DIGITAL VLSI DESIGN OPTIMIZATION VIA MACHINE LEARNING

A Design Project Report

Presented to the Engineering Division of the Graduate School of Cornell University

in Partial Fulfillment of the Requirements for the Degree of Master of Engineering, Electrical and Computer Engineering

by

Hengzhi Yao, Lingbo Kou, Shang-Tzu Chen, Shaoke Xu

Project Advisor: Christoph Studer, Zhiru Zhang

Degree Date: May, 2015

Abstract

Master of Electrical Engineering Program

Cornell University

Design Project Report

Project Title: Digital VLSI Design Optimization via Machine Learning

Author: Hengzhi Yao, Lingbo Kou, Shang-Tzu Chen, Shaoke Xu

Abstract:

The overall goal of this project is to use machine-learning algorithms to optimize digital VLSI circuits. Nowadays, VLSI circuit optimization is mostly carried out by humans, which is tedious, time consuming, and expensive. In this project, we aim to use Search and Markov-chain Monte-Carlo (MCMC) methods to automatically determine the key design parameters of digital signal processing circuits, such as the arithmetic precision, in order to minimize human efforts in the circuit optimization process. By doing this, we will be able to automatically perform optimization of digital circuits parameter to aim at the performance that increase the efficiency, lower resource utilization, and reduce costs.

Report Approved by

Project Advisor: Date:

Executive Summary

This project includes the research on various algorithm and implementation of these algorithms for the optimization of fixed point design.

In the first semester, everyone implements the Quantization function, FIR filter, cost function, and plots the relationship between MSE and fractional bits. And then implements Exhaustive Search, Random Walk, Directed Walk I.

In the second semester, Hengzhi does all of the rest of the search methods, including: directed walk II, Simulated Annealing, and Genetic Algorithm. Shaoke, Shang-Tzu and Lingbo worked on the MCMC alogrithm.

Contents

1 Introduction

Virtually all digital application-specific integrated circuits (ASICs) use fixed-point arithmetic to improve throughput, area, and power efficiency, in comparison to floating point-based designs. The basic idea of using fixed-point for efficient application-specific integrated circuits is to adapt the arithmetic precision to the application and use fast and energy-efficient arithmetic circuits. There are many advantages of using fixed-point to reach efficient ASICs, such as the silicon area can be significantly reduced, the power consumption can also be reduced, and the maximum clock frequency can be increased. Identifying the number of required bits, however, is a highly nonlinear optimization problem that strongly depends on the application. The main disadvantage is the fact that fixed-point optimization is a difficult optimization problem.

For this project, we first need to be familiar with 2's complement arithmetic and fixed-point numbers. The advantages of using 2's complement are that addition can be carried out with standard adder circuits, such as ripple, Kogge-Stone, etc., the multiplicand can also be carried out with standard multiplier circuits, and sign can be extracted easily.

1.1 Fixed-point representation

In modern computers, floating point numbers are used to represent numbers with a fractional part. However, floating-point numbers are not the only way to represent fractional numbers. The fixed-point representation is also widely used essentially in ASICs.

The fixed-point format that we adopted here inherit all the advantages carried out by 2's complement, such as logic and arithmetic shifts, comparisons, leading-zeros and leading-ones detection, logic functions...etc. In addition, fixed-point representation has some advantages over floating point numbers, including arithmetic circuits with

fixed-point implementation requires significantly smaller area and power consumption. Moreover, the maximum clock frequency can be increased.

For fixed point design, binary representation is used to represent fractional number by extending the pattern to include negative exponents. B is defined as the total number of bits that we use to express a number, which includes the integer bit part (I) and the fractional bit part (F), and is denoted as 's', I.F. The range of the number that can be used to represent by using B bit is $(-2^{(B-1)}/2F)$ to $(2^{(B-1)}-1)/2F$.

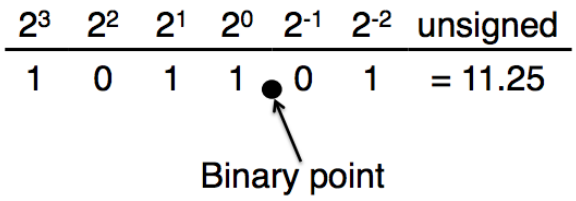

Figure 1: An example of fixed point representation of real number

In Figure 2, it is shown that 11.25 is represented as fixed-point number, with the integer bit number (I) of 4 and the fractional bit number (F) of 2.

To convert from real number to fixed-point number, we simply use the equation below, with the assumption that there is sufficient amount of integer bits given:

$$y = round(x \cdot 2^F)/2^F$$

F is the number of fractional bits.

1.2 Basic operations

There are some key arithmetic operations we wish to implement, including addition, subtraction, and multiplication. For example, multiplication of two numbers with B1 and B2 bits respectively, the resulting number will require B1+B2 bits.

Resizing is also a key operation we wish to implement. Since our goal is to reduce the overall number of bits, we have to reduce the number of fractional bits as well as the number of integer bits. There are different ways to resize fixed-point numbers in hardware, and these methods will affect the most significant number (MSB), least significant number(LSB), and the precision. One issue to be address is that the fewer bits that we use to represent the number, the less accurate the precision is going to be. In resizing the integer part, if we wish to use more bits to represent the integer part, then it is easy and very straightforward, we can just use sign extension. However, if we want to reduce the number of integer bits then we need to be careful because this may affect the performance, and the resizing type that is chosen also affects hardware complexity as well. There are basically two methods to reduce the number of integer bits, including saturation (Sat)and wrapping (Wrp).

For saturation, the result is clipped to maximum representable value (Figure 2). Though saturation in hardware requires additional comparators, it is usually more accurate for final results.

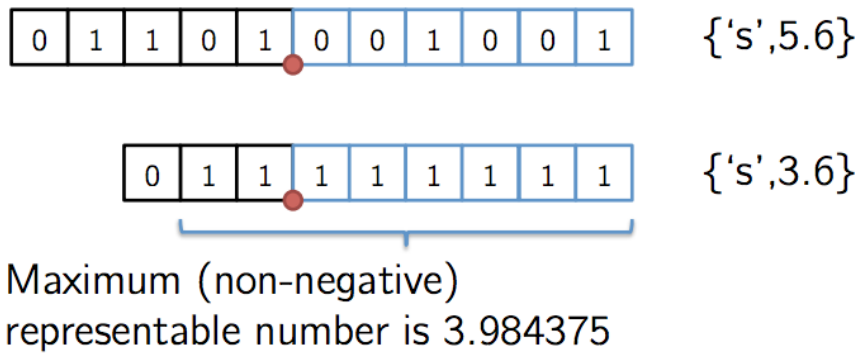

Figure 2: Saturation[9]

For wrapping in integer part, we just throw away the leading bits. This may look like a bad idea to implement because it might lead to a totally off number compared to the correct one, but depending on application, one can get correct results without

hardware overhead (Figure 3). Wrapping does not require additional hardware but can lead to catastrophic results.

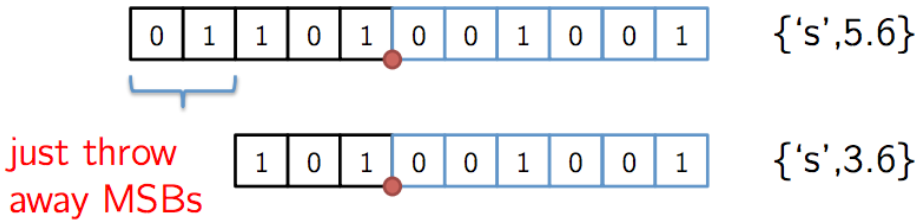

Figure 3: Wrapping[9]

As for the resizing of fractional part, increase the number of fractional bits is also straightforward. However, to reduce the fraction bits may cause some problems, and there are two ways to reduce the fractional bits, including rounding (Rnd) and truncation (Trc).

Rounding removes fractional bits by performing rounding to the next best representation (Figure 4). This requires increment circuitry but is slightly more accurate.

For truncation, it just remove unused fractional bits (Figure 5). Truncation is free,

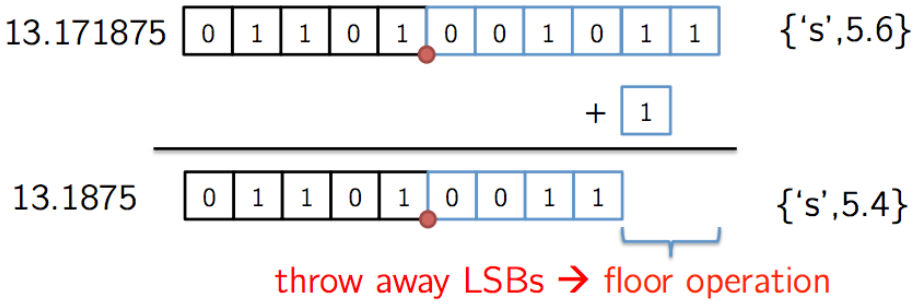

Figure 4: Rounding[9]

but it is also less accurate.

In conclusion, the most common choice for resizing is wrapping for integer part and truncation for fractional part.

1.3 Optimization

For the fixed-point design and optimization in this project, the common approach is to define a quality metric for the application, such as mean-square error (MSE) to floating-point results, or error rate. We want to successively convert a floating-point model into fixed-point model using Python. The way to do this is by looking at signal statistics, and simulate algorithms with the quality metrics that is chosen, and then use grid search with representative data.

We use the quantization function to convert real number to fixed point number and resize it to certain number of integer bits and fractional bits.To convert the floating-point model step by step, we need to quantize every step after an operation is performed (Figure 6). We need to ensure that quality metric is met in every step after the quantization of every operation is performed whenever we quantize more and more signals. After all the arithmetic operations, the output signal is compared with the floating point version of the signal and the MSE is calculated. Our goal is to find the optimized combination of integer bits (I) and fractional bits (F) to reduce the total number of bits (B) as well as maintain a reasonably low MSE, minimizing the quantization cost. I and F may be different for different arithmetic operation, so these paramters form a high dimensional design space, which is exponentially difficult to explore and requires an effective way to find the solution.

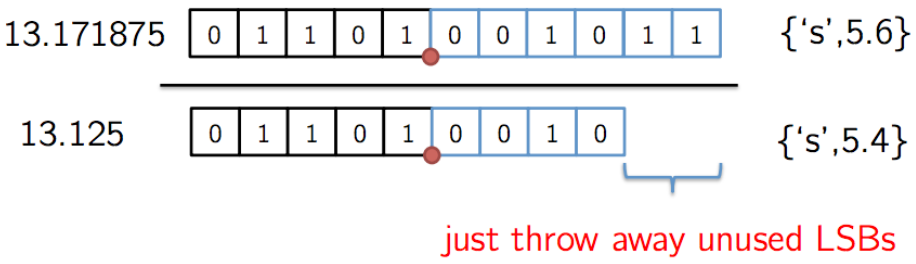

Figure 5: Truncation[9]

The strategy that we use to find initial integer bits is to extract the signal statistics, find its dynamic range, and set the integer bits. To determine the initial fractional bits, we need to make an educated guess, and then we can use random walk to help us figure out which combination of total bits meet our chosen quality metrics criteria.

1.4 Challenges

Though there are many advantages to adapt fixed-point representation in ASICs' applications, the optimization of fixed-point design is a huge challenge. Since most of the current ASICs are still optimized through human interaction, and there are not much literature reference in the optimization of fixed-point designs. There are only exsisting results for basic DSP opterations, such as FFTs, FIR/IIR filters, and CORDICs, ...ect.. Though the automated fixed-point optimization was studied in research, but there are not yet any useful applications regarding the optimization design protocol for fixed-points representation. One of the main problems invovlving in the optimization of fixed-point is that the potential search space is large and therefore to find the desired fixed-point bits is very time consuming.

Figure 6: Quantization steps[9]

1.5 Application

In this project, the application we mainly used for simulation is finite impulse response (FIR) filter. And we use lots of audio files (including music and speech) as input data. A high-pass FIR filter with cut-off frequency at 440Hz is applied to these audio files. The reason we use FIR for the simulations is that FIR circuit is typical digital signal processing circuit and requires the optimization of various key parameters in terms of fixed point design.

Figure 7 shows the block diagram for our project. We compared the results of fixed point design and floating point reference after applying the FIR filter and calculated the MSE.

Besides FIR filter, we also tried other application circuit like CORDIC (COordinate Rotation DIgital Computer). We implement the CORDIC algorithm in Python and compare the result of fixed point design with floating point reference.

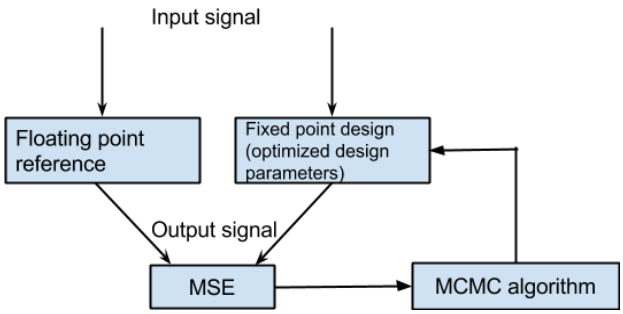

Figure 7: The block diagram for our project

2 Methodology

In this section, several algorithms implemented in our simulations are introduced. In addition, some Python packages of MCMC are presented in this part.

2.1 Exhaustive Search(ES)

Exhaustive search is the simplest one. We just try every combination of integer and fractional bits to calculate the MSE. Then from those combinations that meet the MSE requirement, we find the one solution with the least bits in total.

2.2 Random Walk(RW)

Random walk consists of a succession of random steps. In this project, we randomly pick up an initial point in the whole search space. Then randomly increase or decrease one bit in the integer bits or fractional bits with equal probability. Calculate the MSE, compare with the expected MSE and decide whether to take the next random step.

2.3 Directed Walk I(DW I)

Say we set the critical condition to -45dB. In the result of previous experiments, we can see that the general trend is: with the increase of any bit, the MSE is going to decrease. This is where directed walk comes:

if MSE > -45dB **then**
  increase any bit at one time
else
  decrease any bit at one time
end if

until find a solution

2.4 Directed Walk II(DW II)

Directed walk I can definitely give us a result that meets the bar, but it is not ideal/best solution. We want the sum of bits to be minimum, so we want the result kind of escaping the local optima. Following this idea, we design the algorithm like this(directed walk II):

if MSE > -45dB **then**

increase any bit with P = 0.7 and decrease with P = 0.3 at one time

else

increase any bit with P = 0.3 and decrease with P = 0.7 at one time

end if

This method sheds interesting light on MCMC method, as well as Simulated Annealing Algorithm. The probability to increase or decrease the number of bit is what MCMC tries to learn from the sample space.

2.5 Simulated Annealing(SA)

Simulate annealing[1] goes one step further. In Directed Walk II, the probability to increase/decrease is set to a constant value, independent with the running time, the position of the search space, the current state the search is at, etc. (so here we can find a lot of ways to improve the algorithm, like to define a evaluation function which runs faster rather than calculating the computationally expensive MSE etc.). However, in Simulated Annealing(SA), the probability to increase/decrease any bit is the variant of running time.[8]

We design the SA algorithm like this:

Define variables: SAControl = 6, in every iteration it increases by one;
ugly: random variable generated from [1, SAControl](uniform distribution), and at the same time we set a constant value c = 3. so with the increase of time, the probability that ugly variable falls into [1,3] is decreasing. Through this mechanism, we implement the temperature variable of SA.[8]

if MSE(next)<MSE(now) **then**
  always accept the better result
else if ugly <= 3 **then**
  if MSE(next)>MSE(now) **then**
    accept and move to the next step
  else
    stay at current state
  end if
end if

until find a solution

There are many variables to be controlled, such as how do you control the temperature variable, how long is the step-length of jumping from current step to the next step, etc. We can use cross-validation to solve this problem.

2.6 Genetic Algorithm(GA)

Follow the Genetic Algorithm[2] design patterns, we design the Genetic Algorithm(GA) like this:

After incorporate the 4 integer bits, now we have 8 parameters. Start with k randomly generated states (k=4), which also known as populations in GA.

Then we define a Fitness Function, which depends on the current MSE of that population, to evaluate the likelihood this population could survive in the next round. The MSE(dB) is lower(better), the likelihood to survive is greater.

Then we select k next-round population according to the Fitness Function independently. Later, we randomly select the position to pair, and after cross-over, we randomly do the mutation for any bit.

As you can see, the number of parameters of GA is even greater than SA, and this is one of the difficult point of the GA.

2.7 Markov-chain Monte-Carlo algorithm (MCMC)

MCMC is used when it is not possible to sample θ directly from $p(\theta|y)$, where $p(\theta|y)$ is the target density and θ is the bit parameters. So instead, sampling is performed iteratively so that each step of the process is expected to draw from a distribution that gets closer and closer to $p(\theta|y)$.

If the simulations have been obtained from the posterior distribution $p(\theta|y)$, we can draw from the predictive distribution of unobserved data $\widetilde{y}$. For each draw of θ from the posterior distribution, just draw one value $\widetilde{y}$ form the predictive distribution, $p(y|\theta)$. The set of simulated $\widetilde{y}$'s from all the θ's characterizes the posterior predictive distribu-

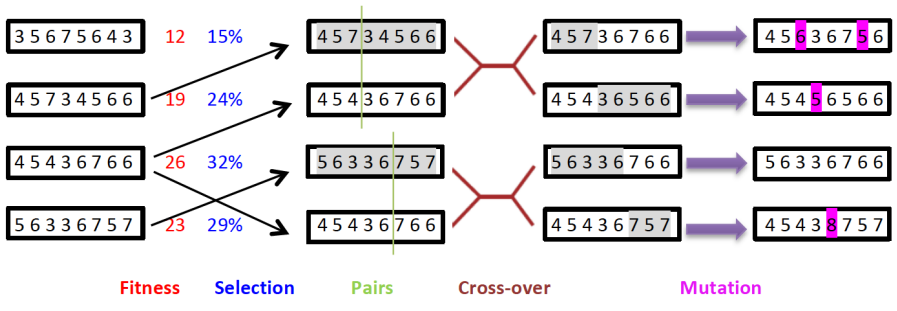

Figure 8: Genetic Algorithm

[10]

tion.

The key is to create a Markov process whose stationary distribution is the specified $p(\theta|y)$ and run the simulation long enough that the distribution of current draw is close to the stationary distribution.

MCMC is based on drawing values of is used when θ from approximate distributions and then correcting those draws to better approximate the target posterior distribution, $p(\theta|y)$. Samples are drawn sequentially, with the distribution of the sampled draws depending on the last value draws; hence, the draws form a Markov chain.

A Markov chain as defined in probability theory, is a sequence of random variables θ^1,θ^2,..., for which, for any t, the distribution of θ^tgiven all previous $\theta's$ depends only on the most recent value, θ^{t-1}. The approximate distributions are improved at each step in the simulation, in the sense of converging to the target distribution.

Several independent sequence of simulation draws are created; each sequence, θ^t, $t = 1, 2, 3, ...$, is produced by starting at some point θ^0 and then, for each t, drawing θ^t from a transition distribution, $T_t(\theta^t|\theta^{t-1})$ that depends on the previous draw, θ^{t-1}.The transition probability distribution must be constructed so that the Markov chain converges to a unique stationary distribution that is the posterior distribution, $p(\theta|y)$.

2.8 Metropolis-Hastings algorithm (MH)

Metropolis-Hastings algorithm is a typical Markov chain Monte Carlo method. MH is an adaption of a random walk that uses acceptance and rejection rule to converge to a specified target distribution. The main procedure of the algorithm include constructing a proper proposal distribution, sampling a new point from the proposal distribution, and acceptance or rejection of the new point.

First, we start to draw from a point θ_0, with $p(\theta^0|y) > 0$,from a starting distribution $p_0(\theta)$.

Second, the starting distribution can be based on an approximation. For $t = 1, 2, 3, ...$: s (a) Sample a proposal $\theta^*$from a jumping distribution or sometimes called proposal distribution at time t, $J_t(\theta^t|\theta^{t-1})$. The jumping distribution of MH is not symmetric, that is $J_t(\theta_a|\theta_b)$ need not to be $J_t(\theta_b|\theta_a)$ for all θ_a, θ_b, t.
(b) The ratio of the densities is, $r = \frac{p(\theta|y)/J_t(\theta|\theta^{t-1})}{p(\theta^{t-1}|y)/J_t(\theta^{t-1}|\theta)}$, where J_t is the proposal distribution, θ is the proposed point, and θ^{t-1} is the original point. So the probability that the next point θ^t will be the proposed θ is min(r,1). Otherwise, θ^t will be the same as θ^{t-1}.

3 Results

This section presents simulation results including the relationship between fixed point parameters and MSE, the comparison of algorithms and the prior probability of Metropolis-Hastings algorithm.

3.1 Quantization function

At the beginning, we wrote the Quantization function in Python. This function dealt with all resizing methods, including Saturation, Wrapping, Rounding and Truncation. Later, we implemented the same function in C and embedded the C function in Python code with Python/C API. It turns out to run 4 times faster than Python function. Code in C and Python is in Appendix.

3.2 MSE calculation

Mean Squared Error(MSE) is an estimator that performs the difference between the estimated value and real value. To get the good audio output, we used MSE to measure the "quality" of our estimated value. The formula of MSE is: $MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i^{predictor} - Y_i\right)^2$. $Y^{predictor}$ is the vector of n predictions and Y is the vector of the true values. In our problem, $MSE = \|value_{fixed} - value_{floating}\|^2$

To simplify the problem, we normalized the initial data and fixed the bits of integer part. In other words, we considered only the fractional bits. However, we still need to provide the integer bits in Quantization function. The integer bits we used were 4. We calculated MSEs in Python from changing only one parameter to changing four parameters. Range of varied parameter is 1 to 10. Since MSEs are small and hard to compare, we took logarithmic function of MSEs. For example, if MSE is 0.0001,

$\log_2 0.0001 = -13.28$, which is a negative number with larger absolute value. Now, all MSEs are negative and have larger absolute value, comparatively. Thus, smaller absolute value signifies smaller initial MSE.

When we fixed three parameters, the results are very straightforward. Generally speaking, more bits of varied parameter got larger log MSEs. It is reasonable since less bit is less precise. Intuitively, MSEs perform different in those resizing methods. The result in Wrapping and Truncation method is shown in the figure 9. The varied parameter here is filter coefficient.

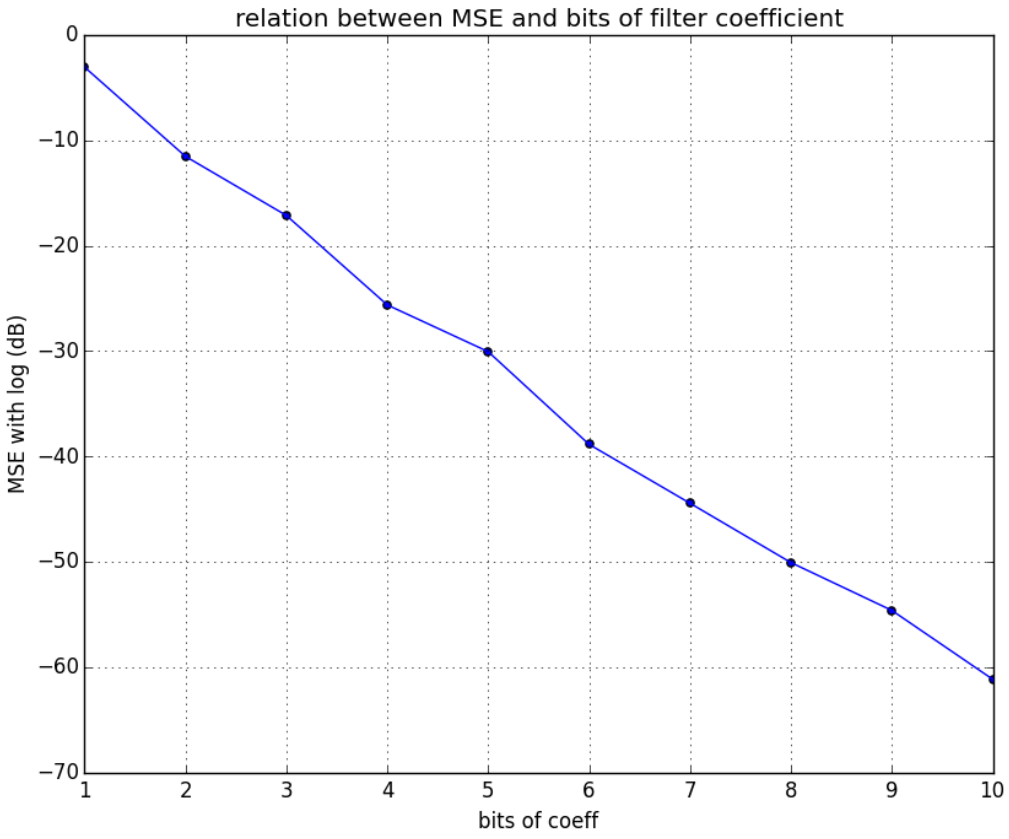

Figure 9: The relation between MSEs and bits of filter coefficient

When we varied three parameters, there are 100 combinations. The combinations of (bits of parameter1, bits of parameter2, bits of parameter3) range from (1, 1, 1) to (10, 10, 10). Intuitively, (1, 1, 1) is the least accurate one but it is less consuming, while (10, 10, 10) is the case that guarantee the best precise but costs too many bits. These two cases are extreme. For other combinations, the accuracies cannot be considered only by the total bits. For example, both (8, 8, 2) and (6, 6, 6) have 18 bits. (6, 6, 6) performs better than (8, 8, 2), since 2 is the bottleneck of the latter MSE. To check three parameters thoroughly, we drew pictures to show all cases. See figure 10-16 for more details. x-Axis in those picture is bits of multiplication and different shapes represent different bits of addition. Filter coefficient is fixed in each picture.

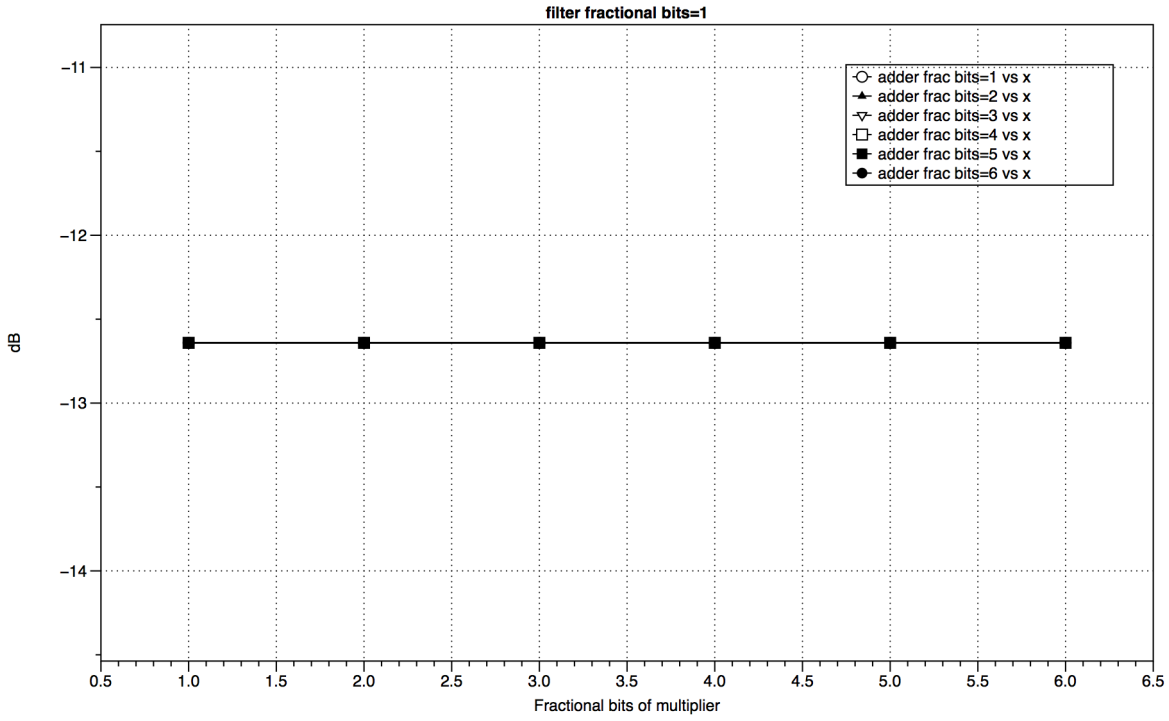

Figure 10: MSEs when bit of filter coefficient is 1

When we varied four parameters, the change of MSEs became more complicated. We recorded MSEs for the combination of four parameters and drew the changes of MSEs in different pictures. Bits of multiplication and addition variables are fixed in each picture. Thus, we had 100 pictures, which are all combinations of multiplication and

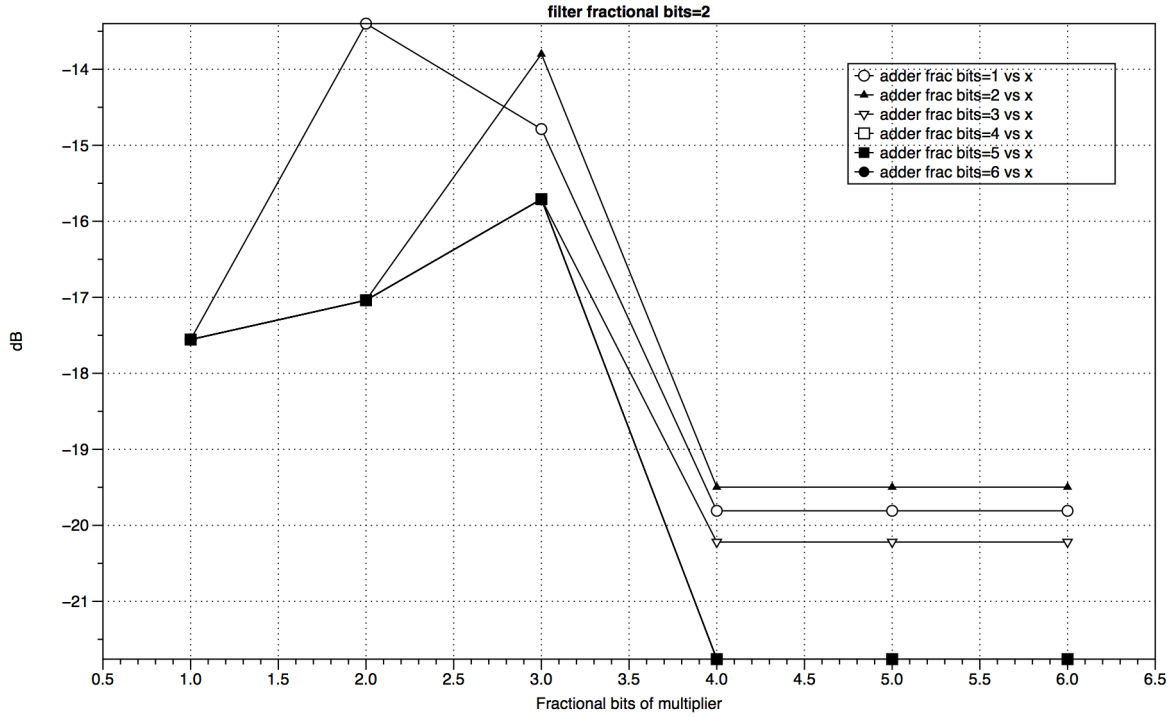

Figure 11: MSEs when bit of filter coefficient is 2

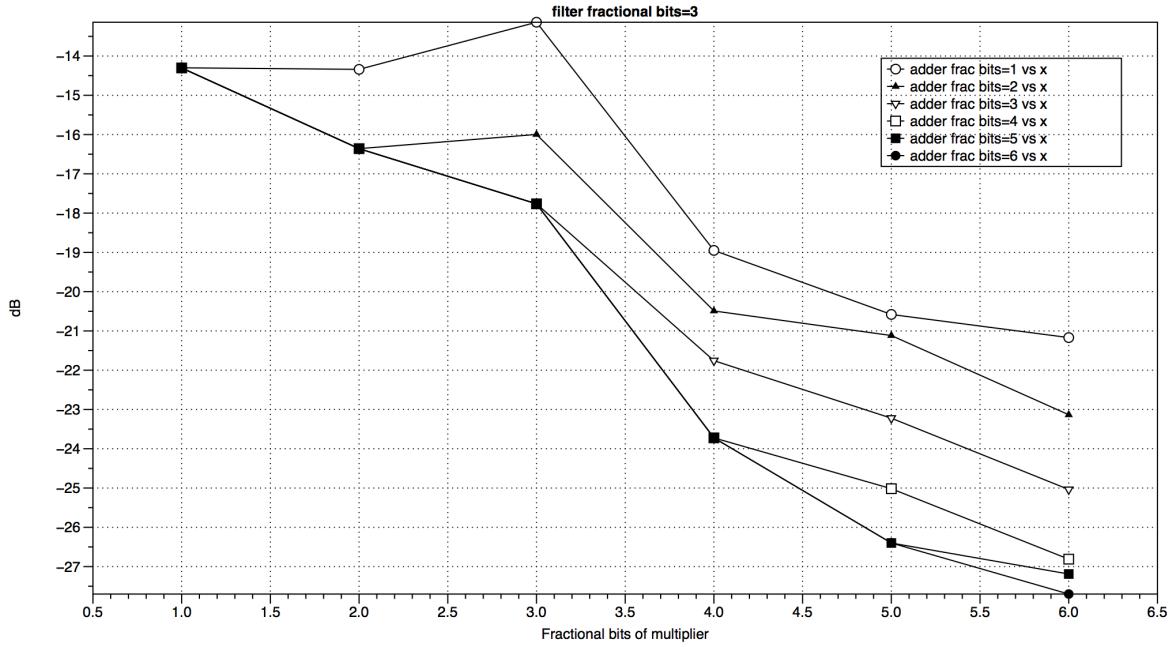

Figure 12: MSEs when bit of filter coefficient is 3

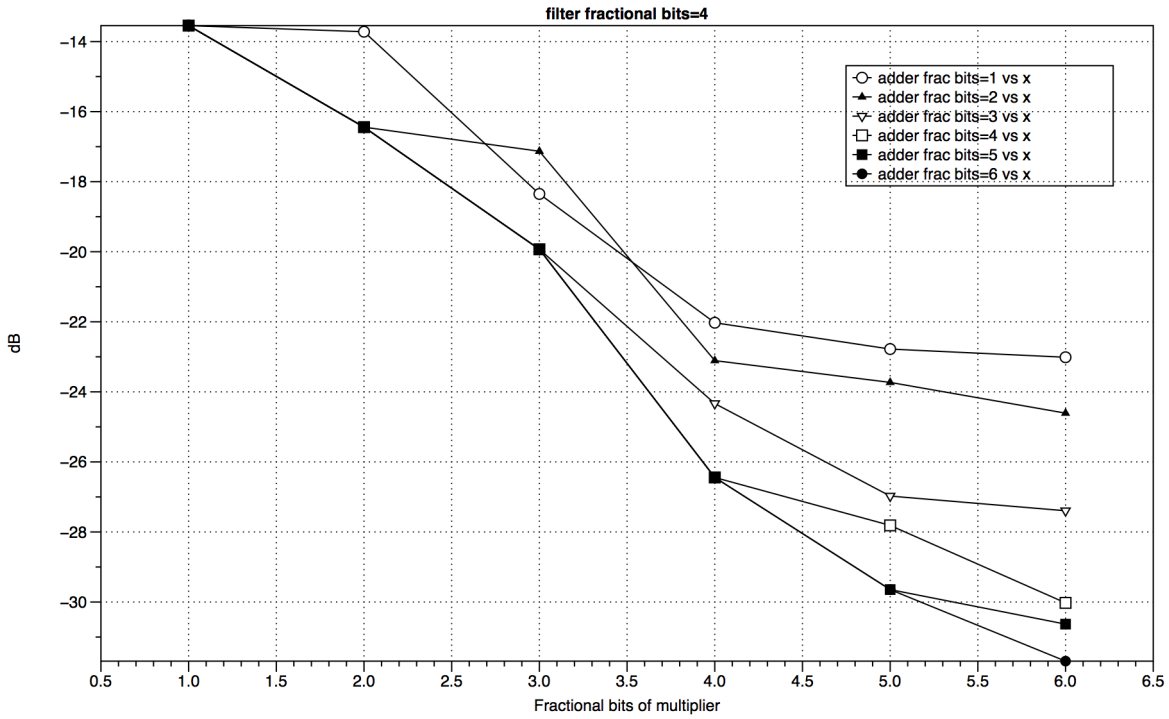

Figure 13: MSEs when bit of filter coefficient is 4

addition.

Intuitively, if any bit is small, the MSE is more likely to be large. The figure 10 is the case that MSEs are far away from threshold. x-Axis in the figure 10 is the bits of signal and y-Axis represents the MSEs. There are 10 lines and different line has different bit of filter coefficient.

From this picture, we noticed that all MSEs here were very large, which was the bad case.

Let's consider the extreme good case. If both multiplication and addition have 10 bits, we drew the figure 17.

If either input signal or filter coefficient has small bit, the MSE is still large.

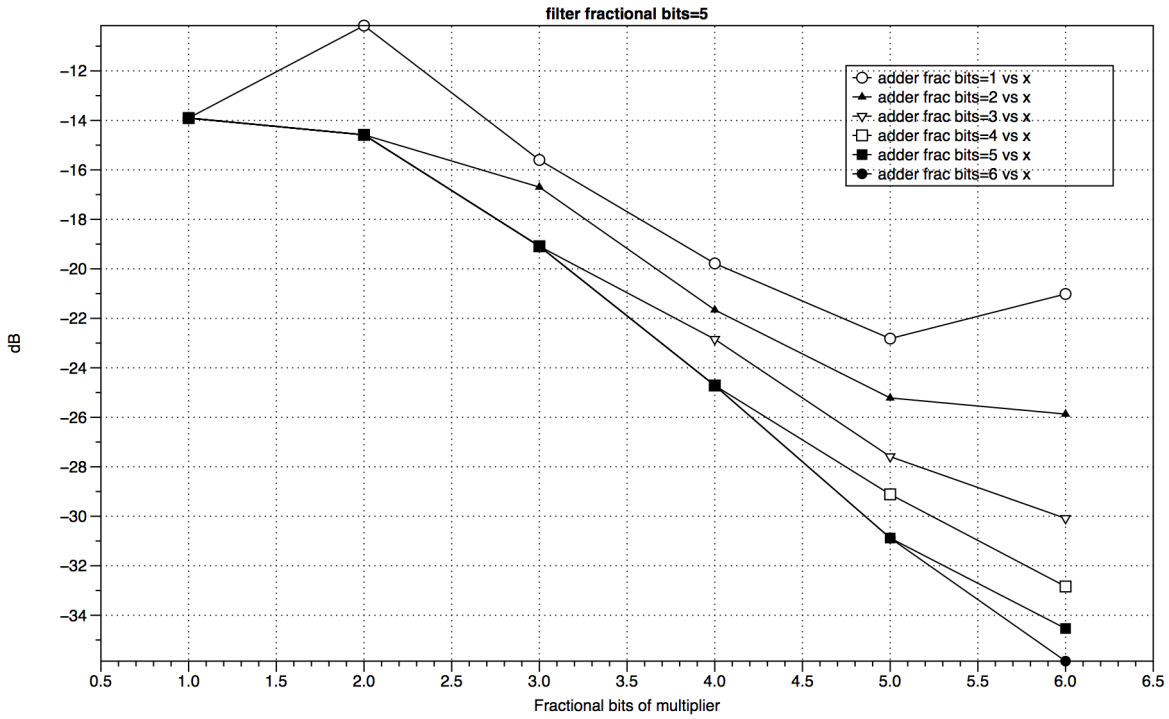

Figure 14: MSEs when bit of filter coefficient is 5

Since it is a trade-off between the good MSE and less number of total bits, we systemized all MSEs and then found the critical condition. The critical condition is the case that MSE can satisfy the requirement and the total bits can be as less as possible. For wrapping and rounding method, the multiplication, addition, filter coefficient and input signal are 8, 8, 8, 6 and 8 bits or 8, 8, 7 and 7 bits, respectively.

The figure 18 contains all MSEs that meet the requirement that MSE $<$ -45dB. Here, negative values in white and green cells, including two yellow cells, are MSEs. Yellow cells in the form of "$number_1\_number_2$" delegate the combinations $(bit_{multiplication}, bit_{addition})$. Cells' value in blue rows are bits of filter coefficient and cells' value in blue columns are bits of input audio signal.

To analyze the critical case, we used the figure 19.

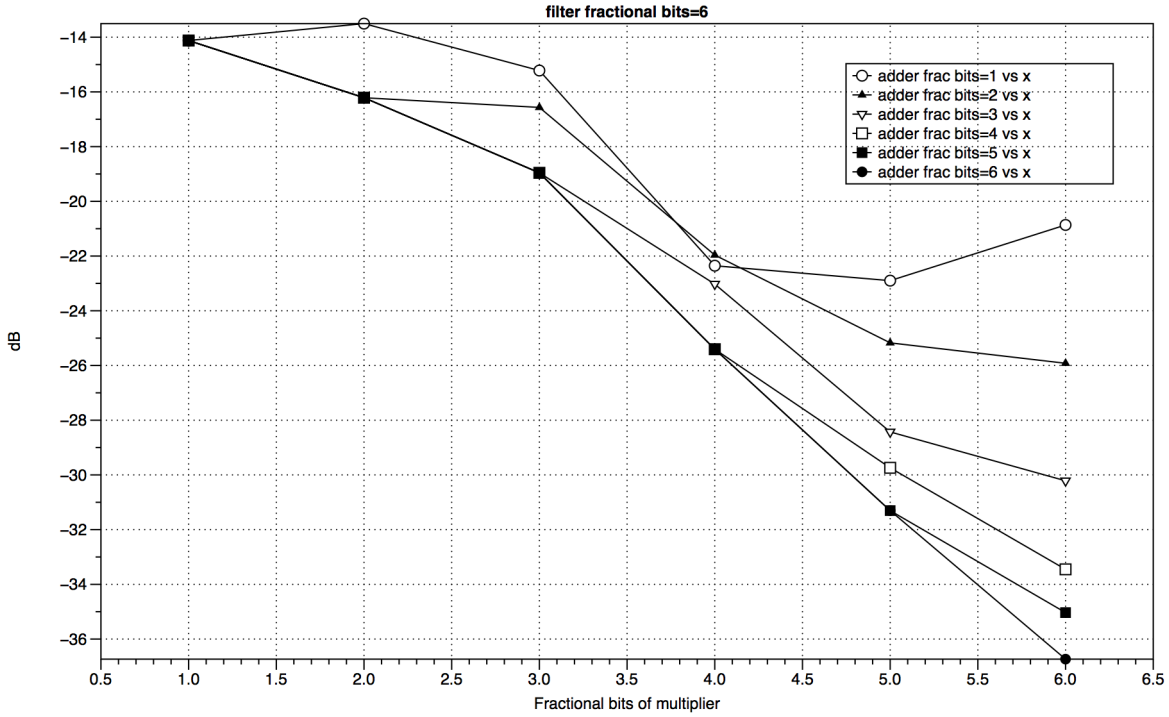

Figure 15: MSEs when bit of filter coefficient is 6

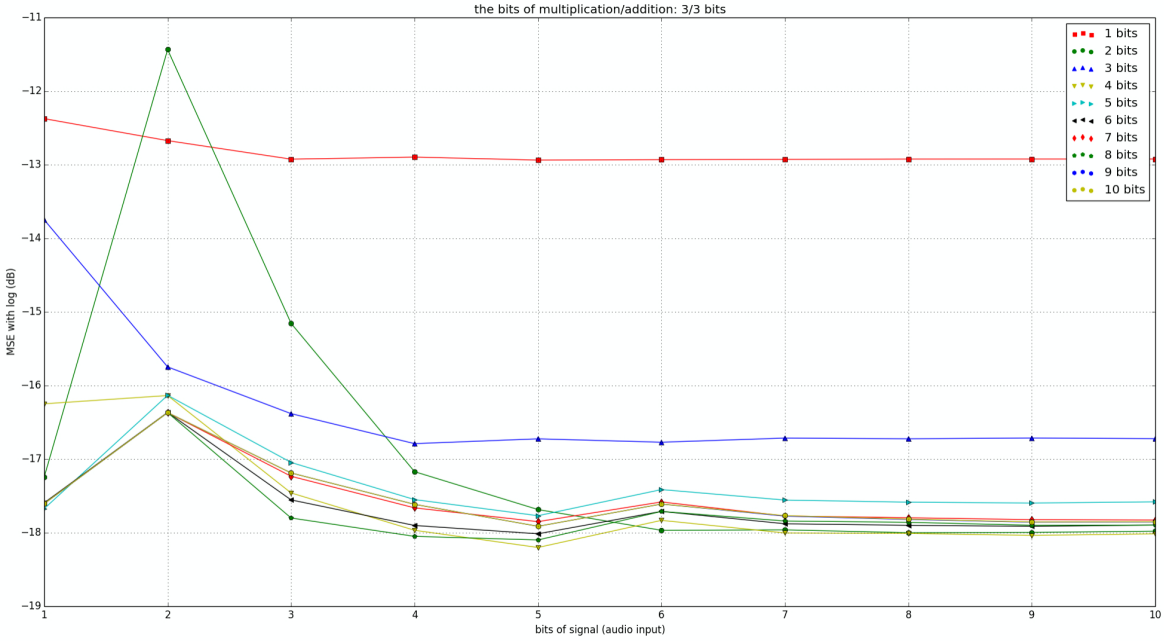

Figure 16: The MSEs if bits of both multiplication and addition are 3

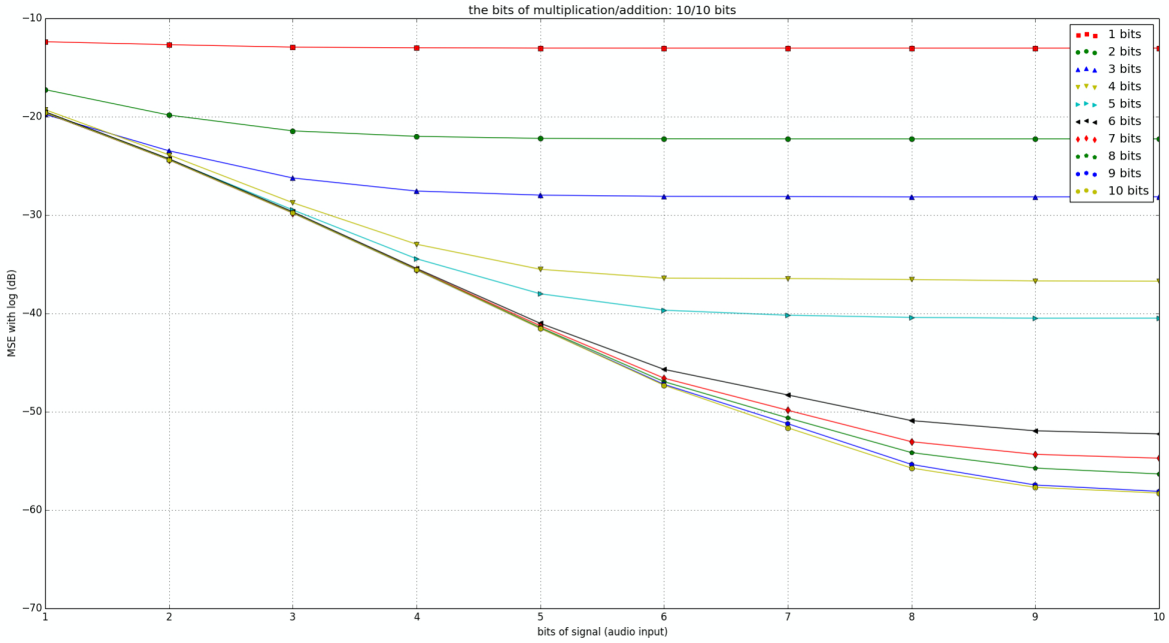

Figure 17: The MSEs if bits of both multiplication and addition are 10

10_10	1	2	3	4	5	6	7	8	9	10
1	-12.37	-12.67	-12.92	-12.99	-13.02	-13.02	-13.02	-13.02	-13.02	-13.02
2	-17.25	-19.83	-21.44	-22	-22.21	-22.24	-22.25	-22.25	-22.25	-22.25
3	-19.76	-23.47	-26.23	-27.56	-27.97	-28.1	-28.12	-28.15	-28.15	-28.14
4	-19.29	-23.88	-28.74	-32.96	-35.51	-36.42	-36.46	-36.55	-36.69	-36.73
5	-19.62	-24.32	-29.47	-34.44	-38	-39.67	-40.18	-40.41	-40.48	-40.48
6	-19.5	-24.27	-29.66	-35.42	-41.01	-45.69	-48.29	-50.9	-51.94	-52.25
7	-19.62	-24.39	-29.79	-35.52	-41.26	-46.58	-49.85	-53.05	-54.33	-54.72
8	-19.6	-24.38	-29.77	-35.54	-41.43	-46.93	-50.62	-54.15	-55.73	-56.32
9	-19.6	-24.37	-29.78	-35.58	-41.51	-47.21	-51.21	-55.36	-57.45	-58.1
10	-19.59	-24.38	-29.78	-35.57	-41.51	-47.3	-51.63	-55.73	-57.69	-58.26

9_9	1	2	3	4	5	6	7	8	9	10
1	-12.37	-12.67	-12.92	-12.99	-13.02	-13.02	-13.02	-13.02	-13.02	-13.02
2	-17.25	-19.83	-21.44	-22	-22.21	-22.24	-22.25	-22.23	-22.24	-22.25
3	-19.76	-23.47	-26.23	-27.56	-27.97	-28.1	-28.13	-28.15	-28.14	-28.13
4	-19.29	-23.88	-28.74	-32.96	-35.51	-35.77	-36.1	-36.49	-36.63	-36.67
5	-19.62	-24.32	-29.47	-34.44	-37.71	-39.13	-39.88	-40.24	-40.31	-40.29
6	-19.5	-24.27	-29.66	-35.21	-40.37	-43.45	-47.32	-49.25	-49.88	-50.09
7	-19.62	-24.39	-29.72	-35.43	-40.93	-44.71	-48.53	-50.52	-51.14	-51.36
8	-19.6	-24.36	-29.74	-35.53	-41.12	-45.15	-49.04	-51.1	-51.82	-52.05
9	-19.6	-24.38	-29.76	-35.56	-41.26	-45.45	-49.49	-51.68	-52.49	-52.66
10	-19.6	-24.38	-29.74	-35.55	-41.32	-45.82	-49.74	-51.84	-52.55	-52.71

9_10	1	2	3	4	5	6	7	8	9	10
1	-12.37	-12.67	-12.92	-12.99	-13.02	-13.02	-13.02	-13.02	-13.02	-13.02
2	-17.25	-19.83	-21.44	-22	-22.21	-22.24	-22.25	-22.23	-22.24	-22.25
3	-19.76	-23.47	-26.23	-27.56	-27.97	-28.1	-28.13	-28.15	-28.14	-28.13
4	-19.29	-23.88	-28.74	-32.96	-35.51	-35.77	-36.1	-36.49	-36.63	-36.67
5	-19.62	-24.32	-29.47	-34.44	-37.71	-39.13	-39.88	-40.24	-40.31	-40.29
6	-19.5	-24.27	-29.66	-35.21	-40.37	-43.45	-47.32	-49.25	-49.88	-50.09
7	-19.62	-24.39	-29.72	-35.43	-40.93	-44.71	-48.53	-50.52	-51.14	-51.36
8	-19.6	-24.36	-29.74	-35.53	-41.12	-45.15	-49.04	-51.1	-51.82	-52.05
9	-19.6	-24.38	-29.76	-35.56	-41.26	-45.45	-49.49	-51.68	-52.49	-52.66
10	-19.6	-24.38	-29.74	-35.55	-41.32	-45.82	-49.74	-51.84	-52.55	-52.71

8_9	1	2	3	4	5	6	7	8	9	10
1	-12.37	-12.67	-12.92	-12.99	-13.02	-13.02	-13.02	-13.02	-13.02	-13.02
2	-17.25	-19.83	-21.44	-22	-22.21	-22.24	-22.18	-22.22	-22.24	-22.25
3	-19.76	-23.47	-26.23	-27.56	-27.97	-27.96	-28.07	-28.1	-28.1	-28.09
4	-19.29	-23.88	-28.74	-32.96	-33.8	-34.68	-35.88	-36.28	-36.38	-36.42
5	-19.62	-24.32	-29.47	-33.87	-36.35	-38.15	-39.32	-39.58	-39.68	-39.68
6	-19.5	-24.27	-29.43	-34.6	-38.01	-42.27	-44.85	-45.74	-46.02	-46.04
7	-19.62	-24.32	-29.63	-35.07	-39.05	-43.14	-45.41	-46.16	-46.39	-46.46
8	-19.59	-24.34	-29.73	-35.25	-39.42	-43.48	-45.69	-46.42	-46.62	-46.66
9	-19.61	-24.34	-29.74	-35.3	-39.59	-43.64	-45.91	-46.52	-46.73	-46.81
10	-19.61	-24.31	-29.73	-35.35	-39.94	-43.89	-46.02	-46.5	-46.7	-46.78

8_10	1	2	3	4	5	6	7	8	9	10
1	-12.37	-12.67	-12.92	-12.99	-13.02	-13.02	-13.02	-13.02	-13.02	-13.02
2	-17.25	-19.83	-21.44	-22	-22.21	-22.24	-22.18	-22.22	-22.24	-22.25
3	-19.76	-23.47	-26.23	-27.56	-27.97	-27.96	-28.07	-28.1	-28.1	-28.09
4	-19.29	-23.88	-28.74	-32.96	-33.8	-34.68	-35.88	-36.28	-36.38	-36.42
5	-19.62	-24.32	-29.47	-33.87	-36.35	-38.15	-39.32	-39.58	-39.68	-39.68
6	-19.5	-24.27	-29.43	-34.6	-38.01	-42.27	-44.85	-45.74	-46.02	-46.04
7	-19.62	-24.32	-29.63	-35.07	-39.05	-43.14	-45.41	-46.16	-46.39	-46.46
8	-19.59	-24.34	-29.73	-35.25	-39.42	-43.48	-45.69	-46.42	-46.62	-46.66
9	-19.61	-24.34	-29.74	-35.3	-39.59	-43.64	-45.91	-46.52	-46.73	-46.81
10	-19.61	-24.31	-29.73	-35.35	-39.94	-43.89	-46.02	-46.5	-46.7	-46.78

Figure 18: MSEs. Yellow cells with two values is $bit_{multiplication}, bit_{addition}$; cells in blue rows are $bit_{coefficient}$; cells in blue columns are $bit_{inputsignal}$

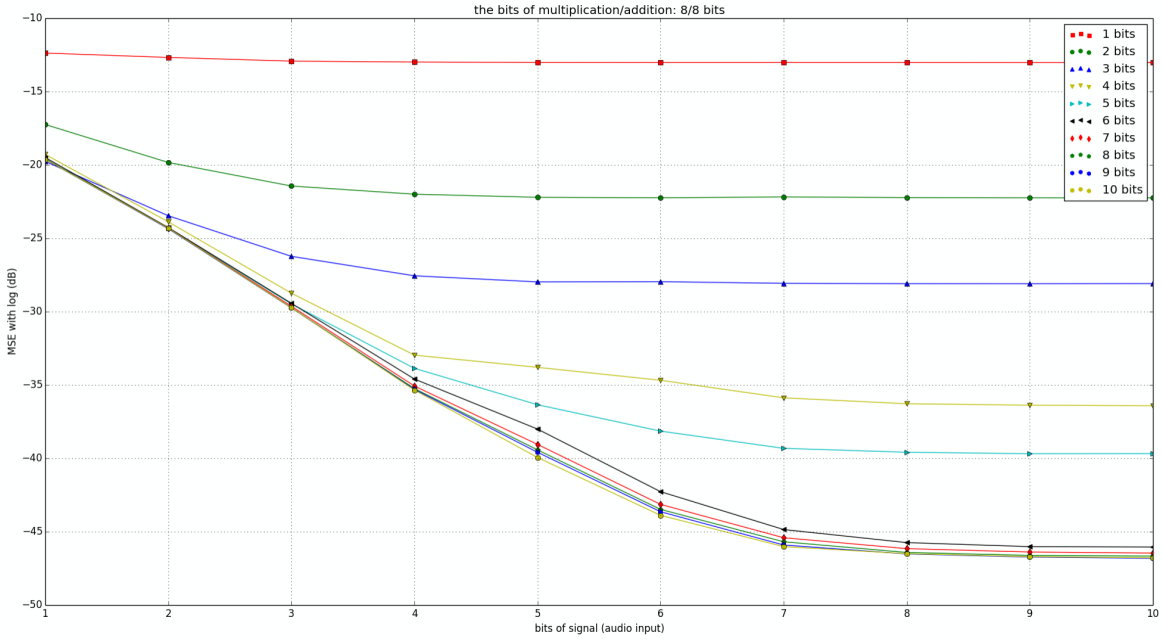

Figure 19: The MSEs if bits of both multiplication and addition are 8

Through those pictures, we can analyze MSEs intuitively.

3.3 Algorithm comparison

1. Exhaustive Search

   Result: can definitely find the best solution; however it takes long time(18h to compute only 4 combinations of fraction bits ranging from 3 to 10.).

2. Random Walk

   The result of this is rather uncertain. Besides the initial condition, sometimes it tries rather unreasonable point like (10,0,0,10), in which the total loss function is restricted by the lowest number of bits with the rest of the number of bits high. From here, we should incorporate heuristic methods.

3. Directed Walk I

Result: overall good, can find one solution in 2-3 minute, but it is usually not the best solution.

4. Directed Walk II

   Result: sometimes good, but usually it is uncertain to give solutions. It can sometimes find a result, but usually it behaves like random walk which leads to unreasonable combinations, and therefore takes long time to run.

5. SA

   Result: better than Directed Walk, in 3-5 minutes it can find the solution near or equal to the best result(In our settings, for a single audio file, and fix the integer bit to 4, and try combinations of 4 fraction bits. The same settings as Exhaustive Search).

6. GA

   Results: though it improves the initial condition, it takes long time to find a solution. Moreover, the 4 population go to the similar state at last.

   Concretely, in our setting, initial condition usually sets around -12dB, after 10 minutes running, it can go to -24dB but 20 minutes no obvious change. We deduce that the 4 populations were tracked at local optima at this time.

Comments: Exhaustive Search and Random Walk methods are just toy example. Essentially, ES provides the reference for later results; but when the number of the parameters is large, it is impossible to run ES(Search space exponentially grows with the number of bits).

DW I can give us a solution in a reasonable time, but not the best result. DW II does not work very well, the probability that we pick randomly(0.7,0.3) sheds light on MCMC methods, in which we learn the probability from the sample space.

SA and GA algorithms are modern AI methods that could give us solutions that approx. the best in a short time. However they have two drawbacks:

1. only deals with one signal sample
2. many parameters should be determined

Though it takes long time to run, it copes with multiple samples at the same time. The method we focused on is MH.

See below table for Search Methods result comparison.

Algorithms	Run-time	Pros	Cons
ES	18 h+	Guaranteed to find a best solution	Run time is unacceptable
RW	uncertain	Illustrate the search in grid space	Uncertain result
DW I	2-3 min	Quickly to find a solution	Solution is usually not optima
DW II	uncertain	Sheds light on SA and MCMC	Behaves like RW(uncertain)
SA	3-5 min	Find approx. the best solution quickly	Need to adjust parameters
GA	long time	Can improve the initial condition	Need to adjust parameters

Table 1: Search methods Result Comparison

3.4 Prior probability of Metropolis Hasting

The prior probability of Metropolis Hasting in our project is $p(y|\theta)$, which is the probability that log MSE is smaller than the threshold, given the bits of parameters. Here, the threshold we chose was -40dB. This value is not special and can be changed to any other reasonable number. The parameter varied was filter coefficient and other parameters, multiplication, addition and input signal were fixed. Thus, to calculate the $p(y|\theta)$, we used hundreds of audios, including both music and speeches. We sectioned audios to shorter sound clips. Each clip has approximately 0.05 second. By processing those audios, we got thousands of millions of sound clips. For each one, we varied

bits of filter coefficient from 1 to 11, fixed bits of other three parameters to 30 and then calculated the MSE. After the calculation, we got a huge matrix in which each column represented a single sound chip, and different rows were different bits. Thus, this matrix had 11 rows and thousands of millions of columns. For each row (bit), we counted the probability that MSE was smaller than the threshold. Finally, we got a vector and each element stored the probabilities. Plotted this vector in Python, we drew the figure 15.

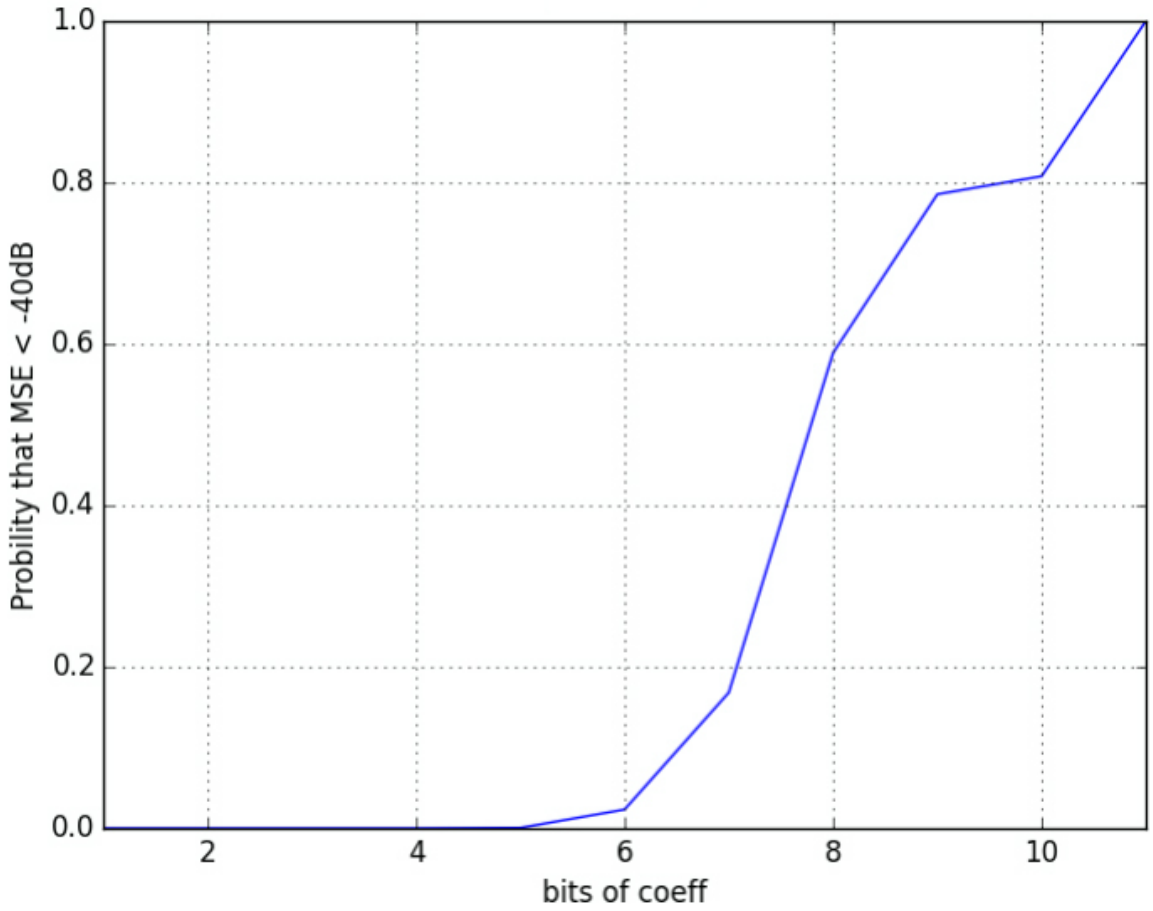

Figure 20: The probability of $MSE < MSE_{threshold}$

Generally speaking, these are the results we got. In fact, the calculation of MSE and prior probabilities are very complex since parameters in every step can be changed. For instance, the bit of every multiplication and every addition could be different as

well. Integer bit is fixed now, which is not the general case. To gain the thorough results, more works would be done in future.

4 Summary

In this project, we propose several ways to optimize the cost in VLSI design with human interaction as less as possible. This is valuable work, since the difficulty when coping with the curse of dimensionality, as well as various of signals, and the structural of digital circuits.

At first stage, we examine several ways to quantize the floating representation for both integer and fraction bits and define the cost function. Then we propose and compare several search algorithms: Exhaustive Search, Random Walk, Directed I and II, Simulated Annealing and Genetic Algorithm. And then the MCMC method is introduced and implemented.

The result shows that Directed Walk I can give us result in a relatively short time, however it is usually not the global optima. Simulated Annealing works in short time to find the approx. the best result, however it is still pretty challenging when facing the curse of dimensionality. Genetic Algorithm works to improve the situation but easily get stuck in local optima(but there are still many parameters and hopefully one combination of parameter will work, but this is also the drawback of GA). MCMC hopefully will work when facing the problem of the curse of dimensionality, and it could deal with thousands of samples at one time. The drawback of MCMC is that it will run relatively longer time.

5 Future Work

5.1 For Search

There is a lot remaining work on SA and GA.

For SA:

1. We only tested few audio files, so future work should be done to generalize this to tons of digital signal files. Just try other files, and very probably you need to have some new parameters to find a good solution faster.
2. Test more parameter combinations using cross-validation etc. There are many parameters, like how do you choose the initial condition(besides random initialization, you could use somewhat like congestion control algorithms in Computer Network [7], after random initialization you could let the bit double by itself until find the solution and decrease then, until find a local optima),how do you control the temperature variable, how fast it should cool down, etc. see above sections for details.

Try self-adaptive search step-length. The step-length could be the dependent variant of the search state. For example, if we set the bar to -45dB, the current state is -15dB, we could increase the number of bit by 3 not just one, whereas when the current state has -40dB, we should carefully increase by just 1 bit, since 1 bit increase might already satisfy our need.

Define new error metrics. Instead of try the computationally expensive MSE, we could define faster error metrics first. For example, when the bar is -45dB, we could first use faster error metrics(heuristics) to find a pre-solution first, which just has lower bar(say -35dB). The faster error metrics can be simple, the intention and intuition behind it is to let us circumvent many local optima quickly, and then around the area near to the best solution, use MSE to calculate it. Sample error metrics like, define

an average avg, and the bit which is below the avg has greater probability to increase, and the greater the deviation(from the avg), the greater probability. For example, when we have a combination of (3,5,5,8), the first bit(3) has the largest probability to increase in the next round, and the last one(8) has the least to increase(or, greater probability to decrease). Through this trick, we could first reach the -35dB quickly, and on the second phase leading to -45dB, use a more elaborate and refined(however more computationally expensive) error metrics, like MSE.

For GA:
Test more parameter combinations using cross-validation etc. There are even more parameters, see above sections for details.

Current work has been stuck at local optima. How to get out of local optima, like incorporating more random factor (eg. increase the prob of mutation), or after a certain period – if cannot get out of local optima, do re-initialization, will still be investigated in the future work.

5.2 For MCMC

More audios are needed to get more accurate results. Also, we can change the unit length of sound clips, such as changing audio length from 0.05 second to 0.5 second or 0.005 second. We also need to calculate prior probabilities for changing only multiplication bit, addition bit and audio input bit, changing two parameters together, three parameters together and all parameters. Next step, we should try to use different asymmetric proposal distributions. There would be a lot of combinations of prior probabilities and distributions. Our goal is to find the best combination.

6 Acknowledgments

We would like to show our deepest gratitude to Professor Christoph Studer, Professor Zhiru Zhang, and Ph.D. student Ramina Ghods, who have provided us with patient and valuable guidance in the project.

7 Appendix

7.1 python code for MSE calculation

github: https://github.com/lk432/MengProject

7.2 Python packages

1. Emcee

   Emcee is an MIT licensed pure-Python implementation of Goodman & Weares Affine Invariant MCMC Ensemble sampler. It is very lightweight. Suppose we execute 100,000 samples, the execution time is approximately 6 second, including burn-in. Since it is pure python, installation is easy with pip. It has not much built-in beyond basic MCMC sampling, and is straightforward and pythonic. [3]

2. Pymc

   Pymc is a python module that can also implement MCMC. Its flexible and extensible and thus it is applicable to a large suite of problems. It has lots of built-in functionality in python. Besides core sampling functionality, pymc has the methods for summarizing output, plotting, goodness-of-fit and convergence diagnostics. To install it, fortran compiler is required. For execution time, it needs about 17 seconds. [4]

3. Pystan

   stan is a package for Bayesian inference with MCMC sampling. It is freedom-respecting, open-source software. It has different interfaces for varied language, such as RStan for R, CmdStan for shell, MatlabStan for Matlab and Stan.jl for Julia. For python, it is pystan. It is not pure python and has lots of built-in functionality in Stan-specific language. Thus, users must learn Stan model

specification language. The installation requires C compiler and Stan and no binaries is available. To execute it, such as 100, 000 samples, it needs 20 seconds compilation and 6 second computation. [5]

4. Winbugs

   winbugs is a stand-alone, self-contained application. It is statistical software for Bayesian analysis using Markov chain Monte Carlo and based on the Bayesian inference using Gibbs Sampling. [6]

References

[1] http://en.wikipedia.org/wiki/Simulated_annealing.

[2] http://en.wikipedia.org/wiki/Genetic_algorithm.

[3] http://dan.iel.fm/emcee/current/.

[4] http://pymc-devs.github.io/pymc/.

[5] https://pystan.readthedocs.org/en/latest/.

[6] http://en.wikipedia.org/wiki/WinBUGS.

[7] Keith W. Ross James F. Kurose. *Computer Networking: A Top-Down Approach (6th Edition)*. Pearson, 2012.

[8] Peter Norvig and Stuart J. Russell. *Artificial Intelligence: A Modern Approach*.

[9] Christoph Studer. Slides: Fixed-point asic design.

[10] Linli Xu. Slides: Artificial intelligence tutorial. University of Science of Technology of China (USTC).