ABSTRACT

Title of dissertation: APPROXIMATE COMPUTING TECHNIQUES FOR LOW POWER AND ENERGY EFFICIENCY

Mingze Gao
Doctor of Philosophy, 2018

Dissertation directed by: Professor Gang Qu
Electrical & Computer Engineering Department
and The Institute for Systems Research

Approximate computing is an emerging computation paradigm in the era of the Internet of things, big data and AI. It takes advantages of the error-tolerable feature of many applications, such as machine learning and image/signal processing, to reduce the resources consumption and delivers a certain level of computation quality. In this dissertation, we propose several data format oriented approximate computing techniques that will dramatically increase the power/energy efficiency with the insignificant loss of computational quality.

For the integer computations, we propose an approximate integer format (AIF) and its associated arithmetic mechanism with controllable computation accuracy. In AIF, operands are segmented at runtime such that the computation is performed only on part of operands by computing units (such as adders and multipliers) of smaller bit-width. The proposed AIF can be used for any arithmetic operation and can be extended to fixed point numbers.

AIF requires additional customized hardware support. We also provide a method that can optimize the bit-width of the fixed point computations that run on the general purpose hardware. The traditional bit-width optimization methods mainly focus on minimizing the fraction part since the integer part is restricted by the data range. In our work, we utilize the dynamic fixed point concept and the input data range as the prior knowledge to get rid of this limitation. We expand the computations into data flow graph (DFG) and propose a novel approach to estimate the error during propagation. We derive the function of energy consumption and apply a more efficient optimization strategy to balance the tradeoff between the accuracy and energy.

Next, to deal with the floating point computation, we propose a runtime estimation technique by converting data into the logarithmic domain to assess the intermediate result at every node in the data flow graph. Then we evaluate the impact of each node to the overall computation quality, and decide whether we should perform an accurate computation or simply use the estimated value. To approximate the whole graph, we propose three algorithms to make the decisions at certain nodes whether these nodes can be truncated.

Besides the low power and energy efficiency concern, we propose a design concept that utilizes the approximate computing to address the security concerns. We can encode the secret keys into the least significant bits of the input data, and decode the final output. In the future work, the input-output pairs will be used for device authentication, verification, and fingerprint.

APPROXIMATE COMPUTING TECHNIQUES FOR LOW POWER AND ENERGY EFFICIENCY

by

Mingze Gao

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2018

Advisory Committee:
Professor Gang Qu, Chair/Advisor
Professor Shuvra S. Bhattacharyya
Professor Manoj Franklin
Professor Donald Yeung
Professor Yu Chen, Dean's Representative

© Copyright by
Mingze Gao
2018

Dedication

To my family —

my parents Shunyou Gao, Yu'e Cao,

and my wife Qunfang Long

Acknowledgments

I would like to express my sincere gratitude to all the people who teach my knowledge, correct my mistakes and give me the confidence to help me achieve this milestone in my life.

My advisor, Professor Gang Qu, is the one I appreciate first and foremost who lead me into the academic world and guide me to enjoy the time of doing research. Beyond my thesis topic, he is also extremely supportive for me to dig some interesting ideas in another research area. Moreover, my financial security was fully guaranteed among these 6 years. He is now more than an academic advisor but a true friend in my entire life.

I would also like to extend my gratitude to other members of my dissertation committee, Professor Donald Yeung, Professor Manoj Franklin, Professor Shuvra S. Bhattacharyya, and Professor Yu Chen for their time and service. Especially, I would thank Professor Franklin, who offer me the recommendation letter when I applied for this PhD program.

It is also my great honor to work in the same research group with these talented people: Tanvir Arafin, Khai Lai, Carson Dunbar, Xi Chen, Qian Wang, Zhaojun Lu and Omid Aramoon. They not only give me fruitful discussions on research but also stress relieving jokes we shared in our lab.

Last, but by no means least, I owe my deepest thanks to my family who always believe in me, support me, and push me to be the best I can be.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 What is Approximate Computing?

Approximate computing is an emerging computation paradigm that utilizes many applications' intrinsic error resilience to improve the power and energy efficiency. The designer can scarify very little accuracy but achieve the significant savings of power and energy consumption.

Approximate computing has found applications in many fields such as multimedia and signal processing, wireless communication and data mining/analysis. One common feature in these applications is that they can tolerate a certain level of errors in the computation. The errors will either only appear in the middle that not affect the overall computational quality or not significant enough to be observed.

1.2 Why Approximate Computing?

Internet of Things (IoT), big data and Artificial Intelligence (AI) are the most popular topics in the next decade, and all of them demand massive computations. The connectivity in IoT allows the devices or subsystems or Things to collect, process, and exchange data in order to accomplish specific applications more effectively.

It was predicted that the total number of IoT devices will reach 30 to 75 billion by 2020. This explosive growth of Things and their diverse applications are inevitably bringing a crucial challenge: how to power these tens of billions of Things since a significant portion of them are wireless and battery operated, where power consumption directly affects their life time. Similarly, as the data volume grows dramatically and the algorithms become more and more complicated, the power consumption of the infrastructures for big data and AI applications increase significantly.

Many of the low power and energy efficiency techniques developed in the past such as dynamic power and thermal management, dynamic voltage and frequency scaling, and multiple threshold voltage design have been well designed and widely applied. But the power/energy savings become less and less when entering the dark silicon era. These approaches are no longer applicable for smaller and smaller nanometer devices due to the manufacturing limits or the physical characteristics changes, or the potentials are exhausted. To this point, approximate computing is a new methodology that provides a novel thought of designing the low power and energy efficiency system.

More importantly, for some applications, there is no need to pursue the extremely accurate computations because 1) Partial computations have very little/no contribution to the final result. For example, in lots of machine learning and deep learning applications, the classification accuracy is usually not very sensitive to the computation accuracy. 2) The quality loss may not be observed by the naked eyes. Especially in the image processing applications, people most likely will not be aware of the errors that happen in only a few pixels. 3) The input data could be mixed

with noise inherently, such as the sensor data collected from the external sources.

"Does accurate computing ever exist?" We may ask this question to ourselves. The modern computer system is the binary system, which uses the finite number of binary bits to asymptotically approach the reals. However, in most cases, the designers are so eager to pursue the precision and put more and more bits to approximate the reals, causing the "over-designed" issue. In some real-world applications, people prefer to convince themselves to accept the results to be accurate if they cannot tell the flaws. By well designed approximate techniques, we can reduce a large portion of computations with the human acceptable quality loss. In this thesis, we consider the results that computed in single float precision as the golden results.

1.3 Approximate Computing in Different Levels

Generally, approximate computing is achieved by redesigning the system at different levels to save power at the cost of accuracy or correctness of the computations. [17] has done the detailed summary of existing approximate computing strategies and techniques. The author reviewed the existing research in both hardware (e.g. memory, circuit, and different processing units like FPGA, GPU, and CPU) and software, like programming frameworks. In a recent survey [27], state-of-the-art approximate computing approaches are classified at the levels of computing, software, compilers, architecture, memory, and circuit. In this section, we briefly introduce the existing research on these different levels.

Approximate Program: In programming level, the designers usually modify

the original code in order to reduce the computations in the non-critical parts. Lots of techniques have been proposed, such as loop perforation [9], code perforation [10], pattern reduction [11], and probabilistic programming [12].

Approximate Architecture: Approximate architecture design focuses on increasing the energy efficiency of the hardware components, such as the processor and memory, of the main computer system. Researchers have proposed several approximate accelerators [6, 15], approximate ISA [16] and compilers for the specific applications. Memory and storage are usually the two of the most energy-consuming components in the modern computer system due to their persistent power supply. Several approximate memory and storage techniques have been proposed [7, 13].

Approximate arithmetic: In the arithmetic field, the power/energy savings can be achieved by well-designed approximate computing function units such as approximate adders [20, 32] and multipliers [19, 48].

Approximate circuit: Approximate logic seeks to provide fine-grained trade-offs between area-power overhead and insignificant logic error during synthesis [8]. Besides, researchers study the physical features of the circuits and apply the traditional low power design techniques combined with approximate computing design concepts, such as the voltage scaling [14].

1.4 Research Contributions

We focus on utilizing the characteristics of different data formats for arithmetic and numerical approximation. Currently, there are mainly three data formats that

are widely used in the modern computing system, the integer and fixed point format, and the floating point format. For the integer and fixed point format, we propose a dynamic solution and a static solution. In the dynamic way, we propose an Approximate Integer Format (AIF) and corresponding computing mechanism. In the static way, we analyze the data range to statistically select the MSBs during computations.

The approximation for floating point format is very trivial, we can simply reduce the length of mantissa bits. Therefore we go beyond the arithmetic level. We propose a framework that can runtimely truncate the unimportant computations in the data flow graph.

Besides the low power and energy efficiency concern, we propose an approximate floating point based security primitive that enables us to embed information during the process of approximate computing. The hidden information can be generated, embedded, and retrieved for several security applications.

1.4.1 Approximate Integer Arithmetic Design

We propose the Approximate Integer Format (AIF) and its corresponding arithmetic operations using this format in Chapter 2. Fixed point can be treated as the integer format with a globally fixed scaling factor. We define the sentinel bits to indicate the importance of the bits. Then we illustrate how sentinel bits can be used to perform basic arithmetic operations of integer numbers. Error analysis of both operations is provided along with some supporting results. Our proposed

AIF and corresponding computing scheme can be used in both hardware level and architecture level. The experimental results demonstrate that the AIF approximate computing scheme can achieve the huge power savings with little loss in qualities.

1.4.2 Bit-width Optimization for Fixed Point Arithmetic

We propose a novel Bit-width optimization approach for the fixed point applications in Chapter 3. This approach does not rely on the special hardware like AIF. In the traditional fixed point system, the position of radix point is fixed. Therefore, the integer bit-width of an intermediate result is restricted by its data range which will be fixed if the input range is deterministic. So we are only able to optimize the bit-width of fraction part. To break this limitation, we take advantage of the dynamic fixed point format, which allows each intermediate result has its own scaling factor. We compute the data range of every intermediate result using the input range as the prior knowledge. Then we estimate the propagation of error which is a function with scaling factor and bit-width as the variables, and derive the function of energy consumption in terms of bit-width of each computation. We propose an efficient optimization strategy to solve the scaling factor and bit-width through balancing the propagated error function and the energy consumption function. The scaling factors remain invariant if the input ranges and the functionality of the application will not change.

1.4.3 Approximate Data Flow Graph Design

For the floating point system, we proposed a novel runtime estimation technique by taking advantage of the IEEE754 floating point format in Chapter 4. The technique can quickly estimate the results of the computations in logarithmic domain. We also three effective data flow graph truncation algorithms. The algorithms can dynamically truncate the non-critical parts of given data flow graph based on the estimated value with the concern of every input. These approximations effectively leverage considerable energy savings with acceptable computation quality. Besides, the above approach is only used to truncate the topology of the data flow graph instead of approximating each computation. Therefore, we can adopt other arithmetic approximation techniques to achieve additional energy savings.

1.4.4 Approximate Computing for Security Concern

We propose an information embedding approach via approximate computing, which can be used in lightweight authentication in Chapter 5. We utilize the obvious fact that the least significant bits (LSB) have much less contribution to the computational quality than the most significant bits (MSB). With carefully selecting the length of MSB, the designer can control the maximum error rate and guarantee the overall quality. Since most IoT devices and applications are insensitive to the minor computational error, we can apply the approximation and embed the secure bits to the LSBs. The embedded secure bits can be used for multiple security purposes.

Chapter 2: Approximate Integer Format

In this chapter, we utilize the fact that the most significant bits (MSBs) have more contribution than the least significant bits (LSBs) and propose a novel approximate integer format (AIF). AIF will neglect the heading zeros of the operands and dynamically truncate off the LSBs based on the operands' magnitudes in order to shorten the bit-width.

2.1 Design Motivation

For approximate arithmetic computing, many approximate adders [33] [20] [32] and multipliers [19] designs have been proposed. Their key design concept is to apply the accurate computation to the MSBs and approximate the LSBs. But the MSBs and LSBs are split statically and not changing with the operand's value. [33] builds a 16-bit approximate adder that only uses 8-bit accurate adder for 8 MSBs and the 8 LSBs were estimated by the OR gates. In some cases, this design could yield good result:

$$\begin{array}{rl} & 0111\ 1010\ 0111\ 1001_2 = 31353 \\ + & 0101\ 1011\ 1011\ 1000_2 = 23480 \\ \hline = & 1101\ 0101\ 1111\ 1001_2 = 54777 \end{array}$$

The absolute error rate is only 0.102% compared to the correct value 54833. However, if the operands are relatively small:

$$\begin{array}{rl} & 0000\ 0000\ 0111\ 1001_2 = 121 \\ + & 0000\ 0000\ 1011\ 1000_2 = 184 \\ \hline = & 0000\ 0000\ 1111\ 1001_2 = 249 \end{array}$$

The error rate becomes 18.4%! The 8 MSBs are all '0's so the accurate adder has no contribution and obviously the OR gates cannot guarantee the accuracy. The rationale behind this is that a statically designed adder cannot dynamically fit the operands in different magnitudes.

Besides, most of the approximate computing units give limited power or delay savings, mainly due to the large range of data, not realizing the full potential of approximation computing. Consider a 32-bit adder, when both operands have small values or contain many tailing zeros, most parts of the adder will be idle but still consuming power. The existing approximate units cannot be aware of the effective bit-with of given data. On the other hand, if one operand or both are large numbers, lower bits will have little impact on the accuracy of the sum and could be neglected to save power when small error can be tolerated.

We extend approximate computing from the existing "static" approach to a more data-aware "dynamic" fashion in order to achieve more significant savings in the resource such as power. For this purpose, we introduce a segmentation based approximate integer format (AIF) and develop the corresponding basic arithmetic operations that can be used at a low level in almost all applications. In AIF,

operands of the arithmetic operation are segmented into blocks, and the notion of sentinel bits are introduced to indicate the importance of each segment in the computation in terms of accuracy loss and power savings. During the arithmetic operation, sentinel bits are used to truncate and round the less important bits to reduce the bit lengths of operands. Then these approximated operands are fed into the accurate computing units. After the computation, the result is post-processed and converted to the AIF format with its own sentinel bits.

These existing approximate computing units are fully-customized in the sense that they can only be used in either addition or multiplication with given accuracy. One may argue that approximate adders can be used for multiplication. But the error generated by each addition will accumulate, causing uncontrollable errors. Since the proposed AIF is only used to select the non-zero MSBs, and the selected data can be used in any arithmetic operations. Therefore, we don't have to customize the adder, multiplier or the divider.

Besides all integer operations, AIF can be easily extended to fixed point operations, making it applicable to almost all possible applications; and incorporate in high level programming language, giving program the control of accuracy-power tradeoff.

The work in this chapter has been published in 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC) [23].

2.2 Related Work

In this section, we look at some prior approaches to approximate arithmetic computing. We focus on arithmetic units as they form the basic blocks in any application and are the most used sub-systems, saving energy at this sub-system results in saving in the entire system.

Kim et al. [34] propose a carry skip scheme in which the input is broken into segments and the carry-in for each segment is speculated using only a preceding subset of bits and neglecting the rest. Hashemi et al. [19] propose a segmentation based multiplier where a smaller accurate core multiplier is used whose inputs are decided by truncating the input bits. The subset bits of the operand are decided by looking for the leading one in the input and selecting the subsequent k-2 bits (k being the accuracy factor). Mahdiani et al. [33] propose an approximate computation block designs that target Neural Networks and fuzzy logic. They use a Lower Part OR Adder which divides an n-bit addition into two smaller parts, the most significant bit part undergoes precise addition while the lower part is ORed to approximate the least significant result bit. Ye et al. [20] propose a re-configurable approximate adder design which allows for adjustment of accuracy with trade-off with power consumption. The input operand bits are divided into blocks of size k. The carry-in for each block is selected from a multiplexer logic choosing between the speculated carry-in and computed carries from the lower significant bits. Zhu et al. [32] propose an error tolerant adder based on truncation of input and extend it in [35]. Their mechanism is to split the input into an accurate part and an inaccurate part. For

the inaccurate part, they check every bit position from left to right and perform regular one bit addition if the two input bits are different or if both are zero. If both the bits turn out to be 1, the traversing stops, and all sum bits from that bit to the least significant bit are set to 1. This mechanism eliminates the whole carry chain for the lower bit orders which reduces the delay and the power consumed. Hu and Qian [36] propose an approximate adder with error reduction. They use a carry speculation strategy in the carry generator block where the most significant bit of the previous block is used as the carry-in to the carry generator block. Therefore, the carry-out from the carry generator is an approximate value.

2.3 AIF: Definition and Error Bound

2.3.1 Operand Segmentation

Our proposed approximate integer format is based on the segmentation of operands. An n-bit positive integer operand N is segmented into B = $\lceil n/k \rceil$ blocks, and with each block contains k bits per block.

Definition 1: A valid block in a positive number is a block that has at least one '1' before or inside it.

Definition 2: The ith sentinel bits st of a number is defined as

$$st[1] = \begin{cases} 1, & \textit{if block } i \textit{ is a valid block} \\ 0, & \textit{if block } i \textit{ is an invalid block} \end{cases}$$

Definition 3: The precision control value 'pc' is the number of valid blocks, from the leftmost one, that will be used in the computation.

For example, when we choose $k = 4$, both $1500_{10} = 0000, \underline{\mathbf{0101,1101}}\mathbf{,1100}_2$, and $800_{10} = 0000, \underline{\mathbf{0011,0010}}\mathbf{,0000}_2$ have three valid blocks. So their sentinel bits are 0111. If we set $pc = 2$, the two underlined valid blocks will participate in the approximate computation.

The key idea of our integer format design is to use only the most important blocks of the data for computation and ignore the rest. To compensate the error, we provide two rounding techniques for the four fundamental arithmetic operations: addition/subtraction and multiplication/division.

2.3.1.1 Classic Rounding

Definition 4: The classic rounding of a number N from the ith least significant bit means that adding the ith bit to the (i+1)th bit and then setting the ith bit and bits to its right to zero.

For example, rounding $N = 263_{10} = 0000, 0001, 0000, 0111_2$ from the 3rd least significant bit changes it to $0000, 0001, 0000, 1000_2$ and from the 4th bit changes it to $0000, 0001, 0000, 0000_2$.

Lemma 1: The error rate of classic rounding by keeping only the 'pc' most significant valid blocks is less than $\varepsilon_r = \frac{1}{2^{k(pc-1)+2}}$

Proof: Consider a number N that can be represented as $b_{m-1}b_{m-2}b_1b_0$ using m blocks, where each block contains k bits. Obviously if b_q is the first valid blocks

($q \in [0, m-1]$), then the blocks from b_{m-1} to b_{q+1} are all '0' or invalid. The smallest value of N is "00...100...0" or $N^{'} = 2^{k(q-1)+1}$. When we pick 'pc' valid blocks and rounding off the blocks from $b_i (i = q - pc)$ to b_0, the largest error introduced with be $R = 2^{ki-1}$ when blocks "100...0" are rounded up and then reset to "000...0". So the maximum round-off error rate is bound by $R/N^{'} = \frac{1}{2^{k(pc-1)+2}}$.

In the above example when $N = 263_{10}$, If we pick the first two valid blocks ($pc = 2$) for rounding, then $N^{'} = 256_{10}$ and the error is $363 - 256 = 7$. The error rate or relative error is

$$\frac{262 - 256}{256} = 2.734\% < \frac{1}{2^{4(2-1)+2}} = 3.125\%$$

Classic rounding is suitable for multiplication and division operations. Because about half of the time, it needs one extra addition operation (when b_i is 1). The power consumption of this extra addition is insignificant for multiplication and division, but may not be negligible for addition and subtraction. Therefore, we propose the following efficient rounding technique which is suitable for addition/subtraction.

2.3.1.2 Efficient Rounding

Definition 5: The efficient rounding of A+B at the ith bit is $A_{trunc} + B_{trunc} + Cin_{round}$, *where* A_{trunc} *and* B_{trunc} *are obtained by truncating the i least significant bits from A and B, and* $Cin_{round} = (A_i \& B_i)$, *the logic AND of the ith bits of A and B.*

Fig.2.1 below provides a simple example of efficient rounding where we choose

to truncate the last i=8 bits.

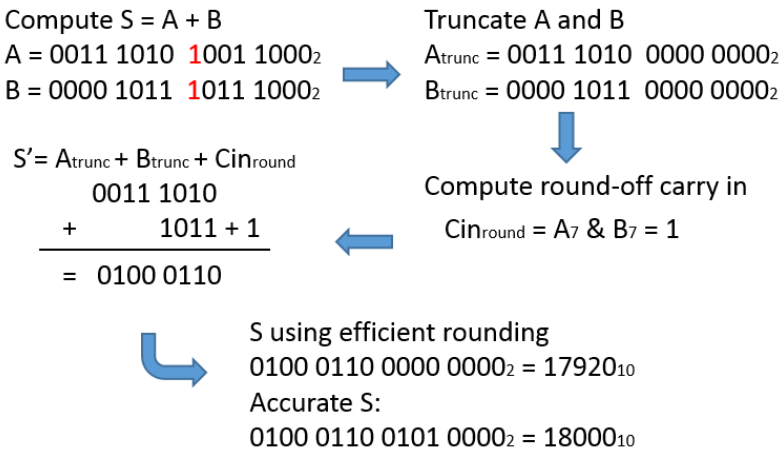

Figure 2.1: AIF addition with efficient rounding

Lemma 2: The largest error introduced by efficient rounding at the ith bit is $2^i + 2^{i-1} - 2$.

Proof: Clearly this happens when one operand ends with "111...1" and the other ends with "011...1".

Efficient rounding is suitable for addition and subtraction because it requires only one additional AND operation. As a comparison, the classic rounding may need one extra addition for each rounding. So it may need three such additions to compute A+B (or A-B). However, classic rounding is unbiased and efficient rounding is biased (always underestimates).

2.3.2 AIF: Definition

In our approach, data is rounded and stored based on its sentinel bits in the so-called Approximate Integer Format (AIF). We illustrate this with a 4-block operand $A = b_3b_2b_1b_0$. There are five possible values of A's sentinel bits st_a: 0000, 0001, 0011,

0111, 1111. For the first four cases, the data A will be stored in following format (no loss of data because b_3 is a 0-block):

st_a	b_2	b_1	b_0

In the last case when st_a equals to 1111, the data A is store as:

st_a	b_3	b_2	b_1

In this case, we drop the last block b_0 and it will introduce error when the block is not 0. However, since the first three blocks (particularly the first and the most significant block) are all valid, this error will be small and its upper bound can be easily obtained similar to Lemma 1.

Starting from the leftmost bit, if the data is partitioned into B blocks. The ith sentinel bits is given by Eq.2.1, ignoring the last term $st[i+1]$ when computing the first bit st[B-1]:

$$st[i] = b_i[0] \mid b_i[1] \mid ... \mid b_i[B-1] \mid st[i+1] \tag{2.1}$$

Eq.2.1 can be implemented as an OR-logic tree. Even though it looks like computing st consumes lots of OR gates, the overhead is very low because: 1) each piece of data only needs to be computed once; 2) hardware implementation can be very efficient; and 3) the computation can be arranged in pipeline when fetching data from memory to register, thus reducing the timing cost

2.4 AIF Computing Mechanism

In this section, we will illustrate how to use the sentinel bits and the AIF to perform approximate addition and multiplication. Subtraction and division can be done with similar methods.

2.4.1 Approximate Addition

The following algorithm shows how to add two operands A and B, which are both stored in approximate data format.

1. Compute the sentinel bits of the result S: $st_s = st_A \mid st_B$;
2. Suppose the leftmost '1' in st_s is in $st[i]$, and we plan to pick pc valid blocks, truncate ith to $(i-pc+1)$th blocks of A and B to obtain A' and B', respectively;
3. Compute $S' = A' + B'$ and $Cout$;
4. Update st_s by $st_s[i+1] = Cout$;
5. Reformulate S in AIF using st_s and S'. Padding 0's if necessary.

In step 3, instead of using the whole length of data, we choose several most significant bits in A and B and do accurate addition of these bits. For example, if we partition the original 16-bit wide data A and B into 4 blocks and we only pick 2 left most valid blocks (pc = 2), we can use a 8-bit full adder instead of a 16-bit adder. This brings the approximate addition reducing the power and delay cost almost by a factor of 2 (see the illustrative example in Fig.2.2).

In step 4, the sentinel bits of the result st_s may change if the carry out the

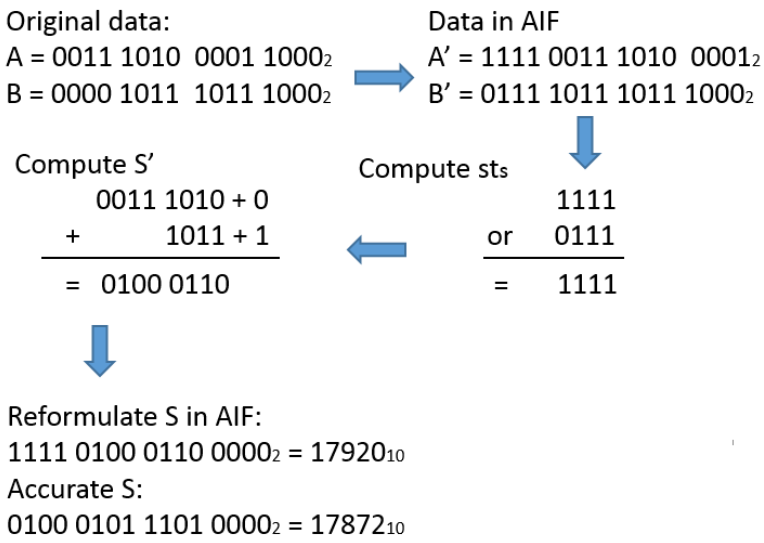

Figure 2.2: An illustrative example of approximate addition in AIF

addition is not zero. Since in step 2 we define that the leftmost leading '1' in st_s is in ith position, so $st[i+1]$ must be '0'. So we let $st_s[i+1] = Cout$, if $Cout$ is '0', nothing changes; if there is $Cout = 1$, $st[i+1]$ will be set to '1'. Notice that if the ith bit is the leftmost and $Cout = 1$, overflow occurs.

Theorem 1: An upper bound of the error rate of the proposed approximate addition in AIF is $2er$*, where* er *is the maximum truncation error rate of the larger operand in step 2.*

Proof: Consider $A + B$ with $A \geq B$ and let er and er_B be the error rates of rounding of A and B, respectively. The error rate of addition will be $Er = \frac{A*er+B*er_B}{A+B}$. Let $B = p * A$, where $p \in (0, 1]$, we have $Er = \frac{A*er+p*A*er_B}{A+p*A} = \frac{er+p*er_B}{1+p}$.

When $p \in (0, er]$, because $er_B < 1$, $Er < \frac{er+p}{1+p} < \frac{2er}{1+p} < 2er$.

When $p \in [er, 1]$, truncation of B will bring the same amount of maximum error with A because both A and B are truncated same number of bits. So $B*er_B = A*er$.

Then $Er = \frac{2*A*er}{A+p*A} = \frac{2er}{1+p} < 2er$.

In the proof, we have given a tighter upper bound $2er/(1+p)$. $2er$ is more practical and easy to estimate from Lemma 1. Apparently, er depends on the value of pc in step 2 and a large pc gives high accuracy.

2.4.2 Approximate Multiplication

In multiplication, each bit in one operand will be multiplied by every single bit in the other operand. So we don't need to truncate the operands at same position as we did for addition. Instead, we truncate them to be the same length and use the following approximation algorithm:

1. Round the leftmost pc valid blocks of A and B into A' and B';
2. Compute $S' = A' * B'$;
3. Compute sentinel bits st_s using st_A and st_B and carry out;
4. Shift and reformulate S in AIF using S' and st_S. Pad 0s if necessary.

In step 2, we use the accurate multiplier with a smaller bit-width. Suppose both A and B are n-bit numbers. After truncating, A' and B' will have $pc * k < n$ bits. So instead of using an n-bit multiplier, our approximation algorithm only requires a $pc * k$ bit multiplier. This is the main source of power reduction in our proposed approximate multiplication.

In step 4, we can also use "routing the interconnects" instead of a shift register. In step 3, computing the sentinel bits is also simple. If there are n_A valid blocks

in A and n_B valid blocks in B, then ns, the number of 1's in st_s, equals to: $n_s = n_A + n_B - 1 + Cout$

Theorem 2: The error rate of our approximation multiplication algorithm is $er_A + er_B + er_A * er_B$, where er_A and er_B are the truncation error rate of A and B respectively.

Proof:

$$\begin{aligned} Er &= \frac{(A * er_A + A) * (B * er_B + B)}{A * B} - 1 \\ &= (er_A + 1) * (er_B + 1) - 1 = er_A + er_B + er_A * er_B \end{aligned}$$

Practically, $er_A << 1$ and $er_B << 1$, so $Er \approx er_A + er_B$

2.5 Beyond Positive Integer Operations

The above discussion is restricted to AIF based approximate computing for positive integers. In this section, we will discuss its extensions.

2.5.1 AIF for Negative Number

Transforming a negative number from the original into approximate data is slightly different with the positive number. For negative number, we cannot use Eq.2.1 to compute the sentinel bits because in most popular representations for negative numbers (such as two's complementary, one's complementary, and sign-and-magnitude), the leading bit is a '1'. So we need to re-define the valid block for

negative number based on its representation. For example, for two's complementary, we have:

Definition 6: A valid block in a negative number is a block that has at least one '0' before or inside it.

To compute the sentinel bits for negative number, we can simply replace "|" (the logic OR) with "&" (logic AND) in Eq.2.1. The conversion to AIF and the corresponding approximate addition and multiplication can be performed similarly to what we have explained for positive numbers.

2.5.2 Fixed Point Number Arithmetic

The AIF is not suitable for floating point number due to its intrinsic feature. However, in most signal processing systems, fixed point format can achieve sufficiently high accuracy while providing power savings. Even though our proposed approximate data format is designed for integer, it is also suitable for fixed point numbers. Recall that all the arithmetic operations on fixed point numbers are the same as integer operations, the position of the decimal point only matters in cases when we want to output the results. Therefore, the proposed approximate arithmetic operations for integers can be used for fixed point number arithmetic as well.

2.5.3 Compute in Caution

Similar to existing approximate computing methodologies, AIF should also be deliberated when used in the following circumstances:

1. Condition criterion, e.g. if, while condition

2. Data value that the result is very sensitive to

3. Functions that have periodical property, e.g. sin, cos, and modulo operation.

2.6 Experiment Results

In this section, we first compare the power and delay of adders and multipliers with different bit width. Then we demonstrate that the overhead of computing the sentinel bits is trivial compared to the power savings they bring us. Next, we use an example, generating the Fibonacci sequence, to verify the accuracy of the proposed AIF based approximate computing. Finally, we apply to five well-known applications in signal and image processing and machine learning: IDCT, FFT, Kmeans, kNN, and SVM.

We build the circuits and approximate computing engine in Verilog and synthesis them using Cadence RTL Compiler with FreePDK 45nm library. To evaluate the approximation qualities, we implement and modify these algorithms in Matlab. Since our designed AIF is suitable only for integers and fixed point numbers, we implement the benchmarks using 32 bit fixed point data format instead of floating point.

2.6.1 Circuit Overhead Comparison

Table.2.1 shows the hardware design parameters for the approximate computing units (adders and multipliers of different bit width). Shaded area indicates the

	cells	area	power(nW)
8-bit checker	10	23.93	61475.68
16-bit checker	17	39.89	132145.68
8-bit adder	91	212.12	752614.84
16-bit adder	230	322.33	2234652.24
32-bit adder	498	1116.93	4818821.94
8-bit multiplier	377	1037.62	2829745.1
16-bit multiplier	1406	4208.68	10815807.06
32-bit multiplier	4916	15126.01	34033690.3

Table 2.1: Design comparison of approximate computing units.

value normalized to that for the 32-bit multiplier. A k-bit checker is the circuitry that exams whether the given k bits contain any '1'. This is used to compute the sentinel bits. This table reveals that:

1. The area and power consumption of adders increases with the bit width linearly.

2. The area and power consumption of multiplier increase approximately with the bit width by a factor of 2.

3. Compared to the adders and multipliers, the area and power overhead of the checker is negligible.

So if we reduce the bit width of each operand, we can allocate computation into smaller adders and multipliers. The power/delay/area savings will be tremendous.

2.6.2 A Simple Example: Fibonacci Sequence

In Fibonacci sequence, a term is the sum of the previous two terms. It is a good benchmark to test whether an approximate method is biased. If it is, the error will accumulate quickly to create uncontrollable error rate.

#	pc=2	pc=3	pc=4	#	pc=2	pc=3	pc=4	#	pc=2	pc=3	pc=4
1~13	0	0	0	22	-0.02732	3.49E-05	0	31	-0.02291	-0.00035	2.30E-06
14	-0.00984	0	0	23	-0.02692	0	0	32	-0.02267	-0.00059	-8.51E-06
15	-0.01317	0	0	24	-0.02707	1.33E-05	0	33	-0.02276	-0.0005	-4.38E-06
16	-0.01691	0	0	25	-0.02912	0.000404	8.24E-06	34	-0.02273	-0.00053	-5.96E-06
17	-0.01858	0	0	26	-0.03225	0.000336	1.02E-05	35	-0.02274	-0.00052	-5.36E-06
18	-0.01985	0	0	27	-0.0375	0.000362	9.44E-06	36	-0.02274	-0.00052	-5.59E-06
19	-0.02173	-0.00044	0	28	-0.03947	0.000352	9.72E-06	37	-0.0328	-0.0001	1.05E-06
20	-0.02905	0.000183	0	29	-0.04118	0.000356	9.61E-06	38	-0.03621	-0.00046	-5.53E-06
21	-0.02625	-5.65E-05	0	30	-0.04205	0.000354	9.66E-06	39	-0.04003	-0.00064	-3.02E-06
								40	-0.04174	-0.00077	-3.98E-06

Table 2.2: Fibonacci sequence error of first 40 terms using approximate addition under different configuration

Table.2.2 lists the error of the first 40 terms in the Fibonacci sequence compute by approximation addition with different parameters. In this example, we partition data into 8 blocks, and pc is the number of valid blocks participating the computation which varies between 2 and 4. From the table, we see that the errors are very small and relatively unbiased. In addition, the accuracy increases as pc value increases. When pc=4, we correctly generate the first 24 terms with no errors. This is better than any of the approximated methods reported.

2.6.3 Real World Applications

We implement 5 well-known applications to evaluate our AIF based approach. In IDCT application, we take the discrete cosine transformed image as the input and use IDCT to recover it. We compare the recovered image with the original image. The error metric we use is PSNR (Peak signal-to-noise ratio). Fig.2.3 shows the different results under different configurations. "32_8_2" means we use the 32-bit fixed point data format, and split the data into 8 blocks. During computing, we only pick 2 valid blocks. From the figure, we can barely see the quality decades

when picking less valid blocks. There are two reasons: 1) The image data will be transformed into 8-bit unsigned integer during visualization. The visualization process will automatically round the data. Our approximate computing method generates different image data but the difference is reduced when visualization. Thats why all the figures have good qualities but their PSNRs are different. 2) Most importantly, this application can tolerant rather high error rate, giving us large design space for approximation computing.

(a) 32_8_2, PSNR = 41.7579 (b) 32_8_4, PSNR = 89.6398

(c) 32_8_6, PSNR =116.2949 (d) original image

Figure 2.3: IDCT approximate result comparison.

We also implement a 16-point FFT. The corresponding error metric is ARES (Average Relative Error Signicance). In the table, the error dramatically decreases when selecting more valid blocks. For Kmeans application, we define the error

metric as the percentage of miss-clustering points. If we pick 4 valid blocks of the data, there will be no miss-clustered points. But we can reduce the power by almost half. kNN and SVM are two most frequently use classifiers in machine learning field. In this chapter, we implement a single-NN classifier (k=1) and a straightforward SVM classifier with linear kernel. We use the MNIST data (http://yann.lecun.com/exdb/mnist/) to evaluate the classification accuracy. Since kNN will take very long time to classify the whole MNIST data, to speed up the experiment, we only use part of the data. The accurate kNN will achieve 93.2% accuracy. The approximation approach almost keeps the same accuracy when decreasing the number of picked valid blocks. We use the whole data to evaluate our SVM classifier. Our SVM is a very straightforward version without optimization. So the accuracy is not as high as current state-of-art SVM. However, our goal is to evaluate the accuracy of the approximation approach, we only need to compare the results within our own implementation. From the table, we can see that there are some minor decreases in accuracy, but the approximation is still worthy considering the large amount of power savings.

AIF modules	IDCT	kNN	FFT	Kmeans	SVM
32_8_2	41.7579	92.9%	0.0081	1.125%	60.94%
32_8_3	60.6398	93.2%	2.79e-4	0.125%	85.11%
32_8_4	89.8324	93.1%	1.61e-5	0%	85.98%
32_8_5	112.0872	93.2%	3.02e-6	0%	85.59%
32_8_6	116.2944	93.2%	7.11e-8	0%	85.42%
baseline	116.2949	93.2%	0	0%	85.42%
Error Metric	PSNR	Classification Accuracy	ARES	Mis-clustered%	Classification Accuracy

Table 2.3: Accuracy comparisons of different applications under different configurations

Table.2.3 shows the qualities of different applications under different configurations. We can clearly see that our approach guarantees the result qualities.

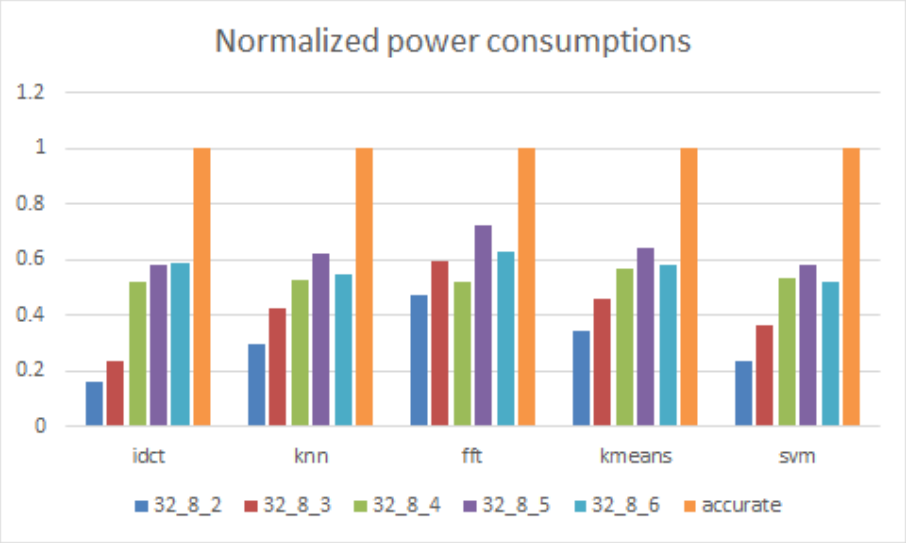

Figure 2.4: Power consumption comparison.

Fig.2.4 lists the power consumption of different configurations. It is very obvious that our approximation approach can largely save the power. The power consumption does not monotonically increase if we continuously pick more valid blocks. This is because that the less block we pick, the more complicated the data selection logic is. Even though fewer blocks consumes less power in computation, but it requires more logic circuit to select data, introducing extra power consumption. Besides, the power consumption is only compared with the fixed point computation. If compared with floating point, our approach will save more power.

2.7 Future Work

In the future, we plan to implement AIF and corresponding computing scheme at higher levels (such as architecture, ISA or software) by introducing appropriate

instructions and data type. This provides an interface for users to control the tradeoff of computation accuracy and resource in the approximate computation paradigm. For example, we can customize the architecture and corresponding ISA of the processor, allowing the approximate operations like:

```
APXLD   F0, 0(R1)
APXLD   F2, 0(R2)
APXADD  F0, F0, F2
APXSD   0(R1), F0
```

Further, we can define apxint as the AIF data type and incorporate it in customized compiler to allow programmer write high level codes as follows:

```
int main(){
    apxint a = 2341;
    apxint b = 546;
    apxint sum = a + b, prod = a * b;
    ...
    return 0;
}
```

Chapter 3: Bit-width Optimization for Fixed Point Arithmetic

In this chapter, we propose a method that utilizes the dynamic fixed point feature to optimize the bit-width of the operands, thus saving the energy consumption. We use data range as the prior information and only select the non-zero MSBs during the computation. Unlike the AIF in chapter 2, this method does not additional hardware support. Our approach reads the software code and transforms it into data flow graph. The optimized data flow graph can also be re-synthesized back to software code, which can be run on any general-purpose platform.

3.1 Design Motivation

The traditional fixed point is represented as $Q_{IB.FB}$, where IB and FB are the number of bits in the integer and fraction part, respectively. In this format, the data range is determined by the value of IB. IB will be fixed if the range is deterministic. FB determines the precision. The fewer FB incurs larger error but saves energy and resource. Therefore, the previous research [41–43] focused on the optimizing the value of FB to obtain lower resource cost with guaranteed accuracy.

However, traditional fixed point format will be less efficient in the computations if the data is within a certain range. For example, if a number is much smaller

than 1, some leading bits in FB will be '0', contributing nothing during the computation. If a number is much bigger than 1, the fraction part will be very insignificant for the result. The source of above inefficiency is that $Q_{IB.FB}$ must reserve a position for radix point regardless the data value. The dynamic fixed point [44] overcomes this drawback by ignoring the radix point. In this format, a given number x can be reshaped to $x = x' * 2^e$, where x' is all the effective bits in integer format, and e is the scaling factor.

The researchers utilized the dynamic fixed point to implement the energy efficient digital filters [44], or train the convolutional neural networks (CNN) with fewer bits [31, 45]. However, they only focused on optimizing some specific applications, but failed to provide a generalized method that applicable for all kinds of computations. For example, the core operations of the applications in [31, 44, 45] are matrix multiplications, where all the data inside a matrix can share a same scaling factor. Therefore, the overall number of scaling factors is very limited and can be iteratively searched and updated (as shown in algorithm 2 in [31]). However, if an application consists of the irregular order of different types of arithmetic operations, we will fall into a dilemma: 1) It is infeasible to group a series of contiguous computations to share the same scaling factor. 2) If allowing each intermediate result has its own scaling factor, allocating the proper scaling factor to each operation with the consideration of both energy efficiency and accuracy will be very difficult. In this chapter, we break the dilemma by solving the problem 2). Our goal is: using the input data range as prior knowledge, allocate the bit-width of x' and the proper value of e for every intermediate result, in order to effectively balance the tradeoff

between the energy cost and the computational accuracy.

To achieve above goal, we first transform the computations into data flow graph (DFG). A data flow graph (DFG), also known as a computational graph, is a directed graph which represents the data dependencies and flow directions between a number of operations. Data flow graph has multiple variants that do not need to be in the real shape of a directed graph. Abstract Syntax Tree (AST) which is used in [21], can be converted to DFG easily. Besides, a slice of software program can also be treated as a DFG where the nodes are the computational operations and the edges are the corresponding variables or intermediate results. Therefore, our proposed approach is transplantable in any other software program variants.

Next, we apply the range analysis to compute the data range of each node in the DFG. Then we use the standard deviation to quantify the error and derive the error function using propagation of uncertainty theory. We generate the energy consumption function by measuring the energy cost of computing each bit under different operations in the synthesized circuits in Cadence. Finally, we solving the optimization problem by setting the constraints of energy to find the best bit-width and the scaling factor assignment. Compared with the existing approaches, the dynamic fixed point gets rid of the radix points, leveraging more optimization space and yielding more energy efficient allocation. Besides, the previous approaches [24, 41] only consider the error propagation of addition/subtraction and multiplication, but our approach can estimate the error of any operation as long as it is differentiable.

The bit-width and scaling factor will be fixed during the execution time. One may argue that the fixed scaling factor and bit-with mean selecting the fixed position

of the operands during the computations, which may contradict the "dynamic" concept we emphasized in Chapter 2. But this approach still stands for two reasons: 1) We analyze the data range of each computation. By using the range, we can pre-decide the non zero MSBs of the operands. Consider the example used in Chapter 2 Design Motivation section, that approximate design will be acceptable if we know those two operands are always big enough as in the first case. 2) The AIF needs additional hardware support. Besides, if we want to implement AIF in high level, we need to modify the architecture support and corresponding ISA and compilers. But the bit-width optimization approach can be easily implemented into programming level without any special demand of hardware platform.

3.2 Related Work

Bit-width optimization problem has been well discussed in last decade. [41] proposes a framework called MiniBit to optimize the bit-width. The authors first apply affine arithmetic to obtain more accurate range analysis than interval arithmetic. Then they use the data range to minimize the integer part (IB). The fraction part (FB) can be optimized by solving the error bound objective function under given constraint. [42] solves the bit-width optimization using divide and conquer approach. They recursively break down the given data flow graph into several subsets and find the pareto optimal solutions to each sub-design. [43] provides a novel error propagation model. They exploit the statistical analysis of the quantization noise coupled with the Additive White Gaussian Noise (AWGN) models. They claim that

using these new model, they can significantly speedup the optimization process of FFT compared with the simulation-based approach.

Most of the bit-width optimization approaches use range analysis as the pre-process to generate the prior knowledge of each node in DFG. The most straightforward approach is Interval Arithmetic [46]. The estimation process of Interval Arithmetic is very fast, but it fails to give the tight bound if the two input variables are correlated. To fixed this problem, Affine Arithmetic [47] is proposed to keep track of the correlations of the input variables to narrow down the possible range.

The bit-width optimization problem can be considered as a sub-problem of approximate computing, whose main goal is to balance the tradeoff between the result quality and computational efforts. More generally in arithmetic fields, researchers are building approximate arithmetic primitives (e.g. approximate adder [20], approximate multiplier [48]) to obtain the energy/power savings.

3.3 Computation in Dynamic Fixed Point Format

In this section, we first give a brief introduction of dynamic fixed point format. Then we use a motivational example to illustrate how we take advantage of this format to achieve significant energy saving. Next, we list the properties and restrictions of this format during the computations.

3.3.1 Preliminary: Dynamic Fixed Point Format

Like $Q_{IB.FB}$ denotes the traditional fixed point format, we use F to represent the dynamic fixed point format.

*Definition: $F_{w,e}$ is the fixed point format of a **signed** number x which can be written as $x = x' * 2^e$, where $w-1$ denotes maximum bit-width of scaled number x' and e is the scaling factor.*

The representation range is in $(-2^{w-1+e}, 2^{w-1+e}]$ with the resolution of 2^e. If a number is 101_2 present in $F_{4,-1}$, then its actual value will be $101_2 * 2^{-1} = 10.1_2 = 2.5_{10}$.

If e is a globally fixed value of all the numbers, F will perform just like the traditional fixed point format Q. On the other hand, if we allow e updating during each operation, F will perform like the floating point format. In our design, we let e to be fixed for an individual operation but independent to others.

3.3.2 A motivational example

Consider the example in Fig.3.1. The range of operands a and b are far apart. In the traditional format, if we want a to be represented more accurately, FB_a need to be large because the leading 9 bits in FB_a will be '0'. On the other hand, the value of b is much bigger than a. Therefore, b's fraction part contributes very less in the multiplication. Maintaining large FB_b is extremely inefficient.

If we ignore the radix point and use 7 bits (including 1 sign bit) to approximately represent a and b, their format can be written as $F_{7,-15}$ and $F_{7,5}$ respectively.

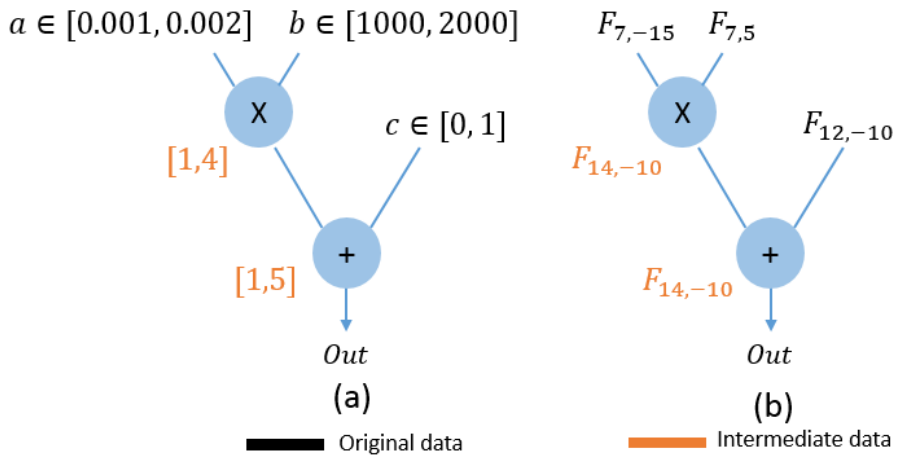

Figure 3.1: A motivational example of using dynamic fixed point format

Let:

$$a = 0.0011903 \approx 0100111_2 * 2^{-15} = 0.00119$$

$$b = 1612.1234 \approx 0110010_2 * 2^{5} = 1600$$

$$c = 1.2344 \approx 010011110000_2 * 2^{-10} = 1.234375$$

Then the error rate of *Out* will be only 0.15%. We can obtain a very low error rate by using only 7 bits in multiplication. If we force all the numbers to obey $Q_{12.12}$ format, the error rate becomes 10.9%. We have the larger error rate and use much more bits in the computations.

3.3.3 Properties and Restrictions

Once the format F is decided for each node in the DFG, it will be fixed during the computation no matter what the inputs will be. Therefore, w and e needs to be carefully selected. Assume a node x is in the range of $[\underline{x}, \overline{x}]$. To avoid the overflow,

the inequality 3.1 must be satisfied:

$$2^{w-1} * 2^{e} \geq max(|\underline{x}|, |\overline{x}|) \tag{3.1}$$

Since our goal is to minimize the number of bits participating into the computation, we can force Eq.3.1 to be an equality. w and e are integers, therefore, we can write Eq.3.1 as:

$$w + e = \lceil log_2(max(|\underline{x}|, |\overline{x}|) \rceil + 1 \tag{3.2}$$

In Fig.3.1.(a), the derived data ranges are in orange color. In Fig.3.1.b, their corresponding F obey the Eq.3.2. We scale c by 2^{-10}, this is because we cannot directly add two number if their scaling factors are different. Given more general case, let $a = a' * 2^{e_a}, b = b' * 2^{e_b}, c = c' * 2^{e_c}$. The example data flow graph can be executed as:

$$Out' = (a' * b') \gg (e_a + e_b - e_c) + c' \tag{3.3}$$

Then the final output will in the form of $Out = Out' * 2^{e_c}$. Since the e of each node will be pre-decided and remains constants during the computation, we can not only synthesis the DFG in hardware level, but also implement it using the software program using Eq.3.3.

The rationale behind F is that we select the most significant effective bits for each operand based on the bits distribution. Compared with [23] which truncates the bits dynamically, our approach takes full advantage of the data ranges and does not need any specific hardware support. The bit-width savings, as well as the energy

savings, comes from the disparity of the operands' range. If a, b and c are all in $[0, 1]$. Then $e_a = e_b = e_c$. This special case under our F based design exactly equals to the traditional fixed point design.

$F_{w,e}$ can be converted to $Q_{w+e.-e}$ if $w + e > 0$ and $e < 0$, as Q does not get rid of the radix point. In F, we do not have these limitations, so we can get a larger design space. If we fix the bit-width w and allow the scaling factor e to change dynamically with the data, the proposed data format will be like the floating point, where $w = 24$ will be the single precision floating point in IEEE754 standard.

3.4 Error Estimation

In this section, we will derive the error function with e as the variable to show how the finite bit-width will affect the final result. The derived error function can either be limited by the user-defined error bound as the constraint for the energy minimization or be considered as the objective function, where we can try to minimize the error given the constraint of energy consumption.

As we introduced in Section 2, the previous research [24, 41] consider the absolute error in the worst case for each operation. However, 1) To guarantee the worst case, the absolute error is most likely over-estimated. Besides, the chance of the worst case happens could be very rare. Use it as the constraint may sacrifice the optimization space. Besides, the absolute error cannot give us the sense of how the errors will vary or disperse under a set of inputs. 2) Their approaches only consider the addition/subtraction and multiplication, and are not suitable for the division

and other complex function like log or cosine.

3.4.1 Propagation Error

We propose an error propagation and estimation method that well addresses the above two issues. We track the standard deviation of the error during the propagation. Given a function $out = f(a, b)$, the standard deviation of out is:

$$\sigma_{out}{}^2 = \left|\frac{\partial f}{\partial a}\right|^2 \sigma_a{}^2 + \left|\frac{\partial f}{\partial b}\right|^2 \sigma_b{}^2 + 2\frac{\partial f}{\partial a}\frac{\partial f}{\partial b}\sigma_a\sigma_b\rho_{ab} \tag{3.4}$$

where σ_a and σ_b are the error standard deviation of input a and b, and the ρ_{ab} is the correlation coefficient. From 3.4, we can easily compute the standard deviation as long as function f is differentiable.

3.4.1.1 Correlation Coefficient Approximation

For the easier computation, if we can eliminate the cross term in inequality 3.4, the output variance will become the linear combination of all the input variances. Let's consider the following two cases.

1. The correlation coefficient is 0

$\rho_{ab} = 0$ means the inputs a and b are independent. In the data flow graph, if a node a and node b are independent, it means $a.predecessors \bigcap b.predecessors = \varnothing$, where $a.predecessors$ is the set of nodes which lay on the path from primary inputs

to node a. Then equation 3.4 becomes:

$$\sigma_{out}{}^2 = \left|\frac{\partial f}{\partial a}\right|^2 \sigma_a{}^2 + \left|\frac{\partial f}{\partial b}\right|^2 \sigma_b{}^2 \tag{3.5}$$

2. The correlation coefficient is not 0

$\rho_{ab} \neq 0$ means $a.predecessors \bigcap b.predecessors \neq \varnothing$. Then the Eq.3.4 can be approximate to:

$$\begin{aligned}\sigma_{out}{}^2 &\leq \left|\frac{\partial f}{\partial a}\right|^2 \sigma_a{}^2 + \left|\frac{\partial f}{\partial b}\right|^2 \sigma_b{}^2 + 2\frac{\partial f}{\partial a}\frac{\partial f}{\partial b}\sigma_a\sigma_b \\ &\leq 2\left|\frac{\partial f}{\partial a}\right|^2 \sigma_a{}^2 + 2\left|\frac{\partial f}{\partial b}\right|^2 \sigma_b{}^2\end{aligned}$$

It has the factor 2 compared with the Eq.3.5.

3.4.1.2 Case Studies for Different Operations

In this chapter, we list the cases of the three most commonly used arithmetic operations: addition, multiplication, and division. The similar approach can be used to analyze some complex functions like *log*. In the following cases, we assume the input $a \in [\underline{a}, \overline{a}]$ and $b \in [\underline{b}, \overline{b}]$. We use Eq.3.5 for simplicity.

1. Addition, $out = a + b$

$$\frac{\partial out}{\partial a} = \frac{\partial out}{\partial b} = 1$$

$$\sigma_{prop}{}^2 = \sigma_a{}^2 + \sigma_b{}^2$$

2. Multiplication, $out = a * b$

$$\begin{aligned} {\sigma_{prop}}^2 &= b^2 {\sigma_a}^2 + a^2 {\sigma_b}^2 \\ &\leq b_{max}^2 {\sigma_a}^2 + a_{max}^2 {\sigma_b}^2 \end{aligned}$$

$$where \quad a_{max} = max(|\underline{a}|, |\overline{a}|), \quad b_{max} = max(|\underline{b}|, |\overline{b}|)$$

3. Division, $out = a/b$

$$\begin{aligned} {\sigma_{prop}}^2 &= \frac{1}{b^2} {\sigma_a}^2 + \frac{a^2}{b^4} {\sigma_b}^2 \\ &\leq \frac{1}{b_{min}^2} ({\sigma_a}^2 + out_{max}^2 {\sigma_b}^2) \end{aligned}$$

$$where \quad b_{min} = min(|\underline{b}|, |\overline{b}|), \quad out_{max} = max(|\underline{out}|, |\overline{out}|)$$

The range of a, b, and out will be pre-decided by the range analysis, so the above equations only have the σ^2 as the unknown variable.

3.4.2 Truncation Error

The above section introduces how to derive the propagation errors. But for the primary inputs, the errors come from the truncation. As we discussed in section 3, if we use finite bits to represent the input data, the resolution will be 2^e, and the absolute truncation error compared with the real value will be in $[0, 2^e)$. Assume this error is uniformed distributed. Then the standard deviation of error after truncation of $input_i$ will be:

$${\sigma_i}^2 = \frac{1}{12}(2^e - 0)^2 = \frac{1}{12} 2^{2e}$$

Besides, if the intermediate results contains the right shifts, as shown in Eq.3.3, it will also contains the truncation error.

1. Addition

$$\sigma_{trunc}{}^2 = \frac{1}{12} 2^{2(e_{out}-e_a)} (e_{out} > e_a) \tag{3.6}$$

The logical term equals to 1 if it is true, otherwise 0.

2. Multiplication

$$\sigma_{trunc}{}^2 = \frac{1}{12} 2^{2(e_{out}-e_a-e_b)} (e_{out} > e_a + e_b) \tag{3.7}$$

3. Division

$$\sigma_{trunc}{}^2 = \frac{1}{12} 2^{2(e_{out}-e_a+e_b)} (e_{out} > e_a - e_b) \tag{3.8}$$

Therefore, the overall errors of the intermediate results equal to the propagation error plus the truncation error:

$$\sigma_{out}{}^2 = \sigma_{prop}{}^2 + \sigma_{trunc}{}^2$$

This is the non-linear function of the variable e.

We define the overall error function as the summation of all the $\sigma_{out_i}{}^2$ of the final output nodes in DFG:

$$Error = \sum_i \sigma_{out_i}{}^2, \quad i \in output\ nodes \tag{3.9}$$

3.5 Constraints and Optimization

In this section, we first illustrate how to formulate the energy consumption function. To reduce the redundant solution searching space, we heuristically add constraints and apply some numerical tricks. We summarize the overall algorithm of our proposed method and discuss how to solve it through public solver.

3.5.1 Energy Consumption

The energy consumption Er can be defined as:

$$Er = \begin{cases} E_+ * BW + C_+ \\ E_* * BW^2 + C_* \\ E_/ * BW^2 + C_/ \\ E_{sft} * e_{diff} + C_{sft} \end{cases}$$

where BW is the bit-width for each operation. $E_+, E_*, E_/$ are the corresponding energy consumption factors, which are the constant values determined by the hardware. The Er of input nodes and constant nodes are '0', as no computation is involved. Er contains the dynamic part and static part. For each operation, the dynamic part is related to the number of bits participate in. It has the linear relationship with the number of bits in addition and shift, and quadratic in multiplication and division. The static part C refers to the static energy consumption of each operation. Besides, if our approach is implemented in programming level, where the energy consumption not only comes from the arithmetic operations but

also the costs of fetching and executing the instructions. We can simply merge these costs into the static part. The static parts will be the constants that only determined by the hardware and architecture. These constant terms do not affect the optimization process of BW when minimizing Er. Therefore, we can omit them for convenience, and only focus on the computational energy cost.

We can also omit the energy cost of shift operation based on the following two reasons: 1) Compared with the other arithmetic operations, the energy cost of shift is much smaller. Besides, if our approach is directly implemented in circuit level, the shift can be replaced by routing without any energy overhead. 2) More importantly, minimizing the truncation error based on the constraints introduced in subsection 3.5.2 will intrinsically and implicitly minimize the shift operation. Therefore, we no longer need to consider it. Based on our experimental results, with or without adding the energy cost of shift yield the same results. So Er can be re-written as:

$$Er = \begin{cases} E_{+} * BW \\ E_{*} * BW^2 \\ E_{/} * BW^2 \end{cases} \tag{3.10}$$

BW should equal to $max(w_a, w_b)$. $max()$ function usually cannot be directly solved by the public optimizer. So we convert it to the linear constraints:

$$BW \geq w_a$$

$$BW \geq w_b$$

Since the energy function will be minimized, this linear constraint is congruent to $max(w_a, w_b)$ in the optimized solution. The overall energy consumption $Energy$ of running the data-flow graph is the summation of Er of all the nodes, and it is the quadratic function with the variables of bit-width w.

3.5.2 Constraints

Up to now, we have built the energy consumption function $Energy$ and the error function $Error$. The next step is to balance the tradeoff between these two functions. Both of these two functions are non-linear, to eliminate the redundant search space, we add constraints for every node based on its operation type.

1. Input nodes and Constant nodes. Input nodes and Constant nodes do not contain any computations, so they will not be counted in the energy function. The only constraint is the Eq.3.2. Its right side will be a constant number if the data range is fixed. Therefore, w can be represented by e.

$$w = \lceil log_2(max(|\underline{x}|, |\overline{x}|)\rceil + 1 - e$$

2. Addition nodes. Due to the natural characteristics of addition, the scaling factor of two operands must be the same. Then the output's scaling factor will be $e_{out} = e_a \pm s$, '+' means out' right shifts s bits, and '$-$' means left shifts. In addition, we can simply let $s = 0$ with the consideration of the following two scenarios: 1) out' right shifts s bits. In this case, we can actually right shift a' and b' s bits first without lose any precision, and still maintains $e_{out} = e_a = e_b$. Besides

the shorter w brings lower energy consumption. 2) out' left shifts s bits. In this case, the out' will be padded with '0' at the end. These redundant '0's increase the bit-width of the adder but contribute nothing to the precision. So the overall constraints of addition are:

$$w_{out} = \left\lceil log_2(max(\left|\underline{out}\right|, \left|\overline{out}\right|)\right\rceil + 1 - e_{out}$$

$$e_{out} = e_a = e_b$$

$$BW_{out} \geq w_a; \quad BW_{out} \geq w_b$$

Besides, if $e_{out} = e_a = e_b$, the truncation error in Eq.3.6 becomes 0.

3. Multiplication nodes. For multiplication nodes, the only different part compared with addition is the constraint of the scaling factor. The w_{out} of addition node will not increase due to the nature of addition. But the bit-width after multiplication will increase, and if we still keep the scenario 1), the bit-width will explosively increase with the data flow graph going deep. Therefore, we allow the right shifts as the relaxation. The overall constraints of the multiplication are:

$$w_{out} = \left\lceil log_2(max(\left|\underline{out}\right|, \left|\overline{out}\right|)\right\rceil + 1 - e_{out}$$

$$e_{out} \geq e_a + e_b$$

$$BW_{out} \geq w_a; \quad BW_{out} \geq w_b$$

If we allow right shifts, the truncation error in Eq.3.7 may not vanish. Moreover, the condition term cannot be handled in most of the public solver since it

is discontinuous and non-differentiable. We adopt the tanh function to mimic this logic function:

$$f(x) = \frac{1 - e^{cx}}{1 + e^{cx}}$$

As we only allow right shift, $x = e_{out} - e_a - e_b$ will fall on the positive side. By increasing c, $f(x)$ can be approximately treated as a boolean function. The function curves is shown in Fig.3.2.

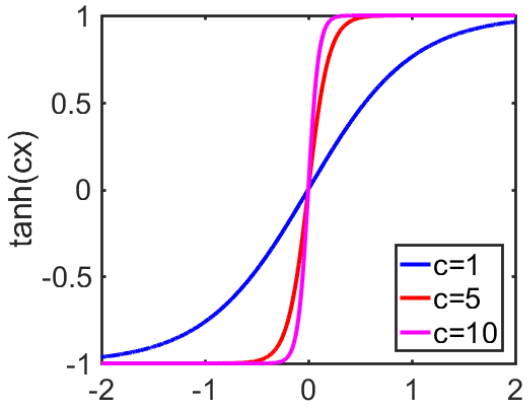

Figure 3.2: Tanh function mimics the boolean logic function

4. Division nodes. In contrast to multiplication, the output's bit-width is smaller than its parents. We add left shifts as the relaxation to prevent the bit-width vanishing. So the constraints are:

$$w_{out} = \left\lceil log_2(max(\left|\underline{out}\right|, \left|\overline{out}\right|)) \right\rceil + 1 - e_{out}$$

$$e_{out} \leq e_a - e_b$$

$$BW_{out} \geq w_a; \quad BW_{out} \geq w_b$$

Since we only allows left shifts in division, the truncation error in Eq.3.8 will be 0.

3.5.3 Functions and Constraints Generation Algorithm

The overall algorithm of combining above sections to generate the energy and error function and constraints is shown below:

Algorithm 1: Bit-width optimization algorithm

Input : A data flow graph G;
A set of input range R
Output: A function of energy cost $Energy(BW_i)$;
A function of estimated error $Error(e_i)$;
A set of constraints $Constraint(BW_i, e_i)$

```
1 Constraint(BW_i, e_i) = ∅
2 Energy(BW_i) = 0
3 topoOrder = TopologySort(G)
4 for each node in topoOrder do
5     compute data range
6     compute σ^2
7     Energy(BW_i) += ER_i
8     add constraint to Constraint(BW_i, e_i)
9 compute Error(e_i) using Eq.3.9
```

At the beginning, we need to sort the nodes in topological order. This is because the pseudocode inside the for loop has data dependency. The range, error, and energy can only be computed after its parents have been computed. The complexity of the algorithm is $O(N+V)$, where N is the number of nodes and V is the number of edges. As we claimed in the introduction, our approach uses data range as prior knowledge and optimize F statically in the design phase, the timing overhead can be negligible.

For the data range analysis, we adopt the minimize function in SciPy.optimize python library. Each intermediate node is expressed by a function of input nodes,

then we use the input node ranges as the variable bounds to minimize/maximize this expression.

3.5.4 Optimization Strategy

From the above algorithm, we obtain the error function, energy function, and constraints. Since our goal is to minimize the energy consumption with guaranteed accuracy. So the straightforward way of the optimization process, as well as adopted by [41] will be:

$$\begin{aligned} &minimize && Energy(BW_i) \\ &subject\ to && Error(e_i) \leq er \\ &and && Constraints(BW_i, e_i) \end{aligned}$$

er is the user-defined maximum error the system can tolerate.

However, this optimization approach usually brings the over-design issue. Because $Error(e_i)$ is an estimated function, and most likely over-estimated. The actual error could be much smaller than er. Hence, limiting the error bound could severely squeeze the optimization space. On the other hand, $Energy(BW_i)$ is a restrict function without over-estimation. The bound for energy will be more easy to reach. Besides, setting the limitation for energy consumption is more intuitive in

the hardware and system design. Hence, we change our optimization strategy to:

$$\begin{aligned} minimize \quad & Error(e_i) \\ subject\ to \quad & Energy(BW_i) \leq engy \\ and \quad & Constraints(BW_i, e_i) \end{aligned} \tag{3.11}$$

engy is the user-defined maximum energy the system can consume. Similar to the data range analysis, we use the SciPy.optimize.minimize function with 'SLSQP' kernel.

3.6 Experiment Results

In this section, we first measure the energy consumption factors for the basic arithmetic operations. To emphasize the advantages in energy saving, we apply our approach to 9 benchmarks and compared with the traditional fixed point design in terms of energy consumption and error.

3.6.1 Energy Consumption for Arithmetics

We implement the adder, multiplier, divider with different bit-width in Verilog and synthesis them using Cadence RTL Compiler with FreePDK 45nm library. Then we compute the averaged energy consumption for a single bit as mentioned in Eq.3.10.

The factors in Table.3.1 do not mean the energy consumption for each oper-

Table 3.1: Energy consumption factors for basic arithmetics

Operations	E_+	E_*	$E_/$
Energy Consumption (pJ)	0.09	0.042	0.088

ation (Addition is by no means more expensive than multiplication). Instead, the overall consumption is related to the bit-width BW shown in Eq.3.10, which is the quadratic for multiplication/division and linear for addition. In the table, E_* is smaller than E_+, this is because the multiplication is usually optimized in parallel. Division cannot be parallelized, so it has very similar energy consumption to addition.

3.6.2 Energy Consumption vs. Accuracy

We run our approach on 9 benchmarks. *poly*, *rgb*, *bsplines*, *DCT* and their corresponding input data ranges are from [41]. *sine*, *sqroot*, *predatorprey*, *turbine*1 and their input data ranges are from [49]. We name the example used in [50] as *vectorDot*, since it can be treated as the vector dot product, which is widely used in machine learning and deep learning field. [41] has one more benchmark named *MatrixMult*, but the data ranges of all the nodes are the same. As we mentioned before, the optimization space comes from the disparity of the data ranges, so our approach gives the same solution with the traditional approach. Therefore we do not list it in the table. To emphasize the advantages of our approach, we use the traditional fixed point design with $FB = 10$ for all nodes as the baseline. We randomly generate 100k inputs within the data range of each benchmark, and use

mean squared error (MSE) and mean absolute percentage error (MAPE) metrics to evaluate the accuracy. In the Table.3.2, we let the constraint of energy consumption in Eq.3.11 to be the same with that of the traditional fixed point design with $FB = 10$. Then we compare their performance in accuracy. The column DFP denotes our proposed dynamic fixed point based approach. Clearly, we can see that given the same amount of energy, our approach provides much smaller errors in both metrics.

Table 3.2: Accuracy comparison under same energy consumption

	MSE		MAPE	
Benchmarks	FB=10	DFP	FB=10	DFP
vectorDot	2.59e-5	3.31e-8	6.25e-4	2.73e-5
poly	6.09e-7	1.27e-7	2.16e-3	9.04e-4
rgb	3.08e-7	7.52e-8	1.68e-2	8.61e-3
bsplines	3.74e-7	3.57e-7	4.21e-2	3.23e-2
DCT	4.52e-7	1.48e-7	1.56e-2	8.04e-3
sine	2.62e-6	4.29e-8	2.45e-3	5.01e-4
sqroot	3.00e-7	3.61e-8	3.62e-4	1.28e-4
predatorprey	1.33e-6	4.39e-8	8.42e-3	1.39e-3
*turbine*1	3.19e-5	1.73e-5	7.12e-4	6.47e-4

In the Table.3.3, we make the error of our approach to be similar to the traditional design. As we mentioned in section 3.5.4, controlling the error to be a specific value would be very difficult, so we cannot make the error of two designs to be exactly the same. We will consider the errors at the same level as long as their difference is within 1x of minor one. Then on average, we can save 20.4% energy. If we ignore *bsplines* and *turbine*1, we can save up to 26.3% energy. For *bsplines*

and *turbine*1, even though we cannot obtain energy savings, we can still provide the better designs with higher accuracy.

Table 3.3: Energy savings under similar errors

Benchmarks	MSE		MAPE		Energy Saving
	FB=10	DFP	FB=10	DFP	
vectorDot	2.59e-5	5.23e-5	6.25e-4	6.46e-4	39.72%
poly	6.09e-7	8.42e-7	2.16e-3	2.44e-3	19.28%
rgb	3.08e-7	2.95e-7	1.68e-2	1.51e-2	18.69%
bsplines	3.74e-7	3.57e-7	4.21e-2	3.23e-2	0%
DCT	4.52e-7	3.66e-7	1.56e-2	1.48e-2	12.58%
sine	2.62e-6	2.38e-6	2.45e-3	3.54e-3	34.06%
sqroot	3.00e-7	2.34e-7	3.62e-4	3.21e-4	17.93%
predatorprey	1.33e-6	6.16e-7	8.42e-3	4.99e-3	41.53%
*turbine*1	3.19e-5	1.73e-5	7.12e-4	6.47e-4	0%

3.6.3 Accuracy vs. Fraction bits

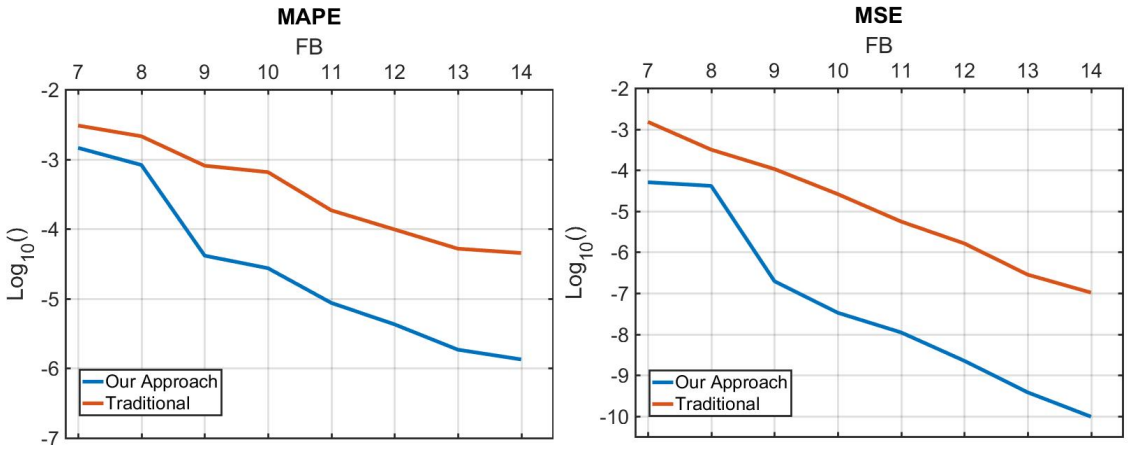

Figure 3.3: Error drops with bit-width increases

In the subsection, we use *vectorDot* benchmark as the example since it now frequently used in machine learning and deep learning applications. We use the

energy consumption of the traditional fixed point design when $FB \in [7, 14]$ as the constraints of Eq.3.11 and minimize the error. We scale the error by log_{10} in the y-axis. Fig.3.3 shows that with the fraction bit-width increase, our approach always yields lower error compared with the traditional design.

3.7 Future Work

There are a few aspects we can expand to make our work more efficient and powerful.

1. Speedup the optimization process

Our method allows each operation has its own scaling factor, therefore, if the graph contains very large number of nodes, the optimization solver could be very slow or yield a less optimal solution.

We propose a compromised solution. In some data flow graphs, lots of the branches appear in the same shape and are independent to each other. Therefore, we can allow these branches have the same combination of scaling factors and only consider one branch in order to reduce the total number of nodes. For example, the element-wised multiplications in matrix multiplication will not affect each other. Since each result will be accumulated later and based on the constraints of addition node in section 3.5.2, they should share the same scaling factor to avoid massive shifts. So the computations in the fully connected neural network can be decomposed and simplified as a small graph that has two multiplications and one addition in scalar level as shown in Fig.3.4.

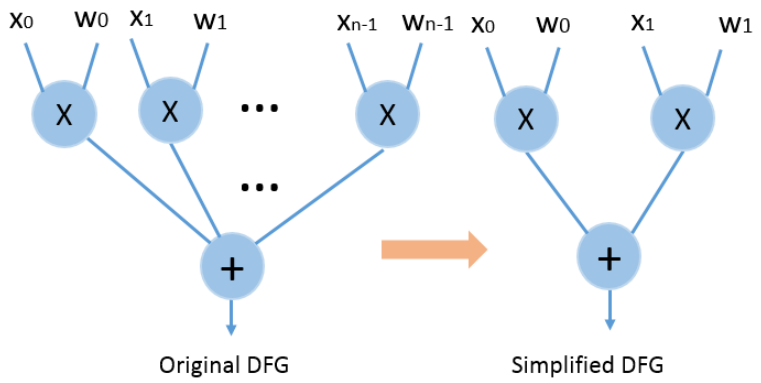

Figure 3.4: Simplify vector multiplication DFG for optimization speedup

2. Combine the optimization with training process

In our current approach, once the data range and the topology of the data flow graph are fixed, the solution of bit-width and scaling factor allocation will be fixed. However, for most of the classification algorithms, the computational error is not strictly related to the classification error. Therefore, we can utilize the training steps to tune the bit-width and scaling factor for every node.

3. Develop a software tools

We can collect the above features and package them as the software tools. The input should contain:

1) a piece software program;

2) the input data range;

3) the expected maximum energy consumption.

The tool will read the software program and decompose it into data flow graph, then use our approach and the input data range to compute the scaling factor and bit-width for each node. Then re-synthesized back to a software program.

Chapter 4: Data Flow Graph Approximation

In this chapter, we proposed a novel runtime estimation technique and three effective data flow graph truncation algorithms. The algorithms can dynamically truncate the non-critical parts of given data flow graph based on the estimated value with the concern of every input. These approximations effectively leverage considerable energy savings with acceptable computation quality.

4.1 Design Motivation

We utilize the error resilient characters of the operations (e.g. add, min/max) in the data flow graph to classify the non-critical nodes. For example, $S = add(A, B)$ where both A and B are intermediate results coming from previous computations, if the absolute value of B is much smaller than A's, the computational error of S will be less related, in other words, less sensitive, to B. Therefore, instead of computing accurately, we can replace B with an estimated value. Similarly, for min/max operation, we can completely neglect the computation of one operand if it is much smaller/bigger than the other one.

The existing approaches [21, 22, 25] share very similar design philosophy with ours. The energy savings can be achieved by approximating the non-critical parts

of the program. These non-critical parts are targeted by using training data or by data range tuning and interval arithmetic. However, all these methods are working in offline regardless every input, as we called "static". To guarantee the accuracy in worst cases, this static feature brings the *false-negative* issue when designers are trying to guarantee the worst cases. For example, a part of the program is sensitive to the final output in general and cannot be approximated by static approaches, but for some given inputs, it becomes insensitive and negligible. Since in [21,22,25], the way of approximation is pre-decided, they cannot foresee these special cases.

Assume $a \in (0,1), b \in (0,1)$ and $c \in (100,101)$. Our goal is to compute $s = a \times b + c$. In the existing DFG approximation techniques, by observing that $a \times b \in (0,1)$, s will be approximated to $0.25 + c$, since $a \times b$ less critical than c. Replacing $a \times b$ does not lose too much accuracy, and more importantly saves a multiplication. However, this approach has two potential weakness:

1) Over estimation issue: If a and b are not strictly in the assumed ranges, permanently remove them may bring huge errors.

2) Under estimation issue: If $b \in (0,1) \cup (100,200)$, then $a * b$ might not be non-critical for c when $b \in (100,200)$. The existing approach will not approximate $a * b$ in this cases. Therefore, we lose change of saving energy when $b \in (0,1)$.

Moreover, in some applications, the existing approaches also fail when the data ranges are not available or not tightly bounded.

Our proposed approach will dynamically consider each input data during execution, and runtimely decide whether a node is critical or not for that input. The major challenge of this "dynamic" fashion is how to estimate the intermediate val-

ues energy efficiently. To address this challenge, we propose a novel estimation approach and corresponding highly hardware-efficient computing mechanism. Before the estimation process, input data is converted into fixed point base-2 logarithmic representation. During the estimation, the operations in the data flow graph are transformed into the computations in logarithmic domain. We use the estimated result to decide the non-critical parts of the graph. Then we approximate the DFG by runtimely replacing these non-critical parts with the estimated values. Our experiments prove that the logarithmic estimation process will be much more energy efficient compared with accurate computing in linear domain, leveraging the overall energy efficiency enhancement of the system.

The major work in this chapter has been published in 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD) [51].

4.2 Related Work

[21] proposed a novel automated approximation technique for the logic synthesis. The authors first transform the exact RTL or behavioral HDL design into an abstract syntax tree (AST). Then they applied training data to locate the approximate parts in the AST and replace them by their approximate variants. Finally, they wrote back the modified AST to RTL or behavioral HDL. In [22], a novel sensitivity computation approach named Automatic Sensitivity Analysis (ASAC) is proposed to extract the sensitivity of the output with respect to the given program data by tuning the data range of variables. Then the non-critical data can be iso-

lated based on the achieved sensitivity information. The energy saving is leveraged by computing and storing these non-critical data approximately. Similarly, interval arithmetic (IA) is a numerical method to compute the bounds of given variables efficiently. But it may suffer from the wrapping effect which outputs large overestimated bounds. [25] proposed an algorithmic differentiation based interval computing approach that can effectively reduce the wrapping effect in many cases. The authors adopt this IA variant to analyze the significance of variables in the program with respect to the final output. The insignificant part of the program can be either executed in low precision or unreliable hardware or simply replaced by constants.

Besides the *false-negative* issue mentioned in the introduction section. The above three approaches can also cause a *false-positive* problem. [21] uses training data to determine where and how to approximate. Whether the training data is sufficient will directly affect the quality of approximation. Insufficient training data may misclassify an important part to be non-critical, leading to over-approximation issue. [22, 25] use input ranges to tune the approximation. If the input ranges are not sufficient, some uncovered inputs may cause significant errors.

To fix these false-positive issues, Khudia et al. in [26] propose a computationally inexpensive framework named Rumba, for online detection and large approximation error correction. Rumba offers the continuous lightweight checks and fixes these errors by recomputing exactly.

Some other work has been reported targeting the approximate computing on DFG. For example, [24] well addressed the approximate resource allocation and binding problem. They proposed a novel error propagation model and scheduling

algorithms to maximize the energy saving within the given accuracy constraints.

4.3 Arithmetic Estimation in Logarithmic Domain

In this section, we demonstrate how to estimate the intermediate results in the data flow graph. We first introduce a novel conversion of the input data to the fixed point expression in logarithmic domain. Then we illustrate how to approximate the linear domain arithmetic operation using the data in logarithmic expression. We target on the floating point operations because 1) floating point cannot be replaced by the fixed point due to its very large data range in lots of applications; 2) floating point format can be easily encoded into logarithmic domain.

4.3.1 Conversion from Floating Point to Logarithmic Representation

There are two widely used types of floating point format: single precision (defined as float in C) and double precision (defined as double). Double precision is very expensive and seldom used in for energy efficiency concern. In our work, we focus on the single precision expression. There are 32 bits in IEEE754 single precision standard consisting of 1 sign bit, 8 exponent bits, and 23 mantissa bits, as shown in the Fig.4.1(a) The value of an single precision floating point number can be computed by

$$sign \times (1 + mantissa) \times 2^{exponent-127} \tag{4.1}$$

By ignoring the sign bit, we can rewrite Eq.4.1 for a given x as $x = 1.m \times 2^e$, where the dot is the radix point. Let x_l to be the logarithmic value of x. The subscript

l denotes the logarithmic representation, and we use this notation for both data and transformed arithmetic operations in the rest of the paper. Then x_l can be approximated as

$$\begin{aligned} x_l &= \log_2(x) = \log_2(1.m \times 2^e) \\ &= \log_2(1.m) + e \qquad (4.2) \\ &\approx m + e = e.m \qquad (4.3) \end{aligned}$$

From Eq.4.2 to 4.3 we use the approximation equation:

$$\log_2(1+x) \approx x \quad \text{for } x \in [0,1] \qquad (4.4)$$

Based on the derived result in Eq.4.3, we can get x_l by directly truncating the floating format without any computation (shown in Fig.4.1(b)).

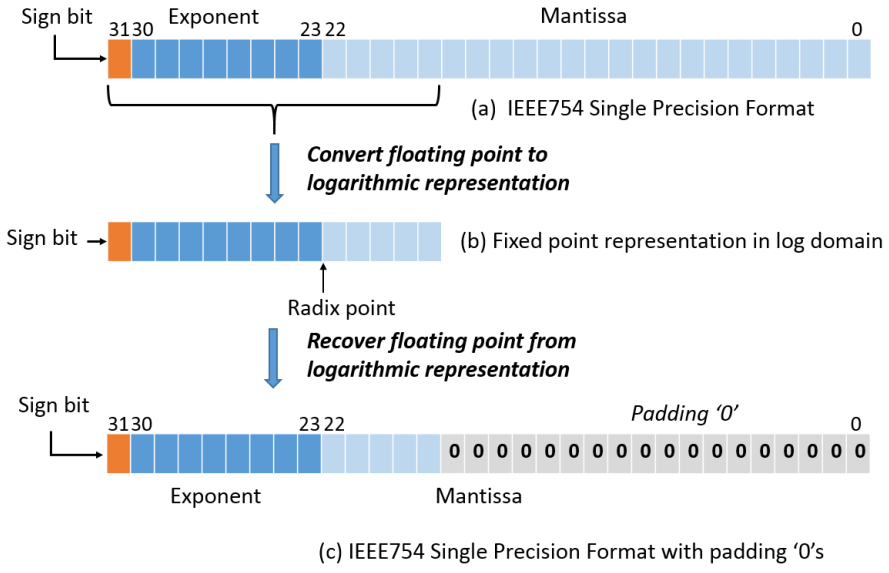

Figure 4.1: Conversion of floating point formation between linear domain and log domain

In Eq.4.4, for $x \in (0, 1)$, $\log_2(1+x)$ is always slightly bigger than x, which can be seen in Fig.4.2. To minimize this error, we empirically truncate mantissa at the 5^{th} bits without rounding the 6^{th}. x_l is represented as fixed point with a sign bit in front. The radix point is at the same position with the floating point format.

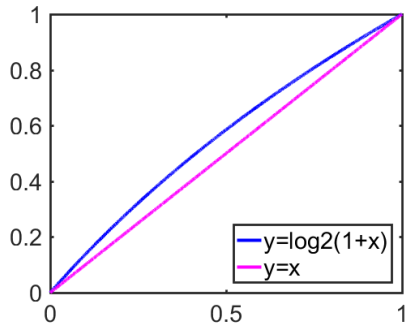

Figure 4.2: The log_2 approximation

When recovering the data from log domain to linear, we can simply pad "0"s to the end and change the data type from fixed point to floating point. The conversion and recovery process is shown in Fig.4.1(c).

4.3.2 Arithmetic Operations in Logarithmic

The above subsection illustrated how to convert the input data into log domain. Next, we discuss how to use this converted data in the basic arithmetics. Let A and B are two input operands in linear domain, and S is the arithmetic output. The converted operands are A_l and B_l, and the output in log domain is S_l. For simplicity, we assume A and B are positive.

4.3.2.1 Accurate Conversion

The following operation can be directly processed without adding error compensation.

1. Multiplication: $S = A \times B$

   $S_l = \log_2 S = \log_2(A \times B) = \log_2 A + \log_2 B = A_l + B_l$

   Multiplication in log domain, marked as mul_l can be done by simply adding the logarithmic operands.

2. Divison: $S = A/B$

   Similar with multiplication, $S_l = A_l - B_l$

3. Square root: $S = \sqrt{A}$

   $S_l = \log_2 S = \log_2 \sqrt{A} = \frac{1}{2}\log_2 A = \frac{1}{2} A_l = A_l \gg 1$

4. Power n: $S = A^n$

   $S_l = \log_2 A^n = n\log_2 A = nA_l$

5. Min/Max or comparison: $S = max(A, B)$

   $S_l = \log_2 S = \log_2 max(A, B) = max(\log_2 A, \log_2 B) = max(A_l, B_l)$

   Comparative operations stay the same with linear domain.

The key character in above operations is that: the computation processes are much cheaper in log domain than those in linear domain. For example, the multiplication and division of floating point unit (FPU) is very energy expensive and time consuming. But in log domain, we can simply use a fixed point adder with 13 bit width. Compared with single precision floating point multiplier, this 13-bit fixed adder can

achieve more than 100x energy saving. Similarly, computing square root is intricate for computer, but in log domain we only need a shift register.

4.3.2.2 Approximate Conversion

Addition and subtraction are much more complicated compared with above operations in section 3.2.1 if we want to compute exactly. We first need to the recovery step in 3.1 to convert the data back to linear domain. Then apply FPU for addition and subtraction. After this, redo the conversion step to get the logarithmic representation of the result. The mathematical step is shown below:

$$\begin{aligned} S_l &= \log_2 S = \log_2(A+B) \\ &= \log_2(recover(A_l) + recover(B_l)) \\ &= \log_2(1.0 \times 2^{A_l} + 1.0 \times 2^{B_l}) \end{aligned} \tag{4.5}$$

However, addition and subtraction are most commonly used in the data flow graph. By following the Eq.4.5, we may get a relatively accurate result but will not obtain energy saving, which is contradicted to our primary desire. Besides, in the data conversion step in 3.1, we already introduced some errors for every primary input data, spending significant energy to do the accurate computing using FPU is much less cost-effective. Since our goal in log domain is to classify the non-critical sub-graph through estimation, we do not need the result to be very accurate as long as we can tell which operations are negligible. To these points, we introduce a novel addition/subtraction estimation with error compensation technique. Without loss

of generality, we assume A and B are positive and $A > B$. Let's reconsider Eq.4.5:

$$\begin{aligned} S &= A + B = 2^{A_l} + 2^{B_l} \\ &= 2^{A_l}(1 + 2^{B_l - A_l}) \\ &\approx 2^{A_l}(1 + 2^{round(B_l - A_l)}) \\ &= 2^{A_l}(1 + 2^{-ed}) \end{aligned} \tag{4.6}$$

Where the exponential difference $ed = round(A_l - B_l)$. Since $A_l > B_l$, ed will be a positive integer, and $2^{-ed} \in (0, 1)$. Therefore:

$$\begin{aligned} S_l &= \log_2 S = \log_2 2^{A_l}(1 + 2^{-ed}) \\ &= \log_2 2^{A_l} + \log_2(1 + 2^{-ed}) \approx A_l + 2^{-ed} \\ &= A_l + 1 \gg ed \\ &= \begin{cases} A_l + 1 \gg ed & \text{if } ed \leq 5 \\ A_l & \text{if } ed \geq 5 \end{cases} \end{aligned}$$

Here we reuse the Eq.4.4. Notice that A_l has only 5 bits in fraction part. So we will omit the compensation if $ed \geq 6$.

We draw a motivational example in Fig.4.3 to illustrate the above derivation. Clearly, we can see that our new proposed logarithmic addition can be estimated by only using two fixed point additions and one shifting. After we convert the data to logarithmic domain using Fig.4.1, the data will be in the fixed point format with the

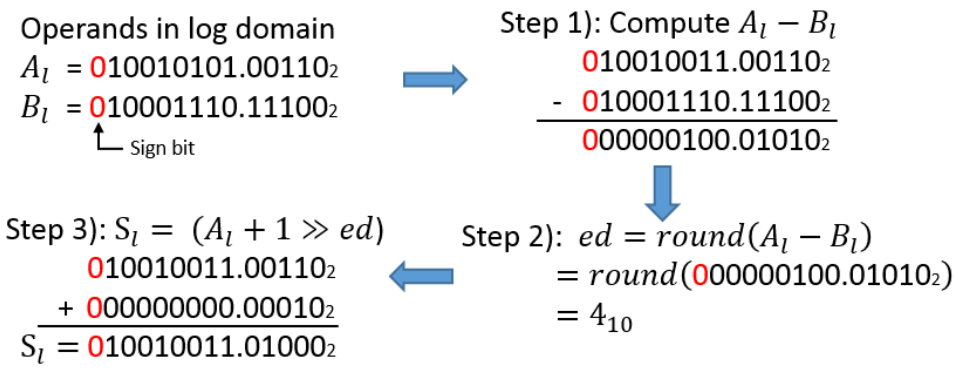

Figure 4.3: Example of addition in log domain

heading bit as the sign bit. In the first step, we compute the difference between these two numbers. Then in the second step, ed is generated by rounding the difference to integer. In the end, we left shift '1' with ed bits and add it as the compensation to the larger number in the converted data.

The estimation process of subtraction is similar with addition at the beginning but has different compensation.

$$\begin{aligned} S &= 2^{A_l} - 2^{B_l} \approx 2^{A_l}(1 - 2^{-ed}) \\ S_l &= \log_2 S = A_l + \log_2(1 - 2^{-ed}) \end{aligned} \tag{4.7}$$

In here, we cannot apply Eq.4.4 for approximation. Instead, we use Look-up Table to a store the approximate value of $\log_2(1-2^{-ed})$. The values are shown in Table 4.1. Similar with addition, we will discard the error compensation if $ed \geq 6$. A special case is when $ed = 0$. It means B_l is just slightly smaller than A_l. But in linear domain, the possible value of $A - B$ could vary within $(0, 0.293A)$. Eq.4.7 cannot handle this special case because $\log_2(0)$ is $-\infty$, which is infeasible in practice. In

the table, we empirically set the compensation value to be -1 for easy computation. Based on the experiments we observe that this value has very minor effect on the overall estimation quality. We can also construct the Look-up table for $\log_2(1+2^{-ed})$

Table 4.1: Look Up Table for subtraction estimation in log domain

ed	$\log_2(1-2^{-ed})$	**binarized approximation**	**binarized value**
0	~	-1.00000_2	-1
1	-1	-1.00000_2	-1
2	-0.415	-0.01110_2	-0.4375
3	-0.1926	-0.00110_2	-0.1875
4	-0.0931	-0.00011_2	-0.09375
5	-0.0458	-0.00001_2	-0.03125

in addition estimation. But in general, shifting operation with only a few steps is cheaper than checking the Look-up Table. Besides, the difference between these two ways of error compensations is negligible based on our experiment.

[29] uses base-2 logarithmic representation to encode the weights in convolutional neural network. It looks like that their approach is very similar to our logarithmic estimation approach. But in fact they are fundamentally different. [29] quantizes $\log_2(x)$ to integer and apply shifting operations during accumulating. In our approach, we use fixed point number to represent $\log_2(x)$, and we only need adder and comparator during the whole estimation process. Apparently our approach provides much higher accuracy and [29] fails to guarantee the computational quality of traditional applications such as DSP.

4.4 Runtime DFG Approximation Algorithm

Section 3.2 introduces how to estimate the intermediate value by transforming the data flow graph into logarithmic domain. In this section, we explain how to use these estimated data to replace the non-critical parts of the DFG.

4.4.1 Non-criticality Truncation

4.4.1.1 Error Resilient and Sensitive Operations

The operations inside a DFG with more than two operands can be decomposed into multiple operations that have only one or two inputs. Given a decomposed DFG, we classify these operations into two types, the error sensitive operations and error resilient operations. Each node in DFG denotes an operation, and each edge can be considered as an intermediate result (except the input and output edges).

Definition 1.1: The node in a DFG with error sensitive operation is defined as the error sensitive node, denoted as n^s.

The nodes with operations like multiplication, division or power are considered as error sensitive nodes since any tiny variation on their inputs could bring significant difference on the output.

Definition 1.2: The node in a DFG with error resilient operation is defined as the error resilient node, denoted as n^r. *An* n^r *is dominated by one input, named as* I_d, *and is less sensitive to the other minor input marked as* I_m.

For $S = add(A, B)$ in linear domain, the dominant input I_d is the one with

larger absolute value of A and B, and I_m is the rest one. For $S_l = add_l(A_l, B_l)$, the dominant input I_d equals to $max(A_l, B_l)$. In log domain, we no longer need to take the absolute value of each inputs. Because the signs of A_l,and B_l also reflect magnitude of their original values in linear domain. Similarly, I_d will be the one with smaller input for $S = min(A, B)$ and $S_l = min_l(A_l, B_l)$ since the output will completely not be affected by the larger one.

We explore this error resilient feature to identify the non-critical inputs and their corresponding branches.

4.4.1.2 Non-criticality Definition and Classification

The non-critical input only occurs when the node is error resilient. All the inputs of error sensitive nodes will be treated as critical inputs. In this chapter, we consider the error resilient node with the addition/subtraction and comparative (such as min/max) operations.

Definition 2: An input I_m is a non-critical input in log domain if and only if $f(I_d - I_m) \geq \delta$, where

$$f(x) = \begin{cases} x & \text{for } add_l/sub_l \\ abs(x) & \text{for comparative operations} \end{cases}.$$

δ in above definition is the threshold we use to control the overall computa-

tional quality. $I_d - I_m = 1$ means that the dominant input is twice larger than the minor input in the linear domain, regardless their signs. The larger δ is, the less critical I_m will be. For the comparative operations, we compute the absolute difference between two inputs. This is because I_d could be either the bigger one or the smaller one depending on the comparison type. The criticality is determined after the estimation process.

4.4.1.3 Truncation and Recomputation

The following steps list how to produce the approximate result for energy efficiency after targeting the non-critical input of given error resilient node n_i^r.

1. Cut off the n_i^r's parent branch that computing the non-critical input;
2. Replace this non-critical input with the estimated value: I_m;
3. Convert the estimated value I_m in log domain to the value in linear domain;
4. Recompute the n_i^r's critical parent branch in linear domain accurately;
5. Compute node n_i^r accurately.

In step 1), all the nodes in the non-critical parent branch is removable no matter it is error resilient or not. Fig.4.4 presents a motivation example of this truncation and recomputation step.

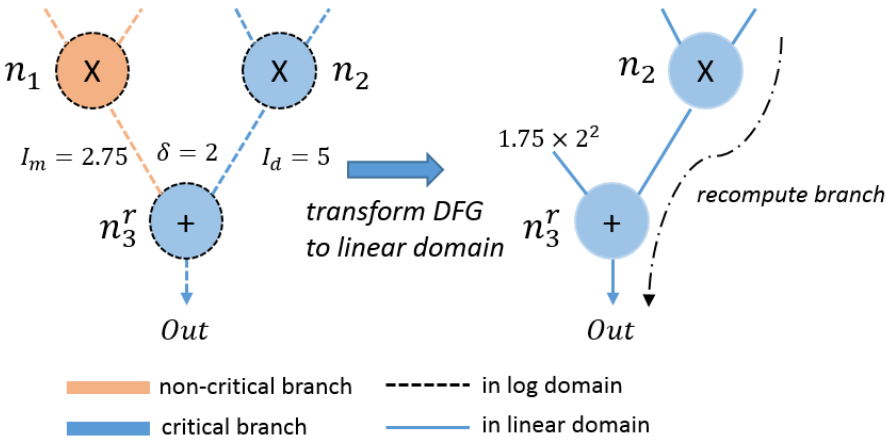

Figure 4.4: Example of truncation and recomputation

4.4.1.4 Error Analysis

In this section, we deduce a theoretical error analysis of the truncation and recomputation steps. We consider addition as an example. Let v_m and v_d are the two accurate input values of n_i^r with I_m and I_d as their logarithmic values. Therefore $v_m = 2^{I_m}$ and $v_d = 2^{I_d}$. Assume the estimation error of I_m and I_d are ε_m and ε_d. So the recovered value of v_m is $2^{I_m+\varepsilon_m}$. The reason we use absolute error ε_m instead of an error rate is that the experimental result shows that the value of ε_m and I_m are highly independent. For simplicity, we omit the conversion and recovery error between the log and linear domain. So the error rate of node n_i^r is:

$$\begin{aligned} er &= \frac{O_{apx} - O_{acc}}{O_{acc}} = \frac{(2^{I_m+\varepsilon_m} + v_d) - (v_m + v_d)}{v_m + v_d} \\ &= \frac{2^{I_m+\varepsilon_m} - v_m}{v_m + v_d} = \frac{2^{I_m+\varepsilon_m} - 2^{I_m}}{2^{I_m} + 2^{I_d}} \\ &\leq \frac{2^{\varepsilon_m} - 1}{2^{\delta-\varepsilon_d+\varepsilon_m} + 1} \end{aligned} \tag{4.8}$$

In Eq.4.8, we use the relation

$$(I_d + \varepsilon_d) - (I_m + \varepsilon_m) \geq \delta$$

O_{acc} and O_{apx} are the accurate and approximate output respectively. Assume $\varepsilon_m = \varepsilon_d = \varepsilon$, then

$$er \leq \frac{2^{\varepsilon} - 1}{2^{\delta} + 1} \tag{4.9}$$

If we let $\varepsilon = 0.5$ and $\delta = 3$ and bring them into Eq.4.9:

$$er \leq \frac{2^{0.5} - 1}{2^3 + 1} \approx 4.6\%$$

With proper selected threshold, we can balance the tradeoff between the computational quality and energy saving.

4.4.2 Runtime Approximation Algorithms

We are now able to determine whether the parent branches of a given node n_i^r is non-critical or not, but we cannot directly truncate all the non-critical nodes we found in the whole data flow graph. Consider the following scenario:

Problem 1: A parent node n_k is in the non-critical branch of n_i^r, but it is also in the critical branch of n_j^r. Simply removing n_k may not cause significant error of n_i^r, but it will have badly impact on n_j^r.

To solve this problem, we propose two efficient algorithms to truncate the non-critical nodes. The algorithms can be integrated with runtime scheduler of data flow graph.

4.4.2.1 GlobalCut Algorithm

The *GlobalCut* algorithm provides an overall consideration of the data flow graph. Instead of removing the non-critical nodes, we translate the problem to: reserve the critical nodes on the path from primary the inputs to the primary outputs. We start from the output nodes and go backward to the inputs, and find out the critical path of each output. The *GlobalCut* algorithm is based on the bread-first search (BFS).

Algorithm 2: *GlobalCut* Algorithm

```
   Input  : A data flow graph G;
            An input set I_i
   Output: Executable nodes list Exec
 1 estimate( G, I_i);
 2 Exec = ∅ ;
 3 for each output node O_i do
 4     Q = queue(O_i); path_i = ∅ ;
 5     while Q! = ∅ do
 6         currNode = Q.top ;
 7         Q.pop() ;
 8         path_i.add(currNode) ;
 9         if currNode is n^s then
10             Q.push(currNode.parents) ;
11         else
12             if I_m is non-critical then
13                 Q.push(node_{I_d}) ;
14             else
15                 Q.push(node_{I_d}, node_{I_m}) ;
16     Exec = Exec ⋂ path_i
```

In this algorithm, $estimate(G, I_i)$ is to estimate the graph in log domain under input I_i. Q is a queue initialized with i_{th} output node O_i. $path_i$ contains all the critical nodes of O_i. $node_{I_d}$ and $node_{I_m}$ are the parent nodes of $currNode$ outputting I_d and I_m. The algorithm is running in $O(V + E)$, where V and E are the numbers of nodes and edges. Normally in DFG, we have $E \approx V$. So the time complexity of *GlobalCut* is linearly to the number of nodes in the graph.

GlobalCut returns the executable nodes remaining in the graph. Approximate output can be obtained by simply executing the nodes in $Exec$ topologically in linear domain.

4.4.2.2 LocalCut Algorithm

The *GlobalCut* will estimate the whole graph using a given input. But the estimated value will not update when we iteratively execute the nodes in $Exec$. Therefore, error ε_m and ε_d increase with the depth of the graph, and propagate to the final output. To solve this drawback, we propose a *LocalCut* algorithm. Different with *GlobalCut*, the *LocalCut* goes forward. It will

1. Iteratively estimates the graph till meeting the next error resilient node n_i^r.
2. Do the truncation and recomputation steps illustrated in section 4.4.1.3 for n_i^r.
3. Go back to 1) using n_i^r's recomputed result.

The detailed pseudocode is listed below. $n_i.parents.val_l$ means the logarithmic value of n_i's parents nodes.

Algorithm 3: *LocalCut* Algorithm

Input : A data flog graph G;
An input set I_i
Output: The final result of G

```
for n_i in G's topological order do
    if n_i is error sensitive then
        estimate(n_i, n_i.parents.val_l);
    else if n_i is error resilient then
        if I_m is non-critical then
            recompute I_d branch in linear domain;
        else
            recompute I_d, I_m branch in linear domain;
    Compute n_i in linear domain;
    Convert n_i to log domain for next estimation ;
```

Similar with *GlobalCut*, the timing complexity of *LocalCut* is also $O(n)$. G's topological order can be precomputed since it is not related to the input values. The *LocalCut* algorithm solves the problem 1 implicitly. If n_k is non-critical to n_i^r, *LocalCut* will replace it by estimated value. While if n_k at same time is critical to n_j^r, the algorithm will recomputed it. But the output might be affected by the execution order of n_i^r and n_j^r. If we compute n_j^r first, the result of n_k is accurate. Then the next time we compute n_i^r, We can directly use the accurate value of n_k instead of the estimated value. But on the other hand, if n_i^r comes before n_j^r, the algorithm will adopt n_k's estimated value to compute n_i^r. The *LocalCut* will not able to determine the optimal order of n_i^r and n_j^r with lack of "global" concern. So this algorithm is greedy.

We draw a example in Fig.4.5 to show the difference between *GlobalCut* and *LocalCut*. With $\delta = 2$, *GlobalCut* approximates one more node. According to the algorithm, we first compare the parents of the node "Out" and if exists non-critical

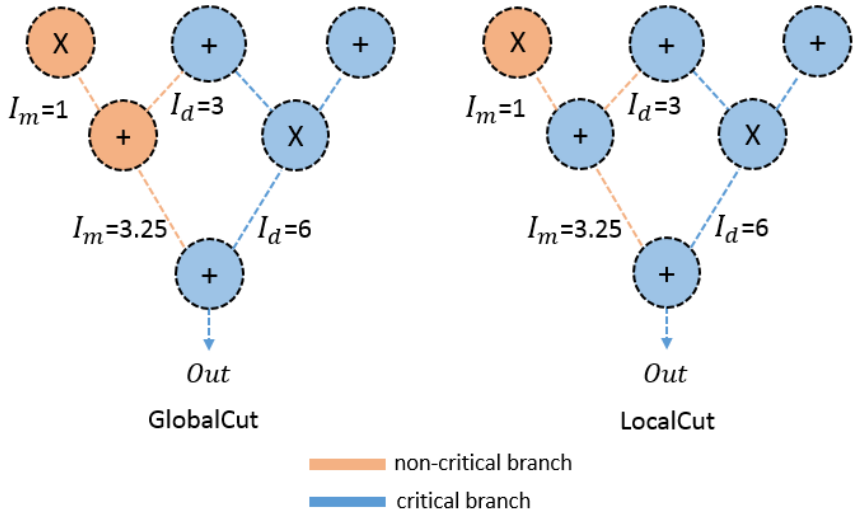

Figure 4.5: A comparative example of GlobalCut and LocalCut

parent, truncate that parent and its predecessors. In comparison, *LocalCut* always executes the error resilient nodes. So typically its energy saving is less than the *GlobalCut*. But the paths in estimation process in *LocalCut* are shorter than those in *GlobalCut*, which can effectively limit the estimated error propagation.

4.4.3 ConditionalCut Algorithm

The above two algorithms can effectively cut the non-critical nodes for every input during runtime. However, this runtime feature is built on the real time decision of the branch's criticality. What's worse, the *GlobalCut* and *LocalCut* needs V_r decisions in the worst cases, where V_r is the number of error resilient nodes. These large number of the decisions will severely affect the efficiency of the algorithms.

To reduce the number of decisions, we propose a static and runtime hybrid algorithm. The static feature can be used to reduce the decision overhead, and the runtime feature guarantees the accuracy. We apply a training procedure for the

given data flow graph. We run the *GlobalCut* algorithm T times (T=100k in this chapter), and count the times of each node being cut. Then we select the most frequently cut nodes and reshape them into C fanout cones. In the runtime, instead of making a potentially large amount of decisions, we only need to determine whether these C fanout nodes is critical or not to their children nodes, and recompute the corresponding cones if critical. We name this new method as *ConditionalCut*.

Definition 3: Let cnt_i denotes the times of node i being cut during T runs. $P_i = cnt_i/T$ denotes the probability of cutting node i, and S_i denotes the energy saving of node i if being cut. The expectation of energy saving will be $E = \sum P_i S_i$.

Lemma 1: If node k is only in node i's cone and node i is the fanout node, then: $P_k = P_i$.

Lemma 2: If node k is in the intersection of cone i and cone j, then: $P_k = P_i + P_j - P(cone_i \bigcap cone_j)$, where "$\bigcap$" means $cone_i$ and $cone_j$ will be cut at the same time.

The *ConditionalCut* can be considered as a problem that: select the nodes to formulate into C cones in order to maximize E. The optimal solution is very difficult to find because: 1) Intuitively we want to add more nodes into cones to achieve more savings, but adding the nodes may change the probabilities of its predecessors (Lemma 1). 2) It is very difficult to calculate the exact probabilities of the intersection nodes (Lemma 2). This is because $cone_i$ and $cone_j$ may or may not be independent due to the graph's shape or even the specific value of the input. To compute the probability of a single intersection node exactly, we need to store all the nodes that were cut in each time amongst T times in total, and count the

conditional appearances. What's worse, we need to count again and again when $cone_i$ and $cone_j$ changes. And things become more complicated for the nodes in the intersection of more than 2 cones. The huge memory and timing costs make this not feasible. To solve this problem, we propose a heuristic approach.

Algorithm 4: *ConditionalCut* Algorithm −− Training

Input : A data flow graph G; Number of conditions C
Output: A list of candidates contains: Conditionally approximate nodes *condNodes*; Accurate nodes *accNodes*

```
1  run GlobalCut T times;
2  for each node n_i in G do
3      count the times of being cut Cut_{n_i};
4  apxNodes = ∅, candidates = [] ;
5  for n_i in the desending order of Cut_{n_i} do
6      apxNodes.add(n_i), condNodes = ∅ ;
7      for n_i in apxNodes do
8          if n_j.children ∩ apxNodes ≠ n_j.children then
9              condNodes.add(n_j);
10     if size(condNodes) ≤ C then
11         accNodes = getAccNodes(condNodes) ;
12         candidates.append([condNodes, accNodes]) ;
13 Function getAccNodes(condNodes)
14     accNodes = [] ;
15     for each output node O_i do
16         Q = queue(O_i) ;
17         while Q ≠ ∅ do
18             currNode = Q.pop() ;
19             if currNode not in condNodes then
20                 Q.push(currNode.parents) ;
21                 accNodes.append(currNode) ;
22     return accNodes;
```

We notice a fact: $cnt_i = \sum_{j \in node_i.children} cnt_j$. This means that the parents always have larger *cnt* than their children, and if we greedily adding nodes in the descending order of *cnt*, the nodes will eventually connect consecutively from the

primary inputs towards the primary outputs. Besides, instead of maximizing E directly, we can evaluate the candidate solution t times and select the one that has the maximum energy saving. The training part of the algorithm is shown in Alg.4.

In this algorithm, line 1-3 counts the cut frequencies of each node with the complexity of $O(TV)$. Then starting from line 5, we iteratively add the most frequently cut node to the candidate pool (apxNodes). Line 8-10 find the fanout nodes amongst the pool and store them into condNodes. Line 9 can be done in $O(1)$ in the real coding, so the complexity of line 8-9 is $O(V)$. Line 11-13 collect the solutions that the number of conditional nodes is no more than C. Line 14-23 generate the node that always needs to be accurately computed with the complexity of $O(V)$. The overall complexity of the training part is $O(TV + V^2)$.

After we get the candidates of accNodes and condNodes from training step, we run each candidate t times and obtain averaged energy savings and select the one with the maximum saving. The pseudocode is shown in the runtime part.

Algorithm 5: *ConditionalCut* Algorithm −− Runtime

Input : A data flow graph G; an input set I_i;
condNodes and accNodes from Training part;
Output: The final result of G

```
1 estimate( condNodes, I_i);
2 for node n_i in the reverse order of accNodes do
3     if a parent p is a conditional node then
4         if p is critical to p.children then
5             recompute p's branch in linear domain ;
6         else
7             use p's estimated value;
8     compute n_i
9 return each O_i
```

In line 2, we traverse the node in the reverse order of accNodes. This is because the reversed accNodes is inherently sorted in topology order. The if statement in line 3 can be predecided after training. The complexity of each run is $O(V)$, so the overall evaluation procedure takes $O(tV)$. Unlike training, we do not need t to be very large, as long as we can figure out the best combinations.

4.4.4 Estimation Timing Overhead Elimination

The estimation process could bring additional timing overhead if the estimated branch will be recomputed. In this subsection, we heuristically provide two techniques for the proposed algorithms to eliminate this timing overhead.

1. Schedule the estimation process in pipeline

This technique is suitable for *GlobalCut* and *ConditionalCut* algorithm. We can estimate G using input set I_{i+1} when computing the result of the input set I_i. This approach does not work for *LocalCut* because the estimation is processed simultaneously with the computation in *LocalCut*.

2. Parallelize estimation and computation

For *LocalCut*, we can estimate the other nodes when executing the undeletable nodes.

Definition 4: A node n_k is undeletable if it can reach both inputs of an error resilient node n_i^r, and there is no other error resilient node lying in the path from n_k to n_i^r.

An example is shown in Fig.4.6. In DFG (a), node n_1 will always be executed no matter which parent branch of n_4^r is cut off. So n_1 is an undeletable nodes in

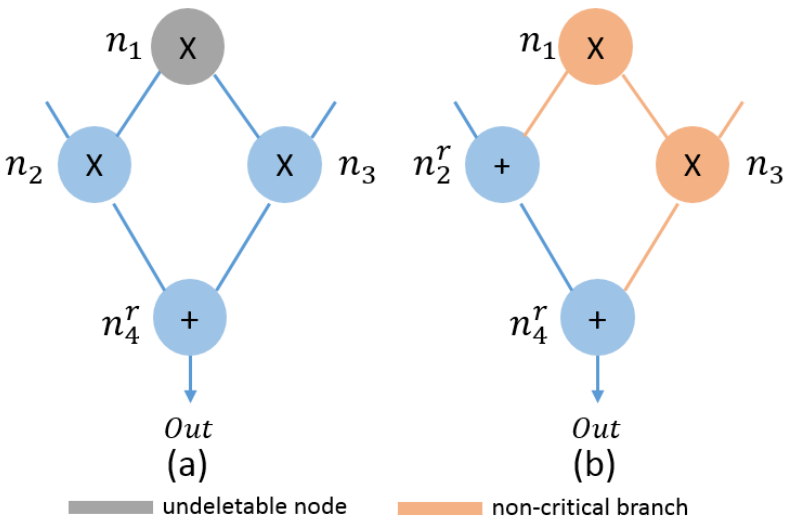

Figure 4.6: Examples of undeletable node

DFG (a). But it is not the case in DFG (b). The branch from n_1 to n_4^r might be broken by other error resilient node n_2^r, if n_1 lies on its non-critical branch and n_3 is also non-critical to n_4^r. The undeletable node can be found before the execution as long as we know the topological order of the nodes. *GlobalCut* algorithm cannot use this technique because even n_i^r itself might be cut off if it lies on the non-critical path of another error resilient node.

4.4.5 Integrating with Approximate Arithmetic

In the traditional approximate arithmetic design field, researchers are focusing on designing the approximate arithmetic units. Most of these designs are incompatible with each other and can only be used individually. However, our proposed approaches can be easily integrated with the existing approximate arithmetic designs. This is because we only truncate the graph, but do not have specific limitations of how to execute the rest nodes during the recomputation process. Therefore, we can

adopt the approximate units for the untruncated nodes to achieve additional power savings. Besides, if I_m is an estimated value, pursuing extremely high accuracy of I_d becomes less worthy. In this chapter, we use a straightforward way to compute the floating point operation approximately. For a single precision format, we use only 10 bits in mantissa.

4.4.6 Estimation in Fixed Point System

Although we utilize the property of IEEE754 floating point format to convert data into logarithmic domain, as shown in Figure 4.1, our approach can still be implemented into the fixed point system. We only need to add a shifting operation to compute the exponent for each fixed point number. When converting back from logarithmic domain to linear domain, we shift the result using the exponent number. However, for a well designed fixed point system, directly using our approach might not be very efficient. Our approach mainly focuses on the floating point system or the DFG with very wide or unknown input ranges.

4.5 Experimental Results

In this section, we first compare the power consumption of some basic arithmetic operations between the linear and log domain. To evaluate the computational qualities and energy savings, we implement *GlobalCut* and *LocalCut* algorithms and test them on the DFG benchmarks [18]. Then we use 10 mantissa bits as an approximate arithmetic technique for these two algorithms to see the influence of errors as

well as the energy savings. Moreover, to emphasize the advantages of the runtime feature in our approach, we compared the error rate of our algorithms with a naive static-based *FixCut* approach at the same energy savings. We also compare the hybrid algorithm *ConditionalCut* with previous two purely runtime algorithms. In the end, we use two machine learning examples to show that the *ConditionalCut* could be extremely powerful.

4.5.1 Arithmetic Operation Power Comparison

We implement the floating point units and our proposed logarithmic arithmetic estimation blocks in Verilog and synthesis them using Cadence RTL Compiler with FreePDK 45nm library. Clearly, we can see from Table 4.2 that the power consumption of arithmetic in log domain are 40x $\sim$ 150x smaller than those in linear domain. The significant saving can be obtained if we use estimated value instead of computing accurately.

Table 4.2: Power Comparison between log and linear domain

Power Consumption (nW)	**add**	**sub**	**mul**	**div**
Log domain	1.0	1.2	0.53	0.53
Linear domain	43.2	43.2	71.2	51.9

4.5.2 Accuracy/Energy Saving vs Threshold

We modify the data flow graph from [18]. For each DFG, we randomly generate 10,000 inputs. We implement our two proposed algorithms with threshold δ from 1 to 4 to approximate these graphs based on each input. We report the averaged

Applications	$\delta = 1$				$\delta = 2$			
	GlobalCut		*LocalCut*		*GlobalCut*		*LocalCut*	
	error	**saving**	**error**	**saving**	**error**	**saving**	**error**	**saving**
hal	6.71%	59.06%	6.73%	59.00%	1.64%	49.68%	1.66%	49.39%
cosine1	8.49%	26.73%	2.80%	16.95%	3.95%	14.99%	0.87%	10.66%
cosine2	10.17%	28.03%	4.29%	17.19%	4.93%	16.15%	1.57%	10.97%
fir1	3.29%	52.58%	2.45%	35.97%	1.74%	36.29%	1.47%	27.47%
fir2	3.43%	51.98%	3.26%	32.35%	1.66%	34.98%	1.66%	22.74%
autoRegFilter	2.79%	41.36%	1.43%	21.58%	0.89%	21.33%	0.46%	12.65%
downsample	3.04%	49.36%	2.24%	4.61%	1.39%	32.51%	0.97%	4.61%
ellipticWaveFilter	0.86%	13.42%	0.76%	5.14%	0.20%	5.71%	0.16%	2.63%
average	4.85%	40.31%	2.99%	24.10%	2.05%	26.46%	1.10%	17.64%

Applications	$\delta = 3$				$\delta = 4$			
	GlobalCut		*LocalCut*		*GlobalCut*		*LocalCut*	
	error	**saving**	**error**	**saving**	**error**	**saving**	**error**	**saving**
hal	0.74%	38.47%	0.77%	38.25%	0.31%	27.88%	0.33%	27.67%
cosine1	2.21%	8.00%	0.34%	6.42%	1.18%	4.39%	0.12%	3.83%
cosine2	2.19%	8.65%	0.51%	6.65%	0.91%	4.70%	0.09%	3.94%
fir1	0.78%	22.96%	0.73%	19.49%	0.30%	13.75%	0.31%	13.05%
fir2	0.56%	19.75%	0.58%	13.30%	0.16%	10.49%	0.17%	7.14%
autoRegFilter	0.24%	10.17%	0.13%	7.07%	0.06%	4.97%	0.04%	3.94%
downsample	0.26%	19.51%	0.17%	4.61%	0.05%	17.58%	0.04%	4.61%
ellipticWaveFilter	0.04%	3.04%	0.03%	1.40%	0.01%	2.23%	0.01%	0.81%
average	0.88%	16.32%	0.41%	12.15%	0.37%	10.75%	0.14%	8.12%

Table 4.3: Accuracy and energy savings of *GlobalCut* and *LocalCut* under different threshold

mean absolute error and energy savings in Table 4.3. From the result, we observe that:

1. Larger threshold can reduce both errors and energy savings.

2. In most cases, *GlobalCut* has more energy savings compared with *LocalCut*. This is due to the fact that *LocalCut* always executes the error resilient nodes.

3. *LocalCut* usually outputs lower error rate than *GlobalCut*. Like we discussed in section 4.4.1, this is because the *GlobalCut* estimates the whole graph at the beginning, but *LocalCut* will do partial estimation using recomputed value.

4. Different benchmark has different performance. For example, *ellipticWaveFilter* has very low errors but its energy savings are much less than the others. This is because the nodes in this benchmark are highly interfered and most of the nodes cannot be easily removed without affecting others.

4.5.3 Integrating with Approximate Arithmetic

As mentioned in subsection 4.4.5, we apply only 10 bits in mantissa during the recomputation process in both algorithms. The results are listed in Table. 4.4.

Applications	$\delta = 1$				$\delta = 2$			
	GlobalCut		*LocalCut*		*GlobalCut*		*LocalCut*	
	error	**saving**	**error**	**saving**	**error**	**saving**	**error**	**saving**
hal	6.7%	76.9%	6.72%	77.17%	1.65%	72.99%	1.67%	73.26%
cosine1	8.99%	61.28%	2.83%	55.97%	4.15%	55.53%	1.01%	53.42%
cosine2	10.29%	62.0%	3.87%	56.06%	5.0%	56.14%	1.28%	53.55%
fir1	3.47%	73.81%	2.4%	66.52%	1.73%	66.52%	1.43%	63.09%
fir2	3.74%	72.71%	3.22%	61.46%	1.77%	64.3%	1.62%	57.58%
autoRegFilter	3.25%	70.65%	1.39%	61.58%	1.01%	61.64%	0.43%	57.96%
downsample	4.43%	70.04%	2.07%	43.6%	1.93%	58.78%	0.86%	43.6%
ellipticWaveFilter	1.22%	51.47%	0.71%	47.76%	0.3%	47.15%	0.17%	46.69%
average	5.26%	67.36%	2.90%	58.76%	2.19%	60.38%	1.06%	56.14%

Applications	$\delta = 3$				$\delta = 4$			
	GlobalCut		*LocalCut*		*GlobalCut*		*LocalCut*	
	error	**saving**	**error**	**saving**	**error**	**saving**	**error**	**saving**
hal	0.76%	68.31%	0.78%	68.71%	0.34%	63.85%	0.36%	64.37%
cosine1	2.38%	52.12%	0.52%	51.68%	1.37%	50.18%	0.31%	50.59%
cosine2	2.3%	52.46%	0.67%	51.78%	1.12%	50.39%	0.31%	50.63%
fir1	0.72%	60.83%	0.69%	59.86%	0.26%	56.99%	0.27%	57.25%
fir2	0.57%	56.92%	0.55%	53.76%	0.16%	52.31%	0.16%	51.22%
autoRegFilter	0.27%	56.67%	0.12%	55.67%	0.08%	54.21%	0.05%	54.32%
downsample	0.32%	51.85%	0.17%	43.6%	0.12%	50.74%	0.11%	43.6%
ellipticWaveFilter	0.09%	45.32%	0.06%	46.16%	0.05%	44.63%	0.05%	45.9%
average	0.93%	55.56%	0.45%	53.90%	0.44%	52.91%	0.20%	52.24%

Table 4.4: Accuracy and energy savings of *GlobalCut* and *LocalCut* with only 10 mantissa bits

Compared with Table.4.3, we can see that: fewer mantissa bits will slightly increase the errors but can achieve about 45% additional savings. The 45% is estimated by:

$$\frac{Table.4.4\ saving - Table.4.3\ saving}{1 - Table.4.3\ saving}$$

One may argue that using the fewer mantissa seems to bring more energy savings than proposed algorithms, which devalues our contribution. But actually our methods are still well worth to be implemented because: 1) We focus on truncating the non-critical part of the graph using very low energy cost. The saving ratios shown in Table.III can be approximately treated to be the portions of the graphs being cut. 2) Our approach does not rely on any specific approximate arithmetic units, which means that no matter with approximate arithmetic units will be used, our approach can always contribute additional savings by reducing the number of computations. For example, reducing the mantissa bits has very limited room for the half precision floating point system, which has 5 bits in exponents and 10 bits in mantissa. But our approach is still applicable.

4.5.4 Comparison with FixCut

We randomly generate 10,000 input as training data, and apply *GlobalCut* algorithm with $\delta = 2$ for each input. We permanently replace the nodes that are most frequently removed in *GlobalCut* with their averaged values under training inputs. We name this static approximation as *FixCut*. Table.4.5 lists the errors

Table 4.5: *FixCut* error compared with *GlobalCut*

benchmarks	GlobalCut	new data	training data
hal	1.64%	109.72%	111.25%
cosine1	3.95%	72.53%	82.85%
cosine2	4.93%	257.99%	231.44%
fir1	1.74%	15.89%	15.41%
fir2	1.66%	16.52%	16.35%
autoRegFilter	0.89%	13.68%	13.68%
downsample	1.39%	18.75%	18.62%
ellipticWaveFilter	0.20%	18.12%	17.85%
average	2.05%	65.40%	63.43%

when running *FixCut* on training data and new data. For comparison, we make the energy savings of *FixCut* to be the same with *GlobalCut*. Obviously, the lack of input oriented concerns causes a tremendous error. The data in column *GlobalCut* is directly copied from Table 4.3, for comparison convenience.

4.5.5 Comparison with ConditionalCut

We force the $\delta = 2$, and use 100k inputs to train the *ConditionalCut* with the number of conditions varying from 3 to 9. The errors and energy savings are listed in Table.4.6.

The results of *GlobalCut* and *LocalCut* are directly copied from Table.4.3 for comparison convenience. C denotes the number of conditions, as well as the decisions. We can see that:

1. Increasing C gives more energy savings but larger errors. (For *hal* benchmark, the results remain the same when $C = 5, 7, 9$. This is because the number of conditions saturated after $C = 5$.)

Applications	*GlobalCut*			*LocalCut*			$C = 3$		$C = 5$		$C = 7$		$C = 9$	
$\delta = 2$	**error**	**saving**	**C**	**error**	**saving**	**C**	**error**	**saving**	**error**	**saving**	**error**	**saving**	**error**	**saving**
hal	1.64%	49.68%	4.96	1.66%	49.39%	5	1.21%	50.81%	1.34%	50.82%	1.34%	50.82%	1.34%	50.82%
cosine1	3.95%	14.99%	23.4	0.87%	10.66%	26	3.45%	6.24%	2.9%	6.93%	3.01%	7.64%	3.32%	8.35%
cosine2	4.93%	16.15%	23.0	1.57%	10.97%	26	1.7%	5.46%	2.51%	7.05%	2.71%	8.01%	2.83%	8.83%
fir1	1.74%	36.29%	9.22	1.47%	27.47%	10	1.02%	15.87%	1.25%	21.2%	1.42%	26.58%	1.51%	29.36%
fir2	1.66%	34.98%	11.20	1.66%	22.74%	15	1.25%	21.0%	1.51%	28.55%	1.61%	31.03%	1.66%	32.31%
autoRF	0.89%	21.33%	10.9	0.46%	12.65%	12	0.17%	6.03%	0.6%	14.56%	0.74%	17.42%	0.81%	18.81%
downsampling	1.39%	32.51%	20.2	0.97%	4.61%	30	0.69%	22.54%	1.1%	24.13%	1.34%	26.79%	1.51%	28.39%
ellipticWaveFilter	0.20%	5.71%	25.12	0.16%	2.63%	26	0.16%	5.16%	0.19%	5.8%	0.2%	6.34%	0.22%	6.74%
average	2.05%	26.46%	16	1.10%	17.64%	18.75	1.21%	16.64%	1.42%	19.88%	1.54%	21.82%	1.65%	22.94%

Table 4.6: Accuracy and energy savings of *ConditionalCut* with different number of conditions

2. The results of *ConditionalCut* are usually between those of *GlobalCut* and *LocalCut*.

3. More importantly, the majority of the savings can be achieved with only 5 or 7 conditions, which is far less than those of the other two algorithms. Therefore, it can be a good compromise between the savings and the runtime efficiency of the algorithm.

4.5.6 ConditionalCut in Machine Learning Applications

The benchmarks we use have massive additions and subtractions. Based on the theoretical error analysis in section IV A.4, these two operations will propagate the errors no matter how large the threshold is. But for comparative operations like min/max, it is possible to get very accurate result with a properly selected threshold. These comparative operations are more error resilient than additions and subtractions, leveraging significant energy savings with minor computation quality loss. In this section, we apply our *ConditionalCut* algorithm in the two well-known

applications: kmeans and perceptron.

The prediction of perceptron is to compute the label y, where $y = sign(W^T x + b)$ (assume linear). W and b are trained weights and bias, and x is the input data. The branch $W^T x + b$ will be cut as long as we know its sign. Therefore, we borrow the idea of *ConditionalCut* and heuristically insert two conditions, "$\geq \delta$" or "$\leq -\delta$" right after $W^T x + b$. We estimate the sign using the approach in section III, and only recompute y for those whose estimated value of $W^T x + b$ is within $(-\delta, \delta)$.

In kmeans application, a point will be clustered to the nearest center. So we need to compute the distances between a given point to all the centers, and find the minimum value amount these distances. We use our approach to quickly estimate the distances, and recompute those that are no larger than the minimum estimated distance plus threshold.

Table 4.7: Prediction error and iteration savings using ConditionalCut

	$\delta = 1$		$\delta = 2$	
Applications	**#iter saved**	**error**	**#iter saved**	**error**
Kmeans	57.22%	0.27%	42.33%	0.0%
Perceptron	91.41%	0.72%	79.34%	0.12%

The error of kmeans is the percentage of the number of points that are mis-clustered. We report the number of iterations saved when computing the distances between the points and centers in Table 4.7.

For perceptron, the error means the percentage of misclassified data compared with accurate computing. We test the pre-trained perceptron classifier with 10,000 randomly generated data. By setting $\delta = 1$, 91.41% of data can be correctly classified by only estimating y instead of accurate computing with only 0.72% error rate.

Our approach can also be implemented for the Convolutional Neural Network (CNN). CNN is one of the most important types of neural network in deep learning field. It manifests an extremely powerful capacity in pattern recognition and object detection. However, as the CNN goes deeper and deeper, the energy consumption becomes the crucial bottleneck for its scalability, transplantability, and mobility. Rectified Linear Unit, as known as ReLu, is the most frequently used activation function in CNN, whose function is $f(x) = max(x, 0)$. We can apply our approximation techniques to quickly estimate the sign of x to decide whether we need to compute x accurately. The process is very similar to the prediction of perceptron. Ideally, we can save close to 50% computations in the convolution part of the CNN.

4.6 Future Work

4.6.1 Software Implementation

In the future, we can build the programming interface with hardware and instructions support for the estimation process in logarithmic domain. For example, the logadd and logmul are the arithmetic operations shown in section 4.3.2.2, logint is the logarithmic representation format of a float number. The logint() and float() are the conversion and recovery steps shown in Fig.4.1. Then we can implement our algorithm in programming level.

Consider a piece of code that computes $a * b + c + d$:

```
float dfg(float a, float b, float c, float d){
    return a*b+c+d;
}
```

If we can observe that $a*b+c$ is frequently cut, then we can implement *ConditionalCut* algorithm and reformulate the code to be:

```
float dfg(float a, float b, float c, float d){
    logint apx = logadd(logmul(a,b), c);
    if (apx + threshold <= logint(d)){
        return float(apx)+d;
    }else{
        return a*b+c+d;
    }
}
```

The if condition judges whether the branch $a*b+c$ is critical to d or not. If critical, recompute the branch, otherwise use the approximate value directly.

4.6.2 Data Flow Graph Scheduling

We can also integrate our graph truncation algorithms with data flow graph scheduling. The truncation algorithms will runtimely change the topology of the graph. Therefore, scheduling for the original full graph will be a waste. In subsection 4.4.4, we mentioned how to integrate the estimation process with the scheduling. In the future, we can propose the randomized scheduling algorithm, which can predict the branches that will be cut, and schedule the rest nodes.

4.6.3 Application Example: Matrix Multiplication

In this subsection, we will illustrate how to use our approximation techniques for matrix multiplication. Matrix Multiplication can be decomposed as the vector multiplication. Let X and Y are two n by 1 vectors and $Z = X^T Y$. Vector multiplication contains element-wised multiplication $z_i = x_i \times y_i$ and an accumulation $Z = sum(z_i)$. Computing z_i is very easy. When converting data into logarithmic domain, the multiplication becomes a single fixed point addition. For the accumulation process, we can sum up zi in any order. We can do summation one by one, or by iteratively pair-wised. This is very trivial.

However, if we watch the logarithmic add equation carefully which is copied below for convenience, we may find that the smaller 2^{-ed} is, the smaller error the approximate addition will have.

$$\begin{aligned} S_l &= \log_2 S = \log_2 2^{A_l}(1 + 2^{B_l - A_l}) \\ &= A_l + \log_2(1 + 2^{-ed}) \approx A_l + 2^{-ed} \\ error &= \log_2(1 + 2^{-ed}) - 2^{-ed} \end{aligned}$$

So if we sum up z_i by pair and unlucky the numbers in each pair are very close, then $error$ might be not very small in each addition. After accumulation, we may not get the relatively accurate result. In this section, we propose a simple but powerful approach that minimizes the accumulation error. The algorithm is shown below:

Algorithm 6: Vector Multiplication in Logarithmic domain

Input : X and Y
Output: Z

1 convert X and Y to X_l and Y_l in log domain ;
2 m = 0 ;
3 **for** *i from 0 to n-1* **do**
4 $\quad {z_l}^i = {x_l}^i \times {y_l}^i$;
5 $\quad$ m = max(m, z_i) ;
6 Z_l = m ;
7 **for** *i from 0 to n-1* **do**
8 $\quad ed_i = round(m - {z_l}^i)$;
9 $\quad Z_l += 2^{-ed_i}$;
10 $Z = recover(Z_l - 1)$;

The basic logic is that:

1) Find the maximum number m of all ${z_l}^i$s, (in line 5).

2) For each ${z_l}^i$, compute the ed_i compared to the maximum number. So ed_i will be the minimum, since $ed_i = round(m - {z_l}^i) \geq round({z_l}^j - {z_l}^i)$ for any j.

3) Sum up 2^{-ed_i} and convert Z_l back to liner domain. In line 10, we use "-1" because there will be a ${z_l}^i$ that equals to m, which should be excluded.

One may argue that after finding m, we can accumulate Z_l by changing line 8 to $ed_i = round(Z_l - {z_l}^i)$. So Z_l will grow larger and larger, therefore making ed_i become smaller. However, compared with this proposal, our approach has two advantages:

1) Accumulating makes Z_l less accurate, leading to less accurate ed_i, and the error could boost. But in algorithm 6, each ed_i is accurately computed.

2) Accumulating makes the second for loop dependable on each iteration.

Chapter 5: Conclusion and Future Work

5.1 Conclusion

In this dissertation, we propose three approximate computing techniques for low power and energy efficiency concern. These techniques are data format oriented but not limited to the arithmetic operations. We focus on three widely used data formats: integer, fixed point, and floating point. The proposed techniques can be implemented on customized hardware platform or on the general-purpose hardware to achieve tremendous power and energy savings. Each technique has its own advantages and allows the designers to pick based on their demand.

For the general integer and fixed point computing, we propose an approximate integer format (AIF) in chapter 2. The AIF splits the operands into several blocks, and use the sentinel bits to indicate whether each block is important or not. During the computation, AIF only picks the non-zero most significant blocks and neglects the LSBs. AIF effectively truncates the bit-width to achieve the energy savings. Moreover, the bits participating in the computations are MSBs, therefore the accuracy is also guaranteed. We also provide the mathematical proof of the error bounds after truncation. Experimental results show that our AIF based approximation computing approach can achieve high accuracy, incurs very little additional overhead,

and save considerable energy. The sentinel bits computation and significant blocks selection can be done very efficiently if the designers are allowed to customize the hardware. AIF simply truncates the operands and not changes the logic of the arithmetic operations. Therefore, it can be implemented into to architecture level, and further into instruction level.

We also provide a technique in chapter 3 that can shrink the bit-width in software program level if the application's input data ranges are known. We analyze the drawbacks of traditional fixed point format and claim the advantage of the dynamic fixed point format which allows every operand has its own bit-width and scaling factor. Our technique first converts the software code into data flow graph. Then we compute the data range of each node in the graph using the primary input ranges. Next, we derive the error function and energy function with bit-width w and scaling factor e as the variables. In the end, we solve the most energy efficient combination of w and e for each node under the energy consumption constraint or maximum error constraint. The data flow graph with resolved w and e can be re-synthesized into the software program. The overall steps do not need to consider the hardware support. This technique can be wrapped as a software tool or be implemented into the compiler.

For the cases that the input ranges not strictly bounded, and the ranges are too wide to adopt fixed point design, we propose a floating point format based data flow graph approximation technique in chapter 4. Rather than focusing on truncating the bit-width, this technique aims to truncate the non-critical parts of the data flow graph. To achieve this goal, we first analyze the importance of each

node. We utilize the structure of IEEE754 format and convert the operands into the logarithmic domain. Then we propose an arithmetic estimation approach in the logarithmic domain. The criticality of a branch is judged by the estimated value. We propose three algorithms to locate the non-critical branches. Then we replace them with the estimated values, and recompute the critical branches accurately. Similar with AIF, this data flow graph approximation techniques can be implemented into programming level. An example has been shown in the future work section in chapter 4.

5.2 Future Work: Approximate Computing for Security Concern

Besides the low power and energy efficiency concerns, we can also use approximate computing to address the security concern. This section introduces a novel design concept which hides the security information via approximate computing. This concept takes both energy efficiency and security into consideration.

5.2.1 Design Motivation

Internet of Things (IoT) in this era becomes ubiquitous in human lives. Billions of devices are connected through internet infrastructure. However, this explosive increase of IoT devices brings several design challenges. IoT devices will collect, process and exchange massive data that could be confidential or privacy-sensitive. Therefore the design and manufacture processes of IoT devices need to consider security and privacy in order to avoid potential malicious attacks and design flaws.

Moreover, powering these tens of billions IoT devices is another challenge that needs to be well addressed. A significant percentage of IoT devices will be wireless and battery driven, so the power consumption directly affects the working time of these devices.

However, these two challenges bring us into a dilemma: we need low power design to achieve longer working time, but we also need to build power hungry security system to prevent attacks. Classical security design based on the modern cryptography becomes unsuitable for IoT devices due to its expensive computational resource demand. To balance the tradeoff between security and resource constraints, lots of hardware primitives based security mechanisms [53, 54] have been proposed to meet the IoT requirements. But all these existing approaches still need additional hardware besides the original functionalities. As a result, many IoT devices such as implantable medical devices do not have any protection on the data.

In this chapter, we show how approximate computing can help to address the security challenges in IoT design with low power concern. We target on the hardware security vulnerability and low power requirements of IoT. We first survey the existing design issues and summarize a couple of mitigations to address these issues. Then we propose an approximate computing based security embedding approach that takes both security and low power into consideration. The proposed approach provides a low power lightweight authentication mechanism without introducing additional hardware for security purpose, which can efficiently save the power consumption.

The design concepts in this section have been published in 2017 IEEE International Symposium on Circuits and Systems (ISCAS) [52].

5.2.2 Security and Privacy Challenge in IoT

IoT devices are suffered from several malicious attacks, such as the malware and Trojan embedding in software and hardware, unsophisticated EDA design tools [37], and untrusted supply chains. Although there exist many well-designed security protocols based on modern cryptography, the low power design requirement of IoT devices makes them unsuitable because they are computationally expensive and power hungry. As a result, many IoT devices such as implantable medical devices do not have any protection on the data. On the other hand, major security threats to IoT device itself exist, as we will survey later in this section.

There is active research on lightweight cryptography which aims to deliver affordable but weak security (for example, with short cryptographic keys) to IoT devices [38] [39] as well as on hardware security primitives such as silicon physical unclonable functions (PUFs) and hardware based random number generator and authentication. These approaches are much more energy efficient than applying the classic cryptographic solutions. However, they add security as a non-functional feature into the system and will need hardware or software support. In the next section, we will show how approximate computing can be used for information hiding to address some of these challenges.

Hardware is the fundamental element in IoT, but it is now becoming a new attacking surface through various physical attacks such as the hardware Trojan injection, side-channel analysis attacks, reverse engineering (RE) and intellectual property (IP) infringements. These problems have been addressed in the hard-

ware design community for more than two decades, but they are unique and more dangerous for IoT devices, which are not fabricated with the latest semiconductor technology and make the above attacks much easier. Next, we briefly describe these attacks and introduce available countermeasures.

Hardware Trojan is a piece of circuit that is implanted to the design or modified from the original design for malicious purposes. It can be as simple as several logic gates, but it can cause severe damage such as altering or disabling certain function units, leaking sensitive information, or shortening the lifetime of IoT devices.

Side channel attack is a classic non-invasive and passive attack by monitoring, measuring and analyzing the systems physical characteristics leaking from side channels when the system is running. These characteristics include timing, current, voltage, electromagnetic radiation, power consumption, optical or acoustic information, etc. Side channel attacks can be very effective, are easy to implement but hard to detect and prevent. Moreover, they target vulnerabilities in the hardware or software implementation instead of the algorithms or protocols. Therefore theoretically proved secure algorithms or protocols may become vulnerable if they are not implemented properly.

Reverse engineering (RE) is the invasive process of extracting IP from an IoT device and reproducing it based on the achieved information with little or no investment in research and development. These low-cost illegitimate products can be sold at a much lower price, giving them unfair competitive edge against the authentic products. Moreover, when the high level functionality of the IoT

device is extracted, the attacker can redesign the device to avoid the infringement of copyright, or insert hardware Trojan into the system for malicious purposes. RE attacks can be very effective against IoT devices because they are simple and not using the latest semiconductor technology (to raise the cost of RE tools).

IP watermarking embeds the signature of the designer to claim the authorship and IP rights. It is used to detect and catch IP piracy. The carefully designed watermark can provide high confidence of IPs authorship, incur low design overhead, and are resilient against various attempts of watermark removal and modification. However, since it is designed to prove IPs authorship, the watermark remains the same for all copies of a given IP and one cannot trace the source of illegally distributed IPs. Digital fingerprinting addresses this concern by embedding IP users information together with the IP authors signature. In that sense, fingerprinted IPs can be viewed as multiple distinct watermarked IPs. Its goal is to identify each copy of the IP and thus protect honest IP users.

While digital watermarking and fingerprinting techniques can deter IP piracy, they do not prevent IP misuse and reverse engineering from happening. **Circuit obfuscation** [55] takes a step towards this direction by making RE harder. It seeks to modify the design and implementation of a circuit in order to make it difficult to interpret the layout and hence increase the cost and complexity of RE attacks. Currently, there are two types of obfuscation techniques: logic encryption and circuit camouflage. Logic encryption is based on the insertion of additional key gates that have the secret key values as part of their input signals. The key gates and thus the entire circuit will malfunction on incorrect key values. Circuit

camouflage is based on replacing original logic gates with configurable logic cells. These configurable cells can be configured to perform as different logic gates, but the difference between configurations is too little to be observed by existing reverse engineering tools.

Physical Unclonable Function (PUF) [56, 57]is a small piece of circuitry embedded in the design that extracts silicon chips fabrication variation and uses such intrinsic physical feature for security applications. PUF has been successfully used for secret key storage, random number generation, chip authentication, intellectual property protection, and anti-counterfeiting. It can be a promising hardware security primitive for IoT devices. However, its usability is limited due to its reliability concerns under different operating environments such as supply voltage, temperature, and humidity as well as circuit aging. Therefore, plenty of work has been proposed to enhance its robustness and stability.

5.2.3 Low Power Techniques for IoT Devices

Most of the IoT devices are resource limited and usually working in the self-sustaining operating environment. Besides, the diversity of IoT devices poses a variety of special low power requirements. Smart power management and low power techniques become the key factors to guarantee the quality and stability of IoT system. In this section, we briefly introduce several advanced research achievements that meet the low power demand in IoT system.

Dynamic Voltage and Frequency Scheduling: Dynamic voltage and Fre-

quency scheduling (DVFS) varies the clock frequency and the supply voltage based on the computation load and deadline requirements to provide an acceptable performance while minimizing the total amount of energy consumption. State-of-the-arts DVFS design has been proposed [1] for the functionalities which are conventionally used in IoT devices.

Circuit Level Low Power Techniques: The power dissipation model in circuit level mainly contains three parts: dynamic power, static/leakage power, and short-circuit power. In today's VLSI technology, dynamic and leakage power are the dominant consumption. Researchers have proposed various techniques targeting the circuit to minimize the power, for example, transistor sizing [2], glitch and path balancing, technology mapping, temperature/thermal aware [3], and dual threshold voltage and input vector control [4], and don't care condition optimization.

Probabilistic Design: Probabilistic design [40] methodology is aiming at the proper design instead of over-design. The majority of the existing design techniques that guarantee the worst case execution often lead the system to be over-designed. Probabilistic design in IoT system utilizes the fact that most devices are fault tolerable and do not require high performance. It uses prior or posterior execution information and takes advantages of the unique features of IoT devices functionalities in order to relax the rigid hardware or software over-designed implementation.

Energy Harvesting: No matter how effective the low power design could be, the IoT devices will be out of power sooner or later. Energy harvesting [5] allows devices collect and store the energy from surrounding environment such as thermal energy, solar power or wind. Cooperating with the low power designs,

energy harvesting can largely prolong the lifetime of the devices.

5.2.4 Security Information Hiding Mechanism and Protocol

To minimize the power cost while still providing a practical solution for the IoT security problems, we introduce a novel approximate computing based approach to embed information for authentication and other security related applications. The idea is inspired by data segmentation where the operands are divided into most significant bits (MSB) and least significant bits (LSB). In approximate computing, the MSB part is used for precise operation and should be preserved, but the LSB part is either ignored or replaced by simple operations such as logical OR. In our approach, we hide information into LSB such that it does not affect approximate computation but can be recovered for security purpose. The implementation of our approach requires slight modification to the arithmetic unit, e.g. adder or multiplier, without building any extra hardware primitives and corresponding systems.

5.2.4.1 Floating Point Format with Security Embedding

In IEEE754 single precision standard (double precision is very expensive and seldom used in IoT devices), the 32-bit data consists of 1 sign bit, 8 exponent bits and 23 mantissa bits, as shown in the figure below. The value of an IEEE 754 number can be computed by $sign \times mantissa \times 2^{exponent}$, where the sign can be 1 and -1 if the leading bit is 0 and 1, respectively; the mantissa is a real value between 1.0 and 2.0 with fractional part represented in binary; the exponent equals to the 8 bits in

the middle subtracting 127. For example, 0,10000000,10010010000111111010000 is $1 \times 1.570795... \times 2$, which is roughly 3.14159 in decimal.

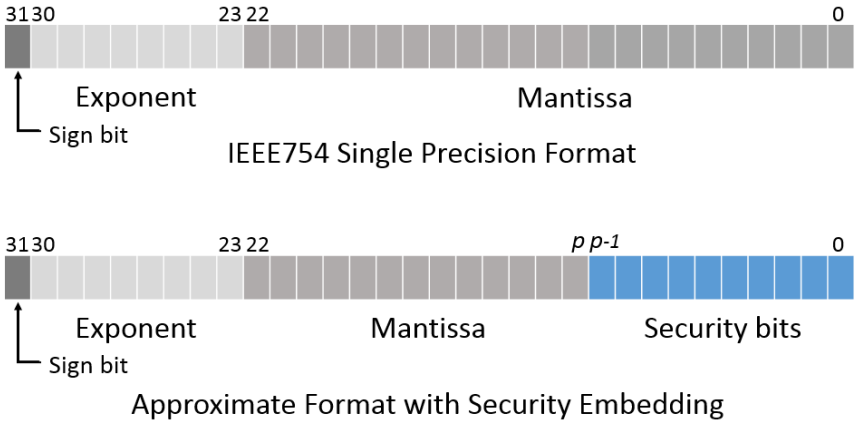

Figure 5.1: Approximate single precision floating point format

Obviously, the LSBs in mantissa will have little impact on the value. Therefore, we propose to use the last p bits, which we call security bits as shown in Fig.5.1, to embed information without changing the rest 32-p bits. In the above example, if we set the last 10 bits to 0s, the value will become 3.1413574, only a 0.0074% decrease from the original value.

Lemma 1: The error introduced by any changes to the last p bits will be less than 2^{p-24} relatively to the original value.

Proof: The maximum error rate when truncating p least significant bits in mantissa will be $er = \frac{maxEr}{minRes+maxEr}$, where $maxEr$ is the maximum error value which is 2^{p-1} and $minRes$ is the minimum residual after truncation which is 2^{23}. So $er = \frac{2^{p-1}}{2^{23}+2^{p-1}} \leq \frac{2^{p-1}}{2^{23}} = 2^{p-24}$. Let $p = 10$, the maximum error rate will be guaranteed no larger than 6.1×10^{-5}. The overall computation quality can be guaranteed with proper selected p.

5.2.4.2 Information Hiding via Approximate Computing

Given two real numbers A and B, we rewrite them in the approximate data format: $A = A' \oplus K_A$and $B = B' \oplus K_B$, where A' and B' are identical to A and B except that the last p mantissa bits are replaced by 0's; K_A and K_B are the last p bits of A and B; $\oplus$ is the bitwise XOR for the last p bits. For any binary arithmetic operation $A \otimes B$, we propose the following method to perform approximate computing and information embedding simultaneously:

1. rewrite A and B in the approximate data format
2. compute $A' \otimes B'$ and rewrite it as $O \oplus K_{O'}$
3. generate a p-bit secret K_S to be embedded
4. return $O' \oplus K_S$ as the result of $A \otimes B$

For example, with A=3.14159 and B=12.31, the previous result of A × B will be 38.6729729. Following the above scheme with p=10, we have:

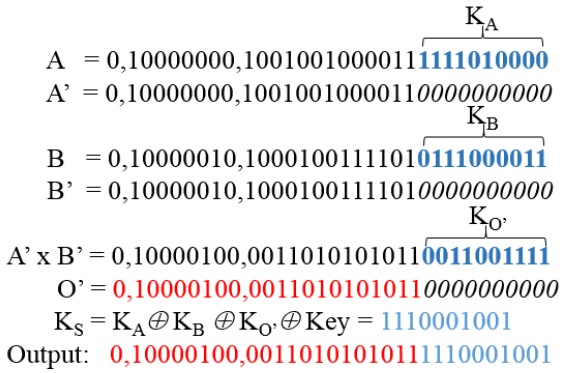

Figure 5.2: An example of information hiding via approximate computing

where we generate the secret K_S using the simple bitwise XOR of K_A,K_B,K_O and a random $Key = 01010101$. The output value is 38.67124, 0.00448% less than the accurate result 38.6729729.

There are two potential threats to this information hiding protocol. First, an attacker can study the values of the operands and the results in order to reveal the Key value, the secret K_S, and the function that generates K_S. Second, authorized parties should be able to reveal the hidden information to verify the watermark and fingerprint or decrypt the encrypted results. An insider attacker or someone who has managed to gain this permission can further alter or forge the hidden information. These attackers can be easily prevented by enhancing the proposed method with additional cost. For example, the simple XOR operation can be replaced by a one-way hash function; information can be hidden on multiple operands and at different computation stage during a complex operation; intrinsic hardware features such PUF and secure memory can be utilized to secure Key and K_S.

The main advantage of our proposed approximate computing based information hiding method is its potential for low cost implementation with the guarantee on the result. This is a result of the fact that we have utilized the energy efficiency of approximate computing and the tolerable error it introduces. The hidden information can be verified easily by disabling the approximate computing. As shown above, this security primitive can provide lightweight security for multiple applications such as authentication and IP protection. Our next step is to find an efficient way to implement a real hardware secure system by using proposed approach.

5.2.5 Information Hiding for Security Applications

The secret K_S can be as simple as a constant or some function of K_A, K_B, $K_{O'}$ and Key. In general, we can write this as $K_S = F(K_A, K_B, K_{O'}, Key)$. We now give several examples where K_S can be generated and used for different purposes.

IP Watermarking: during the design and implementation of an IP, we can either use IP owners digital signature as the Key or pick selective operand values to enable the above proposed steps. By enabling, we mean a mechanism such as returning $A' \otimes B'$ directly (and skipping steps 3 and 4) unless the operands match the given values when the watermark is embedded or a watermark verification signal is activated. Then by checking the output error, we can reveal the watermark.

Device authentication/fingerprinting: similar to watermarking, the unique fingerprint of each device can be embedded as the error value in the LSB. For the same verification inputs or operands, we can select different Key values and different F()'s so we will be able to distinguish all the individual devices.

Lightweight encryption: Since the result will be encrypted, we can use the entire 32-bit space for information hiding. For example, an encryption key up to 32 bits can be generated from function F() based on the values of operands and some given Key values. This encryption key will be used to encrypt the (approximated) computational result, for example by the efficient bitwise XOR operation. This is a symmetric encryption and the result can be decrypted easily once the key is available.

Bibliography

[1] L. Yuan and G. Qu. "Analysis of Energy Reduction on Dynamic Voltage Scaling-Enabled Systems", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems Vol. 24, No. 12, pp. 1827-1837, December 2005.

[2] J. Gu, L. Yuan, Z. Chen, and G. Qu "Improving Dual Vt Technology by Simultaneous Gate Sizing and Mechanical Stress Optimization," IEEE/ACM International Conference on Computer Aided Design (ICCAD'11), November 2011.

[3] J. Yu, Q. Zhou, G. Qu, and J. Bian, "Behavioral Level Dual-Vth Design for Reduced Leakage Power with Thermal Awareness," Design, Automation and Test in Europe (DATE'10), pp. 1261-1266, March 2010.

[4] L. Yuan, and G. Qu. "Simultaneous Input Vector Selection and Dual Threshold Voltage Assignment for Static Leakage Minimization," IEEE/ACM International Conference on Computer Aided Design (ICCAD'07), pp. 548-551, November 2007.

[5] M. Gorlatova, J. Sarik, G. Grebla, M. Cong, I. Kymissis and G. Zussman, "Movers and Shakers: Kinetic Energy Harvesting for the Internet of Things," in IEEE Journal on Selected Areas in Communications, vol. 33, no. 8, pp. 1624-1639, Aug. 2015.

[6] T. Yeh, P. Faloutsos, M. Ercegovac, S. Patel and G. Reinman, "The Art of Deception: Adaptive Precision Reduction for Area Efficient Physics Acceleration," 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007), Chicago, IL, 2007, pp. 394-406. doi: 10.1109/MICRO.2007.9

[7] Yuntan Fang, Huawei Li, and Xiaowei Li. "SoftPCM: Enhancing energy efficiency and lifetime of phase change memory in video applications via approximate write". In Asian Test Symposium (ATS), 2012.

[8] M. R. Choudhury and K. Mohanram, "Approximate logic circuits for low overhead, non-intrusive concurrent error detection," 2008 Design, Automation and Test in Europe, Munich, 2008, pp. 903-908. doi: 10.1109/DATE.2008.4484789

[9] S. Misailovic, S. Sidiroglou, H. Hoffmann and M. Rinard, "Quality of service profiling," 2010 ACM/IEEE 32nd International Conference on Software Engineering, Cape Town, 2010, pp. 25-34. doi: 10.1145/1806799.1806808

[10] H.Hoffmann,S.Misailovic,S.Sidiroglou, A. Agarwal, and M. Rinard, "Using code perforation to improve performance, reduce energy consumption, respond to failures," Massachusetts Inst. Technol. (MIT), Cambridge, MA, USA, Tech. Rep. MIT-CSAIL-TR-2009-042, 2009.

[11] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. "Paraprox: pattern-based approximation for data parallel applications". In Proceedings of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS '14). ACM, New York, NY, USA, 35-50. DOI: http://dx.doi.org/10.1145/2541940.2541948

[12] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum, "Church: A language for generative models," in Proc. Uncertainty Artif. Intell., 2008, pp. 220229.

[13] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. In IEEE/ACM International Symposium on Microarchitecture (MICRO), 2013.

[14] Debabrata Mohapatra, Vinay K Chippa, Anand Raghunathan, and Kaushik Roy. Design of voltage-scalable meta-functions for approximate computing. In Design, Automation and Test in Europe (DATE), 2011.

[15] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In IEEE/ACM International Symposium on Microarchitecture (MICRO), 2012a.

[16] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: approximate data types for safe and general low-power computation. In ACM Conference on Programming Language Design and Implementation (PLDI), 2011.

[17] Sparsh Mittal. "A Survey of Techniques for Approximate Computing". ACM Comput. Surv. 48, 4, Article 62 (March 2016), 33 pages. DOI: https://doi.org/10.1145/2893356

[18] http://www.ece.ucsb.edu/EXPRESS/benchmark/

[19] S. Hashemi, R. I. Bahar and S. Reda, "DRUM: A Dynamic Range Unbiased Multiplier for approximate applications," 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Austin, TX, 2015, pp. 418-425. doi: 10.1109/ICCAD.2015.7372600

[20] Rong Ye, Ting Wang, Feng Yuan, Rakesh Kumar, and Qiang Xu. 2013." On reconfiguration-oriented approximate adder design and its application". In Proceedings of the International Conference on Computer-Aided Design (ICCAD '13). IEEE Press, Piscataway, NJ, USA, 48-54.

[21] Kumud Nepal, Yueting Li, R. Iris Bahar, and Sherief Reda. 2014. "ABACUS: a technique for automated behavioral synthesis of approximate computing circuits". In Proceedings of the conference on Design, Automation & Test in Europe (DATE '14).

[22] Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. 2014. "ASAC: automatic sensitivity analysis for approximate computing". SIGPLAN Not. 49, 5 (June 2014), 95-104.

[23] M. Gao, Q. Wang, A. S. K. Nagendra and G. Qu, "A novel data format for approximate arithmetic computing", 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), Chiba, 2017, pp. 390-395.

[24] Chaofan Li, Wei Luo, S. S. Sapatnekar and Jiang Hu, "Joint precision optimization and high level synthesis for approximate computing", 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), San Francisco, CA, 2015, pp. 1-6.

[25] J. Riehme and U. Naumann "Significance analysis for numerical models" 1st Workshop on Approximate Computing (WAPCO) 2015

[26] D. S. Khudia, B. Zamirai, M. Samadi and S. Mahlke, "Rumba: An online quality management system for approximate computing", 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), Portland, OR, 2015, pp. 554-566.

[27] Qiang Xu, Nam Sung Kim, and Todd Mytkowicz."Approximate Computing: A Survey". In: vol. 33. 1.Feb. 2016, pp. 822.

[28] Mingze Gao, Qian Wang, Md Tanvir Arafin, Yongqiang Lyu, Gang Qu: "Approximate Computing for Low Power and Security in the Internet of Things". IEEE Computer 50(6): 27-34 (2017)

[29] Daisuke Miyashita, Edward H. Lee and Boris Murmann: "Convolutional Neural Networks using Logarithmic Data Representation". In CoRR. abs/1603.01025. 2016, http://arxiv.org/abs/1603.01025

[30] N. M. Ho, E. Manogaran, W. F. Wong and A. Anoosheh, "Efficient floating point precision tuning for approximate computing," 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), Chiba, 2017, pp. 63-68. doi: 10.1109/ASPDAC.2017.7858297

[31] Matthieu Courbariaux and Yoshua Bengio and Jean Pierre David, "Low precision arithmetic for deep learning", CoRR 2014, http://arxiv.org/abs/1412.7024

[32] N. Zhu, W. L. Goh, W. Zhang, K. S. Yeo and Z. H. Kong, "Design of Low-Power High-Speed Truncation-Error-Tolerant Adder and Its Application in Digital Signal Processing," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 18, no. 8, pp. 1225-1229, Aug. 2010. doi: 10.1109/TVLSI.2009.2020591

[33] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie and C. Lucas, "Bio-Inspired Imprecise Computational Blocks for Efficient VLSI Implementation of Soft-Computing Applications," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 57, no. 4, pp. 850-862, April 2010. doi: 10.1109/TCSI.2009.2027626

[34] Y. Kim, Y. Zhang and P. Li, "Energy Efficient Approximate Arithmetic for Error Resilient Neuromorphic Computing," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 23, no. 11, pp. 2733-2737, Nov. 2015. doi: 10.1109/TVLSI.2014.2365458

[35] Ning Zhu, W. L. Goh and K. S. Yeo, "An enhanced low-power high-speed Adder For Error-Tolerant application," Proceedings of the 2009 12th International Symposium on Integrated Circuits, Singapore, 2009, pp. 69-72.

[36] J. Hu and W. Qian, "A new approximate adder with low relative error and correct sign calculation," 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, 2015, pp. 1449-1454.

[37] G. Qu and L. Yuan, "Design THINGS for the Internet of Things An EDA perspective," 2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), San Jose, CA, 2014, pp. 411-416. doi: 10.1109/ICCAD.2014.7001384

[38] Katagi, Masanobu and Moriai, Shiho, "Lightweight Cryptography for the Internet of Things", Sony Corporation, 2012.05

[39] M. T. Arafin, M. Gao and G. Qu, "VOLtA: Voltage over-scaling based lightweight authentication for IoT applications," 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), Chiba, 2017, pp. 336-341. doi: 10.1109/ASPDAC.2017.7858345

[40] Shaoxiong Hua, Gang Qu and S. S. Bhattacharyya, "An energy reduction technique for multimedia application with tolerance to deadline misses," Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451), 2003, pp. 131-136. doi: 10.1109/DAC.2003.1218868

[41] D. U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk and G. A. Constantinides, "Accuracy-Guaranteed Bit-Width Optimization," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 25, no. 10, pp. 1990-2000, Oct. 2006. doi: 10.1109/TCAD.2006.873887

[42] J. Chung and L. W. Kim, "Bit-Width Optimization by Divide-and-Conquer for Fixed-Point Digital Signal Processing Systems," in IEEE Transactions on Computers, vol. 64, no. 11, pp. 3091-3101, Nov. 1 2015. doi: 10.1109/TC.2015.2394469

[43] S. Lee and A. Gerstlauer, "Fine grain word length optimization for dynamic precision scaling in DSP systems," 2013 IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC), Istanbul, 2013, pp. 266-271. doi: 10.1109/VLSI-SoC.2013.6673287

[44] D. Williamson, "Dynamically scaled fixed point arithmetic," [1991] IEEE Pacific Rim Conference on Communications, Computers and Signal Processing Conference Proceedings, Victoria, BC, 1991, pp. 315-318 vol.1. doi: 10.1109/PACRIM.1991.160742

[45] Philipp Gysel, Mohammad Motamedi and Soheil Ghiasi, "Hardware-oriented Approximation of Convolutional Neural Networks", arXiv 2016, https://arxiv.org/abs/1604.03168

[46] R. Moore, "Interval Analysis", Englewood Cliffs, NJ: Prentice-Hall. 1966

[47] Henrique de Figueiredo, Luiz & Stolfi, Jorge. (2003). "Affine Arithmetic: Concepts and Applications."

[48] M. Imani, D. Peroni and T. Rosing, "CFPU: Configurable floating point multiplier for energy-efficient computing," 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, 2017, pp. 1-6. doi: 10.1145/3061639.3062210

[49] FPBench, http://fpbench.org/

[50] Eva Darulova, Viktor Kuncak, Rupak Majumdar, and Indranil Saha. 2013. "Synthesis of fixed-point programs". In Proceedings of the Eleventh ACM International Conference on Embedded Software (EMSOFT '13). IEEE Press, Piscataway, NJ, USA, Article 22, 10 pages.

[51] M. Gao and G. Qu, "Energy efficient runtime approximate computing on data flow graphs," 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Irvine, CA, 2017, pp. 444-449. doi: 10.1109/ICCAD.2017.8203811

[52] M. Gao and G. Qu, "A novel approximate computing based security primitive for the Internet of Things," 2017 IEEE International Symposium on Circuits and Systems (ISCAS), Baltimore, MD, 2017, pp. 1-4. doi: 10.1109/ISCAS.2017.8050360

[53] Md Tanvir Arafin and Gang Qu. 2016. "Secret Sharing and Multi-user Authentication: From Visual Cryptography to RRAM Circuits". In Proceedings of the 26th edition on Great Lakes Symposium on VLSI

[54] M. T. Arafin, M. Gao and G. Qu, "VOLtA: Voltage over-scaling based lightweight authentication for IoT applications," 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), Chiba, 2017, pp. 336-341. doi: 10.1109/ASPDAC.2017.7858345

[55] S. M. Awan, S. Rashid, M. Gao and G. Qu, "Security through obscurity: Integrated circuit obfuscation using don't care conditions," 2016 International Conference on Control, Automation and Information Sciences (ICCAIS), Ansan, 2016, pp. 64-69. doi: 10.1109/ICCAIS.2016.7822437

[56] M. Gao, K. Lai, J. Zhang, G. Qu, A. Cui and Q. Zhou, "Reliable and Anticloning PUFs Based on Configurable Ring Oscillators," 2015 14th International Conference on Computer-Aided Design and Computer Graphics (CAD/Graphics), Xi'an, 2015, pp. 194-201. doi: 10.1109/CADGRAPHICS.2015.54

[57] Mingze Gao, Khai Lai, and Gang Qu. 2014. "A Highly Flexible Ring Oscillator PUF". In Proceedings of the 51st Annual Design Automation Conference (DAC '14). ACM, New York, NY, USA, Article 89, 6 pages. DOI: https://doi.org/10.1145/2593069.2593072