Autotuning the Intel HLS Compiler using the Opentuner Framework

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
In Partial Fulfillment of the Requirements
For the Degree of Master of Science
In the Department of Electrical and Computer Engineering
University of Saskatchewan

by

Chandler Janzen

Saskatoon, Saskatchewan, Canada

© Copyright Chandler Janzen, September, 2019. All rights reserved.

Permission to Use

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, it is agreed that the Libraries of this University may make it freely available for inspection. Permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professors who supervised this thesis work or, in their absence, by the Head of the Department of Electrical and Computer Engineering or the Dean of the College of Graduate Studies and Research at the University of Saskatchewan. Any copying, publication, or use of this thesis, or parts thereof, for financial gain without the written permission of the author is strictly prohibited. Proper recognition shall be given to the author and to the University of Saskatchewan in any scholarly use which may be made of any material in this thesis.

Disclaimer

Reference in this thesis/dissertation to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement, recommendation, or favoring by the University of Saskatchewan. The views and opinions of the author expressed herein do not state or reflect those of the University of Saskatchewan, and shall not be used for advertising or product endorsement purposes.

Request for permission to copy or to make any other use of material in this thesis in whole or in part should be addressed to:

Head of the Department of Electrical and Computer Engineering
57 Campus Drive
University of Saskatchewan
Saskatoon, Saskatchewan, Canada
S7N 5A9

OR

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9 Canada

Abstract

High level synthesis (HLS) tools can be used to improve design flow and decrease verification times for field programmable gate array (FPGA) and application specific integrated circuit (ASIC) designs. The Intel HLS Compiler is a high level synthesis tool that takes in untimed C/C++ as input and generates production-quality register transfer level (RTL) code that is optimized for Intel FPGAs. The translation can, however, require multiple iterations and manual optimizations to get comparable synthesized results to that of a solution written in a hardware descriptive language. The synthesis results can vary greatly based upon coding style and optimization techniques, and typically require an in-depth knowledge of FPGAs to fully optimize the translation which limits the audience of the tool. The extra abstraction that the C/C++ source code presents can also make it difficult to meet more specific design requirements; this includes designs to meet specific resource usage or performance based metrics. To improve the quality of results generated by the Intel HLS Compiler without a manual iterative process that requires an in-depth knowledge of FPGAs, this research proposes a method of automating some of the optimization techniques that improve the synthesized design through an autotuning process. The proposed approach utilizes the PyCParser library to parse C source files and the OpenTuner Framework to autotune the synthesis to provide a method that generates results that better meet the needs of the designer's requirements through lower FPGA resource usage or increased design performance. Such functionality is not currently available in Intel's commercial tools.

The proposed approach was tested with the CHStone Benchmarking Suite of C programs as well as a finite impulse response filter. The results show that the commercial HLS tool can be automatically autotuned through placeholder injection using a source parsing tool for C code and using the OpenTuner Framework to autotune the results. For designs that are small in nature and include conducive structures to be autotuned, the results indicate resource usage reductions and/or performance increases of up to 40% as compared to the default Intel HLS Compiler results. The method developed in this research also allows additional design targets to be specified through the autotuner for consideration in the synthesized design which can yield results that are better matched to a design's requirements.

Acknowledgments

I am grateful for the opportunity to study at the University of Saskatchewan and for the support of the Electrical and Computer Engineering Department. I have received a great deal of support and assistance since I started my research. First, I would like to thank my supervisor, Dr. Brian Berscheid for his guidance and countless hours of investment into my research. His expertise was invaluable in the progression of the research.

Second, I'd like to acknowledge my colleagues for their support and suggestions throughout this journey. In particular: Rory Gowen, Jason Pannell, Dr. Daniel Teng, and Dr. Eric Salt have been very helpful in offering advise and suggestions along the way. Their support and banter over ideas, methods, and paradigms kept things moving along.

Finally, I would like to express my gratitude to my family. Without their support, none of this would have been possible. To my wife, Marcee Janzen, and my kids: Kain, Bella, Nicholas, and Walter; thank you for the much needed encouragement.

Table of Contents

Permission to Use i

Disclaimer i

Abstract iii

Acknowledgments iv

Table of Contents iv

List of Tables ix

List of Figures xii

List of Abbreviations xv

1 Introduction 1

1.1 FPGAs 1

1.2 HLS 4

1.2.1 Intel HLS Compiler 5

1.2.2 Other HLS Tools 6

1.2.3 Design Flow Using Intel HLS 8

1.2.4 Intel HLS Compiler Advantages Over Traditional HDL Development 10

1.2.5 Challenges of Using the Intel HLS Compiler 11

1.3 Intel HLS Compiler Automation 13

1.3.1 Autotuning Using OpenTuner 13
1.3.2 PyCParser: Parsing C Code and Injecting Compiler Directives 16
1.3.3 Docker 17
1.4 Research Objective 17
1.5 Thesis Outline 19

2 Autotuning Parameters 21
2.1 Introduction 21
2.2 Intel HLS Optimization Techniques 21
2.2.1 HLS Interfaces 21
2.2.2 Data Types 23
2.2.3 Local Memory Optimizations 24
2.2.4 Loop Optimizations 27
2.2.5 Coding Practice Related to HLS Optimization 30
2.3 Autotunable Parameters 30
2.3.1 Choosing Tunable Parameters 30
2.3.2 Defining Parameter Ranges 32
2.3.3 Injecting Parameters Into Source Code 36
2.3.4 Limitations 37

3 OpenTuner Configuration and Execution 40
3.1 OpenTuner Framework 40
3.1.1 Search Techniques 41

3.1.2 Configuration Manipulator 43
3.1.3 Objectives 45
3.2 OpenTuner Usage for Tuning Intel HLS 45
3.2.1 Search Techniques 45
3.2.2 Search Space Size 45
3.2.3 Results Generation 46
3.2.4 Compilation Types 48
3.2.5 Adjusting Optimization Preferences 51
3.2.6 Specifying Targets 53
4 Results 55
4.1 Impact of Individual Parameters 56
4.1.1 Memory Optimizations 57
4.1.2 Loop Optimizations 59
4.2 Autotuning the Intel HLS Compiler 62
4.2.1 Use of CHStone 63
4.3 Establishing the Reference for the Autotuner 66
4.3.1 CHStone Applications 66
4.3.2 DSP FIR Filter 67
4.4 Setup of Autotuning for Intel HLS Compiler Applications 68
4.4.1 OpenTuner Configuration 68
4.4.2 Search Space Size and Techniques 71

4.5 CHStone Autotuning Results 74
4.5.1 Establishing a Starting Point 74
4.5.2 Correlation Between Estimated vs Post-Mapped and Estimated vs Post-Fitted Results 76
4.5.3 WNS CHStone Application Results 78
4.5.4 Other CHStone Application Results 85
4.6 DSP FIR Filter Autotuning Results 91
4.6.1 Verification of Results 91
4.6.2 Autotuning with Targets Introduced 100
5 Summary and Conclusions 104
5.1 Summary 104
5.1.1 Benefits of Autotuning 104
5.1.2 Limitations of Autotuning 106
5.2 Thesis Contributions 107
5.3 Future Work 109
A DSP FIR Filter Example 110
B User Defined Configuration File 111
B.1 autotuner_config.json 111
B.2 run.sh 113
C Post-Mapped and Post-Fitted CHStone Results 116
D Verilog Code for FIR Filter 125
References 134

List of Tables

2.1 Optimizing Interfaces to Components [1] 22

2.2 Optimization Parameters for Autotuning System 32

2.3 Loop Unroll Ranges for a Shift Register Circuit 34

2.4 Loop Unroll Ranges for a Shift Register Circuit Implemented in an FIR Filter Circuit 35

2.5 Default Configuration Parameters and their Parameter Ranges 36

2.6 Original Code Versus Generated Code 37

2.7 Using the Blacklist Option 38

3.1 OpenTuner Built-In Search Techniques 42

3.2 Impact on the Search Space 46

3.3 Optimization Strategies and Associated Weights 52

4.1 Memory Optimizations for Local Variables Using Mapped Resources 57

4.2 Memory Optimizations for Local Variables Using Fitted Resources 58

4.3 Results of the Individual Impact of a Memory Optimization Normalized to the hls_singlepump Solution for the *data* Variable 59

4.4 Loop Unrolling Examples 60

4.5 Results of the Individual Impact of Loop Unrolling 60

4.6 Loop Coalescing Examples 61

4.7 List of CHStone Applications and Their Functions 62

4.8 Modifications Made to Blowfish 65

4.9 Initial CHStone Application Results with Estimated Resources 67

4.10 Initial CHStone Application Results with Post-Mapped Resources for the Cyclone V FPGA Family 68

4.11 Initial CHStone Application Results with Post-Fitted Resources for the Cyclone V FPGA Family 69

4.12 Initial FIR Filter Results 69

4.13 Autotuner Configurations: Weights and Target Penalty Factor 70

4.14 Search Space Size of Each Test Program 72

4.15 Initial Time to Compile (Without Injected Optimizations) 72

4.16 Repeated Tests with Different Seeds 75

4.17 Same-Seed Results for mips 76

4.18 Same-Seed Results for mips 78

4.19 Relative Improvement Based on Estimated Resources 83

4.20 Relative Improvement Based on Post-Mapped Resources 84

4.21 Relative Improvement Based on Post-Fitted Resources 85

4.22 Initial CHStone Application Results Versus Best WNS Results 89

4.23 Initial CHStone Application Results Versus Best-In-Category Results 89

4.24 Modelsim Signals Generated By the Intel HLS Compiler 95

4.25 Summary of FIR Filter Synthesis Results 99

4.26 Summary of mips with DSP Block Targets Specified, Same Seed Configuration, Varying Penalty Factor 101

List of Figures

1.1 Internals of an FPGA [2] 1

1.2 LegUp Design Flow 7

1.3 Typical Design Flow When Using Vivado HLS [3]. 8

1.4 Typical Design Flow When Using the Intel HLS Compiler [4]. 10

1.5 Process Flow for Using OpenTuner 15

2.1 Pipelined Loop with Three Stages and Four Iterations [5] 29

3.1 Overview of the Major Components in the OpenTuner Framework [6] 41

3.2 OpenTuner's Exploration Phase and Exploitation Phase [7] 43

3.3 OpenTuner's Hierarchy of Included Parameters [6]. 44

3.4 Intel HLS Compilation Stages 48

3.5 Cyclone V Adaptive Logic Module (ALM) [8] 50

3.6 Design Flow When Performing a Full Quartus Compilation From Command Line or Tcl Script [9] 50

4.1 Same Seed Comparison of Tool Levels - mips 77

4.2 WNS Estimated Balanced Results 80

4.3 WNS Estimated Area Results 81

4.4 WNS Estimated Performance Results 82

4.5 CHStone Estimated Resources - ALUTs 86

4.6 CHStone Estimated Resources for adpcm 88

4.7 CHStone Post-Fitted Resources for mips 90

4.8 FIR Filter Estimated Results 92

4.9 FIR Filter Estimated vs Post-Fitted Results 93

4.10 FIR Filter Estimated vs Post-Fitted Results 94

4.11 FIR_Filter Verilog HDL Results 96

4.12 Intel HLS Initial Result - No Optimization Specified 96

4.13 Intel HLS Initial Result - Fully Unrolled 97

4.14 Intel HLS Initial Result - Partially Unrolled 97

4.15 Mips with a Target of 3 DSP Blocks and Varying Penalty Factors 102

C.1 WNS Estimated Balanced Results 116

C.2 WNS Estimated Area Results 117

C.3 WNS Estimated Performance Results 118

C.4 WNS Post-Mapped Balanced Results 119

C.5 WNS Post-Mapped Area Results 120

C.6 WNS Post-Mapped Performance Results 121

C.7 WNS Post-Fitted Balanced Results 122

C.8 WNS Post-Fitted Area Results 123

C.9 WNS Post-Fitted Performance Results 124

List of Symbols & Abbreviations

AC	Algorithmic C
ALM	Adaptive Logic Module
ALUT	Adaptive Lookup Table
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuit
AST	Abstract Syntax Tree
DPI	Direct Programming Interface
DSP	Digital Signal Processing
FF	Flip-Flop
FIFO	First In First Out
FIR	Finite Impulse Response
$\mathbf{F_{max}}$	Maximum Operating Frequency
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
GDB	GNU Project Debugger
HDL	Hardware Descriptive Language
HLS	High-Level Synthesis
IP	Intellectual Property
LE	Logic Element
LLVM	Low Level Virtual Machine
PLY	Python Lex-Yacc
RAM	Random-Access Memory
ROM	Read-Only Memory
RTL	Register Transfer Level
Tcl	Tool Command Language
VHDL	VHSIC Hardware Description Language
WNS	Weighted Normalized Sum
WNV	Weighted Normalized Value

1. Introduction

1.1 FPGAs

An FPGA (Field Programmable Gate Array) is a device used to produce a digital circuit that is configurable and programmable by a customer or designer. FPGAs can be used in similar places as other controller based circuits including microprocessors and microcontrollers, but can also be used to implement basic digital logic designs. To use an FPGA, the designer would typically write a program using a hardware descriptive language (HDL) in which the language translates directly to the hardware inside of the device.

An FPGA consists of three types of configurable elements: perimeter input/output blocks, a core array of configurable logic blocks, and the resources for interconnection of these blocks. Each configurable logic block can be composed of Flip-Flops (FFs), Look-up Tables (LUTs), blocks of RAM, and other application specific logic elements that vary by manufacturer and FPGA family. Each FPGA has a fixed number of configurable elements that can be used in a design, and some designs will tend to use more of one resource than another. For example, circuits that perform a lot of multiplications will benefit greatly from dedicated digital

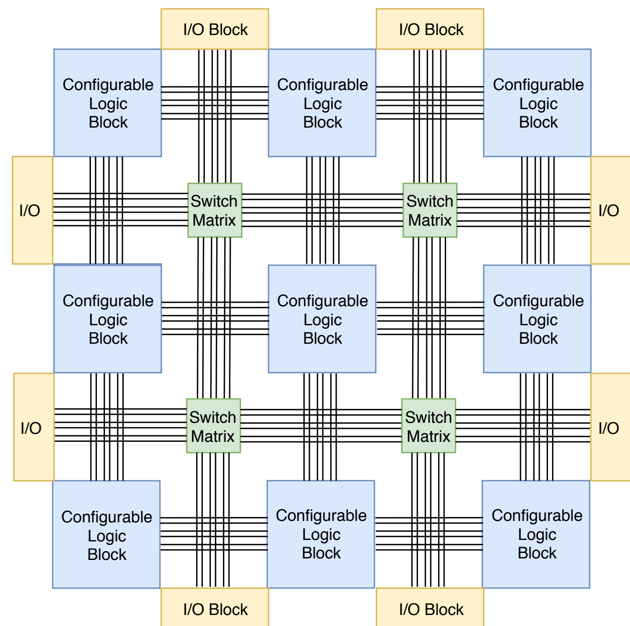


Figure 1.1: Internals of an FPGA [2]

signal processing (DSP) blocks that exist in many FPGAs. However, when all of the DSP blocks have been used, any additional DSP operations must be performed from generic lookup tables which can be very inefficient for large designs.

When using an FPGA to implement a digital circuit, the designer writes a program using a hardware descriptive language that instructs a compiler on how to connect hardware to perform the required functional task. The compiler will perform a series of analysis and synthesis steps to produce a hardware structure that takes into consideration the particular FPGA that is being targeted. It will optimize, re-configure the interconnects and wire the logic blocks together to generate a functional hardware equivalent of the HDL code. Since there are very few limitations on the way an FPGA can be configured, multiple individual circuits and designs can be configured on the same device. This allows the FPGA to handle application specific tasks and parallel data processing very well.

FPGAs are not the only option for implementing digital logic circuits. Two other common alternatives are microprocessors and ASICs. Microprocessors differ from FPGAs in that they are limited to processing data using preconstructed hardware that is not reconfigurable and use a fixed instruction set. This forces microprocessors to perform operations sequentially and limits the extent to which operations can be performed in parallel. FPGAs are much more flexible in design and can support a high degree of parallelism, enabling high data rate applications. One can even implement microprocessors inside of the FPGA (among other designs and custom instructions).

ASICs (application specific integrated circuits) are similar to FPGAs in that a hardware descriptive language is used to organize the hardware, but ASICs are not reprogrammable - the hardware is permanently etched into the silicon. For example, a computer's CPU is an ASIC. ASICs are very expensive to initially create, but in mass quantities they become the economical choice. ASICs also provide higher performance than FPGAs because the hardware can be etched into the silicon in an optimal configuration for the specific application, including the exact required resources for the design and creating shorter routing interconnects. ASICs offer the best performance and power efficiency, but are very unforgiving for future updates and time to market.

FPGAs were introduced in the 1980's and have become increasingly popular ever since due largely to their flexibility and reconfigurability. FPGAs provided industry with reduced upfront costs for lower quantity production runs than ASIC designs, and this gap continues to widen over time [10]. FPGAs have proven themselves to provide a great balance in terms of performance, time to market, cost, and reliability. However, FPGAs also have some challenges. First, the use of an FPGA requires knowledge of a hardware descriptive language such as VHDL or Verilog programming languages. These HDL languages are not as simple to use or learn as procedural languages (such as C/C++) and require knowledge of digital system fundamentals. Furthermore, achieving an efficient and high performance FPGA implementation requires detailed knowledge of the specific FPGA family so that the available logic resources can be used in the most efficient way. Relatively few engineers and computer scientists have this expertise, a fact which limits the market for FPGAs.

Designs have increased in size and complexity over time. As projects increase in size, so has the average percentage of FPGA project time spent in verification. The 2018 Wilson Research Group Functional Verification Study, commissioned by Mentor Graphics, shows an increase in average percentage of FPGA project time spent in verification which indicates an increase in verification complexity as designs grow. Similarly, the demand for verification engineers outpace the demand for design engineers [11, 12]. FPGAs are difficult to verify because of their parallel nature in design, and because of difficulty in testing timing constraints and considerations [13]. According to a study performed in 2018, only 16% of all FPGA projects were able to achieve no bug escapes into production, and almost half of those bugs are related to functional or logical problems. [12]. Procedural languages have many debugging tools such as GDB or Valgrind to verify functionality of the code, and timing considerations are rarely an issue in procedural languages. FPGAs do not have any source level debugging tools and often require the use of simulators, logic analyzers, or implementation tests to verify the designs. This limits the audience that FPGAs can be used by and as designs get larger, so do their costs to verify.

1.2 HLS

In an attempt to enable a larger audience to use FPGAs and reduce the time to verify FPGA designs, some companies have developed High Level Synthesis (HLS) compilers to assist in the development with FPGAs. HLS compilers take a more common procedural language such as C or C++, and convert the code to a hardware descriptive language or Register Transfer Level (RTL) solution that can be programmed directly to the FPGA. There are several HLS tools on the market. Some of these tools are commercially available, while others are open-source. The first generation of HLS tools originated in the 1990's, but these tools were not widely adopted because of limitations in their ability to generate RTL translations that adequately met the needs of the design from their input sources [14]. The complexity of the synthesis transformations increases greatly with the design size which puts more emphasis on the HLS tool and general coding style of the designer [15]. In more recent years, the popularity of using HLS tools has increased. Research is being performed in many different areas of using the tools and an overall improvement in synthesized results has occurred [16–18].

Recent major investments by FPGA companies have spawned two popular commercial HLS tools: Intel HLS Compiler (2017) and Vivado HLS (2012). IntelFPGA (formerly Altera) and Xilinx have a commanding share of the FPGA market, and have each developed their own HLS compiler which target their FPGAs [19]. Another HLS compiler which has gained some traction since its inception in 2011 is LegUp. LegUp was developed at the University of Toronto and has become a popular choice for research because of its academic and open-source nature [20].

This research focuses on the Intel HLS Compiler for two main reasons. First, it is one of the most recent HLS tools to reach the market, and it has not yet been explored to a large extent in research. Second, its architecture lends itself well to the design of FPGA-based accelerator modules, which are expected to be a major application for FPGAs in the future. The following sections discuss the Intel HLS Compiler and the competing HLS tools in more detail.

1.2.1 Intel HLS Compiler

The Intel HLS Compiler takes untimed C/C++ as input and generates production quality register transfer level (RTL) code that is optimized for Intel FPGAs. The Intel HLS Compiler aims to benefit the user by allowing for a higher level of abstraction as well as improved verification times. With this tool, conventional C/C++ development can be used to both develop and verify designs targeted for FPGA devices.

Intel HLS synthesizes a C/C++ function into an RTL design in the form of an Intellectual Property (IP) file. These generated IP files can be incorporated into Platform Designer or Quartus Prime, Intel's FPGA design software. A C/C++ source file is created by the designer which implements functions that can be marked as a *component*. The component is the designation to the Intel HLS Compiler that the function and its contents are to be converted into an RTL solution. The component directive has no impact when testing the code for functionality using a traditional C/C++ compiler, and a new compiler provided by Intel (i++) can be used to generate RTL code from the C/C++ code in the marked function. The designer can choose which functions are to be marked as a component in their design, and only the components will be synthesized into RTL. This allows the designer to also code a procedural based testbench in their C/C++ source code to allow for easier testing. Intel has also added the capability to perform functional testing through interfacing with their FPGA simulator: ModelSim - Intel FPGA Edition. This provides the advantage of allowing for more realistic FPGA functional verification to be performed in a procedural source file. The Intel HLS Compiler uses System Verilog Direct Programming Interface (DPI) to allow the C/C++ code in the hardware executable to interface with System Verilog via the Modelsim simulator. Then, an inter-process communication library is used to pass testbench input data to the RTL simulator and return the data back to the x86 testbench. Any call to the component function in the C program will automatically invoke Modelsim and communicate via DPI [5]. This allows for functional testing of a component as an individual function, but also as components in a larger FPGA design in which there can be multiple components operating in flight at the same time (procedural languages normally cannot handle parallel function calls). Each compile using i++ will (by default) generate a

hardware executable that can be used to test functionality, a Modelsim testbench that uses simulation to further verify functionality and timing requirements, and a Quartus project for actual FPGA implementation for each individual marked component.

As previously discussed, there are many ways to implement a given hardware function inside an FPGA. FPGA developers make many design decisions when writing HDL in order to tailor the implementation to the user's cost and performance requirements. In an HLS design flow, the compiler generates RTL directly from C/C++. The conversion of a C/C++ function into an RTL implementation is only partially optimized by default. The compiler will make many assumptions in the process and will automatically perform optimizations based on those assumptions. In most cases, the C/C++ code will need to be adjusted by placing Intel HLS Compiler specific directives in the code to help direct the compiler's translation process. After several (manual) iterations of optimization involving changes to the C code and compiler optimizations, the goal is for the result to be similar in cost and performance to a solution written in HDL. Intel's target is to be within 10-15% resource usage in comparison to hand-written HDL design. [21].

1.2.2 Other HLS Tools

Although the main focus of this research is the Intel HLS Compiler, this section provides background information on other HLS tools available on the market. According to a survey performed in 2016, there are over 17 active HLS tools in use [22]. Each of these tools differ by input source file language, techniques used, FPGAs supported, and domains that they are targeting. For example, CoDeveloper is specifically designed for the image streaming domain. Others, such as VivadoHLS and LegUp are designed for all domains. Most of these tools use a variation of C as an input language to the tool (including C++, C#, and SystemC) and output a hardware descriptive language equivalent or RTL solution. For comparison, a popular academic tool and a popular commercial tool will be discussed: LegUp and VivadoHLS.

LegUp

LegUp is an open-source high-level synthesis research infrastructure being actively developed at the University of Toronto since early 2010. LegUp can translate a C program into verilog which can then be verified using a simulator. Similar to Intel HLS, LegUp requires Modelsim and Quartus to synthesize the verilog for an FPGA. LegUp can compile an entire C program to hardware, or it can compile user designated functions to hardware while the remaining program segments are executed in software on the soft TigerMIPS processor or ARM processor such as the one available on the DE1-SoC board [23].

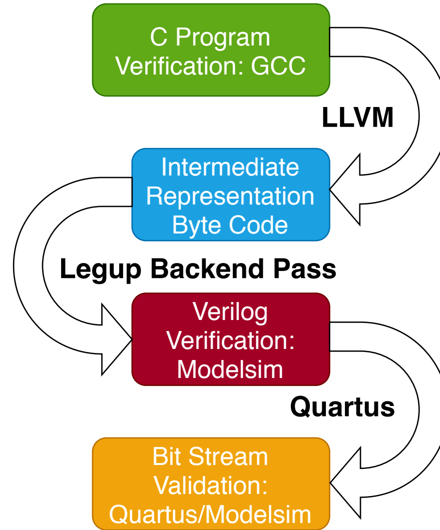


Figure 1.2: LegUp Design Flow

LegUp uses Low Level Virtual Machine (LLVM) to translate the source code to byte code which is an intermediate representation of the source file. LLVM offers more flexibility than GCC because it is modular, easily allows for additional compiler passes, and allows for intermediate representation of the file after each pass [23], [24]. Optimizations for LegUp are performed during the LLVM intermediate representations of each compiler pass. Unlike Intel HLS, insertions into the source files are not usually performed. LegUp also outputs a user-readable verilog file, as opposed to the Intel HLS Compiler's IP file which keeps the actual verilog implementation abstracted from the designer [23].

LegUp also has some limitations on standard C programming: it does not support recursive functions or dynamic memory. Functions, arrays, global variables, floating arithmetic, and pointers are all supported.

Vivado HLS

Vivado is also based upon LLVM compilations. Similar to LegUp, Vivado originated in 2011 and includes a full design environment. Vivado supports more languages, allowing for

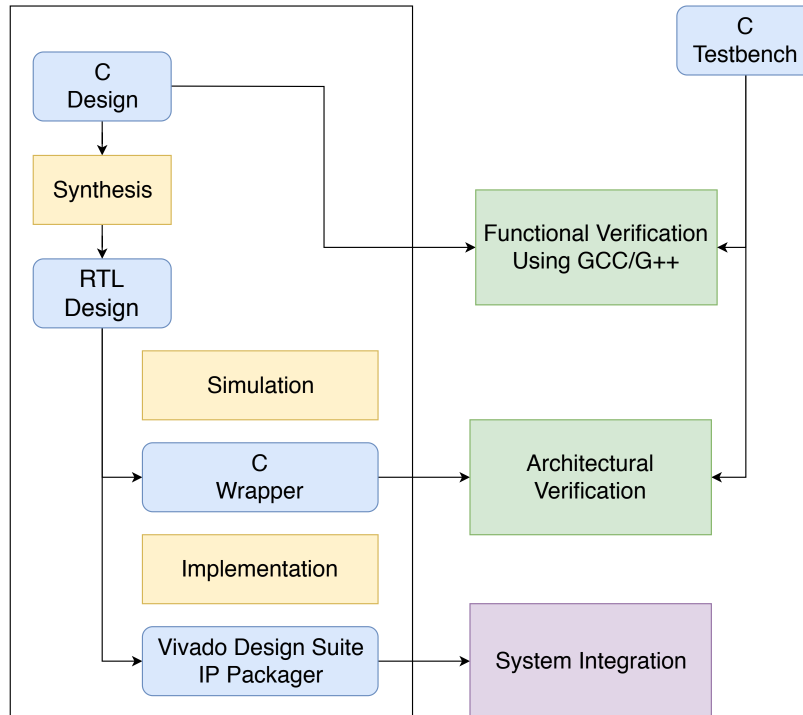


Figure 1.3: Typical Design Flow When Using Vivado HLS [3].

C/C++ and System C as inputs, and outputs VHDL, Verilog, and SystemC. Optimizations are performed during the LLVM compilation process, which is similar to LegUp [3].

Optimizations in Vivado are performed through specific directives used in the design environment. Optimizations can also be specified through TCL scripts from a command line. Vivado only supports Xilinx FPGAs. Similar to Intel HLS, Vivado outputs IP files which are designed to be incorporated into other solutions or to be programmed to a Xilinx FPGA directly [25].

1.2.3 Design Flow Using Intel HLS

Design using an HLS compiler begins with developing a C/C++ testbench and component. The component in this case is simply a function which implements the algorithm to be synthesized, which is usually a small component or portion of a larger design. The algorithm in the component is developed and verified until the C/C++ code is functionally correct. At this stage, the component can be debugged using standard C debugging tools.

The second stage of the design process involves optimizing the design for specific FPGA targets and features. This step generates RTL code for the component and gives estimates for resource usage. The designer at this stage can do initial optimizations based on resource usage from reports generated by the Intel HLS Compiler. By manually using code restructuring, compiler directives, and Intel HLS Compiler specific constructs, a semi-optimized design can be achieved. However, due to the extra layer of abstraction (the addition of procedural code) it can be difficult to know the impact of a change from the procedural language (C/C++) to the final synthesized FPGA design. This causes the designer to repeat the optimization step several times until a satisfactory result is achieved. Furthermore, this optimization requires detailed FPGA expertise, which is contrary to the goal of making FPGAs more accessible to a wider audience.

After optimization, verification of the design using simulation-based tools (Modelsim) allows for additional verification of the design functionality [4]. Following this, full synthesis of the design in Quartus Prime is used to get accurate quality of result metrics such as F_{max} and resource usage results. Further iterations of updates and optimization occur until the designer has found a design which meets all of the requirements of the design specification. Finally, the IP file for the component can be extracted and used with Intel Quartus Prime or Platform Designer for larger system integration.

The procedure can be summarized as follows:

1. Create C/C++ code and testbench.
2. Emulate, test, and debug C/C++ code using conventional C/C++ tools.
3. After functional correctness is verified, mark functions to be synthesized as *component* and recompile using i++.
4. Review generated reports from co-simulation and accompanying Modelsim files. These reports give indications for what/where to optimize the C/C++ code, and the Modelsim files are used to verify HLS constructs, HLS attributes, resets and reset conditions.
5. Update the C/C++ code to improve flow or use special compiler directives to optimize the *component* functions.

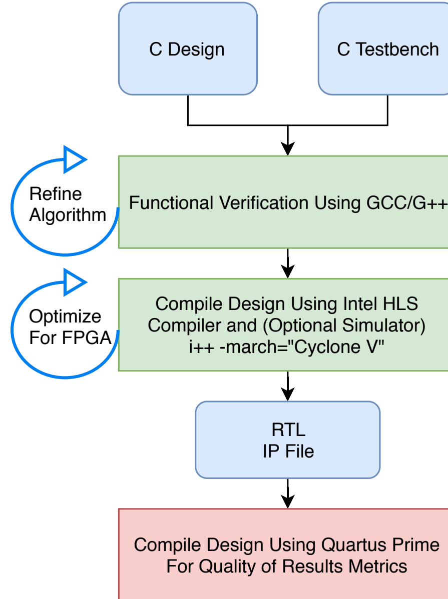


Figure 1.4: Typical Design Flow When Using the Intel HLS Compiler [4].

6. Re-iterate until satisfied with performance and resource usage (repeat for each component).
7. Compile IP in Quartus to generate more accurate resource usage and synthesis report, as well as generate an F_{max} for the design.
8. Verify the solution at the RTL level for reset conditions and timing violations.
9. Integrate the IP file into an HDL project or FPGA system.

1.2.4 Intel HLS Compiler Advantages Over Traditional HDL Development

HLS tools offer many advantages to developers, mostly in the realm of improved development times and ease of use [26]. Some of these advantages include:

Code Development: In most situations, the sequential and easy to follow flow of C/C++ code leads to faster development and verification of algorithms. The ability to implement different algorithms quickly allows the developer to determine which will have the best performance measurements and metrics without the need for a lengthy HDL coding cycle [11].

Debugging: Traditional debugging tools such as print statements, GDB, or Valgrind offer quick line-by-line troubleshooting which is currently unparalleled in the HDL development environments. Problems such as memory leaks, invalid pointers, uninitialized values, and memory allocations are easier to troubleshoot using traditional methods [12].

Less RTL Knowledge Needed: Sequential programming is easier to learn and deploy. Use of HDL tools requires a knowledgeable background in HDL, synthesis, and RTL circuits.

Automation: The Intel HLS Compiler provides scripts that automatically generate Modelsim testbenches to verify RTL circuits generated.

Floating Point Arithmetic: Floating point arithmetic has no order/combination limitations because it is not architecture specific when using the Intel HLS Compiler [1].

Development Phases: Troubleshooting an HDL design is difficult because issues can be related to functionality, the interface, or the timing. Development using an HLS tool allows for the segregation of each of these items, which reduces the debugging scope and simplifies the debugging process.

1.2.5 Challenges of Using the Intel HLS Compiler

HLS compilers, including Intel's HLS Compiler, have several challenges for the designer to overcome. Some of these challenges include:

Result Efficiency: The compiler usually generates less efficient results in terms of performance and resources used; a typical goal is to be within 10-15% of a hand-coded HDL solution if proper optimization techniques are followed [21].

Iterations: Most designs using the Intel HLS Compiler will require a multi-iterative procedure which can reduce some of the benefit of using an HLS tool.

Optimization Techniques: Although the developer will require less knowledge of RTL synthesis, some knowledge of hardware and Intel HLS Compiler specific optimization techniques is required to get to target performance and resource usage.

Timing Violations: Functional equivalent circuits can be developed using the Intel HLS Compiler, but some issues can only be solved using HDL simulation tools such as Modelsim. For example, reset conditions and timing violations are not observed in C/C++ development.

Coding Limitations: Although C/C++ is used as the input into the Intel HLS Compiler, some limitations on coding style and standard coding constructs exist. For example, loops cannot be unrolled if their count is dynamic. Pointer arithmetic is also not supported [1].

Multiple Loops: A current limitation of the Intel HLS Compiler is regarding loops that are coded at the same level. Multiple loops cannot be ran in parallel, although they can still be pipelined. To achieve a more efficient design, the developer would have to place each loop body in their own component. This further restricts coding style.

Abstraction: Intel HLS provides a layer of abstraction from the RTL synthesis. Although abstraction can be a good for improving modularity of code, it can also make it difficult for the developer to meet specific targets and goals of the finished design. Higher levels of abstraction make it more difficult to evaluate the effects of changes and code style, particularly if a resource usage target or performance metric is to be met [27].

Manual Intervention: Many optimizations require the user to manually adjust or correct the C/C++ code. This adjustment can become cumbersome and time consuming, and generally requires FPGA specific expertise. For example, data type sizing, memory space allocations, establishing dependencies, and trade-offs between performance and resources used.

Memory: Intel HLS has many limitations and complications related to memory. When merging data, intermediate storage is needed. Manual specification of concurrent invocations is needed to take advantage of sharing memory. Intel HLS does not support dynamic memory allocation, and only a single dimension of a multidimensional array is used to infer banking configurations as opposed to analyzing each dimension for optimal configuration [1].

Resource Usage Trade-off: Intel's tools have predetermined default optimizations that will attempt to fit designs in accordance to their preference. The designer may, on the other hand, prefer different allocations. For example, a designer may prefer save some DSP blocks by placing some of the design in LUTs instead. This preference must be manually specified, and the trade-off of DSP blocks to LUTs will be unknown until a synthesis is performed. Furthermore, manual repetitive iterations will also need to be performed until the correct balance is achieved. This process can be very time consuming, especially in larger designs.

Inability to Specify Design Targets: The Intel HLS Compiler, when used in conjunction with Quartus, only allows for three different optimization schemes: area, performance, and balanced. Usually there is a performance decrease when less resources are used, and vice versa. These schemes will bias the solution to either use less resources, perform better, or attempt to balance this trade-off. However, it does not take into consideration any specific requirements that should be met in a particular design, and does not offer any variation in the trade-off. In some cases the optimal solution from the designer's perspective may never be achieved because of the rigidity of the optimization schemes.

With all of the challenges of using an HLS tool, it is apparent that more can be done to improve the usability and performance of these tools. The central idea of this research is to improve these tools by automating some of the manual portions of the optimization procedure, and introducing a method of allowing the designer to explore different trade-offs of resource usage automatically. To automate the Intel HLS Compiler and the optimization process, several additional tools will be introduced: OpenTuner, PyCParser and Docker.

1.3 Intel HLS Compiler Automation

1.3.1 Autotuning Using OpenTuner

HLS compilers started making an appearance in the 1990s but did not gain any real traction until the early 2000s. This is due to a number of reasons including: targeting the

wrong audience, lack of support, and hard to validate results [14]. In more recent years, popularity of these compilers has increased but they still struggle with the requirement of manual feedback from the programmer to create optimal and usable solutions which makes the process manual, heavily interactive, and cumbersome. In an attempt to reduce the manual portion of the iterative process, it is hoped that a software solution can be used to automate the exploration of the design space and help achieve target goals in the design.

In general, the process of adjusting program parameters and running the program after each adjustment is known as autotuning. The goal is to improve the program's outcome by finding a desirable set of configuration parameters. This research aims to apply the autotuning concept to HLS compilation. Several software packages, known as autotuners, exist to facilitate this type of design space exploration across various domains. OpenTuner is the first to introduce a general framework to describe complex search spaces for program autotuning [6].

This research attempts to use OpenTuner to autotune the Intel HLS Compiler to assist with the generation of different FPGA configurations that meet targets specified by the user. For example, the user should be able to request the cheapest possible implementation of a module that can run at a clock speed of at least 200Mhz. It is believed that OpenTuner can be modified to assist in this search, which would currently require a great deal of manual intervention. OpenTuner also has the advantages of utilizing an ensemble of search techniques which work together to find an optimal solution, and a database of results is provided for post processing by the designer. Autotuning requires performing multiple compilations and can take a significant amount of time to perform, especially when autotuning HLS tools that are themselves slow to run. This drawback, however, is offset by the automated capabilities that allow the autotuner to run over night or when computers are not normally being used and reducing manual developer effort.

How OpenTuner Works

OpenTuner provides the framework for autotuning. It has all of the components needed to perform autotuning without integration of domain specific requirements [6]. In the current

OpenTuner framework there are three major components that are provided: a configuration manipulator that is responsible for determining configurations (sets of compilation parameters) to run from a search space, a measurement interface that is used to quantify the results of a configuration, and a database.

To use OpenTuner, the user first defines the search space by creating a configuration manipulator. This manipulator contains a user specified list of parameters that will be controlled by the manipulator. In the scope of this research, the configuration parameters are automatically generated to be passed to the Intel HLS Compiler. The parameters are Intel HLS Compiler specific directives that will change the way the compiler synthesizes the RTL solution. These parameters can be either primitive (int, float, etc.) or complex (bool, enum, etc.). Each parameter used in the configuration could have an impact on the synthesized solution of the compiler, providing a new synthesized solution that yields different resources used and performance metrics (for better or for worse).

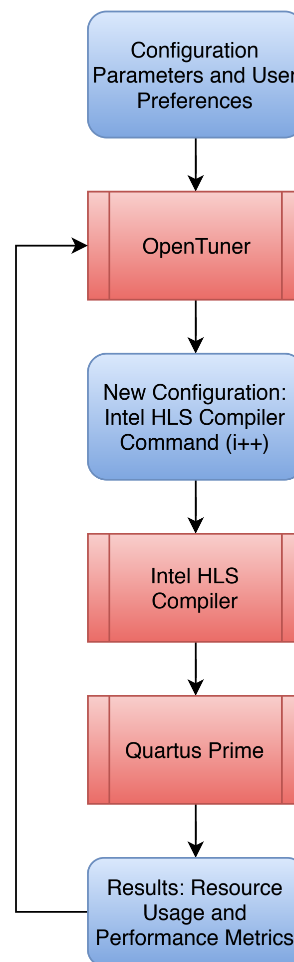


Figure 1.5: Process Flow for Using OpenTuner

The user can also specify which search techniques are to be used; these techniques utilize the results from the database of previous runs to determine what the next configuration should be. OpenTuner supports the use of multiple search techniques, and has many techniques already built into the framework. It also supports the addition of user defined search techniques. Once a configuration has ran, the results are passed to a measurement interface where the results are compared to user-defined objectives. These objectives set the premise of how *good* a configuration is. OpenTuner supports multiple objectives, and several variants on each objective; a typical use case is an object which minimizes time; however, compilation time is not a useful metric for

the present application. To make use of OpenTuner for HLS optimization, the OpenTuner codebase must be extended to allow FPGA resource usage targets to be specified.

Each run of a configuration will yield a result based upon a user-specified objective. In the context of this research a calculated value that reflects the resource utilization and performance measurements is defined. The autotuner framework then attempts to adjust configuration parameters in order to minimize the target metric. As more results populate the database, the autotuner is able to deduce which techniques are typically yielding better results and will bias itself to using the best-performing search technique [6]. The end result of the autotuner will be a new C source file with automatically injected parameters and a list of values for those parameters that yield a particular synthesized RTL solution.

1.3.2 PyCParser: Parsing C Code and Injecting Compiler Directives

The Intel HLS Compiler currently supports both C and C++ as inputs to the translator, but this research focuses on C. The reason is that C is currently the dominant choice in embedded system design. By focusing on C, this research may enable a vast library of legacy C code to be ported to FPGAs. C++ is gaining popularity in DSP systems and multicore embedded systems, but only proves efficient when code space is abundant [28]. In the particular use case of autotuning C code for the Intel HLS Compiler, the C code needs to be parsed and conditioned to accept tuning parameters specific to the needs of the Intel HLS Compiler. In this case, most optimization strategies involve user specified compiler directives, Intel HLS Compiler specific constructs, or in-line code directives using *pragmas* [4]. A portion of this research includes assisting the user by automatically detecting the portions of the C code which are candidates for optimization, then inserting placeholders within the C code for the auto-generated configuration parameters. In order to automatically add the placeholders, this research uses a C parsing tool called PyCParser to interpret the C file. PyCParser is a complete parser of the C language, written in pure Python using Python Lex-Yacc (PLY) parsing library. It parses C code into an Abstract Syntax Tree (AST) and can serve as a front-end for C compilers or analysis tools [29].

1.3.3 Docker

This research includes the use of several different tools (OpenTuner, Intel HLS Compiler, Quartus Prime, PyCParser) that have all been designed to be used in different working environments. In order to get everything to run properly on one system, a complex setup of virtual environments and specific library versions is required. To assist in replicating this particular environment, a program called Docker is used. Docker is a tool which creates a specific environment to run applications in objects called containers. Containers allow a developer to package up an application with all of its requirements (such as specific library versions and dependencies) and deploy them as a one-touch setup. This is particularly useful for this application because of the specific working environment needed, as well as simplifying the process of adjusting autotuner runs and automating tests. Docker provides a performance advantage over virtual machines because it allows for the full utilization of a computer's available resources [30]; a feature that is needed because of the resource heavy utilization of Quartus and the Intel HLS Compiler.

1.4 Research Objective

The primary objective of this research is to investigate whether the translation of HLS to RTL provided by the Intel HLS Compiler can be improved without requiring specific FPGA expertise. Improvements in the translation can be interpreted as either better performance, more efficient translations which lead to reduced resources needed, or translations that are tailored to specific design requirements (such as resource usage limits for a module) that would normally be ignored in the standard translation process provided by the Intel HLS Compiler.

The HLS optimizations under consideration target several different aspects of FPGA design including loop manipulation, memory interfacing, memory optimization, data typing and sizing, and Intel's recommended good coding practices for use with the Intel HLS Compiler. An investigation into which of these optimizations can be performed automatically without user intervention and whether these automatic optimizations have an impact on

functional results will be performed. An investigation into the types of designs that are best tailored to these optimization techniques, as well as further investigation into the impact of specifying target requirements for a practical design environment will occur.

In order to achieve these objectives, a system which can automate the Intel HLS Compiler optimization process will be constructed. This will involve the following major steps:

- Provide an interface for the designer to specify design requirements, targets, and preferences
- Apply PyCParser to inject placeholders around specific coding constructs in the given C source file based upon the preferences provided by the designer and pass a list of tuneable parameters and their associated ranges to OpenTuner
- Extend OpenTuner to dynamically create a search space based upon the given parameters and ranges
- Use OpenTuner to construct an Intel HLS Compiler command that passes a configuration's list of parameters to the HLS tool for the Intel HLS Compiler to run
- Develop a cost function based upon the results of the Intel HLS Compiler run that represents how *good* the result is in comparison to an initial (untuned) result and pass this value to the objective metric within OpenTuner
- Collect data over a variety of programming domains for analysis to determine if the autotuner is finding configurations that show improvements over the initial untuned result, and determine which designs the autotuner is conducive to provide better (or worse) results
- Introduce a new cost function that includes specific design targets based upon a design's requirements and see if the autotuner is able to find synthesized solutions that are optimized to a scheme but also meet the specified design targets

1.5 Thesis Outline

This thesis is divided up into five chapters:

Chapter 1: Intel HLS Compiler and the OpenTuner Framework

Chapter 2: Autotuning Parameters

Chapter 3: OpenTuner Configuration and Execution

Chapter 4: Autotuning Results

Chapter 5: Thesis Summary and Conclusions

Chapter 1 provides details into the background of the environment and domain being autotuned. This chapter discusses what the Intel HLS Compiler is, why there is a need for such a tool, and some of the challenges that are included with the tool. To create an environment to autotune the Intel HLS Compiler, several different components work together to provide pre-conditioning of the input program, the OpenTuner Framework for generating different configurations and runs for the instance, and containerizing of the environment for easier deployment and testing. Each of these components are introduced with relevant background information.

Chapter 2 focuses on a more in-depth view of the autotuning environment, as well as a detailed explanation of what the Intel HLS Compiler does and how this tool can be optimized to generate a final optimized RTL solution. Each of the optional optimization parameters are explained and a discussion of which parameters chosen to be autotuned by the OpenTuner Framework ensues. Our desire to analyze the effects of changing each of these chosen parameters leads to the development and design of a full autotuning environment that yields a more flexible and usable translation of the input program to the Intel HLS Compiler.

Chapter 3 details what the results of each autotuner run produces, as well as how the OpenTuner Framework chooses subsequent configurations based on previous results. Several parameters and their impact on the final RTL design are explored. Configuration options to

the environment are introduced to give the designer more flexibility in the direction of the autotuner, and the effects of the parameters are discussed from a practical design perspective.

Chapter 4 discusses the results of autotuning several applications across different domains such as those presented in the CHStone benchmarking applications and in that of DSP filters. This chapter also elaborates more on the practical design configuration options, and introduces the concept of targets based on requirement specifications from the designer. Further investigation shows the effect on an autotuning run when targets are added, and how the final RTL result can benefit from them.

Chapter 5 summarizes the thesis, covers the research done, and highlights the results. Following the summary is a list of the contributions of the thesis. Some suggestions for future work conclude the document.

2. Autotuning Parameters

2.1 Introduction

As mentioned in the previous chapter, there are many optimization and configuration options provided by the Intel HLS Compiler which can control how a C algorithm is converted into RTL for implementation on an FPGA. The focus of this research is on finding ways to automate the design space exploration process through autotuning. This chapter introduces many of the optimization techniques available in the Intel HLS Compiler, and evaluates which of these are suitable for inclusion in the autotuning scheme. The process of parsing the C code to detect potentially suitable optimizations and inject the necessary placeholders for autotuning is also discussed.

2.2 Intel HLS Optimization Techniques

Intel provides ample documentation on optimization strategies and areas for their Intel HLS Compiler; manual optimizations to the C source files are needed in most situations to reach their target goal of 10-15% higher resource usage in comparison to a hand-coded HDL solution. These optimizations are broken up into the following categories: HLS interfaces, data types, local memory optimizations, loop optimizations, and good coding practice related to optimizing the Intel HLS Compiler [1].

2.2.1 HLS Interfaces

HLS components are essentially just ordinary C/C++ functions which can be called in a standard C/C++ fashion. When passing information into or out of a component, standard pointer notation can be used. This can be greatly optimized by changing how the information

is passed. There are two main types of interfaces supported by the Intel HLS Compiler: avalon streaming interfaces and avalon memory mapped interfaces. Streaming interfaces provide point-to-point unidirectional data flow that is synchronous to a clock signal. Memory mapped interfaces provide a master/slave configuration in which communication is done by an interconnect. All pointer interfaces become memory mapped interfaces and default to 64 bit addresses unless otherwise specified. For example:

Default Interface Code	**Optimized Interface Code**

component **void** vector_add(**int*** a,
int* b,
int* c,
int N)
{
#pragma unroll 8
for (**int** i = 0; i < N; ++i) {
c[i] = a[i] + b[i];
}
}
	component **void** vector_add(
	
	ihc :: mm_master<**int**, ihc::aspace<1>,
	ihc :: dwidth<8*8*__sizeof__(**int**)>,
	ihc :: align<8*__sizeof__(**int**)> >& a,
	
	ihc :: mm_master<**int**, ihc::aspace<2>,
	ihc :: dwidth<8*8*__sizeof__(**int**)>,
	ihc :: align<8*__sizeof__(**int**)> >& b,
	
	ihc :: mm_master<**int**, ihc::aspace<3>,
	ihc :: dwidth<8*8*__sizeof__(**int**)>,
	ihc :: align<8*__sizeof__(**int**)> >& c,
	
	int N)
	{
	#pragma unroll 8
	for (**int** i = 0; i < N; ++i) {
	c[i] = a[i] + b[i];
	}
	}

Table 2.1: Optimizing Interfaces to Components [1]

Table 2.1 includes default code that includes three pointer inputs to the component. By default, all three pointers would be mapped to a single avalon memory mapped interface with a 64-bit wide data bus, but this would cause stalling due to multiple accesses through the same memory interface and extra digital logic to incorporate stallable arbitration logic to schedule these accesses is required. To optimize this, three separate memory mapped interfaces could be used to prevent the memory stalling, and correct bit sizing for the interfaces could be specified as shown in the optimized interface code.

2.2.2 Data Types

The data types used to represent variables in a C component can have a significant impact on the FPGA RTL representation after HLS. While standard C data types are fully supported, it is also possible to use the Algorithmic C (AC) data types that Mentor Graphics provides under the Apache license [5]. Intel has developed optimized versions of the AC data types to allow the Intel HLS Compiler to generate efficient hardware on Intel FPGAs. The key advantages of using these data types over standard implementations are:

- Likely smaller resource consumption
- Custom bit widths for variables
- Support for larger than 64 bit types
- Improved handling for integer promotion
- Special API functions provided by Mentor Graphics may be used

Floating point arithmetic can also be optimized. By default, floating point numbers must adhere to the floating point standard (IEEE 754) [31], but if a design is robust enough to deal with small inaccuracies in floating point arithmetic, the floating point relaxed and floating point conversion options can be passed to the Intel HLS Compiler. The floating point conversion flag allows for intermediate rounding and conversion operations during chained floating point operations when DSP blocks cannot be used, but the design would

no longer follow the (IEEE 754) standard. The floating point relaxed flag allows for the compiler to generate shallower adder trees for sequential additions [32]. The floating point relaxed option also allows the compiler to use hardened dot-product IPs when it detects a pattern that would benefit from them. This is more efficient than traditional chaining of multiply-adds.

Use of the AC data types does require manual insertion of the correct bit sizes, and knowledge of what the appropriate size for a data type or variable is needed. Adjusting the bit sizes creates a risk of breaking code functionality and therefore they should be used with caution and additional verification needs to be performed after making adjustments.

2.2.3 Local Memory Optimizations

The Intel HLS Compiler will attempt to automatically configure and optimize declared variables and arrays into local memory blocks in the FPGA. Most of the Intel FPGAs use the M10K or M20K architecture for local memory which can be used for either RAM or ROM applications. Although the compiler attempts to select appropriate memory settings based on information available to it, there are several considerations related to local memory optimizations:

- Local memory translation is dependent on the C data type used by the programmer.
- Local memory cannot be dynamically allocated. Architecture (including banking config, width, depth, and interconnect) is customized at compile time.
- Performance is dependent on the banking configuration and interconnect bus configuration. The Intel HLS Compiler will automatically configure the memory banks and the interconnects, but they can also be manually customized to improve performance.
- Memory is usually organized into 10Kb or 20Kb blocks of dedicated memory resource based on the M10k or M20k structure available in the FPGA.
- Local memory blocks can support multiple concurrent accesses through a dual port mode.

- Local memory can be combined to make larger sizes in both width and depth.
- Local memory can be forced into single or dual port mode. Each block is 20 bits wide in dual port mode or 40 bits in single port mode.
- Local memory can be used to make shift registers
- Using FIFO buffers if available
- Local memory supports customizable read-during-write operations

When writing C code, attention should be paid to how many loads and stores are used on a port. With no optimization, sharing ports can greatly reduce performance [4]. This is because pipelines stall due to arbitration for concurrent accesses. The key to high local-memory efficiency is stall-free memory accesses. In general, there are several ways to optimize memory to achieve this goal. The compiler will automatically use replication and double pumping, which are the two main forms of local memory optimization. The compiler will also analyze access patterns on memory and will use coalescing, splitting, banking optimization and port sharing to optimize the memory. The advantages of creating stall-free memory include:

- Fixed and reduced latency
- Fewer resources are used
- Stall-free memory can be included in stall-free execution regions of a pipeline
- Simpler interconnects when no arbitration is needed
- Memory access can be scheduled more efficiently.

The Intel HLS Compiler employs several techniques to manipulate and improve memory performance. These techniques include:

Double Pumping This technique doubles the effective number of ports in a memory by using twice the clock frequency for operations. In consequence, some additional multiplexers are used to route data into and out of the memory block.

Replication Replication involves using multiple blocks to hold the same memory. This is particularly useful for memory which will be read often. The major consequence to this technique is stores need to go to both blocks. Replication is always done automatically by the compiler and is transparent to the user. It usually results in a simpler interconnect and has no negative impacts on F_{max}. The only downside to replication is the additional resource usage. Components in the Intel HLS Compiler are infinite loops that never end. Each component can also have multiple invocations which would require replication of memory as well. By default, the compiler will replicate memory for each invocation, but the designer can tell the compiler the number of max concurrent invocations using <hls_max_concurrency(N)>. The compiler will then only replicate the necessary amount of memory needed, and will re-use memory for other invocations.

Coalescing Arrays which are used in concurrent accesses can be merged into larger words, depending on the data type sizes. This will reduce the number of accesses needed for the operations.

Banking In the event that an array is not used with consecutive accesses, the compiler will automatically separate memory into different areas to optimize access. For example, a double array might be broken down into two single arrays to take advantage of banking. An N-bank configuration can handle 4·N requests per clock as long as each request address is on a different bank (assumes dual port, double pumping and no replication are used). A requirement of banking (and coalescing) is that no potential out-of-bounds index addressing can occur. To convince the compiler that no out-of-bounds condition will occur, proper masking of index arguments must be used (for example, A[x] vs A[x & 0x3]). The compiler will assume the lower index of a multidimensional array is the known index. Manual configuration is needed if this is not true (for example, A[x][i] vs A[i][x]).

Splitting In an optimal configuration, each variable would get its own memory system that would include separate addresses and ports. For arrays, bottlenecks can occur on memory access. However, if the array is split into separate memory systems, it can perform better. The compiler will automatically split arrays, but can only do so if it can prove that a pointer refers to exactly one array.

Port Sharing Arbitration logic is needed when there are more read and write sites than ports available. The Intel HLS Compiler will determine if two mutually exclusive operations can be connected to the same port. For example, if the logic is pipelined or when two different for-loop blocks are used sequentially.

Registers Memory can be forced into registers as well. Registers provide fast access and are always stall-free. They are useful for small variables, arrays and scalars because of their fast access. If loops are unrolled, arrays are usually stored in registers. Many devices also provide shift registers which are ideal for hardware optimization. They are automatically inferred from access patterns, and Intel HLS Compiler must recognize the pattern for the shift register to be optimized in hardware.

By default, the Intel HLS Compiler makes memory-based optimization choices automatically based on the number of accesses. Specifically, the compiler may enable any or all of the options listed above. When choosing memory options, the compiler places in order of priority: stall-free memory access and stores, improving overall F_{max} in the design, and lastly optimizing for lower resource utilization. However, the user may wish to prioritize these goals differently and enable a different set of memory options. Manual intervention in the form of compiler directives placed in the C code can be used to force a particular memory configuration.

2.2.4 Loop Optimizations

There are four types of loop optimizations which can influence the implementation of a loop in C.

Data Optimizing the data such that instructions operate on different pieces of data. GPUs

are good at this.

Thread Level Multiple threads execute concurrently to execute instructions in parallel. Multi-core CPUs are good at this.

Instruction Level Executing multiple instructions at the same time.

Pipeline Parallelism Multiple instructions are in flight at the same time, but are executing different parts of the instruction. Most modern processors have pipeline stages and FPGAs are the best at this type of parallelism.

The Intel HLS Compiler automatically analyzes each loop in the C source file for dependencies. It attempts to reduce the amount of clock cycles needed by pipelining the stages and launch the next iteration as soon as possible. This tends to improve the performance without extra hardware. The Intel HLS Compiler will also pipeline components because each component is treated as an infinite loop. Each instruction that is found to be independent can be parallelized and each sequential instruction can be stamped out and executed in a pipelined fashion. In the ideal case, a new iteration is started on each clock cycle. The number of clock cycles needed to start a new iteration is referred to as the iteration interval. A high iteration interval will create a bottleneck in the execution flow of a program.

Loops can also be *unrolled*. Unrolling is the repetition of hardware to allow parallelism. This can only be performed if no dynamic loop count variables are used. This results in a much higher performance/throughput in the component, at the compromise of increased hardware resource utilization. The entire loop can be unrolled, or the loop can be unrolled into segments. Unrolling does not always have a linear relationship for how much to unroll verses performance and area gains. This is caused by potentially created increased critical path delays that are difficult to account for due to actual layout considerations and FPGA specific hardware blocks [33].

Nested loops can create complex analysis unless most or all of the logic is located in the inner-most loop. If logic is found in multiple layers of the loop, the compiler usually results in much higher iteration interval values.

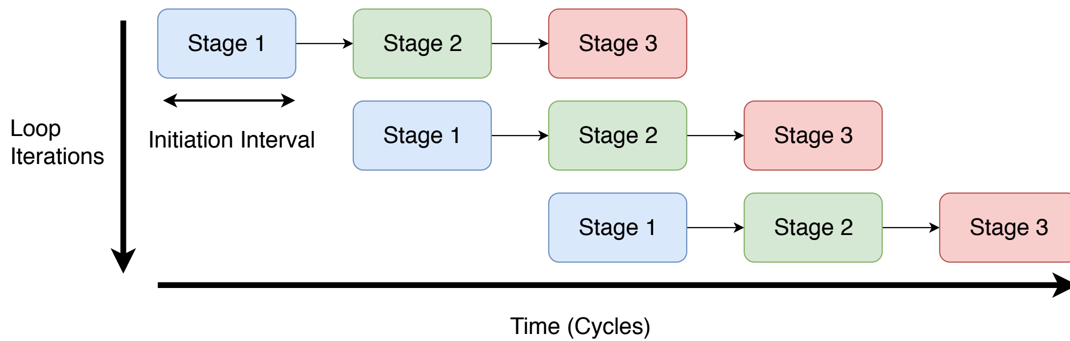


Figure 2.1: Pipelined Loop with Three Stages and Four Iterations [5]

Loops can be manually optimized using a number of compiler specific directives. These directives, or *pragmas* are a means for the programmer to pass information to the compiler. As of this date, all pragmas are related to loop unrolling and include:

#pragma unroll <n> Unrolls the loop by replicating hardware to increase performance. The compiler does this automatically, but can be forced to unroll a specific amount by specifying <n>.

#pragma ivdep Asserts memory array accesses are independent across loops. The designer becomes responsible for functionality as dependencies are ignored by the compiler. The designer can also specify a safe length option. This guarantees the compiler that there are no dependencies for the next 'n' iterations.

#pragma loop_coalesce <n> Combine nested loops to help reduce overhead and resources. The number of levels to coalesce is specified by 'n'. Although this can save resources, it can also create complex loop exit conditions which can increase the iteration interval.

#pragma II <n> Specify a new iteration interval that is greater than the current amount.

#pragma max_concurrency <n> Allows for 'n' iterations to be in flight at one time. By default the compiler will maximize this value, but the user can reduce it if needed to match timings of other components. Reducing this will sacrifice performance for local memory savings which is useful for non-critical loops.

2.2.5 Coding Practice Related to HLS Optimization

Intel provides an entire document dedicated to coding practices that will improve the HLS translation. In order for the translation to reach target goals, the designer must be aware of the nuances and restrictions of the Intel HLS Compiler. A summary of some of the key concepts include:

- Avoid pointer aliasing. The compiler does not make assumptions when using pointers, and is unable to resolve dependencies which leads to the inability to synthesize or optimize. In some cases which are unavoidable, the 'restrict' keyword can be used to tell the compiler that there will never be a dependency between two variables.
- Construct well-formed loops. Exit conditions should compare against a bound integer, and use a simple induction increment.
- Minimize loop carried dependencies. This includes pointer arithmetic, complex array indicies, non-linear indexing, multiple index variables in the same subscript location and reading data written by a previous iteration.
- Convert nested loops into a single loop if possible. This can be done manually or by using *#pragma loop_coalesce* option.
- Declare variables in the deepest scope possible. This will reduce serial regions and reduce the amount of extra resources needed to carry information into the next scope.
- Move unnecessary operations out of loop bodies to prevent extra resources from being generated.

2.3 Autotunable Parameters

2.3.1 Choosing Tunable Parameters

The purpose of autotuning some of the user specifiable optimization options is to reduce manual intervention and ultimately the requirement for designer expertise throughout the

optimization process and RTL translation. For the current study, the key factor in determining whether a parameter can be tuned automatically is whether the parameter could potentially impact functionality of the component. Some parameters can only be used if the designer has foresight of how the component is going to be used; others may have an impact on synthesis but won't effect the functional result of the component. This research focuses on parameters of the latter type.

Table 2.2 summarizes which parameters are detected by our system and added into the autotuning search space, are optionally included if requested, or are not included because they require the designer to manually insert for the reasons specified [1]:

Parameter	**Included**	**Reason**
Intel HLS Interfaces	No	Knowledge of external connections are necessary for proper connection to Intel HLS Compiler Interfaces.
AC_DataTypes	No	Knowledge of the usage of the data to determine proper sizing is required by the designer.
Single/Double Pumping	Yes	Offers a trade-off between using additional resources to improve performance.
Replication	No	Replication is always done automatically by the compiler and cannot be manually controlled.
Banking	No	Currently unimplemented, but could be explored further in future work.
Splitting	No	The compiler automatically does this provided it can prove that a pointer refers to exactly one array.
Registers/Memory Allocation	Yes	Forcing memory into registers or memory will provide a trade-off between performance and resource usage.

Loop Unrolling	Yes	Controlling how much to unroll a loop offers a trade-off between performance and resource usage.
Ignore Potential Dependencies	Optional	The designer is responsible for functionality as dependencies are ignored with this option.
Loop Coalesce	Yes	Combining loops can help reduce overhead and resources. Adjusting the number of loops to coalesce can potentially improve resource usage if the loop exit conditions don't get too complicated.
Iteration Interval	No	Forcing a specific iteration interval is only needed to make a component match timing of another component or system, which requires system-level knowledge from the designer.
Floating Point Relaxed/Conversion	Optional	The designer needs to decide if the design allows for small errors in floating point calculations, timings, or reduced precision.

Table 2.2: Optimization Parameters for Autotuning System

2.3.2 Defining Parameter Ranges

The search space for the autotuner grows exponentially with each configuration option considered. The combination of the large search space and the long compile times of the HLS Compiler and Quartus mapping and fitting tool can make it infeasible to exhaustively search through the entire space for the optimal solution. Reducing the size of the search space is one way to improve the end result of the autotuning process.

One example of how the search space can be selected appropriately for a given piece of C code centers around loop unrolling. Recall that loop unrolling is the replication of hardware for an iteration of a loop which allows the loop iterations to be ran in parallel.

When choosing to unroll a loop, the designer can specify how much to unroll the loop versus allow the compiler to pipeline stages of the loop. The more the loop is unrolled, the fewer clock cycles it takes to get a result on the output; however, unrolling generally increases the required hardware resources. This trade-off is one that the autotuner can use to help discover different solutions that may be more optimal for a design's requirements.

The maximum amount that a loop can be unrolled is limited by the number of iterations in the loop, while the minimum is always going to be (one), which implies that the loop is to be entirely pipelined without any loop unrolling. In both of these cases, there are some inefficiencies that can be eliminated by limiting the options considered by the autotuner. For example, choosing to unroll by a factor of the number of the loop iterations will yield a solution in which the hardware is always fully used. Unrolling by a non-factor value will cause portions of the hardware to be stalled when other portions of the circuit are still processing data.

The elimination of non-factor loop unroll values significantly reduces our total search space. There are situations in which specifying the optimal loop unroll amount will yield significantly improved resource usage. This is due to allowing the compiler to use built-in dedicated hardware blocks for a specific task. For example, observe the following code that implements this shift register portion of an FIR Filter:

```
index = 0;
#pragma unroll N
for(index = (129); index > 0; index--)
  data[index] = data[index-1];
data[0] = data_in;
```

Most Intel FPGAs have built-in shift registers that can perform the above operation very efficiently. The above code is a candidate for the autotuner to loop unroll; however, the optimal solution is likely the one that uses the built in shift registers.

	ALUTs	FFs	RAM Blocks	DSP Blocks
(No pragma)	656	580	1	0
unroll (default)	182	183	1	0
unroll 1	656	580	1	0
unroll 2	1098	1140	2	0
unroll 3	2076	2777	14	0
unroll 4	1874	3286	20	0
unroll 5	2164	3700	23	0
unroll 6	2456	4115	26	0
unroll 42	15064	26223	190	0
unroll 43	15302	26536	193	0
unroll 44	15704	27053	196	0
unroll 128	58036	86485	640	0
unroll 129	182	183	1	0

Table 2.3: Loop Unroll Ranges for a Shift Register Circuit

Table 2.3 shows that fully unrolling the design leads to significant resource savings compared to other unroll amounts. This is because of better utilization of specialized hardware in the design (dedicated shift registers). When observing the impact of forcing a specific unroll amount that is not fully unrolling the loop, the results show an increase in resources used at a fairly consistent rate. This increase could be attributed to more complicated access patterns and restrictions that prevent the use of the dedicated shift registers. Also, unrolling the circuit does not have the same consistent relationship for circuit delay, and all unroll amounts between the half-way point and the full unroll amount will all yield the same circuit delay (which is not optimal). This is further discussed in Section 4.6.1. The table also shows that not specifying the unroll directive will yield the same result as not unrolling the circuit (unroll value of one); a circuit that is small in design but does not leverage the built-in hardware shift registers efficiently and is the worst in circuit delay/throughput. Specifying that the loop should be unrolled but not specifying how much will yield the same result as fully

unrolling the circuit (unroll 129 in this case). This yields the circuit that has the highest throughput and the fewest resources and is likely viewed as the optimal circuit choice in this case. Other factors which may influence a designer's loop unrolling decisions that are not considered here are the initiation interval (latency of a circuit) and the initial setup time of the circuit, discussed in more detail in Section 4.6.1.

	ALUTs	FFs	RAM Blocks	DSP Blocks
(No pragma)	942	1003	2	2
unroll (default)	2895	5020	207	2
unroll 1	942	1003	2	2
unroll 2	1391	1572	3	2
unroll 3	2369	2309	15	2
unroll 4	2167	3718	21	2
unroll 5	2457	4132	24	2
unroll 6	2749	4547	27	2
unroll 42	15357	26655	191	2
unroll 43	15595	26968	194	2
unroll 44	15997	27485	197	2
unroll 128	58329	86917	641	2
unroll 129	2895	5020	207	2

Table 2.4: Loop Unroll Ranges for a Shift Register Circuit Implemented in an FIR Filter Circuit

Another interesting factor that should be considered is how the modifications to a circuit impacts the usability of the circuit; a circuit may be optimal by itself, but when used in a larger system it may have more complicated access patterns and a negative impact on the rest of the design. This is demonstrated in Table 2.3 and 2.4. The original shift register used the fewest resources when fully unrolled, but when applied in a larger system, more complicated access patterns result in a significant amount of extra resources to build the circuit in comparison to not unrolling the circuit at all (although throughput is not

considered here). This demonstrates that there is a need for observing different optimization strategies on the program file as a whole instead of individually testing portions of the circuit because of the impact that one optimization may have on the entire circuit.

The following table summarizes the options that are considered by the autotuner for each default included configuration parameter:

Parameter	**Values/Options for Autotuner**
Register/Memory Allocation	hls_register, hls_singlepump, hls_doublepump
Loop Unrolling	Factors of N where N is number of iterations in the loop
Loop Coalesce	1 to N where N is the number of layers to a nested loop
Loop Concurrency	Factors of N where N is number of iterations in the loop

Table 2.5: Default Configuration Parameters and their Parameter Ranges

2.3.3 Injecting Parameters Into Source Code

One of the objectives of this research is to make the C to RTL translation easier and quicker for the designer. In an attempt to assist the designer with Intel HLS Compiler specific optimization strategies that the autotuner can tune, placeholders for the optimization parameters are injected automatically into the input source files. The autotuner can then substitute a corresponding optimization directive for the current configuration. These placeholders need to be put on specific lines in the code for the corresponding optimization technique. For example, the loop_unroll pragma needs to be on the line just above the for-loop, and the hls_memory pragma needs to be at an array declaration.

Parsing C code for a C-to-C translation has many challenges; different coding styles and preprocessor capabilities can lead to a substantial amount of differences from one functional piece of code to another. To assist with interpreting these differences, a parser called PyCParser is used break input source files into AST (Abstract Syntax Tree) nodes. AST nodes provide a grammar independent representation of the source code which can then be parsed through to identify key lines of interest.

Placeholders are injected directly into the source code for each of the optimization options specified in the configuration file for the autotuner. Some of these optimization options are enabled by default, but can be disabled; while others are disabled by default and can be enabled if the designer chooses to do so. This is done to prevent the autotuner from adjusting configuration parameters that could impact the functionality of the code.

Table 2.6 shows an example of the placeholder injection process. Note that the parser has identified one loop, one array, and one variable in the original C code and injected corresponding placeholders and pragmas in the code for use by the autotuner.

<table>
<tr><th>Original Code</th><th>Generated Code</th></tr>
<tr><td><pre>

signed long mul_out[15];
signed long out;
for(i = 15; i >= 0; i--)
{
   out += mul_out[i];
}</pre></td><td><pre>
num_pump_1
    signed long mul_out[15];
variable_0
    signed long out;

#pragma unroll loop_unroll_1
#pragma max_concurrency loop_concurrency_1
#pragma loop_coalesce loop_coalesce_1
    for(i = 15; i >= 0; i--)
    {
       out += mul_out[i];
    }</pre></td></tr>
</table>

Table 2.6: Original Code Versus Generated Code

2.3.4 Limitations

There are several limitations of the Intel HLS Compiler which prevent certain code styles, C constructs, and specific optimization techniques from being used. Some of these situations are accounted for in the automated insertion script, but some situations are difficult to account for such as user added levels of abstraction through typedefs, unions, etc. The Intel

HLS Compiler is more robust when manual optimization is *not* specified, and can synthesize an acceptable design that would otherwise throw a compiler error when compiler directive is specified for a code segment. In other situations, there could be sensitive sections of code in which the designer does not want optimization strategies applied.

To account for some of these challenges, our parsing solution allows the designer to force a line of code to not be optimized by the autotuner. This will bypass the placeholder injections, and allow for manual optimization parameters to be specified. For example, the Intel HLS Compiler does not allow for pointer math to be used in a loop which utilizes constant data types from a global space.

<table>
<tr><th>Broken Code</th><th>Fixed Code</th></tr>
<tr><td>
<pre>
/*Global Variable Space*/
const int h[24] = {
  12, -44, -44, 212
};

/*Inside a function:*/
const int *h_ptr;
h_ptr = h;
#pragma unroll 5
  for (i = 0; i < 10; i++)
    {
      xa += 2 * (*h_ptr++);
      xb += 2 * (*h_ptr++);
    }
</pre>
</td><td>
<pre>
/*Global Variable Space*/
const int h[24] = {
  12, -44, -44, 212
};

/*Inside a function:*/
const int *h_ptr;
h_ptr = h;
//blacklist
  for (i = 0; i < 10; i++)
    {
      xa += 2 * (*h_ptr++);
      xb += 2 * (*h_ptr++);
    }
</pre>
</td></tr>
</table>

Table 2.7: Using the Blacklist Option

Table 2.7 demonstrates how using an optimization strategy generates a compiler error. On the left, specifying an unroll amount for a for-loop which utilizes pointer math on a

constant variable defined in global space generates an error in the Intel HLS Compiler. Should the designer remove the pragma unroll statement, the Intel HLS Compiler is able to synthesize the design properly. Since the autotuner will automatically attempt to put a placeholder above the for-loop (which includes unroll statements), the designer must specify that the for loop must not be optimized; otherwise, an error will be generated. This is done by using the *//blacklist* command just above the problematic loop.

3. OpenTuner Configuration and Execution

This chapter describes the overall architecture of the OpenTuner Framework. It then explains the modifications that were made to OpenTuner in order to allow autotuning of the Intel HLS Compiler.

3.1 OpenTuner Framework

Figure 3.1 provides an overview of the OpenTuner framwork, which attempts to solve the autotuning problem. The autotuning problem is that of a search problem [6]. The search space is comprised of a combination of inputs and parameters that collectively make a *configuration*. In the present application, parameters are used to tweak the C to RTL translation to work towards optimizing an objective function representing the *quality* of a compile to find a better solution than what the default translation would normally achieve. In order to establish a baseline for comparison, the first translation is always the default translation (with no specified optimization parameters). This baseline is used to assist with determining the effect of adjusting the configuration. Each adjustment of the configuration is done by a *configuration manipulator*, which makes adjustments to its current configuration based on previous results and applying a search space technique to the collection of results in a common database to generate the next configuration. Over time, it is hoped that the configuration manipulator will hone in on desirable values of the configuration parameters.

Each result is analyzed by a *measurement* process which uses a user defined measurement function to determine the worth of the result. This is necessary to analyze the area versus performance trade-off, and also to take into consideration how close to a design target the current result is. This calculated result becomes the basis of the user defined *objectives* which

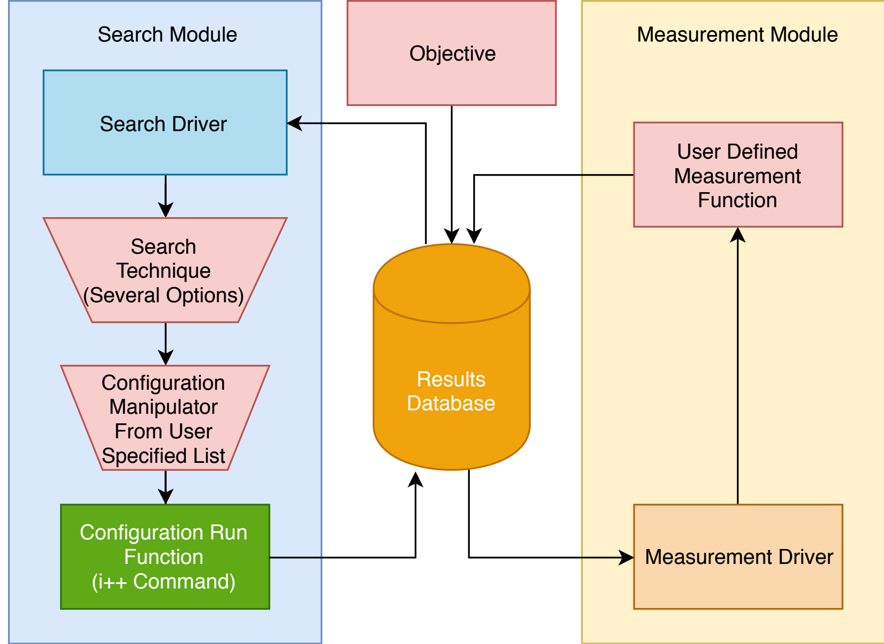


Figure 3.1: Overview of the Major Components in the OpenTuner Framework [6]

the OpenTuner Framework is attempting to meet.

3.1.1 Search Techniques

The first configuration that OpenTuner calculates establishes the basis for comparison; this configuration includes no forced optimizations and allows the Intel HLS Compiler to optimize using all default parameters and optimization strategies. All subsequent compilations use the new source file generated with injected placeholders for forced optimization parameters. The second configuration will be from a random starting point based upon a set of seed values. The seed values can be specified by the user, or can be left to the random number generators. As results are stored into the database, user specified search techniques will start utilizing the data from the database to determine future configurations to try.

OpenTuner includes a wealth of search space techniques which are implementations of standard, well-studied mathematical optimization algorithms. Users can choose to define their own search technique if preferred. Some of the included search space techniques are listed in Table 3.1.

Technique	Description
Differential Evolution	New candidate solutions are created based on combining existing solutions.
Greedy Mutation and Variants	Modifying one or more values based on a probability of change.
Nelder-Mead and Variants	Treats the N-dimensional problem with n+1 points, looking for valleys using the centroid.
Pattern Search	Analyzing previous results for defined patterns and adjusting inputs accordingly
Hill Climbing Variants	Making incremental changes to an existing solution.
Particle Swarm Optimization	A population of candidate solutions that move around the search space.
Pure Random	All parameters are chosen randomly.

Table 3.1: OpenTuner Built-In Search Techniques

OpenTuner also provides a mechanism to utilize multiple search space techniques at the same time. The user is able to specify a collection of techniques to explore, and will subsequently bias towards techniques that perform well while techniques which perform poorly are allocated fewer tests [6]. Using ensembles of techniques provide OpenTuner with two distinct advantages. First, some search techniques may naturally perform better for a given search space than another technique. By employing multiple techniques at the same time, it is much more likely to arrive at a better solution because of a higher likelihood of using a technique that is conducive to the search space. Second, most techniques utilize all existing results from the database. This allows one technique to use data points which another technique has unveiled and expands the data to include more information that would normally not be present for a single search technique. This helps prevent a technique from stalling or getting stuck in a local optimum due to a bad starting point [6].

A detailed discussion and analysis of optimization techniques for multidimensional search

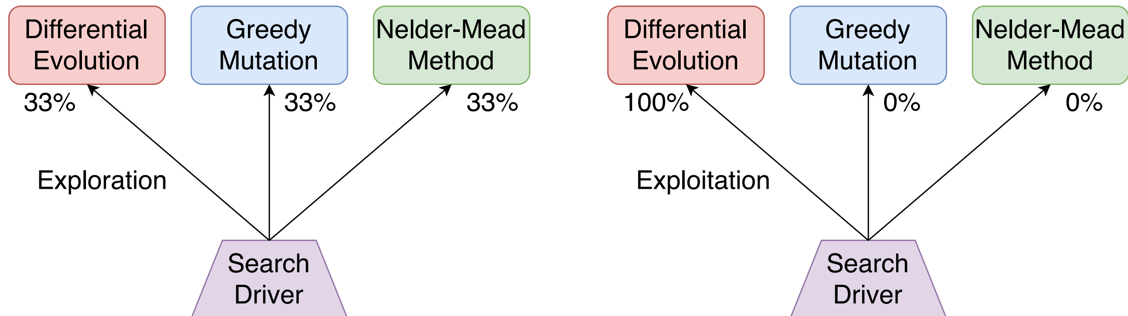


Figure 3.2: OpenTuner's Exploration Phase and Exploitation Phase [7]

spaces is outside the scope of this thesis. In this work, the AUC Bandit Meta Technique is used which is a collection of four different techniques: differential evolution, uniform greedy mutation, normal greedy mutation, and random Nelder-Mead. This is the core meta technique which OpenTuner uses as it is based on an optimal solution to the classical multi-armed bandit problem [6]. This meta-technique encapsulates a fundamental trade-off between exploitation (using the best known technique) and exploration (estimating the performance of all techniques) [7]. It does so by using a sliding window to analyze which techniques have been used the most in the window, and also assigns credit to techniques which performed well for the given search space.

3.1.2 Configuration Manipulator

The configuration manipulator serves two main purposes in OpenTuner. First, it provides a means for the user to specify a list of parameters over which OpenTuner should search for optimal solutions. Each parameter can be assigned a type and a range of valid values. For each built in type of parameter that OpenTuner provides, a collection of helper functions are also included which assist OpenTuner in evaluating the search techniques. OpenTuner's parameters are stored as a hierarchy with two main categories. Primitive parameters are those that have a numeric value with both an upper and lower bound. Complex parameters have a variable set of manipulation operators, and allow parameters to be customized for domain-specific structures to be included in the search space. The selection of data types for autotuneable parameters can impact the performance of the autotuner, and is an important design decision. Specifically, complex parameters are more difficult for search techniques to

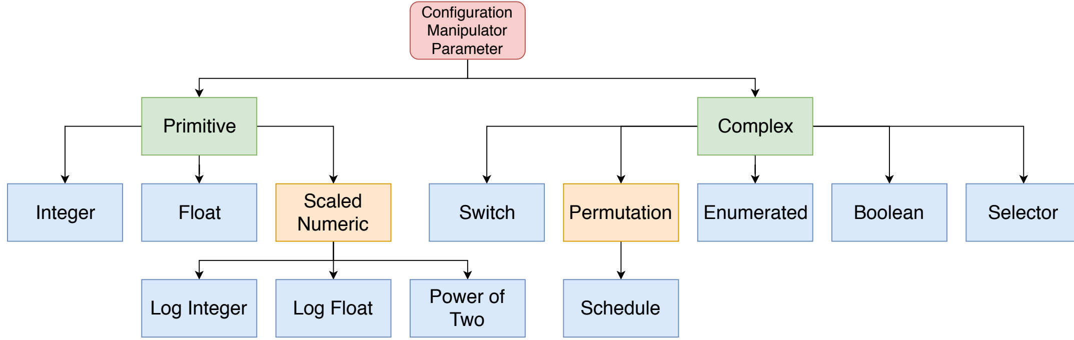


Figure 3.3: OpenTuner's Hierarchy of Included Parameters [6].

explore because of the lack of bounds and lack of defined relationship between operators.

Some complex parameters can also be included in the primitive space. For example, boolean, switch, and enum types can easily be translated to an integer representation. However, in the context of the search techniques they are better represented as complex parameters because integer based parameters usually follow gradients, whereas complex types cannot be assumed to follow gradients. In the context of autotuning Intel HLS parameters, most parameters have been reduced in their possible ranges to decrease the search space size. It is a limitation of the OpenTuner Framework that the integer primitive types do not include lists of integers.

To accommodate the limitation of integer primitive types in the OpenTuner Framework, lists of possible inputs (such as the factors of the number of iterations of a loop) are passed in as enumerated types. It should be noted that in the current implementation of OpenTuner, the best implementation for some of the parameters would be an extension of a scaled numeric type where the factors are listed in an order which implies a corresponding gradient. However, for this initial proof-of-concept system, parameters which are subset lists of a full range are currently defined using the enumerated type. In particular, the parameters for loop unrolling and loop concurrency would be best specified with a numeric type instead of an enumerated type. Similar to the powers of two type, a type for factors could be created. This potential for future work is discussed further in Section 5.3.

3.1.3 Objectives

By default, OpenTuner supports multiple user defined objectives. The framework defines multiple static fields to address these objectives which include time, accuracy, energy, size, confidence and user defined data. Each field can be used as a minimizing function, maximizing function, threshold based function, or a combination of objectives and functions. For example, minimizing time could be the defined objective of the autotuner, and the user can pass the execution time of a program as the input to this field. The autotuner would then use the specified search techniques and configurable parameters to attempt to find solutions which minimize the execution time of a program.

As discussed in the following sections, customized objective functions have been defined in order to facilitate the design space exploration of HLS translation.

3.2 OpenTuner Usage for Tuning Intel HLS

3.2.1 Search Techniques

The AUC Bandit Meta Technique is intended to provide a robust technique for problems with unknown probabilistic outputs. This technique also provides a good balance between exploitation and exploration of search techniques. Each type of problem (and every time the original design changes) may have a different technique that will more efficiently hone in on an optimal solution. For simplicity and consistency across different designs and for the other reasons listed above, the AUC Bandit Meta Technique is used as the search technique for all result generation in this research.

3.2.2 Search Space Size

The search space of a given configuration is determined by the number of included optimization parameters. For example, each variable that is declared in the source program can be optimized through memory optimizations by specifying *hls_register, hls_singlepump, or hls_doublepump.* If these three options are included in the optimizer, every C variable which is to be placed into memory increases the size of the search space by a factor of 3.

For-loops can be optimized by specifying an unroll amount and a concurrency amount. The default included parameters for each for-loop is based upon the number of loop iterations. Given N iterations of a loop, valid configuration parameters included any factors of N for both the loop unroll amount and the concurrency amount. Thus, the search space increases by *(Factors of N)*^{2}. The impact on the search space is outlined in Table 3.2 for the default included parameters.

	Increase in Search Space Size
Variable Declaration	3
Loop of N iterations	(Factors of N)^2
Nested Loop of N Levels	N

Table 3.2: Impact on the Search Space

3.2.3 Results Generation

A closer look into what results are produced from the Intel HLS Compiler is needed before determining the objective of the autotuning process. A report is generated by the Intel HLS Compiler that outlines estimated resource usage based on the current implementation each time the tool is used to translate a C source file to RTL. The key components that are extracted from this report include the number of logic modules used as adaptive lookup tables (ALUTs) or adaptive logic modules (ALMs), the amount of FPGA memory used (RAM bits or RAM blocks), the number of FFs used, the number of DSP blocks used, and the maximum operating clock frequency for the circuit (F_{max}).

A weighted normalized sum is used as the basis of comparison for configuration results because it provides a quantifiable representation of the trade-off between performance and resources used based upon the designer's preferences [26]. Each individual component is assigned a weight which provides a relative worth to itself verses the other components. All values are then normalized to the initial HLS compile to provide the designer with a quantifiable representation of how much improvement the autotuner's solutions have provided in comparison to the compilation with default optimization. Resource based metrics will

be found to be improved if they are smaller than the initial estimate/translation and the performance based metric (F_{max}) is improved if it is greater than the initial synthesis; thus, two equations are used to represent the weighted normalized values.

The relative improvement for each result component relating to resource usage can be found by:

$$WNV_n = \frac{W_n \times (X + 1)}{X_i + 1} \tag{3.1}$$

WNV_n : Weighted Normalized Value of an individual component (FF, ALUT/ALM, RAM bit/block, DSP Block)

W_n : Weight of an individual component, specified by the designer

X : Configuration result for an individual component

X_i : Initial configuration result for an individual component

The relative improvement for the performance metric (F_{max}) can be found by:

$$WNV_n = \frac{W_n \times (X_i + 1)}{X + 1} \tag{3.2}$$

The relative improvement of a configuration can be found by summing the individual weighted normalized values.

$$WNS = \sum_n WNV_n \tag{3.3}$$

WNS : Weighted Normalized Sum of a configuration

In this research's implementation of autotuning the Intel HLS Compiler, minimizing the WNS value is used as the objective of the autotuner. After each configuration completes, the WNS value for that configuration is computed and stored in the database, and the autotuner subsequently attempts to find configurations that minimize the WNS value.

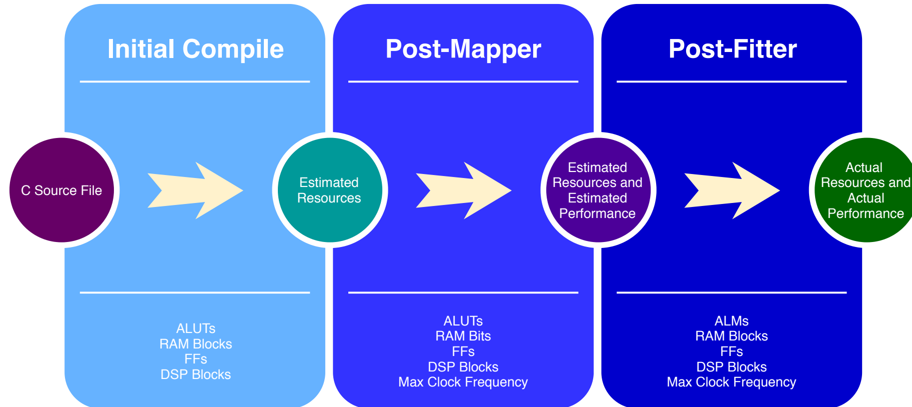


Figure 3.4: Intel HLS Compilation Stages

3.2.4 Compilation Types

The process that the Intel HLS Compiler follows is a multi-stage translation process that leverages an HLS tool and Intel's Quartus Prime software to translate a design to RTL, map the design to a technology and finally fit the design to a specific FPGA. The first stage is an initial compile done by the translation tool to give the designer feedback on their current implementation. This stage includes an estimate of how many resources are going to be used (including ALUTs, FFs, RAM blocks and DSP blocks). At this stage there has been no timing analysis performed yet; thus no performance metrics are generated. The C source file is translated to a functional RTL equivalent based on the architecture specified by the designer. This RTL equivalent is not finalized at this stage because it has not been mapped to a selected technology or fitted to a particular FPGA; hence, it is considered an estimate. This first stage is (relatively) quick to perform.

Included with this initial compile is an optional functional simulation performed by Modelsim and an Intel HLS Compiler specific report which provides detailed information about the current design. The Modelsim-based verification process is used to ensure that the design is in fact functionally correct, and can be verified through an accompanying C/C++ testbench in the original source file. The generated report is intended to be viewed by the designer so that an iterative process can be used to improve the overall quality of the component. Any portions of code that cannot be translated properly are identified.

The autotuner is intended to be used after the functional verification process is complete. Once the designer has ensured that the component is functionally correct and that all of the code can be translated to RTL, they can then run the autotuner to improve the C to RTL translation. If the designer is only interested in optimizing the resource usage of an RTL translation, they could autotune their design based on this first stage. Doing so will yield results much more quickly than performing the latter stages of the compilation process, but will not yield a fully implementable solution on a physical FPGA.

The second stage of the process is to run the Quartus mapper. The mapper translates a hardware descriptive language (HDL) to a technology specific RTL equivalent. Since the RTL has already been generated, the input to the mapper is the pre-generated RTL design which is then adjusted to a technology. This adjusts the resource usage outputs to those that are specific to a technology. For example, the Cyclone V family of FPGAs utilize 8-input adaptive logic modules (ALMs) as seen in Figure 3.5 which include an 8-input fractured look-up table (LUT) with four dedicated registers [8].

For OpenTuner to leverage the Quartus mapper, a Tcl script that utilizes the Quartus shell from command line has been created. The Intel HLS Compiler provides an option to perform a full Quartus compile (post-fit analysis) with its standard command; however, it is advantageous for the designer to only perform the Quartus mapper. The mapper is more accurate than the initial estimation and is much faster to perform than the fitter. Using a Tcl script allows the automation of testing a design with the Intel HLS Compiler and the Quartus shell tool. Additional shell tools can be ran following the mapper such as database merging and timing analysis of the post-mapped results. This generates performance metrics that are imported into OpenTuner. Resource usages are also imported into OpenTuner (ALUTs, RAM bits, DSP blocks, and FFs).

The final stage in generating a fully implementable RTL solution is the Quartus fitter. The fitter generates accurate quality-of-results (QoR) metrics that were mentioned in the mapper. The fitter requires a previously ran analysis and synthesis (post-mapper) solution prior to being ran. The fitter places and routes the design for a specific FPGA device. Following the fitter is another timing analysis to update the maximum performance metrics

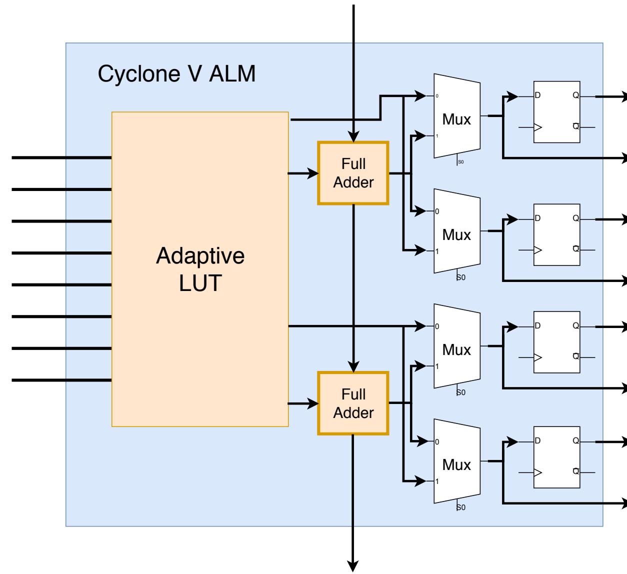


Figure 3.5: Cyclone V Adaptive Logic Module (ALM) [8]

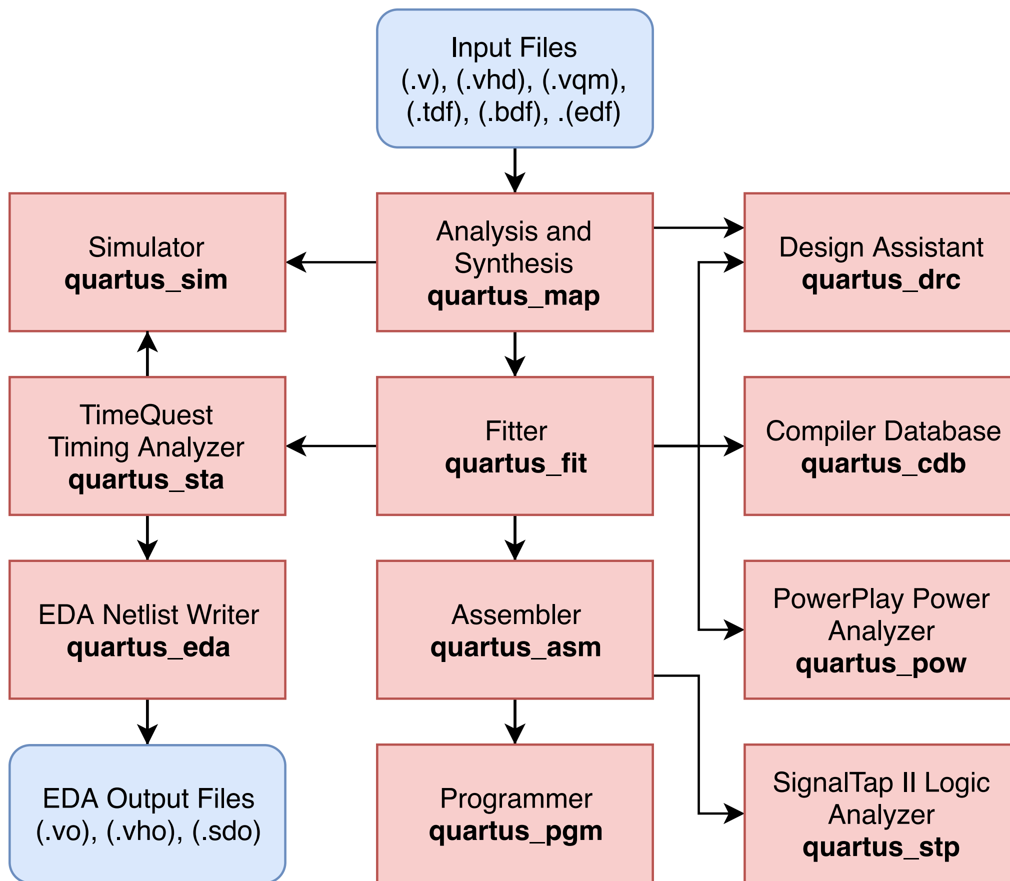


Figure 3.6: Design Flow When Performing a Full Quartus Compilation From Command Line or Tcl Script [9]

to include wiring differences, critical path adjustments, and further optimization that is performed by Quartus. The ALUTs are now converted to ALMs and the RAM bits are converted to RAM blocks, but they still can be analyzed using the weighted normalized sum computations found in equations 3.1 and 3.3. Although using the post-fitter solution as the result metric for OpenTuner provides the highest accuracy of results, it also takes the longest to perform and requires the most computer resources to compile. Therefore, using post-fit results may limit the extent to which OpenTuner can explore the search space for a particular design in a *reasonable* amount of time.

3.2.5 Adjusting Optimization Preferences

One intent of this research is to assist with the C to RTL translation for practical design circuits; adding the capability for the designer to adjust the objective, search space, and relative worth of each result provides a practical approach to the autotuning solution.

Adjusting the Search Space

The default parameters for the search space, as outlined in Section 2.3.1, include several autotunable parameters that are guaranteed not to break the functionality of the design. This includes loop unrolling, loop concurrency, allocating variables to memory or registers, single pumping or double pumping the memory, and coalescing nested loops. There are also some optional parameters that the user can choose to enable which will increase the search space. These include ignoring potential memory dependencies that would otherwise prevent the compiler from performing optimizations, specifying the number of potential concurrent implementations of a component, and relaxing floating point arithmetic. In the current implementation, options to adjust or optimize component interfacing and specifying memory optimizations (such as sizing, banking, interfaces, etc) are not implemented in the autotuner. The user can, however, manually insert these optimizations and adjustments and the autotuner will respect them. Similarly, the user can disable any of the default optimization strategies, or manually specify an option in the original source file which will take precedence over the autotuner's injected parameters.

Adjusting Result Weights

An important feature to make the OpenTuner implementation practical is the ability to adjust the weights of the items generated from each configuration result. In a standard design process using Quartus Prime, the designer has the ability to specify an optimization strategy for the design. These include balanced, performance, and area focused optimization. The designer usually has insight into the limiting factors in their design; if they suspect space will be an issue, they could attempt to optimize with the area strategy. The autotuner constructed in this research continues with this scheme by giving the designer the ability to adjust weights attached to each of the generated results or choose a different weight scheme. Table 3.3 shows the default weights and schemes chosen for this research.

Metric	Area	Balanced	Performance
F_{max}	2	4	8
ALUTs / ALMs	8	4	2
FFs	8	2	4
RAM bits / RAM blocks	8	2	2
DSP blocks	8	2	2

Table 3.3: Optimization Strategies and Associated Weights

The value chosen for the weights should be proportionately represented in relation to the other weights for the items of interest; the actual value of the weight is not as important as the relation to the others. As each result is generated, the *relative improvement* amount will scale in accordance to the weights chosen and the number of each resource that is synthesized in the result. See equations 3.1 and 3.3 for more information on how the weights are used. The framework also allows for the designer to specify multiple optimization schemes such as those found in Table 3.3. This allows for quicker exploration of the different possible solutions by quickly adjusting the optimization scheme.

3.2.6 Specifying Targets

Introducing targets (or goals) for the autotuner adds a new function to the compiler and translator that wasn't previously available. The previous optimizations performed by the Intel HLS Compiler acted on a standard view of *generating a design that is optimal in a category*. With the introduction of targets to the tool, the designer can *generate a design that is optimal in a category while focusing on meeting target specifications*. This is a more practical use of the compiler and translator as more specific objectives can be specified for a solution, and multiple objectives can be accounted for.

The adjustments made to the autotuner to incorporate specific targets are done through the WNS objective as before. With each specified target/goal, a penalty is imposed on the weighted normalized value for that component if the target has not been met. The penalty that is applied becomes increasingly heavy the further from the target that the current configuration generates. Starting with Equation 3.1 as the basis, targets and their penalty can be introduced with:

$$WNS = \sum_{n} WNV_n \times P_n \tag{3.4}$$

WNS : Weighted Normalized Sum of a configuration

WNV_n : Weighted Normalized Value of an individual component

P_n : Penalty for not meeting the target value for an individual component

The penalty is computed as:

$$P_n = \begin{cases} 1 & X_n \leq T_n \\ (\frac{X_n+1}{T_n+1})^{T_p} & X_n > T_n \end{cases} \tag{3.5}$$

P_n : Penalty for an individual component

X_n : Configuration result for an individual component

T_n : Initial configuration result for an individual component

T_p : Penalty factor (user selectable, default = 2.0)

Equation 3.5 may be used for all resource related values that are extracted from the compilation reports, however a slight variation is needed when comparing the performance metric; $\mathrm{F_{max}}$ should be maximized instead of minimized. For the performance metrics, the equation changes to:

$$P_n = \begin{cases} (\frac{T_n+1}{X_n+1})^{T_p} & X_n < T_n \\ 1 & X_n \geq T_n \end{cases} \tag{3.6}$$

If targets are specified then the weighted normalized values found in Equation 3.4 are used instead of those found in Equation 3.1. In the current implementation of the autotuner, targets may be specified as either absolute values or percentages of the available resources in the specific FPGA device.

The penalty factor (T_p) is specified by the user, with a default value of 2.0. To implement more harsh penalties for not meeting the target, increasing this factor slightly provides an exponential penalty imposition. Similarly, if targets are soft goals for the designer, lowering the penalty factor imposes a reduced penalty for not meeting the target.

A configuration file allows the user to select both the weights and the penalty factor for the project. An example of the configuration file is provided in Appendix B.

4. Results

The need for comparing quantitative results from one HLS tool to another is fulfilled through standardized benchmark suites. A common benchmark suite that researchers have been using for HLS translations is the CHStone Benchmark Suite, developed in 2008 [34,35]. The suite consists of 12 programs written in C, with the programs spread across several different programming domains. These include arithmetic, media applications, cryptography, and processor designs. The CHStone Benchmark Suite was designed to stress HLS compilers by using a variety of C coding styles across various programming domains. As such, due to the previously mentioned coding style limitations in the Intel HLS Compiler, many of the C programs do not natively work with the Intel HLS Compiler without adjustment.

The Intel HLS Compiler is designed to have clear separation between the *component* and the *testbench*. The function to be synthesized needs to be marked as the component, and all other C program code is assumed to be testbench related or passed into the component via an interface connection. The CHStone Suite does not make such assumptions, and so a manual reconfiguration of the files was performed to clearly separate the component from the testbench. Similarly, the Intel HLS Compiler has strict C programming *best practices* guidelines that should be followed for the synthesis to work properly. This includes avoiding dynamically allocated memory, not clearly defining loop lengths and avoiding memory dependencies (for a full list of limitations, see Section 2.3.4). Since the CHStone applications do not always follow Intel's C programming best practices, many of the CHStone applications do not compile in the Intel HLS Compiler out of the box.

Some of the CHStone applications can be compiled and synthesized with some *simple* modifications performed while others would require extensive rewrites. In this research, upon

performing the manual separation of the testbench from the component, a test is conducted to see if the component will synthesize:

```
i++ source.c -o test-fpga --simulator none --quartus-compile
```

The above command will yield a full RTL solution for the given source file. Each of the CHStone applications have a single marked function as *component*. For this research, it is assumed that the Intel HLS Compiler will properly translate the C source file into functionally equivalent RTL. Modelsim simulations are generally intended to increase confidence in the final design through function verification. Therefore, in this research, the simulation step is not performed during each autotuning iteration as the design space is explored. Eliminating the simulator reduces compilation times which will reduce the overall time to synthesize greatly.

The following sections describe a series of tests that were performed to evaluate the performance of the Intel HLS Compiler. First, in Section 4.1, the impact of individual configurations parameters is investigated for a typical DSP filtering application found in appendix A. Sections 4.2 to 4.6 then describe the results obtained when autotuning a variety of designs, including those from the CHStone benchmarking suite.

4.1 Impact of Individual Parameters

Using a standard DSP Finite Impulse Response (FIR) Filter (C source code included in appendix A), we can observe the effects of some of the parameters that are being autotuned. To test the individual impacts, manual optimization pragmas are injected to only a single line to observe the impact on the final synthesis. These results will vary for each level of synthesis that is performed: estimated resources, post-mapper, or post-fitter synthesis; the latter two also being effected by Quartus optimizations. The rest of the source file is left to default optimization strategies performed by the Intel HLS Compiler.

There are two purposes for performing these tests. First, they can provide insight into how the individual configuration parameters can influence the HLS results. Second, they can

be used to evaluate the similarities and differences between the resource estimates provided by the Intel HLS Compiler and those obtained from a full Quartus compilation.

4.1.1 Memory Optimizations

Memory optimizations are performed on all variables in the filter design. Prior to running the Quartus mapper or Quartus fitter, the Intel HLS Compiler must first generate an estimate of the logic resources required for the FPGA implementation. Tables 4.1 and 4.2 include the associated estimated resource usage from the Intel HLS Compiler and the Quartus mapper and fitter, respectively, for the memory optimization tests.

	Quartus Mapper Results					Intel HLS Compiler Estimates			
Test	ALUTs	FFs	RAM Bits	DSPs	F_{max} (Mhz)	ALUTs	FFs	RAM Blocks (M10K)	DSPs
hls_singlepump **static int** data[65];	479	850	4164	2	101.1	1082	974	1	2
hls_doublepump **static int** data[65];	452	1139	4171	2	101.1	1146	1308	2	2
hls_register **static int** data[65];	5822	10349	24	2	63.0	6215	9032	0	2
hls_singlepump **static int** index = 0;	479	850	4164	2	101.1	1082	974	1	2
hls_doublepump **static int** index = 0;	479	850	4164	2	101.1	1082	974	1	2
hls_register **static int** index = 0;	479	850	4164	2	101.1	1082	974	1	2

Table 4.1: Memory Optimizations for Local Variables Using Mapped Resources

As can be seen in Table 4.1, optimization of some variables from the original C code can have a significant impact on the final result, while others have no impact at all. This will vary depending on usage of the variables in the component whether they are inputs/outputs of the component and how the memory access patterns are analyzed by the tool. In some cases, variables can be optimized away which is the case for the *index* variable. Arrays are treated in a similar fashion to variables, except the impact is multiplicative based on

the size of the array. Both hls_singlepump and hls_doublepump are directives to store the variable into memory (RAM), but the addition of double pumping the clock creates for more complicated access logic which is demonstrated by an increase in ALUTs and FFs.

	Quartus Fitter Results					Intel HLS Compiler Estimates			
Test	ALMs	FFs	RAM Blocks	DSPs	F_{max} (Mhz)	ALUTs	FFs	RAM Blocks	DSPs
hls_singlepump **static int** data[65];	329	683	1	2	216.12	1082	974	1	2
hls_doublepump **static int** data[65];	385.5	977	2	2	101.1	1146	1308	2	2
hls_register **static int** data[65];	4912	10511	0	2	113.51	6215	9032	0	2
hls_singlepump **static int** index = 0;	329	683	1	2	216.12	1082	974	1	2
hls_doublepump **static int** index = 0;	329	683	1	2	216.12	1082	974	1	2
hls_register **static int** index = 0;	329	683	1	2	216.12	1082	974	1	2

Table 4.2: Memory Optimizations for Local Variables Using Fitted Resources

When observing the results normalized to the hls_singlepump solution for *data* variable as shown in Table 4.3, we can see that introducing a double pumped memory frequency typically yields an increase in area usage, although it allows for portions of the circuit to perform at double the system clock frequency, but actual fitting of the circuit to the device yields a decrease in maximimum operating clock frequency. This decline in performance is the result of additional access logic created from more complicated memory accessing. This shows that doublepumping the memory does not always create performance increases. Similarly, forcing variables into registers can have a significant impact on resource usage and performance metrics.

	Estimated				Mapped					Fitted				
Test	ALUTs	FFs	RAMs	DSPs	ALUTs	FFs	RAMs	DSPs	F_{max}	ALUTs	FFs	RAMs	DSPs	F_{max}
hls_singlepump	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
hls_doublepump	1.06	1.34	2.0	1.0	0.94	1.34	1.002	1.0	1.0	1.17	1.43	2.0	1.0	0.47
hls_register	5.75	9.3	0.0	1.0	12.2	12.2	0.006	1.0	0.62	14.9	15.4	0.0	1.0	0.53

Table 4.3: Results of the Individual Impact of a Memory Optimization Normalized to the hls_singlepump Solution for the *data* Variable

4.1.2 Loop Optimizations

There are three loop optimization techniques that are being autotuned by the OpenTuner Framework: loop concurrency, loop unrolling, and loop coalescing.

Loop Concurrency

Adjusting the maximum loop concurrency offers a trade-off of reduced performance for reduced resource usage. To benefit from increasing the maximum loop concurrency, a loop needs to be able to benefit from private copies of a variable to improve the throughput of the loop. This can occur only when the scope of a component memory (through its declaration or access pattern) is limited to the loop being optimized. In such cases, loop concurrency will provide a trade-off between throughput performance and area usage. The FIR filter under consideration in this section does not benefit from adjusting the loop concurrency; static memory declarations that are used in the scope of the component do not benefit from loop concurrency.

Loop Unrolling

Loop unrolling allows for the replication of hardware to improve throughput of a loop. To observe the effect of loop unrolling, any loop can be unrolled by using the *#pragma unroll* followed by a number which represents how many times to unroll the loop: a value of one implies that the loop does not get unrolled and the maximum value unrolls the loop fully (bound by the number of iterations of the loop).

Shift Register	Accumulator

#pragma unroll N
for(index = (64); index > 0; index−−)
{
  data[index] = data[index−1];
}
	#pragma unroll N
	for(index = (64); index >= 0; index−−)
	{
	  data_out += data[index] ∗ c[index];
	}

Table 4.4: Loop Unrolling Examples

Using post-fitted synthesis results, we can see the effect on both examples over a range of unrolling values for N found in Table 4.5. Note that the shift register has a maximum loop iteration of 64 and the accumulator has a maximum loop iteration of 65.

Test	ALMs	FFs	DSPs	RAM Blocks	F_{max}
Shift Register N = 1 Accumulator = default	329	683	2	1	216.12
Shift Register N = 32 Accumulator = default	2975	9972	2	64	170.53
Shift Register N = 64 Accumulator = default	2830	5021	2	116	169.69
Accumulator N = 1 Shift Register = default	329	683	2	1	216.12
Accumulator N = 32 Shift Register = default	3250	9698	64	32	139.96
Accumulator N = 65 Shift Register = default	6038	21929	58	65	163.45

Table 4.5: Results of the Individual Impact of Loop Unrolling

The results in Table 4.5 demonstrate that default optimization strategy for the FIR filter

is to not unroll the loops (default N = 1). This implies that loop unrolling must be specified by the designer to take advantage of the increase in throughput performance. Another interesting observation is the reduction of resources that the shift register realizes when fully unrolling. This shows that the compiler is able to better utilize the built-in hardware for a shift register when fully unrolling. The accumulator shows a drop in performance when not unrolling by a factor of the number of iterations of the loop. It is for this reason that the autotuner only selects factors of the loop iteration when choosing loop unrolling (and loop concurrency) values in the search space.

Loop Coalescing

Loop coalescing is the merging of nested loops into a single looping structure. This can simplify designs, but can also complicate memory access logic. The number specified for loop coalescing determines how many nested layers of loops it should attempt to coalesce. The default FIR filter does not include any nested loops; however, a slight adjustment can be made (for demonstration purposes):

```
#pragma loop_coalesce N
for(index = 64; index > 0; index--)
{
  data[index] = data[index-1];
  for(index = 64; index >= 0; index--)
  {
    data_out += data[index] * c[index];
  }
}
```

	ALMs	FFs	DSPs	RAM Blocks	F_{max}
N = 1	515	1338	4	2	202.27
N = 2	349	951	2	2	201.57

Table 4.6: Loop Coalescing Examples

The example shown in Table 4.6 yields improved resource usage by coalescing the two loops into a single solution; however, it should be noted that coalescing loops does not always

yield improved resources as memory access patterns can become increasingly complicated in some designs.

4.2 Autotuning the Intel HLS Compiler

The CHStone benchmark suite has been developed for C-based HLS, and has been a common benchmark for HLS related research. The suite has become a popular choice because it requires no additional libraries or extensions, and covers a variety of different domains and programming styles. The suite consists of twelve programs which are selected from domains such as arithmetic, media processing, security and microprocessor [34]. Table 4.7 summarizes the twelve CHStone programs considered in this research.

Application	Domain	Description
DFADD	Arithmetic	Double-Precision Floating-Point Addition
DFMUL	Arithmetic	Double-Precision Floating-Point Multiplication
DFDIV	Arithmetic	Double-Precision Floating-Point Division
DFSIN	Arithmetic	Sine Function for Double-Precision Floating-Point Numbers
MIPS	Processor	Simplified MIPS Processor
ADPCM	Encryption	Adaptive Differential Pulse Code Modulation Decoder and Encoder
GSM	Communcations	Linear Predictive Coding Analysis of Global System for Mobile Communication
JPEG	Media	JPEG Image Decompression
MOTION	Media	Motion Vector Decoding of the MPEG-2
AES	Encryption	Advanced Encryption Standard
BLOWFISH	Encryption	Data Encryption Standard
SHA	Encryption	Secure Hash Algorithm

Table 4.7: List of CHStone Applications and Their Functions

4.2.1 Use of CHStone

The Intel HLS Compiler has many limitations in coding style and application. The tool is designed to synthesize a component while also providing the means to incorporate a testbench (written in C/C++) that can communicate with Modelsim - Intel's simulation tool. The CHStone applications are designed and developed to provide a standard benchmarking suite written in C (not C++), and does not consider the specific requirements of the Intel HLS Compiler. As such, limitations within the Intel HLS Compiler were identified in many of the CHStone applications.

All of the CHStone applications needed minor adjustments to work with the Intel HLS Compiler. Namely, the component must be identified and marked, and a clear separation between the testbench and component is needed for the tool to synthesize properly. In doing so, the testbench portion of the program gets ignored by the Intel HLS Compiler and does not get synthesized into RTL. In some cases, program-specific modifications were made to circumvent compiler errors, but modifications to the basis of the CHStone applications is avoided to maintain comparable results.

Adpcm (mostly) worked after blacklisting, or not optimizing, two sections of code that used pointer arithmetic in memory accesses, as mentioned in Table 2.7. Since adpcm uses large constructs and structures in its design, compilation times varied drastically: from 6 minutes to more than 3 hours per compile. The variance in compilation times is attributed to large loop unrolls that create large amounts of hardware, which exponentially increases the complexity of the RTL solution (and inherently takes a long time to compile).

Applications That Would Not Compile

Three applications would not compile with the Intel HLS Compiler. Major changes and reorganizations of their underlying C code would be required for successful compilation, which is contrary to the purpose of a benchmark program. The reasons that they would not compile varied and are discussed below.

JPEG Fails to compile because of an illegal use of global variable in the component. Pointer

to pointer types are not supported by the Intel HLS Compiler in synthesized code.

SHA Fails to compile because the application uses complex address math which is not supported for load from filescope constant data. This includes pointer math for the accesses to data. More specifically:

```
tmp |= 0xFF & *p2++;
```

is not allowed by the Intel HLS Compiler.

MOTION This program uses older C style syntax such as K&R C style function definitions. The Intel HLS Compiler provided with Quartus treats all input files as C++14 [5]. The compiler does not support files conforming to older C++ standards. Motion has many syntax errors because the Intel HLS Compiler uses C++ as its foundation.

Applications That Compiled, But Were Not Autotunable

Five applications would compile using the default settings of the Intel HLS Compiler, but were not practical to autotune. The four applications related to arithmetic (dfadd, dfmul, dfdiv, dfsin) were successful for every attempt to synthesize, but did not vary in results because they do not contain any constructs that were conducive to the autotuning process. Namely, the two main areas of optimization that are being autotuned in this research included local memory optimizations for large memory constructs and loop optimizations; neither of which are present in the arithmetic programs. As such, each successive configuration yielded the same result (which is the same result as the default synthesis). Since the autotuner does not yield varying results, these applications are not included in the results to follow.

Blowfish differed from the arithmetic programs in that it would compile with only a slight modification. In the case of blowfish, an error is generated without modification:

> *Compiler Error: Complex address math not supported for load from filescope constant data.*
> *Compiler Error: Please simplify and ensure that there isn't pointer math for the accesses to the data set named: bf_init_S*

To overcome the errors, modifications need to be made as shown in Table 4.8.

Blowfish Original Code	**Blowfish Modified Code**

<pre>void local_memcpy (BF_LONG * s1,
    const BF_LONG * s2, int n)
{
  BF_LONG *p1;
  const BF_LONG *p2;
 	
  p1 = s1;
  p2 = s2;
 	
  while (n-- > 0)
    {
      *p1 = *p2;
      p1++;
      p2++;
    }
}</pre>
	<pre>void local_memcpy (BF_LONG * s1,
	    const BF_LONG * s2, int n)
	{
	  BF_LONG *p1;
	  const BF_LONG *p2;
	
	  p1 = s1;
	  p2 = s2;
	
	  while (n-- > 0)
	    {
	      p1[n-1] = p2[n-1];
	    }
	}</pre>

Table 4.8: Modifications Made to Blowfish

With the simple modification made to the pointer arithmetic, blowfish would yield a successful synthesis, but yielded many different errors when optimization placeholders were injected. Blowfish includes several large storage constants for the encryption keys. In the autotuning procedure, these variables could be placed into registers or memory; however, if registers are chosen as an option, another error occurs:

> *Compiler Error: hls_register attribute is given but cannot implement the storage in register*

Blowfish is also large and complex in design, which yields for complex and overly aggressive

optimization that takes large amounts of resources and time to compile. As such, not enough data is able to be collected to show autotuning results. Therefore, blowfish is also eliminated from the results section of this thesis.

Applications That Were Autotuned

The remaining four applications (aes, mips, adpcm, and gsm) were successful to compile and were conducive to the autotuning procedure. Although some size limitations prevented full results from adpcm and gsm, enough results were collected to make conclusions as discussed later in this chapter.

4.3 Establishing the Reference for the Autotuner

The autotuner point of reference is that of the original RTL synthesis that does not include any manually (or automatically) injected optimization techniques. This allows the compiler to yield a solution which is optimized using default tactics and strategies.

4.3.1 CHStone Applications

Only four of the applications were able to be autotuned by the autotuner; however, the solutions for the other applications are included here for completeness. Nine of the twelve CHStone applications were able to be compiled and synthesized with minimal alterations to the original source code. Table 4.9 shows the estimated synthesis results without performing any further Quartus compilations or optimizations (mapper and fitter). Three of the applications were unable to compile, as discussed in the previous section.

Included in Table 4.9 is a side-by-side comparison of the effects of changing the FPGA architecture. Specifically, two FPGA families were considered: Cyclone V and Arria 10. It is noted that changing the architecture has a major impact on the synthesis results. This is caused by differences of the internals of the FPGA families; each logic element, DSP block, and RAM block contain different components and sizes as well as different quantities of available elements. The Intel HLS Compiler takes these differences into account and attempts to optimize for the chosen FPGA family. For example, in the GSM application,

	Cyclone V				Arria 10			
CHStone Application	ALUTs	FFs	RAMs	DSPs	ALUTs	FFs	RAMs	DSPs
adpcm	952120	903635	1071	38	194430	193607	553	417
aes	82537	127156	696	0	75793	117497	242	0
blowfish	86588	84124	609	0	86385	86443	99	0
dfadd	8436	2880	12	0	8327	2431	4	0
dfdiv	76116	96135	29	28	55704	54487	5	28
dfmul	5250	2000	13	8	5267	1891	5	8
dfsin	103528	118777	62	46	78593	67411	12	46
gsm	365230	311910	91	7	42111	34011	8	56
mips	9005	5903	9	4	8202	3689	9	4

Table 4.9: Initial CHStone Application Results with Estimated Resources

the compiler's optmizations have resulted in far more DSPs being used in the Arria 10 than the Cyclone V, with a corresponding large decrease in ALUT and FF usage. To ensure that these differences don't impact the autotuner's configuration manipulator, the Cyclone V will be used for the remainder of this research.

Similarly, the same nine CHStone applications could further compile with Quartus using the mapper, fitter, and timing analysis provided by Quartus Prime. The results of these compilations are shown in tables 4.10 and 4.11.

4.3.2 DSP FIR Filter

Another circuit that is to be analyzed is a common finite impulse response (FIR) filter structure used in digital signal processing (DSP) applications. The filter chosen uses 129 taps to iteratively shape and conform an input signal to digital television specifications.

CHStone Application	ALUTs	FFs	DSPs	RAM Bits	F_{max}
adpcm	300417	887482	44	22712282	42.11
aes	94272	188092	0	2897868	50.45
blowfish	92361	148905	0	2064574	44.79
dfadd	13957	22553	0	77552	47.52
dfdiv	25863	84875	22	103330	48.04
dfmul	8709	16825	12	63680	52.56
dfsin	39859	113182	48	235352	26.34
gsm	58763	144522	14	226360	48.47
mips	2923	5116	6	6210	82.52

Table 4.10: Initial CHStone Application Results with Post-Mapped Resources for the Cyclone V FPGA Family

4.4 Setup of Autotuning for Intel HLS Compiler Applications

4.4.1 OpenTuner Configuration

OpenTuner is a highly adjustable and configurable framework for autotuning solutions. In addition to the built-in framework, in this research, a set of additional configuration options and modes have been added to OpenTuner that are specific to the autotuning of the Intel HLS Compiler. These configuration options will be used for all subsequent tests using the OpenTuner Framework.

- Memory optimizations may be selected for all variables and arrays, with valid options including *hls_register, hls_singlepump, and hls_doublepump.*
- Loop concurrency on all *for loops* with valid options including factors of the number of iterations.
- Loop unrolling on all *for loops* with valid options including factors of the number of iterations.
- Loop coalescing on all *for loops* with valid options ranging from one up to the number

CHStone Application	ALMs	FFs	RAMs	DSPs	F_{max} (Mhz)
adpcm	230514	433417	1223	287	195.66
aes	120812	205643	176	0	198.69
blowfish	97426	156802	179	0	145.39
dfadd	9806	13566	9	0	281.37
dfdiv	19621	47149	10	22	295.77
dfmul	6708	10876	6	12	270.64
dfsin	31446	66061	28	48	187.69
gsm	23860	37881	9	65	169.12
mips	2062	3307	10	6	329.06

Table 4.11: Initial CHStone Application Results with Post-Fitted Resources for the Cyclone V FPGA Family

	ALUTs	FFs	RAM Blocks/Bits	DSPs	F_{max}
Estimated	1082	974	1	2	N/A
Post-Mapped	479	850	4164	2	101.1
Post-Fitted	329	683	1	2	216.12

Table 4.12: Initial FIR Filter Results

of nested for loops.

- Compiler Parallelism is set to one. This refers to the number of configurations *in-flight* at a time; some projects are very large and consume most of a computer's RAM, so the configurations are run one at a time to avoid slow-downs and out-of-memory errors.

The user configuration settings are spread across three different files:

autotuner_config.json contains the list of default and optional configurable parameters that are not autotuned by default. Additionally, target values and weights are specified in this file. All simulations will use the same set of weights, and three different optimization strategies will be tested. If targets are also specified, the penalty for not

meeting the target (as per Equation 3.5 and 3.6 is specified by the *Target Penalty Factor*. The default autotuner configuration is summarized in Table 4.13 below.

	ALUTs	FFs	RAMs	DSPs	F_{max}
Balanced	4	2	2	2	4
Area	8	8	8	8	2
Performance	2	4	2	2	8
Target Penalty Factor	2.0	2.0	2.0	2.0	2.0

Table 4.13: Autotuner Configurations: Weights and Target Penalty Factor

intel_hls_parameters.json contains the full list of injected parameters, as well as all possible configurations for each placeholder. These values can be overwritten, but this file is also auto-generated. All autotuner configurations will use the default generated file.

run.sh contains the run-time configurations. This includes:

duration: The limit for how many seconds to run the autotuner.

parallelism: How many configurations to run in parallel (defaulted to one).

max_iterations: How many iterations to run before finishing.

filename: Name of the input source file that contains main().

optimize_strategy: Choice of balanced, area, or performance.

compute_fmax: True or false to compute post-mapped results.

quartus_compile: True or false to compute post-fitted results.

technique: Designer's choice of search space techniques (discussed below). AUCBanditMetaTechniqueA is used by default.

chip_type: Defaulted to Cyclone V. Intel HLS supports the Cyclone V or later families of FPGAs. A specific chip can be specified, but the 5CEFA9F23I7 is chosen by default.

generate_new_seed: Use previously generated seed file, or generate a new file.

seed_file: Name of the seed file to use. Seed files are in JSON format and provide a list of configuration parameters and their starting values. If no seed file is specified, a new random file is automatically generated.

{output files}: Designer's choice of output file names (CSV, logs, etc.)

Examples of autotuner_config.json and run.sh are provided in Appendix B.

Each configuration result is stored in a common database, as well as a CSV file for further analysis. Reports are generated with a timeline of the best *WNS* values and when the autotuner found them. Every configuration result is assigned a random identifier so that the results of the translation can be viewed after the autotuner has finished.

4.4.2 Search Space Size and Techniques

The OpenTuner framework provides a large set of available search techniques for exploring a variety of search spaces. The framework also provides provisions for creating custom or altered techniques. A unique feature of OpenTuner is the ability to apply multiple techniques to a search space, and allocate more tests to techniques which tend to perform better. Since all results are stored in a common database, every technique benefits from the results generated by other techniques. The method that OpenTuner explores the search space is not a focus of this research, and as such the AUCBanditMetaTechniqueA will be used for all configurations. This group of techniques includes four techniques: Differential Evolution, Uniform Greedy Mutation, Normal Greedy Mutation, and the Random Nelder Mead technique. For more information on these techniques, please see Section 3.2.1.

The search space grows for every variable and loop defined in the source file as discussed in Section 3.2.2. For the given list of test programs, the approximate search space size is shown in Table 4.14. It is clear from Table 4.14 that exhaustive exploration of the search space will not be possible for all but the smallest designs.

Program	Search Space Size
adpcm	$10^{76.95}$
aes	$10^{20.1}$
mips	$10^{12.56}$
gsm	$10^{27.76}$
FIR Filter	$10^{4.44}$

Table 4.14: Search Space Size of Each Test Program

As the complexity of a program increases, and as resource usage increases, so does the average time to compile. Each additional stage of the Quartus tool (simulator, mapper, fitter, timing analysis, etc) that is used to improve the quality of results also increases the time to compile. Therefore, reducing the number of stages and simplifying the design will greatly reduce the time to compile, which in turn provides better exploration of the search space.

	Time to Compile (seconds)			
	Estimated	Post-Mapped	Post-Fitted	Test Computer
adpcm	480	7427	*	Intel i9-9900k 64GB DDR4 RAM NVME M.2 SSD
aes	120	2112	3526	
mips	7	82	165	
gsm	44	1115	1595	
FIR Filter	5	50	78	

Table 4.15: Initial Time to Compile (Without Injected Optimizations)

** adpcm post-fit failed with message "failed to allocate memory".*

Table 4.15 shows the duration in seconds for the compile time. This compile time does not include any simulation or verification. The command for each of these include:

`Estimated:`

```
  time i++ source.c -march="Cyclone V" -o output --simulator none
     -g0 --fpga-only
```

```
Post-Mapped:
  time ( i++ CHStone_Autotune/mips/mips.c -march="Cyclone V"
    -o results/output.prj --simulator none -g0 --fpga-only &&
    cp postmap.tcl results/output.prj/quartus &&
    sh quartus_sta_postmap.sh results/output.prj/quartus )
```

```
Post-Fitted:
  time i++ source.c -march="Cyclone V" -o output --simulator none
     -g0 --fpga-only --quartus-compile
```

To be able to get a performance metric, a Quartus timing analysis tool must be used. As seen in Table 4.15, there is a significant time savings by eliminating the fitter and autotuning based on the post-mapped results. This optimization, however, is only available when using the Cyclone V family of FPGAs because later families require some level of the fitter to have ran to be able to perform a timing analysis. The timing analysis performed by the post-mapper is not going to have the same quality of results in comparison to the fitter, but provides sufficiently accurate results for the autotuning process [9].

If resource usage is the only metric of interest, a significant time savings is provided by eliminating the mapper and fitter steps and relying upon the estimated resources of the Intel HLS Compiler. This will greatly speed up the autotuning and provide a much better exploration of the search space. It is also important to note that different solutions will use varying amounts of resources, and in many cases will use significant amounts of extra resources; these solutions will take much longer than the initial compiles done without injected optimizations.

4.5 CHStone Autotuning Results

The goal of the autotuner is to provide different synthesized solutions to a given function written in C. With each adjustment of the configuration, a potentially different solution will be realized. The *objective* of the autotuner is to find better solutions than the default solution provided by the Intel HLS Compiler based upon the configuration settings provided by the designer as mentioned in Section 4.4.1. The initial benchmark in which all comparisons are made is the result of the synthesis without any injected optimization parameters; the Intel HLS Compiler uses all default optimization strategies. In all of the figures below, the initial starting point is highlighted with a red line, allowing easy comparisons against the autotuned configuration results. Values indicated above the line are worse (with the exception of F_{max} performance) and all values below the line show improvements.

Occasionally, there will be a truly optimal solution that yields the highest maximum operating clock frequency for performance, and uses the fewest amount of resources. However, in many cases no one solution will be the best in all categories. In such cases the preferred solution is the one that best meets the designer's requirements. The designer specifies these preferences through the weights that are applied to the results returned by the Intel HLS Compiler. Refer to Table 4.13 for the weights used for the tests performed below.

4.5.1 Establishing a Starting Point

The autotuning process starts from a random seed configuration (unless a seed file is specified and provided). This starting point can have a significant impact on the autotuning results as the number of compilations performed is very small in comparison to the size of the search space. To determine the impact on the results by adjusting the starting point, the same design is repeated with different randomized seed configurations. For this test, mips and aes from the CHStone Benchmarking Suite will be used.

Design	Seed	Best WNS Value (Initial = 34)	Best WNS Iteration	Design	Seed	Best WNS Value	Best WNS Iteration
mips	1	26.93595	90	aes	1	30.27241	205
mips	2	26.93595	91	aes	2	30.27898	72
mips	3	26.93595	48	aes	3	30.27241	75
mips	4	26.93595	69	aes	4	30.27580	220
mips	5	26.93595	162	aes	5	30.27559	206
mips	6	26.93595	37	aes	6	30.27241	203
mips	7	26.93595	47	aes	7	30.27241	165
mips	8	26.93595	151	aes	8	30.27241	197
mips	9	26.93595	46	aes	9	30.27580	73
mips	10	26.93595	105	aes	10	30.27241	188

Table 4.16: Repeated Tests with Different Seeds

The repeated tests shown in Table 4.16 all use the same designs, but a different *seed*, or starting configuration. Each test was ran for 1000 iterations, and the first iteration to achieve the lowest weighted normalized sum value is recorded. The results show that all tests eventually achieved results of similar quality, but that the initial seed value has a significant impact on the number of iterations required to find the best weighted normalized sum value. In the case of aes, some seed values did not allow the autotuner to hone into the same solutions as other configurations. This problem becomes more apparent the larger the design. Larger designs have exponentially larger search spaces, and typically take longer to compile. This will result in a poorer exploration of the search space and a higher dependence on a good starting seed value.

4.5.2 Correlation Between Estimated vs Post-Mapped and Estimated vs Post-Fitted Results

A series of compilations was performed to investigate the relationship between the estimates from the different compilation levels. To isolate the effect of each additional compilation level, the same seed (starting point) was used in each case. The mips design was used for this comparison due to its relative short compile time.

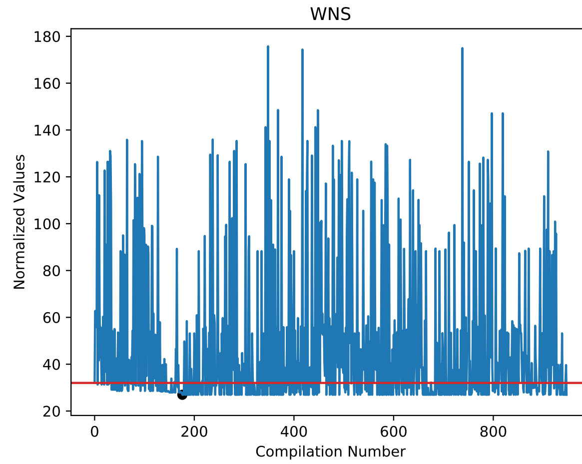
Source of Results	ALUTs/ALMs	FFs	RAM Blocks (M10K)/Bits	DSPs	F_{max} (Mhz)
Estimated	8723 (ALUTs)	5893	3 (Blocks)	4	
Post-Mapped	2946 (ALUTs)	4888	6146 (Bits)	6	82.52
Post-Fitted	2164 (ALMs)	4465	9 (Blocks)	6	185.49

Table 4.17: Same-Seed Results for mips

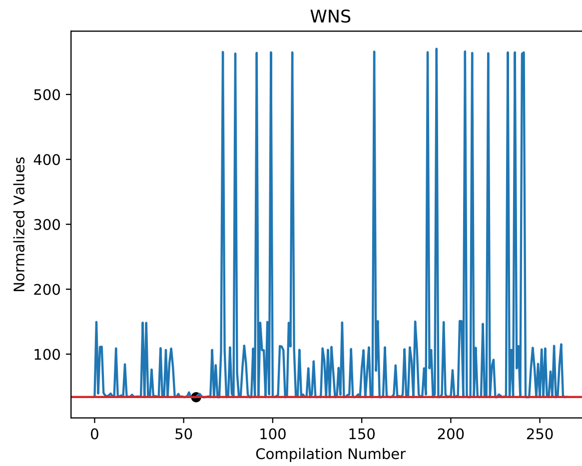
Table 4.17 and Figure 4.1 show that there is no clear correlation between estimated results, post-mapped results and post-fit results. Each of the levels of the tool yield different results, and report slightly different metrics which makes direct comparisons difficult. For example, estimated and post-mapped results report number of ALUTs, but post-fit results yield the number of ALMs used. ALMs also include some FFs, so the number of FFs used by the post-fit results will likely show lower than total accumulated FFs.

Another comparison that is of interest to the designer is the comparison of how the final optimal configurations differ after autotuning. Table 4.18 simplifies the configuration comparison by showing only the parameters that had the greatest impact on the results for this design. The major contributions are the loop unrolling and the local memory storage of the array types.

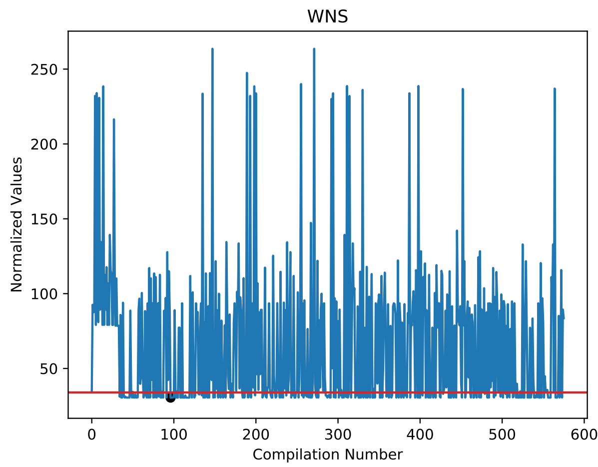
The results show that the best configuration identified by the autotuner is remarkably consistent, regardless of which set of estimates is used. As noted earlier, the resource and performance estimates from the Intel HLS Compiler are less accurate than those from the Quartus mapper and fitter; however, the Quartus mapper and fitter are typically slow to run,



(a) Mips Estimated



(b) Mips Post-Mapped



(c) Mips Post-Fitted

Figure 4.1: Same Seed Comparison of Tool Levels - mips

Parameter	Estimated	Post-Mapped	Post-Fitted
loop_unroll_0	1	1	1
loop_unroll_1	1	1	2
loop_unroll_2	1	1	2
array_0	hls_doublepump	hls_doublepump	hls_doublepump
array_1	hls_singlepump	hls_singlepump	hls_singlepump

Table 4.18: Same-Seed Results for mips

thereby limiting the ability of the autotuner to explore design space and find good solutions. The estimated and the post-mapped resources showed very similar resulting configurations and only differ through local storage of individual variables (not shown). Post-fit results, however, do show some differences on the number of loop unrolling, although the impact of the differences is small. These differences are likely related to the performance increase that the fitter realizes when actually fitting the component. As mentioned earlier, the fitter will yield the highest quality of results, but also takes the longest to compute. Table 4.18 does show that an autotuning system based on the Intel HLS Compiler estimates narrow to a configuration that is close to optimal for resource usage, and the post-mapped results gives a similar analysis with performance metrics included, although the true optimal solution must be found using post-fit results. All levels of the tool could be found useful, depending on the stage of the design process that the designer is currently exploring. If the designer is interested in finding a configuration that potentially optimizes the design by reducing resources, the Intel HLS Compiler estimated results would be valuable. If the designer is interested in meeting specific design requirements in terms of resource usage or performance, the post-fit results may be the level of interest, and post-map gives a balance in-between these two extremes.

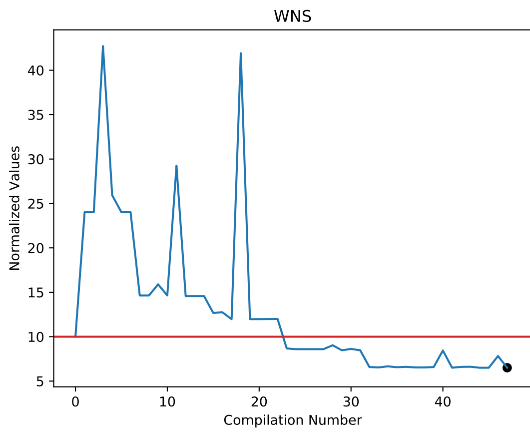
4.5.3 WNS CHStone Application Results

Next, the proposed autotuning system was applied to the entire CHStone benchmarking suite. As discussed earlier, only four of the CHStone applications (namely adpcm, aes,

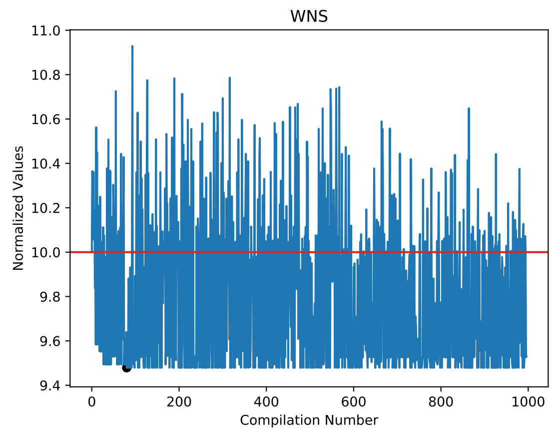
mips and gsm) yielded useful results, with the remainder of the applications either failing to compile or always optimizing to the same design. The autotuning process was repeated a number of times for each design in order to investigate the effect of various design goals (area vs balanced vs performance) and various sources of per-configuration results (Intel HLS Compiler estimates vs Quartus post-mapper estimates vs Quartus post-fitter results). In each case, the WNS values described by Equation 3.3 were used to give an overall indication of how *good* a configuration is.

An example set of results is shown in Figures 4.2 to 4.4 for the balanced, area and performance optimization goals based on the Intel HLS Compiler estimated results. In adpcm, aes, and mips an improvement is demonstrated with the autotuner identifying configurations with normalized values below the starting point (10 in the case of Figure 4.2, as per the balanced weighting scheme found in Table 4.13). For gsm, however, the default Intel HLS Compiler compilation yielded the best result, such that all autotuned configurations were inferior in terms of WNS. It should be noted that only a small portion of the search space has been explored for each of the results, and it is possible that a more optimal solution could be realized if the autotuner was given more time or more processing power.

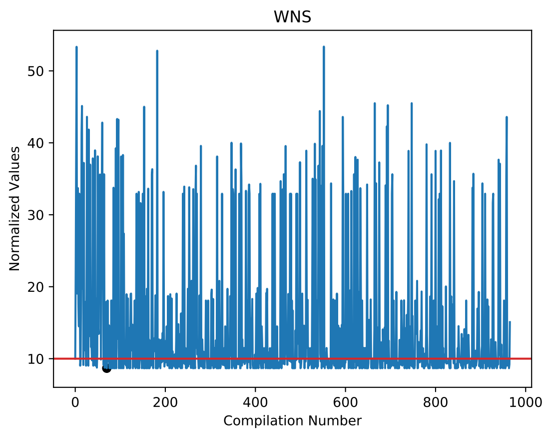
When observing the results in Figure 4.2, large differences in the number of compilations performed by the autotuner may be observed. This is attributed to the difference in the compile time for the different compilation levels, the size of the designs, and for how long the autotuner was ran for the given simulation. For the purpose of the research conducted, the number of compilations needed to find the optimal result is not of primary interest because the methods of exploring the search space are not being discussed. The focus of the research is whether the autotuner is able to find configurations that are optimal or useful to the designer. As such, the autotuner is ran for (up to) 1000 iterations, or for the longest period of time available at the time (usually overnight, but varies). Most of the results show an optimal configuration is found within the first third of an autotuning run, but this is not conclusive because the autotuner is unable to fully explore the search space. In practice, an organization or designer could likely achieve better results by running the autotuner on a larger server or cluster of computers in order to better explore the design space.



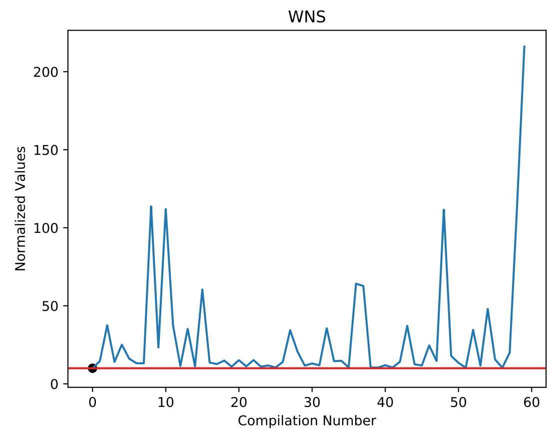
(a) adpcm



(b) aes

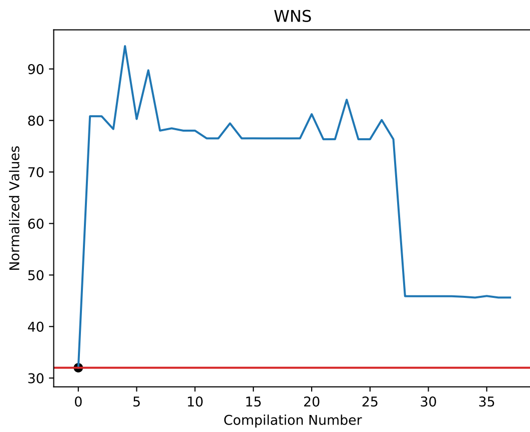


(c) mips

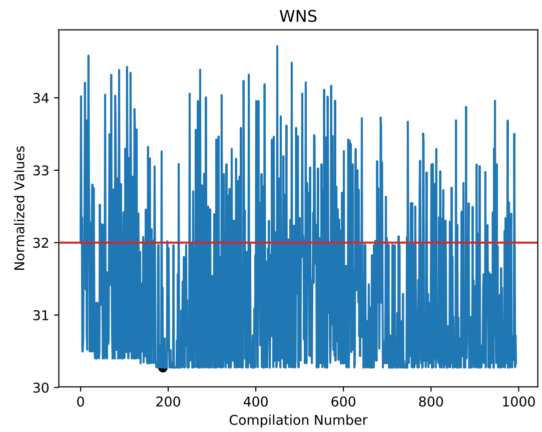


(d) gsm

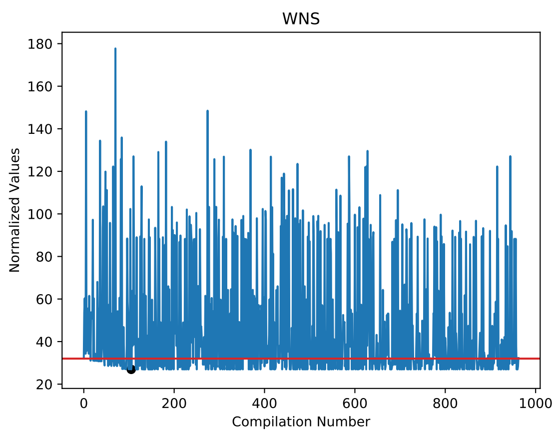
Figure 4.2: WNS Estimated Balanced Results



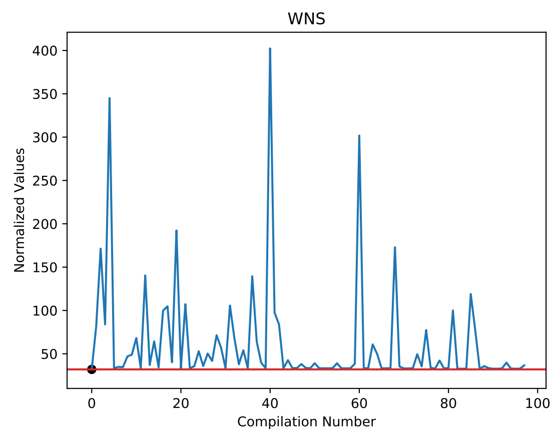
(a) adpcm



(b) aes

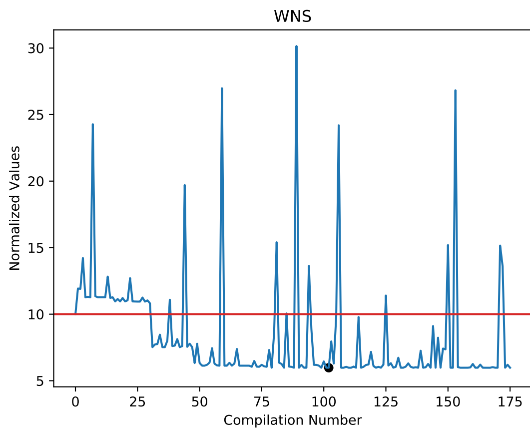


(c) mips

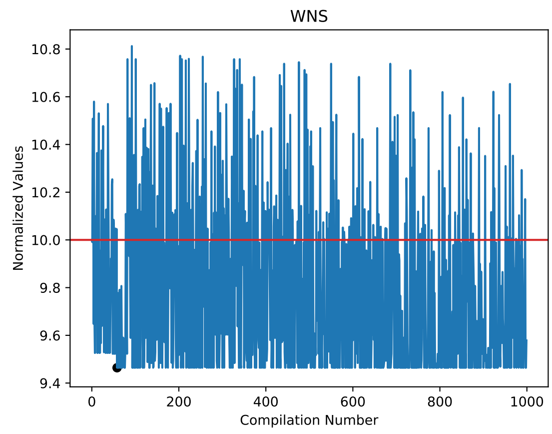


(d) gsm

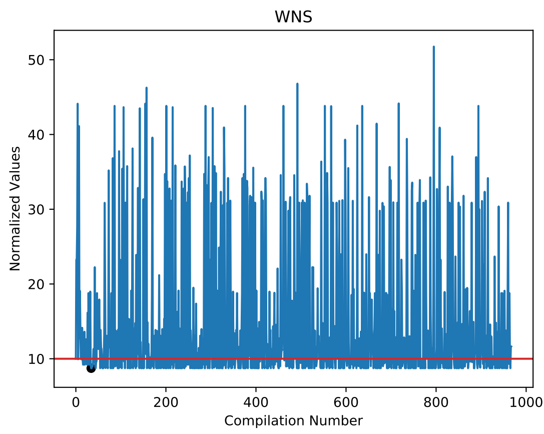
Figure 4.3: WNS Estimated Area Results



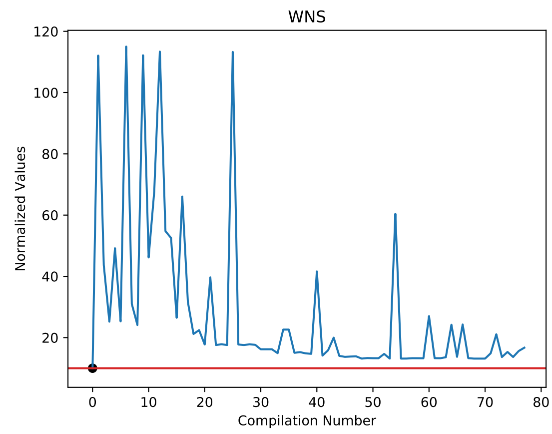
(a) adpcm



(b) aes



(c) mips



(d) gsm

Figure 4.4: WNS Estimated Performance Results

The complete set of autotuning results for all optimization goals and all sources of results is plotted in a series of figures found in Appendix C. In some cases, compilations failed to produce any data points because the computer ran out of resources when compiling. For completeness, the graphs are presented in the figures but will be blank when no data is collected. Overall, it is important to note that only the lowest point in each graph is important, as it represents the best configuration identified by the autotuner. To aid the visualization process, this optimal point is highlighted with a black circle.

The relative improvement provided by the autotuner is represented by the difference between the WNS of the default Intel HLS Compiler configuration (red line) and that of the best autotuned configuration (black circle). These values are based upon the estimated resources produced by the Intel HLS Compiler. The relative improvement is calculated as:

$$Relative\ Improvement(x_{best}, x_{initial}) = \frac{x_{initial} - x_{best}}{x_{initial}} * 100\% \tag{4.1}$$

In the event that the autotuner does not find a better configuration than the initial (default) configuration, Equation 4.5.3 will yield a relative improvement of 0% which shows no improvement was found. Table 4.19 summarizes the relative improvement for each of these designs with each optimization goal.

	Balanced			Area			Performance		
	Initial	Best	Improvement (%)	Initial	Best	Improvement (%)	Initial	Best	Improvement (%)
adpcm	10	6.505	34.95	32	32	0	10	5.976	40.24
aes	10	9.479	5.21	32	30.272	5.4	10	9.464	5.36
mips	10	8.671	13.29	32	26.936	15.825	10	8.731	12.69
gsm	10	10	0	32	32	0	10	10	0

Table 4.19: Relative Improvement Based on Estimated Resources

The normalized results of the estimated resource usage shows that the autotuning process can find solutions that are reduced in resource usage. Adpcm shows a large variation in the relative improvement when different goals are used, ranging from 0% for an area-based optimization to over 40% for a performance-based optimization. As per the discussion in Section

4.5.1, this may be attributed to the starting point of the initial configuration. Adpcm is also the largest of the CHStone Applications tested, which limits the number of compilations performed by the autotuner during the test duration. Aes and mips show a more constent level of improvement with aes being just over 5% and mips being 12% to 16%. Gsm, on the other-hand, shows no improvement from the autotuner.

Each of the graphs shown in Figures 4.2, 4.3, and 4.4 show a downward trend as the number of autotuner iterations increases. This shows that the autotuner's search techniques are intelligently selecting parameters that yield better results. There is also a large discrepancy of the number of compilations performed for each of the CHStone applications; this is caused by adpcm taking significantly longer to compile than mips and aes. As the number of resources used increases, so does the compilation time. Table 4.15 shows the initial time to compile for each design. Gsm has significantly more variance in the number of resources used from one iteration to the next. Although the initial number of resources used is relative small, forced optimization using configurations that have poor values caused an extreme amount of resources to be used and would cause an overly complicated design. These compilications would eventually cause a time-out with an error message *Failed to allocate memory*. To fix this, some of the C-code would need to be re-written to avoid unnecessarily complicated access patterns to memory that are difficult to synthesize. This includes (many) function calls that are not needed and difficult function return values. Gsm also showed the highest variance in resources used from one configuration to another, with a minimum of less than 500k ALUTs and a maximum of over 12 million ALUTs, shown in Figure 4.5.

	Balanced			Area			Performance		
	Initial	Best	Improvement (%)	Initial	Best	Improvement (%)	Initial	Best	Improvement (%)
aes	14	13.6462	2.53	34	33.1319	2.55	18	17.3114	3.83
mips	14	13.9094	0.647	34	33.6240	1.106	18	17.8169	1.017
gsm	14	14	0	34	34	0	18	18	0

Table 4.20: Relative Improvement Based on Post-Mapped Resources

Introducing the mapper adds performance metrics (F_{max}) into the WNS calculations. This results in a similar trend of improved results over time for aes and mips, but with sig-

nificantly less compilations performed in a similar amount of time. The relative improvement is also reduced, but still shows an improvement of up to 4%. As the mapper takes a significant amount of time to compile, adpcm was unable to yield any successful compilations. This could be because the Quartus mapper is unable to fit the whole design into the computer's available memory, or the complexity of the design just takes too long to adequately perform in the time allotted. Similarly, gsm was able to get some successful compilations done, but not enough data points are produced to provide any meaningful results.

	Balanced			Area			Performance		
	Initial	Best	Improvement (%)	Initial	Best	Improvement (%)	Initial	Best	Improvement (%)
aes	14	13.377	4.45	34	32.736	3.72	18	17.312	3.82
mips	14	12.659	9.58	34	29.699	12.65	18	16.376	9.02
gsm	14	14	0	34	34	0	18	18	0

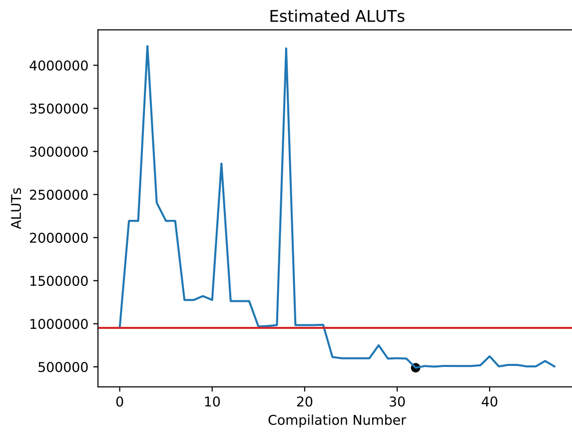
Table 4.21: Relative Improvement Based on Post-Fitted Resources

Introducing the fitter shows results that more closely coincide with the estimated resources from aes showing approximately 4% improvement and mips showing over 10% improvement. The fitter suffers from the same problem as the mapper in that compilations with larger designs take too long, such that the autotuner may not have time to sufficiently explore the design space. The fitter is ran in addition to the mapper, and uses even more computer resources to perform. It does, however, provide the most accurate representation of what will happen in a real FPGA as the resource usage and performance metrics are actual and not estimated.

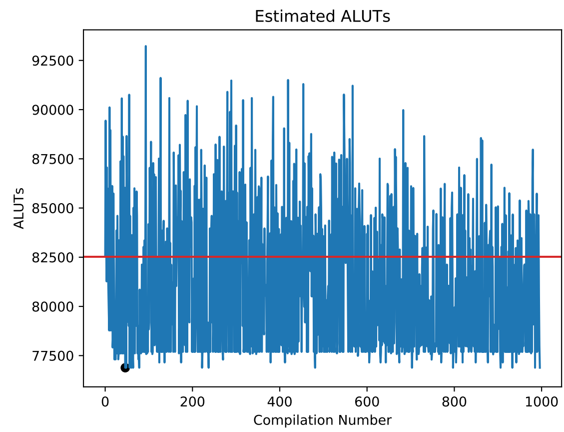
4.5.4 Other CHStone Application Results

To confirm that adjusting the optimization scheme actually benefits the end result, this section takes a closer look into the individual metrics which contribute to the results. Figure 4.5 shows the estimated amount of ALUTs for the CHStone applications using a balanced weight scheme during the autotuning process.

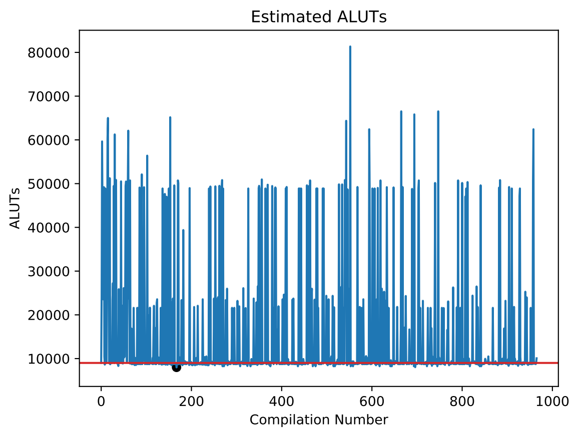
Gsm could, on a good compile, yield less than 400k estimated ALUTs used in a design, but if the configuration is chosen poorly it would require over 12 million ALUTs (among



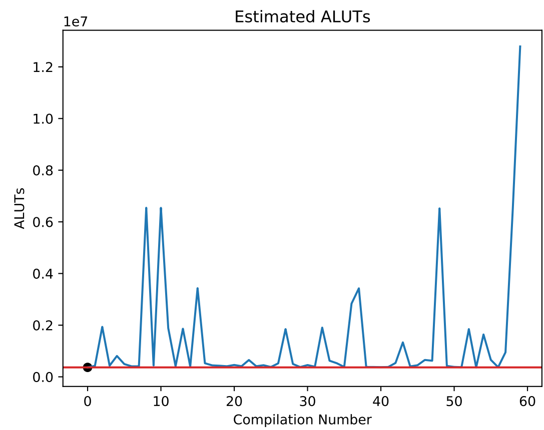
(a) adpcm



(b) aes



(c) mips



(d) gsm

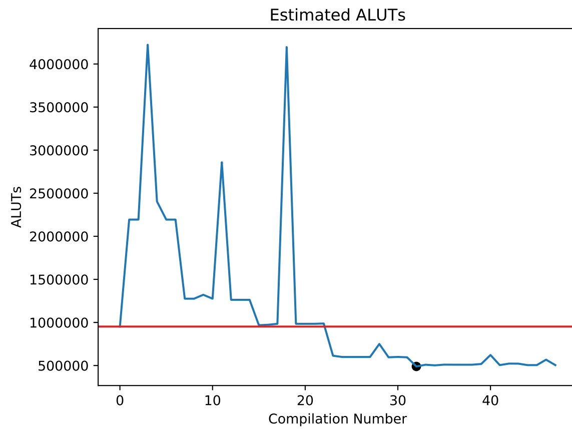
Figure 4.5: CHStone Estimated Resources - ALUTs

other resources). Given that the largest Cyclone V FPGA have around 500k ALUTs, this is clearly an unreasonable result. This causes a huge bottleneck in the autotuner as such compilations would take a very long to compile, and occasionally even timeout with a failed to allocate memory error (on a 64 GB RAM computer) or crash the autotuner. For this reason, the number of compilations successfully performed for gsm and adpcm varied because compilations would complete in a timely manner until a bad configuration is chosen, which would then halt the autotuner. Adpcm follows in a similar manner, but does show significant improvement from the initial compilation: from just under one million ALUTs down to approximately 500k.

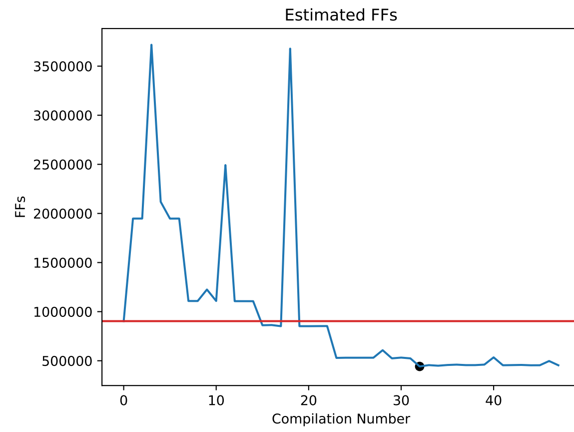
Smaller designs, on the other hand, show a much more consistent amount of ALUTs. This is likely due to the fact that the design space is more fully explored by the autotuner due to shorter compile times. There are still the occasional *bad* configurations chosen, but they show less variation in comparison to the larger designs. This is likely because the larger designs user larger data structures which have a larger impact on the resources used, or the designs used more loops of higher iterations, and if a configuration is chosen poorly for each of the loops on the same configuration, a design will be created that is too large for the computer to handle. Essentially, a larger design with more parameters leaves more room for the compiler to go wrong.

Table 4.22 and 4.23 summarize the initial (pre-autotuning) and post-autotuning results for each of the designs considered. It is clear from the tables that autotuning has uncovered solutions which are advantageous from the perspective of resource usage and/or performance. Specifically: adpcm, aes and mips all showed solutions that improved ALUTs, FFs, RAM blocks, or DSP blocks at some trade-off of the other resources. Identifying such solutions manually without an autotuner would be difficult and time-consuming.

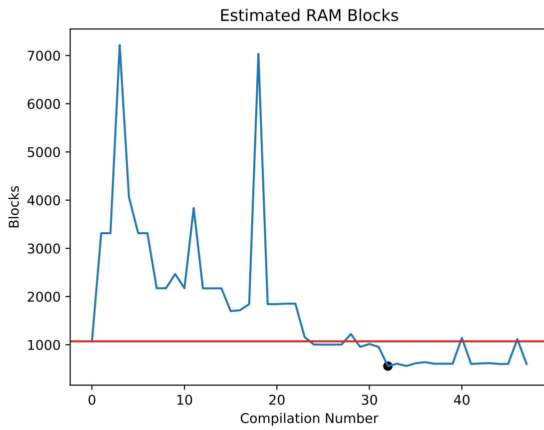
When observing the four resource elements resulting from a compilation for adpcm as shown in Figure 4.6, a strong correlation between ALUTs, RAMs, and FFs occurs for non-optimal configurations and the time taken to compile. That is to say, if the number of ALUTs, FFs or RAM blocks used is sufficiently high, and the time to compile is long, it is very likely that a bad configuration is chosen. If the autotuner could preemptively determine



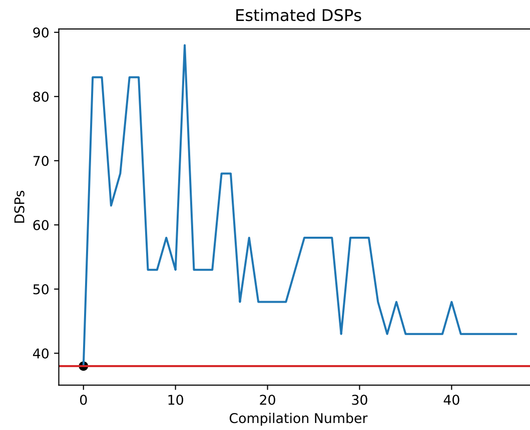
(a) Adpcm ALUTs



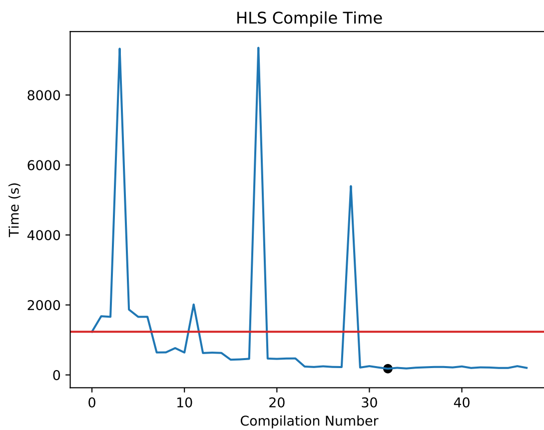
(b) Adpcm FFs



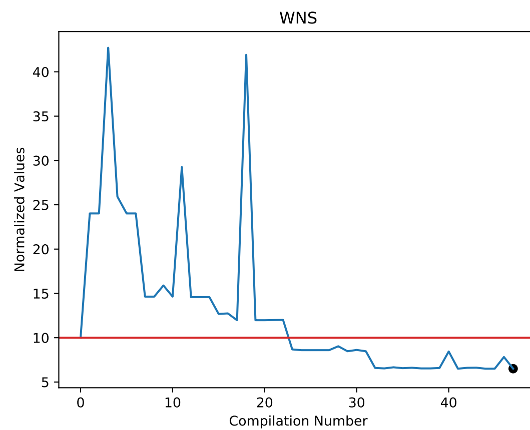
(c) Adpcm RAM Blocks



(d) Adpcm DSP Blocks



(e) Adpcm Intel HLS Compile Time



(f) Adpcm WNS Results

Figure 4.6: CHStone Estimated Resources for adpcm

	Initial Estimate (WNS = 10)				Best WNS Autotuned Result				
CHStone Application	ALUTs	FFs	RAM Blocks	DSPs	WNS	ALUTs	FFs	RAM Blocks	DSPs
adpcm	952120	903635	1071	38	6.5053	504794	453843	601	38
aes	82522	127156	696	0	9.4790	77690	118743	642	0
gsm	365230	311910	91	7	10	365230	311910	91	7
mips	9005	5903	9	4	8.6713	8723	5893	3	4

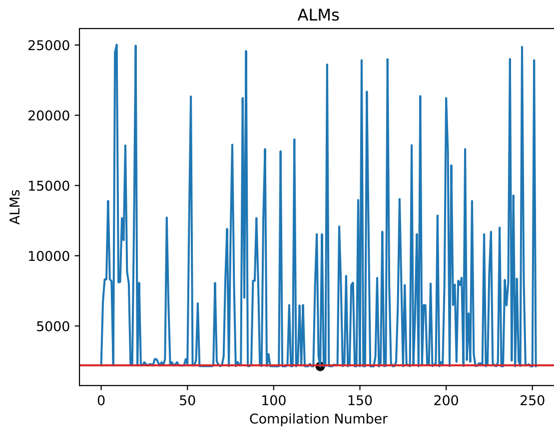
Table 4.22: Initial CHStone Application Results Versus Best WNS Results

	Initial Estimate				Best-In-Category Autotuned Result			
CHStone Application	ALUTs	FFs	RAM Blocks	DSPs	ALUTs	FFs	RAM Blocks	DSPs
adpcm	952120	903635	1071	38	489255	442017	557	38
aes	82522	127156	696	0	76883	118743	612	0
gsm	365230	311910	91	7	365230	311910	91	7
mips	9005	5903	9	4	7988	4516	1	4

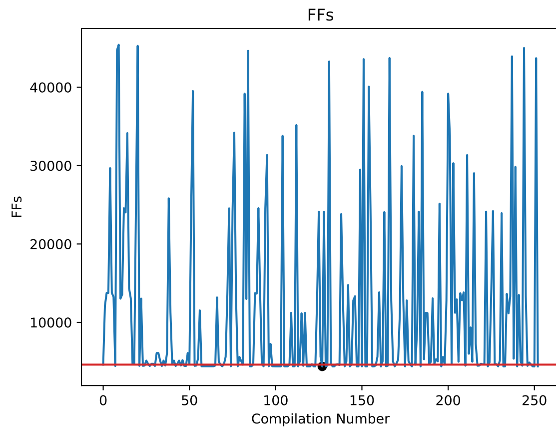
Table 4.23: Initial CHStone Application Results Versus Best-In-Category Results

if a configuration is going to be a bad configuration, a significant amount of compile time could be saved; three of the forty eight compiles account for over 30% of the accumulated compile time. Since a strong correlation between estimated results and post-fitted results can be found (as shown in Figure 4.10), poor estimated results could be used to preemptively determine that the fitter is not worth running. This potential extension to the tool is discussed further in Section 5.3.

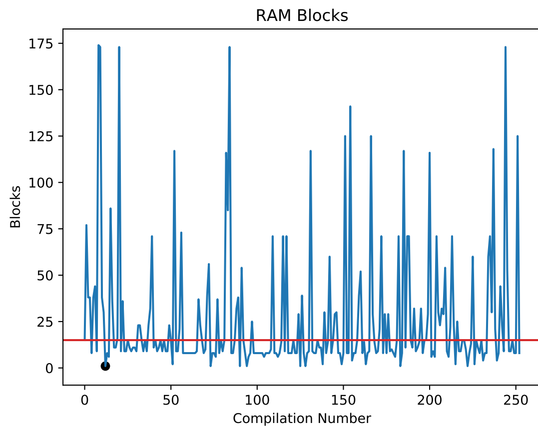
Similarly, the post-fit results in Figure 4.7 for mips show the same correlation between resource usage and time to compile as the estimated resources for adpcm.



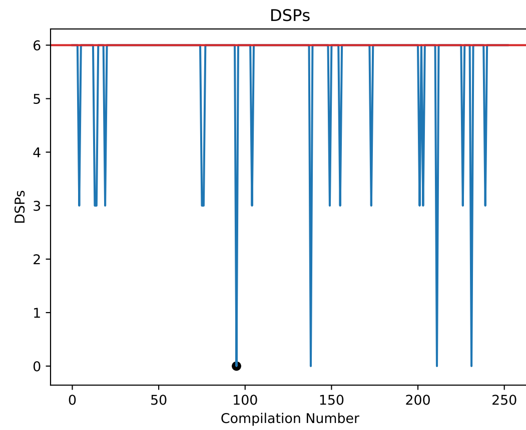
(a) Mips ALMs



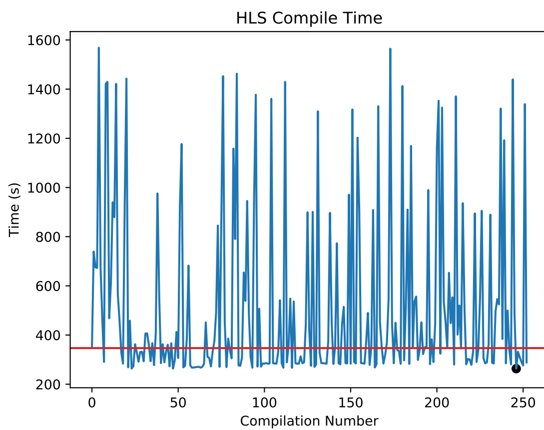
(b) Mips FFs



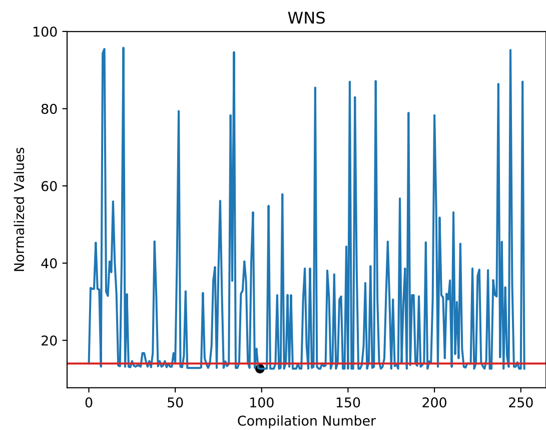
(c) Mips RAM Blocks



(d) Mips DSP Blocks



(e) Mips Intel HLS Compile Time



(f) Mips WNS Results

Figure 4.7: CHStone Post-Fitted Resources for mips

4.6 DSP FIR Filter Autotuning Results

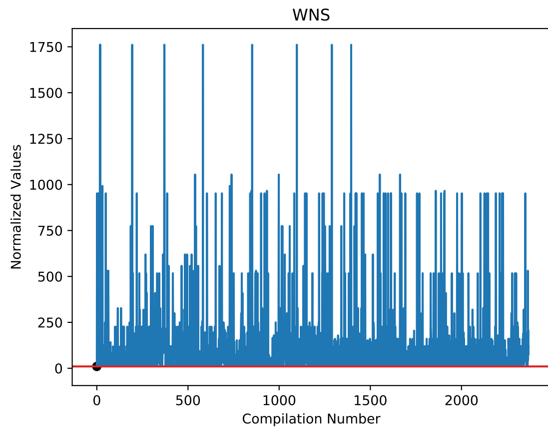
The DSP FIR Filter used for this test is a 129 tap finite impulse response filter. The filter involves two loops; one for shifting data into the component and the other for multiplying and accumulating the data by a pre-defined set of coefficients. In the design of the component, the coefficients are stored as local variables to the component. This ensures that they become an internal resource to the component. If they are declared as global variables they would need to be passed into the component as a streamed memory interface, which is out of the scope of the current autotuner. The complete C code for this filter is provided in Appendix A.

The complexity of the FIR Filter is low; there are only two for loops and a few variables. This makes the search space relatively small, which allows for the autotuner to perform a more complete search. With the way the current C source code is written, the Intel HLS Compiler is able to come up with a solution that is fairly optimized in terms of resources, as shown in Figures 4.8 to 4.10.

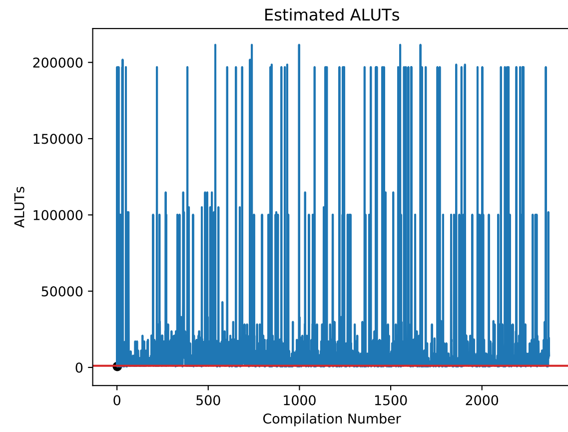
Integrating the full Quartus compile shows that the initial solution yields the highest F_{max} as well as the lowest amount of resources. This is, however, misleading as the performance for this component should include additional metrics beyond just F_{max} such as *throughput*. The throughput for a filter represents the maximum rate at which data may flow through the filter. If the design takes multiple clock cycles to generate an output sample, the throughput is decreased. Conversely, if the filter can process multiple samples within a single clock cycle, the throughput is increased. This is discussed further in Section 4.6.1.

4.6.1 Verification of Results

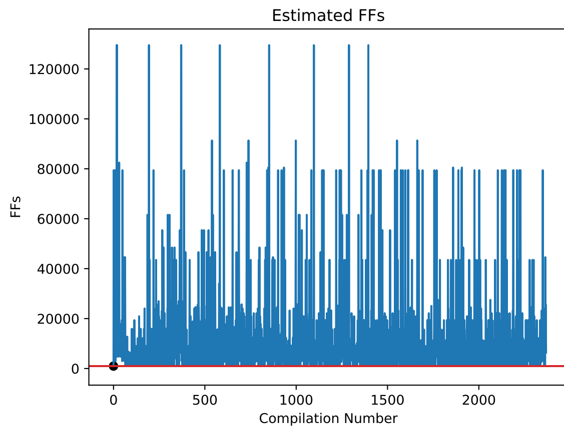
This research has assumed (and chosen optimization parameters to ensure) that the Intel HLS Compiler always produces a result that is functionally equivalent to the original C source file, and that the tool itself does not need to be checked for correctness. As a sanity check, a test comparing the expected and actual outputs for the FIR Filter can be performed. The Intel HLS Compiler creates an IP file that can be compiled in Quartus. This compiled



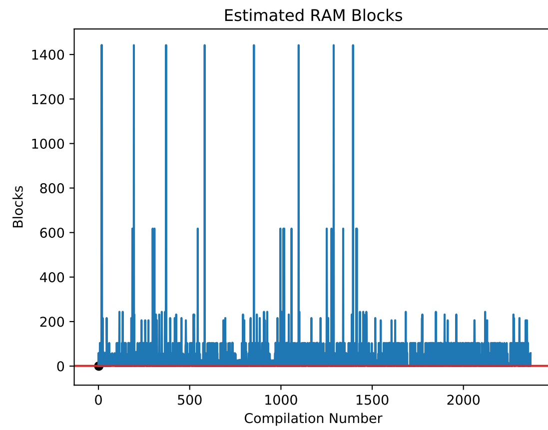
(a) WNS Results



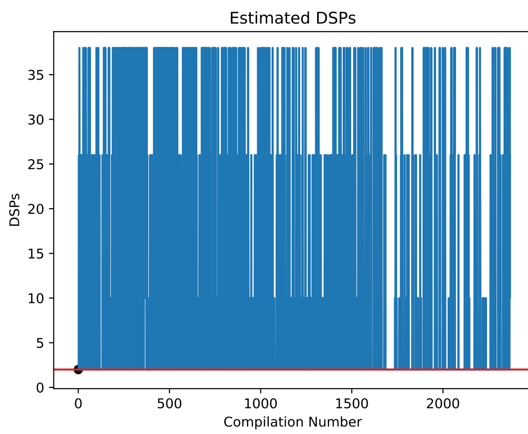
(b) Estimated ALUTs



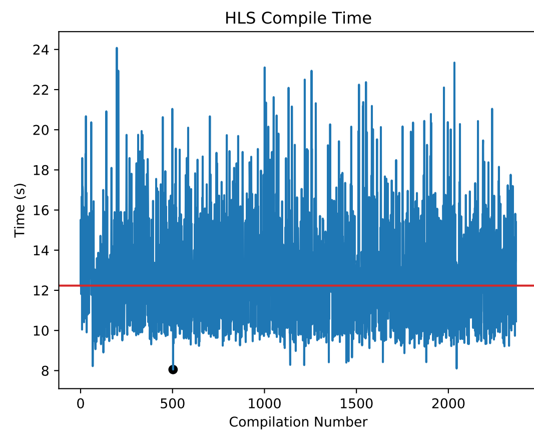
(c) Estimated FFs



(d) Estimated RAM Blocks

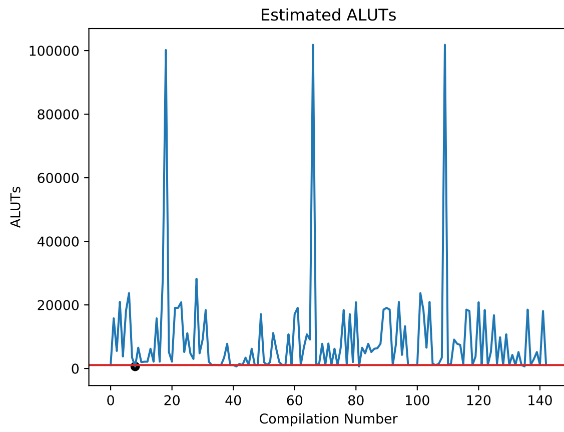


(e) Estimated DSP Blocks

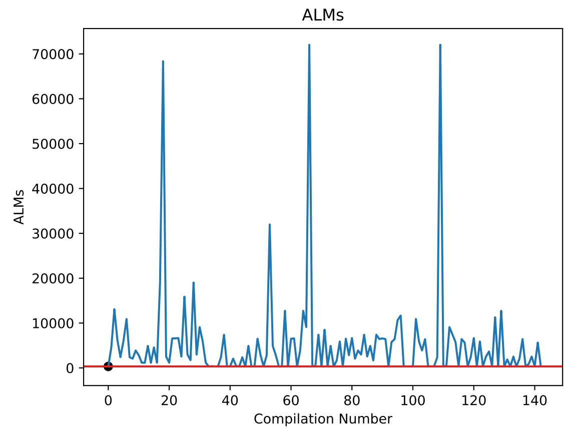


(f) Intel HLS Compile Time

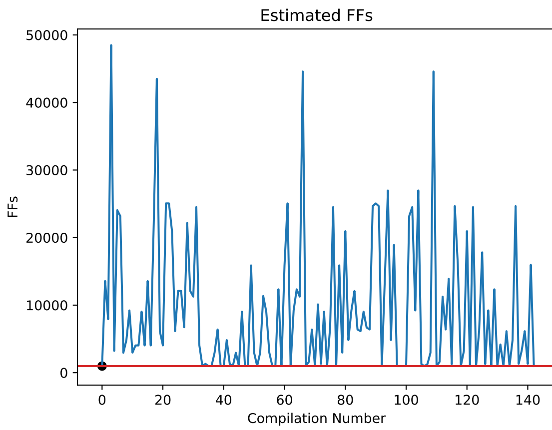
Figure 4.8: FIR Filter Estimated Results



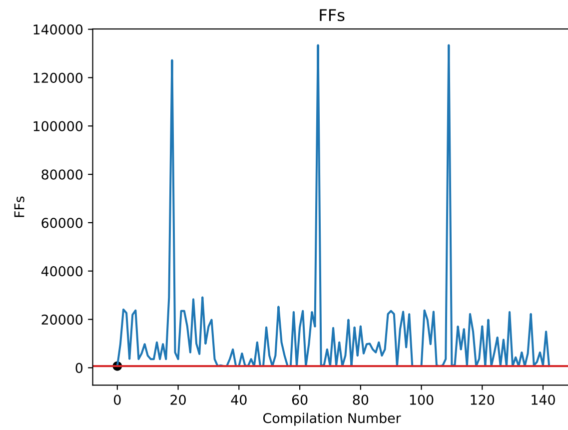
(a) Estimated ALUTs



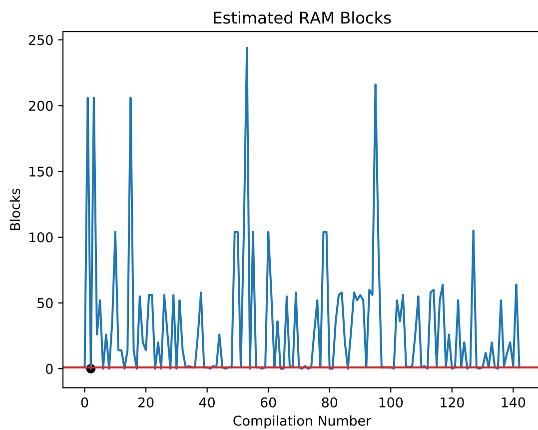
(b) Post-Fitted ALMs



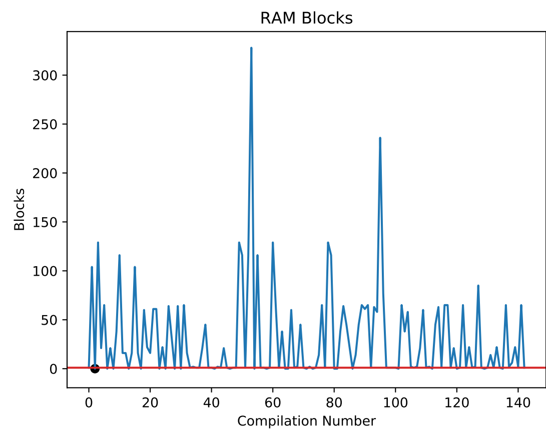
(c) Estimated FFs



(d) Post-Fitted FFs

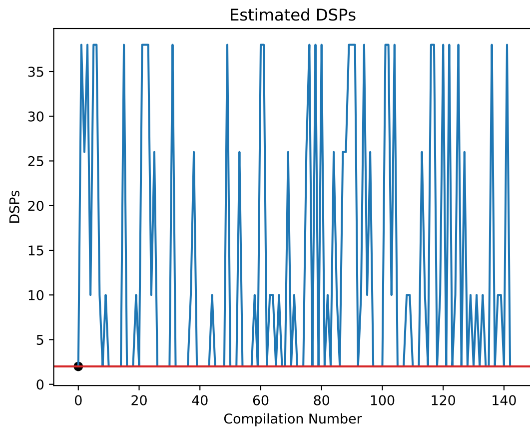


(e) Estimated RAM Blocks

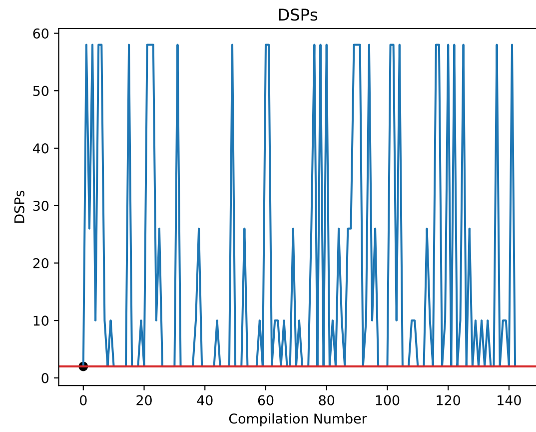


(f) Post-Fitted RAM Blocks

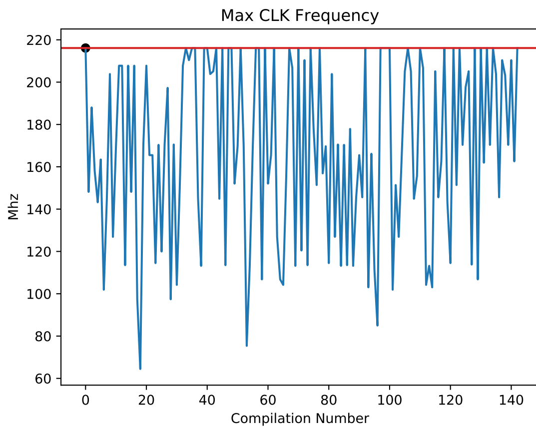
Figure 4.9: FIR Filter Estimated vs Post-Fitted Results



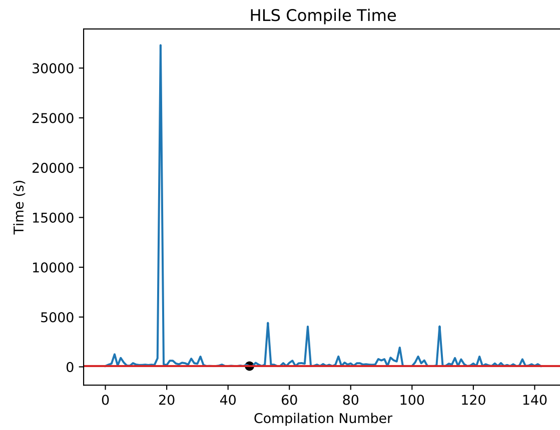
(a) Estimated DSP Blocks



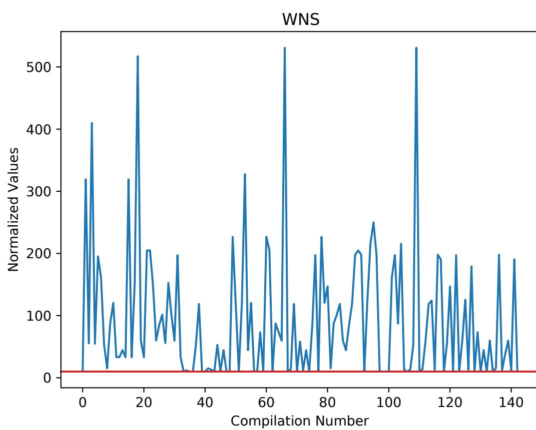
(b) Post-Fitted DSP Blocks



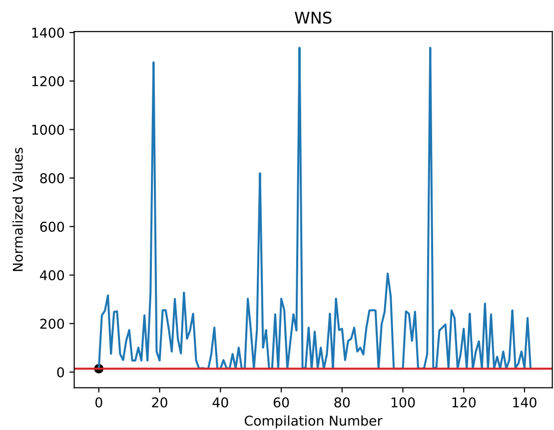
(c) Post-Fitted F_{max}



(d) Post-Fitted Compile Time



(e) Estimated WNS Results



(f) Post-Fitted WNS Results

Figure 4.10: FIR Filter Estimated vs Post-Fitted Results

result can then be simulated for analysis in Modelsim. Each synthesized solution contains the signals found in Table 4.24.

Signal	Auto-Generated	Function
clk	yes	Base clock for the system.
clock2x	yes	Included if double-pumping is used by the module.
<parameter list>	no	Every parameter to the component is included as an input/output.
data_out	no	the return value of the component is passed out of the module.
busy	yes	Indicates if the component is processing data.
done	yes	Indicates the data_out is valid
start	yes	Signal to the component to start processing data.
stall	yes	Signal to the component to delay processing data.

Table 4.24: Modelsim Signals Generated By the Intel HLS Compiler

The FIR filter produces a sequence of outputs that resembles the accumulated sum of the coefficients with a unit input to the filter (a value equal to one). Using Modelsim, a measure of the delay from reset to the first data out, and a measure of the periodicity of the outputs can be realized. The first delay becomes a static constant that reflects the setup time of the circuit. The periodicity, in the case of the FIR filter, represents the maximum sampling rate of the circuit. For testing purposes, a 1 Mhz clock is used. For a comparison of the Intel HLS Compiler results, a similar FIR filter was designed using Verilog HDL. The verilog solution utilizes more advanced DSP techniques such as multiplier sharing and bit sizing to reduce hardware resource utilization. The code for the verilog solution is provided in Appendix D.

Figure 4.11: FIR_Filter Verilog HDL Results

Figure 4.12: Intel HLS Initial Result - No Optimization Specified

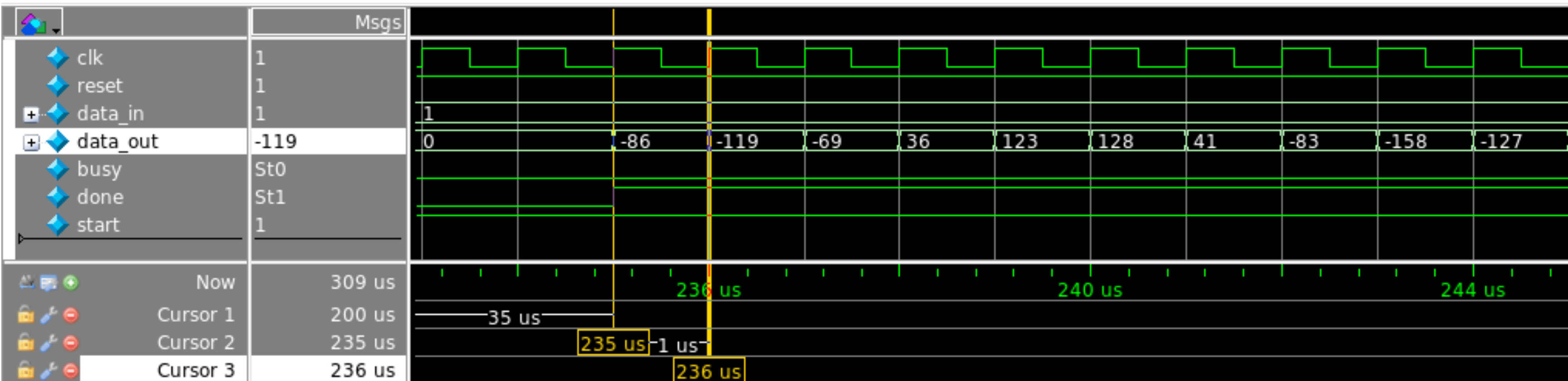


Figure 4.13: Intel HLS Initial Result - Fully Unrolled

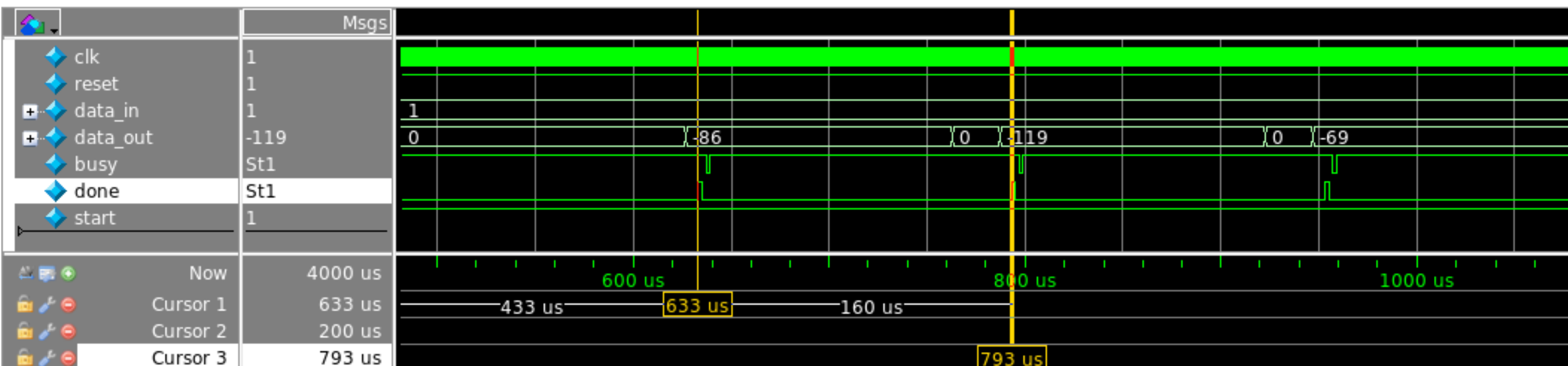


Figure 4.14: Intel HLS Initial Result - Partially Unrolled

Figure 4.11 shows the simulation results for the verilog solution. This solution shows a 79 clock cycle delay from the reset to the first valid data out of the circuit. The period of the output is equal to 8 clock cycles; this delay is caused by the multiplier sharing circuit. The outputs also show the correct accumulator values, but as the values progress a small amount of error is introduced into the system. The verilog solution uses strict bit sizing to barely allow for no overflows, but also creates small rounding errors in the design. This causes some minor discrepancies in the results from the actual versus the expected results. In traditional DSP design, it is the job of an experienced and trained DSP engineer to mathematically analyze the effects of these quantization errors to ensure correct functionality of the design.

Figure 4.12 shows the solution that the initial Intel HLS Compiler translation produces, which is used as the comparison for all future autotuner configurations. This solution uses all of the default optimization techniques that the tool performs automatically. No additional modifications were performed from the C source file to the RTL solution; no bit sizing was performed and no careful structuring was used to attempt to create an ideal/optimized solution. As expected, the testing showed that the output samples generated by the Intel HLS Compiler version of the filter matched the expected values perfectly. The solution is one that has not been unrolled at all; therefor, this solution will yield the highest operating clock frequency and the lowest amount of resources needed to implement the circuit (without further optimization such as bit sizing). It is noted, however, that the solution has a much lower *throughput* for the system. Specifically, 285 clock cycles elapse between valid samples of data out. This relates directly to the maximum sampling rate of the filter, and is not currently considered by the Intel HLS Compiler or the autotuner, as it requires a level of understanding of the functionality of the design.

Similarly, the same HLS solution can be fully unrolled. This produces a solution which has maximized the throughput of the system, and valid data comes out on every clock edge. From a performance perspective, this solution may be the best solution, but since the throughput was not taken into account in the autotuner, this solution will not appear to be optimal for the autotuner. The autotuner only observes the maximum clock frequency that the circuit can operate at; a metric that Quartus provides based on a timing analysis.

To be able to account for the design specific metrics (such as the maximum throughput of the system), a customized Modelsim simulation would need to be performed in conjunction with the procedure already being performed by the autotuner. This would add another level of complexity and time to the compilations. Once the Modelsim simulation is performed, a frequency measurement could be performed on the auto-generated 'done' signal. This signal identifies to the designer that the output has stabilized and is valid for processing.

An example of a partially unrolled solution is also provided in Figure 4.14. This solution uses the *#pragma unroll 10* optimization directive which gets the compiler to unroll the loop 10 times. This shows that partially unrolling the design does yield a solution in-between the two extremes of fully unrolling and not unrolling the loops.

To get a more accurate comparison, the throughput of the circuit should be considered. For comparison reasons, we can define the throughput of the circuit as:

$$Throughput = \frac{F_{Max}}{N_{cycles}} \tag{4.2}$$

$Throughput$: Maximum output of a circuit, millions of valid outputs per second

F_{Max} : Maximum operating clock frequency of the circuit, Mhz

N_{cycles} : Number of cycles between valid data outputs, clock cycles

	ALMs	FFs	RAMs	DSPs	F_{Max}	Throughput
Verilog HDL	2009	3471	0	65	136.22	17.0275
Initial HLS	312	690	2	2	225.38	0.7908
Unrolled HLS	11775	38873	1	60	193.42	193.42
Partially (10) Unrolled HLS	9725.5	16738	56	20	116.86	0.7304

Table 4.25: Summary of FIR Filter Synthesis Results

The throughput for an FIR filter is related to the maximum sampling rate of the circuit, but as the application of the component changes, so will the definition of what the throughput

represents. For example, an image processing component may have its throughput reflect the frame rate, or an encryption algorithm may reflect the number of encryptions per second. Not all examples will be truly periodic, and every component would require special attention and application into the autotuner. It is for this reason that the throughput of the component becomes a difficult metric to calculate. Automated analysis of the throughput for an arbitrary input design for inclusion in the autotuning process is left as a subject for future investigation.

4.6.2 Autotuning with Targets Introduced

Introducing targets allows the designer to specify more rigid requirements for a synthesis translation. For example, if a certain number of DSP blocks have been budgeted for a circuit, any solution that is created above that amount should be penalized. This will allow the autotuner to bias its search for configurations toward those that meet the target requirements.

From autotuner_config.json, targets can be introduced by specifying a resource value that the synthesis should attempts to reach. If the target value is not met, an applied penalty is given to the normalized worth of that configuration. This will cause the autotuner to bias towards solutions and search techniques that meet the target as per Equation 3.4 and 3.5.

The example below shows the user settings used to specify the target for the number of DSP blocks to be 3 for mips with a penalty factor of 2.0 for not meeting the target:

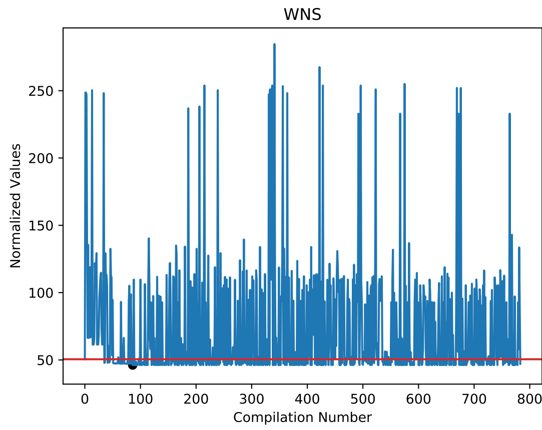
```
"target_settings":
{
    "_comment": "only choose one of value or percentage.",
    "fmax":0,
    "ALUTs":"inf",
    "FFs":"inf",
    "RAMs":"inf",
    "DSPs":3,
    "ALUTs_as_Percentage":100.0,
```

```
        "FFs_as_Percentage":100.0,
        "RAMs_as_Percentage":100.0,
        "DSPs_as_Percentage":100.0
    },
    "target_weights":
    {
        "target_penalty_factor":2.0,
    .
    .
    .
```

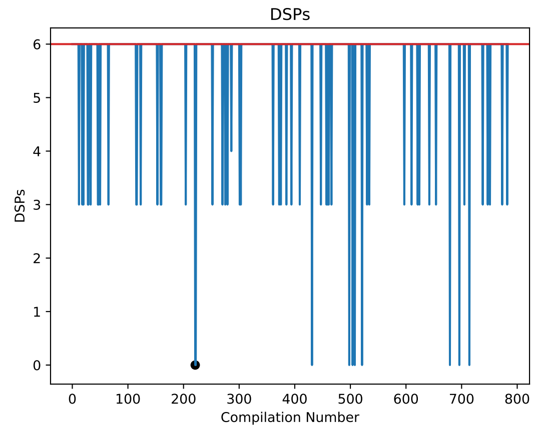
To see the impact of adding a target value, a comparison between setting different targets while keeping the same seed for each test and using the area weighting scheme (to emphasize resource usage over performance) was conducted. Figure 4.15 and Table 4.26 show the impact of adding a target on the number of DSP blocks.

	Initial WNS	Best WNS	ALMs	FFs	RAM Blocks	DSPs	F_{max}
Initial			2202	4606	15	6	178.41
No Target	34.0	29.6987	2122	4374	8	6	188.64
3 DSP Blocks Penalty Factor: 2.0	50.5	46.1987	2122	4374	8	6	188.64
3 DSP Blocks Penalty Factor: 4.0	101.03	88.8355	10897	23689	8	0	143.7
3 DSP Blocks Penalty Factor: 6.0	255.783	89.1643	11540	24412	2	0	160.64

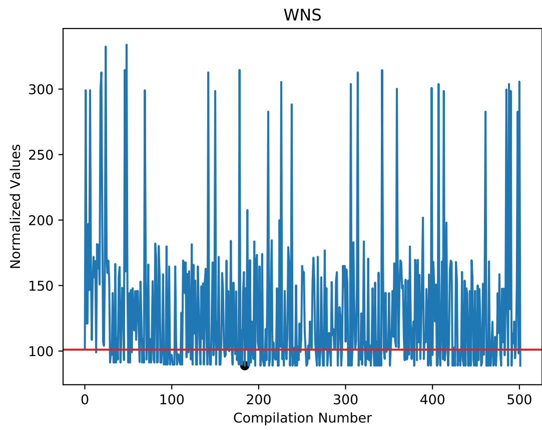
Table 4.26: Summary of mips with DSP Block Targets Specified, Same Seed Configuration, Varying Penalty Factor



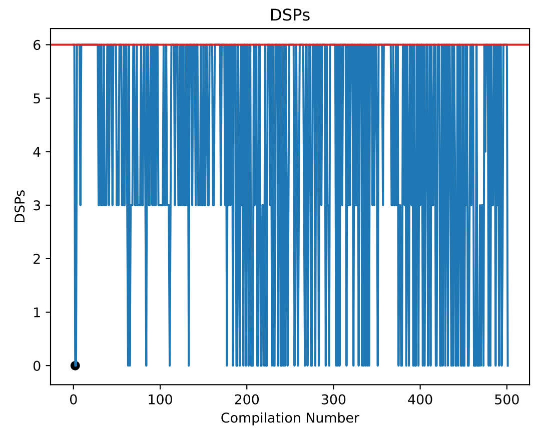
(a) Penalty Factor = 2.0



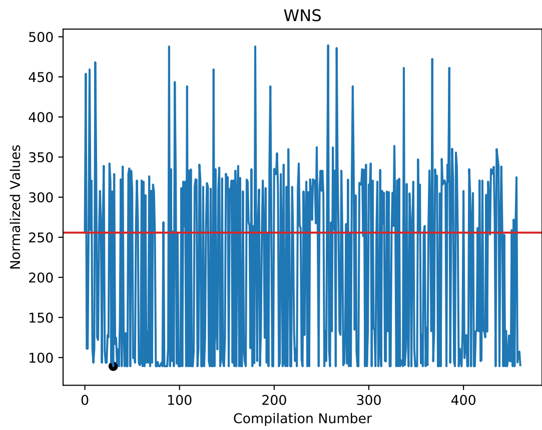
(b) Penalty Factor = 2.0



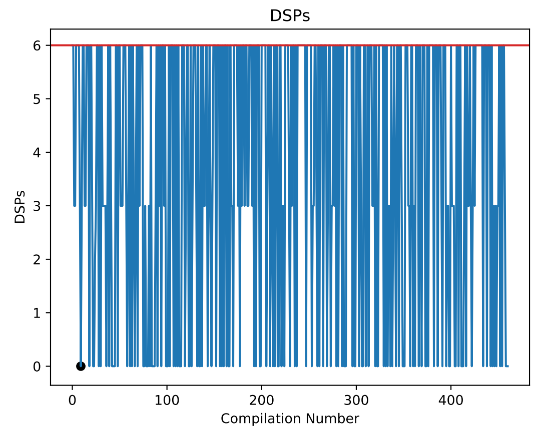
(c) Penalty Factor = 4.0



(d) Penalty Factor = 4.0



(e) Penalty Factor = 6.0



(f) Penalty Factor = 6.0

Figure 4.15: Mips with a Target of 3 DSP Blocks and Varying Penalty Factors

As the targets get more strict, the worse the initial synthesis results become. This is shown in Table 4.26's initial normalized results. Similarly, how *hard* the constraint is can be set by adjusting the penalty for not meeting the target. A low penalty factor would indicate a soft requirement, while a higher penalty factor would indicate a hard requirement. With a penalty factor of 2.0, the target of 3 DSP blocks is not considered optimal when the solution with 3 DSP blocks yields five times the other resources; however, as the penalty factor for the DSP target increases, the more the autotuner is likely to meet the target at the expense of the other resources. Figure 4.15 shows that increasing the penalty factor causes the autotuner to put more emphasis on reducing the number of DSP blocks in the final design, and as such more attempts at finding an optimal solution with reduced DSP blocks occurs. As desired, when the penalty factor is sufficient, the autotuner is able to identify designs which meet the target, as highlighted in Table 4.26.

Targets can be specified for a single resource, or multiple targets could be specified for single design. Each additional target will impose a penalty if not met as per Equations 3.5 and 3.6; although, the autotuners ability to find valid configurations that meet all of the targets becomes increasingly difficult as more restrictions apply to the design.

5. Summary and Conclusions

5.1 Summary

In this research, it has been shown that the OpenTuner Framework can be used to autotune the Intel HLS Compiler, a commercial HLS tool, for C to RTL translations. There are several major areas of optimization strategies than can be employed to a C source file that improve the translation, but this process currently involves a manual iterative procedure that can prove to be lengthy and requires specialized knowledge of FPGAs. Using the autotuner, two types of optimizations can be automatically applied to the source file through placeholders and autotuned to find parameters for better solutions. These areas are loop optimizations and local memory optimizations.

For some designs that use structures conducive to the optimizations being autotuned, improved FPGA hardware implementations can be realized through autotuning. This process works if the search space is well defined; otherwise, long compile times and complex optimizations can limit the usefulness because of poor exploration of large search spaces and unknown variable dependencies by the compiler. Solutions that are small components of a larger design are best to autotune because their time to compile and search space stays small enough for the autotuner to properly explore the search space to find the optimal solutions. Larger solutions are too taxing on the personal computer's hardware limitations unless server-grade computers or clusters are employed.

5.1.1 Benefits of Autotuning

Autotuning allows the designer to obtain improvements in the synthesis translation from C to RTL without making repetitive adjustments to the source code or having a deep knowl-

edge of the FPGA internals. The autotuned parameters are automatically injected into the source code, which assists the designer without manual intervention. The autotuner will automatically search for configurations that yield results that move towards a user specified weighting scheme. Several schemes can be built into the framework which allows a designer to set the focus of the autotuner to bias towards solutions that are optimized for area, performance, or a balance of both. This allows the designer to potentially create a better RTL synthesis of the C source code in an effortless manner.

In some situations the autotuner will not provide results that meet the requirements of a design; however, the autotuner results are stored into a database for post-processing. This allows for the designer to sift through the results in the database to search for a configuration that was not deemed optimal by the autotuner but may still be more useful for their design specifications than an existing solution. For example, there may be a linear relationship between the number of DSP blocks used in a design and the maximum throughput of the circuit, and the best solution for the design is one that has a balance of both. This is not ideal as it requires post-processing of the results, but can still provide valuable time savings over manually checking new configurations.

The autotuner gives three different levels of compilations: an estimate of the resources used, post-mapped results that give an estimate to the performance (maximum operation clock frequency) of the circuit, and post-fitted results. The post-fitted results have the highest quality of results and yields the final solution that would be fit onto the specific FPGA, but also takes the longest time to compile - a problem that limits the exploration of a large search space. The estimated and post-mapped results provide solutions that are a good estimate to the post-fitted results in finding a configuration that is close to optimal, and can more efficiently explore a large search space in a given amount of time.

To further the practical application of autotuning HLS tools, the specification of target requirements adds an additional feature that does not exist currently in HLS tools. The addition of the ability to specify design targets will allow the autotuner to find solutions that meet more specific design targets; the targets can be adjusted from soft targets to hard targets through some JSON formatted configuration files. The addition of targets may cause

a formerly optimal solution to no longer be the best solution for the specific requirements of a design. In any case, the autotuner will explore the search space to find a better solution or verify that no other optimal solutions exist (which is sometimes just as valuable to the designer). In the case of mips, a higher penalty factor is needed to convince the autotuner to reduce the number of DSP blocks in the design at the expense of higher ALMs and FFs; a trade-off that would normally not be considered by the Intel HLS Compiler or Quartus. This provides much more flexibility in specifying the requirements by the designer.

5.1.2 Limitations of Autotuning

There are several limitations imposed by the Intel HLS Tool in terms of coding style and structures used. Specifically, pointer arithmetic and complex exit conditions are difficult for the tool to synthesize, among other limitations regarding code structuring and dependency resolution. The addition of the autotuner and associated tools impose some further restrictions on the coding style; namely, support for C++ is dropped by using the python C parsing tool. This limitation could be reduced (or eliminated) through manual placement of the optimization placeholders or adding more support for source parsing tools.

Long compilation times proved to prevent the autotuner from exploring enough of the search space for optimal solutions in many cases. This causes the best identified configuration to be dependent on the initial seed. To assist with the long compile times, three levels of synthesis results are selectable by the autotuner framework which allows the user to trade off quality of estimates versus compilation time and computer hardware requirements. When a design is large enough to cause the computer use all of its RAM, compilations slow down greatly or cause compilation errors.

The CHStone Benchmarking Suite, which offers a variety of programs over different application domains, was used to test the autotuner. Four of these programs were unable to be autotuned due to coding limitations because of unsupported syntax or coding practices that do not adhere to Intel's *best practices guidelines*. Others, such as the four arithmetic programs, were autotunable but did not contain any loops or large local memory structures to be tuned and are thus not applicable to this method. This limited the number of appli-

cations to be tested to only four out of twelve CHStone applications. Of the four CHStone applications that were autotunable, three showed improvements that ranged from 4% to 40% depending on the optimization scheme and compilation used. Gsm and the DSP FIR filter did not show any improvements beyond the initial translation with default optimization strategies.

The applications that did not show improvement by using the autotuner could still benefit from autotuning if additional optimization goals were to be incorporated into the autotuner. For example, the maximum sampling rate of the FIR filter is established by the maximum throughput of the circuit and is an important metric of the performance of the filter; a metric that is not automatically generated by Intel HLS Compiler or by performing a Quartus compile. Without including this metric into the performance of the filter, the optimal solution will always appear to be the one with the fewest resources used and highest operating clock frequency; however, this solution may in some cases have the slowest throughput and thus the lowest maximum sampling rate of the filter. This could be improved upon by incorporating more information into the autotuner such as the latency and throughput of the synthesized design. Successful integration of the simulator into the autotuner could expand the qualitative results of the autotuner to include design-specific measurement attributes.

5.2 Thesis Contributions

This thesis has developed a method of using an autotuner, such as the OpenTuner Framework, to autotune a commercial HLS tool to improve the synthesis translation from C to RTL automatically. The research from this thesis helps identify problems and situations in which this process works well, and situations in which this process is still lacking.

The main contributions of this thesis are:

1. This thesis provides a method in which an autotuner can be used to autotune a commercial HLS tool for optimization of the tradeoff between resource usage and maximum operating clock frequency. The OpenTuner Framework can be used to autotune the Intel HLS Tool through automatic placement of placeholders for optimization param-

eters that are passed in through the i++ command. The OpenTuner Framework can perform the synthesis translation, store the results in a database, and adjust the configuration through a configuration manipulator and a pre-defined search technique.

2. This thesis verifies that the improvement of the HLS translation can be performed automatically without any HDL or FPGA knowledge. In the current implementation of the autotuner, the design must contain some loops or (large) local memory structures to be autotuned.

3. This thesis shows that smaller components of a design are the most conducive to autotune. Large designs are not conducive to autotuning because of computer hardware limitations that lead to long compile times. Long compile times prevent the autotuner from properly exploring a large search space.

4. This thesis verifies that the starting point for the autotuner is important for the final results of the configuration. As the source file grows, the search space will also grow exponentially. The autotuner will likely only be able to explore a small fraction of the search space and so the starting point of the search technique becomes important so the autotuner can hone in on optimal solutions.

5. This thesis introduces the concept of applying specific design targets, such as resource usage or performance metrics, to an autotuner to adjust the Intel HLS Compiler conversion process. It shows that an autotuner can be used to apply optimization parameters that adjust the synthesis results to meet the targets specified. For practical applications, the addition of specified resource or performance targets will yield different solutions than what the default Intel HLS Compiler translation would normally produce. Such solutions have great practical value, particularly when a component is to be included as part of a larger design and must meet a resource or performance budget. The targets can be specified as hard targets (must meet) or soft targets (should meet) by adjusting a penalty term which is applied to configurations not meeting the target requirement.

5.3 Future Work

There are several areas that have been identified for future investigation on the process of autotuning a commercial HLS tool automatically without any HDL or FPGA specific knowledge:

1. The addition of the simulator to the autotuner procedure would allow for better performance metrics to be measured. These performance metrics could be used to better select a configuration based on performance metrics that are more specific to a design's requirements, such as throughput of the component.

2. More tunable parameters could be introduced to the autotuner if some intelligent checking for functionality is performed to ensure that the parameters being tuned do not break functionality. For example, the data types of the variables in the C code could be autotuned.

3. An exploration into intelligently selecting a starting point could yield better autotuner results in fewer iterations. This would make the process more practical and more efficient.

4. Clustering computers together to solve one autotuner's task could yield a better exploration of the search space and better results from the autotuner. Communication between the cluster would need to be explored as well as proper database handling.

5. The OpenTuner Framework could be built upon to better tailor the needs of HLS translations. This includes adding proper configuration types for factored inputs so that search techniques can find optimal solutions faster, and incorporating design targets and associated search techniques. Preemptively determining if a configuration would yield poor results could be explored, and using the correlation between estimated results and post-fitted results could be developed.

A. DSP FIR Filter Example

```
#include "HLS/hls.h"
component int FIR_Filter(int data_in)
{
  static signed int data_out;
  static int data[130];
  static int index = 0;
  static const int coefficient [130] = {
  -86, -33, 50, 105, 87, 5, -87, -124, -75, 31, 120, 128, 45, -73, -143, -111, 3, 117, 146, 65,
   -70, -156, -121, 16, 155, 182, 57, -139, -257, -185, 58, 310, 372, 152, -239, -538, -495,
   -68, 505, 832, 619, -90, -885, -1212, -740, 358, 1430, 1715, 849, -801, -2247, -2433,
   -942, 1579, 3621, 3627, 1012, -3204, -6600, -6427, -1056, 8934, 20685, 30129, 33740,
   30129, 20685, 8934, -1056, -6427, -6600, -3204, 1012, 3627, 3621, 1579, -942, -2433,
  -2247, -801, 849, 1715, 1430, 358, -740, -1212, -885, -90, 619, 832, 505, -68, -495, -538,
  -239, 152, 372, 310, 58, -185, -257, -139, 57, 182, 155, 16, -121, -156, -70, 65, 146, 117,
   3, -111, -143, -73, 45, 128, 120, 31, -75, -124, -87, 5, 87, 105, 50, -33, -86};
  data_out = 0;
  index = 0;
  for(index = (129); index > 0; index--)
  {
    data[index] = data[index-1];
  }
  data[0] = data_in;
  for(index = (129); index >= 0; index--)
  {
    data_out += (data[index]) * ( coefficient [index]);
  }
  return data_out;
}
```

B. User Defined Configuration File

B.1 autotuner_config.json

```
"hls_advanced_config":
{
    "pragma_unroll": "True",
    "pragma_loop_coalesce": "True",
    "pragma_max_concurrency": "True",
    "pump_and_port": "True",
    "reg_mem":"True",
    "hls_max_concurrency": "False",
    "pragma_ivdep": "False",
    "floating_point_relaxed": "False",
    "floating_point_conversion": "False",
    "floating_point_precision": "Flase",
    "array_banks_width": "False"
},
"target_settings":
{
    "fmax":0,
    "ALUTs":"inf",
    "FFs":"inf",
    "RAMs":"inf",
    "DSPs":"inf",
```

```
    "ALUTs_as_Percentage":100.0,
    "FFs_as_Percentage":100.0,
    "RAMs_as_Percentage":100.0,
    "DSPs_as_Percentage":100.0
},
"target_weights":
{
    "target_penalty_factor":2.0,
    "area": {
        "fmax": 2,
        "ALUTs": 8,
        "FFs": 8,
        "RAMs": 8,
        "DSPs": 8
    },
    "balanced": {
        "fmax": 4,
        "ALUTs": 4,
        "FFs": 2,
        "RAMs": 2,
        "DSPs": 2
    },
    "performance": {
        "fmax": 8,
        "ALUTs": 2,
        "FFs": 4,
        "RAMs": 2,
        "DSPs": 2
    }
}
```

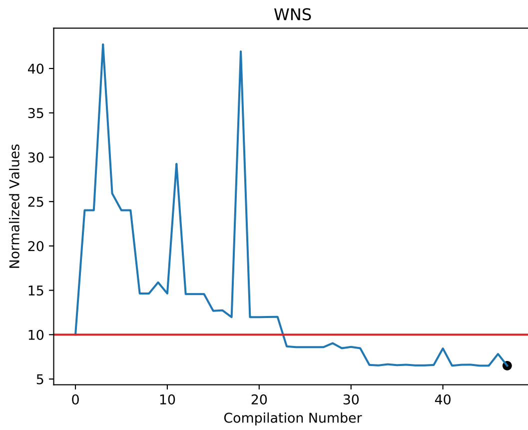
B.2 run.sh

```
#!/bin/bash

virtualenv sandbox
source sandbox/bin/activate
pip install pycparser
pip install enum
pip install sqlalchemy
pip install fn
pip install numpy
export LM_LICENSE_FILE=
export PATH=/opt/intelFPGA/18.1/modelsim_ase/bin: \
       /opt/intelFPGA/18.1/quartus/bin:$PATH
source /opt/intelFPGA/18.1/hls/init_hls.sh
export LD_PRELOAD=/usr/lib64/libtcmalloc.so

duration=864000 #in seconds
parallelism=1 #number of concurrent compiles
max_iterations=1000 #limit control
name="adpcm"
optimize_type="balanced" #balanced, area, or performance
#Only put either compute_fmax or quartus_compile true, not both
compute_fmax=1 #do a partial quartus compile (Mapper Only)
quartus_compile=0 #do a full quartus compile (Fitter included)
technique="AUCBanditMetaTechniqueA"
chip_type="Cyclone V" #Cyclone V, Stratix V, Arria10
hls_project_directory="sourcefiles" #relative paths (./) not supported
results_directory="results" #relative paths (./) not supported.
main_file_name="source.c" #the file that contains the component
```

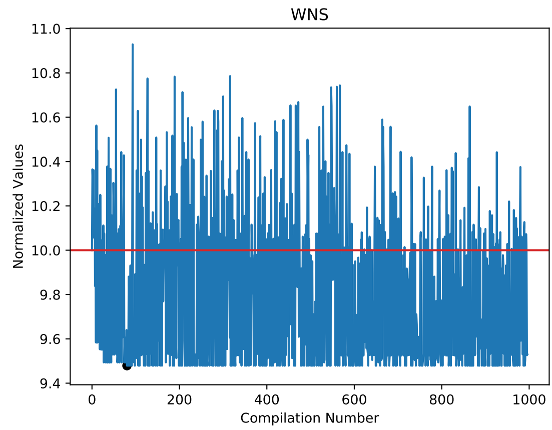
```
final_result="log_best.txt"
other_results="log_other.txt"
detailed_log_name="log_detailed.txt"
csv_log_name="csv_log_"
csv_log_name+=$name
csv_log_name+="_"
csv_log_name+=$optimize_type
csv_log_name+="_"
if [ $compute_fmax -eq 1 ]
then
  csv_log_name+="map"
elif [ $quartus_compile -eq 1 ]
then
  csv_log_name+="fit"
else
  csv_log_name+="est"
fi
csv_log_name+=".csv"
seed_file="seed.json" #NOTE: OpenTuner supports multiple seed files...
                      #NOTE: seed.json is the auto-generated name
generate_new_seed=1 #can be "y/n" "yes/no" "1/0" etc.
                    #Tells autotuner to create a new seed.json
auto_parameter_injection=1 #if not set, user must populate or provide
                           #results/new_sourcefile.c and
                           #intel_hls_parameters.json

#SEE ALSO: autotuner_config.json for more settings
```

```
python autotuner.py \
        --stop-after=$duration \
        --parallel-compile \
        --parallelism=$parallelism \
        --no-dups \
        --technique=$technique \
        --test-limit=$max_iterations \
        --results-log=$final_result \
        --results-log-details=$other_results \
        --hls-directory=$hls_project_directory \
        --results-directory=$results_directory \
        --seed-configuration="$seed_file" \
        --generate-new-seed=$generate_new_seed \
        --log-detailed=$detailed_log_name \
        --compute-fmax=$compute_fmax \
        --main-file-name=$main_file_name \
        --quartus-compile=$quartus_compile \
        --chip-type="$chip_type" \
        --optimize_type=$optimize_type \
        --log-csv="$csv_log_name" \
        --label="$csv_log_name" \
        --auto-parameter-injection=$auto_parameter_injection
```

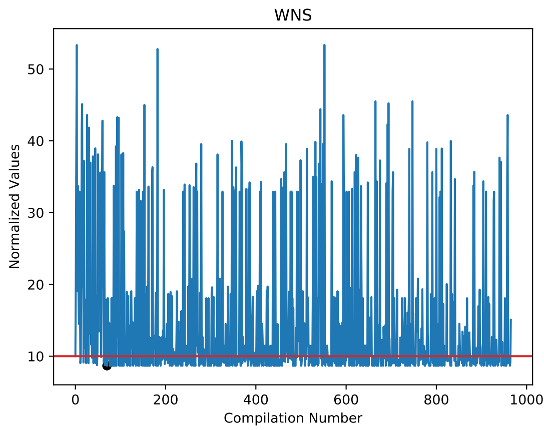
C. Post-Mapped and Post-Fitted CHStone Results



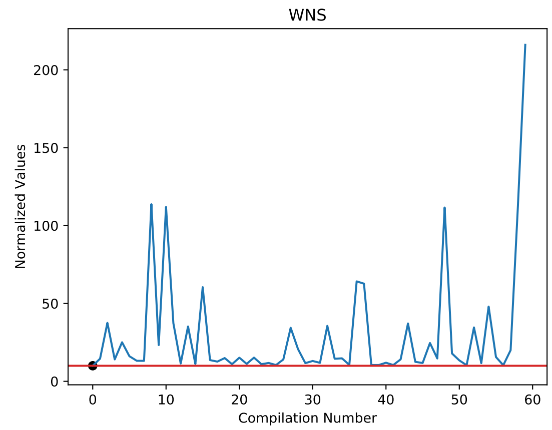
(a) adpcm



(b) aes

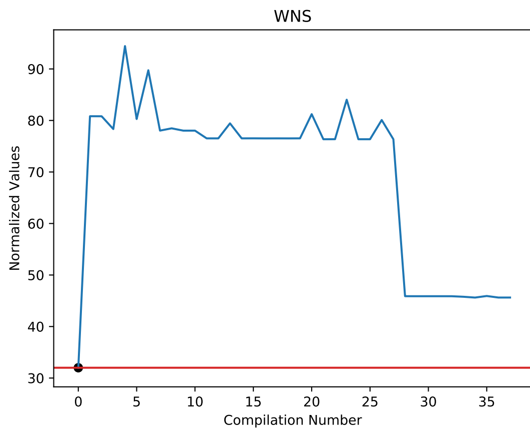


(c) mips

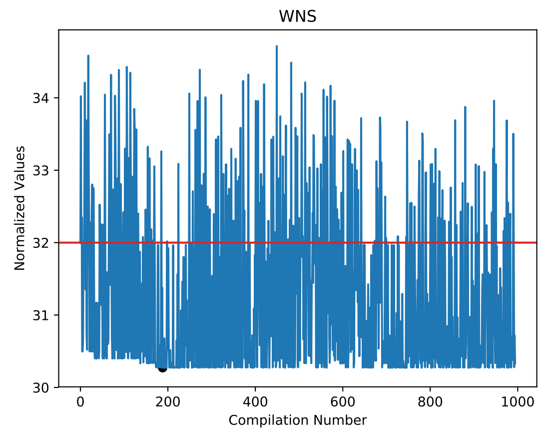


(d) gsm

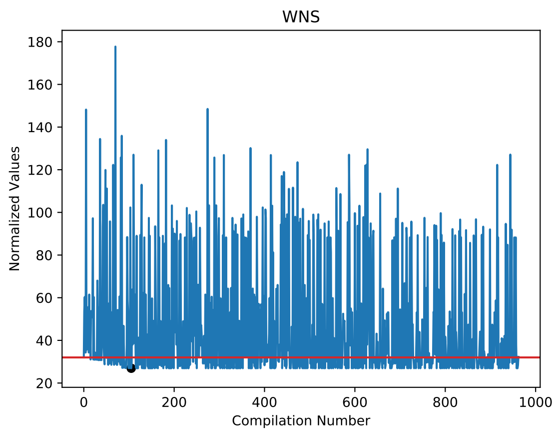
Figure C.1: WNS Estimated Balanced Results



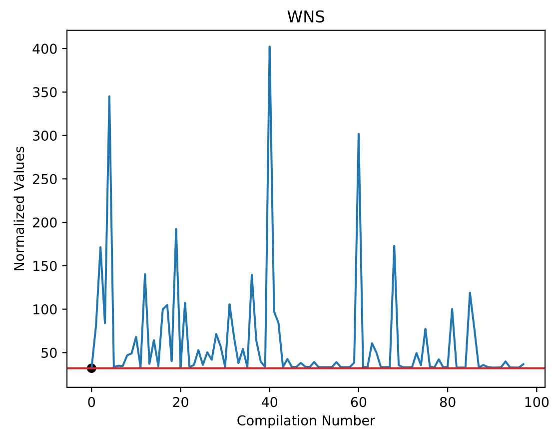
(a) adpcm



(b) aes

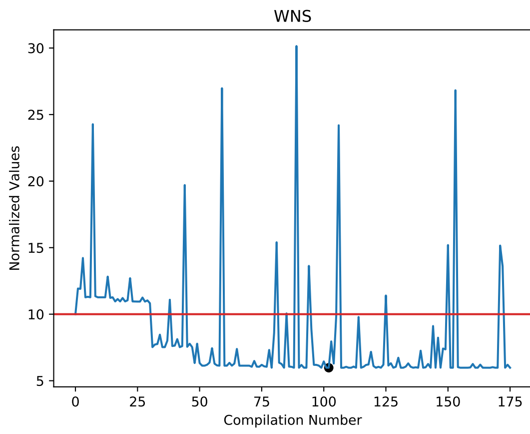


(c) mips

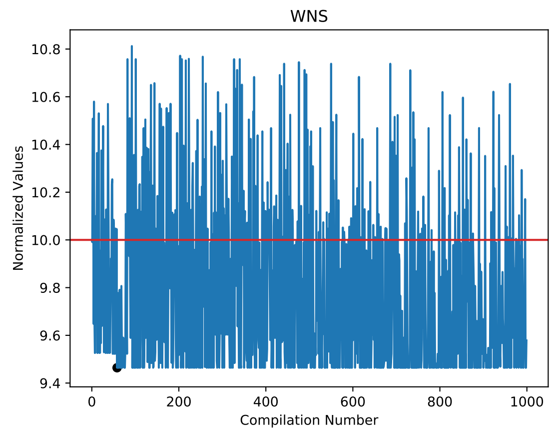


(d) gsm

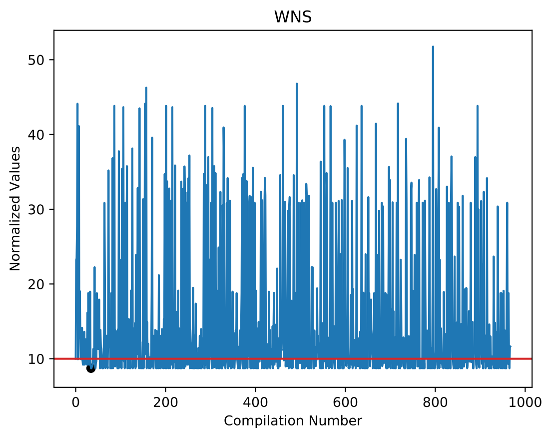
Figure C.2: WNS Estimated Area Results



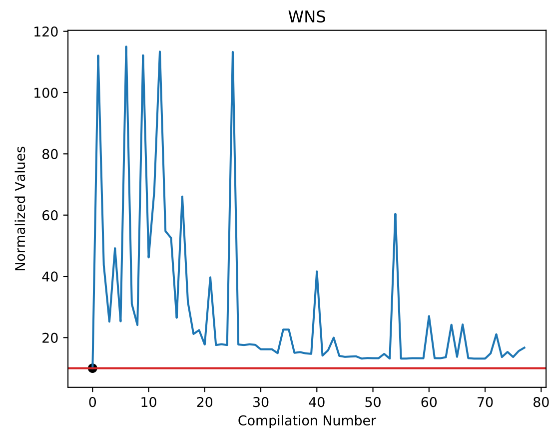
(a) adpcm



(b) aes

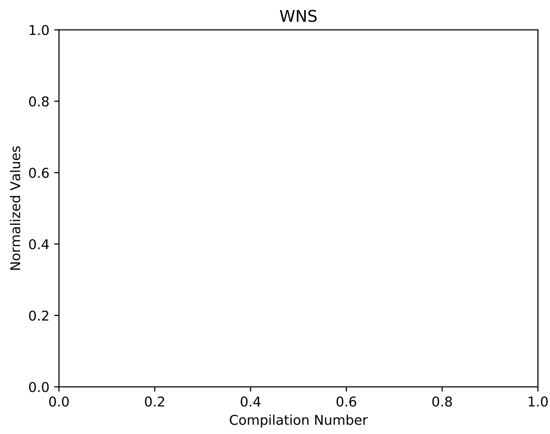


(c) mips

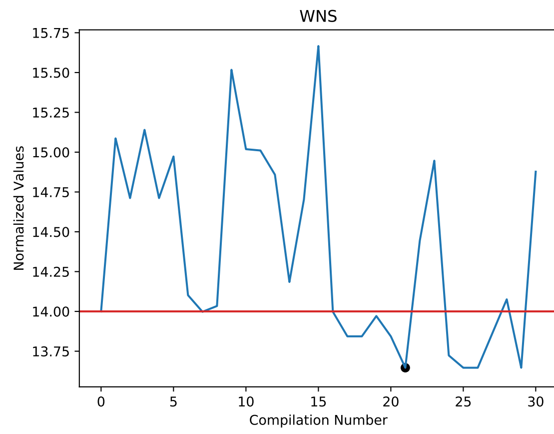


(d) gsm

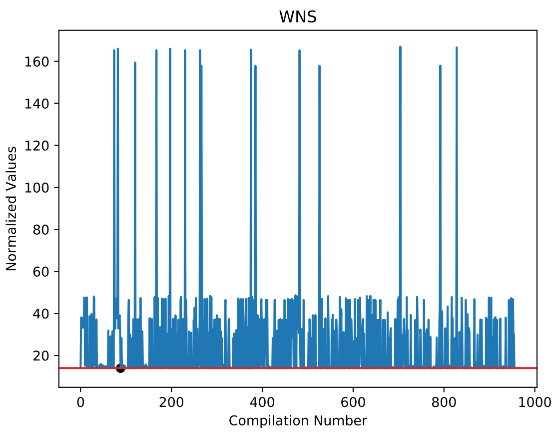
Figure C.3: WNS Estimated Performance Results



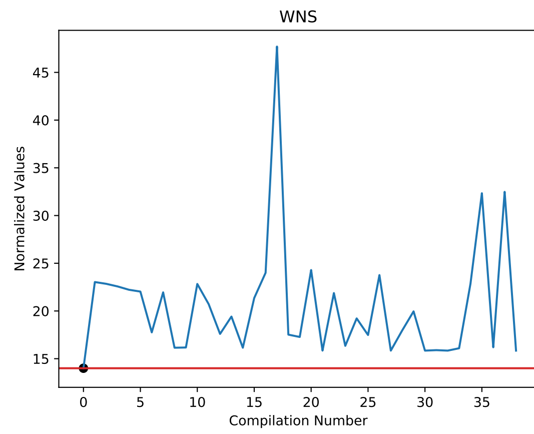
(a) adpcm



(b) aes

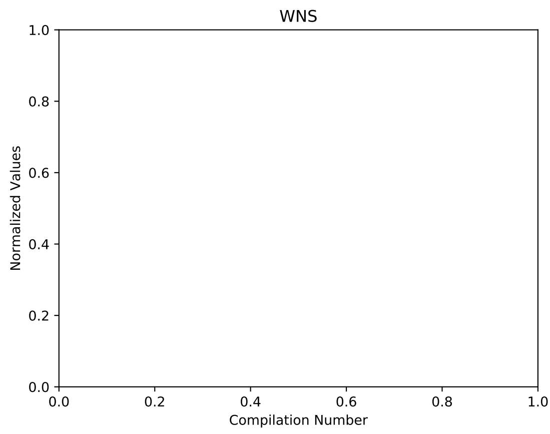


(c) mips

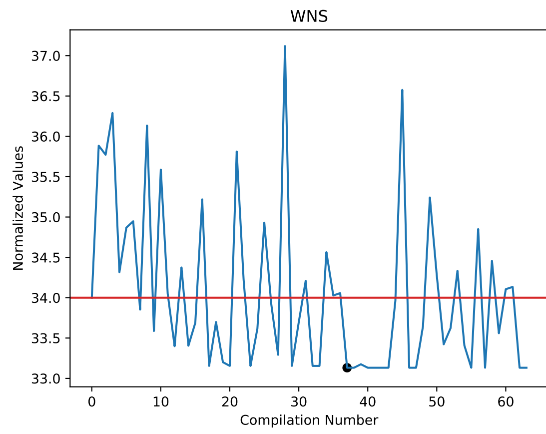


(d) gsm

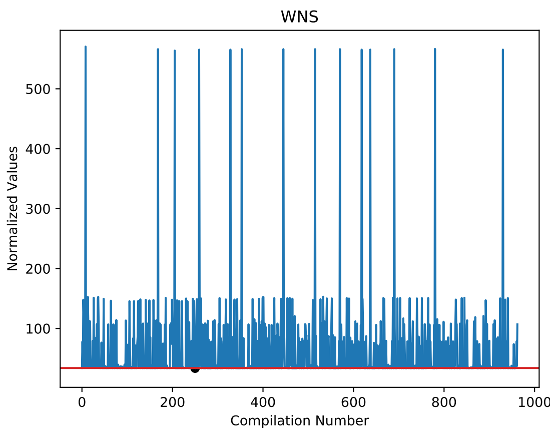
Figure C.4: WNS Post-Mapped Balanced Results



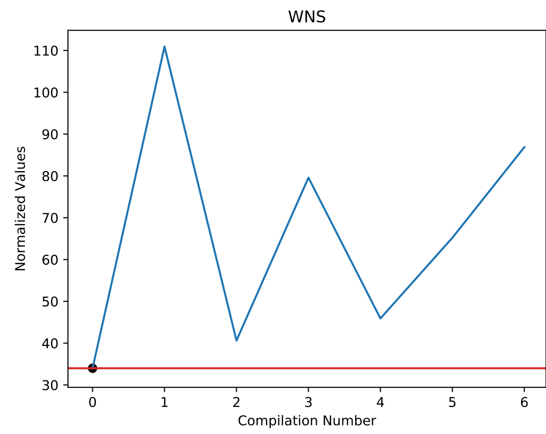
(a) adpcm



(b) aes

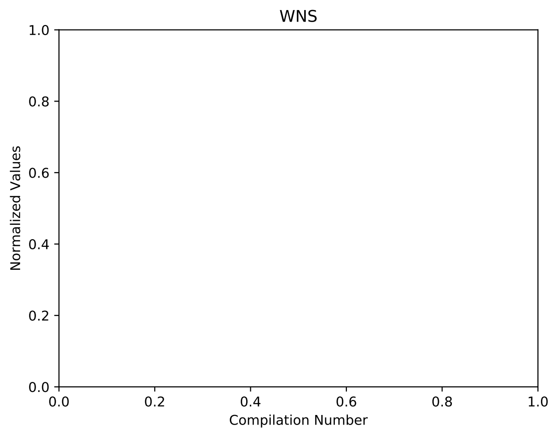


(c) mips

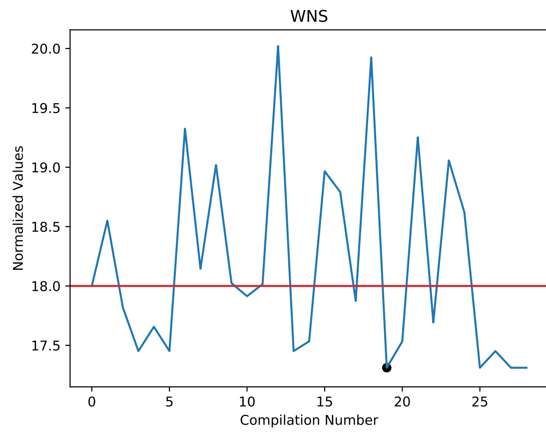


(d) gsm

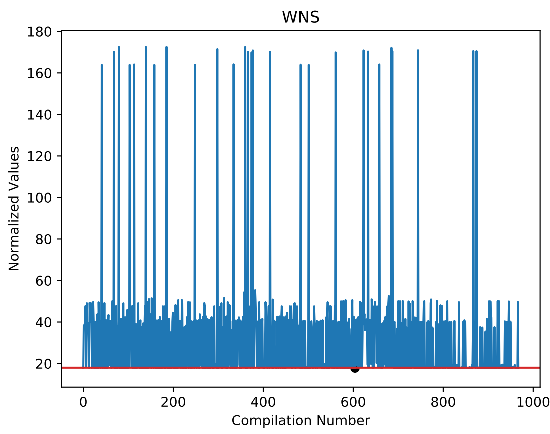
Figure C.5: WNS Post-Mapped Area Results



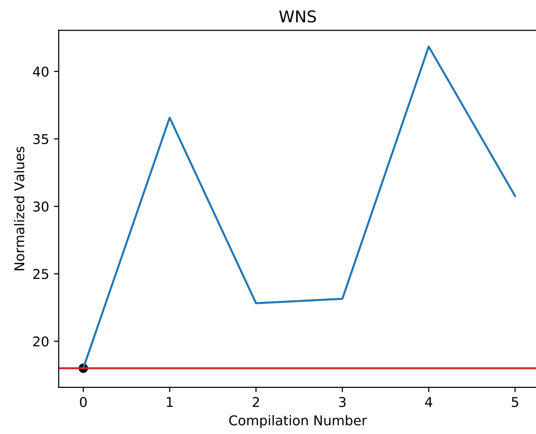
(a) adpcm



(b) aes

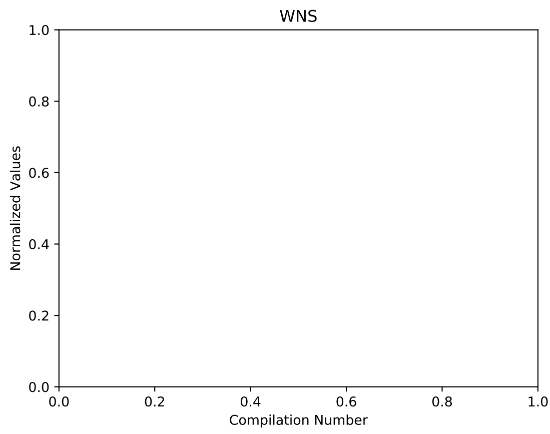


(c) mips

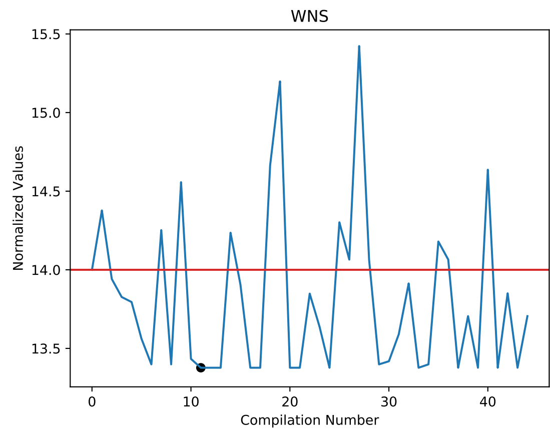


(d) gsm

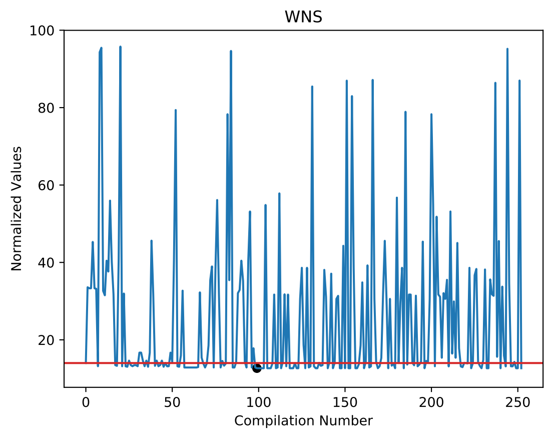
Figure C.6: WNS Post-Mapped Performance Results



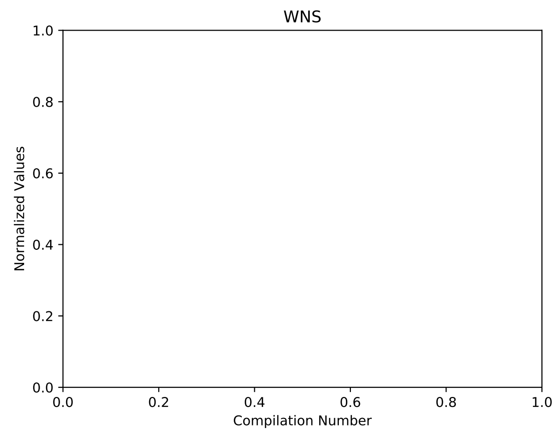
(a) adpcm



(b) aes

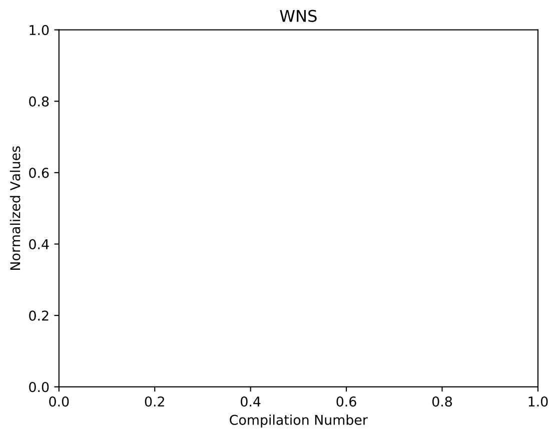


(c) mips

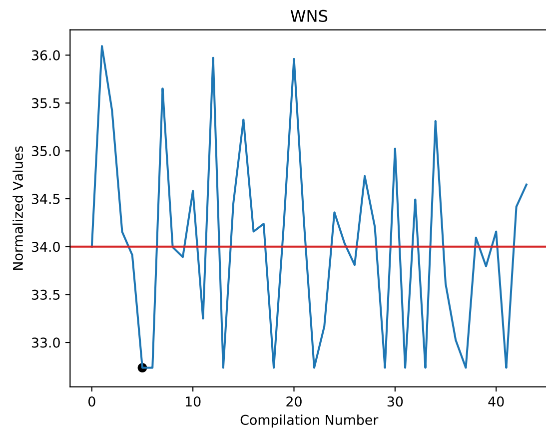


(d) gsm

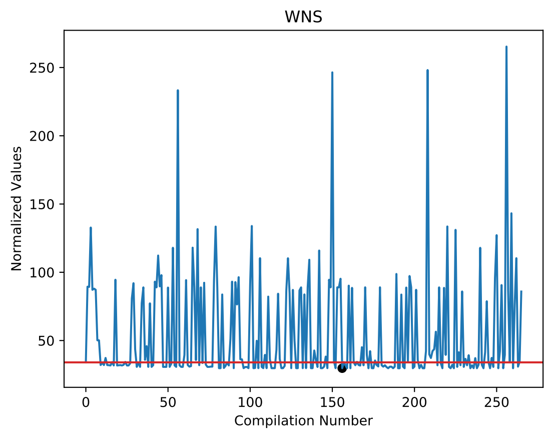
Figure C.7: WNS Post-Fitted Balanced Results



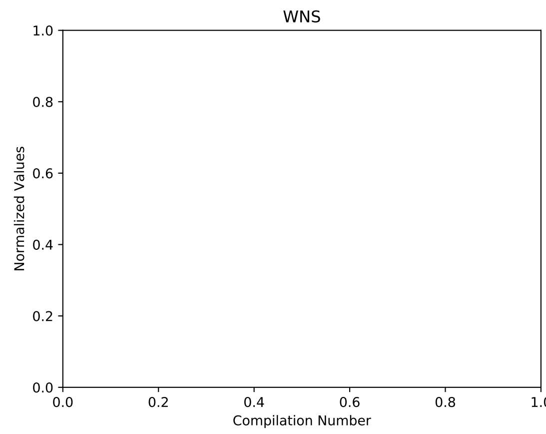
(a) adpcm



(b) aes

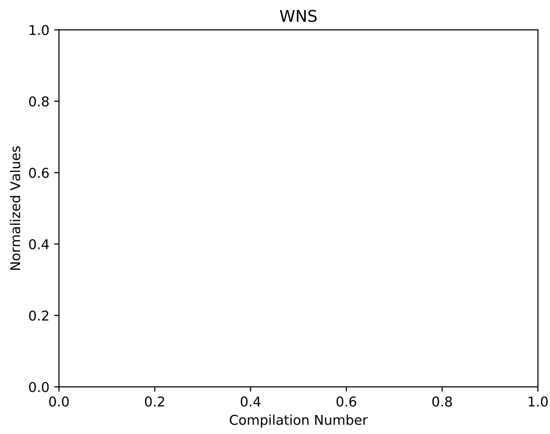


(c) mips

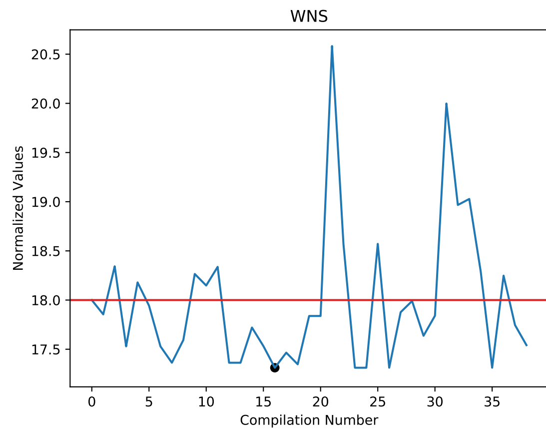


(d) gsm

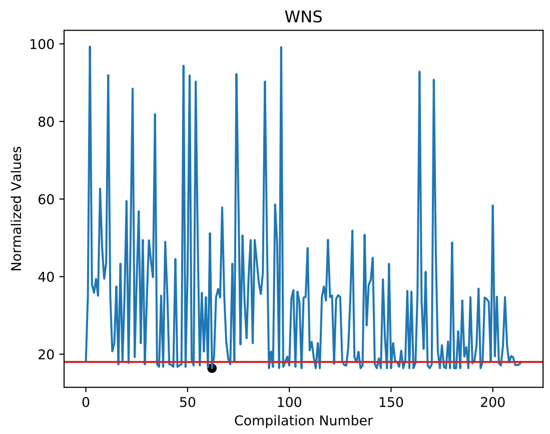
Figure C.8: WNS Post-Fitted Area Results



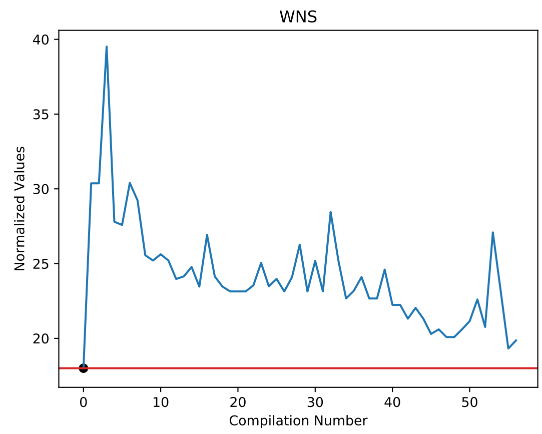
(a) adpcm



(b) aes



(c) mips



(d) gsm

Figure C.9: WNS Post-Fitted Performance Results

D. Verilog Code for FIR Filter

Code © Ethan Paramor, 2019. Used with permission.

```
module FIR_Filter (
  input fast_clk,
  input sys_clk,
  input clk_ena,
  input reset,
  input [2:0] RX_sync,
  input signed [17:0] x_in,
  output reg signed [17:0] y
  );

integer i;
reg [2:0] mux_counter;

reg signed [17:0] x [128:0]; // [1s17] Delay registers as 129x 18-bit arrays
reg signed [17:0] b [64:0]; // [1s17] multiplier coefficients , as 65x 18-bit arrays

reg signed [17:0] mult_data_in [8:0]; // [2s16] Data in (from sum_level_1), post multiplexer
                                      // for fast multiplier , as 9x 18-bit packed arrays
reg signed [17:0] mult_coef_in [8:0]; // [1s17] Coefficient in, post multiplexer for fast
                                      // multiplier , as 9x 18-bit packed arrays
reg signed [35:0] mult_result [8:0];  // [3s33] multiplier output (untrimmed), as 9x 36-bit
                                      // packed arrays
reg signed [17:0] mult_trim [8:0]; // [1s17] trim multiplier output, as 9x 18-bit packed arrays
reg signed [17:0] mult_out_fast [64:0]; // [1s17] multiplier outputs (trimmed, clocked at
                                        // fast_clk ), as 65x 36-bit packed arrays
reg signed [17:0] mult_out_pipe [64:0]; // [1s17] multiplier outputs (trimmer, pipelined from
```

```
                                            // fast_clk  registers ),  as 65x 36-bit packed arrays
reg signed [35:0] mult_out_noShare[64:0]; // [1s17] Non-shared multiplier setup,
                                            // 65x 18-bit packed arrays

reg signed [17:0] sum_level_1 [64:0];  //[1s17] 1st level of adders, as 65x 18-bit packed arrays
reg signed [17:0] sum_level_2 [32:0];  //[1s17] 2nd level of adders, as 33x 18-bit packed arrays
reg signed [17:0] sum_level_3 [16:0];  //[1s17] 3rd level of adders, as 17x 18-bit packed arrays
reg signed [17:0] sum_level_4 [8:0];  //[1s17] 4th level of adders, as 9x 18-bit packed arrays
reg signed [17:0] sum_level_5 [4:0];  //[1s17] 5th level of adders, as 5x 18-bit packed arrays
reg signed [17:0] sum_level_6 [2:0];  //[1s17] 6th level of adders, as 3x 18-bit packed arrays
reg signed [17:0] sum_level_7 [1:0];  //[1s17] 7th level of adders, as 2x 18-bit packed arrays
reg signed [17:0] sum_level_8;  //[1s17] 8th level of adders, as 1x 18-bit packed array

always @ (posedge fast_clk)
  if (reset == 1'b1)
    mux_counter <= 3'd0;
  else
    mux_counter <= mux_counter + 3'd1;

// Initial signal input
always @ (posedge sys_clk)
  if (reset == 1'b1)
    x[0] <= 18'sd0;
  else if (clk_ena == 1'b1)
    x[0] <= {x_in[17], x_in [17:1]};  // [2s16] Sign extend input

// Set up delay registers
//129 delay registers total, take previous value on next clock cycle
always @ (posedge sys_clk)
  if (reset == 1'b1)
    begin
      for(i=1; i<129; i=i+1)
        x[i] <= 18'sd0;
    end
  else if (clk_ena == 1'b1)
    begin
```

```
        for(i=1; i<129; i=i+1)
          x[i] <= x[i-1]; // [2s16]
      end

// First level of Adders (65)
/*
  Add mirrored x values, according to symmetry
*/
always @ (posedge sys_clk)
  if (reset == 1'b1)
    begin
      for (i=0; i<64; i=i+1)
        sum_level_1[i] = 18'sd0;
    end
  else if (clk_ena == 1'b1)
    begin
      for (i=0; i<64; i=i+1)
        sum_level_1[i] = x[i] + x[128-i]; // [2s16]
    end

// Pipeline x[64] to maintain timing with sum_level_1
always @ (posedge sys_clk)
  if (reset == 1'b1)
    sum_level_1[64] = 18'sd0;
  else if (clk_ena == 1'b1)
    sum_level_1[64] = x[64]; // [1s17]

always @ *
  for(i=0; i<=64; i=i+1)
    mult_out_noShare[i] <= sum_level_1[i] * b[i];

// Second Level of Adders (33)
always @ (posedge sys_clk)
  if (reset == 1'b1)
        begin
                for(i=0; i<32; i=i+1)
```

```
                        sum_level_2[i] <= 18'sd0; // [1s17]
        end
        else if (clk_ena == 1'b1)
        begin
  for(i=0; i<32; i=i+1)
     sum_level_2[i] <= mult_out_noShare[2*i][33:16] + mult_out_noShare[2*i+1][33:16];//[1s17]
    end

// Pipeline sum_level_1[64] to maintain timing with sum_level_2
always @ (posedge sys_clk)
  if (reset == 1'b1)
        sum_level_2[32] <= 18'sd0;
  else if (clk_ena == 1'b1)
        sum_level_2[32] <= mult_out_noShare[64][33:16]; // [1s17]

// Third Level of Adders (17)
always @ (posedge sys_clk)
  if (reset == 1'b1)
        begin
                for(i=0; i<16; i=i+1)
                        sum_level_3[i] <= 18'sd0; // [1s17]
        end
        else if (clk_ena == 1'b1)
                begin
                  for(i=0; i<16; i=i+1)
        sum_level_3[i] <= sum_level_2[2*i] + sum_level_2[2*i+1]; // [1s17]
    end

    // Pipeline sum_level_2[32] to maintain timing with sum_level_3
    always @ (posedge sys_clk)
      if (reset == 1'b1)
            sum_level_3[16] <= 18'sd0;
      else if (clk_ena == 1'b1)
            sum_level_3[16] <= sum_level_2[32]; // [1s17]

// Fourth Level of Adders (9)
```

```
always @ (posedge sys_clk)
  if (reset == 1'b1)
       begin
              for(i=0; i<8; i=i+1)
                     sum_level_4[i] <= 18'sd0; // [1s17]
       end
       else if (clk_ena == 1'b1)
              begin
                for(i=0; i<8; i=i+1)
       sum_level_4[i] <= sum_level_3[2*i] + sum_level_3[2*i+1]; // [1s17]
    end

    // Pipeline sum_level_3[16] to maintain timing with sum_level_4
    always @ (posedge sys_clk)
      if (reset == 1'b1)
       sum_level_4[8] <= 18'sd0;
      else if (clk_ena == 1'b1)
       sum_level_4[8] <= sum_level_3[16]; // [1s17]

// Fifth Level of Adders (5)
always @ (posedge sys_clk)
  if (reset == 1'b1)
       begin
              for(i=0; i<4; i=i+1)
                     sum_level_5[i] <= 18'sd0; // [1s17]
       end
       else if (clk_ena == 1'b1)
              begin
                for(i=0; i<4; i=i+1)
       sum_level_5[i] <= sum_level_4[2*i] + sum_level_4[2*i+1]; // [1s17]
    end

  // Pipeline sum_level_4[8] to maintain timing with sum_level_5
  always @ (posedge sys_clk)
    if (reset == 1'b1)
       sum_level_5[4] <= 18'sd0;
```

```
    else if (clk_ena == 1'b1)
        sum_level_5[4] <= sum_level_4[8]; // [1s17]

// Sixth Level of Adders (3)
always @ (posedge sys_clk)
  if (reset == 1'b1)
        begin
                for(i=0; i<2; i=i+1)
                        sum_level_6[i] <= 18'sd0; // [1s17]
        end
        else if (clk_ena == 1'b1)
                begin
                  for(i=0; i<2; i=i+1)
        sum_level_6[i] <= sum_level_5[2*i] + sum_level_5[2*i+1]; // [1s17]
    end

  // Pipeline sum_level_5[4] to maintain timing with sum_level_6
  always @ (posedge sys_clk)
    if (reset == 1'b1)
        sum_level_6[2] <= 18'sd0;
    else if (clk_ena == 1'b1)
        sum_level_6[2] <= sum_level_5[4]; // [1s17]

// Seventh Level of Adders (2)
always @ (posedge sys_clk)
        if (reset == 1'b1)
                sum_level_7[0] <= 18'sd0; // [1s17]
        else if (clk_ena == 1'b1)
                sum_level_7[0] <= sum_level_6[0] + sum_level_6[1]; // [1s17]

    //Pipelining
    always @ (posedge sys_clk)
      if (reset == 1'b1)
        sum_level_7[1] <= 18'sd0; // [1s17]
      else if (clk_ena == 1'b1)
        sum_level_7[1] <= sum_level_6[2]; // [1s17]
```

```
// Eighth (Final) Level of Adders (1)
always @ (posedge sys_clk)
  if (reset == 1'b1)
    sum_level_8 <= 18'sd0;
  else if (clk_ena == 1'b1)
    sum_level_8 <= sum_level_7[0] + sum_level_7[1];

//Pipeline Final Output
always @ (posedge sys_clk)
        if (reset == 1'b1)
                y <= 18'sd0;
        else if (clk_ena == 1'b1)
                y <= sum_level_8;

/*************************************************************/
//                         Coefficients
/*************************************************************/
// Beta = 0.14, Theoretical MER of 43 dB

always @ (negedge reset)
begin
b[0] <= -18'sd86;
b[1] <= -18'sd33;
b[2] <= 18'sd50;
b[3] <= 18'sd105;
b[4] <= 18'sd87;
b[5] <= 18'sd5;
b[6] <= -18'sd87;
b[7] <= -18'sd124;
b[8] <= -18'sd75;
b[9] <= 18'sd31;
b[10] <= 18'sd120;
b[11] <= 18'sd128;
b[12] <= 18'sd45;
b[13] <= -18'sd73;
```

b[14] <= −18’sd143;

b[15] <= −18’sd111;

b[16] <= 18’sd3;

b[17] <= 18’sd117;

b[18] <= 18’sd146;

b[19] <= 18’sd65;

b[20] <= −18’sd70;

b[21] <= −18’sd156;

b[22] <= −18’sd121;

b[23] <= 18’sd16;

b[24] <= 18’sd155;

b[25] <= 18’sd182;

b[26] <= 18’sd57;

b[27] <= −18’sd139;

b[28] <= −18’sd257;

b[29] <= −18’sd185;

b[30] <= 18’sd58;

b[31] <= 18’sd310;

b[32] <= 18’sd372;

b[33] <= 18’sd152;

b[34] <= −18’sd239;

b[35] <= −18’sd538;

b[36] <= −18’sd495;

b[37] <= −18’sd68;

b[38] <= 18’sd505;

b[39] <= 18’sd832;

b[40] <= 18’sd619;

b[41] <= −18’sd90;

b[42] <= −18’sd885;

b[43] <= −18’sd1212;

b[44] <= −18’sd740;

b[45] <= 18’sd358;

b[46] <= 18’sd1430;

b[47] <= 18’sd1715;

b[48] <= 18’sd849;

b[49] <= −18’sd801;

```
b[50] <= -18'sd2247;
b[51] <= -18'sd2433;
b[52] <= -18'sd942;
b[53] <= 18'sd1579;
b[54] <= 18'sd3621;
b[55] <= 18'sd3627;
b[56] <= 18'sd1012;
b[57] <= -18'sd3204;
b[58] <= -18'sd6600;
b[59] <= -18'sd6427;
b[60] <= -18'sd1056;
b[61] <= 18'sd8934;
b[62] <= 18'sd20685;
b[63] <= 18'sd30129;
b[64] <= 18'sd33740;
end
endmodule
```

References

[1] "Intel High Level Synthesis Compiler." https://www.intel.com/content/www/us/en/programmable/documentation/nml1505158467345.html, April 2019 (accessed 10-June-2019). UG-20107.

[2] A. Pandit, "Introduction to FPGA and It's Programming Tools." https://circuitdigest.com/tutorial/what-is-fpga-introduction-and-programming-tools, 2019, (accessed 10-June-2019).

[3] M. Santarini, "Vivado, Inside the New Xilinx Design Suite." https://www.techdesignforums.com/practice/technique/vivado-xilinx-overview/, 2012 (accessed 10-June-2019).

[4] "Intel High Level Synthesis Compiler." https://www.intel.com/content/www/us/en/programmable/documentation/ewa1462824960255.html, April 2019 (accessed 10-June-2019). UG-20037.

[5] "Intel HLS Compiler Reference Manual." https://www.intel.com/content/www/us/en/programmable/documentation/ewa1462824960255.html, June 2019 (accessed 10-June-2019). MNL-1083.

[6] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. Amarasinghe, "OpenTuner: An Extensible Framework for Program Autotuning," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 303–315, Aug 2014.

[7] J. Ansel, "Autotuning Programs with Algorithmic Choice," *MIT - CSAIL*, 2014. Presentation Slides.

[8] "Cyclone V Device Overview." `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_51001.pdf`, May 2018 (accessed 10-June-2019). CV-51001.

[9] "Introduction to the Quartus II Software." `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/intro_to_quartus2.pdf`, 2010 (accessed 10-June-2019). MNL-01055-1.0.

[10] S. M. Trimberger, "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology," *Proceedings of the IEEE*, vol. 103, pp. 318–331, March 2015.

[11] H. D. Foster, "FPGA Verification Challenges and Opportunities." `https://verificationacademy.com/verification-horizons/november-2018-volume-14-issue-3/fpga-verification-challenges-and-opportunities`, 2018.

[12] H. Foster, "2018 FPGA Functional Verification Trends," pp. 40–45, 12 2018.

[13] C. Spear and G. Tumbush, *SystemVerilog for Verification*. Springer, 2012.

[14] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design Test of Computers*, vol. 26, pp. 18–25, July 2009.

[15] M. Dossis, "High-Level Synthesis: A Practical Perspective," *Advances in Robotics and Automation*, vol. 3, no. 3, pp. 1–3, 2014.

[16] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, S. Brown, and J. Anderson, "The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs," in *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 89–96, April 2013.

[17] Y. Uguen, F. de Dinechin, and S. Derrien, "A High-Level Synthesis Approach Optimizing Accumulations in Floating-Point Programs Using Custom Formats and Operators,"

in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 80–80, April 2017.

[18] E. Homsirikamol and K. G. George, "Toward a New HLS-Based Methodology for FPGA Benchmarking of Candidates in Cryptographic Competitions: The CAESAR Contest Case Study," in *2017 International Conference on Field Programmable Technology (ICFPT)*, pp. 120–127, Dec 2017.

[19] J. Johnson, "List and Comparison of FPGA Companies." `http://www.fpgadeveloper.com/2011/07/list-and-comparison-of-fpga-companies.html`, 2011, (accessed 10-June-2019).

[20] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. Brown, and J. Anderson, "LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, 09 2013.

[21] "Intel HLS Compiler Product Brief." `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/products/hls/hls-production-brief.pdf`, 2019 (accessed 10-June-2019).

[22] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, pp. 1591–1604, Oct 2016.

[23] "LegUp Documentation, Release 4.0." `http://legup.eecg.utoronto.ca/docs/4.0/legup-4.0-doc.pdf`, October 2015 (accessed 10-June-2019).

[24] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, S. Brown, and J. Anderson, "The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs," in *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 89–96, April 2013.

[25] Huan Li and Wenhua Ye, "Efficient Implementation of FPGA Based on Vivado High Level Synthesis," in *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*, pp. 2810–2813, Oct 2016.

[26] P. Bruel, A. Goldman, S. R. Chalamalasetti, and D. Milojicic, "Autotuning high-level synthesis for fpgas using opentuner and legup," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–6, Dec 2017.

[27] M. Belwal and T. S. B. Sudarshan, "Source-to-Source Translation: Impact on the Performance of High Level Synthesis," in *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pp. 951–956, May 2017.

[28] C. Kuan, J. Li, C. Chen, and J. K. Lee, "C++ Compiler Supports for Embedded Multicore DSP Systems," in *2011 40th International Conference on Parallel Processing Workshops*, pp. 214–221, Sep. 2011.

[29] E. Bendersky, "PyCParser." `https://pypi.org/project/pycparser/`, (accessed 10-June-2019).

[30] "Docker." `https://www.docker.com/resources/what-container`, (accessed 10-June-2019).

[31] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

[32] P. White, "Advanced QRD Optimization with Intel HLS Compiler," *Intel Corp.*, 2017. WP-01277-1.0.

[33] P. R. Panda, N. Sharma, S. Kurra, K. A. Bhartia, and N. K. Singh, "Exploration of Loop Unroll Factors in High Level Synthesis," in *2018 31st International Conference on VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID)*, pp. 465–466, Jan 2018.

[34] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii, "CHStone: A Benchmark Program Suite for Practical C-Based High-Level Synthesis,"

in *2008 IEEE International Symposium on Circuits and Systems*, pp. 1192–1195, May 2008.

[35] A. Dubey, A. Mishra, and S. Bhutada, "Comparative Study of CHStone Benchmarks on Xilinx Vivado High Level Synthesis Tool," 2015.