Automatic bit-width minimization framework based on information theoretic degradation measures.

Jeppe Græsdal Johansen

May 31, 2012

AALBORG UNIVERSITET

Electronic Systems, ASPI
Institute 8
Fredrik Bajers Vej 7 A1
9220 Aalborg Ø
http://es.aau.dk/

Title:
Automatic bit-width minimization framework based on information theoretic degradation measures.

Project period:
10th semester, 2012

Project group:
1042

Participant:
Jeppe Græsdal Johansen

Supervisor:
Yannick Le Moullec

Scientific consultant:
Alex Aaen Birklykke

Number of copies: 3

Number of pages: 37

Appendices: CD-ROM

Finished 31th of May 2012

Abstract:

Mapping DSP programs on to FPGA platforms requires programmers and designers to have very deep knowledge about the target system, and how it will behave in the final environment.

Since those kinds of platforms often have very tight constraints on both speed requirements and area requirements, lots of experimentation must be performed to get the implementation right.

Some methods have been developed to facilitate this mapping process that rely on simulation of the algorithm and statistics about the errors; however, in very constrained systems surprising results might provide better solutions.

This project investigates the use of information theoretic methods to measure degradations of the output an implementatation, compared to an ideal implementation.

A methodology is proposed which will allow a computer program to perform this mapping from a high-level description of an algorithm, to a tuned high-level description of the same algorithm with the bit-widths optimized.

In the end a test is conducted where a complex non-linear algorithm is mapped using two proposed measures using traditional statistics, and information theoretic metrics. The mapping shows that the optimized implementations perform very close.

Preface

This report documents the work done by group 1042 during their project on the 3th semester project of the master with specialization in Applied Signal Processing and Implementation at Aalborg University and is titled: *Automatic bit-width minimization framework based on information theoretic degradation measures.*.

This report documents the work done to develop a method to solve the problem of transforming a mathematical description of a computer algorithm into a realizable computer implementation. This problem is a significant part of the development cycle, since it directly affects the precision of the realized program, its power usage, and the performance. All of those factors depend on each other, and the programmer responsible for the mapping has to be able to make a satisfactory trade-off while writing the final program. Accuracy effects are hard to estimate, but more important is speed and power usage, which are very hard to predict while developing such programs.
The ever faster growing complexity of communications standards, combined with the reprogrammability and flexibility of FPGAs makes it a very interesting platform to look at. FPGAs can be tuned down to the gate level in all operations, which means that fine-grained tuning of a program to a desired precision is entirely possible, and the effects very clearly visible.
The introduction will go into more detail about this problem, and how it can be solved by traditional means.
The analysis section will explain how calculations are done in digital systems, and how the mapping from continuous signals to discrete signals affect the contents. Further, it will explain measures for determining the similarity between two signals.
The method and specification section will explain the proposed method for mapping an algorithm to a final circuit. It will also explain the domain specific language developed to describe those algorithms.
The implementation section will detail the construction of program that implements this method, and the optimization strategies involved.
The test section shows tests of a number of different common algorithms. Those tests are compared to other methods of mapping the algorithms.
Finally, there will be a conclusion and discussion about the proposed methodology, and the results of the project.

Jeppe Græsdal Johansen

Contents

Part I

Report

1 Introduction

Digital Signal Processing(DSP) is a field that involves processing measured data from the physical world in a computer. This could be everything from sound signals, radio frequency signals, to images from cameras. For all of those the process can be loosely described as illustrated in Figure 1.1. A signal is measured from an analog transducer, which is discretized by an Analog to Digital Converter(ADC). The digital signal is fed to a processing unit which processes the data according to a given algorithm, and gives a result. In the end the result is fed back into the physical world, either as an analog signal from a Digital to Analog Converter(DAC), or some other tangible representation.

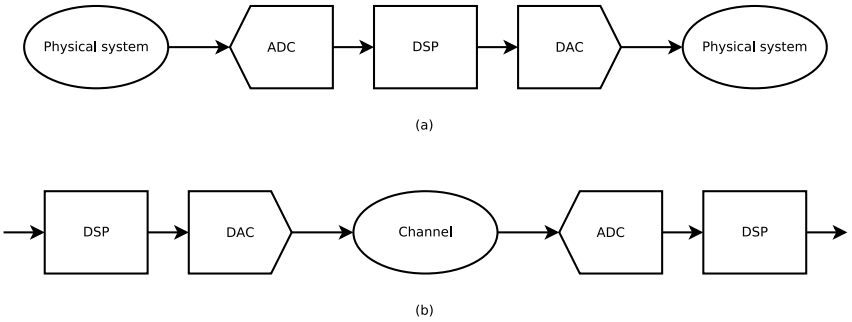

Figure 1.1: Image showing the typical structures of systems that either transform analog signals (a), or communicate through real-world media (b).

When the signal is discretized a certain loss of information occurs since an infinite precision value cannot be represented precisely in digital logic. Binary numbers which are usually used in DSP systems encode decimal numbers in base 2. This means that simply adding a bit to a number will double the information encodable in the value. The same logic could be applied to analog to digical conversion: simply extending the width of the conversion would minimize the loss of information; however, extending the conversion often results in a slower conversion and more expensive hardware. This means that a good trade-off often is desired.

When creating a DSP system the designer usually has access to a set of overall requirements that the final system must fulfill. From there transducers can be selected that match the specific requirements.

The DSP algorithm is then created that can process the data coming from the transducers properly. This is often done in high-level languages such as Matlab or C/C++ on powerful desktop computers. In such environments the available resources in terms of processing power and storage is very high, and allows for very fine precision of calculations. For battery driven devices, such as mobile phones or satelites, such processing power is often infeasible, meaning that the algorithm will have to be transformed from a high-level algorithm into a target platform specific implementation. This can be a trivial task, or it can be extremely complicated depending on the complexity of the algorithm.

Commonly, DSP platforms have been working with the fixed point number representation, which model real numbers as integer numbers implicitly divided by a constant factor. This means that real arithmetic can be performed using integer math, which is very fast in execution time and easy to implement in CMOS Integrated Circuits (IC). Modern desktop computers usually use a real number representation called floating point. This format gives a much wider dynamic range, and esssentially allows the comma to move, hence the name floating point. This kind of real math has a much higher precision since very small numbers can be represented in the same format as very large numbers, however the hardware required to implement such operations usually makes the overall implemen-

tation slower than integer math operations, and require considerably more physical space on an IC.

When using fixed point arithmetic lots of considerations come into play which are not required to the same degree when using floating point. Under most circumstances, when using floating point numbers, a programmer does not have to be aware of possible overflows in intermediate calculations; however, if $2+2$ does not result in 4 but suddenly results in -4 because a number does not allow a safe range to be expressed, then the whole algorithm might fail.

As DSP systems become more and more complex, more resources are required. This has implications both in physical area, but also power consumption in the devices affected. In modern hearing aid devices custom chips are made that must contain advanced signal processing algorithms, but at the same time they should fit in a physically tiny space in the ear together with microphones and speakers which put a rather extreme tanglible constraint on the functionality that can be included in the DSP system. Further, platforms like those also does not have a long battery life, so special considerations should be made towards power consumption efficiency.

For CMOS devices power consumption occurs when a CMOS transistor changes output state (dynamic consumption), and to a smaller extent, when an output is constantly driven low (static consumption.

$$\begin{aligned} P_total &= P_{dynamic} + P_{static} \\ P_{dynamic} &= \tfrac{1}{2} C \cdot f \cdot V^2 \end{aligned} \tag{1.1}$$

Looking at the terms of Equation 1.1 three things can be adjusted. The output capacitance C, the voltage V, or the frequency f. The output capacitance can only be changed by modifying the physical layout of the chip or the design created, which is hard to do as a recursive design process. An output capacitance is generated when the transistor output drives another transistor input. The voltage is a parameter which is interesting to look at, but since lowering this can produce non-deterministic behaviour it is not directly interesting to look at in the scope of deterministic circuits.

Instead of trying to transform a CMOS circuit consisting of highlevel operations, looking at algorithm transformation that reduce the number of transistors required or simply eliminating the switching of certain ranges of transistors could be a method worth looking into. This concept is already used explicitly in lossy data compression, for example used in JPEG image compression, MPEG layer 3 audio compression, etc. If a DSP algorithm is transformed from a floating point description into a fixed point description where the required amount of transistors is minimal, but the loss of information also is minimal, then a more space and power efficient circuit might be able to be constructed.

1.1 The mapping problem

Due to many factors fixed-point arithmetic is often preferred in DSP systems over floating point. The most important are usually the final unit price, power efficiency, and speed [3].

Designing fixed-point applications is harder than designing floating point applications though. Many popular high-level languages, such as C, do not allow a programmer to transparently work with fixed-point arithmetic which can make code very hard to write, and maintenance very complicated. The basic problem is that mapping an algorithm from a description that considers the algorithm as working with infinitely precise real numbers to a finite precision implementation is very hard to automatically do for a compiler.

Many methods have been developed that deal with the problem of mapping integer problems to variably word-sized architectures, such as [7] and [6]. Those methods have shown that optimizing the amount of bits used in all

calculations can give very good results.

In [8] a method is proposed that is based on simulation of algorithms where an optimization process tries to minimize a performance evaluation function while searching for an implementation which complies with a given hardware cost constraint. This is a two-step process where an optimal starting point is also found by simulation. In [6] the method proposed uses a hybrid static analysis and simulation based method to optimize a given algorithm. Instead of simulating a circuit to generate an minimum configuration, a pessimistic estimate is generated by a set of rules which can derive such a solution. This has the implications that the solution will not be able of overflowing in calculations.

In [5] a simulation based mapping methodology is proposed, which uses the same core principle as in [8], but at a mixed algorithm and RTL level, because sharing of functional units was found to be instrumental to a faster mapping of complex systems. Their solution is a two-step procedure that works on both the algorithmic and platform specific level. This approach hampers the flexibility of the overall mapping process, since the distinction between algorithm and implementation is constrained to a single methodology, whereas the method propsed in [8] uses a more target architecture agnostic approach.

1.2 Error metrics

The basic challenge with a simulation based method is that specifying and measuring the acceptable amount of inaccuracy introduced in the system after a mapping is hard to do. Most programming languages do not have any facilities to specify such measures, and introducing them would either mean designing new languages, or changing or doing compile time analysis in existing ones. Neither of those tasks are easy to do which is why many of the simulation based methods have used measures such as Signal to Quantization Noise Ratio (SQNR), Signal to Noise Ratio (SNR), or Mean Square Error (MSE).

Those measures are based on statistics about the signals being transformed from a given set of input data, as illustrated in Figure 1.2. The goal is to use as few bits in a system while keeping the error between an ideal system and the optimized system low enough.

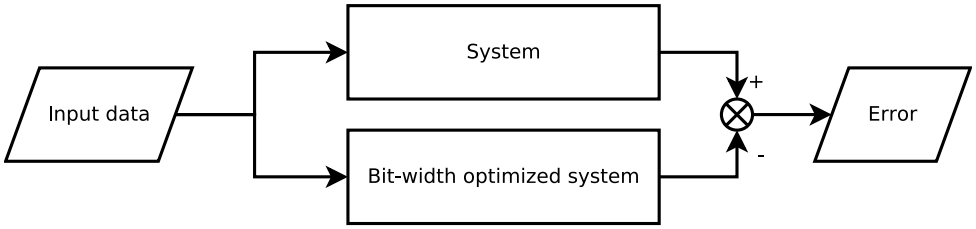

Figure 1.2: An illustration of the general structure of a simulation based bit-width mapping problem. An ideal system is given and a bit-width optimized system is desired. The problem can be constructed as illustrated here such that the two systems are fed the same input, and the error between their outputs is measured.

A method that has not, to the authors knowledge, been applied to this problem is information theory. Information theory is often used in compression algorithms to remove redundant or non-principal components of a signal, such that rare events are removed and only the information that is deemed important remains. It is easy to see how this principle can be applied to the bit-width mapping problem, since an error with a value near infinity might be accepted if it only occurs very rarely. Statistics based measures can easily be swayed by such errors unless directly constructed to handle those situations.

Information based methods have been applied in many fields, including electronics, such as in [1] where boolean

truthtables are mapped into an architecture where they have to be implemented with as few binary multiplexers as possible. They propose a method based on genetic algorithms (GA) optimizing a error metric based on hamming distance and mutual information (MI). They state that their method shown successful results.

1.3 Problem specification

Mapping a high-level algorithm description, to a target specific implementation is a hard task. Especially in very resource constrained systems.

Methods have been proposed that try to automatically map such systems based on simulations and simple optimization; however, most methods proposed base their error functions on linear error metrics.

The problem specification of this project is thus formulated as:

> Can a methodology be developed that can automatically tune a system of equations to use as few bits in intermediate calculations as possible while minimizing degradations of the result? Further, can information theoretic error metrics be used?

1.4 Project scope and limitations

The scope of this project is DSP algorithms on FPGA platforms. The concept behind the methodology could be applicable to DSP platforms too; however, the immediate gains and effects are not as easily found. Working with FPGAs provide an easy measure of benefit from a given degradation in terms of area usage.

The highest priority is to get the methodology working for simple networks of equations, such that a linear set of equations can be tuned, but if possible also for dynamic control and data flow.

2 Analysis

2.1 Number representation

To find out how calculations are affected by different degradations, an analysis of number systems used in discrete systems is performed.

2.1.1 Binary numbers

Binary numbers are a simple way of denoting integer numbers using digital logic. A positive integer x can be denoted in binary as

$$x = \sum_{i=0}^{N-1} y_i 2^i$$

where y_i is the binary digit (bit) of y, and N is the bitwidth of the binary number. When including negative numbers one of two formats are usually used to denote this: one's complement, and more often two's complement.

One's complement is a format where a sign bit is stored together with the digits. The sign bit indicates that a number is positive if 0, and negative if 1. This brings the fortunate situation that a negation is simple to do, simply invert the sign bit. Zero can be stored both as $+0$ and -0, which can be confusing.

The most prevalent format is two's complement. Here a non-zero sign bit indicates that a number is negative, but 1 is subtracted from a negative integer value, so that 0 only can be represented as $+0$. Thus, we will only consider this in the following.

2.1.2 Digital representation of real numbers

Real numbers cannot be expressed exactly in a computer due to the finite amount of resources available, so a number of number notations are usually used to store and process real numbers:

- Fixed-point one's complement
- Fixed-point two's complement
- Floating point

Fixed-point numbers

Fixed-point numbers is an implicit notation which relies on simple integer arithmetic.

A real number x, can be expressed as an integer y when scaled by a constant factor c. This factor is determined by the amount of fractional precision required.

$$y = round(x \cdot c)$$

Where *round* is a function that rounds towards 0.

A normal way of denoting this precision is the Q notation. 1Q15 denotes a 16 bit integer, with 1 bit of quotient information and 15 bits of fraction information. For a 1Q15 number, c would be $2^{15} = 32768$.
A sign bit is often implicitly part of the quotient part, so a 1Q15 number will have 0 bits for an actual quotient part.

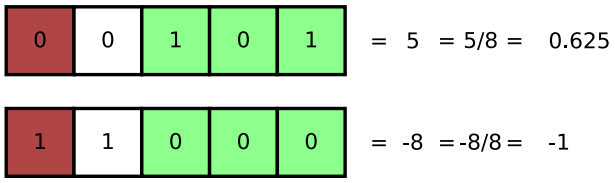

Figure 2.1: Two examples of 2Q3 fixed-point numbers. The red block indicate the sign bit, the white block is the quotient, and the green blocks the fraction.

One can easily see that using this format makes a range of calculations very simple. The expression $x_0 + x_1$, where x_0 and x_1 are two fixed-point numbers with the same format, can be expressed as $y_0 c + y_1 c$, which can be turned into $(y_0 + y_1)\, c$, where the scaling still is implicit. So addition of fixed-point numbers can be done as simple integer additions, as long as the format is the same.

Floating point numbers

Floating point numbers are usually denoted in the form

$$x = s \cdot (1 + m) \cdot 2^{e-c}$$

where s is the sign, m is the mantissa (the fraction), and e is the exponent. c is an unknown constant that varies from format to format. The IEEE 754 standard denotes two commonly used formats that use 32 and 64 bits, respectively in total. Floating point numbers are relatively easy to use in general purpose calculation since they have a large dynamic range, such that the users do not have to worry about whether they are using huge or tiny numbers.

Furthermore, given their construction most formats make it possible to define special constants, such as NaN(Not a number), and $\pm\infty$. However, like one's complement numbers, floating point numbers on this format can also represent ± 0.

When designing DSP systems, floating point numbers are often avoided since they are costly to implement in FPGA fabric, or require calculations in DSPs that are considerably slower than fixed-point calculations unless dedicated hardware to do the calculations is available.

2.1.3 Fixed-point arithmetic

Conversion between fixed-point and real numbers

Positive integers are simple to convert to a fixed-point format, since their fractional part always is 0. The required amount of bits for the quotient for a positive integer x is $round\,(\ln_2(x)) + 1$. For negative integers, an extra sign bit must be added.

Converting real numbers is harder. Numbers such as 0.5 or 0.25 can easily be converted since they can be represented as a sum of integer powers of two (2^{-1} and 2^{-2} respectively). However, a number like 0.3 can only be precisely expressed as an infinite series of powers of two. This has the implication that the designer will have to decide what width is sufficient. On DSP systems with a permanent natural word size, such as 16 bit or 32 bit, it would be straight forward to just give the fractional part the bitwidth of what is left after the quotient part is determined; however, on architectures with no natural bit-width the issue is not as trivial.

For example, with a natural wordsize of 16 bit, 7.3 could be converted to a $3Q13$ format. This would give an actual value of 7.300048828125.

Addition and subtraction

As explained previously, fixed point numbers can be added and subtracted from each other as long as they are stored in the same format. If they are not in the same format some transformation will have to be performed first. Further, it must be observed that to prevent arithmetic overflows some considerations to bitwidths must be observed.

The addition of two fixed-point numbers x and y in format q_xQf_x and q_yQf_y can be written as $round\left(x \cdot 2^{f_x}\right) + round\left(y \cdot 2^{f_y}\right)$. To calculate this with a fixed point result the two numbers are adjusted to a common fractional width, the largest of the two:

$$\begin{aligned}
round\left(result \cdot 2^{f_x}\right) &= round\left(x \cdot 2^{f_x}\right) + round\left(y \cdot 2^{f_y}\right) \cdot 2^{f_x - f_y} && f_x > f_y \\
round\left(result \cdot 2^{f_x}\right) &= round\left(x \cdot 2^{f_x}\right) + round\left(y \cdot 2^{f_y}\right) && f_x = f_y \\
round\left(result \cdot 2^{f_y}\right) &= round\left(x \cdot 2^{f_x}\right) \cdot 2^{f_y - f_x} + round\left(y \cdot 2^{f_y}\right) && f_x < f_y
\end{aligned}$$

The same is true for subtraction.

Quotient bits (Overflow possible)	$max(q_x, q_y)$
Quotient bits (Overflow impossible)	$max(q_x, q_y) + 1$
Fractional bits	$max(f_x, f_y)$

Table 2.1: Required amount of bits as a result of additions and subtractions, to avoid loss of information.

Table 2.1 shows how many bits are required for addition and subtraction operations. The quotient is listed twice: with the possibility of overflow and without. This is important to remember, because given two integers with a similar discrete range of -10 to 10, the largest values possible for the sum or difference of those is -20 and 20, which of course cannot be represented in the previous range.

Overflow can have unforeseen consequences for the results of an algorithm, which is why the possibility of it should be avoided unless explicitly sought after. In the case where it is the desired behaviour a modulo operator, or remainder operation can be applied.

Multiplication

A multiplication of two numbers x and y is expressed as:

$$\begin{aligned}
round\left(result \cdot 2^{f_{result}}\right) &= round\left(x \cdot 2^{f_x}\right) \cdot round\left(y \cdot 2^{f_y}\right) \\
&= round\left(x \cdot 2^{f_x} \cdot y \cdot 2^{f_y}\right) \\
&= round\left(x \cdot y \cdot 2^{f_x + f_y}\right)
\end{aligned}$$

Quotient bits	$q_x + q_y$
Fractional bits	$f_x + f_y$

Table 2.2: Required amount of bits as a result of multiplication.

Comparison

Comparisons is a binary operation that yields a boolean result. Two types of comparisons are looked at.

The first is a similarity comparison. Either the $=$ or the $\neq$ operator. Those can be implemented simply by looking at the bits of two values. Either they are all the same, or some of them are different. In the case where two different Q-formats are compared, the number with the shorter quotient width is sign-extended[1], and a number with a shorter fractional part has zeros appended.

The other type of comparison operation is domain tests, such as $<$, $\leq$, $>$, and $\geq$. Those can not be implemented just by testing bits except for trivial cases; however, they can be implemented using subtraction.

The comparison $x < y$ can be rewritten as $x - y < 0$. Testing whether a two's complement value is under 0 is only a matter of looking at the sign-bit, which is 1 for all negative values. The same follows for $x \leq y$, which can be rewritten as $x - y \leq 0$, however testing for equality at the same time as testing the sign-bit can be costly in hardware and execution time, so the expression is often transformed into $\overline{x > y}$.

2.1.4 Immediate effects of bit truncation

There are different means of lowering the required amount of bits in a fixed-point number. Either by removing most significant bits, or by removing least significant bits. Those operations give different results, based on how and where they are applied. When removing most significant bits, one can either apply a saturating or an overflow operator.

Saturating operator

Saturation means that when saturating a number to a new given domain, numbers that lie outside that domain are set to the nearest number in the domain.

Overflow operator

An overflow operator will "clamp" the value to the given domain in such a way that a value that is outside is offset by the product of the domain width and the divisor.

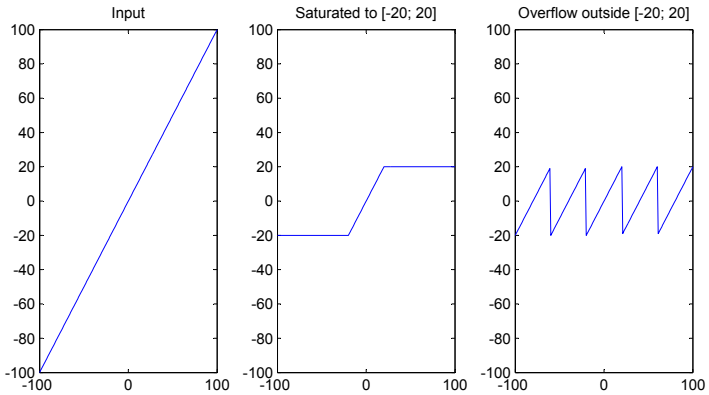

Figure 2.2: An example of a sequence of numbers with a domain of [-100;100] with both saturation and overflow operators applied with a domain of [-20;20].

Saturation is most often used in audio and video processing, where the dynamic range often is known beforehand, and where overflow can result in unpredictable behaviour, or reduced quality.

[1] Prepending the number of required bits, with the same value as the sign-bit

Overflow is more often an undiscovered side effect in systems with a permanent bitwidth; however, some algorithms rely explicitly on overflow, such as in redundancy checks, where a sum is calculated for a large sequence of numbers. One can imagine the sum of the number sequence $4, 2, 4, -6, -4$ in an signed accumulator with a bitwidth of 4. The maximum values that can be stored in the accumulator are -8 to 7. The value of the accumulator with overflow while adding will be $4, 6, -6, 4, 0$, which is what a designer would expect.

General purpose processors do not often have saturated arithmetic since it involves more costly operations that are not used very often. Overflow with powers of two is "free" on most digital systems due to the way binary arithmetic works: bits can simply be set to zero implicitly.

Figure 2.2 shows how those two operators affect a linear input signal, and the error after the operations. The most important part to note is that using the overflow operation, even though the result looks jaggy, the errors remain piecewise constant, and are multiples of the overflow boundaries.

Least significant bit truncation

When cutting off bits from the fraction of a number the range of the number is preserved, while the precision of the number is reduced.

One can either add noise or simply truncate the bits to 0. Adding noise can be beneficial in systems that present visual data, since reduction in a color range lead to visible transistions between colors; an effect known as banding. On the other hand, just truncating an integer value towards 0 introduces an error, as denoted in Equation 2.1.

$$\begin{aligned} x_{trunc} &= \frac{round\left(x \cdot 2^b\right)}{2^b} \\ e_x &= x - x_{trunc} \end{aligned} \tag{2.1}$$

Given a uniformly distributed number over the former domain of x, the distribution of the error e_x after truncation will be a uniformly distributed random value from 0 to $2^b - 1$. For more complex distribution the error distribution is obviously different.

2.2 Calculating implementation costs

To measure what kind of gain one implementation of an algorithm has over another, a cost function must be created.

In this project the logical measure to look at is area. Area is the amount of hardware or instructions required by a given implementation of an algorithm. The area required directly affect other aspects of a complete system. In FPGAs less area would lead to less transistors being able to switch which could mean less power consumed, and also the overall amount of transistors driven would be lower which could potentially lead to higher operational frequencies attainable. In a DSP less memory would be required to hold the data of intermediate results and instructions.

2.2.1 Area cost in FPGAs

When looking at FPGAs they are composed of small cells of functional units. Those are often very limited in use, but when combined they can create very complex logic circuits. Different FPGA manufacturers use different terminology, but most of the general principles behind are the same. The smallest unit in an FPGA, using Xilinx terminology, is called a slice. A slice consists of a LUT, a carry logic unit, and a 1-bit register. FPGAs from other manufacturers can consist of very different variations of this design, but overall a design like this has traditionally been used [2].

LUTs are used to create simple truth-table logic. Using a 4-input LUT you can create a boolean function of 4 input bits; 16 different combinations. Many LUTs can be chained together to create very complex blocks of functionality. Combined with the carry logic unit LUTs can be chained to create very simple adders and subtracters.

When looking at how much area a design takes up in an FPGA, normally an amount of LUTs or slices is used. In this project the terms LUTs and slices will be used interchangably to refer to a single, whole slice.

2.2.2 Area cost in DSPs

Although DSP systems are not discussed as much in this project, the cost of having many instructions in a DSP program can be detrimental to the execution speed, and memory requirements. Even though some combinations of many instructions can be a lot faster than a single slow instructions, they take up more memory which will strain the bus. Making this tradeoff is often a good decision; however, it requires a lot of knowledge about the system.

A simple example could be multiplication. Very often multiplication is a slightly slower operation than additions or logical shift operations. Multiplications with constant integer factors can be turned into a sum of logical shifts of the input number. An example could be $x \cdot 6$. In this case 6 is the same as $2^2 + 2^1$, so the original multiplication can be written as $x \cdot 2^2 + x \cdot 2^1$ which costs 3 instructions instead of 1. Knowing how to transform such expressions is of course highly dependent on the target DSP used, but in most cases some basic heuristics can be applied.

To simplify the project only FPGAs will be discussed from now on.

2.2.3 Estimating area cost in FPGAs

Estimating how many resources that will actually be required from an FPGA implementation of an algorithm is complicated since it relies on many target-specific parameters. The author of this report has previously worked on a project involving this problem where a method was proposed for a specific FPGA to estimate the costs of high-level DSP algorithms [4]. It was shown that for some designs, which follow a simple model, it was able to estimate the actual resource costs with an error of 6% for LUTs and 25% for hardware multipliers. The way it works is by generalizing all functional operations of an algorithm into a certain type of integer operation, for example: adders, multipliers, multiplexers, etc.

This estimation method was based on simple functions approximations fitted onto resource requirement data recorded from actual mappings of simple test programs onto the specific FPGA architecture using regression. That method was inspired by similar methods developed and tested by others who showed similar results. Since the method is straight forward to implement and very fast, it is chosen for this project. All the different types of operations and their similar cost in LUTs and hardware multipliers can be seen in Table 2.3.

2.3 Degradation measure

To quantify how a system is affected by truncation of bits or overflow a method to calculate or measure the degradation of a signal passed through a system is needed.

This section will investigate different methods of measuring this degradation. Two methods will be investigated, statistical analysis and information theory analysis. u

Operation	LUT cost	HW Multiplier cost
Adder	x	0
Multiplication(HW multiplier)	$\begin{cases} 0 & x <= 18 \\ ceil\left(\frac{x}{18}\right) & x > 18 \end{cases}$	$ceil\left(\frac{x}{18}\right)^2$
Multiplication(LUT based multiplier)	$\frac{x^2}{2}$	0
Multiplexer	$\frac{xy}{2}$	0
Constant comparison(power of two)	$\frac{c}{4}$	0
Compare equality/non-equality	$\frac{x}{2}$	0
Bitwise logic operation	x	0
Absolute value	x	0

Table 2.3: Approximations of LUT and HW multiplier costs of basic arithmethic, boolean, and logic operations on a Spartan-3. Ceil is a function that rounds towards $+\infty$. x is the input bit-width of the corresponding operation, y is the number of multiplexer inputs for a multiplexer, and c is the bit-width of a constant. The table comes from [4].

2.3.1 Statistical analysis

Mean Square Error

Mean square error (MSE) is a function that calculates the error between an estimator function and a desired function. The function is here defined as

$$MSE(X,\hat{X}) = \frac{1}{N}\sum_{i}^{N}(\hat{X}_i - X_i)^2$$

Because the function is only a sum of squares, one of the benefits of this measure is that it runs in linear time and is very fast to compute. This is quite a large benefit in simulation based methods where lots of different solutions has to be tried.

Signal to noise ratio

Signal to noise ratio (SNR) calculates the ratio between the signal and the noise in a signal. This is a more subjective measure since it is often hard or impossible to give a good estimate of the noise-floor in arbitrary signals. An often used metric is $SNR(X) = \frac{A(X)}{A(X_{Noise})}$, where A can be calculated as the root mean square (RMS).

A simple way to define the function for use in degredation measurement is to define the error between the optimized output and the desired output as the noise. That way the definition becomes:

$$SNR(X,\hat{X}) = \frac{RMS(X)}{RMS(X-\hat{X})}$$

.

2.3.2 Information based analysis

Entropy

Information describes how much information that are needed to recognize some event from another. If you, for example, have a very noisy system and need to recognize a signal in that system you need more information than in a noise-less system. The noisy system is said to have a higher entropy.

Shannon's definition of entropy $H(X)$, where X is a sequence of discrete values, is given as:

$$H(X) = -\sum_{x} p(x)log(p(x))$$

where x is each unique value occuring in X, and $p(x)$ is the probability massfunction (pmf) of x occuring in X. For a completely random sequence where $p(x_1) = p(x_2)$ for all combinations of x_1 and x_2 then the sequence would consist of maximum entropy, and is said to convey no information. In the opposite case, if a signal only takes on one value, it is said to contain no entropy.

To calculate the entropy of a given sequence first the pmf must be found. The pmf $p(x)$ of a sequence X is a function, which can be defined as the number of times that a distinct value x occurs in the sequence divided by the total number of elements of the sequence X. One can conclude from this definition that $\sum_x p(x) = 1$ is true, when $X \neq \{\}$.

To measure degradation using entropy is interesting. Figure 2.3 shows a few examples of pmf's of errors. Calculating the entropy of the error can be used can give a measure of how close a signal behaves a desired signal. A signal that looks exactly like a desired signal will have an error entropy of 0. Using error entropy is not a good measure intuitively since a signal that looks nothing like the desired signal might lead to the same error entropy. An easy example could be the case where the error entropy between X and $X + 20$ is calculated. In this case the error is always 20, so the entropy is 0.

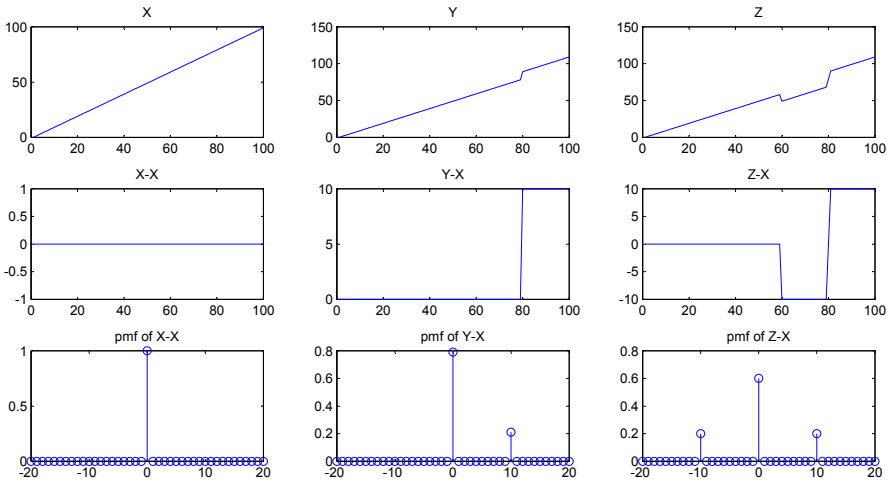

Figure 2.3: Examples of error entropy. The first row shows three sequences of numbers X, Y, and Z. The second row shows the errors between the corresponding function and X. The last row shows the pmf's of the errors above.

Mutual information

Mutual information (MI) gives a measure of how much information is shared between two sequences of discrete values. The function is defined as:

$$I(X;Y) = H(X) + H(Y) - H(X,Y)$$

$H(X,Y)$ is the joint entropy function between two sequences X and Y. It can be seen as finding the entropy of a 2D histogram of the error for a given desired value. The function is defined as:

$$H(X,Y) = -\sum_x \sum_y P(x,y) log(P(x,y))$$

where $P(x,y)$ is the probability of y when x is expected. For two identical inputs $H(X;X) = H(X) + H(X) - H(X,X)$. It can be easily seen that $P(x,y)$ is the same as $p(x)$ only when $x = y$, which means that the joint entropy $H(X,X) = H(X)$ in which case $H(X;X) = H(X)$. One can argue that the highest amount of mutual information between two sequences is the amount of entropy in the sequence with the least amount of entropy.

Mutual information is more interesting than error entropy since it does not just consider the identical error values as the same, but groups them into bins based on the desired value. In this sense it gives a more precise image of the actual error compared to the desired signal. The main disadvantage of this method is that it is considerably harder to calculate, because of the larger number of required bins when deriving the pmf. Also it shares the same traits as error entropy in that the topology of the optimized function does not have to match the desired signal to give an identical amount of mutual information. The same example used for error entropy will also give the same result for mutual information.

However, from a intuitive perspective mutual information can give an even worse result. If one considers two transformations $f(x) = x$ and $g(x) = -x$, then the mutual information between any identical data passed through those will be the same as the information of the input data. An observation seen in line with this is made in [1], which finds that mutual information alone does not lead to any results that necessarily give anything like the desired output when optimizing.

2.3.3 Kullback-Leibler Divergence

Kullback-Leibler divergence (KLD) is another information based method which measures a distance between the distributions of two sequences. This can be used to get a picture of how much two probability mass functions overlap. The KLD is defined as:

$$D_{KL}(X||Y) = \sum_i p(i) log\left(\frac{p(i)}{q(i)}\right))$$

where $p(i)$ and $q(i)$ are the probability mass function of the probability distribution X and Y respectively. The condition $p(i) > 0$ and $q(i) > 0$ must be true for all i where either $p(i) > 0$ or $q(i) > 0$. For two identical probability distributions the result will be $1 \cdot log(1) = 0$.

The KLD measure alone is not symetric meaning it can give a negative result depending on which order the parameters are given. This can give problems when using it to compare two values; however, it can be made symetric using the expression

$$D_{KL}(X||Y) + D_{KL}(Y||X)$$

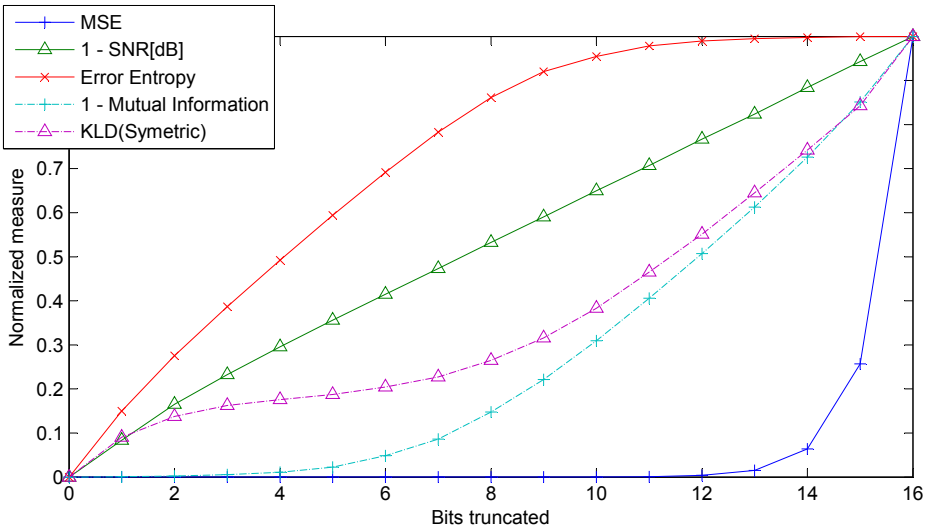

Figure 2.4: A combined graph showing how a number of presented error metrics behave on a bit-truncated signal, where the bottom axis show how many of the least significant bits that are being cut off. The input signal is a 1Q15 uniformly distributed sequence from -1 to 1 with a length of 1024 elements.

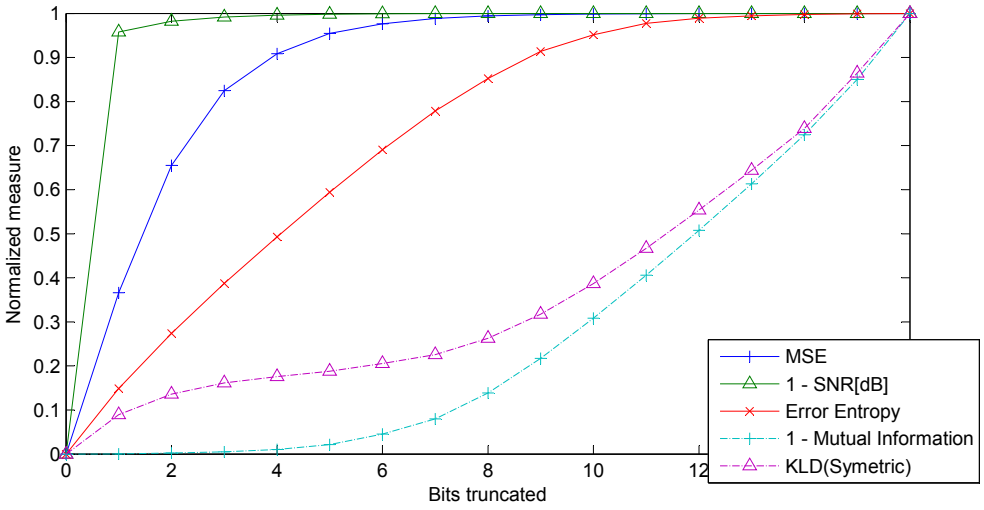

Figure 2.5: A combined graph showing how a number of presented error metrics behave on a signal with an overflow operator applied, where the bottom axis show how many of the most significant bits that are being cut off. The input signal is a 1Q15 uniformly distributed sequence from -1 to 1 with a length of 1024 elements.

As it can be seen in Figure 2.4 and 2.5 the functions based on information theory methods behave in the same way as statistic based methods in the sense that they always have the same general trend. One thing noticable is that the information theory based methods indicate the same overall error measure no matter if overflow or bit truncation is used, whereas the MSE and SNR measures immediately indicate very large errors when bit truncation is used.

2.4 Deriving an optimal solution

Since all parameters in the system are integers it is not possible to directly derive an optimal solution which minimizes the degradation for a given area requirements. This type of problem is an integer programming problem, which means that to find a solution some sort of search algorithm must be employed.

Due to the nature of the calculations in fixed-point arithmetic this problem is not linear and a search that try all possible solutions, an exhaustive search, would grow infeasible very fast depending on how complex the algorithm to be mapped is. To solve such problems combinatorial optimization is often used. In combinatorial optimization a typical objective is to find a solution X that minimizes y in the equation

$$y = f(X)$$

where X is a set of parameters that can be changed. Each combination of X is said to be a solution, but one or more of those are global maximizers. A local maximum does not have any nearby maximums, but it might not be a global maximum.

Two types of combinatorial optimization algorithms are global search algorithms, and local search algorithms. A global search is an algorithm that can potentially discover any global maximum. To do this it must incorporate some mechanism to avoid "getting stuck" in a local maximum. Different algorithms employ different methods to do this, but employing this mechanism typically means that the algorithm is more easily prone to tuning problems. Some parameters might cause the algorithm to not converge fast enough, or some others might cause it to avoid local maximums so much that it never tries to converge.
Another type of algorithm is the local search. This search typically only searches in the nearby area of a given initial solution. Only considering nearby solutions has the benefit that convergence usually is virtually guaranteed,

but it might make it harder to find a global maximum.

3 Methodology and specification

3.1 Introduction

In the problem specification a basic problem was formulated: "Can a methodology be developed that can automatically tune a system of equations to use as few bits in intermediate calculations as possible while minimizing degradations of the result? Further, can information theoretic error metrics be used?".

From this question, concepts were analysed that pertain to the basic challenges behind this problem.

First, two different ways of representing real number arithmetic were explained, concluding that the by far most useful one in a resource constrained performance setting, like FPGAs, is fixed-point arithmetic; however, using fixed-point required the programmer to be aware of a variety of resource allocation rules that complicates the task of working with those.

Secondly, the issue of estimating the implications that different designs have in hardware requirements, is introduced. Estimating those depends entirely on the kind of target architecture that an algorithm is aimed for, and the kind of performance requirements they have to comply with. Different strategies were explained that could be used to estimate those, but none of those are very easy to handle ahead of actually implementing the circuit in code when doing it by hand.

Third, some different methods of actually measuring the loss in accuracy between different implementations of a design are explained. The case is made that analytical analysis of an algorithm can be overly complicated, and that simulated analysis can be used for both linear and non-linear systems.

In conclusion it is seen that tuning a system according to all the possible factors is an combinatorial optimization problem, which simply does not have a trivial solution; however, there exists many different methods to optimize such systems.

3.2 Proposed methodology

A methodology is proposed that binds all of the concept investigated together. A flowchart of the methodology is shown in Figure 3.1.
The process proceeds as follows:

1. An operation graph describing the functional operations of an algorithm is constructed by a computer program from a textual or graphical description.
2. Each node of the graph is tagged with two integer variables, the total bit-width used and the least significant bit of the calculation. Those values are calculated based on the input nodes for the corresponding operation such that no arithmetic overflow or loss of accuracy is possible. The resulting configuration is the initial solution to the following optimization process.
3. According to some optimization strategy the configuration is then updated. The step can change any of the tagged integer variables of the solution.
4. The required amount of resources for the current configuration is estimated according to a target-specific estimation function.
5. If the required amount of resources is less or equal to the allowed amount then the optimization is terminated, and the current configuration is output in a human readable format. Otherwise the process continues to step

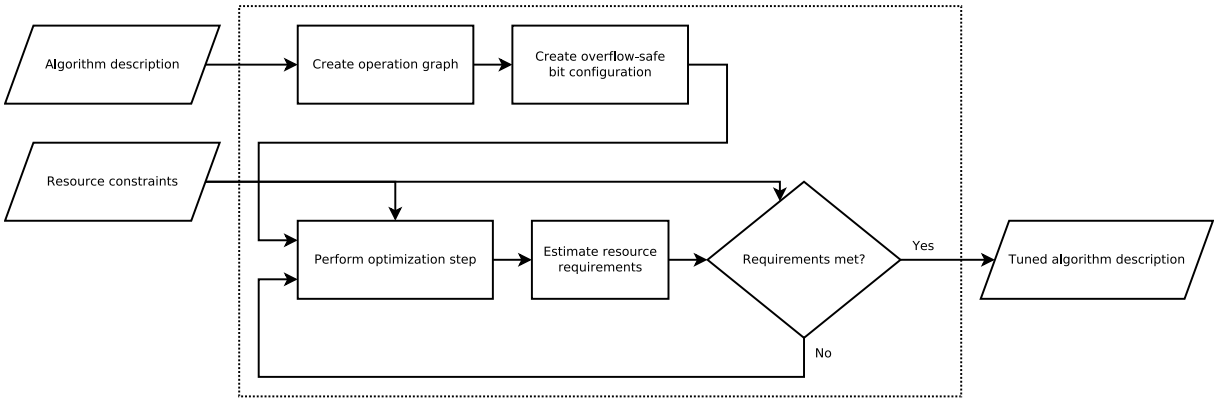

Figure 3.1: The content of the stippled line box comprise the steps of the proposed methodology. The input is constituted of an algorithm description and a target-specific set of resources constraints. The output is a tuned algorithm description, like the input, but annotated with bit allocation information.

3.

3.3 Algorithm dataflow description

The rest of this report considers a system where an algorithm or DSP system is specified as a network of inputs, outputs, arithmetic operations, and intermediate values.

This can be represented as a graph $G = (I, O, V)$, with a set of input vertices I, output vertices O, and arithmetic operation vertices V.

An input vertex is a tuple $< value, q, f >$ of a real value $value$, and a set of integers q and f, describing the amount of quotient and fraction bits, respectively, which specifies how much information the value contains.

An output vertex is a tuple $< q, f >$ of a set of integers q and f, describing the amount of quotient and fraction bits that it can maximally hold, respectively.

The arithmetic operation vertices are tuples $< op, q, f >$. q and f describe the amount of bits like the previous nodes, and op describes what type of operation is performed. This operation can be any of the operations listed in Table 3.1.

$+$ or $-$	A binary addition or subtraction.
$*$	A multiplication.
$=$ or $<$	A conditional selection operator. Will select one of two values based on a comparison of two other values. This means it will have 4 inputs.
$clamp(x)$	A unary overflow operator. Will clamp the input value to a constant real boundary, which must be composable as a power of two.
$trunc(x)$	A unary truncation operator. Will truncate the input value x numbers of LSBs.
$range(x, y)$	A unary operator that combines a clamping and truncation operation in one. Both the resulting bit-width x, and the LSB y is constant.
$Constant$	Node which outputs a real constant. It takes no inputs.

Table 3.1: List of operations allowed in the operation graph.

Using those operations, most highlevel mathematical constructs can be expressed. Figure 3.2 shows an example of how a more complex graph could implement non-linear functions such as an absolute value function using only

those primitives. Even bounded iterative dataflow such as a *while*[1] loop could be modelled using this method, although such constructs would yield very large and complicated graphs.

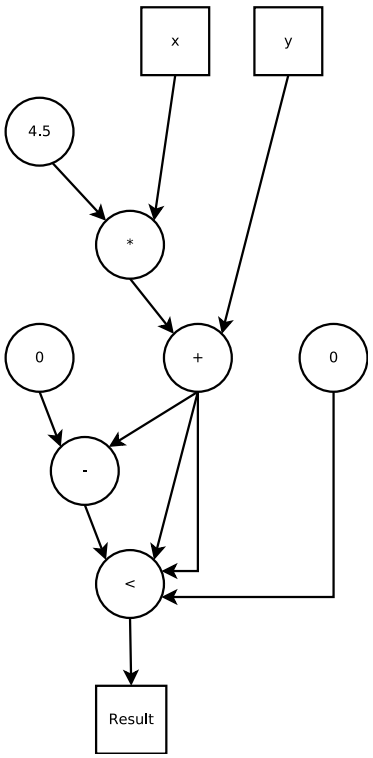

Figure 3.2: An example of a graph, showing an implementation of the mathematical expression $|4,5x+y|$. The selection operator selects $0-(4,5x+y)$ if the condition $4,5x+y<0$ is true, and selects $4,5x+y$ otherwise.

The graph is allowed to be cyclic, as long as some simple requirements are fulfilled. Having cyclic graphs can be used to model time-variant behaviour, such as FIR (finite impulse response) filters. To avoid accuracy problems, a node that is connected to a predecessor node must not have a bit-width that is wider than the predecessors other inputs. This can be ensured using a range operator, which will implicitly redefine the range allowed in the output of a node even though the programmer might know that this range always never will be exceeded. The specific reason for this simple rule will be explained in Section 3.4. To simplify this check all nodes that introduce this form of feedback must be range nodes. If another feedback chain is detected the algorithm should consider the algorithm ill-defined.

3.4 Safe bit-width allocation

To create the initial configuration the a "safe" configuration is calculated. The term safe is used to signify that the implementation will be as accurate as possible, and will not have any unpredictable behaviour that could break the expected dataflow such as unexpected overflow. This further means that adding more resources to the implementation will not create a more accurate implementation, and the solution derived can thus be considered optimal.

The algorithm to derive the safe bit width allocation runs in linear time and is described in Listing 3.1. To determine the safe range of operations a set of simple rules that can be seen in Table 3.2. Those rules are determined based on the analysis of fixed-point arithmetic in Section 2.1.3.

[1] Iterative control flow structure used in most programming languages, which will end, but with no predictable termination point except for a worse-case point.

```
procedure DeriveSafeBitWidths
   For all constant nodes N
      EstimateConstantBitWidth(N);
      Visit(N);
   End for;

   For all range and input nodes N
      Visit(N);
   End for;

   For all unvisited nodes N with all inputs visited
      DeriveBitWidth(N);
      Visit(N);
   End for;

   If unvisited nodes = {} then
      Success
   Else
      Graph ill defined
   End if;
```

Listing 3.1: Algorithm for allocating safe bit-widths for all nodes in a graph. The `DeriveBitWidth` function does a simple look-up in Table 3.2 to determine the resulting bit-width and LSB. The `EstimateConstantBitWidth` function is explained in Listing 3.2.

```
procedure EstimateConstantBitWidth(N)
   If N.Value = 0 then
      N.Width = 0
      N.LSB = 0
   Else
      N.Width = ConstantMaxWidth
      N.LSB = Floor(log2(abs(N.Value)))–N.Width
   End if;
```

Listing 3.2: Algorithm for estimating a "safe" bitwidth of constant nodes. The algorithms strength is that it is very simple, and allocates a constant amount of bits for all constants. For simple constants, such as 2 and -0.5, which could be represented in 2 bits this of course gives a pessimistic estimate, but it should be relatively easy for the optimizer later to simply remove those.

An example illustrating how this algorithm works is shown in Figure 3.3.

Op	**Bits**	**LSB**
$+$	$max(bits_a + lsb_a, bits_b + lsb_b) - min(lsb_a, lsb_b) + 1$	$min(lsb_a, lsb_b)$
$-$	$max(bits_a + lsb_a, bits_b + lsb_b) - min(lsb_a, lsb_b) + 1$	$min(lsb_a, lsb_b)$
$*$	$bits_a + bits_b$	$lsb_a + lsb_b$
$=$ and $<$	$max(bits_a + lsb_a, bits_b + lsb_b) - min(lsb_a, lsb_b)$	$min(lsb_a, lsb_b)$
$clamp(x)$	$Bits$	lsb_a
$trunc(x)$	$bits_a$	y
$range(x,y)$	x	y

Table 3.2: Table showing the amount of bits to allocate for each operation node.

In the case where a feedback loop is created - a nodes output is connected back in the system, such as illustrated in Figure 3.4 - the estimation becomes impossible. In Figure 3.4.a the input to the addition node is the output of the node itself, so determining a safe range becomes an infinitely recursive process, which of course does not have a solution. Instead a range node is introduced as shown in Figure 3.4.b, which limits the possible range, and the system becomes well-defined.

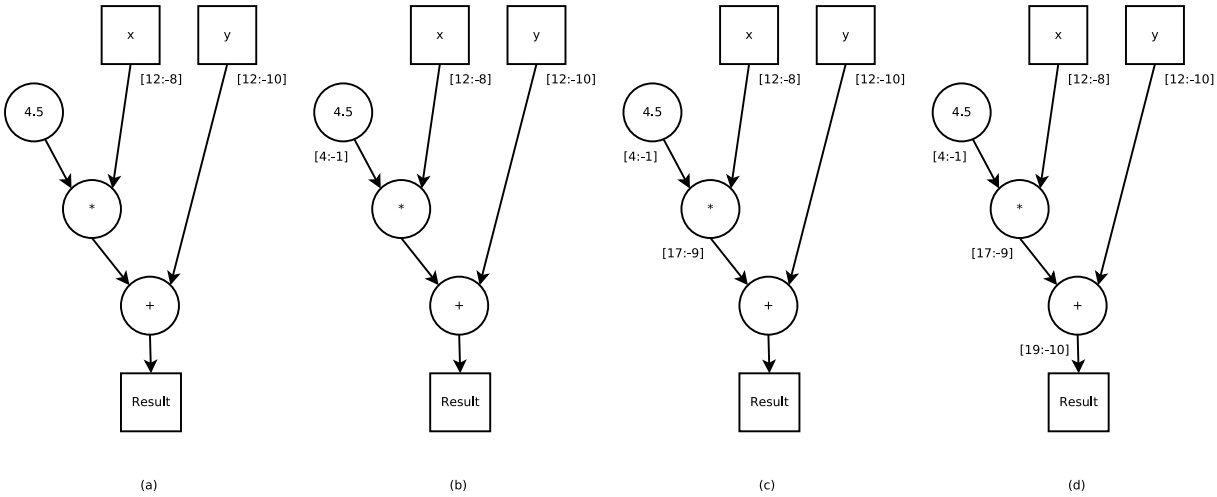

Figure 3.3: A simple example showing the graph for the function $4,5x+y$ in (a). x and y are both 12 bit signed values, with a LSB of -10. In (b) the constant node is visited, and a number of bits is allocated for it, also the two input nodes, x and y, are visited. In (c) the addition is visited, and finally in (d) the multiplication is visited and the algorithm has finished successfully.

3.5 Optimization

When the initial configuration has been derived for a safe configuration, the optimization step starts.

The solution space considered is spanned in two dimensions, hardware requirements and accuracy degradation only, since the topology of the algorithm calculations is always preserved. Because of the topological preservation of the graph the speed performance, in terms of operating frequency, is assumed to always be very close to equal for all possible solutions.

The problem with optimizing these two parameters is that most solutions do not necessarily have any solutions except for the two extremes; i) where a solution has completely degraded the accuracy but use no resources, and ii) the opposite case where the resources are the same as the in the "safe" initial solution and no accuracy is lost.

To solve this problem a method is proposed where only degradation is maximized, but the solution is constrained towards a maximum number of resources. By varying the number of maximum resources from the estimate for the initial solution all the way down to no resources, it is possible to map the best degradations for each amount of resources possible. This can be done both as part of the minimization problem by punishing solutions outside the allowed amount of resources, but also as part of the optimization algorithm itself, depending on the specific strategy implemented.

Two functions to maximize are proposed.

Objective function 1

The first method is based on a traditional method of a combined MSE and SNR measure. The function to maximize is:

$$f_1(y,\hat{y}) = \left(\frac{1}{MSE(y,\hat{y})+1}+1\right)(SNR(y,y-\hat{y})+1)$$

The reason the MSE term is inverted is because a lower MSE value is desired. By adding 1 to each term, the optimization avoids becoming stuck on a sub-optimal solution where one term might become 0.

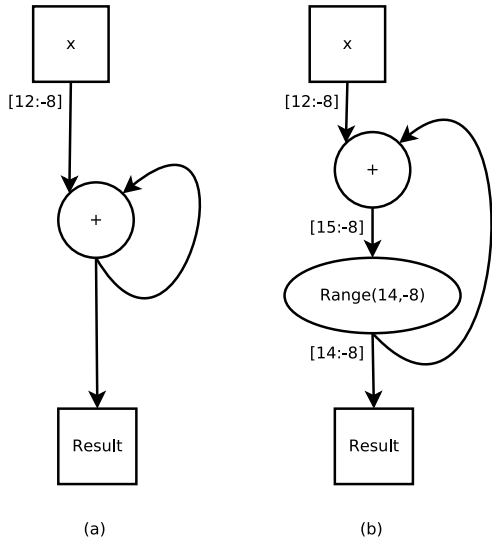

Figure 3.4: Example of feedback in a graph. (a) shows an ill-defined system where the algorithm is not able to determine a safe range, but by introducing a range node in (b) which limits the allowable range of the sum allows the algorithm to determine the worst-case bit-width in the system.

Objective function 2

The second method is based largely on information metrics. It consists of the terms of the mutual information, the error entropy, and the MSE.

$$f_2(y,\hat{y}) = \left(\frac{1}{MSE(y,\hat{y})+1}+1\right)\left(\frac{1}{I(Y,\hat{Y})+1}+1\right)\left(\frac{1}{H(y-\hat{y})+1}+1\right)$$

Here both the error entropy and the MSE is inverted, because both should be minimized. The mutual information should be maximized.

3.6 Termination criteria for optimization

Knowing when to end the optimization can be a good thing. Some optimizations, especially those based on local search, usually converge quickly and steadily towards a solutions and their "optimal" endings will usually be easy to discover. But global search algorithms rely in many cases on randomness, which can be hard to discover. Since it practically is impossible to prove that a solution is optimal three other termination strategies are usually employed. A simple solution is to let the search algorithm run until a human supervisor decides that the algorithm most likely will not converge further.
Another would be to put a time limit on the algorithm. Let it run for a predetermined amount of time, which could be varied depending on the complexity of the algorithm.
Finally a method based on convergence could be applied. Simply calculate how much the solution improved or deteriorated over the last predetermined timeframe, and terminate the optimization process if this falls under some measure.

Those methods all have their pros and cons, which means that they should be chosen to fit each specific problem.

4 Implementation

This section will explain how a solution was created that tries to solve the problem as postulated in section 1.3.

4.1 Overview

To solve the problem, a program is created that takes a high-level algorithm description and turns it into an optimal bit allocated set of code, as shown in figure 4.1.

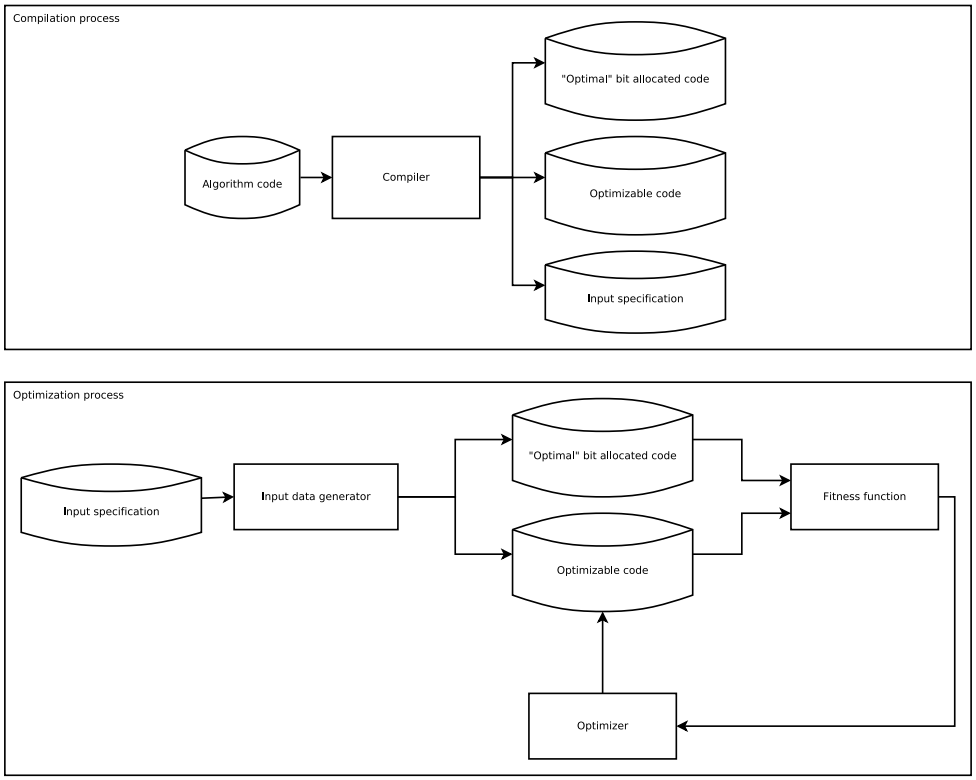

Figure 4.1: The structure of the programs involved in the optimization process.

The entire process consists of two steps:

- The compilation process
- The optimization process

4.1.1 Compilation process

The first step is to process a humanly readable description of the algorithm into a format that a computer program can easily work with. This can be done in a high-level language, and is a well-studied problem.

The result of this process is three descriptions.

First, an optimal description is generated, which describes how the algorithm works under optimal conditions, i.e. when fully precise calculations are performed. This description is used as a baseline for further comparisons, and the algorithm in Section 3.4 explains how to derive this.

Secondly, an optimizable description is generated. This description explains all the intermediate calculations are performed in the algorithm code, but does not assign any particular precision to them.

Third, an input description is made which describe which kind of input the algorithm expects. This could be either a database of samples, or as a description about common distributions, for example normal distributions, uniform distributions, etc.

4.1.2 Optimization process

The optimization process takes the descriptions generated from the compilation process.

The input description is used to generate one or more sequences of input samples. Those sequences are used in the optimization process, either all through the process, or updated between optimization steps.

When the input is generated the optimization process itself starts. This consists of an iterative process where the input data is run through the two sets of code. This results in two sets of output, Y and $\hat{Y}$, where Y is the desired output and $\hat{Y}$ is the degraded output. Those are fed into a fitness function that can give a measure of how degraded the signal is compared to the desired signal.

Given the degradation measure the optimization algorithm can then decide how to tune the optimizable code, and either run a new optimization step, or terminate the process. Those termination conditions vary from algorithm to algorithm, but either the specified amount of optimization has been reached successfully, or it has taken too much time to reach it and the process is deemed not converging.

4.2 Optimization strategies

Three different search algorithms were investigated to test how they perform on various types of systems.

Overall, two types of searches were tried global optimization, and local search.

4.2.1 Simulated annealing

Simulated annealing (SA) is a global optimization technique that is modelled after how metal cools down from a liquid state. The higher the temperature the faster the molecules of the molten metal travels, and is more likely to move further. As the temperature falls the molecules travel slower and slower until they finally sit still.

Simulated annealing optimizes a single configuration at a time, and for each optimization iteration creates a new configuration based on the current. The new configuration has a single bit or value changed. The fitness function of both configurations are calculated, and if the fitness of the new one is higher then this configuration chosen as the next, otherwise the old one is chosen.

There is however another possibility. The optimization has a temperature parameter which is slowly descending towards zero from some high value. The higher the temperature, the more likely it is that the next configuration will be chosen to be the randomly generated one if the fitness is lower. When the temperature reaches zero only configurations that has a higher fitness value can be chosen.

Having the possibility of chosing a configuration with a worse fitness value means it is possible to avoid local minimums while optimizing.

4.2.2 Genetic algorithm

Another global optimization technique implemented is genetic algorithms. This method is based on how biological systems propagate their species through natural selection and mutation.

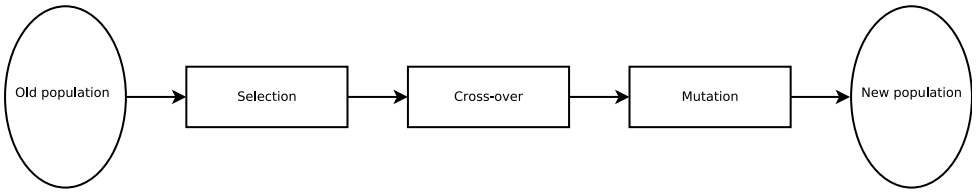

Figure 4.2: An iteration of a genetic algorithm. A set of genomes from the old population is selected based on some set criteria, and some new material might be introduced. Those will then be combined in the crossover process, and finally some mutation might be performed. The resulting genomes are the next generation of the population, and will be subject to the same process next in the next iteration.

A large pool, the genepool, contains a population of individuals, where each individual constitutes a potential solution. For each individual its fitness value is found, given some fitness function.

After this a selection process occurs which transfers a portion of the current genepool to the mating pool. After this there might be added new individuals to the mating pool to keep the number of individuals constant.

From the mating pool the genomes are combined in a crossover operator. This process involves splitting and combining the string of genes in two selected genomes from the mating pool. This produces two sets of offspring that go on to the mutation operation.

When offspring are generated there's a certain probability that a little mutation happens to some of the genes in the genome. This is what happens in the mutation process. After this step the individuals are transferred to new genepool: the next generation.

Those steps can repeat many times until a termination criteria is fulfilled.

4.2.3 Hill climbing

The problem with global search algorithms like genetic algorithms and simulated annealing is that they aren't guaranteed to converge at a very fast speed, or at all, towards an acceptable solution.

Hill climbing is an example of a local search that will always converge towards a local optimum.

The way it works is that it, just like simulated annealing, starts out with an initial configuration and then for each iteration selects a new configuration. Hill climbing does this in a way that is called breadth first. For a given solution all possible neighbouring solutions are investigated, and the one with the largest fitness value is selected as the new one, until no new solution exists. At that point the optimization can terminate.

Since this method can only find local optima, different methods can be used to do a wider search across the search space. If the system includes memory, such that all visited solutions are stored in a database, the algorithm resembles a tabu search.

Another method that can be used is to start out with many different initial configurations, or accept sub-optimal solutions between iterations.

5 Test

A simple non-linear functions is implemented. This function is described in a complexity that would otherwise require a lot of resources, and is then optimized with respect to the two different degradation measures proposed.

5.1 Exponential function

To calculate the exponential function, $f(x) = e^x$, taylor polynomials can be used to generate a sufficiently precise approximation.

The taylor polynomial for e^x is $\sum_{n=0}^{N} \frac{x^n}{n!}$ where N is the order desired. The higher the order the more precise the approximation becomes; however, more calculations and more resources would be required.

There are many ways this can actually be calculated in a graph, figure 5.1 shows one possible way to implement a third order approximation. The input is defined as a 10 bit random variable, which can take on the values from -1 to $1 - 2^{-9}$ $(0,998046875)$.
The actual version tested uses the same structure of building the resulting graph as seen in figure 5.2, but expands the desired order to 7.

5.1.1 Safe bit allocation

The first step after having designed the graph of the algorithm is to create the initial solution. Following the steps of the algorithm proposed gives the result shown in Figure 5.3. Looking at it one can notice that the sizes grow dramatically fast.
Using the Spartan-3 FPGA resource estimator, and assuming that no hardware multipliers can be used, this design would require 27337 LUTs, which is so many that it would not even fit in the largest available version[1] in the Spartan-3 series [2].

5.1.2 Optimization

The optimization strategy employed for this test is hill-climbing only due to lack of time. Figure 5.4 shows how the optimization iterates using the two different degradation measures.

It is easy to see that the initial solution is overly pessimistic, since there is no noticable degradation in the first part of each optimization run using either methods. The big differences between the two happens when the signal starts to become noticably degraded between iteration 2500 and 3000.

5.1.3 Results

The solution was optimized for the two methods using a local search algorithm. This resulted in two solution spaces being generated. Figure 5.5 shows three different local maximums for each optimization method for 50, 100, and 150 LUTs. It shows that both methods attain roughly the same results.

[1] The XC3S5000 with 16640 slices (8320 CLBs).

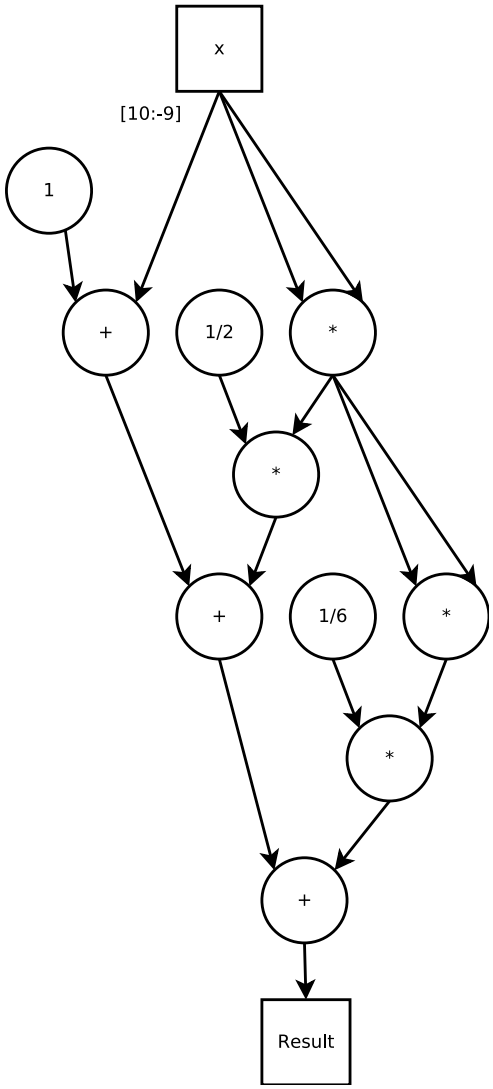

Figure 5.1: Third order exponential function taylor polynomial approximation. The input variable x is shown as having 10 bit resolution with 9 fractional bits.

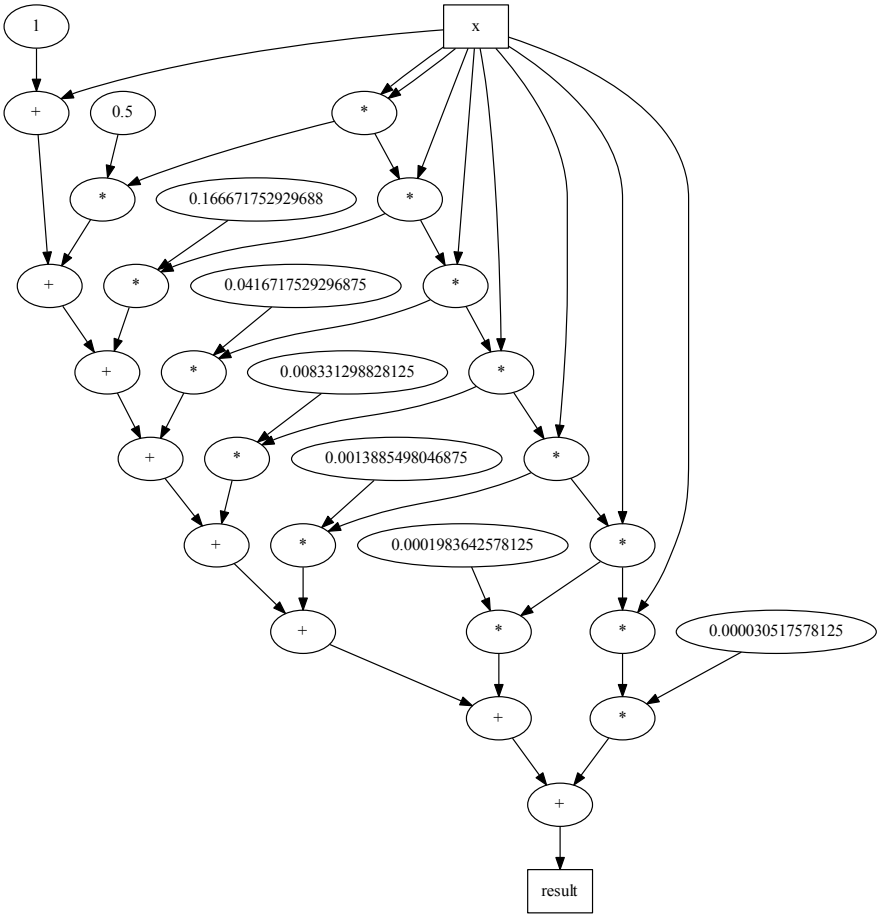

Figure 5.2: 7th order exponential function approximation.

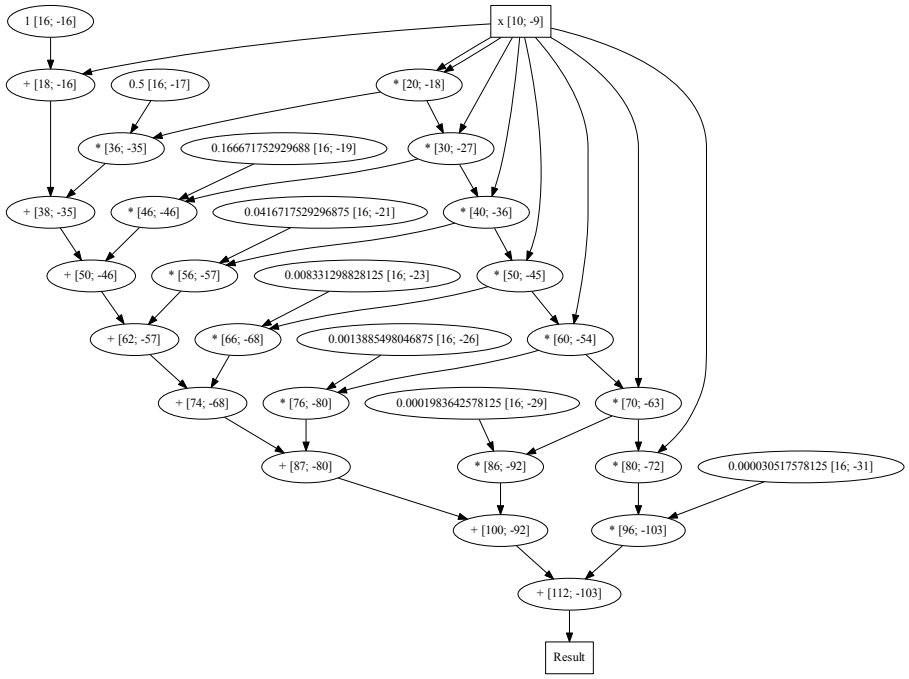

Figure 5.3: Safe bit-width allocation of 7th order exponential function approximation. The [Width; LSB] tag indicate the amount of bits for the fixed-point result of a node. Constructed using a constant maximum bit-width of 16.

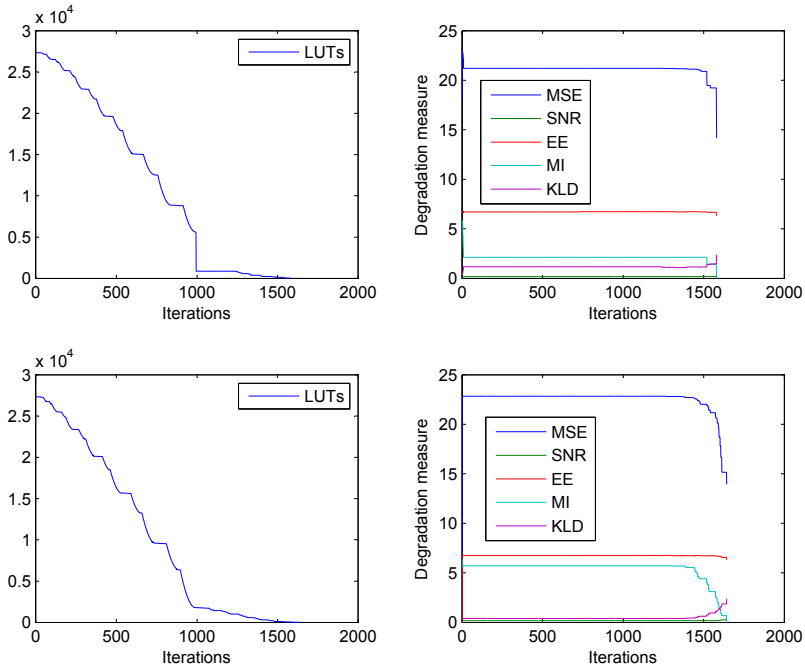

Figure 5.4: The chart shows how each hill-climbing iteration degrades the solution for each optimization method. The two charts on top show method 2 based on information theory, and the two lower ones show the MSE and SNR method.

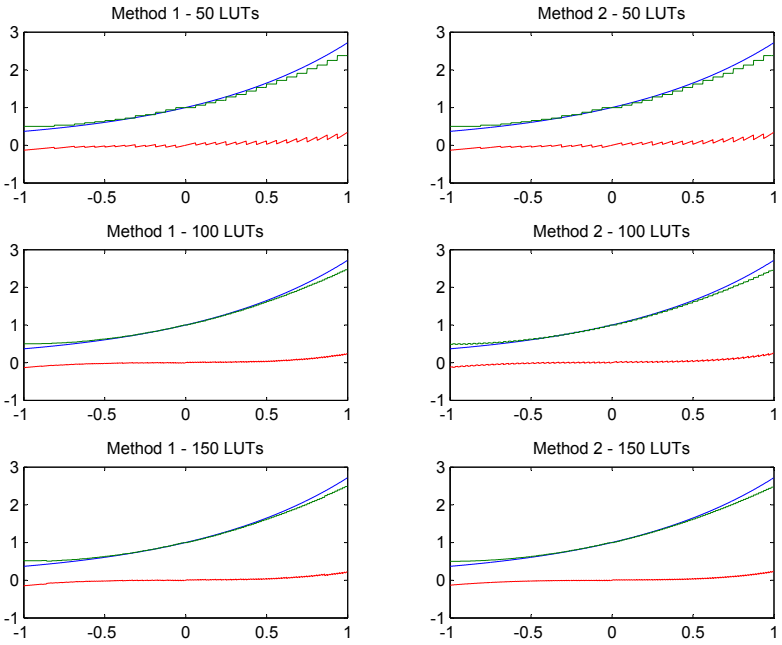

Figure 5.5: This chart shows each local solution for 3 combinations of LUT requirements for each method used.

6 Conclusion

6.1 Discussion

The project started out by asking:

> Can a methodology be developed that can automatically tune a system of equations to use as few bits in intermediate calculations as possible while minimizing degradations of the result? Further, can information theoretic error metrics be used?

To the first question, the answer is yes. Applying both local searches and global search algorithms to the overall problem produced results as expected, just as others have shown before.

Testing out whether information theory metrics could be used in this kind of optimization proved to be a little harder, since the metrics do not necessarily work the way one would expect of a statistical measure. Overall, methods that combined different information theoretic methods proved to give results that looked like those generated using statistical measures; however, alone they had a hard time keeping any part of the "topology" of the original function. Pairing a statistical measure, in this case MSE, with information metrics proved to give interesting results.

An interesting phenomenom that was observed was that combining a statistical measure and an entropy measure such as error entropy often made it possible for a global optimization strategy to convergence faster in very complex graphs. Due to the random and generally slow nature of the optimization algorithm, this was hard to reproduce.

6.2 Future work

Generating an initial solution proved to be a simple task, but the current method is far from optimal. The algorithm generates solutions that will typically be overly pessimistic given the hard requirement of no precision lost. The proposed algorithm only does a forward estimation pass, while a backward pass would make it possibly to indicate unneeded bits. The method chosen to allocate bits for constants is very simple, but suffers from a similar issue. Solving this issue would require extra thought about how constants should actually be represented in the algorithm description given to the compiler.

Due to their computational nature the time it takes to calculate some of the information theoretic metrics is considerably longer due to the histogram based calculations. Compared to the statistics based methods this creates an incentive not to use those methods. As future work investigations into using approximations of those functions could be interesting.

Investigating hybrid optimizations methods would be also be adviced. When mapping very complex graphs the computation time for each iteration grows very quickly, especially when using the information theory methods. Hill-climbing was shown to be a very good method for optimization due to its simple nature. Combining this global searches or multithreading would be trivial to do, and could potentially lead to very big speedups.

The application of this methodology to DSP architectures was also discussed. Many of the architectural properties are the same, yet due to the less flexible nature of DSP platforms this is an even more natural issue on such platforms. Investigating how this methodology performs there could be worth looking into.

6.3 Conclusion

A methodology was proposed that can tune the bit-widths of operations in a DSP algorithm. This methodology was developed based on an analysis of the challenges behind the problem of mapping DSP algorithms from infinite precision floating point descriptions onto an FPGA architecture which uses fixed-point arithmetic.

In the proposed methodology two metrics are proposed to facilitate the automatic mapping of a description. One based on statistical analysis, and one partially based on information theory. Those methods were found to provide a good mapping.

An implementation of the proposed methodology was constructed that could map arbitrarily constructed DSP algorithms to a tuned fixed-point implementation; however, due to lack of time a maximum of 32-bit precision can be used in the mapping process. Expanding this is a doable task, but it would require a good deal more time, and it would further deteriorate the speed of the optimization process.

Using the proposed methodology a complex non-linear DSP algorithm was mapped onto an FPGA architecture. The results showed that the two error metrics proposed gave roughly the same result.

Bibliography

[1] A.H. Aguirre and C.C. Coello. Gate-level synthesis of boolean functions using information theory concepts. In *Computer Science, 2003. ENC 2003. Proceedings of the Fourth Mexican International Conference on*, pages 268 – 275, sept. 2003.

[2] Spartan-3 fpga family data sheet. Datasheet, December 2009.

[3] C. Inacio and D. Ombres. The dsp decision: fixed point or floating? *Spectrum, IEEE*, 33(9):72 –74, sep 1996.

[4] Jeppe G. Johansen. Fpga-oberon - design and implementation of a system description language based on the oberon language. Aalborg University, dec. 2011.

[5] Ki-Il Kum and Wonyong Sung. Combined word-length optimization and high-level synthesis of digital signal processing systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(8):921 –930, aug 2001.

[6] D.-U. Lee, A.A. Gaffar, R.C.C. Cheung, O. Mencer, W. Luk, and G.A. Constantinides. Accuracy-guaranteed bit-width optimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(10):1990 –2000, oct. 2006.

[7] G. Lhairech-Lebreton, P. Coussy, D. Heller, and E. Martin. Bitwidth-aware high-level synthesis for designing low-power dsp applications. In *Electronics, Circuits, and Systems (ICECS), 2010 17th IEEE International Conference on*, pages 531 –534, dec. 2010.

[8] Wonyong Sung and Ki-Il Kum. Simulation-based word-length optimization method for fixed-point digital signal processing systems. *Signal Processing, IEEE Transactions on*, 43(12):3087 –3090, dec 1995.