Using Hard Macros to Accelerate FPGA Compilation for Xilinx FPGAs

Christopher Michael Lavin
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/2933

This Dissertation 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.
Using Hard Macros to Accelerate FPGA Compilation for Xilinx FPGAs

Christopher Michael Lavin

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

Doctor of Philosophy

Brent E. Nelson, Chair
Brad L. Hutchings
David A. Penry
Michael D. Rice
Michael J. Wirthlin

Department of Electrical and Computer Engineering
Brigham Young University
April 2012

Copyright © 2012 Christopher Michael Lavin
All Rights Reserved
ABSTRACT

Using Hard Macros to Accelerate FPGA Compilation for Xilinx FPGAs

Christopher Michael Lavin
Department of Electrical and Computer Engineering, BYU
Doctor of Philosophy

Field programmable gate arrays (FPGAs) offer an attractive compute platform because of their highly parallel and customizable nature in addition to the potential of being reconfigurable to any almost any desired circuit. However, compilation time (the time it takes to convert user design input into a functional implementation on the FPGA) has been a growing problem and is stifling designer productivity.

This dissertation presents a new approach to FPGA compilation that more closely follows the software compilation model than that of the application specific integrated circuit (ASIC). Instead of re-compiling every module in the design for each invocation of the compilation flow, the use of pre-compiled modules that can be “linked” in the final stage of compilation are used. These pre-compiled modules are called hard macros and contain the necessary physical information to ultimately implement a module or building block of a design. By assembling hard macros together, a complete and fully functional implementation can be created within seconds.

This dissertation describes the process of creating a rapid compilation flow based on hard macros for Xilinx FPGAs. First, RapidSmith, an open source framework that enabled the creation of custom CAD tools for this work is presented. Second, HMFlow, the hard macro-based rapid compilation flow is described and presented as tuned to compile Xilinx FPGA designs as fast as possible. Finally, several modifications to HMFlow are made such that it produces circuits with clock rates that run at more than 75% of Xilinx-produced implementations while compiling more than 30× faster than the Xilinx tools.

Keywords: FPGA, rapid prototyping, design flow, hard macros, Xilinx, XDL, RapidSmith, HMFlow, open source, placer, router
ACKNOWLEDGMENTS

I would first like to thank my wife Ashley and daughter Katelyn for their patience and support in allowing me to finish this work. Several long days, nights and Saturdays were necessary for this dissertation to meet completion and their long suffering and love provided me a great motivation to finish. I am also grateful for my parents and their love and support by providing me the great start in life to allow me to reach this achievement.

I would like to thank Dr. Brent Nelson for taking me under his wing five and a half years ago. I am grateful for the patience he had to allow me to find out on my own what I should research for this dissertation. The extra time Dr. Nelson made for me to discuss my ideas or challenges and his example helped shape my talents and skills to help me become the engineer I am today.

I would also like to thank Dr. Brad Hutchings for his significant contributions to this work and my development as an engineer. Dr. Hutchings went out of his way to provide time and support as an unofficial co-advisor to this work. His insights were invaluable and added significantly to this dissertation and helped me grow as a graduate student.

Thanks to Dr. Michael Rice for the opportunity to work with him and the Telemetry Lab on the Space-Time Coding project. That opportunity turned out to be a rich experience that laid the groundwork for several of the other accomplishments I have made as a graduate student.

Thanks also to Dr. Michael Wirthlin for all the support and time he took to help me in various endeavors. Thanks to Dr. David Penry and the entire committee for their valuable insight on this dissertation.

There were also several students whose example and work helped me significantly as a graduate student that I would like to thank: Joseph Palmer, Nathan Rollins, Jon-Paul Anderson, Brian Pratt, Marc Padilla, Jaren Lamprecht, Philip Lundrigan, Subhra Ghosh,
Brad White, Jonathon Taylor, Josh Monson and all the other students in the Configurable Computing Lab.

   Special thanks also to Neil Steiner and Matt French at USC-ISI East, Dr. Peter Athanas and the Virginia Tech Configurable Computing Lab as well as the entire Gremlin project for their insight and ideas that helped me more fully understand FPGAs and their architecture.

   This research was supported by the I/UCRC Program of the National Science Foundation under Grant No. 0801876 through the NSF Center for High-Performance Reconfigurable Computing (CHREC).
# TABLE OF CONTENTS

**LIST OF TABLES** ......................................................... ix

**LIST OF FIGURES** ..................................................... xi

Chapter 1  Introduction ................................................ 1
  1.1 Motivation ......................................................... 1
  1.2 Preview of Approach .............................................. 3
  1.3 Contributions of this Work ..................................... 3

Chapter 2  Background and Related Work .............................. 5
  2.1 FPGA Architecture ............................................... 5
    2.1.1 FPGA Primitives ............................................ 5
    2.1.2 Chip Layout and Routing Interconnect ....................... 8
  2.2 Conventional FPGA Compilation Flow ........................... 10
  2.3 Related Work in Accelerating FPGA Compilation ............... 12
    2.3.1 Using Pre-compiled Cores to Accelerate FPGA Compilation . 12
    2.3.2 Accelerating Placement Techniques ........................ 14
    2.3.3 Routability-driven Routing ................................ 16
    2.3.4 Summary and Overview of this Work ........................ 16

Chapter 3  RapidSmith: An Open Source Platform for Creating FPGA
          CAD Tools for Xilinx FPGAs ..................................... 19
  3.1 Introduction ..................................................... 19
  3.2 Related Work ................................................... 20
    3.2.1 Torc ......................................................... 21
  3.3 XDL: The Xilinx Design Language ............................... 21
    3.3.1 Detailed FPGA Descriptions in XDLRC Reports ............... 22
    3.3.2 Designs in XDL .............................................. 24
  3.4 RapidSmith: A Framework to Leverage XDL and Provide a Platform to Create FPGA CAD Tools ............................................. 26
    3.4.1 Xilinx FPGA Database Files in RapidSmith ................... 27
    3.4.2 Augmented XDLRC Information in RapidSmith ................. 29
    3.4.3 XDL Design Representation in RapidSmith ................... 30
    3.4.4 Impact of RapidSmith ....................................... 30

Chapter 4  HMFlow 2010: Accelerating FPGA Compilation with Hard
          Macros for Rapid Prototyping .................................... 33
  4.1 Preliminary Work ................................................ 33
    4.1.1 Selection of a Compiled Circuit Representation ............. 34
    4.1.2 Experiments Validating Hard Macro Potential ............... 35
    4.1.3 Hard Macros and Quality of Results ........................ 42
    4.1.4 Hard Macros and Placement Time ............................ 42
<table>
<thead>
<tr>
<th>Chapter 5</th>
<th>HMFlow 2011: Accelerating FPGA Compilation and Maintaining High Performance Implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Preliminary Work</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Experiments</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Conclusions on Preliminary Work</td>
</tr>
<tr>
<td>5.2</td>
<td>Comparison of HMFlow Using Large Hard Macros vs. Small Hard Macros</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Upgrading HMFlow to Support Large Hard Macros</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Upgrading HMFlow to Support Virtex 5 FPGAs</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Large Hard Macro Benchmark Designs for HMFlow</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Comparisons of Large and Small Hard Macro-based Designs</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Comparisons of Large Hard Macros with HMFlow vs. Xilinx</td>
</tr>
<tr>
<td>5.3</td>
<td>Modification to HMFlow for High Quality Implementations</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Hard Macro Simulated Annealing Placer</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Register Re-placement</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Router Improvements</td>
</tr>
<tr>
<td>5.4</td>
<td>Performance Analysis</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Result Measurement Fairness</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Results of Three HMFlow 2011 Improvements</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Results of Optimizing HMFlow 2011 Improvements</td>
</tr>
<tr>
<td>5.5</td>
<td>Techniques for Reducing Variance</td>
</tr>
<tr>
<td>5.5.1</td>
<td>T1: Move Acceptance a Function of Hard Macro Port Count</td>
</tr>
<tr>
<td>5.5.2</td>
<td>T2: Small Hard Macro Re-placement</td>
</tr>
<tr>
<td>5.5.3</td>
<td>T3: Cost Function Includes Longest Wire</td>
</tr>
<tr>
<td>5.5.4</td>
<td>Configuration Comparison of Techniques</td>
</tr>
<tr>
<td>5.6</td>
<td>Runtime Results</td>
</tr>
<tr>
<td>5.7</td>
<td>Conclusions</td>
</tr>
</tbody>
</table>

**Chapter 6 The Big Picture: The Compilation Time vs. Circuit Quality Tradeoff**
6.1 Motivation ............................................................... 117
6.2 Implications for FPGA Designers ................................. 120
6.3 Contributions .......................................................... 120
6.4 Conclusions ............................................................ 121
6.5 Future Work ........................................................... 122

REFERENCES ................................................................... 123
# LIST OF TABLES

2.1 Virtex 4 Routing Interconnect Types ........................................ 9
2.2 Virtex 5 Routing Interconnect Types ........................................ 10

3.1 RapidSmith Device Files Performance ........................................ 29

4.1 Baseline Runtimes for each Test Design .................................... 41
4.2 Performance of each Test Design Using Hard Macros ...................... 41
4.3 Comparison of Baseline vs. Hard Macro Designs ............................ 43
4.4 Benchmark Design Characteristics ........................................... 56
4.5 Fine-grained Hard Macro Compile Times .................................... 59
4.6 Router Performance Comparison: Xilinx vs. RapidSmith ................... 60
4.7 Hard Macro Placer Algorithm Comparison ................................... 60
4.8 Runtime Performance of HMFlow and Comparison to Xilinx Flow ........ 62

5.1 Width/Height Area Group Aspect Ratio Configurations .................... 74
5.2 Slice Counts for Large Hard Macro Benchmark Virtex 5 Designs ........ 80
5.3 Coarse-grained Hard Macro Compile Times .................................. 83
5.4 Runtime Comparison of HMFlow with Large Hard Macros vs. Xilinx ....... 83
5.5 Clock Rate Comparison of HMFlow with Large Hard Macros vs. Xilinx ... 84
5.6 All HMFlow 2011 Improvement Configurations Tested ....................... 99
5.7 HMFlow 2011 Benchmark Clock Rates of Single (Default) Run (in MHz) .. 99
5.8 HMFlow 2011 Benchmark Clock Rates of Average of 100 Runs (in MHz) .. 100
5.9 HMFlow 2011 Benchmark Clock Rates of Best of 100 Runs (in MHz) .... 100
5.10 Variance of HMFlow 2011 (C7) and Xilinx in 100 Compilation Runs .... 102
5.11 Average Variance and Frequency using Variance-reducing Techniques ... 112
5.12 Compilation Runtime for Several HMFlow 2011 Configurations vs. Xilinx .. 114
5.13 Clock Rate Summary (in MHz) for HMFlow 2011 Configurations vs. Xilinx . 114
LIST OF FIGURES

2.1 General Logic Abstractions in a Xilinx Virtex 5 FPGA ................. 6
2.2 Common Xilinx FPGA (Virtex 5) Architecture Layout ................. 8
2.3 Conventional FPGA Compilation Flow (Xilinx) ....................... 11

3.1 RapidSmith and XDL Interacting at Different Points within the Xilinx Tool Flow .......................................................... 25
3.2 RapidSmith Abstractions for (a) Devices and (b) Designs ............. 27
3.3 Screenshots of Graphical Tools Provided with RapidSmith to Browse (a) Devices or (b) Designs ........................................... 31

4.1 Hard Macro Creation Flow .................................................. 36
4.2 Compilation Flow for Experiment #1 (Conventional Xilinx Flow) .... 38
4.3 Compilation Flow for Experiment #2 (Modified Xilinx Flow) ...... 40
4.4 Block Diagram of Multiplier Tree Design .............................. 40
4.5 Block Diagram of HMFlow .................................................. 44
4.6 Screenshot of an Example System Generator Design .................. 45
4.7 Front-end Flow for an HMFlow Hard Macro ........................... 47
4.8 (a) A FIR filter Design Compiled with the Xilinx Tools (b) The Same Filter Design Compiled with an Area Constraint to Create a Hard Macro ................. 48
4.9 A Histogram of Hard Macro Sizes of All Hard Macros in the Benchmarks ................................................................. 57
4.10 Graph Showing Percentage of Routed Connections in Each Benchmark Before the Design is Sent to the Router ........................................... 57
4.11 (a) Average Runtime Distribution of HMFlow (b) HMFlow Runtime as a Percentage of Total Time to Run HMFlow and Create an NCD File ................. 64
4.12 A Comparison Plot of the Benchmark Circuits Maximum Clock Rates When Implemented with HMFlow and the Xilinx Tools ....................... 65

5.1 General Pattern of Hard Macro Placement on FPGA Fabric .......... 70
5.2 Delay of a Path Within a 21\times21 Bit LUT-multiplier Hard Macro Placed in a Grid of 400 Locations on a Virtex 4 SX35 FPGA ......................... 71
5.3 A More Severely Impacted Path Caused by a Hard Macro Straddling the Center Clock Tree Spine of the FPGA ........................................... 72
5.4 A PicoBlaze Hard Macro Placed at 3700 Locations on a Virtex 5 FPGA .... 73
5.5 (a) Illustrates a FIR Filter Implemented in System Generator (b) Shows the FIR Filter Converted to a Subsystem to be Turned into a Hard Macro ................. 77
5.6 (a) Comparison of Runtime for Large and Small Hard Macro Versions of 3 Benchmarks on HMFlow 2010a (b) Comparison of Clock Rate for Large and Small Hard Macro Versions of 3 Benchmarks on HMFlow 2010a ....................... 81
5.7 The Number of Existing Routed Connections in the Large Hard Macro Benchmark as a Percentage of Total Connections ....................... 82
5.8 Block Diagrams of HMFlow 2010a and HMFlow 2011 .................. 85
5.9  (a) Representation of a Set of Hard Macros Drawn with a Tight Bounding Box (b) An Approximated Bounding Box for the Same Hard Macros for Accelerating the Simulated Annealing Hard Macro Placer  

5.10  An Illustration of the Problem of an Approximating Bounding Box Where the Box Changes Size Based on Location  

5.11  A Simple Example of a Register (A) Being Re-placed at the Centroid of the Original Site of Register A and Sinks B, C and D  

5.12  Representation of Virtex 4 and Virtex 5 Long Line Routing Resources  

5.13  Plot of Benchmark Brik1 Uphill Accepted Moves for Each Hard Macro  

5.14  T1 Variance of Scaling Factor Q Parameter Sweep  

5.15  T1 Variance of Scaling Factor Q Parameter Sweep (Zoomed)  

5.16  T2 Variance of Hard Macro Size Re-placement Parameter Sweep  

5.17  T2 Variance of Hard Macro Size Re-placement Parameter Sweep (Zoomed)  

5.18  T3 Variance of Longest Wire Scaling Factor Parameter Sweep  

5.19  T3 Variance of Longest Wire Scaling Factor Parameter Sweep (Zoomed)  

5.20  Percentage of Total Wire Length to Overall Cost Function Output on Brik1 Benchmark Using T3  

6.1  Generalized Representation of Current Tool Solution Offerings for Compilation Runtime vs. Quality of Result  

6.2  Quality vs. Runtime Tradeoff for HMFlow 2011 and Xilinx Tools
CHAPTER 1. INTRODUCTION

1.1 Motivation

Field programmable gate arrays (FPGAs) have steadily gained traction as a computational platform for several years. With each fabrication process shrink, FPGAs have been able to steadily grow in capacity from simple chips used as glue logic to massively parallel processing platforms capable of hundreds of GFLOPS (billions of single-precision floating point operations per second). They have also expanded to include a variety of dedicated cores such as block memories, processors, Ethernet MACs, PCI Express cores and multi-gigabit transceivers (MGTs) to provide more robust solutions to a greater number of design projects. As FPGA compute capabilities have increased, they have become a popular choice to replace other computational platforms such as application specific integrated circuits (ASICs) and general purpose processors (CPUs).

FPGAs are becoming a more attractive alternative to ASICs in many applications for two main reasons. First, FPGAs are re-programmable in that their circuits can essentially be changed as many times as the user desires. This feature alone helps drive down costs in development projects as changing an ASIC after photolithography masks have been created is difficult and often will require a new set of the very expensive masks to be created in order to change the circuit behavior.

Second, FPGAs have little to no non-recurring engineering (NRE) costs when compared with the costs of producing leading-edge technology ASICs. An FPGA can be purchased, designed and programed to be fully functional with almost no up front cost. ASICs, however, will require a lengthy development, verification and expensive fabrication process that can cost several million dollars before a single functional die is produced. In addition, with each fabrication technology process shrink, ASIC NRE costs rise significantly—driving more projects and engineers to choose FPGAs over ASICs. It is expected that FPGAs will
continue to consume the ASIC project market for all but the highest volume production runs.

FPGAs are also an attractive alternative to CPUs in embedded applications as FPGAs can offer much higher performance and also better performance per Watt. However, one aspect of FPGAs that has continually plagued their adoption for many typical CPU applications has been lengthy compilation times. Software compilation for CPUs is often completed within seconds whereas FPGA compilation can take hours or even days. Such a large disparity in compile time puts FPGA designers at a significant design disadvantage by limiting the number of edit-compile-debug turns that can be completed per day. Poor designer productivity due to lengthy FPGA development times can often negate the performance savings offered by FPGAs. This drawback in productivity can also drive engineers to use CPUs as design solutions when computational power is not critical.

Long compilation time has been a major roadblock for widespread FPGA adoption and is mostly due to the driving force behind the CAD tools used to compile their designs. Those projects that do leverage FPGAs (mostly as ASIC alternatives) are concerned with two main metrics: area and clock rate. Area and clock rate directly correlate with the cost of the project and FPGA vendors must optimize their CAD tools for these two metrics above all else to ensure viability in the FPGA market arena. In addition, since many engineers leveraging FPGAs are already familiar and accustomed to the lengthy development cycle presented by ASICs, the use of the similar FPGA development and compilation process has not produced much motivation or effort to reduce compilation times.

However, as FPGAs have increased in size and capacity, their usage in new kinds of applications that do not demand the highest clock rate or minimal area utilization have become more prevalent. One such example is National Instruments LabVIEW FPGA design tool which only targets an implementation clock rate of 40 MHz, substantially lower than modern FPGA capabilities. The largest complaint of the LabVIEW FPGA product from its consumers has been the compilation time of their designs [1]. With the past few decades of FPGA CAD being narrowly focused on optimizing for area and clock rate, no effective commercial solutions are available to solve the problem of lengthy FPGA design compilation times.
1.2 Preview of Approach

The goal of this work is to create and explore FPGA CAD tool techniques that can provide acceptable FPGA implementation results while reducing compilation time for FPGAs by an order of magnitude or more. More broadly, the goal is to enable FPGA design engineers greater flexibility in trading circuit quality for faster compilation time and further shed light on the largely unexplored tradeoff.

The particular approach of this work is to emulate the largely successful software compilation model. In software, lengthy compilations are avoided by using pre-compiled libraries which are linked in at the final stage of executable generation. Conventional FPGA compilation is in stark contrast to this model in that each time a design is compiled, each and every component is re-compiled from scratch regardless of whether it has been changed since the previous compilation or not.

To avoid excessive re-compilation, this work heavily utilizes a construct in hardware that largely parallels the pre-compiled libraries in software. This hardware construct is called a hard macro\(^1\). A hard macro is a pre-synthesized, pre-mapped, pre-placed and pre-routed hardware block that has been designed with re-use in mind and can potentially be instantiated at several different locations on the FPGA fabric. As hard macros are device family specific, they are tied to a particular architecture, however, this is similarly the case for pre-compiled libraries in software. By creating designs purely assembled from hard macros, compilation time can be reduced significantly as shown by the techniques outlined in this dissertation.

1.3 Contributions of this Work

Several attempts have been made to accelerate FPGA compilation, often focusing on certain steps of the compilation process [2] [3], with some techniques even using hard macros [4]. However, this work is unique in that it combines the approach of both algorithmic improvement and leveraging pre-compiled information to accelerate the compilation flow\(^1\).

---

\(^1\)It should be noted that the term hard macro referred to here and throughout this dissertation is strictly applied to circuits realized in programmable FPGA fabric. It does not refer to hard macros in an ASIC.
used by FPGAs. Additionally, this approach is flexible enough to accommodate designs from all application domains.

The main contributions of this dissertation are listed below:

- Though some research efforts to perform CAD tool experimentation on commercial FPGAs has been performed in the past (such as [5], [4], [6] and [7]), the task has been quite difficult due to a lack of a unified framework and tools. In order to implement the several algorithms, ideas and techniques of this work, RapidSmith, a framework for FPGA CAD tool creation was developed and released as open source to the research community (Chapter 3).

- HMFlow 2010 - Demonstration of a complete, custom FPGA compilation flow leveraging hard macros for rapid prototyping purposes. (Chapter 4).

- HMFlow 2011 - Demonstration of several techniques to improve on HMFlow 2010 that allow for higher quality implementation with minimal runtime increases (Chapter 5).

- Provide additional insight to the tradeoff between compilation time and circuit quality using the several techniques in HMFlow 2010 and HMFlow 2011 (Chapter 6).

The use of hard macros to directly accelerate commercial FPGA compilation of general purpose designs is a little studied area of research. In addition, providing insight into the potentially valuable tradeoff of compile time vs. circuit quality has also received little attention in the research literature. An overview of FPGA compilation and related work will be outlined in Chapter 2 with Chapters 3, 4 and 5 describing in detail the accomplishments of this dissertation. In Chapter 6 I conclude and reflect on the potential impact of the contributions made in this work.
CHAPTER 2. BACKGROUND AND RELATED WORK

This chapter will describe the basics of FPGA architecture, the conventional FPGA compilation flow and related work in accelerating FPGA compilation.

2.1 FPGA Architecture

In order to implement a wide variety of circuits, FPGAs use configurable logic and routing interconnect to provide a rich framework on which to realize a desired circuit. The logic components of an FPGA contain facilities to perform computations such as arithmetic, memory storage, I/O communications and other processing. The routing interconnect is a massive network of programmable wires that allow connections to be made in between logic elements. This section aims to provide the reader with a rudimentary understanding of FPGA architecture (specifically Xilinx FPGAs) to better understand the contributions found later in this dissertation.

2.1.1 FPGA Primitives

FPGAs have a set of primitives that perform either logic, arithmetic, storage or I/O communications. Each family of Xilinx FPGAs share the same set of logic primitives and are characterized by a set of input and output pins with zero or more configurable options. The most popular primitives are slices, IOBs, block RAMs and multipliers or DSPs.

Slice

At the heart of configurable logic is the programmable look up table (LUT). LUTs can be programmed to implement any Boolean function of complexity only limited by the number of inputs. Modern FPGAs have LUT input sizes of between 3 and 6 inputs and correspondingly have between 8 and 64 ($2^3$ and $2^6$) bits of configuration (memory cells).
LUTs are organized hierarchically into different units of abstraction to aid the CAD tools in mapping and implementing a design onto an FPGA. For example, as shown in Figure 2.1, Virtex 5 Xilinx FPGAs use the terms slice and configurable logic block (CLB) to denote groups of LUTs and groups of slices respectively. LUTs are also accompanied by a D flip-flop, often to store the results of the LUT output and provide a method for maintaining state in the FPGA. A slice will also contain other useful elements such as multiplexers, carry chains (used for faster addition and subtraction), and some kinds of LUTs can be configured as small RAMs.

**Input Output Buffer (IOB)**

In order to interface an FPGA design with off-chip signals, input/output buffers (IOBs) are needed to provide the necessary circuitry and signaling. IOBs can be configured...
to interface with a number of different voltage standards and can be configured as input, output or tri-state buffers.

**Block RAMs**

When FPGAs were first created they consisted of a purely homogeneous array of CLBs and slices. However, as FPGAs grew in capacity and capability, FPGA vendors saw the utility in adding block memories on chip to aid in computation. Xilinx added block RAMs to its FPGAs. The capacities and quantity of block RAMs differ from one device to the next. As one example, Virtex 5 block RAMs contains 36 kilobits of memory and different Virtex 5 parts can have a little as a few dozen to as many as several hundred of these memories on a single chip.

**Multipliers and DSP**

In recent years, FPGAs have been quite successful at accelerating computations in digital signal processing. Since a multiplier implemented out of several LUTs consumes a significant portion of FPGA resources, FPGA vendors began to introduce dedicated multipliers into the FPGA fabric. These multipliers were improved in later families to become a more fully-featured block called a DSP which is capable of several operations, the most popular of which is a multiply-accumulate.

**Other Primitives**

Several other primitive cores exist on an FPGA for a variety of different applications. Xilinx includes a primitive called the Internal Reconfiguration Access Port (ICAP) that allows the FPGA to access its own configuration circuitry. Ethernet MACs and PCI Express cores have also become popular primitives to allow FPGAs to more easily communicate with common I/O standards without consuming large portions of FPGA resources. Xilinx has also included hard processors within the FPGA fabric. The Virtex II, Virtex 4 and Virtex 5 families contained parts which had one or two PowerPC processors on chip to enable software to interact with the FPGA fabric and build system-on-chip type systems.
2.1.2 Chip Layout and Routing Interconnect

One of the major challenges of FPGA architecture is not only providing programmable logic functionality but doing so in a way that allows actual implementation of real circuits. If logic in different parts of the chip cannot communicate, the FPGA is worthless. Therefore, the layout of the chip and its routing interconnect architecture are important and must be accomplished in an intelligent fashion.

Virtually all modern FPGAs follow a layout style called an island-style architecture. Island-style FPGAs consist of creating a regular array of islands of logic with a sea of interconnect resources in between them. Generally, each island of logic is accompanied by a switch box or switch matrix which acts as the entry point to access the vast sea of interconnect resources. A high level representation of a typical Xilinx FPGA layout is given in Figure 2.2.
The particular layout in Figure 2.2 most closely resembles the layout of the Virtex 5 family. The black boxes represent logic and the white boxes are switch boxes. All logic blocks access the main routing interconnect through a switch box to its left. Each CLB is paired with one switch box, whereas larger logic blocks such as DSPs and block RAMs are paired with 2.5 and 5 switch boxes respectively. Xilinx FPGAs typically arrange logic so that each column of logic blocks are of the same type. If the example in Figure 2.2 were extended, it would be seen that repeating CLBs, block RAMs, and DSPs would be above and below those shown until the top and bottom of the chip were reached.

It should also be noted that logic blocks such as CLBs and DSPs have direct interconnect to the adjacent blocks above and below. These direct connections are carry chains which allow for faster arithmetic computation by chaining the critical computations closer together.

The routing found in Figure 2.2 is simplified for ease of understanding. The actual routing interconnect found in the Virtex 5 is more complex with several different types of routing interconnect. Most general purpose routing starts and terminates in the switch box and Xilinx provides a variety of types of routing interconnect to provide a robust interconnection routing structure. The most common routing resources provided in the Virtex 4 and Virtex 5 architectures are listed in Tables 2.1 and 2.2.

Table 2.1: Virtex 4 Routing Interconnect Types

<table>
<thead>
<tr>
<th>Virtex 4 Routing Resource</th>
<th>Switch Box Hops</th>
<th>XDL Name REGEX Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMUX</td>
<td>1, 2</td>
<td>^OMUX[0-9]+</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>1, 2</td>
<td>^[NSEW]2BEG[0-9]+</td>
</tr>
<tr>
<td>HEX</td>
<td>3, 6</td>
<td>^[NSEW]6BEG[0-9]+</td>
</tr>
<tr>
<td>LONG LINE</td>
<td>6, 12, 18, 24</td>
<td>^L[HV][(0)][(6)][(12)][(18)][(24)]</td>
</tr>
</tbody>
</table>

Some wire names start with one of four letters, N, S, E, or W (North, South, East or West) that indicate the direction of the wire. For example, a Virtex 4 DOUBLE called N2BEG0 is a wire that connects to the first and second switch boxes directly above the source switch box. Note that the DOUBLE TURN and PENT TURN routing resources in Virtex 5 change direction after 1 and 3 switch box hops, respectively. Long lines are
Table 2.2: Virtex 5 Routing Interconnect Types

<table>
<thead>
<tr>
<th>Virtex 5 Routing Resource</th>
<th>Switch Box Hops</th>
<th>XDL Regular Expression Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE</td>
<td>1, 2</td>
<td>^[NSEW][LR]2BEG[0-9]+$</td>
</tr>
<tr>
<td>DOUBLE (TURN)</td>
<td>1, 2</td>
<td>^[NSEW][NSEW]2BEG[0-9]+$</td>
</tr>
<tr>
<td>PENT</td>
<td>3, 5</td>
<td>^[NSEW][LR]2BEG[0-9]+$</td>
</tr>
<tr>
<td>PENT (TURN)</td>
<td>3, 5</td>
<td>^[NSEW][NSEW]2BEG[0-9]+$</td>
</tr>
<tr>
<td>LONG LINE</td>
<td>6, 12, 18</td>
<td>^L[HV][(0)</td>
</tr>
</tbody>
</table>

bi-directional so they only use two letters, H and V to denote horizontal and vertical wires. Also note that the regular expressions in Tables 2.1 and 2.2 do not cover all wire names in XDL, simply some of the more common ones.

By providing a variety of wire interconnect lengths, Xilinx FPGAs are able to accommodate routing demands much better than if a single standard length was used. A variety of wire lengths tailors a route’s needs to the routing resource and minimizes routing interconnect delay by using resources that meet the distance requirements of the connection. Long lines are the obvious choice for long distance connections as they are much faster than chaining several shorter lengths of wire together. Conversely, DOUBLE wires are efficient ways of connecting 1 and 2 switch box hops away.

2.2 Conventional FPGA Compilation Flow

A diagram showing the steps involved in the compilation of a typical Xilinx FPGA design is given in Figure 2.3. FPGA designers will often describe the circuit to be implemented in a register transfer level (RTL) based language such as VHDL or Verilog. These files are parsed and then synthesized into a netlist which describes a list of circuit elements such as gates and flip flops with an accompanied list of their interconnections. The Xilinx RTL synthesizer is called xst and outputs a Xilinx netlist called an NGC file. The synthesizer will often perform logic optimization and infer common circuit constructs from the RTL provided. That is, it will find ways in which to reduce the circuit size by exploring more efficient implementations of the logic and also match RTL patterns for common constructs used in FPGAs such as hard multipliers and block RAMs.
The netlist produced by the synthesizer is then technology mapped into the specific components offered by FPGAs such as LUTs, registers, carry chains and so forth. The Xilinx tool which performs technology mapping is called ngdbuild and creates a technology mapped netlist called an NGD file.

After the netlist has been technology mapped, it must be packed into the FPGA primitives such as slices and DSP48s. The packing process is performed by the Xilinx tool called map. The output of map is a Xilinx-specific netlist that contains a list of FPGA primitives and nets describing their interconnections. The proprietary netlist format that Xilinx uses is called the NCD (Native Circuit Description) file. As will be discussed in the next chapter, Xilinx also offers an open ASCII-based version of the NCD file called XDL.

The final steps in implementation are the placement and routing steps. Placement involves assigning FPGA primitives to a specific location on the FPGA fabric on which to reside. The routing phase then decides which programmable switches are turned on and off to create the desired connections between primitives. Placement and routing are often performed by Xilinx par, however, newer architectures such as Virtex 5 and later have
their placement performed in the map process. NCD files are also produced by par as the NCD/XDL format can represent designs that are unplaced, placed and routed.

The final step for FPGA implementation is generating a bitstream or BIT file. A bitstream is a large binary file where its bits correspond to resources or memories in an FPGA. Generating a bitstream involves changing bit values in the BIT file to reflect the desired behavior of the FPGA as specified by the primitive netlist. Xilinx uses a program called bitgen to convert physical netlist files (NCD) to BIT files. After a BIT file is created, it can be downloaded directly to the chip or programmed into a PROM to be loaded into the FPGA on reset or power up.

2.3 Related Work in Accelerating FPGA Compilation

Two major approaches to accelerating FPGA compilation are: (1) make some attempt to preserve intermediate implementation data such as pre-compiled cores and (2), modify or develop new algorithms that will accomplish tasks such as placement and routing faster. As these are the two major thrusts of this work, this section will discuss other relevant work in those respective research areas.

2.3.1 Using Pre-compiled Cores to Accelerate FPGA Compilation

As described in the introduction of this dissertation, software compilation speeds benefit significantly from using pre-compiled libraries to store commonly used routines that do not need to be recompiled for each compilation. By linking these libraries into the final executable, compilation time is reduced significantly. However, the traditional approach of FPGA compilation has often been to re-compile everything for each invocation of the compilation flow. This is often wasteful as many of the design files do not change from run to run. There have been several attempts in the past to leverage this fact to try and accelerate FPGA compilation. The approach taken in this work is to use pre-compiled blocks called hard macros, however, other techniques such as bitstream cores, macroblocks, virtual fabrics and a tool called ReCoBus also demonstrate ways in which intermediate design information can be reused to reduce compilation time.
Bitstream Cores

As the bitstream is the final implementation format of the design, it presents an attractive opportunity to attempt to assemble a design from cores stored as portions of a bitstream. Horta and Lockwood [8] demonstrated the creation of bitstream-based relocatable cores that are quite similar in nature to hard macros. Similar efforts are reported in [9] where bitstream hard cores were used in a network-on-chip to provide accelerated logic emulation and prototyping. Unfortunately, bitstream hard cores are less flexible than hard macros because they must reside between restrictive configuration boundaries and require matching bus macro interfaces to be present in both the cores as well as the existing FPGA configuration. Bitstream-based cores are also more difficult to construct as much of the bitstream formats used by FPGA vendors are proprietary.

ReCoBus

ReCoBus [4] is a tool chain that attempts to generate complete systems on-the-fly. It leverages partially reconfigurable modules that can be loaded into an FPGA at runtime. By reusing modules that can be swapped in and out of the FPGA, it can avoid lengthy compilation time when attempting to reconfigure the overall system. To accomplish this task, a system-level ReCoBus hard macro is generated based on specifications provided by the user. Each core requires an interface template to be wrapped around its interface to provide compatibility with the overall ReCoBus system. These cores are then converted to special partial bitstreams that will allow the cores to be swapped into and out of the FPGA at runtime.

The ReCoBus system does provide a way to reduce compilation time, however each time the functionality of a core must be changed, it must be completely re-compiled. The bus-based nature of the approach limits its flexibility. However, leveraging a bitstream linker, ReCoBus can rapidly reconfigure systems with cores that have already been compiled.
Intermediate Virtual Fabrics

Intermediate virtual fabrics are another strategy that exploits reuse of pre-compiled structures [10]. This technique abstracts the FPGA to a domain-specific fabric created off-line and customized for a particular application domain. These fabrics accommodate macroblocks which are placed and routed quickly onto the fabric. This technique is effective if the intermediate fabric has already been built for a particular application, Coole et al. claim an average place and route speedup of 554×. However, the speed comes at a cost. Each newly created intermediate fabric still requires a lengthy place and route and the area overhead of building an intermediate fabric can consume up to 40% of the FPGA’s resources.

2.3.2 Accelerating Placement Techniques

Placement of FPGA primitives can largely determine the overall quality of the resulting implementation and is therefore a critical step in producing high quality of result circuits. However, to obtain high quality results, placement (as with its counterpart process routing) has been well known for long runtimes. Some work has been done in exploring different techniques to alleviate the significant time requirement by creating macroblocks from design elements or clustering blocks together and modifying the optimization algorithms involved.

Macroblocks

One way to reduce the placement problem size is to reduce the number of objects that need to be placed. Tessier [5] demonstrated Frontier, a placement tool that utilized macroblocks and floorplanning to accelerate placement. The macroblocks used by Frontier were similar to hard macros in that some (not all) contained relative placement information and accordingly could be placed as a complete unit directly onto the FPGA fabric. These macroblocks were compiled before the Frontier placement process and are different from the hard macros used in the approach of this dissertation for two reasons. First, not all macroblocks contained relative placement information and thus, the inner primitives were allowed to float and be manipulated by the placer. This could ultimately reduce the overall
effectiveness of a macro-based approach. Second, the macroblocks did not contain relative routing information.

The first step of Frontier decomposes the FPGA into a set of placement bins of equal size. Next, the macroblocks are grouped into clusters such that each cluster could fit into a placement bin. If not all clusters can fit into the bins, the algorithm is restarted with a larger placement bin size. After all clusters have been assigned to a placement bin, they are subsequently swapped to minimize bin placement costs (which were determined by inter-bin connectivity). After bin placement, the macroblocks within the clusters are each placed within their respective placement bin based on neighboring connectivity. At this point, Frontier can terminate, or, if the placement is calculated to be difficult to route, a low temperature simulated annealing process can be performed to further improve the quality of the placement.

Tessier was successful in achieving $17 \times$ speedup in placement or a $2.6 \times$ speedup in overall place and route when compared to the Xilinx PAR-M2.1 in macro-based mode (which is no longer is available in current Xilinx tools). When compared to the flattened design mode of the PAR-M2.1 software, placement achieved a speedup of $50 \times$ and an overall speedup of $8 \times$ for both placement and routing.

Although macroblocks can be effective at accelerating placement, hard macros have an additional advantage over macroblocks. Hard macros have the potential to contain routing information which means a hard macro contains a placed and routed implementation of a circuit and thus, no further effort is necessary to map it to the FPGA fabric. The presence of finished routes in the hard macro can have a significant impact on routing time as several nets in a design can already be routed before the final routing process. Nevertheless, macroblocks can be an effective technique to reduce overall placement time without impacting design implementation quality.

**Clustering and Hierarchical Simulated Annealing Placement**

Sankar and Rose [2] proposed a similar objective to that of this work. That is, there exists a growing set of FPGA designers who are willing to trade circuit quality for shorter compile times. Sankar and Rose were able to demonstrate a $52.4 \times$ speedup in the
placement process if users are willing to accept a 30% increase in total routing wire length. To accomplish this, Sankar and Rose used a multi-level clustering technique to group the primitives of a design into clusters based on their connectivity. This step was followed by a custom simulated annealing algorithm to iteratively improve the placement.

2.3.3 Routability-driven Routing

Swartz et al. [3] as with [2], also proposed trading quality for faster compile times. However, instead of optimizing placement, Swartz et al. focuses on accelerating routing. The method used is a basic PathFinder algorithm that attempts to reduce the total number of iterations required to resolve congestion. It also has mechanisms for trying to identify un-routable scenarios quickly up front to avoid spending a lot of time on an impossible routing task.

The significant methods used by Swartz et al. are first to use a directed depth-first search expansion when routing a net. The cost function is modified to direct the search towards the desired sink much more quickly than the conventional breath-first search. The second method used is to filter out the routing resources considered for expansion during the search. By being more selective of the routing resources that are followed during the routing process, much time savings can be obtained by avoiding traversal of fruitless paths.

Overall the techniques used are quite effective, however, they were implemented using VPR [11] which makes comparison to commercial tools difficult. Swartz et al. states that the routability-driven router can route a 20,000 logic block circuit in 23 seconds on a 300 MHz sparcstation.

2.3.4 Summary and Overview of this Work

Although many of the techniques mentioned in this section have proven to be effective, they each focus on a single corner of the larger compilation time problem. The goal of this dissertation is to demonstrate a comprehensive approach to accelerating FPGA compilation that combines the usefulness of hard macros for design re-usability with a complete end-to-end compilation flow tuned at each step for rapid compilation. The approach taken here
also focuses on demonstrating the techniques on commercial FPGAs as such work is proof of
concept in its own right. Using these different techniques, this dissertation will demonstrate
a new way to compile FPGA designs that is significantly different than the conventional
approach.
CHAPTER 3. RAPIDSMITH: AN OPEN SOURCE PLATFORM FOR CREATING FPGA CAD TOOLS FOR XILINX FPGAS

At the onset of this work, there was no organized way to build CAD tools for commercial FPGAs and far less available to implement a rapid prototyping compilation flow. This chapter motivates the need for a new FPGA CAD tool framework called RapidSmith, a Java-based Xilinx CAD tool framework written specifically to support the endeavors of this work.

3.1 Introduction

One of the biggest challenges of performing FPGA CAD experiments on commercial FPGAs is the difficulty in gaining access to a suitable database containing all of the native resources on an FPGA. This challenge has caused many FPGA researchers to gravitate toward hypothetical FPGA architectures found in popular tools such as VPR [11] to perform CAD experiments.

At the onset of this work, it was decided that in order to properly address the long standing problem of time-consuming compilation times for commercial FPGAs, it would be necessary to create custom CAD tools for commercial devices. However, as already mentioned, the FPGA vendor must make available a means to obtain a database of FPGA resources and also import/export methods to allow low-level access to FPGA designs.

Altera provides a research-based interface to their tools called the Quartus University Interface Program (QUIP) [12]. QUIP provides some detail about their devices that enable researchers the ability to create tools such as front-end synthesis, technology mappers, placers and global routers. However, it does not allow its users the ability to create detailed routers which would be an essential part of accelerating the FPGA compilation flow. Ultimately,
after examining the offerings of Altera, it was concluded that the provided interface lacked certain features to allow the implementation of this work’s experiments and meet its goals.

Xilinx provides a more complete interface called the Xilinx Design Language (XDL) and an accompanying tool *xdl* [13] that provides enough information to create all the tools enabled by QUIP, and additionally that of a detailed router. It also allows conversion of proprietary Xilinx netlist formats (NCD and NMC) to be converted to and from the human readable ASCII XDL netlist format. This conversion functionality enables the user to make very specific and detailed changes to the netlist of a design. One of the disadvantages of XDL, however, is that it can be quite cumbersome to use. For example, the text-based database report files generated by the *xdl* tool can reach over 70 gigabytes. This makes direct access of such files infeasible for tools such as placers and routers. In addition, there are some minor gaps in the information that is provided by XDL report files (also called XDLRC files) that must be acquired through other means. Despite these shortcomings, Xilinx FPGAs (using XDL as their interface) was choosen to demonstrate this work.

Given the usefulness but unfortunate shortcomings of XDL, it was clear that a CAD tool framework to leverage XDL would be necessary to carry out the experiments described later in this work. The XDL framework created for this purpose is called RapidSmith. The remainder of this chapter summarizes related work, describes XDL and RapidSmith and how RapidSmith is able to overcome the shortcomings of XDL and provide an effective rapid prototyping framework for FPGA CAD tools.

### 3.2 Related Work

Some researchers have used XDL before to perform certain kinds of experiments such as a bus macro generator [7], half latch detection and removal [14], C-slow re-timing [15], power estimation [16], floor planning tools [4], routing constraints [17][18] and run-time reconfiguration [19]. Despite these and several other efforts which leverage XDL, a unified open source solution to facilitate the use of XDL has never materialized until recently. Steiner [6] and Kepa et al. [20] both demonstrate functional XDLRC-derived device databases, however they have not (at the time of writing) been openly released.
3.2.1 Torc

Torc [21] is a very similar and capable set of tools and framework to that of RapidSmith. Developed by University of Southern California’s Information Sciences Institute (USC-ISI), Torc aims to provide a more comprehensive tool set than RapidSmith, including front end tools such as technology mapping and packing. It also proposes support for other FPGA vendors if the information can be obtained which contrasts with RapidSmith goals as it was written specifically for Xilinx devices. Torc leverages XDL in the same ways as RapidSmith, however it is written in C++ whereas RapidSmith is in Java. Ultimately Torc was not used because it was not available in time for this work and the ideas and concepts of this work were able to be more quickly prototyped in Java than C++.

3.3 XDL: The Xilinx Design Language

RapidSmith is based on the interface provided by Xilinx called the Xilinx Design Language (XDL). XDL is provided mainly in the form of an executable, xdl. The Xilinx tool xdl provides three important capabilities (options) that enable the creation of an FPGA CAD tool framework for Xilinx FPGAs:

1. -report: Generates complete FPGA device reports detailing placement sites and an exhaustive routing graph (without timing data)

2. -ncd2xdl: Performs NCD to XDL conversion (allowing conventional Xilinx designs to be converted to the open XDL format)

3. -xdl2ncd: Performs XDL to NCD conversion (allowing XDL-manipulated designs to be re-injected into several locations within the Xilinx design flow)

With these options, the xdl tool provides detailed device information and can import and export designs to and from the Xilinx flow. This section serves as a short reference for XDL as it is no longer fully documented by Xilinx and a correct understanding of XDL is necessary in order to fully understand the value and capabilities of RapidSmith.
3.3.1 Detailed FPGA Descriptions in XDLRC Reports

The first option supplied by xdl is the -report option. This option takes a Xilinx part name as input and outputs a file with the extension .XDLRC that is a report file detailing all of the tiles, primitive sites and routing resources (without timing data) that are available in the given part. To obtain the most detailed information possible, the options -pips -all_conns are also included in the command. These XDLRC report files are generally several gigabytes in size and some XDLRC file sizes are shown later in the chapter in Table 3.1.

XDLRC report files have a very predictable and regular structure making them easy to parse and harvest valuable FPGA architectural information. These reports have two main sections: tiles and primitive definitions. An example of some XDLRC report file syntax is shown in Listing 3.1. The tile section as seen in lines 10–15, describe a complete floor plan layout of a device and details all of the separate resources found in the tiles such as primitive sites and routing resources such as wires and PIPs. Line 10 gives the dimensions (height and width) in tiles of the FPGA layout. The primitive definition section as shown in lines 16–20 is family specific (Virtex 4, Virtex 5, etc.) and is included in every XDLRC file generated by xdl. It contains a listing of all primitives supported by the device’s family giving detailed information about each such as inputs, outputs, internal connections and resources.

Listing 3.1: Basic Structure of an XDLRC Report File

```plaintext
1 # =======================================================
2 # XDL REPORT MODE $Revision: 1.6 $
3 # time: Thu Apr 16 13:58:39 2009
4 # cmd: c:\Xilinx\10.1\ISE\bin\nt\xdl.exe -report -pips -all_conns xc4vfx12
5 # ==============================================================
6 (xdl_resource_report v0.2 xc4vfx12sf363-12 virtex4
7 # **************************************************************************
8 # * Tile Resources *
9 # **************************************************************************
10 (tiles 73 66
11  (tile 0 0 EMPTY_CORNER_X0Y72 EMPTY_CORNER 0
12    (tile_summary EMPTY_CORNER_X0Y72 EMPTY_CORNER 0 0 0)
13  )
14  )
15 )
16 (primitive_defs 41
17  (primitive_def BSCAN 8 10
18  )
19 )
```
Tiles

XDLRC report files abstract the FPGA fabric into a two dimensional array of sections called “tiles” which are conceptually laid out edge to edge in a checker board fashion. Each tile is specified with a unique name and a type. The tile name often includes an X and Y coordinate as a suffix and the beginning of the name often exemplifies its type. For example, a tile named CLB_X2Y3 is of type CLB and is the 3rd CLB to the right of the origin and 4th above the origin (the origin being the bottom left corner of the FPGA). An interesting aspect of the X and Y coordinate pairs are unique to the tile type rather than to every tile declared in the grid. For example, a tile called CLB_X2Y3 will have a switch box tile directly to its left which will have the same X and Y coordinates will be called INT_X2Y3.

Tiles are mainly used as a reference to specify locations of certain resources. Tiles contain declarations of FPGA resources such as primitive sites, wires, and PIPs.

Primitive Sites

A primitive site is a location where an instance of an FPGA primitive can reside. Each primitive site has a type that is compatible with one or more primitive instance types. For example, a SLICEL is a type of primitive available in Virtex 4 parts. Half of all slice primitive sites in Virtex 4 parts are type SLICEL and the other half are type SLICEM. A SLICEL primitive can be placed on either a SLICEL or SLICEM site, however, the converse is not true. As mentioned earlier, a list of all primitive types is included in every XDLRC report file.
Wires

Wires declared in XDLRC reports refer to wires which cross tile boundaries. A wire is simply un-programmable metal that exists on the FPGA fabric as part of the routing graph. A wire always begins in a tile and can span one or more tiles. One of the peculiar attributes of XDLRC is that even though a wire represents a continuous piece of metal in the FPGA fabric, a separate name is given to each segment of the wire for each tile it occupies. Thus, a wire which spans 3 tiles will have three separate names attached to it depending on which tile is being referenced.

PIPs

Programmable interconnect points, or PIPs, are the configurable part of the FPGA routing graph. PIPs are completely contained within a tile (they do not straddle a tile) and define a potential connection that can exist between two wire segments. In most cases, thousands of PIPs are defined in the switch matrix tiles of an FPGA, most of which allow a certain wire segment to connect to one of many different wire segments. Some PIPs are used simply to denote a one-to-one connection between two wire segments within a tile and are sometimes referred to as “fake PIPs” as they do not affect the bitstream. These fake PIPs exist simply to complete the connectivity of the routing graph as their absence would make routing impossible with XDLRC reports alone.

3.3.2 Designs in XDL

The second and third options provided by xdl are the XDL/NCD conversion functions which enable proprietary Xilinx NCD files (native netlists for Xilinx FPGAs) to be converted to and from the open XDL format. A design in XDL is equivalent in most respects to Xilinx NCD files except that it is an ASCII file format that is human readable and it also is directly incompatible with other Xilinx tools in the flow. However, by converting XDL to NCD files, XDL can be re-injected into several different steps of the Xilinx flow such
Figure 3.1: RapidSmith and XDL Interacting at Different Points within the Xilinx Tool Flow as after packing (`map`), after placement (`par -r`), and after routing (`par -p`) as shown in Figure 3.1.

XDL design files contain four types of declarations: design, module, instance and net. The first statement found in XDL designs is the design statement and occurs only once in each design file. It specifies the design name and part that the design is targeting. The module declaration is not a commonly used construct in XDL but is powerful in that it enables a level of hierarchy in designs and is capable of representing a hard macro (a pre-placed and routed design block). The majority of the contents of an XDL design exist as instances and nets.

**Instances and Placement**

All logic elements of a design are contained within the instances declared in the XDL file. An instance is an instantiation of a particular primitive type (SLICEL, SLICEM, etc.) that has a unique name, a type, an optional primitive site assigned to it and a list of attributes and values which configure the instance. Since assignment to a primitive site is optional, XDL files can represent designs that are unplaced, partially placed or fully placed. Placement of a design takes place when instances are assigned specific primitive site locations in the
final implementation. Therefore, creating a tool to automatically place a design has very little overhead when using XDL as it simply implies creating a mapping between instances and primitive sites.

Nets and Routing

In order to specify connections between instances, the net declaration is used. A net has a unique name, a type, a list of input/output pins and optionally, a list of PIPs. Each net has one outpin (source) and one or more inpins (sinks). Pins are declared by using the unique instance name and pin on the instance. PIPs define how a net is routed and the presence of PIPs in a net indicate that the net is at least partially routed if not fully routed. Therefore, the task of a router is simply to assign a list of PIPs to a net. Nets cannot be routed if a design is not placed (PIPs require placed instances to make connections), however, XDL can represent designs that are un-routed, partially-routed or fully routed.

3.4 RapidSmith: A Framework to Leverage XDL and Provide a Platform to Create FPGA CAD Tools

There are several challenges that exist to successfully use XDL to build new CAD tools and leverage it as a design exchange method. Although XDLRC report files are extensive and provide a complete view of an FPGA architecture, their representation is extremely inefficient as the files can exceed 70 gigabytes in size. This makes direct use of XDLRC report files infeasible for creating design tools. Another challenge exists due to lacking information in the XDLRC report files that is necessary for efficient placement and routing. An open source XDL parser and data structure is not provided with the xdl tool making XDL manipulation often delegated to Perl scripts created for each specific task.

RapidSmith provides two core packages to deal with the challenges of using XDL. First, RapidSmith provides a compact XDLRC-equivalent database file format and corresponding data structure (pictured in Figure 3.2a) that is fast loading and suitable for all
kinds of applications. RapidSmith also includes the lacking information that can aid in making more effective placement and routing tools.

Second, RapidSmith provides a complete XDL parser and corresponding data structure (pictured in Figure 3.2b) to easily facilitate manipulating XDL designs and enable researchers to build highly customized CAD tools for Xilinx FPGAs.

### 3.4.1 Xilinx FPGA Database Files in RapidSmith

A very important attribute of an FPGA CAD database is its size and consequently, its load time from disk. RapidSmith employs three major strategies in order to control its device database size and load time; (1) aggressive wire and object reuse, (2) careful pruning of unnecessary wires, and (3) customized serialization with compression.

#### Wire and Object Reuse

Wires are the major reason for the gigantic size of the XDLRC report files. FPGAs are generally very regular replicated structures and the same wires can appear identically all over the chip. Efficiently accommodating the infrequent irregularities that occur in the routing structure is key to creating an efficient and much smaller device database. Therefore, rather than define a wire object for every wire in every tile, RapidSmith employs a unique set of wires that are specified by a name and a set of tile offsets that remove reference to
any particular tile in the chip, making them reusable. This reusability factor is able to dramatically reduce the size required to represent wire connections on the device.

**Wire Graph Pruning for Efficiency**

The XDLRC report contains wires that are present simply for the sake of completeness, but, do not contribute more useful information to the actual routing structure. For example, in a HEX wire (a wire which spans 7 switch matrices), connections to the wire can only be made in 3 of 7 tiles which it spans. The 4 wire segments in tiles that do not connect to other wires are implied by the overall structure and thus, removed by RapidSmith. By removing these wires, significant memory savings are obtained as well as faster routing times as the removed wires are no longer examined when expanding connection searches.

**Serialization and Compression**

RapidSmith uses custom serialization to store only the essential data needed to reconstruct the device databases. The default Java serialization routines were considered, but were later abandoned as they turned out to be inefficient and caused files to load more slowly. RapidSmith uses the Hessian 2.0 serialization compression protocol [22] to further reduce the size of the device file. The Hessian protocol provides low-level serialization APIs and also compresses/decompresses the data found in the device files with minimal impact on load time.

**Overall Performance**

Using all of the techniques mentioned, RapidSmith database files are able to achieve a file size compression of over 10,000× when compared to the original XDLRC report files and all but the largest files can be loaded in less than a few seconds. A summary of large parts and their corresponding device file statistics are shown in Table 3.1. Measurements shown in Table 3.1 were recorded on a Windows 7 64-bit workstation with a Core i7-860
Table 3.1: RapidSmith Device Files Performance

<table>
<thead>
<tr>
<th>Xilinx Part Name</th>
<th>XDLRC Report Size</th>
<th>Compressed File Size</th>
<th>Java Heap Usage</th>
<th>Load Time From Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>V4 SX55</td>
<td>3.5GB</td>
<td>539KB</td>
<td>34MB</td>
<td>0.299s</td>
</tr>
<tr>
<td>V4 FX140</td>
<td>8.0GB</td>
<td>1546KB</td>
<td>70MB</td>
<td>0.616s</td>
</tr>
<tr>
<td>V4 LX200</td>
<td>10.0GB</td>
<td>1010KB</td>
<td>61MB</td>
<td>0.602s</td>
</tr>
<tr>
<td>V5 FX200T</td>
<td>9.4GB</td>
<td>1227KB</td>
<td>60MB</td>
<td>0.585s</td>
</tr>
<tr>
<td>V5 TX240T</td>
<td>10.0GB</td>
<td>1111KB</td>
<td>56MB</td>
<td>0.620s</td>
</tr>
<tr>
<td>V5 SX240T</td>
<td>11.9GB</td>
<td>1135KB</td>
<td>61MB</td>
<td>0.630s</td>
</tr>
<tr>
<td>V5 LX330</td>
<td>12.5GB</td>
<td>1069KB</td>
<td>69MB</td>
<td>0.622s</td>
</tr>
<tr>
<td>V6 CX240T</td>
<td>8.5GB</td>
<td>937KB</td>
<td>35MB</td>
<td>0.460s</td>
</tr>
<tr>
<td>V6 SX475T</td>
<td>17.7GB</td>
<td>1506KB</td>
<td>61MB</td>
<td>0.814s</td>
</tr>
<tr>
<td>V6 LX760</td>
<td>22.8GB</td>
<td>1758KB</td>
<td>77MB</td>
<td>1.068s</td>
</tr>
<tr>
<td>V7 855T</td>
<td>32.0GB</td>
<td>2634KB</td>
<td>115MB</td>
<td>1.408s</td>
</tr>
<tr>
<td>V7 1500T</td>
<td>53.0GB</td>
<td>4985KB</td>
<td>263MB</td>
<td>2.653s</td>
</tr>
<tr>
<td>V7 2000T</td>
<td>73.6GB</td>
<td>5956KB</td>
<td>301MB</td>
<td>3.339s</td>
</tr>
</tbody>
</table>

processor, 8GB of DDR3 RAM and 1TB 7200RPM SATA hard disk. The 32-bit Oracle JVM ver. 1.6.0.22 was used for Java bytecode execution.

3.4.2 Augmented XDLRC Information in RapidSmith

The XDLRC report neglects to inform the user about primitive sites that can support more than one primitive type. The lack of this information provides two inefficiencies or challenges: (1) placements produced without this information may not be complete or may result in inefficient placements and (2) certain primitives do not have native sites of their own and therefore must reside on sites of a different type. When a primitive resides on a site of a different type, there can be pin name mis-matchings that can cause problems for a router.

Using RapidSmith, we have built special programs to automatically compile a complete listing of primitive site compatibilities and pin name mappings. This information is included with the RapidSmith distribution and integrated into the APIs to provide a seamless solution. Without the help and supplementation provided in RapidSmith, XDL users
would have to recompile this information manually or re-implement several of the programs already included with RapidSmith.

3.4.3 XDL Design Representation in RapidSmith

As already shown, Figure 3.2 illustrates how RapidSmith data structures represent the data contained in XDL. Each class in RapidSmith will often represent a one-to-one relationship with XDL/XDLRC constructs that make using RapidSmith intuitive to use with XDL designs and XDLRC reports. A fully optimized XDL parser and export method are included with RapidSmith to populate its data structures and produce Xilinx-compatible XDL for re-injection into the Xilinx flow. RapidSmith also includes graphical tools to browse and explore devices and designs as shown in Figure 3.3. The Device Browser, shown in Figure 3.3a, illustrates how the user can explore the resources and wires available in a device in RapidSmith and demonstrates the framework which can be used by researchers to create custom graphical CAD tools that interact with the FPGA fabric. The Device Browser leverages the classes outlined in Figure 3.2a. The Design Explorer application, shown in Figure 3.3b, allows the user to load XDL designs and, optionally, timing reports to rapidly search through the different constructs and leverages the classes found in Figure 3.2b.

The APIs available in the design package of RapidSmith are particularly rich with over 700 documented methods for design manipulation. These methods include functionality such as placement/un-placement of instances, creating new instances from scratch, determining the type of nets (wire, VCC, GND, CLK, etc), changing the source of a net, un-routing a net, manually routing a net (adding PIPs), and hundreds of other useful design transformations.

3.4.4 Impact of RapidSmith

RapidSmith has been released as an open source project under the GPL license and is available at http://rapidsmith.sourceforge.net. In the first 12 months since its release, it has been used and/or cited in 12 published works. In addition, RapidSmith has been used
Figure 3.3: Screenshots of Graphical Tools Provided with RapidSmith to Browse (a) Devices or (b) Designs
as project material in graduate-level courses taught at both Brigham Young University and Virginia Tech.
CHAPTER 4.  HMFLOW 2010: ACCELERATING FPGA COMPILATION WITH HARD MACROS FOR RAPID PROTOTYPING

With the creation of RapidSmith as described in Chapter 3, the necessary pre-requisites were in place to create a custom, end-to-end FPGA compilation flow based on hard macros. The primary goal of this custom CAD flow is to provide extremely fast compilation that would be useful in rapid prototyping scenarios where achieving high clock rates were not necessarily a priority. That is, in creating such a compilation flow, its optimization and implementation were dictated by trying to answer the following question:

“Using hard macros and completely ignoring circuit quality, how fast can FPGA compilation be made to run?”

This chapter describes the preliminary work, the various components and results of such an endeavor to create a fast, rapid prototyping FPGA compilation flow.

4.1 Preliminary Work

This section describes the benefits of using hard macros and why they were chosen as the mechanism for storing intermediate compiled circuit information. It also describes a series of preliminary experiments that prove hard macros are capable of accelerating compilation and that they are a feasible approach to creating a complete FPGA compilation flow.
4.1.1 Selection of a Compiled Circuit Representation

Storing pre-compiled circuit information can be done in a number of ways. The most common method used by Xilinx and other IP vendors is as a synthesized netlist. A netlist describes the primitive components that compose a design as well as a list of nets describing their interconnections. Compiling designs strictly from netlists would provide some compilation time acceleration, but it would be limited at best since it only allows the user to skip the synthesis step.

Another technique to store a compiled circuit representation would be a relationally-placed macro (RPM). Xilinx has a specific RPM format that allows FPGA primitives to be described in relation to each other in terms of relative placement on the FPGA fabric. As RPMs exist at the primitive netlist level (NCD/XDL), RPMs allow the user to skip synthesis, mapping, packing and placement.

The most comprehensive method for storing pre-compiled FPGA circuit descriptions, however, is a hard macro. Hard macros have all the benefits of RPMs with the additional feature of being able to store relative routing information. Hard macros are able to provide a more complete implementation of a circuit, yet, are still able to maintain a similar level of placement flexibility with RPMs by leveraging the regularity of FPGA routing structures. Ultimately, hard macros can skip all of the major steps in FPGA compilation: synthesis, mapping, packing, placement and routing.

Although hard macros contain pre-compiled information, they must be assembled into a design, placed at some location on the FPGA fabric, and connections in between hard macros must be routed. However, these steps are much less complex than conventional FPGA design compilation. For example, one of the advantageous side-effects of using hard macros as the common design element of FPGA design is that the number of objects requiring placement in the final placement phase is significantly reduced. Typical FPGA designs often contain 10,000s of primitives that must be placed, whereas hard macro-based designs only contain a few dozen to a few hundred hard macros. For this and the reasons mentioned above,
hard macros were chosen as the compiled circuit representation to obtain the maximum savings in compilation time.

It should be mentioned that the placement of hard macros onto an FPGA versus the placement of macroblocks in ASIC design are different problems. Automating the placement of hard macros onto FPGA fabric is harder in that there are several more constraints placed on hard macros than placing a macroblock in a conventional ASIC design. An FPGA does not have a perfectly uniform fabric, and modern FPGAs have varying patterns of resources such as block RAM and DSPs that can make placement challenging for a hard macro that contains such components. Also, hard macros can be routed and must use specific routing resources which must align exactly with those specified within the hard macro or it cannot be placed. These two constraints make the granularity by which large hard macros can be placed, very rough and non-linear. Even though research on macro-based ASIC placement has been performed [23], it cannot be directly applied in the case of hard macros on reconfigurable logic.

4.1.2 Experiments Validating Hard Macro Potential

At the onset of this work, it was necessary to prove the feasibility of using hard macros as a basic design element and their potential to accelerate FPGA compilation. In order to accomplish this task, it was crucial to understand how well such a hard macro-based tool flow is currently supported by the Xilinx tools. To determine this, a few experiments were conducted. The first experiment aimed to find out how well hard macros were supported by the Xilinx tools. The second experiment then tried to determine how much potential speedup a hard macro-based design flow could offer.

Hard Macro Creation

The preliminary task of these experiments was to first build a set of hard macros which could then be assembled into a design. Although the Xilinx tools do support hard
macros, our experiments have found that there are some significant limitations to using them with the conventional flow. The first limitation is the hard macro creation process. Xilinx documents a method for creating hard macros [24] using FPGA Editor. Although this is a tedious manual process, FPGA Editor can be scripted to some extent [25] for creating hard macros.

To create a set of hard macros for use in our experiments, we developed a Hard Macro Generator tool (HMG) and associated flow as shown in Figure 4.1. Specific hard macro generation programs have been created before as in [7] and [25], however, at the time, a tool to create general hard macros from arbitrary RTL has not been created before. All circuit manipulations in the experiments were performed in XDL using RapidSmith as described in Chapter 3. The hard macro creation flow leverages the first four steps of the conventional Xilinx design flow as shown in Figure 4.1. Designs were prepared by connecting ports intended to become terminals on the hard macros to IOBs in the FPGA. In addition, reasonably tight area constraints were placed on the design to make the hard macros more compact and increase their place-ability (or increase the number of valid locations on which they could reside).
Once a design was adequately prepared, it was implemented using the conventional Xilinx tool flow which produced a completely placed and routed FPGA implementation with only the circuitry bound for inclusion in the hard macro. After converting the designs into XDL, the HMG tool would take the XDL as input and convert it into a stand-alone hard macro. The steps taken to create a hard macro from the original XDL are listed below:

1. All IOBs were removed and hard macro port objects were created and attached to corresponding XDL Pins.

2. Measures were taken to ensure that each net had only one hard macro port as more would cause an error.

3. All GND and VCC sources (TIEOFFs and SLICEs) were removed as they are an illegal construct in a hard macro. Separate ports were created in their place with a special naming convention to indicate a GND or VCC input should be supplied.

4. All clock nets and circuitry were removed (our hard macros only used a single clock domain) and in their place a clock port was added to the external port list of the hard macro.

The result was an XDL representation of the hard macro which was then converted to the Xilinx NMC format (the Xilinx hard macro counterpart format to NCD).

**Experiment #1: Find Out Support Level for Hard Macros in Xilinx Tools**

With a small set of hard macros to use for hard macro design, experiment #1 set out to determine how robust the Xilinx tools are when operating on hard macro-based designs. To create complete design implementations for this experiment, structural VHDL was written which instantiated the hard macros and specified their inter-connections. This VHDL was then passed through the Xilinx tool flow starting with synthesis (`xst`) through place and route (`par`). The compilation flow for experiment #1 is simply the conventional
Xilinx flow (xst, ngdbuild, map and par) as shown in Figure 4.2. Since the design was an interconnected set of black boxes in the VHDL, no real synthesis was performed nor was there any mapping or packing done. That is, although the hard macro designs were processed by xst, ngdbuild and map, no real computation or translation was performed as the hard macros already existed at the native netlist level. Therefore, the bulk of the runtime was spent in PAR\(^1\).

![Hard Macros instanced as black boxes in structural VHDL](image)

Figure 4.2: Compilation Flow for Experiment #1 (Conventional Xilinx Flow)

**Results of Experiment #1: Mixed Support for Hard Macros**

Several compilations of different hard macro-based designs were completed using the flow shown in Figure 4.2 with varying results. The sample designs used in these tests included a QPSK demodulator as well as various designs created by inter-connecting various open source cores found on the web which are listed later in the section describing Experiment #2. The results of these tests can be grouped into three categories:

1. For one collection of tests the design successfully placed and routed, but the resulting tool chain time was significantly longer than an equivalent process starting from a standard HDL-based design. As indicated in [24], it is known that using hard macros actually causes Xilinx PAR to run more slowly.

\(^1\)A final hard macro-based design flow does not use xst, ngdbuild, map, or par of the Xilinx flow but rather would directly combine, place, and route the hard macros at the XDL level—this technique was employed for the experiments only to make it possible to quickly obtain results.
2. For another portion of the tests, no valid placement for the hard macros in the design could be found and the placement phase would fail. At times PAR would fail with an error message and at other times placement would simply never terminate within a reasonable amount of time.

3. Finally, for some tests, placement would complete successfully but the router would fail with an error message that it could not route all the nets in the design.

These results suggest that, in spite of the intuitive advantages that a hard macro-based design flow would seem to offer, Xilinx PAR does not work well with such a flow. The results further suggest that placement was the problem—either it failed to place the designs, or it seemed to create an un-routable placement.

**Experiment #2: Hard Macros in an Augmented Xilinx Flow**

The second experiment was designed to bypass the Xilinx placement step to understand the router’s handling of hard macros. To do so, the original VHDL-based structural design was processed as before but only up to mapping and packing. The result was an unplaced NCD file (the overall flow for experiment #2 is shown in Figure 4.3). The hard macros were then placed by hand using a custom created XDL macro placer tool. That is, placement of each block was chosen manually and then a custom created XDL-based tool was used to modify the un-placed XDL design file to reflect that placement. The resulting placed design was then passed to the Xilinx router (PAR) for completion.

To make an accurate measurement of runtime improvement, a baseline (non-hard macro) design was created for each test. The baseline tool runtime was determined by the total runtime of the tools from the designer’s original RTL implementation (VHDL or Verilog files) to a fully placed and routed design in NCD format. The baseline measurement omits bitstream generation as this process cannot be accelerated due to the proprietary nature of the bitstream (it consumes identical runtime for both the experiment and baseline circuits).
The command line parameters to the Xilinx tools were set to obtain the fastest runtime (ignoring timing constraints, standard effort levels, and avoiding timing-based MAP).

The first design is a multiplier tree made of a $20 \times 20$ bit LUT-based multiplier with a pipeline depth of 5. The Multiplier tree has 15 identical instances of the multiplier where the arrangement of the multipliers is that of a binary tree as shown in Figure 4.4. This design was chosen to estimate the speedup that a hard macro-based flow would produce when the synthesis work load is small (the multiplier was synthesized only once).

The second design is a collection of five different cores: CORDIC, AES decryptor, Twofish (a symmetric key block cipher), FM Receiver, and a Hilbert Analytic Filter. All
cores were placed in the design and connected together to form a data-path with inputs and outputs connected to external IO pins. Although this design has a smaller hardware footprint than the multiplier tree, its heterogeneous nature implies that more time will be spent during synthesis and will potentially lead to increased speedup for the hard macro based flow.

Results of Experiment #2: Reductions in Runtime

Table 4.1: Baseline Runtimes for each Test Design

<table>
<thead>
<tr>
<th>Design</th>
<th>XST</th>
<th>NGDBuild</th>
<th>MAP</th>
<th>PAR</th>
<th>Total Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mult-Tree</td>
<td>46.7s</td>
<td>9.0s</td>
<td>17.7s</td>
<td>64.2s</td>
<td>137.5s</td>
</tr>
<tr>
<td>Heterogeneous</td>
<td>80.2s</td>
<td>4.9s</td>
<td>10.4s</td>
<td>35.5s</td>
<td>131.0s</td>
</tr>
</tbody>
</table>

The baseline results for each design are shown in Table 4.1. All runtimes were obtained on a desktop workstation with an Intel Core 2 Duo 3.0GHz (E6850) processor with 4GB of RAM, running Windows XP Pro SP3 and Xilinx ISE 11.4. All designs targeted a Virtex 4 SX35 (xc4vsx35ff668-10). Comparisons were made primarily on runtime, however, clock rate and hardware utilization are included as well to demonstrate the trade-offs between design runtime and QOR.

Table 4.2: Performance of each Test Design Using Hard Macros

<table>
<thead>
<tr>
<th>Design</th>
<th>Custom Placer</th>
<th>XDL to NCD</th>
<th>PAR -p (route only)</th>
<th>Total Runtime</th>
<th>Speedup Over Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mult-Tree</td>
<td>4.3s</td>
<td>14.3s</td>
<td>25.7s</td>
<td>44.3s</td>
<td>3.1×</td>
</tr>
<tr>
<td>Heterogeneous</td>
<td>4.3s</td>
<td>12.1s</td>
<td>25.5s</td>
<td>41.9s</td>
<td>3.1×</td>
</tr>
</tbody>
</table>

As can be seen in Table 4.2, the hard macro designs were both built 3.1× faster than their corresponding baseline build. These results offer promise to a design flow created entirely from hard macros and would enable significantly faster build times. What is also
notable about these results is that they are two very different designs but both achieve the same speedup by using hard macros. This could be caused by a floor on the performance of PAR (actually, just the router since in these experiments we did our own placement). That is, both hard macro implementations took almost an identical amount of time to run PAR. It is possible that for designs of a certain size, PAR may not be able to execute much faster than about 25 seconds. If this were the case, it may be beneficial to create a faster router for these kinds of situations.

### 4.1.3 Hard Macros and Quality of Results

The QOR of each circuit (as seen in Table 4.3) implementation is somewhat mixed. The multiplier tree baseline and hard macro implementation have surprisingly similar hardware utilization and maximum clock rates. This is an excellent result as it shows (in some cases) that using hard macros will not be much worse than using the Xilinx tools (when optimized for runtime). However, the heterogeneous design baseline produced a higher clock rate and significantly smaller hardware footprint. This is probably due to the fact that the synthesizer was able to optimize some of the logic across the different block boundaries leading to lower hardware utilization and a higher clock rate. No such optimization was possible in the hard macro implementation. It must be noted, that the Xilinx tools were configured for the fastest runtime in all of the experiments and it is likely that if configured for best quality of result, clock rates of the baselines would be higher.

### 4.1.4 Hard Macros and Placement Time

Note that the runtimes measured in the augmented flow included the time it took to route the hard macros together, but not the amount of time it took to place them. Placement times were assumed to be negligible for these experiments based on the following

---

2This meant running the Xilinx tools without a timing constraint (or providing a very loose one when necessary), setting all effort level parameters to their lowest setting (-std) and disabling timing-based placement where applicable.
analysis. Placement time for the hard macro experiments can be estimated by assuming
that placement runtime behavior is approximately linear in the number of design elements
to be placed. The baseline designs consisted of 4597 and 1757 slices and consumed 35.1s
and 25.5s of placement time, respectively (placement time was calculated by running par
with the -r option, which only runs the placer). The hard macro designs consisted of 15
and 5 blocks. Extrapolating using the linear-behavior assumption, placement times can
be estimated to be 0.11s and 0.07s, respectively, for the hard macro experiments. Adding
these values back into the measured runtimes does not change the result from that shown in
Table 4.2. This substantial reduction occurs because the hard macro placer only places the
top-level hard macros—placement time for their constituent cells is eliminated as they were
previously placed and routed. Note that the presumption of linear behavior is conservative.
If the placer’s behavior is super-linear, the hard macro approach will perform better than
this analysis indicates, relative to the baseline. It will be shown later in Section 4.3.3 with
the implementation of a hard macro placer that this approximation proved accurate.

<table>
<thead>
<tr>
<th>Design</th>
<th>Slices</th>
<th>Clock Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mult-Tree (baseline)</td>
<td>4592</td>
<td>148 MHz</td>
</tr>
<tr>
<td>Mult-Tree</td>
<td>4830</td>
<td>150 MHz</td>
</tr>
<tr>
<td>Heterogeneous (baseline)</td>
<td>1746</td>
<td>82 MHz</td>
</tr>
<tr>
<td>Heterogeneous</td>
<td>2741</td>
<td>61 MHz</td>
</tr>
</tbody>
</table>

4.1.5 Conclusions on Preliminary Hard Macro Experiments

With concrete evidence that hard macros have potential to accelerate FPGA compi-
lation, the implementation of a complete, automated hard macro design flow ensued. This
hard macro flow, called HMFlow for short, operates completely on hard macros and designs
in XDL. The remainder of this chapter is devoted to HMFlow’s components and performance.
4.2 HMFlow 2010: A Rapid Prototyping Compilation Flow

HMFlow 2010 is the vehicle by which the effectiveness of hard macros is demonstrated for accelerating FPGA compilation. The first implementation of HMFlow was created in 2010 and is referred to as HMFlow 2010 to differentiate it from an improved version of the flow developed in 2011 (aptly named HMFlow 2011). Throughout the remainder of this chapter, any reference to HMFlow is referring to HMFlow 2010 (unless otherwise specified).

HMFlow begins by importing designs created using Xilinx System Generator as seen in Figure 4.5. Design data are stored in Simulink Model Files (.MDL) that are parsed by the design parser and mapper tool. The mapper also identifies each block in a design and its corresponding hard macro. If the hard macro does not exist in the hard macro cache, the mapper invokes the hard macro generator to create one for the mapper and to store it in the cache.

Once all of the hard macros have been created or retrieved from the cache, they are given to the design stitcher, which will “stitch” all of the hard macros together. The design stitcher also inserts appropriate I/O buffers (IOBs) and clock generation circuitry. The design is then passed to the hard macro placer and router to be exported as a final placed and routed implementation in XDL.
The remainder of this section describes in detail the design entry techniques, algorithms and steps that have been constructed to realize and implement HMFlow.

4.2.1 Xilinx System Generator

Although HMFlow can potentially be used with any design entry tool, System Generator was chosen for this work because it provided several benefits out-of-the-box. System Generator is well-suited to hard macros because it is a block-based design tool. HMFlow automatically converts each of the System Generator blocks into a hard macro as shown in Figure 4.6. This allievates the need of the designer to either specify where hard macro boundaries occur or placing this burden on the tools to dynamically discover hard macro boundaries. Also, since HMFlow uses System Generator designs as input, existing System Generator simulation tools can be leveraged for HMFlow-bound designs.

Another advantage of using System Generator is its design compilation path. HMFlow provides a rapid prototyping path so designs can be quickly tested on an FPGA. However,
once the design is functionally correct, a high quality implementation is needed for design deployment. Since HMFlow accepts designs that are System Generator compatible, a high QOR is easily obtained (albeit with much longer runtime) by simply using the conventional Xilinx System Generator tools.

4.2.2 Simulink Design Parsing

The first step of HMFlow is to parse the System Generator design stored in the Simulink Model file (MDL). HMFlow has a custom built JavaCC-based parser to perform this task and populates a custom made Simulink-compatible data structure.

One drawback of using System Generator is that information about design blocks, such as bit widths of ports, are not stored in the MDL file. To obtain this missing data, the parser has to recalculate these widths on-the-fly during parsing and it must be done on a block-by-block basis. Despite this technical challenge, HMFlow is able to support over 75% of the most commonly used System Generator blocks in HMFlow designs.

4.2.3 Hard Macro Cache and Mapping

The mapper receives the design in the form of a populated data structure from the parser and iterates over all of the blocks in the data structure to identify the System Generator blocks. For each block it finds, an MD5 hash is generated from its type and parameters to uniquely identify the block and its corresponding hard macro. The mapper then uses the hash to query the hard macro cache to find out if the hard macro has already been built. If the hard macro has already been built, it is loaded from the cache and given to the mapper. If the hard macro is not in the cache, the mapper invokes the hard macro generator.

4.2.4 Hard Macro Creation

Xilinx does not provide any method for automated hard macro creation of arbitrary designs. The only method provided is a tedious, manual process using FPGA Editor. There-
fore, an automated hard macro generator was created using RapidSmith. This hard macro
generator is an improved version of that used in preliminary experiments as mentioned in
Section 4.1.2.

To create a hard macro in HMFlow, a design block (NGC netlist from System Gener-
ator) is implemented with the conventional Xilinx tools. Designs are prepared by inserting
special port-identifying signal pass-through hard macros which are attached to each of the
inputs and outputs of a design as shown at the top of Figure 4.7. These special port hard
macros remain untouched as the design passes through the Xilinx tools and specify the ports
to create in the final hard macro.

After synthesis of the hard macro-bound design, resource utilization counts are ex-
tracted from the synthesis report to generate slice, DSP48 and BRAM area placement con-
Figure 4.8: (a) A FIR filter Design Compiled with the Xilinx Tools (b) The Same Filter Design Compiled with an Area Constraint to Create a Hard Macro

...constraints as shown in Figure 4.7. Constraint generation aims to generate the hard macros in a fairly regular region of the chip and provide at least 50% extra resources inside of the constraint than what was calculated and reported by the synthesis tool, XST. To illustrate the need for area constraints, consider the implemented filter design in Figure 4.8a. The conventional vendor tools cause a design to be sparsely spread out over the entire chip, making its reusability nearly impossible. However, with a proper area constraint applied during the compilation process, the implementation can be nicely compacted into a rectangular form as shown in Figure 4.8b.

Listing 4.1: UCF Area Constraint Example

```ucf
# All hard macro instances are declared as part of area group block
INST "sysgen_fifo_x0/fifo_13312/percent_full(4)1" AREA_GROUP = "block";
INST "sysgen_fifo_x0/fifo_13312/percent_full(3)1" AREA_GROUP = "block";
INST "sysgen_fifo_x0/fifo_13312/percent_full(2)1" AREA_GROUP = "block";
INST "sysgen_fifo_x0/fifo_13312/percent_full(1)1" AREA_GROUP = "block";
INST "sysgen_fifo_x0/fifo_13312/percent_full(0)1" AREA_GROUP = "block";
INST "sysgen_fifo_x0/fifo_13312/comp0.core_instance0" AREA_GROUP = "block";
```
These constraints are applied through a typical Xilinx user constraint file (UCF) following typical Xilinx conventions. An example of the automatically generated UCF for hard macro creation can be seen in Listing 4.1. The constraints also ensure that hard macros are compact and maximize the number of valid locations on which they can be placed on the FPGA. It should be noted, however, that area constraints can only restrict placement of FPGA primitive instances and only suggests (not constrains) the router to try and keep routes within the area. When area constraints are too tight, it is common for routing to “spill out” of the area constraint. The routing spill out varied depending on the tightness of the constraint. When the area constraint contained 50% of area overhead or more, it was rare to have any routing spill over in the designs tested. However, when the area constraint overhead was in the ranges of 0-25%, it was common to have routes leave the area constraint by as much as 2 to 4 tiles before returning inside the area constraint. Therefore, routing spill out is avoided as it can increase the overall size of the hard macro and ultimately reduce its place-ability.

Once the placed and routed design (NCD file) is generated with the Xilinx tools, it is converted to XDL for manipulation by the hard macro generator in the RapidSmith framework. The hard macro generator then performs the same transformations (see Section 4.1.2) on the design to convert it to a Xilinx compatible hard macro. The hard macro is then stored in the hard macro cache using a similar compact format to that used in RapidSmith device files as described in Section 3.4.

4.2.5 XDL Design Stitcher

After the mapper has gathered all of the hard macros corresponding to a design, the design stitcher must perform the “stitching” of all the hard macros together into a single
implementation. There are several steps involved to create a complete implementation that are listed below:

1. Create module instances (instances of the hard macros) in the design. In some cases, hard macros can describe logic that is simply a routing connection and a special block is instantiated in those situations.

2. Create IOB instances for external input/output.

3. Create clock circuitry (DCMs, BUFGs, etc.).

4. Create nets and populate connections to mirror original design.

5. Clean up un-needed nets and pins.

The first step of the stitcher is to create instantiations of all the hard macros in the design. This is easily done with just a few RapidSmith API calls. However, some of the System Generator blocks do not always generate a hard macro because, by their own nature, they do not contain any memory or logic. These kinds of blocks are called routing blocks as they only contain routing connection information. For example, System Generator has a concatenate block which will convert two signals into a single bus signal. This block only aids conceptually in the design process but does not produce additional hardware. The stitcher has facilities to create an empty hard macro object that would contain all of the wire mappings that occur later when the nets are instantiated. The stitcher automatically identifies these routing blocks and processes them accordingly.

Another special case of a design block are the System Generator gateway blocks used to describe signals leaving and entering the FPGA. From these blocks, IOBs are inferred and created in XDL from a few routines found in the stitcher. IOBs are device family specific and predetermined parameters and attributes are set by default for each family, however, most of the parameters that are user customizable in the gateway block can also change how IOBs are created.
HMFlow only supports single clock domain designs and thus, clock circuitry is quite predictable based on System Generator designs. The stitcher instantiates the appropriate digital clock manager (DCM) based on the family architecture specified in the design and also a BUFG primitive to drive the clock network. In the Virtex 4 series, an additional step is required to avoid the NBTI effect [26] which requires the connection of all unused DCMs to an on-chip oscillator, the PMV primitive. The stitcher handles all of the instantiations and adds the new primitive instances to the design.

Once all of the instances and module instances exist in the design, the nets that interconnect them must be created and connected to mirror the connections in the original design. In order to accomplish this process, each hard macro contains a special mapping object that tells the stitcher how to connect the outputs of each bit of each signal to other blocks in the design. This mechanism is especially handy for routing blocks or any blocks that contain signals that pass through without interacting with any logic. Nets and pins are created and connected to hard macro ports based on the specified mappings. This process completes once all hard macro outputs have been connected.

The final stage iterates through the design and removes any stray pins or dangling nets that are no longer connected due to the net merging process. At this point, the design is completely mapped to the FPGA and could be output to XDL. However, to avoid slow hard disk access times, the design is passed directly to the hard macro placer.

4.2.6 Hard Macro Placer

Since HMFlow uses hard macros instead of FPGA primitives as the objects that are placed onto the FPGA fabric, there is a significant reduction in problem size. Even though the hard macros produced from System Generator design blocks are relatively fine-grained, they still result in a 10-20× reduction of objects requiring placement. However, the algorithm used to perform placement also affects how fast placement will occur. Throughout
the development of HMFlow, three different placers were created, a recursive bi-partitioning placer, a random placer and finally a greedy heuristic placer.

The most commonly used FPGA placement algorithm is simulated annealing [27]. This is due to its ability to obtain high quality implementations (which are often characterized by minimized wire length) at the expense of longer runtimes. In most cases, this is quite desirable because of the need to obtain a high QOR. However with rapid compilation as a goal for HMFlow, different algorithms were tested for their ability to perform placement quickly.

**Recursive Bi-partitioning Placer**

Our first attempt at a fast placer followed the approach of Maidee et. al [28] which used a recursive bi-partitioning algorithm. This involves partitioning the hard macros of a design into two separate groups such that the number of connections between the two groups is minimized. The two groups were created and optimized using the heuristic developed by Kernighan and Lin [29]. By recursively dividing each subgroup in this manner, it would ultimately reduce total wire length of the nets in the design. Our attempts using this implementation did show significant acceleration of the placement process, however, the quality of most of the resulting implementations resulted in clock rates averaging approximately 67 MHz on Virtex 4 parts. This result may be due to our choice of partitioning parameters or the lack of a final simulated annealing step as used in [28].

**Random Placer**

Through various experiments, random placements of several hard macro designs were attempted. The results were surprising in that most of the designs were still route-able and produced similar implementation quality to that of the recursive bi-partitioning placer. The random placer was extremely fast, placing most designs in a fraction of a second. This was
due to the fact that each hard macro was placed only once rather than being swapped several times as in the partitioning placer.

A Greedy Heuristic Placer

The conclusion made from the surprising performance of the random placer is that rapid placement can be achieved by placing each hard macro once, but doing so in a manner that made an attempt at optimizing the placement. This would improve on the quality of the random placement, but still retain the fast execution time of the random placer.

A greedy heuristic was developed to take advantage of the random placer findings. The heuristic was simple in that it depended mostly on the connectivity between hard macros. The heuristic dictates that hard macros are placed one-by-one starting with those containing DSP48 or BRAM primitives as those are the most scarce primitives on the FPGA fabric. Following the DSP/BRAM hard macros, placement priority is given to the largest (greatest number of occupied tiles) hard macros first, as they may be the next most difficult to place.

When a hard macro is placed, the 1st, 2nd, or 3rd most highly connected hard macro to the current hard macro is queried to see if it has already been placed. If so, the current hard macro is placed as close as possible to its highly connected companion. If none of the three hard macros are already placed, the hard macro is placed arbitrarily into one of nine bins designated on the FPGA fabric. This process proceeds until all hard macros are placed. This technique has proven successful on all benchmark designs used when resource utilization is 50% or less of the FPGA. Results comparing the three placement algorithms are deferred to Section 4.3 where the benchmark designs used are described in detail.

4.2.7 Detailed Design Router

The Xilinx router, par, has an option (−p) to only perform the routing of a design, however, if a design is partially routed, those routes are not guaranteed to remain intact. This is a problem if the output of HMFlow were to create a placed hard macro design in
XDL and then converted to NCD for \texttt{par} to route because the hard macros are flattened in the XDL-to-NCD conversion process. This would negate any savings provided by hard macros which contained routed nets. To leverage pre-routed routes from hard macros, a custom, full design router was created using RapidSmith for HMFlow.

Traditionally, the most popular FPGA router algorithm has been PathFinder [30] because of its ability to negotiate routing conflicts in an effective way. However, PathFinder requires several iterations over the nets of a design in order to complete successfully. It is this iterative nature that optimizes circuit speed at the expense of longer runtime. For this reason, a different algorithmic approach was taken for HMFlow.

A simple FPGA routing technique is that of a maze router and is the principle algorithm used in this work. A maze router is fast because it only makes a single pass through all the nets in a design. However, unlike PathFinder, FPGA routing resources are assigned on a first-come, first-served basis which can cause significant routing conflicts given certain scenarios.

To overcome these potential conflicts, the router in HMFlow uses a congestion avoidance technique. When a design is first loaded in the router, it is analyzed for potential conflicts (which are mostly architecture-specific) by looking for hot spot routing switch boxes or certain input pins that have a unique input path. In these cases, the routing resources are reserved for the net which will require them most to complete the route. This technique has been successful in all designs tested when FPGA slice utilization has been 50% or less. Not all designs have successfully routed when utilization exceeds half of the FPGA. Despite this limitation, the router used in HMFlow has proven to be 3-10× faster than the fastest configuration of the Xilinx router (\texttt{par -p}) execution\textsuperscript{3} and allows for preservation of internal hard macro routes. This router is also capable of routing arbitrary designs and can be used for general purposes.

\textsuperscript{3}Because of the limitation by which \texttt{par} can be timed for execution, the performance comparison represents the time it takes to load a design from disk, route the design and then save the routed design to disk for \texttt{par} and the router used in HMFlow. It is expected that if file access times were removed, the HMFlow router would be faster than this estimate.
4.3 Results

This section describes the efforts to ensure adequate benchmark designs that properly illustrate HMFlow’s capability to scale to large designs and large commercial FPGAs. Using these benchmarks, comparison data of the three placement algorithms developed for HMFlow is put forth. Finally, comparison data of the benchmarks implemented with HMFlow and conventional Xilinx tools is presented and analyzed.

All data presented in this section was measured on an HP workstation running Windows XP SP3 32-bit, with an Intel Core 2 Duo (E6850) 3.0 GHz processor and 4GB RAM. All FPGAs tested are from the Xilinx Virtex 4 series with a speed grade of $-10$ and Xilinx ISE ver. 12.3 was used for all experiments. RapidSmith and HMFlow Java implementations used the Oracle JVM ver. 1.6.0_21.

4.3.1 Benchmark Designs

In order to adequately demonstrate the effectiveness of hard macros on FPGA compilation time and their potential for design size scalability, a wide variety of design types and sizes were compiled and used. These designs, outlined in Table 4.4, contain a variety of algorithms, processing cores and data paths. Some of the smaller designs include circuits such as a PicoBlaze, FIR filters, polyphase filters, state machines and 1024-point FFTs/IFFTs. Most of the larger blocks were taken from a very large experimental telemetry receiver design [31] which required three large FPGAs for its implementation. The telemetry receiver design was an excellent source of benchmark circuits on account of its large size and because the design was originally implemented and optimized in System Generator.

As shown in Table 4.4, three different Virtex 4 devices were targeted, the SX35, SX55 and LX200 which is the largest part in the series. The table also shows that the 13 benchmark designs range in size from 150 slices to over 23,000 slices. BRAM and DSP48 primitive counts are representative of the typical circuits used and are present to illustrate that HMFlow and hard macros can be used with heterogeneous types of primitives without
Table 4.4: Benchmark Design Characteristics

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>V4 Part Name</th>
<th>Slices Used</th>
<th>BRAMs Used</th>
<th>DSP48s Used</th>
<th>HMs* / HMIs**</th>
</tr>
</thead>
<tbody>
<tr>
<td>pd_control</td>
<td>SX35</td>
<td>150</td>
<td>1</td>
<td>0</td>
<td>12/21</td>
</tr>
<tr>
<td>polyphaseFilter</td>
<td>SX35</td>
<td>680</td>
<td>8</td>
<td>4</td>
<td>30/79</td>
</tr>
<tr>
<td>aliasingDDC</td>
<td>SX35</td>
<td>806</td>
<td>1</td>
<td>3</td>
<td>25/78</td>
</tr>
<tr>
<td>dualDivider</td>
<td>SX35</td>
<td>1832</td>
<td>0</td>
<td>6</td>
<td>39/542</td>
</tr>
<tr>
<td>computeMetric</td>
<td>SX35</td>
<td>2551</td>
<td>56</td>
<td>40</td>
<td>64/332</td>
</tr>
<tr>
<td>fft1024</td>
<td>SX35</td>
<td>2553</td>
<td>8</td>
<td>12</td>
<td>48/313</td>
</tr>
<tr>
<td>filtersAndFFT</td>
<td>SX35</td>
<td>5203</td>
<td>25</td>
<td>31</td>
<td>92/588</td>
</tr>
<tr>
<td>frequencyEst</td>
<td>SX55</td>
<td>6988</td>
<td>31</td>
<td>72</td>
<td>249/757</td>
</tr>
<tr>
<td>dualFilter</td>
<td>SX55</td>
<td>11173</td>
<td>33</td>
<td>26</td>
<td>93/901</td>
</tr>
<tr>
<td>trellisDecoder</td>
<td>LX200</td>
<td>16973</td>
<td>61</td>
<td>53</td>
<td>196/1328</td>
</tr>
<tr>
<td>filterFFTCM</td>
<td>LX200</td>
<td>18883</td>
<td>81</td>
<td>12</td>
<td>149/920</td>
</tr>
<tr>
<td>multibandCorr</td>
<td>LX200</td>
<td>19732</td>
<td>52</td>
<td>23</td>
<td>90/1472</td>
</tr>
<tr>
<td>signalEst</td>
<td>LX200</td>
<td>23841</td>
<td>126</td>
<td>47</td>
<td>390/1448</td>
</tr>
</tbody>
</table>

**HMs = Unique hard macros in the design  
***HMIs = Total hard macro instances in the design

issue. The final column of Table 4.4 shows the number of unique hard macros the design contains and also the total number of instances of hard macros in the design. The latter number illustrates the number of objects the placer must place whereas the former illustrates the number of hard macros that exist in the cache and must be created. In most cases, hard macros are reused on average between 2 and 10 times illustrating another reuse benefit of hard macros.

**Hard Macro Granularity**

One of the determining factors that measure how well hard macros can accelerate FPGA compilation is their size or granularity and also the percentage of routes that are preserved inside of hard macros. The hard macros used to demonstrate HMFlow 2010 are of a very small granularity, having an average footprint of 1 to 6 tiles on the FPGA fabric. A histogram of the hard macros used in all of the benchmarks is shown in Figure 4.9.
Figure 4.9: A Histogram of Hard Macro Sizes of All Hard Macros in the Benchmarks

Figure 4.10: Graph Showing Percentage of Routed Connections in Each Benchmark Before the Design is Sent to the Router
With smaller hard macros, it is less likely that they contain a significant percentage of routed connections. Figure 4.10 shows the percentage of connections (source to sink pairs) in the benchmarks that are already routed due to routes being included in the hard macros. Note that the percentages do not include connections such as clock, ground or power as these cannot be included in hard macros and are relatively fast to route. Since the granularity of the hard macros in the benchmarks is small, no one design has more than 25% of the connections already routed. As will be seen later in this section, this causes extra work for the router.

**Hard Macro Compilation Time**

This work investigates how fast FPGA compilation can be made to run if all hard macros are readily available at compile time. This is the case with HMFlow when certain kinds of design modifications are made such as re-wiring connections in between hard macros, changing constant values or having compiled the user set of hard macros beforehand. However, it is obvious that a designer is likely to change the design such that one or a few hard macros need to be compiled from one execution of HMFlow to the next and would thus make hard macro compilation time part of the total compilation time of the FPGA design.

As this work leveraged the Xilinx flow to compile the hard macros, it is far from an efficient realization of how fast hard macro compilation could potentially be made to run. The average compile time for a fine-grained hard macro in HMFlow is about 2 minutes. A list of some hard macro compilation times is given in Table 4.5. These hard macro compile times could be avoided by using XDL generators that generate the hard macro on-the-fly. Such generators were demonstrated by Ghosh [32] and illustrate that hard macro generators can produce circuits of similar quality to that produced by the Xilinx tools and do so in a fraction of a second. The issue of hard macro compilation time could thus be eliminated by using such hard macro XDL generators.

58
Table 4.5: Fine-grained Hard Macro Compile Times

<table>
<thead>
<tr>
<th>Hard Macro Name</th>
<th>Tiles Occupied</th>
<th>Compile Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay23</td>
<td>4</td>
<td>116 secs</td>
</tr>
<tr>
<td>Delay14</td>
<td>4</td>
<td>116 secs</td>
</tr>
<tr>
<td>Delay1</td>
<td>4</td>
<td>120 secs</td>
</tr>
<tr>
<td>Negate</td>
<td>10</td>
<td>122 secs</td>
</tr>
<tr>
<td>Inverter</td>
<td>1</td>
<td>136 secs</td>
</tr>
<tr>
<td>Logical3</td>
<td>1</td>
<td>136 secs</td>
</tr>
<tr>
<td>Mux4</td>
<td>1</td>
<td>136 secs</td>
</tr>
<tr>
<td>Delay25</td>
<td>1</td>
<td>137 secs</td>
</tr>
<tr>
<td>Delay33</td>
<td>2</td>
<td>137 secs</td>
</tr>
<tr>
<td>Delay</td>
<td>1</td>
<td>147 secs</td>
</tr>
<tr>
<td>Mux3</td>
<td>4</td>
<td>148 secs</td>
</tr>
</tbody>
</table>

4.3.2 RapidSmith Router Performance

The RapidSmith router used in HMFlow is a full design router, capable of routing arbitrary designs in XDL and is not specific to designs containing hard macros. Because of this attribute, it makes it reasonable to compare the router against the Xilinx router in par. To measure the performance of the RapidSmith router, the benchmark circuits were placed and then flattened (all references to hard macros were removed with circuitry remaining intact) and exported to an XDL file. The designs were flattened for a more fair comparison between the RapidSmith router and Xilinx router because it takes a significant amount of additional time for the Xilinx tools to load designs that contain several hard macros. The execution times in Table 4.6 represent the time it took each router to load the design implementation, route it, and then save it to a file.

As can been seen from Table 4.6, the RapidSmith router runs over $3 \times$ to almost $10 \times$ faster than the Xilinx router in par. The smaller designs ran the fastest because of the lightweight RapidSmith device files mentioned in Section 3.4.1 and the execution penalty mentioned earlier in this section for par. The execution parameters for par were set to route only, all effort levels were set to standard (fastest run time) and to use the -x option which ignores any given timing constraints and generates new ones suitable for the circuit.
Table 4.6: Router Performance Comparison: Xilinx vs. RapidSmith

<table>
<thead>
<tr>
<th>Design Name</th>
<th>Xilinx Router Runtime</th>
<th>RapidSmith Router Runtime</th>
<th>Speedup over Xilinx</th>
</tr>
</thead>
<tbody>
<tr>
<td>pd_control</td>
<td>14.1 secs</td>
<td>1.4 secs</td>
<td>9.9x</td>
</tr>
<tr>
<td>polyphaseFilter</td>
<td>16.6 secs</td>
<td>2.6 secs</td>
<td>6.5x</td>
</tr>
<tr>
<td>aliasingDDC</td>
<td>16.6 secs</td>
<td>2.7 secs</td>
<td>6.3x</td>
</tr>
<tr>
<td>dualDivider</td>
<td>22.9 secs</td>
<td>4.0 secs</td>
<td>5.8x</td>
</tr>
<tr>
<td>computeMetric</td>
<td>34.0 secs</td>
<td>8.0 secs</td>
<td>4.2x</td>
</tr>
<tr>
<td>fft1024</td>
<td>29.7 secs</td>
<td>6.5 secs</td>
<td>4.5x</td>
</tr>
<tr>
<td>filtersAndFFT</td>
<td>47.2 secs</td>
<td>14.8 secs</td>
<td>3.2x</td>
</tr>
<tr>
<td>frequencyEst</td>
<td>89.5 secs</td>
<td>20.3 secs</td>
<td>4.4x</td>
</tr>
<tr>
<td>dualFilter</td>
<td>196.2 secs</td>
<td>38.8 secs</td>
<td>5.1x</td>
</tr>
<tr>
<td>trellisDecoder</td>
<td>210.3 secs</td>
<td>59.2 secs</td>
<td>3.5x</td>
</tr>
<tr>
<td>filterFFTCM</td>
<td>452.0 secs</td>
<td>76.7 secs</td>
<td>5.9x</td>
</tr>
<tr>
<td>multibandCorr</td>
<td>312.3 secs</td>
<td>82.2 secs</td>
<td>3.8x</td>
</tr>
<tr>
<td>signalEst</td>
<td>516.4 secs</td>
<td>107.9 secs</td>
<td>4.8x</td>
</tr>
</tbody>
</table>

Table 4.7: Hard Macro Placer Algorithm Comparison

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Runtime (seconds)</th>
<th>Clock Rate (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>REC</td>
<td>RAND</td>
</tr>
<tr>
<td>pd_control</td>
<td>0.266</td>
<td>0.047</td>
</tr>
<tr>
<td>polyphaseFilter</td>
<td>3.281</td>
<td>0.063</td>
</tr>
<tr>
<td>aliasingDDC</td>
<td>3.234</td>
<td>0.047</td>
</tr>
<tr>
<td>computeMetric</td>
<td>15.86</td>
<td>0.157</td>
</tr>
<tr>
<td>fft1024</td>
<td>11.703</td>
<td>0.094</td>
</tr>
<tr>
<td>frequencyEst</td>
<td>29.156</td>
<td>0.391</td>
</tr>
<tr>
<td>filterFFTCM</td>
<td>203.9</td>
<td>2.765</td>
</tr>
<tr>
<td>Averages</td>
<td>38.2</td>
<td>0.51</td>
</tr>
</tbody>
</table>

REC: Recursive bi-partitioning placer, RAND: Random placer, FAST: Fast heuristic placer

4.3.3 Hard Macro Placer Algorithms

As mentioned in Section 4.2.6, three placers were created during the development of HMFlow: a recursive bi-partitioning placer (REC), a randomplacer (RAND) and a fast heuristic placer (FAST). Only a limited set of benchmarks were completed when the REC and RAND placer were developed and tested, therefore comparisons between the three placers are made only on that subset of benchmark designs. Each benchmark design was placed
three times, once for each placer as listed in Table 4.7. Clock rates were obtained after each placement was routed with the HMFlow router and measured with the Xilinx tool, trace.

As can be seen from the listed averages at the bottom of Table 4.7, the FAST placer had the fastest execution of the three placers and produced, on average, the highest quality circuit\(^4\). These results show that the custom heuristic designed for hard macro placement is capable of producing higher quality results than the RAND and REC placer, yet, still execute with speed similar to that of the RAND placer.

### 4.3.4 HMFlow Performance

After the complete realization of HMFlow, all 13 benchmark circuits were implemented with both the Xilinx tools and HMFlow. Since the benchmark circuits were created as Xilinx System Generator designs, the Xilinx tools runtime included the time elapsed during the following steps:

- System Generator NGC generation (outputs netlist)
- NGDBuild (outputs NGD)
- map (outputs mapped circuit in NCD format)
- par (outputs placed and routed circuit)

Some steps of the Xilinx flow allow execution to be optimized to reduce overall runtime. Therefore, options such as overall-effort-level in map and par were set to their lowest setting to ensure that the comparison is made with the fastest execution of the Xilinx tools. Although theoretically possible, bitstream creation (.NCD to .BIT) could not be legally accelerated by HMFlow due to licensing issues, thus, bitgen execution time is omitted as it would be identical for either of the design flows.

\(^4\)It should be noted that after the prototype RAND placer was implemented, the FAST placer benefited significantly from optimization and is shown to be the fastest. It is very likely that the RAND placer could have benefited from some optimization and could have run as fast if not faster than the FAST placer.
Table 4.8: Runtime Performance of HMFlow and Comparison to Xilinx Flow

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Simulink Runtime</th>
<th>Mapper / Design Runtime</th>
<th>Hard Macro Runtime</th>
<th>Design Runtime</th>
<th>XDL Export Runtime</th>
<th>HMFlow Runtime</th>
<th>Xilinx Runtime</th>
<th>HMFlow Speedup</th>
<th>XDL to NCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>pd_control</td>
<td>0.093s</td>
<td>0.735s</td>
<td>0.187s</td>
<td>0.016s</td>
<td>0.219s</td>
<td>0.062s</td>
<td>1.3s</td>
<td>65.6s</td>
<td>50×</td>
</tr>
<tr>
<td>polyphaseFilter</td>
<td>0.094s</td>
<td>0.75s</td>
<td>0.219s</td>
<td>0.015s</td>
<td>1.406s</td>
<td>0.11s</td>
<td>2.6s</td>
<td>60.3s</td>
<td>23.2×</td>
</tr>
<tr>
<td>aliasingDDC</td>
<td>0.11s</td>
<td>0.765s</td>
<td>0.219s</td>
<td>0.016s</td>
<td>1.453s</td>
<td>0.125s</td>
<td>2.7s</td>
<td>62.2s</td>
<td>23.1×</td>
</tr>
<tr>
<td>dualDivider</td>
<td>0.313s</td>
<td>0.89s</td>
<td>0.203s</td>
<td>0.047s</td>
<td>2.407s</td>
<td>0.218s</td>
<td>4.1s</td>
<td>96.6s</td>
<td>23.7×</td>
</tr>
<tr>
<td>computeMetric</td>
<td>0.281s</td>
<td>0.891s</td>
<td>0.641s</td>
<td>0.047s</td>
<td>6.359s</td>
<td>0.609s</td>
<td>8.8s</td>
<td>160.8s</td>
<td>18.2×</td>
</tr>
<tr>
<td>fft1024</td>
<td>0.235s</td>
<td>0.937s</td>
<td>0.297s</td>
<td>0.047s</td>
<td>4.953s</td>
<td>0.375s</td>
<td>6.8s</td>
<td>119.3s</td>
<td>17.4×</td>
</tr>
<tr>
<td>filtersAndFFT</td>
<td>0.328s</td>
<td>0.984s</td>
<td>0.797s</td>
<td>0.188s</td>
<td>12.312s</td>
<td>0.75s</td>
<td>15.4s</td>
<td>254.1s</td>
<td>16.5×</td>
</tr>
<tr>
<td>frequencyEst</td>
<td>0.437s</td>
<td>1.5s</td>
<td>0.578s</td>
<td>0.219s</td>
<td>18.11s</td>
<td>1.171s</td>
<td>22s</td>
<td>373.5s</td>
<td>17×</td>
</tr>
<tr>
<td>dualFilter</td>
<td>0.469s</td>
<td>1.313s</td>
<td>1.203s</td>
<td>0.437s</td>
<td>34.672s</td>
<td>1.656s</td>
<td>39.8s</td>
<td>469s</td>
<td>11.8×</td>
</tr>
<tr>
<td>trellisDecoder</td>
<td>0.656s</td>
<td>1.719s</td>
<td>1.422s</td>
<td>0.547s</td>
<td>54.015s</td>
<td>2.5s</td>
<td>60.9s</td>
<td>824.6s</td>
<td>13.5×</td>
</tr>
<tr>
<td>filterFFTCM</td>
<td>0.516s</td>
<td>1.937s</td>
<td>1.641s</td>
<td>0.984s</td>
<td>69.933s</td>
<td>3.046s</td>
<td>78.1s</td>
<td>1021.3s</td>
<td>13.1×</td>
</tr>
<tr>
<td>multibandCorr</td>
<td>0.828s</td>
<td>1.797s</td>
<td>1.844s</td>
<td>1.859s</td>
<td>73.297s</td>
<td>5.781s</td>
<td>85.4s</td>
<td>786.2s</td>
<td>9.2×</td>
</tr>
<tr>
<td>signalEst</td>
<td>0.843s</td>
<td>2.328s</td>
<td>2.157s</td>
<td>1.531s</td>
<td>107.547s</td>
<td>15.375s</td>
<td>129.8s</td>
<td>1508.7s</td>
<td>11.6×</td>
</tr>
</tbody>
</table>

1We define HMFlow speedup as the time it takes Xilinx to create a placed and routed implementation divided by the time it takes HMFlow to create a placed and routed implementation from the same System Generator design.
The total runtime of HMFlow is measured as the sum of time elapsed during the following steps:

- Simulink Parser (reads/parses System Generator design)
- Mapper/Hard Macro Cache (maps/retrieves hard macros)
- Design Stitcher (combines hard macros together)
- Hard Macro Placer (places all hard macros)
- Design Router (routes all un-routed nets)
- XDL Export (outputs placed and routed XDL design file)

Table 4.8 contains the runtimes for the individual steps in HMFlow, the total runtimes of HMFlow and Xilinx tools and the speedup of HMFlow over the Xilinx tools. A preliminary observation is that the speedup obtained by HMFlow is much greater (23-50×) for the smaller designs than the larger designs (9.2-13.5×). There are a few possible causes for this phenomenon. First, the Xilinx tools write out intermediate design files between each step of the design process and must also load device database files for each step. This is in contrast to HMFlow which loads design information once (during Simulink parsing and hard macro cache accesses) and loads the RapidSmith device files once for the entire execution of HMFlow. Therefore, the Xilinx tools spend a disproportionate amount of time reading/writing files for the smaller designs. A second possible cause for the faster speedup of smaller designs could be attributed to the algorithms used in the Xilinx tools. As Xilinx must accommodate very large designs on very large parts, their algorithms might be tuned for larger designs and thus, perform better. However, our suspicions lie with the former explanation rather than the latter.

If the runtimes of all benchmarks are averaged, Figure 4.11a shows a pie chart illustrating the runtime distribution of HMFlow. As can be seen from Figure 4.11a, the majority
of the time is spent in the router. This illustrates the speed at which all the other steps in HMFlow operate. A completely placed implementation can be obtained from HMFlow in a matter of seconds for even the largest benchmarks. Although the RapidSmith router runs $3-10 \times$ faster than the Xilinx router, it is still the most time consuming process in HMFlow. This is likely because of the granularity of hard macros used in the benchmark designs. The hard macros do not contain a significant percentage of all nets in the final design, most of the nets exist between the external ports of the hard macros. For this reason, the router must route almost all of the nets in the design in the final step of HMFlow. Chapter 5 details an improved version of HMFlow and will concentrate on creating larger, more routing-dense hard macros to help alleviate this problem.

One challenging issue with HMFlow and the Xilinx tools is the time it takes to create an NCD file from an XDL file. The NCD and XDL file formats can represent all of the necessary details to describe an implementation, however, XDL must be converted to NCD in order to create a bitstream as bitgen only accepts NCD files as input. The far right column of Table 4.8 lists the runtime to generate an NCD file for the resulting HMFlow benchmark implementation. As designs get larger, the conversion time escalates.
Figure 4.11b shows the average runtime distribution if the runtime of HMFlow and XDL conversion are combined. This challenge is specifically associated with the Xilinx tools, not the use of hard macros or HMFlow. The long conversion times could easily be overcome if appropriate Xilinx proprietary knowledge were obtained to build a more efficient conversion tool or to use a different FPGA vendor that would provide the necessary native netlist access.

Figure 4.12: A Comparison Plot of the Benchmark Circuits Maximum Clock Rates When Implemented with HMFlow and the Xilinx Tools

In order to achieve the dramatic reduction in runtime, HMFlow trades implementation clock rates for faster runtime as shown in Figure 4.12. On average, HMFlow produces implementations that are 2-4× slower than those created by the Xilinx tools\(^5\). For rapid prototyping purposes, this tradeoff may be acceptable because even though the design will run 2-4× slower, overall, it will be executing 10,000s of times faster than simulation.

\(^5\)The reader is reminded that the Xilinx tools are likely to have produced a higher quality circuit, however, the results shown in Figure 4.12 represent the fastest runtime configuration of the Xilinx tools.
4.4 Conclusion

HMFlow has demonstrated that rapid compilation of FPGA designs using hard macros is possible. It has been shown that pre-compiled blocks can be used to construct functional and correct designs an order of magnitude faster than conventional vendor tools. Although QOR is traded for faster compilation time, this technique paves the way for a completely new approach to FPGA compilation that will have the potential to change the way engineers design and debug projects on FPGAs.
CHAPTER 5. HMFLOW 2011: ACCELERATING FPGA COMPILATION AND MAINTAINING HIGH PERFORMANCE IMPLEMENTATIONS

After the success of HMFlow 2010 in achieving rapid compilation for FPGAs, it was time to take a step back and consider alternative approaches. Of particular interest was exploring ways of maintaining rapid compilation, but also improving HMFlow implementation clock rates to have speeds comparable to those produced by vendor tools. The major approach taken in HMFlow 2011 was to (as hinted at in Section 4.3.4) increase the size of the hard macros used in the flow. That is, rather than use fine-grained hard macros such as adders, registers, multiplexers, etc., the approach would use system-level sized blocks such as FIR filters, FFTs, DDS/mixers, micro-controllers and so forth.

These larger hard macros benefit HMFlow in two major ways. First, larger hard macros contain a much higher percentage of the total routes in the design. This allows timing closure information (the timing-sensitive routing information) to be stored in the form of routed nets that can be re-used with each instantiation of the hard macro. In addition, it significantly reduces routing time. As an example, the frequency estimator benchmark used to demonstrate HMFlow 2010 has 11,226 un-routed nets (66% of total nets) before the design reached the router. However, a large hard macro version of the frequency estimator benchmark, has only 854 un-routed nets (5% of total nets) before the design reached the router. This resulted in a significant reduction in total runtime for the HMFlow 2011 router as the router only had to route the last 5% of nets (rather than 66%) in the design to complete the implementation.

The second main advantage of larger hard macros is that the larger hard macro-based designs will typically absorb many of the smaller hard macros prevalent in the benchmarks used in HMFlow 2010. This substantially reduces the total number of objects requiring
placement during the placement process, ultimately reducing the problem size. For instance, the frequency estimator benchmark used to demonstrate HMFlow 2010, requires 757 hard macro instances to be placed. However, a large hard macro version of the benchmark only has 31 hard macro instances to be placed, a 24\times reduction in problem size.

This chapter describes the preliminary work performed to validate the ideas of using large hard macros and their potential to be the major building block of a high performance, rapid compilation flow. Once large hard macros for high performance are validated, a modified HMFlow 2010 (HMFlow 2010a) is augmented to support large hard macros and a newer Virtex 5 architecture. This sets the stage for HMFlow 2011 with a number of new improvements and techniques that are described and implemented. Finally, several further improvements are made on the new placer in HMFlow 2011 to help increase robustness and finalize HMFlow 2011.

5.1 Preliminary Work

Two major questions needed to be answered if large hard macros could be feasible for both creation and design implementation. They are:

1. If a large hard macro is created and timing verified to run at clock rate X at one location, will it be able to run at the same or similar rate at another location on the FPGA fabric?

2. What is the best way to create a hard macro in terms of size, shape and density?

Question #1 is simply a question to verify if the FPGA fabric can be relied upon to provide consistent timing behavior of a hard macro regardless of where it might be placed. This is important because when a design is implemented with HMFlow 2011, its clock rate is determined not only by the critical path in between hard macros, but the the longest critical path within the hard macros themselves. As each hard macro is created, it’s longest critical path is calculated for the location on which it was created. Question #1 aims to
discover how accurate the critical path measurement is when the hard macro is placed at other locations on the FPGA.

Question #2 aims to reveal information that ultimately improves the performance of HMFlow, specifically, how to increase the place-ability (maximizing the number of valid locations a hard macro can be placed on the FPGA) of hard macros and how to reduce their critical paths. Increasing the place-ability of a hard macro depends largely upon its shape and size. Smaller, more uniformly sized hard macros have many locations on which they can be placed. However, large hard macros, especially oddly-shaped ones, have fewer valid locations on which they can be placed. Also, at this point in the research, it was not well understood as to what parameters affected the actual performance or critical path of a hard macro.

5.1.1 Experiments

To answer the questions presented in the previous section, a series of experiments were conducted.

Experiment #1: Timing Variability of Hard Macros on FPGA Fabric

Experiment #1 is designed to answer Question #1 as presented in the previous section. That is, it aims to measure the timing variability of hard macros as they are placed at different locations across the FPGA fabric. To do this, two different hard macros containing a reasonable set of internal routes were created. The first hard macro created was a 21×21 bit LUT-based multiplier on a Virtex 4 SX35 FPGA. Relative to the FPGA fabric size, the hard macro was small, but did contain several timing sensitive routing paths that could be accurately measured and variations in its timing behavior could easily be detected.

To measure the variability of the FPGA fabric, a separate implementation containing a unique placement for each valid location of the hard macro was created. In this case, the multiplier could be validly placed at 400 different locations on the Virtex 4 SX35. These
implementations were produced by writing a custom program in RapidSmith that would iteratively find each valid location at which the multiplier hard macro could be placed, create a unique implementation with the hard macro placed at that location and then save it to an XDL file. At the end, there were 400 XDL files, each with a single instance of the multiplier hard macro placed at a unique location.

The placement locations formed a grid pattern on the FPGA fabric as demonstrated by Figure 5.1. Each shaded rectangle in the grid represents a single implementation of the hard macro placed there with every rectangle being populated in a separate implementation. For each unique placement of the hard macro, the timing results of each placement populated each measurement in the grid. For each of the 400 implementations, the XDL file was converted to NCD and then provided as input to Xilinx Trace (\texttt{trce}) to calculate the minimum clock period at which the multiplier could be clocked and delay values for all of the paths within the multiplier. All of the 400 \texttt{trce} report files (.TWR) were analyzed and two types of deviations in timing were found. The first is represented in Figure 5.2 as a particular path’s delay in the multiplier hard macro is plotted in a grid with the Z axis representing the amount of delay (in nanoseconds).

Note the regular sharp hills that span the Y direction, these correlate very well with the clock branches found at those locations on the FPGA (the graph is rotated about 100° counter-clockwise from a normal X/Y perspective). These deviations, however, are quite
Figure 5.2: Delay of a Path Within a 21×21 Bit LUT-multiplier Hard Macro Placed in a Grid of 400 Locations on a Virtex 4 SX35 FPGA

minimal (≈ 10 ps) and are caused by the hard macro straddling a clock distribution branch in the FPGA fabric. This deviation is quite small and very acceptable for the purposes of HMFlow 2011.

However, a second type of timing deviation was found among the trace reports and is shown in Figure 5.3 (rotated about 45° counter clockwise). Here the timing deviation is much more significant (≈ 250 ps) and is due to the multiplier hard macro straddling the center column or clock distribution spine of the FPGA. If examined closely, the same ridges present in Figure 5.2 can be seen, although they are dwarfed by the major ridge down the center column of the FPGA. The delay measured in this grouping of paths is more problematic but can easily be avoided by simply disallowing any hard macro to be placed such that it straddles the center column of the chip.

71
Figure 5.3: A More Severely Impacted Path Caused by a Hard Macro Straddling the Center Clock Tree Spine of the FPGA

An additional experiment was performed to verify that this behavior holds true for the Virtex 5 series of FPGAs. A Xilinx PicoBlaze (Ken Chapman’s programmable state machine) hard macro was created for the Virtex 5 SX240T FPGA and placed in 3700 locations. Each implementation was created as with the multiplier and all 3700 trace reports were analyzed. The critical path delay for each placement is plotted in Figure 5.4. In this scenario, the hard macro was not allowed to straddle the center column of the FPGA to avoid the significant timing variation experienced with the multiplier.

The PicoBlaze had the same behavior as the path shown in Figure 5.2 (the Z axis is critical path delay in nanoseconds). That is, the timing deviation was minimal (∼10 ps) and was caused by the clock distribution branches found at regular intervals in the FPGA fabric. Note that as with Figure 5.2, the plot has been rotated about 100° counter-clockwise from a normal X/Y perspective.
Figure 5.4: A PicoBlaze Hard Macro Placed at 3700 Locations on a Virtex 5 FPGA

**Experiment #2: Optimal Hard Macro Creation Parameters**

The most important aspect of hard macro creation is the area constraint applied to it during its placement phase. In fact, there are actually multiple area constraints that are often applied when creating hard macros as the logic will often contain slices, DSP48s and/or block RAMs. Each primitive type requires its own area group constraint and calculating their size and shape can affect the hard macro’s place-ability.

Experiment #2 is designed to answer Question #2 as described in the previous section. It aims to find out what are the optimal parameters in creating hard macro area constraints such that place-ability and clock rate are maximized. In order to do this, large hard macros were selected and several thousand runs of the Xilinx tools were executed, each with a slightly different configuration of area constraint and timing constraint. Specifically there were 4 different hard macro designs, a 1024 point FFT, 18×18 bit LUT-based multiplier, a PicoBlaze and a MicroBlaze.

To vary the area group constraints, a width/height metric was used as area group constraints are generally rectangular. Width and height test values were calculated in terms
of slice coordinates ranging from 1 to 5 in both dimensions. Table 5.1 shows all of the test configurations for the area group aspect ratio parameter (duplicate ratios are denoted by ‘-’).

<table>
<thead>
<tr>
<th>Height</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1/1</td>
</tr>
<tr>
<td>2</td>
<td>1/2</td>
</tr>
<tr>
<td>3</td>
<td>1/3</td>
</tr>
<tr>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>5</td>
<td>1/5</td>
</tr>
</tbody>
</table>

In addition, a density metric that determined how loose or tight the area constraint should be. This was calculated as a function of the xst synthesis report utilization estimates for the hard macro design of interest. This metric only applied to slices as DSP48 and block RAM counts are always estimated accurately for the hard macros tested. For example, if xst reported that a design required 100 slices, a density metric of 50% overhead would produce an area constraint that contained at least 150 slices. A density metric of 0% overhead produced an area constraint that contained at least 100 slices. In some situations, the shape of the area constraint often would produce an area constraint slightly larger (by just a few slices) than what was specified by the density metric. For this parameter, overhead values ranged from 0 to 150% overhead, in increments of 10%.

One additional parameter was varied in this large set of Xilinx tool runs, the minimum clock period constraint. All of the 4 hard macros tested were implemented in a single clock domain and required a clock period timing constraint. In each unique parameter instance, 10 different timing constraints were run 100 ps apart, starting with a constraint where the tools could not achieve the constraint and then lengthening the constraint 10 times. The constraints were in the range of 4 to 5 ns but were design dependent.

Ultimately with the 4 designs and three parameters, several thousand runs of the Xilinx tools (ngdbuild, map, par, and trce) were executed to obtain the timing perfor-
mance of each implementation. The results of these runs will be presented in a future thesis by Jaren Lamprecht at Brigham Young University in 2012\(^1\). Ultimately, no correlation was found between the area group aspect ratio and density metric that inhibited the Xilinx tools from achieving timing closure. Therefore, hard macro area constraints can be generated to maximize their place-ability without regard for trying to meet timing closure. This outcome provided significant flexibility to the hard macro creation process as area constraints could be optimized for a single parameter rather than two opposing parameters.

5.1.2 Conclusions on Preliminary Work

After completing the experiments, the questions put forth at the beginning of this section can now be answered. That is, will hard macros be able to run at the same or similar clock rates if placed at different locations? The answer is yes. Timing variation is quite minimal (~ 10 ps) on both Virtex 4 and Virtex 5 FPGAs as long as the hard macro does not straddle the center column clock spine of the chip.

To answer question #2, what is the best way to create a hard macro in terms of size, shape and density? From the experiments conducted, it was learned that the aspect ratio and density had little to no effect on the timing closure capability of a hard macro-bound design. Therefore, the area aspect ratio and density can be chosen for optimal place-ability. Ultimately, the density constraint found to be best was 50% and an aspect ratio of 1/1 (width=length).

5.2 Comparison of HMFlow Using Large Hard Macros vs. Small Hard Macros

After determining that large hard macros are a feasible and an effective approach of minimizing compile time while increasing circuit quality, a number of benchmark designs containing several large hard macros were needed. In addition, minor changes would be required to HMFlow in order to support larger hard macros in designs and the methods of

\(^1\)The reader is directed to Jaren Lamprecht’s thesis for a full detailed description and analysis of this work. Only summarizing statements of the outcomes are presented here.
indicating how they should be created. Virtex 5 would be the FPGA platform of target for 2011 and thus, additional upgrades to HMFlow would be required to handle the additional FPGA architecture.

Ultimately, six large designs were chosen and re-fashioned from the benchmarks used to demonstrate HMFlow 2010. Three of these designs were the same logic as their small hard macro counterpart, however, implemented with large hard macros rather than small ones. These benchmarks were tested to measure the impact of using large hard macros against small ones in the areas of runtime and quality of result.

\subsection*{5.2.1 Upgrading HMFlow to Support Large Hard Macros}

The design theory behind implementing large hard macros is that it would enable a designer to encapsulate a building block of his or her design into a reasonably finished module. That is, the designer could use the small hard macro paradigm to develop, debug and verify the module. Then, once the block was satisfactorily correct and verified, it could be tagged to become a large hard macro so that it could be compiled for high quality.

In order to allow the designer this capability, the Subsystem construct found in the MathWorks Simulink environment was leveraged. Simulink is the environment used to implement Xilinx System Generator designs and recognizes subsystems as a valid design construct. For example, consider Figure 5.5a, which shows a FIR filter constructed in System Generator. In HMFlow 2010, each block is converted into a hard macro, in this case, there would be 22 hard macros created (the 4 constant blocks actually just become routing). However, the entire FIR filter can be encapsulated inside of a subsystem as shown in Figure 5.5b, which is bound to become a single large hard macro.

The changes in HMFlow were actually quite minimal in order to support subsystems. The first change occurred in the Simulink parser which had to identify top-level subsystems so that they could be converted to a single hard macro. Only top-level subsystems or subsystems at the highest level of hierarchy were recognized for hard macro creation because
Figure 5.5: (a) Illustrates a FIR Filter Implemented in System Generator (b) Shows the FIR Filter Converted to a Subsystem to be Turned into a Hard Macro
Simulink allows for multiple subsystem hierarchy. This change in the parser required slight changes to the Simulink data structure and hard macro creation process but were still fairly minimal. All of these changes were implemented, tested and verified before any other changes were made to HMFlow.

5.2.2 Upgrading HMFlow to Support Virtex 5 FPGAs

As with all research, the technology on which it was performed quickly becomes obsolete. To try and minimize the obsolescence of the technology used to demonstrate HMFlow, it was upgraded to support Virtex 5 FPGAs in addition to Virtex 4. This required a number of changes to HMFlow because of the differences in primitive types such as IOBs and slices. Some of the changes that were made are summarized as follows:

- **Hard Macro Creation**
  - Port hard macro identification had to change to accommodate Virtex 5 slice differences
  - Synthesis reports output statistics in a different format for Virtex 5 so parsing facilities had to change
  - The layout and LUT counts of Virtex 5 slices compared to Virtex 4 slices are different and require a separate equation for area constraint calculation

- **Stitcher**
  - The IOB creation had to be modified to accommodate changes to IOBs in Virtex 5 primitives
  - Clock generation circuitry was modified to support Virtex 5 architectures
  - Virtex 5 did not require instantiation of the PMV primitive as with Virtex 4 designs
• Router

- The handling of static sourced nets (VCC and GND) had to be treated differently due to the significant change in routing architecture

- RapidSmith had to include missing information for pin name mapping on some primitives (see Section 3.4.2)

With the additional features of (1) large hard macro support through subsystems and (2) Virtex 5 support in HMFlow, it has changed significantly enough to differentiate it from HMFlow 2010 as described in Chapter 4. The modified HMFlow 2010 with these two additional features will be referred to as HMFlow 2010a throughout the remainder of this dissertation. It should be emphasized that all of the steps and algorithms used in HMFlow 2010 are the same as those used in HMFlow 2010a.

5.2.3 Large Hard Macro Benchmark Designs for HMFlow

Of the six benchmark designs selected for demonstrating HMFlow for use with large hard macros, three were logically equivalent to three of the original benchmarks used in demonstrating HMFlow 2010, namely frequencyEstimator, multibandCorrelator and trellisDecoder. The other three benchmarks (brik1, brik2 and brik3) each represented an entire design that occupied an entire FPGA in the original telemetry receiver [31] selected as a major source for benchmark circuits.

Large hard macros were created in each benchmark by encapsulating major components into subsystems, often by also inserting pipeline registers at their boundaries. After all the subsystems were finalized, all six benchmarks were built using HMFlow 2010a to support large hard macros and targeted the Virtex 5 SX240T FPGA. The slice count of the six benchmark designs is shown in Table 5.2. It should also be noted that the three non-brik benchmarks actually existed as a single large hard macro inside one of the brik benchmarks. The frequencyEstimator became a single hard macro in brik2, the trellisDecoder became
Table 5.2: Slice Counts for Large Hard Macro Benchmark Virtex 5 Designs

<table>
<thead>
<tr>
<th>Design Name</th>
<th>Virtex 5 Slices</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency estimator</td>
<td>3970</td>
</tr>
<tr>
<td>trellis decoder</td>
<td>5929</td>
</tr>
<tr>
<td>brik3</td>
<td>7505</td>
</tr>
<tr>
<td>brik2</td>
<td>7552</td>
</tr>
<tr>
<td>multiband correlator</td>
<td>8258</td>
</tr>
<tr>
<td>brik1</td>
<td>9598</td>
</tr>
</tbody>
</table>

a single hard macro in brik3 and the multiband correlator became a single hard macro in brik1.

5.2.4 Comparisons of Large and Small Hard Macro-based Designs

To measure the impact of larger hard macros on HMFlow, a comparison was to be made between the small hard macro benchmarks and the large hard macro benchmarks. Both sets of benchmarks were built with the newly created HMFlow 2010a and targeted the Virtex 5 SX240T FPGA. This would provide a fair comparison of runtime and quality of result when using large hard macros vs. small hard macros. The runtimes for the benchmarks and their maximum achievable clock rates are graphed in Figure 5.6.

As can be seen from Figure 5.6a, using larger hard macros over smaller hard macros can reduce compilation times significantly, by at least 2.5–4×. This is a very significant reduction in that the smaller hard macro versions compile at least 10× faster than the fastest Xilinx compilations. This shows very positive implications for using larger hard macros as it validates the previous conjectures on such reductions.

Figure 5.6b shows that using larger hard macros will improve the maximum achievable clock rate by 60–70%. This is an excellent result as no algorithms were changed in HMFlow 2010a to achieve this result. The fact that more routes were being packed into the hard macros and being routed with the higher quality Xilinx router made a significant difference in overall clock rate.
Some of the benefit attributed to the larger hard macros is the increased number of routed nets encapsulated within the hard macros. The large hard macro benchmarks include over $3.7 \times$ the routed connections as found on average in the smaller hard macro benchmarks as described in Section 4.3.1. This removes a significant burden from off the router as the large hard macro benchmarks have over 60% of their connections already routed (not including clock, power and ground nets) before arriving at the routing stage. The percentage of routed connections in each benchmark is shown in Figure 5.7.

**Large Hard Macro Compilation Time**

As mentioned in Section 4.3.1, this work aims to find out how fast FPGA compilation can be made to run assuming that hard macros are already built. Large hard macros generally take longer to compile than small hard macros and as large hard macros contain more routing, the hard macros perform better if they are created to run at faster clock rates.
Figure 5.7: The Number of Existing Routed Connections in the Large Hard Macro Benchmark as a Percentage of Total Connections

Although large hard macro compile times are longer than small hard macro compile times and cannot be solved as easily with XDL generators as described in [32], they do offer the capability to capture timing closure information. By capturing the routing configuration that meets a difficult timing constraint that might take the Xilinx router several hours or days to generate in a hard macro, that timing closure process does not need to be repeated. Each subsequent bug fix found in other parts of the design will not necessitate the long and difficult compilation process that would be needed to re-achieve timing closure because such
Table 5.3: Coarse-grained Hard Macro Compile Times

<table>
<thead>
<tr>
<th>Hard Macro Name</th>
<th>Tiles Occupied</th>
<th>Compile Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>digital_gain.amplifier</td>
<td>62</td>
<td>160s</td>
</tr>
<tr>
<td>new_correlator_fifo</td>
<td>110</td>
<td>168s</td>
</tr>
<tr>
<td>ad_dynamic_range_calc</td>
<td>34</td>
<td>177s</td>
</tr>
<tr>
<td>Polyphase_filter</td>
<td>610</td>
<td>200s</td>
</tr>
<tr>
<td>pd_control</td>
<td>117</td>
<td>230s</td>
</tr>
<tr>
<td>Reindexer</td>
<td>645</td>
<td>260s</td>
</tr>
<tr>
<td>Aliasing_DDC</td>
<td>491</td>
<td>320s</td>
</tr>
<tr>
<td>multi_band_correlator1</td>
<td>12672</td>
<td>1324s</td>
</tr>
</tbody>
</table>

Table 5.4: Runtime Comparison of HMFlow with Large Hard Macros vs. Xilinx

<table>
<thead>
<tr>
<th></th>
<th>frequency_est.</th>
<th>trellis_dec.</th>
<th>brik3</th>
<th>brik2</th>
<th>multiband_cor.</th>
<th>brik1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx</td>
<td>392s</td>
<td>461s</td>
<td>605s</td>
<td>852s</td>
<td>498s</td>
<td>849s</td>
</tr>
<tr>
<td>HMFlow</td>
<td>7.0s</td>
<td>10.9s</td>
<td>12.9s</td>
<td>13.1s</td>
<td>11.8s</td>
<td>14.2s</td>
</tr>
<tr>
<td>Speedup</td>
<td>56×</td>
<td>42×</td>
<td>47×</td>
<td>65×</td>
<td>42×</td>
<td>60×</td>
</tr>
</tbody>
</table>

information is already captured in the hard macro (obviously if a bug exists in the hard macro in question, it must be recompiled).

One could potentially envision a development process where a hybrid approach of using both small and large hard macros to design a block. A designer could start by using small hard macros to quickly converge on a functional circuit. Then, once the block is debugged and verified, it could be collapsed into a single large hard macro and compiled to meet the difficult timing constraint. The time it takes to compile the design as a large hard macro will take longer, but will avoid subsequent compilations where timing closure is difficult to reach.

5.2.5 Comparisons of Large Hard Macros with HMFlow vs. Xilinx

With large hard macros making a positive impact in performance and quality of HMFlow implementations, it begs the question, how close is this combination to the implementations produced by Xilinx? To answer this question, the remaining three benchmark designs (brik1, brik2 and brik3) were also compiled using HMFlow 2010a and then all six
benchmarks were compiled with the Xilinx tools using the same configuration as described in Section 4.3.4. Runtimes were measured on all six benchmarks for both compilation flows and are shown in Table 5.4. The average speedup for larger hard macros designs compiled with HMFlow 2010a over conventional Xilinx tools is about $52 \times$, this is substantially more speedup than what was achieved using smaller hard macros as reported in Section 4.3.4.

Although runtimes were significantly reduced for the HMFlow-generated implementations, clock rates remained relatively low when compared with the Xilinx tools. Clock rates obtained for the six large hard macro benchmark circuits for HMFlow and Xilinx are shown in Table 5.5. The HMFlow implementations run between 2.5-3.5× slower than the Xilinx implementation. These clock rates are 60–70% faster than those obtained when using small hard macro versions of the benchmarks, but there is still plenty of room for improvement. Because of this large disparity in clock rate, a number of techniques were devised to improve clock rate which are discussed in the following section.

### 5.3 Modification to HMFlow for High Quality Implementations

In order to improve clock rates of large hard macro-based benchmark designs compiled with HMFlow 2010a, several new techniques and modifications were required. The result is a new version of HMFlow, called HMFlow 2011. HMFlow 2011 builds on HMFlow 2010a and, additionally contains a completely new placer, a register re-placement step in between placement and routing and an optimization tweak to the router. The major differences between the two versions of the flow are illustrated in Figure 5.8. Each of these performance optimizations are described in the remainder of this section.

<table>
<thead>
<tr>
<th></th>
<th>frequency_est.</th>
<th>trellis_dec.</th>
<th>brik3</th>
<th>brik2</th>
<th>multiband_cor.</th>
<th>brik1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx</td>
<td>237MHz</td>
<td>225MHz</td>
<td>225MHz</td>
<td>199MHz</td>
<td>243MHz</td>
<td>207MHz</td>
</tr>
<tr>
<td>HMFlow</td>
<td>82MHz</td>
<td>64MHz</td>
<td>64MHz</td>
<td>80MHz</td>
<td>80MHz</td>
<td>81MHz</td>
</tr>
<tr>
<td>Slowdown</td>
<td>2.9×</td>
<td>3.5×</td>
<td>3.5×</td>
<td>2.5×</td>
<td>3.0×</td>
<td>2.5×</td>
</tr>
</tbody>
</table>
5.3.1 Hard Macro Simulated Annealing Placer

As described in Section 4.2.6, a common algorithm used for FPGA placement is simulated annealing [27] as it is effective at producing good results, however, at the expense of longer runtimes. Regardless of the characteristic runtimes for the algorithm, a simulated annealing placer for hard macros was created out of a need to determine the potential quality of hard macro-placed designs. It was important to understand how much circuit quality was being given up by the heuristic placer used in HMFlow 2010/HMFlow 2010a. A very inefficient and slow placer was implemented that gave much better results than the placer used in HMFlow 2010, but at the cost of significantly longer runtimes (a few minutes
versus a few seconds). Performance and runtime comparisons of both placers are deferred to Section 5.4 where a more careful analysis can be presented.

It was found later that accelerating the schedule of the simulated annealing placer so it finished much sooner had very little impact on the overall quality of the resulting implementation. By using a bounding box optimization (to detect overlap) for the hard macros and a faster cooling schedule, the slow, high quality simulated annealing placer became a significant improvement over the HMFlow 2010 placer with a reasonable increase in runtime.

**Basic Algorithm**

The basic idea behind simulated annealing is to mimic the heat treatment process used in metallurgy, often to improve the organization of the structure of metals in making them more ductile and work-able. Annealing occurs by the diffusion of atoms within the metal so that the metal progresses towards an equilibrium state. By heating the metal to a very high temperature and then causing it to cool at a controlled rate, the atoms have the needed energy to break bonds and move toward an equilibrium.

In much the same way, the FPGA placement problem can be solved where the logic primitives can be “heated” and then slowly cooled so that they move toward an equilibrium, or high quality placement. The basic algorithm for simulated annealing is given in Algorithm 1.

The process begins by creating some initial state or initial placement, $S$, as a starting point for the algorithm. A starting temperature, $t$, is chosen or calculated, generally it is a large value to allow several uphill moves (moves which increase the total cost of the system) to be accepted at the beginning of the process. Then, several moves will be made and evaluated at each temperature step. When a move decreases the total system cost (calculated by the function $Cost()$), a move is always accepted. If the move increases the total system cost, it is accepted with some probability which is a function of the magnitude of the change in cost.
Algorithm 1 Basic Simulated Annealing

1: $S \leftarrow \text{createInitialPlacedSolution}()$
2: $t \leftarrow \text{initialStartingTemp}$
3: while not reachedStoppingCriteria() do
4:    for $i = 0$ to $\text{movesPerTemperatureStep}$ do
5:        $S_i \leftarrow \text{generateRandomMove}()$
6:        if $\text{Cost}(S) \geq \text{Cost}(S_i)$ then
7:            $S \leftarrow S_i$
8:        else if $e^{(\text{Cost}(S) - \text{Cost}(S_i))/t} \geq \text{random}[0,1]$ then
9:            $S \leftarrow S_i$
10:        end if
11:    end for
12:    $t \leftarrow t \times \text{temperatureReduceRate}$
13: end while
14: return $S$

and also the temperature (see line 8 of Algorithm 1). When the temperature is high, uphill moves are more likely to be accepted. However, when the temperature is low, uphill moves are less frequently accepted.

After $\text{movesPerTemperatureStep}$ moves has been made, the temperature is decreased by some percentage, $\text{temperatureReduceRate}$. This process repeats over and over until some stopping criteria is met which is often when the system cost does not change after several moves.

Customization of Simulated Annealing for Hard Macros

The implementation of the simulated annealing placer for hard macros created for HMFlow 2011 had a number of customizations which changed or deviated from the basic algorithm shown in Algorithm 1. The customizations are listed below:

1. The number of moves per temperature step varies based on the acceptance rate of the moves being generated. The number of moves per temperature step increase when the acceptance rate is near a more productive acceptance rate which was determined to be approximately 44% [33].

87
2. The inner `for` loop was replaced with a `while` loop as the moves per temperature step were only counted when they were accepted moves rather than counting total moves.

3. The initial starting temperature was set to a value equal to 1.5 times the initial system cost.

4. The process stopped if either the move acceptance rate fell below 2%, or, the temperature had dropped below a value of 0.01.

5. The cost function was the sum total measure of Manhattan distances between all port connections in between hard macros.

Most of these changes or customization were chosen simply from many iterative attempts at placement and trial and error. The number of modification and performance tweaks to the algorithm are limitless and would be far beyond the scope of this dissertation in trying to explore all the effects of each particular parameter. However, several optimizations are described later in this chapter that do explore and evaluate the performance of particular changes.

**Challenges of Hard Macros in a Simulated Annealing Placer**

One of the main challenges of performing simulated annealing on a design which is composed of mostly hard macros is that it takes much more effort to evaluate a valid move when compared with conventional designs. This is due to the way in which a move must be validated for a hard macro versus a single primitive instance. Regular FPGA designs are composed primarily of primitive instances. Instances are placed on compatible primitive sites and the check required to see if the placement is valid simply requires a comparison to make sure the primitive site is compatible and that it is unoccupied. However, when a hard macro is moved, it may contain dozens or even hundreds of primitive instances in addition to several routed nets. In a potential move, each instance must be checked for a compatible
site as well as being unoccupied, and, each PIP within each routed net must also be verified to make sure the move is valid.

Because of the significant bottleneck of checking hard macros for valid moves, trying millions of moves (a typical quantity for regular simulated annealing placers) takes an extremely long time and is quite unreasonable for a rapid compilation approach. To help accelerate the number of moves the placer could perform each second, a few optimizations were employed. First, during the creation of each hard macro, all of its valid placement locations were pre-computed off-line and stored with the hard macro in the hard macro cache. This saved several seconds of runtime when initializing the placer and allowed the move generator to choose from a set of a valid placement locations rather than randomly choosing locations that often would not support the hard macro.

Second, rather than using a tight bounding box around each hard macro as shown in Figure 5.9a, each hard macro had an approximating bounding box calculated around all of its logic and routing as shown in Figure 5.9b. The bounding boxes accelerated move
validation in that only the bounding box needed to be moved and tested for overlap with other bounding boxes in the design rather than each primitive instance and PIP in the hard macro. The bounding box approximation did restrict to some degree the flexibility of how hard macros could be placed—the bounding boxes contained some empty space making for some inefficiencies—but the trade-off reduced runtime to make it a feasible placement technique.

Another challenge that arose from using the approximating bounding box was that the Xilinx FPGA fabric is not always uniform and can skew the bounding box sizes based on where it was originally calculated. To illustrate this concept, consider Figure 5.10. On the left of the figure is the outline of a hard macro that consumes two horizontally adjacent CLB tiles. This would create a a bounding box of 1 tile high and 3 tiles wide. However, now consider the same hard macro on the right side of Figure 5.10. An extra column of configuration tiles is found in between the two columns of CLBs. This creates a bounding box of 1 tile high and 4 tiles wide.

The problem occurs when a hard macro with the smaller 1x3 bounding box is calculated and then moved to a spot where it should have a 1x4 bounding box. When hard macros
are big enough, they can span multiple columns and rows of tiles that ultimately cause the bounding box to be too small for a specific placement. This problem, on rare occasion, can lead to overlap of hard macro placements which is placement failure.

In trying to remedy this issue, it was determined that trying to compensate for the placement-varying bounding box would eliminate much of the runtime savings it created, thus an alternative solution was chosen. The solution implemented was that all hard macros would have to be checked for valid placements at the end of the simulated annealing process. If any hard macros were part of an invalid placement (overlap) the smaller of the two hard macros was replaced at the closest valid location to its final placer-decided location. The situation occurs quite infrequently (once or twice in approximately 10 placement runs) and did not significantly impact placement quality but did allow the preservation of the bounding box runtime acceleration optimization.

5.3.2 Register Re-placement

One of the biggest problems of large hard macro placement is the inherent presence of long distance connections between hard macros that often become critical paths in design implementation. The placer can reduce these paths to some extent, however, because of the rigid nature of hard macros, circuitry is not nearly as fluid and malleable as in conventional design compilation flows.

To try to reduce the effects of this problem, a new technique called register re-placement is introduced. Register re-placement occurs after the placement phase but before routing in HMFlow 2011. This technique leverages the common occurrence of registers being present at hard macro ports. The basic process for register re-placement is given in Algorithm 2.

The process begins by identifying all external hard macro nets (those nets connected exclusively to hard macro ports). From the nets, all connections (a connection being a single source pin to a single sink pin) can be extracted and then for all connections that are longer
Figure 5.11: A Simple Example of a Register (A) Being Re-placed at the Centroid of the Original Site of Register A and Sinks B, C and D
Algorithm 2 Optimal Register Re-placement

1: $C \leftarrow \text{identifyHardMacroPortConnections}()$
2: $T \leftarrow 10$
3: $S \leftarrow \{\}$
4: for each Connection $c$ in $C$ do
5: \hspace{1em} if $c$.getTotalLength() $\geq T$ then
6: \hspace{2em} $r_s \leftarrow c$.getSourceRegister()
7: \hspace{2em} $r_d \leftarrow c$.getDestinationRegister()
8: \hspace{2em} if $r_s \neq \text{null}$ and not $r_s \subseteq S$ then
9: \hspace{3em} $P \leftarrow \text{getAllPinLocations}(c)$
10: \hspace{3em} $\text{MoveRegister}(r_s, \text{findCentroid}(P))$
11: \hspace{3em} $S$.add($r_s$)
12: \hspace{2em} else if $r_d \neq \text{null}$ and not $r_d \subseteq S$ then
13: \hspace{3em} $P \leftarrow \text{getAllPinLocations}(c)$
14: \hspace{3em} $\text{MoveRegister}(r_d, \text{findCentroid}(P))$
15: \hspace{3em} $S$.add($r_d$)
16: \hspace{1em} end if
17: \hspace{1em} end if
18: end for

than a certain length (Manhattan distance from source pin to sink pin) of 10 tiles, register re-placement is considered for evaluation. If a connection has a register at its source pin, the register is moved to the centroid (arithmetic mean) of all the net’s source and sink pins. The centroid is the center of the smallest circle which encloses all of the points of interest. In this case, it would be the original location of the register and all of the net’s sink locations.

To illustrate the centroid of a net and how it is used to move a register, consider Figure 5.11 where the top picture represents a portion of the FPGA fabric after a hard macro design has been placed. The output of register A in the large hard macro on the left, is driving three separate sinks B, C and D in three other hard macros. Due to the placement, the connections are very far apart, especially from register A to sink B. In the bottom picture of Figure 5.11 it shows register A moved to the centroid (the center of the smallest circle that still includes the 4 points). This re-placement of the register significantly decreases the longest path the router would have to route in this particular case, ultimately paving the way for a higher quality implementation.
5.3.3 Router Improvements

As seen in Chapter 4, the router used in HMFlow 2010 consumes the largest fraction of the total runtime of any step in the flow. This is largely due to the higher percentage of routes it has to complete when compiling smaller hard macro-based designs. Since measuring the results of HMFlow 2010, small incremental improvements have been made to reduce the runtime of the router by profiling and testing different code transformations but no algorithmic changes. With the goal of improving clock rates with HMFlow 2011, the router was re-visited to find additional improvements with minimal impact on increasing runtime.

One of the major inefficiencies found in the HMFlow 2010 router was the fact that it often did not use long line routing resources efficiently when connections had to be made over a long distance. This resulted in long distance connections being routed with several shorter hops of less efficient resources that ultimately added up to a significant amount of propagation delay for the net. This behavior was mostly a side effect of the driving force behind the router’s original implementation which was “route as fast as possible.”

Long Lines

A long line in Xilinx FPGAs is the longest routing wire available. In the Virtex 4 architecture, long lines spanned 24 switch boxes, however, they were reduced in length to

![Virtex 4 Horizontal Long Line](image)

![Virtex 5 Horizontal Long Line](image)

Figure 5.12: Representation of Virtex 4 and Virtex 5 Long Line Routing Resources
18 switch boxes in the Virtex 5 architecture. The long line is also the only bi-directional routing wire available in modern Xilinx devices. They are also placed in both horizontal and vertical connections. Long lines are also capable of connecting to every 6th switch box in its path as illustrated in Figure 5.12.

By using long lines, routing connections that are very far apart can be connected with relatively few long lines. In Virtex 4, the next longest wire is the hex line providing a connection distance of 6 switch boxes. In Virtex 5, the next longest routing resource is the pent line which provides a connection distance of 5 switch boxes.

Unfortunately, the timing information of the wires found in Xilinx FPGAs is not available publicly and therefore, wire delay cannot be used for comparison. However, it is clear that the number of wire segments used in routing a connection can often have a bigger impact on delay rather than a connection’s length and so if the use of a long line can significantly reduce the number of total wire segments in the routed connection, it can also significantly reduce delay.

**Long Line Router**

To overcome the long distance routing inefficiency of the router, a specialized long line router was developed to specifically route long distance connections during the routing process. This long line router finds very good routes using as many long lines as possible to get closer to the sink of the connection. Once the route has used as many long lines as possible, the routing is then finished by the main routing algorithm.

The long line router is essentially a maze router that only uses long line resources. It is invoked by the main router when it encounters a connection to be routed that has a Manhattan distance of 12 switch boxes or more between source and sink. Through preliminary testing on a handful of designs and routes, it was determined that the minimum distance for which the long line router provided benefit was to invoke it for distances of 12 switch boxes or more. This is because there is some extra distance involved from connecting the source
to the starting point of the long line as well as the terminal point of the long line to the sink. These shorter connections cannot be routed as efficiently as with the Xilinx tools for lack of Xilinx proprietary timing information. Therefore, in order to avoid situations where the long line router produces routes that have a longer delay than the conventional HMFlow router\(^2\), this threshold is imposed.

In addition, long lines can only be used in discrete hops of 6, 12 and 18 hops as previously shown in Figure 5.12 and if a route is only using 6 hops worth of routing, it still incurs the delay of 18 hops. Therefore, setting a threshold of 12 would reduce the likelihood of the long line router utilizing a long line for a distance of only 6 hops. The goal of the threshold was simply to invoke the long line router when it would be likely to provide benefit and avoid situations where it could introduce unnecessary delay into a route.

Once the long line router is invoked, the first task is to find the closest and most efficient entry point to a nearby long line. This is a higher quality search than is performed by the main router and takes a bit longer to complete when compared with equivalent work loads on the main router. If no available long line resource is available, the long line router fails and returns the routing task to the main router to complete the net without long line optimization. This occurs quite infrequently, but it does happen when congestion is high around the source of the connection.

Once an efficient path to a long line has been found, the long line router will change modes to only search out a path using long line resources. Again, the algorithm is based on a maze router and will end prematurely if too much congestion is encountered. However, if a partial route using long lines was found, the long line router will provide the partial route to the main router as a starting point. This happens more frequently, but still provides some propagation delay improvements.

\(^2\)It may be interpreted by the reader that the conventional HMFlow router has access to proprietary Xilinx timing information. This is not true. The only reason it may produce routes with lower delays for short routes than the long line router is the fact that it is not being constrained to use a long line and has greater flexibility to explore for a better solution.
When the long line router exhausts the options to find the closest long line exit to the sink, it returns the partially routed path back to the main router to finish the final connection. Regardless of whether the long line router succeeds in finding a partial or complete long line path, the main router’s parameters change to try and obtain a higher quality path as the longest paths are more likely to become a critical path in the final implementation. These changes do increase router runtime to an amount of 15-20% on average.

5.4 Performance Analysis

Once all three of the HMFlow 2011 improvements were complete, it was necessary to perform analysis on their impact on quality of result. The ultimate goal of these improvements was to improve implementation clock rate while limiting runtime increase. In order to accurately and fairly measure quality of result of each improvement, a variety of configurations were created and run with all 6 of the large hard macro benchmark designs.

5.4.1 Result Measurement Fairness

By introducing the simulated annealing algorithm into the hard macro placer, the quality of result and variability become a function of the random number generator seed used in the algorithm. If the seed is always the same, then the results will always be repeatable but potentially limit quality. However, it is common practice to run the Xilinx compilation process with different seeds (Xilinx calls them table cost entry values). In fact, Xilinx has provided users with a specific tool called SmartXplorer that allows several compilation jobs to be run either in series or in parallel on a cluster where each job uses a different seed or configuration. This technique can be quite effective in obtaining higher quality results as it is very likely that the default seed may not provide the best possible implementation.

Due to the variability of some configurations of HMFlow, a single result does not accurately portray the true performance of the tools and additional metrics are given to supplement the results. For example, the Xilinx tools have a default mode which will produce
a repeatable result of a particular quality. However, it can also be configured to run with 99 other table cost placement entries which will vary the performance obtained. There will often be better resulting circuits found by running all 100 configurations versus a single run. Therefore, three results are provided for each configuration and each benchmark. The first is a single run with a default configuration, the second is the average (arithmetic mean) of 100 runs with different seeds and third, the best result obtained from the same 100 runs which produced the average.

One of the major reasons for providing all three metrics is that some designers may have a cluster of computers suitable for farming out the 100 jobs to run in parallel. In this scenario, the best of 100 results could be obtained rather quickly and would be the metric of interest. However, other designers may not have such a luxury and could only run 2 or 4 in parallel and would be more interested in the average (or expected) result. To aid decision making to both groups of designers, both metrics are included in addition to the single run result when relevant.

5.4.2 Results of Three HMFlow 2011 Improvements

To get a sense of the impact of each improvement added to HMFlow 2011, all 8 possible configurations of the three improvements within the flow were tested separately on all 6 benchmarks. The first configuration (Configuration 0) is the absence of all three improvements and is the same as HMFlow 2010a. This configuration is considered the baseline against which all 7 other configurations are compared. All of the configurations are summarized in Table 5.6.

As can be seen from Table 5.6, there are two placers (the heuristic hard macro placer from HMFlow 2010 and the new simulated annealing hard macro placer added in HMFlow 2011), an optional register re-placement step, and two routers (the router used in HMFlow 2010 and the new long line-enabled router in HMFlow 2011). All 8 configurations were tested with the 6 large hard macro benchmark circuits. Those configurations that did not use the
simulated annealing hard macro placer (C0, C2, C3, and C5) were not run 100 times as the heuristic placer was not dependent on a random seed as its input.

5.4.3 Results of Optimizing HMFlow 2011 Improvements

The results of the eight configurations of HMFlow 2011 improvements are shown in Tables 5.7, 5.8, and 5.9. Each table includes results from the Xilinx tools compiling the benchmarks to provide a comparison. The Xilinx tools were run in ‘Performance Evaluation Mode’ that attempts to obtain good results in a reasonable amount of time without a timing constraint. This mode was chosen because it most closely matched the goals of HMFlow 2011. Also, all units in the three tables are in MHz unless otherwise specified.

Table 5.7: HMFlow 2011 Benchmark Clock Rates of Single (Default) Run (in MHz)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
<th>C7</th>
<th>Xilinx</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency_est.</td>
<td>116</td>
<td>146</td>
<td>156</td>
<td>128</td>
<td>189</td>
<td>165</td>
<td>177</td>
<td>194</td>
<td>227</td>
</tr>
<tr>
<td>trellis_decoder</td>
<td>62</td>
<td>88</td>
<td>88</td>
<td>74</td>
<td>111</td>
<td>114</td>
<td>123</td>
<td>128</td>
<td>251</td>
</tr>
<tr>
<td>brik3</td>
<td>66</td>
<td>118</td>
<td>77</td>
<td>84</td>
<td>115</td>
<td>81</td>
<td>132</td>
<td>122</td>
<td>241</td>
</tr>
<tr>
<td>brik2</td>
<td>82</td>
<td>99</td>
<td>122</td>
<td>101</td>
<td>139</td>
<td>141</td>
<td>113</td>
<td>153</td>
<td>203</td>
</tr>
<tr>
<td>multiband_corr.</td>
<td>67</td>
<td>83</td>
<td>87</td>
<td>94</td>
<td>127</td>
<td>109</td>
<td>114</td>
<td>146</td>
<td>250</td>
</tr>
<tr>
<td>brik1</td>
<td>65</td>
<td>153</td>
<td>127</td>
<td>80</td>
<td>167</td>
<td>144</td>
<td>171</td>
<td>189</td>
<td>200</td>
</tr>
<tr>
<td>Average</td>
<td>76</td>
<td>115</td>
<td>109</td>
<td>93</td>
<td>141</td>
<td>126</td>
<td>138</td>
<td>155</td>
<td>229</td>
</tr>
<tr>
<td>Improvement</td>
<td>-</td>
<td>1.5×</td>
<td>1.43×</td>
<td>1.22×</td>
<td>1.85×</td>
<td>1.65×</td>
<td>1.81×</td>
<td>2.04×</td>
<td>2.99×</td>
</tr>
</tbody>
</table>
Table 5.8: HMFlow 2011 Benchmark Clock Rates of Average of 100 Runs (in MHz)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
<th>C7</th>
<th>Xilinx</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency_est.</td>
<td>116</td>
<td>144</td>
<td>156</td>
<td>128</td>
<td>170</td>
<td>165</td>
<td>167</td>
<td>186</td>
<td>220</td>
</tr>
<tr>
<td>trellis_decoder</td>
<td>62</td>
<td>100</td>
<td>88</td>
<td>74</td>
<td>124</td>
<td>114</td>
<td>123</td>
<td>139</td>
<td>247</td>
</tr>
<tr>
<td>brik3</td>
<td>66</td>
<td>108</td>
<td>77</td>
<td>84</td>
<td>116</td>
<td>81</td>
<td>119</td>
<td>123</td>
<td>238</td>
</tr>
<tr>
<td>brik2</td>
<td>82</td>
<td>91</td>
<td>122</td>
<td>101</td>
<td>131</td>
<td>141</td>
<td>115</td>
<td>153</td>
<td>198</td>
</tr>
<tr>
<td>multiband_corr.</td>
<td>67</td>
<td>101</td>
<td>87</td>
<td>94</td>
<td>151</td>
<td>109</td>
<td>124</td>
<td>177</td>
<td>253</td>
</tr>
<tr>
<td>brik1</td>
<td>65</td>
<td>144</td>
<td>127</td>
<td>80</td>
<td>169</td>
<td>144</td>
<td>169</td>
<td>168</td>
<td>203</td>
</tr>
<tr>
<td>Average</td>
<td>76</td>
<td>115</td>
<td>109</td>
<td>93</td>
<td>143</td>
<td>126</td>
<td>136</td>
<td>158</td>
<td>226</td>
</tr>
<tr>
<td>Improvement</td>
<td>-</td>
<td>1.5×</td>
<td>1.43×</td>
<td>1.22×</td>
<td>1.88×</td>
<td>1.65×</td>
<td>1.78×</td>
<td>2.06×</td>
<td>2.97×</td>
</tr>
</tbody>
</table>

Table 5.7 represents clock rates obtained for each of the benchmarks when executing a single default compilation run of the tools. Table 5.8 shows the average clock rates obtained of compiling each benchmark 100 times, each compilation using a different seed or table cost entry. For those configurations where a seed does not have an effect (heuristic placer configurations, C0, C2, C3, and C5), the values are the same as those in Table 5.7. Interestingly, the single run and average of 100 run results are very similar in their average clock rates and average improvement in clock rates over the baseline, C0. This validates the notion of a single run being a good representation of a tool’s performance. This observation holds true also for the Xilinx tool’s results.

Table 5.9: HMFlow 2011 Benchmark Clock Rates of Best of 100 Runs (in MHz)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
<th>C7</th>
<th>Xilinx</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency_est.</td>
<td>116</td>
<td>194</td>
<td>156</td>
<td>128</td>
<td>208</td>
<td>165</td>
<td>217</td>
<td>228</td>
<td>264</td>
</tr>
<tr>
<td>trellis_decoder</td>
<td>62</td>
<td>124</td>
<td>88</td>
<td>74</td>
<td>157</td>
<td>114</td>
<td>157</td>
<td>176</td>
<td>280</td>
</tr>
<tr>
<td>brik3</td>
<td>66</td>
<td>139</td>
<td>77</td>
<td>84</td>
<td>146</td>
<td>81</td>
<td>146</td>
<td>157</td>
<td>260</td>
</tr>
<tr>
<td>brik2</td>
<td>82</td>
<td>134</td>
<td>122</td>
<td>101</td>
<td>167</td>
<td>141</td>
<td>160</td>
<td>180</td>
<td>205</td>
</tr>
<tr>
<td>multiband_corr.</td>
<td>67</td>
<td>148</td>
<td>87</td>
<td>94</td>
<td>207</td>
<td>109</td>
<td>172</td>
<td>223</td>
<td>270</td>
</tr>
<tr>
<td>brik1</td>
<td>65</td>
<td>191</td>
<td>127</td>
<td>80</td>
<td>214</td>
<td>144</td>
<td>205</td>
<td>212</td>
<td>214</td>
</tr>
<tr>
<td>Average</td>
<td>76</td>
<td>155</td>
<td>109</td>
<td>93</td>
<td>183</td>
<td>126</td>
<td>176</td>
<td>196</td>
<td>249</td>
</tr>
<tr>
<td>Improvement</td>
<td>-</td>
<td>2.03×</td>
<td>1.43×</td>
<td>1.22×</td>
<td>2.4×</td>
<td>1.65×</td>
<td>2.31×</td>
<td>2.57×</td>
<td>3.26×</td>
</tr>
</tbody>
</table>

Table 5.9 represents the best clock rates obtained from the best of 100 compilation runs. These results could be obtained very quickly if all 100 compilation runs could be run
in parallel such as on a supercomputer or cluster. When compared with averages of the single and average of 100 runs, the best of 100 runs results are approximately 50% better. This provides a significant advantage to those with easy access to parallel computing power. However, the advantage is less if the designer is using the Xilinx tools which only produce approximately 25-30% better clock rates.

**Impact of Improvements**

The simulated annealing hard macro placer had the greatest impact in performance improvement. Compared to baseline (C0) implementations, replacing the heuristic placer with the simulated annealing version (C1) increased clock rates on average by 50%. The next most impactful improvement was the addition of the register re-placement step (C2) that improved clock rates by 43%. The improvement of the long line optimized router (C3) had the smallest impact when measured in isolation, providing improved clock rates of only 22%. These results illustrate the impact a good or bad placement can have on the overall implementation clock rate. The router is also likely to be disadvantaged as the timing information on which a router could make intelligent decisions is not freely available at a sufficient level of detail.

Dual combinations of the improvements (C4, C5, and C6) were closely additive in their improvement of clock rate showing how each improvement approach was mostly independent of the others. C7, the configuration which combined all three of the improvement techniques gave the best clock rate improvement of $2 \times$ or $2.5 \times$ in the best of 100 runs case.

**Comparison of HMFlow 2011 to Xilinx**

Xilinx still produced the fastest clock rates on average when compared to HMFlow 2011. Xilinx implemented design clock rates were on average almost 50% faster than the C7 clock rates. However, if the C7 results of Table 5.9 are compared to the Xilinx results
of Table 5.7, the resulting implementation clock rates are much closer and in two cases (frequency estimator and brik1) are actually faster than those produced by Xilinx.

Variance of Results

It was obvious from the results that the simulated annealing placer introduced variance of the quality of the results when comparing 100 runs of the same design. The lingering question after these results were obtained was, how much variance existed and how did it compare to the variance of the Xilinx tools? Table 5.10 shows a summary of the variance of the 100 runs for both HMFlow 2011 and Xilinx for the 6 benchmarks.

Table 5.10: Variance of HMFlow 2011 (C7) and Xilinx in 100 Compilation Runs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>HMFlow 2011</th>
<th>Xilinx</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency_est.</td>
<td>0.368</td>
<td>0.094</td>
</tr>
<tr>
<td>trellis_decoder</td>
<td>0.637</td>
<td>0.050</td>
</tr>
<tr>
<td>brik3</td>
<td>2.054</td>
<td>0.027</td>
</tr>
<tr>
<td>brik2</td>
<td>0.437</td>
<td>0.007</td>
</tr>
<tr>
<td>multiband_corr.</td>
<td>0.460</td>
<td>0.011</td>
</tr>
<tr>
<td>brik1</td>
<td>0.396</td>
<td>0.020</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>0.725</strong></td>
<td><strong>0.035</strong></td>
</tr>
</tbody>
</table>

The variance $\rho$ was calculated across the entire population of 100 implementation results for each benchmark using the standard equation

$$\rho = \sum_{i=1}^{n} (x_i - \mu)^2 \quad (5.1)$$

where $\mu$ is the arithmetic mean calculated by

$$\mu = n^{-1} \sum_{i=1}^{n} x_i. \quad (5.2)$$

The variance of HMFlow 2011 is significantly higher than that of the Xilinx tools, more than $20 \times$ higher. This illustrates the ability for the Xilinx tools to produce results
that are much closer together than those of HMFlow 2011. The reason this is significant is the concern that if a designer were to use HMFlow in a single run scenario, it is much more likely to obtain a poor result because of the large variation of the results than it would be if the designer were to use the Xilinx tools.

5.5 Techniques for Reducing Variance

To try and improve the robustness of the new improvements found in HMFlow 2011, three techniques were considered to attempt to minimize the total variance of the results produced by the compilation flow. It was determined through some preliminary tests that the source of much of the variability in HMFlow 2011 was due to the simulated annealing placer. Therefore, the techniques to reduce variability changed the behavior of the simulated annealing placer. The three techniques are summarized below:

1. Change move acceptance (when a move increases the system cost) to be a function of the hard macro port count

2. Perform a small hard macro re-placement optimization step (similar to register re-placement)

3. Change the cost function to include a factor of the longest wire length between any two hard macros

Each technique will be referred to hereafter as T1, T2 and T3 respectively. The remainder of this section will describe each technique to reduce variance in detail and also present results that show the best configuration to use in each case. Finally, all possible configurations of the three techniques are compared and the best performing (lowest variance) is selected.
5.5.1 T1: Move Acceptance a Function of Hard Macro Port Count

T1 is a technique that was designed to address a behavior that was noticed in placing large hard macros during the simulated annealing process. The hard macros that had several ports (generally larger hard macros) were being moved less often than other hard macros with fewer ports. This often led to lower quality placements because the larger hard macros would not benefit from as many moves as other hard macros. The equation of interest to this problem is the move acceptance equation

\[ r < e^{-\frac{\Delta C}{T}}. \] (5.3)

The acceptance equation determines how often a move is accepted when it increases the total system cost (moves that decrease total cost are always accepted). When the net change in total system cost \( \Delta C \) is large, as is often true in the case of moves involving hard macros with several ports, the likelihood of accepting the move is lower. The current temperature \( T \) also reduces the likelihood of accepting the move as it decreases. The right hand side of Equation 5.3 is compared to a random number \( r \) that is generated inside the interval \( (0, 1] \). If the right hand side of the equation is larger than \( r \), the move is taken otherwise it is discarded.

To attempt to equalize the number of moves or perturbations each hard macro received, the acceptance equation was changed from Equation 5.3 to

\[ r < e^{-\frac{\Delta C}{T + QP}}. \] (5.4)

Two factors were added to the denominator of the right hand side of the exponential. First, the number of ports \( P \) involved in the hard macro move and second, a scaling factor \( Q \) to control the impact of the change. If \( Q = 0 \), then the performance should be roughly (random seed differences) the same as the variance of C7 as shown in Table 5.10.
After making this change to the acceptance equation, some preliminary tests were conducted to verify that the change indeed behaved as expected. To illustrate the behavior of this technique, consider Figure 5.13 which numbers all of the hard macros from benchmark brik1 from 1 to 45. Each accepted uphill move (moves with a positive $\Delta C$) for each hard macro is tabulated for three scenarios: $Q$ set to 0, 10, and 20. The port counts on these 45 hard macros vary from 1 to 88 and the hard macros are sorted in increasing port count from left to right along the X-axis. When $Q$ is 0 (baseline or C7), many of the hard macros with larger port counts do not have as many accepted uphill moves. However, as $Q$ increase to 10 and then 20, the hard macros with higher port counts receive a more equal share of the total accepted uphill moves.

With the desired behavior of this technique controlled by the parameter $Q$, a parameter sweep was conducted in order to determine the best value of $Q$ to reduce variance of the
implementation output. A graph of the scaling factor sweep of $Q$ is shown in Figure 5.14 and an enlarged version of the range 0 to 11 in Figure 5.15.

Figure 5.14 shows some impact to the variance in the interval where $0 < Q < 11$. However, when $Q > 11$, the variance rises significantly. This is likely due to a significant reduction in the number of accepted uphill moves which allow the simulated annealing process to escape local minima. Overall, the best performing scaling factor was 2.0 with a variance of 0.492.

5.5.2 T2: Small Hard Macro Re-placement

T2 is a technique that mimics the behavior of the register re-placement improvement in HMFlow 2011. It was found in some preliminary testing that the register re-placement step reduced variability in the quality of results produced by HMFlow 2011 and was thought that exploiting that technique within the placer could provide further reduction of variability.
The technique works by modifying placement after the simulated annealing process has completed. It iterates over all of the small hard macros (which are quite likely to contain registers) in a design and performs re-placement on them so that they are placed at the centroid of all of their port connections using the same approach as described in Section 5.3.2. This is a greedy optimization that attempts to reduce wire length and critical paths so that the final implementation will be closer to the ideal clock rate.

The main question in implementing such a technique is, up to what size hard macro should be moved? There should be a ceiling on the effectiveness of moving ever larger hard macros as moving larger hard macros will tend to create more problems than it solves. In order to answer this question, a parameter sweep in hard macro tile size was performed. Hard macro size can be calculated by the number of tiles it occupies. The results of the parameter sweep of hard macro size versus variance of results is shown in Figure 5.16 and an enlarged version of the graph from 0 to 100 tiles in Figure 5.17.

Figure 5.15: T1 Variance of Scaling Factor Q Parameter Sweep (Zoomed)
Figure 5.16: T2 Variance of Hard Macro Size Re-placement Parameter Sweep

The average variance plot experiences a step like behavior at about 150, 400 and 2000 tiles. This is due to a benchmark experiencing significant quality reduction when trying to move hard macros of a certain size. The lowest variance results were obtained when only moving hard macros of size 150 tiles or less. The lowest variance result (0.499) was obtained when moving hard macros of 20 tiles or less and this value was chosen as the final configuration for this technique.

5.5.3 T3: Cost Function Includes Longest Wire

In preliminary tests it was found that total wire length between hard macros was a better performing cost function than the longest wire in between any two hard macros. However, when using large hard macros that contain timing verified connections that have been placed and routed with the higher quality Xilinx tools, only the connections in between
Figure 5.17: T2 Variance of Hard Macro Size Re-placement Parameter Sweep (Zoomed)

hard macros will limit the maximum achievable clock rate. This would indicate that the longest wire in between any two hard macros would likely become the critical path for the design. Therefore, reducing the longest path in between any two hard macros would likely produce a better result.

T3 combines the total wire length function and longest wire scaled by some factor into a single cost function. Again, as in the previous techniques, a parameter sweep was conducted to find the best variance-minimizing value for the scaling factor of the longest wire in the cost function. Results of the parameter sweep are shown in Figure 5.18 and an enlarged area of the graph (values 3000 to 100000) is shown in Figure 5.19.

Using the parameter sweep for T3 uncovers a significant reduction in variance with the minimum variance at 0.278 when using a scaling factor of 16384 on the longest wire. Because of the significant improvement, a more fine-grained sweep was performed around 16384 and can be seen in both Figure 5.18 and 5.19 with the several jagged peaks and valleys
Figure 5.18: T3 Variance of Longest Wire Scaling Factor Parameter Sweep

Figure 5.19: T3 Variance of Longest Wire Scaling Factor Parameter Sweep (Zoomed)
around 16384. Unfortunately, the fine-grained sweep did not uncover any better performing scaling factors. However, this technique turned out to be the most significant reduction of all 3 techniques used to try to minimize variance.

To understand the effect this technique has on the cost function, consider Figure 5.20. This figure plots total wire length as a percentage of the total cost of the system for each cost function evaluation for the brik1 benchmark. This data was taken using the T3 configuration with the scaling factor set to 16384. The total wire length only becomes approximately 0.5–2% of the total system cost, which seems insignificant, however it appears that this combination produces an effective balance to find good solutions.
5.5.4 Configuration Comparison of Techniques

With three different techniques shown to reduce variance of results in HMFlow 2011 to some degree, it would be advantageous to determine if such techniques could be combined to further reduce variation of results. Therefore, experimental runs were conducted where all possible combinations of techniques were combined and their results are given in Table 5.11.

Table 5.11: Average Variance and Frequency using Variance-reducing Techniques

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Variance</th>
<th>Freq.</th>
<th>Avg. 100 Run Freq.</th>
<th>Best 100 Freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline (C7)</td>
<td>0.725</td>
<td>155MHz</td>
<td>158MHz</td>
<td>196MHz</td>
</tr>
<tr>
<td>T1</td>
<td>0.492</td>
<td>160MHz</td>
<td>159MHz</td>
<td>200MHz</td>
</tr>
<tr>
<td>T2</td>
<td>0.499</td>
<td>160MHz</td>
<td>158MHz</td>
<td>196MHz</td>
</tr>
<tr>
<td>T3</td>
<td>0.278</td>
<td>175MHz</td>
<td>165MHz</td>
<td>195MHz</td>
</tr>
<tr>
<td>T1+T2</td>
<td>0.451</td>
<td>164MHz</td>
<td>158MHz</td>
<td>199MHz</td>
</tr>
<tr>
<td>T1+T3</td>
<td>0.34</td>
<td>157MHz</td>
<td>161MHz</td>
<td>197MHz</td>
</tr>
<tr>
<td>T2+T3</td>
<td>0.303</td>
<td>165MHz</td>
<td>164MHz</td>
<td>193MHz</td>
</tr>
<tr>
<td>T1+T2+T3</td>
<td>0.307</td>
<td>159MHz</td>
<td>163MHz</td>
<td>200MHz</td>
</tr>
<tr>
<td>Xilinx</td>
<td>0.035</td>
<td>229MHz</td>
<td>226MHz</td>
<td>249MHz</td>
</tr>
</tbody>
</table>

The results shown in Table 5.11 are calculated based on each tool configuration compiling all 6 large hard macro benchmark designs and averaging their variance, single run implementation clock rate, average implementation clock rate of 100 runs and best implementation clock rate of 100 runs. The baseline (C7) configuration defined in Table 5.6 and Xilinx tools data are repeated for reference.

The HMFlow 2011 tool configuration with the minimum variance is T3 implemented by itself. No combination of techniques including T3 was able to further reduce variance when compared to T3 alone. T3 also had the highest average single run frequency of any configuration (175MHz) and also the highest frequency of the average of 100 runs (165MHz). It would appear that reducing variance also increases average implementation clock rate as the resulting frequencies are grouped tighter together. The only metric in which T3 did not perform better than the other configurations is the best frequency of 100 runs. This also
could be explained as caused by reducing variance as the best implementations are pushed closer to the average.

T1 and T2 did reduce variance when compared with the baseline (C7), however, their variance was almost twice as much as that of T3. T1 and T2 also did not have a significant impact on the implementation clock rate of the average of 100 runs—matching or slightly better than baseline. The combinations of the techniques did reduce variance more than T1 or T2 alone, T3 simply performed better.

Overall, the techniques did not drastically reduce the variance of the placer. This is likely due to the secondary motive of trying to compile quickly and also using a variety of hard macro sizes (large and small). However, T3 did reduce variance in results by over $2.5 \times$ and also slightly improved performance. For the effort, the improvements were certainly worthwhile.

5.6 Runtime Results

Now that circuit quality has been optimized for the different configurations presented in this chapter, the runtimes of each configuration should also be reported and compared. Table 5.12 shows the runtime for each of the 6 benchmarks as compiled with the major HMFlow revisions presented in this chapter. Runtime is measured in seconds and is defined as the total time to compile a design starting from the time taken to read in a design’s source files and ending after creating a placed and routed implementation file (XDL or NCD).

As can be seen from Table 5.12, the baseline C0 has the fastest runtime performance of any tool configuration. This is largely due to its usage of the lower quality but faster placer and router algorithms present in HMFlow 2010. When the newer simulated annealing, register re-placement and long-line optimized router are introduced (C7), runtimes increase by a little less than 50%. T3 boasts the same algorithmic improvements as C7 but with a significant reduction in variance of result and roughly has the same runtime performance of
Table 5.12: Compilation Runtime for Several HMFlow 2011 Configurations vs. Xilinx

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline (C0)</th>
<th>C7</th>
<th>HMFlow 2011 (T3)</th>
<th>Xilinx</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency_estimator</td>
<td>6.82s</td>
<td>10.01s</td>
<td>10.91s</td>
<td>392.06s</td>
</tr>
<tr>
<td>trellis_decoder</td>
<td>11.81s</td>
<td>18.2s</td>
<td>18.57s</td>
<td>461.34s</td>
</tr>
<tr>
<td>brik3</td>
<td>13.81s</td>
<td>28.56s</td>
<td>27.81s</td>
<td>605.16s</td>
</tr>
<tr>
<td>brik2</td>
<td>15.26s</td>
<td>20.94s</td>
<td>19.98s</td>
<td>851.61s</td>
</tr>
<tr>
<td>multiband_correlator</td>
<td>12.45s</td>
<td>18.82s</td>
<td>18.8s</td>
<td>497.91s</td>
</tr>
<tr>
<td>brik1</td>
<td>15.98s</td>
<td>15.56s</td>
<td>15.66s</td>
<td>848.79s</td>
</tr>
<tr>
<td>Average Runtime</td>
<td>12.69s</td>
<td>18.68s</td>
<td>18.62s</td>
<td>609.48s</td>
</tr>
<tr>
<td>Speedup (over Xilinx)</td>
<td>48.0×</td>
<td>32.6×</td>
<td>32.7×</td>
<td>-</td>
</tr>
</tbody>
</table>

C7, the only modification of T3 was to add one extra component to the cost function in the placer which is a very small percentage of total runtime.

Runtime, however is only half of the story. Looking at both runtime and quality of result provides a better picture of the actual tradeoff occurring as a result of the various configurations of HMFlow. As performance numbers are scattered throughout this chapter, they are summarized in Table 5.13 for convenience. The clock rates shown in Table 5.13 are measured in MHz and were all taken from a single compilation run of the tools.

Table 5.13: Clock Rate Summary (in MHz) for HMFlow 2011 Configurations vs. Xilinx

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline (C0)</th>
<th>C7</th>
<th>HMFlow 2011 (T3)</th>
<th>Xilinx</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency_estimator</td>
<td>116</td>
<td>194</td>
<td>182</td>
<td>227</td>
</tr>
<tr>
<td>trellis_decoder</td>
<td>62</td>
<td>128</td>
<td>166</td>
<td>251</td>
</tr>
<tr>
<td>brik3</td>
<td>66</td>
<td>122</td>
<td>131</td>
<td>241</td>
</tr>
<tr>
<td>brik2</td>
<td>82</td>
<td>153</td>
<td>173</td>
<td>203</td>
</tr>
<tr>
<td>multiband_correlator</td>
<td>67</td>
<td>146</td>
<td>196</td>
<td>250</td>
</tr>
<tr>
<td>brik1</td>
<td>65</td>
<td>189</td>
<td>199</td>
<td>200</td>
</tr>
<tr>
<td>Average Clock Rate</td>
<td>76</td>
<td>155</td>
<td>175</td>
<td>229</td>
</tr>
</tbody>
</table>

Overall, the results can be viewed as a positive outcome for HMFlow. The final configuration of HMFlow 2011 (T3) can produce implementations over 30× faster than the Xilinx tools and still obtain clock rates that are 75% of the implementations produced by the best Xilinx efforts. Put another way, HMFlow produces a placed and routed implementation that runs 175 MHz on average and can be obtained in less than 20 seconds. This is in stark
contrast to Xilinx which will take over 10 minutes to produce a design that will only run about 30% faster. This is a tremendous result for HMFlow as it demonstrates that rapid compilation is a very feasible accomplishment in FPGA CAD and could radically change the way designers create and implement designs for FPGAs.

5.7 Conclusions

The beginning of this chapter began with HMFlow 2010—optimized for rapid compilation times, utilized small hard macros and the Virtex 4 architecture. Then, the flow was augmented to support large hard macros and the Virtex 5 architecture which created HMFlow 2010a. Once those modifications were in place, three new improvements were added (simulated annealing placer, register re-placement, and long line-optimized router) to create HMFlow 2011. Several configurations of these techniques were tested with C7 as the best performing. However, it was noted that variance of result was significant and more improvements were attempted to reduce it. This resulted in more experiments and configurations with T3 being the best configuration overall.

From where this chapter began with HMFlow 2010, average clock rates have improved by almost $3 \times$ and even though implementation clock rate variance is still significant, it was reduced by $2.5 \times$. All of this was accomplished while delivering compilation times that are still over $30 \times$ faster than the conventional Xilinx flow. Given these performance numbers, HMFlow offers an attractive alternative to conventional FPGA compilation techniques and has the potential to increase designer productivity with its rapid compilation benefits.
CHAPTER 6. THE BIG PICTURE: THE COMPILATION TIME VS.
CIRCUIT QUALITY TRADEOFF

In Chapter 4, the major motivation was to implement a compilation flow (HMFlow 2010) that could compile as fast as possible without regard for circuit quality. With the success of HMFlow 2010, efforts described in Chapter 5 were focused on improving quality while trying to also maintain short runtimes in creating HMFlow 2011. With the tradeoff of compilation runtime and clock rate pulling in opposite directions and affecting FPGA design methodologies in a variety of ways, trying to understand and convey the implications of such a tradeoff is difficult to condense down to a nicely formatted \LaTeX{} table. Something more is required to endow the reader with any reasonable sense of how the tradeoff behaves and its implications for FPGA designers.

The purpose of this chapter is to provide the reader a view of the big picture of why such significant efforts were made to build a rapid compilation flow and what impact these results can have on the future of FPGA design.

6.1 Motivation

It should first be made clear why the compilation time vs. quality of result tradeoff is significant. To do this, consider a rough representation of current tool implementation quality vs. runtime offerings in Figure 6.1 where compilation time is on the horizontal axis and quality of result (clock rate) is on the vertical axis. FPGA vendors provide very high quality tools to produce high quality of result implementations as this has been what markets have generally demanded over the past several years. FPGA design starts have typically required the fastest clock rate possible, using as much of the FPGA as possible for
the cheapest price. Given these difficult pressures on FPGA vendors, compilation runtime has typically had to suffer as most of the effort to improve tools has been expended in producing higher quality of result. Hence, the current offerings of FPGA vendor tools is shown on the right-hand side of Figure 6.1.

However, in more recent years, the demands of FPGAs have been changing. With the benefits of Moore’s Law continually doubling FPGA logic capacities every few years, the critical pressure that existed previously—to fill an FPGA to capacity in order to cut costs—is starting to be replaced by growing development costs driven mostly by longer development times. FPGAs have benefited significantly from Moore’s Law, but workstation processors, on which the FPGA vendor tools run, have not. Although FPGA vendor tools continue to improve, the time to compile an FPGA circuit is getting longer with each generation.
Figure 6.2: Quality vs. Runtime Tradeoff for HMFlow 2011 and Xilinx Tools

With changing market pressures that are starting to push for faster compile times, it is advantageous to enable new ways of compiling FPGA circuits that could allow FPGA designers the opportunity to take advantage of the left-hand side of Figure 6.1. HMFlow accomplishes this task by demonstrating compile times over $30 \times$ faster than the conventional flow. In Figure 6.2, the major revisions of HMFlow 2011 are plotted with the Xilinx tools to produce a tradeoff graph similar to the one presented in Figure 6.1. Based on the data in the plot of Figure 6.2, it would indicate that there exists a very steep ramp at the beginning of the tradeoff. That is, when runtimes are very short, increasing runtime by a small amount would increase clock rate by a larger amount. However, when runtimes are long, adding additional runtime will not produce as much gain in clock rate.
6.2 Implications for FPGA Designers

The significance of this tradeoff was also presented in [34] as the authors recognized the utility of choosing a compilation scheme to match the design situation rather than providing a one-compile-scheme-fits-all solution. Providing the ability to trade small amounts of quality for significantly faster runtimes is extremely useful to an FPGA designer and it could motivate new compilation strategies for rapid prototyping. FPGA designers could work in a rapidly compiling “debug mode” that could significantly increase the number of edit-compile-debug turns per day he or she could accomplish. By accelerating compilation times, development times become shorter and less costly. Another benefit of this approach is that once a design is fully verified, the high quality tools can still be used to produce the final high quality implementation. The results of this dissertation shows that there is merit in approaching compilation from a different angle than the one traditionally taken by FPGA vendors.

6.3 Contributions

To summarize, the major contributions of this dissertation are listed below:

1. Provided an open source framework to create custom FPGA CAD experiments on commercial FPGAs called RapidSmith. RapidSmith has received over 500 downloads worldwide and received the Community Service Award at the 21st International Conference on Field Programmable Logic and Applications Conference in 2011. RapidSmith served as the foundation framework for all of the research performed in this dissertation, has served and continues to serve as a resource for other research projects at Brigham Young University and at other universities internationally.

2. Demonstrated a complete custom rapid compilation flow leveraging hard macros called HMFlow 2010 that compiles designs 10× faster or more than the fastest conventional vendor tools by accepting clock rates 2–4× slower than vendor tools. HMFlow 2010
targeted real Virtex 4 FPGAs and produced functional implementations demonstrating the potential of hard macros to serve as a rapid compilation technology.

3. Demonstrated an improved HMFlow 2011 that leverages large hard macros to preserve valuable timing closure information to accelerate compilation of high performance designs. HMFlow 2011 included a number of improvements over its predecessor to improve clock rates to an average of 75% of the best implementations provided by Xilinx while still compiling an average of 30× faster.

4. Provided insights into the largely unexplored compilation time vs. circuit quality tradeoff curve to enable new kinds of compilation approaches for rapid prototyping. These insights will enable new techniques of compilation and design that could dramatically increase FPGA designer productivity and ultimately lower development costs for FPGA designs.

6.4 Conclusions

Engineering is about creating solutions to problems and then further enhancing those solutions to become better, less expensive, more efficient, lower power, smaller and faster. FPGA circuit design is both a solution and problem that has existed for several years. FPGAs have enabled significant computational performance in embedded applications for attractive performance per Watt rates. However, the problem of long expensive design times has been growing and new techniques to solve this growing problem are needed.

This dissertation aims to provide valuable insight and results that FPGA compilation can be accomplished over an order of magnitude faster than what is conventionally done if the user is willing to trade a little bit of clock rate performance for a lot of compilation speedup. Pre-compiled modules or hard macros enable such compilation performance and this dissertation demonstrates their effectiveness in HMFlow 2010 and HMFlow 2011. These experiments and results were performed on actual commercial parts and are proof of con-
cept in their own right. The adoption of hard macros and the techniques presented in this
dissertation would significantly reduce designer development time by enabling rapid compi-
lation and bring FPGA designer productivity much closer to that of their software designer
counterparts.

6.5 Future Work

This dissertation branches out into an area of relatively unexplored research. Al-
though the goals of this dissertation have been met, the work performed herein has only
scratched the surface of exploring the implications of creating hard macros for general pur-
pose design. Future work could expand into such areas as hard macro creation, representa-
tion, placement and routing.

The artificial challenges presented by XDL and its conversion time to NCD is certainly
a step that would need to be addressed for commercial usage of these techniques. However, if
similar device information could be obtained from other FPGA vendors, the same techniques
could also be applied to their respective architectures and the same performance and results
could be achieved.

As detailed Xilinx timing information is proprietary and not publicly available, it was
not used or leveraged for the work presented in this dissertation. If persons with such access
to timing information were able to implement the techniques presented here, additional
optimizations would be quite productive, especially in the routing algorithm to produce
better implementations.

It is theoretically possible with detailed knowledge of the proprietary Xilinx bitstream
that hard macros could be pieced together at the bitstream level as was mentioned in Chap-
ter 2. If such information were available, future work could include such techniques and
provide even further acceleration of the compilation flow.
REFERENCES


125