Improving the Run-time and Memory Scalability of FPGA CAD Algorithms

by

Scott Yin Lunn Chin

B.Eng. Electrical Engineering, University of Victoria, 2003
M.A.Sc. Electrical and Computer Engineering, University of British Columbia, 2006

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES
(Electrical and Computer Engineering)

The University Of British Columbia
(Vancouver)

August 2011

© Scott Yin Lunn Chin, 2011
Abstract

Field-Programmable Gate Arrays (FPGAs) are pre-fabricated integrated circuits that can be configured as any digital system. Configuration is done by Computer-Aided Design (CAD) tools. The demand placed on these tools continues to grow as advances in process technology continue to allow for more complex FPGAs. Already, for large designs, compile-times of an entire work day are common and memory requirements that exceed what would be found in a typical workstation are the norm. This thesis presents three contributions toward improving FPGA CAD tool scalability.

First we derive an analytical model that relates key FPGA architectural parameters to the CAD place-and-route (P&R) run-times. Validation shows that the model accurately captures the trends in run-time when architecture parameters are changed.

Next we focus on the CAD tool’s storage requirements. A significant portion of memory usage is in representing the FPGA. We propose a novel scheme for this representation that reduces memory usage by 5x-13x at the expense of a 2.26x increase in routing run-time. Storage is also required to track metrics used during routing. We propose three simple memory management schemes for this component that further reduces memory usage by 24%, 34%, and 43% while incurring a routing run-time increase of 4.5%, 6.5%, and 144% respectively. We also propose a design-adaptive scheme that
reduces memory usage by 41% while increasing routing run-time by only 10.4%.

Finally, we focus on the issue of long CAD run-times by investigating the design of FPGA architectures amenable to fast CAD. Specifically, we investigate the CAD run-time and area/delay trade-offs when using high-capacity logic blocks (LBs). Two LB architectures are studied: traditional and multi-level. Results show that for the considered architectures, CAD run-time can be reduced at the expense of area; speed improved. For example, CAD run-time could be reduced by 25% with an area increase of 5%. We also show that the run-time trade-offs through these architectural changes can be complementary with many previously published algorithmic speed-ups.
Preface


Parts of this dissertation’s research content have been published. A preliminary version of Chapter 3 has been published in [3] and the completed work has been submitted.
for review in [6]. The work from Chapter 4 was published in [1] and [2]. Portions of Chapter 5 were published in [5] and the complete study has been submitted for review in [7]. An overview of this work was published in [4]. In all cases, I designed and conducted the research, analyzed the results, and prepared the manuscripts with guidance and editing from my supervisor Dr. Steve Wilton.
# Table of Contents

Abstract ........................................................................ ii  
Preface ......................................................................... iv  
Table of Contents ............................................................. vi  
List of Tables ................................................................... xi  
List of Figures ................................................................... xii  
Glossary .......................................................................... xv  
Acknowledgments ............................................................. xvii  
Dedication ....................................................................... xviii  

1 Introduction ................................................................. 1  
1.1 Motivation ............................................................... 1  
1.2 Scalability ............................................................... 3  
1.3 Research Objective and Contributions ...................... 5  
1.3.1 Analytical Models Relating FPGA Architecture and FPGA CAD Run-time ........................................... 6  
1.3.2 Storage-Efficient Representations for FPGA Architectures ......................................................... 8  
1.3.3 Towards Scalable CAD Run-time Through Architecture ............................................................. 9  
1.4 Thesis Organization .................................................. 10  

2 Background and Related Work ........................................ 11
2.1 FPGA Architecture Review ............................................. 11
  2.1.1 Logic Block Architecture ........................................ 13
  2.1.2 Routing Architecture ............................................. 14
2.2 FPGA CAD Review ..................................................... 16
  2.2.1 Placement ......................................................... 18
  2.2.2 Routing .......................................................... 19
2.3 Related Work on FPGA CAD Storage Requirement Scalability .... 21
2.4 Related Work on FPGA CAD Run-time Scalability ................ 22
  2.4.1 Improving Classical Techniques .................................. 22
  2.4.2 Alternative Heuristics .......................................... 23
  2.4.3 Parallelization .................................................... 24
  2.4.4 Incremental Compile .............................................. 26
  2.4.5 Floorplanning and Hierarchy .................................... 26
  2.4.6 Hardware Acceleration .......................................... 27
2.5 Related Work on Affecting FPGA CAD Run-time through Architecture 28
2.6 Related Work on Analytical FPGA Models .......................... 29
2.7 Summary .................................................................... 30

3 Analytical Models Relating Architecture and CAD Run-time ........ 31
  3.1 Framework ............................................................ 32
    3.1.1 Architectural and Circuit Assumptions ....................... 32
    3.1.2 CAD Assumptions ............................................... 33
    3.1.3 Logic Utilization Model Usage ................................ 34
  3.2 Model Relating Placement Run-time to Architecture .............. 34
    3.2.1 Average Time Per Swap $C_{\text{swap}}$ .................... 35
    3.2.2 Swaps Attempted Per Temperature Step $N_{\text{swaps}}$ .... 36
    3.2.3 Number of Temperature Steps $N_{\text{steps}}$ ............... 36
      3.2.3.1 Temperature Update Scheme ............................. 36
      3.2.3.2 Final Temperature $T_f$ ................................. 38
      3.2.3.3 Initial Temperature $T_o$ ............................... 40
    3.2.4 Placement Run-time Model Summary .......................... 43
  3.3 Modeling Routing Run-time ........................................ 43
    3.3.1 Number of Routing Iterations $N_{\text{iter}}$ .................. 45
3.3.2 Average Number of Expansions $N_{exp}$ .................................. 46
3.3.3 Expansions to Leave a Logic Block $E_{LBout}$ .......................... 46
3.3.4 Expansions Per Wire Segment $E_{wire}$ ................................. 48
  3.3.4.1 Number of CB Connections From a Wire $E_{cbox}$ ............ 48
  3.3.4.2 Number of SB Connections from a Wire $E_{sbox}$ (Bi-
directional Routing) ........................................... 49
  3.3.4.3 Number of SB Connections from a Wire $E_{sbox}$ (Single
                 Driver Routing) ....................................... 50
3.3.5 Wires Explored $W$ .................................................. 50
3.3.6 Routing Run-time Model Summary ................................... 53
3.4 Run-time Model Validation ............................................. 54
  3.4.1 Benchmarks ....................................................... 54
  3.4.2 Model Calibration ................................................ 55
  3.4.3 Placement Run-time Validation Results .......................... 55
  3.4.4 Routing Run-time Validation Results ........................... 56
3.5 Example Applications .................................................. 58
  3.5.1 Architectural Choices Under A CAD Run-Time Constraint .... 60
  3.5.2 CAD Run-Time Optimization with Die Area Constraint ....... 61
3.6 Summary ..................................................................... 62

4 Storage-Efficient Representations for FPGA Architectures ......... 63
  4.1 Overview and Further Motivations ..................................... 63
  4.2 Routing Resource Graph .............................................. 66
  4.3 Architectural Data ........................................................ 67
    4.3.1 Tile Type Representation ........................................ 69
    4.3.2 Connections Within the Same Tile Type ...................... 69
    4.3.3 Connections Between Different Tiles of the Same Instance .. 70
    4.3.4 Connections Between Different Tile Types ................... 71
    4.3.5 Modelling Long Wires ............................................ 72
    4.3.6 Depopulated Connection Boxes and Switch Boxes .......... 73
    4.3.7 Changes To The Routing Algorithm ........................... 73
  4.4 Architectural Data - Experimental Results ........................ 76
    4.4.1 Experimental Framework For $N_{array}, W$, and $L$ ........... 77
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.2</td>
<td>Experimental Results for Array Size $N_{array}$</td>
<td>77</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Experimental Results for Channel Width $W$</td>
<td>80</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Experimental Results for Wire Length $L$</td>
<td>81</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Impact on Routing Time</td>
<td>82</td>
</tr>
<tr>
<td>4.4.6</td>
<td>Impact on Priority Queue Memory Requirements</td>
<td>84</td>
</tr>
<tr>
<td>4.4.7</td>
<td>Impact on Routing Solution</td>
<td>86</td>
</tr>
<tr>
<td>4.5</td>
<td>Temporary Routing Data</td>
<td>86</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Lifespan of TRDOs</td>
<td>86</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Simple Memory Management Schemes</td>
<td>88</td>
</tr>
<tr>
<td>4.5.2.1</td>
<td>Scheme 1</td>
<td>88</td>
</tr>
<tr>
<td>4.5.2.2</td>
<td>Scheme 2</td>
<td>88</td>
</tr>
<tr>
<td>4.5.2.3</td>
<td>Scheme 3</td>
<td>88</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Design-Adaptive Memory Management Scheme</td>
<td>89</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Design-Adaptive Memory Management Scheme</td>
<td>89</td>
</tr>
<tr>
<td>4.5.5</td>
<td>Design-Adaptive Memory Management Scheme</td>
<td>89</td>
</tr>
<tr>
<td>4.5.6</td>
<td>Design-Adaptive Memory Management Scheme</td>
<td>89</td>
</tr>
<tr>
<td>4.6</td>
<td>Experimental Results - Temporary Routing Data</td>
<td>90</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Impact on Memory</td>
<td>90</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Impact on Run-Time</td>
<td>92</td>
</tr>
<tr>
<td>4.6.2.1</td>
<td>Cache Performance Overhead</td>
<td>92</td>
</tr>
<tr>
<td>4.6.2.2</td>
<td>Additional Code Overhead</td>
<td>93</td>
</tr>
<tr>
<td>4.7</td>
<td>Summary</td>
<td>94</td>
</tr>
</tbody>
</table>

5 Towards Scalable CAD Run-time Through Architecture   

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Architecting High-Capacity Logic Blocks</td>
<td>98</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Flat Logic Block Architecture</td>
<td>98</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Multi-Level Logic Block Architecture</td>
<td>100</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Logic Block I/O and Interconnect Demand</td>
<td>100</td>
</tr>
<tr>
<td>5.1.3.1</td>
<td>Equation Derivation</td>
<td>101</td>
</tr>
<tr>
<td>5.1.3.2</td>
<td>Equation Validation</td>
<td>103</td>
</tr>
<tr>
<td>5.2</td>
<td>Experimental Methodology</td>
<td>103</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Architectural Assumptions</td>
<td>104</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Area Model, Delay Model and Circuit Design</td>
<td>105</td>
</tr>
<tr>
<td>5.2.3</td>
<td>CAD and Benchmarks</td>
<td>106</td>
</tr>
<tr>
<td>5.3</td>
<td>Experimental Results: Flat Logic Block Architectures</td>
<td>108</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Area</td>
<td>109</td>
</tr>
</tbody>
</table>
List of Tables

Table 3.1 Model inputs .............................................. 33
Table 3.2 Placement run-time model summary .......................... 44
Table 3.3 Routing run-time model summary .......................... 53
Table 3.4 Benchmarks details ........................................ 54
Table 3.5 Parameter values for routing model validation ............. 55
Table 3.6 Parameter values for placement model validation and applications 56
Table 4.1 Valid coordinates for tiles in a typical homogeneous FPGA ..... 75
Table 4.2 Dependencies for validity of edges .......................... 76
Table 4.3 Device sizes for some commercial FPGAs ..................... 79
Table 4.4 CAD run-time results for synthetic benchmark circuits in seconds . 83
Table 4.5 CAD run-time results for MCNC benchmark circuits in seconds . 83
Table 4.6 Dynamic memory management scheme memory reductions ........ 92
Table 4.7 Routing run-time overhead of each scheme .................. 93
Table 5.1 Synthetic benchmark circuits ................................ 108
Table 5.2 Comparison of results against related work .................. 124
Table B.1 Architecture parameters ..................................... 155
Table B.2 Number of nodes of each resource type per tile. And number of edges from each resource type .................. 157
Table B.3 Example node and edge memory usage ........................ 157
Table B.4 Model validation architecture space .......................... 158
List of Figures

Figure 1.1  Altera Quartus II recommended RAM for five device generations. 4
Figure 1.2  Relative increase in FPGA logic capacity and CPU speeds. 5

Figure 2.1  FPGA architecture overview 12
Figure 2.2  FPGA logic block architecture overview 13
Figure 2.3  FPGA routing architecture overview 15
Figure 2.4  Typical FPGA CAD flow 17

Figure 3.1  Two placements of a net with the same half perimeter bounding box. 42
Figure 3.2  Single-driver routing architecture LB output pin connections. 47
Figure 3.3  Bi-directional routing switch box connections for $L = 2$ 49
Figure 3.4  Single-driver routing switch box connections for $L = 2$ 49
Figure 3.5  Routing search wavefront for breadth-first and depth-first search. 52
Figure 3.6  Placement run-time model validation 57
Figure 3.7  Routing run-time model validation 59
Figure 3.8  Application 1: logic architecture tradeoffs with placement run-time constraint 60
Figure 3.9  Application 2: CAD run-time optimization with die area constraint 62

Figure 4.1  Example portion of the routing resource graph 66
Figure 4.2  Illustration of key idea behind new architecture representation. 68
Figure 4.3  The five tiles required to model an homogeneous FPGA. 69
Figure 4.4  Connections within the same tile instance. 70
Figure 4.5  Connections to different instances of the same tile type. 70
Figure 4.6  Assumed tile coordinate system 71
Figure 4.7  Instance-dependant connections. 72
Figure A.1 Placement run-time model validation for bi-directional routing architectures. ........................................ 153
Figure A.2 Routing run-time model validation for bi-directional routing architectures. ........................................ 154
Figure B.1 Equation validation (a) through (b) ........................................ 158
Figure B.1 Equation validation (c) to (g) ........................................ 159
Figure C.1 Additional validation results ........................................ 161
## Glossary

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>BLE</td>
<td>Basic Logic Elements</td>
</tr>
<tr>
<td>CAD</td>
<td>Computer-Aided Design</td>
</tr>
<tr>
<td>CB</td>
<td>Connection Box</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Arrays</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>HPWL</td>
<td>Half Perimeter Bounding Box Wirelength</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuits</td>
</tr>
<tr>
<td>JIT</td>
<td>Just-in-Time</td>
</tr>
<tr>
<td>LB</td>
<td>Logic Blocks</td>
</tr>
<tr>
<td>LUT</td>
<td>Look-Up Tables</td>
</tr>
<tr>
<td>MUX</td>
<td>Multiplexor</td>
</tr>
<tr>
<td>PE</td>
<td>Processing Element</td>
</tr>
<tr>
<td>PQ</td>
<td>Priority Queue</td>
</tr>
<tr>
<td>QA</td>
<td>Quantum Annealing</td>
</tr>
<tr>
<td>RR</td>
<td>Routing Resource</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>RNG</td>
<td>Random Number Generator</td>
</tr>
<tr>
<td>RRG</td>
<td>Routing Resource Graph</td>
</tr>
<tr>
<td>SA</td>
<td>Simulated Annealing</td>
</tr>
<tr>
<td>SB</td>
<td>Switch Box</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>STWL</td>
<td>Steiner Tree Wirelength</td>
</tr>
<tr>
<td>TRDO</td>
<td>Temporary Routing Data Object</td>
</tr>
<tr>
<td>VPR</td>
<td>Versatile Place-and-Route, a commonly-used academic FPGA place-and-route tool</td>
</tr>
</tbody>
</table>
Acknowledgments

My foremost gratitude goes to my academic supervisor, Dr. Steve Wilton. Throughout my studies at UBC, Dr. Wilton provided me with a tremendous amount of support, guidance and opportunities to learn. His dedication to the growth of his students is admirable. He has helped me achieve more out of my graduate experience than I could possibly have imagined. All I can say in this small amount of space is that I consider myself extremely fortunate to have had the opportunity to work under his supervision.

In addition to my supervisor, I thank all the members of my examining committees for taking the time to provide insightful feedback during the various stages of my research: Dr. Sathish Gopalakrishnan, Dr. Shahrriar Mirabbasi, Dr. Res Saleh, Dr. Guy Lemieux, Dr. Tor Aamodt, Dr. Alan Hu, Dr. Konrad Walus and Dr. Russell Tessier.

I thank all my lab mates for the technical discussions, the laughs, the camaraderie, and the company on those late nights prior to deadlines. The list of names is really too long and they know who they are :). Their company has made working in the lab a real pleasure. I have many fond memories, and I look forward to the continued friendships.

Finally, I greatly appreciate the financial support provided by the National Sciences and Engineering Research Council of Canada, the University of British Columbia, and Altera Corporation.
Dedication

I dedicate this thesis to my wife Marie, my parents Philip and Yvonne, my extended family, and my friends. They have shaped me into the person that I am today.
Chapter 1

Introduction

1.1 Motivation

Field-Programmable Gate Arrays (FPGA) are pre-fabricated Integrated Circuits (IC) that can be configured to implement any digital system. Using an FPGA to implement a digital system provides many advantages compared to developing a custom Application Specific Integrated Circuit (ASIC). When developing an ASIC, the steps from the preparation of the design for fabrication to the point of receiving a working chip require significant expertise, time, and money. The major advantage of using an FPGA is that it circumvents many of these steps. The FPGA vendor has taken care of the need for some expertise by handling the transistor-level design, layout, low-level electrical effects, fabrication, and validation of the chip itself. Since the FPGA is immediately ready for programming and use, time-to-market for the product is reduced. Furthermore, the design and fabrication cost of the FPGA chip is amortized across all of the FPGA vendor’s customers. This makes small to medium volume IC products economically feasible.

However, using an FPGA does come with some disadvantages. Since FPGAs are
generic and programmable, they contain a significant amount of circuitry overhead. Compared to a custom chip, an FPGA will require more silicon area to implement the same digital system. The extra circuitry also leads to additional capacitance within the chip. This in turn leads to slower circuit speeds, and higher power dissipation. The study in [68] reports an average 21x increase in area, 4x decrease in speed, and 12x increase in power consumption. Despite these disadvantages, the trade-offs are acceptable for many applications.

FPGAs are configured by setting the state of a large number of configuration memory bits inside the FPGA. Each bit controls one or more programmable switch or participates in determining the function of a programmable logic element. Modern FPGAs contain millions of configuration bits so determining the values of these bits manually is not feasible. Instead, FPGA users employ sophisticated Computer-Aided Design (CAD) software tools which are normally supplied by the FPGA vendor. These tools convert a digital design, often specified in a hardware description language (HDL), to a bitstream using a series of transformations. In brief, the HDL is first converted to logic gates and mapped to the basic programmable function blocks in the FPGA architecture. These function blocks are then mapped to physical locations on the FPGA (placement). Finally, paths through the FPGAs programmable routing network are determined to implement the connections between the function blocks (routing). At the end of these steps, the functionality of the resources on the FPGA has been determined, and the CAD tool can produce the appropriate values for all configuration bits.

Advances in process technology have allowed for a dramatic increase in the capacity of FPGAs and this scaling is continuing at a steady pace\(^1\) This has shifted the use of FPGAs from implementing simple glue logic to implementing entire systems. How-\footnote{In fact, FPGA vendors are often early adopters and drivers of new process technologies \cite{53, 87, 95}.}
ever, this scaling places increasing demands on the FPGA CAD tools. Already, for very large designs, compile-times of an entire work day are common, and memory requirements that exceed what would be found in a common desktop workstation are the norm [13] [126]. As FPGAs continue to grow, the problem will become worse. Unless the scalability of FPGA CAD tools is addressed, the long run-times and large memory requirements will become a hindrance to future FPGA scaling, leading to increased costs and design times for companies that use these devices. The focus of this thesis is to address this scalability problem. More specifically, this thesis aims to improve the run-time and memory-requirement scalability of FPGA CAD tools.

1.2 Scalability

The concept of scalability is central to this thesis, and thus we discuss its meaning in this subsection. As process technology improves, the logic capacity of state-of-the-art commercial FPGAs increases. Moore’s Law predicts that the capacity of an integrated circuit will increase exponentially over time. This means that the number of logic elements that must be placed, and the number of nets that must be routed in a fully-utilized FPGA also increase exponentially over time.

The run-time of an FPGA CAD tool depends directly on the size of the FPGA and user circuit. As we will show later in Chapter 3, the order of the run-time of a placement algorithm grows faster than linear with respect to the number of placeable logic elements [31]. The memory requirements are typically determined by the number of available routing resources on the chip; assuming an architecture that supports a Rent’s Exponent\(^2\) greater than 0.5\(^3\), the number of routing tracks also grows faster than linear with the

\(^2\)The Rent’s Exponent is a parameter in Rent’s Rule, an empirically observed relationship between interconnect demand and number of logic gates [73].

\(^3\)For example, the Altera Cyclone and Apex architectures have been reported to support Rent’s Expo-
Figure 1.1: Altera Quartus II 64-bit v10 recommended RAM for five device generations [13]. Data based on Stratix EP1S80, Stratix II EP2S180, Stratix III EP3SE260, Stratix IV EP4SE820, Stratix V 5SGSB8 devices.

The trend predicted by Moore’s Law also applies to the processors used to run the FPGA CAD algorithms. As more transistors become available, processor architects are now using them to implement more processor cores rather than making each core faster. Assuming algorithms can be perfectly parallelized, this leads to a somewhat linear relation between processing power and transistor count. However, perfect parallelism is, at current, rarely possible except for a small set of problems [81].

The summary of the above is that, over time, the processing requirements of FPGA CAD tools will increase faster than O(n) where n is the available transistors in a state-of-the-art device.
Figure 1.2: Relative increase in FPGA logic capacity (number of logic elements), and uni-processor CPU speeds since 1999. CPU speeds are based on SPEC CINT2000 benchmarks [81].

the-art integrated circuit, while the processing capacity available in modern processors will increase at or less than $O(n)$. This is highlighted in Figure 1.2 which compares the trends in uni-processor CPU speeds and FPGA logic capacity.

These trends in memory requirement and run-time are not sustainable, and this issue is the focus of this thesis. The long-term goal of this research is to create a CAD tool and architecture such that the CAD run-time and memory requirements scale at the same rate as the anticipated scaling of processing capacity (i.e., at or less than $O(n)$).

1.3 Research Objective and Contributions

To summarize the previous sections, the goal of this thesis is to improve the run-time and memory scalability of FPGA CAD tools as FPGA capacity increases due to process technology scaling. This dissertation presents three contributions towards this goal:
1. An analytical model that relates key FPGA architecture and CAD run-time. This work was published in [38], and to our knowledge is the first published description for such a run-time model.

2. Novel algorithm-independent data-representations of FPGA architectures and dynamic memory management schemes that reduce the order at which CAD storage requirements scale. This work was published in [35, 36] and to our knowledge is the first published research focused on the memory requirements of modern FPGA CAD tools.

3. An investigation into the design of FPGA architectures that are amenable to fast CAD. We explored a concrete example by performing a detailed study on architectures with high-capacity logic blocks. This work was published in [37].

The next three subsections will briefly describe each contribution. The details for each contribution will be presented as individual chapters in this dissertation.

1.3.1 Analytical Models Relating FPGA Architecture and FPGA CAD Run-time

The first contribution of this thesis is an analytical model that relates key architecture parameters to the CAD run-time. Although studies exist that describe the worst-case computing efficiencies of the underlying CAD algorithms [57, 91, 93], the novelty of our model is that it draws out the essence of how architectural choices affect the typical performance of these algorithms when applied to FPGA CAD.

Traditional metrics for FPGA architecture optimization are speed, silicon area, and power. To optimize for these metrics, timing [19], area [19], and power [96] models have been proposed. These models work in conjunction with an experimental method-
ology: gather a suite of benchmark circuits, generate a family of architectures using a parameterized architecture model\(^4\), map the benchmarks to each architecture, evaluate the metric with the model, then choose the optimal architecture based on the metric.

Recently, analytical models have been developed that estimate some of these metrics without the need to run experiments \([43, 45, 52, 56, 62, 107]\). Analytical models have several advantages:

1. *Provide a body of theory:* An experimental methodology can show us which architectures work well, but does not give any direct insight as to why. On the other hand, analytical equations succinctly describe the relationships between the metric and architecture.

2. *Quick to evaluate:* Evaluating a set of closed-form equations is fast. In contrast, an experimental approach is very time-consuming because CAD run-times are significant. To compound the issue, sampling the architecture space of a parameterized architecture model leads to a large number of architectures. Each benchmark in the suite would need to be mapped to each of these architectures.

3. *Do not require benchmark circuits:* Large sets of realistic benchmarks are difficult to acquire as they often contain proprietary intellectual property. In addition, some characteristics of the circuit (e.g. circuit size and interconnect demand) may affect the metric under optimization. Acquiring circuits that also meet the range of desired characteristics can be difficult.

The major challenges tackled in the derivation of our model were, balancing simplicity with accuracy, and bringing out the architecture’s impact on CAD run-time by

\(^4\)For example, the parameterized FPGA architecture model found in the academic VPR place-and-route CAD suite \([19, 82]\)
keeping the model as independent as possible of other factors (such as circuit properties and code-specific implementation details). The detailed derivation, description, validation, and application of this model is described in Chapter 3.

1.3.2 Storage-Efficient Representations for FPGA Architectures

The second contribution described in this dissertation focuses on the scalability of the FPGA CAD tool’s storage requirements. Although the CAD tool’s memory usage can be reduced somewhat through code-level optimizations, the ideas presented here are high-level techniques that can apply to a wide range of FPGA routing algorithms. These techniques are novel in that they rely on insights into FPGA architectures, and lead to long-term benefits by reducing the order at which the memory requirements scale.

We focused on the storage requirements of the routing stage in the FPGA CAD flow because it is one of the most memory intensive steps [83]. The goal of the routing stage is to choose appropriate paths in the FPGA routing network to form the nets of the circuit being implemented by the FPGA. Unlike ASIC routers, FPGA routers can only use the prefabricated routing resources (prefabricated wire segments and programmable connections) in the FPGA routing network. Representing all of the available routing resources to the routing algorithm requires a significant amount of memory. This representation is necessary regardless of the routing algorithm being used. As FPGA capacity increases (over time with process technology scaling), the amount of available routing resources also increases, and in turn increases the storage requirements of the CAD tools.

By leveraging the fact that FPGA architectures are very regular, we developed a novel storage-efficient scheme to represent the routing resources. To make the ideas concrete, we implemented the techniques into VPR (an academic FPGA place-and-route tool) [19]. This allowed us to experimentally evaluate the benefits of the scheme, and
also to work out many of the details needed to make the technique work in practice. Our implementation has been released for public use as a software patch for VPR.

In addition, we proposed and evaluated three simple and one design-adaptive dynamic memory management scheme to further reduce the storage requirements needed during routing. These results are also independent of code-specific details and highlight the impact of architecture and circuit properties on CAD storage requirement. Chapter 4 will describe these techniques and results in detail.

1.3.3 Towards Scalable CAD Run-time Through Architecture

The third contribution described in this thesis focuses on the run-time scalability of the FPGA CAD tools. We proposed that FPGA architectures be designed with CAD run-time as an optimization metric. This is a divergence from the traditional architecture design process where area, speed, and power are optimized, and then the CAD algorithms are optimized for run-time once the architecture is fixed. The point is not to abandon algorithmic approaches for improving CAD run-times, but to provide another complementary avenue for dealing with it. There is very limited research on this approach.

In Chapter 5, we describe general approaches towards designing architectures that are amenable to fast scalable CAD, one of which is to design architectures that lead to a reduced problem size in the computationally expensive FPGA CAD stages. Some steps in the FPGA CAD flow take much longer than others. It may then be possible to modify the architecture so that the problem size presented to these computationally expensive stages is reduced, thus directly reducing their run-times. As a concrete example, we investigated architectures with high-capacity logic blocks. Such architectures require fewer blocks to implement a digital design, which in turn reduces the number of blocks
that need to be placed and the number of nets that need to be routed. In other words, the problem size presented to the placement and routing CAD stages is reduced. We performed a detailed architecture study on this approach that highlights the ability to trade off traditional design metrics (area and speed) for CAD run-time. As a part of this study, we derive and validate an equation for determining the amount of I/O needed by high-capacity logic blocks. Finally, we illustrate that CAD run-time reductions through architectural changes can still be complementary to algorithmic opportunities for reducing CAD run-time.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides additional background information on FPGA architecture and CAD. In addition, this chapter puts into context the research contributions described in this dissertation by summarizing related work. The first contribution, describing the new analytical model, is in Chapter 3. The second contribution, describing storage-efficient architecture representations and dynamic memory management schemes, is presented in Chapter 4. The third contribution on affecting run-time scalability through architecture is presented in Chapter 5. Chapter 6 summarizes the dissertation’s key contributions, conclusions and limitations, and describes various avenues for future research.
Chapter 2

Background and Related Work

In this chapter, we first review FPGA architectures in Section 2.1 and FPGA CAD in Section 2.2. We then frame the context of this dissertation by providing a complete literature survey on related works in FPGA CAD storage requirement scalability (Section 2.3), FPGA CAD run-time scalability (Section 2.4), affecting FPGA CAD run-time through architecture (Section 2.5), and analytical FPGA models (Section 2.6).

2.1 FPGA Architecture Review

This thesis focuses on the island-style FPGA architecture [19]. The island style architecture has been used in many generations of commercial FPGAs[1] and academic research has primarily focused on this architecture. This architecture has been commercially proven to scale in logic capacity according to Moore’s Law and thus it is appropriate for us to focus on the scalability of its associated CAD tools. Before reviewing the island-style architecture in more detail, we note that several newer vendors are producing FPGAs that diverge from this architecture. For example, Achronix Semiconductor

1Such as those from Altera, Xilinx, and Actel
Corporation has developed an asynchronous architecture [3]. The FPGAs produced by Tabula use an island-style architecture, but apply time-multiplexing to increase the effective logic capacity of the logic resources [11]. In the academic literature, hierarchical routing architectures have also been proposed [4, 30, 115]; these architectures are commercially difficult to scale however, because tree-based routing networks are difficult to lay-out in an efficient manner. We do not consider any of these architectures further.

The architecture of an FPGA consists of three fundamental components: configurable logic resources, I/O resources, and a configurable interconnect network. In an island-style architecture, these resources are arranged in a flat two-dimensional mesh as shown in Figure 2.1. The configurable logic resources are further broken down into general Logic Blocks (LB), and specialized blocks such as memory blocks and Digital Signal Processing (DSP) blocks. We refer to FPGAs that contain only LBs as homogeneous and those that also contain specialized blocks as heterogeneous. Heterogeneous FPGAs are now the norm as specialized blocks are much more efficient at implementing various common circuit elements such as memories and arithmetic operators.
2.1.1 Logic Block Architecture

FPGAs typically use Look-Up Tables (LUT) as their fundamental programmable logic resource. A $K$-input LUT can implement any combinational function of $K$ inputs by storing the corresponding truth table in the LUTs $2^K$ configuration memory bits. To implement sequential elements, LUTs are grouped with flip-flops to form Basic Logic Elements (BLE). A common grouping is to pair LUTs with flip-flops as shown in Figure 2.2a. In this organization, a configurable Multiplexor (MUX) is used to control whether the output of the BLE is sequential or combinational.

To improve density and speed, $N$ BLEs are grouped together to form an LB (also known as a cluster) as shown in Figure 2.2b [5]. The local interconnect can route any of the $N$ BLE outputs to any BLE input through a set of feedback connections. Each LB also has $I$ unique inputs which can be supplied to the BLE inputs. Typically, $I$ is less than the total number of BLE inputs within the LB since real circuits have shared nets. The interconnect within an LB (intra-cluster routing) has less capacitance than the
routing resources on the global interconnect (inter-cluster routing), and is hence faster and more power-efficient.

Commercial LBs typically contain additional circuitry such as carry-chains to efficiently implement certain arithmetic functions, control signals such as flip-flop set and resets, and clock signals to support multiple clock domains.

2.1.2 Routing Architecture

The configurable interconnect is made up of prefabricated wire segments, and programmable connections. Wire segments run along tracks and tracks are grouped together to form channels. Channels run vertically and horizontally along the rows and columns of the FPGA mesh. The channel width refers to the number of tracks per channel and is denoted by $W$. The channel width for all channels is typically the same. Horizontal (vertical) wire segments can span one or more rows (columns) of LBs. This wire segment length is denoted by $L$.

A Connection Box (CB) is the junction and topology in which global routing wire segments connect to the pins of an LB. A Switch Box (SB) is the junction and topology in which wire segments connect to other wire segments. Although maximum flexibility would be achieved by having fully connected SBs and CBs where any incident wire segment may connect to any other incident wire segment or pin, the area overhead makes this impractical. SBs and CBs are typically sparsely populated with programmable connections [74, 88]. The actual topology is important to maximize the flexibility of the programmable routing network, while minimizing area speed and power overhead.

Initially, FPGAs employed bi-directional routing networks wherein signals can be programmed to flow in either direction along a track. An example of this is shown in Figure 2.3a. Modern FPGAs now employ single-driver uni-directional routing as it leads
(a) Bi-directional routing architecture

(b) Single-driver routing architecture

Figure 2.3: FPGA routing architecture overview
to better density and speed [75]. In this type of network, the direction of signal flow for a track is pre-determined. This type of network is illustrated in Figure 2.3b.

The fraction of tracks passing through a CB that can connect to an LB input pin is denoted by $F_{cin}$. In Figure 2.3a and 2.3b, $F_{cin} = 1$ because all four tracks are accessible to the LB input pin. Similarly, the fraction of tracks to which an LB output pin can connect is denoted by $F_{cout}$. In Figure 2.3a, $F_{cout} = 0.75$ since the LB output pin can only connect to 3 out of 4 tracks.

The area, speed, and power consumption of modern FPGAs are now dominated by the routing network. To give a sense of scale, Figures 2.3a and 2.3b depict a routing network with a channel width of $W = 4$. However, in high-capacity FPGAs, the channel width is in the range of 200 and beyond (Table 4.3 in Chapter 4 shows some examples of commercial FPGA channel widths).

### 2.2 FPGA CAD Review

Due to the enormous number of resources on an FPGA that require configuration, sophisticated Computer-Aided Design (CAD) tools are used. Figure 2.4 shows a typical FPGA CAD flow. The user provides as input a Hardware Description Language (HDL) description of the circuit to be implemented on the FPGA. The output is a set of configuration bits used to configure all of the elements on the FPGA. The FPGA vendor typically supplies these tools along with FPGA architecture descriptions and related semiconductor process information.

First, the HDL is transformed into a netlist of simple boolean gates and flip-flops through high-level synthesis. The process of technology mapping transforms this simple netlist into a netlist of elements available on the FPGA, namely LUTs, flip-flops, and any specialized blocks available on the FPGA such as memories and DSPs. Clustering (or
Packing) then takes the technology-mapped netlist and groups the LUTs and flip-flops into LBs so that the resulting netlist consists of LBs and specialized blocks. The purpose of Placement is to assign each block in the netlist to a physical location on the FPGA, in other words, to map the logical netlist to the physical FPGA. Finally, the purpose of Routing is to find appropriate paths through the FPGA global interconnect to implement all of the connections between the netlist blocks. At this point determining the values of all SRAM configuration bits is straightforward and carried out by the CAD tools. Once the mapping solution has been determined, area, timing, and power models can be used to evaluate the expected performance of the implemented circuit.

Since this dissertation focuses on the placement and routing stages, the following subsections will describe the placement and routing algorithms in more detail.
2.2.1 Placement

The most popular heuristic for FPGA placement is Simulated Annealing (SA) [67] due to its ability to easily incorporate optimization objectives and legality constraints. Although other approaches have been proposed, such as partitioning-based [84], force-directed [78] and quadratic [127] analytical placers, SA is often cited as the most widely used algorithm for FPGA placement. A recent study compared the performance of SA based FPGA placement to ASIC analytical placers and concluded that “simulated annealing based [FPGA] placement would still be in dominant use for a few more device generations.” [20]

Algorithm 1 describes the generic SA algorithm. Let $S$ denote the current placement solution and $T$ denote the current temperature. SA starts with a random legal initial placement followed by many iterations of improving the placement through random swaps of placeable objects. Swaps that improve the objective cost function are always accepted. Swaps that degrade the cost are sometimes accepted to prevent the optimization from becoming trapped in a local extrema. The cost function can simultaneously incorporate multiple objectives (such as minimizing area, delay, routing congestion [19] and/or power [71]), and legality constraints (such as clock domain restrictions [72], or voltage islands [77]).

An annealing schedule is used to describe the various aspects of the anneal such as the ExitCriterion (when to terminate the anneal), InnerLoopCriterion (number of swaps to evaluate at each given temperature), $T_o$ the initial temperature, and how to update the temperature. The temperature (and the severity of cost degradation) directly controls the probability of accepting a bad move. Initially, $T$ is large so that most moves are accepted. As the temperature decreases, and the solution begins to converge to a good
Algorithm 1 Simulated annealing placement

1: $S \leftarrow \text{Initial Placement}$
2: $T \leftarrow T_0$
3: while $\text{ExitCriterion} == \text{False}$ do
4:   while $\text{InnerLoopCriterion} == \text{False}$ do
5:     $S_{\text{new}} \leftarrow \text{Perform Random Swap}$
6:     $\Delta C \leftarrow \text{Cost}(S_{\text{new}}) - \text{Cost}(S)$
7:     $r \leftarrow \text{Random}(0, 1)$
8:     if $r \leq e^{-\Delta C/T}$ then
9:       $S \leftarrow S_{\text{new}}$
10: end if
11: end while
12: $T \leftarrow \text{Update Temperature}$
13: end while

one, the probability of accepting bad moves decreases. Due to the enormous number of
swaps evaluated, SA can take a significant amount of time to converge.

2.2.2 Routing

Unlike typical ASIC routers that perform two-step routing (global routing that assigns
nets to channels followed by detailed routing that assigns specific tracks within a chan-
nel), modern FPGA routers perform both global and detailed routing concurrently. The
prefabricated nature of the FPGA routing fabric causes difficulties for two-step routing.
The most common approach for FPGA routing is based on the Pathfinder algorithm, a
negotiation-based router [90].

The key idea in Pathfinder is that during the routing process, it allows multiple nets to
use the same routing resource (which leads to an illegal routing solution). However, this
overuse is resolved through multiple routing iterations. When a resource is overused, its
associated cost is increased proportionally to the amount of overuse to reflect the amount
of demand. Over the course of multiple routing iterations, nets start to find alternative
Algorithm 2 PathFinder-based VPR router

1: repeat
2:   for all nets $i_{net}$ do
3:     $PriorityQueue = \text{source node of } i_{net}$
4:     for all sinks of $i_{net}$ do
5:       $m = \text{top}(PriorityQueue)$
6:       while $m \neq \text{sink}$ do
7:         $\text{expand_neighbours}(m)$
8:       $m = \text{top}(PriorityQueue)$
9:     end while
10:   $Route = \text{Traceback through nodes leading to sink}$
11:   Add all nodes in traceback to $PriorityQueue$
12: end for
13: end for
14: for all Routing Resources on the FPGA do
15:   Update cost of Routing Resource
16: end for
17: until There Are No Overused Resources

Algorithm 3 $\text{expand_neighbours}(m)$ function

1: for all fanout nodes $n$ of $m$ do
2:   Calculate cost of adding $n$ to route path
3:   Add $n$ to $PriorityQueue$ with cost
4: end for

paths and only the most critical nets can justify the cost of the high demand resources. Routing iterations repeat until there are no longer any overused resources, thus leading to a legal routing solution.

Algorithm 2 and 3 outlines the Pathfinder algorithm. Within each routing iteration, nets are routed one after another. The routing of a net begins from the source pin to the sink pin. For nets with multiple sinks, each sink is routed sequentially. Starting at the source node, neighbour expansion (summarized in Algorithm 3) is performed, meaning all routing resources (wire segments and pins) that can be reached from the source are scored based on a cost function, and inserted into a Priority Queue (PQ). Next, the
routing resource with the best score, \( m \), is removed from the PQ. If this resource is the target sink, the routing is complete for this sink. Otherwise, neighbour expansion is performed on \( m \) and the next resource with the best score is removed from the PQ. This is repeated until the sink is found. This process effectively creates an expansion wavefront that emits from the source and eventually reaches the target sink. Once all nets have been routed, the cost associated with each resource is increased based on whether it is overused. As long as overused resources exist (illegal solution), a new routing iteration will take place where all nets are re-routed.

### 2.3 Related Work on FPGA CAD Storage Requirement Scalability

To our knowledge, there are only two studies in the literature that have looked at memory reduction of FPGA CAD algorithms. The authors of [25] improve the memory scalability of the QuickRoute [79] algorithm used to route on FPGAs with pipelined routing architectures [100, 102]. For each net, a routing resource node cannot be explored more than once for a given path. Tracking this information lead to a storage requirement that scaled at a rate in \( O(n^2) \) where \( n \) is the number of routing resources. The authors employed an encoding algorithm to compress this information and the resultant storage requirement scaled at a rate in \( O(n \cdot \log(n)) \). This work does not directly apply to modern commercial FPGAs as they typically do not employ pipelined routing architectures.

The second related work proposes a Just-in-Time (JIT) compiler for FPGAs which configures an FPGA when it is about to execute [83]. Their routing algorithm uses a simplified routing resource graph where each node represents a switch box, and edges represent routing resources between the switch boxes. Although their technique produces a significant reduction in memory footprint, it only applies to a specific simpli-
fied architecture aimed for fast JIT compilation. Also, this technique still models every
switch box and connection explicitly, resulting in large graphs if scaled to large FPGA
devices. The technique proposed in Chapter 4 addresses both of these drawbacks.

2.4 Related Work on FPGA CAD Run-time Scalability

The development of fast FPGA CAD tools has been an active area of research. Most
work on fast FPGA CAD has focused on placement as it is often cited as the most time
consuming stage [81, 109], but efforts on fast routing [23, 27, 47, 99, 109, 119], and
fast technology-mapping [32, 33, 55] do exist. In this section we focus on reviewing the
studies related to placement. We loosely categorize the research on fast FPGA placement
as follows and review them in the following sub-sections.

- improving classical heuristics [16, 17, 19, 49, 80, 92, 118]
- alternative heuristics [2, 18, 51, 64]
- parallelization [1, 26, 39, 42, 58, 81, 98]
- incremental compilation [76, 103, 108, 113]
- floorplanning and hierarchy [24, 97, 104, 112, 130]
- and hardware acceleration [54, 60, 123]

2.4.1 Improving Classical Techniques

Betz introduced three new features in his SA-based VPR placer which dramatically im-
proves on SA run-times [19]: an adaptive annealing schedule so that less time is wasted
in iterations that do not enhance the placement quality, a linear routability-congestion
model which can be evaluated faster than previous non-linear routing-demand based
congestion models, and an incremental net bounding-box update scheme that computes the cost of a potential swap.

Some studies propose methods to quickly generate a good initial placement for SA. Space-filling Curves are proposed in [17] and [16] uses recursive bi-partitioning of the circuit. In [49], short term memory is introduced to reduce the amount of time spent on evaluating bad moves.

The authors of [118] use intelligent directed moves in addition to the random moves in traditional SA. The authors of [80] apply the Quantum Annealing (QA) optimization heuristic to the FPGA placement problem. QA is similar to SA except that it also allows for a “tunneling” effect along with the hill-climbing effect to move out of local optimization minimums. The QA implementation showed a 18.3% reduction in run-time and 7.2% improvement in critical-path delay over that of conventional VPR.

In [92], the authors compare the trade-offs between place-and-route run-time and quality for a number of placement and routing algorithms. Some of these algorithms were trade-off oriented in that they had parameters that could be tuned to vary the run-time versus quality trade-off. For example, compared to the commercial Xilinx CAD tools at the time, 3x run-time speedup could be achieved at the expense of 1.27x degradation in mapping quality, or a 1.6x improvement in quality could be achieved at the expense of 2x longer CAD run-times.

2.4.2 Alternative Heuristics

A number of non-SA heuristics have been proposed for the FPGA placement problem. The algorithm in [18] starts by constructively generating a good legal initial placement followed by a greedy iterative improvement phase. To construct the initial placement, the algorithm randomly chooses and places a seed block. A high fanout block connected
to the seed is then placed near the seed. This new block becomes the next seed and the process is repeated until all blocks have been placed. In the next phase, they use an Immediate Neighbour Local Search method which employs range windows to limit the deviation from the initial placement.

Placement and routing in commercial and academic FPGA CAD are usually performed as distinct steps. Since the routing solution is highly dependent on the placement solution, studies have suggested that a more tightly coupled flow will lead to higher quality mappings. In terms of run-time, giving the placement a better notion of routability will lead to less time being spent re-placing a design because it was unroutable. Another advantage is that the router will encounter fewer hard-to-route nets (time consuming). Since a full detailed route after each placement iteration is too time-consuming, an analytical routability model which can be evaluated very quickly can be used [28]. The authors of [64] argue that these models are not accurate, and present a heuristic that generates a set of possible routes for each net using a fast minimum steiner-tree formulation.

Some studies have proposed to use alternative meta-heuristics such as Tabu-Search [51]. However, it can be argued that if properly formulated and tuned, all meta-heuristics can perform with comparable quality and run-time scalability.

2.4.3 Parallelization

With the growth in per-core performance waning, CPUs are moving deeper into the multi-core processor paradigm where dual and quad core processors are already the norm. Consequently, parallel software is becoming a necessity for increased performance. Several parallel FPGA placement algorithms have been proposed in the literature [21, 26, 58, 81, 120]. In addition, both of the major FPGA vendors, Altera [81] and Xilinx [98], have very recently included basic multi-core support in their respective
CAD tools. Parallelization will no doubt play a significant role in combating long CAD run-times. We review some of the more recent efforts here.

Altera has published their efforts at parallelizing the SA placer in their Quartus II CAD suite [81]. Ideally, run-time speed-ups would increase linearly with the number of processors. However, due to cross-core communication requirements, this can be difficult to achieve. The Altera paper showed a 1.7x and 2.2x speedup for two and four cores respectively. Most notably, their method emphasized several requirements that make their speedups more conservative compared to earlier publications: the algorithm must run on commodity hardware, must be deterministic (produce the same solution across multiple executions), and must have serial equivalency (produce the same solution regardless of the number of processing cores used). The authors suggest that with improved memory subsystems on upcoming multicore processors, their techniques will scale to 16 processors.

Birk et al. [21] describes a parallel implementation of VPR’s SA placer using transactional memory\(^2\). They show a near linear speedup compared to their single thread implementation, but an overall slowdown compared to the original VPR. In addition, their implementation is currently non-deterministic and does not have serial equivalence. The paper does discuss possible methods to overcome these shortcomings.

Wang et al. [120] parallelized VPR’s SA placer and the associated timing analysis. Their algorithm is deterministic and yields significant speedup for up to 25 threads. Although there is an 8% and 11% degradation in wirelength and critical-path delay, these degradations are generally independant of the number of cores.

The authors of [39] target an NVIDIA GTX280 Graphics Processing Unit (GPU)\(^2\)Transactional Memory is a framework aimed at transparantly support regions of code that require atomicity and consistency, thus simplifying some of the challenges in writing parallel software [48].
with 240 cores divided into 30 groups (streaming multiprocessors). The authors highlight the challenge of dealing with the GPU’s Single Instruction Multiple Data (SIMD) architecture and constrained memory architecture (where memory access is optimized for contiguous memory regions). Their approach achieved 10x speedup with 3% wire-length degradation, but did not address determinism or scalability.

### 2.4.4 Incremental Compile

In a traditional FPGA design flow, even a small change to the design (at the HDL level) would require a full CAD iteration to remap it to the FPGA. However, in many cases, the change is relatively small such that most of the FPGA’s configuration does not need to be changed. The idea of incremental compile is to only re-configure the parts of the design that have changed. The speedup compared to a full iteration through the CAD flow can be dramatic. Two limitations are that a full mapping must be performed in the beginning, and too many incremental recompiles tend to lead to degraded overall quality, resulting in a need to fully recompile the design.

Incremental FPGA synthesis [33, 108], placement [76, 103, 113], and routing [113] techniques have been described in the academic literature and are supported by commercial CAD tools [12].

### 2.4.5 Floorplanning and Hierarchy

Floorplanning-based speedup techniques leverage the fact that large-scale designs are often developed using pre-defined macroblocks or using hierarchical hardware design languages. This hierarchy in the circuit provides locality information, which can then be exploited to guide the CAD tools. However, this is more difficult when targeting FPGAs than when targeting ASICs due to the prefabricated resource constraints. The system
in [24] is restricted to data-path circuits. The Frontier tool in [112] uses information in pre-placed macroblock libraries but also allows blocks to swap freely in the overall placement. Floorplanning for FPGAs containing heterogeneous resources are considered in [104] and [130]. Some argue that the logical locality may not translate well to physical locality [31], and would thus lead to degraded mapping quality.

From another perspective, floorplanning is a form of hierarchical placement. In [97] the authors propose Ultra-Fast VPR, a fast hierarchical FPGA placer that recursively coarsens the problem to obtain a simpler, and thus faster to solve, placement problem. The coarser (simpler) problems are solved and used as a guide to optimize the next finer level. This type of multilevel approach is taken by most modern ASIC placement tools [29, 61, 66, 117].

2.4.6 Hardware Acceleration

The authors of [123] propose to associate a Processing Element (PE) for each logic block site on the FPGA for which placement is to be performed. Each PE then negotiates with its neighbours to determine whether a local swap should occur. This algorithm was implemented on a Xilinx Virtex2 FPGA and they found speed-ups of several orders of magnitude. The scalability of this technique is limited by the capacity of the FPGA on which it is implemented. Although the technique is very interesting, FPGA devices are not large enough to implement enough PEs to target very large FPGA placement.

In [54], the authors implement an analytical placer which uses a graphic processing unit (GPU) as a co-processor to perform most of the matrix computations, resulting in a speedup of 3x.
2.5 Related Work on Affecting FPGA CAD Run-time through Architecture

This section reviews studies, of which there are very few, on creating architectures for fast CAD. The Plasma architecture [15] was designed with fast mapping in mind. The Plasma FPGA was custom-built for the Teramac project [14], an extremely large-scale configurable hardware system that contains 1728 Plasma FPGAs. Fast mapping was achieved by using an extremely dense routing fabric employing crossbars and long wire segments. By employing an extremely flexible and resource-rich routing fabric, the placement and routing algorithms do not need to spend as much time to find a legal solution. Although this routing fabric has a large area overhead, the entire Teramac system could be configured in under two hours.

A chapter in [50] briefly discusses the effects of FPGA architecture on CAD runtime. This chapter focused on routing with discussion centered on increasing the richness of the routing fabric. This reduces the difficulty of the routing problem, which in turn reduces the number of routing iterations needed to find a legal routing solution. This is similar to the Plasma approach.

The SFRA architecture [121] employs a pipelined corner-turn interconnect that allows for very fast routing run-times. The SB topology causes turning connections (connections between horizontal and vertical wire segments) to be a limited resource while having abundantly available straight-through connections. The CBs are fully-populated crossbars. The SFRA CAD routing stage uses two-step routing. First, a custom global routing algorithm aimed at minimizing turns is employed; the abundance of straight-through connections make parts of this step easy and fast. For detailed routing, a fast greedy channel packing algorithm is possible because of the fully-populated CBs. Com-
pared to the Xilinx CAD flow targeting a Xilinx Virtex E device, the SFRA achieved two orders of magnitude reduction in routing time at the expense of a 3.9x area increase.

In [23], a parallel routing algorithm is proposed that only works when a specific FPGA switch box topology is used: the Disjoint switch box [74]. The Disjoint topology partitions the routing fabric into track domains. Routing resources in different track domains never connect to each other. For this reason, the routing problem can be parallelized once each net has been assigned to a track domain. The study showed near-linear speedup on up to eight cores, at the expense of an increase in the number of required tracks. This increase is due to the previously known decreased flexibility of the Disjoint switch block [89]. Although not directly stated in the original paper, this approach can be considered as designing architectures that are amenable to parallellizable CAD tools, an avenue of future research that will be discussed in the concluding chapter of this dissertation.

All of the studies reviewed in this sub-section target the routing stage. In Chapter 5, we investigate an architectural change that affects both placement and routing run-time.

2.6 Related Work on Analytical FPGA Models

Recently, several publications have presented analytical equations that relate FPGA architectural parameters to various important metrics. The relationship between logic block architecture and area efficiency was described in [56] and [69]; The model in [69] also takes into account cluster-based FPGAs and cluster architecture. The relationship between logic block architecture and post-techmapping and post-clustering depth was described in [44]. In [107], a wirelength model for FPGAs is presented that describes the average post-placement wirelength. Both homogeneous and heterogeneous FPGAs were considered in this model. The model in [52] relates the minimum required channel
width to parameters describing the routing architecture.

These models can be used in conjunction with area, delay and power models for fast early-stage architectural investigation. For example, [105] and [106] showed how the above models could be used with numerical optimization techniques to help determine appropriate architectural and transistor-level parameters. For all of the above models, the purpose is to provide a means to quickly narrow down the architecture space before moving onto more accurate but more time-consuming experimental tuning.

Other related studies include analytical methods to predict the routability of a circuit [22, 43, 65] and the behaviour of Rent’s Exponent during placement [94].

### 2.7 Summary

This chapter first reviewed the main concepts and nomenclature on FPGA architecture and CAD. We also reviewed the emerging body of theory on analytical FPGA models. In the next chapter (Chapter 3), we build upon this theory to develop a model that relates key FPGA architecture parameters to the FPGA CAD run-time. This chapter also reviewed the two pieces of work related to FPGA CAD storage requirements. Chapter 4 will present a significant contribution towards improving the scalability of FPGA CAD storage requirements. Finally, the rest of this chapter discussed the many pronged efforts towards tackling FPGA CAD run-time scalability. In Chapter 5, we will investigate further the designing FPGA architectures that are amenable to fast CAD.
Chapter 3

Analytical Models Relating Architecture and CAD Run-time

This chapter describes an analytical model that relates key FPGA architectural parameters to the FPGA CAD run-time. More specifically, we focus on the placement and routing stages of the CAD flow because they are typically the most time consuming stages [81, 110]. A generic model that encompasses all possible algorithms used to perform placement and routing would not be very accurate. Hence, we focus on two well accepted algorithms: Simulated Annealing based placement, and negotiation-based routing (these algorithms were reviewed in Chapter 2). This allows us to develop practical models while still incorporating many of the important ideas used in modern FPGA placement and routing tools. This also allows us to validate our model against a real CAD tool, namely the academic Versatile Place-and-Route (VPR) [82] CAD suite. These derivations can then serve as a framework for deriving models for other types of placement and routing algorithms.
The remainder of this chapter is organized as follows. Section 3.1 presents our framework and assumptions. The placement and routing run-time models are presented in Sections 3.2 and 3.3 respectively. Section 3.4 validates the models. Section 3.5 uses two examples to demonstrate how this model can be used by an FPGA architect and Section 3.6 summarizes the chapter. This work was first published in [38].

3.1 Framework

We first describe the architectural, circuit, and algorithmic assumptions made during the derivation of our model. In addition, our model makes use of a separate FPGA logic density model [69] which will also be described further here.

3.1.1 Architectural and Circuit Assumptions

We assume an island-style FPGA architecture as reviewed in Chapter 2. Furthermore, we make the following architectural assumptions:

- The FPGA is homogeneous (containing only LBs). Generalizing for heterogeneous FPGAs that contain embedded blocks such as memories and DSP blocks is not considered in this dissertation.
- All wire segments are of the same length $L$.
- The number of outgoing wires to which each incoming wire can connect is $F_s = 3$. This has been shown to give good area and speed results [19].
- The Wilton switch-block topology [122] is used.

Our primary goal is to relate architecture and CAD run-time. However, we cannot completely ignore the properties of the circuit being implemented. We describe the
Table 3.1: Model inputs

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Architecture:</strong></td>
<td></td>
</tr>
<tr>
<td>$K$</td>
<td>Number of inputs per look up table</td>
</tr>
<tr>
<td>$N$</td>
<td>Number of lookup tables per logic block</td>
</tr>
<tr>
<td>$I$</td>
<td>Number of inputs per logic block</td>
</tr>
<tr>
<td>$N_x, N_y$</td>
<td>FPGA array dimensions</td>
</tr>
<tr>
<td>$W$</td>
<td>Routing channel width</td>
</tr>
<tr>
<td>$L$</td>
<td>Wire segment length</td>
</tr>
<tr>
<td>$F_{Cout}$</td>
<td>LB output pin connection box flexibility (Fractional)</td>
</tr>
<tr>
<td>$F_{Cin}$</td>
<td>LB input pin connection box flexibility (Fractional)</td>
</tr>
<tr>
<td><strong>Circuit:</strong></td>
<td></td>
</tr>
<tr>
<td>$p$</td>
<td>Rent’s exponent of a given circuit</td>
</tr>
<tr>
<td>$n_2$</td>
<td>Number of 2-input boolean gates in a given circuit</td>
</tr>
<tr>
<td>$n_{io}$</td>
<td>Number of primary I/Os in a given circuit</td>
</tr>
<tr>
<td><strong>Logic Utilization (Estimated using Lam/Wilton Model [69]):</strong></td>
<td></td>
</tr>
<tr>
<td>$n_c$</td>
<td>Number of used logic blocks</td>
</tr>
<tr>
<td>$i$</td>
<td>Number of used inputs per logic block</td>
</tr>
<tr>
<td>$o$</td>
<td>Number of used outputs per logic block</td>
</tr>
<tr>
<td>$f_{avg}$</td>
<td>Average circuit fanout</td>
</tr>
</tbody>
</table>

circuit using three parameters: the Rent’s Exponent $p$ (this describes the routing demand of the circuit), number of 2-input boolean gates $n_2$, and number of primary I/Os $n_{io}$ (these two parameters describe the size of the circuit).

The architectural and circuit parameters that serve as inputs to our model are summarized in Table 3.1.

3.1.2 CAD Assumptions

As stated before, we focus on accurately modeling two well accepted algorithms that can serve as a framework for other derivations. For placement, we consider a simulated annealing based placement algorithm employing a Half Perimeter Bounding Box Wire-length (HPWL) cost function. We also assume the adaptive annealing schedule found in VPR. The pseudo-code for this algorithm was shown in Algorithm 1 (Chapter 2).

For routing, we assume a negotiation-based router such as the one employed by VPR. The pseudo-code for this algorithm was shown in Algorithm 2 (Chapter 2). Any further algorithm details will be reviewed when necessary during the model derivation.
3.1.3 Logic Utilization Model Usage

Since the placement CAD stage places a technology-mapped and clustered netlist, our placement run-time model requires some information regarding the technology mapping and clustering efficiency. More specifically, our model requires as input the number of LBs \( n_c \), the average number of used LB input pins per LB \( i \), the average number of used LB output pins per LB \( o \), and the average fanout of each net \( f_{avg} \). These properties can be estimated using the Lam/Wilton FPGA logic density model [69], and the architectural and circuit parameters from Table 3.1. These terms are summarized at the bottom of Table 3.1 and will be used in parts of our model.

3.2 Model Relating Placement Run-time to Architecture

We first consider the run-time required by placement. Recall that SA is based on performing a large number of random swaps, and proposed swaps are accepted or rejected based on a cost function. Essentially, the total placement run-time depends on the number of attempted swaps, and the time it takes to evaluate an attempted swap. The following equation describes this relationship.

\[
Time_{place} = C_{swap} \cdot \sum_{T=T_i}^{T_f} N_{swaps}(T) \tag{3.1}
\]

where \( C_{swap} \) is the average time required to evaluate a swap, \( N_{swaps}(T) \) is the number of swaps attempted at temperature \( T \), and \( T_i \) and \( T_f \) are the initial and final temperatures. The following section will show that \( N_{swaps}(T) \) is constant and independent of \( T \) in the VPR annealing schedule. Therefore, we simply need to determine the number of temperature steps \( N_{steps} \) and Equation[3.1] can be simplified as follows.
\[ Time_{place} = C_{swap} \cdot N_{steps} \cdot N_{swaps} \]  

Relating back to the pseudo-code in Algorithm 1 (page 19), \( N_{steps} \) models the number of iterations for the outer loop on Line 3, \( N_{swaps} \) models the number of iterations for the inner loop on Line 4, and \( C_{swap} \) models the average amount of time needed to execute Lines 5-10. These three terms will be modelled in the next three subsections.

### 3.2.1 Average Time Per Swap \( C_{swap} \)

In general, the time required to evaluate a proposed swap is the time needed to calculate the change in cost, \( \Delta C \), due to the swap. For the HPWL cost function, this involves a re-calculation of the HPWL for each net that would be perturbed. We can then express \( C_{swap} \) as a function of the average number of nets perturbed in a swap \( net_{aff} \), and the average time needed to re-calculate the HPWL of a single net \( C_{net} \); this is shown in Equation 3.3. We assume that \( C_{net} \) is a constant for a given software implementation of the placement algorithm, compiled and running on a given machine.

\[ C_{swap} = C_{net} \cdot net_{aff} \]  

The average number of nets perturbed depends on the average number of LBS involved in a swap, and the average number of nets connected to an LB. The number of blocks involved in a swap can be one (when a block is moved to an unoccupied LB location), or two (when two blocks swap locations). On average, the number of blocks involved will be 1 plus the probability of choosing an occupied LB location. Since there are \( N_x \cdot N_y \) (this is the FPGA array size) LB sites from which to choose, and \( n_c \) occupied LBs, the probability of uniformly choosing an occupied LB is \( \frac{n_c}{N_x \cdot N_y} \). The average num-
ber of nets connected to an LB is the sum of the average number of used LB input pins \(i\), and the average number of used LB output pins \(o\). The equation for \(nets_{aff}\) is described in Equation 3.4.

\[
nets_{aff} = \left( 1 + \frac{n_c}{N_x \cdot N_y} \right) \cdot (i + o)
\]  

(3.4)

### 3.2.2 Swaps Attempted Per Temperature Step \(N_{swaps}\)

Equation 3.5 describes the number of swaps attempted at each temperature step according to VPR’s annealing schedule [19] where \(InnerNum\) is an annealing schedule parameter (default value of 10), and \(N_{blocks}\) is the total number of blocks to be placed.

\[
N_{swaps} = InnerNum \cdot N_{blocks}^{4/3}
\]

(3.5)

\(N_{blocks}\) can be expressed by Equation 3.6 where \(n_c\) is the number of logic blocks in the post technology-mapped netlist, and \(n_{io}\) is the number of primary I/Os in the circuit. \(n_c\) and \(n_{io}\) are inputs to the model.

\[
N_{blocks} = n_c + n_{io}
\]

(3.6)

### 3.2.3 Number of Temperature Steps \(N_{steps}\)

To model \(N_{steps}\) we need to know the initial \(T_o\) and final temperature \(T_f\), and the temperature update scheme. These terms are modeled in the following sub-sections.

#### 3.2.3.1 Temperature Update Scheme

The adaptive temperature-update scheme in Equation 3.7 encourages the anneal to spend more time near productive temperatures [19]. Productivity is measured using the swap
acceptance ratio at each temperature \( \alpha = \frac{\text{Swaps accepted}}{\text{Swaps proposed}} \).

\[
T_{\text{new}} = \gamma \cdot T_{\text{old}} \quad \gamma = \begin{cases} 
0.5, & \alpha > 0.95 \\
0.9, & 0.8 < \alpha \leq 0.95 \\
0.95, & 0.15 < \alpha \leq 0.8 \\
0.8, & \alpha < 0.15 
\end{cases} \tag{3.7}
\]

Starting at the initial temperature \( T_o \), we can calculate each intermediate temperature until the final temperature \( T_f \) as shown in Equations 3.8 to 3.10. Note that there are \( N_{\text{step}} \) temperatures (including \( T_o \) and \( T_f \)) by definition.

\[
T_1 = \gamma_0 \cdot T_o \\
T_2 = \gamma_1 \cdot T_1 \\
\vdots \\
T_f = \gamma_{N_{\text{steps}}-1} \cdot T_{N_{\text{steps}}-1} \tag{3.10}
\]

By recursively substituting Equations 3.8 to 3.10 we come to Equation 3.11:

\[
T_f = \left( \prod_{i=0}^{N_{\text{steps}}-2} \gamma_i \right) \cdot T_o \tag{3.11}
\]

We found that VPR’s annealing schedule caused the anneal to spend most of its time in the range where \( 0.15 < \alpha \leq 0.8 \) (corresponding to \( \gamma = 0.95 \)). Based on this observation, our model approximates \( \gamma \) with a constant value of 0.95 (the sensitivity of this approximation will be discussed in Section 3.4.3). In other words, \( \gamma_0 = \gamma_1 = \ldots = \gamma_{N_{\text{steps}}-1} = 0.95 \). This allows us to rewrite Equation 3.11 as Equation 3.12.
Finally, we rearrange Equation 3.12 to produce an expression for \( N_{\text{steps}} \) as shown in Equation 3.13. Expressions for \( T_f \) and \( T_o \) are derived in the following sub-sections.

\[
N_{\text{steps}} = \log_{0.95} \frac{T_f}{T_o} + 1
\]  

(3.13)

### 3.2.3.2 Final Temperature \( T_f \)

In VPR, the anneal terminates when the following condition is met: \( T < \varepsilon \cdot \frac{\text{Cost}}{n_{\text{nets}}} \), where \( \varepsilon = 0.005 \) and \( n_{\text{nets}} \) is the number of nets in the netlist being placed. We will use the upper bound as the final temperature as expressed in Equation 3.14. At this temperature, \( \text{Cost} \) is the final placement cost which we denote as \( \text{Cost}_f \).

\[
T_f = \varepsilon \cdot \frac{\text{Cost}_f}{n_{\text{nets}}}
\]  

(3.14)

The \( n_{\text{nets}} \) term will cancel out in the final set of equations, so we do not need to consider how to express this term. Next we model the final placement cost. As mentioned earlier, we assume the HPWL cost function of VPR shown in Equation 3.15. This cost is basically the sum of each individual net’s HPWL.

\[
\text{Cost} = \sum_{j \in \text{nets}} \left( q(f_j) \cdot \frac{\text{HPWL}(j)}{W} \right)
\]  

(3.15)

where \( \text{HPWL}(j) \) and \( f_j \) are the HPWL and fanout of net \( j \), respectively. \( W \) is the FPGA channel-width but will be eliminated later in the derivation, and thus it does not affect
the placement run-time. $q()$ is a HPWL wirelength compensation factor for nets with more than three terminals \[^1\] [34]. This term is looked up from a table of pre-computed known constants, so no further effort is needed to express this term analytically.

If we know the average HPWL and fanout for a net, we can eliminate the summation in Equation 3.15 and write Equation 3.16:

$$\text{Cost} = n_{nets} \cdot q(f_{avg}) \cdot \frac{HPWL_{avg}}{W}$$  \hspace{1cm} (3.16)

By dividing both sides of Equation 3.16 by $n_{nets}$, as shown in Equation 3.17, we can express the average cost per net at any given point of the anneal if we also know the corresponding average HPWL at that point.

$$\frac{\text{Cost}}{n_{nets}} = q(f_{avg}) \cdot \frac{HPWL_{avg}}{W}$$  \hspace{1cm} (3.17)

The point of interest in this derivation is the end of the anneal. We are interested in the final cost per net (as needed by Equation 3.14). We can determine this if we can determine the final HPWL. This is described in Equation 3.18:

$$\frac{\text{Cost}_f}{n_{nets}} = q(f_{avg}) \cdot \frac{HPWL_{avg,f}}{W}$$  \hspace{1cm} (3.18)

where $HPWL_{avg,f}$ is the average post-placement HPWL of a net in the circuit. This term can be estimated using the Smith/Das/Wilton FPGA wirelength model [107]. Note that the Smith/Das/Wilton model expresses wirelength using a Steiner tree construction. We therefore need to relate the Steiner Tree Wirelength (STWL) to the HPWL. We do this

\[^1\]Half Perimeter Bounding Box Wirelength typically underestimates the wiring needed for nets with more than three terminals.
by empirically measuring the HPWL to STWL ratio $R_{\text{hpwl-stwl}}$ in Equation 3.19 where $WL_{sdw}$ is the wirelength predicted by the Smith/Das/Wilton model.

$$HPWL_{avg,f} = R_{\text{hpwl-stwl}} \cdot WL_{sdw}$$  \hspace{1cm} (3.19)

We found that the value for $R_{\text{hpwl-stwl}}$ was fairly uniform across a collection of twenty MCNC circuits using five different placement randomization seeds $^{2}$; it was measured to be 0.54. This is consistent with previous empirical studies. Hanan $^{[59]}$ reported that the HPWL is the same as the STWL for nets with two or three terminals. For nets with four or five terminals, the HPWL is between 0.5 and 1 times the minimum Steiner tree. We now have all the expressions needed to model the final temperature $T_f$.

### 3.2.3.3 Initial Temperature $T_o$

The initial temperature in VPR is determined as follows:

- Generate a random placement of the $n_c + n_{io}$ blocks
- Generate $n_c + n_{io}$ alternative random placements by performing and accepting $n_c + n_{io}$ swaps.
- Calculate the standard deviation $\sigma$ in cost across this set of random placements.
- Set the initial temperature as $T_o = 20\sigma$

This sets the initial temperature high enough such that all swaps are accepted with high probability at the beginning of the anneal. Using this observation, we begin our

$^{2}$Since SA is based on random swaps, its implementation uses Random Number Generators (RNGs). By re-running the same experiments and supplying different seed values to the RNG, and then averaging the results, we can reduce the impact of noise introduced by the RNG.
derivation of an expression for the initial temperature $T_o$ by relating it to the probability of a swap being accepted which we denote as $P_{accept}$.

Recall from Line 8 in Algorithm 1 (page 19) that good swaps ($\Delta C < 0$) are always accepted and bad swaps ($\Delta C > 0$) are accepted based on the random event $r \leq e^{-\Delta C / T}$ where $r$ is a uniformly distributed random variable in [0,1]. The probability of accepting a swap, in general, can therefore be expressed by the following equation.\(^3\)

$$P_{accept} = \text{Prob}(\Delta C \leq 0) + \text{Prob}(\Delta C > 0) \cdot e^{-\Delta C / T}$$

Equation 3.20 is then applied to the conditions at the beginning of the anneal. Immediately after a random initial placement there is, on average, an equal probability of generating a good and bad swap $\text{Prob}(\Delta C \leq 0) = \text{Prob}(\Delta C > 0) = 0.5$. This was experimentally verified. $T$ is replaced with the initial temperature $T_o$. Making these substitutions into Equation 3.20 and solving for $T_o$ gives Equation 3.21.

$$T_o = \frac{-\Delta C_{bad}}{\ln(2 \cdot P_{accept} - 1)}$$

(3.21)

Recall that the initial temperature is set high enough such that there is a high probability of acceptance for bad swaps. Using Equation 3.21, we determine an appropriate temperature such that a bad swap (a swap that has a change in cost of $\Delta C_{bad}$) will have a high probability of being accepted $P_{accept,j}$. This temperature will be the approximation for the initial temperature. We define a high acceptance probability as being in the range of $P_{accept,j} = [0.96, 0.99]$. This was experimentally verified.

\(^3\)Prob $\left( r < e^{-\Delta C / T} \right) = e^{-\Delta C / T}$ since $r$ is uniformly distributed.
Figure 3.1: Two placements of a net with the same half perimeter bounding box.

$\Delta C_{\text{bad}}$ is modeled with three terms: the number of nets affected by an average swap $n_{\text{nets,aff}}$ (Equation 3.4), the fraction of these nets that will see an increase in their bounding box due to the swap $P_{\text{increase}}$, and the change in cost when one net’s average bounding box increases to the maximum $\Delta C_{\text{one net}}$ (this is a worst-case change in cost):

$$\Delta C_{\text{bad}} = n_{\text{nets,aff}} \cdot P_{\text{increase}} \cdot \Delta C_{\text{one net}}$$

We develop a simplified approximation for $P_{\text{increase}}$. Given the HPWL of an average net after the random initial placement, $HPWL_i$, the number of LB sites within the bounding box can range from $HPWL_i$ in the best case when all LBs connected to the net are in the same row (or column), to $(HPWL_i/2)^2$ in the worst case when the bounding box is a square. This is illustrated in Figure 3.1. To be conservative, we use the worst case scenario. Thus, the number of LB sites outside of the bounding box is $N_x \cdot N_y - (HPWL_i/2)^2$. If the block moves to one of these sites, then its bounding box will grow. Since there are $N_x \cdot N_y$ sites to uniformly choose from, the probability of increasing the bounding box can be described by:

$$P_{\text{increase}} = \frac{N_x \cdot N_y - (HPWL_i/2)^2}{N_x \cdot N_y} = 1 - \frac{(HPWL_i/2)^2}{N_x \cdot N_y}$$
To estimate the average HPWL after an initial random placement $HPWL_i$, we again use the Smith/Das/Wilton wirelength model [107]. However, instead of using the post-placement Rent parameter, we use a Rent parameter value that reflects an initial random placement, $p_i$. Previous studies [94] showed that the Rent parameter of the placement starts off quite high and decreases in a highly correlated way with the placement cost. Although we do not have an expression for the random-placement Rent parameter we found that setting this value in the range of 0.9 to 0.99 worked well for our run-time model predictions. This range is consistent with the findings in [94].

For a typical net (one that has the average bounding box wirelength of $HPWL_i$), the worst case change in cost occurs when its new bounding box goes from the average HPWL to span the entire FPGA. A bounding box that spans the entire FPGA is $(N_x + N_y)$. Hence, the change in cost of one net undergoing such a bounding box increase can be expressed by (recall the cost function of Equation 3.15):

$$\Delta C_{\text{one net}} = q(f_{\text{avg}}) \cdot \frac{(N_x + N_y) - HPWL_i}{W}$$

\[ (3.24) \]

### 3.2.4 Placement Run-time Model Summary

The model equations that relate placement run-time to the architecture are summarized in Table 3.2. Validation of this model is described in Section 3.4 and Section 3.5 presents some example applications of this model.

### 3.3 Modeling Routing Run-time

This section describes the routing run-time model. We first define two terms. A Routing Resource (RR) is expanded when it is scored and placed into the priority queue.
Table 3.2: Placement run-time model summary

<table>
<thead>
<tr>
<th>Description</th>
<th>Equ. #</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Placement Time</td>
<td>3.2</td>
<td>$T_{\text{place}} = C_{\text{swap}} \cdot N_{\text{steps}} \cdot N_{\text{swaps}}$</td>
</tr>
<tr>
<td>Time per swap</td>
<td>3.5</td>
<td>$C_{\text{swap}} = C_{\text{net}} \cdot n_{\text{nets}}$</td>
</tr>
<tr>
<td>Nets affected per swap</td>
<td>3.4</td>
<td>$n_{\text{nets}} = (1 + \frac{n_{\text{nets}}}{N_{\text{x}}N_{\text{y}}}) \cdot (i + o)$</td>
</tr>
<tr>
<td>Swaps per Temperature</td>
<td>3.5, 3.6</td>
<td>$N_{\text{swaps}} = \text{InnerNum} \cdot n_{\text{c}} + n_{\text{id}}^{4/3}$</td>
</tr>
<tr>
<td>Temperature Steps</td>
<td>3.13</td>
<td>$N_{\text{steps}} = \log_{0.95} \frac{T_{\text{f}}}{T_{\text{o}}} + 1$</td>
</tr>
<tr>
<td>Final Temperature</td>
<td>3.14, 3.18, 3.19</td>
<td>$T_{\text{f}} = \epsilon \cdot q(f_{\text{avg}}) \cdot \frac{R_{\text{hpwl}} - W_{\text{stwl}} \cdot W_{\text{sdw}}}{W}$</td>
</tr>
<tr>
<td>Initial Temperature</td>
<td>3.21, 3.22</td>
<td>$T_{\text{o}} = -\frac{\text{nets}<em>{\text{aff}} \cdot P</em>{\text{increase}} \cdot \Delta C_{\text{one net}}}{\ln(2 \cdot P_{\text{accept}} - 1)}$</td>
</tr>
<tr>
<td>(part of $T_{\text{o}}$ Equation)</td>
<td>3.23</td>
<td>$P_{\text{increase}} = \frac{N_{\text{x}}N_{\text{y}} - (N_{\text{hpwl}}/2)^{2}}{N_{\text{x}}N_{\text{y}}}$</td>
</tr>
<tr>
<td>(part of $T_{\text{o}}$ Equation)</td>
<td>3.24</td>
<td>$\Delta C_{\text{one net}} = q(f_{\text{avg}}) \cdot \frac{(N_{\text{x}}+N_{\text{y}}) - N_{\text{hpwl}}}{W}$</td>
</tr>
</tbody>
</table>

This corresponds to an iteration of the inner-most loop of Algorithm 2 (Lines 15-16 in Chapter 2). A RR is explored when it is the best choice resource and removed from the priority queue for review. This corresponds to Line 6 in Algorithm 2. In other words, the router first explores a RR. Then, if it is the target sink, a route for the net has been found. Otherwise, expand all neighbours and repeat. The run-time of the routing algorithm arises from performing many expansions when searching for each sink of each net over multiple routing iterations. This relationship is summarized as follows:

$$T_{\text{routing}} = C_{\text{expansion}} \sum_{i_{\text{iter}}=1}^{N_{\text{exp}}} \sum_{i_{\text{net}}=1}^{n_{\text{net}}} N_{\text{exp}}(i_{\text{net}}, i_{\text{iter}})$$ (3.25)

where $C_{\text{expansion}}$ is the time needed to perform an expansion (constant for a given compilation of the tool running on a given machine), $N_{\text{iter}}$ is the number of routing iterations needed and $N_{\text{exp}}(i_{\text{net}}, i_{\text{iter}})$ is the number of expansions performed in iteration $i_{\text{iter}}$ to route net $i_{\text{net}}$. As described in the following subsections, we model the number of expansions as an average. Therefore, Equation 3.25 is simplified to the following:
\[ T_{\text{routing}} = C_{\text{expansion}} \cdot N_{\text{iter}} \cdot n_{\text{nets}} \cdot N_{\text{exp}} \] (3.26)

### 3.3.1 Number of Routing Iterations \( N_{\text{iter}} \)

Deriving an analytic expression for the number of routing iterations is difficult due to the congestion-negotiation aspect of the routing algorithm. Every single RR (every single wire segment and pin on the FPGA) has a historical congestion cost which may change from iteration to iteration, and a present congestion cost which may change between the routing of each net [19]. This suggests that a detailed model would likely need to treat both congestion terms of each RR as random variables. However, the number of RRs in an FPGA is tremendous (this is the main reason for the large memory requirements as focused on in the next Chapter). To reduce the complexity of our models, we take a semi-analytical approach.

Intuitively, the number of required iterations depends on how “hard” it is to route the circuit. This difficulty can be quantified by comparing the minimum-required channel width to route the circuit, \( W_{\text{req}} \) (a measure of the circuit’s interconnect demand), to the channel width of the FPGA, \( W \) (a measure of the FPGA’s interconnect supply). \( W_{\text{req}} \) captures the impact of the routing architecture’s flexibility as specified by key routing architecture parameters (i.e., \( F_{\text{Cin}}, F_{\text{Cout}}, F_s, L \)), and can be estimated with the Fang/Rose channel width model [52].

We experimentally found that as the routing difficulty is relaxed (increasing \( W \), or decreasing \( W_{\text{req}} \) through other parameters such as \( L, F_{\text{Cin}}, F_{\text{Cout}}, \) etc.), the number of routing iterations decreased linearly until there were 20% more tracks than required. After this point, the number of iterations remained relatively constant on average. This point is often used as the threshold between high-stress and low-stress routing scenarios.
In our model, we will assume the more common low-stress scenario where there is 20% more routing tracks available. This allows the model to use a constant value for $N_{iter}$. This assumption is valid for many real world scenarios where commercial FPGAs are used, and it still allows us to use the model for its original purpose of investigating how the architecture impacts CAD run-time. Enhancing the model to more accurately model this term for high-stress routing could be an area of future research.

### 3.3.2 Average Number of Expansions $N_{exp}$

We use Equation 3.27 to model the average number of expansions that occur for a single net in a single routing iteration. $E_{LBout}$ describes the number of expansions performed to leave the source LB of the net being routed. $WE$ describes the average number of wire segments explored to route a net in a routing iteration. $E_{wire}$ describes the average number of expansions from a wire segment. These three terms are described in more detail in the following sub-sections.

$$N_{exp} = E_{LBout} + WE \cdot E_{wire}$$  \hspace{1cm} (3.27)

### 3.3.3 Expansions to Leave a Logic Block $E_{LBout}$

Assuming that all LB output pins are logically equivalent, $E_{LBout}$ arises from expanding $N$ LB output pins, and all $W \cdot F_{Cout}$ wire segments that can be reached from each of the $N$ LB output pins. Note that during routing, all $N$ output pins will be expanded even if there are less than $N$ output pins used by the LB. This is summarized in Equation 3.28.

$$E_{LBout} = N + N \cdot W \cdot F_{Cout}$$  \hspace{1cm} (3.28)
For single-driver routing architectures, an LB output pin can only drive a wire segment through an SB MUX, and SB MUXes are located only at a wire segment’s start tile. This can place a restriction on $F_{cout}$ as follows. For wire segments that span one tile $L = 1$ (as shown in Figure 2.3b), there is no restriction because each wire segment starts and ends at each tile. Figure 3.2 shows $L = 3$ wire segments. The LB output pin can only connect to wire segments in two of the six tracks, leading to a maximum $F_{Cout}$ of $2/6 = 0.33$. In general, there is a relationship between the maximum possible $F_{Cout}$ and $L$ \[82\]. In our model, we use the following effective $F_{Cout}$ for single-driver routing architectures. Bi-directional routing architectures do not have this constraint.

$$ F_{cout\_singlydriver} = \begin{cases} 1/L, & F_{cout} > 1/L \\ F_{cout}, & \text{otherwise} \end{cases} \quad (3.29) $$

---

4Segments rotate to new tracks as shown so that a single tile can be designed and then replicated for the fabrication mask layout of the FPGA.
3.3.4 Expansions Per Wire Segment $E_{wire}$

$E_{wire}$ describes the average number of routing resources that can be reached from a wire segment. There are two components as expressed in Equation 3.30: LB input pins reached through Connection Boxes (CBs), and other wire segments reached through Switch Boxes (SBs).

$$E_{wire} = E_{cbox} + E_{sbox}$$ (3.30)

3.3.4.1 Number of CB Connections From a Wire $E_{cbox}$

In general, a CB is surrounded by two logic blocks (illustrated in Figure 2.1 in Chapter 2). A wire passing through this CB will thus encounter $2 \cdot (I/4)$ input pins assuming that the $I$ input pins are evenly distributed across all four sides of a logic block. However, only a fraction, $F_{Cin}$, of these pins will have a programmable connection from the wire. In addition, each wire segment will pass through $L$ CBs. These relationships are expressed in Equation 3.31. Although wires also pass by I/O blocks, this occurs much less often, especially for large FPGAs. Hence we do not model this scenario for simplicity.

$$E_{cbox} = \frac{I}{2} \cdot F_{Cin} \cdot L$$ (3.31)

As an optimization in the VPR implementation (the tool that we will validate against in the next section), LB input pins are never expanded unless they belong to the target LB (it is not necessary because we know that this LB is not the net’s sink). We have included it here for completeness but set $E_{cbox} = 0$ when we validate our model.
3.3.4.2 Number of SB Connections from a Wire $E_{box}$ (Bi-directional Routing)

Next we describe connections in SBs for bi-directional routing architectures. We will use a horizontal wire segment for explanation, but the same reasoning also applies to vertical wire segments. At either ends of the horizontal wire segment, there is a programmable connection going to the left and the right, leading to two connections. For turning connections to vertical segments, there are two connections if the vertical segment terminates at the SB, and one connection if the vertical segment passes through. This is illustrated in Figure 3.3. The fraction of vertical tracks that terminates is $1/L$, and the remaining $(1 - 1/L)$ fraction of tracks pass through. This leads to an average of
of $2 \cdot (1/L) + 1 \cdot (1 - 1/L) = (L + 1)/L$ turning connections at each SB. Since each wire
segment interacts with $L + 1$ SBs (one at each end, and $L - 1$ that are passed through), a
wire segment has $(L + 1) \cdot (L + 1)/L$ turning connections overall. This is also shown in
Figure [3.3] The total number of other segments that can be reached from a segment in a
bi-directional routing architecture is summarized in Equation [3.3]

### 3.3.4.3 Number of SB Connections from a Wire $E_{esbox}$ (Single Driver Routing)

In the case of single-driver routing, there is only one connection in the same direction
of the wire segment occurring at the end of the wire segment. This is shown in Fig-
ure [3.4] For connections to vertical segments, there are two connections if the vertical
segment terminates at the SB. However, unlike bi-directional routing, if the vertical seg-
ment passes through, no turning connection is possible because wire segments can only
be driven at the vertical segment’s start. Again, the fraction of vertical segments that
terminate at an SB is $1/L$. This leads to an average of $2 \cdot (1/L)$ turning connections at
each SB. SB connections only occur at $L$ SBs (unlike $L + 1$ in the bi-directional case
since turning connections do not occur at the start tile). On average, each wire segment
has $2 \cdot (1/L) \cdot L = 2$ turning connections. The total number of wire segments that can be
reached from a wire segment in a single-driver routing architecture is thus 3.

$$E_{esbox} = \begin{cases} 
2 + \frac{(L+1)^2}{L}, & \text{bi-directional} \\
3, & \text{single-driver}
\end{cases} \quad (3.32)$$

### 3.3.5 Wires Explored $WE$

Determining the number of wire segments explored is also a difficult problem. It is
equivalent to determining the number of graph nodes visited in an A* search, which in
turn, has been extensively studied [57, 93]. Unfortunately, these studies focus on worst-case bounds and have not provided any practical methods to estimate the average number of nodes visited. In addition, these studies focus on search graphs that are very simplified compared to those used in a negotiation-based FPGA router. For these reasons, we took a semi-analytical approach to model this component using both empirical observations and some theories on A* search. We propose the following form and perform linear regression analysis to determine the set of fitting terms $\beta$.

$$WE = \left(\frac{WL}{L}\right)^2 (\beta_4 W - \beta_3 W (1 - F_{\text{Cout}})) + \beta_2 \frac{W}{F_{\text{Cin}}} \left(\frac{WL}{L}\right) + \beta_1 \left(\frac{WL}{L}\right)$$  \hspace{1cm} (3.33)

In A*, the number of nodes explored in the search graph can be related to the solution path length (in units of graph nodes) [93]. In our case, the search graph is the routing resource graph [19], a graph node is a routing resource (a wire segment for the purpose of this section) and the solution path length is the average wirelength of a routed net $WL$ (in units of logic block tiles).

There are four terms in Equation 3.33 that describe four general scenarios during the routing of a net. The $\beta$ factors can be seen as an indication of the frequency of occurrence of each scenario. However, these $\beta$ factors are essentially random variables as they depend on the evolution of the congestion cost terms, and also the tie-breaking mechanism of the priority queue insertion implementation. This is a difficult to predict source of error in our model.

Two of these terms are linear with respect to $WL$ and the other two are quadratic. For the linear terms, certain conditions allow the A* search to be very directed and appear more like a depth-first search. In these scenarios, the number of nodes explored is linearly related to the solution path length. The quadratic terms describe the scenarios...
Figure 3.5: Routing search wavefront for breadth-first and depth-first search on a two-terminal net. Shaded regions show where routing resources have been explored.

where many wire segments look equally as appealing (many resources have equal or very similar cost), thus resulting in a more breadth-first exploration. This occurs most often in the early routing iterations when the costs for each routing resource are fairly uniform. The effect of depth-first and breadth-first search is highlighted in Figure 3.5 for the routing of a two terminal net.

When breadth-first exploration occurs, every track in a channel is explored as the wavefront moves along the channel. Therefore, the channel width $W$ is a factor. However, when the breadth-first scenario occurs immediately after leaving the CLB, only some of the tracks in the channel are reachable if $F_{\text{Cout}} < 1$. More specifically, $W(1 - F_{\text{Cout}})$ tracks are not reachable. Hence, these are not explored and are subtracted from the overall count. These effects do not occur when the search is direct (the linear terms).

Since routing occurs from source to sink, $W/F_{\text{Cin}}$ describes the number of chances to make the final connection into the LB from a channel. Since the breadth-first scenario proceeds along all tracks in the channel, it can easily find an input pin connection and is thus not affected by limitations imposed by $F_{\text{Cin}}$. The directed search does not have this luxury and can explore down a path only to find that there is no connection into the
Table 3.3: Routing run-time model summary

<table>
<thead>
<tr>
<th>Description</th>
<th>Equ. #</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routing Time</td>
<td>3.26</td>
<td>$T_{routing} = C_{expansion} \cdot N_{iter} \cdot n_{nets} \cdot N_{exp}$</td>
</tr>
<tr>
<td>Avg. Expansions per net</td>
<td>3.27, 3.28</td>
<td>$N_{exp} = (N + N \cdot W \cdot F_{Cout}) + WE \cdot E_{wire}$</td>
</tr>
</tbody>
</table>
| Avg. Expansions per Wire              | 3.30, 3.31, 3.32, 3.32 | $E_{wire} = \begin{cases} 
\frac{L}{2} \cdot F_{Cin} \cdot L + 2 + \frac{(L+1)^2}{L}, & \text{bi-directional} \\
\frac{L}{2} \cdot F_{Cin} \cdot L + 3, & \text{single-driver} 
\end{cases}$ |
| Avg. Wires Explored per net           | 3.33   | $WE = \left(\frac{WL}{L}\right)^2 (\beta_4 W - \beta_3 W (1 - F_{Cout}))$ + $\beta_2 \left(\frac{WL}{L}\right) + \beta_1 \left(\frac{WL}{L}\right)$ |

CLB. This leads to additional exploration to find an input pin. The $\beta_2$ term models these extra "backtracking" explorations, and the $\beta_1$ term models wires explored from typical directed search. Finally, $WL$ is normalized by $L$ since we want the solution path to be expressed in units of routing resource graph nodes.\(^5\)

The $WE$ term requires the post-routing wirelength $WL$. This value is also affected by architecture (for example, less flexible routing architectures will lead to more round-about paths and hence longer wirelengths). However, there are currently no studies, to our knowledge, that model post-routing wirelength in FPGAs as a function of architecture. Modeling this is an open research problem.

3.3.6 Routing Run-time Model Summary

The model equations that relate routing run-time to the architecture are summarized in Table 3.3. Validation of this model in described in Section 3.4.

\(^5\)There is one routing resource graph node for each wire segment regardless of how many LBs the segment spans. More details on the routing resource graph can be found in Section 4.2 in Chapter 4.
### Table 3.4: Benchmarks details

<table>
<thead>
<tr>
<th>Training Set</th>
<th>Validation Set</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MCNC Circuits</strong></td>
<td><strong>Circuit</strong></td>
</tr>
<tr>
<td>s38584</td>
<td>6158</td>
</tr>
<tr>
<td>clma</td>
<td>4875</td>
</tr>
<tr>
<td>frisc</td>
<td>3124</td>
</tr>
<tr>
<td>spla</td>
<td>1978</td>
</tr>
<tr>
<td>bigkey</td>
<td>1682</td>
</tr>
<tr>
<td>diffeq</td>
<td>1341</td>
</tr>
<tr>
<td>tseng</td>
<td>1068</td>
</tr>
<tr>
<td>alu4</td>
<td>1011</td>
</tr>
<tr>
<td>ex1010</td>
<td>896</td>
</tr>
<tr>
<td>s298</td>
<td>756</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Synthetic Circuits</th>
<th><strong>Circuit</strong></th>
<th><strong>n</strong></th>
<th><strong>p</strong></th>
<th><strong>Circuit</strong></th>
<th><strong>n</strong></th>
<th><strong>p</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>synth01</td>
<td>52482</td>
<td>0.668</td>
<td>synth06</td>
<td>41748</td>
<td>0.670</td>
<td></td>
</tr>
<tr>
<td>synth02</td>
<td>39498</td>
<td>0.661</td>
<td>synth07</td>
<td>31278</td>
<td>0.679</td>
<td></td>
</tr>
<tr>
<td>synth03</td>
<td>31198</td>
<td>0.679</td>
<td>synth08</td>
<td>24053</td>
<td>0.663</td>
<td></td>
</tr>
<tr>
<td>synth04</td>
<td>23406</td>
<td>0.662</td>
<td>synth09</td>
<td>21517</td>
<td>0.645</td>
<td></td>
</tr>
<tr>
<td>synth05</td>
<td>19940</td>
<td>0.645</td>
<td>synth10</td>
<td>17376</td>
<td>0.617</td>
<td></td>
</tr>
</tbody>
</table>

#### 3.4 Run-time Model Validation

We evaluate the accuracy of our models by comparing the predicted run-times to measured run-times of a real CAD tool. We will first discuss the benchmarks used for validation as well as the tuning of model parameters, followed by the validation results.

##### 3.4.1 Benchmarks

For validation, we used the benchmark circuits listed in Table 3.4. Twenty of these are real circuits from the MCNC benchmark suite [128]. To supplement these, we also include ten larger synthetic circuits that mimic modern system-level designs [85]. Out of these thirty benchmarks, we use fifteen of them to act as a training set for determining the $\beta$ fitting terms of Equation 3.33. The other fifteen circuits are used for validation of the model. To divide these benchmarks, we first sorted the circuits by size and then assigned them in an alternating fashion.
Table 3.5: Parameter values for routing model validation

<table>
<thead>
<tr>
<th></th>
<th>$C_{\text{expansion}}$</th>
<th>$N_{\text{iter}}$</th>
<th>$\beta_1$</th>
<th>$\beta_2$</th>
<th>$\beta_3$</th>
<th>$\beta_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>290ns</td>
<td>20</td>
<td>-42.9178</td>
<td>0.1277</td>
<td>1.3521e-06</td>
<td>8.2096e-07</td>
</tr>
</tbody>
</table>

3.4.2 Model Calibration

Next we calibrate the $C_{\text{net}}$ and $C_{\text{expansion}}$ terms to the specific software implementation and computing machine. In this work, we compare our model against VPR version 5.02 running on a 2.16GHz Intel Core 2 Duo with 2GB of RAM. $C_{\text{net}}$ and $C_{\text{expansion}}$ were respectively measured as 225ns and 290ns. Performance details of this model applied to VPR v4.30 and bi-directional routing architectures can be found in Appendix A. The fitted $\beta$ terms for the routing run-time model are summarized in Table 3.5.

3.4.3 Placement Run-time Validation Results

To acquire measured run-times, we use the bounding-box linear congestion cost function in VPR and the default annealing schedule. We use all circuits listed in Table 3.4. For the measured placement time of a circuit, we take the average from three differently seeded placements. To acquire predicted run-times, we used the model described in Section 3.2 and the Smith/Das/Wilton wirelength model [107] to predict the post-placement wirelength and initial placement wirelength (using $p_i = 0.9$). The acceptance probability for calculating $T_o$ was set to $P_{\text{accept},i} = 0.95$. For values that can be predicted by the Lam/Wilton model (i.e. $n_c$, $f_{\text{avg}}$, $i$, $o$), we used the measured values to reduce any possible modelling inaccuracies from that model.

We found our model was very sensitive to the value of $\gamma$. Using a value of 0.95 causes very large over-predictions because it ignores the fact that time is spent at temperatures with extremely high and low acceptance ratios at the beginning and end of the anneal. To be more accurate, the expected value of $\gamma$ should be used. This was experi-
Table 3.6: Parameter values for placement model validation and applications

<table>
<thead>
<tr>
<th>Value</th>
<th>C&lt;sub&gt;net&lt;/sub&gt;</th>
<th>p&lt;sub&gt;i&lt;/sub&gt;</th>
<th>P&lt;sub&gt;accept&lt;/sub&gt;</th>
<th>γ</th>
</tr>
</thead>
<tbody>
<tr>
<td>225ns</td>
<td>0.9</td>
<td>0.95</td>
<td>0.915</td>
<td>0.915</td>
</tr>
</tbody>
</table>

mentally measured as 0.915 which was then used in our model. Table 3.6 summarizes the model parameter values that we will use during validation of our model.

Figures 3.6a and 3.6b show that our model is able to capture the impact of architecture on placement run-time when LUT and cluster size are varied, respectively. Each data point describes the run-time averaged across all circuits. Figure 3.6c shows a circuit by circuit breakdown of prediction error through a correlation plot for all placements generated in this section. Points above the line indicate over-prediction by the model and points below the line indicate under-prediction. Although the goal of the model is to capture the average impact on run-time, this correlation plot shows that the predictions for each circuit can be fairly accurate. There are some outliers due to the stochastic nature of simulated annealing. The main cause is when the average value of γ for the specific placement trial differs from the value (0.915) used by the model.

3.4.4 Routing Run-time Validation Results

To generate measured routing run-times, we use our validation benchmark set listed in Table 3.4. Each validation set circuit was placed three times with different seeds. This effectively gave us 45 circuits. As will be described, we directly varied F<sub>Cout</sub>, F<sub>Cin</sub>, L, and W. Measured times are taken starting after the routing resource graph is constructed in VPR<sup>6</sup>, to the end of the routing stage.

To gather predicted run-times, we used the model described in Section 3.3. To estimate WE we need the post-routing wirelength. Since there are currently no models

---

<sup>6</sup>The construction of this graph can take a significant amount of time for large FPGAs.
in the literature that model this, we used the measured post-routing wirelength to validate our model. As described in Section 3.3.1, $N_{iter}$ is modeled as a constant during low-stress routing. We set $N_{iter} = 20$ as observed during the experimental calibration for $WE$. The correlation of predicted versus measured $WE$ for every circuit in each experiment of this section is shown in Figure 3.7a.

To validate that our routing run-time model can capture the impact of the architecture, we perform three sets of experiments. In the first set, we vary $F_{Cout}$ from [0.2, 0.4, 0.6, 0.8, 1.0] and $L$ from [1, 2, 3, 4]. The combinations of these two parameters generates
20 architectures. For each value of $L$, we set $F_{Cout} = 0.2$ and perform a binary search route $^7$ to determine the minimum required channel width, $W_{req}$, for the circuit. We then increase this value by 20% to ensure low-stress routing conditions. This gives us a different channel width for each value of $L$ and circuit. Using these channel widths, we route the circuits to each of the 20 architectures. This is performed for all 45 circuit placements. The second experiment set jointly varies $F_{Cin}$ and $L$ in the same way. In the third experiment, we fix $F_{Cout}$, $F_{Cin}$, and $L$, and determine the minimum channel width for each circuit. We then choose the largest value in the suite and add 20% more tracks to ensure low-stress routing. Starting at this +20% value of $W$, we increase $W$ in increments of +10% up to +80% to investigate the impact of channel width on routing run-time.

Figures 3.7b, 3.7d, 3.7c, and 3.7e show the accuracy of our model when $F_{Cin}$, $F_{Cout}$, $L$, and $W$ are respectively varied. We show a representative set in each plot. For example in Figure 3.7b and 3.7d, we have data for $L = 1, 2, 3, 4$ but only show the case of $L = 1$. Overall, the trends are accurately captured. Finally, Figure 3.7f shows a circuit-by-circuit breakdown of run-time estimation errors for every routing performed in this section.

### 3.5 Example Applications

In this section, we show two applications of the run-time models to highlight some possible uses for FPGA architects.

---

$^7$Binary search route is an experimental method to find the minimum channel width $W_{req}$ that will allow the placed circuit to be routed. This is done by routing the circuit to different values of $W$ using a binary search.
Figure 3.7: Routing run-time model validation
3.5.1 Architectural Choices Under A CAD Run-Time Constraint

We pose the question of given a fixed amount of placement run-time that we can tolerate, $Time_{\text{max}}$, what logic architectures are possible? Figure 3.8 was generated using a purely analytical flow, combining results from the Lam/Wilton FPGA Logic Utilization model [69], the Smith/Das/Wilton FPGA Wirelength model [107] and our placement run-time model. We used the same parameter values as in Section 3.4.3 (for $P_{\text{accept,i}}, p_i$, and $\gamma$) targeting a user circuit with one million 2-input gates and a Rent parameter of 0.7. The number of inputs per cluster was set to $I = \frac{K}{2}N + 2$ [19]. The only unspecified LB architecture parameters are LUT size $K$ and cluster size $N$.

Each line in the plot corresponds to a placement run-time constraint ranging from $Time_{\text{max}} = 1, 2, 4, 6, 8, \text{ and } 10 \text{ hours}$. A point on a line indicates the minimum values for $K$ and $N$ that need to be used to not exceed a placement time of $Time_{\text{max}}$. Any values above and to the right of the curve are acceptable and satisfy $Time_{\text{place}}(K,N) \leq Time_{\text{max}}$. The intuition behind this is that increasing either $K$ or $N$ decreases the number of clusters
$n_c$, which means we need to place fewer blocks. Note that although we show continuous curves, $K$ and $N$ are discrete values. The continuous lines indicate the boundary values.

To highlight the power of such models, gathering this data using the models only takes several minutes; whereas an experimental approach would likely take days of data collection.

### 3.5.2 CAD Run-Time Optimization with Die Area Constraint

Architects often design new architectures based on a baseline architecture. For example, designing an FPGA for the next product generation using the current generation device as a basis. Using this as motivation, we pose the following question. *Given a baseline architecture with an area of $A_{\text{base}}$, and given that we can tolerate an area increase of $A_{\text{inc}}\%$, how much placement run-time reduction can be achieved?* This is an optimization problem to minimize CAD run-time in the presence of an area constraint.

We answer this using our run-time model and the area model described in [105] and [106]. This area model also uses the Fang/Rose channel-width estimation model described in [52]. Again, we employ a completely analytical flow. Since we only have three architecture parameters, $K, N$ and $I$, we first sweep these parameters in the range of $K = [2..7], N = [2..20], I = [N \cdot K..K/2 \cdot N + 2]$ and record into a table the corresponding area and run-times given by the models. Then given a baseline architecture with $A_{\text{base}}$ and a tolerable area increase of $A_{\text{inc}}$, we prune all entries in the table that exceed the acceptable amount of area increase. Of the remaining entries, we can find the architecture with the lowest placement run-time and compare it to our baseline run-time. This is graphically shown in Figure 3.9 using a baseline architecture of $(K, N, I) = (4, 10, 22)$. The x-axis shows tolerable area increase, and the y-axis shows the placement run-time normalized to the baseline run-time. The shape of this graph is due to the discrete nature
Figure 3.9: Application 2: CAD run-time optimization with die area constraint of the architecture parameters. As with the previous application, the analytical flow allows for extremely fast data collection compared to using an experimental methodology.

3.6 Summary

This chapter described an analytical model that relates key FPGA architectural parameters to the FPGA CAD place and route run-times. The major challenge tackled in the derivation of our model was balancing simplicity with accuracy, and to expose the architecture’s impact on the metric of run-time. The model was validated against the commonly used VPR academic FPGA place-and-route tool. Through this validation, we showed that the model can accurately capture the trends in run-time when various key architecture parameters \((K, N, F_{\text{Cin}}, F_{\text{Cout}}, W, L)\) are changed. Although we have made some assumptions regarding the algorithms and architecture, our derivations provide a sound methodology for deriving run-time models for different placement and routing algorithms or architectures. The chapter concluded with two example applications of the model in the context of FPGA architecture design.
Chapter 4

Storage-Efficient Representations
for FPGA Architectures

4.1 Overview and Further Motivations

This chapter describes a novel storage-efficient representation of FPGA architectures and dynamic memory management schemes aimed at improving the storage-requirement scalability of FPGA CAD tools. Unlike a CAD tool targeting an ASIC design flow, a CAD tool targeting an FPGA must be aware of every programmable switch and prefabricated routing resource on the device. As FPGA device sizes grow, the memory needed to store this information grows even faster. This trend was highlighted in Figure 1.1 of Chapter 1. Although the memory capacity of the workstations used to run the CAD tools is also scaling according to Moore’s Law, reducing the memory footprint of FPGA CAD tools is important for two main reasons: cost and CAD run-time.

In terms of cost, FPGA power-users typically have access to modern workstations. However, there are many designers that are designing circuits for FPGAs using low-cost
workstations that contain limited amount of RAM. FPGAs provide a low-cost low-risk entry to integration; this advantage is somewhat reduced if FPGA CAD tools require expensive workstations.

Memory usage has a significant impact on computation performance and hence software run-time. When the CPU needs to access the main-memory (RAM), there is usually a stall in the CPU’s processing as it waits for the relatively long memory access to complete. Because of this, various caching strategies have emerged as processor speeds have increased. Even with advanced caching techniques, a cache miss still requires a time-consuming main-memory access. In general, reducing the amount of memory usage can lead to faster run-times of the CAD tools because of more efficient cache usage. At the other extreme, if the CAD tools require more RAM than that is available on the workstation, the workstation begins to use the swap memory on the hard disk to supplement the RAM. Accessing the hard disk is much slower than accessing RAM memory. If the CAD tool is constantly reading and writing memory to the hard drive (thrashing), then the perceived speed of the CAD tools is drastically reduced. Furthermore, as processors move further into the multi-core paradigm, reducing storage requirements will lower the amount of inter-core communication, and reduce the strain on the main-memory bandwidth, both of which will increase perceived performance.

For these reasons, reducing the storage requirements of FPGA CAD algorithms is important. One of the most memory intensive steps in the FPGA CAD flow is routing [83]. The goal of the router is to efficiently use the prefabricated routing resources to implement each connection in the circuit being implemented by the FPGA. To do this, the routing algorithm needs to store two types of data during its execution regardless of the specific routing algorithm being used:
1. Architectural Data – The router requires a map of all physical routing resources available on the FPGA. As the number of programmable elements and wiring tracks increase, this map grows very quickly in size, and tends to be a dominating factor in the overall memory footprint of the FPGA CAD flow.

2. Temporary Routing Data – Storage is also required for each programmable resource to track various metrics and bookkeeping information used during routing. This data also has poor scalability since it is required for every single routing resource, but it is much smaller than the architectural data requirements.

We focus on improving the scalability of both components. Although software memory usage can be very implementation-specific, the research that we present in this chapter are high-level techniques; they can apply to any FPGA routing algorithm. The proposed techniques are specific to the domain of FPGAs, and lead to long-term benefits by reducing the order at which the CAD tool memory requirements scale.

The remainder of this chapter is organized as follows. Section 4.2 explains how FPGA CAD tools typically model the architectural data using a Routing Resource Graph. Section 4.3 presents the new representation for improving the scalability of the architectural data. To make the technique concrete, we implemented it into the academic VPR framework [19] and experimentally quantified the impact on memory usage and runtime. Section 4.4 presents these results. Section 4.5 proposes four dynamic memory-management schemes for reducing the temporary routing data storage requirements. Section 4.6 presents the corresponding results. Section 4.7 summarizes the chapter. Part of this work has been published in [36] and a journal version of the complete work has been published in [35].
4.2 Routing Resource Graph

The map of the physical wire segments and programmable switches can be represented by a Routing Resource Graph (RRG) [19, 90]. The RRG is a directed graph where each wire and logic block pin on the FPGA is represented by a node. In addition to wire and pin nodes, there are source and sink nodes to model pins that are logically equivalent (these nodes do not correspond to a physical element on the FPGA). Programmable connections between the resources are represented by directed edges. Directional connections such as tristate buffers are modeled by a single edge, and bi-directional connections such as pass transistors are modeled by a pair of edges. Figure 4.1 shows an example of the RRG for one logic block tile containing a 2-input lookup table (LUT) and a channel width of 2.

Each node of the RRG includes not only connectivity information, but also coordinates for mapping each node to its corresponding physical resource, information for the timing model, and possibly some algorithm-specific information. The size of this graph is directly proportional to the number of wire segments, pins, and programmable connections in the FPGA, and is very large for modern devices. A more detailed analysis of the impact of architecture on the size of this graph can be found in Appendix B.
4.3 Architectural Data

This section describes a storage-efficient scheme to replace the traditional routing resource graph discussed in the previous section. This scheme takes advantage of the regular tiled nature of FPGAs. Rather than explicitly representing all programmable resources in the FPGA, we divide the FPGA into tiles. Each tile corresponds to a tile type, where all tiles within a tile type are identical (with the possible exception of the starting and ending point of wire segments that span more than one block). For example, a tile type might contain a single logic block and its surrounding routing channels. An FPGA with heterogeneous blocks (such as memories or DSP blocks) would contain additional tile types consisting of the heterogeneous blocks. Additional tile types are required for the I/O blocks along the periphery of the FPGA. In brief, the number of tile types is very small compared to the total number of tiles in the entire FPGA. We then model the routing resources within a tile type only once. This is the key idea and we illustrate this point further through an example in Figure 4.2.

Figure 4.2a shows an FPGA with 35 logic block tiles \((N_x \cdot N_y = 7 \cdot 5 = 35)\). Figure 4.2b shows the corresponding full routing resource graph (edges are not shown). In general, the number of nodes in this graph, and hence the graph’s storage requirement, scales at a rate in \(O(N_x \cdot N_y \cdot (W/L + N + I))\) where \(W\) is the number of tracks per channel, \(L\) is the wire segment length, and \(N + I\) is the number of pins per logic block. Together, \((W/L + N + I)\) denotes the number of nodes per logic block tile. Figure 4.2c shows the new representation where the logic block tile is modeled only once. The number of nodes for this representation scales only at a rate in \(O(W + N + I)\). In other words, the storage requirements no longer increase as more logic block tiles are added. This is significant.

---

1 See Appendix B for more details.
as there are tens of thousands of logic block tiles in a modern FPGA [11, 125], and adding more logic blocks is the typical way to scale FPGA capacity.

While this main idea may appear straightforward, there are many challenging considerations related to modeling the programmable connections (edges), supporting additional tile types, and making the routing algorithm understand the new architecture representation. Sections 4.3.1 to 4.3.6 describe the modeling details, and Section 4.3.7 describes how the FPGA router is modified to use the new scheme.
4.3.1 Tile Type Representation

We illustrate our technique on a homogeneous FPGA without memories or DSP blocks. For such an FPGA, we require five tile types: CLB, IO_TOP, IO_BOTTOM, IO_LEFT, IO_RIGHT. These five tiles are shown in the first row of Figure 4.3. By replicating these tile-types accordingly, the FPGA being represented can be reproduced. The second row shows a graph representation of each tile, where each pin and wire segment has been replaced by a graph node.

Programmable connections are modeled differently depending on whether they are between nodes in the same tile, or between nodes in different tiles. In the following three subsections, we describe the specifics of modelling the connections.

4.3.2 Connections Within the Same Tile Type

For programmable connections between resources that lie within the same tile instance, the edges are modeled in the traditional manner as shown in Figure 4.4.
4.3.3 Connections Between Different Tiles of the Same Instance

When a programmable connection exists between routing resources that belong to different instances of the same tile type, we model using an edge that wraps around as shown in Figure 4.5. The upper connection in this example is a connection between two horizontal wire segments. In our graph of the tile, both of these wire segments are represented by the same node. When this happens the node emits an edge that returns to itself. The edge is labeled with two new quantities: $\Delta x$ and $\Delta y$. These values specify to which adjacent tile instance the edge connects. We assume the coordinate system shown in Figure 4.6 so that in this example, the $\Delta$ values for the edge are $(+1, 0)$. All edges are labeled in this way.

In the lower connection in Figure 4.5, a horizontal wire segment connects to a vertical wire segment. In this case, the two resources are represented by different nodes but are still part of the same tile type. By specifying the $(\Delta x, \Delta y)$ values consistently ($(-1, 0)$ in this example) the connectivity information is completely preserved.
4.3.4 Connections Between Different Tile Types

Programmable connections between tiles of different tile types can be handled in a similar way. However, sometimes the type of an edge’s destination tile depends on the coordinates of the tile in the FPGA array. For example, consider a logic block tile at the centre of an FPGA. The neighbouring tiles in this case will be other logic block tiles. On the other hand, a logic block tile on the periphery of the FPGA would have some neighbours that are I/O block tiles.

To model this, edges are added from the source node to all potential destination nodes across all potential tile types. This is shown graphically in Figure 4.7 in which two edges emit from the horizontal track in the right-most logic block tile. One edge wraps around to the same logic block tile, while the other edge connects to the I/O block tile type. The two edges are given the same $(\Delta x, \Delta y)$ labels ($(-1, 0)$ in this case).

For any given instance of the logic block tile, only one of the two edges corresponds to a physical routing switch; which edge depends on the coordinates of the tile. It is up to the routing algorithm to keep track of the $(x, y)$ tile location during maze routing, and
only expand those edges that correspond to physical switches. More details regarding the router are provided in Section 4.3.7.

4.3.5 Modelling Long Wires

Wires that span more than one tile require special attention. The graph that we have described thus far implies the use of wire segments that span a single tile. Consider a wire segment that spans four logic block tiles (i.e. \( L = 4 \)). At each instance of the logic tile, the wire may terminate, or it may pass through to the neighbouring instance. We can represent this using two edges, one to represent a hard connection (wire passing through), and one to represent a programmable connection (wire terminating). The choice of which edge to use is handled by the router depending on the current \((x,y)\) location being explored by the maze router.

The edge that corresponds to the hard connection is different than the edges discussed previously because it does not correspond to a programmable switch. We use a special type of edge called a delayless edge to represent this. In our implementation, these delayless edges are treated differently in that they do not result in a new element being placed into the priority queue. When a delayless edge is encountered, the router immediately performs neighbour expansion on the node pointed to by the delayless edge.
4.3.6 Depopulated Connection Boxes and Switch Boxes

So far, we have implicitly assumed that the connection boxes within each logic block tile are identical. However, as described in [19], this may not be the case; in a depopulated architecture, long wires do not connect to all of the connection boxes that they pass through. Although this could be modeled by creating a separate tile type for each logic block with a unique connection block pattern, it is more memory efficient to represent all possible connection block connections and leave it to the router to determine at run-time which edges correspond to physical switches (as in Section 4.3.4) based on the (x,y) location that is currently being explored. Architectures with depopulated switch blocks (in which not every switch block contains connections to all tracks) can be handled in the same manner.

It is interesting to note that creating separate tile types could be a way to trade-off memory usage and run-time as we will see in Section 4.4.5 that the proposed technique leads to some runtime overhead.

4.3.7 Changes To The Routing Algorithm

In general, the router has two new tasks:

1. To keep track of the (x,y) tile coordinate of each routing resource that is visited during maze expansion.

2. To determine which edges represent physical connections at the given (x,y) tile location and expand only these edges.

As an example, we implemented the proposed representation into the VPR router. The VPR router is based on the negotiation-based PathFinder algorithm [90]. The pseudo-code for this router is described in Algorithm 2 and 3 as reviewed in Chapter
2. Routing of a net progresses as follows. As long as the sink has not been found, the node with the best score is removed from the Priority Queue (PQ) and its neighbouring nodes are scored and added back into the PQ. The PQ is initialized with the source node of the net being routed. Since the router always knows the \((x,y)\) coordinates of the source and sinks being routed, the \((x,y)\) coordinates of every subsequent node that is added to the PQ can be calculated using the \((x,y)\) coordinate of the source node as the starting point, and the relative positioning information stored in the \((\Delta x, \Delta y)\) edge labels.

Normally, when expanding the neighbours of a node, the router iterates over the nodes fanout edges and adds each fanout node to the PQ as described in the `expand_neighbours()` function of Algorithm 3 (invoked in Line 7 of Algorithm 2). To facilitate our proposed representation, modifications to the router are needed in the form of replacing the `expand_neighbours()` function in Algorithm 3 with the one described in Algorithm 4.

Wire nodes are handled differently than other resources because they can span multiple tiles (i.e. wires where \(L > 1\)). Line 3 of Algorithm 4 uses a loop to iterate across all tiles for which the wire segment spans.

As described in the previous subsections, not all edges in the new graph will correspond to physical connections at each tile of the FPGA. Therefore, the router needs to perform a check before adding a fanout node to the PQ. This check is performed by the `fanout_exists(m,n)` function as follows.

First, we need to determine whether the tile type in which the fanout node \(n\) belongs corresponds to the expected tile type at the \((x,y)\) location of node \(n\). For example, if \(n\) is part of the logic block tile type in the graph, but the \((x,y)\) location of \(n\) is actually an I/O tile on the physical FPGA, then node \(n\) will not be added to the PQ. Valid coordinates for each tile type are summarized in Table 4.1 (or refer to Figure 4.6).
Algorithm 4 Pseudo-code highlighting changes to neighbour expansion

1: {Wires handled separately}
2: if \( m \) is a wire then
3: while \((x, y) \neq (x, y)\) of wire end tile do
4: for fanout nodes \( n \) of node \( m \) do
5: if fanout_exists\((m, n)\) then
6: Calculate cost of adding \( n \) to route path
7: Add \( n \) to \textit{PriorityQueue} with cost
8: end if
9: end for
10: \((x, y) + = (\Delta x, \Delta y)\)
11: end while
12: else
13: for fanout nodes \( n \) of node \( m \) do
14: if fanout_exists\((m, n)\) then
15: Calculate cost of adding \( n \) to route path
16: Add \( n \) to \textit{PriorityQueue} with cost
17: end if
18: end for
19: end if

Table 4.1: Valid coordinates for tiles in a typical homogeneous FPGA

<table>
<thead>
<tr>
<th>Tile Type</th>
<th>( x ) Range</th>
<th>( y ) Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLB</td>
<td>( 1 &lt; x &lt; N_x + 1 )</td>
<td>( 1 &lt; y &lt; N_y + 1 )</td>
</tr>
<tr>
<td>IO_TOP</td>
<td>( 1 &lt; x &lt; N_x + 1 )</td>
<td>( y = N_y + 1 )</td>
</tr>
<tr>
<td>IO_BOTTOM</td>
<td>( 1 &lt; x &lt; N_x + 1 )</td>
<td>( y = 0 )</td>
</tr>
<tr>
<td>IO_LEFT</td>
<td>( x = 0 )</td>
<td>( 1 &lt; y &lt; N_y + 1 )</td>
</tr>
<tr>
<td>IO_RIGHT</td>
<td>( x = N_x + 1 )</td>
<td>( 1 &lt; y &lt; N_y + 1 )</td>
</tr>
</tbody>
</table>

After the tile type of \( n \) has been confirmed as valid, we then need to check whether a programmable connection between \( m \) and \( n \) exist at the \((x, y)\) location of \( m \) and \( n \). For example, if \( m \) is a logic block output pin, and \( n \) is a wire segment, then a programmable connection exists as determined by the connection box architecture at the given tile. Table 4.2 summarizes the parameters that determine whether a programmable edge exists.
Table 4.2: Dependencies for validity of edges

<table>
<thead>
<tr>
<th>Resource type of m</th>
<th>Resource type of n</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>wire segment</td>
<td>wire segment</td>
<td>Track number of ( m ) and ( n ) ((x,y)) location of ( m ) and ( n ) switch box topology</td>
</tr>
<tr>
<td>wire segment</td>
<td>logic block or I/O block output pin</td>
<td>Track number of ( m ) Pin number of ( n ) ((x,y)) location of ( m ) and ( n ) connection box topology</td>
</tr>
<tr>
<td>logic block or I/O block output pin</td>
<td>wire segment</td>
<td>Pin number of ( m ) Track number of ( n ) ((x,y)) location of ( m ) and ( n ) connection box topology</td>
</tr>
</tbody>
</table>

In terms of overheads, the router changes for handling Task 1 requires the addition of several new fields to each PQ data object. These fields are used to track the \((x,y)\) location of the associated routing resource, and to properly handle traceback. This leads to a small and insignificant memory overhead that is dependent on the PQ size. We quantify this overhead in Section 4.4.6. Handling Task 2 leads to a run-time overhead and is quantified in Section 4.4.5.

4.4 Architectural Data - Experimental Results

In this section, we measure the impact of the proposed technique on the router’s storage requirements. We first describe the experimental framework in Section 4.4.1. In Sections 4.4.2 to 4.4.4 we measure the memory reductions as a function of the FPGA array size \( N_{array} \), channel width \( W \), and segment length \( L \). In Section 4.4.6, we show that the memory overhead introduced into the PQ data to be insignificant. In addition, we quantify the proposed techniques impact on run-time in Section 4.4.5.
4.4.1 Experimental Framework For \( N_{array} \), \( W \), and \( L \)

For the experiments in Section 4.4.2, 4.4.3, and 4.4.4 we pause the VPR process after it loads all of the architectural and temporary routing data structures, but before the routing algorithm actually starts. The PQ memory requirements are not included here but is discussed in more detail in Section 4.4.6.

After pausing, we measure the virtual memory usage of the VPR process. Virtual memory is the amount of memory that the process requires regardless of how much physical memory is actually installed on the machine. We take this measurement using the UNIX command `top`. Top’s ability to give detailed memory usage information is sufficient for our purpose.

In the following experiments, we route the same circuit to a number of different architectures that vary in \( N_{array} \), \( W \), and \( L \). For each architecture, we assume that logic blocks use 4-input LUTs (\( K = 4 \)), consist of ten LEs (\( N = 10 \)), and have 22 unique inputs (\( I = 22 \)).

4.4.2 Experimental Results for Array Size \( N_{array} \)

Figure 4.9 shows the memory requirements of the router using the new architecture representation and the original VPR router as a function of the array size \( N_{array} \) (Note that \( N_{array} = N_x = N_y \)). For small array sizes (i.e. \( N_{array} = 25 \)), the memory footprint of the proposed technique is only slightly better (1.7x smaller). As the array size increases, the difference in memory footprint becomes more dramatic. At \( N_{array} = 200 \), the memory footprint of the proposed technique is 6.5x smaller. Note that a 200x200 architecture with a channel width of \( W = 150 \) is comparable to current commercial devices \(^2\) (see Table 4.3 for some example device sizes).

\(^2\)At the time that this study was performed
Although the storage required for the architectural data is no longer a function of \( N_{\text{array}} \), the storage required to implement the temporary routing data structures is still proportional to \( N_{\text{array}} \). This explains why our storage requirements still grow slightly as \( N_{\text{array}} \) increases.

Figure 4.9 shows how the memory reductions depend on \( N_{\text{array}} \) for various fixed channel widths. The vertical axis shows the memory reductions (baseline memory footprint / new memory footprint); larger values are desired. Each line corresponds to a
Table 4.3: Device sizes for some commercial FPGAs [8][9][10][11][124][125].

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Device Family</th>
<th>(N_x)</th>
<th>(N_y)</th>
<th>(W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Altera</td>
<td>StratixIII (Largest)</td>
<td>120</td>
<td>120</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>StratixIII (Smallest)</td>
<td>45</td>
<td>45</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>StratixII (Largest)</td>
<td>100</td>
<td>96</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>StratixII (Smallest)</td>
<td>33</td>
<td>26</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>CycloneIII (Largest)</td>
<td>88</td>
<td>88</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>CycloneIII (Smallest)</td>
<td>20</td>
<td>20</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>CycloneII (Largest)</td>
<td>68</td>
<td>68</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>CycloneII (Smallest)</td>
<td>19</td>
<td>19</td>
<td>80</td>
</tr>
<tr>
<td>Xilinx</td>
<td>Virtex4 (Largest)</td>
<td>192</td>
<td>116</td>
<td>220</td>
</tr>
<tr>
<td></td>
<td>Virtex4 (Smallest)</td>
<td>64</td>
<td>24</td>
<td>220</td>
</tr>
<tr>
<td></td>
<td>Virtex5 (Largest)</td>
<td>240</td>
<td>108</td>
<td>220</td>
</tr>
<tr>
<td></td>
<td>Virtex5 (Smallest)</td>
<td>80</td>
<td>30</td>
<td>220</td>
</tr>
</tbody>
</table>

different channel width. The wire length is \(L = 8\) (we experimented with \(L = 1, L = 4,\) and \(L = 16\) and the conclusions were similar). As shown in the graph, as \(N\) increases, the memory savings increases but reaches an asymptote.

The asymptote can be explained using a similar rational as Amdahl’s Law where, beyond a certain point, the architectural data becomes insignificant compared to the temporary routing data. Consider the original representation. Assume the storage requirements for the architectural information and temporary routing information to grow at a rate of \(a \times N_{array}^2\) and \(t \times N_{array}^2\), respectively, where \(a\) and \(t\) are constants. Hence, the overall footprint grows at \((a + t) \times N_{array}^2\). In the new representation, denote the storage requirements for the architectural and temporary routing information to grow at a rate of \(A\) and \(t \times N_{array}^2\), respectively (the storage requirements for the architectural information is not a function of \(N_{array}\) in the new representation). This gives an overall growth rate of \(A + t \times N_{array}^2\). Note that \(t\) is the same constant in both representations because the temporary routing information representation is unchanged in the proposed technique.

Using these definitions, the memory reduction factor is \(((a + t) \times N_{array}^2)/(A + t \times N_{array}^2)\).
Figure 4.10: Achievable memory reductions for architectures with various channel widths $W$ and wire segment lengths $L$.

$N_{array}^2$. When $N_{array}$ becomes large, $A$ becomes insignificant and the ratio approaches $(a/t) + 1$. This value becomes the asymptote for the amount of memory that can be saved for the family of architectures with a given $W$ and $L$.

By using the case of $N_{array} = 200$, we can conservatively approximate the asymptote. Figure 4.10 shows the achievable memory reductions using this approximation. Each line represents an architecture with a different wire segment length $L$. This figure shows that for small channel widths ($W = 50$), we can achieve a 3x-6x reduction in memory footprint. For larger channel widths, we can achieve 5x-13x reduction depending on $L$.

4.4.3 Experimental Results for Channel Width $W$

Figure 4.11 shows the storage requirements as a function of channel width for a family of architectures with $N_{array} = 100$, and $L = 4$. The storage requirements of both repre-
Figure 4.11: Memory footprint for architectures with various channel widths. \( N_{array} = 100, L = 4 \).

sentations increase linearly but differ by a constant factor. This is due to the absence of the dependance on \( N_{array} \) in the new representation. As an example, suppose we increase \( W \) by 1. In the CLB tile of the proposed technique, the number of nodes is increased by two (one for the horizontal channel and one for the vertical channel). But using the original representation, the number of nodes for CLB channels is increased by \( N_{array} \times 2 \) because there are \( N_{array} \) CLB tiles.

### 4.4.4 Experimental Results for Wire Length L

Figure 4.12 shows that the memory reduction also changes with \( L \). As \( L \) increases, less storage is required for wire resources. This reduces both the connectivity and temporary information in the original representation. In the proposed technique, this only reduces the temporary information. Since the connectivity information in the proposed technique is fixed and relatively small, its overall footprint decreases much faster than the full RRG, leading to an overall decrease in memory usage.
Figure 4.12: Memory reductions for architectures with various values of $W$ and $L$ ($N_{array} = 200$).

4.4.5 Impact on Routing Time

The additional steps added to the router described in Section 4.3.7 will increase the router’s execution time. In this section, we measure this increase. Both versions of VPR (with and without our architecture representation) were instrumented to count the number of executed clock ticks at three points in the flow: immediately before placement, after placement, and after routing. These three values allow us to calculate the placement time, routing time, and total place-and-route run-time. All measurements were performed on an Intel Core 2 Duo 2.16GHz processor.

We are interested in how the proposed technique performs across different benchmark circuits because the number of nets and sinks affect the amount of time spent on routing. We use two suites of benchmark circuits. The first suite consists of the twenty largest MCNC circuits. The second suite consist of seven synthetic circuits created by stitching together a number of MCNC circuits using the technique described in [70]. Although these circuits are much larger than the twenty largest MCNC circuits, they are still not as big as some of the architectures that we investigated in the previous subsec-
Table 4.4: CAD run-time results for synthetic benchmark circuits in seconds.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>$N_{array}$</th>
<th>$W$</th>
<th>Place Time</th>
<th>Baseline Run-times</th>
<th>New Run-times</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Route</td>
<td>Total</td>
<td>Route</td>
</tr>
<tr>
<td>30seq20com</td>
<td>43</td>
<td>72</td>
<td>321.07</td>
<td>100.16</td>
<td>421.23</td>
<td>224.88</td>
</tr>
<tr>
<td>3lrg50sml</td>
<td>35</td>
<td>56</td>
<td>171.14</td>
<td>44.48</td>
<td>215.62</td>
<td>103.19</td>
</tr>
<tr>
<td>50mixedsize</td>
<td>44</td>
<td>54</td>
<td>298.85</td>
<td>59.73</td>
<td>358.68</td>
<td>134.67</td>
</tr>
<tr>
<td>60mixedsize</td>
<td>43</td>
<td>62</td>
<td>312.91</td>
<td>92.29</td>
<td>405.20</td>
<td>208.52</td>
</tr>
<tr>
<td>70s1423</td>
<td>40</td>
<td>47</td>
<td>205.47</td>
<td>68.18</td>
<td>273.65</td>
<td>151.28</td>
</tr>
<tr>
<td>Lotsaffs2</td>
<td>33</td>
<td>59</td>
<td>131.49</td>
<td>29.97</td>
<td>161.46</td>
<td>69.95</td>
</tr>
<tr>
<td>70sml</td>
<td>33</td>
<td>99</td>
<td>198.79</td>
<td>89.04</td>
<td>287.83</td>
<td>194.88</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: CAD run-time results for MCNC benchmark circuits in seconds.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>$N_{array}$</th>
<th>$W$</th>
<th>Place Time</th>
<th>Baseline Run-times</th>
<th>New Run-times</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Route</td>
<td>Total</td>
<td>Route</td>
</tr>
<tr>
<td>alu4</td>
<td>11</td>
<td>39</td>
<td>6.56</td>
<td>2.55</td>
<td>9.11</td>
<td>6.13</td>
</tr>
<tr>
<td>apex2</td>
<td>11</td>
<td>50</td>
<td>7.99</td>
<td>2.64</td>
<td>10.63</td>
<td>6.39</td>
</tr>
<tr>
<td>apex4</td>
<td>10</td>
<td>24</td>
<td>4.08</td>
<td>1.06</td>
<td>5.14</td>
<td>2.1</td>
</tr>
<tr>
<td>bigkey</td>
<td>27</td>
<td>18</td>
<td>15.99</td>
<td>4.79</td>
<td>20.78</td>
<td>9.75</td>
</tr>
<tr>
<td>clma</td>
<td>22</td>
<td>65</td>
<td>75.47</td>
<td>41.23</td>
<td>116.70</td>
<td>95.2</td>
</tr>
<tr>
<td>des</td>
<td>32</td>
<td>23</td>
<td>25.88</td>
<td>8.50</td>
<td>34.38</td>
<td>19.1</td>
</tr>
<tr>
<td>difeq</td>
<td>10</td>
<td>27</td>
<td>8.49</td>
<td>1.33</td>
<td>9.82</td>
<td>2.85</td>
</tr>
<tr>
<td>dsip</td>
<td>27</td>
<td>21</td>
<td>16.36</td>
<td>6.77</td>
<td>23.13</td>
<td>11.98</td>
</tr>
<tr>
<td>elliptic</td>
<td>16</td>
<td>46</td>
<td>24.63</td>
<td>7.94</td>
<td>32.57</td>
<td>19.82</td>
</tr>
<tr>
<td>ex1010</td>
<td>10</td>
<td>9</td>
<td>3.09</td>
<td>0.54</td>
<td>3.63</td>
<td>1.31</td>
</tr>
<tr>
<td>ex5p</td>
<td>5</td>
<td>8</td>
<td>0.96</td>
<td>0.08</td>
<td>1.04</td>
<td>0.11</td>
</tr>
<tr>
<td>frisc</td>
<td>15</td>
<td>59</td>
<td>27.55</td>
<td>15.21</td>
<td>42.76</td>
<td>36</td>
</tr>
<tr>
<td>misex3</td>
<td>10</td>
<td>42</td>
<td>6.55</td>
<td>2.38</td>
<td>8.93</td>
<td>5.57</td>
</tr>
<tr>
<td>pdc</td>
<td>16</td>
<td>72</td>
<td>25.53</td>
<td>17.60</td>
<td>43.13</td>
<td>39.81</td>
</tr>
<tr>
<td>s298</td>
<td>9</td>
<td>23</td>
<td>3.34</td>
<td>1.06</td>
<td>4.40</td>
<td>2.53</td>
</tr>
<tr>
<td>s38417</td>
<td>21</td>
<td>34</td>
<td>44.22</td>
<td>6.43</td>
<td>50.65</td>
<td>12.75</td>
</tr>
<tr>
<td>s38584</td>
<td>22</td>
<td>37</td>
<td>56.91</td>
<td>11.90</td>
<td>68.81</td>
<td>25.6</td>
</tr>
<tr>
<td>seq</td>
<td>11</td>
<td>52</td>
<td>10.22</td>
<td>4.56</td>
<td>14.78</td>
<td>10.21</td>
</tr>
<tr>
<td>spla</td>
<td>15</td>
<td>59</td>
<td>19.62</td>
<td>12.61</td>
<td>32.23</td>
<td>30.72</td>
</tr>
<tr>
<td>tseng</td>
<td>11</td>
<td>19</td>
<td>6.22</td>
<td>0.77</td>
<td>6.99</td>
<td>1.46</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

However, we found the impact on CAD run-time to be generally independent of circuit size. Each circuit was placed and routed to the minimum array size and channel width.

Table 4.4 reports the run-time results for the suite of synthetic circuits. Columns
2-3 shows the minimum array size and channel width required to place and route the circuit. Columns 4-6 show the placement, routing, and total run-times when using the original version of VPR. Columns 7-8 shows the new routing and overall run-times and Columns 9-10 shows the ratios. The average increase in routing time is 2.26x and the average increase in overall compile time is 1.28x. The overhead for each circuit is close to the average, meaning that the compile time overhead is generally independent of the FPGA size. This is because the overhead depends mainly on the extra time required to perform the `fanout_exists()` boolean evaluations (described in Section 4.3.7).

The average increase in route time and overall place-and-route run-time for the MCNC suite were found to be 2.18x and 1.30x respectively as summarized in Table 4.5. These results are similar to that of the Synthetic benchmark suite.

### 4.4.6 Impact on Priority Queue Memory Requirements

In this section, we discuss the impact of our technique on the PQ storage requirements. We will show that increased memory usage in the PQ from our technique does not diminish the memory reductions reported in the previous sections. First recall that the PQ is used by the router to prioritize the nodes of the expansion wavefront. We are interested in the peak size of the PQ. The peak size of the PQ can increase for a number of reasons such as when the source and sink are far apart, when a net has very high fanout, or when the route has to take a roundabout path (due to congestion).

In Figure 4.13, we routed the MCNC and Synthetic circuits to a minimum-sized, minimum-channel width FPGA. In each case, we kept track of the peak PQ size across all nets and routing iterations. The y-axis shows the peak number of routing resources in the PQ. The x-axis shows the FPGA size $N_{array}$ used for the circuit. Note that we separate out the low-density circuits (in that only a small percentage of logic blocks on
the FPGA are actually used by the circuit)in the MCNC suite (bigkey, des, and diffeq) because they are I/O limited and only use 15% of the logic blocks. Therefore, they should have a much smaller effective FPGA size.

Although the peak PQ size for a given FPGA size varies, there is an increasing trend in the envelope as shown by the two trend-lines. The steeper line marks the envelope for the Synthetic and high-density MCNC benchmarks. The other line marks the envelope for the low-density MCNC circuits. To be conservative, we extrapolate the steeper trend-line to a 200x200 FPGA. Here, the trend predicts a maximum PQ size of 120644 nodes. In VPR, for example, one PQ entry requires 24 Bytes, leading to a peak PQ memory requirement of 2.9MB. With our representation, a PQ entry requires several new fields to perform traceback, bringing it to 30 Bytes, leading to a peak PQ memory requirement of 3.6MB. This is less than 1% of the overall memory usage when using our proposed techniques. An increase in the PQ memory requirements does not diminish the memory reductions brought by our techniques.

**Figure 4.13:** Maximum priority queue size for MCNC and META circuits.
4.4.7 Impact on Routing Solution

An important property of this technique is that all of the architectural information is maintained exactly. This means that there is no change to the detailed routing architecture as seen by the router. This was verified by building the full RRG from our modified graph, and then comparing it to the RRG built in the traditional manner.

4.5 Temporary Routing Data

After the memory requirement of the architectural data is reduced as described in the previous sections, the storage for the Temporary Routing Data becomes dominant. Normally, a Temporary Routing Data Object (TRDO) is allocated for each routing resource on the FPGA. This object contains various scoring parameters, traceback information, and other bookkeeping information needed by the routing algorithm.

In this section, we conduct a study on the lifespan of these TRDOs through the application of four memory management schemes. From this investigation, we find that only a portion of these TRDOs are in use at any given time. Taking advantage of this observation allows for reductions in the temporary routing data storage requirements. Although we use the VPR router to perform this investigation, the results of this study can apply to any FPGA routing algorithm with a maze-routing component. The techniques presented here are independent of and complementary to the technique presented earlier to reduce the architectural data storage requirements.

4.5.1 Lifespan of TRDOs

We begin our analysis of TRDO lifespans by making three observations:

1. We only need to allocate TRDO storage for routing resources that are explored during the maze-routing process. Unexplored resources are not scored and do not
require any bookkeeping information.

2. In practice, the number of resources explored is always less than the total number of resources on the chip [19]. For low and medium-density designs, the pins (and source/sink routing resource nodes) associated with unused logic blocks will never be explored. These circuits will also have a lower interconnect demand than that supplied by the FPGA, meaning that a significant number of the wiring resources will also be unused, further reducing the number of nodes that must be explored during routing. Even in high-density designs, the routing congestion often varies across the design [114]. This leaves regions of low routing-congestion and thus a high amount of unused routing resources.

3. The routing of different nets, or even the same net between different routing iterations, leads to the exploration of different sets of routing resources. This means that the TRDO for a given routing resource may not have to persist throughout the entire routing process.

An upper and lower bound can be written for the number of resources that will be explored. In the worst case (upper bound), all routing resources are explored during the routing of a net. This should never happen in a negotiation based router unless there is an architectural flaw that makes a route impossible even when ignoring congestion. In the best case (lower bound), the router only explores the resources used in the final routing solution. Although this never happens in practice (we would need an oracle router to achieve this), a directed-wavefront expansion during maze-routing brings the number of explored resources close to the best case.
4.5.2 Simple Memory Management Schemes

Using the observations in the previous subsection, we can reduce the peak temporary routing data memory requirements by intelligently allocating storage for a TRDO only when we know for certain that a routing resource will be explored. In addition, we can deallocate the space for a TRDO when we know that the data in a TRDO will no longer be used. An important task is to determine when and how often to perform this deallocation such that we do not incur a significant run-time overhead.

Here we describe three simple memory management schemes. Each scheme allocates memory for a resource when it is first explored (a node is deemed “explored when it is removed from the priority queue). Memory deallocation is handled as follows:

4.5.2.1 Scheme 1

In this scheme, once memory is allocated for a resource, it persists until the end of the routing process. In other words, no dynamic memory deallocation is employed.

4.5.2.2 Scheme 2

At the end of a routing iteration, the TRDO for a routing resource is deallocated if the routing resource is not occupied by a net and if its scoring parameters have not been changed from initial default values (for example, in a negotiation-based router, historical congestion costs must persist between routing iterations).

4.5.2.3 Scheme 3

In this scheme, memory deallocation occurs at the end of routing each net. This can lead to a significant run-time overhead for circuits with a large number of nets. However, this is also the most aggressive deallocation scheme possible and will lead to the
largest memory reduction. To reduce the run-time overhead, it is possible to perform deallocation after routing every $x$ nets where $x$ is a knob that can be arbitrarily adjusted. However, we do not investigate this knob further. Instead, we set $x$ to 1 for this scheme.

### 4.5.3 Design-Adaptive Memory Management Scheme

We also present a design-adaptive algorithm that employs a memory-aware net ordering strategy. The key idea in our design-adaptive scheme is to order the nets intelligently to minimize the amount of allocation and deallocation required. To accomplish this, we first divide the FPGA into (possibly overlapping) square regions of size $N_{\text{region}}$ by $N_{\text{region}}$ logic blocks, where $N_{\text{region}}$ depends on the design (to be discussed below). Each net is then assigned to the region that shares the most overlap with that nets bounding box. During routing, the router iterates across each region and routes the nets belonging to that region before proceeding to the next region. Note that while routing a net, all routing resources (including those not in the region) are available to be explored. Intuitively, however, most routing will occur within the region. After routing the nets in a region, the router performs a deallocation pass.

Compared to Scheme 3 in Section 4.5.2.3, this new algorithm will reduce the number of unnecessary deallocations, improving the run-time. The exploration for nets within a region will often visit the same routing resources. By not deallocating the TRDOs for these routing resources until all nets assigned to a region have been routed, fewer unnecessary deallocations will be performed. The peak memory reduction capabilities, however, will be somewhat worse than Scheme 3, since a net will rarely be routed entirely within a region. In the next section, we experimentally evaluate the run-time and memory reduction capabilities of this algorithm.

An important consideration is how to choose the size, shape, and location of the
regions. In our experiments, we fix \( N_{\text{region}} = \frac{\overline{B}}{2} \) where \( \overline{B} \) is the average half-perimeter of the bounding box of each net in the design (so a design that tends to have smaller net bounding boxes will have smaller regions). We also assume that regions are overlapped such that the start coordinate of a region is \( \frac{N_{\text{region}}}{2} \) logic blocks to the right of the previous region and each row of regions is \( \frac{N_{\text{region}}}{2} \) logic blocks below the previous row of regions. Unlike all other techniques in this chapter, re-ordering the nets may change the routing solution slightly\(^3\). However, experimentally we determined that the impact on the minimum channel width and overall circuit delay was negligible.

### 4.6 Experimental Results - Temporary Routing Data

In this section, we quantify the impact on memory and run-time of the schemes described in the previous section.

#### 4.6.1 Impact on Memory

To experimentally measure the impact of our algorithms on the memory required to store the temporary routing data, we mapped the 20 largest MCNC circuits to a minimum-sized, minimum channel width FPGA. We assumed logic blocks to have 22 inputs and consist of ten 4-input LUT and register pairs. The routing architecture uses length 4 wire segments and the Wilton switch-box [122].

Rather than measuring the peak memory usage directly, we keep track of the maximum number of TRDOs allocated at any given time during routing. Figure 4.14 shows this quantity normalized by the total number of routing resources available on the FPGA for each benchmark circuit (this is equal to the reduction in memory needed to store all TRDOs, compared to a router that maintains a TRDO for every resource). For each

---

\(^3\)The impact will not be significant unless net-ordering is a specific component of the routing algorithm being used. This is not the case in VPR’s PathFinder algorithm.
Figure 4.14: Temporary data memory reductions for the four memory management schemes and the lower-bound. Results shown for routing to minimum channel width $W_{\text{min}}$.

circuit, we show the results for all three simple memory deallocation schemes, as well as the design-adaptive scheme. We also show the theoretical minimum that could be obtained; this quantity is the ratio of the number of routing resources used in the final routing solution divided by the total number of routing resources on the chip.

As the graph shows, the memory reductions are significant. On average, our algorithms reduce the peak memory needed to store TRDOs by between 24% and 43% of the original value, compared to a theoretical minimum of 51%.

To mimic low to medium-density designs, we repeated the experiments assuming channel widths of 1.25x $W_{\text{min}}$, 1.5x $W_{\text{min}}$, and 2x $W_{\text{min}}$ where $W_{\text{min}}$ is the minimum channel width required to route the circuit. These results are summarized in Table 4.6. As the table shows, the memory reduction is more pronounced for architectures with wider channels, since a smaller proportion of the routing tracks need to be explored.
Table 4.6: Average achievable temporary routing data memory reductions for three memory management schemes. Lower bound is also shown.

<table>
<thead>
<tr>
<th>Channel Width</th>
<th>Scheme 1</th>
<th>Scheme 2</th>
<th>Scheme 3</th>
<th>Design Adaptive</th>
<th>Theoretical Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00X $W_{\text{min}}$</td>
<td>0.76</td>
<td>0.66</td>
<td>0.57</td>
<td>0.59</td>
<td>0.49</td>
</tr>
<tr>
<td>1.25X $W_{\text{min}}$</td>
<td>0.75</td>
<td>0.61</td>
<td>0.51</td>
<td>0.53</td>
<td>0.44</td>
</tr>
<tr>
<td>1.50X $W_{\text{min}}$</td>
<td>0.72</td>
<td>0.56</td>
<td>0.45</td>
<td>0.49</td>
<td>0.40</td>
</tr>
<tr>
<td>2.00X $W_{\text{min}}$</td>
<td>0.67</td>
<td>0.50</td>
<td>0.38</td>
<td>0.41</td>
<td>0.34</td>
</tr>
</tbody>
</table>

4.6.2 Impact on Run-Time

Table 5 summarizes the impact on routing run-time. Column 1 lists the benchmark names. Column 2 lists the routing time on an unmodified version of VPR. Column 3-6 lists the routing time using the three different schemes as well as the design-adaptive method. Column 7-10 the ratio of each run-time to the run-time of the unmodified VPR.

We verified that the change in route time for different channel widths were within expected noise margins, and hence show only the run-time results for routing to a minimum channel-width FPGA.

There are three sources of run-time overhead. The first source is a potential decrease in cache performance due to memory fragmentation. The second source is the need to check whether a TRDO has been allocated each time a routing resource is removed from the priority queue. The third source is the time to perform the memory deallocation. In the following subsections, we consider each of these sources.

4.6.2.1 Cache Performance Overhead

First consider the overhead due to possible cache performance degradation. This quantity can be estimated as follows. Observe that in Scheme 1, the run-time overhead is due to a possible cache performance degradation as well as the code required to check if a TRDO has been allocated for a given resource. We measured the cache miss rate of
Table 4.7: Routing run-time overhead of each scheme

<table>
<thead>
<tr>
<th>Design</th>
<th>Baseline</th>
<th>Scheme 1</th>
<th>Scheme 2</th>
<th>Scheme 3</th>
<th>Design Adaptive 1</th>
<th>Design Adaptive 2</th>
<th>Design Adaptive 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>alu4</td>
<td>1.15</td>
<td>1.20</td>
<td>1.22</td>
<td>2.09</td>
<td>1.26</td>
<td>1.047</td>
<td>1.057</td>
</tr>
<tr>
<td>apex2</td>
<td>1.56</td>
<td>1.64</td>
<td>1.67</td>
<td>2.82</td>
<td>1.56</td>
<td>1.052</td>
<td>1.069</td>
</tr>
<tr>
<td>apex4</td>
<td>0.52</td>
<td>0.53</td>
<td>0.54</td>
<td>0.93</td>
<td>0.38</td>
<td>1.028</td>
<td>1.049</td>
</tr>
<tr>
<td>bigkey</td>
<td>1.38</td>
<td>1.43</td>
<td>1.46</td>
<td>2.94</td>
<td>2.19</td>
<td>1.036</td>
<td>1.054</td>
</tr>
<tr>
<td>clma</td>
<td>10.93</td>
<td>11.64</td>
<td>11.77</td>
<td>37.50</td>
<td>12.15</td>
<td>1.064</td>
<td>1.076</td>
</tr>
<tr>
<td>des</td>
<td>3.02</td>
<td>3.15</td>
<td>3.21</td>
<td>7.92</td>
<td>3.92</td>
<td>1.044</td>
<td>1.064</td>
</tr>
<tr>
<td>difeq</td>
<td>0.42</td>
<td>0.44</td>
<td>0.45</td>
<td>1.03</td>
<td>0.46</td>
<td>1.046</td>
<td>1.070</td>
</tr>
<tr>
<td>dsip</td>
<td>1.87</td>
<td>1.94</td>
<td>1.99</td>
<td>4.75</td>
<td>1.90</td>
<td>1.039</td>
<td>1.067</td>
</tr>
<tr>
<td>elliptic</td>
<td>2.28</td>
<td>2.38</td>
<td>2.42</td>
<td>6.90</td>
<td>2.46</td>
<td>1.045</td>
<td>1.065</td>
</tr>
<tr>
<td>ex1010</td>
<td>0.22</td>
<td>0.23</td>
<td>0.23</td>
<td>0.39</td>
<td>0.20</td>
<td>1.038</td>
<td>1.060</td>
</tr>
<tr>
<td>ex5p</td>
<td>0.05</td>
<td>0.06</td>
<td>0.06</td>
<td>0.08</td>
<td>0.07</td>
<td>1.025</td>
<td>1.057</td>
</tr>
<tr>
<td>frisc</td>
<td>3.27</td>
<td>3.44</td>
<td>3.51</td>
<td>9.04</td>
<td>3.50</td>
<td>1.055</td>
<td>1.074</td>
</tr>
<tr>
<td>misex3</td>
<td>1.14</td>
<td>1.18</td>
<td>1.20</td>
<td>2.06</td>
<td>1.19</td>
<td>1.039</td>
<td>1.054</td>
</tr>
<tr>
<td>plc</td>
<td>5.36</td>
<td>5.65</td>
<td>5.72</td>
<td>12.54</td>
<td>6.18</td>
<td>1.055</td>
<td>1.068</td>
</tr>
<tr>
<td>s298</td>
<td>0.47</td>
<td>0.49</td>
<td>0.50</td>
<td>0.84</td>
<td>0.46</td>
<td>1.045</td>
<td>1.058</td>
</tr>
<tr>
<td>s38417</td>
<td>2.59</td>
<td>2.73</td>
<td>2.78</td>
<td>16.34</td>
<td>3.43</td>
<td>1.055</td>
<td>1.074</td>
</tr>
<tr>
<td>s38584</td>
<td>3.11</td>
<td>3.28</td>
<td>3.34</td>
<td>21.55</td>
<td>3.79</td>
<td>1.056</td>
<td>1.076</td>
</tr>
<tr>
<td>seq</td>
<td>1.83</td>
<td>1.91</td>
<td>1.94</td>
<td>3.39</td>
<td>2.00</td>
<td>1.045</td>
<td>1.063</td>
</tr>
<tr>
<td>spla</td>
<td>3.47</td>
<td>3.66</td>
<td>3.71</td>
<td>8.07</td>
<td>3.52</td>
<td>1.054</td>
<td>1.070</td>
</tr>
<tr>
<td>tseng</td>
<td>0.41</td>
<td>0.42</td>
<td>0.43</td>
<td>1.01</td>
<td>0.47</td>
<td>1.038</td>
<td>1.066</td>
</tr>
<tr>
<td>geo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.045</td>
<td>1.064</td>
</tr>
</tbody>
</table>

Scheme 1 and the unmodified version of VPR using the Valgrind instrumentation framework\textsuperscript{4} [116]. Scheme 1 showed an average increase in data cache miss rate of 0.2% and conclude that any change in cache performance is negligible.

4.6.2.2 Additional Code Overhead

As concluded from Section 4.6.2.1, all of the overhead in Scheme 1 is for determining when to allocate memory. As shown in Table 4.7 this increases the run-time by 4.5%.

In Scheme 2, there is an additional overhead of another 2%; this is the time required to perform memory deallocation once after each routing iteration. The circuits in this suite required 25-30 routing iterations to complete. The run-time overhead in Scheme

\textsuperscript{4}We used the cachegrind tool within version 3.6.1 of the Valgrind framework and measured cache performance of data read and writes.

93
3 is very significant (144%). This is because we perform a memory deallocation pass after each net is routed. For example, ex5p has the fewest number of nets (94 nets), and hence has the smallest increase in run-time (1.48X), whereas s38584 has the most nets (3000 nets) and has the largest increase in run-time (6.94X). As expected, the run-time of the design-adaptive algorithm is more modest (10.4%) but achieves memory reductions comparable to that of Scheme 3.

4.7 Summary

The chapter described the second contribution of this dissertation where the focus was on the memory-requirement scalability of FPGA CAD tools. Although software memory usage can be very implementation-specific, this research is independent of the algorithm implementation details, and can apply to a wide range of FPGA routing algorithms.

We first described a novel scheme to represent the FPGA’s programmable resources. A routing algorithm needs this representation so it can determine what FPGA resources are available to use in order to implement a design’s nets. By leveraging the regularity of the FPGA architecture, the proposed representation does not need to explicitly model every programmable resource. To make this idea concrete, we implemented this scheme into VPR, and showed that the memory usage can be reduced by over an order of magnitude (5x-13x) for modern FPGAs. More importantly, we showed that the rate at which the memory requirements scaled was reduced. Due to extra steps carried out by the router to facilitate the new representation, the routing run-time and overall place-and-route run-time increased on average by 2.26x and 1.28x respectively.

Aside from information used to describe the architecture, storage is required for each programmable resource to track various algorithm specific metrics used during routing. We made several observations on when storage for this information is required for each
programmable resource. Based on these observations, we proposed three simple dynamic memory management schemes that varied in their aggressiveness to reduce memory usage. A design-adaptive scheme was also described. We quantified the memory and run-time trade-offs of each scheme, as well as the bounds on achievable memory reductions using an experimental methodology. The simple schemes were able to further reduce memory usage in high-density mappings by 24%, 34%, and 43% while incurring an additional routing run-time increase of 4.5%, 6.5%, and 144% respectively. The design-adaptive scheme achieved memory reductions comparable to that of the most aggressive simple scheme (41%) while incurring only a fraction of the run-time overhead (10.4%). For low-density mappings, memory reductions of 33%, 50%, 62% and 59% could be achieved for the three simple and design-adaptive schemes, respectively.

This chapter has presented an important step towards tackling the issue of increasing memory requirements in FPGA CAD tools. Code optimizations may delay the problem somewhat, but a long-term solution requires significant changes in CAD algorithms or FPGA architectures as shown here. We have made the memory-efficient version of VPR available at http://www.ece.ubc.ca/~scottc/downloads.html for public use.
Chapter 5

Towards Scalable CAD Run-time Through Architecture

This chapter focuses on the run-time scalability of FPGA CAD tools. As reviewed in Chapter 2, reducing the run-time of FPGA CAD tools has long garnered much interest. However, almost all of the focus has been on improving the algorithms. While this must remain an important approach, this chapter investigates the design of FPGA architectures with CAD run-time as an optimization metric. There is very limited research in this direction, and the aim of this chapter is to motivate more attention towards this approach. This is a divergence from the traditional architecture design process where area, delay, and power are optimized, and then the CAD algorithms are optimized for run-time once the architecture is fixed.

We have identified several possible approaches to designing such an architecture, which will be discussed as future work in the next chapter. In this chapter, we investigate one such approach: *Architectures that lead to a reduced problem size in the com-
Computationally expensive stages of the FPGA CAD flow. The FPGA CAD flow contains a number of stages as shown in Figure 2.4 (Chapter 2). Some stages, such as placement and routing, take much longer than others. As suggested by our model in Chapter 3, it should be feasible to modify the architecture so that the problem size presented to the computationally expensive stage(s) is reduced, thus directly reducing the run-time of these stages.

As a concrete example, this chapter presents an architecture study focused on architected high-capacity logic blocks to reduce CAD run-time. Intuitively, increasing the logic capacity of each Logic Block has three effects on CAD run-time:

1. The number of logic blocks needed to implement the user circuit is reduced, thus directly reducing the placement problem size and run-time.

2. The number of nets that need to be routed outside of the logic blocks is reduced, thus directly reducing the routing problem size and run-time.

3. Increasing the LB size may increase the run-time of upstream CAD stages, such as clustering.

On the other hand, the performance (area and delay) of the FPGA may also be affected. We look at two approaches to building high-capacity logic blocks (LB): directly scaling traditional LB architectures, and using a multi-level LB architecture. We explore LB sizes well beyond the range looked at in previous studies and approach this study from the new perspective of CAD run-time. To aid in the LB design, we describe a new analytical relationship for determining the number of inputs per LB which accounts for interconnect demand. In addition, we compare two different LB packing algorithms to show that reducing CAD run-time through architectural changes and algo-
rithmic changes can be complementary. Finally, we compare the results of this work to that of four previously published algorithmic approaches to trading off quality for CAD run-time speedups.

This chapter is organized as follows. Section 5.1 describes the architectures that are investigated in this study as well as the equation used to determine the amount of I/O needed by a logic block. Section 5.2 describes the experimental methodology. Section 5.3 and 5.4 present the experimental results. Section 5.5 performs the comparison of results against related works. Parts of this chapter have been published in [37]

5.1 Architecting High-Capacity Logic Blocks

We considered two ways to architect high-capacity logic blocks. The first approach is to directly increase $N$ in the traditional logic block architecture that we have assumed throughout this dissertation (reviewed in Chapter 2). We refer to this architecture as flat. The second approach uses a partly hierarchical construction with two levels of hierarchy. We refer to this as multi-level. We do not consider more than two levels of hierarchy.

5.1.1 Flat Logic Block Architecture

The flat LB architecture was described in Section 2.1.1 (Chapter 2) and is shown again in Figure 5.1a. We use $K$, $N$, and $I$ to denote the LUT size, number of LEs per LB, and number of LB inputs, respectively. Similar to previous studies, we assume a fully-connected local interconnect topology as shown in Figure 5.2, each LE can be programmably connected to any of the LB input pins or LE output pins. This is consistent with previous studies [5], and the assumptions in the VPR place-and-route CAD suite that were used in the experiments.

To make the area results more realistic, we use the local interconnect architecture
Figure 5.1: Two logic block architectures investigated

Figure 5.2: Logic block local interconnect architecture used to route the I LB inputs and N BLE outputs to any of the K · N BLE inputs described recently in [129]. Full logical connectivity is maintained but the number of inputs to each of the multiplexor is reduced from $(I + N)$ to $(I + N - K + 1)$, thus reducing the number of required transistors. This is possible because the inputs of a LE are logically equivalent. Although other forms of depopulation have been described in [74], we do not apply them in this study because they may incur a routing run-time overhead.
5.1.2 Multi-Level Logic Block Architecture

This architecture consists of two levels. The first level is the same as the flat architecture from the previous sub-section. We use $K$, $N_1$, and $I_1$ to denote the LUT size, number of LEs per LB, and number of inputs available to the first level. The second level consists of a second interconnect with the same topology as the first level, and $N_2$ first-level blocks. The number of inputs available at the second level is parameterized as $I_2$. Figure 5.1b shows this architecture. This LB architecture fits well into an island-style architecture and does not require any changes to the global routing architecture or CAD tools.

5.1.3 Logic Block I/O and Interconnect Demand

Determining the appropriate number of inputs ($I$, $I_1$, and $I_2$) for the LB is critical to the area and packing efficiency of the architecture. Too few inputs (input-limited) forces LEs to go unused, and too many leads to an unnecessarily large silicon area due to the size of the local interconnect. The total number of LE inputs within a LB is $K \cdot N$. However, due to shared and feedback connections (which are implemented by connections already inside the LB) in the circuits being implemented, LBs typically do not need the maximum of $K \cdot N$ unique inputs.

Betz [19] proposed the formula in Equation 5.1 for choosing the number of LB inputs, and found that if an architecture was constructed according to this formula, 98% of BLEs would be used on average. This formula was based on experimental data with LUTs of size $K = 4$, and LBs in the range of $N = 1$ to $N = 16$. Ahmed [5] later generalized this relationship to a form, shown in Equation 5.2, that includes LUT size. This generalized formula was also based on experimental data using architectures with LUT sizes in the range of $K = 2$ to $K = 7$, and LB sizes in the range of $N = 1$ to $N = 10$. This equation has since become a well-accepted rule of thumb in academic studies.
\[ I = 2 \cdot (N + 1) \quad (5.1) \]

\[ I = \frac{K}{2} (N + 1) \quad (5.2) \]

We found that this relationship becomes less suitable when scaling to larger LB sizes. The main shortcoming is that it does not directly account for the interconnect demand of circuits being implemented on the FPGA. The end result is a higher than necessary number of input pins per LB for large LBs. Using the analytical Lam/Wilton FPGA logic density model [69], we propose a more suitable relationship that accounts for interconnect demand of the circuits being implemented on the FPGA.

### 5.1.3.1 Equation Derivation

Starting with the same goal of Equation 5.2, we aim to determine a value \( I \) such that 98% of the BLEs are utilized on average. Let us denote the number of \( K \) input LUTs required to implement a circuit as \( n_k \). We then denote the number of LBs needed to cover these LUTs as \( n_c \). Each LB contains \( N \) LUTs so the total number of programmable LUTs available across all LBs is \( n_c \cdot N \). If 98% of all available LUTs are used to implement the circuit, we can then form the following equality:

\[ n_k = 0.98 \cdot n_c \cdot N \quad (5.3) \]

We then use Equation 5.4 directly from [69].\(^1\) This equation models the number of LBs needed to implement a circuit that has been technology mapped to \( n_k \) LUTs.

\(^1\)The model in [69] contains two equations for \( n_c \) depending on whether the logic architecture leads to N-Limited, or I-Limited packing. In our scenario, we state that only 98% of BLEs are used because \( I \) is chosen in a way to limit full occupancy. Therefore, we use the I-Limited equation.
implemented; \( p \) and \( f \) denote the Rent’s Exponent and average fanout of the circuit, respectively. \( \gamma \) is a model parameter of the Lam/Wilton model that describes the average number of unused LUT input pins. The full derivation of Equation 5.4 can be found in [69], and its accuracy has been validated against experimental results.

\[
n_c = n_k \cdot \frac{K + 1 - \gamma}{I \cdot (1 + \frac{1}{f})}
\]  

(5.4)

Substituting Equation 5.4 into Equation 5.3 and solving for \( I \) yields Equation 5.5.

We can see that this has similar form to the “rule-of-thumb” equation previously shown in Equation 5.2. However, there are now interconnect-related terms, \( p \), and \( f \).

\[
I = (0.98 \cdot N)^p \cdot \frac{K + 1 - \gamma}{1 + \frac{1}{f}}
\]  

(5.5)

We can interpret the Rent’s exponent \( p \) in Equation 5.5 as the Rent’s exponent of the architecture. In other words, it describes the architecture’s interconnect supply. From an FPGA architect’s point of view, this would need to be high enough to support the interconnect demand of user circuits that would be implemented on the FPGA. As a reference point, previous studies have quantified the architecture Rent exponent of the Altera Cyclone [7] architecture and Altera Apex [6] architecture to be 0.7256 [94], and 0.78 [63], respectively. Those studies also quantified the interconnect demand of 77 industrial circuits and found the Rent parameter to range from 0.5-0.8 with an average of 0.6. This is consistent with the benchmark circuits (described later in Section 5.2.3) that are used in this study.
5.1.3.2 Equation Validation

To validate the equation, we compared the values that it suggests to experimental results, as well as to the traditional rule-of-thumb. To acquire experimental results, we first mapped the benchmark circuits summarized in Table 5.1 to LUTs using Flowmap [40]. Next, we mapped the netlist of LUTs to LBs of increasing size $N$ using the timing-driven clustering algorithm, TV-Pack [86]. For each LB size, we varied the number of inputs to the LB until the average number of LEs used in each LB reached 98%. We repeated the experiments using a second clustering algorithm, iRac [101] aimed at minimizing the number of global nets. Each data point is the average across the benchmark suite. We use the average Rent’s parameter and average fanout of the benchmark suite. Figures 5.3a and 5.3b show validation data for LUT sizes of $K = 4$ and $K = 6$ respectively. Additional results in the range of $K = 2$ to 7 can be found in Appendix C.

In the range of $N = 2$ to 10 (the range for which the rule-of-thumb was derived [5]), both equations produce similar values, and the rule-of-thumb works quite well. However, beyond $N = 10$, the figures show that the proposed equation is more suitable. Hence, for the remainder of this chapter, we will use Equation 5.5 to determine the number of input pins per logic block. For the multi-level architecture, we determine $I_1$ by substituting $N_1$ into the equation, and we determine $I_2$ by substituting in the effective logic block size $N_2 \cdot N_1$ (this is equivalent to the amount of logic within the LB).

5.2 Experimental Methodology

In the next section, we will experimentally quantify the CAD run-time, area, and delay trade-offs for the architectures described in Section 5.1. This section will describe

---

2The original implementation described in [101] was not publicly available. We used a version previously implemented by Marvin Tom at the University of British Columbia.
the experimental methodology including architectural and circuit assumptions, area and delay models, CAD tools and benchmarks.

5.2.1 Architectural Assumptions

We assume a homogeneous FPGA. The routing architecture is based on single-driver directional routing [75] which is consistent with VPR 5.0 and commercial devices. Each LB output pin drives an isolation buffer which then drives $F_{\text{Cout}} \cdot W$ SB routing multiplexors, where $W$ is the number of routing tracks per channel and $F_{\text{Cout}}$ is the percentage of the $W$ tracks to which the output pin can connect. Wire segments connect to LB inputs through an isolation buffer which then drives $F_{\text{Cin}} \cdot (2 \cdot I/4)$ input pins, where $F_{\text{Cin}}$ is the fraction of wire segments in the routing channel that can drive a given input pin. We assume that LB pins are evenly distributed along all four sides of the LB. Therefore, there are $I/4$ input pins on each side, and two LBs surrounding each channel. Wire segments entering a SB can connect to $F_s$ outgoing wire segments.

In our experiments, we set $F_{\text{Cout}} = 0.1$, $F_{\text{Cin}} = 0.15$, $F_s = 3$ and assume that wire

Figure 5.3: Validation of equation to determine number of logic block inputs for 98% logic utilization.
segments span a single LB. The channel width $W$ is set to the minimum that still enables the circuit to be routed. This is determined independently for each circuit and each set of architecture parameter values. Similarly, the FPGA grid size is set just large enough to accommodate all of the required LBs for each circuit. The number of inputs per LB is chosen based on an architecture Rent’s parameter of $p = 0.8$.

I/O blocks are found on the periphery of the FPGA. Each I/O block contains a number of pads that allow for off-chip communication. To maintain a similar I/O Pad to I/O Block density across all architectures, we set the number of pads per block according to Equation 5.6 [86].

$$PadsPerIOblock = \lceil 2 \cdot \sqrt{N_{eff}} \rceil$$ (5.6)

$N_{eff}$ describes the number of LEs inside the LB. $N_{eff} = N$ for the flat architecture, and $N_{eff} = N_2 \cdot N_1$ for the multi-level architecture.

### 5.2.2 Area Model, Delay Model and Circuit Design

We use an area model based on counting the number of minimum-width transistor areas (MWTA) needed to implement the architecture. In this area model, SRAM cells are implemented with six minimum-size transistors. LUTs are implemented as full binary trees of NMOS pass-transistors and all pass transistors are minimum width. Multiplexors are implemented as two-stage pass transistor trees, which is consistent with VPR’s assumptions. Routing SBOX drivers are implemented as three stage buffers similar to [75]. All other buffers are two-stage. Buffers are sized to minimize delay using HSPICE. We use circuit-level designs similar to those described in [62]. We assume a 45nm process throughout and use the Predictive Technology Model [131] to acquire process information for HSPICE.
As the size of a LB increases, the physical distance between LBs in the FPGA grid also increases. This directly affects the physical length (as illustrated in Figure 5.4) of the routing wire segments which in turn affects the SB driver size. This effect is taken into account during the sizing of the SB drivers. We first size all drivers in the architecture except for the wire driver. These sizings, along with an initial guess of the SB driver size, are then supplied to the area model to obtain a MTWA count for a single tile in the FPGA. We then convert this count to a physical area assuming a 60% layout density [19]. This allows us to estimate the physical wire length and size the SB driver. The process is repeated using the new SB driver size until the SB driver size converges. This iterative flow is summarized in Figure 5.5.

5.2.3 CAD and Benchmarks

We use a typical academic CAD flow similar to those used in previous LB architecture studies [5, 86]. First, benchmark circuits are technology mapped to LUTs using Flowmap [40]. We then use a timing-driven packing algorithm T-VPack [86] to pack LUTs and registers into logic blocks. Placement and routing is performed using VPR 5.0 [82]. For comparison, we repeat the experiments using a second packing algorithm iRac aimed at minimizing the number of global nets.
In the multi-level cluster architecture experiments, we apply two iterations of the T-VPack algorithm to pack a circuit. Although multi-level packing algorithms have been proposed [41, 46], these more sophisticated algorithms would likely incur additional run-time. We first wanted to investigate the area and delay trade-offs under more optimistic CAD run-time conditions. In addition, implementations of these algorithms were not readily available. To perform timing analysis during place and route for the multi-level LBs, we implemented straight-forward extensions to the timing-graph construction in VPR’s code. The core timing analysis algorithms were unchanged.

Run-time measurements were performed on Xeon 2.6GHz processors. When quantifying the routing run-time, we only measure the time needed to route to the minimum channel-width architecture (as opposed to measuring the entire binary-search routing process needed to determine the minimum channel width). Placement and routing run-time excludes the time needed for memory allocation of data structures. Clustering time for the multi-level architectures include both iterations for the first and second levels.

The random nature of the simulated annealing placement algorithm inserts some
Table 5.1: Synthetic benchmark circuits

<table>
<thead>
<tr>
<th>Circuit</th>
<th>4-LUTs</th>
<th>Nets</th>
<th>Rent</th>
<th>Avg. Fanout</th>
</tr>
</thead>
<tbody>
<tr>
<td>synth01</td>
<td>15818</td>
<td>38047</td>
<td>0.654</td>
<td>3.30</td>
</tr>
<tr>
<td>synth02</td>
<td>8831</td>
<td>21288</td>
<td>0.658</td>
<td>3.38</td>
</tr>
<tr>
<td>synth03</td>
<td>10906</td>
<td>26598</td>
<td>0.641</td>
<td>3.51</td>
</tr>
<tr>
<td>synth04</td>
<td>20007</td>
<td>48460</td>
<td>0.631</td>
<td>3.46</td>
</tr>
<tr>
<td>synth05</td>
<td>18115</td>
<td>42917</td>
<td>0.609</td>
<td>3.40</td>
</tr>
<tr>
<td>synth06</td>
<td>11786</td>
<td>28300</td>
<td>0.657</td>
<td>3.47</td>
</tr>
<tr>
<td>synth07</td>
<td>26461</td>
<td>63934</td>
<td>0.648</td>
<td>3.44</td>
</tr>
<tr>
<td>synth08</td>
<td>20961</td>
<td>50008</td>
<td>0.646</td>
<td>3.44</td>
</tr>
<tr>
<td>synth09</td>
<td>12085</td>
<td>29020</td>
<td>0.659</td>
<td>3.36</td>
</tr>
<tr>
<td>synth10</td>
<td>10009</td>
<td>24014</td>
<td>0.648</td>
<td>3.38</td>
</tr>
<tr>
<td>synth11</td>
<td>16867</td>
<td>38661</td>
<td>0.605</td>
<td>3.44</td>
</tr>
<tr>
<td>synth12</td>
<td>13131</td>
<td>31161</td>
<td>0.692</td>
<td>3.33</td>
</tr>
<tr>
<td>synth13</td>
<td>3160</td>
<td>7973</td>
<td>0.648</td>
<td>3.38</td>
</tr>
<tr>
<td>synth14</td>
<td>12405</td>
<td>31085</td>
<td>0.689</td>
<td>3.14</td>
</tr>
<tr>
<td>Avg.</td>
<td></td>
<td></td>
<td>0.649</td>
<td>3.39</td>
</tr>
</tbody>
</table>

noise in the measured results (area, critical path delay, and CAD run-time). To reduce the effects of this noise, all experiments were performed three times with different seed values to initialize the placement algorithm’s random number generator. Each data point in the following set of experiments is an average of the three trials.

We used the synthetic benchmark circuits summarized in Table 5.1 because the circuits in the traditional MCNC benchmark suite are too small for this study. The synthetic benchmarks mimic modern system-level designs and were generated using the circuit generator published in [85]. We use the MCNC suite of circuits to form the IP library needed by the circuit generation tool.

5.3 Experimental Results: Flat Logic Block Architectures

Figure 5.7 shows the impact of LB size on area, delay, and CAD runtime for the traditional flat architecture. Each data point is the average over the entire benchmark suite. For area and delay, both logic and routing components are shown along with the total.
Logic refers to the component incurred from the LBs, and routing refers to the component incurred by the global routing resources. A breakdown of clustering, placement, and routing components are shown for CAD run-time. Results from using both TV-Pack and iRac packing tools are shown.

5.3.1 Area

Three competing trends arise as $N$ increases that impact the total FPGA area:

1. The local interconnect within each LB increases in complexity and size, thus increasing the area of each LB. The logic area per tile is shown in Figure 5.6a.

2. The minimum-required channel width increases (shown in Figure 5.8a), thus increasing the routing area in each FPGA tile. This occurs because the number of signals entering a block increases as observed by Rent’s Rule. The routing area per tile is shown in Figure 5.6a.

3. The number of LBs, and hence number of FPGA tiles, required to implement the circuit decreases. This trend is shown in Figure 5.6b.

Figure 5.7a and 5.7b show the overall impact on area. Trends 1 and 3 lead to an overall roughly linear increase in logic block area as $N$ increases. Trends 2 and 3 lead to an overall decrease in routing area as $N$ increases. The total area has a minimum region between $N = 10$ and $N = 16$. Results obtained using the iRac clustering algorithm have a similar trend but leads to lower total area for all values of $N$. This is expected as iRac enables a smaller minimum-required channel width (highlighted in Figure 5.8a), leading to a lower routing area. These observations on area are consistent with previous studies ([5] investigated up to a size of $N = 10$ for a 180nm process and [19] investigated up to a size of $N = 20$ for a 350nm process).
5.3.2 Delay

The impact of LB size, \( N \), on the critical path delay of a circuit implemented on a flat architecture is shown in Figure 5.7c and Figure 5.7d when using TV-Pack and iRac, respectively. There are three competing trends that impact the critical path delay:

1. The delay through a single LB increases as the local interconnect grows in complexity; more transistors are used to implement the interconnect leading to more capacitance.

2. The number of nets implemented in the local interconnect within an LB increases. This leads to an increasing portion of the critical path delay being due to the local LB interconnect and LEs.

3. At the same time, a decreasing amount of the critical path delay will be due to the global routing interconnect.
Figure 5.7: Impact of logic block size on area, delay, and CAD run-time (flat architecture)
Trends 1 and 2 lead to an overall increase in logic delay (delay from the portion of the critical path implemented within LBs). Trend 3 results in an overall decrease in routing delay (delay from the portion of the critical path implemented in the global routing network). The total critical path delay decreases as the LB size increases. We would expect the critical path delay to eventually increase again as the local interconnect becomes very large and slow (consider the extreme case where the entire FPGA is a single LB). However, an interesting observation is that this does not occur even for sizes of $N = 50$.

As expected, the timing-driven TV-Pack flow leads to lower overall critical path delay for all LB sizes compared to the iRac flow; TV-Pack packs more timing critical nets into the relatively faster local interconnect. Figure 5.8d shows a comparison of the number of global nets on the critical path when using TV-Pack and iRac.

5.3.3 CAD Run-Time

The logic block size has a significant impact on placement and routing runtime as shown in Figure 5.7e and 5.7f. When the packing efficiency of the architecture is not constrained by the number of LB inputs (such as the case in this study since we appropriately chose the number of LB inputs), the number of LBs required to implement the user circuit is proportional to $1/N$. Since the placement problem size is proportional to the number of required LBs, we see an inversely proportional trend in placement run-time.

The routing run-time also decreases as fewer nets need to be routed using the global routing network (Figure 5.8b shows the number of global nets as a function of LB size).

Furthermore, the choice of clustering algorithm affects the placement and routing run-times. Figure 5.9a shows a comparison of placement run-time when using TV-Pack and iRac. iRac packing solutions lead to faster placement run-times for all LB sizes.
As shown by our model in Chapter 3, the placement run-time depends on the average number of nets connected to an LB; when there are fewer nets, less time is required to evaluate the cost of a potential placement swap. Since iRac packing solutions have fewer global nets compared to TV-Pack solutions, using iRac leads to faster placement run-times. Using iRac also leads to faster routing run-times for two reasons. First, since there are fewer global nets to route, each routing iteration becomes faster. Secondly, iRac requires a smaller routing channel-width than TV-Pack, as discussed earlier, leading to fewer neighbours per routing resource.

This observation is an example of how an algorithmic change to reduce CAD run-time (using a routability-driven packing algorithm versus a purely timing-driven algorithm) can be complementary to an architectural change to reduce CAD run-time (using high-capacity logic blocks).

5.3.4 Trade-offs Between CAD Run-time and Quality of Results

In this section, we discuss the trade-offs between CAD run-time and the quality of results when varying the logic block size within the flat LB architecture. Figure 5.10a shows the CAD run-time versus area trade-offs. Each data point corresponds to a specific LB size $N$, and is the average of results across the benchmark suite. Area and run-time values have been normalized to those obtained for the $N = 14$ architecture when using the TV-Pack flow. $N = 14$ serves as a suitable baseline because it is within the minimum area range shown in Figure 5.7a and 5.7b and this is roughly the value used in commercial devices. Trade-off curves for results obtained by both the TV-Pack and iRac flows are shown.

Based on these trade-off curves, there is no incentive to use any LB architectures with $N < 14$. Doing so would increase both area and CAD run-time. For LB sizes of
Figure 5.8: Impact of logic block size on various mapping properties

For $N \geq 14$, there is a clear trade-off between area and CAD run-time. Figure 5.10b presents a close-up of this region. Exposing this relationship gives architects a trade-off that can be employed when tackling CAD run-time. For example, when using the timing-driven TV-Pack flow, LBs with $N = 22$ give a 25% reduction in CAD run-time at the expense of 5% area increase when compared to the baseline LB architecture of $N = 14$.

The trade-off curves obtained from both the TV-Pack and iRac flows have a similar shape. However, since iRac produces more area-efficient mappings (as discussed in Section 5.3.1), its curve is shifted downwards (lower area) compared to the TV-Pack curve.
And since iRac also produces solutions that are faster to place and route (as discussed in Section 5.3.3), its curve is also shifted to the left (smaller run-times) compared to the TV-Pack results.

Figure 5.10c shows the trade-off between CAD run-time and critical path delay. According to these trade-off curves, there is no negative impact to using the largest LB size. The trade-off curves obtained using the TV-Pack and iRac flow have similar shapes and yield the same conclusion. However, as TV-Pack produces mappings with lower critical path delays, the trade-off curve corresponding to the TV-Pack results is shifted downwards compared to the trade-off curve corresponding to the iRac results.

Figure 5.10d presents a close up of the region for $N \geq 14$. Earlier in this section, we looked at the example of using an LB with $N = 22$ compared to the baseline of $N = 14$, and showed a 25% decrease in CAD run-time with a 5% area increase. Figure 5.10d shows that when delay is also considered, LBs with $N = 22$ also achieves a 10% reduction in critical path delay.

Overall, there are clear trade-offs between CAD run-time and the quality of results.
(area and delay) when changing the size of the LBs. In general, this approach can reduce CAD run-time and critical path delay at the expense of area. This helps motivate the investigation of the multi-level LB architecture to determine whether it leads to better area results.
5.4 Experimental Results: Multi-Level Logic Block Architectures

In this set of experiments, we considered three families of the multi-level LB architecture, each with a different fixed number of level-1 blocks \((N_2 = 2, 4, \text{ and } 6)\). The size of each level-1 block, \(N_1\), was then varied within each family to vary the overall logic block size. For comparison to the flat architecture, we use the term effective logic block size, \(N_{\text{eff}} = N_2 \cdot N_1\) when describing the size of a multi-level logic block. This corresponds to the total number of LEs within the LB. \(N_{\text{eff}}\) is thus analogous to \(N\) in the flat architecture in that they both describe an LB with the same amount of logic capacity (number of LEs), and hence also the same number of LB inputs (since the number of inputs per LB is set according to the analytical equation described in Section 5.1.3 in both cases).

5.4.1 Area

Figures 5.11a, 5.11b, and 5.12a show the logic, routing, and total area for different effective LB sizes. Results from the flat architecture are superimposed for comparison. The choice of packing algorithm does not impact logic area as both algorithms are able to fully pack a majority of the LBs. The family with \(N_2 = 2\) requires more logic area for all investigated sizes compared to the flat architecture. For the \(N_2 = 4\) and \(N_2 = 6\) families, the multi-level architecture requires a smaller logic area beyond \(N_{\text{eff}} = 25\). The cause for these differences is in the number of transistors needed to implement the local interconnect of the LBs. For the flat architecture, the local interconnect in an LB is a single crossbar; the local interconnect in a multi-level LB consists of two smaller crossbars (one for each level). The number of transistors needed to implement the multi-level crossbars increases less quickly with LB size compared to the flat architecture’s crossbar with an equivalent size. However, there is an area overhead to having two
crossbars. For large enough logic blocks \(N_{\text{eff}} \geq 25\), the overhead of having the two crossbars is offset by the better scaling.

When comparing routing area under the TV-Pack flow, the multi-level architectures incur a larger routing area than the flat architecture for all LB sizes. When using the iRac flow, the multi-level architecture achieves a lower routing area beyond \(N_{\text{eff}} = 26, 28,\) and 42 for the \(N_2=2, 4,\) and 6 families, respectively. This corresponds to the points at which the flat architecture’s minimum required channel-width exceeds that of the multi-level architectures (channel width as a function of LB size is shown in Figure 5.8a).

These factors give the multi-level architecture better overall area-efficiency beyond \(N_{\text{eff}} = 40\) for both \(N_2=4\) and 6 when using TV-Pack. When using iRac, this occurs at \(N_{\text{eff}}=40, 32,\) and 36 for the \(N_2=2, 4,\) and 6 families, respectively.

### 5.4.2 Delay

Figure 5.11c, 5.11d and 5.12b show the logic, routing, and total critical path delay for different effective LB sizes. Results from the flat architecture are superimposed for comparison. The routing delay for the multi-level architecture is worse for all families and LB sizes compared to the flat architecture regardless of packing algorithm. The reason is that the multi-level architecture absorbs fewer critical path nets so more of the critical path is implemented using the global routing resources. This is highlighted in Figure 5.8d which shows the number of global nets on the critical path as a function of LB size.

The logic delay for the multi-level architecture is worse for all configurations investigated. This is due to the two levels of local interconnect. Consider the scenario where the output of an LE drives the input of an LE within the same LB, but in different level-1 blocks. This scenario is illustrated in Figure 5.13. This signal would need to
Figure 5.11: Comparison between flat and multi-level architectures for various components of area, delay and CAD run-time.
Figure 5.12: Comparison of total area, total delay and total CAD run-time across all investigated flat and multi-level architecture families.

pass through two levels of interconnect. For a flat block of the same effective size, there is only one level of interconnect. Since each level of interconnect consists of two-stage multiplexors, the signal in the multi-level scenario would pass through more transistors (two two-stage multiplexors) and be slower. We investigated the use of different multiplexer implementation styles such as one-hot encoding to see whether we could mitigate this effect in the multi-level local interconnect. However, our results showed that the two-stage multiplexer scheme still led to the best delay performance. Overall,
Figure 5.13: Example scenario for extra delays through multiplexors for a local connection within a multi-level logic block.

the multi-level architecture is less speed-efficient compared to the flat architecture.

### 5.4.3 CAD Run-time

Figure 5.11c, 5.11f and 5.12c show the placement, routing, and total CAD run-time for the multi-level architectures. These figures show that the multi-level LB architectures require longer placement and routing run-times compared to the flat architecture. The reason for this is that there are more global nets when employing the multi-level logic blocks as shown in Figure 5.8b. Recall from the earlier discussion in this chapter that more global nets means it takes more time to evaluate a potential placement swap (longer placement time), and means there are more nets to route (longer route time). The cause of more global nets is in the greedy nature of the two-step packing. When packing the first level, consideration is only given to nets that can be completely absorbed into the level-1 blocks. This makes it more challenging for the second iteration to completely absorb nets.
Figure 5.14: Trade-offs between CAD run-time and quality of results when employing different logic block sizes (All architectures)

### 5.4.4 Trade-offs Between CAD Run-time and Quality of Results

This section discusses the trade-offs between CAD run-time and quality of results when varying the logic block size within the multi-level architecture. Figures 5.14a to 5.14d illustrate these trade-offs. Results from the flat architecture (Figures 5.10a to 5.10d) are super-imposed for comparison. As before, each data point corresponds to a specific effective LB size $N_{\text{eff}}$. Area, delay and CAD run-time values have been normalized to those obtained with the $N = 14$ flat architecture when using the TV-Pack flow.
Figure 5.14b shows that the increase in area as $N_{ef}$ increases (moving from right to left) is not as steep as the flat architecture because the multi-level architecture becomes more area-efficient for larger LB sizes. However, since the multi-level architecture leads to longer CAD run-times compared to the flat architecture (discussed in Section 5.4.3), the multi-level architecture’s trade-off curves are shifted to the right compared to the flat architecture’s curves. The summary of this is that although the multi-level architectures can be more area-efficient for a specific logic block size, the flat architecture can produce better CAD run-time reductions for a specific amount of area increase by choosing an appropriate LB size along the flat architecture trade-off curve.

Figure 5.14d illustrates a similar conclusion for critical path delay. The multi-level architecture leads to a higher critical path delays in all cases and longer CAD run-times. Hence, for a specific amount of desired CAD run-time reduction, the flat architecture can achieve a better delay.

### 5.5 Comparison to Other Published Work

Before concluding this chapter, we compare the trade-offs observed in this architecture study against four other relevant studies. These four studies were chosen because they investigated similar trade-offs but through algorithmic changes. Table 5.2 provides a summary of these comparisons. All values are with respect to results obtained using VPR [19]. In all cases except VPR -fast (more on this later), results are presented as reported in the original studies.

For the study by Mulpuri and Hauck, VPR -fast, and our work, the combined place-and-route run-time speedups, and post-routing results are presented. For Ultra-fast VPR and the parallel placer by Wang and Lemieux, only placement run-time, and pre-routing results (such as bounding-box cost and pre-routing estimated critical path delay) are
Table 5.2: Comparison of results against related work. Note that Ultra-fast VPR and the parallel placer report only placement run-time speedup whereas all others report combined place-and-route run-time speedup. A “-” means the metric was not reported in the original study.

<table>
<thead>
<tr>
<th>Reference</th>
<th>CAD Speedup</th>
<th>Delay</th>
<th>Area</th>
<th>BB Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>This work</td>
<td>2.2x, 1.34x</td>
<td>0.9x</td>
<td>1.35x</td>
<td>-</td>
</tr>
<tr>
<td>Mulpuri and Hauck [92]</td>
<td>5x, 2.2x</td>
<td>2.5x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>VPR -fast [19]</td>
<td>2.5x, 1.8x</td>
<td>1.01x</td>
<td>1.03x</td>
<td>-</td>
</tr>
<tr>
<td>Ultra-fast VPR [97]</td>
<td>75x, 7x</td>
<td>-</td>
<td>-</td>
<td>1.8x</td>
</tr>
<tr>
<td>Parallel Placer [120]</td>
<td>123x</td>
<td>1.08x</td>
<td>-</td>
<td>1.11x</td>
</tr>
</tbody>
</table>

reported. It is important to keep this difference in mind because a speed-up in placement achieved at the expense of placement cost typically leads to a slow-down in routing. We describe each work in more detail below.

First, for our work, we cite two data points from the trade-off curves of the flat architecture using the TV-Pack flow. At one extreme, $N = 50$, we could achieve a 2.2x speedup in CAD run-time, and a 0.9x delay (an improvement in the FPGA’s operation speed), at the expense of 1.35x area. For a more acceptable area increase, we previously discussed the $N = 22$ architecture that achieved a 1.34x CAD run-time speedup at the expense of a 1.05x area increase.

Mulpuri and Hauck [92] investigated the achievable trade-offs between CAD run-time and critical path delay for a number of placement and routing algorithms. Their results show that 5x CAD run-time speedup could be achieved at the expense of a 2.5x increase in delay. For a more acceptable quality result, they also report a 2.2x CAD run-time speedup at the expense of only 1.8x increase in delay.

Ultra-fast VPR [97] is a hierarchical FPGA placer that was reviewed in Section 2.4.5.
This work showed a CAD run-time versus placement bounding-box (BB) cost trade-off curve. Different points on the curve could be achieved by varying a set of parameters used to control the algorithm. At one extreme of the curve, they report a 75x placement run-time speedup at the expense of 1.8x increase to BB cost. More conservatively, they also report a 7x placement run-time speedup at the expense of 1.05x BB cost increase.

The parallel placement algorithm by Wang and Lemieux [120] showed incredible speedups. Trade-off in placement run-time and quality could be achieved by varying either the number of threads or parameters specific to the parallel algorithm. One point cited in the paper is a 123x placement run-time speedup at the expense of 1.08x increase in (estimated) delay and 1.11x increase in BB cost. This result was for 25 threads.

The VPR tool has a -fast options which reduces the number of attempted swaps per temperature\(^3\) during placement by a factor of 10. We directly measured the speed-ups of VPR -fast on a baseline architecture with \(N = 14\) logic blocks. On this baseline architecture, the results showed a 7.3x speedup in placement run-time but a 1.9x slow-down in routing run-time; the lower quality placement solution made routing more challenging. Overall VPR -fast provided a 2.34x speedup. We then investigated the impact of this fast mode on our family of high-capacity logic blocks. To do this, we repeated the experiments from Section 5.3 (the flat architecture) and used the VPR -fast option when placing the benchmark circuits. On average, a CAD run-time speedup of 2.5x was achieved at the expense of a 1.01x and 1.03x increase in delay and area, respectively. This speed-up from VPR -fast was on top of the speed-ups afforded by the high-capacity logic blocks.

There are three conclusions from this set of comparisons. First, the trade-offs between CAD run-time and quality afforded through the high-capacity logic blocks is com-

\(^3\) Achieved by setting the anneal parameter \(Innernum = 1\).
parable to previously published trade-offs afforded through algorithmic efforts. Second, the experiments using VPR -fast continue to confirm that algorithmic efforts to reduce CAD run-time can be complementary with architectural efforts. This is especially true for the architectural method presented in this chapter as it is compatible with the traditional CAD flow. Finally, our work is the only method of those discussed here that shows a reduction in critical path delay alongside CAD speedups.

5.6 Summary

This chapter focused on the use of architectural changes to address the CAD run-time problem. To explore a concrete example, we investigated the use of high-capacity logic blocks to directly reduce the placement and routing problem sizes. We looked at two ways to architect these blocks: directly scaling the traditional logic block architecture, and a multi-level logic block architecture. We first derived an equation to determine the amount of I/O needed for both types of logic blocks, and showed that it is much more suitable than previously published rule-of-thumb formulas when scaling to larger logic block sizes. Then, using a detailed experimental methodology, we quantified the CAD run-time versus quality of result (area and delay) trade-offs of these architectures. Finally, we compare the results of this study with several published algorithmic studies on trading off quality for CAD run-time speedups.

We showed that there is a clear ability to reduce CAD run-time at the expense of area-efficiency for the traditional architecture; critical path delay did not suffer. For example, using a traditional logic block with $N = 22$ compared to a baseline of $N = 14$ can reduce CAD run-time by 25% and critical path delay by 10% at the expense of a 5% area increase. We found the multi-level architecture to be more area-efficient beyond certain logic block sizes, but less delay-efficient when compared to the flat architec-
ture. However, due to the inability to pack as many nets into the logic blocks, place and route run-times are longer for the multi-level architectures. The summary of this is that although the multi-level architectures can be more area-efficient for a specific logic block size, the flat architecture can produce better CAD run-time reductions for a specific amount of area increase. We also showed that using a logic block packing algorithm focused on minimizing the number of global nets (iRac) compared to a purely timing-driven algorithm (TV-Pack) could reduce placement and routing run-time. This run-time reduction was on top of the reductions achieved through using high-capacity logic blocks.

In our comparison against four previously published algorithmic methods to trade off quality for CAD run-time, we showed that our results are comparable, and further show that architectural and algorithmic approaches to reducing CAD run-time can be complementary. Although the architectures explored in this chapter are relatively straightforward, this study validates the call for more research into using architectural changes to reduce CAD run-time.
Chapter 6

Conclusion

In this concluding chapter, we first summarize the research and contributions that were presented. Then, we discuss the limitations within the presented research, and suggest possible future work to address these limitations. Finally, we present some possible directions for longer-term future research.

6.1 Dissertation Summary

As transistor process technologies continue to improve, FPGA logic capacities also continue to increase. This scaling places an increasing demand on the corresponding FPGA CAD tools used to map a user circuit to an FPGA. Users of modern high-capacity FPGAs already experience mapping times that exceed an entire work day, and the memory requirements of these tools already exceed the amount available on a typical commodity desktop computer. As FPGAs continue to grow, the problem will become worse. Unless the scalability of the FPGA CAD tools is addressed, the long run-times and large memory requirements will become a hindrance to future FPGA scaling, leading to increased costs and design-times for companies that use these devices. This dissertation described
three contributions towards addressing this problem.

Chapter 3 described an *analytical model that relates key FPGA architectural parameters to the FPGA CAD place and route run-times*. Although studies exist that describe the worst-case computing efficiencies of the underlying CAD algorithms [57, 91, 93], the novelty of our model is that it draws out the essence of how architectural choices affect the typical performance of these algorithms when applied to FPGA CAD. The model contributes to an emerging body of theory on FPGA architectures [43, 44, 52, 69, 107], and helps FPGA architects directly consider CAD run-time during early-stage architecture design. The major challenge tackled in the derivation of our model were, balancing simplicity with accuracy, and to expose the architecture’s impact on the metric of run-time. The model was validated against the commonly used VPR academic FPGA place-and-route tool. Through this validation, we showed that the model can accurately capture the trends in run-time when various key architecture parameters (\(K\), \(N\), \(F_{cin}\), \(F_{cout}\), \(W\), \(L\)) are changed. The chapter concluded with two example applications of the model in the context of FPGA architecture design. This work was published in [38], and to our knowledge is the first published description for such a run-time model.

Chapter 4 focused on the scalability of the FPGA CAD tool’s storage requirements. Although software memory usage can be very implementation-specific, the research that we presented is independent of the specific algorithm implementation details, and can apply to a wide range of FPGA routing algorithms. The proposed techniques are specific to the domain of FPGAs, and lead to long-term benefits by reducing the order at which the CAD tool memory requirements scale. Specifically, we first described a novel scheme to represent the FPGA’s programmable resources. A routing algorithm needs this representation so it can determine what FPGA resources are available to use in order to implement a design’s nets. By leveraging the regularity of the FPGA architecture, the
proposed representation does not need to explicitly model every programmable resource. To make this idea concrete, we implemented this scheme into VPR, and showed that the memory usage can be reduced by over an order of magnitude (5x-13x) for modern FPGAs. More importantly, we showed that the rate at which the memory requirements scaled was reduced. Another important property is that the representation is still exact. Due to extra steps carried out by the router to facilitate the new representation, the routing run-time and overall place-and-route run-time increased on average by 2.26x and 1.28x respectively. A preliminary version of this technique was published in [36], and to our knowledge was the first published research focused on the memory requirements of modern FPGA CAD tools. Our memory-efficient version of VPR has also been made publicly available to the research community.

A second memory-related study was performed to investigate the lifespan of the temporary routing data. Aside from information used to describe the architecture, storage is required for each programmable resource to track various algorithm specific metrics used during routing. We made several observations on when storage for this information is required, and based on these observations, we proposed three simple dynamic memory management schemes that varied in their aggressiveness to reduce memory usage. A design-adaptive scheme was also described. We quantified the memory and run-time trade-offs of each scheme, as well as the bounds on achievable memory reductions using an experimental methodology. The simple schemes were able to further reduce memory usage in high-density mappings by 24%, 34%, and 43% while incurring an additional routing run-time increase of 4.5%, 6.5%, and 144% respectively. The design-adaptive scheme achieved memory reductions comparable to that of the most aggressive simple scheme (41%) while incurring only a fraction of the run-time overhead (10.4%). For low-density mappings, memory reductions of 33%, 50%, 62% and 59% could be
achieved for the three simple and design-adaptive schemes, respectively. The complete study from Chapter 4 was published in [35].

Chapter 5 focused on the use of architectural changes to address the CAD run-time problem. To explore a concrete example, we investigated the use of high-capacity logic blocks to directly reduce the placement and routing problem sizes. Using such blocks means fewer blocks are required to implement a user design. This in turn reduces the number of blocks that need to be placed and the number of nets that need to be routed, thus directly reducing the placement and routing problem sizes. We looked at two ways to architect these blocks: directly scaling the traditional logic block architecture, and a multi-level logic block architecture. We first derived an equation to determine the amount of I/O needed for both types of logic blocks, and showed that it is much more suitable than previously published rule-of-thumb formulas when scaling to larger logic block sizes. Then, using a detailed experimental methodology, we quantified the trade-offs between CAD run-time and quality of result (area and delay) for these architectures. Finally, we compare the results of this study with several published algorithmic studies on trading off quality for CAD run-time speedups.

We showed that there is a clear ability to reduce CAD run-time at the expense of area-efficiency for the traditional architecture; critical path delay did not suffer. For example, using a traditional logic block with $N = 22$ compared to a baseline of $N = 14$ can reduce CAD run-time by 25% and critical path delay by 10% at the expense of a 5% area increase. We found the multi-level architecture to be more area-efficient beyond certain logic block sizes, but less delay-efficient when compared to the flat architecture. However, due to the inability to pack as many nets into the logic blocks, place and route run-times are longer for the multi-level architectures. The summary of this is that although the multi-level architectures can be more area-efficient for a specific logic
block size, the flat architecture can produce better CAD run-time reductions for a specific amount of area increase. We also showed in that CAD run-time reductions in this architectural manner can be complementary to algorithmic opportunities for reducing CAD run-time. We showed that using a logic block packing algorithm focused on minimizing the number of global nets (iRac) compared to a purely timing-driven algorithm (TV-Pack) could reduce placement and routing run-time. This run-time reduction was on top of the reductions achieved through using high-capacity logic blocks.

In our comparison against four previously published algorithmic methods to trade off quality for CAD run-time, we showed that our results are comparable, and further showed that architectural and algorithmic approaches to reducing CAD run-time can be complementary. This work was published in [37].

6.2 Limitations and Short Term Future Work

This section summarizes the limitations of the research described in this dissertation and points to possible avenues of future work that can address them.

6.2.1 Analytical Models Relating FPGA Architecture and FPGA CAD Processing Requirements

There are two notable limitations to the analytical models from Chapter 3. The first limitation is that a number of architectural and CAD assumptions were made in the derivation of the models. The models assumed only homogeneous FPGAs, all wire segment lengths are of the same length, and a specific switch box connectivity \((F_s=3)\). Generalizing these assumptions is straight-forward, and would make the models more flexible. In terms of CAD assumptions, the models assume Simulated Annealing placement with a bounding-box wirelength cost function, and Pathfinder based routing. Attempting to
generalize the models to encompass different algorithms may not be a suitable path. For example, force-directed (analytical) placement is a very different algorithm than simulated annealing placement. It is unlikely that the run-time of both algorithms can be expressed accurately under a single model. Success would be more likely if separate models were developed. That being said, the steps taken to derive the presented models could serve as a good foundation for the derivation of other similar models.

The second limitation is that parts of the model are based partly on empirical observations. We have not found a way to accurately analytically model the number of routing iterations and the number of wires explored in the routing run-time model. Doing so would provide some very interesting insight and also make the model more flexible.

6.2.2 Storage-Efficient Representations for FPGA Architectures

The main limitation to the architecture representation proposed in Chapter 4 is that it incurs an increase in routing run-time. Reducing this run-time overhead is likely possible through additional efforts in improving the current software implementation. Another limitation is that the current implementation of our memory-efficient representation only supports FPGAs that consist only of logic blocks. The techniques that were described do support heterogeneous blocks (such as memory and DSP blocks); we simply did not implement this portion during the validation of the idea. Addressing the above limitations would be straightforward and possibly beneficial to users of VPR. However, these are mainly engineering tasks and may not have much direct research benefit.

6.2.3 Towards Scalable CAD Run-time Through Architecture

For the study on architecting high-capacity logic blocks in Chapter 5, a notable limitation is that we did not apply specialized multi-level packing algorithms when mapping
to the multi-level logic blocks. These more sophisticated algorithms would likely incur additional run-time. We first wanted to investigate the area and delay trade-offs under more optimistic CAD run-time conditions. In addition, implementations of these algorithms were not readily available. That being said, such tools may somewhat improve the performance (area and delay) of the mapped circuits for the multi-level architectures. This would be interesting to quantify.

A second limitation is that some of the architectural assumptions are simplified compared to modern commercial FPGAs. For example, modern FPGAs contain elements like carry-chains, DSP and memory blocks, fracturable LUTs, and fast paths through LUTs. More accurate modelling of these components would improve the accuracy of the area and delay results in the presented study. Although the two limitations discussed so far would have an impact on the CAD run-time versus performance trade-off curves, they will not change the curves’ overall trends. One of the key contributions of this study is that it illustrated that such a trend exists.

Finally, when scaling to larger logic blocks, the local interconnect area becomes very significant. We used a transistor depopulation technique that maintains full logical connectivity. We could investigate the impact of more aggressive depopulation techniques such as those described in [74]. This increases routing time however, but quantifying the overall impact to the CAD run-time versus area trade-off curve would be interesting.

6.3 Long-Term Directions for Future Work

The previous section discussed short-term future work for addressing some of the limitations of the research presented in this dissertation. In this section, we describe some possible long-term research directions. We break this down into the three areas of analytical models, CAD memory-requirement scalability, and CAD run-time scalability.
6.3.1 Analytical Models

Modeling the impact of FPGA architecture on various metrics is still an emerging area of research. Since our model is the first published model that relates FPGA architecture to CAD run-time, there is room for improvement and many orthogonal directions of research. Some ideas include:

- Model the relationship between architecture and CAD run-time for other CAD stages such as synthesis, technology mapping, clustering, timing analysis, etc.

- Derive models for other placement and routing algorithms and cost functions such as force-directed placement, or timing-driven placement.

- Analytical modelling for FPGAs is still a fairly new topic. One aspect that has not been looked at yet is the impact of architecture on post-routing wirelength. This is needed in our routing run-time model to make a completely analytical flow.

- Parallelizing the CAD algorithms will play an important role in tackling the long CAD run-times. Some aspects of the FPGA architecture may affect the amount of achievable parallelism. Modelling this relationship between architecture and achievable parallelism could lead to further insight on designing architectures amenable to parallel FPGA CAD tools. For example, fine-grain parallel routing could be performed by parallelizing the neighbour expansion step. In the case of a wire segment, the amount of parallelism is limited by the number of routing resources to which the wire can connect.

In general, as more research is performed on other aspects of FPGA architecture modelling, additional applications of our run-time model will emerge. For example, there does not yet exist any models that describe post-routing critical path delay, or
power dissipation. If these models were to exist, early stage CAD run-time versus delay or power studies could be performed.

6.3.2 CAD Memory-Requirement Scalability

Identifying areas of research into improving CAD memory-requirement scalability is challenging because memory usage can be very implementation specific. In this dissertation, we focused on efficiently representing the FPGA architecture which is an aspect that is required for any FPGA CAD tool. The user circuit netlist is another aspect that would be required by any FPGA CAD tool, and its size would also increase over time as process technology scales. Efficiently representing the netlist and its associated information such as the timing graph to the CAD tool could be an area of future research.

As was shown in Chapter 4, the proposed architecture representation achieves significant memory reduction at the expense of increased CAD run-time. In general, reducing memory usage far below the amount of available RAM may not be significantly beneficial. Investigating ways to trade-off memory reduction and CAD run-time could allow a user to meet the memory limit imposed by the amount of available RAM, while minimizing the CAD run-time penalty.

6.3.3 CAD Run-time Scalability

There are many possible directions towards tackling the CAD run-time problem. We will discuss future work in terms of architectural changes to reduce CAD run-time as this was the focus in the dissertation.

- Investigate architectures with features to recoup degraded mapping quality of fast CAD algorithms. Most fast CAD techniques lead to degraded mapping quality (i.e. area efficiency, delay, routability, and or power) [92]. It is then conceivable
to over-design the architecture, or optimize the architecture to compensate for this degradation. The end goal would be to achieve equivalent mapping quality and faster CAD at the expense of absolute area.

- Investigate architectures that are amenable to hierarchical mapping algorithms. ASIC CAD flows experience the same CAD scalability problem faced by FPGA tools except that they often face problems of an even larger scale. To address this, many ASIC placement algorithms are now hierarchical in that they iteratively coarsen the problem (reduce problem size), solve the coarse problem and use this solution to guide the solving of a finer-grain level. Applying these same hierarchical approaches to FPGA placement can be challenging because the resources on FPGAs are pre-fabricated and this imposes additional constraints. An interesting research question is whether FPGA architectures can be developed that would alleviate these constraints, thus allowing hierarchal ASIC mapping algorithms to be applied to FPGAs.

- Architectures that are amenable to parallel CAD tools. Now that commodity processors continue deeper into the multi-core paradigm, parallel CAD tools will definitely be an important direction for tackling the CAD run-time problem. An obstacle in achieving efficient speedups through parallelism is to reduce or eliminate various forms of data dependencies. Some of these dependencies may be introduced by the architecture. Research into the limitations on parallelism imposed by the architecture, and ways to design the architecture to alleviate these limitations is an interesting research topic. Each CAD stage presents an individual sub-problem in this regard (i.e. architectures amenable to parallel technology mapping, clustering, placement, routing, etc.).
Bibliography


[87] G. Martin. Achronix to build the worlds most advanced field programmable gate arrays (fpgas) on intel 22nm process technology. Achronix Semiconductor Press Release, Oct 2010. → pages 2


146


[116] Various Authors. Valgrind. URL www.valgrind.org → pages 93


Appendices
Appendix A

Run-time Model Validation for Bi-directional Architectures

This appendix summarizes the validation results of the run-time models presented in Chapter 3 when applied to bi-directional routing architectures. The methodology is exactly the same as that in Section 3.4 with the exception of the use of VPR v4.30 and bi-directional routing architectures. Figures C.1a and C.1b show that the model can capture the trends the impact of architecture on run-time when LUT size and cluster size are varied, respectively. Figure A.1 shows a correlation plot for all placement experiments.

Figures A.2b, A.2d, A.2c, and A.2e show the accuracy of the model when $F_{Cin}$, $F_{Cout}$, $L$, and $W$ are respectively varied. Again, we show a representative set in each plot. For example in Figure A.2b and A.2d we have data for $L = 1, 2, 3, 4$ but only show the case of $L = 1$. Overall, the trends are accurately captured. Finally, Figure A.2f shows a circuit-by-circuit breakdown of run-time estimation for every routing performed in this section. Figure A.2a shows a circuit-by-circuit breakdown for $WE$, wires explored.
**Figure A.1:** Placement run-time model validation for bi-directional routing architectures.
Figure A.2: Routing run-time model validation for bi-directional routing architectures.
Appendix B

VPR Routing Resource Graph

Memory Usage Analysis

This appendix describes a simple analysis on how the memory usage of VPR’s routing resource graph scales. The architecture parameters used in this analysis are summarized in Table B.1.

B.1 Counting Nodes and Edges

We first count the average number of nodes and edges in each logic block (LB) and IO tile. We then multiply these values by the number of LB and IO tiles in the FPGA. The

Table B.1: Architecture parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Number of lookup tables per logic block</td>
</tr>
<tr>
<td>I</td>
<td>Number of inputs per logic block</td>
</tr>
<tr>
<td>N_x, N_y</td>
<td>FPGA dimensions</td>
</tr>
<tr>
<td>W</td>
<td>Routing channel width</td>
</tr>
<tr>
<td>L</td>
<td>Wire segment length</td>
</tr>
<tr>
<td>F_{Out}</td>
<td>LB output pin connection box flexibility (Fractional)</td>
</tr>
<tr>
<td>F_{In}</td>
<td>LB input pin connection box flexibility (Fractional)</td>
</tr>
<tr>
<td>IO_JRAT</td>
<td>I/O pads per I/O block</td>
</tr>
</tbody>
</table>
number of nodes per tile depends on the number of pins in the tile, and the number of wire segments that start in that tile. The number of edges in a tile depends on the number of resources of each type in the tile (e.g. LB output pins), and the number of resources to which the resource can connect (i.e. $F_{cout} \cdot W$ wire segments can be reached from an LB output pin). Table B.2 summarizes the number of resources of each type per tile, and the number of programmable connections from each of these resources. The total number of edges in each tile is basically the dot product (cartesian product) of the "Nodes per tile" column and the "Fanout Edges” column. Note that the $E_{wire}$ term is described in Section 3.3.4 of Chapter 3.

Equation B.1 and B.2 describe the number of nodes per logic block tile and I/O tile respectively. Equation B.3 and B.4 describe the number of edges per logic block tile and I/O tile respectively. Equation B.5 describes the overall memory requirements where $C_{node}$ and $C_{edge}$ denote the actual memory usage for a single node and edge respectively.

$$Nodes_{LB} = 2 \cdot \frac{W}{L} + I + N + \frac{2}{\text{Wire segments}} + \frac{2}{\text{Pins}} + \frac{2}{\text{Source and sink}}$$  \hspace{1cm} (B.1)

$$Nodes_{IO} = \frac{1}{2} \cdot \frac{W}{L} + IO_{RAT} + \frac{2}{\text{Wire segments}} + \frac{2}{\text{Pins}} + \frac{2}{\text{Source and sink}}$$  \hspace{1cm} (B.2)

$$Edges_{LB} = N + I + N \cdot W \cdot F_{cout} + 2 \cdot \frac{W}{L} \cdot E_{wire}$$ \hspace{1cm} (B.3)

$$Edges_{IO} = IO_{RAT} \cdot (2 + W \cdot F_{cout}) + \frac{1}{2} \cdot \frac{W}{L} \cdot E_{wire}$$ \hspace{1cm} (B.4)
Table B.2: Number of nodes of each resource type per tile. And number of edges from each resource type.

<table>
<thead>
<tr>
<th>Tile Type</th>
<th>Resource Type</th>
<th>Nodes per tile</th>
<th>Fanout Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLB</td>
<td>source</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>sink</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>input pin</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>output pin</td>
<td>N</td>
<td>$W \cdot F_{\text{Cout}}$</td>
</tr>
<tr>
<td></td>
<td>wire segments</td>
<td>$2 \cdot \frac{N}{L}$</td>
<td>$E_{\text{wire}}$</td>
</tr>
<tr>
<td>IO</td>
<td>source</td>
<td>1</td>
<td>IO_RAT</td>
</tr>
<tr>
<td></td>
<td>sink</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>input pins</td>
<td>IO_RAT</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>output pins</td>
<td>IO_RAT</td>
<td>$W \cdot F_{\text{Cout}}$</td>
</tr>
<tr>
<td></td>
<td>wire segments</td>
<td>$\frac{N}{L}$</td>
<td>$E_{\text{wire}}$</td>
</tr>
</tbody>
</table>

Table B.3: Example node and edge memory usage

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Memory usage (VPR 4.30)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{\text{node}}$</td>
<td>Memory required per node</td>
<td>72 bytes</td>
</tr>
<tr>
<td>$C_{\text{edge}}$</td>
<td>Memory required per edge</td>
<td>3 bytes</td>
</tr>
</tbody>
</table>

$$MEM = \frac{N_x \cdot N_y \cdot \left( C_{\text{node}} \cdot \text{Nodes}_{\text{LB}} + C_{\text{edge}} \cdot \text{Edges}_{\text{LB}} \right)}{\text{LB tiles}} + \frac{2 \cdot \left( N_x + N_y \right) \cdot \left( C_{\text{node}} \cdot \text{Nodes}_{\text{IO}} + C_{\text{edge}} \cdot \text{Edges}_{\text{IO}} \right)}{\text{IO tiles}}$$  \hfill (B.5)

### B.2 Equation Validation

The values for $C_{\text{node}}$ and $C_{\text{edge}}$ were found to be 72 bytes and 3 bytes, respectively, in the VPR 4.30 CAD tool. This was found by inspecting the data structures used to implement nodes and edges. These values are summarized in Table B.3.

To validate the equations against each architecture input parameter, we generate a family of architectures by sweeping that one parameter and holding all other parameters constant. The sweep range, and fixed value for each input parameter are summarized in...
Table B.4: Model validation architecture space

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Baseline Value</th>
<th>Sweep Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>4</td>
<td>[10, 14, 18, 22]</td>
</tr>
<tr>
<td>$I$</td>
<td>32</td>
<td>[8, 16, 24, 32, 40]</td>
</tr>
<tr>
<td>$N_x, N_y$</td>
<td>200</td>
<td>[25, 50, 100, 150, 200]</td>
</tr>
<tr>
<td>$W$</td>
<td>150</td>
<td>[50, 100, 150, 200, 250, 300]</td>
</tr>
<tr>
<td>$L$</td>
<td>4</td>
<td>[1, 2, 4, 8, 16]</td>
</tr>
<tr>
<td>$F_{cin}$</td>
<td>0.4</td>
<td>[0.2, 0.4, 0.6, 0.8, 1.0]</td>
</tr>
<tr>
<td>$F_{cout}$</td>
<td>0.4</td>
<td>[0.2, 0.4, 0.6, 0.8, 1.0]</td>
</tr>
</tbody>
</table>

(a) FPGA Array Size

(b) Channel Width

Figure B.1: Equation validation (a) through (b)

Table B.4: We then supply each architecture to both VPR, to acquire measured memory usage, and to the equations described in this chapter to acquire predicted memory usage.

Figures B.1a to B.1g shows the measured memory usage and predicted memory usage while varying each architecture parameter. This shows that the equations accurately capture the trends. There is a slight under-prediction because some of the less significant aspects of memory usage in the VPR tool are not modeled (e.g. the circuit netlist, and various miscellaneous support data structures).
Figure B.1: Equation validation (c) to (g).
Appendix C

Logic Block I/O Formula -
Additional Validation Results

This appendix provides additional validation data for the formula presented in Section 5.1.3 used to choose the number of inputs per logic block. Figure 5.3 had shown validation data for LUT sizes of $K = 4$ and $K = 6$. Figure C.1 in this appendix shows additional data for $K = 2, 3, 5,$ and $7$. 
Figure C.1: Additional validation results