2015-03-01

Preemptive Placement and Routing for In-Field FPGA Repair

Joshua E. Jensen

Brigham Young University - Provo

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

Part of the Electrical and Computer Engineering Commons

BYU ScholarsArchive Citation

https://scholarsarchive.byu.edu/etd/4417

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.
Preemptive Placement and Routing for In-Field FPGA Repair

Joshua E. Jensen

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

Master of Science

Michael J. Wirthlin, Chair
Brad L. Hutchings
Brent E. Nelson

Department of Electrical and Computer Engineering
Brigham Young University
March 2015

Copyright © 2015 Joshua E. Jensen
All Rights Reserved
ABSTRACT

Preemptive Placement and Routing for In-Field FPGA Repair

Joshua E. Jensen
Department of Electrical and Computer Engineering, BYU
Master of Science

With the growing density and shrinking feature size of modern semiconductors, it is increasingly difficult to manufacture defect free semiconductors that maintain acceptable levels of reliability for long periods of time. These systems are increasingly susceptible to wear-out by failing to meet their operational specifications for an extended period of time. The reconfigurability of FPGAs can be used to repair post-manufacturing faults by configuring the FPGA to avoid a damaged resource. This thesis presents a method for preemptively preparing to repair FPGA devices with wear-out faults by precomputing a set of repair circuits that, collectively, can repair a fault found in any logic block of the FPGA. This approach relies on logic placement and routing to create “repair” circuits that avoid specific logic blocks. These repairs can be used when a specific resource has failed. New placement and routing algorithms are proposed for generating such repair circuits. The number of repairs needed to create a complete repair set depends heavily on the utilization of the FPGA resources. The algorithms are tested against several benchmarks and with multiple area constraints for each benchmark. Using this work, on average 20 repair configurations were needed to repair 99% of permanent faults.

Keywords: FPGA, Repair, Fault-Tolerance, Placement, Routing
I would like to thank my advisor, Dr. Michael Wirthlin, for giving me the opportunity to pursue my goals as a graduate student. His advice, support and dedication made this work possible.

I also want to thank my committee members and all of the faculty and staff of the BYU Department of Electrical and Computer Engineering. They have been great examples and their instruction helped me learn and grow as a student.

Most importantly, I want to thank my wife Hannah for her unceasing love and support. Her patience and encouragement motivated me in this work. She is my inspiration.

This work has been sponsored by Cisco Systems and by the I/UCRC Program of the National Science Foundation under Grant No. 1265957.
# Table of Contents

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Summary of Approach ................................................. 2
1.2 Key Results ................................................................. 3
1.3 Outline ............................................................... 3

2 Background and Related Work 4

2.1 Faults ............................................................................. 4
   2.1.1 Permanent Faults ......................................................... 4
   2.1.2 Transient Faults ........................................................... 5
2.2 Overview of Approach .................................................. 6
   2.2.1 Benchmark Designs .................................................... 8
   2.2.2 Repair Placement and Routing Objectives .................... 9
2.3 Related Work .............................................................. 11

3 FPGA Architecture and Mapping Tools 13

3.1 FPGA Architecture ...................................................... 13
   3.1.1 Virtex-4 Overview ..................................................... 14
   3.1.2 CLB Tiles .............................................................. 14
# Conclusion

6.1 Future Work ......................................................... 60

**Bibliography** 62

**A Placement in RapidSmith** 65

A.1 Simulated Annealing Placer ......................................... 65

A.1.1 Key Classes ....................................................... 65

A.1.2 Key Steps .......................................................... 67

A.1.3 Code ............................................................... 67

**B Routing in RapidSmith** 74

B.1 PathFinder Routing ................................................... 74

B.1.1 Data Structures .................................................... 74

B.1.2 Key Classes ........................................................ 75

B.1.3 Key Steps .......................................................... 77

B.1.4 Code ............................................................... 77

**C Additional Routing Approach** 86

C.1 Shadow Routing ....................................................... 86

C.1.1 Algorithm Overview ............................................... 87
List of Tables

2.1 Benchmark Circuits ................................................. 9
4.1 Baseline and Vendor Placement Tools ............................ 29
4.2 Bounding Box Area Constraints .................................. 32
4.3 Number of Repair Circuits using Naive Repair Placement .. 36
4.4 Number of Repair Circuits using Cost Repair Placement .... 40
4.5 Number of Shadow Sites and Execution Time of Shadow Placement .... 45
4.6 Hybrid Placer Execution Time .................................... 48
4.7 Double Faults ....................................................... 49
4.8 Repair Placement Algorithms Result Summary .................. 49

5.1 Baseline and Vendor Routing Tools ............................. 53
5.2 Cost Repair Router Execution Time ............................... 57
5.3 Cost Using Cost Repair Router ................................... 57
5.4 Number of PIPs Using Cost Repair Router ....................... 58
5.5 Minimum Clock Period(ns) using Cost Repair Router .......... 58
5.6 Percentage of TWCs repaired using Cost Repair Router ........ 58
## List of Figures

2.1 FPGA Repair ......................................................... 7
2.2 Initial and Repair Configurations ................................. 7
2.3 Operational Repair .................................................. 8
2.4 Server Repairing ...................................................... 9

3.1 FPGA Architecture .................................................. 14
3.2 Configurable Logic Block (CLB) ................................. 15
3.3 Diagram of SLICEL ................................................... 16
3.4 DSP Tile .............................................................. 17
3.5 BRAM Tiles in Device .............................................. 18
3.6 Programmable Switch Matrix ..................................... 19
3.7 Long Lines .......................................................... 20
3.8 Xilinx Design Flow .................................................. 21
3.9 Design Flow with Custom Place and Route ..................... 22
3.10 RapidSmith and XDL .............................................. 22

4.1 Design Flow for Baseline Placement ............................ 25
4.2 Bounding Box ........................................................ 26
4.3 Relationally Placed Groups ....................................... 28
4.4 Two Placements for the Multxor Design ........................ 31
4.5 Repair Circuits for 67% Utilized Design ....................... 31
4.6 Artificial Area Constraint ......................................... 33
4.7 Repair Example ....................................................... 34
Chapter 1

Introduction

It is increasingly difficult to design and manufacture semiconductor systems that maintain acceptable levels of reliability while simultaneously taking advantage of the lower power, higher operating speeds, and greater density offered by modern sub-micron fabrication technologies [1]. There is also greater variation in transistor parameters as the technology node decreases causing a greater circuit failure rate. In addition, the effects of wear-out due to electromigration are more pronounced at smaller geometries [2]. To obtain the full benefits of future technologies, future semiconductor systems must tolerate a greater number and variety of permanent faults. The lifetime of these systems could be significantly increased if they are designed to support some form of repair to address manufacturing faults or wear-out faults that occur after manufacturing.

Because of the fine-grain reconfigurability of FPGAs, it is possible to address such wear-out faults by creating a new FPGA configuration for a specific design with a modified placement and routing of a specific design. This configuration, called a repair configuration, performs the same function as the original circuit but is placed and routed in such a way to avoid the resource or resources that have permanently failed. There are many previous efforts that investigate ways of repairing an FPGA by reconfiguring the circuit [3, 4, 5, 6]. These papers exploit the fact that there are many unused resources in a mapped FPGA design, and repair circuits can take advantage of these unused resources to avoid faulty resources. The various approaches in the papers differ in terms of percentage of permanent faults tolerated, execution time to generate a repair configuration, and the cost of overhead of the repair. This thesis presents a new approach that anticipates any possible fault before it occurs and does so in a reasonable amount of time with little cost overhead.
1.1 Summary of Approach

This thesis presents a technique for generating a set of circuits that tolerates permanent faults within an FPGA. Each circuit is called a “repair”. A single repair circuit can generally tolerate a large number of permanent faults, but multiple circuits are needed to anticipate all possible permanent faults. This thesis identifies new methods to generate a complete set of repair circuits for before the fault occurs. Most other approaches create the repair circuit after the fault occurs. A large set of pre-computed repair circuits, called a “repair set” is created before the system is deployed and made available for use after a permanent fault is found. When a fault occurs and its location identified, one of the pre-generated repair circuits is configured onto the FPGA to “repair” the permanent fault. Anticipating every possible permanent fault requires a large amount of computation time upfront, however, when there is a permanent fault, the FPGA can immediately be reconfigured to tolerate that fault.

The objective of this work is to generate a repair set for all FPGA faults including faults within the programmable logic, fixed functions (DSP, BRAM, etc.), and routing resources. The primary mechanism for repairing logic and fixed function blocks is through circuit placement. Placement is the process of assigning logic and fixed function blocks to physical locations on the device. Multiple circuit placements are performed to produce a placement that will avoid each allocated resource of a design mapped to a particular device. After placement the circuit is ready to be routed. Routing uses wires and programmable interconnect to connect the logic in the design. Routing resources are repaired during routing by generating multiple configurations that avoid different routing resources.

This work is based on an in-house placement and routing infrastructure that targets the commercially available Xilinx Virtex-4 family of FPGAs. The placer and router successfully performs FPGA placement and routing that generates valid configurations for the Virtex-4 family. Several repair placement algorithms were developed from this baseline placer to generate repair placement configurations. One repair routing algorithm was developed from the baseline router to route the placements generated by the repair placers. This thesis will present four repair placement approaches and one repair routing approach and contrast the execution time and quality of the results.
1.2 Key Results

This work increases the lifetime of FPGAs by tolerating 99% of permanent faults using 20 repair configurations with only 5% reduction in placement and routing quality. A complete set of repair configurations could be generated without the placement and routing algorithms presented in this thesis, but would require more time. This could be done by running placement and routing for each resource in the original design and explicitly restricting the placement and router. Xilinx tools may be capable of avoiding specific resources, but the option is not publicly available. However, this approach would linearly increase in run time as the size of the design increases. The placement and routing algorithms presented in this thesis are able to speed up execution time of the process by reusing place and route between iterations while generating repairs.

The primary contribution of this thesis is the ability to anticipate every possible fault and preemptively generate corresponding repair configurations before failure occurs. This was done by creating (1) a basic placer, (2) a basic router, (3) four placement approaches that generate placement configurations for repair, and (4) one routing approach that generates routed configurations for repair.

1.3 Outline

The remainder of this thesis is organized as follows. Chapter 2 discusses background information on permanent faults and provides an overview of the approach used in this thesis. Related work is also discussed. Chapter 3 presents an overview of FPGA architecture and provides additional background information explaining the tools used to develop the algorithms presented in this thesis. Chapter 4 introduces the placement process and the repair placement approaches. Chapter 5 introduces the routing process and describes the repair routing approaches. Chapter 6 concludes this thesis.
Chapter 2

Background and Related Work

The purpose of this thesis is to provide new methods to tolerate permanent faults to increase the availability and lifetime of a device. Although this thesis focuses solely on permanent faults, it is important to understand the difference between types of faults. This chapter describes the types of faults and possible causes for those faults in FPGAs. A summary of approach of the methods is given, and the benchmark designs to test the novel methods introduced in this work are also discussed.

Faults within an FPGA are not a new problem and many fault tolerant techniques have been developed. The final section of this chapter discusses several techniques related to the method presented in this thesis and contrasts the different approaches.

2.1 Faults

A fault is an incorrect state or abnormal condition of hardware or software resulting from failures of components, physical interference from the environment, operator error, or incorrect design. The two types of faults are permanent and transient. The two fault types will be discussed in more detail. The work in this thesis focuses solely on tolerating permanent faults.

2.1.1 Permanent Faults

Permanent faults, also known as “hard errors”, are continuous and stable faults. Permanent faults are the result of an irreversible physical change in the device. Permanent faults can occur either during the manufacturing process or as a result of wear-out over time after manufacturing.

During the manufacturing process, contamination such as dust can produce defects [7]. These defects, called particle defects, typically manifest as shortened or broken wires. Another defect caused during manufacturing is variation in transistor behavior. Transistor behavior can be characterized by the number of dopant atoms in the transistor channel [8]. There is ran-
domness in the manufacturing process where the number of dopant atoms can vary between transistors. In the 1-micron technology there were thousands of dopant atoms and the randomness of a few dopant atoms between transistors had nearly negligible consequences. However, in the 32- to 16-nm technology there are only tens of dopant atoms in the transistor channel. Randomness in a few dopant atoms at this feature size can render some resources unusable, introducing a permanent fault in the FPGA.

Another cause of permanent faults is wear-out over an extended period of time. One way wear-out occurs is that temperature fluctuation can eventually cause cracks to appear [7]. Some forms of wear-out occur more rapidly as technology size decreases. As the feature size in semiconductor devices shrinks, the density of electric current in metal wires increases. This accelerates device wear-out due to electromigration [9]. Electromigration is the phenomenon where metallic atoms are transported from one location and deposited in another. As the atoms move from a specific location, voids form in the wires ultimately resulting in electrical discontinuity and a permanent fault in the FPGA.

Permanent faults can cause FPGAs to function incorrectly. A short in a route, an open in a route, or wrong logic are examples of how permanent faults can affect FPGAs. If a design uses the resource that has the permanent fault, the FPGA will not perform the function correctly.

Permanent faults are difficult to repair because they represent a physical state of the device. However, they can be tolerated by reconfiguring the device to avoid the faulty resource. This thesis presents new methods to tolerate permanent faults by preemptively generating configurations to be used in the presence of a permanent fault.

2.1.2 Transient Faults

A transient fault, also known as “soft error”, is an incorrect state of a bit somewhere in the system [10]. Unlike permanent faults, transient faults are not caused by physical damage to the hardware, but are a result of temporary environmental conditions. Some of those conditions are temperature, humidity, pressure, voltage, vibrations and electromagnetic interference. Another common cause of a transient fault is a Single-Event Upset (SEU).

A SEU occurs when ionizing radiation changes the state of a digital memory element [11]. As an ionized particle passes through the device, charge can be transferred from one area to another. This charge transfer can disrupt the internal state of a memory cell by changing its voltage.
There are three main sources of radiation that cause SEUs: alpha particles; high-energy neutrons from cosmic radiation; and borophosphosilicate glass in the device [12]. Alpha particles come from within the device itself. Devices are made of materials which have naturally occurring radioactive impurities. As the unstable radioactive material decays to a lower energy state, alpha particles are emitted. Cosmic radiation is emitted by celestial bodies such as the sun [11]. The earth’s atmosphere filters the ionizing radiation and reduces the probability of an SEU, however devices at high altitude and in space applications are much more susceptible. The third source of ionizing particles is a secondary source that occurs when cosmic radiation reacts with the boron in borophosphosilicate glass [12]. Boron is composed of two isotopes and one of the isotopes is unstable. When exposed to neutrons, the isotope breaks apart into a lithium isotope and an alpha particle.

These particles do not cause any permanent damage within the device [11]. SEUs can be corrected to restore the device to proper working order. The most common approach to correct SEUs combines triple modular redundancy (TMR) with configuration scrubbing [13]. For TMR, circuit modules are implemented three times. When there are no errors, all modules will have the same output. A majority voter is used to determine which module is in error when the outputs differ. After isolating the error, scrubbing repairs the memory by setting it back to the correct values. As long as two modules are working, a TMR system functions correctly. Scrubbing significantly reduces the probability that SEUs will affect two modules simultaneously and disrupt the system.

Although transient faults are not within the scope of this thesis, there is a lot of work done to address this problem and is the focus of other research efforts. This thesis focuses solely on new methods to tolerate permanent faults.

2.2 Overview of Approach

Permanent faults can be “masked”, meaning the device can continue functioning despite the existence of the fault. This is done by utilizing the reconfigurability of the FPGA and using unused resources to perform the function previously performed by the faulty resource. When a fault occurs, the device is reconfigured to avoid the fault (see Figure 2.1). The device on the left shows a configured FPGA using the resources shaded in blue. The red \( x \) signifies the resource at fault. The device on the right has been reconfigured to avoid the faulty resource.
Although it is relatively easy to mask one fault, it can be challenging to mask all possible faults. Anticipating all possible permanent faults for in-field repair of FPGA circuits is possible by generating a large set of circuit configurations before the circuit is deployed. This thesis presents new placement and routing tools based on RapidSmith to generate the set of configurations. An initial circuit configuration is generated that performs the operation of the circuit when there are no permanent faults on the FPGA device. The system is configured with this initial configuration and the FPGA uses this initial configuration until a fault is found. In addition to the initial configuration, a set of repair circuit configurations is generated. These repair configurations are used to replace the initial circuit configuration when a permanent fault is found (see Figure 2.2).
During normal operation, the system will employ a mechanism for periodically detecting permanent faults within the initial configuration (see Figure 2.3). Although not the focus of this work, there is a large body of previous work in fault detection and isolation for FPGAs [14]. If a permanent fault is detected, the system identifies the location of the fault and selects a repair configuration that repairs the permanent fault. The set of repair configurations could be cached locally within the system or be made available on an external server and accessed remotely if the system does not have sufficient memory resources. Because the repair configuration has already been created, a repair can be performed relatively quickly and without additional computation.

![Figure 2.3: Operational Repair](image)

One possible application for this repair strategy is for an FPGA design that is deployed in a large number of products and which have network access to a central server as suggested by Figure 2.4. The central server maintains a set of repair configurations for the design and responds to failure messages from the product in the field. When the server receives a failure message, it identifies the corresponding repair configuration within the repair database and sends the repair configuration to the product in the field. The failed product in the field is then repaired using the appropriate repair configuration. With time, multiple products in the field may fail and the server will provide a different repair circuit for different products since the failure of each product is unique.

### 2.2.1 Benchmark Designs

Five benchmarks will be used to evaluate the various placement and routing algorithms described in this thesis (see Table 2.1). These benchmarks are designs that have been used for other research projects. The five benchmarks were chosen to provide a wide range of design styles and sizes. All of the designs have been mapped to various devices within the Xilinx Virtex-4 family of FPGAs. The sizes of the benchmark designs are summarized in Table 2.1 in.
terms of the number of slices (i.e., the sum of SLICEL and SLICEM primitives), the number
of BRAM and DSP elements, and the number of relationally placed groups (RPGs). Although
these benchmarks utilize a relatively low percentage of their respective devices, artificial area
constraints are introduced later in this thesis as a way to mimic high utilization. The benchmarks
are limited and more testing could be done, however the benchmarks are adequate to provide a
proof of concept of the algorithms.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Device</th>
<th>Slice(utilization)</th>
<th>BRAM, DSP</th>
<th>RPGs</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>fx12</td>
<td>21 (.56%)</td>
<td>0, 0</td>
<td>1</td>
</tr>
<tr>
<td>system_test0</td>
<td>fx12</td>
<td>316 (5.8%)</td>
<td>2, 0</td>
<td>23</td>
</tr>
<tr>
<td>mult18</td>
<td>sx55</td>
<td>653 (2.7%)</td>
<td>0, 0</td>
<td>17</td>
</tr>
<tr>
<td>crazy</td>
<td>fx12</td>
<td>3373 (62%)</td>
<td>32,24</td>
<td>0</td>
</tr>
<tr>
<td>multxor</td>
<td>lx160</td>
<td>10585 (16%)</td>
<td>0, 0</td>
<td>473</td>
</tr>
</tbody>
</table>

2.2.2 Repair Placement and Routing Objectives

The primary goal of the placement and routing approaches described in this thesis is
to determine a valid FPGA placement and routing on a fully functional device and a unique
“repair configuration” for every possible permanent fault in the device. A repair configuration
is a unique configuration of a circuit design that avoids one or more specific FPGA resources. If any of these resources fail, its corresponding repair circuit can be configured onto the device and can allow the circuit to continue operating on the faulty FPGA. If a repair configuration is created for every possible resource failure, the circuit can be configured to operate on an FPGA in the presence of any single resource failure. Tolerating any single permanent fault significantly increases lifetime of the device and addresses the growing problem of device wear-out. The technique presented in this thesis could be expanded to support repair of more than one fault, but that is not within the scope of this work.

The primary disadvantage of this approach is the large amount of computation that must be performed to create repair configurations for the circuit. Creating a custom placement and routing of a circuit for every possible resource failure will obviously take far more time than traditional placement and routing for a fully functional device. However, performing all this placement and routing computation before the circuit is first configured has a number of advantages. First, all the repair information is available when a fault occurs and no additional resource mapping computation is needed to create a repair circuit. This simplifies the process of performing a repair. Second, it is possible to guarantee a repair circuit for each failure. Because the repair configurations are available before failure, there is no concern that a repair circuit cannot be determined. Third, it is more efficient to identify all of the repair configurations at the same time during global placement and routing than it is to create repair circuits one at a time as needed. Since all of the design and device information is resident, many of the data structures can be reused and information shared during the process. This savings in time is similar to the time savings performed using incremental placement and routing approaches [4].

The repair placer and router must generate a complete set of repair configurations under several constraints. First, the repair placer and router should minimize the impact of the repair process on the circuit quality. Circuit quality will be defined in a later chapter. Modifying the placement of a circuit for a repair may impact the timing of the circuit. It is important to minimize this impact and generate configurations that have similar timing characteristics of the original circuit configuration.

Second, it is important to minimize the overall size of the repair database. The repair database may be part of an embedded system and may consume a large amount of memory. To make this approach feasible, the size of this repair database needs to be carefully controlled. As described earlier, a repair placement may repair a large number of faults. To reduce the number
of repair patches in the repair database, the repair placement should attempt to cover as many faults as possible with each repair configuration.

Third, it is important to minimize the time required to generate the repair placement set. Performing repair placement will certainly increase the runtime of the placement routing process. In some cases, repair placement will be significantly longer than the conventional placement. While this approach for repair must accept longer run-times, the run-times must be carefully controlled and managed.

There is generally a trade-off between the quality of a repair configuration and the time needed to generate the repair. A successful placement and routing repair approach must maintain high quality circuits while completing in an acceptable amount of time.

2.3 Related Work

There has been great interest in taking advantage of unused logic and routing resources of FPGAs to use for repairing permanent faults. FPGAs can be reconfigured to utilize previously unused resources in place of a faulty one. This allows the FPGA to continue functioning correctly despite the presence of a fault. Most FPGA designs, even those that are heavily utilized, contain unused resources (routing and logic). The goal of these techniques is to exploit these unused resources with repair configurations that use these unused resources to avoid a faulty resource.

Approaches for repairing permanent faults can be viewed in two categories. The first approach generates a repair configuration after the fault occurs, such as in [4]. This approach is good for repairing any permanent fault, however there is downtime due to the computation time needed to generate the repair.

The second approach generates repairs prior to failure, such as in [15]. With this approach, the repair is immediately available to reconfigure the device, however the approaches found using this method are limited in the number of faults they can repair.

Incremental approaches have been introduced that allow the ability to perform partial rerouting for repairing with much less effort by saving the routing history [4]. Without history the entire design would have to be ripped up and rerouted in the event of a fault in the interconnect. This work also supports the ability to address “Intra-cluster” faults by using unused resources within a cluster. A cluster is multiple LUT/FF pairs tightly connected together. Faults in clusters can be addressed by substituting unused cluster interconnect, LUTS, and flip-flops.
The authors in [16] describe a yield enhancement scheme that allocates spare interconnect resources to tolerate functional faults. This is achieved by adding redundant routing tracks during placement and routing. Then decisions are made when the device is configured to determine which routing tracks should be use. The additional redundant hardware results in greater area and timing penalties, but only a single bitstream and a bitstream controller is needed to repair non-identical devices.

Several efforts exploit repair by pre-allocating rows or columns of logic resources and “shifting” the design to avoid these resources in the event of a fault [3, 17]. For example, in [17], the design is partitioned into tiles. Within each tile there is a spare logic block to be used in the event of a permanent fault. Every tile has multiple configurations so that any single fault within a tile can be repaired.

The Choose-Your-Own-Adventure (CYA) router focuses on repairing interconnect defects. In [18], the authors describe CYA as a method for embedding test structures and repair information within a bitstream. Decisions are made at configuration load time to determine which configuration information, including alternative repair structures, to use.

The work presented in this thesis is most similar to the approach presented in [15] where there are two or more implementations of the design. In [15], each implementation is encouraged to utilize different resources as much as possible. All configuration bitstreams are stored in a memory. The bitstreams are then loaded sequentially to configure the device and tested for correct functionality. When a bitstream is found to function correctly, the configuration procedure terminates. This technique was developed specifically to negate faults that occurred during the manufacturing process. The downside with this approach is that with only several bitstreams, it is possible for all of them to fail to avoid the defective resource and render the device unusable. Another downside is it requires memory to store the additional bitstreams.

This thesis presents new methods for increasing reliability by generating repairs prior to failure. Unlike the other approaches, these new methods anticipate every possible permanent fault and generate repair configurations. The focus of the work in this thesis is to generate a repair set for all faults within FPGA resources. The specific resources and the tools used to generate repair configurations are discussed in the next chapter.
Chapter 3

FPGA Architecture and Mapping Tools

The work presented in this thesis was developed for the Xilinx Virtex-4 architecture. This chapter describes the basics of FPGA Architecture and provides a broad overview of the specific family of FPGAs used for testing, the Xilinx Virtex-4. This chapter describes the specific resources in the Virtex-4 architecture that will be targeted for repairs. This chapter also explains the design flow that takes a user generated design and processes it to be configured onto the device. Knowledge of the design flow provides a better understanding of where in the tool flow the work presented in this thesis is taking place.

In addition, this chapter describes the Xilinx Design Language (XDL), which is the file format the algorithms presented read in and manipulate to perform placement and routing. This chapter also describes the RapidSmith toolkit which is the environment the algorithms were developed in. RapidSmith fully supports the Virtex-4 architecture[19]. Although the work in this thesis is very specific to the Virtex-4 architecture, the ideas could be applied to other architectures.

3.1 FPGA Architecture

Field-programmable gate arrays (FPGAs) are integrated circuits designed to be configured after manufacturing. FPGAs are composed of programmable logic and memory blocks which are connected using programmable interconnects to perform a specific task. The large number of unique configurations allows FPGAs to perform a wide variety of complex tasks. They can be reconfigured at any time to perform a different function.

Certain types of FPGAs have blocks in addition to programmable logic and memory that perform specific functions. For example, some FPGAs have digital signal processing blocks (DSPs) that are used to increase performance. The type of blocks available on an FPGA depends on the vendor, family, and specific device. The tests and results for this thesis are performed
on the Xilinx Virtex-4 architecture [20]. This section describes the resources on the Virtex-4 family of FPGAs that are relevant to the repair algorithms presented in this thesis.

3.1.1 Virtex-4 Overview

The Virtex-4 is an island style FPGA arranged in a two dimensional grid of sections called “tiles” similar to what is shown in Figure 3.1. There are several different types of tiles: CLB, DSP, BRAM, and interconnect. Every tile of the same type is identical. A specific tile is identified by its (X,Y) location on the device (e.g. CLB(40, 49)). Many of the tiles can be broken down into smaller components called primitive types. Primitive types are the smallest atomic unit in the repair process. The inputs and outputs of primitive type are called pins. The tiles and corresponding primitive types are described in the following section.

![Figure 3.1: Configurable Logic Blocks (CLBs) in island style FPGA with Programmable Switch Matrices (PSMs) and routing interconnect [21].](image)

3.1.2 CLB Tiles

The Configurable Logic Blocks (CLBs) are the resource for implementing logic on the FPGA. A CLB occupies a single tile on the device and is connected to a switch matrix to access
the general routing structure (see Figure 3.2). Each CLB is comprised of four interconnected slices.

One example of a primitive type found within a tile is a slice. Each slice is made up of two look up tables (LUTs), two storage units, wide-function multiplexers, carry logic, arithmetic gates, and routing interconnect (see Figure 3.3). The Virtex-4 uses four input LUTs that can each implement any single four input function. The LUTs can then be connected together to form more complex functions.

The CLB is divided into two columns with two slices in each column as shown in Figure 3.2. There are two types of slices, the pair in the right column is SLICEL and the pair in the left column is SLICEM. The two different slices are two of the primitive type resources that need to be repaired. There is a distinction between a SLICEL and a SLICEM. The SLICEL column-pair can only be used as logic, where as the SLICEM column can be used as logic, distributed RAM, or as a shift register [20]. In terms of repair, a SLICEM can be used to replace a faulty SLICEL, but a SLICEL cannot be used in place of a SLICEM.

Figure 3.2: Arrangement of slices in a CLB [20].
3.1.3 DSP Tiles

The Digital Signal Processing (DSP) tile contains two DSP slices (see Figure 3.4). A DSP slice is another primitive type that needs to be repaired. Each slice supports many independent functions, including multiplier, multiplier followed by an adder, or barrel shifter. The slices can also be connected together to form wide math functions. These function could be implemented using more general logic resources such as CLBS. However, the use of DSPs can de-
crease the amount of more general logic resources resulting in lower power, higher performance, and efficient device utilization [22].

Figure 3.4: DSP Tile Containing Two DSP Slice Primitives [22].

3.1.4 BRAM Tiles

The final primitive type that the placement algorithms repair is the block random access memory (BRAM) [20]. The Virtex-4 BRAM block can store up to 18 kilobits of data. The BRAMS have true dual ports meaning data can be written to either or both ports and can be read from either or both ports. The BRAMS are cascadable, meaning they can be linked together to enable deeper and wider memory implementation. Each BRAM can also be configured as a FIFO memory using dedicated hardware without the need for additional CLB logic. Figure 3.5 shows how BRAM tiles fit in the device in relation to the other tiles.
### 3.1.5 Interconnect Tile

There are two main types of routing resources on an FPGA used to connect the pins on primitives, programmable interconnect points (PIPs) and physical wires. A PIP is used to connect two wires. PIPs are typically most concentrated in the interconnect tiles, but can be found in other types of tiles as well. These interconnect tiles, also called programmable switch matrices (PSMs), are dispersed throughout the FPGA. Figure 3.6 shows a PSM (shaded gray square in center). The PIPs are the diamond shapes in the figure where the wires intersect.

The other type of routing resource is a wire. Wires vary in length. Some wires only connect adjacent tiles while others span across several. The length and quantity of wires depends on the device. Most wires are unidirectional, meaning the signal must start at one end and travel to the other. One example of a wire that is not unidirectional is a long line.

The Virtex-4 architecture has long lines that cover relatively large distances. Long lines are bidirectional, so the signal can travel either direction. Specific caution must be taken when using long lines (see Figure 3.7). There are multiple exits along the wire, but the signal must start at one end or the other. When one end of the long line is being used, the other end cannot be used by another signal.

### 3.2 Xilinx Design Flow

The Xilinx design flow is the process for creating implemented designs for FPGAs [23]. This process is broken up into a number of steps (see Figure 3.8). The first step is Design Entry. In this step, a design is typically created using a hardware description language (HDL). HDLs
Figure 3.6: Detail of Programmable Interconnect Associated with CLB [21].

are able to describe a design at a high level of abstraction, but also has the capability to control the gate-level details. HDLs are technology independent making it easy to generate the design for different technologies without making any changes to the design.

After creating a design in HDL it needs to be synthesized. Synthesis optimizes the design for a specific device and converts it into a structural netlist. The netlist contains the primitives on the FPGA and the connections between them.

After synthesis, the design is implemented. Implementation is the process of taking the previously generated netlist and preparing the design to be configured onto a specific device. This is done using a number of discrete steps. First, the Xilinx NGDBuild program reads in all of the netlists and combines them into a single netlist. Using the information in the netlist
Figure 3.7: Example of long lines. Long lines can be entered on either end and can be exited at several points along the path, like in A and B. However, both ends can not be used simultaneously as shown in C.

it then creates a Native Generic Database (NGD) file. This file logically describes the design in terms of generic primitives. The NGDBuild also adds user constraints to the design by extracting them from a user constraint file (UCF). After NGDBuild, the NGD file is ready to be mapped to the specific device family.

Mapping is the process of pairing the generic logic in the NGD file to the specific primitives (slices, BRAMS, etc.) in the target FPGA. Mapping is performed using Xilinx's MAP program. MAP outputs a native circuit description (NCD) file, which is a physical description of the design mapped to the specific FPGA.

After mapping, the NCD file is ready to be placed using the PAR (place and route) program. During placement, the placer assigns each primitive to a physical site on the device. Placement is controlled by a number of factors such as the length of connections and the available routing resources. After the design is placed, PAR generates a new NCD file that now contains the placement information.

After placement, PAR routes, or connects, the components using the wires in the FPGA as defined by the netlist. Upon completion, the PAR program outputs an NCD file containing the fully placed and routed design. The NCD file is then passed on to the bitstream generator (BitGen).

The final step in design implementation is the creation of a bitstream (BIT) file by BitGen. A BIT file contains all the information needed to configure a design onto a device. BitGen reads in the fully placed and routed NCD file and generates a BIT file. The BIT file is then used to configure the device.

The work presented in this thesis takes place during the place and route stage of the design flow (see Figure 3.9). The design is processed through the commercial Xilinx tools up
through mapping. Next, the NCD file is converted and processed through the modified repair placer and router. Finally, the bitstream is generated using commercial tools.

### 3.3 RapidSmith

There are several tools that can manipulate FPGA designs. These tools work with the Xilinx Design Language (XDL) file format. An XDL file contains all of the information of an NCD file, but it is human readable. The specifics of XDL are detailed in the following subsection.
Device databases have been derived from XDL [24, 25]. An open source tool called Torc was developed for custom research applications, CAD tool development, and architecture exploration using XDL files [26]. This thesis manipulates XDL files using a tool called RapidSmith.

RapidSmith is a set of tools and APIs aimed to manipulate XDL [19]. It provides an environment to explore experimental placement and routing algorithms on FPGAs. RapidSmith is written in Java and allows designers to write Java code to manipulate and modify designs in XDL format. Although the use of the Java programming language for this project will result in slower run-times for placement and routing when compared to natively compiled approaches, the ability to exploit the existing XDL libraries and device databases significantly improved the software development productivity of the project. Figure 3.10 shows where RapidSmith tools work in relation to the design flow. All the placers and routers presented in this thesis were written in Java using RapidSmith’s environment.
The placement and routing code developed for this thesis is layered on top of Rapid-Smith. RapidSmith provides the basic capabilities to perform placement and routing [27]. For placement the three basic capabilities are: First, creating objects of resources that need to be placed. The object contains information such as the connectivity, size, shape and classification of the object; Second, providing a complete set of valid placement locations where the objects can be placed; Third, an infrastructure so that changes can be made to placement quickly.

The basic building block used by RapidSmith for placement is an *instance*. Instances are an instantiation of a primitive type (such as SLICEL). The location on a device where an instance can reside is called a primitive site. Instances are placed by assigning them to a compatible primitive site. RapidSmith provides methods to create an instance, get all the compatible primitive sites for that instance, and set the placement of the instance.

For routing, RapidSmith provides a compact routing graph by using efficient techniques to store and retrieve routing resources (nodes) and routing connections (edges). Methods in RapidSmith include getting all the connections of a node, and calculating the Manhattan distance between nodes. RapidSmith sets the route by setting the PIPs in the XDL.

RapidSmith provides all the functionality to place instances and set routes, but the actual approach for complete placement and routing is determined by the user. The next few chapters of this thesis describe placement and routing algorithms, as well as introduce the new placement and routing approaches developed to repair permanent faults.

### 3.3.1 XDL

The NCD file is a proprietary format of Xilinx and is not human-readable. Xilinx has created another format that is human-readable called the Xilinx Design Language (XDL) [19]. XDL contains the same information as NCD. XDL can represent a design in various stages of the place and route process. For example, a design can be unplaced and unrouted, partially placed and unrouted, partially placed and partially routed, fully placed and unrouted, fully placed and partially routed, or fully placed and fully routed. The XDL file can be modified to change the state of placement and routing. The following XDL snippet shows the format for an instance of a primitive.

The first line contains the name of the instance “*mySlice*”, the primitive type “*SLICEL*” and the location. The rest of the snippet is the “cfg” string which contains a list of attributes to define the content and functionality of the instance. Placement can be changed by modifying
the first line. For example, to unplace the instance, replace “placed CLB_X14Y4 SLICE_X23Y8” with “unplaced”. As long as the modifications do not violate the constraints of the architecture the XDL file can be converted back to NCD format.

Xilinx has tools which convert NCD files to XDL files and vice versa. This tool is an executable called xdl. After a modified XDL file has been converted back to NCD, the Xilinx tools can be used to finish placement and routing if not already completed, and to generate the bitstream.

This thesis presents new algorithms to generate repair configurations to tolerate permanent faults in Xilinx Virtex 4 FPGAs. These permanent faults can occur in CLBs, DSPs, BRAMs, and in routing interconnect. The algorithms were written in Java and use RapidSmith to manipulate XDL to create a custom placement and route. The placement algorithms are described in the following chapter.
Chapter 4

Placement

Placement is the process of assigning primitives to physical locations on the device. Placement is a difficult problem in that there is no efficient way to find an absolutely optimal solution. Instead, heuristic methods are used to find approximate solutions. Placement is generally done by first performing a random placement, and then using a heuristic approach to improve the quality by moving the primitives to more optimal locations. This chapter describes the baseline placement algorithm as well as addresses the challenges associated with placement. The new approaches for generating repair placement configurations are also introduced.

4.1 Baseline Placement Algorithm

A conventional placement algorithm was developed to perform logic placement for Xilinx Virtex-4 designs. Although the focus of this paper is not on the development and improvement of conventional placement algorithms, a conventional placer is necessary to provide a baseline of comparison between the repair placement approaches described later in this paper.

The baseline placer operates on XDL files that are created from vendor tools as shown in Figure 4.1. Vendor tools are used to synthesize the design, perform technology mapping, and convert the internal design representation into the XDL text file. At this point, our baseline placer reads the XDL, performs placement, and saves the placement into the XDL file. The file is converted back into the binary format and vendor tools are used to perform routing and generate a valid FPGA bitstream.

![Figure 4.1: Design Flow for Baseline Placement](image-url)
The baseline placer was based on the well-known VPR FPGA placement approach [28]. An initial, legal placement is created by randomly choosing sites for FPGA resources. After creating a legal initial placement, a heuristic approach is used to explore placement perturbations (moves) in an attempt to improve the cost of the design placement [29]. The cost is determined by estimating the wire length of the routing within the design [28]. This estimate is made by measuring the x and y dimensions of the bounding box (see Figure 4.2) that contains all terminals of each net as follows,

\[
Cost = \sum_{i \in \text{AllNets}} q(i) \cdot (bb_x + bb_y)
\]  

(4.1)

where \(q(i)\) is a fanout-based correction factor, \(bb_x\) is the x dimension of the bounding box and \(bb_y\) is the y dimension of the bounding box. Placement moves are randomly selected and accepted using a carefully controlled annealing schedule.

Figure 4.2: This figure shows the bounding box of a net. This box is 3×3.

4.1.1 Simulated Annealing

Simulated annealing is a heuristic approach that occasionally allows a move resulting in a higher placement cost [30]. Allowing some changes reduces the likelihood of becoming stuck in a local optima. The probability of making a bad change is dependent on the user-defined
temperature $t$. At high temperature almost every move is taken, but as the temperature cools the number of bad moves taken decreases. Other user-defined parameters are the number of placement changes to attempt per temperature, how much the temperature should decrease each iteration and the ending criteria. Changing the parameters affects the run time of the heuristic and the quality of the solution.

Algorithm 1 Simulated Annealing Heuristic

1: Perform initial random placement
2: Set initial temperature $t$
3: while !(stop criteria) do
4:   for $i = 1$ to number of moves to attempt do
5:     Select a move
6:     Compute cost of placement after move
7:     If accept(move, $t$), take move
8:     Else reject move
9:   end for
10: Reduce temperature $t$
11: end while

The approach for Simulated Annealing is outlined in Algorithm 1. It begins by randomly placing all the primitives and initializing the temperature. For each temperature, a predetermined number of moves will be attempted. Next a move is selected. A move can either move a primitive to an unoccupied location, or if the location is occupied, swap locations with the residing primitive. If the move decreases the overall cost of the placement, it is accepted. If the move increases the cost, the move is randomly determined to be accepted or rejected based on how much it increases the cost and the current temperature. After the predetermined number of moves are attempted, the temperature decreases and the process repeats. This continues until the stop criteria is met. Typical stop criteria is that the cost of placement has not recently improved or the temperature has decreased a set amount. For more information on how the Simulated Annealing placer was implemented see Appendix A.

There are challenges associated with placement, such as restrictions on where and how primitives can be placed. This is addressed in the next section about placement groups. Following that section the results of the baseline placer are compared with the Xilinx commercial placer using the metrics described. Finally, artificial area constraints are discussed as a way to make designs appear to utilize more of the device.
4.1.2 Placement Groups

To support placement of designs on the commercially available Virtex-4 architecture, a number of issues had to be addressed. First, the Virtex-4 architecture is not homogeneous and there are several different primitive types whose placement locations are not interchangeable (i.e., SLICEL, SLICEM, BRAM, DSP, etc.). Second, there are a number of architecture dependent placement restrictions that complicate the selection of valid placement locations. For example, SLICEL primitives can be placed at either a SLICEM or SLICEL location while SLICEM primitives can only be placed at a SLICEM location. Third, many design instances have relational placement constraints with other instances when certain architecture-specific features are used. For example, all instances using the same carry chain must be placed in the same column and be placed vertically adjacent in the appropriate carry chain order (see Figure 4.3).

Figure 4.3: This figure shows three primitives (A,B,C) that are constrained by a carry chain to be placed in the same column in the correct order. Moving the primitives individually (left) allows the possibility of placement violations. Combining them into a relationally placed group (top) guarantees that placement constraints will not be violated (right).
To address these issues, this placer identifies and organizes all netlist instances into relationally placed groups (RPGs) and performs placement on multi-site RPGs rather than individual netlist instances. The placement of groups is more challenging as the placement process must consider the shape of each group and find a placement in which no groups overlap (this is especially challenging with very tight placement constraints). All benchmark designs were successfully placed and validated using the vendor design rule checker.

4.1.3 Baseline Placement Results

The execution time and overall cost of circuits placed with our baseline placer are shown in Table 4.1. The cost represents the quality of result of the placement and is determined by measuring the total wire length using a bounding box span (see Equation 4.1). It is interesting to note that for some designs, the baseline placer produces a placement that has a lower cost than the corresponding Xilinx placement. This suggests that Xilinx uses a different cost function and the cost function used here could be improved.

<table>
<thead>
<tr>
<th>Design</th>
<th>Baseline RapidSmith</th>
<th>Xilinx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time(s)</td>
<td>Cost ns</td>
</tr>
<tr>
<td>top</td>
<td>.71</td>
<td>200</td>
</tr>
<tr>
<td>system_test0</td>
<td>4.4</td>
<td>5834</td>
</tr>
<tr>
<td>mult18</td>
<td>35.7</td>
<td>9160</td>
</tr>
<tr>
<td>multxor</td>
<td>3937</td>
<td>348792</td>
</tr>
</tbody>
</table>

To provide a reference, the execution time and cost for vendor placement is also provided. To facilitate comparison, the same cost function was used for both the vendor placement and our baseline placement. On average, our placer executes 6.7 times slower and generates circuits that have a 19% higher clock period than the vendor tools.

The final metric shown in the table is the minimum clock period. The minimum clock period is the time it takes a signal to traverse the longest path. This constrains how quickly instructions can be executed. The lower the minimum clock period, the faster the device can operate. Although a design must be placed and routed to find the minimum clock period, this metric can used to measure the quality of a placement. This is done with an external routing
step using Xilinx's tools. After routing, the minimum clock period is found by running a Xilinx trace on the routed design.

The baseline placement approach described here is inferior to a commercial placement approach in both execution time and quality of results. There are many methods that could improve the quality and run time of the placer. However, the focus of this work is not to replicate commercial quality placement but to demonstrate the feasibility of a placement approach that considers repair.

4.1.4 Area Constraints

As seen in Table 2.1, most of the benchmark designs use a relatively small amount of the device’s FPGA resources. It is relatively easy to repair designs with low FPGA utilization because these designs leave a large number of unused FPGA resources. With a large set of unused resources, it is relatively easy to create a repair design that replaces a used resource from the original design with an unused resource.

Any design that uses less than 50% of an FPGA’s resources can repair any permanent fault with a single repair circuit. Figure 4.4 demonstrates this point. This figure demonstrates two different placements of the multxor benchmark design. The placement on the left is the original placement and would be used when there is no fault in the device. The placement on the right is the “repair” placement and would be used if there is a permanent fault in the device that affects any of the resources used in the original placement of the design. Since no resources used in the left placement are used in the right placement, the device can be configured with the right placement to repair any failure within the device that affects the left placement. Only one alternative placement, the right placement, is needed for a repair configuration.

Generating repair circuits is more difficult for designs that consume more than 50% of the circuit resources. When more than 50% of the device is utilized, more than one repair circuit is needed. For example, if a design uses 67% of the FPGA resources, only 33% of the idle resources are available for a repair. In this case, at least two repair circuits are needed (see Figure 4.5). One repair configuration could repair half of the design’s original resources and the other repair configuration could repair the other half of the original resources.

The number of repair configurations needed to repair any circuit fault grows as the utilization of the device increases. Note that if the device is 100% utilized, there are no idle resources for a repair and no repairs can be made. The minimum number of repair configurations
Figure 4.4: Two Placements for the multxor Design. The Left Placement is the Original Placement and the Right Placement is the Repair Placement.

Figure 4.5: Repair Circuits for 67% Utilized Design

required to cover all utilized resources is the number of resources allocated by the design, $A$, divided by the number of idle resources in the device, $I$, as shown in Equation (4.2). The number of idle resources is the total number of resources available on the device, $R$, minus the number of resources allocated by the design (i.e., $I = R - A$),
\[ N_{\text{min}} = \left\lceil \frac{A}{I} \right\rceil = \left\lceil \frac{A}{R - A} \right\rceil = \left\lceil \frac{1}{R/A - 1} \right\rceil = \left\lceil \frac{u}{1 - u} \right\rceil. \] (4.2)

This expression can also be represented in terms of the utilization of the device, \( u = A/R \).

For example, if a device contains 10,000 elements (i.e., \( R = 10,000 \)) and a design mapped to the device uses 8,500 of these elements (i.e., \( A = 8,500 \)), then there are 1,500 idle resources in the device (i.e., \( I = 1,500 \)) and the device has a utilization factor of .85. This design will need a minimum of six repair configurations to repair every resource on the device:

\[ N_{\text{min}} = \left\lceil \frac{1}{10,000/8,500 - 1} \right\rceil = \left\lceil \frac{1}{1.18 - 1} \right\rceil = \lceil 5.667 \rceil = 6. \]

Equation (4.2) represents the minimum number of configurations and in practice more repair configurations will be needed.

A set of artificial placement constraints have been created for the benchmark designs to emulate higher resource utilization. These artificial placement constraints will define a rectangular region within the FPGA that achieves a predetermined utilization. Table 4.2 describes five different area constraints for each of the benchmarks. These constraints force a utilization of 50% (A), 75% (B), 90% (C), 95% (D), and 99% (E). There is no 95% or 99% constraint for the top design since this design is too small. Using Equation (4.2), these constraints will require at least 1, 4, 10, 20, and 100 repair configurations respectively.

<table>
<thead>
<tr>
<th>Table 4.2: Bounding box area constraints for five different utilization levels. The bounding box is defined by the number of slices wide and the number of slices long.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraint</td>
</tr>
<tr>
<td>Design</td>
</tr>
<tr>
<td>top</td>
</tr>
<tr>
<td>system_test0</td>
</tr>
<tr>
<td>mult18</td>
</tr>
<tr>
<td>crazy</td>
</tr>
<tr>
<td>multxor</td>
</tr>
</tbody>
</table>

For example, the rectangular constraint to provide a 90% utilization for the mult18 design is 25,29. This means that all of the slices in the mult18 design must be placed within a square region that is 25 slices wide and 29 slices high (see Figure 4.6). The total area of this
bounding box constraint is 725 slices. The utilization of this constrained region for the `mult18` design is \( \frac{653}{725} = 90.1\% \).

**Figure 4.6:** This figure is of an example FPGA at the slice level. It shows how the `mult18` benchmark must be placed in an area that is 25 slices wide and 29 slices long when it is under the 90% area constraint.

In practice, more repair designs will be needed to provide a complete database of repairs than the amount indicated in Equation (4.2). Minimizing the number of repair circuits may lead to some repair circuits with a poor quality of result. For example, Repair #1 in Figure 4.5 is not a particularly good placement result - this particular placement is split into two parts with a significant distance between the two parts. The repair placement algorithms presented later in the next section seek to minimize the number of repair circuits needed while maintaining acceptable levels of quality.
4.2 Repair Placement

Placement determines which logic blocks or fixed functions will be used by the device. A permanent fault in a logic block or fixed functions can be tolerated by modifying the placement to avoid the faulty resource. In order to anticipate every possible fault in the logic blocks and fixed function, a repair placer must generate a complete set of repair placements so that for each resource used in the initial placement, there is a repair placement that avoids that resource.

The repair placement described in this work will generate an initial placement of a design. This initial placement specifies the placement of each primitive of a given design. Each FPGA site used in this initial placement must have a corresponding repair placement. A corresponding repair placement is a unique placement of the same design such that the given FPGA site is not used in the design. Figure 4.7 shows an example of an initial placement on a device alongside an alternate repair placement. Multiple repair placements will be needed to repair every site used in the initial placement.

![Figure 4.7](image)

**Figure 4.7:** This figure shows two possible placements of a design on a device. The device has 16 possible sites. A colored in square represents an occupied site. Placement A is the initial placement. In this example site $s_{2,3}$ has failed. Placement B is an alternate placement where site $s_{2,3}$ is no longer used, thereby repairing the device.

This section describes the four different repair placement approaches. They are called Naive, Cost Repair, Shadow, and Hybrid. An algorithm overview is given for each approach. The benchmarks were run through each approach and the results are summarized. Finally, the
results between the different approaches are compared and contrasted and the advantages and
disadvantages of each approach are discussed.

4.2.1 Naive Repair Placement

The first approach for repair placement is relatively simplistic and thus called the “Naive”
approach. The strategy behind the Naive placer is that with each iteration at least one resource
will be repaired. This is done by selecting a resource that needs to be repaired and eliminating
the possibility of it being used in the current iteration of the placer, thereby guaranteeing that
it will not be used. This process is repeated until all resources have been repaired.

Algorithm Overview

This approach, summarized in Algorithm 2, begins by performing a single, initial place-
ment using the conventional placement algorithm described in Chapter 5. This initial placement
is used on the device when there are no device failures. This initial placement defines a set, $S$,
of placement sites that are used in the initial placement (see line 3).

<table>
<thead>
<tr>
<th>Algorithm 2 Naive Repair Placement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: $D \leftarrow$ set of all possible placement sites</td>
</tr>
<tr>
<td>2: Perform initial placement</td>
</tr>
<tr>
<td>3: $S \leftarrow$ set of occupied sites in initial placement</td>
</tr>
<tr>
<td>4: while $S \neq \emptyset$ do</td>
</tr>
<tr>
<td>5: choose $s \in S$, remove $s$ from $S$</td>
</tr>
<tr>
<td>6: Remove site $s$ from devise database</td>
</tr>
<tr>
<td>7: Perform repair placement(site $s$ not available)</td>
</tr>
<tr>
<td>8: $R \leftarrow$ set of occupied sites in repair placement</td>
</tr>
<tr>
<td>9: $G \leftarrow (D - R) \cap S$ (sites repaired)</td>
</tr>
<tr>
<td>10: $S \leftarrow S - G$</td>
</tr>
<tr>
<td>11: Add site $s$ into device database</td>
</tr>
<tr>
<td>12: end while</td>
</tr>
</tbody>
</table>

The algorithm proceeds by performing multiple, independent placements on the design
with the goal of “repairing” one or more sites used in the original design. A “repair” placement
is an alternative placement of the circuit primitives such that some logic or fixed function sites
used in the original placement are not used. When choosing a move, the placer randomly selects
a site from all possible sites in the device database. To insure that the placement repairs at least
one site, one of the physical placement sites, \( s \), used in the original placement is removed from
the device database (line 5). This prevents the placer from selecting \( s \) when choosing random
placement moves. Since the site \( s \) will never be selected during placement, it will not be used in
the repair placement and the repair placement can be used when site \( s \) fails. Due to the random
nature of the simulated annealing placer, the repair placer will likely repair other sites that were
used in the original placement (i.e., some of the sites used in the original placement are not used
in the repair placement). Those sites repaired during a repair placement are removed from the
set of sites needing a repair (line 10). The algorithm will iterate until there are no more sites
needing a repair (line 4).

Results

The Naive repair placer was performed on all five benchmarks and each area constraint.
The results of this placement approach are summarized in Table 4.3. This table indicates how
many repair placement circuits were created to repair all sites used in the initial placement.
For example, 114 different placement configurations were required to repair all 3429 instances
for the crazy benchmark using the 75\% (B) area constraint. For this design, each placement
circuit repaired on average 30 placement sites.

Table 4.3: Number of Repair Circuits using Naive Repair Placement

<table>
<thead>
<tr>
<th>Design</th>
<th>No Constraint</th>
<th>A(50%)</th>
<th>B(75%)</th>
<th>C(90%)</th>
<th>D(95%)</th>
<th>E(99%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>31</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>system_test0</td>
<td>10</td>
<td>40</td>
<td>71</td>
<td>112</td>
<td>149</td>
<td>243</td>
</tr>
<tr>
<td>mult18</td>
<td>2</td>
<td>51</td>
<td>142</td>
<td>294</td>
<td>340</td>
<td>467</td>
</tr>
<tr>
<td>crazy</td>
<td>N/A</td>
<td>76</td>
<td>114</td>
<td>224</td>
<td>413</td>
<td>929</td>
</tr>
<tr>
<td>multxor</td>
<td>47</td>
<td>150</td>
<td>441</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Time</td>
<td>25x</td>
<td>56x</td>
<td>132x</td>
<td>152x</td>
<td>243x</td>
<td>468x</td>
</tr>
</tbody>
</table>

As expected, the number of repair circuits required to repair all sites increases as the
area constraint tightens. The number of repair circuits generated by this approach, however,
is far higher than the minimum number of repairs suggested by Equation (4.2). For example,
the minimum number of repair circuits for crazy using the 75\% benchmark (B) is 5 while 114
were created using this approach. Not all of the results have been computed for the multxor
benchmark. Naive Repair Placement for this benchmark takes a very long time. For example,
it took over 17 days to generate the 441 repair placements for constraint B. It is anticipated that the C, D, and E constraints would require several months to complete.

The final row of Table 4.3 reports the average increase in execution time of the Naive Repair Placer over the conventional placer. As expected, the execution time increases with each tighter area constraint. For these benchmarks, the execution time is growing non-linearly and this approach will take a tremendous amount of time to repair large circuits with a high utilization. The cost of the repair circuits using Equation (4.1) was measured and compared with the cost of the corresponding initial configurations. The average cost of the repair configurations was found to be 2% higher than the cost of the initial configurations. Since a complete placement is performed during each iteration, it is expected that the cost of the naive repair configurations will be very similar to the cost of the initial configuration.

Completing the Naive Repair Placer takes a very long time because relatively few repairs are performed during each placement iteration. The number of unique repairs performed during each iteration of the multxor benchmark for the 75% B constraint is shown in Figure 4.8. As seen in this figure, the first few iterations of the algorithm perform a large number of repairs (over 900 in the first iteration). However, due to the random nature of site selection, the number of repairs performed quickly decreases. Even though each iteration is guaranteed to repair a single site, fewer than 100 repairs are actually performed after about 30 iterations of the algorithm and only a handful of repairs are made in the final iterations of the algorithm.

4.2.2 Cost Repair Placement

A second repair placement approach was developed to address the long execution time of the Naive Repair approach. This second approach, called Cost Repair, operates similarly to the Naive Repair but it attempts to complete all the repairs using far fewer iterations of the placer. This is done by introducing a new cost function that gradually increases the cost of all sites still needing repair with each iteration. This encourages the placer to select sites with lower costs that do not need to be repaired.

Algorithm Overview

Like the Naive approach, this algorithm begins with an initial placement and performs multiple independent placements to provide repairs for the sites used in the initial placement.
The objective of the Cost Repair approach is to perform more repairs during each placement iteration to reduce the number of repair placements required to repair all sites.

A modified cost function is used to meet this objective. Specifically, this modified cost function is used to encourage the placer to perform more repairs during each iteration of the placer. This is done by charging a site-specific cost for each primitive site used in the design (see Equation 4.3). Placement sites that are unused in the original placement or that have been repaired have no cost - they can be used by the repair placer without incurring additional costs. Sites that have not been repaired (i.e., elements of set \( S \) on line 3) are given an initial cost. Because the use of these sites will incur additional cost, the placer will attempt to avoid sites that need to be repaired and favor those sites that do not need to be repaired (i.e., sites that were not used in the original placement). The placer will not avoid all sites needing repair, however, since avoiding all of these sites will significantly increase the wire length bounding box.

\[
\text{Cost} = \sum_{i \in \text{AllNets}} q(i) \cdot (bb_x + bb_y) + \sum_{j \in \text{Sites}}.
\] (4.3)
Initially, the cost of using sites that need to be repaired is low. As seen in the Naive approach, the random nature of placement will repair a large number of sites during the early iterations. To encourage repair in later iterations of placement, the cost of sites needing repair is increased after each iteration (line 11). The cost of using a site needing repair grows exponentially: $C_{i+1} = C_i \cdot (1 + \mu)$, where $\mu$ is an experimentally determined parameter.

In addition to a modified cost function, the Cost Repair approach will insure the placer makes greater forward progress by removing more than one site from the device database during each iteration. The number of sites removed depends on the utilization of the device and the number of used resources needing repair. Specifically, the minimum of either 5% of the idle sites will be removed or 5% of the used resources are removed as shown in Equation 4.4. For designs with high utilization, fewer sites will be removed to maintain high quality placement. This calculation is rounded up to insure that at least one resource is removed during each iteration.

$$n_t = \min \left\{ \lceil 0.05 \times I \rceil, \lceil 0.05 \times A \rceil \right\}.$$  \hspace{1cm} (4.4)

**Results**

The Cost Repair placement approach was applied to all of the benchmarks under each of the placement constraints. The number of placements needed to repair all sites in the original design is summarized in Table 4.4. These results indicate that the Cost Repair is able to provide a complete repair set with far fewer repair circuits than the Naive approach (see Table 4.3). For
example, the number of repair placements needed for crazy using constraint B drops from 114 using the Naive approach down to 15 using the Cost Repair approach. A similar reduction in the number of repair circuits is seen for the other benchmarks and constraints.

<table>
<thead>
<tr>
<th>Design</th>
<th>No Constraint</th>
<th>A(50%)</th>
<th>B(75%)</th>
<th>C(90%)</th>
<th>D(95%)</th>
<th>E(99%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>161</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>system_test0</td>
<td>12</td>
<td>7</td>
<td>11</td>
<td>10</td>
<td>17</td>
<td>85</td>
</tr>
<tr>
<td>mult18</td>
<td>4</td>
<td>5</td>
<td>11</td>
<td>11</td>
<td>18</td>
<td>111</td>
</tr>
<tr>
<td>crazy</td>
<td>15</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>21</td>
<td>32</td>
</tr>
<tr>
<td>multxor</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>21</td>
<td>108</td>
</tr>
<tr>
<td>Time</td>
<td>8.7x</td>
<td>8.2x</td>
<td>12.9x</td>
<td>14.8x</td>
<td>22.1x</td>
<td>28.5x</td>
</tr>
</tbody>
</table>

The primary reason that the placement time has been decreased is the ability of the Cost Repair approach to repair a large number of sites during each iteration of the algorithm. Figure 4.9 plots the number of repairs performed during each iteration of the multxor benchmark under the B constraint.

Most of the work is done during the first seven iterations of the algorithm where an average of 1400 repairs are performed during each iteration.

The bottom row of Table 4.4 indicates the increase in time required for Cost Repair placement when compared with the original, non-repair placement. As expected, the cost increases as the area constraints tighten. On average and across all benchmarks and constraints, the Cost Repair placer executed 13.9× longer than the original, non repair placement. On average, the overall placement cost of the Cost Repair was 105% of the cost of the original placements.

4.2.3 Shadow Placement

While successfully able to create valid repair sets, the Naive and Cost Repair approaches for repair placement do not reuse any placement information from one placement iteration to the next. Another style of repair placement, called Shadow Placement, was developed to exploit the advantages of incremental placement. Unlike the previous two approaches, Shadow Placement will generate a complete repair set within a single placement iteration.
Like traditional placement, a unique FPGA resource is allocated for each circuit primitive in the netlist. This resource is used during the initial configuration and is named the “main” resource. In addition, a replacement resource, named the “shadow” resource, is also allocated for each circuit primitive in the netlist. This shadow resource is reserved for a potential repair if the original resource becomes faulty. Shadow placement operates by placing each circuit primitive at two locations - once for its “main” site (when there are no faults in the device) and once for its “shadow” site (when the main site is at fault).

The “main” resource for each primitive in the netlist must be unique and distinct from the resources used by all other primitives in the netlist. The resource allocated for “shadow” sites, however, can be shared with the “shadow” sites of other circuit primitives. Since this repair strategy will only guarantee the repair of any single resource failures, there is no need to allocate a dedicated repair resource for each circuit primitive. In fact, shadow resources are shared as much as possible to reduce the overhead of the repair circuit. These shadow resources are distributed throughout the device as dictated by the placer to insure that adequate redundant resources are available while still minimizing the placement cost.
Figure 4.10 demonstrates the placement of the main and shadow primitives. In this simplified example, there are four netlist primitives: L1, L2, L3, and L4. Dedicated “main” resources are allocated for each of these primitives and are annotated as bold in the figure. Only one primitive is allocated for each primary resource (site (0,1) for L1, site (1,1) for L2, site (2,1) for L3, and site (0,0) for L4). Shadow resources are also allocated for each primitive but these resources can be shared. The shadow resources are indicated in the figure with braces (i.e., \{L1\}, \{L2\}, \{L3\}, and \{L4\}). In this example, three shadow primitives (L1, L2, and L4) are allocated at the site (1,0) and L3 is allocated to site (2,0).

![Figure 4.10: Placement of “Main” and “Shadow” Resources.](image)

When the device is functioning normally and without fault, the four primitives are placed at their “main” sites ( (0,0), (0,1), (1,1), and (2,1)) and the sites (1,0) and (2,0) are idle. If a permanent fault occurs at site (1,1) where primitive L2 is located, a new circuit placement is used in which primitive L2 is moved to its shadow site (1,0) (see Figure 4.11).

This example is not necessarily an efficient placement of the shadow resources. In this example, two shadow resources are allocated for four primary resources for a 50% overhead. Ideally, shadow placement will share more shadow resources and significantly reduce the overhead to support repair.
Algorithm Overview

The Shadow Placement approach is based on the conventional simulated annealing placer described previously. Unlike the previous repair placement approaches, the Shadow repair placer will determine a repair within a single placement iteration. The first difference occurs during the generation of the initial placement. The algorithm begins by placing all of the shadow resources for the design. In particular, the algorithm chooses a small set of placement sites within the device in which all shadow resources reside. The size of this set is based on the size of the largest placement group. Placing all shadow resources at the same site is done to maximize the remaining sites for main placement. If too many shadow sites are allocated during the initial placement, it may not be possible to find sufficient sites for the main resources and the placement cannot proceed. After placing all shadow resources, the main resources are randomly placed within the remaining sites.

The second major difference is the cost function. The cost function is based on the cost function of Equation (4.1) but is modified to take into account the location of the shadow resources. The wire length of the routing is estimated by measuring the $x$ and $y$ dimensions of the bounding box that contains both the main and shadow terminals of each net (Figure 4.12 demonstrates this difference). If only the main placement sites are considered, this bounding
box has a size of 6 \((x = 3, y = 3)\). Since the shadow sites for L1 and L3 lie outside of this bounding box, the bounding box for Shadow placement is expanded. In this case, the bounding box size for this net is 8 \((x = 4, y = 4)\). To minimize the cost function, the resources allocated for shadow sites must be placed relatively close to its corresponding site and ideally will not increase the bounding box of a net.

The third difference between the conventional placer and the Shadow Placer is the way in which potential placement moves are created. Like the conventional annealing placer, a random resource is chosen as a candidate for movement. Since each resource has a corresponding main and shadow resource, the main or shadow version of the resource is randomly selected. Next, a
random target site is selected as a potential destination for the selected resource. If the selected
target site is free, the move is considered for acceptance. If the target site is not free, the shadow
placer will attempt to swap the resource with the resource(s) located at the target site. If the
target site contains shadow resources, all of the target shadow resources are swapped with the
randomly selected resource.

Results

An important figure of merit for the Shadow Placer is the number of shadow sites allo-
cated during placement. Table 4.5 summarizes the number of shadow sites allocated for each
benchmark and constraint pair. As the constraint tightens, fewer idle resources are available
and fewer shadow sites are allocated. Although the Shadow Placement executes in a single
placement iteration, each iteration must place twice as many resources (i.e., a “main” and a
“shadow” for each resource). The average increase in time of the Shadow placer over the conven-
tional placer is summarized in the final row of Table 4.5. On average, the Shadow placer takes
2.9× longer than the conventional placer and unlike the previous approaches, the amount of
time required to complete a shadow placement does not increase with a tighter area constraint.

<table>
<thead>
<tr>
<th>Design</th>
<th>No Constraint</th>
<th>A(50%)</th>
<th>B(75%)</th>
<th>C(90%)</th>
<th>D(95%)</th>
<th>E(99%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>29</td>
<td>23</td>
<td>16</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>system_test0</td>
<td>218</td>
<td>122</td>
<td>69</td>
<td>31</td>
<td>19</td>
<td>N/A</td>
</tr>
<tr>
<td>mult18</td>
<td>482</td>
<td>167</td>
<td>110</td>
<td>52</td>
<td>26</td>
<td>N/A</td>
</tr>
<tr>
<td>crazy</td>
<td>884</td>
<td>878</td>
<td>646</td>
<td>357</td>
<td>175</td>
<td>55</td>
</tr>
<tr>
<td>multxor</td>
<td>4094</td>
<td>1773</td>
<td>826</td>
<td>431</td>
<td>282</td>
<td>101</td>
</tr>
<tr>
<td>Time</td>
<td>2.12x</td>
<td>3.43x</td>
<td>3.01x</td>
<td>3.1x</td>
<td>2.48x</td>
<td>3.03x</td>
</tr>
</tbody>
</table>

The primary disadvantage of the Shadow Placer, however, is the lower quality of some
repairs. Unlike the other repair placement techniques, the shadow placer must reserve a large
number of extra resources for placement of shadow sites. These shadow sites are spread out
throughout the area constraint and increase the size of the net bounding boxes. In some cases,
shadow sites are placed very far from their corresponding main sites and significantly increase
the placement cost. On average, the cost of all circuit repairs using this approach is 41% higher
than the original non-repair placement.
The cost of each repair is not the same with shadow placement - some repairs lower the cost while others increase the cost. Figure 4.13 provides a distribution of the cost deviation from the initial placement of the `multxor` benchmark. As seen in this distribution, many repairs do not change the cost or even reduce the cost of the placement. However, for this tight constraint (E), the cost increases for a large number of repairs. There are a large number of repairs that significantly increase the placement cost. In summary, the Shadow repair approach has a significant advantage in terms of execution time as the placer only performs one placement iteration. The primary disadvantage of this approach, however, is the relatively low quality of some circuit repairs.

![Figure 4.13: Distribution of Wirelength Cost Deviation for Shadow Repairs.](image)

### 4.2.4 Hybrid Placement

The fourth and final approach is a hybrid approach that combines the iterative placement style of the Naive and Cost Repair approaches with the concept of pre-allocated shadow resources in the Shadow approach. The theory is that this approach can achieve similar quality placements
to the Cost Repair approach while simultaneously reducing execution time like the Shadow approach.

**Algorithm Overview**

The overall algorithm structure is listed in Algorithm 4. Like the Naive and Cost Repair approaches, this algorithm will perform repair placement during multiple iterations where each iteration will repair some subset of the device resources. This outer loop will initiate a new iteration (line 3) until all resources have been repaired.

**Algorithm 4 Hybrid Repair Placement**

1. $D \leftarrow$ set of all possible placement sites
2. $S \leftarrow$ set of occupied sites in initial placement
3. while $S \neq \emptyset$ do
4. choose $S' \subseteq S$, remove $S'$ from $S$
5. Perform Shadow Placement (shadow repair for all $s \in S'$
6. Perform repair placement(site $s$ not available)
7. Identify all resources, $R \in S$ that can be repaired with no increase in cost
8. Remove $R$ from $S$
9. end while

Each iteration of the loop will perform a repair on a random subset of resources in the design using the Shadow Placement style of repair (i.e., preallocate repair sites for the set of resources to repair). Currently, the algorithm will select 5% of the design resources for repair during each iteration. Unlike the Shadow Placement approach described in the previous section, only a subset of resources are repaired. A smaller number of repairs are performed in each iteration as the overall quality of the circuit will increase when fewer circuit resources are preallocated for repair. A unique Shadow Placement is performed during this iteration and will guarantee a repair for each resource by allocating a shadow location (line 5).

After completing Shadow placement, a number of primitive sites are reserved and allocated for repair. After completing the shadow placement, this algorithm looks to see if any of the other groups can be repaired by placing shadow sites at any of the currently allocated shadow sites. If the cost of any such repair is 5% higher than the average cost of a repair, the repair is rejected and added back to the set of groups still needing to be repaired. Rejecting repairs that increase the cost more than 5% is done to increase the quality of the repairs.
Results

The execution time of the Hybrid placer when applied to the benchmarks are summarized in Table 4.6. The overall increase in execution of the Hybrid placer over the non-repair placer is 15.6. Because the Hybrid placer performs multiple iterations, it takes longer than the Shadow repair approach. It is likely there are ways to speed up the Hybrid placer, but that will be left for future work.

The quality of the Hybrid repair placer, however, is far superior than the Shadow placer as expected. On average, the Hybrid repair placer created repair circuits that are 2% higher cost than the conventional placer.

<table>
<thead>
<tr>
<th>Constraint</th>
<th>No Constraint</th>
<th>A(50%)</th>
<th>B(75%)</th>
<th>C(90%)</th>
<th>D(95%)</th>
<th>E(99%)</th>
<th>× Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>14.3</td>
<td>19.2</td>
<td>13.0</td>
<td>15.9</td>
<td>16.2</td>
<td>15.4</td>
<td></td>
</tr>
</tbody>
</table>

4.2.5 Repairing Multiple Faults

Although the algorithms described in this thesis were not designed to repair more than one fault, it is likely that a given repair circuit will repair more than one fault. The repair circuits for the crazy benchmark were analyzed to see how many double faults can be repaired. With 3373 slices in this design, there are 3373×3372=11,373,756 possible two-slice pairs. Each pair of circuit resources used in the original design is checked against each of the repair configurations. If there is a repair configuration that does not use both of the faulty sites, the repair configuration is able to repair the given double fault. If either of the sites are used in the repair configuration, the repair configuration cannot repair the double fault. The results of this analysis are shown for all five area constraints in Table 4.7. For loose area constraints, the repair configurations can repair most double faults. However, as the area constraint tightens, fewer and fewer double faults can be repaired.
Table 4.7: Percentage of Two-Site Faults That Can Be Repaired For The crazy Benchmark

<table>
<thead>
<tr>
<th></th>
<th>A(50%)</th>
<th>B(75%)</th>
<th>C(90%)</th>
<th>D(95%)</th>
<th>E(99%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>64.2%</td>
<td>43.0%</td>
<td>13.8%</td>
<td>5.5%</td>
<td>3.6%</td>
</tr>
</tbody>
</table>

4.3 Summary

This chapter presents a baseline placement algorithm and four placement algorithms to support in-field repair. These algorithms pre-allocate repair resources for all circuit primitives in the device for use in the presence of a permanent fault. Four different algorithms were presented that vary in approach, execution time, and quality of repair result. The results of each approach are summarized in Table 4.8. The Naive repair approach is computationally expensive and is not feasible for large, highly utilized designs. The Shadow repair approach is less computationally intensive but generates repair circuits that are inferior to traditional placement approaches. The Cost and Hybrid repair approaches create circuits that are comparable in quality to traditional placement but require about $14 \times$ more execution time than a conventional placer. This work demonstrates that it is possible to perform repair placement that anticipates all possible repairs within a reasonable amount of time. After placement, the circuits are ready to be routed which is the topic discussed in the next chapter.

Table 4.8: Execution Time and Placement Algorithms When Compared to Non-Repair Placement

<table>
<thead>
<tr>
<th>Technique</th>
<th>Run Time</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive Repair</td>
<td>$210 \times$</td>
<td>1.02</td>
</tr>
<tr>
<td>Cost Repair</td>
<td>$13.9 \times$</td>
<td>1.05</td>
</tr>
<tr>
<td>Shadow Repair</td>
<td>$2.9 \times$</td>
<td>1.41</td>
</tr>
<tr>
<td>Hybrid Repair</td>
<td>$15.6 \times$</td>
<td>1.03</td>
</tr>
</tbody>
</table>
Chapter 5

Routing

Routing is the process of connecting input and output pins of primitives. There are several challenges associated with routing. FPGAs have fixed amount of routing resources available [31], and this causes routes to compete for resources to find the best possible path. This problem is especially true in highly congested areas. A router must be able to allocate resources so that all pins can be routed, while simultaneously creating a high quality route for all pins.

This chapter explains the basics of routing. It demonstrates what it means to route from a source to a sink. It also explains the PathFinder routing algorithm that is used to solve the routing congestion problem. It additionally shows the results from the baseline router and compares them with the Xilinx commercial router. Finally, the repair routing algorithm is described and the results are discussed.

5.1 Baseline Routing Algorithm

A conventional routing algorithm was developed to perform routing for the designs placed by the baseline placer. Like the placer, the baseline router was written in Java and built upon the RapidSmith toolkit [19, 27]. The baseline placer operates on the XDL files created from the baseline placer. The router reads the XDL, performs routing, and saves the completed placement and routing in an XDL file (see Figure 5.1). The file is then converted back into the binary format and vendor tools are used to generate a valid FPGA bitstream.

Figure 5.1: Design Flow for Baseline Routing
The baseline router was based on the well-known VPR FPGA routing approach which is based on the PathFinder negotiated congestion algorithm [28]. The router connects subsets of pins in a design. A subset of pins that need to be connected together are assigned to the same route (also called a net). The source pin is where the net originates and the sink pin is where it terminates. A net can have multiple sinks, but it can only have one source. Each net is routed individually one at a time. The method for routing a net is shown in Algorithm 5.

It is done by creating nodes for the source and sink pins. A node is an instance of the Node class which is RapidSmith’s technique to store and retrieve routing resources [19]. Nodes are added to a priority queue to await processing. The cost function used to sort the queue is the Manhattan distance of the current node to the sink node. The shortest distance has highest priority. Initially, the current node is the source node and the priority queue contains only the source. The current node is processed by adding all of its adjacent nodes to the priority queue. The algorithm continues processing the first node in the queue until the the sink has been found or there are no nodes left in the queue.

**Algorithm 5** Route Source to Sink

1: Current node $\textit{currNode} \leftarrow$ source node  
2: Initialize priority queue $\textit{pq}$  
3: while $\textit{currNode} \neq$ sink node do  
4:   for each node $\textit{adjNode}$ adjacent to $\textit{currNode}$ do  
5:      Calculate cost of $\textit{adjNode}$  
6:      Add $\textit{adjNode}$ to $\textit{pq}$  
7:   end for  
8:   if $\textit{pq}$ is empty then  
9:      return sink not found!  
10:  end if  
11: $\textit{currNode} \leftarrow$ highest priority node in $\textit{pq}$  
12: end while

In the PathFinder algorithm, each net is initially routed using the shortest path from source to sinks regardless of whether other nets are already using the routing resources (see Algorithm 6). After every net has been routed, the routes are compared to find which resources are being shared between routes. Any nets that are sharing resources are ripped up and flagged to be routed again. The cost of the shared resource is increased to encourage routes to find alternate paths. By gradually increasing the cost, a resource will be allotted to the net that most
needs it. The router then performs another iteration of routing, evaluating shared resources and increasing resource costs. This process continues until there are no shared resources between routes.

Algorithm 6 PathFinder Algorithm

1: Route each net
2: Check for shared resources between nets
3: while shared resources exist do
4: Increase cost of shared resources
5: Rip up and reroute nets
6: Check for shared resources between nets
7: end while

The baseline router has some slight variations from other PathFinder approaches. Some implementations rip up the entire route for a net using a shared resource [31]. Some even rip up every route between each iteration. Like the VPR router [28], this baseline router only rips up the shared resource, leaving the rest of the route behind to be used in the next iteration. This way the net can be rerouted very quickly because there are already a large number of connections in place. For more information on how the PathFinder router was implemented in RapidSmith see Appendix B.

5.1.1 Baseline Routing Results

The execution time, overall cost, and minimum clock period of circuits routed with our baseline router are shown in Table 5.1. A fourth metric is also used to determine quality which is the total number of PIPs used in the configuration. The multxor benchmark is not included because it would require several months for the baseline router to complete. The cost represents the quality of result of the route by measuring the total wire length using a bounding box span (see Equation 4.1). The original cost function measures a bounding box around just the logical resources because during placement the routes are unknown. During routing the routes can extend outside of the original bounding box. The cost function is modified to expand the bounding box to include routing resources and provide a more accurate estimation of total used wire length (see Figure 5.2).

To provide a reference, the execution time and cost for vendor routing is also provided. To facilitate comparison, the same cost function was used for both the vendor routing and
Table 5.1: Routing Time (s) and Quality of Result for Baseline and Vendor Routing Tools

<table>
<thead>
<tr>
<th>Design</th>
<th>Baseline RapidSmith</th>
<th>Xilinx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Cost</td>
</tr>
<tr>
<td>top</td>
<td>5.47</td>
<td>482</td>
</tr>
<tr>
<td>system_test0</td>
<td>33.92</td>
<td>8352</td>
</tr>
<tr>
<td>mult18</td>
<td>28.03</td>
<td>12058</td>
</tr>
<tr>
<td>crazy</td>
<td>453.5</td>
<td>81354</td>
</tr>
</tbody>
</table>

Figure 5.2: This figure shows the bounding box of a net for two stages of a design. On the left, the net has been placed but not routed so the bounding box only encompasses logical resources. On the right, the net has been placed and routed so the bounding box is expanded to include routing resources.

The baseline routing approach described here is inferior to a commercial routing approach in both execution time and quality of results. There are many ways to improve the quality and runtime of the approach. However, the focus of this work is not to replicate commercial quality routing but to demonstrate the feasibility of a routing approach that considers repair.

5.2 Repair Routing

The goal of the repair routing approach described in this thesis is to determine a valid FPGA route on a fully functional device for the various placements generated by the repair placers. Some of the placers output numerous placement files and the router must simply route each
placement. Other placers only generate one placement file and provide additional information that the router needs to generate the completely placed and routed repair configurations.

Like with placement, the disadvantage of this approach is the large amount of computation that is required to create the repair configurations. Generally, the router must be run repeatedly, although in some cases only partial re-routes are needed. The advantages are that the router can usually retain information between routes to speed up execution time, and it is possible to know before failure which routing resources can be repaired.

The repair router described in this work will generate the initial route of a design. This initial route specifies a set of routing resources which are used when no permanent faults are present. In RapidSmith, a specific physical routing connection is defined by the tile object (which specifies the location) and the wire. This pair is called a Tile Wire Connection (TWC) and is the routing resource that is repaired by the routers in this thesis. Each TWC in the set needs a corresponding repair, which is an alternate routing configuration where that resource is not used. The router tracks what resources have been repaired after each iteration.

This section describes the repair routing approach developed. The Cost Repair Router is used for both the Naive and Cost Repair placement algorithms. A possible implementation to route the shadow placement is described in Appendix C. It is important to note that unlike placement resources, it is difficult to guarantee that all routing resources can be repaired.

Because of the placement, some routing resources are critical and cannot be repaired. For example, if the $YQ$ pin of a CLB is the source of a net (see Figure 5.3), the wire attached to that pin must be used in the route, and it cannot be avoided. The only way to repair the wire is to move the CLB to another location. Another possible reason for not being able to route resources is that in highly congested areas there may not be any spare resources available.

The routing approaches are relatively simplistic in that they do not actively seek to repair specific resources. The placement algorithms actively repaired primitive sites by prohibiting the use of a site or by duplicating resources for each primitive. The general repair routing approach is to route a placement or a partial placement and then check to see which resources were naturally repaired. There are likely ways to encourage the routers to actively repair more resources, but that is a topic for future work.

An example of routing resources being repaired is shown in Figure 5.4. In this simplified device, there are nine primitive sites. There are also 36 wires (gray horizontal and vertical lines) available to connect the primitives. The black boxes are programmable switch matrices that
connect the wires. Normally there are additional routing resources connecting the primitives to the wire grid, but for demonstrative purposes they can be ignored. In the initial configuration, the sites are occupied by primitives X, Y and Z. The source of the net comes from X and there are two sinks in Y and Z. The initial configuration uses four wires in the wire grid to route the source and sinks. Two of the used wires are labeled, one is A, the other B. In the repair configuration, X, Y and Z were moved to different locations and routed again. This configuration does not use wire A, therefore wire A has been repaired. However, wire B is still used. A different configuration would be required to repair wire B.

5.2.1 Cost Repair Router

The first approach for repair routing is designed to work in tandem with the naive placer and the cost repair placer. Although it can be used with the naive placer, the naive placer generates an order of magnitude more repair placement configurations than the cost repair placer. For time constraints, the cost repair router was only tested using the placements generated by the cost repair placer.
Figure 5.4: Initial and repair configurations of a simple device. There are nine primitive sites and a grid of wire segments. The black boxes are programmable switch matrices to connect wire segments. The repair configuration repairs wire $A$, but it does not repair wire $B$.

Algorithm Overview

As stated earlier, naive and cost repair placers generate a number of placed XDL files. The cost repair router simply iterates though each placement and routes them one at a time using the baseline router. The first placement to be routed is the original placement to be used when there are no permanent faults. After it is completely routed, a set $S$ is defined as the tile wire connections (TWCs) that are used in the initial route.

\begin{algorithm}
\begin{algorithmic}
\STATE $P \leftarrow \text{set of all repair placements}$
\STATE $D \leftarrow \text{set of all TWCs}$
\STATE Perform initial routing
\STATE $S \leftarrow \text{set of TWCs used in initial route}$
\FORALL{ $p$ in $P$ }
\STATE Perform repair routing on $p$
\STATE $R \leftarrow \text{set of TWCs used in route}$
\STATE $G \leftarrow (D - R) \cap S$ (sites repaired)
\STATE $S \leftarrow S - G$
\ENDFOR
\end{algorithmic}
\end{algorithm}

The algorithm proceeds by getting the next repair placement and performing a completely new, independent route. No routing information is saved between iterations. After
completing all iterations, if the set $S$ is not empty, then there are routing resources that were not repaired. A routing resource is repaired when it is not used in one or more of the routing sequences.

**Results**

The Cost Repair routing approach was applied to all of the benchmarks placed by the Cost Repair Placer. The number of routed repair circuits is equal to the number of repair placements as shown previously in Table 4.4.

Table 5.2 indicates the increase in time required for Cost Repair routing when compared with the original, non-repair routing. This is due to the need to route every repair file individually. As expected, the time cost increases as the area constraints tighten. On average and across all benchmarks and constraints, the Cost Repair router executed approximately $100\times$ longer than the original, non-repair router.

The quality of the routes are determined by the cost, the number of pips, and the minimum clock period. The results of these three metrics are summarized in Table 5.3, Table 5.4, and Table 5.5 respectively. On average, the overall cost of circuits generated by the Cost Repair router was 105% of the cost of the original circuits.

<table>
<thead>
<tr>
<th>Constraint</th>
<th>No Constraint</th>
<th>A(50%)</th>
<th>B(75%)</th>
<th>C(90%)</th>
<th>D(95%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>3.63</td>
<td>3.263</td>
<td>82.91</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>system_test0</td>
<td>545.35</td>
<td>277.82</td>
<td>486.07</td>
<td>464.37</td>
<td>1169.79</td>
</tr>
<tr>
<td>mult18</td>
<td>1225</td>
<td>2817</td>
<td>3492</td>
<td>472</td>
<td>17948</td>
</tr>
<tr>
<td>crazy</td>
<td>24660</td>
<td>N/A</td>
<td>26237</td>
<td>27598</td>
<td>34017</td>
</tr>
<tr>
<td>× Increase</td>
<td>43.5</td>
<td>58.2</td>
<td>69.2</td>
<td>80.6</td>
<td>107</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constraint</th>
<th>No Constraint</th>
<th>A(50%)</th>
<th>B(75%)</th>
<th>C(90%)</th>
<th>D(95%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>729</td>
<td>469</td>
<td>472</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>system_test0</td>
<td>7258</td>
<td>7117</td>
<td>7522</td>
<td>7447</td>
<td>8543</td>
</tr>
<tr>
<td>mult18</td>
<td>13783</td>
<td>16059</td>
<td>18082</td>
<td>18066</td>
<td>16955</td>
</tr>
<tr>
<td>crazy</td>
<td>81405</td>
<td>N/A</td>
<td>77909</td>
<td>90860</td>
<td>90709</td>
</tr>
</tbody>
</table>
Table 5.4: Number of PIPs Using Cost Repair Router

<table>
<thead>
<tr>
<th>Design</th>
<th>No Constraint</th>
<th>A(50%)</th>
<th>B(75%)</th>
<th>C(90%)</th>
<th>D(95%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>985</td>
<td>838</td>
<td>842</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>system_test0</td>
<td>12888</td>
<td>12849</td>
<td>13014</td>
<td>12952</td>
<td>13361</td>
</tr>
<tr>
<td>mult18</td>
<td>16733</td>
<td>17324</td>
<td>17707</td>
<td>17713</td>
<td>17484</td>
</tr>
<tr>
<td>crazy</td>
<td>108657</td>
<td>N/A</td>
<td>108196</td>
<td>111474</td>
<td>111818</td>
</tr>
</tbody>
</table>

Table 5.5: Minimum Clock Period(ns) using Cost Repair Router

<table>
<thead>
<tr>
<th>Design</th>
<th>No Constraint</th>
<th>A(50%)</th>
<th>B(75%)</th>
<th>C(90%)</th>
<th>D(95%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>4.319</td>
<td>2.244</td>
<td>2.582</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>system_test0</td>
<td>1.425</td>
<td>1.203</td>
<td>1.241</td>
<td>1.504</td>
<td>1.3675</td>
</tr>
<tr>
<td>mult18</td>
<td>3.04</td>
<td>3.252</td>
<td>3.954</td>
<td>3.889</td>
<td>3.541</td>
</tr>
<tr>
<td>crazy</td>
<td>7.953</td>
<td>N/A</td>
<td>9.708</td>
<td>7.467</td>
<td>7.271</td>
</tr>
</tbody>
</table>

As stated previously, the Cost Repair router was not designed to actively repair every routing resource. Table 5.6 summarizes the percentage of TWCs repaired for each benchmark. Nearly 100% of all TWCs were repaired for the crazy benchmark. On average, approximately 3% of routing resources are not repaired by the Cost Repair Router.

Table 5.6: Percentage of TWCs repaired using Cost Repair Router

<table>
<thead>
<tr>
<th>Design</th>
<th>No Constraint</th>
<th>A(50%)</th>
<th>B(75%)</th>
<th>C(90%)</th>
<th>D(95%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>84.9</td>
<td>78.3</td>
<td>87.7</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>system_test0</td>
<td>97.86</td>
<td>97.73</td>
<td>97.65</td>
<td>98.22</td>
<td>98.42</td>
</tr>
<tr>
<td>mult18</td>
<td>99.97</td>
<td>96.5</td>
<td>96.33</td>
<td>97.76</td>
<td>97.79</td>
</tr>
<tr>
<td>crazy</td>
<td>99.9</td>
<td>N/A</td>
<td>99.87</td>
<td>99.87</td>
<td>99.89</td>
</tr>
</tbody>
</table>

5.3 Summary

This chapter presents the baseline routing algorithm and a routing algorithm to support infield repair. This repair algorithm routes the placed designs generated by the Naive and Cost Repair placers described in Chapter 4. The Cost Repair approach creates circuits that
are comparable in quality to traditional routing but require $100 \times$ more execution time than a conventional router. A theoretical approach that works with Shadow placement is proposed in Appendix C. This work demonstrates that it is possible to perform repair routing that anticipates nearly all possible repairs, but at a very high run-time cost.
Chapter 6

Conclusion

This work is the only known method that makes it possible to anticipate and generate repair configurations for nearly all possible permanent faults before the fault occurs. These new placement and routing approaches increase the lifetime of FPGAs by repairing 99% of permanent faults with 5% overhead in cost.

This thesis presents four placement algorithms and one routing algorithm to support in-field repair. The placement algorithms pre-allocate repair resources for all logic primitives in the device for use in the presence of a permanent fault. The routing algorithms route the repair placements generated by the placers, while repairing nearly all routing resources. The end result is a set of repair files that can be used to reconfigure the device in the event of a permanent fault.

There is a trade-off between execution time and quality of repair between the different algorithms. The Shadow approach executes quickly, but the injection of shadows produces repair circuits of lower quality than traditional approaches. The Cost Repair approach creates circuits comparable to traditional methods, but executes an order of magnitude slower.

The difficulty in manufacturing semiconductors and the increased effects of wear-outs at small geometries makes it necessary to address permanent faults. Because of the fine-grain reconfigurability of FPGAs, repairs of permanent faults can be made by modifying the placement and routing of a design. Systems that take advantage of this capability can significantly increase their lifetime and availability.

6.1 Future Work

There are many opportunities for the work presented in this thesis to be expanded. For example, the execution time of the Hybrid placer could likely be decreased so it is more of a balance between the Cost Repair and Shadow approaches. Another area that can be improved is that ideally all resources on the device should be repaired. Currently a small percentage of
routing resources are not repaired. There are likely methods that could be used to encourage the router to repair the last few routing resources.

Another item of future work is to develop routers that can route the Shadow and Hybrid placements. A possible implementation of a Shadow router is outlined in Appendix C. This approach could then be adapted to work with Hybrid placements.

Finally, the algorithms presented attempt to repair all possible single faults. If more than one permanent fault is present and the device is highly utilized, it is unlikely this approach can repair the device. The algorithms could likely be modified to encourage repair of multiple faults.
Bibliography


Appendix A

Placement in RapidSmith

A baseline placer was developed in RapidSmith in order to explore the various repair placement approaches as well as to provide a baseline for comparison. This appendix provides details on the implementation of the placer in RapidSmith. The baseline placer uses Simulated Annealing and is based on the VPR FPGA placement approach [28].

A.1 Simulated Annealing Placer

This section provides additional information for the Simulated Annealing placer that was developed in RapidSmith. The Simulated Annealing algorithm was summarized previously in Section 4.1.1. This section describes key classes and their important methods, and it summarizes some key steps to provide insight into how the code was written. This section also includes actual code of the main parts of placement.

A.1.1 Key Classes

PlacementGroup

This class represents the atomic unit for placement. Because of placement constraints between instances (i.e., instances of a carry chain), multiple instances may be part of an atomic placement group. In other words, all instances of a given placement group have a specific relative placement constraints and when one of the instances are placed, all of the instances in the group are correspondingly placed. This class does not maintain any placement information. All of the placement information is stored in the PlacerState. Every placement group has a single instance designated as the anchor. All other instances of the group are placed relative to the anchor. This class is extended to form multiple instances placement groups and single instance placement groups.

DesignPlacementGroups

Creates all of the “static” information necessary for a design to be placed on a particular device. This static information includes PlacementGroup objects of a design and PlacementGroup objects that cannot be placed by this placer. This class also manages a map between instances in the design and the PlacementGroup object that they belong to. This facilitates the identification of PlacementGroups during placement.

Key Method:

- createPlacementShapeGroups() - Identify all of the atomic placement groups and create the data structure for each group. This is determined using the shape information of XDL instances as specified by the _INST_PROP attributes found within an XDL instance.
**PlacerState**

This class manages the dynamic placement state of all placement groups during the placement process. The placement state changes frequently during placement and this state is managed locally. The actually placement state is not set in the XDL file until after placement and by calling the “finalizePlacement” method.

Key Methods:

- placeGroup() - Modify placement information of a group. This method assumes that the PrimitiveSite anchor used for placement is valid. If the new site is null, unplaceGroup will be called.
- unplaceGroup() - Remove the placement information for a given group. The group is considered “unplaced” with no location after calling this method.
- isGroupOverlapping() - Determines if the group will overlap another group if placed at the primitive site.

**PlacerMove**

Represents an atomic move between one or more PlacementGroup objects. In some cases, this will be a single object that is being moved from its current location to an empty, available location. In other cases, this move will involve multiple groups (usually a swap between groups at different locations). The set of moves should have already been checked for validity (i.e., that the atomic set of moves are mutually compatible).

Key Methods:

- makeMove() - Places all Instances specified in this move at their new PrimitiveSites.
- undoMove() - Returns all Instances of this move back to their previous PrimitiveSites.

**NetRectangleCostFunction**

A cost function that determines system cost based on nets and their distances. Based on the VPR cost function [28].

Key Methods:

- calcSystemCost() - Calculates the entire systems cost.
- calcIncrementalCost() - Calculates new cost after a move is taken.

**DisplacementRandomInitialPlacer**

An initial placer that tries to randomly place PlacementGroups. If a PlacementGroup cannot be placed randomly, other PlacementGroups are displaced to make room for it.
BasicPlacer

A basic placer. Contains a random number generator, a design, and placer state. Contains a number of methods for managing the movement and placement of placement groups. Other placement algorithms can be developed that use the methods in this class.

Key Method:

- proposeMove() - Propose a new location for a given PlacementGroup. This method will continually identify a random locations until it finds a valid location for the group. Once a valid location is found, it calls createSwapMove(PlacementGroup, PrimitiveSite) to create a move involving a swap. Note that while a valid location for the group may be found, its corresponding swap move may not be legal and the move is not created. Note that this method does not actually perform the move - it simply finds a valid location.

SimulatedAnnealingPlacer

This placer extends the basic placer to form a simple implementation of simulated annealing. At the beginning of the anneal, most moves are accepted even if they increase the system cost. As it cools, fewer moves are accepted. The annealing schedule is determined by the initial temperature, the number of moves (or steps) to attempt per temperature, how quickly the temperature should cool, and the exit criteria.

A.1.2 Key Steps

1. Initialize placer state to keep track of all placement information.
2. Create Placement Groups from XDL shapes.
3. Initial random placement of PlacementGroups.
4. Initialize the annealing schedule.
5. For a temperature.
   (a) For number of steps per temperatures.
      • Identify a move.
      • Make move.
      • Calculate new cost.
      • If cost is lower or a random number is less than the move threshold keep move, else undo move.
   (b) Check to see if annealer is making progress.
   (c) Calculate new temperature.

A.1.3 Code

```java
public boolean simulatedAnnealingPlace(InitialPlacer initialPlace,
                                      PlacerCostFunction cost,
                                      boolean ignoreInitialPlacement, PlacerEffortLevel level) {
```
PlacerEffortLevel placeEffort = level;

// Perform initial placement
System.out.println("Instances: " + design.getInstances().size());
ArrayList<PlacementGroup> allGroups =
    new ArrayList<PlacementGroup>(state.getPlacementGroups());
boolean initialPlaceSuccessful = initialPlace.initialPlace(state,
    ignoreInitialPlacement);

// Check to see if the initial placer was successful or not
if (!initialPlaceSuccessful) {
    System.out.println("Unsuccessful initial place");
    return false;
}

// Initialize annealing schedule
float currCost = cost.calcSystemCost();
float initialCost = currCost;
float oldTempCost = currCost;
System.out.println("Initial placement cost: " + initialCost);

if (DEBUG >= DEBUG_MEDIUM)
    System.out.println("== MAKING MOVES FOR INITIAL TEMPERATURE ==");
float temperature = findInitialTemperature(allGroups, currCost, cost)*1.5f;
if(temperature != temperature){
    temperature = (float) 0.20377909;
}

float fractionOfMovesAccepted = 0;
//float alpha = 0;
//float endTemperature =
//    0.005f*currCost/NetRectangleCostFunction.getRealNets(design).size();
int numRealNets = NetRectangleCostFunction.getRealNets(design).size();
//float endTemperature = .005f*currCost/numRealNets;

int numMoves = 0;
// TODO: Use the constraint rather than the device size
int maxRangeLimit = design.getDevice().getColumns() +
    design.getDevice().getRows();
int rangeLimit = maxRangeLimit;

// Default values for Normal Mode
int MAX_TEMPERATURES_BELOW_COST_THRESHOLD = 5; //was 10
float percentageThreshold = 1.0f;
boolean usePercentMode = false;
// Set steps per temperature for different modes
float qualityMultiplier = HIGHER_QUALITY_MULTIPLIER;
if (placeEffort == PlacerEffortLevel.LOW){
qualityMultiplier = qualityMultiplier * .4f;
usePercentMode = true;
percentageThreshold = 5.0f;
MAX_TEMPERATURES_BELOW_COST_THRESHOLD = 3;
}
else if (placeEffort == PlacerEffortLevel.MEDIUM){
    qualityMultiplier = qualityMultiplier * .75f;
    usePercentMode = true;
    percentageThreshold = 1.0f;
    MAX_TEMPERATURES_BELOW_COST_THRESHOLD = 5;
}
else if (placeEffort == PlacerEffortLevel.HIGH){
    usePercentMode = true;
    percentageThreshold = 0.5f;
    MAX_TEMPERATURES_BELOW_COST_THRESHOLD = 10;
}
else if (placeEffort == PlacerEffortLevel.HIGH_L){
    qualityMultiplier = qualityMultiplier * .4f;
    usePercentMode = true;
    percentageThreshold = 0.5f;
    MAX_TEMPERATURES_BELOW_COST_THRESHOLD = 10;
}
else if (placeEffort == PlacerEffortLevel.HIGH_M){
    qualityMultiplier = qualityMultiplier * .75f;
    usePercentMode = true;
    percentageThreshold = 0.5f;
    MAX_TEMPERATURES_BELOW_COST_THRESHOLD = 10;
}
else if (placeEffort == PlacerEffortLevel.HIGH_H){
    qualityMultiplier = qualityMultiplier * 1.5f;
    usePercentMode = true;
    percentageThreshold = 0.5f;
    MAX_TEMPERATURES_BELOW_COST_THRESHOLD = 10;
}

int stepsPerTemp = (int)(Math.pow(allGroups.size(), 1.33f) * 
    qualityMultiplier);
System.out.println("Max Range Limit = "+maxRangeLimit+" steps per
temp="+stepsPerTemp);

// Initialize time counter
long initTime = System.currentTimeMillis();
long currTime = initTime;
long lastTime;
int lastMoves = 0;

System.out.println("Initial placement cost: " + currCost + " Initial 
    Temperature: " + temperature);

// Flag that indicates whether another temperature iteration should proceed
boolean keepGoing = true;
float COST_THRESHOLD = .05f * currCost/numRealNets;

int numTemperaturesBelowCostThreshold = 0;

System.out.println(" Cost Threshold = " + COST_THRESHOLD + " num temperatures below cost threshold=" + MAX_TEMPERATURES_BELOW_COST_THRESHOLD);
System.out.println("-----------------------------");
if (DEBUG >= DEBUG_MEDIUM)
    System.out.println("== STARTING TO MAKE REAL MOVES ==");

while (keepGoing) {
    int numTempMoves = 0;
    int numTempMovesAccepted = 0;

    // This loop will perform a single move. It will be done "stepsPerTemp" times.
    for (int j = 1; j < stepsPerTemp; j++) {
        if (DEBUG >= DEBUG_MEDIUM) {
            System.out.println("= Searching for a move at current cost of " + currCost);
        }

        // Identify a move
        PlacerMove move = null;
        while (move == null) {
            toSwapIdx = rng.nextInt(allGroups.size());
            PlacementGroup toSwap = allGroups.get(toSwapIdx);
            if (rangeLimit <= maxRangeLimit) {
                move = proposeMove(toSwap, true, rangeLimit);
            } else {
                move = proposeMove(toSwap);
            }
        }

        move.makeMove();
        float newCost = cost.calcIncrementalCost(move);

        float deltaCost = newCost - currCost;
        if (DEBUG >= DEBUG_MEDIUM) {
            System.out.print(" Move Cost=" + newCost + " (delta=" + deltaCost + ")");
        }

        boolean acceptMove;
        if (deltaCost < 0) {
            // if the cost is lowered, always accept the move.
            acceptMove = true;
        } else {
            if (DEBUG >= DEBUG_MEDIUM) {
                System.out.println("= Checking move with cost of " + newCost + ", saving " + deltaCost + ");
            }

            acceptMove = cost.evaluateMove(move, true);
        }

        if (acceptMove) {
            numTempMovesAccepted++;
            if (DEBUG >= DEBUG_MEDIUM) {
                System.out.println("= Accepted move with cost of " + newCost + ", saving " + deltaCost + ");
            }
        } else {
            numTempMoves++;
            if (DEBUG >= DEBUG_MEDIUM) {
                System.out.println("= Rejected move with cost of " + newCost + ", saving " + deltaCost + ");
            }
        }
    }
    numTemperaturesBelowCostThreshold++;

    keepGoing = !allTemperaturesDone;
}

System.out.println(" Number of temperatures below cost threshold was " + numTemperaturesBelowCostThreshold + ");
if (DEBUG >= DEBUG_MEDIUM) {
    System.out.println(" MOVE ACCEPTED");
}
else {
    // Accept some moves that increase the cost. The higher the increase
    // in cost, the lower the percentage of moves will be accepted.
    tempDiffCost = currCost - oldTempCost;
    System.out.println(" MOVE ACCEPTED");
    System.out.print(" Threshold="+moveThreshold+" rand="+r);
    acceptMove = false;
    if (DEBUG >= DEBUG_MEDIUM)
        System.out.println(" MOVE REJECTED");
}

if(acceptMove) {
    currCost = newCost;
    numTempMovesAccepted++;}
else {
    if (DEBUG >= DEBUG_MEDIUM)
        System.out.println(" Undo Move");
    move.undoMove();
    if (Math.abs(costIfMoveIsRejected-currCost) > MAX_DIFFERENCE) {
        System.out.println("Warning: Undo cost mismatch: initial cost " +
                           currCost + " new cost:" + newCost
                           + " after undo " + costIfMoveIsRejected+
                           " difference="+(costIfMoveIsRejected-currCost));
        //newMove.debugMove();
        //System.exit(-1);
    }
    numMoves++;
    numTempMoves++;}

float tempDiffCost = currCost - oldTempCost;
oldTempCost = currCost;
float differencePercentage = tempDiffCost/oldTempCost*100f;
boolean percentThresholdExceeded = differencePercentage <= 0 &&
-differencePercentage < percentageThreshold;
boolean tempThresholdExceeded = tempDiffCost > 0 || (-tempDiffCost) <
COST_THRESHOLD;
if (usePercentMode && percentThresholdExceeded || !usePercentMode &&
tempThresholdExceeded) {
    numTemperaturesBelowCostThreshold++;
    //System.out.println("Didn't meet threshold "+costThreshold+"
    #"+numTemperaturesBelowCostThreshold);
    if (numTemperaturesBelowCostThreshold >=
MAX_TEMPERATURES_BELOW_COST_THRESHOLD /* && rangeLimit == 1 */) {
        keepGoing = false;
        if(!usePercentMode)
            System.out.println("Did not meet threshold of "+COST_THRESHOLD+"for "+numTemperaturesBelowCostThreshold
            "+" consecutive temperatures");
        else
            System.out.println("The delta cost percent fell below -"+percentageThreshold+"% for "+numTemperaturesBelowCostThreshold
            "+" consecutive times.");
    }
} else
    numTemperaturesBelowCostThreshold = 0;

    // Compute Time
    lastTime = currTime;
currTime = System.currentTimeMillis();
int moves = numMoves - lastMoves;
long dTime = currTime - lastTime;
float movesPerMiliSecond = (float) moves / (dTime);
//int movesPerSecondInt = Math.round(movesPerSecond);
System.out.println("\tTime: "+(float) dTime / 1000 +" seconds. "+ moves+"moves. Moves per second: "+
(float)movesPerMiliSecond * 1000);
lastMoves = numMoves;

    // Compute new cost
    fractionOfMovesAccepted = (float)numTempMovesAccepted/(float)numTempMoves;
temperature = findNewTemperature(fractionOfMovesAccepted, temperature);

    //Check to see if temperature is NaN (not a number). Exits loop if
    //temperature is invalid.
    if (temperature != temperature){
        keepGoing = false;
    }

    String diffPercent = String.format("%3.3f",tempDiffCost/oldTempCost*100f);
System.out.println("\t\tNew cost="+currCost+ " delta cost: "+ tempDiffCost+ " ("+diffPercent+"%)");
//endTemperature =
0.005f*currCost/NetRectangleCostFunction.getRealNets(design).size();
rangeLimit = findNewRangeLimit(fractionOfMovesAccepted, rangeLimit,
maxRangeLimit);

System.out.println("\tRange Limit: " + rangeLimit);

//work harder in more productive parts of the anneal
if(rangeLimit < maxRangeLimit) {
    stepsPerTemp = (int)(Math.pow(allGroups.size(), 1.33f) *
    qualityMultiplier);
}
}

// Done. Reached the ending condition.
if (DEBUG >= DEBUG_LOW) {
    System.out.println("Final Placement:");
    for (PlacementGroup pg : state.getPlacementGroups()) {
        System.out.println(" +pg+:"+state.getGroupAnchorSite(pg));
    }
}

//System.out.println("Final cost: " + currCost);
long timeInMiliSeconds = (System.currentTimeMillis()- initTime);
float movesPerSecond = (float) numMoves / timeInMiliSeconds * 1000;
System.out.println("Final cost: " + currCost+ " ("+
    (currCost/initialCost)*100+"% of initial cost:"+
    initialCost+")");
System.out.println(numMoves+" Moves in "+(float)timeInMiliSeconds/1000+" seconds ("+movesPerSecond+" moves per second)");
state.finalizePlacement();
return true;
}
Appendix B

Routing in RapidSmith

A baseline router was developed in RapidSmith in order to explore the repair routing approach as well as to provide a baseline for comparing the results. This appendix provides details on the implementation of the router in RapidSmith. The baseline router uses the PathFinder algorithm and is based on the VPR FPGA routing approach [28].

B.1 PathFinder Routing

This section provides additional information for the PathFinder router developed in RapidSmith. The algorithm was summarized previously in Section 5.1. This section defines several new data structures used by this router. It also describes key classes for the PathFinder router and the important method associated with the class, and it summarizes some key steps to provide insight into how the code was written. This section also includes actual code of the main parts of routing.

B.1.1 Data Structures

It is essential to understand a number of the data structures in RapidSmith in order to understand, maintain, and expand the PathFinder router. The purpose of this section is to provide a summary of the key data structures created specifically for this router.

TileWireConnection

Represents a specific routing connection on the device. It is composed of a Tile object (which specifies the exact location) and an integer ‘Wire’. This class is made to include only these two members so that it will be as small as possible. Many of these objects will be created during routing so it is essential to keep this as efficient as possible. This class is similar to the Node class in RapidSmith.

NodeEdge

Similar to the RapidSmith Node object. This class is used for routing when finding routing connections between a source and a sink. It represents a TileWireConnection (i.e., a specific routing location) and a reference to a parent NodeEdge. The reference to a parent is used for backwards traversal to recreate the routing path from the source to the sink. The reference to the parent is done with a reference to a parent NodeEdge object (i.e., a reference to the parent).

This class also has a reference to a WireConnection object which specifies the WireConnection used to connect the parent to the child. This class is most similar to the Node class in RapidSmith. While this class is similar to the TileWireConnection class and the RouteEdge
class, its primary purpose is for searching during the route process. In this infrastructure, the NodeEdge is only used for searching. Once a path has been found, RouteEdge objects are used.

**RoutingForest, RoutingTree, AbstractRoutingForest, DefaultRoutingForest, and DefaultRoutingTree**

These interfaces and classes all represent the connectivity of an actual route. The purpose of the complex class organization is to facilitate reuse of a few single data structures (contained mainly within DefaultRoutingTree).

**Route**

Represents the routing information of a net. Specifically, it represents (1) the connection locations of the sources and sinks of the net and (2) the routing information between the source and sinks. The connection information of the sources and sinks is available without any routing. The actual routing information may be nonexistent, partial, or complete. The routing information is maintained by a few data structures containing RouteEdge objects.

**RouteEdge**

A routing edge in a routing tree graph. It encapsulates all of the information associated with a route between two points: the source TileWireConnection, the sink TileWireConnection, and the WireConnection that maps the two. A RouteEdge does not contain any parent or children data structures - this information is maintained within the Route object.

**B.1.2 Key Classes**

**RoutableDesign**

This class is used to prepare the design for routing.

Key Method:

- prepDesign() - Prepares the design by calling the StaticSourceNetSplitter which splits up the static sources into multiple nets. This method also calls the RouteThroughFinder to determine which connections cannot be used for routing.

**RoutedDesign**

Contains an XDL design and the Route objects that are associated with each Net. This class manages the mapping between Net objects of an XDL design and their corresponding Route objects. This is used throughout the routing process to access the current status of the route.

Key Method:

- RoutedDesign() - Primary constructor. Creates a new, unrouted Route for each Net in the design.
RoutingCost

Provides a cost function for routing resources. The cost is zero if the target and source are the same. If not the same, the cost is a constant times the Manhattan distance plus 1 (all non equal connections have a distance of at least one)

BaseRouter, BreadthFirstSearchRouter, AStarRouter

The BaseRouter sets up the routing infrastructure and and provides a number of helpful methods for building routers including a set of main functions, pre/post processing of netlists and debug statements. The BreadthFirstSearch router extends the BaseRouter and will find the the lowest cost route for each net using a breadth first search. It is not very efficient but is readable and provides a number of methods that can be used by other routing strategies. The AStarRouter extends the BreadthFirstSearchRouter and adds the AStar heuristic to speed routing time.

PathFinderAstarRouter

This router extends AStarRouter to implement the PathFinder algorithm described previously.

Key Methods:

- routeNets() - Most of the PathFinder conflist resolution work is done here. This method sorts the routes to route based on total fanout distance and performs an iteration of calling routeNet() for each route. It then checks for any conflicts between routes. Any resource shared between routes is removed from the route, the cost for the resource is incremented, and the affected routes are added to a list to be routed again. This process iterates until no sharing exists.

- routeNet() - Returns the set of new connections needed to complete the route. This does not include the source connection or any of the sink connections (even if this is a new route). If the net has previously been routed, the previous route connection information is used so only a partial reroute may be necessary. The source and all connections still connected to the source are considered sources. This method calls routeSink() for each sink of the net to perform the actual route.

- routeSink() - Returns the leaf node of the branch added for this sink. The method performs the actual route. It initializes a priority queue and adds all the source nodes based on cost. The method iterates by taking the lowest cost node off the queue. If the node is the sink it is finished. If not, it calls processAdjacentNodes() and adds adjacent nodes to the queue. This process repeats until the sink has been found or it has processed a predetermined limit of nodes.

- processAdjacentNodes() - Process all adjacent children of the given node on the stack. These nodes may have been visited and in this case, update their cost (determines if cost cheaper through this node). If the node has not been visited, then set the parent of the node as this parent, set its cost, and estimate its cost to the end.
**TileWireConnectionSharingHistoryMap**

Represents the sharing and sharing history of TileWireConnection objects. In the pathfinder algorithm, TileWireConnection objects will be allocated by more than one Route as the algorithm proceeds. By the end of the algorithm, TileWireConnection objects will be allocated by only one route.

This class tracks the ownership of TileWireConnection objects by various routes as the algorithm proceeds. In addition, it tracks the history of TileWireConnection objects so that costs may be determined at run-time. The sharing map is cleared after each iteration and the history is kept during the full route.

This object has two maps: one map that provides a floating point value for each connection (its history) and another that provides a Set of Net objects that are using the connection. The history is used to calculate the cost of using a connection.

**B.1.3 Key Steps**

1. Prep design by splitting static source net and finding routethroughs.
2. Initialize RoutedDesign. Create new Route object for each Net.
3. Route nets.
   - (a) Route net.
     - Route all unrouted sinks.
     - Prune route by removing any dangling branches from the previous iteration that are no longer used by the route in this iteration.
   - (b) Find resources shared between routes and update cost.
   - (c) Remove shared resource from routes to prepare for next iteration

**B.1.4 Code**

```java
public RoutedDesign routeNets(RoutedDesign rDesign, List<Route> sortedRoutesToRoute) {

    Comparator<Route> costSorter = new RouteTotalCostComparator();

    // A map between a net and the set of connections that are in conflict with the given net. This is reset at the end of each iteration.
    Map<Route, Set<TileWireConnection>> netConflictConnectionMap = new HashMap<Route, Set<TileWireConnection>>();

    // A map between a net and the number of iterations the net has had a conflict. This map is used to identify high conflict nets.
    Map<Route, Integer> routeSharingIterationMap = new HashMap<Route, Integer>();

    // A map between a net and the new connections added during the iteration
    Map<Route, Set<TileWireConnection>> routeConnectionsUsedInIterationMap = new HashMap<Route, Set<TileWireConnection>>();
```

77
// flag that indicates that there are shared nets in a routing iteration
boolean sharedNets = false;
// iteration counter for debug and accounting purposes
int iteration = 0;
// Net number
int netNum;
// max history
float maxHistory = 0;

// A list of integers that contains the number of resources shared at each
// iteration. This is used to evaluate progress (or lack thereof) of routing.
ArrayList<Integer> sharedResourcesPerIteration = new ArrayList<Integer>();
int minSharedResourcesInIteration = Integer.MAX_VALUE;
int numCyclesNotMakingProgress = 0;
int iterationsOfNotMakingProgressBeforeGivingUp = design.getNets().size();
    //10 * (int) Math.sqrt(design.getNets().size()); // square root of the
    // number of nets?
boolean makingProgress = true;
int firstIterationResourceConflicts = 0;
int previousTilemapSize = 0;

// PathFinder algorithm: iterate through as many iterations as necessary until
// the net is routed.
do {
    totalNodesVisited = 0;
    iteration++;
    if (DEBUG >= DEBUG_GENERAL_INFO) {
        System.out.println("=====================");
        System.out.println("== Iteration #"+iteration);
        System.out.println("=====================");
    }
    //if (iteration > 1)
    //DEBUG = DEBUG_SINK_SUMMARY;
    long iterationStartTime = System.currentTimeMillis();

    // Clear the sharing data structures
    //sharingMap.clearAllNetSharing();
    routeConnectionsUsedInIterationMap.clear();

    netNum = 0;
    totalConnections = 0;
    maxDesignQueue = 0;
    for(Route route : sortedRoutesToRoute){
        Net net = route.getNet();
        netNum++;

        int oldDebug=0;
        if (netNum == NET_TO_QUERY) {

oldDebug = DEBUG;
DEBUG = DEBUG_QUEUE;
}
if (iteration == DEBUG_ITERATION_CHANGE) {
    DEBUG = DEBUG_LEVEL_DURING_ITERATION_CHANGE;
}

// Check to see if the route needs to be routed (some nets don't need any routing). This has nothing to do with whether the net has been routed previously, this is a check to see if the net needs any routing at all.
if (!route.netHas2orMorePinsAndRequiresRouting())
    continue;

if (DEBUG >= DEBUG_NET_SUMMARY) {
    System.out.println("Routing net "+net.getName()+" fanout="+net.getFanOut()+" total dist="+route.getTotalFanoutDistance()+" ("+netNum+" of "+design.getNets().size()+")");
}

// If we are not reusing routes, ignore the route generated in the previous iteration and start over
if (!REUSE_ROUTES)
    route.clearRoute();

// Get the new connections for the route
int connectionsBeforeRoute = route.size();
routeNet(route);
int connectionsDuringRoute = route.size() - connectionsBeforeRoute;
// Prune the unnecessary branches (after routing, there may be dangling branches in the route from a previous iteration that are not used by the route in this iteration. This prune removes all such branches).
route.pruneRoute(true);
// Determine route connections
addRouteConnectionsToSharing(route);

if (DEBUG >= DEBUG_SINK_DETAIL)
    System.out.println(route.toStringNodes());

totalConnections += connectionsDuringRoute;

if (maxNetQueue > maxDesignQueue)
    maxDesignQueue = maxNetQueue;

if (netNum == NET_TO_QUERY) {
    DEBUG = oldDebug;
}
}

// Iterate over all of the sites and see if any are shared.
// If so, update the history table of the site.
sharedNets = false;
int numResourcesShared = 0;
int totalSharing = 0;
float totalHistory = 0;

if (DEBUG >= DEBUG_ITERATION_RESOURCE_SHARING)
    System.out.println("=====================");

// Save a copy of the routes that were recently routed
List<Route> completedRoutes = new ArrayList<Route>(sortedRoutesToRoute);

// Start a new list of routes that need rerouting
List<Route> routesNeedingRerouting = new ArrayList<Route>(sortedRoutesToRoute.size());
Set<TileWireConnection> usedSites = new HashSet<TileWireConnection>(sharingMap.getUsedSites());
netConflictConnectionMap.clear();

// Iterate over all of the used sites in the current routing. Find routing
// conflicts and prepare for the next iteration.
for (TileWireConnection ptwc : usedSites) {
    int sharing = sharingMap.getNumNetsSharing(ptwc);
    if (sharing > 1) {
        numResourcesShared++;
        totalSharing += sharing;

        float connectionHistory = sharingMap.getConnectionHistory(ptwc);
        totalHistory += connectionHistory;
        sharedNets = true;

        // Update history
        sharingMap.incrementConnectionHistory(ptwc, sharing);

        TileWireConnection otherLongLineEnd =
            longLine.otherSourceEndOfLongLine(ptwc, we, tileConnectionMap);
        if (TEMP_LONG_LINE_DEBUG && longLine.isLongLineSourceConnection(ptwc, we)) {
            System.out.format("\tLong Line Source conflict for resource
                "+ptwc.toString(we)+" (%.1f)",
                sharingMap.getConnectionHistory(ptwc));
            if (otherLongLineEnd != null)
                System.out.format("-Other:"+otherLongLineEnd.toString(we)+" (%.1f)%n",
                sharingMap.getConnectionHistory(otherLongLineEnd));
            else
                System.out.println("-Other: non existant");
        }
    }
}
routesNeedingRerouting.add(n);
//netsToReRoute.add(n);
//Route route = netRouteMap.get(n);
n.removeConnection(ptwc);
if (DEBUG >= DEBUG_SINK_DETAIL)
    System.out.println("Removing resource "+ptwc.toString(we)+" from net "+ n.getNet().getName());
// See if the other end of a long line needs to be removed
if (otherLongLineEnd != null) {
    n.removeConnection(otherLongLineEnd);
    //System.out.println("Removing other end "+otherEnd.toString(we)+" from "+ptwc.toString(we));
}
}

// Remove net sharing
sharingMap.clearNetSharing(ptwc);
//if (otherLongLineEnd != null)
//    sharingMap.clearNetSharing(otherLongLineEnd);

//if (DEBUG >= DEBUG_ITERATION_RESOURCE_SHARING) {
for (Route net : netsAtThisConnection) {
    Set<TileWireConnection> netConflictSites =
        netConflictConnectionMap.get(net);
    if (netConflictSites == null) {
        netConflictSites = new HashSet<TileWireConnection>();
        netConflictConnectionMap.put(net, netConflictSites);
    }
    netConflictSites.add(ptwc);
}
}

// At this point, we have a new list of routes needing routing. Sort the list of routes that need to be routed during the next iteration based on their actual cost.
Collections.sort(routesNeedingRerouting, costSorter);
sortedRoutesToRoute = routesNeedingRerouting;

for (Route r : routesNeedingRerouting) {
    Net net = r.getNet();
    Set<TileWireConnection> prunedNodes = r.pruneRoute(false);
    if (DEBUG >= DEBUG_SINK_DETAIL) {
        if (prunedNodes != null && prunedNodes.size() > 0) {
            System.out.println("Pruning from net "+net.getName());
            for (TileWireConnection prune : prunedNodes)
                System.out.println(" "+prune.toString(we));
        }
    }
    System.out.println(r.toString());
}
Integer i = routeSharingIterationMap.get(r);
if (i == null) {
    routeSharingIterationMap.put(r, 1);
} else {
    routeSharingIterationMap.put(r, i+1);
}

if (DEBUG >= DEBUG_ITERATION_RESOURCE_SHARING) {
    List<Route> sortedConflictNets = new ArrayList<Route>(netConflictConnectionMap.keySet());
    Collections.sort(sortedConflictNets, new NetConflictComparator(netConflictConnectionMap));
    for (Route route : sortedConflictNets) {
        Set<TileWireConnection> netConflictSites = netConflictConnectionMap.get(route);
        System.out.println(netConflictSites.size() + " conflicts for net "+route.getName()+" ("+(float)netConflictSites.size() / route.getConnectionsReachableFromSource().size() * 100+
                       "% of "+route.getConnectionsReachableFromSource().size()+")
                        conflict iteration "+routeSharingIterationMap.get(route));
        System.out.print(" ");
        for (TileWireConnection twc : netConflictSites)
            System.out.print(twc.toString(we)+" ");
        System.out.println();
    }
}

// Evaluate the progress of the routing
if (firstIterationResourceConflicts == 0)
    firstIterationResourceConflicts = numResourcesShared;
if (numResourcesShared < minSharedResourcesInIteration) {
    minSharedResourcesInIteration = numResourcesShared;
    numCyclesNotMakingProgress = 0;
} else
    numCyclesNotMakingProgress++;
int previousResources = 0;
if (sharedResourcesPerIteration.size() > 0)
    previousResources = sharedResourcesPerIteration.get(sharedResourcesPerIteration.size()-1);
sharedResourcesPerIteration.add(new Integer(numResourcesShared));

int numIterationsInAverage = 10;
int actualNum = (iteration < numIterationsInAverage ? iteration : numIterationsInAverage);
int recentAvg = 0;
for (int i = 0; i < actualNum; i++) {
    recentAvg += sharedResourcesPerIteration.get(iteration-i-1);
}
recentAvg = recentAvg / actualNum;
int conflictsPerIteration =
    (numResourcesShared-firstIterationResourceConflicts)/iteration;
if (iteration > 1 && (conflictsPerIteration > -1) &&
    numCyclesNotMakingProgress >=
    iterationsOfNotMakingProgressBeforeGivingUp) {
    System.out.println(" No longer making progress - abort" +conflictsPerIteration);
    makingProgress = false;
}

// General iteration debug message
if (DEBUG >= DEBUG_GENERAL_INFO) {
    System.out.println("=====================");
    long iterationEndTime = System.currentTimeMillis();
    long iterationTimeInSeconds = (iterationEndTime-iterationStartTime)/1000;
    long totalTimeInSeconds = (iterationEndTime - routeStartTime)/1000;
    System.out.println("== End Iteration "+iteration+" iteration time =
    "+iterationTimeInSeconds+" sec. Total time="
    +totalTimeInSeconds+" sec");
    System.out.format("== Visited=%d Visit/sec=%d connections=%d
    visits/connection=%d%n",
        totalNodesVisited,
        (totalNodesVisited*1000/(iterationEndTime+1-iterationStartTime)),
        // +1 so we don’t have a divide by zero
        totalConnections,(totalConnections > 0 ?
        (totalNodesVisited/totalConnections) : 0));
    float reroutepercent = ((float) completedRoutes.size()) /
        design.getNets().size() * 100;
    System.out.format("== Reroute nets "+completedRoutes.size()+" (%.1f%%)
        Used sites=%d%n",
        reroutepercent,usedSites.size());
    int tileMapSize = tileConnectionMap.getSize();
    System.out.println("== Max queue in iteration: "+maxDesignQueue+"
        tilemap size="+tileMapSize+" (+"+
        (tileMapSize-previousTilemapSize)+")");
    previousTilemapSize = tileMapSize;
    if (numResourcesShared > 0) {
        float avgHistory = totalHistory/numResourcesShared;
        if (avgHistory > maxHistory)
            maxHistory = avgHistory;
        System.out.format("== Shared="+numResourcesShared+" (%.3f%%)
            History=%.2f "+
            (maxHistory==avgHistory ? "*" : "") +%".2f)%n",
            (float)totalSharing/numResourcesShared,avgHistory,maxHistory);
    } else
        System.out.println("== No resources shared!");
    System.out.println("== delta="+(numResourcesShared-previousResources)+
        //" avg="+recentAvg+ " ("+(numResourcesShared-recentAvg)+")"]"+" min="+minSharedResourcesInIteration+
        "+(numResourcesShared-minSharedResourcesInIteration)+")]+" +
        "+overall="+(numResourcesShared-firstIterationResourceConflicts)+"};
" ("+(numResourcesShared-firstIterationResourceConflicts)/iteration +"/it)"+" no progress="+numCyclesNotMakingProgress+ "+iterationsOfNotMakingProgressBeforeGivingUp+"));
System.out.println("=====================");
}

} while(sharedNets && makingProgress);

// Debug message if the design did not route
if (!makingProgress) {
System.out.println("**** Route could not be found ****");
Map<TileWireConnection, Set<Route>> conflictSites = new
HashMap<TileWireConnection, Set<Route>>();
for (Route net : netConflictConnectionMap.keySet()) {
Set<TileWireConnection> conflictConnections =
netConflictConnectionMap.get(net);
for (TileWireConnection cc : conflictConnections) {
Set<Route> netsWantingThis = conflictSites.get(cc);
if (netsWantingThis == null) {
 netsWantingThis = new HashSet<Route>();
 conflictSites.put(cc,netsWantingThis);
}
 netsWantingThis.add(net);
}
}
for (TileWireConnection conflictconnection : conflictSites.keySet()) {
System.out.println(conflictconnection.toString(we)+" history="+
 sharingMap.getConnectionHistory(conflictconnection)+" nets wanting 
 resource="+ 
 conflictSites.get(conflictconnection).size());
for (Route n : conflictSites.get(conflictconnection))
 System.out.println("\t"+n.getNet().getName());
}

// Clear the incomplete route of the unrouted net (don’t want to send a 
 partially routed route to XDL)
for (Route r : sortedRoutesToRoute)
r.clearRoute();

// print out sharing nets
if (DEBUG >= DEBUG_ITERATIONRESOURCE_SHARING) {
List<Route> reroutedRoutes = new ArrayList<Route>();
for (Route r : reroutedRoutes) {
 System.out.println(routeSharingIterationMap.get(r)+" iterations for "+r.getNet().getName()+" connections="+ 

r.getConnectionsReachableFromSource().size();

// Print routing resource utilization
int totalConnections = 0;
for (Route r : rDesign.getRoutes()) {
    //Route r = netRouteMap.get(net);
    Set<TileWireConnection> connections = r.getConnectionsReachableFromSource();
    if (connections != null)
        totalConnections += r.getConnectionsReachableFromSource().size();
}
System.out.println("Total Connections=\n"+totalConnections+" Connections/Net=\n"+(totalConnections/rDesign.getRoutes().size()));

if (DEBUG >= DEBUG_GENERAL_INFO) {
    long totalTime = System.currentTimeMillis() - routeStartTime;
    System.out.println("Seconds/Iteration=\n"+((float)totalTime/1000/iteration));
}

return rDesign;
}
Appendix C

Additional Routing Approach

This appendix outlines a possible repair routing approach that can be used to route a Shadow placement. This approach could then be adapted to route a Hybrid placement as well. This appendix describes the approach and provides an overview of the algorithm.

C.1 Shadow Routing

Unlike the Cost Repair router described previously, the Shadow Router seeks to take advantage of incremental routing by reusing routing information from one routing iteration to the next. An example is shown in Figure C.1. In this example, the Shadow router routes the original placement (shown left). The Shadow Router generates a repair for primitive Y by moving it to its shadow location. The router then performs a partial reroute by rerouting all nets (shown in red) associated with Y. In this example, net XYZ takes a new path, while net AB remains the same.

![Diagram showing original configuration and shadow configuration](image)

Figure C.1: Image showing the original configuration of main sites and route (left) and repair configuration for primitive Y and corresponding route (right). Only the red route associated with Y is rerouted. The purple route for A and B is reused from the original configuration to the shadow configuration.
The Shadow Router operates on the placement file generated by the Shadow Placer which contains a main site and a shadow site for each circuit primitive in the netlist. The Shadow Router begins by routing the main sites as the baseline router would, and completely ignores all shadow sites (e.g. \{A\}, \{B\}, \{X\}, etc. shown in original configuration of Figure C.1). This generates the original route configuration to be used when there are no permanent faults on the FPGA. A set containing the TWCs used in the initial route is created to track which TWCs are repaired.

**Algorithm 8** Shadow Repair Routing

1: \(PG \leftarrow \text{set of all placement groups}\)
2: \(D \leftarrow \text{set of all possible TWCs}\)
3: Perform initial route on main sites only
4: \(S \leftarrow \text{set of TWCs used in initial route}\)
5: \textbf{for all } \text{pg in } P \textbf{ do}
6: \quad Move pg to shadow site
7: \quad Rip up all nets associated with pg
8: \quad Perform partial reroute on pg's net
9: \quad Resolve routing conflicts
10: \(R \leftarrow \text{set of TWCs in repair route}\)
11: \(G \leftarrow (D - R) \cap S \) (TWCs repaired)
12: \(S \leftarrow S - G\)
13: Restore placement and route to initial configuration
14: \textbf{end for}

The Shadow Router next iterates through and routes all shadow placements. It does this by selecting one placement group and moving the group to its shadow site. All nets associated with that placement group are ripped up and flagged to be rerouted. It is important to note that the routing information from the original route is still available and used. Only nets affected by relocating the placement group need to be rerouted. The router performs a partial reroute by routing the flagged nets. It is possible that reroutes can cause conflicts with the existing routes. The next step is to identify and resolve the conflicts using the PathFinder algorithm described previously. After completing a conflict free route, the router determines if any TWCs were repaired by the route. Finally, the placement group is returned to its original location in preparation for the next repair route.