Title:

Error Distribution Estimation for Fixed-point Arithmetic using Program Derivatives

Authors:

Soramichi Akiyama, Ryota Shioya, Yuto Miyatake, and Tongxin Yang

Published in:

The 25th International Symposium on Quality Electronic Design (ISQED)

Link to IEEE Xplore:

https://ieeexplore.ieee.org/document/10528683

Error Distribution Estimation for Fixed-point Arithmetic using Program Derivatives

Soramichi Akiyama*, Ryota Shioya†, Yuto Miyatake‡ and Tongxin Yang§¶
*College of Information Science and Engineering, Ritsumeikan University, Osaka, Japan
†Graduate School of Information Science and Technology, The University of Tokyo, Tokyo, Japan
‡Cybermedia Center, Osaka University, Osaka, Japan
§Sony Semiconductor Solutions Corporation, Kanagawa, Japan
Email: s-akym@fc.ritsumei.ac.jp, shioya@ci.i.u-tokyo.ac.jp, yuto.miyatake.cmc@osaka-u.ac.jp, yangtongxin@ieee.org

***Abstract*—Fixed-point arithmetic is widely used because of its efficiency in latency, area, and power consumption. However, determining the number of bits assigned to each variable while considering the balance of efficiency and the error of the program output is challenging. To ease this burden, we (1) propose a new method that estimates the statistical error distributions of the program output when fixed-point arithmetic is used, and (2) implement an error estimation system for HLS programs based on our method. The main idea is to apply an error propagation model based on program derivatives to the distributions of data and their errors. This achieves estimating not only the range of the errors but also the statistical aspects of them without feeding a lot of input data to the program. Furthermore, the input data and their errors can be tweaked at the distribution level, allowing an easy-to-conduct robustness analysis of chosen precision. Our experiments show that our method can estimate error distributions for various operators and a realistic application well, and the estimated results can be used for different types of analyses that help the user determine precision in fixed-point arithmetic.**

***Index Terms*—Fixed-point Arithmetic, Error Estimation, Program Derivatives**

I. Introduction

Fixed-point arithmetic is widely used when losing computational accuracy in return for efficiency is acceptable. Examples include, but not limited to, FPGA application designs [1], artificial intelligence [2], and specialized hardware for industrial applications [3]. For example, Cabello *et al.* [4] use fixed-point arithmetic on an FPGA to implement Gaussian Filter, whose output is typically displayed to human eyes. They show that the look-up table usage is 24.2× smaller than that of the floating-point counterpart with less than 5% Root Mean Squared Error in the final images when 11-bit fixed-point arithmetic is used.

Choosing the right *precision* for fixed-point numbers used in fixed-point arithmetic is key for efficiency. The precision of a fixed-point number refers to the number of bits assigned to it (i.e., bit-width). It must be chosen small enough not to cancel the efficiency, but large enough not to incur too much *error*. An error means the difference between the output of a fixed-point program and the ideal (or mathematical) result. Fig. 1 shows the overhead of fixed-point multipliers ($z = x \ast y$) with various precision. We use FreePDK's 45 nm process for synthesis and simulation and the power simulations were conducted on 1,000,000 random inputs. The operating conditions for synthesis were typical (a 1.00 process factor, 1.1 V power supply, and 25°C operating temperature). All multipliers were synthesized and optimized using the default compiler options. Synopsys Power Compiler was used to estimate power consumption from switching activity interchange format files generated from the Synopsys VCS. The precision of x, y, and z are swiped between 12 bits and 15 bits, while the number of bits assigned to the integer part is fixed to 4. For example, 12_13_14 in the graph means that the fractional parts of x, y, and z are assigned 8 bits, 9 bits, and 10 bits, respectively. The vertical axes show the overhead compared to the 12_12_12 case. The efficiency of this multiplier differs by as much as

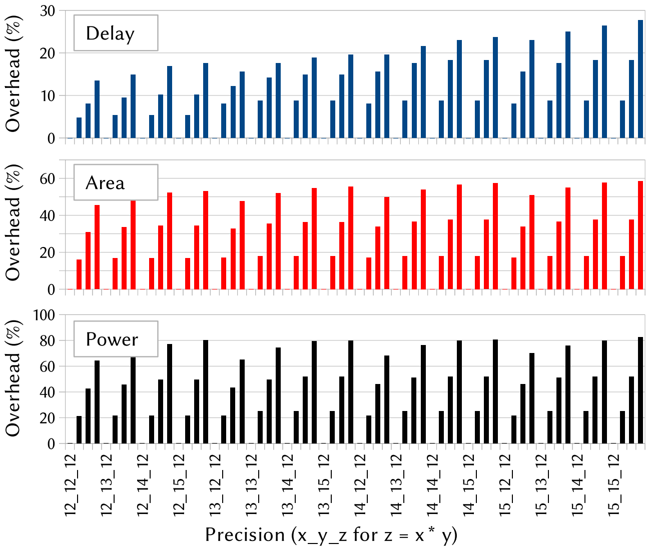

Fig. 1. Relative overhead of delay, area, and power of fixed-point multipliers that calculate z = x * y with various precision. Label i_j_k represents that the precision of x, y, and z are i, j, and k, respectively (every fourth of them are shown for brevity). The baseline of the comparison is 12_12_12.

This work was supported by JSPS Grants-in-Aid for Scientific Research Grant Number JP23H03388.

¶This work was done while the author was with The University of Tokyo, Japan.

80+% depending on the precision.

Determining the right precision is key but difficult because the errors are both code- and data- dependent, and the statistical aspects of errors must be taken into account. To be concrete, we break down the difficulty into the following four challenges:

1) The errors cannot be statically estimated because they depend not only on the program code but also on the input data.
2) The errors must be estimated based on a large amount of real data rather than on a small amount of sampled data to not miss the long-tail of errors.
3) The statistical aspects of errors differ largely from a normal (gaussian) distribution and from each other in different precision.
4) The number of possible precision is large in hardware circuit designs and FPGA configurations, unlike CPU and GPU programs.

To tackle these challenges, we propose a new method to estimate the error *distribution* of a program's output for given precision. A distribution (formally defined in Section III-A) of a set of data represents the statistical aspects of them, and the user can calculate not only the average and the standard deviation but also the probability that the data falls into any given range. Our method estimates the distributions of errors by modeling how errors propagate based on derivatives of operators used in the target program.

Our method solves the aforementioned challenges as follows. First, it considers both the program code and the input data to estimate errors. Second, the input to our method is distributions of input variables created from a large amount of real data. This enables our method to properly take the long-tail of errors into account. Third, it outputs distributions of the errors so that the user can investigate the statistical aspects of them to determine precision. This includes calculating the probability that the error falls into a certain range and tweaking the distributions of values/errors of the input to consider extreme-case scenarios (e.g., when the error of input data is much higher than expected). Fourth, our method is faster than repeatedly executing the target program normally to estimate distributions of errors. This is because our method manipulates distributions of variables directly based on the program code.

The contributions of this paper are as follows.

- We are the first to apply program derivatives and distribution estimation for determining the precision of fixed-point arithmetic as far as we know.
- We give concrete algorithms to estimate errors of programs that consist of `+`, `*`, and `sin` operators.
- Based on the algorithms, we implement an error estimation system that targets High-Level Synthesis (HLS) programs written in C++.
- With the estimation system, we show that the error distributions of the aforementioned operators and a realistic application can be estimated well.
- We also provide two types of analyses on errors estimated by our system; i.e., the probability that they fall within 1σ from the average (σ denotes the std. dev.), and how they change when the errors of input variables are increased.

II. Research Background

A. Details of the Challenges

Here we elaborate on the details of the challenges in determining the precision for fixed-point arithmetic.

1) **Code- and data- dependency**: The errors depend not only on the program code but also on the input data. Let us consider a simple program `y = 1.1b * x` and we assign 1 bit to the fractional parts of `x` and `y`. The error of `y` depends on the value of `x`, as well as of course on the program code. For example, when `x = 1.0b`, the error of `y` is 0 as it can be accurately calculated. On the other hand, when `x = 1.1b`, the error of `y` is `0.01b` as the ideal result of `1.1b * 1.1b` is `10.01b` and it is truncated to `10.0b` because of the precision of `y`.
2) **Long-tail of errors**: Estimating errors based only on few representative data may overlook the long-tail of errors. Khudia *et al.* [5] applied loop perforation, another efficient but error-prone computing method, to calculate the brightness of images. They report that the average error was 5% while the maximum was 23%, and only 2 out of 800 images caused 20+% error (Figure 3 in [5]).
3) **Statistical aspects of errors**: Determining precision only from the average, maximum value, and minimum value of errors cannot capture statistical aspects of them. This is because the distributions of errors can be far from a normal distribution, and even worse, look very different from precision to precision. For example, the right-hand side of Figure 9 shows distributions of errors of a program `y = sin(x)` in two different precision. They are far from a normal distribution and from each other. In the top-right case, the probability that the error is contained within 1σ from the average is 0.600, which is largely different from the same probability for a normal distribution (0.683).
4) **Large configuration space**: Arbitrary precision can be specified for fixed-point numbers in designing circuits and FPGA applications. This large configuration space requires an efficient method to estimate the error for each precision, which is a different story from CPUs and GPUs that only support a handful of precision options.

B. Existing Methods and Drawbacks

We describe three existing methods to estimate errors of programs using fixed-point arithmetic and point out how they do not solve the challenges.

1) **Sampling-based**: This method selects a few representative input data and determines the precision based on the errors incurred for the selected data. Although it is significantly faster than the other two methods, it cannot consider the long-tail of errors. Li *et al.* [6] models errors of variables with their average and standard deviation,

which restricts their model to only estimate the standard deviation of applications.

2) **Range-based**: This method calculates the maximum possible error for given precision by mathematical theories such as interval arithmetic [7] and affine arithmetic [8]. The fundamental problem of these theories is that they are too conservative; they guarantee that the actual errors never exceed the estimated maximum, but they say nothing about how close actual errors could be to the maximum error in realistic cases.

3) **Brute-force**: This method feeds all available input data (we refer to it as the *dataset*) to the target program and actually calculates incurred errors to determine precision. The problem of this method is twofold. First, executing the target program for many precision options for every data in the dataset is time-consuming. For example, Minerva [9] repeatedly trains a DNN with various precision to determine the best one, which multiplies the amount of training time by the number of possible precision options. Second, it can only estimate errors for input data that is available. It means that considering extreme-case scenarios for determining precision requires preparing (generating) such data so that they can be fed into the target program.

C. Precision of Fractional Part

This paper particularly focuses on determining the precision of the fractional parts of fixed-point numbers. In this context, we merely say the *precision* of a fixed-point number to refer to the number of bits assigned to its fractional part hereafter. In addition, we assume that the most significant bit of an integer part represents the sign, and a real number r is rounded into the largest fixed-point number r_f such that $r \geq r_f$.

The fractional parts are the main concern in fixed-point arithmetic. Suppose we assign A and B bits to the integer and fractional parts of a fixed-point number X, respectively. A decides the maximum and minimum values that X can represent, i.e., X satisfies $-2^{A-1} - \alpha \leq X \leq 2^{A-1} - 1 + \alpha$ (lemma A). Here, α is the value of the fractional part that is less than 1. B decides how much X can be drifted from its real value X^*, i.e., $|X^* - X| \leq 2^{-B}$ (lemma B). The difficulties of determining A and B are quite different because lemma A holds for any variable in a fixed-point program while lemma B holds only for the input variables due to error propagation. An example of lemma B not always holding is given in Section III-B.

III. Error Distribution Estimation

We propose a method that estimates the *distribution* of errors to mitigate the drawbacks of the above-mentioned methods. First, we deal with values that the program manipulates with their distributions. This enables capturing the statistical aspects of them without considering all the actual values. Second, we model the error propagation through the code in a form that can be applied to distributions of values.

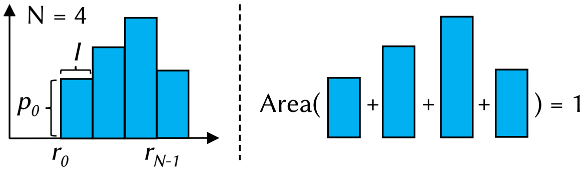

Fig. 2. Visual illustration of a distribution.

A. Distribution of Values

A distribution is a step-wise *probability density function (PDF)* of a random variable r. It is defined by N tuples $\{t_0, ..., t_{N-1}\}$ where $t_i = \{r_i, p_i\}$ and the length of the steps l (i.e., $l = r_1 - r_0 = r_2 - r_1 = ...$). We define $r_0 = r_m$ and $l = (r_M - r_m)/N$, where r_m and r_M are the minimum and maximum possible values of r, respectively. This results in $r_{N-1} = r_0 + (N-1) \times l = r_M - l$. p_i is defined so that the probability that r falls into $[r_i, r_i + l)$ is $p_i \times l$. This makes the integration of the distribution within $[r_0, r_{N-1} + l)$ be 1. Figure 2 illustrates a distribution and its parameters.

A distribution of a random variable can be built by creating and normalizing its histogram. To illustrate this, let us suppose all the possible values of a random variable r are $0.01, 0.02, 0.03, 0.06$ and we build a distribution with $N = 2$, which means that $l = 0.025$. Because there are three possible values of r in $[0.01, 0.035)$ and one value in $[0.035, 0.06)$, the distribution of x can be built as $\{\{0.01, 30\}, \{0.035, 10\}\}$.

An advantage of dealing with values by their distributions is that operations among them can be executed by constant time with regard to the number of actual possible values in the distributions. Suppose that x and y are independent real-valued random variables and let $z = x + y$. X, Y, and Z denote the PDFs of x, y, and z, respectively. We will write $Z = X \oplus Y$ in association with $z = x + y$. It is well-known that $Z(z)$ is given by

$$Z(z) = \int X(z-y)Y(y)dy. \tag{1}$$

Because a distribution in our definition is a step-wise PDF, the same applies to the distributions of values in a program and the integration can be calculated with $O(min(N_X, N_Y))$ time where N_X and N_Y are the numbers of tuples of X and Y, respectively. We introduce the $\otimes$ operator in the same way, and $Z = X \otimes Y$ can be calculated by

$$Z(z) = \int \frac{1}{|y|} X(\frac{z}{y}) Y(y) dy, \tag{2}$$

which again takes constant time with regard to the number of actual possible values.

B. Error Propagation Modeling

Error propagation is modeled in two components: *error amplification* and *error injection*. The former models how the error of a variable affects the error of its successor variable through an operator (e.g., +, `sin`), and the latter models how new error is introduced by assignments. In this section, we first

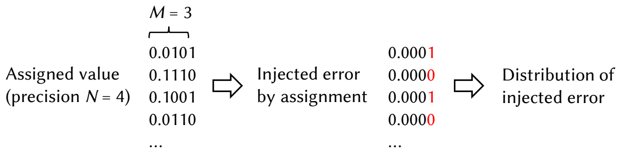
M = 3
Assigned value (precision N = 4)
0.0101
0.1110
0.1001
0.0110
...
Injected error by assignment
0.0001
0.0000
0.0001
0.0000
...
Distribution of injected error

Fig. 3. Modeling injection error when $N = 4$, $M = 3$.

look at an example that explains the two error components, and then show the details of the modeling.

Example: Let us consider variables `x` and `y` whose precision is 1 and a simple program `y = 1.1b * x`. Under the rounding method we assume (rounding to the smaller side), the error of an input variable is always smaller than 2^{-N} where N is the precision. For example, if the real value of `x` is `0.111b`, the error of `x` itself is `0.011b` which is smaller than 2^{-1}. However, the result of `1.1b * x` is `0.11b` and the error against the ideal value (`1.1b * 0.111b = 1.0101b`) is `0.1001b` that exceeds 2^{-1}, i.e., the error is amplified by the `*` operator. Even worse, the result `0.11b` is rounded to `0.1b` due to the assignment to `y`, i.e., more error is injected by an assignment. This yields the final error of `0.1101b` which is way beyond the error of `x`.

Error Amplification is modeled by derivatives of expressions. Let x_i ($i = 0, ..., n$) be the input variables and y be the output variable of an operator: $y = op(x_0, ..., x_n)$. Then, the error of y, denoted as $e(y)$, is approximated as

$$e(y) \approx \sum_i \left(e(x_i) \times \frac{\partial y}{\partial x_i} \right) \tag{3}$$

where $e(x_i)$ is the error of x_i. This equation is based on the concept of *total derivative*; If x_i changes slightly by $e(x_i)$, then y changes by $e(x_i) \times \frac{\partial y}{\partial x_i}$, as long as $e(x_i)$ is small.

Based on equation (3), we show how error amplification by operators $+$ and $\times$ are calculated. For $y_0 = x_0 + x_1$, the error of y_0 (denoted as $e(y_0)$) is:

$$\frac{\partial y_0}{\partial x_0} = 1, \quad \frac{\partial y_0}{\partial x_1} = 1 \tag{4}$$

$$\therefore e(y_0) = e(x_0) + e(x_1) \tag{5}$$

For $y_1 = x_0 \times x_1$, the error of y_1 (denoted as $e(y_1)$) is:

$$\frac{\partial y_1}{\partial x_0} = x_1, \quad \frac{\partial y_1}{\partial x_1} = x_0 \tag{6}$$

$$\therefore e(y_1) = x_1 \times e(x_0) + x_0 \times e(x_1) \tag{7}$$

The intuition behind equation (7) is that $e(x_0)$ is amplified by the value of x_1 and the same applies to $e(x_1)$ and x_0.

Equation (3) applies to other differentiable operators as well. For example when $y = sin(x)$, $e(y)$ can be calculated in the same way as for + and $\star$ as follows.

$$\frac{\partial y}{\partial x} = cos(x) \tag{8}$$

$$\therefore e(y) = e(x) \times cos(x) \tag{9}$$

Error Injection is modeled with the number of bits dropped by an assignment. We denote the precision of variable `x` as `pr(x)`. When assigning a value that has N bits of precision to `x` and `pr(x)` $= M$ satisfies $N > M$, the number of bits dropped is $N - M$ for the rounding method assumed. We statistically model the injected error by approximating that the lower $N - M$ bits of an assigned value are uniformly random. For the easiest case, if $N - M$ is 1, the injected error is 0 in half of the cases and 2^{-N} in the other half. These error values with their possibilities are used to build the distribution of injected errors. Figure 3 illustrates this with $N = 4$ and $M = 3$.

The precision N of an assigned value depends on how the value is calculated: $N = max(\mathrm{pr(a)}, \mathrm{pr(b)})$ for `a + b` and $N = \mathrm{pr(a)} + \mathrm{pr(b)}$ for `a * b`. For example, the value of `a * b` can have 1s down to the second bit from the decimal point when `pr(a) = pr(b) = 1` (e.g., if `a = b = 0.1b`, then `a * b = 0.01b`). The precision of a result of an irrational operator (e.g., `sin`) is assumed to be a large value such as 64.

Error propagation of a whole program is calculated by applying the aforementioned two models recursively from output variables until we reach input variables. The errors of input variables are calculated by quantizing input data with the specified precision. The use of real input data enables considering the data-dependency and the long-tail of the errors. It is also possible to intentionally increase the errors of input variables (for example, by multiplying the quantization errors by a small constant) to emulate extreme-case scenarios.

C. Estimating Error Distribution

To estimate the error distribution of a target application, we recursively apply the error propagation model to distributions of values that the application manipulates. Because we apply the model to *distributions* of values, the output of the model (i.e., e) is also the *distribution* of errors.

This application of the model requires (1) replacing the $+$ and $\times$ operators in the model with the $\oplus$ and $\otimes$ operators, (2) calculating the distribution of $cos(x)$ given the distribution of x, and (3) calculating the distributions of the input variables of the application and their errors. Requirement (1) is done by following the equation for $Z(z)$ in Section III-A. Requirement (2) is equivalent to a well-known variable conversion of random variables. Requirement (3) can be done from the actual values in the dataset and their errors for specific precision.

Estimating the error distribution for given precision can be done with constant time w.r.t the dataset size once the distributions of the input data and their errors are built. This is because each of the $\oplus$ and $\otimes$ operators in the model takes constant time as shown in Section III-A. The effect of the non-constant part, i.e., building the distributions of the input data and their quantization errors, is evaluated in Section V.

IV. Estimation System Implementation for HLS

As a concrete example of applying our method, we implement an error estimation system for programs written in C++ language, which is the de-facto standard for High-Level

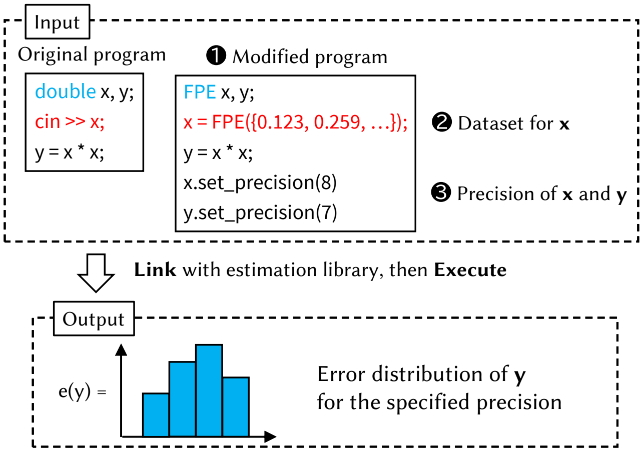

Fig. 4. System overview.

Synthesis (HLS). Note that this example does *not* narrow the applicability of our method down to C++ only. For example, creating an error propagation graph (Section IV-C) can also be done for Register-Transfer Level languages (e.g., Verilog) by tracking information flow using existing work [10].

Fig. 4 illustrates the overview of the estimation system based on our method. In this system, a user provides:

❶ A modified user program for which the error distribution is estimated. We explain how it is modified later.
❷ The dataset to feed into the input variable(s).
❸ The precision of variable(s) for which fixed-point numbers are used.

The dataset contains input data for which the error distribution is estimated. The modified program is linked with our estimation library and executed as a normal executable. This program then outputs the error distribution of the specified output variable(s) for the given dataset and precision.

A. The FPE class

The `FPE` class abstracts all the required information to estimate errors so that the modification to the user program can be as small as possible. The user replaces the type of a variable to `FPE` when that variable is to be mapped to a fixed-point number. An `FPE` class instance contains:

1) the operator that is used to calculate it,
2) the operands that it depends on through the operator,
3) the distribution of the values (not the errors) of the variable that it corresponds to, and
4) the pointer to a memory space to store its precision.

Suppose the user decides to use fixed-point numbers for variables `x`, `y`, and `z` in a program `z = x + y`. In this case, the `FPE` instance for `z` records (1) that the operator is $+$, (2) that the operands are `x` and `y`, (3) the distribution of the values of `z`, and (4) a pointer to a single `int` to store the precision. The distribution of the values is calculated from the distributions of the values of `x` and `y` by the $\oplus$ operator. We use a pointer for the precision store so that the pointed memory space can be shared by multiple instances.

```
void application() {
  double x[10] = {/* input data */}, y = 0.0;

  for(int i = 0; i<10; i++)
    y += x[i];
}
```

Fig. 5. Usage of our estimation system: user program *before* modification

```
void application() {
  FPE x[10], y(0.0); // double => FPE

  // Dataset: retrieved by any legal code
  std::vector<double> x_data[10];

  // Initialize input variables by dataset
  for(int i = 0; i<10; i++)
    x[i] = FPE(x_data[i]);

  // The main part is unmodified
  for(int i = 0; i<10; i++)
    y += x[i];

  y.set_precision_all(8);  // Set precision
  y.set_precision(7);      // Set precision
  estimate(y);  // Estimate error distribution
}
```

Fig. 6. Usage of our estimation system: user program *after* modification

For example, an expression `y = y + x` has two 'y's that have different values but the same precision unless the circuit is specially crafted to use different precision for them.

B. Modification of User Program

The user program is modified in three ways. Fig. 5 shows a simple program that we use to explain how it is modified, and Fig. 6 is the modified version of it.

First, the types of variables for which the user decides to use fixed-point numbers are replaced with `FPE`. For example, the user can decide to replace the types of `x[0]`, ..., `x[9]`, and `y` because they are the input and output variables. In addition to variables that the user directly decides to replace, any variable that depends on a replaced variable must also be replaced.

Second, the dataset for each input variable must be provided. The program must pass a `std::vector` that contains the dataset for an input variable to its constructor provided by the `FPE` class. The dataset can be retrieved by any legal C++ code (e.g., reading from a file or reading from the standard input).

Third, the precision of variables replaced with `FPE` must be defined. This can be done either in two methods: (a) by setting the precision of one variable to a specific value, and (b) by setting the precision of one variable and all variables that it depends on to a specific value. Method (b) is supported so that the user does not need to specify every single variable one by one. For example in Fig. 6, `y.set_precision_all(8)` sets the precision of `y` and all variables that it depends on (e.g., `x[0]`) to be 8. The precision of `y` (but not other variables) is then overwritten to 7 by `y.set_precision(7)`.

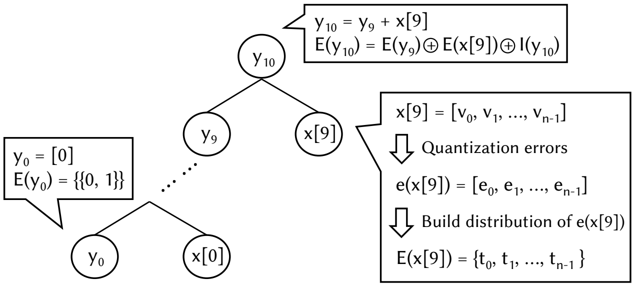
$y_{10} = y_9 + x[9]$
$E(y_{10}) = E(y_9) \oplus E(x[9]) \oplus I(y_{10})$
y_{10}
y_9
$x[9]$
$x[9] = [v_0, v_1, ..., v_{n-1}]$
Quantization errors
$e(x[9]) = [e_0, e_1, ..., e_{n-1}]$
Build distribution of e(x[9])
$E(x[9]) = \{t_0, t_1, ..., t_{n-1}\}$
$y_0 = [0]$
$E(y_0) = \{\{0, 1\}\}$
y_0
$x[0]$

Fig. 7. Error propagation graph for the sample application and how error distribution is estimated using the graph.

C. Error Propagation Graph

To recursively apply the error propagation model through variables, our estimation system generates an *error propagation graph* of the modified user program. It represents how errors propagate from input to output. The nodes of a graph consist of all variables whose types are class FPE, and edges connect the output variable of each expression and its operands. For example, Fig. 7 shows the error propagation graph for the program in Fig. 6. Note that the variable y has multiple nodes corresponding to it (e.g., y_{10}, y_9, ...) because it is reused in the main computation loop.

Our system dynamically creates an error propagation graph when a modified user program is executed. To do this, the modified program #includes the C++ header file that implements the FPE class and is compiled as a normal C++ program. The operators on FPE instances (e.g., +) are overloaded so that they investigate the error propagation graphs of the operands and set a new error propagation graph to the return value (which is another FPE instance). The complete error propagation graph is acquired from an output variable at the end of a run of the modified program. Abstraction by the class FPE and operator overloading make it easy for users to modify their programs because the main computation part does not need to be modified as shown in Fig. 6.

We explain how the error distribution of an output variable is estimated using Fig. 7. Here, E(v) denotes the distribution of errors of a variable v. The error distribution of the output variable y_{10} is estimated as follows.

1) The amplification part and the injection part of $E(y_{10})$ are estimated separately and then combined. The amplification part is calculated based on equation (5) because y_{10} is the sum of y_9 and x[9]. Thus, the amplification part is $E(y_9) \oplus E(x[9])$. In addition, the injection part is further calculated using the precision of y_{10}, y_9, and x[9] and we denote it as $I(y_{10})$. Given the two parts, we get $E(y_{10}) = E(y_9) \oplus E(x[9]) \oplus I(y_{10})$.
2) The same procedure is recursively applied until every term can be calculated from the input variables. For example, E(x[9]) is calculated from the actual values of x[9] that are given to our system as the dataset.

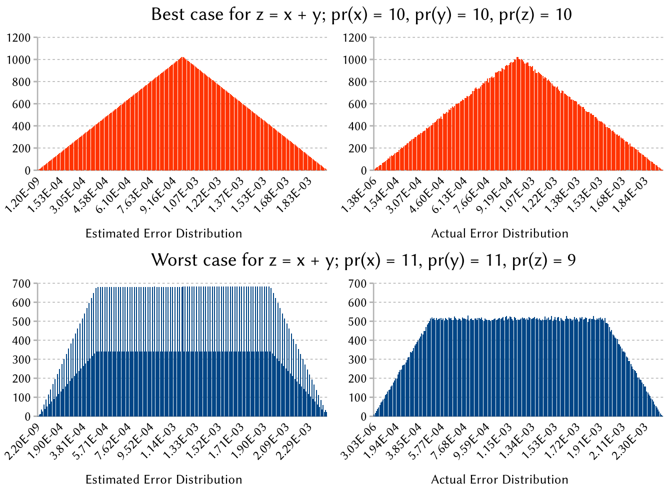

Fig. 8. Estimated/actual error dists. for best/worst cases in z = x + y.

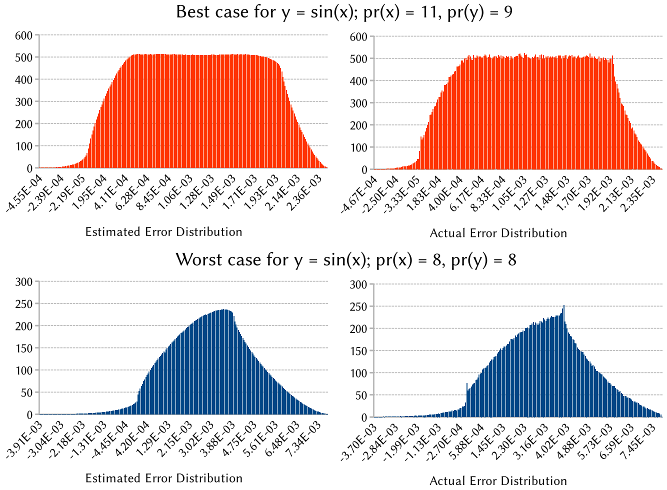

Fig. 9. Estimated/actual error dists. for best/worst cases in y = sin(x).

V. Experimental Results

A. Estimation Accuracy for Micro Benches

In this section, we evaluate how well our method can estimate the error distribution for single operators using three programs z = x + y, z = x * y, and y = sin(x). The programs are implemented in C++. The input variables obey normal distributions whose mean is 0 and the standard deviation is 1, and we swipe the precision between 8 and 11.

The actual error distributions (baselines) are acquired by comparing the results of programs written with floating-point and fixed-point variables. For + and *, we change the variable types from double to ap_fixed [11] to make a fixed-point version of each program. For sin, an ap_fixed is up-casted to a double to be fed to a standard sin function, and the return the value is re-casted to ap_fixed.

Hellinger Distance (we refer to it as the *distance*) is used to quantitatively compare the estimated and actual distributions. It is a well-known metric (e.g., Section 3.2 in [12]) to measure the similarity of probability density functions (PDFs). The distance H between two PDFs f and g is given by

$$H(f,g)=\frac{1}{\sqrt{2}}\sqrt{\int\left(\sqrt{f(x)}-\sqrt{g(x)}\right)^2dx} \qquad (10)$$

which has three characteristics: (1) $0 \leq H \leq 1$ for any PDFs, (2) $H = 0$ when f and g are identical, and (3) H increases as f and g become further apart.

Fig. 8 and Fig. 9 show a comparison of best- and worst-case estimations for `x + y` and `sin(x)`, respectively. The best case means that the distance between the actual and estimated error distributions is the smallest among all swiped precision, and the worst case means that it is the largest. We omit this visual representation for `x * y` due to the space limit. In the best cases, the estimated error distributions are almost identical to the actual ones except for some small dithering. In the worst case for `x + y`, the two distributions do not match as closely.

Fig. 10 shows the distance for all tested precision in each program. The horizontal axes are the output precision and each line corresponds to the input precision. For + and $\star$, the label i_j means that the precision of `x` (denoted as `pr(x)`) is i and the precision of `y` (denoted as `pr(y)`) is j.

B. Statistical Aspects of Errors

In this section, we show the importance of considering statistical aspects of errors using our method. To this end, we calculate the average (denoted as μ), the standard deviation (denoted as σ), and the probability P that the error falls into a certain range $[\mu - \sigma,\ \mu + \sigma]$. These values can be easily calculated from a distribution because a distribution in this paper is a probabilistic density function (PDF).

TABLE I
STATISTICAL ANALYSIS FOR ERROR DIST. OF Y = SIN(X)

	μ	σ	P
pr(x) = 11, pr(y) = 9, estimated	0.00112	0.000585	**0.600**
pr(x) = 11, pr(y) = 9, actual	0.00112	0.000589	0.602
pr(x) = 8, pr(y) = 8, estimated	0.00303	0.00163	0.667
pr(x) = 8, pr(y) = 8, actual	0.00311	0.00173	0.680
normal distribution	-	-	**0.683**

Table I shows μ, σ, and P calculated for the estimated and actual error distributions for variable `y` in `y = sin(x)`. The table also shows P for a normal distribution. The precision of `x` and `y` are the same as in Fig. 9. We draw two takeaways from the table. First, μ, σ, and P calculated from the estimated error distribution are close to the ones calculated from the actual error distribution even in the worst case (`pr(x) = pr(y) =` 8). Second, P is largely different from a normal distribution when `pr(x) = 11` and `pr(y) = 9` (0.600 vs. 0.683). This result demonstrates the importance of considering statistical aspects of errors when determining precision.

C. Estimation Accuracy and Speed for Application Bench

In this section, we evaluate our method with a more realistic application in two aspects: (a) the execution time and (b) the distance between the estimated and actual error distributions. The baseline is to repeatedly execute the application using fixed-point arithmetic for all data in the dataset. The execution time is measured on an Intel Xeon Platinum 8276 with 192 GB of DD4-3200 memory running Debian GNU/Linux 11. We fix the number of tuples N of a distribution to 256.

We use Gaussian Filter with a 3x3 filter implemented in C++ for this purpose. For the dataset, we collect 3K images in the order of their filenames from each of the 10 categories of CIFAR-10 dataset [13] (30K images in total). The images are converted to gray-scale and each pixel value is normalized to $[0, 1]$. We estimate the error distribution of the value of the center pixel of the output images.

Fig. 11 shows the comparison of the execution times of our method and the brute-force method. We omit the time it takes to load images from the storage to memory. The horizontal axis shows the number of images, and the vertical axis shows the average execution time across all the tested precision. We swipe the precision of the pixels of the input images, the pixels of the output image, and the intermediate variables between 8 bits to 11 bits. The execution time of the brute-force method is proportional to the number of images because the method iterates over all the images. In contrast, the execution time of our method increases less rapidly, giving a more significant speedup when the size of the dataset is larger. Our method does not run in exactly constant executing time because it still iterates over all images to build distributions of the input variables. However, this effect becomes negligible when the main computation of the target application takes more time.

The max, min, average, and standard deviation of the distance between the actual and estimated error distributions were 0.362, 0.096, 0.128, and 0.046, respectively. The fact that the average is only 0.128 suggests that our method works well even when the target program consists of many operators.

D. Robustness Analysis

An advantage of our method compared to the brute-force method besides speed is the ability to conduct *robustness analyses*. A robustness analysis consists of three steps:

1) The user chooses specific precision by our method based on their error requirements.
2) The user *tweaks* input distributions of our method with extreme-case scenarios in mind.
3) The user confirms that their error requirements are still met even in these extreme-case scenarios.

Tweaking a distribution can be done either for the distributions of input variables, the distributions of their errors, or both. For example, the user can double the quantization errors of input variables to tweak the distributions of them. This reflects an extreme-case scenario where the errors of the input data are much higher than expected. Conducting the same analysis by the brute-force method is quite troublesome because the user must generate such input data. On the other hand, our method enables tweaking at the distribution level.

As an example of robustness analysis, we apply it to the gaussian filter application. We assume that the user has chosen 8 bits, 10 bits, and 11 bits for the precision of the input pixels,

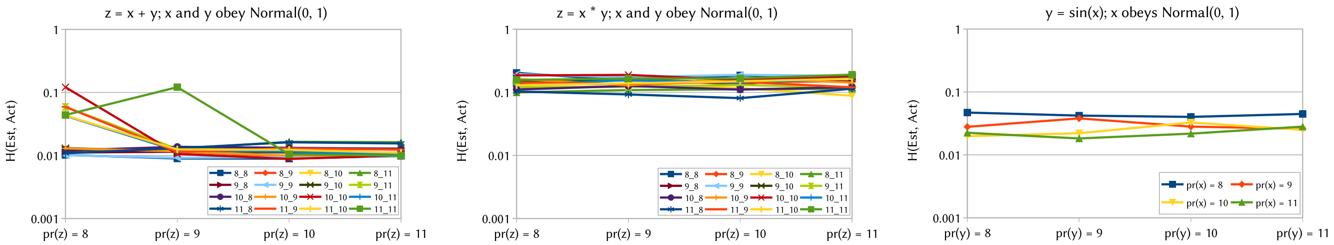

Fig. 10. Distance of actual and estimated error distributions for z = x + y, z = x * y, and y = sin(x). The input variables obey Normal(0, 1).

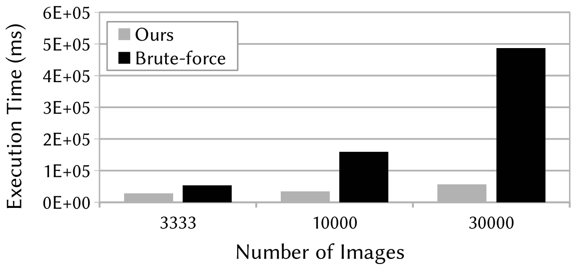

Fig. 11. Execution time of our method and the brute-force method.

intermediate variables, and output pixels, respectively, and has decided to tweak the error distribution of the input pixels by multiplying the quantization errors by 1.1. Figure 12 shows the estimated error distribution without tweaking (top) and with tweaking (bottom). The vertical lines indicate the tops of the distribution shapes. The figure shows that the peak of the distribution is shifted to the right because of the increased quantization errors, while the overall shapes (e.g., maximum error) of the distributions remain the same. The user can then compare their error requirements to these results to see if the precision chosen is robust to this extreme-case scenario.

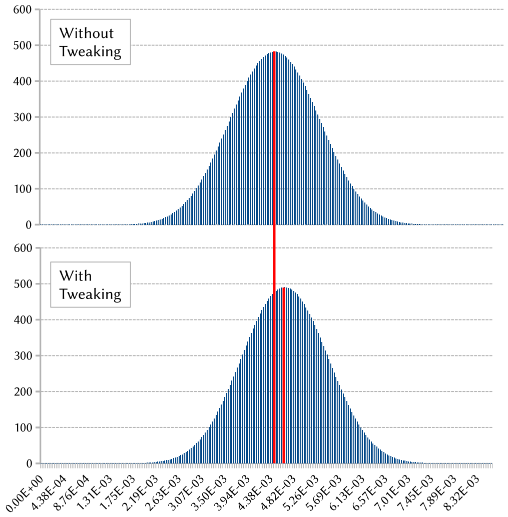

Fig. 12. Error distributions of Gaussian Filter estimated for a particular precision. Top: input error distributions are not tweaked. Bottom: same precision, but the input error distributions are tweaked.

VI. Related Work

Earlier studies on errors incurred by fixed-point arithmetic only reveal the largest possible error, but not the statistical aspects of errors. Simić *et al.* [8] apply model checking to validate that the largest error incurred to a variable does not exceed a threshold by rewriting an input program to a form that model checking is applicable to errors. Fang *et al.* [14] use affine arithmetic, an improved methodology of interval arithmetic, to compute the range of errors that a variable can contain from the range of the input and the precision.

PreAxC [15] is the only work we know that predicts the error distribution of approximated programs. They take a program as a DFG (Data Flow Graph) and feed it to graph neural networks. The major difference from our work is that they focus on circuit-level programs (i.e., a DFG node is a functional unit in their method), while we handle more generic programs such as those written in C++. This restricts their DFG nodes to be additions or multiplications only.

Little is studied on how to efficiently determine the precision of fixed-point numbers. As a result, applications that leverage fixed-point numbers merely use precision that satisfies their error requirements [1], [16] but may overlook further optimization opportunities. A few exceptions we are aware of include Minerva [9], which optimizes the precision of fixed-point numbers for DNNs beyond convenient options (e.g., 8 bits). However, they do it by repeatedly training a DNN for each precision configuration, which takes a long time.

Error analysis using derivatives has been done since the old days in the context of rounding a real number to a floating-point number for scientific computing [17], [18]. As far as we know, we are the first to combine it with distribution of values to determine precision of fixed-point arithmetic.

VII. Conclusion

To ease the burden of determining precision of fixed-point arithmetic, we proposed a new technique that estimates the error distributions of the application output when fixed-point arithmetic is used. Our method allows not only determining precision based on real data, but also analyzing the robustness of chosen precision by considering extreme-case scenarios. By implementing an example estimation system based on our method, we showed that the error distributions can be properly estimated for some primitive operators (+, ⋆, and sin) as well as for a real application.

References

[1] S. Fox, J. Faraone, D. Boland, K. Vissers, and P. H. Leong, "Training deep neural networks in low-precision with high accuracy using FPGAs," in *International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 1–9.

[2] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning (ICML)*, 2015, pp. 1–10.

[3] S. Shuvo, E. Hossain, and Z. R. Khan, "Fixed point implementation of grid tied inverter in digital signal processing controller," *IEEE Access*, vol. 8, pp. 89 215–89 227, 2020.

[4] F. Cabello, J. Leon, Y. Iano, and R. Arthur, "Implementation of a fixed-point 2d gaussian filter for image processing based on FPGA," in *Conference Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, 2015, pp. 28–33.

[5] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Rumba: An online quality management system for approximate computing," in *International Symposium on Computer Architecture (ISCA)*, 2015, pp. 554–566.

[6] C. Li, W. Luo, S. S. Sapatnekar, and J. Hu, "Joint precision optimization and high level synthesis for approximate computing," in *Design Automation Conference (DAC)*, 2015, pp. 1–6.

[7] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics (SIAM), 2009.

[8] S. Simić, A. Bemporad, O. Inverso, and M. Tribastone, "Tight error analysis in fixed-point arithmetic," *Formal Aspects of Computing*, vol. 34, no. 1, sep 2022.

[9] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *International Symposium on Computer Architecture (ISCA)*, 2016, p. 267–278.

[10] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *Design, Automation & Test in Europe Conference (DATE)*, 2017, pp. 1691–1696.

[11] Xilinx, "Overview of arbitrary precision fixed-point data types," https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Overview-of-Arbitrary-Precision-Fixed-Point-Data-Types, 2022.

[12] L. Cam and G. Yang, *Asymptotics in Statistics*. Springer New York, 1990.

[13] A. Krizhevsky, "The CIFAR10 dataset," https://www.cs.toronto.edu/~kriz/cifar.html, 2009.

[14] C. F. Fang, R. A. Rutenbar, and T. Chen, "Fast, accurate static analysis for fixed-point finite-precision effects in DSP designs," in *International Conference on Computer-Aided Design (ICCAD)*, 2003, pp. 275 – 282.

[15] L. Sathidevi, A. Sharma, N. Wu, X. Jiao, and C. Hao, "PreAxC: Error distribution prediction for approximate computing quality control using graph neural networks," in *International Symposium on Quality Electronic Design (ISQED)*, 2023, pp. 1–7.

[16] J. Lu, L. Zhao, K. Chen, P. Deng, B. Li, S. Liu, and Q. An, "Real-time FPGA-based digital signal processing and correction for a small animal PET," *IEEE Transactions on Nuclear Science*, vol. 66, no. 7, pp. 1287–1295, 2019.

[17] F. L. Bauer, "Computational graphs and rounding error," *SIAM Journal on Numerical Analysis*, vol. 11, no. 1, pp. 87–96, 1974.

[18] M. P. W. Mutrie, R. H. Bartels, and B. W. Char, "An approach for floating-point error analysis using computer algebra," in *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, 1992, p. 284–293.