Implementation of an API for and the Folding of FPGA Routing Resource Graphs in VTR

Ethan Steiner Rogers
Brigham Young University

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

BYU ScholarsArchive Citation
https://scholarsarchive.byu.edu/etd/9414

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact ellen_amatangelo@byu.edu.
Implementation of an API for and the Folding of FPGA Routing Resource Graphs in VTR

Ethan Steiner Rogers

A thesis submitted to the faculty of Brigham Young University in partial fulfillment of the requirements for the degree of Master of Science

Brent E. Nelson, Chair
    Mike Wirthlin
    Jeffrey Goeders

Department of Electrical and Computer Engineering
    Brigham Young University

Copyright © 2022 Ethan Steiner Rogers
    All Rights Reserved
ABSTRACT

Implementation of an API for and the Folding of FPGA Routing Resource Graphs in VTR

Ethan Steiner Rogers
Department of Electrical and Computer Engineering, BYU
Master of Science

FPGAs provide parallel computing that is fit for speeding up the computation of a large range of problems. Programming an FPGA involves a complex tool flow for which several CAD tools have been developed. These tools compute solutions to many problems such as packing, placement, and routing, which map a circuit design onto an FPGA. These computations require a great deal of memory, of which the Routing Resource Graph contributes the most of any individual data structure. If the RRGraph could be represented in a more compact manner, performance of the tool flow algorithms may be improved due to an increase in memory caching benefits. This work presents four variations on RRGraph folding which vary in memory usage reduction and runtime, with the most aggressive method reducing the RRGraph size by up to 4× while maintaining similar performance to the original representation.

Keywords: Routing Resource Graph, VTR, F4PGA, SymbiFlow, cache
ACKNOWLEDGMENTS

I would like to express appreciation for the counsel provided to me by my advisor, Dr. Brent Nelson who provided valuable suggestions and feedback for this work.

I would also like to thank my wife, Karinne Rogers, who spent many hours discussing this work, supporting me, and helping edit this thesis.

Gratitude is given to the many developers who gave feedback regarding the implementation of the RRGraph API. Of particular note is Vaughn Betz, Henner Zeller, and Xifan Tang.

Additional thanks goes to some of my immediate family members, Parker Rogers, Steve Rogers and Dan Rogers, who provided technical knowledge and insights into different aspects of this work.

Finally, special thanks goes to Google which funded the efforts that were put into this work.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title Page</td>
<td>i</td>
</tr>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>iii</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>iv</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td><strong>Chapter 1  Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Implementation Details</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Contributions</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Outline</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Summary of Results</td>
<td>3</td>
</tr>
<tr>
<td><strong>Chapter 2  Background and Related Work</strong></td>
<td>4</td>
</tr>
<tr>
<td>2.1 Versatile Place and Route</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Verilog to Routing</td>
<td>6</td>
</tr>
<tr>
<td>2.2.1 VTR Design Flow</td>
<td>6</td>
</tr>
<tr>
<td>2.2.2 VTR Usage Models</td>
<td>8</td>
</tr>
<tr>
<td>2.3 F4PGA</td>
<td>8</td>
</tr>
<tr>
<td>2.3.1 F4PGA Design Flow</td>
<td>9</td>
</tr>
<tr>
<td>2.3.2 Interfacing with VPR</td>
<td>10</td>
</tr>
<tr>
<td>2.3.3 Conversion Between F4PGA and VTR RRGraphs</td>
<td>11</td>
</tr>
<tr>
<td><strong>Chapter 3  Routing Resource Graph Representations</strong></td>
<td>12</td>
</tr>
<tr>
<td>3.1 F4PGA Representation</td>
<td>12</td>
</tr>
<tr>
<td>3.2 VTR Representation</td>
<td>15</td>
</tr>
<tr>
<td>3.3 Structure Comparison</td>
<td>16</td>
</tr>
<tr>
<td>3.4 Comparison of Actual Devices</td>
<td>17</td>
</tr>
<tr>
<td>3.4.1 Generating the RRGraphs</td>
<td>18</td>
</tr>
<tr>
<td>3.4.2 Comparing the Data Structures</td>
<td>18</td>
</tr>
<tr>
<td>3.4.3 Memory Usage</td>
<td>19</td>
</tr>
<tr>
<td>3.4.4 Access Time Comparison</td>
<td>21</td>
</tr>
<tr>
<td>3.5 Conclusion</td>
<td>24</td>
</tr>
<tr>
<td><strong>Chapter 4  Routing Resource Graph API</strong></td>
<td>26</td>
</tr>
<tr>
<td>4.1 Pros of an RRGraph API</td>
<td>26</td>
</tr>
<tr>
<td>4.1.1 Ease of Altering the Routing Resource Graph</td>
<td>26</td>
</tr>
</tbody>
</table>

iv
A.2 Switches for stratix_arch Device ................................. 71
Appendix B Benchmark Results ........................................ 72
Appendix C Cache Reference Results ................................. 74
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Comparison of RRGraph Data Members</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>VTR Memory Usage for XC7A50T</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>F4PGA Memory Usage for XC7A50T</td>
<td>20</td>
</tr>
<tr>
<td>3.4</td>
<td>Test 1 Runtime in Seconds</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>Test 2 Runtime in Seconds</td>
<td>22</td>
</tr>
<tr>
<td>3.6</td>
<td>Test 3 Results in the Form VTR / F4PGA</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>Runtime (s) in the Form Original / API</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>API Implementation Statistics</td>
<td>34</td>
</tr>
<tr>
<td>4.3</td>
<td>QoR Comparison Before and After node_direction() Pull Request Changes</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>Cache Relative Performance</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Node Data Repetition</td>
<td>43</td>
</tr>
<tr>
<td>5.3</td>
<td>Edge Repetition (in Thousands of Edges)</td>
<td>43</td>
</tr>
<tr>
<td>5.4</td>
<td>Benchmarks</td>
<td>45</td>
</tr>
<tr>
<td>5.5</td>
<td>nodes_all_attr Results in the Form Flat / Folded</td>
<td>51</td>
</tr>
<tr>
<td>5.6</td>
<td>switches_subsets Results in the Form Flat / Folded</td>
<td>54</td>
</tr>
<tr>
<td>5.7</td>
<td>dest_switch_subsets Results in the Form Flat / Folded</td>
<td>57</td>
</tr>
<tr>
<td>5.8</td>
<td>nodes_edges Results in the Form Flat / Folded</td>
<td>58</td>
</tr>
<tr>
<td>5.9</td>
<td>Size of Different F4PGA RRGraphs.</td>
<td>58</td>
</tr>
<tr>
<td>B.1</td>
<td>Benchmark Descriptions</td>
<td>72</td>
</tr>
<tr>
<td>B.2</td>
<td>nodes_all_attr Results in the Form Flat / Folded</td>
<td>72</td>
</tr>
<tr>
<td>B.3</td>
<td>switches_subsets Results in the Form Flat / Folded</td>
<td>73</td>
</tr>
<tr>
<td>B.4</td>
<td>dest_switch_subsets Results in the Form Flat / Folded</td>
<td>73</td>
</tr>
<tr>
<td>B.5</td>
<td>nodes_edges Results in the Form Flat / Folded</td>
<td>73</td>
</tr>
<tr>
<td>C.1</td>
<td>Cache References for k6_arm_core</td>
<td>74</td>
</tr>
<tr>
<td>C.2</td>
<td>Cache References for stratixiv_cholesky</td>
<td>75</td>
</tr>
<tr>
<td>C.3</td>
<td>Cache References for directrf</td>
<td>75</td>
</tr>
<tr>
<td>C.4</td>
<td>Cache References for linux_arty</td>
<td>75</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

2.1 Potential VTR Design Flows [1] . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.2 F4PGA Design Flow [2] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.3 F4PGA Device Generation [2] . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

3.1 F4PGA Representation of Routing Resources [3] . . . . . . . . . . . . . . . . . . 14
3.2 VTR Representation of Routing Resources [3] . . . . . . . . . . . . . . . . . . . 16
3.3 Corresponding nodes in VTR and F4PGA . . . . . . . . . . . . . . . . . . . . . 19
3.4 F4PGA get_edges() Implementation . . . . . . . . . . . . . . . . . . . . . . . . 23

4.1 Flat Representation of Node Data Along with Accessors . . . . . . . . . . . . 27
4.2 Folded Representation of Node Data Along with Accessors . . . . . . . . . . 28
4.3 Compute Node Length . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.4 API Compute Node Length . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.5 Instance of node_type() Implementation . . . . . . . . . . . . . . . . . . . . . . 34
4.6 Instance of node_length() Implementation . . . . . . . . . . . . . . . . . . . . . 35

5.1 Examples of Temporal and Spatial Locality . . . . . . . . . . . . . . . . . . . . 39
5.3 Original RRGraph Accesses of Node and Edge Data Structures . . . . . . . . 46
5.2 Flat RRGraph Representation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.4 nodes_all_attr Folded RRGraph . . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.5 switches_subsets Folded RRGraph . . . . . . . . . . . . . . . . . . . . . . . . . 53
5.6 Obtain Edges Using dest_switch_subsets Folded RRGraph . . . . . . . . . 55
5.7 dest_switch_subsets Folded RRGraph . . . . . . . . . . . . . . . . . . . . . . . 56
5.8 RRGraph Comparison . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

A.1 First 5 and Last 5 Switch Types for xc7a50t_test . . . . . . . . . . . . . . . 70
A.2 All Switch Types for stratix_arch . . . . . . . . . . . . . . . . . . . . . . . . . 71
CHAPTER 1. INTRODUCTION

When designing any computer program, the primary objective is correctness. If the program does not provide a correct solution, it does not matter how fast it can run or how little memory it uses. Once correctness is ensured, however, optimization seeks to reduce runtime and memory usage. One method of reducing these metrics involves optimizing the algorithms used by the program. Sorting a list, for example, can be done correctly many ways with some algorithms requiring less steps than others and consequently, less time [4].

Another optimization method deals with the trade off between data structure size and access time. Data that is represented in a simple array requires only one CPU instruction to access. When data is accessed, the CPU copies it into the cache to speed up future accesses. If a data structure becomes too large for its entirety to fit in the cache, every request for data that is not already in the cache will require a time intensive access to main memory. One solution to this problem is to represent the same data in a smaller, more complex data structure that a larger percentage of will fit in the cache. Such a data structure may take several more instructions to access due to its complexity, but its improved ability to fit in the cache means it will not require as many accesses to main memory. Depending on the complexity of the data structure and how many additional instructions it takes to access, it has the potential of being faster to access than the original data structure.

In the design of FPGA CAD tools there are large data structures that are used to represent different characteristics of a device. The Routing Resource Graph (RRGraph) is a data structure of particular interest, as it can grow to be as large as 2.5 GiB for certain FPGAs. Due to the nature of FPGAs, there exists a certain amount of repetition within device layouts that manifests itself in the RRGraph. This repetition can be used to compress or fold the RRGraph and reduce the memory needed to represent it. A smaller, yet more
complex, representation of this data structure has the potential of being faster to access than
the original flat representation.

This thesis explores the trade offs of size versus access speed of the Routing Resource
Graph to understand:

1. What is the smallest way to represent the RRGraph?

2. What is the smallest representation without performance loss?

3. What are the representations that are in between greatest reduction in memory and
   best performance?

4. Does a representation exist that will reduce both memory and runtime for every device?

1.1 Implementation Details

This research will be done in the context of Versatile Place and Route (VPR) [5],
as VPR provides a Routing Resource Graph, FPGA tool flow, and several synthetic archi-
tectures for devices ranging in size and complexity. In addition to these synthetic devices,
benchmarks will also be run on F4PGA devices which more closely represent commercial
FPGAs, specifically Xilinx 7 Series devices.

1.2 Contributions

The contributions of this thesis are:

1. This work conclusively demonstrates that folding the RRGraph will still give a speed
   up for certain devices despite its increased complexity.

2. New knowledge is introduced regarding the foldability of VTR and F4PGA device
   representations.

3. This work provides the addition of an RRGraph API within VTR which enabled this
   research and provides opportunity for several other research interests to be explored
   such as the possibility of supporting different RRGraph representations beyond VTR’s
   current RRGraph representation.
4. An infrastructure is provided by this work for RRGraph folding research to continue in the future.

1.3 Outline

The remainder of this thesis will be presented in the following format. Chapter 2 will give a background of VPR and F4PGA including discussion of recent papers about them. Chapter 3 will discuss the RRGraph representations currently in use by VPR and F4PGA including discussion of each representation along with a comparison of data members and memory usage. Chapter 4 will explain the need for an RRGraph API within VPR and detail the steps that were taken to implement such an API. Chapter 5 will put forth several RRGraph folding methods that have been implemented in this research along with memory usage and runtime results. Chapter 6 will present some analysis of results and conclude. Additional information, references, and results are included in the appendix.

1.4 Summary of Results

The results demonstrated in later chapters show that the effectiveness of folding the RRGraph varies based on several factors. These factors include the RRGraph folding method being used, the circuit being implemented, and the device being represented. Each of these parameters has an impact on RRGraph memory reduction and the runtime of VTR. Below are some of the primary findings.

- The most compact RRGraph is folded to be $4 \times$ smaller than its original size.
- The best performing RRGraph runs slightly faster than the original representation while reducing memory usage by 20%.
- In general, F4PGA [2] devices are more complex to fold, resulting in less reduction in memory usage as compared with VTR devices.
CHAPTER 2. BACKGROUND AND RELATED WORK

Much of this work builds upon and uses open source projects that have been in development for several years. VPR, which was first released to the public in 1997, has been at the forefront of open source FPGA CAD tools for over two decades. VPR has been utilized or referenced by hundreds of research efforts to improve packing [6], [7], placement [8]–[10], and routing [11]–[13]. Additionally, some have attempted to expand beyond purely academic use and target commercial devices with VPR [2], [14], [15].

F4PGA, formerly known as SymbiFlow, is younger, having its first release only a few years ago in 2018 [2]. As each of these tools plays a large role in this work, this chapter will give an overview of them by reviewing several of their research papers.

2.1 Versatile Place and Route

This overview of VPR is based upon the initial release of VPR [5], while features of more recent versions of VPR are discussed in Section 2.2. VPR was the graduate work of Vaughn Betz, and an in-depth description of its features can be found in his dissertation [16].

VPR is an open source FPGA CAD tool focused on flexibility and the ability to target a large variety of FPGA architectures. Altering the architectural description of a hypothetical FPGA and rerunning the VPR flow can be done in an automated fashion, and thus allows quick and versatile exploration of different architecture features. VPR’s initial paper [5] gives an overview of its features, including methods used for packing, placement and routing. A brief synopsis of each of these steps will now be discussed.

The packing step in VPR is handled by VPACK, a tool for packing logic blocks that was created as a part of VPR. VPACK is capable of targeting a range of different logic blocks. The paper [5] does not give a detailed explanation of VPACK’s implementation,
but simply states that it takes as input a technology-mapped circuit and outputs a netlist containing FPGA logic blocks for use in VPR.

For its placement algorithm, VPR uses simulated annealing which is an algorithm that can be used for solving a large variety of problems incrementally. Simulated annealing follows a similar pattern no matter what the problem at hand is.

1. A cost function is defined, which measures the quality of a given solution  
2. An initial solution is determined  
3. A random change is made to the solution and the cost function is used to determine the change’s impact on solution quality  
4. The change to the solution is accepted or rejected based on a variety of parameters  
5. This process is repeated until certain termination criteria are met, and the final solution is accepted  

The cost function used by VPR for placement is termed a linear congestion cost function, and it discourages placements that require routing in locations on the FPGA with narrower channels. This cost function gave the best results compared to several others that were tested by VPR’s creators. In addition to the cost function, several other parameters used in simulated annealing were fine-tuned during VPR’s development to yield the best placement.

When it was first released [5] VPR’s routing algorithms outperformed all previously published FPGA routers that were available for comparison. Their routing algorithm was based on the Pathfinder negotiated congestion algorithm [17], [18] which was developed at the University of Washington. The algorithm starts by routing every net using the shortest possible path. After the initial pass, each net is assigned a cost based on how many of its connections were overused. An overused connection is one where the number of nets that use the connection is greater than its capacity. After each pass, the cost of overused connections is increased so that nets that are not critical will be routed on other paths. The circuit is routed many times, with each subsequent routing effort seeking to minimize cost. Once there are no overused nodes the routing is deemed successful. The routing algorithm allows
connections to be made up to three channels outside the bounding box of the given net terminals. This routing method is an effective and relatively simple method to route the nets of a design.

As of its initial release, the output of VPR included the placement and routing files along with other metrics such as maximum net length, timing analysis, track count, and routed wirelength. These data points allow researchers to gain a quick understanding of how different architectures perform, expanding the possibilities for FPGA architecture research.

2.2 Verilog to Routing

Verilog to Routing (VTR) adds many new features beyond the initial functionality of VPR, and was first openly published in 2012 [19]. VTR provides more of an entire tool flow, whereas VPR only performs a portion of the flow. At its core, VTR uses ODIN-II [20] for synthesis and elaboration, ABC [21] for logic optimization, and VPR for packing, placing, routing, and analysis. The most recent paper about VTR [1] describes improvements to quality, run-time, and memory usage in addition to the ability to better describe hierarchical FPGA architectures. The paper also gives an overview of VTR’s design flow and improvements to packing, placement, and routing compared to prior versions of VPR.

2.2.1 VTR Design Flow

Verilog to Routing maps a given circuit onto a desired FPGA architecture. It takes two arguments as input:

1. an FPGA architecture description (i.e. arch.xml)

2. an HDL file which describes a circuit (i.e. counter.v)

In order to accomplish the task of mapping a design onto an FPGA, VTR makes use of several already existing tools. Several possible design flows are given in Figure 2.1 which is taken from the paper on VTR 8 [1].
VTR adds several features beyond VPR’s initial functionality that can be seen in Figure 2.1. First of all, VPR takes as input a technology mapped netlist, whereas VTR takes an HDL design. This feature allows VTR to function more independently as hardware applications are designed in an HDL such as verilog rather than as netlists. Another feature of VTR shown in the figure is its flexibility to target different use cases. For example, the Routing Resource Graph can now be input to VPR at several of its stages, which allows for other CAD tools to generate their own Routing Resource Graph for use rather than relying on VPR to generate it based upon the architecture file. Additionally, VPR is now compatible with the possibility of other CAD tools performing the packing, placement, or routing. This may be useful when a packer, placer, or router is being designed individually as the rest of the tool flow can be completed by VTR. Also, there are three options included for how to convert from the initial HDL design into the netlist, which allows for further flexibility. Overall, the design flow of VTR provides users with many possibilities for execution depending on their individual use cases and objectives.
2.2.2 VTR Usage Models

As this work will be referencing different use models of VTR in this paper, some use models are highlighted here.

1. **Minimum Channel Width** - This use model generates the routing resource graph several times with decreasing channel widths in order to determine the minimum channel width at which the circuit may be mapped onto the FPGA.

2. **From File** - This use model reads the RRGraph from a file and then performs packing, placement and routing with a fixed channel width.

Each of these methods caters to different objectives, with the first method being more aimed at research into device architectures and the second method more fit for usage with commercial devices. When targeting a commercial FPGA, the channel width of the architecture cannot be altered, and thus the primary objective is to perform the FPGA toolflow. This method is used by F4PGA, as it uses VTR to target commercial devices. The results presented in Chapter 5 were obtained using the “From File” usage model. Further explanation of F4PGA and its relation to VTR will now be discussed.

2.3 F4PGA

Similar to VTR, F4PGA is a fully open source FPGA CAD tool which aims to provide a tool flow that can evaluate new FPGA architectures. Unlike VTR, however, F4PGA also seeks to provide a tool flow that can target commercial devices and provide functional bitstreams. In a recent paper [2] several features of F4PGA are explained, including the role that VTR plays in F4PGA’s design flow. F4PGA was previously known as SymbiFlow and as such, any further references to SymbiFlow in figures or otherwise are vestigial. The F4PGA project can be found on GitHub [22]. As F4PGA is relatively new, some URLs included in this work still contain the SymbiFlow URL. It should be noted that these sites may be ported over to the F4PGA naming schema in the near future.
2.3.1 F4PGA Design Flow

Several aspects of F4PGA’s design flow are similar to VTR’s. The usage of Yosys for synthesis, ABC for tech-mapping, and VPR for packing, placement and routing are implemented in both design flows, with these steps being required for F4PGA and optional for VTR. Thus, this portion of the design flow may be identical for each open source CAD tool. Below is a graphical representation of F4PGA’s design flow as included in [2].

![F4PGA Design Flow](image)

The primary differences between VTR’s and F4PGA’s design flows are at the beginning and end of each tool flow, with the body of each being similar. F4PGA starts by generating a device model that represents commercial devices well enough that it can generate a bitstream to program the respective devices. VTR, on the other hand, uses architectures that model devices well for experimentation and analysis, but not well enough to produce a bitstream. Furthermore, VTR’s design flow ends after timing analysis whereas F4PGA’s picks up there and takes the necessary steps to produce a bitstream.
2.3.2 Interfacing with VPR

VPR performs the tasks of packing, placement and routing within F4PGA’s design flow. Although VPR provides interfacing functionality, some work had to be done by developers of F4PGA in order to interface to and from VPR within F4PGA. The work required at the end of F4PGA’s design flow involved modifying VPR to output a FASM file, which specifies which bits within the bitstream need to be set in order to generate the bitstream. The beginning of the flow took more work as several input files including the RRGraph and architecture files had to be created for use in VPR. The generation of the device model which is used by VPR is shown in Figure 2.3 which comes from [2].

![Figure 2.3: F4PGA Device Generation][1]

The device information in the “Low-Level Device DB” block of Figure 2.3 comes from third party projects like Project X-Ray and Project Icestorm which provide a device description based on Xilinx Artix-7 and Lattice iCE40 FPGAs respectively. Several Python scripts are used to take the device description in that database and generate files for use in VPR including the architecture file and Routing Resource Graph. Once these files are generated, they can be input into VPR for it to perform packing, placement and routing. F4PGA then takes the resulting FASM file through the rest of its design flow and the bitstream is generated.

---

[1]: https://example.com/figure2.3.png
2.3.3 Conversion Between F4PGA and VTR RRGraphs

Figure 2.3 includes a line between “SymbiFlow DB” and “Construct RR graph” which represents the conversion from the F4PGA RRGraph representation to the corresponding RRGraph representation in VTR. As these representations differ, this conversion requires a firm understanding of each representation to accomplish. As this work aims to fold the routing resource graph, Chapter 3 gives a summary of each RRGraph representation along with comparisons between data members and memory usage. This explanation will provide a basis for the graph folding discussed in Chapter 5.

Further explanation of any necessary details of VTR or F4PGA that have not been covered here will be given in the context of examples that are used later.
CHAPTER 3. ROUTING RESOURCE GRAPH REPRESENTATIONS

In order to route a design onto an FPGA, the router must know several pieces of information about the device layout such as where wires can be connected together. Other data, including resistance and capacitance of the nets is also necessary to perform a timing analysis of your routing result. This and other information is provided to the router through a data structure called the Routing Resource Graph, which can be represented in many different ways.

The RRGraph is the largest data structure at play in the FPGA tool flow. Because of this, both the size and speed of access of the RRGraph play a critical part in the performance of packing, placement and primarily routing. In order to optimize these algorithms, the RRGraph should be represented in the most compact, efficient manner possible.

To better understand some variations of RRGraph representation, this chapter explores two methods of representing routing resources. The first method comes from F4PGA, which was discussed in Chapter 2. The second RRGraph representation comes from VTR which was also discussed in Chapter 2. This chapter will detail and compare the structure, memory usage, and performance of each RRGraph representation method. Chapter 5 will explore several methods of folding the RRGraph representation used by VTR.

3.1 F4PGA Representation

F4PGA represents the fabric of an FPGA as a grid of tiles that each contain zero to many tile wires, pips, and site pins. Wires that are electrically connected are grouped into nodes. Pips provide configurable connections between certain nodes. Definitions of these data members and others that define the representation are given below.

\textbf{tile} - x, y location in grid that contains tile wires, pips and site pins and is of a certain \textbf{tile_type}. 

\textbf{12}
tile_type - Every tile of a given tile type has the same sites, tile wires and pips.

tile wire - Partial net that exists within one tile, but can be connected to other tile wires as part of a node to span multiple tiles.

wire_in_tile - wire for a given tile type that contains capacitance, resistance and pip information. Every tile of this type will have a wire that has the attributes mentioned.

node - Set of tile wires that form a complete electrically connected group.

pip - Programmable interconnect point that can be turned on to connect two tile wires within the same tile together.

pip_in_tile - pip for a given tile type that has a source and destination wire_in_tile. Every tile of a given type will have the same pip_in_tiles.

site - Location within a tile that contains site pins.

site pin - Sink/source that connects a tile wire to/from a site. May only connect to one tile wire.

This representation is similar in structure to the Xilinx representation of the RRGraph because the Xilinx RRGraphs are used to construct the F4PGA RRGraphs. Much work has been done by contributors to the F4PGA project to document their RRGraph representation, and a helpful graphical description they have created is included as Figure 3.1 [3].

Several of the data members used in F4PGA’s RRGraph representation can be found in Figure 3.1. For example, a node consisting of 3 tile wires is at the top of the figure and spans 3 tiles. In the bottom of the figure a tile wire sinks into a site pin which is connected to a site. Additionally, a tile wire is connected to another tile wire via a PIP in the middle of the figure.
References to `pip_in_tile` and `wire_in_tile` are missing from Figure 3.1 because these last two data members are not physically represented on the device. However, they still play a role in the layout of the device. For example, “Tile N” in Figure 3.1 has a certain tile type which determines the `pip_in_tile` and `wire_in_tile` instances that exist within that tile. Every other tile of the same type will have the same `pip_in_tile` and `wire_in_tile` instances. This methodology avoids repetition of wires and pips where many tiles share these data members.
3.2 VTR Representation

Although VTR’s representation ultimately includes the same routing resource information, the structure is quite different from F4PGA’s representation. Several terms are shared between the two representations, with some of them meaning different things. The nodes, for example, in VTR are different than the nodes in F4PGA. The nodes in F4PGA represent a collection of tile_wires, whereas the nodes in VTR are the nodes of a graph as used in graph theory. The nodes of this graph consist of wires and pins, and the edges define which nodes can be connected to each other. An edge’s weight specifies the type of connection it makes between two nodes. Definitions and further explanation of these concepts are given below.

- **node** - routing resource that is one of three main types
  - CHANx/CHANY - net that may span several units in the x or y direction (technically lies in between tiles)
  - SOURCE/SINK - site pin
  - IPIN/OPIN - pins connected to SOURCE/SINK nodes that allow multiple routing paths to begin at a SOURCE and end at a SINK

- **edge** - connects two nodes together, and consists of a destination node and switch type
  - Destination Node - Specifies the node ID to which the source node can connect to via this edge
  - Switch Type - Specifies the type of connection the edge makes between nodes. Will be either configurable or non-configurable.
    * **configurable edge** - programmable connection that can be turned on to connect two nodes (analogous to a pip)
    * **non-configurable edge** - connection between two nodes that is always on (analogous to a short)
In an effort to understand and convert between the two representations, much work
has been done by contributors to the F4PGA project to document the VTR representation,
and a helpful graphical description they have created is included as Figure 3.2 [3].

![Figure 3.2: VTR Representation of Routing Resources [3]](image)

### 3.3 Structure Comparison

Because these two data structures represent the same physical devices there are many
parallels between them. For example, configurable edges in the VTR representation are
similar to pips in the F4PGA representation as each of these data members is used to
connect wires together. Similarly, a group of CHANX or CHANY in the VTR representation
is analogous to a routing node in the F4PGA representation. Table 3.1 gives a side by side comparison. Rows marked “N/A” signify data structures that are only represented in VTR.

<table>
<thead>
<tr>
<th>VTR</th>
<th>F4PGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHANX/CHANY</td>
<td>routing node</td>
</tr>
<tr>
<td>SOURCE/SINK</td>
<td>site pin</td>
</tr>
<tr>
<td>IPIN/OPIN</td>
<td>N/A</td>
</tr>
<tr>
<td>Configurable Edge</td>
<td>PIP</td>
</tr>
<tr>
<td>Non-configurable Edge</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Each representation comes with its own pros and cons. One of the disadvantages of VTR’s representation is that non-configurable edges are required to represent wires that span more than just x or y. On the other hand, F4PGA’s method of defining a node as a collection of tile wires allows a given node to span both x and y directions. This distinction requires VTR to include edges that represent connections between CHANX and CHANY that are always on (non-configurable edges), and these shorts are not represented by F4PGA. Further comparison of these data structures will be given below.

3.4 Comparison of Actual Devices

Up to this point, our comparisons made between the two RRGraph representations have been purely structural. In order to more fully understand the benefits and drawbacks of each representation method, routing resource graphs that represent the same device have been generated for both VTR and F4PGA. These RRGraphs have been used for comparison and testing.
3.4.1 Generating the RRGraphs

In order to obtain corresponding RRGraphs, several scripts were used from the f4pga-arch-defs repository on GitHub [23]. In order to use VPR in F4PGA’s tool flow, f4pga-arch-defs generates a VTR Routing Resource Graph from F4PGA’s own RRGraph. As discussed earlier, these routing resource graphs each represent the same routing resource information, but use two separate representation methods. The device that will be used for the examples in this chapter was generated using these discussed methods and is representative of the Xilinx Artix-7 XC7A50T device.

3.4.2 Comparing the Data Structures

In order to fairly compare the two representations, there are certain things that need to be taken into account, such as the fact that several CHANX / CHANY VTR nodes may be required to represent the corresponding F4PGA node. This is due to the fact that CHANX / CHANY only span one direction, but F4PGA’s nodes may span multiple directions. For clarity, definitions for each type of node are included here.

- **VTR Node** - May be a wire, pin, or src / sink.
- **F4PGA Node** - Collection of tile wires.
- **Routing Node** - Refers to all F4PGA nodes and VTR nodes of type CHANX / CHANY.

Figure 3.3 shows the correspondence between VTR and F4PGA for a set of connected routing nodes.

There are 2 F4PGA nodes and 3 VTR nodes in Figure 3.3. VTR nodes 101 and 20779 are part of the same F4PGA node, so two VTR nodes are equivalent to one F4PGA node. VTR node 21123, on the other hand, is the only VTR node that makes up its corresponding F4PGA node. The nodes used for this example come from the RRGraphs for the Xilinx Artix-7 XC7A50T device.
3.4.3 Memory Usage

Because the representations differ in how they represent the RRGraph, they also differ in the amount of memory required to represent it. The VTR representation adds the concept of CHANX / CHANY and removes the F4PGA concept of nodes and wires. VTR does not make use of the repeated patterns for tiles of a given type (i.e. pip_in_tile, wire_in_tile), whereas F4PGA does. Because F4PGA stores data on a per tile basis, its representation is smaller. Raw numbers regarding the memory usage of the VTR and F4PGA representation are given in Tables 3.2 and 3.3 respectively. The values in the “Memory Usage (MiB)” column were computed by multiplying the number of bytes required to represent the data member in the “Item” column by the number in the “Count” column.
Table 3.2: VTR Memory Usage for XC7A50T

<table>
<thead>
<tr>
<th>Item</th>
<th>Count</th>
<th>Memory Usage (MiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>559,476</td>
<td>8.54</td>
</tr>
<tr>
<td>edge</td>
<td>4,302,474</td>
<td>41.03</td>
</tr>
<tr>
<td>total</td>
<td>4,861,950</td>
<td>49.57</td>
</tr>
</tbody>
</table>

About 80% of the memory used by the VTR representation is taken up by the edges. Although each edge takes less memory than each node, there are $7.69 \times$ more edges than nodes, so their overall impact on memory usage is great.

Table 3.3: F4PGA Memory Usage for XC7A50T

<table>
<thead>
<tr>
<th>Item</th>
<th>Count</th>
<th>Memory Usage (MiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>375,659</td>
<td>3.58</td>
</tr>
<tr>
<td>wire</td>
<td>1,465,821</td>
<td>27.96</td>
</tr>
<tr>
<td>wire_in_tile</td>
<td>26,578</td>
<td>0.16</td>
</tr>
<tr>
<td>pip_in_tile</td>
<td>10,563</td>
<td>0.10</td>
</tr>
<tr>
<td>tile</td>
<td>13,898</td>
<td>0.08</td>
</tr>
<tr>
<td>tile_type</td>
<td>24</td>
<td>0.00</td>
</tr>
<tr>
<td>total</td>
<td>3,358,364</td>
<td>31.87</td>
</tr>
</tbody>
</table>

The bulk of memory used by F4PGA is in the wires, as the other data structures collectively only make up 12% of the RRGraph. This is due to the fact that every wire is represented, but many of the other data structures are patterns that remain the same for every tile of a given type. Each wire must be represented because they are uniquely a part of specific nodes. Every wire of a certain wire_in_tile type has the same resistance, capacitance, and pips, but not necessarily the same nodes.
Data Member Count Comparison

There are several interesting comparisons to be made between corresponding data members of the VTR and F4PGA representations given in Tables 3.2 and 3.3.

- Edges - VTR has 4,302,474 edges and F4PGA has significantly less, with only 10,563 pip_in_tiles. This may be because
  - F4PGA does not represent the non-configurable edges while VTR does
  - pip_in_tiles are only represented once for each tile_type
  - F4PGA does not represent SOURCE or SINK nodes, so there are no pips to connect them

- Routing Nodes - VTR has 559,476 nodes and F4PGA has nearly half as many with only 375,659 nodes. This may be because
  - Several VTR nodes may be grouped together as one F4PGA node
  - F4PGA does not represent SOURCE or SINK nodes that VTR does

Consequently, F4PGA ends up representing a smaller number of data members. This is one of the main reasons for its smaller RRGraph representation.

3.4.4 Access Time Comparison

Several tests were created and run to compare the access time of each routing resource graph. Although an exact comparison is difficult due to the differences in the representations, the best comparison possible has been sought. The 3 tests below each give a brief explanation of the test along with runtime results for each RRGraph representation. The tests were written in Python and run on an Intel Core i7-9700K CPU @ 3.60GHz.

Test 1: Get Random Data Member

This test randomly iterates over all nodes in the given routing resource graph and queries specific information from the node, such as x, y, resistance, and capacitance. Results for this test using RRGraphs for the Artix-7 XC7A50T device can be found in Table 3.4.
The time taken to access data members in F4PGA was longer in part because the accesses are more complex. Retrieving a node’s x coordinate, for example, requires accessing the node’s tile first, and then accessing that tile’s x coordinate. In the VTR representation, x coordinates are stored individually for each node, so they can be accessed directly.

Test 2: Get All Edges

This test randomly iterates over all nodes in the given routing resource graph. It then determine all other nodes that can be connected to from the current node. For VTR, this means for a given node (i.e. CHANX) the test finds all edges where the node is the source of that edge. For F4PGA, this means for a given node the test finds all other nodes that can be connected with via a PIP. For this to occur, a node must contain a wire with a pip that connects to another wire that is part of the destination node. Results for this test using RRGraphs for the Artix-7 XC7A50T device can be found in Table 3.5.

<table>
<thead>
<tr>
<th>VTR</th>
<th>F4PGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.54</td>
<td>3.14</td>
</tr>
</tbody>
</table>

The results for Test 2 show that the VTR representation is about 6x faster at accessing every edge. Similar to Test 1, this is likely due to the fact that finding the edges for F4PGA is more complicated than VTR. The VTR method can directly access its edge data structure to determine which nodes can be connected with. F4PGA, on the other hand, requires a series of accesses that look like the following pseudocode.
The code in Figure 3.4 iterates over each wire of a node to find which of the node’s wires can connect to other wires via a pip. The resulting nodes which can be connected to using the given pips are then returned. In order to do this using F4PGA’s current representation, the initial wire must link to its wire_in_tile which then gives a destination wire_in_tile. The wire that is of the destination wire_in_tile type and in the initial wire’s tile will be a part of the node that can be connected to.

Reducing the runtime of Test 2 for F4PGA could be achieved by creating helper data structures that minimize intermediate accesses. The most basic example would be to store a list of connectable nodes for each node. This data structure would directly give the desired result for Test 2 without nearly as much overhead. This is just one example of the trade off between memory usage and runtime. In this instance, a data structure that stores information directly would likely provide a speedup over the more complex representation.

**Test 3: Route a Given Distance**

This test randomly iterates over all nodes in the given routing resource graph. For each node the test attempts to traverse a certain distance by finding a path of connecting nodes that begin with the input node. To determine which node to travel to next, the test connects with the node that gives the furthest x or y distance traveled. If no path can be found that reaches the goal distance, the test returns that no path was found. Table 3.6 gives the results for the VTR and F4PGA representations of the RRGraph.
Table 3.6: Test 3 Results in the Form VTR / F4PGA

<table>
<thead>
<tr>
<th>Goal Distance</th>
<th>Potential Paths Found</th>
<th>Not Found</th>
<th>Runtime in Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>219,328 / 168,575</td>
<td>340,148 / 390,901</td>
<td>14.22 / 21.64</td>
</tr>
<tr>
<td>25</td>
<td>96,207 / 80,616</td>
<td>463,269 / 478,860</td>
<td>23.61 / 29.75</td>
</tr>
<tr>
<td>100</td>
<td>8,046 / 6,073</td>
<td>551,430 / 553,403</td>
<td>32.92 / 38.66</td>
</tr>
<tr>
<td>250</td>
<td>114 / 112</td>
<td>559,362 / 559,364</td>
<td>34.22 / 39.35</td>
</tr>
</tbody>
</table>

The results for this final test show VTR running faster than F4PGA, but not by too large of a margin. For each of the goal distances, F4PGA couldn’t find quite as many paths as VTR did. This may be due to differences in their node representations.

3.5 Conclusion

The tests presented in this chapter are not meant to provide a conclusive comparison between VTR and F4PGA devices, but primarily serve to give a high level comparison of the general differences and similarities between each representation method.

Both the VTR and F4PGA Routing Resource Graph representations come with their own intricacies that take time to understand. Determining which nodes can be connected with via pips in F4PGA’s representation is not readily understood. On the other hand, VTR’s edges allow for a quick and simple way to know which other nodes can be connected to, but the fact that nodes in VTR vary between wires, pins, and source/sinks can be confusing at first. Overall, each method has complexities that can be understood with time and effort, each providing an effective way of representing the available routing resources of a device.

Of particular interest to this work is the possibility of reducing the size of a routing resource graph by removing duplicated data that is shared by many nodes and edges in the graph. The F4PGA RRGraph already takes advantage of many repeated patterns in this manner via its `wire_in_tile` data member by storing data for every tile based upon tile_type. In order to experiment with the memory usages and performance of different folded RRGraphs, a CAD tool that could run the FPGA tool flow with various folded RRGraphs
was needed. VTR provides the necessary requirements for this task, but as of the beginning of this work the only RRGraph that could be used by VTR was its default flat graph. In order to use alternate RRGraph representations within VTR, an API was needed. Chapter 4 discusses the implementation of such an API and Chapter 5 explores several RRGraph Folding possibilities and their respective performance and memory reduction within VTR.
CHAPTER 4. ROUTING RESOURCE GRAPH API

This chapter will discuss VTR’s need for an API to access the Routing Resource Graph and detail the pros and cons of such an API. Because this work implemented a majority of the API, implementation details will also be discussed and specific use cases will be given where the API has already been useful.

Early on in this research, several of the head developers of VTR discussed the need for an RRGraph API. At the heart of the discussion was the hope that the Routing Resource Graph could be represented more efficiently and the fact that without an API, altering the structure of the RRGraph would require major refactoring of VTR. Every altered access into the RRGraph within VTR would require implementation-specific details. An API allows for more rapid and simple alterations to be made to the RRGraph structure. These concepts, along with other pros and cons of an RRGraph API will be discussed in this chapter.

4.1 Pros of an RRGraph API

The existence of an API for the Routing Resource Graph within VTR presents many benefits. Most notably, the API allows for simple and intuitive altering of the underlying structures that make up the RRGraph. Additionally, the API provides the possibility of abstracting away low-level implementation details from code that accesses the RRGraph, thus making those pieces of code simpler.

4.1.1 Ease of Altering the Routing Resource Graph

Functions throughout VTR that access the RRGraph don’t care how the underlying data is represented. Because of this, the representation of nodes and edges can be altered in any number of ways as long as the fundamental data attributes (location, node type, timing attributes etc.) are represented.
Perhaps the best way to illustrate this principle would be through a few examples. Figure 4.1 includes the original data structure which represents routing resources along with the access methods for the data members \texttt{xlow} and \texttt{type}.

```c
struct t_rr_node_data {
  int16_t cost_index = -1;
  int16_t rc_index = -1;
  int16_t xlow = -1;
  int16_t ylow = -1;
  int16_t xhigh = -1;
  int16_t yhigh = -1;
  t_rr_type type = NUM_RR_TYPES;

  union {
    Direction direction;  // Valid only for CHANX/CHANX
    unsigned char sides = 0x0;  // Valid only for IPINs/OPINs
  } dir_side;
  uint16_t capacity = 0;
};

vtr::vector<RRNodeId, t_rr_node_data> nodes;

int node_xlow(RRNodeId id){ /* Method for accessing xlow */
  return nodes[id].xlow;
}

t_rr_type node_type(RRNodeId id){ /* Method for accessing type */
  return nodes[id].type;
}
```

Figure 4.1: Flat Representation of Node Data Along with Accessors

The flat representation in Figure 4.1 uses a vector called \texttt{nodes} to store a struct of data for every node. Retrieving data members is as simple as directly accessing the \texttt{nodes} vector as shown on lines 19 and 23. This method makes accessing node attributes simple, but
also requires a large amount of storage as every attribute for every node is stored. Figure 4.2 shows another method of representing nodes which takes advantage of patterns of attributes that nodes share.

```c
struct t_rr_folded_node_data {
    int16_t xlow = -1;
    int16_t ylow = -1;
    int16_t cost_index = -1;
    int16_t rc_index = -1;
    int16_t dx = -1;
    int16_t dy = -1;
    t_rr_type type = NUM_RR_TYPES;
    union {
        Direction direction; // Valid only for CHANX/CHANY
        unsigned char sides = 0x0; // Valid only for IPINs/OPINs
    } dir_side;
    uint16_t capacity = 0;
};
```

The folded representation in Figure 4.2 reduces the amount of repeated data by storing node patterns where possible. The shared data is stored in the `node_data` vector defined on
line 18. Each member of that vector is of the `t_rr_folded_node_data` type which is given starting on line 1. An example of accessing the folded data for `node_type()` is on line 23. In this manner, the data is folded and requires half the initial storage or less, depending on the device. See Chapter 5 for information regarding this folding method’s performance, which is called `nodes_all_attr` in that chapter.

Although the code changes were small between the above two representations, implementing the folded version throughout VTR would require major refactoring without an API. Accesses to every node attribute would need to be changed to fit the new representation. With an API in place, however, the necessary code changes would be as simple as altering the code that describes the RRGraph structure from what is in Figure 4.1 to what is in Figure 4.2.

### 4.1.2 Abstracting Away Implementation Details

The purpose of the Routing Resource Graph is to provide VTR with the necessary information about the architecture to perform valuable computations. One such computation, routing, requires information such as which nodes can be connected to each other and where the nodes are located on the device. These pieces of information, and others, are provided through means of the RRGraph, but the underlying structure makes no difference to the router as long as the data is provided.

With an API in place, many implementation details can be hidden inside it. The calling routines can then simply request the information they require rather than computing it themselves. Consider, for example, the problem of computing the length of a wire. Without an API, this is done with the code in Figure 4.3.

```c
1 int node_length = 1 + node_xhigh(node) - node_xlow(node) +
2       node_yhigh(node) - node_ylow(node);
```

Figure 4.3: Compute Node Length
Every time VTR needs to know the length of a node, this code would have to be duplicated. Additionally, considering the many contributors to VTR, each developer may come up with their own way of evaluating the node length if they were unaware of currently used methods. The implementation of an API solves these issues by localizing every instance of computing node length into one location. The code in Figure 4.4 shows the current implementation of \texttt{node\_length()} in the API.

```cpp
inline int node_length(RRNodeId node) const {
  VTR_ASSERT(node_type(node) == CHANX || node_type(node) == CHANY);
  int length = 1 + node_xhigh(node) - node_xlow(node)
    + node_yhigh(node) - node_ylow(node);
  VTR_ASSERT_SAFE(length > 0);
  return length;
}

/* Calling code */
int node_length = rr_graph_view.node_length(node_id);
```

Figure 4.4: API Compute Node Length

With the \texttt{node\_length()} function now in the API, every piece of code that needs to know the node length can simply call the API. Other functions of this nature implemented by this work include:

- \texttt{node\_is\_initialized()}
- \texttt{nodes\_are\_adjacent()}
- \texttt{node\_is\_inside\_bounding\_box()}
- \texttt{node\_coordinate\_to\_string()}

Each of the above functions removes implementation details from the calling code and simplifies VTR. Code changes for optimization of these functions can now be made in
one unified location within the API and the benefit of such optimizations will be seen by every piece of code that calls the API function.

4.2 Potential Drawbacks

Although the API presents many beneficial features, not every aspect of it is positive. One drawback consists of the potential performance hit that may come due to the addition of the API. Another disadvantage is that the creation of the API requires many hours and much deliberation that could be spent elsewhere.

4.2.1 Performance Impact of the API

Early on in the discussions of whether or not to build an API for the RRGraph, performance impact was of concern. The addition of another layer of abstraction has the possibility to slow down code, especially when the code in question is heavily used throughout VTR.

After having implemented a majority of the API, benchmarks have been run for the following sets of device and design pairs. These benchmarks are listed in order of increasing architecture size and can be found within VTR’s code base.

- **EArch** refers to the `tseng.blif` circuit implemented on VTR’s EArch architecture.

- **k6** refers to the `arm_core.blif` circuit implemented on VTR’s k6_frac_N10_frac-chain_mem32K_40nm architecture.

- **stratixiv** refers to the `cholesky_mc_stratixiv_arch_timing.blif` circuit implemented on VTR’s stratixiv_arch architecture.

Most of the results in Table 4.1 are essentially a tie. Although there was a slight increase in runtime of 3.07% for large devices after the API was implemented, the benefits of the API outweigh this small slowdown. With an API in place, all of the positive features mentioned in section 4.1 are available, including the main focus of this work, which is graph folding.
Table 4.1: Runtime (s) in the Form Original / API.

<table>
<thead>
<tr>
<th>Task</th>
<th>EArch</th>
<th>k6</th>
<th>stratixiv</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routing</td>
<td>0.13 / 0.14</td>
<td>4.9 / 5.84</td>
<td>99.92 / 96.26</td>
</tr>
<tr>
<td>Lookahead</td>
<td>0.1 / 0.1</td>
<td>1.89 / 1.88</td>
<td>81.62 / 82.36</td>
</tr>
<tr>
<td>Packing</td>
<td>0.82 / 0.8</td>
<td>20.02 / 20.58</td>
<td>102.29 / 103.09</td>
</tr>
<tr>
<td>Placement</td>
<td>0.46 / 0.46</td>
<td>13.61 / 13.45</td>
<td>163.85 / 172.36</td>
</tr>
<tr>
<td>VTR Entire Flow</td>
<td>1.85 / 1.84</td>
<td>44.92 / 46.21</td>
<td>645.13 / 659.98</td>
</tr>
</tbody>
</table>

4.2.2 Work Required to Implement the API

At the beginning of the API implementation, it was difficult to know exactly how long it would take to properly add the RRGraph API into VTR. Because the effort would likely take many months, several measures were put into place to ensure the API did not negatively alter the functionality or performance of VTR. Each of the over 20 pull requests was carefully reviewed for style, brevity, and performance impact. A set of regression tests was used to benchmark the quality of results for each new API and ensure that VTR maintained similar runtime and memory usage to before. With each of these constraints in place, implementing the API took over 6 months of development.

The implementation of a majority of the API was put into place by the author of this work with several other developers providing feedback and approval to merge each pull request into VTR. The process was broken up into several pull requests, each of which implemented one to many API functions that replace accesses into the RRGraph. Although the API was not the only feature in development during this period, it did take a large portion of discussion time during weekly meetings and development time outside the meetings.

4.3 Implementation

The creation of an API for the routing resource graph presented an open ended problem with many solutions varying from a relatively low level wrapper class to an implementation with a high level of abstraction where many implementation details are hidden.
Although VTR will move towards further abstraction in the future, the best first step was to implement the API at a lower level to begin with, keeping in mind that once that was finished, adding higher level functions to hide implementation details would be easier.

4.3.1 API Structure

VTR interacts with the Routing Resource Graph both in creating it and in accessing it. In accordance with these two methods, the RRGraph API consists of the RRGraphBuilder class for creation of the Routing Resource Graph and RRGraphView class for data access. The RRGraphView side of the API was implemented by the author of this work, while the RRGraphBuilder was implemented by other collaborators. The functions found in Table 4.2 are implemented by the RRGraphView portion of the API, with each row representing an individual pull request into VTR’s master branch. Negative values in the “Flow Speedup” column indicate that VTR ran faster than before in the pull request for that row. The entries marked as “N/A” indicate pull requests for which the flow speedup values were not obtained.

Many of the functions in Table 4.2 are still very low level, but the API provides a simple way to abstract away some of them. For example, \texttt{node\_rc\_index} could be replaced completely by \texttt{node\_R()} and \texttt{node\_C()} as the RC index is only ever used to access resistance and capacitance values. The sheer number of files changed in all of the pull requests is evidence of the benefit of an API, because without the API, all of these files would have to be changed based on each individual RRGraph alteration. With the API in place, only a handful of files must be changed to implement a new RRGraph structure.

4.3.2 Individual API Addition

Although there were a handful of exceptions, many of the necessary code changes to add each individual function to the API were similar. Low level functions, like \texttt{node\_type()}, that make a direct access into the RRGraph data structure were directly replaced by calls to the RRGraphView API. These replacements would look something like the code in Figure 4.5.
Table 4.2: API Implementation Statistics

<table>
<thead>
<tr>
<th>Function(s)</th>
<th>Instances Replaced</th>
<th>Files</th>
<th>Flow Speedup</th>
<th>Merged On</th>
</tr>
</thead>
<tbody>
<tr>
<td>node_type()</td>
<td>189</td>
<td>36</td>
<td>1.42%</td>
<td>Jun 14</td>
</tr>
<tr>
<td>node_capacity()</td>
<td>35</td>
<td>14</td>
<td>1.10%</td>
<td>Jun 21</td>
</tr>
<tr>
<td>node_direction()</td>
<td>14</td>
<td>4</td>
<td>2.00%</td>
<td>Jul 12</td>
</tr>
<tr>
<td>node_fan_in()</td>
<td>9</td>
<td>5</td>
<td>0.11%</td>
<td>Jul 30</td>
</tr>
<tr>
<td>node_R()</td>
<td>19</td>
<td>8</td>
<td>-0.69%</td>
<td>Aug 26</td>
</tr>
<tr>
<td>node_C()</td>
<td>13</td>
<td>7</td>
<td>N/A</td>
<td>Oct 11</td>
</tr>
<tr>
<td>node_rc_index()</td>
<td>2</td>
<td>1</td>
<td>N/A</td>
<td>Oct 27</td>
</tr>
<tr>
<td>node_xlow()</td>
<td>113</td>
<td>28</td>
<td>0.31%</td>
<td>Sep 8</td>
</tr>
<tr>
<td>node_xhigh()</td>
<td>54</td>
<td>21</td>
<td>N/A</td>
<td>Nov 22</td>
</tr>
<tr>
<td>node_ylow()</td>
<td>112</td>
<td>28</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>node_yhigh()</td>
<td>55</td>
<td>21</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>node_cost_index()</td>
<td>24</td>
<td>14</td>
<td>-1.31%</td>
<td>Sep 29</td>
</tr>
<tr>
<td>is_node_on_specific_side()</td>
<td>16</td>
<td>5</td>
<td>N/A</td>
<td>Oct 11</td>
</tr>
<tr>
<td>node_type_string()</td>
<td>13</td>
<td>7</td>
<td>N/A</td>
<td>Oct 27</td>
</tr>
<tr>
<td>node_side_string()</td>
<td>2</td>
<td>1</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>node_pnc_num()</td>
<td>18</td>
<td>12</td>
<td>N/A</td>
<td>Nov 22</td>
</tr>
<tr>
<td>node_pin_num()</td>
<td>18</td>
<td>9</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>node_class_num()</td>
<td>9</td>
<td>6</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>node_track_num()</td>
<td>9</td>
<td>4</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

1. `rr_type = device_ctx.rr_nodes[inode].type(); // Original`
2. `rr_type = rr_graph.node_type(inode); // API`

Figure 4.5: Instance of node_type() Implementation

On the other hand, more complex functions, like `node_length()` replaced larger code blocks that implemented the same purpose. Finding these cases was more difficult than simply using control-F to search VTR because they were not all implemented in the same way. An example of inserting the `node_length()` API is included in Table 4.6.

The majority of APIs created fell into one of the two categories listed above, but there were a few that required further work. Some of the other changes were things like altering argument lists, creating enumerations, and small fixes that were made to VTR in passing.
4.3.3 Quality of Results Verification

In order to ensure that each pull request did not introduce major reduction in performance, a QoR regression was utilized. The regression consisted of several tests which stress VTR in a series of different ways to perform a thorough examination of VTR’s features. These regression tests were run on VTR before and after the changes in each pull request and the quality of results were compared. If there was a major increase in runtime, then further work was required to prevent that. The QoR results for `node_direction()` are given in Table 4.3 for reference.

The QoR results for this particular pull request show that the overall runtime of VPR was 98% of the original runtime after the implementation of the `node_direction()` API. Every step of VTR’s flow was faster besides the pack_time step, which was only slower by 0.83%. These results indicate that the changes made in this pull request did not negatively impact performance and thus, the pull request was acceptable to merge. A similar analysis of QoR results was made for each API pull request, ensuring the performance of VTR was not significantly reduced due to the API implementation.

4.4 API Conclusion

Implementing an API for the Routing Resource Graph into VTR took a significant amount of time but came with many benefits. Altering the structure of the Routing Resource Graph is now much more feasible and simple. Additionally, many low-level calls and repeated code blocks have been removed from VTR. As will be seen in Chapter 5, the implementation
and exploration of many folded RRGraphs was made possible through the creation of this API.

Table 4.3: QoR Comparison Before and After node_direction() Pull Request Changes

<table>
<thead>
<tr>
<th>Task</th>
<th>PR percentage of initial runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>odin_synth_time</td>
<td>93.02</td>
</tr>
<tr>
<td>abc_synth_time</td>
<td>99.86</td>
</tr>
<tr>
<td>max_vpr_mem</td>
<td>99.98</td>
</tr>
<tr>
<td>pack_time</td>
<td>100.83</td>
</tr>
<tr>
<td>place_time</td>
<td>99.26</td>
</tr>
<tr>
<td>min_chan_width_route_time</td>
<td>98.94</td>
</tr>
<tr>
<td>crit_path_route_time</td>
<td>98.22</td>
</tr>
<tr>
<td>vtr_flow_elapsed_time</td>
<td>98.00</td>
</tr>
</tbody>
</table>
CHAPTER 5. PERFORMANCE AND MEMORY BENEFITS FROM GRAPH FOLDING

The focus of this research has been to explore and implement folded versions of the Routing Resource Graph in the hopes that a smaller representation would bring with it a reduced memory footprint and increased performance. These benefits would be a result of the more compact RRGraph fitting better inside the CPU caches, leading to fewer cache misses and reducing the number of accesses to main memory. Further explanation of these principles is given in Section 5.1 of this chapter.

With an API in place in VTR, the underlying organization of the Routing Resource Graph can now be altered. Sections 5.1 and 5.2 discuss several principles that were taken into consideration during the process of designing and implementing each folded RRGraph. Section 5.3 explores several variations of folding the RRGraph including each method’s folded representation, memory usage and performance metrics. Section 5.5 gives a high-level comparison of all of the folded RRGraphs.

5.1 CPU Cache

Computers contain several levels of memory storage which vary in size and access time. These different memory levels, called cache levels, help reduce the runtime of computer programs by making relevant data faster to access. When a program makes a request for data that is not already in registers, the processor looks for the data beginning at L1 of the cache and proceeding on to L2, and L3 caches. If a particular level of cache does not contain the requested data, a cache miss occurs and the processor continues to the next, larger level in search of the data. If the data is not found in any of the cache levels, it must be fetched from main memory, which takes much longer than fetching it from anywhere in the cache. Once the data is found, it is used by the program and copied to the lower levels of cache.
so that it can be accessed more quickly the next time it is used. Table 5.1 includes relative access times for each of the memory levels. The values in the “size” column of the table are for the processor the tests in this chapter were run on (Intel i7-9700K) and were obtained using the `lscpu` command. Because access time in cycles for each level of cache could not be obtained for the processor that was used for testing, the values in the “Access Time” and “Relative Access Time” columns are given for a similar processor (Intel i7-6900K) [24].

Table 5.1: Cache Relative Performance

<table>
<thead>
<tr>
<th>Memory Level</th>
<th>Size</th>
<th>Access Time</th>
<th>Relative Access Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>256 KiB</td>
<td>4 cycles</td>
<td>-</td>
</tr>
<tr>
<td>L2</td>
<td>2 MiB</td>
<td>12 cycles</td>
<td>3 × L1</td>
</tr>
<tr>
<td>L3 (LLC)</td>
<td>12 MiB</td>
<td>36 cycles</td>
<td>9 × L1 or 3 × L2</td>
</tr>
<tr>
<td>RAM</td>
<td>62 GiB</td>
<td>230 cycles</td>
<td>58 × L1 or 19 × L2 or 6.4 × L3</td>
</tr>
</tbody>
</table>

The number of cycles necessary to access data not in the cache is 58× greater than the number of cycles needed to access data in L1 of the cache. Because of the stark increase in access time from lower to higher levels of cache, it is important to keep the cache in mind during the development of computer programs. Algorithms and data structures should be designed in a way that takes advantage of the structure and functionality of the cache.

5.1.1 Temporal and Spatial Locality

Two of the relevant design methods for optimizing cache usage involve the temporal and spatial accessing of data structures. Temporal locality refers to the use and reuse of specific data within a relatively short period of time. When data is accessed repeatedly, it remains in the lower levels of cache and can be accessed quickly. Alternatively, data that is not accessed frequently will be replaced in the cache by other data and must be fetched from main memory later, resulting in a much slower request.

Spatial locality refers to the access of data that is stored in relatively close locations in memory. This method is useful due to the manner in which the cache handles data
requests. When the cache retrieves a requested piece of data from main memory, it copies
the requested data and surrounding data in a chunk of 64 Bytes (could be smaller or larger
depending on the CPU) called a cache line. If the next data the program needs was stored
close to the original requested data, then it will already be available in the cache and not
require another access to main memory. The most simple application of spatial locality is
the accessing of data from arrays in a sequential manner. Examples of both spatial and
temporal locality are given in Figure 5.1

```c
// Temporal Locality - Specific data is accessed repeatedly in short time
A[0];
A[1000];
A[0];
A[1000];

// Spatial Locality - Array is accessed sequentially
for (i=0; i < A.size(); i++){
    A[i];
}
```

Figure 5.1: Examples of Temporal and Spatial Locality

Temporal locality is exhibited in lines 2-5 of Figure 5.1 because the data in A[0] and
A[1000] is accessed multiple times in a short amount of time. Spatial locality is exhibited
in lines 8-9 because data in the array is being accessed sequentially. Knowledge of these
methods and of the functionality of the cache are of particular import in the design and
implementation of the Routing Resource Graph which can grow to the level of multiple
gigabytes in some cases. References to the cache and its role in the performance of several
representation methods will be included in the following sections.
5.1.2 Cache Optimization and Folding

This section discusses findings that have to do with folding methods and code that optimizes cache performance. The task of folding the RRGraph and the task of optimizing that folded RRGraph for use in the cache are two problems that must be solved concurrently because of how the solution to each problem affects the other.

Folded Data Pattern Index

One method of folding a set of data structures consists of storing patterns of repeated data in a separate array. Instead of storing data within the data structure at hand, a pattern index is stored that indexes into the other array that contains the repeated patterns.

The number of patterns needed to represent the folded data determines the minimum number of bytes required to represent each pattern index. One byte can represent 256 different unique values, so if there are less than 256 patterns, the pattern index can be one byte in size. Two bytes can represent 65,536 pattern indexes and four bytes can fit 4,294,967,296 different patterns. Four bytes is the maximum size that could be needed for a pattern index because if there are more than 4.3 billion patterns, a pointer (which is 4 bytes) can be used instead of a pattern index.

The minimum size of the pattern index impacts the potential benefit of folding the nodes and edges of the RRGraph. Many of the different groupings of data members exceed 65,536 patterns for large devices and thus pattern indices must be stored in 4 bytes. When folding the RRGraph nodes, if every data member is included in the node pattern, the pattern size would be 16 bytes. Regardless of how many patterns are found, the node pattern index of 4 bytes must be stored for every node. This would yield a $4 \times$ reduction in size in the best case because the node data was only 16 bytes to begin with. This limitation prevents the node reduction from going to less than 25% of the original size if only nodes are considered. Additionally, as the RRGraph increases in size, a $4 \times$ reduction in the size of the nodes becomes increasingly less likely to fit inside the cache, decreasing the benefits of the cache that would be seen with smaller data structures.
Data Storage

Another aspect of designing for cache optimization is determining how to store the data structures. There are two main guidelines for this problem that cater to how the cache functions.

1. Data members that are used often together should be stored in the same struct or object.

2. Struct sizes should be a power of 2, such as 2, 4, 8 or 16 bytes.

The first guideline is beneficial because of how it takes advantage of temporal locality. If data members that are used often together are stored together then the CPU will only have to request one chunk of data for each set of data members, rather than one chunk of data for each individual data member. If those requests happen to be to main memory, then this technique will reduce the percentage of cache misses, increasing performance.

The second guideline is beneficial due to the manner in which the cpu accesses data from an array or vector. When data is requested, the processor must compute the data's address offset from the beginning of the array it is contained in. This address can be computed by multiplying the element’s index by the size of each element. When the struct’s size is a power of 2, however, this multiplication can be replaced with bit shifting because multiplication or division by a power of 2 is synonymous with bit shifting left or right respectively. Bit shifting can be performed faster than multiplication, with the amount of speedup depending on the processor in use. For this reason, storing data in structs that are a power of 2 improves cache performance.

5.2 Node and Edge Repetition

At the beginning of this research the task arose to determine how much repetition of data exists within the nodes and edges of a Routing Resource Graph. The degree to which the data structure could be compressed depends largely on the result of this task. It was expected that many of the synthetic architectures used by VTR would contain a larger percentage of node and edge patterns than commercial-like architectures such as those found in F4PGA devices. Results for node and edge repetition are given below.
5.2.1 Node Repetition

Table 5.2 contains the percentage of repeated node patterns within several architectures. Two nodes have the same pattern if they share \( x_{\text{low}}, y_{\text{low}}, x_{\text{high}}, y_{\text{high}}, \text{rc\_index}, \text{cost\_index}, \text{type}, \text{direction}, \text{side} \) and \( \text{capacity} \).

The “% patterns of initial” row of Table 5.2 shows the number of node patterns that are needed to represent all node data for a given device. This value for EArch is 20.8% which means that this device needs \( 15,052 \times 0.208 \), or 3,131 node patterns to represent all of its nodes. The memory required to store the node patterns and node indexes for EArch is 45.8%. This value differs from the “% patterns of initial” value because of the node pattern indexes that must be stored in addition to the node patterns.

The equations below show how much memory is required for storing node data. Equation 5.1 gives the original memory usage of only nodes and Equation 5.2 gives the memory usage when node patterns are utilized. \( S_n \) represents the storage size in bytes for just nodes, \( S_p \) represents the storage size in bytes when using node patterns, \( N \) represents the number of nodes, and \( P \) represents the number of node patterns. The 16 is a constant that represents the number of bytes required to represent the data for a single node, and 4 is the number of bytes required to represent a node pattern index.

\[
S_n = N \times 16 \tag{5.1}
\]

\[
S_p = N \times 4 + P \times 16 \tag{5.2}
\]

\( S_p/S_n \) represents the percentage of initial memory usage when using node patterns and is given in the “% size of initial” row of Table 5.2. In the best possible case, the number of node patterns will be essentially zero and the resulting memory usage for nodes will be \( \frac{N \times 4}{N \times 16} \) or \( \frac{1}{4} \) of the initial memory usage.

Several conclusions can be drawn from the data in Table 5.2. First of all, the F4PGA devices (50t, 100t, 200t) all contain about the same percentage of foldability. As the devices scale in size, the percentage of node patterns needed to represent all nodes remains
at about 12%. Additionally, VTR’s Stratix IV is the least foldable because it requires the most node patterns of all the devices shown.

Table 5.2: Node Data Repetition

<table>
<thead>
<tr>
<th>Category</th>
<th>EArch</th>
<th>Stratixiv</th>
<th>50t</th>
<th>100t</th>
<th>200t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>15,052</td>
<td>3,364,308</td>
<td>2,934,684</td>
<td>5,425,322</td>
<td>11,715,490</td>
</tr>
<tr>
<td>Folded Patterns</td>
<td>3,127</td>
<td>1,127,893</td>
<td>365,600</td>
<td>666,698</td>
<td>1,420,371</td>
</tr>
<tr>
<td>% patterns of initial</td>
<td>20.8%</td>
<td>33.5%</td>
<td>12.5%</td>
<td>12.3%</td>
<td>12.1%</td>
</tr>
<tr>
<td>% size of initial</td>
<td>45.8%</td>
<td>58.5%</td>
<td>37.5%</td>
<td>37.3%</td>
<td>37.1%</td>
</tr>
</tbody>
</table>

5.2.2 Edge Repetition

Nodes make up around 25% of the RRGraph memory used during the runtime of VTR, leaving 75% for the edges. Edges are made up of switches and destination nodes. The switches take up around 25% of the RRGraph memory, while the destination nodes take up around 50%. Because the edges make up a majority of the memory, reducing their size via folding or some other method is impactful. Although there are many ways to find repetition within the RRGraph’s edges, this section presents the foldability of edges as found by the dest_switch_subsets folding method that will be described in Section 5.3.5. In Table 5.3 “Edges in Folded Patterns” refers to the number of edges that are stored within the edge patterns of dest_switch_subsets.

Table 5.3: Edge Repetition (in Thousands of Edges)

<table>
<thead>
<tr>
<th>Category</th>
<th>EArch</th>
<th>Stratixiv</th>
<th>50t</th>
<th>100t</th>
<th>200t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edges</td>
<td>114</td>
<td>36,745</td>
<td>22,358</td>
<td>41,262</td>
<td>89,419</td>
</tr>
<tr>
<td>Edges in Folded Patterns</td>
<td>53</td>
<td>20,381</td>
<td>4,748</td>
<td>7,895</td>
<td>14,812</td>
</tr>
<tr>
<td>% edges of initial</td>
<td>46.75%</td>
<td>55.47%</td>
<td>21.22%</td>
<td>19.13%</td>
<td>16.56%</td>
</tr>
<tr>
<td>% size of initial</td>
<td>53.86%</td>
<td>62.26%</td>
<td>68.99%</td>
<td>67.23%</td>
<td>65.00%</td>
</tr>
</tbody>
</table>
The results in Table 5.3 show that the foldability of edges for F4PGA devices increases as the devices increase in size. This may be due to the fact that many of the structures present in the 50t device are repeated in the 100t and 200t devices. In general, the edges for VTR devices are more foldable than the edges for F4PGA devices as can be seen by comparing the values in the “% size of initial” row of Table 5.3. This is because, although the “% edges of initial” value is smaller for F4PGA devices than for VTR devices, the other data structures in the dest_switch_subsets folding method are much larger for F4PGA than for VTR. This is because the dest_switch_subsets folding method stores an edge pattern for every set of consecutive switch_ids and VTR devices have longer sets of consecutive switches than F4PGA devices. Consequently, F4PGA devices require more patterns and pattern indexes than VTR devices, causing their overall size to increase.

5.3 RRGraph Representations and Implementations

This section describes several methods of representing the Routing Resource Graph including VTR’s current method and four folded methods presented by this work. Memory usage and runtime results of each RRGraph will be included within its section. After all of the representation methods are presented, a general comparison of them will be given in Section 5.5.

5.3.1 Benchmarks

A total of 7 benchmarks were run to test the performance of each folded RRGraph compared to the original RRGraph representation. Results within this chapter will include results from 3 benchmarks for VTR devices and 1 for F4PGA devices. The results for the remaining 3 benchmarks is included in the appendix. All tests were run on a machine with an Intel Core i7-9700K CPU @ 3.60GHz. The benchmarks are included in Table 5.4.

These benchmarks were chosen because they use RRGraphs ranging in size with the arm_core benchmark being the smallest and the directrf benchmark being the largest. linux_arty was included because it uses an F4PGA device. The k6_frac_N10_frac_chain_mem32K_40nm is a heterogeneous architecture with carry chains designed for VTR 7.0 [25].
The stratixiv_arch [1] is a VTR architecture aimed at modeling Intel’s Stratix IV device [26]. Lastly, the xc7a50t_test device is an F4PGA device that aims to model Xilinx’s XC7A50T device [27].

Table 5.4: Benchmarks

<table>
<thead>
<tr>
<th>Name</th>
<th>description</th>
<th>device</th>
<th>LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>arm_core</td>
<td>processor</td>
<td>k6_frac_N10_frac_chain_mem32K_40nm</td>
<td>12,224</td>
</tr>
<tr>
<td>cholesky_mc</td>
<td>matrix decomposition</td>
<td>stratixiv_arch</td>
<td>28,598</td>
</tr>
<tr>
<td>directrf</td>
<td>communications/DSP</td>
<td>stratixiv_arch</td>
<td>442,169</td>
</tr>
<tr>
<td>linux_arty</td>
<td>Linux-based SoC</td>
<td>xc7a50t_test</td>
<td>13,808</td>
</tr>
</tbody>
</table>

5.3.2 Original Representation

The RRGraph representation that is currently in place in VTR represents the necessary data directly in several vectors. The terms RRNode and RREdge refer to a node or edge in the RRGraph respectively. RREdgeId is an integer that indexes into the edge data structures. The data structures for the flat RRGraph are given below.

- **node_storage** - Vector containing a struct of xlow, ylow, xhigh, yhigh, cost_index, rc_index, capacity, direction, side and node type for each RRNode
- **edge_src_node** - Vector containing source node for each RREdge
- **edge_dest_node** - Vector containing destination node for each RREdge
- **edge_switch** - Vector containing switch id for each RREdge
- **node_first_edge** - vector containing first RREdgeId for each node

Figure 5.2 gives an example of how the flat RRGraph would represent data for 3 nodes and 5 edges. In the figure, the arrows coming from the node_storage vector point to the attributes that are stored for each node. In this manner, it can be seen that
node 0 is of node type CHANY and has an rc_index of 1 in addition to its other attributes. The edges are represented via 3 vectors, which are all indexed with the same RREdgeId. The out-going edges of a given node are defined as a range of RREdgeIds going from node_first_edge[node_id] up until node_first_edge[node_id+1]. Because the last node is at the end of the node_first_edge array, an extra element is stored to determine the last node’s edge range. The node_first_edge vector shows that node 0’s first edge has RREdgeId 0 which can be used to index into the edge data structures and determine that edge 0’s dest_node is 2 and switch is of type 0.

An example of how the flat RRGraph’s data members are accessed by VTR’s timing driven router is given in Figure 5.3. Functions that serve as getters of the data at hand have been replaced with direct accesses for clarity.

```cpp
// iterate over edges from a given node
auto edges = rr_graph.edge_range(from_node);
for (RREdgeId from_edge : edges) { // from edge is an RREdgeId
    RRNodeId to_node = rr_graph.edge_dest_node[from_edge];
    short switch_id = rr_graph.edge_switch[from_edge];

    // expand neighbours of from_node
    int to_xlow = rr_graph.node_storage[to_node].xlow;
    int to_ylow = rr_graph.node_storage[to_node].ylow;
    int to_xhigh = rr_graph.node_storage[to_node].xhigh;
    int to_yhigh = rr_graph.node_storage[to_node].yhigh;
    // code continues below...
}
```

Figure 5.3: Original RRGraph Accesses of Node and Edge Data Structures

The code in Figure 5.3 primarily serves the purpose of illustrating how the original RRGraph representation is used inside of VTR. The edge_range() function on line 2 retrieves a sequential range of edge ids that begin at the first edge of the input node (from_node) and end at its last edge. Line 3 begins the iteration over each of the edge ids, and the next line fetches the destination node via the edge_dest_node vector. The sequen-
Figure 5.2: Flat RRGraph Representation
tial nature of edge ids means these accesses take advantage of spatial locality. In contrast, lines 8-11 do not take advantage of spatial locality due to the range of indexes that may be used to index into `node_storage`. The index will be the destination node (`to_node`) of each out-going edge from the source node (`from_node`), and destination node ids fanning out from a given source node have no sequential ordering.

As these data structures grow in size, features such as spatial locality play a larger role in performance because a smaller percentage of the entire RRGraph fits into the cache. With larger RRGraphs, only data accesses that are purely sequential will take advantage of spatial locality.

5.3.3 Folding Method 1: Nodes All Attributes

This folding method and the next 3 folding methods present RRGraph representations that have been implemented by this work. This method focuses on folding the nodes of the RRGraph, while the remaining folding methods fold the destination nodes, switches, or both. Although they can be folded jointly, the best folding methods have come from folding nodes or edges individually. When this is done, the memory savings from folding is limited to the percentage of the RRGraph that is made up of what kind of elements you are folding. Nodes make up about 25% of the RRGraph in most cases, so if you fold only nodes, the smallest possible RRGraph would be 3/4 of the original size. Alternatively, a folded graph where only edges are being folded would end up as 1/4 the original size in the best case. To achieve the smallest RRGraph, it is possible to join two folding methods such that the nodes and edges are each folded with separate methods.

The folding method discussed in this section is the simplest of the folding methods presented by this work and only folds the nodes of the RRGraph. The algorithm that folds the nodes proceeds as follows:

1. For every node, its node pattern consists of a set of every attribute of that node.
2. If the node pattern has not been found previously, it is added to a list of node patterns.
3. The index of the current node pattern is stored for the current node.
Once the nodes have been folded as described above, the access of node data takes one extra step beyond the flat representation. Instead of accessing the node’s data directly, the code first accesses the node’s pattern index and then uses that index to access its data. This folding method is represented visually in Figure 5.4 with the top of the figure showing the flat representation and the bottom showing how those nodes would be folded.

The different colors in Figure 5.4 each represent a unique node pattern. Each unique pattern contains a distinct set of xlow, xhigh, ylow, yhigh, rc_index, cost_index, capacity, type, direction and side. The arrows indicate which nodes contain the separate patterns. In this example, the second, fourth and fifth nodes all share the same node pattern. This folding method takes advantage of the observation that many nodes within the RRGraph share all of the same node attributes.

Table 5.5 gives the runtime and memory usage results of the nodes_all_attr folded RRGraph in the form flat / folded. The last three rows contain RRGraph memory usage and the maximum memory used by VTR during runtime. The values in the “VTR Memory (MiB)” row refer to the maximum resident set size as reported by VTR. The first six rows refer to the time taken to complete each respective task of the VTR flow. The rows that are of most interest to this work are the “RRGraph Size” and “Entire VTR flow” rows as they provide information regarding reduction in memory of just the RRGraph and the corresponding runtime of VTR. The VTR benchmarks in Table 5.5 are sorted left to right by increasing device size. The F4PGA device is given in the furthest right column.

On average, the nodes_all_attr Folded RRGraph is about 88% of the initial RRGraph size. Considering that the nodes take up 25% of the memory of the RRGraph, this folding method reduces the nodes memory usage from 25% to 13% or by about 2×. The time taken to run VTR using the nodes_all_attr Folded RRGraph is slightly longer than for the original representation, which is likely due to the extra step needed to obtain the node pattern index.
Figure 5.4: nodes_all_attr Folded RRGraph
Table 5.5: nodes_all_attr Results in the Form Flat / Folded

<table>
<thead>
<tr>
<th>Category</th>
<th>arm_core</th>
<th>cholesky_mc</th>
<th>directrf</th>
<th>linux_arty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routing (s)</td>
<td>3.5 / 3.7</td>
<td>66.1 / 74.3</td>
<td>1112.7 / 1294.8</td>
<td>277.2 / 303.6</td>
</tr>
<tr>
<td>Lookahead (s)</td>
<td>1.8 / 2.0</td>
<td>79.6 / 86.5</td>
<td>688.9 / 767.2</td>
<td>N/A</td>
</tr>
<tr>
<td>Packing (s)</td>
<td>16.4 / 16.3</td>
<td>91.0 / 90.7</td>
<td>799.6 / 811.7</td>
<td>96.4 / 99.7</td>
</tr>
<tr>
<td>Placement (s)</td>
<td>9.3 / 9.2</td>
<td>121.8 / 120.4</td>
<td>5458.9 / 5595.3</td>
<td>62.6 / 63.9</td>
</tr>
<tr>
<td>Entire VTR flow (s)</td>
<td>34.7 / 35.0</td>
<td>503.4 / 529.5</td>
<td>9189.6 / 9706.4</td>
<td>459.8 / 489.5</td>
</tr>
<tr>
<td>Entire VTR flow (%)</td>
<td>100.86%</td>
<td>105.18%</td>
<td>105.62%</td>
<td>106.46%</td>
</tr>
<tr>
<td>RRGraph Size (MiB)</td>
<td>222 / 188</td>
<td>4808 / 4415</td>
<td>27123 / 24806</td>
<td>3416 / 2869</td>
</tr>
<tr>
<td>RRGraph Size (%)</td>
<td>84.75%</td>
<td>91.83%</td>
<td>91.46%</td>
<td>84.00%</td>
</tr>
<tr>
<td>VTR Memory (MiB)</td>
<td>417 / 383</td>
<td>6940 / 6562</td>
<td>43575 / 41186</td>
<td>4013 / 3466</td>
</tr>
</tbody>
</table>

Two different ways of storing the node pattern indexes have been explored. The first method simply has an array that stores the 4 byte pattern index individually for each node. Since the pattern index is 4 bytes and the original data pattern is 16 bytes, $4 \times$ is the greatest reduction in memory usage when the indexes are stored this way.

An alternate method sorts the nodes in such a way that all nodes with the same pattern index are stored sequentially by id. The only data stored for this method is the first node id of a given pattern. This array will have as many elements as the number of node patterns that are found, which is often $5 \times$ smaller than the number of nodes. When this method is used, a binary search must be performed each time a node’s pattern index is needed. When this method was implemented, the reduction in memory usage was slightly greater than nodes_all_attr. Additionally, the performance suffered due to the time taken by the binary search. As a result it was not pursued further and when the term nodes_all_attr is used in the balance of this thesis it refers to the variation where a node pattern index is stored for each node.
5.3.4 Folding Method 2: Switches Subsets

The previous folding method only compressed the nodes within the RRGraph. This folding method focuses on folding just the switches. In the exploration of the RRGraph it was noticed that the consecutive switches for a given node are often grouped in chunks of the same switch_id. For example, a node may have the switch_id sequence 1, 1, 1, 2, 2, 2, 2, which could be expressed as three 1s followed by four 2s. The switches_subsets folding method takes advantage of these repeated switch patterns by finding instances where a switch pattern is a subset of another, often larger switch pattern. This is best illustrated with the help of Figure 5.5.

This folding method stores a single switches_master_list that contains the data necessary to represent every set of switch_ids. This master list is shown on the lower right side of Figure 5.5. The node_switch_idx array stores each node’s starting index into switches_master_list. The num_edges array specifies how many out-going switches each node has. To determine its switches, Node 0’s switch_id list begins at the third switch of the switches_master_list and continues for 7 switches. Node 1’s switch_id list begins at the fourth switch and continues for 4 switches, and so on for nodes 2 and 3. The number of switches that are a part of each node’s switch_id list is included in the num_edges vector.

The effectiveness of the switches_subsets folding method relies heavily on the presence of consecutive runs of the same switch_id. An example where this method would not work well is the switch_id sequence 2, 3, 1, 7, which would be expressed as one 2, one 3, one 1, one 7. The absence of any consecutive run of matching switch_ids renders this folding method ineffective because fewer subsets exist. Thus, the reduction in memory used by the switches via this method depends largely upon the structure of the device being folded.

Within VTR generated architectures, switch_ids are repeated enough such that the switches_subsets folding method reduces the number of switches represented to essentially zero. F4PGA generated architectures do not fold nearly as well with the switches_subsets method and the percentage of switches that must be represented is around 30% as compared to the nearly 0% required for VTR architectures. Of all the folding methods explored, this folded RRGraph runs in the smallest amount of time for VTR devices while still reducing memory usage.
Table 5.6 gives the runtime and memory usage results of the \texttt{switches\_subsets} folded RRGraph in the form flat / folded. On average, the \texttt{switches\_subsets} Folded RRGraph is about 86\% of the initial RRGraph size. Even though the switches can be folded to take up little space, the switches themselves only make up 25\% of the RRGraph’s overall size, so the overall reduction can be at best 25\% smaller than the original RRGraph. Additionally, the runtime using this Folded RRGraph is about the same as the original representation, but slightly smaller for the \texttt{arm\_core} and \texttt{directrf} benchmarks. This speedup is likely due to the cache benefits discussed in Section 5.1 that are the result of a smaller representation.
Table 5.6: switches_subsets Results in the Form Flat / Folded

<table>
<thead>
<tr>
<th>Category</th>
<th>arm_core</th>
<th>cholesky_mc</th>
<th>directrf</th>
<th>linux_art</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routing (s)</td>
<td>3.5 / 3.5</td>
<td>66.1 / 67.7</td>
<td>1112.7 / 1124.8</td>
<td>277.2 / 283.4</td>
</tr>
<tr>
<td>Lookahead (s)</td>
<td>1.8 / 1.8</td>
<td>79.6 / 82.6</td>
<td>688.9 / 690.0</td>
<td>N/A</td>
</tr>
<tr>
<td>Packing (s)</td>
<td>16.4 / 16.1</td>
<td>91.0 / 91.1</td>
<td>799.6 / 809.8</td>
<td>96.4 / 97.2</td>
</tr>
<tr>
<td>Placement (s)</td>
<td>9.3 / 9.2</td>
<td>121.8 / 122.1</td>
<td>5458.9 / 5398.9</td>
<td>62.6 / 63.6</td>
</tr>
<tr>
<td>Entire VTR flow (s)</td>
<td>34.7 / 34.3</td>
<td>503.4 / 511.5</td>
<td>9189.6 / 9144.5</td>
<td>459.8 / 468.7</td>
</tr>
<tr>
<td>Entire VTR flow (%)</td>
<td>98.85%</td>
<td>101.61%</td>
<td>99.51%</td>
<td>101.94%</td>
</tr>
<tr>
<td>RRGraph Size (MiB)</td>
<td>222 / 182</td>
<td>4808 / 4038</td>
<td>27123 / 22367</td>
<td>3416 / 3207</td>
</tr>
<tr>
<td>RRGraph Size (%)</td>
<td>82.14%</td>
<td>83.99%</td>
<td>82.46%</td>
<td>93.88%</td>
</tr>
<tr>
<td>VTR Memory (MiB)</td>
<td>417 / 377</td>
<td>6940 / 6199</td>
<td>43575 / 38746</td>
<td>4013 / 3804</td>
</tr>
</tbody>
</table>

5.3.5 Folding Method 3: Dest Switch Subsets

Every edge of the RRGraph has a destination node and switch id. This folding method finds patterns of edges (dest_node, switch_id pairs) that match between different nodes. The number of these patterns that can be found is low when the destination nodes are represented absolutely. However, when the destination nodes are stored via an offset from the first destination in a list of edges, many patterns are found.

Every node has one to many pattern indexes that determine the edge patterns that contain the node’s edges. Each edge pattern is comprised of a list of relative offsets with the same switch_id. To determine a node’s out-going edges, offsets inside each of the node’s edge patterns are added to its first destination node.

To illustrate the functionality of the dest_switch_subsets folded RRGraph, the code in Figure 5.6 will be used to find the edges for node 1 from Figure 5.7. First of all, line 3 finds the first destination node, which is 10 for node 1. Line 4 iterates over the pattern ids, starting with pattern 2 and then proceeding to pattern 1. Line 5 assigns the switch id for every edge of a given pattern, which will be switch 0 for the first pattern and switch 2 for the second pattern. The destination nodes are then determined by adding the first
destination node to the destination difference values which are stored for each pattern. For node 1 and pattern 2, this would be $10 + 0 = 10$, $10 + 3 = 13$ and then $10 + 4 = 14$. This yields the first 3 edges of node 1 to have destination nodes 10, 13 and 14 all with switch 0 which matches the flat representation. The remaining edges of node 1 are found by following the same process using its second pattern. In this manner the edges can be built for every node in the RRGraph.

```python
# Obtain all edges (destination node, switch) for every node
for node in nodes:
    first_dest = node["first_dest"]
    for pattern in node["patterns"]:
        switch_id = pattern["switch"]  # the switch is the same for every edge of pattern
        for dest_diff in pattern["dest_diff"]:
            destination_node = first_dest + dest_diff
```

Figure 5.6: Obtain Edges Using dest_switch_subsets Folded RRGraph

Table 5.7 gives the runtime and memory usage results of the dest_switch_subsets folded RRGraph in the form flat / folded. On average, the dest_switch_subsets Folded RRGraph is about 53% of the initial RRGraph size. This folding method reduces size more than the previous folding methods because it is folding both the destination nodes and the switches, which make up more of the RRGraph storage than the nodes or switches on their own. Despite its $2\times$ reduction in size, this folding method has the greatest slowdown thus far, taking 6% longer to run than the flat representation on average. This slowdown may be the result of the increased complexity dest_switch_subsets requires to access the edges of a node. Additionally, node patterns are not necessarily stored in contiguous memory which may result in more cache misses as compared with the original RRGraph.
Figure 5.7: dest_switch_subsets Folded RRGraph
Table 5.7: dest_switch_subsets Results in the Form Flat / Folded

<table>
<thead>
<tr>
<th>Category</th>
<th>arm_core</th>
<th>cholesky_mc</th>
<th>directrf</th>
<th>linux_arty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routing (s)</td>
<td>3.5 / 3.7</td>
<td>66.1 / 74.1</td>
<td>1112.7 / 1280.0</td>
<td>277.2 / 308.0</td>
</tr>
<tr>
<td>Lookahead (s)</td>
<td>1.8 / 1.9</td>
<td>79.6 / 96.8</td>
<td>688.9 / 816.6</td>
<td>N/A</td>
</tr>
<tr>
<td>Packing (s)</td>
<td>16.4 / 16.3</td>
<td>91.0 / 91.4</td>
<td>799.6 / 816.4</td>
<td>96.4 / 99.1</td>
</tr>
<tr>
<td>Placement (s)</td>
<td>9.3 / 9.2</td>
<td>121.8 / 120.6</td>
<td>5458.9 / 5477.8</td>
<td>62.6 / 63.3</td>
</tr>
<tr>
<td>Entire VTR flow (s)</td>
<td>34.7 / 34.9</td>
<td>503.4 / 546.7</td>
<td>9189.6 / 9672.7</td>
<td>459.8 / 508.6</td>
</tr>
<tr>
<td>Entire VTR flow (%)</td>
<td>100.58%</td>
<td>108.60%</td>
<td>105.26%</td>
<td>110.61%</td>
</tr>
<tr>
<td>RRGraph Size (MiB)</td>
<td>222 / 87</td>
<td>4808 / 2767</td>
<td>27123 / 13751</td>
<td>3416 / 2268</td>
</tr>
<tr>
<td>RRGraph Size (%)</td>
<td>39.06%</td>
<td>57.55%</td>
<td>50.70%</td>
<td>66.39%</td>
</tr>
<tr>
<td>VTR Memory (MiB)</td>
<td>417 / 282</td>
<td>6940 / 4928</td>
<td>43575 / 30130</td>
<td>4013 / 2865</td>
</tr>
</tbody>
</table>

Of all the folding methods explored thus far, this method gives the greatest reduction in memory usage for the edges, yielding a 3x reduction in some cases. Similar to the switches_subsets folding method, this method works better when more repeated switches are stored. Because of this, this method works better for VTR than for F4PGA.

5.3.6 Folding Method 4: Nodes and Edges

This folding method combines the nodes_all_attr folding method and a more compact implementation of the dest_switch_subsets folding method in order to arrive at the smallest Folded RRGraph that this work presents. No visual representation is provided for this folding method, but the RRGraph represents the nodes as given in Figure 5.4 and the edges as given in Figure 5.7.

Table 5.8 gives the runtime and memory usage results of the nodes_edges folded RRGraph in the form flat / folded. On average, the nodes_edges Folded RRGraph is about 42% of the initial RRGraph size, with the RRGraph for the arm_core taking up 4x less space than the flat representation. This folding method produces the smallest RRGraphs of the folding methods, but also takes the longest amount of time to run averaging 13% longer runtime than the original RRGraph.
Table 5.8: nodes_edges Results in the Form Flat / Folded

<table>
<thead>
<tr>
<th>Category</th>
<th>arm_core</th>
<th>cholesky_mc</th>
<th>directrf</th>
<th>linux_arty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routing (s)</td>
<td>3.5 / 4.0</td>
<td>66.1 / 84.0</td>
<td>1112.7 / 1439.3</td>
<td>277.2 / 330.8</td>
</tr>
<tr>
<td>Lookahead (s)</td>
<td>1.8 / 2.2</td>
<td>79.6 / 125.7</td>
<td>688.9 / 1098.5</td>
<td>N/A</td>
</tr>
<tr>
<td>Packing (s)</td>
<td>16.4 / 16.4</td>
<td>91.0 / 93.9</td>
<td>799.6 / 815.8</td>
<td>96.4 / 97.2</td>
</tr>
<tr>
<td>Placement (s)</td>
<td>9.3 / 9.3</td>
<td>121.8 / 123.1</td>
<td>5458.9 / 5433.1</td>
<td>62.6 / 62.8</td>
</tr>
<tr>
<td>Entire VTR flow (s)</td>
<td>34.7 / 36.0</td>
<td>503.4 / 612.9</td>
<td>9189.6 / 10193.7</td>
<td>459.8 / 535.8</td>
</tr>
<tr>
<td>Entire VTR flow (%)</td>
<td>103.75%</td>
<td>121.75%</td>
<td>110.93%</td>
<td>116.53%</td>
</tr>
<tr>
<td>RRGraph Size (MiB)</td>
<td>2212 / 56</td>
<td>4808 / 2437</td>
<td>27123 / 11588</td>
<td>3416 / 1724</td>
</tr>
<tr>
<td>RRGraph Size (%)</td>
<td>25.26%</td>
<td>50.69%</td>
<td>42.72%</td>
<td>50.48%</td>
</tr>
<tr>
<td>VTR Memory (MiB)</td>
<td>417 / 251</td>
<td>6940 / 4584</td>
<td>43575 / 27968</td>
<td>4013 / 2321</td>
</tr>
</tbody>
</table>

Table 5.9: Size of Different F4PGA RRGraphs.

<table>
<thead>
<tr>
<th>RRGraph Size</th>
<th>nodes_all_attr</th>
<th>switches_subsets</th>
<th>nodes_edges</th>
<th>dest_switch</th>
<th>flat</th>
</tr>
</thead>
<tbody>
<tr>
<td>50t (MiB)</td>
<td>2869.30</td>
<td>3206.80</td>
<td>1724.40</td>
<td>2268.00</td>
<td>3416.00</td>
</tr>
<tr>
<td>100t (MiB)</td>
<td>5570.00</td>
<td>6092.90</td>
<td>3349.80</td>
<td>4369.70</td>
<td>6582.90</td>
</tr>
<tr>
<td>200t (MiB)</td>
<td>12504.20</td>
<td>13478.10</td>
<td>7468.80</td>
<td>9685.10</td>
<td>14694.90</td>
</tr>
<tr>
<td>50t (%)</td>
<td>84.00%</td>
<td>93.88%</td>
<td>50.48%</td>
<td>66.39%</td>
<td>100.00%</td>
</tr>
<tr>
<td>100t (%)</td>
<td>84.61%</td>
<td>92.56%</td>
<td>50.89%</td>
<td>66.38%</td>
<td>100.00%</td>
</tr>
<tr>
<td>200t (%)</td>
<td>85.09%</td>
<td>91.72%</td>
<td>50.83%</td>
<td>65.91%</td>
<td>100.00%</td>
</tr>
</tbody>
</table>

5.4 Scaling F4PGA Devices

The F4PGA devices that have been used in this work range in size with the xc7a50t_-test device being the smallest and the xc7a200t_test device being the largest. The linux_arty benchmark used the xc7a50t_test device, but benchmarks have also been run on the other devices. Table 5.9 shows the absolute and relative reduction in memory usage of several folded F4PGA RRGraphs as compared with the flat representation. Overall,
the RRGraph size percentage of the initial RRGraph size when using a given folding method remains fairly constant between the devices. For example, the percentage of initial size for RRGraphs using dest_switch_subsets varies from 66.39% to 65.91% which is not a large range. Similar to VTR devices, the nodes_edges folding method gives the greatest memory reduction, with a 2× reduction in size.

5.5 Comparing All RRGraph Representations

The average memory usage and runtime of each RRGraph representation is given in Figure 5.8 with values being computed by taking the average of the “Entire VTR flow (%)” and “RRGraph Size (%)” rows of the tables discussed above. The general trend is that as memory usage is decreased via folding, the runtime increases. This trend may be due to the additional instructions required to access more complex RRGraph representations. The switches_subsets and flat RRGraphs have about the same runtime, with the switches_subsets using less memory. Similarly, the nodes_all_attr and dest_switch_subsets RRGraphs run in about the same amount of time with dest_switch_subsets being much smaller.

Each of the RRGraph representation methods explored in this chapter comes with unique pros and cons. switches_subsets has the best performance of the folded methods for VTR benchmarks. nodes_edges has the greatest memory reduction for every benchmark, but with reduced performance. When the nodes are the only thing being folded as in nodes_all_attr, memory reduction is limited and runtime is increased. Due to this observation, folding the edges is a crucial part of reducing the memory usage of the RRGraph. Moreover, F4PGA devices proved to be more difficult to fold than VTR devices. This was expected to some degree at the beginning of this research due to the inherent complexity of representing a commercial device with all of the irregularities found therein.

Overall, there was a slight reduction in memory usage with little increase in runtime when folding the RRGraph with switches_subsets. For other folding methods, the complexity of folding the graph led to an increase in runtime. Appendix C includes cache performance results that show that some folding methods had less cache misses than the flat representation. Because of the folded graph’s complexity, however, they also had more cache
references than the flat representation. Thus, reducing the memory used by the RRGraph does lead to increased performance of the CPU cache, but more complex accesses often times increase the runtime by a greater amount than the smaller memory usage reduces it.

Figure 5.8: RRGraph Comparison
CHAPTER 6. CONCLUSION

The task of Folding VTR’s Routing Resource Graph presents a trade off between memory usage and performance. Of the many possibilities for folding, this work has implemented over 15 different variations of folding the RRGraph, seeking to find the folding method that optimized both memory usage and performance as compared to the RRGraph before any folding. Of the 15 folding variations mentioned, 4 have been presented in this work. Results for both memory usage and runtime vary depending on the device in use and the circuit being implemented. Smaller devices tend to have a greater performance increase using the folded RRGraphs as compared to larger devices. This is likely due to the fact that folding a small device often reduces its size such that the folded representation fits in the cache more completely, whereas this is less likely for large devices. Size reduction results depend largely on the architecture being compressed and the degree to which it can be folded. Devices with more repeated data for nodes and edges fold to smaller sizes than devices that contain fewer patterns.

This chapter will summarize the results of this work and detail the contributions. The impact of this work will be discussed, and some future work will be suggested.

6.1 Understanding the Slowdown Due to Using Folded RRGraphs

A large portion of this work has presented the idea that if the RRGraph could be reduced in size, the performance would also increase. If this were true, then why did the smallest RRGraph also have the longest average runtime? The answer to this question is not obvious, but there are several factors that may partially give an explanation.

First of all, the more compact representations often take more instructions to access. For example, the \texttt{nodes\_all\_attr} has a pattern index that must first be accessed and then a node’s attributes can be obtained using that index. Although this may seem like a
small difference when compared to the flat representation, accessing a node’s data happens frequently enough that this extra instruction makes a difference.

Secondly, folded representations often store data in a manner that is not sequential. For example, the edge pattern indexes used in the `dest_switch_subsets` folding method have no specific ordering and thus the edge patterns they access may not be stored next to each other in memory. Because of this, `dest_switch_subsets` does not inherently take advantage of spatial locality.

Finally, although it has been optimized greatly already, there is still room to improve the code within VTR that accesses each Folded RRGraph. The initial implementation of `dest_switch_subsets` introduced a 30% slowdown in VTR for one of the benchmarks. After altering how data structures were being represented and optimizing algorithms that accessed edges, the slowdown was reduced to 8%. With more effort, the implementation that accesses each folded RRGraph could be optimized further.

Each of these factors plays a role in explaining the slowdown seen due to using some of the folded RRGraphs.

### 6.2 Potential Use Models

There are several use models for using folded RRGraphs that each cater to different requirements. The use model presented in this work involves storing the RRGraph in a folded manner and using it within VTR in its folded state. This provides the benefits of reduced disk space needed to store the RRGraph, reduced RAM used during VTR’s runtime, and in some cases a slight speedup.

Another use model would be to store the RRGraph in a folded state, and then unfold the graph into the flat representation at the beginning of VTR’s flow. This would allow for the RRGraph to be stored on disk at 2-4× smaller than its original size, but would still give similar performance as the original RRGraph as the unfolding would take little time.
6.3 Contributions and Impact

This work has spanned several open source projects and contributed improvements that may impact the future use models of both VTR and F4PGA. Major contributions are listed briefly below, with larger explanations given in the Sections 6.3.1 through 6.3.4.

1. This work helped implement an API for accessing the RRGraph in VTR.

2. New knowledge was introduced regarding the foldability of VTR and F4PGA devices.

3. Memory usage of RRGraphs was reduced by $2\times$ or more while maintaining similar performance.

4. An infrastructure was provided within VTR for further RRGraph folding research to continue in the future.

6.3.1 RRGraph API Implementation

The development and implementation of an API for the Routing Resource Graph was discussed in Chapter 4. All of the benchmarking and usage of folded RRGraphs within VTR was only possible because of the API which was developed. Additionally, the API introduces the possibility of VTR using other RRGraph representations in the future as long as they provide the API with the necessary information. Overall, the implementation of an API for RRGraph creation and access increases the flexibility of VTR.

6.3.2 Knowledge of VTR and F4PGA Devices

Section 5.2 and the entirety of Chapter 3 provide new knowledge regarding several comparisons between VTR and F4PGA RRGraphs. Research was performed to determine the uniformity and repetition within each set of devices in addition to comparing memory usage and data member count. These data points give insight into each method of representing the Routing Resource Graph.
6.3.3 Reduced Memory Usage of RRGraphs

The end goal of this research was to explore potential memory and runtime reductions via compressing the large data structure that is the RRGraph. In this process over 15 different folding variations were implemented, each with unique memory usage and performance metrics. Ultimately a folding method was found that yields on average a 2× memory reduction and little performance slowdown. Some other folding methods yield greater memory reduction with more slowdown while others yield better performance with little memory reduction. There are many trade-offs to be considered while folding the RRGraph, and the ideal representation depends on the use case at hand. Use cases where memory is critical may favor the more compact RRGraphs, while performance-based applications with no memory requirement would favor the flat RRGraph.

6.3.4 Infrastructure for Future RRGraph Folding

Another contribution of this work includes an infrastructure within VTR to make future graph folding research simple and streamlined. Several scripts have been included in a public GitHub repository [28] that perform some of the graph folding methods discussed thus far. Documentation concerning the necessary steps to implement a new folded RRGraph within VTR has also been included in the mentioned repository. Each of these items will help future developers that desire to explore other folding methods.

6.4 Future Work

As this research concludes, there remain several items of future work that may be pursued, which vary in depth and impact.

6.4.1 Implementation into F4PGA

One of the more simple tasks which remains is that of implementing the folded RRGraph into F4PGA. The current version of VTR in use by F4PGA is a fork of VTR’s main repository, so there currently exists only a partial API within it. If it were desired, updating
F4PGA’s version of VTR to be current with VTR master should not be overly difficult, and would introduce new opportunities for research.

6.4.2 RRGraph Folding

Despite the many folding methods which have been implemented over the course of this work, unexplored graph folding methods still exist which may give improved results. Additionally, there may be room for optimization of the code that accesses the folded RRGraphs presented in this work. With the RRGraph API and RRGraph folding infrastructure, new folding approaches should be relatively straightforward to implement and experiment with.

6.4.3 Further Development of the API

This work has helped implement an API within VTR that is currently in its infant stages, and much work remains to polish and refine it. Many functions within the API are still very low level and could be replaced by higher level functions with some work. In addition, other RRGraph representations, such as F4PGA’s representation, could be considered and the API could potentially cater to them by using higher level API calls that are not as closely tied to the RRGraph representation in use.

For example, VTR currently requires individual nodes to only traverse in the X or Y direction. A higher level API could allow for nodes to be L-shaped, for example, and hide the fact that VTR represents L-shaped wires by using nodes of type CHANX and CHANY connected by a non-configurable switch. With a change like this in place, more complex wires that are common in F4PGA devices could be represented as a single node instead of multiple nodes connected by non-configurable switches that are always on.

6.5 Conclusion

Over the course of this research, different RRGraph representations have been studied, an API has been developed, and RRGraph folding has been explored and implemented to achieve a 2× memory reduction while maintaining performance. The greatest memory savings increase up to 4× reduction for some devices, while yielding a slight increase in
runtime. Cache benefits have been observed as a result of RRGraph folding, but some devices are so large that the folded RRGraph is still too large to yield cache benefits. Overall, there are many trade-offs involved in RRGraph folding and the individual use case and constraints must be considered when choosing the RRGraph representation.
REFERENCES


APPENDIX A. DATA SETS

A.1 Switches for xc7a50t_test Device

```xml
<switch id="0" name="__vpr_delayless_switch__" type="mux"><timing/>
```

```xml
<switch id="1" name="buffer" type="mux"><timing Cin="7.70000012e-16" Cout="4.00000001e-15" R="551" Tdel="1.78000004e-10"/>
```

```xml
<switch id="2" name="routing_R0_C0_Tdel14.317513749999994e-11" type="mux"><timing Tdel="4.31751371e-11"/>
```

```xml
<switch id="3" name="routing_R0_C0_C0_Tdel1.375e-13" type="mux"><timing Tdel="1.37499997e-13"/>
```

```xml
<switch id="4" name="routing_R0_0_C0_0_Tdel1e-12" type="mux"><timing Tdel="9.99999996e-13"/>
```

```xml
...
```

```xml
<switch id="232" name="routing_R0_00171220225_C0_Tdel1e-12" type="mux"><timing
    R="0.00171220221" Tdel="9.99999996e-13"/>
```

```xml
<switch id="233" name="routing_R0_001337678375_C0_Tdel1e-12" type="mux"><timing
    R="0.0013376784" Tdel="9.99999996e-13"/>
```

```xml
<switch id="234" name="routing_R0_01e-12_Tdel1e-12" type="mux"><timing
    Cinternal="9.99999996e-13" Tdel="9.99999996e-13"/>
```

```xml
<switch id="235" name="routing_R0_C0_Tdel11.754943508222875e-38" type="mux"><timing Tdel="1.17549435e-38"/>
```

```xml
<switch id="236" name="routing_R0_001375_C0_Tdel0.0" type="mux"><timing R="0.00137499999"/>
```

Figure A.1: First 5 and Last 5 Switch Types for xc7a50t_test
A.2 Switches for stratix_arch Device

Figure A.2: All Switch Types for stratix_arch

```
<switch id="0" name="_vpr_delayless_switch_" type="mux"><timing/>
<switch id="1" name="ipin_cblock" type="mux"><timing Cin="1.4699995e-15" R="2231.5"/>
<switch id="2" name="seg4_driver" type="mux"><timing Cin="6.00000023e-16" Cout="4.81999991e-15" R="450" Tdel="5.89999993e-11"/>
<switch id="3" name="seg16_driver" type="mux"><timing Cin="1.8000007e-15" Cout="1.44999996e-14" R="150" Tdel="8.69999975e-11"/>
```
APPENDIX B. BENCHMARK RESULTS

Additional benchmarks beyond those given in Chapter 5 are included here. These benchmarks were run on the F4PGA xc7a50t_test device.

Table B.1: Benchmark Descriptions

<table>
<thead>
<tr>
<th>Name</th>
<th>description</th>
<th>device</th>
<th>LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>minilitex</td>
<td>LiteX-based SoC</td>
<td>xc7a50t_test</td>
<td>6,323</td>
</tr>
<tr>
<td>picosoc_50</td>
<td>SoC with picorv32 CPU running @50MHz</td>
<td>xc7a50t_test</td>
<td>4,110</td>
</tr>
<tr>
<td>picosoc_100</td>
<td>SoC with picorv32 CPU running @100MHz</td>
<td>xc7a50t_test</td>
<td>4,087</td>
</tr>
</tbody>
</table>

Table B.2: nodes_all_attr Results in the Form Flat / Folded

<table>
<thead>
<tr>
<th>Category</th>
<th>minilitex_arty</th>
<th>picosoc_50</th>
<th>picosoc_100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routing (s)</td>
<td>173.8 / 180.4</td>
<td>42.9 / 43.4</td>
<td>110.0 / 113.7</td>
</tr>
<tr>
<td>Packing (s)</td>
<td>30.9 / 32.3</td>
<td>21.5 / 21.9</td>
<td>21.0 / 21.5</td>
</tr>
<tr>
<td>Placement (s)</td>
<td>19.6 / 19.3</td>
<td>11.5 / 11.3</td>
<td>14.0 / 13.8</td>
</tr>
<tr>
<td>Entire VTR flow (s)</td>
<td>238.9 / 244.3</td>
<td>89.6 / 87.7</td>
<td>158.6 / 160.1</td>
</tr>
<tr>
<td>RRGraph Size (MiB)</td>
<td>3654.4 / 3107.5</td>
<td>3687.3 / 3141.3</td>
<td>3687.7 / 3141.2</td>
</tr>
<tr>
<td>VTR Memory (MiB)</td>
<td>3889.6 / 3342.6</td>
<td>3861.2 / 3315.3</td>
<td>3861.0 / 3315.3</td>
</tr>
</tbody>
</table>
Table B.3: switches_subsets Results in the Form Flat / Folded

<table>
<thead>
<tr>
<th>Category</th>
<th>minilitex_arty</th>
<th>picosoc_50</th>
<th>picosoc_100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routing (s)</td>
<td>173.8 / 172.6</td>
<td>42.9 / 42.3</td>
<td>110.0 / 110.4</td>
</tr>
<tr>
<td>Packing (s)</td>
<td>30.9 / 30.8</td>
<td>21.5 / 21.1</td>
<td>21.0 / 20.7</td>
</tr>
<tr>
<td>Placement (s)</td>
<td>19.6 / 19.6</td>
<td>11.5 / 11.3</td>
<td>14.0 / 14.0</td>
</tr>
<tr>
<td>Entire VTR flow (s)</td>
<td>238.9 / 237.5</td>
<td>89.6 / 88.2</td>
<td>158.6 / 158.8</td>
</tr>
<tr>
<td>RRGraph Size (MiB)</td>
<td>3654.4 / 3445.4</td>
<td>3687.3 / 3478.8</td>
<td>3687.7 / 3479.1</td>
</tr>
<tr>
<td>VTR Memory (MiB)</td>
<td>3889.6 / 3680.5</td>
<td>3861.2 / 3652.5</td>
<td>3861.0 / 3653.1</td>
</tr>
</tbody>
</table>

Table B.4: dest_switch_subsets Results in the Form Flat / Folded

<table>
<thead>
<tr>
<th>Category</th>
<th>minilitex_arty</th>
<th>picosoc_50</th>
<th>picosoc_100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routing (s)</td>
<td>173.8 / 194.0</td>
<td>42.9 / 50.6</td>
<td>110.0 / 123.9</td>
</tr>
<tr>
<td>Packing (s)</td>
<td>30.9 / 32.1</td>
<td>21.5 / 21.9</td>
<td>21.0 / 21.4</td>
</tr>
<tr>
<td>Placement (s)</td>
<td>19.6 / 19.5</td>
<td>11.5 / 11.2</td>
<td>14.0 / 13.8</td>
</tr>
<tr>
<td>Entire VTR flow (s)</td>
<td>238.9 / 273.3</td>
<td>89.6 / 110.4</td>
<td>158.6 / 185.8</td>
</tr>
<tr>
<td>RRGraph Size (MiB)</td>
<td>3654.4 / 2502.3</td>
<td>3687.3 / 2536.3</td>
<td>3687.7 / 2536.2</td>
</tr>
<tr>
<td>VTR Memory (MiB)</td>
<td>3889.6 / 2737.4</td>
<td>3861.2 / 2710.4</td>
<td>3861.0 / 2710.3</td>
</tr>
</tbody>
</table>

Table B.5: nodes_edges Results in the Form Flat / Folded

<table>
<thead>
<tr>
<th>Category</th>
<th>minilitex_arty</th>
<th>picosoc_50</th>
<th>picosoc_100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routing (s)</td>
<td>173.8 / 205.3</td>
<td>42.9 / 53.6</td>
<td>110.0 / 129.6</td>
</tr>
<tr>
<td>Packing (s)</td>
<td>30.9 / 31.2</td>
<td>21.5 / 21.4</td>
<td>21.0 / 20.9</td>
</tr>
<tr>
<td>Placement (s)</td>
<td>19.6 / 19.4</td>
<td>11.5 / 11.3</td>
<td>14.0 / 13.8</td>
</tr>
<tr>
<td>Entire VTR flow (s)</td>
<td>238.9 / 289.6</td>
<td>89.6 / 119.2</td>
<td>158.6 / 196.9</td>
</tr>
<tr>
<td>RRGraph Size (MiB)</td>
<td>3654.4 / 1961.3</td>
<td>3687.3 / 1992.0</td>
<td>3687.7 / 1994.4</td>
</tr>
<tr>
<td>VTR Memory (MiB)</td>
<td>3889.6 / 2196.4</td>
<td>3861.2 / 2166.0</td>
<td>3861.0 / 2168.5</td>
</tr>
</tbody>
</table>
APPENDIX C. CACHE REFERENCE RESULTS

Results regarding the number of last-level cache references ("LLC Refs") and missed last-level cache references ("Missed LLC Refs") are included in the tables below for several of the benchmarks defined in Chapter 5. The "Missed LLC Refs (%)" row refers to the ratio of "Missed LLC Refs" / "LLC Refs". The folding methods included as columns in these tables use the following naming convention.

- Nodes - Nodes All Attributes Folding Method
- Switches - Switches Subsets Folding Method
- Edges - Dest Switch Subsets Folding Method
- All - Nodes and Edges Folding Method
- Flat - Original Representation

Table C.1: Cache References for k6_arm_core

<table>
<thead>
<tr>
<th>Category</th>
<th>Nodes</th>
<th>Switches</th>
<th>Edges</th>
<th>All</th>
<th>Flat</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLC Refs</td>
<td>8.53e+09</td>
<td>8.26e+09</td>
<td>8.58e+09</td>
<td>8.66e+09</td>
<td>8.43e+09</td>
</tr>
<tr>
<td>Missed LLC Refs</td>
<td>5.00e+08</td>
<td>4.93e+08</td>
<td>4.83e+08</td>
<td>4.61e+08</td>
<td>5.23e+08</td>
</tr>
<tr>
<td>Missed LLC Refs (%)</td>
<td>5.86%</td>
<td>5.97%</td>
<td>5.63%</td>
<td>5.32%</td>
<td>6.21%</td>
</tr>
<tr>
<td>VTR Entire Flow (s)</td>
<td>34.97</td>
<td>34.27</td>
<td>34.88</td>
<td>36.05</td>
<td>35.00</td>
</tr>
</tbody>
</table>
Table C.2: Cache References for stratixiv_cholesky

<table>
<thead>
<tr>
<th>Category</th>
<th>Nodes</th>
<th>Switches</th>
<th>Edges</th>
<th>All</th>
<th>Flat</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLC Refs</td>
<td>1.44e+11</td>
<td>1.28e+11</td>
<td>1.41e+11</td>
<td>1.63e+11</td>
<td>1.29e+11</td>
</tr>
<tr>
<td>Missed LLC Refs</td>
<td>2.87e+10</td>
<td>2.83e+10</td>
<td>3.13e+10</td>
<td>3.32e+10</td>
<td>2.88e+10</td>
</tr>
<tr>
<td>Missed LLC Refs (%)</td>
<td>19.97%</td>
<td>22.12%</td>
<td>22.16%</td>
<td>20.41%</td>
<td>22.28%</td>
</tr>
<tr>
<td>VTR Entire Flow (s)</td>
<td>529.48</td>
<td>511.53</td>
<td>546.70</td>
<td>612.90</td>
<td>508.10</td>
</tr>
</tbody>
</table>

Table C.3: Cache References for directrf

<table>
<thead>
<tr>
<th>Category</th>
<th>Nodes</th>
<th>Switches</th>
<th>Edges</th>
<th>All</th>
<th>Flat</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLC Refs</td>
<td>2.16e+12</td>
<td>1.98e+12</td>
<td>2.12e+12</td>
<td>2.30e+12</td>
<td>1.99e+12</td>
</tr>
<tr>
<td>Missed LLC Refs</td>
<td>8.67e+11</td>
<td>8.43e+11</td>
<td>8.79e+11</td>
<td>9.01e+11</td>
<td>8.45e+11</td>
</tr>
<tr>
<td>Missed LLC Refs (%)</td>
<td>40.21%</td>
<td>42.57%</td>
<td>41.52%</td>
<td>39.23%</td>
<td>42.45%</td>
</tr>
<tr>
<td>VTR Entire Flow (s)</td>
<td>9706.41</td>
<td>9144.55</td>
<td>9672.72</td>
<td>10193.72</td>
<td>9189.60</td>
</tr>
</tbody>
</table>

Table C.4: Cache References for linux_arty

<table>
<thead>
<tr>
<th>Category</th>
<th>Nodes</th>
<th>Switches</th>
<th>Edges</th>
<th>All</th>
<th>Flat</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLC Refs</td>
<td>8.75e+10</td>
<td>8.48e+10</td>
<td>9.65e+10</td>
<td>1.03e+11</td>
<td>8.47e+10</td>
</tr>
<tr>
<td>Missed LLC Refs</td>
<td>9.90e+09</td>
<td>1.06e+10</td>
<td>1.15e+10</td>
<td>1.14e+10</td>
<td>1.05e+10</td>
</tr>
<tr>
<td>Missed LLC Refs (%)</td>
<td>11.32%</td>
<td>12.53%</td>
<td>11.87%</td>
<td>11.07%</td>
<td>12.42%</td>
</tr>
<tr>
<td>VTR Entire Flow (s)</td>
<td>479.98</td>
<td>469.75</td>
<td>509.25</td>
<td>534.63</td>
<td>460.84</td>
</tr>
</tbody>
</table>