formulation to model the unfavourable lengthening effect of longer paths. Our model uses FU latencies that are pro-rated with $1/J$ and D-FIFO latencies that are bounded by a reduced depth $(1/J) \times d_{OV}$:

$$d_{OV}^l = \left\lceil \frac{1}{J} \times d_{OV} \right\rceil$$  \hspace{1cm} (5.8)
We derive this approximation from the expression that equates the latency of a shorter path $A$ with that of a longer and also lengthened path $B$ that has $J$ routing hops for each edge:

$$L_A = L_B$$

$$\sum_{FU_s(A)} T_i + \sum_{edges(A)} (b_i + 1) = \sum_{FU_s(B)} T_i + J \sum_{edges(B)} (b_i + 1)$$

The latencies of paths are given as a sum of FU latencies ($T_i$) and a sum of the D-FIFO ($b_i$) and D-REG (1) latencies for each routing hop. The latencies of the DDPUs surrounding the FUs are included in the FU latencies ($T_i$). After dividing both sides of the expression with $J$, we obtain the following form:

$$\frac{1}{J} \sum_{FU_s(A)} T_i + \frac{1}{J} \sum_{edges(A)} (b_i + 1) = \frac{1}{J} \sum_{FU_s(B)} T_i + \sum_{edges(B)} (b_i + 1)$$

The above equation assumes that D-FIFOs on longer paths have a latency of 2 (minimum D-FIFO latency) because longer paths execute at their minimum latency, whereas shorter paths are lengthened (i.e., they exercise the available latency margin). Thus, the longer paths in the model will not be affected by the reduced D-FIFO depth.

The CheckBalanced function and the above ILP formulation with the reduced D-FIFO depth $d_{OV}^J$ and pro-rated FU latencies can test if the DFG’s minimal mapping is balanced. However, it does not tell us which paths are imbalanced and by what latency.

In order to determine the latency difference of imbalanced paths we extend the ILP model used by the CheckBalanced function. We start by creating a balancing DFG for the minimal DFG mapping (an example is shown in Figure 5.8). Each edge $(u, v)$ has an unknown variable $b_{uv}$ in order to represent the latency of the edge’s D-FIFO. We then extend the balancing DFG with an additional unknown variable $(c_{uv})$ for each graph edge. This variable indicates the latency deficit on the corresponding edge, i.e.,
the latency that is required beyond the latency of the D-FIFO in order to balance this path with longer paths that join it.

Figure 5.8: Balancing DFG created for the minimal mapping of an example DFG.

The D-FIFO latency \((b_{uv})\) is bounded, while the latency deficit \((c_{uv})\), which is not bounded, can be arbitrarily large:

\[
\begin{align*}
\forall (u,v) \in E & : 2 \leq b_{uv} \leq d_{OV}^J - 1 \\
\forall (u,v) \in E & : 0 \leq c_{uv}
\end{align*}
\]  

(5.11a)  
(5.11b)

If a balanced mapping can be achieved with the latency margins of the D-FIFOs that already exist on the edges (modeled with \(b_{uv}\) variables), then no balancing nodes should be inserted into the DFG. Thus, the solution of the model should produce a value of 0 for all \(c_{uv}\). However, if a balanced mapping cannot be achieved only with the latency margins of \(b_{uv}\) variables, then we should insert the minimum number of balancing nodes
in order to balance the DFG. Consequently, the model must minimize both the sum of all $b_{uv}$ (as explained in Section 5.5) and the sum of all $c_{uv}$. However, it must prioritize minimizing the sum of all $c_{uv}$ in order to minimize the number of balancing nodes.

This is achieved by multiplying each $c_{uv}$ with a factor of 2 in the objective function:

\[
\text{minimize: } \sum_{(u,v) \in E} b_{uv} + \sum_{(u,v) \in E} 2c_{uv} \quad (5.12a)
\]

As a result, for a given path, the ILP model will first allocate latency to the $b_{uv}$ variables and only allocate latency to the $c_{uv}$ variables when the $b_{uv}$ variables have been maxed out (i.e., $b_{uv} = d_{OV}^J - 1$).

After solving the ILP model, the number of balancing nodes ($N_{uv}$) inserted on an edge $(u, v)$ is computed as:

\[
N_{uv} = \left\lceil \frac{c_{uv}}{d_{OV}^J} \right\rceil \quad (5.13)
\]

For example, if $c_{uv} = 67$ and $d_{OV}^J = 20$ then we insert 4 balancing nodes ($N_{uv} = 4$).

Our experiments suggest that the average lengthening of a DFG edge is 3 ($J = 3$): a DFG edge is on average implemented using 3 routing hops by the place-and-route algorithm. We empirically determined that a suitable value for $1/J$ is in the $[0.2, 0.33]$ range. We use a value of $1/J = 0.2$ in our experiments. An exhaustive sweep of parameter values is left for future work.
Chapter 6

Overlay Physical Design and Compilation to FPGA

The physical design aspect of this thesis tackles the two key challenges involved in compiling mesh-of-FUs overlays to the FPGA fabric: 1) maintaining high $f_{\text{MAX}}$ as the mesh size is scaled up; and 2) developing a methodology for the quick compilation of custom overlays.

When scaled to span the size of large modern FPGAs, the mesh-of-FUs overlays suffer a significant drop in $f_{\text{MAX}}$ [128]. Our investigation shows that, even for mesh-of-FUs overlays built with distributed control and elastic pipelines, the default flow of the FPGA CAD (i.e., a flat flow) is not able to exploit the regular structure and distributed control of the circuit. We show that this is a result of the overlay circuit’s large size, which in turn limits the ability of CAD tools to successfully optimize it. This optimization challenge is certainly shared with other large FPGA circuits. The drop in the $f_{\text{MAX}}$ of large mesh-of-FUs overlays reduces their performance, thus hindering their use as accelerators.

Furthermore, mesh-of-FUs overlays can be customized for an application domain, which enables them to leverage the flexibility of the FPGA fabric. A mesh often contains a number of different FU types (e.g., mul, div, exp, sqrt), and the overlay is customized
through choice and arrangement of FUs within the mesh. In order to make the creation of mesh-of-FUs easier, it is necessary to provide a software programmer (a non-expert in FPGA CAD) with a methodology for the quick creation of a custom overlay for the target application domain. This methodology should allow for the quick compilation of new overlays, the quick modification of existing ones, and for the pre-compiled components of existing overlays to be reused to build new overlays.

The above two challenges are exacerbated by the exponential increase in FPGA size. A viable approach to addressing the problem of exponentially increasing size is to explore flows that are amenable to parallelization [43]. Today, a compilation flow that addresses both challenges is an open research question.

We address the above challenges by implementing a *tile-based bottom-up* CAD flow that utilizes the hierarchical physical design techniques– partitioning and floorplanning–offered by commercial CAD tools. In our flow, an overlay is divided into *tiles*, or groups of adjacent overlay cells. The tiles are then compiled onto a coarse-grain rectangular floorplan, which allows differently sized FUs to be tightly placed within a tile’s region, and thus make efficient use of the region’s resources. In addition, we insert “mirrored” elastic buffers on paths between neighbouring tiles to ensure that the inter-tile paths are not a bottleneck for \( f_{\text{MAX}} \).

Our tile-based bottom-up flow enables the *independent* compilation of each tile to its respective region in the floorplan. Furthermore, a large overlay can be compiled with high \( f_{\text{MAX}} \) by only “stitching” a set of pre-compiled tiles, obviating the need to redo the placement-and-routing of critical paths. The use of rectangular coarse-grained floorplans allows overlays of the same mesh size to be compiled to the same floorplan (*template floorplan*), which facilitates the design of a *tile library*– a collection of pre-compiled tiles–that can be re-used to quickly stitch together new overlays. Moreover, independent tile compile makes the parallel compilation of tiles feasible, thereby presenting an opportunity for reducing overlay compilation time.
We experimentally evaluate the bottom-up flow on 5 custom overlays and compare it to the flat flow. We also compare it to the top-down flow in which the tiled overlay is compiled at once in a single CAD project without inserting inter-tile elastic buffers. Thus, the top-down flow demonstrates the benefit of using only partitioning and floorplanning.

Our results show that: (1) the bottom-up flow achieves significantly higher \( f_{\text{MAX}} \) than the flat flow (345 MHz vs. 244 MHz), and slightly higher \( f_{\text{MAX}} \) than the top-down flow; (2) there is relatively little \( f_{\text{MAX}} \) degradation in the bottom-up flow (no more than 22%) as the overlay’s mesh is grown from \( 1 \times 1 \) to \( 12 \times 15 \) overlay cells (\( f_{\text{MAX}} \) degrades up to 45% and 26% with the flat and top-down flows); (3) the use of inter-tile buffers adds no more than 8% resource overhead; (4) on a single machine, the compile times for the bottom-up flow are faster than the flat and top-down flows by 19% and 11%, respectively; and (5) the bottom-up flow allows us to build a library of 45 pre-compiled tiles that can be used to stitch together new overlays in 35 minutes while achieving a high \( f_{\text{MAX}} \). The flat and top-down flows do not provide this capability.

Our work on the physical design and compilation to the FPGA fabric of mesh-of-FUs overlays makes the following contributions. It introduces a tile-based bottom-up flow that achieves a high \( f_{\text{MAX}} \) that degrades little as the size of the overlay increases. This flow can also reduce the mesh-of-FUs compilation time because it enables parallel tile compilation and fast tile stitching. Furthermore, the bottom-up flow offers a method that enables the creation of a pre-compiled tile library that facilitates reuse.

It is important to note that our results are necessarily target device and CAD tool specific. However, all modern FPGAs and CAD tools are built upon two core principles, namely, a BLE-based island-style architecture and a flat flow consisting of clustering, placement, and routing. Thus, it is likely that the tile-based bottom-up methodology will be beneficial across the spectrum of modern FPGAs.

We begin this chapter by outlining the characteristics of the target FPGA device and the CAD tool in Section 6.1. We then describe the use of physical regions in Section 6.2.
In Section 6.3 we present the flat flow. Sections 6.4 and 6.5 describe the top-down flow and the floorplanning methodology. The tile-based bottom-up flow is described in Section 6.6, and tile reuse is presented in Section 6.7. Finally, Section 6.8 introduces the overlay-to-FPGA functional layout matching.

6.1 Target FPGA Device and CAD Tool

<table>
<thead>
<tr>
<th>Part number</th>
<th>EP4SGX530KH40C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed grade</td>
<td>2</td>
</tr>
<tr>
<td>ALMs</td>
<td>212,480</td>
</tr>
<tr>
<td>ALUTs</td>
<td>424,960</td>
</tr>
<tr>
<td>FFs</td>
<td>424,960</td>
</tr>
<tr>
<td>18-bit DSPs</td>
<td>1,024</td>
</tr>
<tr>
<td>M9Ks</td>
<td>1,280</td>
</tr>
</tbody>
</table>

Table 6.1: Stratix IV FPGA characteristics

We explore our tile-based bottom-up flow using an Altera Stratix IV FPGA device whose characteristics are shown in Table 6.1. The majority of the target Stratix IV device is comprised of logic resources (166 logic columns). The device has 4 DSP columns and 14 M20K columns that are spread throughout it. We have previously elaborated on the key features of ALMs, DSPs, and M20Ks blocks in Section 2.1.3, and we have also described the non-uniformity and asymmetry of the layout that is common to the families of Stratix devices.

We use the Altera Quartus II 12.1sp1 CAD tool to compile the overlays to the target FPGA device, and all of our flows use the following recommended Quartus II performance optimization parameters in order to help achieve higher $f_{\text{MAX}}$ [129]: synthesis optimizations for speed with timing-driven synthesis; highest fitter effort with placer and router effort multipliers of 10; and physical synthesis optimizations for performance with extra effort. In addition, we set the target clock rate to 500 MHz.
6.2 Physical Regions

In all three of the flows we designed (i.e., flat, top-down, and bottom-up), we constrain the entire overlay to a rectangular physical region which we call the *overlay’s physical region*. In the top-down and bottom-up flows, this region is further floorplanned such that it contains a set of smaller rectangular regions; this process is detailed later on in Section 6.5.

The granularity of physical regions and floorplanning in Altera’s Quartus is at the LAB level. The size of a physical region is measured as the total number of LABs (or ALMs) in the region, and it affects the logic utilization\(^1\) and the \(f_{\text{MAX}}\) of the circuit constrained to that region. We observed that a smaller region size results in tighter packing (higher logic utilization), but also in routing congestion. In contrast, using regions that are too large wastes resources. Figure 6.1 illustrates this effect for a single add/sub (A) overlay cell (consumes 1010 ALMs) that has been compiled to physical regions of different sizes. As the figure shows, \(f_{\text{MAX}}\) increases with region size, while the logic utilization drops.

In order to balance resource usage and \(f_{\text{MAX}}\), we target a physical region size that results in 80-85% logic utilization. We experimentally determined this region size (the number of LABs) for each type of overlay cell (e.g., add/sub, exp, log) independently. The total region size of an \(H\times W\) mesh overlay is computed as the sum of the sizes of its cells’ regions.

6.3 Flat Flow

The flat flow is the basis for the top-down and bottom-up flows whose designs extend the flat flow by incorporating additional constraints.

The flat flow is designed as an extension of the default push-button flow of Quartus

\(^1\)We define a region’s logic utilization as reported by Quartus II: the ratio between utilized ALUT/FF pairs and the total number of pairs within the region.
II. We do so by configuring Quartus II with the performance optimization parameters given in Section 6.1, and also by applying two design constraints: the overlay’s physical region and the overlay’s I/O constraints. The entire overlay circuit, which is described in Verilog, is then compiled using the default Quartus II push-button flow.

We constrain the overlay’s I/O as follows: each overlay input and output port (32-bit data and two 1-bit synchronization signals) is connected to a virtual I/O port which is realized as a register (overlay I/O registers); the placement of the I/O registers is constrained to the outside of the overlay’s physical region (as shown in Figure 6.2a); the exact location of the overlay I/O registers is decided by the Quartus II placer; the use of the overlay’s I/O registers for virtual I/O ensures that the paths to and from the overlay’s I/O ports are optimized during compilation; consequently, this guides Quartus II to place the overlay’s boundary cells close to the physical region boundary.

6.4 Top-Down Flow

We design our top-down flow by augmenting the flat flow with partitioning, fine-grain overlay I/O constraining, and floorplanning. In the top-down flow, the entire overlay that has been partitioned and floorplanned is compiled in a single Quartus II project. In
Chapter 6. Overlay Physical Design and Compilation to FPGA

Figure 6.2: Constraints for an overlay I/O register connected to the boundary overlay cell (2,1) of a 4×4 mesh overlay: (a) I/O register allowed anywhere in the large region surrounding the overlay; (b) I/O register allowed only within a fine-grained I/O region.

In this section, we describe partitioning and I/O constraining; our floorplanning approach and trade-offs will be discussed in Section 6.5.

We use partitioning at the cell-level with each overlay cell being designated as a design partition in Quartus II. The logic resources of different partitions are clustered separately by the CAD tool, and this in turn helps the tool to localize the placement of the logic resources of each overlay cell. Furthermore, we hypothesize that this partitioning scheme also helps to expose the mesh topology of the overlay circuit to the Quartus II placer.

In addition, our top-down flow utilizes fine-grain overlay I/O constraining: we create a rectangular region for the overlay I/O registers that connect to each boundary overlay cell, as shown in Figure 6.2b. An overlay with a mesh size of H×W has 2×(H+W) fine-grain I/O regions. Fine-grained I/O constraining may improve $f_{\text{MAX}}$ by guiding the placement— as performed by the CAD tool— of the overlay’s boundary cells towards their “ideal” location. They are pulled towards a small I/O region to which I/O registers are constrained.

6.5 Floorplanning Granularity: Cells vs. Tiles

Floorplanning constrains the possible locations of overlay cells, and thus simplifies the placement task performed by the CAD tool. A key challenge in overlay floorplanning is
finding the optimal size and shape for the physical regions that comprise the floorplan. Floorplanning decisions are often driven by the circuit granularity at which floorplanning is applied and the circuit topology. Since the overlay has a regular rectangular mesh topology, we opt to use rectangular physical regions. In this section, we elaborate on and explore two levels of granularity for floorplanning: cell-based and tile-based.

In cell-based floorplanning we constrain each overlay cell to its own physical region. Figures 6.3a and 6.3b show a functional layout of an 8×10 overlay and its cell-based floorplan, which is designed by arranging the physical regions in a grid. Physical regions in the same row/column are aligned and have the same height/width. The height of each row is determined by the height of the tallest region of any overlay cell in the row, and the width of a floorplan column is determined by the widest region of any overlay cell in the column. Overlay cells vary in resource use by up to 5X due to the different amounts of resources used by their floating-point FUs. This leads to oversized physical regions for smaller overlay cells and thus wasted resources, which is indicated by the black areas in Figure 6.3b. One possible way to improve cell-based floorplanning is to use irregular floorplans, but this approach requires a floorplan design for each overlay. However, we seek a methodology that allows for reuse and quick overlay compilation, and, as such, using irregular and full custom floorplans for each overlay is not a viable approach because it lacks the required flexibility.

We propose tile-based floorplanning as a more flexible and resource-efficient alternative to cell-based floorplanning. A tile represents a rectangular section of an overlay’s functional layout, i.e., a rectangular group of overlay cells. In this approach, we use partitioning at the tile level and at the cell level within the tiles. The functional layout of the example 8×10 overlay from Figure 6.3a is partitioned into four 4×5 tiles. Figure 6.3c shows four physical regions arranged in a 2×2 floorplan grid. Each tile is compiled to one of the physical regions by the FPGA CAD tool. The figure shows the tight placement— as produced by the FPGA CAD tool— of small and large overlay cells within the tile's
Figure 6.3: (a) 8×10 mesh overlay - functional layout with four 4×5 tiles, (b) compiled with top-down flow and cell-based floorplanning, (c) compiled with top-down flow and tile-based floorplanning, requires a smaller overlay region on Stratix IV – figures (b) and (c) have matching scale.
physical region, which results in the overall overlay floorplan being smaller than if we were to use the cell-based approach.

Tile-based floorplanning also allows a larger degree of flexibility in terms of an overlay’s functional layout while still being resource efficient. Due to 1-to-1 cell-to-region mapping in cell-based floorplanning, only those cells whose physical regions contain DSP blocks can be cells with a DSP-based FU. In contrast, in tile-based floorplanning any cell within a physical region that contains DSP blocks can potentially be a DSP-based cell. Similarly, having multiple cells per tile allows for a choice of a mix of cells in each tile that both makes good use of resources and does not over-limit the functional layout of an overlay. Nonetheless, functional layouts in tile-based floorplanning are not as flexible as the functional layouts of non-floorplanned overlays. In summary, tile-based floorplanning provides a balance of the benefits of both cell-based floorplanning and no floorplanning by combining floorplanning with a degree of flexibility.

6.6 Bottom-Up Flow

As in the top-down flow, our bottom-up flow also uses partitioning at the tile and cell levels and tile-based floorplanning. However, in the bottom-up flow, each overlay tile is compiled *independently* in a separate Quartus II projects to generate the post-fit netlist (i.e., the internal place-and-route) for the tile. Subsequently, in the top-level project, the post-fit netlists for the tiles are imported and then “stitched” together to form an overlay: only the paths between the tiles are routed, while the post-fit netlists of the tiles are preserved. Our bottom-up flow does not relocate the post-fit netlists of compiled tiles, that is, a post-fit netlist is only used in the physical region to which it was compiled to. Thus, for an overlay that is partitioned into 9 tiles (and thus 9 physical regions), we need to generate all of the 9 post-fit netlists in their respective physical regions.

The main challenge in a bottom-up flow is to ensure that the inter-tile paths do
Chapter 6. Overlay Physical Design and Compilation to FPGA

not create a bottleneck for $f_{\text{MAX}}$, thus enabling the independent compilation of all of the overlay’s tiles. This is done by inserting “mirrored” elastic buffers on the inter-tile paths, as will be described in Section 6.6.1.

The bottom-up flow has two advantages over the top-down flow. The first is its use of divide-and-conquer compilation: tiles can be compiled in parallel on different machines, thus speeding up overlay compilation. The second advantage is its ability to reuse pre-compiled tiles across different overlays, which enables even faster overlay compilation by only stitching together a set of pre-compiled tiles. Section 6.7 describes our approach for tile reuse across overlays via the use of a template floorplan and a pre-compiled tile library.

6.6.1 Independent Tile Compile and Inter-Tile Elastic Buffers

In order to ensure that inter-tile paths are not timing critical, and thus enable the independent compilation of tiles, we: (1) use “mirrored” elastic buffers which we insert at the tile boundary cells, thus ensuring that each inter-tile path is registered at both ends (source and sink tile); and (2) apply fine-grain I/O constraints to the boundary cells of each tile (i.e., I/O ports of each tile). This ensures that inter-tile EBs are placed close to the tile’s physical region boundary so as to minimize the routing delays between two adjacent tiles.

A 2-slot elastic buffer is used in elastic and latency insensitive pipelines to register data, valid, and stall signals (detailed in Section 2.4). Circuits built using these pipelines often use a single type of 2-slot EB throughout the entire circuit [105, 107, 130]. While all of the signals are registered in each EB, the path between two of the same EBs that are connected back-to-back passes through a mux and adds a timing delay. In contrast, we make use of two types of 2-slot EBs—called inter-tile EBs—and we insert a pair of them on each inter-tile path. The tile-out EB is inserted at each tile output port, connecting to the tile-in EB at each tile input port, as shown in Figure 6.5a. This results in inter-tile
paths that are *registered at both ends* (i.e., no logic is on the path), thus reducing their timing delay. The reduction in timing delay is due to the “mirrored” structure of the two EBs: the tile-out EB has no logic (i.e., no muxes) on its output side, while the tile-in EB has no logic on its input side. Observe that, if we instead used only one type of EB (e.g., tile-out EB) at all of the tile ports, the inter-tile paths would not be registered at both ends because they would pass through a mux.

To compile an individual tile we first insert tile-in and tile-out EBs at each tile input and output port, respectively. We then connect each EB to a tile I/O register whose location is constrained to a small physical region, as shown in Figures 6.5b/c. As a result, the CAD tool places the inter-tile EBs close to tile I/O registers, and hence close to the physical region boundary. This ensures that the two inter-tile EBs in adjacent tiles are...
Figure 6.5: The bottom-up compilation of a 4×8 overlay by stitching two 4×4 tiles; tiles 1 and 2 are compiled independently using fine-grained tile I/O constraints.

placed close to one another even though the two tiles are compiled independently (shown in Figure 6.5d). Note that tile I/O registers are not part of the final tile circuit; they only serve to “pull” the inter-tile EB close to the tile boundary during the compilation of the tile.

For a tile with H×W overlay cells, a total of 4×(H+W) inter-tile EBs are inserted at the tile’s boundary. This adds resource overhead, which we measure in Section 7.9.5.

### 6.6.2 Impact of Tile Stitching on Clock Skew and Routing Conflicts

The clock skew\(^2\) for a pair of registers within a tile can change during tile stitching. The reason for this effect is as follows. The post-fit netlist of a tile also includes the clock

\(^2\)The difference in the arrival time of a clock signal at two different registers.
paths that have been routed along the branches of the global clock network. Some of the
clock network branches span across the pairs of adjacent tiles (i.e., they cross the physical
region boundaries of adjacent tiles). As a result, two adjacent tiles may be compiled such
that they use a number of the same clock network branches to route the clock paths.
Since all the tiles use a common global clock it is legal for clock network branches to
be *shared* by two adjacent tiles. When two tiles that share clock branches are stitched
together, they will both add load to the shared branches, thus changing the clock delays
on them. These changes in clock delay can cause a swing from a positive clock skew to
a negative one. Thus, a data path within a tile may become timing-critical and degrade\(f_{\text{MAX}}\).

A tile’s internal logic (i.e., LUTs, DSPs) is guaranteed to be contained within its
physical region. However, the routing wires that are used to connect the tile’s internal
logic may cross outside the region’s boundary. Similarly to the effect with the clock
branches described above, adjacent tiles may use the same routing wire, however they
would use it for different data signals. This causes a routing conflict because it is not legal
for different data signals to share the same routing wire. The CAD tool resolves these
conflicts by re-routing the affected nets during stitching [71]. Re-routing can increase
tile-stitching time if there is a large number of nets that are affected by routing conflicts,
but it can also negatively impact \(f_{\text{MAX}}\) if the remaining routing resources are slow, thus
causing the re-routed nets to become critical paths.

### 6.7 Tile Library and Template Floorplan

Overlays that have the same mesh size (e.g., 12×15) and similar total logic resource usage
can be compiled to the same floorplan; we term this a *template floorplan*. The set of tiles
comprising these overlays forms a *tile library*.

We illustrate this approach using the example shown in Figure 6.6. A floorplan with
nine physical regions (R_{1,1} to R_{3,3}) that spans the majority of the Stratix IV device is shown in Figure 6.6a, while Figures 6.6b/c show two overlays (OV_A and OV_B) that were compiled \textit{from-scratch} to the template floorplan.

New \textit{tile-reuse} overlays can be built by reusing the tiles from the library. An example of such an overlay is OV_C which has been built by reusing the tiles from OV_A and OV_B (shown in Figure 6.6d). For each region R_{i,j} we pick a tile T_{i,j} either from OV_A or from OV_B. In this case, tiles T_{1,3}, T_{2,1}, T_{3,2}, and T_{3,3} are from OV_A while the rest are from OV_B.

Overlays with different mesh sizes and/or large differences in resource usage can use different template floorplans. In addition, in our flow, a tile T_{i,j} is always used at its original region (R_{i,j}), i.e., we do not relocate pre-compiled tiles. The use of different template floorplans and tile relocation can be explored as part of future work.

6.8 Overlay-to-FPGA Functional Layout Matching

The overlay cell’s programmable routing and DDPUs only use the FPGA’s logic resources. Similarly, many FUs (e.g., \textit{addsub}, \textit{shift}, and \textit{abs}) use only logic resources. Since columns of logic resources are abundant across an FPGA device, overlay cells with logic-only FUs can be implemented anywhere on the device.

However, overlay cells that contain a DSP-based FU (e.g., a floating point \textit{mult} or \textit{div}) impose constraints on their placement on an FPGA device. In order to achieve the high f_{\text{MAX}} of 400 MHz, the logic resources and DSP blocks of such cells must be placed close together (i.e., clustered). Since the FPGA device has far fewer DSP columns than logic columns, the physical placement of DSP-based overlay cells is a challenge both in terms of the CAD flow and overlay’s functional layout design.

We propose a technique that we refer to as overlay-to-FPGA functional layout matching. In essence, this technique entails that: 1) the overlay’s functional layout is designed
Figure 6.6: (a) a template tile-based floorplan with 9 regions on a Stratix IV device, (b)(c) OV_A and OV_B (each has a 12×15 mesh with nine 4×5 tiles) compiled from-scratch to the template floorplan, (d) OV_C is compiled by stitching a mix of tiles from OV_A and OV_B.
Figure 6.7: An example of matching a functional layout of a 3x10 overlay to a section of Stratix IV FPGA with two DSP columns. The desired physical placement of three overlay DSP-based cells from DSP overlay columns is shown. Overlay cell codes: D=div, L=log, S=sqrt, A=addsub, C=abs-cmp and R=routing-only cell.

such that DSP-based overlay cells are arranged as a subset of overlay columns (referred to as *DSP overlay columns*); and 2) that the DSP overlay columns are positioned within the overlay’s functional layout to facilitate their placement close to the FPGA’s DSP columns. The positioning of DSP overlay columns within the overlay is chosen such that the amount of logic resources used by logic-only overlay columns between DSP overlay columns corresponds to the amount of logic resources on the device between the FPGA’s DSP columns (i.e., the layouts are *matched*).

Figure 6.7 shows an example of matching the functional layout of a 3x10 overlay to a section of a Stratix IV device with two DSP columns. In this example, 6 logic-only overlay columns can fit between the two FPGA DSP columns and 2 logic-only overlay columns can fit to the left. Thus, we match the overlay’s functional layout to the FPGA’s layout by positioning the DSP overlay columns to columns 3 and 10 of the overlay. Figure 6.7b also illustrates the desired physical placement of three DSP-based overlay cells as a result of overlay-to-FPGA layout matching. The logic resources of these cells (depicted as blobs) are clustered close to the DSP columns.
6.8.1 Floorplanning and Functional Layout Matching

The use of floorplanning for the design of an overlay constrains the design space of the overlay’s functional layouts. Specifically, a floorplan’s physical region that does not contain any DSP blocks cannot implement DSP-based overlay cells. As a result, the section of the overlay’s functional layout that is assigned to such a physical region cannot contain any DSP-based overlay cells.

However, floorplanning is very effective when used together with overlays whose functional layouts are matched to the FPGA layout. This is because physical regions “enforce” the physical location of DSP overlay columns to the vicinity of the FPGA’s DSP columns.

In Section 6.5 we discussed the relative merits of cell- and tile-based floorplanning. Cell-based floorplanning provides tighter placement constraints but is also more restrictive in terms of the overlay’s potential functional layouts as only cells whose physical regions contain DSP blocks can be DSP-based cells. Tile-based floorplanning is more flexible in that any overlay cell assigned to a tile with DSP blocks can potentially be a DSP-based cell. As a result, tile-based floorplanning allows for coarser overlay-to-FPGA layout matching, and thus enabling a larger design space for the overlay’s functional layouts.
Chapter 7

Experimental Evaluation

In this chapter, we experimentally evaluate the overlay architecture, the DFG-to-overlay mapping algorithm, and the physical design flows.

We prototype two large overlays on a Stratix IV FPGA and evaluate both their performance and that of the DFG-to-overlay algorithm using 16 DFGs extracted from 7 parallel code sections. We show that the overlays achieve high $f_{\text{MAX}}$ and that maximum throughput can be achieved for all DFGs, which validates the overlay architecture. We also show that very little time is required to map each DFG to an overlay—within 7 seconds—which validates the heuristics that are used in the DFG-to-overlay mapping algorithm. Furthermore, we evaluate our tile-based bottom-up overlay compilation flow and demonstrate its benefits using a set of 5 custom overlays. We also compare the bottom-up flow to the flat and top-down flows.

This chapter is organized as follows: Section 7.1 describes the benchmark DFGs; Section 7.2 describes the evaluation platform; Section 7.3 details the design of the two prototype overlays used to evaluate the performance of the benchmark DFGs; Section 7.4 outlines the metrics used to evaluate the overlays as well as the execution of the DFGs on the overlays; Section 7.5 presents the $f_{\text{MAX}}$ of the two prototype overlays as well as the performance of the DFGs on those overlays; Section 7.6 evaluates the DFG-to-overlay
mapping algorithm, including the compile time, algorithm convergence, balancing nodes insertion, DFG routability and balancability and the impact of overlay D-FIFO depth on DFG throughput.

In Sections 7.7 and 7.8 we show the resource usage and resource overhead of the two prototype overlays. We measure the resource overhead involved in using overlays to execute DFGs by comparing their resource usage to: 1) an ideal FUs-only overlay, and 2) direct DFG compilation to FPGA fabric (i.e., a hard-wired DFG circuit).

Section 7.9 presents the evaluation of the physical design flows for overlay compilation to FPGA fabric. We first describe 5 benchmark overlays, and we then compare the flat, top-down, and bottom-up flows in terms of the $f_{\text{MAX}}$ they achieve for these overlays. We also measure the resource cost of using inter-tile EBs in the bottom-up flow. Next, we demonstrate the benefits of the tiled-based bottom-up flow including tile library and tile reuse, overlay compile time reduction, and the resource efficiency of tile-based floorplanning. Finally, we investigate the impact of overlay-to-FPGA functional layout matching on the flat flow.

### 7.1 Benchmark DFGs

The work in this thesis uses DFGs as the input model for the overlay and mapping algorithm. Thus, our evaluation is focused on the mapping and execution of several DFGs that are extracted from several applications.

We manually extract the DFGs from parallel code sections that are taken from applications that are common targets for acceleration. As discussed in the introduction, parallel code sections do not generally represent full applications; however, these code sections are most often the ones that are offloaded to an accelerator. Thus, the throughput of the execution of the DFGs on the overlay serves as the performance measurement of only the parallel code sections and not the entire application. However, the DFGs
do represent the bulk of the parallel code sections; thus, the throughput of the DFGs is representative of the throughput of the parallel code sections. Since the overlay is not integrated into a full system, the testing of the DFGs and the measurement of their throughput are done through a surrounding test harness. We elaborate on the design of the test harness in Section 7.2.

We manually extract 16 DFGs from 7 parallel code sections (i.e., parallel loops and data-parallel kernels), and we describe these DFGs using the PyDFG API introduced in Section 5.2.1. The resulting DFGs and their characteristics are shown in Table 7.1.

In the remainder of this section we provide a description of: 1) the salient characteristics of the benchmark DFGs; 2) the process of manually extracting DFGs from parallel code sections; 4) the parallel code sections from which the DFGs are extracted; 3) the extraction details for each DFG; and 5) the DFG replication method for each DFG. In addition, Appendix B shows the topology and compute operations for each DFG.

The DFGs are divided into two sets: 1) a floating-point set containing six DFGs that perform 32-bit single-precision floating-point operations; and 2) an integer set containing ten DFGs that perform 32-bit integer operations. The table shows the number of compute nodes in each DFG as well as the number of balancing nodes that were inserted into the DFGs. We elaborate on the results of the balancing node insertion in Section 7.6.2.

The input nodes for each DFG are broken-down into regular (i.e., streamed) input nodes and constant input nodes (the input node types were defined in Section 5.2.1). The DFGs differ in size (i.e., the total number of nodes) as well as in the compute operations of their nodes. The DFGs also vary in their topology, which is characterized by the DFG connectivity patterns (e.g., diverge-converge, reduction, fan-outs) shown in Appendix B. This variability of DFG characteristics exercises the ability of the prototype overlays to route and balance the DFGs.

We also replicate small DFGs to create larger DFGs. The extensions 2x, 3x, etc. next to the DFG names in Table 7.1 indicate the number of times the smaller DFGs are
Table 7.1: DFG characteristics. The table also contains the number of balancing nodes inserted by the DFG-to-Overlay mapping algorithm, and the ratio of input nodes to compute nodes.

<table>
<thead>
<tr>
<th>DFG</th>
<th>Data type</th>
<th>compute nodes</th>
<th>balancing nodes</th>
<th>in (regular) nodes</th>
<th>in (const) nodes</th>
<th>in (all) nodes</th>
<th>out nodes</th>
<th>total nodes</th>
<th>in / compute</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-Body</td>
<td>SP FP</td>
<td>20</td>
<td>17</td>
<td>7</td>
<td>4</td>
<td>11</td>
<td>3</td>
<td>51</td>
<td>0.55</td>
</tr>
<tr>
<td>N-Body-2x</td>
<td>SP FP</td>
<td>40</td>
<td>34</td>
<td>11</td>
<td>4</td>
<td>15</td>
<td>3</td>
<td>92</td>
<td>0.38</td>
</tr>
<tr>
<td>N-Body-3x</td>
<td>SP FP</td>
<td>60</td>
<td>51</td>
<td>18</td>
<td>8</td>
<td>26</td>
<td>6</td>
<td>143</td>
<td>0.43</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>SP FP</td>
<td>68</td>
<td>59</td>
<td>3</td>
<td>16</td>
<td>19</td>
<td>2</td>
<td>148</td>
<td>0.28</td>
</tr>
<tr>
<td>MatMul</td>
<td>SP FP</td>
<td>63</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>24</td>
<td>9</td>
<td>96</td>
<td>0.38</td>
</tr>
<tr>
<td>MatMulAdd</td>
<td>SP FP</td>
<td>72</td>
<td>0</td>
<td>24</td>
<td>9</td>
<td>33</td>
<td>9</td>
<td>114</td>
<td>0.46</td>
</tr>
<tr>
<td>RGB2YIQ</td>
<td>int32</td>
<td>21</td>
<td>0</td>
<td>3</td>
<td>11</td>
<td>14</td>
<td>3</td>
<td>38</td>
<td>0.67</td>
</tr>
<tr>
<td>RGB2YIQ-2x</td>
<td>int32</td>
<td>42</td>
<td>0</td>
<td>6</td>
<td>11</td>
<td>11</td>
<td>6</td>
<td>65</td>
<td>0.40</td>
</tr>
<tr>
<td>RGB2YIQ-4x</td>
<td>int32</td>
<td>84</td>
<td>0</td>
<td>12</td>
<td>22</td>
<td>34</td>
<td>12</td>
<td>130</td>
<td>0.40</td>
</tr>
<tr>
<td>RGB2YIQ-5x</td>
<td>int32</td>
<td>105</td>
<td>0</td>
<td>15</td>
<td>33</td>
<td>48</td>
<td>15</td>
<td>168</td>
<td>0.46</td>
</tr>
<tr>
<td>SAD</td>
<td>int32</td>
<td>47</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>32</td>
<td>1</td>
<td>80</td>
<td>0.68</td>
</tr>
<tr>
<td>SAD-2x</td>
<td>int32</td>
<td>94</td>
<td>0</td>
<td>36</td>
<td>0</td>
<td>36</td>
<td>2</td>
<td>132</td>
<td>0.38</td>
</tr>
<tr>
<td>GaussianBlur</td>
<td>int32</td>
<td>49</td>
<td>1</td>
<td>25</td>
<td>6</td>
<td>31</td>
<td>1</td>
<td>82</td>
<td>0.63</td>
</tr>
<tr>
<td>GaussianBlur-2x</td>
<td>int32</td>
<td>98</td>
<td>2</td>
<td>30</td>
<td>6</td>
<td>36</td>
<td>2</td>
<td>138</td>
<td>0.37</td>
</tr>
<tr>
<td>MatMul</td>
<td>int32</td>
<td>63</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>24</td>
<td>9</td>
<td>96</td>
<td>0.38</td>
</tr>
<tr>
<td>MatMulAdd</td>
<td>int32</td>
<td>72</td>
<td>0</td>
<td>24</td>
<td>9</td>
<td>33</td>
<td>9</td>
<td>114</td>
<td>0.46</td>
</tr>
</tbody>
</table>

replicated; the absence of this notation indicates that the DFG is not replicated.

The process of manually extracting a DFG from a parallel code section—a body of a parallel loop or a body of a data-parallel kernel—is as follows. First, the compute operations within a parallel code section are represented by the DFG’s compute nodes. Next, the data dependencies between the compute operations are represented by the edges between the DFG’s compute nodes. Afterwards, false dependencies (i.e., write-after-read and write-after-write), if they exist, are manually removed by renaming. Input nodes are created for any data that is consumed by the code section (i.e., the data that is defined before the parallel code section). Finally, output nodes are created for any data that is produced by the code section.

**N-Body.** An N-body simulation numerically approximates the evolution of a system of N bodies in which each body continuously interacts with every other body. The N-Body DFG is extracted from a data-parallel kernel in CUDA [131, 132]. The N-Body DFG represents the body of the bodyBodyInteraction function, which is called from the innermost loop of the data-parallel kernel calculate_forces via the tile_calculation function. The bodyBodyInteraction function contains all of the floating point operations that the kernel executes. The N-Body-2x DFG is constructed to compute the interaction of 2
bodies on a third body, while the $N$-Body-$3x$ DFG contains the $N$-Body and $N$-Body-$2x$ DFGs as two connected components.

**BlackScholes.** The Black-Scholes model is a closed-form solution of a partial differential equation that computes call and put option prices for European options [132, 133]. Our BlackScholes DFG represents the body of the BlackScholesBody function, which computes the call and put options for a single input option price. The BlackScholesBody function represents the entire body of the loop in the BlackScholes CUDA kernel. Thus, the BlackScholes DFG executes the same set of the floating-point operations that the CUDA kernel executes.

**MatMulAndAdd.** This DFG performs matrix multiplication and addition on small matrices, or blocks. Specifically, it multiplies blocks $A$ and $B$ and then adds a blocks $C_{in}$ to the product: $C_{out} = A \times B + C_{in}$ where $A$, $B$, $C_{in}$, and $C_{out}$ are $3 \times 4$, $4 \times 3$, $3 \times 3$, and $3 \times 3$ blocks, respectively. The MatMulAndAdd DFG is extracted from a blocked matrix multiplication function, which features 6 nested loops wherein the 3 innermost loops perform matrix multiplication and addition on blocks [134, 135]. All of the compute operations of the matrix multiplication function reside within the innermost loop; and the MatMulAndAdd DFG represents the 3 innermost loops for the case when the sizes of the blocks $A$ and $B$ are $3 \times 4$ and $4 \times 3$, respectively. The DFG is constructed by first fully unrolling the 3 innermost loops and then computing all of the elements of the output block ($C_{out}$) in parallel. This is possible because there are no dependencies between the elements of the output block. Each element of the output block $C_{out}$ is a dot product of two 4-element vectors; these dot products are formed using a reduction topology. There is an integer and a floating-point version of this DFG.

**MatMul.** This DFG is a subset of the MatMulAndAdd DFG in that it performs only the block multiplication, $C_{out} = A \times B$, using the same block sizes. Specifically, the MatMul DFG has 63 compute nodes in total; it has 9 addition operations less than the MatMulAndAdd DFG, which has 73 compute nodes in total. Since the MatMulAndAdd
Chapter 7. Experimental Evaluation

DFG contains all of the compute operations of the 3 innermost loops from the function described above, the *MatMul* DFG contains 88% of those compute operations. There is an integer and a floating-point version of this DFG.

**GaussianBlur.** Gaussian Blur is a 2D convolution with a circularly symmetric filter [136]. For each pixel in the image, it computes a dot product of the 2D filter weights with the input pixels within a 2D window surrounding the output pixel. Our implementation assumes a single colour channel (e.g., grayscale) and that the computation is done in fixed-point number representation with 32-bit integers. The *GaussianBlur* DFG computes one output pixel and uses a $5 \times 5$ filter and a $5 \times 5$ input window. This DFG corresponds to all of the compute operations of the body of the OpenCL kernel that computes a 2D convolution for a single output pixel [137]. The reduction topology of the DFG can be obtained by fully unrolling the kernel’s two nested loops that traverse the 2D window. The *GaussianBlur-2x* DFG computes two adjacent output pixels for a $5 \times 5$ filter and a $6 \times 5$ input window that contains two overlapping $5 \times 5$ input windows.

**SAD.** Sum of absolute differences (SAD) measures the similarity between a block of an input image and a template block [138]. It is computed as a Manhattan distance between the image block and the equally-sized template block. This distance is computed for every block in the image by sliding the template block over the image. The SAD DFG computes a Manhattan distance between a $4 \times 4$ image block and a $4 \times 4$ template block. This DFG corresponds to the compute operations that are obtained by unrolling the 2 innermost loops that compute the distance between two $4 \times 4$ blocks from the SAD OpenCL kernel [139]. We strip out the boundary condition checks and instead assume that the input image is zero-padded (this affects only the blocks on the image boundary). The *SAD-2x* DFG takes as its input a $4 \times 5$ image block that comprises two overlapping $4 \times 4$ blocks and a single $4 \times 4$ template block. The SAD-2x DFG then computes the distance between each block and the template block.

**RGB2YIQ.** The *RGB2YIQ* DFG takes one pixel encoded in RGB format as input
and converts it to the YIQ format. This DFG corresponds to the body of the loop from the RGB-to-YIQ conversion benchmark [140]. The $RGB2YIQ-2x$, $RGB2YIQ-3x$, $RGB2YIQ-4x$, and $RGB2YIQ-5x$ DFGs convert 2, 3, 4, and 5 pixels in parallel, respectively.

### 7.2 Evaluation Platform and Methodology

We use a Stratix IV EP4SGX530 FPGA of the DE4, which is a high-end development board [141]. We use Altera’s Quartus II 12.1sp1 CAD tool to compile the overlays to the FPGA fabric and measure $f_{\text{MAX}}$. Section 6.1 provides the details of the FPGA device and the Quartus II settings we use. We use a machine with an Intel quad-core processor (i5-2500 3.30 GHz) and 24 GB of RAM. Quartus II is configured to use all four cores. We use this compute platform to report the compile time of the three CAD flows and the DFG-to-overlay mapping flow.

As described in Section 4.3, the overlay reconfiguration controller loads the DFG bitstream into the overlay and initiates overlay execution. For the purposes of verifying the execution correctness, an overlay that is under test is surrounded by a test harness that uses additional logic: the overlay’s input ports have data stream generators which feed data into the overlay, and the overlay’s output ports have checksum circuits that verify the correctness of the output data. We first verify the operation and measure the throughput of smaller overlays (4x4 and 8x8 overlay instances) on the board. We then verify that the results and throughput obtained via cycle-accurate RTL simulation of the overlays using Mentor Graphics ModelSim [142] match the board results. We eschew the design of a board test harness that scales to the size of larger overlays. Instead, for larger overlays, we rely on RTL simulation to verify their operation and measure throughput. We also use RTL simulation for the architectural exploration of the overlays (e.g., sweeping of D-FIFO depths).
7.3 Overlay Prototypes

We design two prototype overlays: 1) a 32-bit integer overlay (OV-i), and 2) a 32-bit single-precision floating-point overlay (OV-f). The OV-i and OV-f are designed to be able to execute all integer and floating-point DFGs from Table 7.1, respectively. The functional layouts of OV-i and OV-f are shown in Figures 7.1a and 7.1c, and we elaborate on their design in the remainder of the section.

The design of OV-i and OV-f is tailored for the compilation with the tiled-based bottom-up flow, and their functional layouts are matched to the layout of the FPGA device (this method is described in Section 6.8). That is, their functional layouts match the FPGA resources available in each tile, and DSP-based cells are present only in those tiles that have DSP blocks. The partitioning of the two overlays into tiles is shown in Figures 7.1b and 7.1d. Both overlays are tiled with a combination of 5 × 4 and 4 × 4 tiles. OV-i has a total of 20 tiles and OV-f has a total of 16 tiles.

OV-i comprises four overlay cell types—addsub-abs, mult, shift, and a routing-only cell—that allow for the operations that appear in the integer DFGs. The addsub-abs is a multi-FU overlay cell comprising two FUs: addsub and abs. In OV-i, adders are the highest count FUs, and are followed by multipliers. We match the functional layout of OV-i to the functional layout of the FPGA’s LAB and DSP columns as follows. The columns of LUT-only high-count addsub-abs cells are positioned across the LAB columns, whereas 3 columns of DSP-based mult cells are positioned to match the layout of the 3 rightmost DSP columns of the FPGA device. The low-count LUT-only shift cells are then interspersed within addsub columns.

The overlay cells form several 3 × 2 and 2 × 2 functional groups that are repeated across the OV-i overlay (shown in Figure 7.1a). These cell groups are then interleaved with one row and one column of routing-only cells that serve to increase the routing resources of the overlay (depicted as empty blocks in Figure 7.1).

OV-f consists of nine overlay cell types that allow for the operations that appear in
Figure 7.1: (a) OV-i 24×16 functional layout, (b) OV-i partitioning into 20 tiles, (c) OV-f 18×16 functional layout, (d) OV-f partitioning into 16 tiles.
the floating-point DFGs. These cell types are as follows: \textit{f-addsub}, \textit{f-mult-dsp}, \textit{f-mult-lut}, \textit{f-abs-cmp}, \textit{f-div}, \textit{f-exp}, \textit{f-log}, \textit{f-sqrt}, and a routing-only cell. The \textit{f-abs-cmp} cell is a multi-FU cell that contains \textit{f-abs} and \textit{f-cmp} FUs. As with OV-i, the highest count FUs in OV-f are adders (\textit{f-addsub}) followed by multipliers (\textit{f-mult-dsp} and \textit{f-mult-lut}, described below) as they are the most common operations in the floating-point DFGs.

The functional layout of OV-f resembles that of OV-i. The columns of LUT-only high-count adder cells (\textit{f-addsub}) are positioned across the LAB columns, whereas 3 columns of DSP-based multiplier cells (\textit{f-mult-dsp}) are positioned to match the layout of the 3 rightmost DSP columns of the FPGA device. In order to increase the number of available multipliers, OV-f also has a set of LUT-based multipliers (\textit{f-mult-lut}) along the two columns on the left side of the overlay\(^1\). The low-count LUT-only FUs (\textit{f-abs-cmp} and \textit{f-sqrt}) are interspersed within the \textit{f-addsub} columns whereas the low-count DSP-based FUs (\textit{f-div}, \textit{f-exp} and \textit{f-log}) are interspersed within the \textit{f-mult-dsp} columns.

The overlay cells of OV-f form several \(3 \times 2\) and \(4 \times 2\) functional groups that are repeated across the OV-f overlay (shown in Figure 7.1b). These cell groups are then interleaved with two rows and one column of routing-only cells that serve to increase the routing resources of the overlay. OV-f is designed to have more routing resources than OV-i because of its greater layout heterogeneity-- there are more cell types and some are sparse (i.e., low-count). OV-f’s greater heterogeneity leads to the lower spatial locality of various cell types and requires more routing resources to connect them. For example, in OV-f, the \textit{f-sqrt} cells are only at the bottom, the \textit{f-exp} cells are only in the middle, and the \textit{f-log} cells are only found at the top.

In this work, the overlay’s functional layout design is done manually. Future work can explore automating this process by accounting for the constraints of the FPGA device’s layout and the tile-based compilation flow as well as accounting for the operation

\(^{1}\)The fourth DSP column is at the very left of the FPGA device. Since OV-f uses the 3 rightmost DSP columns, it would have to use the entire FPGA device in order to use the fourth DSP column. OV-f uses 75\% of FPGA device ALM resources, and we provide the details in Section 7.7.
Chapter 7. Experimental Evaluation

<table>
<thead>
<tr>
<th>Overlay</th>
<th>HxW</th>
<th># Cells CR / R / Total</th>
<th>FIFO depth</th>
<th>$f_{\text{MAX}}$</th>
<th>Peak throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>OV-f</td>
<td>18x16</td>
<td>104 / 184 / 288</td>
<td>64</td>
<td>312</td>
<td>32.4 GFLOPS</td>
</tr>
<tr>
<td>OV-i</td>
<td>24x16</td>
<td>170 / 214 / 384</td>
<td>32</td>
<td>355</td>
<td>60.4 GOPS</td>
</tr>
</tbody>
</table>

Table 7.2: Overlay characteristics (CR=compute-routing cells, R=routing-only cells).

requirements of the target DFGs. Furthermore, future work can also investigate the sensitivity of DFG performance on the variations in the overlay’s functional layout.

Table 7.2 summarizes the characteristics of the two overlays. The height ($H$) and width ($W$) reflect the entire overlay as they include both the regular compute-routing cells (i.e., with FUs) and routing-only cells. In the table, the compute-routing cells are denoted with $CR$ and routing-only cells are denoted with $R$.

We design OV-i with 32-slot D-FIFOs and OV-f with 64-slot D-FIFOs at the input ports of their cells. OV-f has larger D-FIFO depths because the floating-point FUs have larger latencies than integer FUs (as described in Section 4.1.1). Consequently, the same imbalance of DFG paths in terms of FU count will result in a larger imbalance in terms of latency on OV-f than it will on OV-i. We explore the impact of D-FIFO depth on throughput in Section 7.6.3.

7.4 Evaluation Metrics

- The first metric we use to evaluate an overlay is its maximum clock frequency, i.e., $f_{\text{MAX}}$. This is determined by the Quartus II static timing analyzer for the slow 85 °C 0.9 V timing model.

- For a DFG mapped to an overlay, we measure its DFG latency (in terms of clocks cycles), which is the time required for data to travel through the DFG, from the overlay’s input ports to its output ports.

- We evaluate the pipelined execution of a DFG on a target overlay using the DFG
throughput \((Th)\), which is defined as the average number of DFG instance completions per cycle. The throughput is computed as follows. First, the time, in terms of the number of clock cycles \(T_{cc}\) needed to execute 65,536 DFG instances \(N_{inst}\), is measured. This number is chosen such that the impact of initial DFG latency\(^2\) on the overall execution time and throughput is negligible. Thus, the measurement of execution time begins with the completion of the first DFG instance (i.e., after the initial DFG latency) and captures the steady state throughput. The execution time is then divided by the number of executed DFG instances to obtain the DFG throughput:

\[
Th = \frac{T_{cc}}{N_{inst}}
\]  

(7.1)

The maximum DFG throughput is 1, i.e., the completion of one DFG instance every cycle. A DFG throughput of 0.5 means that one DFG instance is completed every other cycle. DFG instance completions are measured using performance counters that are located at the overlay’s output ports.

- We calculate the DFG operation throughput \((Th_{OP})\) as the product of the DFG throughput, the number of compute nodes in the DFG \(N_{comp}\), and the overlay’s \(f_{MAX}\). It represents the number of operations executed per unit of time on the overlay and is the overall measure of the DFG’s performance on the overlay. This metric is expressed as:

\[
Th_{OP} = Th \times N_{comp} \times f_{MAX}
\]  

(7.2)

The operation throughput is reported in GOPS (billions of integer operations per second) or GFLOPS (billions of floating-point operations per second), depending

\(^2\)230 cycles on avg., see Section 7.5.3.
on which is appropriate.

- We also measure the DFG-to-Overlay \textit{compile time} of the DFG-to-Overlay mapping tool and report it in seconds. In addition, we measure a set of metrics that characterize a given DFG-to-Overlay mapping. These metrics include the average number of seeds needed to achieve a routable and a balanced mapping, DFG routability, and DFG balancability. We discuss this in detail in Section 7.6.

- Finally, we measure the \textit{FPGA resource usage} for each overlay and its \textit{resource overhead}, which serves as an indication of the overhead associated with using an overlay. The first metric that we measure is the ratio of the overlay’s resource usage to that of the FUs that comprise it, which represents the theoretical minimum amount of resources required to realize the functionality of a DFG that uses all of the overlay’s FUs. The second metric that we measure is the ratio between the resource usage of the overlay’s cells that realize the DFG and the resource usage of a direct DFG implementation in FPGA fabric (i.e., hard-wired DFG circuit without an overlay’s programmability). The direct implementation of the DFGs in FPGA fabric is generated by a DFG-to-FPGA tool that we implement. This tool is described in Appendix A. In addition to resource usage, the DFG-to-FPGA tool allows us to compare the compile time, $f_{\text{MAX}}$, and DFG latency of the DFG-to-overlay and DFG-to-FPGA approaches.

### 7.5 Performance and Scalability

#### 7.5.1 $f_{\text{MAX}}$

OV-i achieves an $f_{\text{MAX}}$ of 355 MHz. The $f_{\text{MAX}}$ of the 20 tiles ($4 \times 5$ and $4 \times 4$ in size) that comprise OV-i ranges from 363 to 408 MHz with an average of 381 MHz. The variability of $f_{\text{MAX}}$ across tiles is caused both by the variability of functional layouts and
Chapter 7. Experimental Evaluation

the variability of FPGA resource layouts. The \( f_{\text{MAX}} \) of OV-i is limited by the \( f_{\text{MAX}} \) of the slowest tile (363 MHz) and is also negatively impacted by the clock skew change during tile stitching (Section 7.9.5 includes a detailed analysis of the bottom-up flow). The critical paths of the slowest tiles of OV-i are between D-FIFOs in neighboring cells with the synchronization logic contributing the most delay.

The tiles comprising OV-i can be viewed as small (4×5 and 4×4) overlays with an average \( f_{\text{MAX}} \) of 381 MHz. In comparison to these small overlays, a 24×16 OV-i has a mesh size that is 19 times larger but only 7% lower \( f_{\text{MAX}} \). This attests to the scalability of our overlay architecture— a 19X increase in mesh size (i.e., scaling up the overlay) comes with only 7% \( f_{\text{MAX}} \) drop.

OV-f achieves an \( f_{\text{MAX}} \) of 312 MHz. The critical paths are contained within the \( f\text{-exp} \) FUs. In addition, three 4×4 tiles of OV-f that contain an \( f\text{-exp} \) FU (see Figure 7.1d) achieve a similar \( f_{\text{MAX}} \) that is limited by the critical paths in the \( f\text{-exp} \) FU. Thus, any tile with the \( f\text{-exp} \) FU and any overlay that is composed of those tiles will have its \( f_{\text{MAX}} \) upper bounded by the \( f\text{-exp} \) FU. Consequently, OV-f achieves the same \( f_{\text{MAX}} \) as a small 4×4 overlay whose FU mix contains the \( f\text{-exp} \) FU (e.g., tiles with \( f\text{-exp} \) described above). In other words, \( f_{\text{MAX}} \) is maintained as the overlay is scaled up 18 times from 4×4 to 18×16 mesh size. Again, this attests to the scalability of the overlay architecture.

7.5.2 Throughput

Table 7.3 shows the achieved throughput and the corresponding operation throughput for each DFG. Each integer DFG is mapped and run on OV-i and each floating point DFG is mapped and run on OV-f. The results are an average of 30 DFG-to-overlay compiles using different starting seeds. The DFG throughput is measured under maximum overlay I/O bandwidth, i.e., data is available (valid) at the overlay input ports every cycle, and the overlay output ports do not stall.

The maximum DFG throughput (\( Th=1 \)) is achieved for all DFGs; that is, in steady-
### Table 7.3: Achieved throughput and latency of the DFGs on OV-i and OV-f.

<table>
<thead>
<tr>
<th>DFG</th>
<th>Overlay</th>
<th>Throughput</th>
<th>GFLOPS/GOPS</th>
<th>DFG latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-Body</td>
<td>OV-f</td>
<td>1</td>
<td>6.2</td>
<td>315</td>
</tr>
<tr>
<td>N-Body-2x</td>
<td>OV-f</td>
<td>1</td>
<td>12.5</td>
<td>365</td>
</tr>
<tr>
<td>N-Body-3x</td>
<td>OV-f</td>
<td>1</td>
<td>18.7</td>
<td>378</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>OV-f</td>
<td>1</td>
<td>21.2</td>
<td>725</td>
</tr>
<tr>
<td>MatMul</td>
<td>OV-f</td>
<td>1</td>
<td>19.7</td>
<td>194</td>
</tr>
<tr>
<td>MatMulAdd</td>
<td>OV-f</td>
<td>1</td>
<td>22.5</td>
<td>218</td>
</tr>
<tr>
<td>RGB2YIQ</td>
<td>OV-i</td>
<td>1</td>
<td>7.5</td>
<td>101</td>
</tr>
<tr>
<td>RGB2YIQ-2x</td>
<td>OV-i</td>
<td>1</td>
<td>14.9</td>
<td>141</td>
</tr>
<tr>
<td>RGB2YIQ-4x</td>
<td>OV-i</td>
<td>1</td>
<td>29.8</td>
<td>149</td>
</tr>
<tr>
<td>RGB2YIQ-5x</td>
<td>OV-i</td>
<td>1</td>
<td>37.3</td>
<td>173</td>
</tr>
<tr>
<td>SAD</td>
<td>OV-i</td>
<td>1</td>
<td>16.7</td>
<td>120</td>
</tr>
<tr>
<td>SAD-2x</td>
<td>OV-i</td>
<td>1</td>
<td>33.4</td>
<td>145</td>
</tr>
<tr>
<td>GaussianBlur</td>
<td>OV-i</td>
<td>1</td>
<td>17.4</td>
<td>124</td>
</tr>
<tr>
<td>GaussianBlur-2x</td>
<td>OV-i</td>
<td>1</td>
<td>34.8</td>
<td>190</td>
</tr>
<tr>
<td>MatMul</td>
<td>OV-i</td>
<td>1</td>
<td>22.4</td>
<td>159</td>
</tr>
<tr>
<td>MatMulAdd</td>
<td>OV-i</td>
<td>1</td>
<td>25.6</td>
<td>182</td>
</tr>
</tbody>
</table>

state pipelined execution, one DFG instance is completed every cycle. Consequently, there are no FU stalls as this would have resulted in less than maximum DFG throughput ($Th<1$).

This result validates the DFG-to-overlay mapping tool, which reports that a balanced mapping was found for all benchmark DFGs. As described earlier, OV-i is designed using D-FIFOs with a depth of 32 and OV-f is designed using D-FIFOs with a depth of 64. Thus, these D-FIFO depths provide a latency-balancing margin that is sufficient to balance the benchmark DFGs. Section 7.6.3 gives a detailed throughput analysis for a range of D-FIFO depths. In addition, it should be noted that four DFGs (three N-Body DFG variants and BlackScholes) require the insertion of balancing nodes in order to achieve a latency balanced mapping on the OV-f overlay. This is elaborated on in Section 7.6.2.

OV-f has an operation throughput of up to 22.5 GFLOPS (achieved by the MatMulAndAdd DFG), and OV-i has an operation throughput of up to 37.3 GOPS (achieved
by the $RGB2YIQ-5x$ DFG).

### 7.5.3 Latency

As described in Chapter 5, the DFG-to-Overlay algorithm optimizes for maximum DFG throughput, potentially at the expense of larger DFG latency. We measure DFG latency in order to characterize the depth of the pipelines that implement the DFG on the overlay. These results are shown in the rightmost column of Table 7.3. As in the throughput experiments, the results are an average of 30 DFG-to-overlay compiles using different starting seeds. The latency of integer DFGs ranges from 101 to 190 clock cycles, while the latency of floating-point DFGs is larger, ranging between 194 and 725 clock cycles. This is both due to the larger latency of floating-point FUs and the deep diverge-converge topology of $BlackScholes$ and $N$-$Body$ DFGs.

In Section 7.8 we compare the DFG latency on the overlay to the latency of direct DFG implementation on the FPGA fabric. These results characterize the “latency overhead” of using the overlay.

### 7.6 DFG-to-Overlay Mapping Tool

In Section 5.2 we described the parameters of the DFG-to-Overlay mapping tool. A summary of the parameters used for this tool is given in Table 7.4. Unless otherwise noted, the set of experiments discussed in this section uses these parameters. Furthermore, the results for a given DFG are an average of 30 runs of the DFG-to-overlay mapping tool. Each run uses a different starting seed, which is provided by an external seed-sweeping script.
### 7.6.1 DFG-to-Overlay Compile time

We first evaluate the DFG-to-overlay mapping tool’s execution time (referred to as the DFG-to-overlay compile time) for the set of DFG benchmarks when they are mapped to the OV-i and OV-f overlays. Table 7.5 shows the DFG-to-Overlay compile times in seconds. In order to characterize our tool’s mapping of the DFGs, we also show the average numbers of PAR loop iterations, balancing loop iterations, and total number of internal seeds. The average number of PAR loop iterations indicates the average number of internal placement seeds that need to be swept in order to find a routable mapping. Similarly, the average number of balancing loop iterations reflects the average number of routable mappings that need to be tested for latency balance in order to find a balanced mapping among them. The total number of PAR seeds measures the total number of internal seeds required to find a balanced mapping.

A balanced mapping for all of the benchmark DFGs is found within the maximum of 40 internal seeds for each run of the DFG-to-Overlay mapping tool. Half of the DFGs compile in under 1 second, while all compile in under 7 seconds.

The integer *MatMulAndAdd* DFG requires the most PAR seeds in total (13.6) to find a balanced mapping, requiring an average of 5 seeds to find a routable mapping and 2.8 routable mappings to find a balanced mapping. Five DFGs (*N-Body-1/2x*, *RGB2YIQ*, *SAD*, and *GaussianBlur*) can be balanced with only 1 PAR seed, which means that they do not rely on the multi-pass strategy of the algorithm. Only four DFGs required more than 4 PAR seeds in total (*FP MatMulAndAdd*, *RGB2YIQ-5x* and integer *MatMul* and...
<table>
<thead>
<tr>
<th>DFG</th>
<th>overlay</th>
<th>PAR loop iter. avg.</th>
<th>balancing loop iter. avg.</th>
<th>total PAR passes (seeds) avg.</th>
<th>total compile time avg. (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-Body</td>
<td>OV-f</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.28</td>
</tr>
<tr>
<td>N-Body-2x</td>
<td>OV-f</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.48</td>
</tr>
<tr>
<td>N-Body-3x</td>
<td>OV-f</td>
<td>1.07</td>
<td>1</td>
<td>1.07</td>
<td>0.78</td>
</tr>
<tr>
<td>Black-Scholes</td>
<td>OV-f</td>
<td>1.27</td>
<td>1.1</td>
<td>1.5</td>
<td>0.97</td>
</tr>
<tr>
<td>MatMul</td>
<td>OV-f</td>
<td>2.18</td>
<td>1.37</td>
<td>3</td>
<td>1.2</td>
</tr>
<tr>
<td>MatMulAndAdd</td>
<td>OV-f</td>
<td>4.78</td>
<td>1.63</td>
<td>7.87</td>
<td>3.17</td>
</tr>
<tr>
<td>RGB2YIQ</td>
<td>OV-i</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.24</td>
</tr>
<tr>
<td>RGB2YIQ-2x</td>
<td>OV-i</td>
<td>1.27</td>
<td>1</td>
<td>1.27</td>
<td>0.45</td>
</tr>
<tr>
<td>RGB2YIQ-4x</td>
<td>OV-i</td>
<td>1.9</td>
<td>1.07</td>
<td>2</td>
<td>1.19</td>
</tr>
<tr>
<td>RGB2YIQ-5x</td>
<td>OV-i</td>
<td>9.53</td>
<td>1.17</td>
<td>10.8</td>
<td>6.73</td>
</tr>
<tr>
<td>SAD</td>
<td>OV-i</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.42</td>
</tr>
<tr>
<td>SAD-2x</td>
<td>OV-i</td>
<td>1.03</td>
<td>1</td>
<td>1.03</td>
<td>0.75</td>
</tr>
<tr>
<td>GaussianBlur</td>
<td>OV-i</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.47</td>
</tr>
<tr>
<td>GaussianBlur-2x</td>
<td>OV-i</td>
<td>3.57</td>
<td>1</td>
<td>3.57</td>
<td>2.1</td>
</tr>
<tr>
<td>MatMul</td>
<td>OV-i</td>
<td>3.89</td>
<td>2.47</td>
<td>9.37</td>
<td>3.3</td>
</tr>
<tr>
<td>MatMulAndAdd</td>
<td>OV-i</td>
<td>5.03</td>
<td>2.8</td>
<td>13.6</td>
<td>5.34</td>
</tr>
</tbody>
</table>

Table 7.5: DFG-to-overlay compile times to OV-f with 64-deep FIFOs, and OV-i with 32-deep FIFOs.

The average number of PAR loop iterations also varies across DFGs. Five DFGs (N-Body-1/2x, RGB2YIQ, SAD and GaussianBlur) can be routed using only one seed on average. GaussianBlur-2x is the hardest DFG to route, requiring an average of 10 internal PAR seeds to find a routable mapping. All of the other DFGs are routable within 5 seeds.

Furthermore, only one balancing loop iteration is required for 9 DFGs. This holds for all N-Body, SAD, and GaussianBlur DFG variants as well as for RGB2YIQ/2x DFGs. All other DFGs require up to 3 balancing loop iterations.

These results validate the multi-pass seed-sweeping strategy of the algorithm, and demonstrate that the algorithm can converge on a balanced mapping relatively quickly. In Section 7.6.4 we perform a DFG routability and balancability study on a larger statistical set of mapping algorithm runs (1000 for each DFG). This study presents additional results...
in favour of the multi-pass strategy.

### 7.6.2 Insertion of Balancing Nodes

As described in Chapter 5, the first step of the DFG-to-overlay mapping algorithm estimates whether the latency margins of the overlay’s D-FIFOs are sufficient to balance the DFG on the overlay. If they are not, then the mapping algorithm inserts balancing nodes into imbalanced DFG paths.

The number of balancing nodes inserted into each DFG by the DFG-to-overlay mapping algorithm is shown in Table 7.1. Four floating-point DFGs (three *N-Body* DFG variants and the *BlackScholes* DFG) have between 17 to 59 balancing nodes inserted. These DFGs require balancing nodes because they themselves are highly imbalanced. This is due to their asymmetric reconverge topology in which some reconverging paths have only one or two compute nodes, while others have ten or more compute nodes. This disparity results in a large latency imbalance between those paths.

The other DFGs have more symmetrical topologies (i.e., there is little or no difference in the number of nodes in their paths), and thus the DFGs themselves have little or no latency imbalance. Correspondingly, the mapping algorithm adds only 1 and 2 balancing nodes to the two *GaussianBlur* DFG variants and estimates that no balancing nodes are required for other DFGs. That is, it estimates that the latency margins accumulated on the routed paths of these DFGs are sufficient for a latency-balanced mapping on the overlay.

### 7.6.3 Impact of the Overlay’s FIFO Depth on DFG Throughput

The results in Section 7.5.2 show that once balancing nodes have been inserted into a DFG, maximum throughput is achieved. This is the case when the D-FIFO depths of OV-i and OV-f are 32 and 64, respectively. In this section, we investigate how using smaller D-FIFO depths affects DFG throughput. Specifically, we perform two experiments. In
Chapter 7. Experimental Evaluation

the first, we use the DFG-to-overlay mapping algorithm to find the smallest balancing FIFO depth for each DFG, i.e., the smallest possible FIFO depth that results in maximum throughput. In the second experiment we validate this result through simulation. We measure DFG throughput for a range of FIFO depths – 4 to 64 – and show that when the FIFO depth is smaller than the smallest balancing FIFO depth the throughput is indeed reduced.

In our first experiment, the mapping algorithm relies on heuristics in the \texttt{FindMostBalancedMapping} function to estimate the smallest balancing D-FIFO depth of a DFG. This is done by configuring the experiment to map the DFGs to OV-i and OV-f that have a D-FIFO depth of 4 (i.e., the smallest D-FIFO depth\(^3\)). We then perform 10 DFG-to-overlay compiles for each DFG. The results show that none of the DFGs can be balanced for a D-FIFO depth of 4. The mapping tool then uses the \texttt{FindMostBalancedMapping} algorithm in each compile and generates up to 30 routable DFG mappings (based on 30 internal placement seeds), selecting the one with the smallest balancing FIFO depth (\(d_{\text{min}}\)). We then average the smallest balancing depths for each DFG across 10 compiles. The results are shown in Figure 7.2.

We can observe that the smallest balancing D-FIFO depth varies across DFGs, ranging from as low as 6.3 for the \textit{SAD-1x} DFG and up to 47 for the \textit{BlackScholes} DFG. An average \(d_{\text{min}}\) of 6.3 for \textit{SAD-1x} indicates that mappings of \textit{SAD-1x} to OV-i instances that are designed using D-FIFOs with a depth that is larger than 6.3 will likely be balanced (i.e., they will achieve \(Th=1\)). DFG mappings on OV-i instances with a FIFO depth smaller than 6.3 will likely not be balanced and will likely achieve less than maximum throughput (\(Th<1\)).

The \(d_{\text{min}}\) ranges from 6.3 to 25.6 for the integer DFGs, and from 28 to 47 for the floating point DFGs. Consequently, the design of OV-i and OV-f with the D-FIFO depths of 32 and 64 enables a wider range of DFGs to achieve maximum throughput. However,

\(^3\)Our D-FIFO implementation inherits the minimum depth limitation of the underlying SCFIFO IP. See Section 4.1.3 for details.
overlays that are customized for specific applications could use lower D-FIFO depths. For example, a custom overlay could be designed for the $RGB2YIQ$, $SAD$, and $GaussianBlur$ DFGs with a D-FIFO depth of only 16 and all three of the DFGs would still achieve maximum DFG throughput.

We also observe that, as DFG size is increased by replication, so too is the smallest balancing FIFO depth. For example, the smallest balancing depth for the three $N$-Body DFG variants increases from 28 to 31 and 40. This also holds for the $RGB2YIQ$, $SAD$ and $GaussianBlur$ DFG variants.

In the second experiment, we validate through simulation that that the DFGs achieve maximum DFG throughput when the overlay’s FIFO depth is equal or larger to the smallest balancing FIFO depth. We also validate that the throughput is lower than maximum when the overlay’s FIFO depth is smaller than the smallest balancing FIFO depth. We also validate that as the overlay’s FIFO depth is further reduced, the throughput of the benchmark DFGs is also reduced. That is, we quantify the throughput of a DFG as a function of the overlay’s FIFO depth.

We first run floating-point DFGs on a set of instances of OV-f that have FIFO depths...
ranging between 4 and 64. We then run integer DFGs on a set of OV-i instances with the same range of FIFO depths. For each overlay’s FIFO depth, we run each DFG 10 times on the overlay and each DFG run uses a different mapping of the DFG (i.e., it uses a different starting seed). We report the average throughput that a DFG achieved across these 10 runs. For clarity, the DFG throughput as a function of overlay’s FIFO depth is shown using four figures that each group the DFGs from the same application. Figure 7.3 shows the results for the \textit{N-Body} DFGs and Figure 7.4 shows the results for the \textit{BlackScholes} and \textit{MatMul} DFGs. The results for the \textit{RGB2YIQ} and \textit{SAD} DFGs are shown in Figure 7.5, while Figure 7.6 shows the results for the \textit{GaussianBlur} and \textit{MatMul} DFGs.

Figure 7.3 shows that the \textit{N-Body} DFG achieves maximum throughput when FIFO depth of OV-f is 32, and that \textit{N-Body-2x} achieves slightly lower than the maximum throughput (0.97). This matches the smallest balancing FIFO depths ($d_{\text{min}}$) of the two DFGs that we measured in the first experiment, 28 and 31, respectively. For a FIFO depth of 16, the DFG throughput for the two DFG is close to 0.6, and it drops to 0.4 and 0.2 for the FIFO depths of 8 and 4.

For \textit{N-Body-3x} we measure an average DFG throughput of 0.96 for a FIFO depth of 48, and an average DFG throughput of 0.77 for a FIFO depth of 32. The $d_{\text{min}}$ of \textit{N-Body-3x} is 40, indicating that for FIFO depths larger than 40, the DFG should achieve maximum throughput. The average DFG throughput for a FIFO depth of 48 that is slightly less than maximum (0.96) occurs because $d_{\text{min}}$ of 40 is an estimate that is obtained as an average across 10 DFG-to-overlay compiles. Specifically, within the set of 10 compiles of \textit{N-Body-3x}, some DFG mappings require a FIFO depth that is larger than 48 to be balanced and achieve maximum throughput. Nevertheless, the average throughput of 0.96 does indicate that most mappings are indeed balanced for a FIFO depth of 48, which validates our heuristics for estimating the smallest balancing FIFO depth.
As shown in Figure 7.4, BlackScholes achieves a similar throughput as N-Body-3x across the measured FIFO depth range. It achieves an average DFG throughput of 0.98 for a FIFO depth of 48, which indicates that most DFG mappings for this FIFO depth are balanced. For FIFO depths lower than 48, the measured throughput is progressively reduced. These results match the estimated $d_{min}$ of 47 that we obtained for the BlackScholes DFG in the previous experiment.

As demonstrated for the floating point DFGs, the measured throughput of integer DFGs also matches the throughput estimated by the average $d_{min}$. Specifically, the
Figure 7.5: Impact of varying the D-FIFO depth of OV-i overlay instances on throughput of $RGB2YIQ$ and $SAD$ DFGs.

Figure 7.6: Impact of varying the D-FIFO depth of OV-i overlay instances on throughput of $GaussianBlur$, $MatMul$ and $MatMulAndAdd$ DFGs.

$RGB2YIQ$ and $SAD$ DFGs have an average $d_{min}$ of less than 8 and achieve the maximum DFG throughput for FIFO depths of 8 and larger. The $GaussianBlur$ DFG has an average $d_{min}$ of 8.4 and achieves an average DFG throughput of 0.98 for a FIFO depth of 8. It achieves maximum throughput for larger FIFO depths. Similar results hold for the rest of the integer DFGs.
Chapter 7. Experimental Evaluation

7.6.4 DFG Routability and Balancability

The DFG-to-overlay algorithm employs a multi-pass strategy that relies on the internal sweeping of placement seeds in order to find routable and balanced mappings. This approach hides the variability of the simulated annealing-based placement from the user. We further validate this algorithm by performing a routability and balancability study of benchmark DFGs using a statistically large set of generated placements (1000 seeds for each DFG). We investigate the percentage of routable and balanced placements within this large sample set and confirm that the algorithm converges on a balanced mapping relatively quickly.

In this experiment, we configure the DFG-to-overlay mapping tool as follows. We use the default set of parameters from Table 7.4, except that we do not limit the number of internal PAR seeds swept in each compile. Instead, we let the tool sweep as many internal placement seeds as necessary to find a balanced mapping\(^4\). Thus, every DFG compile results in a balanced mapping.

We run the DFG-to-overlay mapping tool 1000 times for each DFG, with each run using a random starting seed. Since every DFG compile runs until a balanced mapping is found, the experiment will generate 1000 balanced mappings for each DFG. We also count the total number of placements that are generated for each DFG within those 1000 runs of the tool, as well as the total number of routable placements.

From this data we compute the percentage of placements that are routable and also the percentage of placements that are routable and balanced (i.e., balanced mappings). The remaining routable placements are imbalanced. Figure 7.7 shows the percentage of placements that are routable and breaks them down into balanced and imbalanced portions.

The above figure shows that all placements of the \textit{N-Body} DFG are routable, and that

\(^4\)This is possible because our previous experiment from Section 7.6.1 demonstrated that all DFGs within the benchmark set can converge to a balanced mapping on OV-i and OV-f.
Figure 7.7: Percentage of placements that are routable and their breakdown into balanced routable placements and imbalanced routable placements.

99% of its placements are both routable and balanced. For 7 of the DFGs, 95% of their placements are routable, and 90% are also balanced (all N-Body variants, RGB2YIQ, both SAD variants, and GaussianBlur). Hence, in most cases, these DFGs can be balanced using only 1 internal seed.

Across all DFGs, the integer MatMul DFG has the lowest percentage of balanced placements (8%), with only every 13th placement being balanced. These results validate the multi-pass seed-sweeping strategy employed in the mapping algorithm. The algorithm is likely to find a balanced placement quickly by sweeping through a relatively small number of internal seeds.

We observe several trends in Figure 7.7. For N-Body DFG variants (mapped to OV-f), DFG replication has a minor impact on routability (4% decrease) and balancability (10% decrease). In contrast, the routability and balancability of RGB2YIQ DFG variants (on OV-i) degrade progressively as the DFG is replicated. The non-replicated variant of RGB2YIQ is 98% routable and balanced. The routability drops to 79%, 49%, and 11% for 2x, 4x, and 5x replication factors, respectively. A similar drop is seen for the GaussianBlur-2x DFG when it is compared to the non-replicated GaussianBlur DFG.
Table 7.6: Resource usage of each overlay and resources used only by the FUs of each overlay.

The replication of *RGB2YIQ* and *GaussianBlur* uses up the majority of overlay’s routing resources, which in turn results in a decrease in DFG routability. Thus, DFGs with topologies similar to *RGB2YIQ* and *GaussianBlur* are likely to be less routable on the overlay architecture than DFGs with topologies similar to *N-Body*.

Furthermore, *MatMul* and *MatMulAndAdd* DFGs are more routable on the OV-f than they are on OV-i (the integer and floating-point variants of these DFGs have the same topology). While the functional layout of adders and multipliers is similar in both of the overlays, OV-f has a higher ratio of routing resources, i.e., its ratio of routing-only to compute-routing cells is 1.8. This ratio is 1.3 for OV-i. Thus, the higher ratio of routing-only cells within the overlay contributes to higher DFG routability.

### 7.7 Resource Usage

In this section, we describe the resource usage of the overlay prototypes in terms of ALM resource count. Table 7.6 shows the resources used by each overlay as well as the resources used by only the FUs that comprise each overlay. The table also breaks down the resource usage in terms of ALUTs, flip-flops, DSPs, and memory bits. Both overlays use around 160K ALMs, which is 75% of the FPGA’s total ALMs.

We also measure the ALM usage of the routing logic, pipeline units (DDPUs), and synchronization logic within a compute-routing cell and within a routing-only cell. On average, these components use 440 ALMs in a compute-routing cell (due to the CAD seed noise of packing, clustering, and placement, there are small variations in ALM
usage among the different cells). In comparison, these components use an average of 355 ALMs within a routing-only cell. The ALM usage of these components within a routing-only cell is less than it is in a compute-routing cell because the former does not have DDPUs and muxes that surround the FU. It should also be noted that these ALM counts include the D-FIFOs, which are realized using MLABs.

Furthermore, we measure the resource usage of the FUs used in the overlays. The floating-point FUs consume 453 ALMs on average, while the integer FUs consume 90 ALMs on average. The floating-point FUs also consume a small amount of memory bits, while integer FUs consume none. Specifically, a floating-point divider ($f$-div) contains an internal ROM that is implemented using memory blocks.

### 7.8 Resource Overhead

In this section, we assess the resource overhead of the overlay by comparing the resource usage of implementing the DFGs on our overlays to that of a direct implementation of the DFGs on the FPGA. We refer to the latter approach as the DFG-to-FPGA approach. Fundamentally, the comparison of the overlay approach to alternative approaches for implementing similar functionality is complicated by the fact that resource usage is affected by several factors. They include: the paradigm of describing functionality, design effort, achieved performance, and compile times. Some of these factors are either hard or impractical to quantify, making direct comparisons difficult.

Nonetheless, we believe that the comparison of the DFG-to-overlay to the DFG-to-FPGA approach is fair because it neutralizes most of the factors that affect the resource usage. Specifically, both of the approaches: 1) use DFGs as the method for describing functionality; 2) are optimized for high-throughput (i.e., DFG throughput of 1); and 3) target an $f_{\text{MAX}}$ of 400 MHz. Thus, the difference between these two approaches is reflected primarily in resource usage, compile time, and DFG latency. It is important
to note that because they differ in compile time, the development cycle time is different for the two approaches. However, the two approaches are otherwise similar in terms of design effort because they both aim to achieve the highest DFG throughput.

We first discuss alternative approaches for evaluating the overlay’s overhead and point out the challenges associated with them. Following this, we present a detailed comparison and analysis of the DFG-to-overlay and DFG-to-FPGA approaches in Sections 7.8.2 through 7.8.5.

### 7.8.1 Discussion of Alternative Approaches

There are several alternative approaches to which the DFG-to-overlay approach can be compared to. One alternative is to compare it to a hard-wired circuit that has been manually designed in HDL to implement the same functionality as that of a DFG. However, this comparison entirely ignores designer effort and design time. With HDL, the programmer is responsible for manually pipelining, scheduling operations, balancing, and controlling the pipeline of the FUs. In contrast, the DFG-to-overlay flow performs these tasks automatically. The programmer is only responsible for creating a DFG description. The resource usage of the HDL-based solution— and consequently the overlay’s overhead in comparison to it— is a direct result of the extent of the manual optimizations performed by the HDL designer. Furthermore, a particular HDL implementation may be targeted for a low-area solution and may thus deliberately trade-off throughput for lower resource usage.

Another alternative is to compare the DFG-to-overlay approach to a C-based HLS implementation of an application, which more closely approximates the design effort of the DFG-to-overlay approach. However, the resource usage of a particular HLS implementation is often highly affected by both the coding style and its target throughput.

One naive way to assess the overhead of the overlay’s programmability is by using the ratio of the overlay’s total resource usage to that of the FUs that comprise it. The
Chapter 7. Experimental Evaluation

latter reflects the minimum resource usage of a circuit that implements a DFG that consists of all of the overlay’s FUs. Furthermore, if we assume that a DFG that uses all of the overlay’s FUs can be implemented on the overlay (i.e., it can be routed), then this overlay-to-FU resource usage ratio estimates the overlay’s overhead over a specialized circuit that implements such a DFG.

Referring to Table 7.6, OV-f and OV-i respectively use 3.4x and 10.6x more ALMs than the FUs alone. The higher ratio for OV-i is due to its smaller FUs. These numbers reflect that the overhead can be high, particularly for the integer overlay. However, it is important to note that no realistic circuit can be built from just the FUs. Given this, it is likely that a specialized realistic DFG circuit will consume more resources. On the other hand, a DFG that uses all of the overlay’s FUs may not be routable on the overlay and may require a larger overlay with more routing resources—and consequently more resource overhead. Thus, these ratios are estimates and not a firm upper bound.

7.8.2 Comparison of the DFG-to-overlay and DFG-to-FPGA Approaches

We present the experimental comparison of the DFG-to-overlay approach to a direct DFG implementation on the FPGA using the DFG-to-FPGA tool that we designed. The tool takes a DFG description as input and produces a balanced pipelined circuit that uses “direct” hard-wired connections between the FUs, i.e., without the resources needed for the programmability of the overlay.

The tool first generates Verilog HDL for the DFG’s circuit. The DFG’s circuit is then compiled using Quartus II default flat flow with the same performance optimization settings we use to compile the overlays to the FPGA (as described in Section 6.1). Since the DFG’s circuit is compiled on its own—there is no host interface or memory subsystem—we surround the circuit with virtual pins to eliminate the impact of I/O pin placement and routing. This gives the placer the freedom to search for the most suitable location
for the circuit on the device. A more detailed description of the tool and its design is given in Appendix A.

One important factor to consider when evaluating the resource overhead of an overlay is the extent to which a DFG uses the overlay’s resources. If a DFG uses most of the overlay’s FUs and most of its routing resources, then the FPGA resource usage of the overlay itself accurately reflects the resource usage of the DFG implementation on the overlay. However, this evaluation becomes more challenging if a DFG is small and uses only a fraction of the overlay’s resources, or if a DFG is large but easy to route and uses only a fraction of the overlay’s routing resources.

Thus, we opt for a methodology that measures DFG-on-overlay resources at the granularity of overlay cells. This is done by adding up the ALM resources of only those overlay cells that are used for the mapping of a particular DFG on the overlay. Specifically, for any overlay cell whose FU is utilized by the DFG’s mapping, we count the ALM resources of the entire cell. For any routing-only overlay cell (w/o an FU) that has any of its routing hops utilized by the DFG’s mapping, we count the ALM resources of the entire routing-only cell. For any overlay cell whose FU is not utilized by the DFG’s mapping, but at least one of its routing hops is, we count all the ALM resources that implement the routing logic of the cell. This is equivalent to counting this cell as a routing-only cell.

Finally, for any cell whose FU is configured as NOP and serves as a balancing node (as inserted by the balancing node insertion algorithm), we count the ALMs of the entire cell (equivalent to a compute-routing cell). Note that the balancing node insertion algorithm also maps balancing nodes to routing-only cells. These cells are subsumed into the ALM count of routing-only cells, as was described earlier.

We term compute-routing cells whose FU is utilized by the DFG’s mapping (either as an operation or a NOP) compute-routing mode cells. Compute-routing cells whose FU is not utilized for the DFG’s mapping and routing-only cells are referred to as routing-only mode cells. In other words, any cell that is used only as a routing resource is referred to
as a routing-only mode cell.

An example of compute-routing cells being used in routing-only mode occurs in the mapping of the floating-point MatMul DFG to OV-f. The DFG requires only add and mult FU types, but the mapping of the DFG to the overlay may use the routing resources of exp and log cells. However, the actual FUs of these cell types are unusable for the mapping of the MatMul DFG; they appear as routing-only cells to the mapping algorithm, i.e., they are usable only in routing-only mode. Thus, for the purpose of resource-usage measurement, the overlay can be considered as having routing-only cells instead of exp and log cells.

The DFG-to-FPGA mapping results are presented in Table 7.7. We measure five metrics: DFG instance throughput, resource usage in terms of ALMs, the latency of a single DFG instance execution in clock cycles (DFG latency), $f_{\text{MAX}}$, and compile time. The results are an average of 5 compiles with different seeds. The DFG instance throughput is 1 for all DFGs. This is the main optimization goal of the DFG-to-FPGA tool, which always generates a balanced DFG pipeline. The resource usage varies from 3.6K ALMs for RGB2YIQ (the smallest integer DFG) to 38K for MatMulAndAdd (the largest floating point DFG). We can observe that the floating-point variant of MatMul is 3 times larger than the integer variant: although the structure of the two circuits is similar, the floating-point FUs are larger and have longer latencies.

In comparison to the OV-i and OV-f overlays which each consume 160K ALMs, the hard-wired circuits of the DFGs are smaller and thus easier to optimize with the default Quartus II flat flow. The smaller size of the DFG circuits (relative to the overlay size), combined with the DFG circuit’s elastic pipelines (as generated by the DFG-to-FPGA tool), enables them to achieve an $f_{\text{MAX}}$ of between 326 and 496 MHz. The average $f_{\text{MAX}}$ across the circuits is 405 MHz. This validates the design of the DFG-to-FPGA tool.

The circuits for the floating-point DFGs have longer DFG latencies than the integer
### Table 7.7: DFGs mapped directly to FPGA fabric using the DFG-to-FPGA flow (DFG-to-HDL and Quartus II tools).

<table>
<thead>
<tr>
<th>DFG</th>
<th>DFG instance throughput</th>
<th>resources (ALMs)</th>
<th>DFG latency (clock cycles)</th>
<th>$f_{\text{MAX}}$ (MHz)</th>
<th>compile time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-Body</td>
<td>1</td>
<td>11187</td>
<td>178</td>
<td>356</td>
<td>693</td>
</tr>
<tr>
<td>N-Body-2x</td>
<td>1</td>
<td>22301</td>
<td>194</td>
<td>354</td>
<td>1329</td>
</tr>
<tr>
<td>N-Body-3x</td>
<td>1</td>
<td>33794</td>
<td>194</td>
<td>357</td>
<td>2125</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>1</td>
<td>34066</td>
<td>373</td>
<td>326</td>
<td>2433</td>
</tr>
<tr>
<td>MatMul</td>
<td>1</td>
<td>31290</td>
<td>48</td>
<td>401</td>
<td>2454</td>
</tr>
<tr>
<td>MatMulAdd</td>
<td>1</td>
<td>37756</td>
<td>64</td>
<td>402</td>
<td>2913</td>
</tr>
<tr>
<td>RGB2YIQ</td>
<td>1</td>
<td>3666</td>
<td>29</td>
<td>446</td>
<td>313</td>
</tr>
<tr>
<td>RGB2YIQ-2x</td>
<td>1</td>
<td>6648</td>
<td>29</td>
<td>433</td>
<td>538</td>
</tr>
<tr>
<td>RGB2YIQ-4x</td>
<td>1</td>
<td>13422</td>
<td>29</td>
<td>424</td>
<td>1126</td>
</tr>
<tr>
<td>RGB2YIQ-5x</td>
<td>1</td>
<td>17232</td>
<td>29</td>
<td>415</td>
<td>1367</td>
</tr>
<tr>
<td>SAD</td>
<td>1</td>
<td>8535</td>
<td>32</td>
<td>496</td>
<td>769</td>
</tr>
<tr>
<td>SAD-2x</td>
<td>1</td>
<td>16119</td>
<td>32</td>
<td>489</td>
<td>1454</td>
</tr>
<tr>
<td>GaussianBlur</td>
<td>1</td>
<td>8016</td>
<td>38</td>
<td>394</td>
<td>822</td>
</tr>
<tr>
<td>GaussianBlur-2x</td>
<td>1</td>
<td>15372</td>
<td>38</td>
<td>348</td>
<td>1537</td>
</tr>
<tr>
<td>MatMul</td>
<td>1</td>
<td>10316</td>
<td>18</td>
<td>419</td>
<td>857</td>
</tr>
<tr>
<td>MatMulAdd</td>
<td>1</td>
<td>12017</td>
<td>23</td>
<td>431</td>
<td>1061</td>
</tr>
</tbody>
</table>

circuits due to the longer latencies of their FUs and deeper DFG pipeline (e.g., *N-Body* variants and *BlackScholes* DFGs). The compile time of the DFGs ranges from 5 minutes for *RGB2YIQ* (the smallest DFG) to 49 minutes for the floating-point *MatMulAndAdd* (the largest DFG). The compile time of the floating-point variant of the *MatMul* DFG is 3x longer than it is for the integer variant. This corresponds to the ratio between the size of the two circuits (in terms of ALMs).

Table 7.8 shows the results generated by the DFG-to-overlay tool for the same five metrics, while using the tool’s default parameters given in Table 7.4. These results are again averaged over 30 compiles with different start seeds. Note that the mapping tool is allowed to map to any cell on the overlay by default. Thus, the overlay region’s listed height and width in the table correspond to the full overlay height and width.

Furthermore, we compare the DFG-to-overlay tool to the DFG-to-FPGA tool across the five metrics in Table 7.9. The two tools are equal in terms of DFG instance throughput—they generate a balanced mapping and have throughput of 1 DFG instance per cycle. Since the overlays have a fixed frequency, all floating-point DFGs run at 312 MHz, while all integer DFGs run at 355 MHz—an average of 339 MHz. This is 16% lower than the
Chapter 7. Experimental Evaluation

Table 7.8: DFGs mapped to the entire overlay mesh using the DFG-to-Overlay flow.

<table>
<thead>
<tr>
<th>DFG</th>
<th>overlay region height</th>
<th>overlay region width</th>
<th>DFG instance throughput</th>
<th>resources (ALMs)</th>
<th>latency (cycles)</th>
<th>$f_{\text{MAX}}$ (MHz)</th>
<th>compile time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-Body</td>
<td>OV-f 18 16</td>
<td>1</td>
<td>51381</td>
<td>315</td>
<td>312</td>
<td>0.28</td>
<td></td>
</tr>
<tr>
<td>N-Body-2x</td>
<td>OV-f 18 16</td>
<td>1</td>
<td>95065</td>
<td>365</td>
<td>312</td>
<td>0.48</td>
<td></td>
</tr>
<tr>
<td>N-Body-3x</td>
<td>OV-f 18 16</td>
<td>1</td>
<td>134046</td>
<td>378</td>
<td>312</td>
<td>0.78</td>
<td></td>
</tr>
<tr>
<td>BlackScholes</td>
<td>OV-f 18 16</td>
<td>1</td>
<td>135588</td>
<td>725</td>
<td>312</td>
<td>0.97</td>
<td></td>
</tr>
<tr>
<td>MatMul</td>
<td>OV-f 18 16</td>
<td>1</td>
<td>133498</td>
<td>194</td>
<td>312</td>
<td>1.2</td>
<td></td>
</tr>
<tr>
<td>MatMulAndAdd</td>
<td>OV-f 18 16</td>
<td>1</td>
<td>143404</td>
<td>218</td>
<td>312</td>
<td>3.17</td>
<td></td>
</tr>
<tr>
<td>RGB2YIQ</td>
<td>OV-i 24 16</td>
<td>1</td>
<td>30111</td>
<td>101</td>
<td>355</td>
<td>0.24</td>
<td></td>
</tr>
<tr>
<td>RGB2YIQ-2x</td>
<td>OV-i 24 16</td>
<td>1</td>
<td>67254</td>
<td>141</td>
<td>355</td>
<td>0.45</td>
<td></td>
</tr>
<tr>
<td>RGB2YIQ-4x</td>
<td>OV-i 24 16</td>
<td>1</td>
<td>121073</td>
<td>149</td>
<td>355</td>
<td>1.19</td>
<td></td>
</tr>
<tr>
<td>RGB2YIQ-5x</td>
<td>OV-i 24 16</td>
<td>1</td>
<td>144552</td>
<td>173</td>
<td>355</td>
<td>6.73</td>
<td></td>
</tr>
<tr>
<td>SAD</td>
<td>OV-i 24 16</td>
<td>1</td>
<td>71203</td>
<td>120</td>
<td>355</td>
<td>0.42</td>
<td></td>
</tr>
<tr>
<td>SAD-2x</td>
<td>OV-i 24 16</td>
<td>1</td>
<td>121472</td>
<td>145</td>
<td>355</td>
<td>0.75</td>
<td></td>
</tr>
<tr>
<td>GaussianBlur</td>
<td>OV-i 24 16</td>
<td>1</td>
<td>82012</td>
<td>124</td>
<td>355</td>
<td>0.47</td>
<td></td>
</tr>
<tr>
<td>GaussianBlur-2x</td>
<td>OV-i 24 16</td>
<td>1</td>
<td>134446</td>
<td>190</td>
<td>355</td>
<td>2.1</td>
<td></td>
</tr>
<tr>
<td>MatMul</td>
<td>OV-i 24 16</td>
<td>1</td>
<td>111440</td>
<td>159</td>
<td>355</td>
<td>3.3</td>
<td></td>
</tr>
<tr>
<td>MatMulAndAdd</td>
<td>OV-i 24 16</td>
<td>1</td>
<td>119731</td>
<td>182</td>
<td>355</td>
<td>5.34</td>
<td></td>
</tr>
</tbody>
</table>

The average DFG-to-FPGA $f_{\text{MAX}}$ of 405 MHz. The DFG-to-overlay tool has at most 22% lower $f_{\text{MAX}}$ (for MatMulAndAdd) and up to 2% higher $f_{\text{MAX}}$ (for GaussianBlur-2x).

The DFG-to-overlay tool generates mappings with much higher DFG latency. Indeed, the DFG latency is up to 2x higher for N-Body variants and BlackScholes, while for the rest of the DFGs it ranges between 3.4x to 8.8x higher. This is expected because the routing paths between the FUs on the overlay can have high latency due to several factors: 1) the overlay has deeply pipelined routing fabric; 2) the tool optimizes primarily for throughput, at the expense of latency; 3) the shortest distance between two connected FUs may be several routing hops because of interspersed routing-only cells; and 4) if a DFG is harder to route, then the number of routing hops between FUs is further increased due to congestion.

The ALM resource usage of the DFGs on the overlay is obtained using the methodology we described earlier in this section. The resource usage ranges from 30K ALMs for RGB2YIQ to 135K ALMs for BlackScholes. The resource overhead for the floating-point DFGs ranges from 3.8x to 4.6x, while it ranges from 7.5x to 10.8x for the integer DFGs. We analyze the resource overhead in detail in Section 7.8.3.

The compile time to the overlay for the floating point DFGs is three orders of magnitude faster (from 900x to 2800x) than the compile time to the FPGA. For the inte-
Table 7.9: Comparison of DFG-to-Overlay and DFG-to-FPGA approaches (DFGs are mapped to the entire mesh of the overlay).

<table>
<thead>
<tr>
<th>DFG</th>
<th>DFG inst. th.</th>
<th>resources</th>
<th>latency</th>
<th>f_{MAX}</th>
<th>compile time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>overlay /</td>
<td>overlay /</td>
<td>overlay /</td>
<td>overlay /</td>
<td>FPGA /</td>
</tr>
<tr>
<td></td>
<td>FPGA</td>
<td>FPGA</td>
<td>FPGA</td>
<td>FPGA</td>
<td>overlay</td>
</tr>
<tr>
<td>N-Body</td>
<td>1</td>
<td>4.59</td>
<td>1.77</td>
<td>0.88</td>
<td>2475</td>
</tr>
<tr>
<td>N-Body-2x</td>
<td>1</td>
<td>4.26</td>
<td>1.88</td>
<td>0.88</td>
<td>2769</td>
</tr>
<tr>
<td>N-Body-3x</td>
<td>1</td>
<td>3.97</td>
<td>1.95</td>
<td>0.87</td>
<td>2724</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>1</td>
<td>3.98</td>
<td>1.94</td>
<td>0.96</td>
<td>2508</td>
</tr>
<tr>
<td>MatMul</td>
<td>1</td>
<td>4.27</td>
<td>4.04</td>
<td>0.78</td>
<td>2045</td>
</tr>
<tr>
<td>MatMulAndAdd</td>
<td>1</td>
<td>3.80</td>
<td>3.41</td>
<td>0.78</td>
<td>919</td>
</tr>
<tr>
<td>RGB2YIQ</td>
<td>1</td>
<td>8.21</td>
<td>3.48</td>
<td>0.80</td>
<td>1304</td>
</tr>
<tr>
<td>RGB2YIQ-2x</td>
<td>1</td>
<td>10.12</td>
<td>4.86</td>
<td>0.82</td>
<td>1196</td>
</tr>
<tr>
<td>RGB2YIQ-4x</td>
<td>1</td>
<td>9.02</td>
<td>5.14</td>
<td>0.84</td>
<td>946</td>
</tr>
<tr>
<td>RGB2YIQ-5x</td>
<td>1</td>
<td>8.39</td>
<td>5.97</td>
<td>0.86</td>
<td>203</td>
</tr>
<tr>
<td>SAD</td>
<td>1</td>
<td>8.34</td>
<td>3.75</td>
<td>0.72</td>
<td>1831</td>
</tr>
<tr>
<td>SAD-2x</td>
<td>1</td>
<td>7.54</td>
<td>4.53</td>
<td>0.73</td>
<td>1939</td>
</tr>
<tr>
<td>GaussianBlur</td>
<td>1</td>
<td>10.23</td>
<td>3.26</td>
<td>0.90</td>
<td>1749</td>
</tr>
<tr>
<td>GaussianBlur-2x</td>
<td>1</td>
<td>8.75</td>
<td>5.00</td>
<td>1.02</td>
<td>732</td>
</tr>
<tr>
<td>MatMul</td>
<td>1</td>
<td>10.77</td>
<td>8.83</td>
<td>0.85</td>
<td>260</td>
</tr>
<tr>
<td>MatMulAndAdd</td>
<td>1</td>
<td>9.96</td>
<td>7.91</td>
<td>0.82</td>
<td>199</td>
</tr>
</tbody>
</table>

In this section, we analyze the results presented in the previous section in order to determine the key factors that contribute to the resource overhead.

7.8.3 Analysis

In this section, we analyze the results presented in the previous section in order to determine the key factors that contribute to the resource overhead.
Table 7.9 shows that floating-point DFGs incur less overhead than integer DFGs. The floating-point variants of *MatMul* and *MatMulAndAdd* incur overhead of 4.3x and 3.8x, respectively, while their integer variants incur 10.8x and 10x, respectively. The breakdown of their overlay cell usage in terms of compute-routing and routing-only modes is shown in Table 7.10. We notice that the floating-point *MatMul* uses only 16% fewer routing-only mode cells than the integer variant but incurs 60% less overhead. This is because the integer FUs are on average 5x smaller than the floating-point FUs. Even though the number of used routing-only mode cells is similar in both variants of *MatMul*, the programmable routing of the integer overlay contributes relatively more to the total DFG’s resource usage due to the smaller FUs of the integer overlay. This results in larger resource overhead for the DFGs implemented on the integer overlay.

Furthermore, the results show that in the majority of cases the smaller DFG variants incur higher overhead than their larger counterparts. For example, *N-Body* and *N-Body-2x* incur more overhead than *N-Body-3x*. Similarly, *MatMul* incurs more overhead than *MatMulAndAdd*, and *RGB2YIQ-2x* and *RGB2YIQ-4x* incur more overhead than *RGB2YIQ-5x*. The breakdown of cell usage suggests that the larger overhead of smaller DFG variants is due to the larger relative number of routing-only cells used. For example, *GaussianBlur* uses 41% of all of the overlay’s cells in routing-only mode, whereas *GaussianBlur-2x* uses 60% but it has twice the compute nodes of *GaussianBlur*. We capture the relationship between the resource overhead and the usage of routing-only cells as the ratio of cells in routing-only mode to cells in compute-routing mode (the rightmost column in Table 7.9). For example, *GaussianBlur* has a ratio of 3.2 whereas *GaussianBlur-2x* has a ratio of 2.3. Moreover, the results show that a higher RO-to-CR ratio corresponds to a higher resource overhead across all comparisons of small vs. large DFG variants.

The larger relative usage of routing-only cells in smaller DFG variants is a combination of two factors. First, the benchmark DFGs have a large number of input nodes, which
Table 7.10: DFGs mapped to entire overlay mesh. It shows: the percentage of ALMs of the entire OV-f/i overlays, the usage of compute-routing and routing-only mode cells and the total usage of cells. Also shown is the ratio of routing-only and compute-routing cells.

is partly due to a large number of input constants (they are fed to the FUs and routed in the same manner as regular input nodes). Furthermore, there are several cases where the smaller DFG variant has a larger input-to-compute node ratio (e.g., this ratio is 0.68 for SAD vs. 0.38 for SAD-2x).

Second, a small DFG with a large input-to-compute node ratio that is mapped to a large overlay will be “pulled apart” in order to route to all of the required input ports. For example, consider the mapping of GaussianBlur (49 compute and 31 input nodes) to a large hypothetical overlay (e.g., 64x64) with an FU layout that has been customized for the DFG. In an ideal case, the compute nodes could be placed on a 7x7 region in one of the overlay corners. Similarly, an ideal mapping of 31 input nodes is 15 and 16 nodes to each side of the overlay corner. Thus, the input nodes span a larger 15x16 region within the hypothetical overlay, which requires routing paths that are well beyond the 7x7 region. In contrast, a large DFG that fills out the overlay can use short routing paths to connect to input ports on all four overlay sides. This analysis suggests that the overhead associated with implementing a DFG on the overlay depends on the overlay’s size, even if the overlay’s functional layout has been customized to the DFG.
7.8.4 Constraining DFG Mapping to Smaller Overlays

In order to evaluate the impact of overlay size on resource overhead we map the benchmark DFGs to smaller overlays that correspond to rectangular regions of the OV-i and OV-f overlays. Each small overlay is created by taking a functional layout of a rectangular region of a larger base overlay (OV-i or OV-f) and then making that region an overlay in itself, i.e., by surrounding it with I/O ports. In contrast, within the base overlay, that same region is surrounded with neighbouring overlay cells. Mapping a DFG to a smaller overlay allows shorter routing to the overlay’s I/O, thus eliminating the “pull apart” effect of the I/O when mapped to a larger base overlay. We manually create a smaller overlay for each DFG by choosing the smallest rectangular region– positioned anywhere in the base overlay, not necessarily a corner region– to which the DFG can be placed and routed.

Table 7.11 shows the characteristics of the smaller overlays: the base overlay from which the region was extracted (OV-i/OV-f), the top-left coordinates of the region, and also the height and width of the region (i.e., the height and width of the smaller overlay). The table also shows the results of mapping the DFGs to those smaller overlays.

Half of the DFGs require the entire overlay layout and do not fit on any subregion of the overlay: \textit{N-Body-3x}, \textit{BlackScholes}, \textit{MatMul}, and \textit{MatMulAndAdd} require the entire 18x16 OV-f, and \textit{RGB2YIQ-fx/5x}, and \textit{integer MatMulAndAdd} require the entire 24x16 OV-i. The rest of the DFGs fit on smaller overlays. For example, \textit{N-Body} fits on a 10x8 region of OV-f (1/4 of its layout), while \textit{RGB2YIQ} fits on a 7x7 region of OV-i (1/8 of its layout).

We assume that the $f_{\text{MAX}}$ of smaller overlays is no more than the $f_{\text{MAX}}$ of OV-f and OV-i. Our evaluation of overlay $f_{\text{MAX}}$ across a range of mesh sizes in Section 7.9 shows that this is a conservative estimate. Even though the tile-based bottom-up flow achieves higher $f_{\text{MAX}}$ for large overlays than the other flows, the smaller overlays can still achieve higher $f_{\text{MAX}}$. In terms of compile time, there are negligible differences between mapping
Table 7.11: DFGs mapped to the smaller overlays that correspond to regions of the base overlays (OV-f/OV-i).

to the smaller overlays and to the entire overlay mesh. The DFG latencies are 10% shorter for some DFGs on the smaller overlays.

The reduction in the DFG resource usage on the smaller overlays compared to their usage on the entire base overlays is more significant. Table 7.12 shows the DFG resource overhead on the smaller overlays with respect to the direct DFG implementation on the FPGA. The analysis of cell usage in terms of compute-routing and routing-only modes is shown in Table 7.13. Additionally, Table 7.14 compares the overhead of mapping the DFGs to the entire base overlays versus mapping to the smaller overlays.

On a smaller 10x8 overlay, N-Body incurs 3.34x overhead with respect to the direct implementation on the FPGA, which is 27% overhead reduction compared to the overhead of mapping to the entire base overlay (4.59x). Similarly, on a 7x7 overlay, RGB2YIQ exhibits 5.24x resource overhead, a 36% reduction compared to the overhead of mapping to the entire base overlay (8.21x). Furthermore, in almost all cases of mapping the DFGs to the smaller overlays, the smaller DFG variants exhibit less overhead than the larger DFG variants, which is the inverse of what occurs when the DFGs are mapped to the entire base overlays. Table 7.13 demonstrates that the lower overhead of the small DFG variants compared to the large DFG variants is due to the lower relative number of
### Chapter 7. Experimental Evaluation

#### Table 7.12: Comparison of DFG-to-Overlay and DFG-to-FPGA approaches (DFGs are mapped to the smaller overlays).

<table>
<thead>
<tr>
<th>DFG</th>
<th>DFG inst. th. overlay / FPGA</th>
<th>resources overlay / FPGA</th>
<th>latency overlay / FPGA</th>
<th>$f_{\text{MAX}}$ overlay / FPGA</th>
<th>compile time FPGA / overlay</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-Body</td>
<td>1</td>
<td>3.34</td>
<td>1.63</td>
<td>0.88</td>
<td>2665</td>
</tr>
<tr>
<td>N-Body-2x</td>
<td>1</td>
<td>3.72</td>
<td>1.82</td>
<td>0.88</td>
<td>2556</td>
</tr>
<tr>
<td>N-Body-3x</td>
<td>1</td>
<td>3.97</td>
<td>1.95</td>
<td>0.87</td>
<td>2724</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>1</td>
<td>3.98</td>
<td>1.94</td>
<td>0.96</td>
<td>2508</td>
</tr>
<tr>
<td>MatMul</td>
<td>1</td>
<td>4.27</td>
<td>4.04</td>
<td>0.78</td>
<td>2045</td>
</tr>
<tr>
<td>MatMul and Add</td>
<td>1</td>
<td>3.8</td>
<td>3.41</td>
<td>0.78</td>
<td>919</td>
</tr>
<tr>
<td>RGB2YIQ</td>
<td>1</td>
<td>5.24</td>
<td>2.76</td>
<td>0.8</td>
<td>1423</td>
</tr>
<tr>
<td>RGB2YIQ-2x</td>
<td>1</td>
<td>8.02</td>
<td>4.14</td>
<td>0.82</td>
<td>1251</td>
</tr>
<tr>
<td>RGB2YIQ-4x</td>
<td>1</td>
<td>9.02</td>
<td>5.14</td>
<td>0.84</td>
<td>946</td>
</tr>
<tr>
<td>RGB2YIQ-5x</td>
<td>1</td>
<td>8.39</td>
<td>5.97</td>
<td>0.86</td>
<td>203</td>
</tr>
<tr>
<td>SAD</td>
<td>1</td>
<td>5.12</td>
<td>3.16</td>
<td>0.72</td>
<td>2024</td>
</tr>
<tr>
<td>SAD-2x</td>
<td>1</td>
<td>6.18</td>
<td>4.16</td>
<td>0.73</td>
<td>989</td>
</tr>
<tr>
<td>GaussianBlur</td>
<td>1</td>
<td>7.04</td>
<td>3.03</td>
<td>0.9</td>
<td>1868</td>
</tr>
<tr>
<td>GaussianBlur-2x</td>
<td>1</td>
<td>8.52</td>
<td>4.5</td>
<td>1.02</td>
<td>708</td>
</tr>
<tr>
<td>MatMul</td>
<td>1</td>
<td>9.91</td>
<td>8.39</td>
<td>0.85</td>
<td>268</td>
</tr>
<tr>
<td>MatMul and Add</td>
<td>1</td>
<td>9.96</td>
<td>7.91</td>
<td>0.82</td>
<td>199</td>
</tr>
</tbody>
</table>

#### Table 7.13: DFGs mapped to the smaller overlays. The DFG resource usage on the smaller overlays in terms of ALMs and as a percentage of the entire base overlay (OV-f/OV-i), the usage of compute-routing and routing-only mode cells in the region and the total usage of overlay cells in the region. Also shown is the ratio of routing-only and compute-routing cells.

<table>
<thead>
<tr>
<th>DFG</th>
<th>ALMs</th>
<th>% of OV-f/i ALMs</th>
<th>cells CR mode (used)</th>
<th>% of CR cells (region)</th>
<th>cells RO mode (used)</th>
<th>% of RO cells (region)</th>
<th>total cells (used)</th>
<th>% of all cells (region)</th>
<th>RO / CR</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-Body</td>
<td>37357</td>
<td>23</td>
<td>25</td>
<td>69</td>
<td>34</td>
<td>43</td>
<td>59</td>
<td>74</td>
<td>1.4</td>
</tr>
<tr>
<td>N-Body-2x</td>
<td>82921</td>
<td>51</td>
<td>50</td>
<td>63</td>
<td>88</td>
<td>52</td>
<td>138</td>
<td>82</td>
<td>1.8</td>
</tr>
<tr>
<td>N-Body-3x</td>
<td>134946</td>
<td>83</td>
<td>75</td>
<td>72</td>
<td>156</td>
<td>54</td>
<td>231</td>
<td>80</td>
<td>2.1</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>135588</td>
<td>84</td>
<td>80</td>
<td>77</td>
<td>155</td>
<td>54</td>
<td>235</td>
<td>82</td>
<td>1.9</td>
</tr>
<tr>
<td>MatMul</td>
<td>133498</td>
<td>83</td>
<td>63</td>
<td>61</td>
<td>189</td>
<td>66</td>
<td>252</td>
<td>88</td>
<td>3</td>
</tr>
<tr>
<td>MatMul and Add</td>
<td>144304</td>
<td>89</td>
<td>72</td>
<td>69</td>
<td>191</td>
<td>66</td>
<td>263</td>
<td>91</td>
<td>2.7</td>
</tr>
<tr>
<td>RGB2YIQ</td>
<td>19215</td>
<td>12</td>
<td>21</td>
<td>84</td>
<td>22</td>
<td>45</td>
<td>43</td>
<td>88</td>
<td>1</td>
</tr>
<tr>
<td>RGB2YIQ-2x</td>
<td>53316</td>
<td>33</td>
<td>42</td>
<td>60</td>
<td>88</td>
<td>55</td>
<td>130</td>
<td>81</td>
<td>2.1</td>
</tr>
<tr>
<td>RGB2YIQ-4x</td>
<td>121073</td>
<td>75</td>
<td>84</td>
<td>49</td>
<td>218</td>
<td>57</td>
<td>302</td>
<td>79</td>
<td>2.6</td>
</tr>
<tr>
<td>RGB2YIQ-5x</td>
<td>144552</td>
<td>89</td>
<td>105</td>
<td>62</td>
<td>252</td>
<td>66</td>
<td>358</td>
<td>93</td>
<td>2.4</td>
</tr>
<tr>
<td>SAD</td>
<td>43718</td>
<td>27</td>
<td>47</td>
<td>72</td>
<td>47</td>
<td>39</td>
<td>94</td>
<td>79</td>
<td>1</td>
</tr>
<tr>
<td>SAD-2x</td>
<td>99658</td>
<td>61</td>
<td>94</td>
<td>70</td>
<td>130</td>
<td>50</td>
<td>224</td>
<td>86</td>
<td>1.4</td>
</tr>
<tr>
<td>GaussianBlur</td>
<td>56470</td>
<td>35</td>
<td>50</td>
<td>52</td>
<td>85</td>
<td>52</td>
<td>135</td>
<td>82</td>
<td>1.7</td>
</tr>
<tr>
<td>GaussianBlur-2x</td>
<td>130959</td>
<td>81</td>
<td>100</td>
<td>59</td>
<td>222</td>
<td>62</td>
<td>322</td>
<td>89</td>
<td>2.2</td>
</tr>
<tr>
<td>MatMul</td>
<td>102257</td>
<td>63</td>
<td>63</td>
<td>42</td>
<td>198</td>
<td>63</td>
<td>261</td>
<td>83</td>
<td>3.1</td>
</tr>
<tr>
<td>MatMul and Add</td>
<td>119731</td>
<td>74</td>
<td>72</td>
<td>42</td>
<td>234</td>
<td>61</td>
<td>306</td>
<td>80</td>
<td>3.3</td>
</tr>
</tbody>
</table>
routing-only cells used. This experiment also shows that mapping a DFG to an overlay that matches it in size results in the high utilization of cells within the overlay. This utilization ranges from 74% for $N$-Body to 93% for $RGB2YIQ-5x$.

Table 7.14 compares the number of compute-routing cells and routing-only mode cells in two DFG-to-overlay mapping experiments. We can observe that mapping the DFGs to the smaller overlays results in a large reduction in the use of routing-only mode cells. The use of compute-routing cells is equal in both cases. The largest reduction in routing-only cells is for the 3 smallest DFGs ($N$-Body, $RGB2YIQ$, and $SAD$), which use 2x, 2.5x, and 2.7x fewer routing-only cells on the smaller overlays, respectively. We can also observe that the reduction in total cell usage closely corresponds to the reduction in ALM usage.

These results show that resource overhead increases when the overlay size is too large and does not match the DFG in size. This is for two reasons: first, the “pull apart” effect of the overlay’s I/O, which we described earlier; and second, too many routing-only cells within the overlay actually end up decreasing the density of compute cells which in turn results in longer routes between them. This suggests that an overlay instance can be optimized by tuning its size and the amount of routing-only cells to a target set of DFGs.

7.8.5 Summary

We evaluate the overhead of DFG implementation on the two prototype overlays against a competing approach that implements DFGs directly in the FPGA fabric. Both approaches have equal DFG instance throughput, and the DFG-to-overlay approach has 18% lower $f_{\text{MAX}}$ on average than the DFG-to-FPGA approach. Thus, the overall throughput of both approaches is comparable. The significant differences between the two approaches lie in resource usage, compile time, and DFG latency.

The results show that the overlay-based implementation for the floating-point DFGs incurs resource overhead between 3.8x and 4.6x with respect to the direct implementation on the FPGA, while for the integer DFGs incurred overhead is between 7.5x and 10.8x.
Table 7.14: DFGs mapped to the entire base overlay and to the smaller overlays (i.e., rectangular regions of base overlays). The resource usage in ALMs, the usage of compute-routing mode cells, routing-only mode cells, and the total usage of cells are compared.

<table>
<thead>
<tr>
<th>DFG</th>
<th>ALMs CR cells cells cells CR RO total mode</th>
<th>ALMs CR cells cells cells CR RO total mode</th>
<th>ALMs CR RO cells entire/ entire/ entire/ entire/ region region region region</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-Body</td>
<td>51381 25 69 94</td>
<td>37357 25 34 59</td>
<td>1.4 1 2 1.6</td>
</tr>
<tr>
<td>N-Body-2x</td>
<td>95065 50 118 168</td>
<td>892921 50 88 138</td>
<td>1.1 1 1.3 1.2</td>
</tr>
<tr>
<td>N-Body-3x</td>
<td>134046 75 156 231</td>
<td>134046 75 156 231</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>135588 80 155 235</td>
<td>133588 80 155 235</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>MatMul</td>
<td>133498 63 189 252</td>
<td>133498 63 189 252</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>MatMulandAdd</td>
<td>143404 72 191 263</td>
<td>143404 72 191 263</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>RGB2YIQ</td>
<td>30111 21 54 75</td>
<td>19215 21 22 43</td>
<td>1.6 1 2.5 1.7</td>
</tr>
<tr>
<td>RGB2YIQ-2x</td>
<td>67254 42 128 170</td>
<td>53316 42 88 130</td>
<td>1.3 1 1.5 1.3</td>
</tr>
<tr>
<td>RGB2YIQ-4x</td>
<td>121073 84 218 302</td>
<td>121073 84 218 302</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>RGB2YIQ-5x</td>
<td>144552 105 252 358</td>
<td>144552 105 252 358</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>SAD</td>
<td>71203 47 128 175</td>
<td>43718 47 47 94</td>
<td>1.6 1 2.7 1.9</td>
</tr>
<tr>
<td>SAD-2x</td>
<td>121472 94 194 288</td>
<td>99658 94 130 224</td>
<td>1.2 1 1.5 1.3</td>
</tr>
<tr>
<td>GaussianBlur</td>
<td>82012 50 159 209</td>
<td>56470 50 85 135</td>
<td>1.5 1 1.9 1.5</td>
</tr>
<tr>
<td>GaussianBlur2x</td>
<td>134446 100 232 332</td>
<td>130959 100 222 322</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>MatMul</td>
<td>111140 63 224 287</td>
<td>102257 63 198 261</td>
<td>1.1 1 1.1 1.1</td>
</tr>
<tr>
<td>MatMulandAdd</td>
<td>119731 72 234 306</td>
<td>119731 72 234 306</td>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

This resource overhead is invested in order to achieve a significant reduction in compile time. For the floating-point DFGs, the DFG-to-overlay tool is three orders of magnitude faster (900x to 2800x) than the DFG-to-FPGA tool, and it is two to three orders of magnitude faster (200x to 2000x) for the integer DFGs. Compared to the DFG-to-FPGA approach, the latency of a single DFG instance execution on the overlay is up to 4x and 9x longer for the floating-point and integer DFGs, respectively. The latency increase is expected since the DFG-to-overlay tool optimizes primarily for DFG throughput.

Our resource overhead analysis shows that a lower overhead for the overlay-based approach can be achieved if the DFGs are mapped to smaller overlays that better match the DFGs in size (the smaller overlays are rectangular regions of OV-f and OV-i). The experiments with smaller overlays show that the overhead for the floating-point DFGs is reduced to a range between 3.3x and 4.3x, while for the integer DFGs to a range between 5.1x and 10x.
7.9 Evaluation of Overlay Physical Design Flows

We evaluate $f_{\text{MAX}}$ for the flat, top-down, and bottom-up flows using five custom 12x15 overlays. In this section, we first describe the benchmark overlays and the experiments that we conducted. We then present results for the three flows. For the bottom-up flow, we measure the cost and benefit of inter-tile EBs and the impact of tile stitching on clock skew change and routing conflicts. We then assess the benefits of the tile-based bottom-up flow. We conclude by investigating the effect of overlay-to-FPGA functional layout matching on the flat flow.

7.9.1 Benchmark Overlays

<table>
<thead>
<tr>
<th>overlay name</th>
<th>abs/ cmp</th>
<th>add/ sub</th>
<th>div</th>
<th>exp</th>
<th>log</th>
<th>mult (DSP)</th>
<th>mult (LUT)</th>
<th>sqrt</th>
<th>routing only</th>
</tr>
</thead>
<tbody>
<tr>
<td>OV$_1$</td>
<td>18</td>
<td>45</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>18</td>
<td>54</td>
</tr>
<tr>
<td>OV$_2$</td>
<td>18</td>
<td>45</td>
<td>18</td>
<td>–</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>18</td>
<td>54</td>
</tr>
<tr>
<td>OV$_3$</td>
<td>18</td>
<td>45</td>
<td>–</td>
<td>–</td>
<td>9</td>
<td>27</td>
<td>–</td>
<td>27</td>
<td>54</td>
</tr>
<tr>
<td>OV$_4$</td>
<td>18</td>
<td>90</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>36</td>
<td>–</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>OV$_5$</td>
<td>–</td>
<td>117</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>63</td>
</tr>
</tbody>
</table>

Table 7.15: Breakdown of five custom overlays (12×15 mesh) across FU type.

<table>
<thead>
<tr>
<th>overlay name</th>
<th>ALUTs</th>
<th>FFs</th>
<th>ALMs</th>
<th>DSPs</th>
<th>M9Ks</th>
<th>$f_{\text{MAX}}$ flat</th>
<th>$f_{\text{MAX}}$ top-down</th>
<th>$f_{\text{MAX}}$ bottom-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>OV$_1$</td>
<td>159503</td>
<td>220343</td>
<td>148259</td>
<td>444</td>
<td>14</td>
<td>210</td>
<td>283</td>
<td>287</td>
</tr>
<tr>
<td>OV$_2$</td>
<td>157184</td>
<td>219595</td>
<td>147300</td>
<td>396</td>
<td>18</td>
<td>241</td>
<td>314</td>
<td>322</td>
</tr>
<tr>
<td>OV$_3$</td>
<td>153512</td>
<td>213737</td>
<td>144546</td>
<td>180</td>
<td>0</td>
<td>236</td>
<td>338</td>
<td>339</td>
</tr>
<tr>
<td>OV$_4$</td>
<td>163458</td>
<td>236792</td>
<td>154919</td>
<td>144</td>
<td>18</td>
<td>244</td>
<td>329</td>
<td>345</td>
</tr>
<tr>
<td>OV$_5$</td>
<td>154216</td>
<td>220908</td>
<td>147050</td>
<td>0</td>
<td>0</td>
<td>261</td>
<td>333</td>
<td>344</td>
</tr>
</tbody>
</table>

Table 7.16: Resource use of five overlays (12×15 mesh) and $f_{\text{MAX}}$ achieved with flat, top-down and bottom-up flows. The bottom-up flow adds resource cost (see section Section 7.9.5).

All 5 of the custom overlays (OV$_1$-OV$_5$) that we designed have a 12×15 mesh and use around 150K ALMs, which is 70% of the FPGA’s ALMs. This is similar to OV-f
and OV-i, which use 75% of the FPGA’s ALMs. Furthermore, the 5 custom overlays are compute-routing cell dominated, meaning that the percentage of compute-routing cells ranges from 65% to 90%. In contrast, the percentage of compute-routing cells in OV-f and OV-i is lower—36% and 44%, respectively. Thus, even though all of the overlays use a similar amount of ALMs, OV1-OV5 have smaller meshes than OV-f/OV-i due to having a higher percentage of compute-routing cells (compute-routing cells consume more resources than routing-only cells).

The breakdown of the functional layouts of OV1-OV5 across FU/cell types is shown in Table 7.15, and the resource usage of the 5 overlays is shown in Table 7.16. OV5 consists of only logic-based FUs (uses only ALMs), while other overlays also use DSPs and M9Ks.

The \( f_{\text{MAX}} \) of an overlay (regardless of its mesh size) is limited by its slowest cell type (i.e., the slowest FU type). For example, OV1 contains a mix of all nine cell types, but its \( f_{\text{MAX}} \) is limited to 325 MHz because of its \( \text{exp} \) cells. In contrast, OV5 consists of only two cell types of which both are fast (add/sub and routing-only) and they pose a higher limit (440 MHz) to the overlay’s \( f_{\text{MAX}} \).

For each 12×15 overlay, we also construct overlay variants with smaller mesh sizes (1×1, 2×2, 4×5, and 8×10). We use the same cell type mix in the 4×5 and 8×10 overlays as the one used in the largest 12×15 overlay. For the variants with 1×1 and 2×2 mesh sizes, we pick the slowest cells type(s). Hence, all overlay variants (across all mesh sizes) have the same upper bound on \( f_{\text{MAX}} \).

### 7.9.2 Experiments

We compile all overlay variants with each flow using 5 different seeds, and we report the highest \( f_{\text{MAX}} \) achieved (i.e., best-of-5). We also measure absolute and relative \( f_{\text{MAX}} \) seed spread for all 12x15 overlays compiled with each flow. The absolute seed spread is the difference between the \( f_{\text{MAX}} \) attained by the best and worst results of the 5 seeds. The
relative seed spread is the ratio of the absolute seed spread to the average $f_{\text{MAX}}$ across 5 seeds. We report the relative seed spread (or simply the seed spread) as a metric of seed variability and as an indicator of whether a seed set contains a large outlier. Across all of the combinations of flow-overlay experiments, the seed spread is no more than 12% and is 6% on average—i.e., there are no large outliers and the overall seed variability is low.

In the flat and top-down flows, an entire overlay circuit is compiled at once for each given seed. In the bottom-up flow the tiles are compiled separately. This allows the selection of the best $f_{\text{MAX}}$ seed for each tile. As a result, the overlay can be stitched together using the tiles with their best seeds (i.e., best-of-5 seeds on a per-tile basis). For the bottom-up flow, we report the resulting overlay’s $f_{\text{MAX}}$ that is obtained after stitching of per-tile best seeds.

7.9.3 Flat Flow

Figure 7.8 shows that the $f_{\text{MAX}}$ degrades with increasing overlay size for all 5 overlays. In comparison to the $1 \times 1$ mesh, the $f_{\text{MAX}}$ for $12 \times 15$ mesh degrades by 33% for OV$_1$, and up to 45% for OV$_4$. The seed spread for the $12 \times 15$ mesh averages out to 8% for all overlays. The 33-45% $f_{\text{MAX}}$ degradation is a detriment to performance, and it also prevents the scaling of mesh-of-FUs overlays. The projected growth of FPGA size will allow for even larger meshes in the future. Hence, a flow that enables the scaling of mesh-of-FUs is critical for the success of mesh-of-FUs overlays.

7.9.4 Top-Down Flow

We use a $4 \times 5$ tile size to evaluate the top-down flow. Thus, the overlays with a $12 \times 15$ mesh size each consist of 9 tiles and are compiled to a floorplan with 9 physical regions. Similarly, the overlay variants with an $8 \times 10$ mesh size each have 4 tiles and are compiled to a 4-region floorplan. The smaller overlay variants with $1 \times 1$, $2 \times 2$, and $4 \times 5$ mesh sizes
are not floorplanned. Figure 7.9 shows the resulting $f_{\text{MAX}}$.

The $f_{\text{MAX}}$ of 1×1 overlay instances (single-cell overlays) compiled with the top-down and flat flows is different because of the different I/O constraints (fine-grained vs. single region). Across all three flows, we use the $f_{\text{MAX}}$ of 1×1 overlay variants compiled with the flat flow as a reference for measuring $f_{\text{MAX}}$ degradation that occurs when overlay mesh size is increased.

For a 12×15 mesh size, the top-down flow improves the $f_{\text{MAX}}$ over the flat flow by 28\% for OV₅ and up to 43\% for OV₃. The $f_{\text{MAX}}$ degradation of 12×15 overlay variants
compared to 1×1 variants is in the 10-26% range. This is less than the degradation that occurs with the flat flow (33-45% range). The seed spread for the 12×15 mesh size averages to 6%.

Furthermore, as the mesh size is increased to 12×15, the top-down flow maintains the $f_{\text{MAX}}$ within 6-9% of the $f_{\text{MAX}}$ achieved for 4×5 mesh size. Thus, the $f_{\text{MAX}}$ of a large overlay is close to the $f_{\text{MAX}}$ of a smaller overlay whose mesh size is equal to the tile size of the large overlay. These results validate our tile-based floorplanning approach.

### 7.9.5 Bottom-Up Flow

The bottom-up flow also uses 4×5 tile size and floorplans with 4 and 9 regions for 8×10 and 12×15 meshes, respectively. Figure 7.10 shows the resulting $f_{\text{MAX}}$. The $f_{\text{MAX}}$ degradation of the 12×15 variants compared to the 1×1 variants is in the 8-22% range, and the seed spread for the 12×15 variants averages to 7%.

Similar to the top-down flow, the bottom-up flow also maintains the $f_{\text{MAX}}$ of the 12×15 variants at the level achieved by the 4×5 variants. Additionally, for the 12×15 variants, it improves upon the top-down flow by 5% for OV$_4$, by 3% for OV$_2$ and OV$_5$, and by 1% for OV$_1$.

![Figure 7.10: Bottom-up flow $f_{\text{MAX}}$ (data points per-tile best-of-5 seeds).](image)
Figure 7.11: Benefit of inter-tile EBs: bottom-up compile of OV\textsubscript{2} with and without inter-tile EBs (data points per-tile best-of-5 seeds).

In all bottom-up experiments, the overlay’s \( f_{\text{MAX}} \) is limited by its slowest tile; that is, the inter-tile paths are not the bottleneck. For example, the slowest tile of OV\textsubscript{3} is \( T_{3,2} \) (339 MHz), which limits the overlay’s \( f_{\text{MAX}} \) even though the other tiles are faster (e.g., \( T_{3,3} \) achieves 359 MHz).

Cost and Benefit of Inter-tile EBs

We measure the resource usage overhead of inter-tile EBs that are inserted on the boundaries of \( 4 \times 5 \) tiles. The increase in the overlay’s logic resource usage (ALMs) due to these EBs is 8%. The floorplan regions and the overlay’s physical region are also increased by the same amount to achieve the same level of logic utilization. We believe this is an acceptable cost as it enables independent tile compilation and tile reuse. Furthermore, we experimentally verify that when inter-tile EBs are absent, the \( f_{\text{MAX}} \) attained for OV\textsubscript{2} with the bottom-up flow drops by 18%, as shown in Figure 7.11.
Impact of Tile Stitching on Clock Skew and Routing Conflicts

As described in Section 6.6.2, tile stitching can change the clock skew within the tiles which may result in $f_{\text{MAX}}$ drop. We compare the $f_{\text{MAX}}$ of the slowest tile for each overlay before and after tile stitching, and we observed that tile stitching caused an average drop of 3% in the $f_{\text{MAX}}$ of the slowest tile. Since the slowest tile determines the overlay’s $f_{\text{MAX}}$, tile stitching carries a 3% $f_{\text{MAX}}$ penalty. For example, the $f_{\text{MAX}}$ of OV$_3$’s tiles before stitching is within the 351 to 362 MHz range, but it drops to the 339 to 359 MHz range after stitching.

We found routing conflicts due to tile stitching to be minimal. On average, only 0.03% of a tile’s internal routing is not preserved and there is no impact on the tile’s $f_{\text{MAX}}$ or the overlay’s $f_{\text{MAX}}$.

7.9.6 Benefits of the Tile-Based Bottom-up Flow

Tile Library for Tile Reuse

The 5 benchmark overlays (OV$_1$ to OV$_5$) were compiled to a common template floorplan using the bottom-up flow. Each overlay has 9 tiles (T$_{1,1}$ to T$_{3,3}$) that are compiled to 9 physical regions (R$_{1,1}$ to R$_{3,3}$). In total, we have 45 pre-compiled tiles that form a small tile library. By reusing the tiles from the library it is possible to use tile-stitching to build around 2 million ($5^9$) different 12×15 overlays. This is because for each of the 9 physical regions (R$_{1,1}$ to R$_{3,3}$), we have a choice of 5 tiles (each originally compiled for one of the 5 benchmark overlays).

We build three tile-reuse overlays by reusing the tiles from the library. The first one is built by choosing the fastest tiles available from OV$_1$-OV$_5$ for each region, while the second one is built by choosing rows of tiles from OV$_3$, OV$_4$, and OV$_5$. Finally, the third one is built by choosing columns of tiles from OV$_2$, OV$_3$, and OV$_4$. These overlays achieve an $f_{\text{MAX}}$ of 347 MHz, 345 MHz, and 339 MHz, respectively. This demonstrates
the utility and effectiveness of our template-based approach for reusing tiles.

**Overlay Compile Time Reduction**

We compare the full compilation time (HDL to bitstream) for the flat, top-down, bottom-up from-scratch and bottom-up tile-reuse flows. Here we report the compile times for the 5-seed experiments for which we previously reported best-of-5 seeds $f_{\text{MAX}}$. Table 7.17 shows the obtained times in hours.

<table>
<thead>
<tr>
<th>Flow</th>
<th>Compile time (5 seeds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flat</td>
<td>18:59</td>
</tr>
<tr>
<td>Top-down</td>
<td>17:07</td>
</tr>
<tr>
<td>Bottom-up (from-scratch)</td>
<td>15:17</td>
</tr>
<tr>
<td>Bottom-up (tile-reuse)</td>
<td>00:35</td>
</tr>
</tbody>
</table>

Table 7.17: Average compile time (hours:minutes) of the benchmark overlays ($12 \times 15$ mesh) using best-of-5 seeds method across different flows.

The top-down compile time is shorter than the flat flow compile time due to a shorter place-and-route time, which is repeated for every seed (synthesis is done only once). In the bottom-up flow, the total place-and-route time is even shorter. This is because the overlay’s full place-and-route netlist is created only once (during the stitching of tiles with the best $f_{\text{MAX}}$), whereas in the flat and top-down flows it is created for every seed. The bottom-up tile-reuse flow takes only 35 minutes as it only needs to import and stitch together 9 tiles from the library.

**Potential for Compile Time Reduction on a Compute Cluster**

Multi-seed compiles are commonly run on multiple machines to reduce the overall compile time. For example, the 5-seed compile for the flat and top-down flows could be run in parallel on a 5-node cluster and achieve a compile time around 4 hours (one seed per node).
In the bottom-up flow all tiles are compiled independently. Thus, our bottom-up flow exposes parallelism at the tile level which can lead to further compile time reductions if it is exploited. For example, if a 45-node cluster is used, the 5-seed bottom-up compile time can potentially be reduced to one hour by compiling all tiles in parallel (9 tiles 5 seeds each). The 1-seed compile of each tile takes 23 min, and importing and stitching together the best seed tiles on a single machine takes 35 min.

**Resource Efficiency of Tile-Based Floorplanning**

We quantify the benefit of using tile-based floorplanning over cell-based floorplanning. Figure 7.12 compares the overlay’s total floorplan size (i.e., the overlay’s physical region) and $f_{\text{MAX}}$ for the two floorplanning approaches. The cell-based approach achieves 8% higher $f_{\text{MAX}}$ than the tile-based approach, but this comes at the expense of a 47% larger overlay floorplan size. Indeed, we are unable to fit the $12 \times 15$ cell-based floorplan on the Stratix IV device. For this experiment, we used the top-down flow on the OV$_2$ overlay.

**Per-Tile Seeds**

We quantify the benefit of an approach that uses the best-of-5 seed on a per-tile basis. For comparison, we compile the overlays with the bottom-up flow such that all tiles are
compiled with the same seed. The per-tile best-of-5 seed approach minimally improves $f_{\text{MAX}}$—by 2%—over the per-overlay best-of-5 seed approach. Thus, even though the per-tile best-of-5 seed approach improves $f_{\text{MAX}}$ slightly, its impact is not significant.

### 7.9.7 Impact of Overlay-to-FPGA Functional Layout Matching

Finally, we investigate the impact of overlay-to-FPGA layout matching on the $f_{\text{MAX}}$ of the flat flow. In other words, we measure whether layout matching alone (without the use of floorplanning) can contribute to higher $f_{\text{MAX}}$. We use two overlays with the same set of cells, but the cells are arranged into two different functional layouts. The layout of the first overlay was designed such that its DSP overlay columns are matched with the FPGA’s DSP columns. The functional layout of the second overlay is designed without matching.

We also construct overlay variants with smaller mesh sizes using the method described in Section 7.9.1, and we compile the overlays using the flat flow. The obtained $f_{\text{MAX}}$ for mesh sizes ranging from $1\times1$ to $12\times15$ is shown in Figure 7.13. We observe that $f_{\text{MAX}}$ is similar for both overlays, and it exhibits significant degradation as the mesh size is increased.
We analyzed the results for the matched overlay and found that this degradation occurs for all seeds. In each compile, some of the overlay cells had their DSP blocks placed either to a non-local (i.e., non-matched) column or they were placed to the matched DSP column but with a large offset within the column. In either case, the long routing delay limits the $f_{\text{MAX}}$ of the overlay. Moreover, as the overlay increases in size, it covers more of the FPGA’s DSP columns, which also become taller. This increases the likelihood of DSP misplacement and in turn results in more $f_{\text{MAX}}$ degradation.

These results show that layout matching alone—without the use of floorplanning—is not sufficient to maintain high $f_{\text{MAX}}$ for large overlays. Our results on floorplan-based flows (top-down and bottom-up) show that layout matching combined with floorplanning is effective in maintaining high $f_{\text{MAX}}$. Floorplan-based flows inherently enforce layout matching through physical constraints, and the resources available in each physical region dictate the possible functional layouts that can be mapped to that region.
Chapter 8

Related Work

The work in this thesis is related to research on mesh-of-FUs architectures on both FPGAs and ASICs. Furthermore, the DFG-to-overlay mapping algorithm is related to research on placement, routing, and latency balancing algorithms. Finally, the tile-based bottom-up flow is related to CAD techniques and flows that utilize partitioning and floorplanning. In this chapter, we review the existing work in these areas.

8.1 Mesh-of-FUs Architectures

There have only been a few mesh-of-FUs FPGA overlay architectures proposed in the literature.

Shukla et al. [96, 143] propose QUKU, which uses a form of data-driven execution and integer FUs, but this architecture was only investigated for small mesh sizes (up to 4x4). Their use of FIFOs is for the implementation of a process-networks programming model. In contrast, we use FIFOs to implement a latency balancing mechanism by embedding them into every routing hop.

McGettrick et al. [95] present a mesh with DSP-only integer FUs, but they use a centralized pipeline control, which limits $f_{\text{MAX}}$ to 196 MHz despite the architecture containing fast DSPs. However, as with QUKU, there is no latency balancing mechanism
present in the architecture. Neither QUKU nor the work by McGettrick et al. offer an application-to-overlay mapping algorithm.

Coole and Stitt [39, 144] present Intermediate Fabrics (IF), which is a centrally controlled mesh-of-FUs with inelastic pipelines. There are a number of major differences between IF and our approach in terms of both overlay architecture and mapping algorithm.

First, IF uses a central controller and a global stall signal for the entire IF overlay. In contrast, we implement elastic pipelines to achieve distributed pipeline control and localized synchronization signals. Furthermore, the self-contained nature of our overlay cells facilitates the compilation of the overlay with a tiled-based bottom-up flow.

We compare the $f_{\text{MAX}}$ and the overall compute capability of our approach to that of IF across several overlay instances. The IF architecture on a Virtex 4 (90nm technology) achieves on average $f_{\text{MAX}}$ of 175 MHz across several application-customized overlay instances with up to 50 FUs (16-bit integer). Also, the IF architecture on a Stratix III (65 nm technology) for a 16-bit integer overlay with 192 FUs achieves an $f_{\text{MAX}}$ of 124 MHz. Conversely, the architecture designed in this work achieves an $f_{\text{MAX}}$ of 355 MHz for a 32-bit integer overlay with 170 FUs (on Stratix IV, 40 nm).

Compared to the Virtex 4 instance of IF, our overlay has an advantage of two FPGA generations; nonetheless, our overlay is 12X better in terms of overall compute capability. Specifically, our overlay has 2X the data-width, 3X more FUs, and 2X higher $f_{\text{MAX}}$ than IF. This combines for a total 12X advantage in terms of overall compute capability.

A similar comparison holds for the Stratix III instance of IF. In this case our overlay has a one FPGA generation advantage, but is 5.8X better in terms of overall compute capability. That is, it has 2X the data-width, a roughly equal number of FUs, and achieves 2.9X higher $f_{\text{MAX}}$. This combines for a 5.8X advantage in overall compute capability.

The overall compute capability as well as the $f_{\text{MAX}}$ advantage of our overlay exceeds the advantage of generational FPGA scaling. Thus, the performance advantage we get
is not only due to the generational improvement of the device, but rather due to the scalability of our approach.

The second difference between IF and our architecture is that the IF architecture requires latency balance for correct execution due to its inelastic and globally stalled pipelines. In contrast, our elastic pipelines are locally stallable at each individual FU. As a result, our architecture can operate correctly even if the DFG mapping is not latency balanced (i.e., the architecture is latency insensitive). Thus, any placement of a DFG that is routable can execute correctly, and latency balancing takes on a performance optimization role. We see this as a key usability advantage.

The third difference between IF and our architecture is that IF uses 8-slot configurable shift-registers at the inputs of the FUs for latency balancing. Thus, two routed paths reconverging at an FU can differ only up to 8 cycles. The IFs also use deeper shift-registers as one of the types of FUs, but they are present in small numbers (up to 5) and only at the boundary of the overlay, which may limit their applicability to balance reconvergent DFG paths. In our work, we embed a deep FIFO (e.g., capacity of 32) in every overlay’s routing hop. As a result, a latency-balancing margin accumulates along every routing path. This allows for the latency balancing of two paths with an arbitrary latency difference.

The fourth difference between IF and our architecture is that our mapping algorithm uses internal seed-sweeping to find placements that are both routable and balanced. Thus, our algorithm can hide from the programmer (to a certain degree) the fact that some DFGs are harder to route and/or balance and require multiple seeds. The IF mapping algorithm also uses a simulated-annealing-based placer, however, it runs only one seed; thus, unroutable or imbalanced mappings are made visible to the user. Furthermore, the IF mapping algorithm can latency balance paths with an imbalance of only up to 8 cycles, otherwise the netlist must be altered manually to use the deeper shift registers. In contrast, our mapping algorithm uses a general ILP-based latency balance
checking together with balancing node insertion and does not pose such restrictions and can support an arbitrary degree of latency imbalance.

Finally, the IF’s mapping algorithm performs timing analysis on routing delays along the IF’s routing tracks in order to improve $f_{\text{MAX}}$. As a result, the $f_{\text{MAX}}$ achieved on an IF instance is seed-dependent and thus susceptible to seed noise, exposing the intricacies of hardware design to the programmer. In contrast, our overlays have a fixed $f_{\text{MAX}}$—the high-performance design of the architecture and the tile-based flow aim to ensure that the worst case $f_{\text{MAX}}$ of the overlay is high. Nonetheless, it is possible that an instance of our overlay would contain a slow FU, which would limit the overlay’s $f_{\text{MAX}}$. For the DFGs that do not utilize these FUs it may be beneficial to use a form of timing analysis that excludes the unused slow FUs, thus allowing the overlay to run at a higher $f_{\text{MAX}}$. We conclude that overall our approach has an advantage in the areas of performance and usability.

Lin et al. [94] explore folding (i.e., time-multiplexing) large 1,000-node DFGs onto a small mesh-of-FUs overlay architecture (4x5 with 20 FUs). These overlays achieve up to 100 MHz. There is no support for latency balancing within the architecture itself, and the burden of finding a latency-balanced DFG mapping falls entirely on the software, which generates a static schedule of operations much like a VLIW processor. The authors also explore various overlay interconnects such as ring, torus, and DFG-specific. In contrast, we design a high-throughput and a high-$f_{\text{MAX}}$ mesh-of-FUs architecture, and we demonstrate scalability to large mesh sizes in terms of $f_{\text{MAX}}$. We also design a latency balancing mechanism that divides the burden of balancing between the hardware and software. The work by Lin et al. [94] on DFG-folding and interconnect customization can potentially be integrated with our work. We leave this direction for future research.

Our work also relates to mesh-of-FUs-like architectures implemented in ASIC, namely, coarse-grain reconfigurable architectures (CGRAs) and massively parallel processing arrays (MPPAs). Common across these architectures is their homogeneous functional lay-
outs with integer-only ALUs [107, 145–149]. In contrast, we design an overlay architecture with heterogeneous functional layouts (with both integer and floating-point FUs) to leverage the FPGA’s ability for application-driven customization.

A majority of ASIC CGRAs are statically scheduled in a fashion similar to a VLIW processor in that the burden of scheduling for high-throughput is entirely on the compiler [145, 150]. Furthermore, a single I/O stall (e.g., a memory access contention or a cache miss) can cause the entire array to be stalled because of a static lock-step schedule across all FUs [145]. In contrast, our architecture is dynamically scheduled (via data-driven execution) and individually stallable at each I/O port and each FU, while the FIFOs and EBs embedded in the architecture’s pipelines provide elasticity to absorb intermittent local stalls.

A form of dynamic data-driven execution is used in two early CGRAs, KressArray [146] and PACT XPP [147]. Although both have pipelined interconnect, their architectures do not have a mechanism for latency balancing. Moreover, KressArray’s architecture assumes that there is a very low bandwidth restriction on the mesh-of-FUs I/O boundary (only a single 32-bit word per cycle), which severely limits the possible throughput of the architecture. Thus, latency balancing for the high-throughput execution of the FUs would be of little value.

Two recent CGRA architectures employ elastic pipelines: Huang et al. [149] focus on the energy-efficiency evaluation of the architecture, while Panda et al. [148] explore the design space of interconnect and stall network implementations. These two works do not explore latency balancing support in hardware or compiler support for latency-balanced mapping. In contrast, we tailor the implementation of elastic pipelines for the FPGA fabric, and we also maximize DFG throughput with a combination of deep elastic FIFOs embedded in the routing and a latency balancing mapping algorithm.

Ambric [105, 106] is a massively parallel processor array (MPPA) in which CPUs are individually programmed with each containing local instruction and data memory.
This programming model is similar to the Kahn Process Network [151]. The CPUs communicate with their neighbours via messages, which are carried via channels based on 2-slot EBs. There are several key differences between Ambric and our mesh-of-FUs overlay architecture.

First, Ambric channels are based on 2-slot EBs, which cannot be used for latency balancing. The buffering capacity between two CPUs can only be increased by manually instantiating a FIFO object in the code which is mapped to a physical RAM block adjacent to the CPU [105]. Thus, latency balancing is limited and is done manually. In contrast, we design a built-in support for latency balancing by embedding deep elastic FIFOs into each overlay’s routing hop. Furthermore, our DFG-to-overlay mapping algorithm automatically finds a balanced mapping through the use of seed-sweeping and an ILP-based latency balance checker.

Second, Ambric architecture is designed for coarse-grain computation: its compute building blocks are CPUs with instruction and data memories. Studies show that Ambric does not perform well for fine-grained computations, such as those similar to our DFGs, and can result in low CPU utilization and low performance [152]. In contrast, the mesh-of-FUs is a “leaner” architecture: its compute building blocks are FUs, which are tuned for the fine-grained computation of the DFGs.

Third, our architectural decisions were driven by the FPGA fabric. For example, we mix EBs with inelastic registers and inelastic FUs to balance FPGA resource use and \( f_{\text{MAX}} \). In contrast, Ambric is tuned for the ASIC fabric and contains no such optimizations. Thus, the direct implementation of the CPU-based Ambric architecture as an FPGA overlay is likely to result in much higher resource cost and lower \( f_{\text{MAX}} \) in comparison to our overlay architecture.

Finally, Ambric is a homogeneous integer-only architecture in that all CPUs are the same and perform only integer operations. In contrast, the mesh-of-FUs overlays in this thesis are heterogeneous with both integer and floating-point operations. Additionally,
it is common for the layout of an FPGA to be non-uniform (i.e., DSP columns are unevenly spaced across the FPGA). These aspects give rise to a number of physical design challenges, which we address in this work. Such challenges do not arise when an Ambric-like architecture is implemented in ASIC.

Since the initial description of our work [40, 128], a number of researchers have adopted mesh-of-FUs overlay architecture and proposed improvements and extensions. Aklah et al. [153, 154] propose FUs that are implemented as PR regions and that can be dynamically loaded. Jain et al. [155–158] explore FUs that are based entirely on DSP block functions to improve resource efficiency. Psarakis [159] introduces reliability-aware mesh-of-FUs where redundancy is achieved by triplicating each FU. Liu et al. [160, 161] develop tools for mapping nested loops to mesh-of-FUs overlay architecture. Cooke et al. [162] propose coupling mesh-of-FUs with an FSM overlay architecture to enable more flexible control flow. Similarly, Ng et al. [163] couple mesh-of-FUs with a soft processor for executing complex control flow. Eslami and Wilton [164] use a mesh-of-FUs overlay to augment a user’s hard-wired circuit, thus enabling debug and fast trigger insertion without circuit re-compile. Belwal et al. [165] explore the use of mesh-of-FUs overlays for seamless application execution on hybrid CPU/FPGA systems.

8.2 Place and Route Algorithms for FPGAs

Our DFG-to-overlay mapping algorithm adapts and extends well-established techniques for simulated-annealing-based placement and PathFinder-based routing from the context of FPGAs [49, 69] to the context of mesh-of-FUs architecture.

A key difference between our techniques and previous work is the problem size. Whereas circuits with one million nodes are mapped to a large FPGA, DFGs with 100’s of nodes are mapped to mesh-of-FUs. This enables several orders of magnitude reduction in application mapping time. We further leverage this to perform an internal seed sweep,
which is a technique that is prohibitively expensive for the compilation of circuits with one million nodes.

A second key difference is that the FPGA CAD employs timing analysis at all stages of the flow. In contrast, we optimize the overlay architecture for high $f_{\text{MAX}}$ and then use that as a fixed operating frequency for the overlay across all DFGs. This obviates the need for timing analysis, which simplifies the DFG-to-overlay mapping algorithm. Finally, in the context of the HDL-to-FPGA CAD flow, the design of a cycle-by-cycle schedule, throughput, and latency of operations is entirely in the hands of an HDL designer. The FPGA CAD tool primarily optimizes $f_{\text{MAX}}$, resource usage, and power. In contrast, the DFG-to-overlay mapping algorithm maps an abstract DFG representation to a data-driven architecture. Thus, any mapping (with an arbitrary throughput and latency of operations) is legal, so long as the data dependencies are satisfied.

In comparison to the state-of-the-art FPGA simulated-annealing placer used in VPR [58, 68], we use a novel $O(1)$ move-generation algorithm that is tuned for heterogeneous coarse-grain architectures. Whereas VPR performs an iterative random search for a legal move within a range-limited window, we pre-compute a Manhattan-distance-sorted legal move list (LML) and a distance index (DI) for each location in the mesh. A move is then generated by $O(1)$ look-ups in DI and in LML.

A recently proposed *smart search space* algorithm for heterogeneous non-column-based FPGA layouts [166] also provides $O(1)$ move generation. The physical distance range in this algorithm depends on the block type, allowing some block types to make larger moves than others. This possibly has a downside of creating displaced blocks because two connected blocks of different types are not allowed to make the same large move. One advantage of this approach over our approach is that it requires less memory because it uses a global table of legal moves.

As mentioned earlier, one of the key differences between our work and FPGA CAD is the use of an internal seed-sweep (e.g., 40 seeds) that is performed by our DFG-to-overlay
tool in a manner that is transparent to the user. This both improves the quality of the results and reduces the seed noise that is inherent to FPGA CAD tools that use only a single-seed. In FPGA CAD tools, a seed-sweep can be performed explicitly by the user via the design space exploration tools (DSE) [167].

The seed-sweeping also allows us to adapt the parameters of the place-and-route algorithm from one seed to the next. We adapt the PathFinder’s routing iteration limit such that, if a placement for a given seed is not routable, we then increase the routing iteration limit for the next seed.

Our work also relates to algorithms that map to a class of FPGA and CGRA architectures that employ pipelined routing [168–172]. The architectures and the circuits that these algorithms map to are inelastic and latency sensitive, which means that the latency of the paths in the circuit must be preserved and cannot be modified. As a result, these algorithms deal with the $N$-delay routing problem: the path between a source and sink must include exactly $N$ registers. In contrast, we map to a data-driven latency-insensitive architecture. The DFG mapping will be functionally correct even if the latency of a routing path is arbitrarily lengthened and even if joining paths are imbalanced. Furthermore, in addition to pipeline registers, we also embedded deep elastic FIFOs into the routing hops, which serve as hardware-support for latency balancing.

8.3 Partitioning, Floorplanning and Bottom-up Compilation

Hillenbrand et al. [173] compile a streaming architecture with 8 streaming engines to a regular 8-region floorplan. The streaming engines are not connected and the system achieves an $f_{\text{MAX}}$ of 144 MHz. Otero et al. [174] propose a CAD tool, called Dreams, that uses a regular floorplan to compile an integer 8-bit mesh-of-FUs in the context of dynamic reconfiguration. However, they do not report on $f_{\text{MAX}}$ nor do they report on
the effectiveness of their floorplanning. In contrast, we show the scalability of our flows and achieve an $f_{\text{MAX}}$ of up to 355 MHz.

LaForest and Steffan [86] use partitioning to improve $f_{\text{MAX}}$ for a homogeneous mesh of soft-processors and conclude that floorplanning is not beneficial. In contrast, we have a heterogeneous mesh-of-FUs where overlay cells differ by $5x$ in resource usage. While we also find partitioning useful, we show that floorplanning is critical for high $f_{\text{MAX}}$, tile reuse, and compile time reduction.

Several works [76, 175–177] automatically create a custom irregular floorplan for a given circuit and optimize for half-perimeter wirelength. Neely et al. [178] use a domain-specific automatic floorplanning algorithm to optimize modular networking pipelines. Majer et al. [179] and Koch et al. [180] use a template floorplan that is based on narrow vertical slots, but in the context of partial reconfiguration environment. In contrast, we have simpler and more regular floorplans, and we use tiles to achieve high $f_{\text{MAX}}$ while being resource efficient. We also allow for tile reuse across overlays.

Hard-macro-based flows [181–184] use a pre-computed library of hard-macros (pre-compiled modules) that are placed and stitched together to create a circuit. These flows mainly aim to reduce compile time, potentially at the expense of high $f_{\text{MAX}}$. In contrast, we seek to achieve high $f_{\text{MAX}}$.

The other mesh-of-FUs overlay architectures explored in the literature [39, 94–96] only use a flat compilation flow. Our tile-based bottom-up flow could be applied to these architectures by first partitioning and floorplanning the mesh into tiles. Additionally, any global control signals and the corresponding control logic that these architectures employ would have to be replicated and localized for each tile (in effect allowing for the independent operation of tiles). Finally, the tiles could also be isolated with the inter-tile EBs proposed in this work or a similar form of inter-tile pipelining.

Since the initial description of our work [40, 128], several researchers have further explored bottom-up techniques for overlay and user circuit compilation. Yue et al. [185]
propose a zero-logic-overhead routerless stitching of adjacent mesh-of-FUs overlay cells by aligning cell interconnect, but this technique was explored only for low $f_{\text{MAX}}$ overlays (120 MHz). Murray and Betz [186, 187] present an automated floorplanning tool. Their initial experiments with the tool do not evaluate $f_{\text{MAX}}$ of circuits with regular mesh architecture.

### 8.4 Alternative FPGA Overlay Architectures

Researchers have also proposed other types of overlay architectures, including: multi-processors and many-cores [83, 85, 188], many-cores with custom cores [84], multi-threaded processors [189, 190], vector processors [87, 88, 97], GPGPU-like soft processors [89, 90], VLIW processors [91, 191, 192], networks-on-chip [92, 93] and mesh-of-soft-processors [86].

Similar to the mesh-of-FUs overlay architecture, these overlay architectures must also expose the massive parallelism of FPGAs to be successful and also provide a useful parallel programming model that can utilize this parallelism. Furthermore, they must exploit the FPGA’s ability to customize an overlay instance to an application and be able to scale with FPGA size while maintaining high $f_{\text{MAX}}$.

Research on multi-processors has investigated systems containing up to 32 processors [83, 85, 188]. A many-core system with custom cores and a ring interconnect with 50 cores was demonstrated but only with $f_{\text{MAX}}$ of up to 160 MHz. Vector processors with up to 32 lanes and an $f_{\text{MAX}}$ of 200 MHz have also been demonstrated [87, 88, 193]. Finally, a soft GPGPU with up to 32 stream processors and an $f_{\text{MAX}}$ of 100 MHz has also been investigated [90].

This set of evaluated architectures demonstrates the challenge of scaling beyond 50 cores and an $f_{\text{MAX}}$ of 200 MHz. However, as FPGAs increase in size, overlay architectures with 100’s of cores (or equivalent FUs/lanes) and eventually 1000’s of cores will become
crucial for the adoption of overlays. The work in this thesis demonstrates a step in that direction, namely, the introduction of mesh-of-FUs overlays with up to 170 FUs and an operating frequency of 355 MHz. Furthermore, the pipelined execution of DFGs fits with the OpenCL and OpenACC programming models that are showing success in the heterogeneous computing space.

Concurrently with our work [40, 128], research on a homogeneous mesh of soft processors [86] confirms our results that scaling and high $f_{\text{MAX}}$ is achievable for mesh architectures. However, [86] lacks a software programming model that would utilize the parallelism of the mesh.

Recent work also extends a vector processor overlay with custom instructions in the form of hard-wired DFG pipelines [193], thus improving the vector processor’s ability to exploit pipeline parallelism exposed by DFGs. Finally, work on VLIW-based overlays presents a system with 8 identical VLIW-cores that execute in a SIMD fashion. This VLIW-based overlay executes OpenCL kernels [192].

This research suggests the growing importance of meshes, pipelined DFG execution, and the OpenCL programming model for overlay architectures. We see a potential for the synergy of the strengths of different overlay architectures. On one hand, mesh-of-FUs offer scalability, high $f_{\text{MAX}}$, and pipeline parallelism at a large scale. On the other hand, VLIWs and vector processors offer instruction-level parallelism and the ability to time-multiplex large DFGs. One possible example of a hybrid architecture would be to extend the mesh-of-FUs architecture by augmenting the FUs with the capability for VLIW, vector, and time-multiplexed style of execution.

Other researchers have further explored overlay architectures that are not based on mesh-of-FUs. Gray [194] proposes a massively parallel processor array (MPPA) overlay based on RISC soft processors that are interconnected with an NoC. Omidian and Lemieux [195] explore algorithms for mapping Khan Process Network (KPN) based programs to MPPA overlays. Cheng and Wawrzynek [196] propose use of data-flow based
architectural templates to improve the quality of circuits generated by HLS tools. Andryc et al. [197] conduct an architectural exploration of GPU overlay architectures on FPGAs. Polig et al. [198] present a mesh of soft-processors tailored for relational operations of text analytics queries.

8.5 Elastic Pipelines

The use of data-driven pipelined execution in this work builds upon the techniques of: (1) synchronous elastic pipelines with data-presence and back-pressure signals [47, 48]; (2) latency-insensitive pipeline design [103]; and (3) non-linear elastic pipelines with join/fork structures [47, 48, 122].

These techniques are used to varying extent in streaming architectures for FPGAs such as SCORE [104], ImpulseC [199] and Altera’s Avalon Streams and OpenCL compiler [200, 201], high-performance multi-FPGA applications [202], packet-switched NoCs on FPGAs [92, 93], Ambric MPPA [105], ASIC CGRAs [148, 149], and processors [47, 48, 122]. We also apply the same techniques to the design of a mesh-of-FUs overlay architecture in FPGA fabric. This context brings a unique set of trade-offs between $f_{\text{MAX}}$ and scalability on the one hand and resource usage on the other.

Cortadella et al. [101] propose a design space consisting of four types of EBs (implemented with flip-flops) based on the physical arrangement of the two data registers and the mux. In this thesis, we explore the design of two types of EBs but also propose three variants of each type. These variants have different control logic implementation which is either tuned for higher $f_{\text{MAX}}$ or better capacity behaviour during stalls. Our use of EB variants and their control logic tuning is novel in both the FPGA and ASIC contexts. Furthermore, our back-to-back use of mirrored stall buffers (the EB variants we term D-BUF1 and D-BUF2) on inter-tile paths for the purpose of isolating tiles is also novel. In the ASIC context, two EB variants were previously proposed [48, 101, 203]
that correspond to the EB variants we term D-EB1 and D-BUF1. In the FPGA context, an EB variant similar to D-EB1 was realized concurrently with our work [130].

In addition, the design of the run-time reconfigurable join and fork synchronization logic proposed in this work is novel. Previous work [47, 48, 102, 122] on synchronization logic has only explored its fixed (i.e., hard-wired) form; that is, the set of senders and receivers that are synchronized with join and fork structures is fixed. In contrast, in our join and forks structures, both the sender and receiver sets are run-time configurable via configuration masks. Furthermore, the reconfigurability of our structures requires that both join and fork structures are distributed (i.e., entirely separate synchronization logic for each receiver/sender). Finally, previous work in the context of ASICs considered only a distributed fork structure (LKFork in [122]), while all join structures are centralized (i.e., senders/receivers share parts of the synchronization logic).
Chapter 9

Conclusions

9.1 Research Summary

In this thesis we explored an approach to tackle the programmability wall of FPGAs that is based on overlays. We proposed, designed, and implemented a high-performance mesh-of-FUs overlay architecture for the pipelined execution of DFGs.

More specifically, we have adopted a DFG programming model for the overlay architecture. A DFG is mapped to an overlay by mapping the compute nodes to the overlay’s FUs and by configuring the overlay’s routing logic to create data flow among the FUs. Multiple instances of the DFG are then executed in pipelined fashion to achieve high-throughput. We selected DFGs as a programming model because they provide a balance between inherent parallelism (via pipelined execution) and their usability in parallel programming models. They are used as a parallel programming abstraction in DSP design and streaming programs and they can also be extracted from the bodies of OpenCL kernels and parallel loops such as OpenMP.

The overlay architecture uses data-driven execution that is designed using elastic pipelines that we tailor to the FPGA fabric. Furthermore, the overlay’s architecture is modular, which means that the overlay’s cells are self-contained and pipeline flow control
is distributed without the need for global control signals. This facilitates high \( f_{\text{MAX}} \) of the architecture and also allows for the use of the tile-based bottom-up flow that enables the overlays to maintain high \( f_{\text{MAX}} \) as their mesh size is scaled up.

We embed deep elastic FIFOs (e.g., capacity of 32) in every overlay’s routing hop in order to provide them with latency margins. This forms the basis for the latency balancing mechanism of the overlay: shorter (lower latency) paths can use their latency margins to “stretch” their latency in order to balance the latency of longer paths.

The DFG-to-overlay mapping tool leverages this mechanism to find the most latency balanced DFG mapping, thus maximizing the throughput of pipelined DFG execution. The tool first uses the simulated annealing-based placer and the PathFinder-based router to find a DFG mapping. It then uses the ILP-based latency balance checker to test whether the latency margins along the shorter routing paths are sufficient to balance the longer paths. Due to the overlay’s coarse-grain architecture and the absence of timing analysis (the operating frequency of each overlay is fixed), these algorithms have very fast run-times. This enables the use of internal seed-sweeping (transparently to the user) to generate multiple DFG mappings and then to pick the most balanced one.

We designed a tile-based bottom-up CAD flow in order to overcome the following limitations that arise when compiling the mesh-of-FUs with the default flat flow of the CAD tool: 1) degradation in \( f_{\text{MAX}} \) when the overlay’s mesh size is increased; and 2) the flat flow does not allow for the reuse of pre-compiled overlay components to build new overlays, which prevents the quick compilation of new overlays and the modification of existing ones. The tiled-based bottom-up flow utilizes partitioning and floorplanning to divide the overlay into tiles. It compiles the tiles onto a rectangular coarse-grain floorplan to achieve a resource-efficient placement of the overlay’s cells within the tiles.

The bottom-up flow also uses inter-tile elastic buffers to ensure that the inter-tile buffers are not a bottleneck for \( f_{\text{MAX}} \), thus enabling the independent compilation of tiles. As a result, an overlay can be compiled and achieve high \( f_{\text{MAX}} \) by only “stitching” together
a set of pre-compiled tiles that have high $f_{\text{MAX}}$. Furthermore, the use of rectangular coarse-grain template floorplans facilitates the design of a tile library that can be reused to stitch new overlays. The independent compilation of tiles enables parallel compilation, which can be used to compile tiles on a compute cluster to significantly reduce overlay compile time.

We implemented two large overlay prototypes on a Stratix IV FPGA: a $24 \times 16$ integer-FU overlay with 170 compute cells that operates at 355 MHz, and an $18 \times 16$ floating-point-FU overlay that has 104 compute cells and runs at 312 MHz. The high $f_{\text{MAX}}$ achieved by these overlays is a combined result of modular design, the matching of the overlay’s layout to the FPGA’s layout, and the tile-based bottom-up flow.

We evaluated the DFG-to-overlay mapping tool using 16 DFGs that have been extracted from 7 parallel code sections. The DFG-to-overlay mapping tool is able to find a latency balanced mapping on the two overlays for all DFGs, thus achieving maximum throughput. This high-performance is achieved due to both high $f_{\text{MAX}}$ and maximum DFG throughput: the integer overlay achieves up to 37 GOPS and the floating point overlay achieves up to 22 GFLOPS. Furthermore, the mapping of the DFGs to the overlays is highly efficient: a DFG can be mapped to an overlay in no more than a few seconds. These results validate the design of a high performance overlay architecture for the pipelined execution of DFGs.

We assess the resource overhead of the two overlays by comparing the resource usage of implementing the DFGs on them to that of DFGs that are compiled directly to the FPGA fabric (i.e., hard-wired DFG circuits). In case of the floating-point DFGs, the overlay-based implementation incurs resource overhead that is between 3.8x and 4.6x compared to the direct approach, while for the integer DFGs the overhead is between 7.5x and 10.8x. Our analysis shows that the resource overhead is reduced when the DFGs are mapped to smaller overlays that better match the DFGs in size—the overhead for the floating-point DFGs is reduced to a range between 3.3x and 4.3x, while for the integer
DFGs to a range between 5.1x and 10x.

The resource penalty of the two overlays is incurred in order to achieve both the high-performance of the overlays and a significantly faster compile time in comparison to that of the DFG-to-FPGA flow. For the floating-point DFGs, the DFG-to-overlay tool is three orders of magnitude faster (900x to 2800x) than the DFG-to-FPGA tool, whereas it is two to three orders of magnitude faster (200x to 2000x) for the integer DFGs.

We experimentally evaluate the tile-based bottom-up flow on 5 heterogeneous overlays and found that the overlay’s high \( f_{\text{MAX}} \) is maintained as the overlay’s mesh size is increased from 4x5 to 12x15 cells. The bottom-up flow achieves 37% higher \( f_{\text{MAX}} \) than the flat flow for large meshes. We also demonstrate that pre-compiled tiles can be reused to stitch together a new overlay in 35 minutes on a single machine while still achieving high \( f_{\text{MAX}} \) for the overlay. By comparison, the same overlay takes 4 hours to compile using the flat-flow.

The use of inter-tile elastic buffers in the bottom-up flow adds no more than 8% resource overhead. These results suggest that our tile-based bottom-up flow is a viable approach for compiling large mesh-of-FUs overlays to the FPGA fabric.

The two overlay prototypes demonstrate the capability of the mesh-of-FUs overlay architecture to be scaled to the size of a large FPGA and expose its massive parallelism. In addition, the integer and floating-point overlay prototypes show that it is possible to design functional layouts for the overlays that are customized for a different set of applications, thus leveraging the flexibility of the FPGA fabric. Furthermore, the tile-based bottom-up flow demonstrates that large custom mesh-of-FUs overlays can be built quickly while maintaining high \( f_{\text{MAX}} \). The DFG-to-overlay mapping tool achieves high-throughput mapping of the DFGs on the overlays and has a fast mapping time. Additionally, it insulates the programmer from the hardware intricacies of FPGA design such as timing analysis and seed noise. Nonetheless, the many benefits of our overlay architecture do come with a modest resource overhead over a hard-wired DFG implementation in FPGA fabric.
9.2 Concluding Remarks

Our results demonstrate the potential advantages of overlays that we alluded to in the introduction. In particular, we show that it is possible to design a high-performance overlay architecture that scales and enables quick stitching of a new overlay from reusable tiles while still achieving high $f_{\text{MAX}}$. Furthermore, we show that it is possible to design a mapping tool that is fast, efficient, insulates the programmer from the intricacies of hardware design and obviates the use of slow FPGA CAD tools. We believe that these results position our overlay architecture as a true contender for tackling the programmability wall of FPGAs. However, in order to realize the promise of this work, several major challenges still need to be addressed.

The first major challenge is preserving the energy efficiency advantage of using FPGA for acceleration. For certain application domains, an FPGA implementation may win on both the performance and energy-efficiency over a CPU or GPU implementation. The use of our overlay approach can bring the ease of programming and development cycles closer to that of a CPU. However, at this point, this comes with a resource overhead that decreases the overall energy-efficiency. In certain cases, the use of overlay may even negate the FPGA energy-efficiency advantage. Thus, overcoming the resource overhead of our overlay is essential in order to maintain the energy-efficiency of FPGAs.

One of the ways in which the resource overhead of our overlay could potentially be reduced is through the use of a large library of pre-compiled overlay tiles, each customized to a specific application-domain in several ways. These can include customizing the functional units, the size of the tiles, the amount of routing-only cells, the data-path widths, and the depth of D-FIFOs. An overlay built from the tiles customized in this fashion will likely have less overhead than a more general overlay that is compatible with a larger class of applications. In this context, tile customization can be seen as an enabler of a trade-off between overlay generality and lower resource overhead.

The second major challenge is the integration of the overlay with the FPGA I/O,
a memory subsystem, and a host CPU as well as interfacing the overlay with the full application. Our work focuses on achieving high-performance of DFGs that represents parallel code sections (e.g., parallel loops or data-parallel kernels). One way of interfacing our overlay with the CPU is by streaming the data into the overlay inputs from the CPU and by streaming overlay output data back to the CPU. However, in a larger application, the calls to a parallel section may be surrounded with complex control flow or the applications may have complex patterns of data access and reuse. These application characteristics require more sophisticated communication and synchronization mechanism than the direct streaming between the host CPU and the overlay. Consequently, the design of a generalized interface between the larger application and the overlay is essential for leveraging our work within a wide range of applications.

Certainly, the availability of the DFG-to-FPGA tool opens the opportunity to use this tool instead of using the overlay. This tool allows the user to “harden” the DFGs—implement a hard-wired DFG directly in FPGA fabric—as a means of avoiding the resource overhead of the overlay. If the application is mature and its functionality does not change often then using the DFG-to-FPGA is probably the optimal approach. However, by using both the DFG-to-overlay and DFG-to-FPGA tools, the user would have the ability to trade-off between compile time and resource usage, which is not possible if only one tool is used. A combined use model of the two tools can speed up the development of new applications. The user can use the DFG-to-overlay tool for fast development cycles in the early prototyping stages. Subsequently, once the application functionality is stable, the resource overhead of the overlay can be recovered by compiling the DFG with the DFG-to-FPGA tool. Furthermore, the recovered resources can be used to increase the throughput by replicating the DFG.

Despite the advantage of lower resource use, the DFG-to-FPGA tool still involves the use of the CAD tool, which prevents its use in scenarios where short compile time and fast reconfiguration are paramount. One scenario in which the overlay is necessary is just-in-
time compilation to the FPGA [46], which requires fast compilation and reconfiguration times. Another scenario where the use of overlay may be necessary is for applications that have many distinct computationally intensive phases that execute at different times. Implementing all of the phases on the FPGA may be impractical and inefficient since only a few phases may be active at a time, or alternatively time-multiplexing the phases by reconfiguring the FPGA between the phases may be prohibitive in terms of reconfiguration time. In contrast, the faster reconfiguration time of overlays has the potential to enable time-multiplexing of a large number of distinct phases on the overlay.

Finally, in this work we manually extract DFGs from parallel code sections. This use model places an undesirable burden on the user who must construct the DFG manually. Clearly, a tool that would translate a parallel code section into a DFG is another key aspect that is necessary to facilitate the use of our overlay architecture. Research on this tool has already been undertaken by others and early results show promise that such tool is indeed feasible [46].

9.3 Future Directions

As discussed in the previous section, there are several major challenges that need to be tackled in order to realize the promise of this work. Future work can extend our work in a number of directions that aim to address these challenges.

The first direction is to reduce the overlay’s resource overhead. Our results show that the high use of routing-only cells is a major contributor to the resource overhead of overlay prototypes. Thus, future work can explore optimizing the overlay’s routing architecture. This work can explore the use of larger routing hops (multi-cell bypass), other routing topologies, and dedicated routing blocks. In addition, the use of routing-only cells could be reduced by exploring custom functional layouts whose sizes and topologies are closely matched to needs of an application domain’s DFGs. Similarly, the routing data-path
widths and the depth of D-FIFOs could also be closely matched to the needs of the DFGs.

Another approach to reducing the overlay’s overhead is to improve the design of functional units. They could be designed to use less logic resources by better using the dedicated hardware within the DSP blocks, both for integer and floating-point functions. Thus, the FPGA’s logic resources would be primarily dedicated to the programmability of the overhead, thus lessening the overall resource usage of the overlay. Furthermore, our cell design with multiple functional units could be improved to allow these functional units to be active at the same time and also feed data directly among themselves. This would allow multiple compute nodes of a DFG be mapped to the same cell, possibly reducing the routing resources that are needed to connect the compute nodes.

Our current overlay prototypes are not integrated with a data source (e.g., a memory subsystem), and our experimental results are generated with an idealized data source. However, the overlay could potentially be integrated with any data source that is independent of the overlay and that has an elastic interface (i.e., stall/valid synchronization). One possible direction is to surround the overlay with a number of tiles that implement the data source system and that connect to the overlay via its elastic I/O ports. On one end, this data source system would interface with the host CPU (e.g., via a PCIe interface), the FPGA’s off-chip memory or the FPGA I/O (e.g., a video-in). On the other end, the system would feed data into the overlay and drain it out the overlay via the overlay’s elastic I/O ports. Some of the candidate data source systems are stream buffers (e.g., sliding-windows [204]), scratchpad buffers [205, 206], and a cache hierarchy [14, 205, 207]. It is expected that different overlay instances would use different types of data source systems. For example, one overlay instance may be surrounded with data source tiles that implement sliding-windows and another one with a mix of small caches and sliding-windows.

In addition to surrounding the overlay with an independent data source system, the
overlay could also include various types of memory blocks within the overlay mesh itself. For example, a number of scratchpad buffers or small caches could be arranged in columns within the overlay. This would enable storing of temporary results within the overlay and also caching of input data for temporal reuse. In order to connect to the surrounding overlay’s cells, these memory blocks could expose an number of elastic interface ports.

An alternative memory interface model is to tightly integrate the overlay and the host CPU on a platform that provides a low-latency shared memory between the CPU and the FPGA [29]. In this model, the overlay’s memory subsystem could be designed to enable the use of DFGs with irregular data access patterns and fine-grained interaction with the CPU.

Future work on interfacing the overlay with a host CPU can build upon the board support packages (BSPs) that are provided by the FPGA vendors and board vendors for the off-the-shelf accelerator boards [208, 209]. BSPs include a number of verified I/O subsystem IPs such as DMA, DDR, and PCIe controllers that can be used to create a full system.

A third direction for future work is to automate the extraction of DFGs from applications. In this work, DFGs are manually extracted from dedicated parallel code sections. Future work can explore front-end compiler passes that would analyze a large application and identify parallel code sections that are suitable for mapping to the overlay and then extract DFGs from them. Alternatively, a user could be given control—via compiler directives—over which code sections should get mapped to the overlay. Future work could also pursue back-end compiler passes that optimize the DFGs for the overlay. One of the possible optimizations is partitioning large DFGs that do not fit on the overlay. Another avenue for compiler research is JIT compilation to the overlay, which could enable user-transparent run-time mapping of hot segments of machine instructions onto the FPGA.

In addition to future directions that aim to address the most pressing outstanding
challenges, future work can also address several other limitations of this work and/or extend its scope. We outline the more salient ones in the remainder of the section.

In this work, the functional layouts of the prototype overlays have been manually designed. A possible direction for future work is to explore the automatic generation of overlay functional layouts for common application domains (e.g., image processing, financial, linear algebra). The required mix of FU types can be learned by profiling the target applications. Similarly, the arrangement of the FUs within the functional layout could be guided by the producer-consumer locality of operations in target applications.

The overlay architecture proposed in this thesis can be further augmented in a number of other ways. There are several approaches for extending the overlay’s functional units. Since our mesh-of-FUs architecture is designed for executing data flow graphs, it could be extended with a dedicated support for control flow, both for branches (by using select units and predication edges) as well as loops (by using iterator units and feedback edges). In addition, functional units could be extended with internal feedback paths, which would allow FUs to have state and could also be used to implement feedback edges in the graphs.

Another avenue for extending the overlay’s functionality is to use a mix of functional units of different data-types and precisions. One approach for enabling the use of different data-types is to add dedicated type-conversion functional units to the overlay. In terms of data precision, overlay’s 32-bit routing paths could carry vectors of lower precision data-types in addition to 32-bit values. For example, the routing could carry two 16-bit or four 8-bit numbers. This could be combined with functional units that operate on vectors of lower precision data (e.g., vector-add of four 8-bit integers).

In terms of the control mechanism for the overlay’s pipelines, one could explore a hybrid inelastic/elastic mechanism in which the overlay is broken up into multiple inelastic regions, with each region using a common enable signal while elastic buffers are used only on the region boundaries. This approach would trade-off flexibility (i.e., re-converging paths within the region must be balanced for correctness and all units move in lockstep)
for lower resource usage (i.e., there is less elastic buffers). Furthermore, the common enable signal of a region would have to be synchronized to the region’s boundary elastic buffers, which may impact \( f_{\text{MAX}} \). The FIFOs in our architecture are MLAB-based, but they could also be implemented using M20Ks, which would allow for a set of very deep balancing FIFOs in the overlay. The mapping algorithm could potentially map highly imbalanced paths to those FIFOs.

There are several ways in which the overlay reconfiguration could be extended. In our architecture, constants are brought into the FUs via overlay inputs and routing. This can be expensive for DSP applications with many coefficients (e.g., a convolution). Thus, a dedicated and more resource-efficient mechanism could be added to the overlay. One candidate approach is to extend the overlay’s reconfiguration mechanism and its bitstream to handle the loading of constant data. Another direction for augmenting the reconfiguration mechanism is adding the ability to store multiple configuration bitstreams. This would enable low-latency time-multiplexing (i.e., context switch) between different DFGs or different parts of a large DFG.

The tile-based bottom-up flow presented in this work explores a tile library containing fewer than one hundred tiles and only two template floorplans. A compelling way to extend our tiled-based bottom-up flow is to build a large library of template floorplans and pre-compiled tiles. The flow could be extended such that new overlays are compiled to the best-matching template floorplan with only the tiles that do not exist in the library being compiled from scratch.

The tile library can be coupled with tools for the automatic generation of application-customized functional layouts. Future work can explore how to combine these in order to reduce overlay overhead.

The tile-based bottom-up flow could also be integrated and automated into existing FPGA CAD tools. Other forms of inter-tile pipelining could be explored in addition to the inter-tile EBs proposed in this work.
Furthermore, the tile-based bottom-up flow can also be extended by exploring the CAD techniques for tile relocation, the partial reconfiguration of tiles, and the range of tile sizes, as well as finding ways to further reduce tile stitching time. In addition, the scalability of the bottom-up and top-down flows on larger devices can also be investigated.

Another direction for future work would be to conduct an in-depth comparison of the performance, scalability and resource overhead of different overlay architectures. This line of inquiry could provide useful insights for the design of hybrid overlay architectures that would incorporate the key strengths of mesh-of-FUs, VLIW, and vector overlay architectures.

The overlay architecture was implemented on a Stratix IV device using Altera’s CAD tool. However, all of the overlay’s building blocks (FUs, DDPUs, and routing logic) can be implemented in any FPGA fabric. Hence, this work can be extended by porting the overlay architecture to devices provided by other FPGA vendors (e.g., Xilinx). In addition, overlay instances with larger meshes could be implemented on larger Altera devices such as Stratix V, Arria 10 and the upcoming Stratix 10.

Finally, our work demonstrates scalability up to 384 overlay cells and 170 FUs. A next step is to investigate the scalability of both the overlay architecture and the mapping algorithm to 1000’s of FUs (on larger devices, such as Altera’s Stratix 10). This endeavour may expose limitations in our work that we are not aware of and thus provide future research directions that are beyond those that we discussed in this section.
Appendix A

DFG-to-FPGA Tool

The DFG-to-FPGA tool takes a DFG description as input and produces a balanced pipelined circuit that implements the DFG with direct hard-wired connections between the FUs, i.e., without the resources needed for the programmability of the overlay (e.g., routing muxes).

The goal of the tool is to provide a reference for measuring the resource cost of the overlay’s programmability to implement DFGs. Thus, we keep other design dimensions (such as throughput, $f_{\text{MAX}}$, and latency insensitivity) invariant across the DFG-to-FPGA and DFG-to-overlay implementations. The DFG-to-FPGA tool inherits the following high-level design goals from the DFG-to-overlay mapping tool:

1. Achieve maximum DFG instance throughput; that is, the DFG should be balanced.

2. Achieve an average $f_{\text{MAX}}$ of 400 MHz across the DFG set.

3. Elastic pipelines and fine-grained stallability, thus enabling the independent stalling of each FU and each overlay’s I/O port. These features give FUs and I/Os the built-in capability for interfacing with various types of memory subsystems whose latency and throughput can vary at run-time (i.e., they are not statically predictable).

As a result of the shared goals, the difference between these two approaches is primarily......
reflected in resource usage, compile time, and DFG latency.

It should also be noted that alternative implementations of the DFG-to-FPGA tool are possible. For example, a tool may target an $f_{\text{MAX}}$ that is lower than 400 MHz (e.g., 200 MHz), or it may use statically scheduled inelastic pipelines (they are latency sensitive). This type of tool may consume fewer resources than our DFG-to-FPGA tool implementation, but it would also differ from the DFG-to-overlay mapping across multiple design dimensions. Hence, unlike our tool, this hypothetical tool does not isolate the comparison only to the resource overhead of the overlay’s programmability.

The DFG-to-FPGA tool maps each DFG node to a dedicated FU, and each DFG edge directly connects the FUs via a pipelined and latency-balanced data-path. The tool uses the same set of FUs as those used in the overlay architecture but with the programmability stripped out, i.e., each FU only implements the operation of the DFG node that is mapped to it. Furthermore, just like in the overlay cell, each FU is surrounded by elastic buffer DDPU’s—specifically, D-SEB2 at the FU’s inputs and D-BUF2 at the FU’s output. These EBs serve to pipeline both the data-path signals and the synchronization signals.

Join synchronization logic is inserted between the input DDPU and the FU, and fork synchronization logic is inserted after the output DDPU. The synchronization logic is hard-wired, which means that it synchronizes with a fixed set of producer and consumer FUs. This is different from the overlay cell where the synchronization logic is programmable.

Next, the DDPU FIFOs are inserted on the paths between the FUs to balance the pipelines. We use MLAB-based D-FIFOs of up to 64 in depth, and we manually determine the insertion points for the balancing FIFOs. This is done by inserting NOPs on the edges of the original DFG before passing it on to the DFG-to-FPGA tool. On paths where the required balancing depth is larger than 64, multiple NOPs are inserted. As a result, the tool will implement the required balancing depth as multiple chained D-FIFO units. This NOP insertion technique could be automated by first allowing the tool to insert
balancing FIFOs of any required depth, and then following that with a pass that splits
them into multiple chained D-FIFOs.

Finally, the tool inserts a D-SEB2 elastic buffer between all back-to-back D-FIFOs
for additional pipelining of the data-path and synchronization signals. Consequently, a
path with a large latency imbalance will have multiple elastic buffers inserted on it. In
effect, this method uses the degree of latency imbalance of a path as a proxy for the
amount of pipelining required on that path.
Appendix B

Benchmark DFGs

This appendix gives a graphical representation of the DFGs that we use in this study.
Figure B.1: N-Body DFG
Figure B.2: N-Body-2x DFG
Figure B.3: N-Body-3x DFG
Figure B.4: BlackScholes DFG
Figure B.5: MatMul DFG (SP FP and int32 variants have the same topology)
Figure B.6: MatMulandAdd DFG (SP FP and int32 variants have the same topology)
Figure B.7: RGB2YIQ DFG
Figure B.8: RGB2YIQ-2x DFG
Figure B.9: RGB2YIQ-4x DFG
Figure B.10: RGB2YIQ-5x DFG
Figure B.11: SAD DFG
Figure B.12: SAD-2x DFG
Figure B.13: GaussianBlur DFG
Figure B.14: GaussianBlur-2x DFG
Bibliography


[27] BEEcube. URL: http://beecube.com (cit. on p. 3).


BIBLIOGRAPHY


