Automated Fixed-Point Analysis and Bit Width Selection in Digital Signal Processing Circuits Using Ptolemy

Derrick S. Gibelyou
Brigham Young University - Provo

Follow this and additional works at: https://scholarsarchive.byu.edu/etd
Part of the Electrical and Computer Engineering Commons

BYU ScholarsArchive Citation
https://scholarsarchive.byu.edu/etd/2757

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.
Automated Fixed-Point Analysis and Bit Width Selection in Digital Signal Processing Circuits using Ptolemy

Derrick S. Gibelyou

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

Michael J. Wirthlin, Chair
Brent E. Nelson
Michael D. Rice

Department of Electrical and Computer Engineering Brigham Young University
August 2011

Copyright © 2011 Derrick S. Gibelyou
All Rights Reserved
ABSTRACT

Automated Fixed-Point Analysis and Bit Width Selection in Digital Signal Processing Circuits using Ptolemy

Derrick S. Gibelyou
Department of Electrical and Computer Engineering
Master of Science

When designing custom hardware to implement signal processing algorithms, it is important to select bitwidths that meet the minimum error requirements while minimizing implementation area. Larger bitwidths reduce error, but increase area, while selecting smaller bitwidths does the opposite.

Finding the set of bitwidths that produces the smallest area that still meets the error requirements has been shown to be NP-hard. To address this problem, many heuristics have been developed. Unfortunately, they are not always well documented and do not have available source code. It is also difficult to know which algorithm to try to use.

This thesis addresses these challenges in several ways. It provides the necessary background information to understand bitwidth optimization algorithms, as well as a survey of the existing literature. It also presents a new framework called “Bitwidth Analysis Tool” (BAT) built on the open source Ptolemy tool. This framework is designed to help implement and compare bitwidth optimization algorithms. Some existing algorithms are implemented within this new framework, and compared with each other on a variety of benchmarks.

The comparison results verify that because the tested algorithms are heuristics, no single algorithm gives the best results in all cases. It is therefore important to test a variety of algorithms to try to find the best answer. The results also show existing algorithms and error models provide a good starting point, but existing error models do not yet provide sufficiently tight bounds to be useful in large complex systems.

Keywords: FPGA, fixed-point, bitwidth analysis, digital signal processing, DSP, Ptolemy
ACKNOWLEDGMENTS

I would like to acknowledge my family for the extreme patience and support they have given me. I would also like to especially thank my wife, without whose support and patience I would not have been able to complete this work. I would also like to thank Edison for the encouragement he gave me to finish my thesis in a timely manner.

I would like to acknowledge the students in the FPGA lab who have helped me in my research over the years.

I would like to thank my adviser, Dr. Wirthlin, for giving me the freedom to pursue this research and for the support he has given me.

I would like thank Dr. George Constantinides, who provided some initial feedback and direction.

I also would like to acknowledge the major sources of funding for the research that went into this thesis. This research is supported by the I/UCRC Program of the National Science Foundation under Grant No. 0801876 through the NSF Center for High-Performance Reconfigurable Computing (CHREC).
# TABLE OF CONTENTS

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

LIST OF TABLES .................................................................. xiii

1 Introduction ................................................................... 1

2 Overview of Bitwidth Optimization Process .................. 5

  2.1 Fixed Point Numbers and Definitions ....................... 7

  2.2 Bitwidth Selection Process .................................. 11

3 Range Analysis ............................................................ 13

  3.1 Interval Arithmetic ............................................. 13

  3.2 Affine Arithmetic .............................................. 14

    3.2.1 Affine Arithmetic in IIR Filters ....................... 17

    3.2.2 Probabilistic Affine Arithmetic ..................... 19

  3.3 Simulation ......................................................... 19

4 Error Analysis ............................................................ 23

  4.1 Using Range Arithmetic to Measure Error ................. 23

  4.2 Using Simulation to Measure Error ....................... 24

    4.2.1 Tools Available for Simulation .................. 25

5 Precision Selection ..................................................... 27

  5.1 Heuristic Competitions ..................................... 27

    5.1.1 Competition Variants ............................. 30
5.2 Other Techniques .................................................. 32
   5.2.1 Optimal Techniques ........................................... 33
   5.2.2 Polynomial Based Technique ................................. 33

6 Implementation of the Bitwidth Analysis Tool ..................... 35
   6.1 The Bitwidth Analysis Tool as a Ptolemy Extension .............. 36
      6.1.1 Bitwidth Director ........................................... 38
      6.1.2 Tokens .................................................... 39
      6.1.3 Actors .................................................... 43
   6.2 Implementing Bitwidth Algorithms Using BAT .................... 44
      6.2.1 Minimum Uniform Bitwidth ................................. 47
      6.2.2 Uniform Bitwidth -1 Bit .................................. 47
      6.2.3 Scaled Uniform Bitwidth -1 Bit ............................ 47
      6.2.4 Min +b Bits ............................................... 49
      6.2.5 Simulation ............................................... 51
      6.2.6 Cost Function ............................................. 51

7 Competition and Error Model Analysis Results ........................ 55
   7.1 Previous Work .................................................. 56
   7.2 Experimental Setup ............................................. 57
   7.3 Test Circuits .................................................. 58
   7.4 Range Analysis Results ....................................... 59
   7.5 Error Analysis ................................................ 62
   7.6 Heuristic Comparisons ........................................ 64
      7.6.1 Area Cost Function Results ............................... 65
      7.6.2 Number of Fractional Bits ................................. 67
      7.6.3 Competition Runtime ...................................... 69
      7.6.4 Measured Error vs. Error Constraint ....................... 70
# LIST OF FIGURES

2.1 Circuit Area and Error vs. Bitwidth .................................................. 6
2.2 Flowchart for the Bitwidth Selection Process ..................................... 11
3.1 The Joint Range of \( x \& y \) Using Interval Arithmetic and Affine Arithmetic. 16
3.2 Example of AA and IA in a Feedback System ..................................... 18
3.3 Affine Arithmetic in IIR Filters ......................................................... 19
5.1 Example of How a Competition Round Proceeds ................................... 29
6.1 Bitwidth Analysis Tool and Ptolemy ................................................. 36
6.2 UML Diagram for Token Classes ....................................................... 40
6.3 UML Diagram for Strategy Classes .................................................... 46
6.4 Algorithm Flow for Uniform Bitwidth -1 Bit ..................................... 48
6.5 Algorithm Flow for Scaled Uniform Bitwidth -1 Bit ............................. 49
6.6 Algorithm Flow for Min+b Bits ......................................................... 50
7.1 Interval Arithmetic v. Affine Arithmetic in a YUV Converter Example ...... 59
7.2 Comparison of Algorithm Variations .................................................. 64
7.3 Area Comparisons for Various Circuits. ........................................... 66
7.4 Fraction Bit Count Comparisons for Various Circuits. .......................... 68
A.1 Model Used for the 4 Tap FIR Filter ............................................... 83
A.2 Model Used for the 5 Tap IIR Filter .................................................. 85
A.3 Model Used for the Farrow Interpolator ............................................ 86
A.4 Model Used for the DCT Circuit ....................................................... 88
A.5 Model Used for the LMS Filter ............................................. 89
A.6 Model used for the BPSK Timing Loop .............................. 90
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>AA and IA Range Compared to Simulation for DCT</td>
<td>61</td>
</tr>
<tr>
<td>7.2</td>
<td>AA and IA Range in DCT</td>
<td>62</td>
</tr>
<tr>
<td>7.3</td>
<td>Calculated Range v. Simulation Range</td>
<td>63</td>
</tr>
<tr>
<td>7.4</td>
<td>Comparison Between Calculated Error and Simulation Error</td>
<td>63</td>
</tr>
<tr>
<td>7.5</td>
<td>Ranking of Competition by Area Cost Function for FIR Systems</td>
<td>65</td>
</tr>
<tr>
<td>7.6</td>
<td>Ranking of Competition by Area Cost Function for IIR Filters</td>
<td>65</td>
</tr>
<tr>
<td>7.7</td>
<td>Ranking of Competitions by Total Fractional Bit Count.</td>
<td>69</td>
</tr>
<tr>
<td>7.8</td>
<td>Ranking of Competitions by Runtime</td>
<td>70</td>
</tr>
<tr>
<td>7.9</td>
<td>Comparison Between the Calculated Error and the Error Constraint for a YUV Circuit</td>
<td>70</td>
</tr>
<tr>
<td>A.1</td>
<td>Filter Coefficients for a 30 tap FIR Filter</td>
<td>84</td>
</tr>
</tbody>
</table>
CHAPTER 1. INTRODUCTION

When designing a digital system, the infinite set of real numbers must be mapped to a specific, finite set of binary values. The size of this set is determined by the number of bits used to store each value, which in turn determines the amount of quantization error in the system. The larger the set of binary values, the less quantization error there will be. However, using a larger set of binary values also increases the area of the circuit.

The process of bitwidth optimization involves balancing the two opposing design requirements of minimizing the area of the circuit and minimizing the quantization error. Minimizing the area involves reducing the bitwidth, which increases the error. Decreasing the error involves using a larger bitwidth, which increases the area. The goal of bitwidth optimization is finding the set of bitwidths that results in the smallest circuit area while satisfying the system quantization error requirements.

When designing signal processing algorithms in software, bitwidth optimization is not generally a concern. This is because the software runs on general purpose hardware such as DSPs and CPUs. These systems have a fixed number of arithmetic units, each with a predetermined bitwidth. General purpose processors need to run code for a variety of applications, so the hardware designers use standard fixed sizes for both integer and floating-point units. These units are wide enough for the vast majority of applications. Because the entire datapath is exercised regardless of the actual size of the data, attempting to optimize the bitwidth does not make the system more efficient, and in some cases can reduce performance [41].

When designing custom hardware, bitwidth optimization can produce significant benefits. A custom hardware design is often single purpose. This means that a designer can
use an arbitrary number of arithmetic units, depending on the application and the design strategy. Because these units are not shared, they can have different bitwidths to meet the specific needs of the function they are implementing. To make the circuit as small as possible, the hardware designer can select an optimal bitwidth for each unit. These smaller units allow either more room for other units, or allow the overall design to use less space and therefore less power.

However, this flexibility does not come without a cost. Design times for hardware are already longer than those for software, and selecting an optimal bitwidth for each operator further increases hardware design time. Finding the optimal bitwidth is difficult because the error as a function of the bitwidths creates a non-convex space. For example, the output error of a circuit with \( n \) operators can be expressed as \( e(bw_1, bw_2, ..., bw_n) \) and will create a non-convex \( n \)-dimensional space. If the candidate bitwidths range from 1 bit wide to 32 bits wide, there are \( 32^n \) points in the space. Because the space is non-convex, all of the points must be tested to ensure that the chosen answer is the best one. In fact, finding the optimal set of minimum bitwidths has been shown to be NP-hard [16].

Many people have explored algorithms for finding a minimum or near minimum set of bitwidths in a reasonable time. Their papers often mention tools that have been developed to implement these algorithms including Synoptix [9; 10; 11], Precis [5], Bitwise [41], BitSize [22], MiniBit [30; 31], Minibit+ [34] and several other, unnamed, tools [2; 3; 8; 39; 42]. These algorithms seem to produce good results.

However, none of these tools are available outside of the labs in which they were developed. In order for a reader to run these algorithms on a circuit, the algorithms must be implemented from scratch, using only the details provided in the corresponding paper. Many papers also leave the implementation of their algorithms somewhat vague, which makes it difficult for an engineer to implement the published algorithm. In addition, critical information surrounding the process of bitwidth analysis is spread among many papers, requiring
the engineer to invest significant time finding and reviewing the literature before attempting to implement an algorithm.

This thesis will address several of the current challenges in bitwidth selection. It provides a survey and summary of some existing techniques, which will serve as a starting point for readers wishing to learn more about bitwidth optimization. This thesis also presents a new and open bitwidth optimization framework, which will allow users to quickly implement and test new or existing algorithms. Three existing algorithms are implemented and compared within this new tool, giving valuable insight into the current state of bitwidth optimization and demonstrating the usefulness of the new framework.

The thesis provides a survey of existing bitwidth optimization techniques, complete with the background information necessary to understand the current literature. The background information presented in the next few chapters provides a starting point by introducing the basic concepts, and by citing papers that can be read for more details on specific aspects of bitwidth optimization. Armed with this background information, the reader will be better able to read and understand the current literature.

Because there are no tools available to test these bitwidth analysis algorithms, I have implemented a new, open source framework for developing and testing bitwidth selection algorithms. The tool is built on the Ptolemy project from Berkley [17] and is available as an open source program so that others who wish to work with bitwidth optimization techniques will have a foundation on which to start. The framework has several working examples of bitwidth analysis algorithms, and is built so that new algorithms can be added easily. With the core portions of bitwidth optimization algorithms already implemented, users can focus on the new aspects of algorithms, rather than starting from scratch. The tool is discussed in more detail in Chapter 6.

The thesis will use this new tool to compare existing bitwidth optimization techniques. One comparison paper was published in 2002, but used only simulation to measure the error, rather than using an analytic error model [4]. Analytic error models provide provable bound,
which is something that simulation cannot do. The comparisons presented here will compare results obtained with analytic error models, and will also include algorithms developed since that paper was written, providing new information for engineers looking to utilize bitwidth optimization techniques.

The thesis is organized as follows. Detailed background information necessary for understanding the bitwidth selection process is presented in Chapters 2, 3, 4, and 5. Second, the new and open framework, named “Bitwidth Analysis Tool”, is described in Chapter 6. Third, the results from comparing the three implemented bitwidth algorithms are presented in Chapter 7. Finally, conclusions and future work are discussed in Chapter 8. In short, this work makes it possible for readers to implement and test bitwidth optimization algorithms on their own circuits, by providing background information, a tool, examples, and comparisons.
CHAPTER 2. OVERVIEW OF BITWIDTH OPTIMIZATION PROCESS

In order to implement and compare bitwidth optimization algorithms, some background is necessary. This chapter will discuss the importance and impact of bitwidth optimization, as well as some key terminology for fixed-point and bitwidth optimization that will be used throughout the paper.

Finding the optimal set of bitwidths is important because we want to build the smallest system possible while maintaining a certain level of accuracy. The accuracy of the outputs of a circuit is a function of the bitwidths used for all of the intermediate values in the system. Rounding error, or quantization error, accumulates throughout the circuit, and will be larger at the output than at any particular internal operator. The process of bitwidth optimization presented in this thesis seeks a custom set of bitwidths which meets a certain error constraint, while optimizing some other characteristic of the implementation such as speed, area, or power consumption [3]. For simplicity this thesis will assume that the primary goal is area reduction. We will call the set of operator bitwidths the configuration of the circuit.

To illustrate the effect that bitwidth has on the area and quantization error of a circuit, we will examine a simple FIR filter, as shown in Figure 2.1a. FIR filters are foundational in signal processing applications, and measuring the quantization error at the output is relatively straightforward. In this particular example, we will use a filter with four taps. Such a filter has only thirteen different bitwidths: the input to the four gain blocks (4), the four coefficients (4), the output of the gain blocks (4), and the output of the final adder (1). If the possible bitwidths range from 1 to 32, then there are $32^{13}$, or 36 trillion unique combinations, which create a non-convex 13-dimensional space. Not all of these combinations will result in a circuit that meets the designer’s error constraints. Because the space is non-
Figure 2.1: Circuit area and error vs. the system uniform bit width. Area is estimated by a cost function, and is measured in LUTs for a Xilinx Virtex part.

convex, however, we cannot easily identify which combinations will or will not meet the error constraint. The goal is to find the combinations that results in the smallest circuit area from the set of combinations that meet the error constraint.

If we limit the space to only configurations where each operator has the same bitwidth, then we have a space with 32 configurations. This space is convex in both area and quantization error. This space will not contain the optimal configuration, but can be visualized, as
shown in 2.1b. This figure illustrates the effect that bit width has on area and quantization error. The left vertical axis shows the area of the filter measured in LUTs, as computed by the cost function described in Section 6.2.6. As the bitwidth increases from 1 bit to 32 bits, we see a 10x increase in the area of the circuit. The right vertical axis shows the quantization error at the output, as measured by the Bitwidth Analysis Tool presented in Chapter 6. It shows that as the bitwidth increases from 1 bit to 32 bits, the quantization error decreases by 11 orders of magnitude.

If a designer were constrained to using uniform bit widths, then such a graph could be used to find the smallest bit width that satisfies the error constraint. However, this small set of bit widths does not contain the optimal configuration. Because the search space in non-convex, finding the optimal solution requires testing every point. Because this is not feasible, many heuristics have been developed in an attempt to search the space intelligently and find a configuration that is sufficiently close to the true optimal configuration. Several such algorithms are summarized in Chapter 5.

2.1 Fixed Point Numbers and Definitions

In order to understand the details of the bitwidth optimization process, an understanding of fixed-point numbers is important. Although fixed-point numbers are not the only way of storing numbers in custom hardware, they are often preferred due to their lower implementation complexity compared with floating-point numbers. Those interested in learning more about custom floating-point types may find [21; 19; 34] of interest. Only fixed-point numbers will be considered in this thesis.

A fixed-point representation of a number \( x = x_{INT} + x_{FR} \) consists of integer and fraction components represented by \( m \) and \( f \) digits respectively [18]. A binary number \( B \) in fixed point can be converted to the decimal number \( x \) by

\[
x = \sum_{i=0}^{i=m} B_i 2^i + \sum_{i=1}^{f} B_i 2^{-i}.
\] (2.1)
Fixed point numbers have a fixed number of digits after the radix (decimal or binary) point. As such, they have an implicit scale factor. For example, the number 1.23 can be stored as 1230 with an implicit scale factor of 1/1000. This can easily be extended to binary numbers by using scale factors that are a power of 2 rather than a power of 10. For example, to store the number 2.5 one could store 5 with a scale factor of 1/2, or 10 with a scale factor of 1/4, etc. The chosen scale factor is known by the hardware in advance. This differs from floating point values, which store both the mantissa (number portion) and the exponent (scale factor).

**Range** The range is the set of values a variable can assume during the computation of an algorithm [23]. It is usually represented as the minimum and maximum values of the set. These values must be known so the fixed-point representation can be chosen with enough bits to avoid any underflow or overflow errors.

**Precision** The precision of a numerical quantity is a measure of the detail in which the quantity is expressed [23]. This is usually measured in bits, but sometimes in decimal digits. It is related to precision in mathematics, which describes the number of digits that are used to express a value. In this thesis precision will refer to the number of fractional bits.

The minimum resolution of a number is determined by the precision according to the equation

\[ r = 2^{-p} \]  

(2.2)

where \( r \) is the minimum resolution and \( p \) is the precision. If a fixed point number has a precision of 2, then the resolution is 1/4, which means that it cannot represent any increment smaller than 1/4. Data types with a higher precision are capable of storing smaller increments than numbers with a lower precision. A higher precision value will create less rounding error than a lower precision number, because the distance between the value to be represented and the nearest representable value will be smaller, but will increase the complexity and size of the implementation.
**Accuracy** Accuracy is the degree of closeness between a stored value and its true value. Since the true value cannot be known exactly, a number of higher accuracy is used as a reference value [23]. While related to precision, it is not the same thing. For example the number 3.1400001 is a precise representation of $\pi$ to 8 significant figures, but is only accurate to 3 significant figures. Accuracy can be lost in a computation through rounding errors, and the accuracy of the final answer will be less than the precision the value implies. This makes knowing the accuracy of a number difficult to determine.

**Quantization Error** Quantization error is the difference between the actual value and the quantized digital value. If the fractional portion of infinite precision value is represented in binary form as

$$U = \sum_{i=1}^{\infty} U_i 2^{-i} \quad (2.3)$$

and is quantized to $f$ bits, then the quantized form will be

$$U_q = \sum_{i=1}^{f-1} U_i 2^{-i} + [U_f + \{1 \text{ or } 0\}] \quad (2.4)$$

where with rounding a 1 or 0 is added to the $f$th bit depending on whether the $(f+1)$th bit is a 1 or 0. With truncation the bits beyond the most significant $f$-bits are dropped.

If the error that occurs during the quantization is denoted by $\epsilon$,

$$\epsilon = U - U_q \quad (2.5)$$

then for rounding $\epsilon$ will lie between $-2^{-f}$ to $2^{-f}$, and for truncation the error will lie in the range $0$ to $2^{-f-1}$ [1]. The quantization error is sometimes considered as an additional random signal called quantization noise because of its stochastic behavior.

This error can be measured in a variety of ways. One way is to measure the minimum and maximum rounding error, and call this the range of the error. This method is in several papers, including [30], [34], and [20]. Another way is the measure the statistical properties
such as the mean and variance of the rounding error. This method can be especially applicable to signal processing systems, which are concerned about the power (variance) of the error relative to the power of the signal. This method is used in papers such as [11] and [13]. Measuring the maximum range of the error is simpler, but is more pessimistic. Measuring the power of the error may allow some occasional quantization error to be much larger than the given bound. However, this may be acceptable for many signal processing applications.

**Error Constraint** The error constraint is the maximum amount of error allowed at the output. This is a design specification, and the designer must make decisions about what bitwidths to use throughout the system to make sure the output error is always less than the error constraint. It could be specified as a Signal-to-Noise ratio (where the quantization error is considered as noise) or as a maximum value for the accumulated error.

**Bitwidth Configuration** A bitwidth configuration, or simply *configuration*, as used in this thesis means a set of operator bitwidth assignments. A particular configuration specifies the bitwidth of each operator in the system.

The *optimal configuration* is the set of bitwidths that will result in the smallest circuit which meets the error constraint. It can be expressed as

$$\arg \min(Area(BW)) \quad \text{when} \quad Error(BW) < error\_constraint$$  \hspace{1cm} (2.6)

where $BW$ is the optimal set of Bitwidths.

The number of possible configurations for a circuit is the number of possible bitwidth assignments raised to the power of the number of operators. For example, if permissible bitwidth assignments ranged from 1 to 32, then the number of possible circuit configurations would be $32^n$, where $n$ is the number operators in the circuit.
Figure 2.2: Flow chart showing the process of bitwidth selection. The boxes Range Selection, Measure Error and Precision Selection will be discussed in Chapter 3, Chapter 4, and Chapter 5 respectively.

### 2.2 Bitwidth Selection Process

Finding the optimal set of bitwidths is difficult because the various circuit configurations create a non-convex search space, which makes the problem NP-Hard. Because the
only way to find the optimal configuration is to try all possible configurations, heuristics have been developed to search the space in an intelligent way attempting to find a “good enough” solution. One very simple heuristic is to choose the smallest uniform bitwidth that meets the error constraint. However this is not efficient because the output error is not equally sensitive to all the of the widths in the system.

Many heuristics follow the general outline shown in Figure 2.2. The flow chart is used to implement the various algorithms discussed in this work. Some other techniques, such as non-heuristic methods do not follow this type of flow, and are discussed in greater detail in Section 5.2. The first step in the process is to find the range of values that each signal can take, as represented by the box labeled Range Selection. This indicates a portion of the data type that must be used to avoid overflow errors. Techniques for measuring the range will be discussed further in Chapter 3 of this work. Second, there must be a way to measure the error of a given circuit configuration, to ensure that the selected configuration meets the error constraint. This is shown in the box labeled Measure Error, and is discussed in more detail in Chapter 4. Finally there must be an algorithm for selecting the configurations to test based on the error measurements of other configurations, as discussed in Chapter 5. This is shown by the large box labeled Precision Selection. Three such algorithms were implemented for this work, and are compared in Chapter 7.
CHAPTER 3. RANGE ANALYSIS

In order to determine if a particular system configuration meets the error bound, the range and the quantization error must be measured. The purpose of range analysis is to find the minimum and maximum values of the signals throughout the system. This is used to ensure that there are sufficient bits to prevent any overflow. This chapter discusses several existing techniques for measuring the range of the values throughout the system. These techniques were incorporated into the new Bitwidth Analysis Tool presented in Chapter 6. The rules presented here can also be used for measuring the range of the quantization error. These concepts are used extensively in the implementation of bitwidth selection algorithms.

There are several methods for calculating the range of the signals within a system. Two of the static methods are interval arithmetic (IA) and affine arithmetic (AA). Simulation can also be used, but cannot provide the definite bounds that static methods can.

In general range does not depend on the precision, and can be found before attempting to find a precision. However, some methods attempt to find the range and the precision simultaneously. A few of these composite methods are discussed in Section 5.2.

3.1 Interval Arithmetic

Interval arithmetic (IA), also known as interval analysis, was developed in the 1960s by Ramon E. Moore [33] to solve range problems. In interval arithmetic, each value is replaced by a range, represented by $x = [x.lo, x.high]$ where $x$ is known to lie between $x.lo$ and $x.high$. Operators then act on these ranges and produce new ranges as results. For
example, the resulting range of an addition is

\[ z = x + y = [x.lo + y.lo, x.hi + y.hi]. \]  

(3.1)

Similar formulas can be derived for other operators, and can be found in the original work by Moore [33].

Unfortunately, interval arithmetic is overly pessimistic, and ranges can explode in even trivial cases. One such example is when \( x = [-1, 1] \) and \( y = [-1, 1] \) and \( x \) and \( y \) are bound by the relationship \( x = -y \). Using Equation 3.1 we have \( z = [-2, 2] \). However, the true value is \( z = 0 \). Because the same input was used twice, with a different sign, interval arithmetic produces a range that is overly pessimistic. These over estimations can accumulate throughout the system and can result in exponential range growth. This problem is addressed by affine arithmetic [20].

### 3.2 Affine Arithmetic

Affine arithmetic (AA) was first developed in [7] to address some of the limitations of interval arithmetic. This section will summarize affine arithmetic as presented in [7]. In affine arithmetic the partially unknown quantity \( x \) is represented by an affine form \( \hat{x} \), which is a first-degree polynomial

\[ \hat{x} = x_0 + x_1 \epsilon_1 + x_2 \epsilon_2 + \cdots + x_n \epsilon_n \]  

(3.2)

where the \( x_n \) are known and the \( \epsilon_n \) are unknown error terms which lie in the interval \( U = [-1, +1] \). Each error term represents a different source of uncertainty, either from previous inputs or from rounding/truncation. The equation

\[ [\hat{x}] = [x_0 - \xi, x_0 + \xi], \quad \xi = \sum_{i=1}^{n} |x_i| \]  

(3.3)
will convert an affine form to a standard interval, where \( x_0 \) is known as the central value, and the \( x_n \) as residuals. A standard interval can be converted into an affine form \( \hat{x} = x_0 + x_1 \epsilon_1 \) using

\[
x_0 = \frac{x_{hi} + x_{lo}}{2}, \quad x_1 = \frac{x_{hi} - x_{lo}}{2}.
\]

(3.4)

In revisiting the example from interval arithmetic, we have

\[
x = 0 + 1_{\epsilon_0}
\]

\[
y = 0 - 1_{\epsilon_0} = -x
\]

\[
z = x + y
\]

\[
= 0 + 1_{\epsilon_0} - 1_{\epsilon_0} = 0.
\]

The true power of affine arithmetic comes when performing an operation on variables that share error symbols. Because it accounts for correlations between signals, affine arithmetic can be used to find signal ranges within feedback systems, such as IIR filters, where interval arithmetic produces explosive bit growth [19; 20]. As an example of affine arithmetic’s ability to reduce range by tracking correlation, consider the values of \( x \) and \( y \) where

\[
\hat{x} = 10 + 2\epsilon_1 + 1\epsilon_2 - 1\epsilon_4
\]

\[
\hat{y} = 20 - 3\epsilon_1 + 1\epsilon_3 - 4\epsilon_4.
\]

From this we know that \( x \) lies in the interval \([6, 14]\), and \( y \) lies in \([12, 28]\). This range is shown by the shaded box in Figure 3.1. Because of the shared error symbols, the pair \((x, y)\) found using affine arithmetic lies in an area half the size of the square found by using standard interval arithmetic. This more limited range is shown by the darker area inside the interval arithmetic square.
Figure 3.1: The joint range of x & y using interval arithmetic and affine arithmetic. The area using affine arithmetic is half the area using interval arithmetic, because of the shared error terms.

Computing the affine forms for affine (linear) operators, such as addition and multiplication by a constant, are straightforward, and produce exact results. However, for non-affine operations, such as general multiplication and square-root, the expression becomes non-affine. For example, if we have $a = [-3, 5]$ then the affine form for $a$ is $a = 1 + 4\epsilon_1$. If we want to find an expression for $x$ where $x = a \times a$ then the affine expression is $x = 1 + 8\epsilon_1 + 16\epsilon_1^2$ which is no longer affine because of the $\epsilon^2$ term. This can be dealt with in a variety of ways. One way is to replace the squared term with a new affine term over the same range but in a new variable, (e.g. $8 - 8\epsilon_2$). This dilutes some of the correlation, and can result in very long expressions. Long expressions can be simplified in a similar manner by replacing several expressions with a shorter one. However this further dilutes the correlations, and has a negative impact on the quality of the final result.
AA can occasionally produce worse estimates than IA. This is due to the errors in the affine estimate of non-affine functions such as general multiplication, rather than a shortcoming of the approach itself. One such example is given in [30] and is reproduced here.

Given that \(a = [-3, 2]\) and \(b = [4, 8]\), we want to find the range of \(d\) where \(d = a \times b\). Using interval arithmetic we get a range for \(d\) of \([-24, 16]\). The affine expressions for \(a\) and \(b\) are

\[
a = -0.5 + 2.5\epsilon_1 \\
b = 6 + 2\epsilon_2.
\]

Using these expressions we get \(d_{AA} = -3 + 15\epsilon_1 - 1\epsilon_2 + 5\epsilon_1\epsilon_2\). We replace the non-affine term \(\epsilon_1\epsilon_2\) with \(\epsilon_3\) and use Equation 3.3. We then get a range of \([-24, 18]\) which is larger than the range given by interval arithmetic. This is due to the error introduced when linearizing the non-affine general multiply.

### 3.2.1 Affine Arithmetic in IIR Filters

Using interval arithmetic in a stable IIR filter results in explosive range growth, because the correlations are lost. However, affine arithmetic can use the correlation in the feedback to find a closed solution for the range of a system that includes an IIR filter. Figure 3.2 shows a simple IIR filter with intermediate values at time=3 for both interval arithmetic and affine arithmetic. The most important values to notice are those after the second add block (labeled \(+2\)). Using interval arithmetic, the maximum possible value is 1.7 + 0.8 = 2.51. Because affine arithmetic is tracking the correlation between the values, we know that 0.81\(\epsilon_1\) from the first tap will nearly cancel the \(-0.8\epsilon_1\) result from the second tap, resulting in \([(0.81\epsilon_1 + 0.9\epsilon_2) + (-0.8\epsilon_1) = 0.01\epsilon_1 + 0.9\epsilon_2 = [-.91, .91]]\). When using interval arithmetic this correlation is lost, and we must choose 2.51 as the maximum value instead.
Figure 3.2: An example highlighting the difference between affine and interval arithmetic in a feedback system.

of 0.91. The IA bound will grow exponentially, and in just a few iterations will no longer be tight enough to be a useful bound.

Any stable IIR filter will have a stable bound produced through affine arithmetic. However the usefulness of the resulting bound is closely related to the location of the poles. As the poles get closer to the unit circle, the accuracy of affine arithmetic goes down, and the convergence time goes up, as described in [19]. There the authors used

\[ \text{Accuracy} = \frac{\text{real error}}{\text{estimated error}} \]

as a metric. The accuracy decreases linearly, while the convergence time, or the number of iterations required until the outputs stabilize, goes up exponentially. When used to find the range of a filter with poles near the unit circle, the range given by affine arithmetic will be much larger than the true range, and will take many more iterations to calculate. On the other hand a filter than has poles close to the origin will have a tighter bound, and the algorithm will not require as many iterations to come to a solution.
Figure 3.3: Accuracy and stability of affine arithmetic in an IIR filter as a function of the pole location. Reproduced from [19].

A small increase in the convergence time can have a large impact on the bitwidth selection algorithms, because each unique set of bitwidths has to be tested, and each test requires the outputs to stabilize.

3.2.2 Probabilistic Affine Arithmetic

Typically the range is computed from an affine form using Equation 3.3. However, when the number of error terms is large, it is unlikely that the error terms will all reach their maximum value simultaneously. The sum of the error terms is Gaussian by the central limit theorem, so the authors of [20] developed a bound based on a specified confidence interval. The authors used a confidence interval of 99.9999% and were able to achieve much tighter bounds in some cases. The derivation for the equations necessary to find this probabilistic bound are discussed in more detail in Section 6.1.2

3.3 Simulation

Another method of determining the range of the signals in a system is simulation. Using simulation to measure range involves executing the circuit with a set of test vectors, and measuring the maximum and minimum values reached by all of the internal nodes
during the iterations. The circuit must be run for a large number of iterations to generate some confidence that the simulation inputs were able to push the internal variables to their maximum values. This leads to one of simulations greatest drawbacks: it cannot provide an absolute bound, because there is no way to verify that the simulation inputs caused all of the interval values to reach their maximum value.

Because simulation cannot provide true bounds, once a range has been found using simulation, it often increased by a “fudge factor”. One such method is presented in [12] and is a combination of the approaches proposed in [26; 29]. In this method the system is simulated using the user provided inputs, and the peak values $P_s$ are recorded for each signal $s$. These are then scaled by a safety factor $k$ (typically $k = 4$, which provides 2 guard bits for each signal) so that $p_s = kP_s$. A tighter bound of $p_s = \lceil \log_2 kP_s \rceil + 1$ was presented in [15]. Because the ranges are scaled, the resulting circuit may be ill-conditioned, with some bitwidth larger then the theoretically necessary. An example of an ill-conditioned circuit would be an adder with input integer widths of $p_a$ and $p_b$, and output integer width $p_c$. The output width $p_c$ should be no larger than $p_a + p_b + 1$. If it is then the circuit is ill-conditioned, then $p_c$ is reduced to $p_a + p_b + 1$. The circuit is conditioned repeatedly until the scaling converges.

Another simulation technique uses *extreme value theory* [43]. It uses the Gumble distribution to model the distribution function of the minima and maxima of each node in the system. After running a number of simulations, the authors use the Gumbel distribution and a confidence interval to determine the necessary integer bitwidths for each node. Their test results show that this method is faster than a brute force simulation, and because it uses confidence intervals, it still provides tighter bounds than static methods such as interval and affine arithmetic.

A simple simulation model is implemented as part of the Bitwidth Analysis Tool presented in Chapter 6. The implementation runs the circuit for a large number of cycles,
and records the minimum and maximum values at each node. It does not apply any of the more advanced techniques discussed in the previous paragraphs.

In summary, measuring range is a necessary first step in bitwidth optimization. As such interval arithmetic, affine arithmetic, and probabilistic affine arithmetic are implemented the Bitwidth Analysis Tool, as detailed in Chapter 6.

Once the range is known, the error of a given configuration can be measured. Many of the same techniques are reapplied when measuring the error of a circuit configuration, and are discussed in Chapter 4. Measuring the range of the values in a system does not seem to have the same challenges that measuring error does. This is because overflow can be completely avoided, so that the range of one operator has a predictable effect on the range of the following operators. Because rounding and quantization errors cannot be eliminated, measuring their effect on the system output becomes much more complicated. Some techniques for measuring the error will be discussed in the next chapter.
CHAPTER 4. ERROR ANALYSIS

In order to determine if a particular system configuration meets the error bound, the quantization error must be measured. Doing so requires that the range of each operator be known, as discussed in the previous chapter. This chapter discusses several existing techniques for measuring the error of the values throughout the system. Several of these methods were implemented and compared in the new Bitwidth Analysis Tool.

In order to find an optimal set of bitwidths, there must be a way of determining the maximum error of a given configuration. Once the error is known, the precision selection algorithm or engineer can choose to increase or decrease the size of the circuit to come closer to meeting the error constraint.

The techniques for measuring error closely mirror those of measuring range. The minimum and maximum values of the error can be found, either through range arithmetic or simulation. In addition, the statistical properties of the error can be measured, either analytically or through simulation. Measuring the statistics such as mean and variance can be especially applicable to signal processing systems, where the error constraint may be expressed in terms of the power(variance) of the error.

4.1 Using Range Arithmetic to Measure Error

There are two ways to quantize a signal: truncation and rounding. Truncation has a higher error, but requires no extra hardware, whereas faithful rounding requires extra hardware in order to achieve it’s lower maximum error. Depending on the chosen rounding model, the maximum error is the value of the least significant bit (LSB) for truncation,
or half of the LSB when using faithful rounding. These error ranges are then propagated through this system using the same rules discussed in Chapter 3.

In [20] the authors create a single affine expression for the entire system, which includes the range error terms and the quantization error terms. This allows the solution to capture the dependency that the range and quantization error may have on each other. However, this approach has been criticized because it makes the equations more complex and the increase in complexity does not justify the minimal gains in the chosen solution [30].

Measuring the quantization error variance analytically can be challenging. Doing so requires knowing the transfer function from each operator to the output to determine how the quantization error at each node affects the output [10; 27]. It also limits the technique to LTI systems, whereas affine arithmetic could be used on some simple non-LTI systems.

However, a recent method attempts to overcome this challenge by using a modified form of affine arithmetic. Using this modified affine arithmetic allows the authors to estimate the transfer function during runtime using the affine error terms. This makes the approach applicable to non-linear systems as well as linear systems. The authors boast runtime improvements of 3 orders of magnitude, and very low estimation error [3].

4.2 Using Simulation to Measure Error

Simulation is often the only method that can be used to measure error in complex systems. While LTI systems lend themselves to static analysis, non-linear systems and systems with complex control logic cannot effectively be analyzed statically. In these systems simulation is often the only way to attempt to verify correctness.

When using simulation, two separate simulations must be run, one at a very high precision, such as double precision floating point, to create a reference, and another at the proposed finite precision. These two simulations can be performed simultaneously. This allows any control flow decisions to be made based on the fixed-point values, which can give a more accurate estimation of the error [6]. However, in iterative systems where the conver-
gence time may change with the chosen precision, running the simulations simultaneously may hide that effect.

Once the differences between the high precision and proposed finite precision models have been measured, there are two ways to evaluate the error. One is to analyze the absolute error introduced by the chosen precision. The other is to find the statistics of the deviations caused by the finite precision. Tracking the statistics of the error can often decrease the number of iterations required to produce a reasonable estimate of the maximum error.

One of the greatest drawbacks of simulation is the fact that the results are only as good as the simulation inputs. When trying to determine the accuracy of the computation, the simulation inputs need to generate the worst case instantaneous deviation for the true result, or a good estimate of the statistical properties of those deviations. The problem then becomes how to create a suitable set of simulation inputs.

Many papers have focused on ways to improve simulation. The simulation method described in [6] runs a floating point and fixed point simulation simultaneously, measures the error and collects statistics on the error. The advantage of running the simulations simultaneously instead of serially is that any control decisions are determined based on the fixed point results, and are the same for both the fixed point and floating point simulations.

The mean ($\mu$), standard deviation ($\sigma$) and maximum absolute error ($\epsilon$) are measured and stored. Since all of these statistics are dependent on the fractional word length, they can then be used to help determine the optimal width. However, the floating and fixed point values may diverge in sensitive feedback systems, due to the high correlation of the error values. In this case the authors of [6] break the feedback loop by manually inserting an error value in the feedback loop.

4.2.1 Tools Available for Simulation

There are a variety of tools that will assist an engineer in measuring the error of a given system using simulation. Such tools include Matlab and Synopsis System Studio [28].
However, these tools do not make any suggestions to the developer about which bitwidths to increase or decrease. They simply help the developer measure the effects of the chosen bitwidth configuration.

Synopsis System Studio has several simulation based tools to assist developers. One of the tools in System Studio converts SystemC data types to native machine data types, improving simulation speed. Because the conversion is done automatically, the code is easier to read, maintain, and there is less room for error. Doing this conversion can bring simulations speeds from 30x slower than floating point to only 1-3x slower than floating point simulations. Another tool in System Studio automatically collects signal statistics through a simulation. This can help determine the necessary range and precision of each signal. Collecting statistics automatically helps keep code clean and easier to maintain [28].

Matlab also has tools for assisting engineers in measuring the system error. There is a fixed point tool-box for implementing custom fixed point data types. This toolbox can be used both in the textual code environment as well as the GUI based Simulink simulation tool, and provides accelerated execution to speed simulation times. These fixed point simulations can then be compared to floating point simulations to determine the error [32; 38].

Once the range and error are measured, bitwidth selection algorithms can make decisions about increasing or decreasing the bitwidth of the operators throughout the system. Although range need to be measured only once, the error must be measured for every new configuration of the circuit. The following chapter discusses a variety of algorithms that use the measured error to search for a near optimal bitwidth.
CHAPTER 5. PRECISION SELECTION

Armed with the range and error measurements an algorithm can search for the optimal set of bitwidths. This chapter summarizes algorithms developed by a variety of authors. Several of the algorithms discussed here are implemented in the Bitwidth Analysis Tool, and more details on their implementation can be found in Section 6.2.

Because finding the optimal set of bitwidths is NP-hard, there are exhaustive techniques and heuristics. Exhaustive techniques will try every possible combination of bitwidths, and take the smallest circuit meeting the error bound. Heuristics will instead attempt to make intelligent decisions to reduce the search space for possible bitwidths, while making no guarantee that the proposed solution is the optimal one.

Precision selection attempts to find the minimum fractional bitwidth for each signal while still meeting the error constraints of the system. There are several techniques to do this, and they have many similarities. Each method must have an error model, or a way to compute the error at the output, as discussed in Chapter 4. After measuring the error, an algorithm makes a decision to modify one or more of the bitwidths in the system, and measure the new error. The general process is outlined in Figure 2.2.

Not all algorithms follow this flow. Most notable are exhaustive methods which test all combinations and ensure the optimal results. There are also some algorithms that find the range and precision simultaneously.

5.1 Heuristic Competitions

Bitwidth optimization heuristics often involve competitions [4]. An example of a competition is shown in Figure 5.1. In this example the width of A and B are fixed, so
Algorithm 1 Pseudocode for a competition to reduce bits

```
while system_error_constraint is met do
    for all operator do
        operator.width--;
        error = measureError();
        score = error × costFunction(operator);
        list.add(score,operator);
        operator.width++;
    end for
    winner = list.get(bestScore);
    winner.width--;
end while
```

there are only three bit widths to optimize. Each is temporarily reduced, and the resulting quantization error is measured. This quantization error is then used to compute a cost function which balances the trade off between system error performance and circuit area. The operator that produces the best trade-off wins the round, and has its bit width permanently reduced. The process repeats until the error constraint can no longer be met.

Another way to think about competitions is in the context of the gradient descent algorithm. Gradient descent searches for the minimum point by moving in the direction of the negative gradient according to the equation

\[ b = a - \gamma \Delta F(a) \] (5.1)

where \( \gamma \) is a small number controlling the distance moved, and \( \Delta F(a) \) is the gradient at the current location. When applied to bit width selection, \( \gamma \) is the number of bits to reduce, and the multidimensional gradient determines which operator should be modified. To determine the gradient, the circuit must be tested \( n \) times, once for each operator. The operator that results in the best circuit, as determined by a cost function, will be the winning operator, and the algorithm will move in that direction. The difference between various competitions can be related to differences in the gradient function, starting point, cost function and the value of \( \gamma \).
(a) A competition can begin by finding a uniform word length, and the error measured.

(b) One operator is reduced, and the error measured.

(c) Another operator is reduced, and the error measured.

(d) Another operator is reduced, and the error measured.

(e) The operator introducing the lowest error is chosen, and the competition continues.

Figure 5.1: Example of how a competition round proceeds.
Below is a list of competitions, highlighting their individual unique approaches. Some competitions reduce bitwidths until no more nodes can be reduced without violating the error constraint. Others increase the bitwidth, ending as soon as the system error constraint is not violated. The names of the competitions are reproduced from the originating paper.

5.1.1 Competition Variants

Min +b bit $^1$ As presented in [4], in this competition each operand is tested to find the minimum bitwidth that meets the error specifications while all other operations are arbitrarily long (floating point). Then each operator is set to this minimum width, and the operators compete to increase their length until the error constraint is met.

In terms of gradient descent, we find the minimum value in each dimension, then start from that point. This point will be below the error constraint, so the algorithm moves in the direction of the gradient until the error constraint is met.

Max -1 bit This procedure starts all operators at the maximum allowed fixed-point width. The operators then compete to lose bits until the error constraint can no longer be met [4].

In terms of gradient descent, this algorithm starts at the maximum value, and tests the area of the circuit in every direction, moving with $\gamma = 1$ in the direction of the best area gain.

Evolutive This procedure sets all operators to floating point, then starts by setting one operator to 0 width, and works up until the error constraint is met. The operator is then increased by one bit, and another operator is increased from 0. Because previous operators keep their width when new operators are tested, the order of operators is important. After all the operator widths have been picked, they compete to lose bits while the error constraint is still met [4].

$^1$This algorithm was implemented and is referenced as “Min” in Chapter 7
In terms of gradient descent, the circuit creates an n-dimensional space, where n is the number of bitwidths to optimize. The algorithm moves along the axis in one direction to find the error constraint. It moves a little farther, then moves away from the axis in another dimension. Once the algorithm has moved away from all of the axes, it will use the “Max -1 bit” algorithm to try to reduce the circuit size.

**Hybrid** This procedure first performs “Min +b bit” followed by “Max -1 bit”. Having a different starting point results in a slightly different solution than the Max -1 bit competition [4].

**Heuristic** This procedure first performs “Min +b bit”. Then the width of each operator is increased by one until the error constraint is met. Once the error constraint is met, the operators compete to lose bits as in “Max -1 bit” [4].

**Exhaustive** This is a brute force method. A “Min +b bits” competition is performed first, and all combinations of bits smaller than those found in the competition are pruned from the search space. All remaining combinations are then tested. [4].

**Uniform Bit-Width -1 bit** 2 This procedure begins by finding the optimal uniform bitwidth, which is not an NP-complete problem. From there, each operator has its width increased by at least one bit to increase the search space. Then operators compete to lose one bit at a time [9]. This is very similar to the “Max -1 bit” procedure, except that the search space is first pruned by finding the minimum uniform bitwidth.

**Scaled Uniform Bit-Width -1 bit** 3 This algorithm is similar to the previous algorithm, but the minimum uniform bitwidth is scaled rather than incremented to enlarge the search space. During the competition, each signal bitwidth is reduced until it violates one of the output error constraints. The signal whose reduction provides the best area improvement is chosen to lose one bit. This makes the algorithm less likely to get trapped in a local minimum [10].

---

2 This algorithm was implemented and is referenced as “UBW” in Chapter 7
3 This algorithm was implemented and is referenced as “Scale” in Chapter 7
In terms of gradient descent, the gradient measures the minimum bitwidth that the operator can have while maintaining the error constraint, rather than measuring the area of the circuit with the given operator’s width reduced by a fixed amount. This means that this competition will proceed in a slightly different direction than the previous competition.

**Binary search Uniform Bit-Width -b bits** Building on the “Scaled Uniform Bit-Width -1 bit” procedure, this algorithm performs a binary search for the uniform bitwidth. Then each operator competes, losing b bits at a time [11; 34]. Setting b to a larger number allows the algorithm to search a larger space in less time, with some diminished ability to search the entire space. Setting the value of b is like setting the value of $\gamma$ in terms of gradient descent.

**Simulated Annealing** Simulated annealing can also be used to find a near optimal bitwidth. Rather than always choosing the best candidate configuration, simulated annealing has the option of choosing a worse configuration depending on the temperature parameter. When the algorithm starts, the temperature is high and the algorithm chooses configurations almost randomly. As the algorithm proceeds, the temperature lowers, and the probability that the algorithm will choose a worse solution than the current solution goes down. This gives the algorithm the opportunity to escape from local minima. Simulated annealing was used to find a solution in [30].

### 5.2 Other Techniques

Not all algorithms can be described as a competition. Specifically some optimal techniques do not search the space as a competition, because all combinations are tried. Others do not use a traditional graph based approach to circuit analysis, but rather create a polynomial to model the output error as a function of the bitwidths in the system. The Bitwidth Analysis Framework does not support these other approaches, but they are included here for completeness.
5.2.1 Optimal Techniques

In [14] the authors present a mixed integer linear programming (MILP) formulation to find the true optimal bit width in a system. They formulated the MILP equations, then used an off the shelf MILP solver. Because of the complexity, finding the optimal solution is only useful for extremely small systems. The authors used this optimal technique to create a series of test benches with which to compare their heuristic approaches. In [11] the same authors found that their heuristic (Binary search minimum uniform bitwidth -b bits) did well compared to the optimal solution.

5.2.2 Polynomial Based Technique

The previous techniques are best suited to DSP applications that do not involve division. The authors of [2] argue that in order for a bitwidth optimization technique to target general algorithms, range and precision must be considered together. This is mostly because of the division operator, where small quantization errors can effect the range of the result, especially as the divisor approaches zero. The same authors propose a technique based on results from real algebra discovered by Handelman [24]. By constructing a Generalized Handelman Polynomial, the authors can solve for the quantization error given a set of bit widths. The authors were able to find bounds that are very close to simulation results, and saw significant improvements in size, frequency and latency of the test circuit. However, this technique has only been tested with small circuits and still has much room for improvement before being useful for large real world circuits.

Of the algorithms presented in this chapter, three were implemented in the Bitwidth Analysis Tool, and are discussed in the next chapter. The implemented algorithms are “Minimum Uniform Bitwidth -1 bit”, “Scaled Minimum Uniform Bitwidth -1 bit” and “Min +b”. These algorithms were chosen because they form a representative set of the other algorithms. Many of the other algorithms are either minor variations of these three, or performed poorly in the paper in which they were presented [4]. The following chapter discusses the imple-
mentation of these three algorithms inside the Bitwidth Analysis Tool, which was developed for the purpose of running these and other bitwidth optimization algorithms.
CHAPTER 6. IMPLEMENTATION OF THE BITWIDTH ANALYSIS TOOL

In order to implement and compare some of the algorithms mentioned in the previous chapter, I built a new framework called Bitwidth Analysis Tool (BAT). Due to the lack of available tools for performing bitwidth analysis and suggesting candidate bitwidths, I needed to write my own tool. This tool is now open source\(^1\), so that any others who wish to use or build on this work may do so.

Some of the key features of the tool are its ability to run multiple algorithms at a time, and the modularity that allows the user to mix and match range techniques, error models, cost functions and bitwidth selection algorithms. In addition, the framework has several working examples of bitwidth analysis algorithms, and is built so that new algorithms can be added easily. This allows users to see a detailed, working example of how a bitwidth algorithm works, and allows users who wish to build their own algorithm to focus on the new aspects of algorithms, rather than starting from scratch.

One of the key features of BAT is the ability to run multiple algorithms on the given input. Because of the heuristic nature of these algorithms, no single algorithm gives the best answer in all cases. Having the results from all of the selected algorithms side by side helps select the best set of bitwidths for the given circuit.

Another feature of the tool is its modularity. This allows an engineer to run any or all of the bitwidth selection algorithms with any one of the error models and cost functions. This also makes it easy to add new algorithms. Range algorithms, error models and cost functions are not tied to specific algorithms, which means that a variety of combinations can be tested with relative ease. New algorithms, error models and cost functions can easily

\(^1\)The tool and source are available from SourceForge at http://sourceforge.net/projects/bitanalysis/.
be added and mixed with existing algorithms. This increases the usefulness of the tool to others who wish to build and test their own bitwidth analysis algorithms.

The Bitwidth Analysis Tool (BAT) has two components. The first component is an extension to the Berkley Ptolemy project [17] and creates a new simulation environment designed specifically for performing bitwidth analysis. The second component is a set of bitwidth analysis algorithms built to run in this new framework. Because the tool can be split into these two parts, it is very easy to add new or modified algorithms, error models or cost functions. Figure 6.1 shows how these components build on each other and Ptolemy.

6.1 The Bitwidth Analysis Tool as a Ptolemy Extension

When faced with the decision to build a new tool, I decided to build on an existing framework, so that I could concentrate on the bitwidth algorithms, rather than spending
time debugging a lower level framework. Ptolemy has been in development since 1996, and so it provides a stable and mature framework with very few bugs.

Another candidate platform was Matlab’s Simulink. While stable, Simulink is not as open as Ptolemy, and I would have been unable to customize the framework to the same extent. Ptolemy provides access to graph functions and internal data structures that are unavailable through the Simulink API. Building on Ptolemy also allows anyone to use the tool, not just those who have a current Matlab license.

Ptolemy is an actor and token based framework. Actors are components that interact with other actors in a model. They interact with each other by using tokens. The model is executed by a director. The director tells the actors when they are allowed to produce and consume tokens, and as such the director defines the model of computation used by the model. The existing directors implement a variety of computational models, including synchronous dataflow (SDF), discrete-event (DE) and continuous time models. The use of tokens allow actors to be data-polymorphic [17]. The tokens hold all of the information about datatypes, which allows actors to be used with multiple types of data, even during the same execution run.

The input to BAT is a model described in the native Ptolemy format. The tool takes the circuit through the process described in Figure 2.2. It first finds the range of all of the signals, then runs a variety of precision selection algorithms as specified by the user. The results from these algorithms are then output to a single report file, where the suggested bitwidths and resulting estimated size/cost can easily be compared. The report contains details on the suggested bitwidths, measured output error and error constraints for each algorithm.

A variety of new objects were created to implement this new framework. They include a new director, new tokens, and new actors, each of which will be described in detail.
6.1.1 Bitwidth Director

In order to run the bitwidth selection algorithms, I created a new director by extending the existing SDF director. This allowed me to take advantage of the existing SDF scheduling logic. Because the BitwidthDirector extends the SDF Director, only models that support the SDF model of computation are supported. This means that the model must be able to be statically scheduled, and that actors must produce and consume a fixed number of tokens during each iteration.

To create the bitwidth analysis environment, I overrode the initialize and postfire functions. The initialize function is called each time the user attempts to run the model. The postfire function is called after each of the actors has fired once, as defined in the original SDF Director.

The initialize function reads the list of algorithms to run, starts a new report and setups up all of the necessary data structures to run the algorithms. The new postfire function is called after each iteration, and tests to see if the algorithm has finished. This is the major difference between the existing director and the new director. Rather than run a specific number of iteration, the BitwidthDirector runs enough iteration to finish all of the bitwidth selection algorithms. None of the other scheduling logic has been modified.

The BitwidthDirector has no knowledge of the bitwidth algorithm. It simply knows whether the algorithms has finished or not. Once an algorithms has finished, the bitwidth director saves the current state and moves on to the next algorithm. In order to move from one algorithm to another, the BitwidthDirector resets all of the operator widths and state elements. After the next iteration, the BitwidthDirector will query a the algorithm class to analyze the results of the iteration. The algorithm class is responsible for changing the widths of the operators according to the rules of the algorithm. After sufficient iterations, this class will inform the BitwidthDirector that the algorithm has finished, and the director will move on to the next algorithm. If all of the algorithms have been run, then the director
will generate a report using the results from all of the algorithms, then return control to the Ptolemy framework.

6.1.2 Tokens

Actors in Ptolemy communicate with each other by passing tokens. Each operator, or actor, consumes and produces tokens every time it is executed. The Ptolemy framework provides a Token class that can be overridden depending the on the type of data in the system. All of the arithmetic logic is stored in the tokens, so that actors can be data-polymorphic, meaning that an actor can operate an different datatypes. For example, the Addition actor takes two tokens of any type, and calls the token’s addition function. This allows the same actor to be used with integers, floats, arrays, matrices, etc. The token representing each datatype is responsible for implementing the necessary logic and error checking for performing the actual addition.

Tokens can be broken down into several groups, including scalars and vectors. Existing scalar vectors store a single value, and do not easily interact with vector tokens. In order to implement range arithmetic as described in Chapter 3, I needed to implement a new scalar token that stored multiple values. This allows them to interact with other scalars (such as constants) more easily. It also gives me a place to implement the rules of range arithmetic as discussed in Chapter 3. Therefore I implemented new tokens for range and error analysis. Figure 6.2 shows the class hierarchy with both existing Ptolemy classes and the new Token based classes.

Any class which extends Token must override the following functions: `absolute()`, `_add(ScalarToken)`, `_divide(ScalarToken)`, `_multiply(ScalarToken)`, `_subtract(ScalarToken)`, `convert(Token)`, and `toString()`. The function with underscores implement the logic for the given arithmetic function. The convert function allows one token to be losslessly converted to another token type. Therefore integer tokens can be
converted to double tokens, but not vice versa. Some tokens do provide functions for lossy conversion.

### 6.1.2.1 Range Tokens

Range tokens represent the range of values that a signal can assume. All of the arithmetic logic for a given data type is stored inside the token. This allows the same actor to be used in a variety of different systems. In fact there could be integer tokens, floating point tokens and fixed point tokens in the circuit at the same time, and the actors would not need to know about the differences.

In order to implement range arithmetic, I implemented a new RangeToken. Although a range token holds more than one value, it is treated as a scalar and extends the existing ScalarToken. This allows the token to be used with other scalars, such as constants.

RangeTokens for interval arithmetic store the maximum and minimum value, and performs arithmetic as mentioned in Section 3.1. A token for affine arithmetic was also created. In addition to the maximum and minimum values, it stores the value and index of each error term in the affine expression. These values are used to perform arithmetic according
to the rules presented in Section 3.2. I also provided a `convert()` function which accepts any scalar token, as well as 2-element arrays. The function will return a corresponding range token.

### 6.1.2.2 Probabilistic Affine Arithmetic Tokens

To implement tokens for the probabilistic affine arithmetic described in Section 3.2.2 I had to create a derivation based on work presented in [20]. The derivation is presented below.

The probability that the run time value represented by the affine form is greater than the chosen bound needs to be small enough to be considered insignificant. This can be written as

\[
P[X > \gamma] = p
\]  

(6.1)

where \( X \) is the sum of the residuals of the affine form, \( \gamma \) is the bound that I would like to find, and \( p \) is an acceptably small probability that the true value will be outside the bound.

To find the area in the right tail of the Gaussian, I use the Q-function. Using the Q-function, the expression for \( \gamma \) can be obtained by noting

\[
P[X > \gamma] = p
\]

\[
= Q \left( \frac{\gamma - \mu}{\sigma} \right)
\]

\[
Q^{-1}(p) = \frac{\gamma}{\sigma}
\]

\[
Q^{-1}(p)\sigma = \gamma
\]

\[
Q^{-1}(p)\sqrt{\frac{1}{3} \sum a^2} = \gamma.
\]
The standard deviation of the sum of the residuals is found by noting that

\[ X = \sum a_i e_i \quad e_i \sim U(-1, 1) \]

\[ \text{var}(e_i) = \frac{1}{12} (1 - (-1))^2 = \frac{1}{3} \]

\[ \text{var}(X) = \sum \text{var}(a_i e_i) \]

\[ = \sum a_i^2 \text{var}(e_i) \]

\[ = \frac{1}{3} \sum a_i^2 \]

\[ \text{std}(X) = \sqrt{\frac{1}{3} \sum a_i^2}. \]

The probabilistic bound is now \( x_0 \pm \gamma \) rather than \( x_0 \) plus the sum of the residuals as described in Equation 3.3.

6.1.2.3 Error Tokens

The ErrorToken’s primary purpose is to measure the quantization error at each point in the circuit. It holds two RangeTokens; one to track the range of the signal, and the second to track the range of the quantization error. This is necessary because the error is highly dependent of the range of the signals. Although the ErrorToken stores several values, it is still acting as a scalar value, and extends ScalarToken. All of the arithmetic operations were overridden to apply the given operation to find new values for the error. Finding the error requires knowledge of the range, so the range is also calculated.

6.1.2.4 Simulation Error Tokens

In addition to bitwidth analysis algorithms, the BAT framework also has support for simulation. Although a user could use the existing Ptolemy framework to run a simple simulation, the Ptolemy framework lacks support for automatically tracking the maximum
and minimum values that occur during the simulation, as mention in Section 4.2. In order to add this functionality, I created a new Simulation Token.

The new simulation token tracks multiple values. There is both a native Ptolemy fixed-point token and a native double precision floating point token inside the simulation token. At each step, the difference between the double token and the fixed-point token is taken and compared to previous minimum and maximum values. If appropriate, the minimum or maximum value is updated, and the simulation continues. The range of the quantization error at the output is then reported to the director once the simulation has finished. Unlike the other algorithms, which share the range and error tokens, the simulation tokens can only be used with the simulation error model, which is described in more detail in Section 6.2.5.

6.1.3 Actors

In Ptolemy actors are components that interact with other actors in a model. They extend the concept of objects to concurrent computation [17]. In the case of digital signal processing models, actors are most often arithmetic operators such as add and multiply.

While Ptolemy has existing actors for these arithmetic operations, the BAT framework has its own set of new actors. There are two reasons that these new actors were created. The first reason is to calculate the cost function. The second reason is to make it easier to import a circuit from another program. These reasons are detailed below.

In order to calculate the cost of an operator, the size of the inputs and the size of the outputs must be known. Normal actors do not save this information, so new actors had to be introduced that would allow the algorithms to evaluate the cost of each of the operators. Currently ten operators are supported. They include Delay, Add, Multiply, Gain, Modulo, Sign, Constant, Union (Mux), Input and Output. These operators were created by copying existing code for the given operation, and changing the class to extend the BitwidthActor instead. Other operators could be added similarly by copying the existing
Ptolemy implementation, and changing the code to extend the BitwidthActor class, which handles all of the quantization code.

Importing models from other programs is made easier if the new model looks the same as the old model. If the existing Ptolemy actors had been used, there would have needed to be a quantizer inserted after each operator. This quantizer would have been responsible for tracking the bit width of the output of the Ptolemy block. Adding blocks would have changed the layout of the model and made importing circuits much more difficult. Having a separate quantizer would also have made calculating the cost of the operator much more difficult.

When extending existing Ptolemy base classes to implement an actor, only a few functions need to be implemented. The most important are the constructor and fire(). The construction sets up the interface of the actor by adding input and output ports. The fire() method is called by the director when the actor is supposed to execute. This is the method where the actor reads tokens from the inputs, acts on them, and then puts a new token on the output, to be read by the succeeding actor.

6.2 Implementing Bitwidth Algorithms Using BAT

With the BAT framework in place, I implemented three of the bitwidth algorithms mentioned in Chapter 5. The implemented algorithms include Uniform Bit-Width -1 bit, Scaled Uniform Bit-Width -1 bit, and Min +b bits. I also implemented a simple cost function as found in [14]. Each algorithm follows the same general outline described in Figure 2.2. The differences are mainly in the selected set of precisions to test, and the decision of which direction to move. All of the algorithms implemented here assume that the input has been bounded by a previous stage, so that the propagated ranges will never be exceeded.

Figure 6.3 shows the hierarchy for the classes that support the bitwidth selection algorithms. Most of the general algorithmic logic is in the Competition class. The specific rules are stored in subclasses, which control the direction of the competition. This is the
case for the three implemented algorithms, which are all subclasses of the Competition class. The details of the implement are discussed later. Any algorithms that follow the same flow can extend this class and implement the following functions.

```java
void nodeFailed(double systemError) This function tells the algorithm what to do if testing a node at a new bitwidth resulted in the system failing to meeting its error constraint. Typically this function would reset the bitwidth to its former value without calculating the cost of making such a move.

boolean nodePassed(double systemError) This function tells the algorithm what to do if the system passed the error constraint as a result of testing a particular node. Typically this function would save the cost of making such a move so that the algorithm could later select this node. It returns true if the competition is ready to move on to another node, or false to continue testing this node.

boolean runUniformBWSearch() This function indicates whether or not to start the algorithm by performing a search for the minimum uniform bitwidth.

void setNewNode(int currentPrecision) This function returns the size of the node to be tested. This allows algorithms to decrease widths by one to test them, or even to have a variable step size during the algorithm. For algorithms that work up, such as Min, this value might be the current width plus the stepsize. For algorithms that work down, the value might be the current width minus the stepsize.

String getName() Returns the human readable name of the algorithm, used for displaying the algorithm in the reports.

These functions are called by the Competition class as it executes the general flow described in Figure 2.2. In order to implement an algorithm that did not follow this flow, a programmer could extend the PrecisionStrategy class. In this case the following functions would need to be implemented.
Figure 6.3: A simplified UML diagram showing the type hierarchy. New Classes are shown in boxes, existing Ptolemy classes are ovals.

```java
long getDuration() Returns the number of milliseconds since the strategy began execution.

String getName() Returns the human readable name of the algorithm, used for displaying the algorithm in the reports

void initialize(List<BitwidthActor> quantizers) Gives a list of the bitwidth actors in the current system, and indicates that model needs to be reset to run the next algorithm.

Token newInstance(Token t) Transform the given token into the correct type of token to use with this strategy. In most cases this is derived from the chosen error model, but in some cases, such as simulation, the strategy requires a particular type of token to function correctly.

boolean postfire(List<BitwidthActor> quantizers) Informs the strategy that an iteration of the systems has completed. This function tests the outputs for convergence, and if the outputs have converged, save the results and makes decisions about changing bitwidths for future iterations.
```
6.2.1 Minimum Uniform Bitwidth

Finding the minimum uniform bitwidth is not an NP-complete problem, and so an exact solution can be found relatively quickly. The system error as a function of the systems uniform bitwidth is a one dimensional convex space, so an efficient algorithm such as a binary search can be used. For a binary search, all of the operators in the system are set to the maximum width. The error is then measured, the next system bitwidth is chosen according to the rules of a binary search. Once the algorithm has found the smallest bitwidth which meets the error bound, that width is chosen.

6.2.2 Uniform Bitwidth -1 Bit

The “Uniform Bitwidth -1 bit” (UBW) competition is a relatively straightforward competition. It begins by finding the minimum uniform bitwidth, as described above. It then starts in the first box of Figure 6.4 by increasing all of the operators by 2 bits. This enlarges the search space and helps avoid getting trapped in a local minima. It then selects a set of configurations to try. The selected set contains all of the adjacent configurations. Adjacent configurations are those where exactly one operator has a different bitwidth than the current configuration. In this case, each operator will be tested at one less fractional bit than the currently selected width. Each of these is tested, and the configuration that provides the greatest area/cost gain is chosen. This continues until there are no more configurations that pass the error constraint. The last successful configuration is selected as the proposed bitwidth.

6.2.3 Scaled Uniform Bitwidth -1 Bit

This competition was introduced in [10]. It also starts by finding the minimum uniform bitwidth, as before. When moving into the first box in Figure 6.5 the bitwidths are doubled, rather than increased by two. The set of configurations to test is somewhat larger this time. It contains a set of configurations for each operator, where each operator is tested
at all smaller bitwidths while the rest of the operators are held constant. The operator that has the best cost at its lowest valid precision is selected to lose one bit, and the competition is repeated.
6.2.4 Min + b Bits

This competition moves through the general flow twice, one after another, and can be thought of as a two phase competition. Instead of finding the minimum uniform bit width,
it begins on the left side of Figure 6.6 by selecting the maximum uniform precision allowed by the framework (in this case, 32).

The set of configurations to try contains 32 configurations: one operator is tested at all possible bitwidths while all other operators stay at the maximum bitwidth. This is repeated for each operator, and the minimum value for which the circuit meets the error constraint is recorded. After all configurations have been tested, the competition moves to the second phase.

Now each operator has a bitwidth that only passes the error constraint when all of the other operators are at full precision. When all of the operators are assigned the minimum
value found in the first phase the system no longer passes the error constraint. The second phase of the competition then selects adjacent configurations and slowly increases the size of the circuit until the error constraint is met.

6.2.5 Simulation

In addition to the analytic methods described above, the BAT framework supports simulation to measure the range and quantization error. The simulation method implemented here performs a simultaneous double precision floating point and fixed point simulation, and calculates the quantization error by measuring the difference.

The first step of the simulation is to generate the input. The input is a uniform random variable generated based on the range of the input specified by the user. Although a Gaussian random variable might better approximate the expected input to the system, as long as the range of the number is significantly larger than the quantization error, the distribution of the quantization error will not be dependent on the distribution of the input [40]. Simulations were done using both a uniform random variable and a clipped Gaussian random variable. The results were the same, indicating that the range is sufficiently large that the distribution of the input does not impact the quantization error. Using a uniform random variable is slightly more computationally efficient, and was chosen for this implementation.

The current implementation performs the simulation one million times. The maximum and minimum values of the range and the error of each operator are recorded and saved. The number of iterations can be changed to either decrease the runtime or increase the confidence interval indicating that no values will be larger than the simulation values.

6.2.6 Cost Function

The cost function is responsible for estimating the area of each operator. The circuit is assumed to be resource constrained, so that the cost of the routing is negligible. This means that the area estimate is the sum of the area of each operator. Other cost functions
could easily be implemented and used in place of the current one. The current cost function is based on the Xilinx optimized cost function presented in [14], and estimates the area in LUTs for a Xilinx Virtex device. The equation for the area of an adder depends on the precision of the first operator \( n_a \), the second operator \( n_b \), the difference between them \( s \), and the rounded precision of the output \( n_o \). The cost function for an adder is

\[
A = \begin{cases} 
  k_1(n_o + 1) + k_2[\max(n_a - s, n_b) - m - n_o + 1] & \text{if } n_o + m \leq \max(n_a - s, n_b) + 1, \\
  k_1[\max(n_a - s, n_b) - m + 2] & \text{otherwise.} 
\end{cases}
\]

(6.2)

Here \( m \) is a function of the binary point position, and can be expressed as \( m = max(p_a, p_b) + 1 - p_o \), where \( p_a, p_b, \) and \( p_o \) are the binary point locations of the inputs and output respectively. The values \( k_1 \) and \( k_2 \) are constants found through experimentation. For the Xilinx Virtex family the authors of [14] used \( k_1 = 1.0 \) LUTs and \( k_2 = 0.5 \) LUTs.

A cost function for a constant coefficient multiple is more difficult because the area can be highly dependent on the value of the coefficient. However, the authors of [14] used a simpler model that has been demonstrated in practice to provide good results. The proposed cost function for a constant coefficient multiplier is

\[
A = k_3 n_c(n_o + 1) + k_4(n_i + n_c - n_o)
\]

(6.3)

where \( n_i, n_c, \) and \( n_o \) are the widths of the input, coefficient and output respectively. The constants were found to be \( k_3 = 0.60 \) LUTs and \( k_4 = -0.85 \) LUTs. General multiplies were not addressed in [14], since the authors there were only concerned with LTI system. However, I adapted this function to estimate the area of a general multiply by substituting the width of the second operand for the width of the constant.

The Bitwidth Analysis Tool builds on the Ptolemy framework by adding a new director, tokens and actors. These additions in turn allow fast development of bitwidth opti-
mization algorithms, as shown by the implementation of the three algorithms described here. The results from running these implemented algorithms are presented in the next chapter.
CHAPTER 7. COMPETITION AND ERROR MODEL ANALYSIS RESULTS

Several existing algorithms were tested and compared using the Bitwidth Analysis Tool presented in Chapter 6. Range algorithms were compared with each other and with the simulated range. Affine arithmetic did very well, and produced bounds close to the simulation bounds. The implemented bitwidth selection algorithms defined in the previous chapter were also tested. The Binary Search Uniform Bitwidth -1 bit algorithm gave good results, but was not the best in all cases. These bitwidth selection algorithms were compared based on a variety of metrics, including

1. Area (as measured by a cost function)
2. Number of fractional bits
3. Runtime
4. Measured error v. error constraint

The most important metric is area. Improving area is the main purpose of bitwidth optimization algorithms, as an engineer who is not worried about area could simply over-design to meet error requirements. A reduction in area also improves other key factors, such as reduced power and increased speed.

Closely related to the area is the total number of fractional bits. While not a design target per say, it is a good indicator of the area of the circuit and can be used as a sanity check on the area cost function. As such it is more useful for an algorithm designer to check the health of an algorithm than it is as a metric for an end user of the algorithm. We will see that some circuits produce higher fractional bit counts and lower areas, by choosing to reduce the width of high cost operators first.
The runtime of the algorithm is also an important metric. A shorter runtime allows the algorithm to run on larger, more complex systems. However the runtime is often correlated with the quality of the answer. Algorithms that work quickly tend to give poorer results than algorithms that take longer, with the exhaustive search methods reaching the extreme level of taking the longest to run and guaranteeing the best results.

Finally, the estimated error vs. the error constraint shows how close the competition was able to come to the error constraint. The farther the estimated error from the error constraint, the more likely it is that an undiscovered optimization can still be made. On average the algorithms were not able to come as close to the error metric as was expected.

7.1 Previous Work

Although many papers present new or improved algorithms, few of them take the time to compare a variety of different algorithms. This is likely due to the difficulty of implementing a wide variety of algorithms without access to the original author’s code.

One comparison paper that does exist is [4]. In that paper, nine bit width optimization algorithms were compared on a variety of small test benches. The only error model used was simulation to measure the maximum error, as discussed in Section 4.2. The authors compared the nine algorithms based on the total number of fractional bits and number of algorithmic iterations. The author’s results show that no algorithm consistently provided the best solution.

The authors of [4] used a simplified cost function that is not adequate. The simplified cost function is only a function of the output bit width, rather than a function of the width and binary point location of the inputs and the outputs as presented in Section 6.2.6. This simplified cost function could cause the algorithms to choose to reduce the width of lower cost operators instead of higher cost operators, because the cost function would not be able to distinguish them. The comparisons here use the more sophisticated area cost function presented in Section 6.2.6.
The previous paper only uses simulation as an error model. Simulation cannot provide bounds, and often underestimates the maximum quantization error. The comparisons presented here use a variety of analytic error models, which give definite bounds on the maximum quantization error. In addition this thesis presents a comparison of the analytic error models.

The comparisons done here also add algorithms that were not yet published at the time [4] was published. These include the Scale and UBW competitions. Most of the bitwidth optimization algorithms presented in [4] are either extensions of the three implemented here, or performed very poorly; therefore they were not implemented for this comparison.

7.2 Experimental Setup

Interval and affine arithmetic were implemented to measure both the range and the quantization error. Probabilistic affine arithmetic was also implemented to measure quantization error. Three of the precision algorithms mentioned in Section 5.1.1 were implemented: “Min +b bit” (Min), “Binary Search Uniform Bit-Width -1 bit” (UBW), and “Scaled Minimum Uniform Bit-Width -1 bit” (Scale), and were paired with each of the quantization error models. While there are several other precision algorithms, this seems to be a foundational set of the algorithms presented in Section 5.1.1, with most other algorithms being derivatives of these three.

Following each precision algorithm, the circuit was simulated to see how closely the simulated range and error corresponded to the range and error measured by the algorithms. In general the simulation range was close to the measured range, but the simulation error was anywhere between 4x and 40,000x smaller than the estimated error.
7.3 Test Circuits

A variety of test circuits were used in evaluating the range and precision algorithms. They can be split into three categories, feed forward, feedback, and non-LTI systems. Data flow graphs of these circuits can be found in Appendix A.

Feed-forward systems are the simplest, and lend themselves to a wider variety of techniques. The feed-forward systems that were tested are

1. 30 tap FIR filter
2. YUV converter
3. DCT circuit
4. Farrow interpolater

Each feed forward circuit was tested using interval and affine arithmetic to find the range, and under all three precision algorithms using both AA and IA as error metrics.

Feedback systems are circuits in which the output has an impact on future computations. The feedback systems that were tested were a 5-tap and 15-tap IIR filter. Because of the feedback, not all of the range techniques are applicable. Most notably methods using interval arithmetic will fail to produce a result in all but the simplest feedback circuits.

Finally there are non-LTI systems that cannot be characterized by a rational transfer function. Two non-LTI test benches were used in these tests: a timing loop from a BPSK receiver and a four-tap LMS adaptive filter. The non-linear systems were only able to be tested under a subset of the algorithms. The timing loop was tested under all three algorithms, but only using the probabilistic affine arithmetic. The other error models became unstable and the algorithms failed to find a viable solution. The LMS filter was only tested using the UBW algorithm. The other precision algorithms were unable to finish because the system too complex.
7.4 Range Analysis Results

Range analysis using affine arithmetic performs very well. In linear, feed-forward systems, affine arithmetic can give exact bounds. If each input is used only once per output, interval arithmetic can also give exact bounds. The range measurements used here assume that the input has already been bounded in a previous stage.

The YUV and DCT examples show the strength of affine arithmetic over interval arithmetic. In each of these circuits, some of the outputs use an input more than once in the computation. In the YUV example shown in Figure 7.1, the range of each signal is shown in a bubble. Bubbles with a single value indicate that interval arithmetic and affine arithmetic give the same range. For example, the range of the inputs is the same for interval and affine arithmetic, and is \([0, 1]\), or from zero to one inclusive. Once the value of R (or B) is reused to compute Cr (or Cb), IA and AA give different results. The IA value is given first, with the AA value below. The AA ranges are smaller, and in fact give the exact bound.
We can find the exact bound by rearranging the equations so that each input appears only once. As implemented, the equations used for the YUV circuit are

\[ Y = 0.222R + 0.7067G + 0.713B \] (7.1)
\[ Cr = 0.6427(R - Y) \] (7.2)
\[ Cb = 0.5384(B - Y). \] (7.3)

After rearranging, they become

\[ Y = 0.222R + 0.7067G + 0.713B \] (7.4)
\[ Cr = 0.6427R - 0.6427(0.222R + 0.7067G + 0.713B) \] (7.5)
\[ = 0.5R - 0.454G - 0.458B \] (7.6)
\[ Cb = 0.5384B - 0.5384(0.222R + 0.7067G + 0.713B) \] (7.7)
\[ = 0.1545B - 0.119R - 0.3805G. \] (7.8)

By inspection we can see that the minimum value \( Cb \) can assume is -0.5, when B is 0 and R and G are both 1, and the maximum value will be 0.1545 when B is 1 and R and G are 0. Similarly \( Cr \) will reach a maximum of 0.5 when R is one and G and B are zero, and a minimum of -0.912 when R is 0 and G and B are one. These exact bounds are smaller than the bounds achieved by interval arithmetic, but are the same bounds found by affine arithmetic. Such arithmetic manipulation can be used to make circuits interval arithmetic friendly, but doing so in this case would require nine multiplies, instead of the five in the chosen implementation.

In the DCT example we can see that the range given by AA is tighter than that given by IA (as shown in Table 7.1). This table shows each operator’s interval and affine range and compares it against the simulated range. IA and AA give the same range for all of the operators that use each input only once, as can be verified by looking at the DCT diagram.
Table 7.1: Range results for a Discrete Cosine Transform DCT circuit. Here the ranges for AA and IA are compared against the simulation results.

<table>
<thead>
<tr>
<th>Node</th>
<th>Sim Range</th>
<th>IA Range</th>
<th>% Larger than simulation</th>
<th>AA Range</th>
<th>% Larger than simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add00</td>
<td>[-1.993,1.992]</td>
<td>[-2.,2.]</td>
<td>0.4%</td>
<td>[-2.,2.]</td>
<td>0.4%</td>
</tr>
<tr>
<td>Add01</td>
<td>[-2.,1.996]</td>
<td>[-2.,2.]</td>
<td>0.1%</td>
<td>[-2.,2.]</td>
<td>0.1%</td>
</tr>
<tr>
<td>Add02</td>
<td>[-1.996,1.995]</td>
<td>[-2.,2.]</td>
<td>0.2%</td>
<td>[-2.,2.]</td>
<td>0.2%</td>
</tr>
<tr>
<td>Add03</td>
<td>[-1.997,1.995]</td>
<td>[-2.,2.]</td>
<td>0.2%</td>
<td>[-2.,2.]</td>
<td>0.2%</td>
</tr>
<tr>
<td>Add04</td>
<td>[-1.997,1.995]</td>
<td>[-2.,2.]</td>
<td>0.2%</td>
<td>[-2.,2.]</td>
<td>0.2%</td>
</tr>
<tr>
<td>Add05</td>
<td>[-1.996,1.995]</td>
<td>[-2.,2.]</td>
<td>0.2%</td>
<td>[-2.,2.]</td>
<td>0.2%</td>
</tr>
<tr>
<td>Add06</td>
<td>[-2.,1.996]</td>
<td>[-2.,2.]</td>
<td>0.1%</td>
<td>[-2.,2.]</td>
<td>0.1%</td>
</tr>
<tr>
<td>Add07</td>
<td>[-1.993,1.992]</td>
<td>[-2.,2.]</td>
<td>0.4%</td>
<td>[-2.,2.]</td>
<td>0.4%</td>
</tr>
<tr>
<td>Add08</td>
<td>[-3.899,3.767]</td>
<td>[-4.,4.]</td>
<td>4.4%</td>
<td>[-4.,4.]</td>
<td>4.4%</td>
</tr>
<tr>
<td>Add09</td>
<td>[-3.776,3.779]</td>
<td>[-4.,4.]</td>
<td>5.9%</td>
<td>[-4.,4.]</td>
<td>5.9%</td>
</tr>
<tr>
<td>Add10</td>
<td>[-3.87,3.836]</td>
<td>[-4.,4.]</td>
<td>3.8%</td>
<td>[-4.,4.]</td>
<td>3.8%</td>
</tr>
<tr>
<td>Add11</td>
<td>[-3.756,3.797]</td>
<td>[-4.,4.]</td>
<td>5.9%</td>
<td>[-4.,4.]</td>
<td>5.9%</td>
</tr>
<tr>
<td>Add12</td>
<td>[-3.845,3.832]</td>
<td>[-4.,4.]</td>
<td>4.2%</td>
<td>[-4.,4.]</td>
<td>4.2%</td>
</tr>
<tr>
<td>Add13</td>
<td>[-3.776,3.779]</td>
<td>[-4.,4.]</td>
<td>5.9%</td>
<td>[-4.,4.]</td>
<td>5.9%</td>
</tr>
<tr>
<td>Add14</td>
<td>[-3.846,3.755]</td>
<td>[-4.,4.]</td>
<td>5.2%</td>
<td>[-4.,4.]</td>
<td>5.2%</td>
</tr>
<tr>
<td>Add16</td>
<td>[-6.535,6.377]</td>
<td>[-8.,8.]</td>
<td>23.9%</td>
<td>[-8.,8.]</td>
<td>23.9%</td>
</tr>
<tr>
<td>Add18</td>
<td>[-6.156,6.93]</td>
<td>[-8.,8.]</td>
<td>22.3%</td>
<td>[-8.,8.]</td>
<td>22.3%</td>
</tr>
<tr>
<td>Add22</td>
<td>[-6.156,6.93]</td>
<td>[-8.,8.]</td>
<td>22.3%</td>
<td>[-8.,8.]</td>
<td>22.3%</td>
</tr>
<tr>
<td>Add23</td>
<td>[-6.337,6.606]</td>
<td>[-8.,8.]</td>
<td>23.6%</td>
<td>[-8.,8.]</td>
<td>23.6%</td>
</tr>
<tr>
<td>Add35</td>
<td>[-5.126,4.79]</td>
<td>[-5.89,5.89]</td>
<td>18.8%</td>
<td>[-5.89,5.89]</td>
<td>18.8%</td>
</tr>
<tr>
<td>Add36</td>
<td>[-4.196,4.318]</td>
<td>[-8.288,8.288]</td>
<td>94.7%</td>
<td>[-5.226,5.226]</td>
<td>22.8%</td>
</tr>
<tr>
<td>Add41</td>
<td>[-3.836,3.87]</td>
<td>[-12.,12.]</td>
<td>211.4%</td>
<td>[-4.,4.]</td>
<td>3.8%</td>
</tr>
<tr>
<td>Add43</td>
<td>[-5.471,5.227]</td>
<td>[-7.89,7.89]</td>
<td>47.5%</td>
<td>[-6.359,6.359]</td>
<td>18.9%</td>
</tr>
<tr>
<td>Add45</td>
<td>[-6.364,6.869]</td>
<td>[-7.89,7.89]</td>
<td>19.3%</td>
<td>[-7.89,7.89]</td>
<td>19.3%</td>
</tr>
<tr>
<td>Gain0</td>
<td>[-2.081,2.074]</td>
<td>[-2.165,2.165]</td>
<td>4.2%</td>
<td>[-2.165,2.165]</td>
<td>4.2%</td>
</tr>
<tr>
<td>Gain1</td>
<td>[-2.672,2.672]</td>
<td>[-2.828,2.828]</td>
<td>5.9%</td>
<td>[-2.828,2.828]</td>
<td>5.9%</td>
</tr>
<tr>
<td>Gain2</td>
<td>[-5.026,4.906]</td>
<td>[-5.226,5.226]</td>
<td>5.2%</td>
<td>[-5.226,5.226]</td>
<td>5.2%</td>
</tr>
<tr>
<td>Gain3</td>
<td>[-2.356,2.652]</td>
<td>[-3.062,3.062]</td>
<td>22.3%</td>
<td>[-3.062,3.062]</td>
<td>22.3%</td>
</tr>
<tr>
<td>Gain4</td>
<td>[-4.481,4.671]</td>
<td>[-5.657,5.657]</td>
<td>23.6%</td>
<td>[-5.657,5.657]</td>
<td>23.6%</td>
</tr>
</tbody>
</table>
in Figure A.4. When compared to the range estimated through simulation, affine arithmetic also does very well. The affine arithmetic range is never more than 25% larger than the range measured in simulation, indicating that the given bound is a tight bound.

Table 7.2 highlights those operators which have a different interval and affine arithmetic range. In this case the affine arithmetic range can be up to 3 times smaller than the interval arithmetic range. In the case of the Add41 signal, AA would require two integer bits, while the interval arithmetic value would require 4 integer bits. A total of 8 bits are saved by using AA ranges rather than IA ranges.

Table 7.2: Range results for a Discrete Cosine Transform DCT circuit. Here the ranges for affine arithmetic and interval arithmetic are compared in the cases where they differ.

<table>
<thead>
<tr>
<th>Node</th>
<th>IA Range</th>
<th>Bits</th>
<th>AA Range</th>
<th>Bits</th>
<th>AA fewer bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add36</td>
<td>[-8.288,8.288]</td>
<td>5</td>
<td>[-5.226,5.226]</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Add41</td>
<td>[-12.,12.]</td>
<td>5</td>
<td>[-4.,4.]</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Add43</td>
<td>[-7.89,7.89]</td>
<td>4</td>
<td>[-6.359,6.359]</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Add51</td>
<td>[-16.178,16.178]</td>
<td>6</td>
<td>[-7.89,7.89]</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Add53</td>
<td>[-10.055,10.055]</td>
<td>5</td>
<td>[-6.359,6.359]</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Because affine arithmetic keeps track of variable correlations, it can be used to analyze systems with feedback, as stated in [19; 20]. The affine arithmetic results for range in the three IIR filters is encouraging. The simulation bounds are quite close to the measured bounds, as shown in Table 7.3 The five tap filters are very close, while the simulated range in the 15 tap filter is about 63% of the measured range. This seems to follow that as the system gets more complex, affine arithmetic begins to lose the tightness of its bound.

7.5 Error Analysis

Error analysis is more difficult than range analysis. This is likely due to the fact that overflow errors can be completely eliminated, making the range dependent only on the
Table 7.3: Calculated Range v. Simulation Range. The Simulation range is relatively close to the bound, implying that the bound is tight.

<table>
<thead>
<tr>
<th></th>
<th>Calculated Range</th>
<th>Simulation Range</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>lowpass (5 tap)</td>
<td>±11.3081</td>
<td>[-11.0401, 10.7364]</td>
<td>96.3%</td>
</tr>
<tr>
<td>arma (15 tap)</td>
<td>±2.5248</td>
<td>[-1.6054, 1.5545]</td>
<td>62.6%</td>
</tr>
<tr>
<td>fir (30 tap)</td>
<td>±1.5269</td>
<td>[-1.3951, 1.2934]</td>
<td>88.0%</td>
</tr>
</tbody>
</table>

Table 7.4: Comparison between Calculated Error and Simulation Error. Simulation provides much tighter estimate of the quantization error, but cannot be used as a bound.

<table>
<thead>
<tr>
<th></th>
<th>Feed-forward</th>
<th>Feed-back</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interval Arithmetic</td>
<td>5,372x</td>
<td>n/a</td>
</tr>
<tr>
<td>Affine Arithmetic</td>
<td>112x</td>
<td>7,378x</td>
</tr>
<tr>
<td>Probabilistic Affine Arithmetic</td>
<td>66x</td>
<td>627x</td>
</tr>
</tbody>
</table>

underlying model. Quantization error on the other hand cannot be completely eliminated, so that the error is dependent on the current circuit configuration. This difficulty can be seen when comparing the distance between the measured range and the simulation range with the distance between the measured error and the simulation error. The measured and simulation ranges are much closer than the measured and simulated error.

Table 7.4 shows that the error model bounds are significantly larger than the simulation estimates. While the simulation bounds are not strict bounds, the large discrepancy shows the limitations of current error models in performing bitwidth optimization. Larger bounds on the possible quantization error means that all of the bitwidth selection algorithms will stop too early, leaving a circuit that is larger than necessary.

The results between various error models is expected. Interval arithmetic provides the largest bounds due to it’s inability to track correlations between variables. Probabilistic affine arithmetic provides a tighter bound than affine arithmetic because it takes the affine arithmetic value and applies a confidence interval to reduce the range. The bounds for feed-
(a) Comparison of variation on the 5 tap IIR filter example.

(b) Comparison of variations on the 30 tap FIR filter example.

Figure 7.2: Comparison of variations of the uniform bitwidth heuristic with other heuristics. While variations produce small changes in the results, they do not change significantly when compared to other heuristics.

forward systems are tighter than those for systems with feedback. Although affine arithmetic can track the correlations in feedback systems, it does so with decreased accuracy, especially as the filter’s poles move closer to the unit circle.

7.6 Heuristic Comparisons

Because the algorithms used are heuristics, there is a wide variety of subtle changes that will produce different, somewhat unpredictable results. To verify that small variations to the Minimum Uniform Bitwidth -b bits (UBW) competition would not change the results significantly, I tested several circuits using different starting points. The test points I used were +2-1, +4-2, and +6-3. This means that after finding the minimum uniform bit width, each operator in the circuit has its width increased by 2, 4 or 6 respectively, after which bitwidths are reduced by 1, 2 or 3 to test the new circuit configuration. Figure 7.2 shows these variations compared to the other two heuristics. The y-axis is the area of the circuit as measured in LUTS by the cost function. The graphs show that while these variations do
change the results in an unpredictable manner, the results compared to other competitions with a fundamentally different approach are still consistent.

### 7.6.1 Area Cost Function Results

Table 7.5 shows the area ranking of the competitions for each of the feed-forward testbenches and Table 7.6 show the area ranking of the competition for each of the IIR filters. The area is measured by a cost function optimized for Xilinx Virtex series parts as discussed in Section 6.2.6. Figure 7.3 compares the algorithm for each of the test benches discussed at the beginning of this chapter. It is interesting to note that the FIR filter, Farrow interpolator and DCT systems have one outlier where the algorithm gets stuck in a local minimum and fails to find a good solution.

#### Table 7.5: Ranking of competitions by area cost function for FIR filters. Size is estimated in LUTs for a Xilinx Virtex Part.

<table>
<thead>
<tr>
<th>30 tap FIR Algorithm</th>
<th>Size</th>
<th>Farrow Interpolator Algorithm</th>
<th>Size</th>
<th>YUV Converter Algorithm</th>
<th>Size</th>
<th>DCT Algorithm</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>UBW PAA</td>
<td>1164</td>
<td>Min AA</td>
<td>34</td>
<td>UBW PAA</td>
<td>97.45</td>
<td>UBW PAA</td>
<td>193.3</td>
</tr>
<tr>
<td>Min PAA</td>
<td>1413</td>
<td>Min IA</td>
<td>51.2</td>
<td>Min PAA</td>
<td>97.45</td>
<td>UBW AA</td>
<td>260</td>
</tr>
<tr>
<td>UBW AA</td>
<td>2308</td>
<td>Scaled AA</td>
<td>56.75</td>
<td>UBW AA</td>
<td>122.7</td>
<td>UBW IA</td>
<td>289.85</td>
</tr>
<tr>
<td>Min IA</td>
<td>2792</td>
<td>UBW AA</td>
<td>81.75</td>
<td>Min AA</td>
<td>126.3</td>
<td>Scaled PAA</td>
<td>494.2</td>
</tr>
<tr>
<td>UBW IA</td>
<td>3127</td>
<td>UBW PAA</td>
<td>91.5</td>
<td>Scaled AA</td>
<td>219.9</td>
<td>Scaled AA</td>
<td>663.1</td>
</tr>
<tr>
<td>Scaled AA</td>
<td>4471</td>
<td>Scaled PAA</td>
<td>94.45</td>
<td>Scaled PAA</td>
<td>334.6</td>
<td>Min PAA</td>
<td>693.25</td>
</tr>
<tr>
<td>Scaled IA</td>
<td>4611</td>
<td>UBW IA</td>
<td>137.4</td>
<td>UBW IA</td>
<td>395.4</td>
<td>Scaled IA</td>
<td>1156.5</td>
</tr>
<tr>
<td>Min AA</td>
<td>14949</td>
<td>Scaled IA</td>
<td>143.05</td>
<td>Min IA</td>
<td>405.4</td>
<td>Min IA</td>
<td>1296.4</td>
</tr>
<tr>
<td>Scaled PAA</td>
<td>6E+05</td>
<td>Min PAA</td>
<td>1775.7</td>
<td>Scaled IA</td>
<td>1272.8</td>
<td>Min AA</td>
<td>5219.1</td>
</tr>
</tbody>
</table>

#### Table 7.6: Ranking of competitions by area cost function for IIR filters. Size is estimated in LUTs for a Xilinx Virtex Part.

<table>
<thead>
<tr>
<th>5 tap Lowpass Algorithm</th>
<th>Size</th>
<th>15 tap ARMA Algorithm</th>
<th>Size</th>
<th>5 tap ARMA Algorithm</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>UBW PAA</td>
<td>205.85</td>
<td>UBW PAA</td>
<td>5298.9</td>
<td>UBW PAA</td>
<td>185.7</td>
</tr>
<tr>
<td>Min PAA</td>
<td>233.95</td>
<td>Min PAA</td>
<td>5593</td>
<td>Min PAA</td>
<td>186.35</td>
</tr>
<tr>
<td>UBW AA</td>
<td>397.8</td>
<td>Scaled PAA</td>
<td>5770.6</td>
<td>UBW AA</td>
<td>353.15</td>
</tr>
<tr>
<td>Min AA</td>
<td>577.5</td>
<td>Scaled AA</td>
<td>6393.2</td>
<td>Min AA</td>
<td>502.6</td>
</tr>
<tr>
<td>Scaled PAA</td>
<td>804.8</td>
<td>UBW AA</td>
<td>6795.3</td>
<td>Scaled PAA</td>
<td>793.4</td>
</tr>
<tr>
<td>Scaled AA</td>
<td>1431.2</td>
<td>Min AA</td>
<td>14107</td>
<td>Scaled AA</td>
<td>1419.8</td>
</tr>
</tbody>
</table>
(a) Area for a 5 tap IIR filter

(b) Area for a 15 tap IIR filter

(c) Area for a 30 tap FIR filter

(d) Area for a Farrow Interpolator

(e) Area count for RGB to YUV converter

(f) Area count for Discrete Cosine Transform

Figure 7.3: Area Comparisons for Various Circuits.
The UBW competition seems to do the best in terms of circuit area, which is the main design goal. It gives the circuit with the smallest area in all but one case, and is never the worst. The Min competition makes a strong showing for second place, but also ranks last on some designs. It is interesting to note, that on the Farrow Interpolator the Min competition is ranked both first and last under different error models. This could be due to the small size of the farrow interpolator. The Scale competition is consistently ranked at the bottom, due to the way it explores the space. It tends to get stuck exploring in one direction, so that one operator is reduced but all of the others remain at the maximum size.

7.6.2 Number of Fractional Bits

While the total number of fractional bits is not a design target, it does give a good indication of the algorithm’s performance. This makes the metric more applicable to the algorithm designer than the end user. Although bit count and area are related there is not a one to one correlation, because some operators such as add have a smaller area per bit than operators such as multiply. If an algorithm consistently gives a higher area but lower bit count than another algorithm, then the cost function or the algorithm need to be tuned so that the algorithm will make better decisions about which operator’s bit width to reduce. Table 7.7 and Figure 7.4 show the results for ranking the competitions by the total number of fractional bits for the algorithms and test benches presented in this thesis.

Only the 5-tap IIR filter has the same rankings for area and bit count. For the 30-tap FIR filter, the only competition that is out of place is Scaled PAA, which is ranked 4th by bit count, but 9th by area. Inspection of the suggested bitwidths shows that Scale chose to reduce smaller circuit elements such as constants and adders, but left the multipliers near their maximum precision. The Farrow interpolator design has a completely different ordering. This could be because of the size of the design. It is one of the smallest test designs, and therefore small variations in the suggested bitwidths can produce a drastic change in the ordering of the competitions.
Figure 7.4: Fraction Bit Count Comparisons for Various Circuits.
Table 7.7: Ranking of competitions by Total bit count. Probabilistic affine arithmetic does well, as expected, and the Linear Competition results in the smallest bit size in all but one cases.

<table>
<thead>
<tr>
<th></th>
<th>Lowpass</th>
<th>ARMA</th>
<th>ARMA_15</th>
<th>FIR_30</th>
<th>Farrow</th>
<th>YUV</th>
<th>DCT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>UBW PAA</td>
<td>UBW PAA</td>
<td>UBW PAA</td>
<td>UBW PAA</td>
<td>UBW AA</td>
<td>Min PAA</td>
<td>UBW PAA</td>
</tr>
<tr>
<td>2</td>
<td>Min PAA</td>
<td>Min PAA</td>
<td>Min PAA</td>
<td>Min PAA</td>
<td>UBW PAA</td>
<td>UBW PAA</td>
<td>UBW AA</td>
</tr>
<tr>
<td>3</td>
<td>UBW AA</td>
<td>UBW AA</td>
<td>UBW AA</td>
<td>UBW AA</td>
<td>Min AA</td>
<td>Min AA</td>
<td>UBW IA</td>
</tr>
<tr>
<td>4</td>
<td>Min AA</td>
<td>Min AA</td>
<td>Scaled PAA</td>
<td>Scaled PAA</td>
<td>UBW IA</td>
<td>UBW AA</td>
<td>Scaled PAA</td>
</tr>
<tr>
<td>5</td>
<td>Scaled PAA</td>
<td>Scaled PAA</td>
<td>Scaled AA</td>
<td>Min IA</td>
<td>Scaled IA</td>
<td>Scaled AA</td>
<td>Scaled AA</td>
</tr>
<tr>
<td>6</td>
<td>Scaled AA</td>
<td>Scaled AA</td>
<td>Min AA</td>
<td>UBW IA</td>
<td>Min IA</td>
<td>Scaled PAA</td>
<td>Scaled IA</td>
</tr>
<tr>
<td>7</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Scaled AA</td>
<td>Scaled AA</td>
<td>Min IA</td>
<td>Min AA</td>
</tr>
<tr>
<td>8</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Scaled IA</td>
<td>Scaled PAA</td>
<td>UBW IA</td>
<td>Min PAA</td>
</tr>
<tr>
<td>9</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Min AA</td>
<td>Min PAA</td>
<td>Scaled IA</td>
<td>Min IA</td>
</tr>
</tbody>
</table>

7.6.3 Competition Runtime

The Scale competition did very well in runtimes because it tends to pick one direction rather than continuing to explore the space in multiple directions. This results in the algorithm tending to favor a single operator and reducing that operator’s bitwidth to exhaustion. This results in good run times, but often very bad circuit sizes.

The Min competition was often the slowest competition, occasionally taking an order of magnitude longer than the other competitions. This is most likely because of the direction that this algorithm moves. It finds a set of minimum bitwidths that are below the error constraint then moves up. Because it starts with a circuit that has too much quantization error, the circuit is often unstable, and the outputs take longer to converge for each iteration. In algorithms that move downward to the error constraint, only the last few steps are testing the marginally stable circuits, rather than starting with unstable circuits and searching for a more stable configuration. The run times for both Min and Scale imply that runtime is a poor metric for judging the quality of an algorithm. Poorer algorithms can run faster or slower, so that runtime does not give an indication of the quality of the algorithm. The runtimes are shown in Table 7.8.
Table 7.8: Ranking of competitions by runtime. The Scaled competition seems to be the fastest, which could account for its poor cost performance. There does not seem to be any clear winner among the error models in timing.

<table>
<thead>
<tr>
<th>Type</th>
<th>Measured error vs. Error Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>UBW</td>
<td>57.5%</td>
</tr>
<tr>
<td>Min</td>
<td>51.1%</td>
</tr>
<tr>
<td>Scale</td>
<td>36.2%</td>
</tr>
</tbody>
</table>

7.6.4 Measured Error vs. Error Constraint

Table 7.9: Calculated Error as a percent of the Error constraint. The farther the final circuit is from the error constraint, the more likely it is that there is an undiscovered optimization remaining.

Table 7.9 shows the average difference between the measured error and the error constraint. UBW comes closest to the error constraint, while the Scale competition is the farthest from the error constraint. This implies that the Scale competition leaves undiscovered optimizations that could reduce the size of the circuit by bringing the output error closer to the constraint.
7.6.5 Competition Results

Based on the resulting circuit area and the distance from the error metric, UBW appears to be the best competition. Min seemed to do on par with UBW, except in the DCT circuit, when Min did significantly worse than the other algorithms. This circuit has multiple inputs and outputs, so it is possible that has a negative effect on the algorithm.

Interestingly Min did both the best and the worst in the Farrow interpolator, under different error models. Although UBW seemed to do better in general, the results verify that they are in fact heuristics, and may get trapped in local minima. To get the best estimate, running multiple algorithms is required.

7.7 Application to Non-Linear Systems

The LMS filter and Timing Loop completed a subset of the competition/error model combinations, but have poor results. The error models are not able to adequately measure the quantization error, which causes the bitwidth selection algorithms to make poor decisions. For example, when using the Scale algorithm the quantization error measured through simulation was much higher than the error measured through the error model. Furthermore only the probabilistic affine arithmetic error model worked on these non-linear systems. The other error models became unstable, and the algorithms were not able to finish.

In the BPSK filter the algorithm made seemingly logical choices, except that the timing loop filter is reduced to a single bit. Reducing the filter to this width would cause the circuit to fail. This decision makes sense when considering that the range algorithms search for the steady state values. In the circuit the loop filter is used to initially lock onto an incoming signal. Once the filter has locked, the values coming from the loop filter are incredible small, and it is easy to see how the algorithm could consider this value insignificant. This underscores a severe limitation to the bitwidth selection algorithms. If transient responses are important to the circuits operation, then the bitwidth selection algorithms may give poor results.
Because these algorithms are only useful for finding the steady state behavior of a system, they have limited application to non-linear systems. However, they could be used to find a starting point from which manual bitwidth selection could be continued.
CHAPTER 8. CONCLUSION

This thesis provides a comprehensive overview for those wishing to learn about bitwidth optimization algorithms, and can help a reader choose an initial set of papers for further research. It also gives comparisons of three existing precision selection algorithms, which are compared on a diverse set of test-benches. These algorithms are implemented within the new Bitwidth Analysis Tool framework. The framework provides a starting point for those who wish to test new algorithms or error models, without the added cost of starting a new implementation from scratch.

Although current algorithms and error models do not seem to provide a final solution, they are useful in their current state. For engineers who are more concerned with engineering time than space, these algorithms provide a slightly optimized circuit without much extra time. For engineers who are more worried about space, the suggested bitwidths can provide a starting point for future optimizations. The over estimations of the error might also be characterized and the error constraint modified so that the suggested bitwidth is even closer to an approved simulation bound.

8.1 Bitwidth Analysis Summary

This thesis has consolidated the background information necessary to understand bitwidth analysis. Much of this information is scattered throughout the literature, and can be very difficult to find and understand in a logical order. Such difficult to find concepts include the fact that affine arithmetic can be used in IIR filters, and that most of the bitwidth selection algorithms can be expressed in terms of either a gradient descent algorithm or as a competition. Many recent papers also lack detail on how affine arithmetic works due to a
lack of space and an implied understanding. This paper has provided a detailed description of these concepts to prepare the reader to understand the current literature.

8.2 Bitwidth Analysis Tool

The Bitwidth Analysis Tool presented here provides an open framework for building and testing bitwidth selection algorithms. It is the only tool available for anyone to run bitwidth analysis algorithms on their own circuits. It has several working examples, and is available for others to build on. It is easy to add algorithms, error models or cost functions, and can run and compare multiple algorithms in a single run.

As heuristics, precision selection algorithms sometimes get stuck in local minimum. This turns a usually strong heuristic into a poor one, as seen in the Section 7.6.1 where on four of the six designs at least one of the algorithms performs an order of magnitude worse than the others. This underscores the value of a tool that runs multiple algorithms, such as the Bitwidth Analysis Tool presented in this thesis.

8.3 Bitwidth Selection Algorithm Comparison

The Scale algorithm did the worst, although it was highly acclaimed in the paper that introduced it [10]. This is likely due to the fact that the algorithm was originally used with an error model that measured the power of the quantization noise directly, rather than measuring the maximum value of the error. Further work would need to be done to verify if this is the case.

The Min algorithm takes much longer to run than the other algorithms, without providing much additional benefit. This longer run time is due to the fact that the algorithm works by finding a smaller configuration that does not meet the error constraint, and then working up to find one that does. Working in this direction means that the algorithm tests some unstable configurations of the circuit, and must wait longer for the values to converge than the other algorithms which test fewer unstable configurations.
The UBW algorithm seems to work well overall. It was not the best in all cases, but was in most cases. If only one algorithm could be chosen, this would be the best choice. However, it is important to note that as it is still a heuristic, it cannot always provide the best solution. That is why it is useful to have a tool that will run multiple algorithms, and compare the results.

One observation that was only apparent in the timing loop circuit example is the fact that these algorithms are only useful for finding the steady state behavior of a system. In a system like the timing loop, where the transient response is important, these algorithms will give poor results.

8.4 Future Work

Finding an error model that produces tight bounds on the quantization error is the largest remaining hurdle to widespread use of bitwidth selection algorithms. With a good error model, the existing selection algorithms can do a better job of searching the space and finding a near optimal solution.

Some promising techniques are on the horizon. The Handelman polynomial representation as discussed in Section 5.2 provides a completely new approach to the bitwidth optimization problem. Although it applies to non-linear systems as well as linear systems, it currently lacks the speed necessary to apply to real world problems.

The Bitwidth Analysis Tool is by no means a production tool. It is designed to assist researchers in exploring bitwidth selection algorithms and error models. There is still some work that could be done to improve the tool.

The largest improvement would be to add support for measuring the power of the quantization noise. This could be done either by providing some way to input the transfer function, or by adding transfer function estimation as proposed in [3]. Another improvement would be increasing the speed and efficiency of the program. Such improvements could
include faster and better ways to detect when a circuit is unstable, or the ability to converge faster when circuits are only marginally stable.
References


international conference on Computer-aided design, page 275. IEEE Computer Society, 2003. 2.1, 3.1, 3.2, 3.2.2, 4.1, 6.1.2.2, 7.4


[28] H. Keding. Pain killers for the Fixed-Point design flow, Mar. 2010. 4.2.1


79


APPENDIX A. TEST BENCHES

This appendix contains the model diagrams for the test benches used.

A.1 4-tap FIR Filter

A simple direct form 4 tap FIR filter. It is an averaging filter with all of the coefficients equal to 0.25.

![Model Used for the 4 Tap FIR Filter](image)

Figure A.1: Model Used for the 4 Tap FIR Filter
A.2 30-tap FIR Filter

A similar filter with 30 taps was also used, but is not pictured. It has the same form as the 4-tap filter. The 30-tap filter is a half band filter generated using the Matlab command “fir1(30,.5);”. Its coefficients are shown in Table A.1.

Table A.1: Filter Coefficients for a 30 tap FIR Filter

<table>
<thead>
<tr>
<th>Tap</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-0.001700397</td>
</tr>
<tr>
<td>2</td>
<td>1.76E-18</td>
</tr>
<tr>
<td>3</td>
<td>0.002937332</td>
</tr>
<tr>
<td>4</td>
<td>-3.28E-18</td>
</tr>
<tr>
<td>5</td>
<td>-0.006730091</td>
</tr>
<tr>
<td>6</td>
<td>6.05E-18</td>
</tr>
<tr>
<td>7</td>
<td>0.014093888</td>
</tr>
<tr>
<td>8</td>
<td>-9.60E-18</td>
</tr>
<tr>
<td>9</td>
<td>-0.026785036</td>
</tr>
<tr>
<td>10</td>
<td>1.33E-17</td>
</tr>
<tr>
<td>11</td>
<td>0.049098961</td>
</tr>
<tr>
<td>12</td>
<td>-1.66E-17</td>
</tr>
<tr>
<td>13</td>
<td>-0.096938333</td>
</tr>
<tr>
<td>14</td>
<td>1.87E-17</td>
</tr>
<tr>
<td>15</td>
<td>0.315619563</td>
</tr>
<tr>
<td>16</td>
<td>0.500808227</td>
</tr>
</tbody>
</table>
A.3 IIR Filter

A simple 5 tap IIR filter. It was configured as a low pass filter with a cutoff frequency of $0.1\pi$. The coefficients found using the Matlab command “yulewalk(4,[0 .1 .1 1],[1 1 0 0]);”. The FIR coefficients are \{0.0107, -0.0122, 0.0123, -0.0038, 0.0044\} and the IIR coefficients are \{1.0000, -3.2340, 4.1252, -2.4466, 0.5675\}.

![Model Used for the 5 Tap IIR Filter](image)

- IIR Coe: \{0.0134, 0.0137, 0.0193, 0.0257, 0.0146, 0.0179, 0.0073\}
- FIR Coe: \{1.0000, -3.0743, 4.7558, -4.3599, 2.4835, -0.8111, 0.1178\}

Figure A.2: Model Used for the 5 Tap IIR Filter
A.4 Farrow Interpolator

A second order Farrow Interpolator as found in the BPSK timing loop depicted in Figure A.6. It is used to help synchronize the incoming signal with the local receiver [37].

![Diagram of Farrow Interpolator](image)

Figure A.3: Model Used for the Farrow Interpolator

A.5 Discrete Cosine Transform

Figure A.4 shows a diagram for an eight point DCT. DCT is commonly used in lossy compression algorithms, such as MP3 and JPEG [36]
A.6 LMS Filter

Figure A.5 is a simple four tap LMS adaptive filter [25]. LMS filters are non-linear systems that attempt to mimic an unknown system.

A.7 BPSK Timing Loop

Figure A.6 is a simplified timing loop for a BPSK receiver [37; 35]. The circuit synchronizes the receiver to an incoming BPSK signal.
Figure A.4: Model Used for the DCT Circuit
Figure A.5: Model Used for the LMS Filter
Figure A.6: Model used for the BPSK Timing Loop