Toward Efficient Static Analysis of Finite-Precision Effects in DSP Applications via Affine Arithmetic Modeling

Claire Fang Fang, Rob A. Rutenbar, Markus Püschel, Tsuhan Chen
{ffang, rutenbar, pueschel, tsuhan}@ece.cmu.edu
Electrical and Computer Engineering
Carnegie Mellon University

ABSTRACT

We introduce a static error analysis technique, based on smart interval methods from *affine arithmetic*, to help designers translate DSP codes from full-precision floating-point to smaller finite-precision formats. The technique gives results for numerical error estimation comparable to detailed simulation, but achieves speedups of three orders of magnitude by avoiding actual bit-level simulation. We show results for experiments mapping common DSP transform algorithms to implementations using small custom floating point formats.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Signal Processing Systems; J.6 [**Computer-Aided Engineering**]: Computer-Aided Design

General Terms

Algorithms, Design, Performance

Keywords

Static error analysis, affine arithmetic, probabilistic error bound, embedded hardware, custom floating-point

1. INTRODUCTION

Modern digital signal processing DSP applications are typically prototyped using floating-point arithmetic, which offers both large dynamic range and high precision for each numerical computation. However, for hardware implementations, the final form rarely uses a full-precision floating-point unit, given issues of silicon area, power, and speed. This creates the common—and still awkward—problem of transforming the application from its initial, for all practical purposes "infinite" precision form, into some final, finite-precision hardware format.

These final finite-precision formats can be either custom fixed-point or reduced-precision (so-called "lightweight" [3]) custom floating-point. The two-part problem for any complex DSP task is how to choose the smallest bit-level number formats, and then how to validate that the format choices maintain the necessary level of numerical precision. Several techniques have been proposed [2, 3]. Roughly speaking, these all share three common characteristics: (a) they are based on detailed simulation to capture the necessary numerical ranges and the maximum error; (b) they first strive to determine the dynamic range of each operand, to avoid catastrophic overflows; and (c) they next strive to choose the right precision to assure acceptable quality in the output.

The essential problem in all these techniques is the need to rely on detailed simulations to find the optimal range and precision of each operand. If we choose a sequence of input patterns that are too few or incorrectly distributed, we may fail to find all the extreme values that real-life use will encounter. If we employ instead a more rigorous detailed simulation strategy, the format optimization process becomes unduly expensive.

A more attractive solution is some form of *static analysis*, which guarantees that we will find the extreme values and the maximum error of each operand, but does not require us to consider a potentially unbounded set of input patterns. We use the term "static" here in the same sense as that from *static timing analysis* for gate-level logic netlists. A single evaluation pass on the network, which computes more than just the delay through each gate, derives useful bounds on each path in the network. These delays are pessimistic—they are upper bounds, but tight enough for timing signoff. For finite-precision format optimization, we seek exactly the same sort of attack: a more pattern-independent evaluation that can create usefully tight bounds on the range/error of each operand in the application.

The obvious approach for such a static analysis draws on techniques from *interval arithmetic* [8], which replaces each scalar value with a bounded interval on the real number line. An algebra of intervals lets us "simulate" the code by performing each operation on the intervals. Unfortunately, conventional interval arithmetic suffers from the problem of range explosion: as more computations are evaluated, the intervals tend to grow without bound. The problem is that correlations among variables are not captured.

A recent solution to this problem is a more sophisticated interval model called *affine arithmetic*, introduced in [1]. The approach explicitly captures some of the correlations among operands, and dramatically reduces the level of pessimism in the final intervals. It has been successfully used in analog circuit sizing [6]. We apply the idea to the novel prob-

DAC 2003, June 2–6, 2003, Anaheim, California, USA.

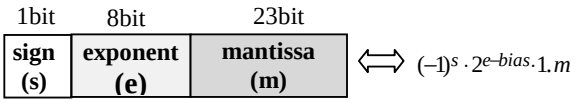


Figure 1: Standard single precision floating-point format

lem of finite-precision error estimation for reduced-precision custom floating point formats: we seek to efficiently estimate, for each operand, the maximum *difference* (the error) between a finite-precision implementation and a reference "ideal" version using standard floating point. We show how to build an explicit numerical model of the error and estimate it quickly using affine arithmetic. We derive a model that estimates a hard upper bound for the error. This bound, however, becomes less accurate as more computations are evaluated. We solve this problem by using probabilistic methods to compute a 'soft" bound that is satisfied with a user-specified probability.

The remainder of the paper is organized as follows. Section 2 gives relevant background on floating point error and affine arithmetic. Section 3 formulates our error analysis model and offers a refinement based on probabilistic bounds. Section 4 present some results for several DSP benchmarks showing the viability of the approach. Application examples on two practical design problems are given. Finally, Section 5 offers concluding remarks.

2. BACKGROUND

2.1 Floating-Point Error

In this section we give a brief overview of the floating-point format and its associated error model, followed by an overview of related work on error analysis, to distinguish our approach.

The floating-point format commonly used in general-purpose processors is the IEEE-standard double-precision or single-precision format, which consists of three fields: sign, exponent and mantissa. Figure 1 shows the single-precision floating-point format, represented by $2^{e-bias}1.m$, where $e \in [0, 256]$, $bias = 127$, $m \in [0, 1]$, and the leading '1' is implicitly specified. The precision of the floating-point format is determined by the mantissa bit-width, and usually 23 bit is sufficient for most applications. This is the standard mantissa bit-width for single-precision format.

However, in application specific floating-point units, more flexible custom formats can be adopted to reduce the hardware cost and the power consumption. In this case, the precision becomes a "tuning" parameter, which can be optimized to minimize the hardware cost while guaranteeing the algorithm's required numerical performance. One important property of floating-point arithmetic is that the rounding error depends not only on the mantissa bit-width, but also on the magnitude of the operands. A conventional error model is given by

$$\begin{aligned} x_f &= x + x \cdot 2^{-(t+1)} \cdot \varepsilon, \\ x_f \circ y_f &= (x_f \bullet y_f) + (x_f \bullet y_f) \cdot 2^{-(t+1)} \cdot \varepsilon, \end{aligned} \tag{1}$$

where x_f is the floating-point representation of a real number x, $\circ$ is the floating-point approximation of an arithmetic operation $\bullet$, $\varepsilon \in [-1, 1]$ is an error term, and t is the mantissa bit-width [11]. The error model (1) makes static error analysis impossible since the magnitude information of x or $x_f \bullet y_f$ is only available at run time.

To quantify the error of an application implemented in custom floating-point format, two approaches can be taken:

- *Simulation* estimates the error as the maximal difference between two implementations over a large number of runs: one using a custom floating-point format and the other using the IEEE-standard double-precision format, which is assumed to have virtually infinite precision [2, 3]. Although this approach is application independent and provides good accuracy, it may be prohibitive for fast design decision making because of the high computational effort.
- *Static analysis* estimates an error bound at compile time. Methods in this category are mostly application dependent [10, 5, 7, 12, 13]: a linear transfer function is required between the inputs and the outputs, and a closed form of the output error variance is developed based on the floating-point error model (1). One application independent approach was introduced in [4], which developed error bound models for common operators based on interval arithmetic (IA). However, it is very likely to lead to unacceptable overestimation since IA does not take into account correlations between intervals.

In this paper, we suggest an alternative approach to application independent static error analysis that combines high accuracy with fast evaluation time.

2.2 Affine Arithmetic

The modeling tool in this paper is *affine arithmetic*, which is an efficient and recent variant of range arithmetic. In this section, we begin with introducing its predecessor, interval arithmetic, and then emphasize the advantage of affine arithmetic.

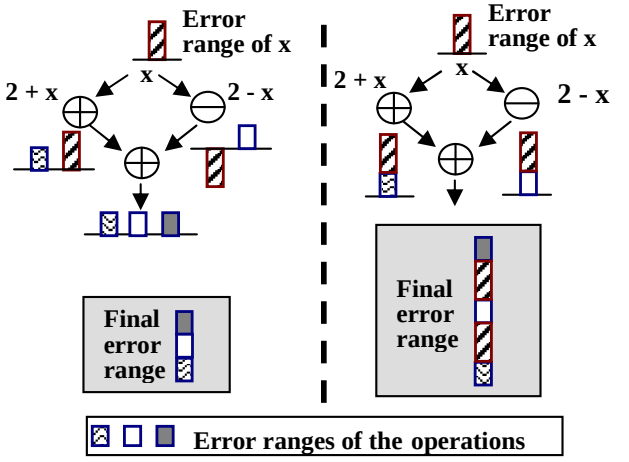
Interval arithmetic (IA), also known as *interval analysis*, was invented in the 1960s by Moore [8] to solve range problems. The uncertainty in a variable x is represented by the interval $\overline{x} = [\overline{x}.lo, \overline{x}.hi]$, meaning that the true value of x is known to satisfy $\overline{x}.lo \leqslant x \leqslant \overline{x}.hi$. For each operation $f : R^m \rightarrow R$, there is a corresponding range extension $\overline{f} : \overline{R}^m \rightarrow \overline{R}$. Taking addition as an example, the corresponding IA operation is obtained as

$$\overline{z} = \overline{x} + \overline{y} = \overline{x}.lo + \overline{y}.lo, \overline{x}.hi + \overline{y}.hi. \tag{2}$$

Analogous formulas can be derived for multiplication, division, square root, and other common mathematical functions [1]. A floating-point error model based on IA was introduced in [4]. The main problem of IA is overestimation, especially for correlated variables. To illustrate this problem, suppose that in (2) $\overline{x} = [-1, 1]$, $\overline{y} = [-1, 1]$, and that x and y have the relation $y = -x$. Using (2), $\overline{z} = [-2, 2]$, while $z = x + y = 0$. The effect of overestimation accumulates along the computation chain, and may result in an exponential range explosion.

Affine arithmetic (AA), or *affine analysis*, is a recent refinement in range arithmetic to alleviate the problem of overestimation in IA [1]. It has been used in areas such as computer graphics and analog circuit sizing [1, 6]. In contrast to IA, AA preserves correlations among intervals. In affine arithmetic, the uncertainty of a variable x is represented as a range in an *affine form* $\widehat{x}$ given by

$$\widehat{x} = x_0 + x_1\varepsilon_1 + x_1\varepsilon_1 + \cdots + x_n\varepsilon_n, \quad \text{with} -1 \leqslant \varepsilon_i \leqslant 1. \tag{3}$$



a) AA-based error range **b) IA-based error range**

Figure 2: Comparison of AA and IA

Each symbol ε_i stands for an independent component of the total uncertainty of the variable x; the corresponding coefficient x_i gives the magnitude of that component. For the affine operations $\widehat{x} \pm \widehat{y}$, $a \pm \widehat{x}$, and $a\widehat{x}$, the resulting affine forms are easily obtained using (3). For other operations (e.g., multiplication), the result, as a function $f(\varepsilon_1, \ldots, \varepsilon_n)$ of the ε_i's, is no longer affine. Thus, to obtain the affine form for the result, first a linear function $f^*(\varepsilon_1, \ldots, \varepsilon_n)$ is selected as an approximation of $f(\varepsilon_1, \ldots, \varepsilon_n)$, and a new noise term indicating the approximation error is estimated and added to the final affine form [1].

The key feature of AA is that one noise symbol may contribute to the uncertainties of two or more variables, indicating correlations among them. When these variables are combined, error terms may cancel out. This advantage is especially noticeable in computations that are highly correlated or of great computational depth. Returning to our previous simple example, suppose that x and y have affine forms $\widehat{x} = 0 + 1\varepsilon$ and $\widehat{y} = -\widehat{x} = 0 - 1\varepsilon$. In this case, the affine form of the sum $\widehat{z} = \widehat{x} + \widehat{y} = 0$ perfectly coincides with the actual range of the variable z.

3. ERROR ANALYSIS VIA AFFINE MODELING

Range arithmetic provides a tool for problems in which precise information is unavailable and an estimation of range offers a good approximation of the solution. We apply range arithmetic to floating-point error analysis. In this section, we first introduce the AA-based floating-point error model and use it to derive "hard" error bounds. Then we use probabilistic methods to refine the model to obtain "soft" probabilistic error bounds.

3.1 AA-based Floating-Point Error Model

As explained in the previous section, AA provides the opportunity for range cancellation. If we think of the floating-point error as a range, we can model the floating-point number representation and computations using AA. Figure 2 is an intuitive example illustrating the difference between AA and IA in the context of floating-point error propagation. The AA-based error ranges carry information about the error sources, which enables error cancellation, while the IA-based error ranges always accumulate, which inevitably leads to overestimation.

Next, we formally derive a floating-point error model based on AA. Our goal is to find a simple form, independent of the exact magnitude information, to represent floating-point numbers and computations. we develop the model in three steps.

AA model for floating-point numbers. According to (1), a floating-point number x_f can be written as

$$x_f = x + x \cdot 2^{-(t+1)} \cdot \varepsilon, \quad \text{with } \varepsilon \in [-1, 1]. \tag{4}$$

Note that the floating-point model (4) is in an affine form. The uncertainty term $x_f \cdot 2^{-(t+1)} \cdot \varepsilon$ is caused by the floating-point approximation, or rounding.

AA model for floating-point number ranges. To apply range arithmetic, in particular AA, to error analysis, we develop now a floating-point model for ranges. Suppose a variable x is in the range $\widehat{x} = [x_0 - x_1, x_0 + x_1]$. Then $\widehat{x}$ can be written as

$$\widehat{x} = x_0 + x_1\varepsilon_r,$$

and, using (4), its floating-point representation is

$$\widehat{x}_f = x_0 + x_1\varepsilon_r + (x_0 + x_1\varepsilon_r) \cdot 2^{-(t+1)} \cdot \varepsilon_x. \tag{5}$$

To reduce (5) to an affine form, we introduce the *bounding operator* B:

$$B(x_0 + \sum_{i=1}^{N} x_i\varepsilon_i) = \sum_{i=0}^{N} |x_i|,$$

which computes a hard upper bound of its argument. Then we apply B to (5) to obtain an upper bound of $\widehat{x}_f$ in affine form with associated error $E(\widehat{x}_f)$:

$$\widehat{x}_f \leqslant x_0 + x_1\varepsilon_r + (|x_0| + |x_1|) \cdot 2^{-(t+1)} \cdot \varepsilon_x, \tag{6}$$

$$E(\widehat{x}_f) = \widehat{x}_f - \widehat{x} \leqslant (|x_0| + |x_1|) \cdot 2^{-(t+1)} \cdot \varepsilon_x. \tag{7}$$

The sign '$\leqslant$' here means that the range on the left is included in the range on the right. In (7), the error is related to both the insufficient knowledge of the exact magnitude of x, and the floating-point rounding error.

AA models for floating-point range computations. Using (6), we now derive error models for the addition and the multiplication of floating-point ranges.

Floating-point addition : $\widehat{z}_f = \widehat{x}_f \oplus \widehat{y}_f$

$$\begin{aligned}
\widehat{z}_f &\leqslant (\widehat{x}_f + \widehat{y}_f) + B(\widehat{x}_f + \widehat{y}_f)2^{-(t+1)}\varepsilon_z \\
&\leqslant \widehat{x} + B(\widehat{x})2^{-(t+1)}\varepsilon_x + \widehat{y} + B(\widehat{y})2^{-(t+1)}\varepsilon_y \\
&\quad + B(\widehat{x}_f + \widehat{y}_f)2^{-(t+1)}\varepsilon_z \\
E(\widehat{z}_f) &= (\widehat{x}_f \oplus \widehat{y}_f) - (\widehat{x} + \widehat{y}) \\
&\leqslant B(\widehat{x})2^{-(t+1)}\varepsilon_x + B(\widehat{y})2^{-(t+1)}\varepsilon_y \\
&\quad + B(\widehat{x}_f + \widehat{y}_f)2^{-(t+1)}\varepsilon_z
\end{aligned} \tag{8}$$

Floating-point multiplication : $\widehat{z}_f = \widehat{x}_f \otimes \widehat{y}_f$

Benchmarks	# of adds	AA error bound	max error	ratio
WHT4	4	0.0029	0.0027	1.08
WHT64	64	0.1094	0.0334	3.27

Table 1: Estimation accuracy vs. algorithm complexity

$$\begin{aligned}\widehat{z}_f &\leqslant (\widehat{x}_f \cdot \widehat{y}_f) + B(\widehat{x}_f \cdot \widehat{y}_f) \cdot 2^{-(t+1)} \cdot \varepsilon_z \\ &\leqslant (\widehat{x} + B(\widehat{x})2^{-(t+1)}\varepsilon_x)(\widehat{y} + B(\widehat{y})2^{-(t+1)}\varepsilon_y) \\ &\quad + B(\widehat{x}_f \cdot \widehat{y}_f)2^{-(t+1)}\varepsilon_z \\ &\approx \widehat{x} \cdot \widehat{y} + \widehat{y}B(\widehat{x})2^{-(t+1)}\varepsilon_x + \widehat{x}B(\widehat{y})2^{-(t+1)}\varepsilon_y \\ &\quad + B(\widehat{x}_f \cdot \widehat{y}_f)2^{-(t+1)}\varepsilon_z \\ E(\widehat{z}_f) &= \widehat{x}_f \otimes \widehat{y}_f - (\widehat{x} \cdot \widehat{y}) \qquad (9) \\ &\leqslant B(\widehat{x})B(\widehat{y})2^{-(t+1)}(\varepsilon_x + \varepsilon_y) + B(\widehat{x}_f \cdot \widehat{y}_f)2^{-(t+1)}\varepsilon_z\end{aligned}$$

Note that we ignore the second order term $\varepsilon_x\varepsilon_y$ in the multiplication model. From (7)–(9), we see that the floating-point error of any range is in an affine form. We can generalize this form and rewrite it as $C_0 + \sum C_i\varepsilon_i$, where the ε_i's, called *error symbols*, are in the range [-1, 1]. The corresponding error bound is given by

$$error\ bound = B(C_0 + \sum_{i=1}^{N} C_i\varepsilon_i) = \sum_{i=0}^{N} |C_i|. \qquad (10)$$

The sharing of error symbols among two or more variables indicates their correlations and offers the opportunity for error cancellation. It is this numerical formulation of the rounding error—the difference between a specific finite precision format and the "ideal" real value—that is the basis for our static evaluation strategy. In section 4, we show the effectiveness of this method compared to the conventional floating-point error model based on IA.

3.2 Probabilistic Bounding

The AA-based error analysis estimates a reasonable bound for the floating-point error, but one drawback is that the estimation accuracy drops with the increasing computational complexity of the target floating-point program. This behavior is shown in Table 1 by comparing a Walsh-Hadamard transform (WHT) of size 4 and of size 64, with 8 and 384 operations, respectively. To obtain the error bounds, we assume a 16-bit mantissa. The AA based error is obtained using the above method; the maximum error (fourth column) is obtained by a simulation over one million random input vectors. In both cases, the largest error of any output is chosen. The fifth column display the ratio between these error bounds. For size 4, the AA-based method is very accurate, for size 64, it is a factor of 3.27 worse than the simulated error. Clearly, for a floating-point program with thousands of arithmetic operations, the AA-based hard error bound may become useless.

The main reason for the decreasing accuracy in the AA based analysis compared to simulation is the increasing unlikeliness that all errors ε_i in (10) are simultaneously close to being maximal. To formalize this behavior, we assume that the error symbols ε_i in (10) are independent random variables uniformly distributed in [-1, 1]. We set $S_N = \sum_{i=1}^{N} C_i\varepsilon_i$ and denote with $\mathcal{N}(0,1)$ a standard normally distributed random variable. Then, by the *central limit theorem*,

$$\frac{S_N}{\sqrt{N}\sqrt{Variance(S_N)}} \to \mathcal{N}(0,1).$$

We use this interpretation to develop a refinement of the AA-based error analysis, which is based on *probabilistic bounds*. To achieve this, we modify the bounding operator B such that it returns a *confidence interval* that bounds the error with a specified high probability λ. We denote this new operator B_λ and define it by

$$B_\lambda(C_0 + \sum_{i=1}^{N} C_i\varepsilon_i) = C_0 + F_\lambda(S_N), \qquad (11)$$

where $F_\lambda(S_N)$ is the probabilistic bound for S_N

$$prob(error \leqslant F_\lambda(S_N)) \geqslant \lambda.$$

To calculate $F_\lambda(S_N)$ for a given λ, we use the inverse CDF (Cumulative Density Function) of S_N for $N = 1, 2, 3$, and Gaussian approximation for $N > 3$. If c is the number of operations in the given program, then the final maximum estimated error K over all outputs satisfies

$$prob(error \leqslant K) \geqslant \lambda^c.$$

4. EXPERIMENTAL RESULTS

4.1 Methodology

Recall our original goal: estimate quickly the numerical errors that accrue given a candidate reduced-precision custom floating-point format. The actual "error" we seek is the maximum difference between the floating-point value computed in this format, and the "ideal" real value, which we take to be the IEEE double-precision version of the computation. With bit-level simulation over a suitably large set of inputs, we can calculate this error. In all the experiments in this section, we compare our estimated error bound against this simulated maximum error.

Two C++ libraries are built to assist the AA-based error analysis and the evaluation. One is a custom floating-point library, called *CMUfloat* [3]. Arbitrary mantissa and exponent width can be specified in the variable declaration. The maximal error is obtained by comparing the output of the *CMUfloat* and double-precision versions of the code simulated with 10^6 randomly generated independent uniformly distributed inputs. We assume a 16-bit mantissa and an 8-bit exponent in all the experiments. The other library is an AA-based floating-point computation library that overloads the C++ arithmetic operators and computes the affine form for each operation in the code, and the relevant bound. This strategy allows us to analyze the program with minimal modification to the source code.

4.2 AA vs. IA

To verify that range cancellation enabled by noise symbol sharing in the AA modeling helps improving the estimation accuracy, we compare the AA-based error bound according to (7)–(9) with the conventional IA-based error bound in a DSP application—the 8-input IDCT (Inverse Discrete Cosine Transform), which is widely used in image and video processing. The inputs are assumed to lie in the range [-128, 128]. As shown in Table 2, the AA-based approach provides a much tighter error bound than the IA-based approach. IA overestimates because it fails to consider the

AA error bound	IA error bound	Max error
0.0431	0.0964	0.0203

Table 2: Error analysis results on IDCT

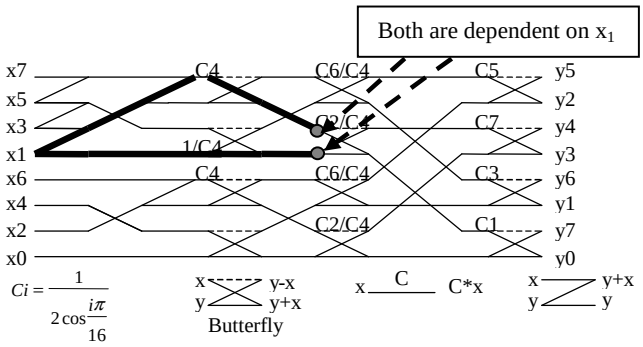


Figure 3: Dataflow of the IDCT algorithm

correlations among variables. We highlight an example of such correlations in the IDCT diagram in Figure 3.

Since correlations in the data path are very common in DSP applications, our AA-based error model significantly improves accuracy compared to the IA-based model, while incurring virtually the same computational cost.

4.3 Benchmark Results

We test the applicability and accuracy of the proposed error model and the probabilistic bounding method on a variety of common signal processing kernels, including WHT, FIR filter, and IDCT. Table 3 displays hard bounds and probabilistic bounds (for $\lambda = 0.9999$) with confidences. Table 4 shows the CPU times needed to compute the estimated bounds and the simulated maximum error. Note that the CPU time required to compute the probabilistic bound is independent of λ. We conclude that our error estimation produces useful estimates significantly faster than simulation.

	Hard bound	Probabilistic bound	Confidence λ^c	Max error
WHT4	0.0029	0.0029	0.9992	0.0027
WHT64	0.1094	0.0325	0.962	0.0334
FIR4	0.0028	0.0023	0.9996	0.0020
FIR25	0.0132	0.0057	0.998	0.0062
IDCT8	0.0431	0.0197	0.9962	0.0203

Table 3: Improvement on accuracy ($\lambda = 0.9999$)

Figure 4 shows the behavior of the probabilistic bound with varying λ (abscissa) for WHT64 with $c = 384$. The confidences λ^c for the bounds are displayed above the bars. Note that there is also a confidence value associated with the simulation result, because the maximum error seen in 10^6 simulations only guarantees that

$$prob(error > maximum\ error) \leqslant \frac{1}{10^6},$$

if we assume uniform error distribution. Therefore the corresponding confidence of the simulated maximum error is 0.999999. The results show that our method generates highly accurate probabilistic bounds that substantially improve the hard bound ($\lambda = 1$) with slight sacrifice in confidence.

	CPU time for probabilistic bound	CPU time for max error
WHT4	0.002	31.40
WHT64	0.173	2854
FIR4	0.005	54.33
FIR25	0.018	331
IDCT8	0.010	282.2

Table 4: Comparison of CPU time (sec)

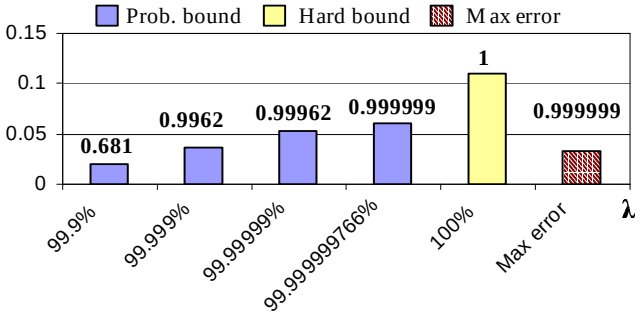


Figure 4: Probabilistic bounds for WHT

4.4 Application Example 1: Exploring the Design Space

We illustrate here how our error analysis tool assists with fast design space exploration in the dimension of numerical precision.

For a given algorithm, various implementations will lead to differing numerical precisions due to different data paths, even with the same floating-point format. In this example, we consider a much more complicated kernel, a DCT type IV of size 64, which requires about 830 arithmetic operations on the data path. We generate four different implementations, based on algebraically different algorithms, using the DSP transform code generator SPIRAL [9], and compare the obtained error bounds. In this experiment, we specify λ to be 0.9999, which offers a confidence of $\lambda^c = 0.92$. The choice of λ does not affect the relative order of the error bounds for the four different algorithms.

In Figure 5, both the probabilistic error bound and the maximum error yield the same order for the four algorithms with respect to numerical accuracy: DCT4 < DCT2 < DCT3 < DCT1, while the probabilistic error bound estimation is about 5000 times faster than running the simulation of one million random inputs. Note that the large spread in accuracy makes the choice of the algorithm a mandatory task.

4.5 Application Example 2: Determining the Minimum Mantissa Bit-width

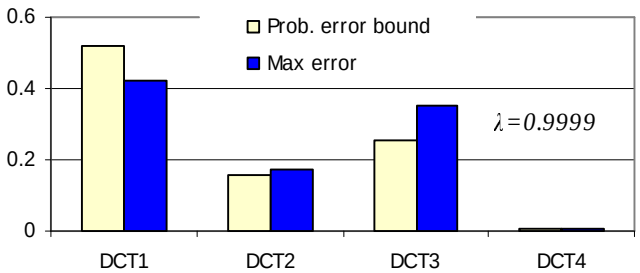


Figure 5: Comparison of four DCT algorithms

A common problem in designing a custom floating-point application is to determine the mantissa bit-width that offers enough precision at minimal hardware cost. From the basic floating-point error model in (1), we know that the error bound is proportional to 2^{-t}, where t is the mantissa bit-width. The methodology for configuring the mantissa bit-width is shown in Figure 6. As depicted in the figure, t_1 is an initial value for the mantissa bit-width. This value can be purely random since it does not affect the final bit-width. Then the AA-based error bound estimation is used to estimate the minimum mantissa bit-width t_2. Finally, the program is simulated using the t_2-bit mantissa with the help of the *CMUfloat* library. Using the result, the mantissa bit-width may be slightly adjusted for optimality.

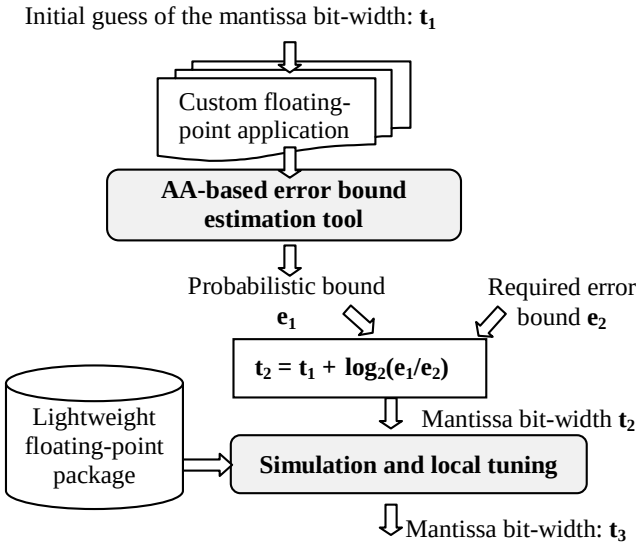


Figure 6: Configuring mantissa bit-width

Following the experiment in the previous section, we now determine the minimal mantissa bit-width for the best algorithm DCT4 (Figure 5). Suppose the required max error e_2 is 0.01, and our initial guess of the mantissa bit-width is 20. Experimental results are shown in Table 5. By using the static error analysis method, a new mantissa bit-width $t_2 = 16$ is estimated. Then after the simulation, the final bit-width is adjusted to $t_3 = 15$.

t_1	e_1	t_2	e_2	t_3
20	4.32e-4	16	1e-2	15

Table 5: Steps in mantissa configuration

5. CONCLUSION

We proposed an efficient static error analysis method based on affine arithmetic and probabilistic bounding, to enable fast custom hardware design of reduced-precision DSP applications. The affine formulation of the rounding error inherent in a custom floating point format is one useful contribution of the work; another important idea is the formulation of the bound itself as a random variable, estimated via confidence intervals. Results for experiments mapping common DSP transform kernels to implementations using small custom floating point formats show that the static analysis bounds with usable accuracy, but is at least three orders of magnitude faster than direct bit-level simulation.

Our static analysis method can provide an estimation for errors, or functions of errors, such as mean square error or signal-to-noise ratio. For applications that rely on measurements not closely related to the error, such as the convergence rate in adaptive filtering, or the perceptual quality in audio processing, our method is not directly applicable.

The modeling method can be adapted to fixed-point error analysis. It is part of our current research work.

6. ACKNOWLEDGMENTS

This work was funded by Semiconductor Research Corporation and Pittsburgh Digital Greenhouse.

7. REFERENCES

[1] L. H. de Figueiredo and J. Stolfi. Self-validated numerical methods and applications. *Brazilian Mathematics Colloquium monograph, IMPA, Rio de Janeiro, Brazil*, July 1997.

[2] F. Fang, T. Chen, and R. Rutenbar. Floating-point bit-width optimization for low-power signal processing applications. In *International Conf. on Acoustic, Speech, and Signal Processing*, May 2002.

[3] F. Fang, T. Chen, and R. Rutenbar. Lightweight floating-point arithmetic: Case study of inverse discrete cosine transform. *EURASIP J. Sig. Proc.; Special Issue on Applied Implementation of DSP and Communication Systems*, 2002(9):879–892, Sept. 2002.

[4] W. Kramer. A prior worst case error bounds for floating-point computations. *IEEE Trans. Comp.*, 47:750–756, July 1998.

[5] T. L. Laakso and L. B. Jackson. Bounds for floating-point roundoff noise. *IEEE Trans. Circ. Sys II: Analog and Digital Signal Processing*, 41:424–426, June 1994.

[6] A. Lemke, L. Hedrich, and E. Barke. Analog circuit sizing based on formal methods using affine arithmetic. In *International Conf. on Computer Aided Design*, Nov. 2002.

[7] B. Liu and T. Kaneko. Error analysis of digital filters realized with floating-point arithmetic. *Proc. IEEE*, 57:1735–1747, Oct. 1969.

[8] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[9] M. Püschel, B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms. *to appear in Journal of High Performance Computing and Applications*.

[10] B. D. Rao. Floating point arithmetic and digital filters. *IEEE Trans. Sig. Proc.*, 40:85–95, Jan. 1992.

[11] P. H. Sterbenz. *Floating-Point Computation*. Prentice-Hall, 1974.

[12] C. Tsai. Floating-point roundoff noises of first- and second-order sections in parallel form digital filters. *IEEE Trans. Circ. Sys II: Analog and Digital Signal Processing*, 44:774–779, Sept. 1997.

[13] C. Weinstein and A. V. Oppenheim. A comparison of roundoff noise in floating point and fixed point digital filter realizations. *Proc. IEEE*, 57:1181–1183, June 1969.