Evaluation of Static Analysis Techniques for Fixed-Point Precision Optimization

Jason Cong[1], Karthik Gururaj[1], Bin Liu[1], Chunyue Liu[1], Zhiru Zhang[2], Sheng Zhou[2], Yi Zou[1]

[1]Computer Science Department
University of California Los Angeles
Los Angeles, CA 90095, USA

[2]AutoESL Design Technologies
Los Angeles, CA 90064, USA

***Abstract*—Precision analysis and optimization is very important when transforming a floating-point algorithm into fixed-point hardware implementations. The core analysis techniques are either based on dynamic analysis or static analysis. We believe in static error analysis, as it is the only technique that can guarantee the desired worst-case accuracy. In this paper we study various underlying arithmetic candidates that can be used in static error analysis and compare their computed sensitivities. The approaches studied include Affine Arithmetic (AA), General Interval Arithmetic (GIA) and Automatic Differentiation (Symbolic Arithmetic). Our study shows that symbolic method is preferred for expressions with higher order cancelation. For programs without strong cancelation, any method works fairly well and GIA slightly outperforms others. We also study the impact of program transformations on these arithmetics.**

I. Introduction

FPGAs provide the capability to customize the bitwidth required for an efficient fixed-point computation. In general, bitwidth optimization for fixed-point design is usually composed of two parts: 1) integer bitwidth (IB) optimization, 2) fractional bitwidth (FB) optimization. IB optimization is based on range information of values, and FB optimization typically requires precision analysis. The output error of the design needs to be smaller than user-specified error tolerance. The error tolerance can be either a worst-case metric or a statistical/stochastic metric. Automatic differentiation (AD) is used in [2] for precision analysis. The approach first obtains the symbolic expression of the gradient/sensitivity. Then it relies on user-specified input data or sampling to convert those symbolic sensitivities into numerical constraints. The approach is a dynamic analysis, but when the number of the samples is large, it can serve as a reference for static analysis techniques. *Affine arithmetic* (AA) [6], as an improved static analysis technique over the traditional *interval arithmetic* (IA) [7], is used in [3]–[5] for precision analysis. AA can often achieve tighter bounds than IA with better estimation of error cancelation. The work in [3] gives statistical interpretations of affine arithmetic.

While there are multiple work using different kinds of analysis techniques and heuristics for bitwidth minimization, usually they do not make head-to-head comparisons. It is not clear how these different analysis and optimization techniques differ on common design examples. For example, MiniBit [4] claims that its results are within 1% of optimal solution. However, optimizations need accurate sensitivity information to build up constraints. If the static analysis it has employed is already too conservative, better solutions may exist.

This paper focuses on quantitative evaluation of various kinds of static analysis techniques. We also suggest two simple schemes/extensions (GIA, AD+AA) that have not been applied in precision analysis previously. Our results show that simple techniques such as AA and GIA are sufficient for expressions that do not have higher order cancelations, while they would still overestimate for higher order expressions.

II. Affine Arithmetic for Precision Analysis

A. *Affine Arithmetic*

In affine arithmetic, the uncertainty of a variable x is represented as

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i \varepsilon_i, \tag{1}$$

where each ε_i is an interval $[-1,1]$, and each x_i is a scalar. Each term $x_i\varepsilon_i$ represents the uncertainty on x caused by some primordial uncertainty i. Using this form, correlations between variables can be captured, and symbolic cancelation of uncertainties are possible.

Additions and subtractions in affine arithmetic only involves the addition and substraction of the coefficients of the affine form.

$$\hat{x} \pm \hat{y} = x_0 \pm y_0 + \sum_{i=1}^{n} (x_i \pm y_i)\varepsilon_i. \tag{2}$$

However, some operations will generate nonlinear terms. In order to maintain the the affine form for every variable, approximations are needed. Take multiplication as an example, we have the resulting

$$\begin{aligned} \hat{x}\hat{y} &= (x_0 + \sum_{i=1}^{n} x_i\varepsilon_i)(y_0 + \sum_{i=1}^{n} y_i\varepsilon_i) \qquad (3) \\ &= x_0y_0 + \sum_{i=1}^{n}(x_0y_i + y_0x_i)\varepsilon_i + (\sum_{i=1}^{n} x_i\varepsilon_i)(\sum_{i=1}^{n} y_i\varepsilon_i) \end{aligned}$$

Note that the following relation between the two intervals

$$(\sum_{i=1}^{n} x_i\varepsilon_i)(\sum_{i=1}^{n} y_i\varepsilon_i) \subseteq (\sum_{i=1}^{n} |x_i|)(\sum_{i=1}^{n} |y_i|)\varsigma, \tag{4}$$

where ς is a new interval $[-1,1]$. Using this result, we can conservatively estimate $\hat{x}\hat{y}$ as $x_0y_0 + \sum_{i=1}^{n}(x_0y_i + y_0x_i)\varepsilon_i +$

$(\sum_{i=1}^{n}|x_i|)(\sum_{i=1}^{n}|y_i|)\varsigma$. This approach can successfully capture the bound of the quadratic term. However, it adds a new interval variable, so that the number of interval variables will grow along with the number of non-linear operations performed. In addition, the new interval variable is independent of any existing interval variable; this results in loss of correlation information and probably a looser bound in further analysis.

B. Application of Affine Arithmetic in Precision Analysis

For a floating-point variable x, let $\bar{x}$ be its finite-precision representation, and let E_x be the rounding error, so that $x = \bar{x} + E_x$. For addition/subtraction, $(x \pm y) - (\bar{x} \pm \bar{y}) = E_x \pm E_y$. For multiplication, we have

$$\begin{aligned} xy - \bar{x}\bar{y} &= \bar{x}\bar{y} + \bar{x}E_y + \bar{y}E_x + E_xE_y - \bar{x}\bar{y} \\ &\approx \bar{x}E_y + \bar{y}E_x. \end{aligned} \tag{5}$$

Note that the term E_xE_y is discarded because both E_x and E_y are typically very small.

To use affine arithmetic for precision analysis, we can express both $\bar{x}$ and E_x as affine expressions. If p bits are used to represent the fractional value of x, we can use $2^{-p}\varepsilon$ to represent the possible values of E_x. However, the terms $\bar{x}E_y$ and $\bar{y}E_x$ are not in affine form, and multiplication in affine arithmetic is needed to convert them into affine form with additional variables.

Suppose we need to perform precision analysis on a simple polynomial $x^2 - 6x$ and the range of x is $[-1,3]$. If we assume the truncation error of x is $2^{-p}\varepsilon_1$, the first order error of x^2 due to the truncation of x is $2x \times 2^{-p}\varepsilon_1$. This is not an affine form, we need to convert x into $1+2\varepsilon_2$ and the final error of x^2 in affine form is $2^{1-p}\varepsilon_1 + 2^{2-p}\varepsilon_3$ where ε_3 is a new interval variable (due to $\varepsilon_1\varepsilon_2$) . The final affine error of $x^2 - 6x$ is $-2^{2-p}\varepsilon_1 + 2^{2-p}\varepsilon_3$ (Note, a common mistake is to put the worst-case bound of $|x|$ into the error expression $2x \times 2^{-p}\varepsilon_1$ directly and simply claim the worst case error of x^2 is $6 \times 2^{-p}\varepsilon_1$. $6 \times 2^{-p}\varepsilon_1$ is indeed a worst case error for x^2, but it can cause incorrect cancelation in later computation stages: $6^{-p}\varepsilon_1 - 6 \times 2^{-p}\varepsilon_1 = 0$. This mistake appears in many AA-based precision analysis or bitwidth optimization papers in literature [3]–[5].)

The affine error in this example can be used to generate a constraint $\inf(|-2^{2-p}\varepsilon_1 + 2^{2-p}\varepsilon_3|) \leq \text{ErrorBound}$, or $8 \times 2^{-p} \leq \text{ErrorBound}$. Here the value 8 is the upper bound of the sensitivity of $x^2 - 6x$ upon x. This sensitivity can be used to decide the number of bits to represent the fractional part of a value. It is also possible to get the upper bound of sensitivity using other methods, as discussed in the following sections.

III. General Interval Arithmetic for Precision Analysis

A. General Interval Arithmetic

If we allow the coefficients of the affine form to be intervals, the analysis for the above example can be greatly simplified and the error is $(2x-6) \times 2^{-p}\varepsilon_1 = (2 \times [-1,3] - 6) \times 2^{-p}\varepsilon_1 = [-8,0] \times 2^{-p}\varepsilon_1$.

This type of arithmetic is called general interval arithmetic (GIA) [8]. This arithmetic is originally developed to reduce the excessive width generated from regular interval operations [7]. The representation and the arithmetic are similar to the affine arithmetic, but the coefficients are intervals rather than scalars.

The addition/subtraction of the GIA also invokes addition/subtraction of the coefficients, but we need to use interval addition/subtraction rather than scalar addition/subtraction. Multiplication will also generate higher-order terms which can be handled as follows.

$$\begin{aligned} \hat{x}\hat{y} &= \tilde{x}_0\tilde{y}_0 + (\tilde{x}_0 + \sum_{i=1}^{n}\tilde{x}_i\varepsilon_i)(\tilde{y}_0 + \sum_{i=1}^{n}\tilde{y}_i\varepsilon_i) \\ &= \tilde{x}_0\tilde{y}_0 + \sum_{i=1}^{n}(\tilde{x}_0\tilde{y}_i + \tilde{y}_0\tilde{x}_i)\varepsilon_i + (\sum_{i=1}^{n}\tilde{x}_i\varepsilon_i)(\sum_{i=1}^{n}\tilde{y}_i\varepsilon_i) \\ &= \tilde{x}_0\tilde{y}_0 + \sum_{i=1}^{n}(\tilde{x}_0\tilde{y}_i + \tilde{y}_0\tilde{x}_i + \sum_{i=1}^{n}\tilde{x}_i[-1,1]\tilde{y}_i)\varepsilon_i \end{aligned} \tag{6}$$

Here the coefficients $\tilde{x}_i$, $\tilde{y}_i$ including $\tilde{x}_0$,$\tilde{y}_0$ are all intervals. The resulting expression is still in a GIA form (affine form with interval coefficients). Interval arithmetic is used to compute coefficients such as $\tilde{x}_0\tilde{y}_i$ etc.

B. Application of General Interval Arithmetic in Precision Analysis

In precision analysis, we can throw away the cross terms and use the following equation.

$$\begin{aligned} \hat{x}\hat{y} - \tilde{x}_0\tilde{y}_0 &= (\tilde{x}_0 + \sum_{i=1}^{n}\tilde{x}_i\varepsilon_i)(\tilde{y}_0 + \sum_{i=1}^{n}\tilde{y}_i\varepsilon_i) - \tilde{x}_0\tilde{y}_0 \\ &= \sum_{i=1}^{n}(\tilde{x}_0\tilde{y}_i + \tilde{y}_0\tilde{x}_i)\varepsilon_i + (\sum_{i=1}^{n}\tilde{x}_i\varepsilon_i)(\sum_{i=1}^{n}\tilde{y}_i\varepsilon_i) \\ &\approx \sum_{i=1}^{n}(\tilde{x}_0\tilde{y}_i + \tilde{y}_0\tilde{x}_i)\varepsilon_i \end{aligned} \tag{7}$$

$\tilde{x}_0$ and $\tilde{y}_0$ are known after range analysis. Interval multiplication and interval addition/subtraction are needed in the computation. In GIA-based precision analysis, the cross terms that we throw away are the multiplication of two rounding errors (which are typically very small); but in AA-based precision analysis, terms coming from $\bar{x}E_y$ and $\bar{y}E_x$ (multiplication of value terms and error terms) are also second order. These terms may not be so small. We throw away the cross term E_xE_y in AA-based analysis, but keep $\bar{x}E_y$ and $\bar{y}E_x$ terms .

Note it is also straightforward to include an additional error interval to accommodate the cross terms that we throw away in both AA and GIA. Eq.(6) is one way to accommodate the cross terms in a pessimistic fashion. We throw away the multiplication of two error terms in AA and GIA, because first-order Taylor approximation used in automatic differentiation (in next section) also does not have those terms. We have to throw away the terms in order to have a fair comparison. In practice, the approximation (first-order Taylor approximation) works if the error terms are relatively small.

It is also very easy to handle enclosure (min and max) operation in GIA. When the program contains some control flow and one variable may take multiple paths, enclosure operation can be used to estimate the maximum error from all the paths. We only need to perform the enclosure operation on each interval coefficient. But in AA, this operation is again non-linear and generates additional interval variables.

Because GIA does not produce additional interval variable for each non-linear operation, it is more scalable for code fragments containing many nonlinear operations. However, affine arithmetic allows cancelations for the additional interval variables generated from non-linear operations. GIA may not get as tighter bound as AA for these cases.

We are not aware of any previous bitwidth optimization work that claims to use GIA directly; however, the work [9] uses two affine error expressions: positive error and negative error. It also designs the error propagation arithmetic for them. Mathematically it is very close to the GIA we have described here (if not exactly the same).

IV. Automatic Differentiation for Static Precision Analysis

The methods we described in the previous sections can get the bound of the sensitivity numerically. Automatic differentiation can get the exact form of the sensitivity symbolically.

Although automatic differentiation is used by [2] as a dynamic analysis technique, we can also bring this into static analysis. After the symbolic sensitivities are obtained, we can use IA or AA to obtain the bound of the sensitivity, and further use these numerical ranges to feed the constraints for the optimization. If we use IA to compute the bound of each sensitivity, it actually is the same as the GIA approach we presented in the previous section. We can also use AA to obtain a possible tighter bound on these sensitivities. For the example we described in Section II, the symbolic sensitivity is $2x-6$. Applying either IA or AA on this sensitivity will get the same bound [-8,0], whose worst case absolute value is 8.

Applying AA on the symbolic sensitivity (we term this as AD with AA) is a second-order method, which keeps track of cancelations up to second order. It can also be viewed as an arithmetic with an affine expression as coefficients. In contrast, GIA uses intervals for coefficients while pure AA uses scalars.

As the symbolic expression of the sensitivity may not be convex, the exact bound may involve global optimization techniques, such as branch and bound or Monte Carlo sampling.

Automatic differentiation cannot handle non-differential operations e.g., the enclosure operation we talk about earlier. Thus it is difficult to get a symbolic sensitivity when control flows present. This paper limits the discussion to DFGs only.

V. Form of Expressions

Different forms of the same expression can generate completely different analysis results. Affine arithmetic and general interval arithmetic can handle first order cancelations very well. For example, they can detect the cancelation effects in expression such as $100x-99x$. However, they are not able to detect higher order cancelations, e.g., $100x^2-99x^2$.

Even if the computation does not contain these trivial cancelations, expressions such as polynomials can still be written in different forms such as monomial form, horner expansion [10] and expanded form. These different forms can also generate different analysis results. We study these effects and compare the results in the next section.

VI. Experimental Results

We implement the static analysis techniques discussed in Sections II-IV in C/C++ langauge as passes in LLVM [1]. These analysis passes are part of our bitwidth optimization engine for floating-point to fixed-point conversion. Four static analysis method are compared: (1) AA-based precision analysis, (2) GIA-based precision analysis, (3) Automatic differentiation with AA for the range analysis of the symbolic sensitivities, (4) Exact bounds of the symbolic expression obtained by automatic differentiation.

A. Effect of Cancelations

First, we compare the effects of cancelations. The computed sensitivities are listed in Table I. For simple low-order polynomial such as x^2-6x, all the four methods generate the same sensitivity. The first order cancelation effect in $100x-99x$ can be detected by AA and GIA, and they can get the exact sensitivity. AA and GIA can not detect higher order cancelation and they generate a very pessimistic estimation for $100x^2-99x^2$ and $100x^3-99x^3$. If we reuse x^2 or x^3 rather than recompute them, AA can detect the cancelation and generate exact result, but GIA cannot because the interval operation on the interval coefficients does not cancel very well. Automatic differentiation with AA as range analysis for the symbolic sensitivity can detect up to second order cancelation. Note we do not perform symbolic expression simplification for the symbolic sensitivity obtained from automatic differentiation.

One may argue that these cancelations are trivial and compilers may detect them automatically. Furthermore, no one would write the code that does a computation like $100x^3-99x^3$ as we all know this is equal to x^3. This may be true for these simple examples. However, high order cancelations still exist in expressions such as $100x^3-99(x+0.0001)^3$ and symbolic cancelation or expression simplification is nontrivial. Similar experiment results like Table I can still be obtained and AA and GIA will still be far away from optimal for these cases.

The number of fractional bits for the variable is related to the logarithm of sensitivity, and a 150X overestimation of sensitivity can be translated to a possible 7-bit difference. Static analysis need to be used with care if the program contains potential higher order cancelations.

B. Form of Expressions

We then study some expressions which do not contain the trivial cancelations. Two examples here are the Taylor expansion of $sin(x)$ and $ln(1+y)$. We use $sin(x)\approx x-\frac{1}{6}x^3+\frac{1}{120}x^5-\frac{1}{5040}x^7+\frac{1}{362880}x^9$ and $ln(1+y)\approx y-\frac{1}{2}y^2+\frac{1}{3}y^3-\frac{1}{4}y^4$. The data flow graph of the two schemes (Monomial or Horner) are shown in Fig. 1. The computed sensitivity (on x or y) is

TABLE I

SENSITIVITY COMPARISON—EFFECTS OF CANCELATIONS: $x \in [-1,3]$

Expression	AA	GIA	AD with AA	Exact
$x^2 - 6x$	8	8	8	8
$100x - 99x$	1	1	1	1
$100x^2 - 99x^2$	798	798	6	6
$y = x^2; 100y - 99y$	6	798	6	6
$100x^3 - 99x^3$	4779	3294	2403	27
$y = x^3; 100y - 99y$	27	3294	27	27

TABLE II

SENSITIVITY COMPARISON—FORM OF EXPRESSIONS: $x \in [0, \pi/2], y \in [0,1]$

Expression	AA	GIA	AD with AA	Exact
$sin(x)$ (Monomial)	1.81563	1.25459	1.70877	1
$sin(x)$ (Horner)	1.76637	1.10667	1.63827	1
$ln(1+y)$ (Monomial)	1.97917	2	1.5	1
$ln(1+y)$ (Horner)	1.45833	1.16667	1.29167	1

shown in Table II. We can see the sensitivity can be more accurately computed by any method if the expressions are written in horner scheme, because there are fewer cancelations in the scheme. From the comparative data in the first three columns, it is hard to say which method is the best. GIA slightly outperforms other methods but not consistently. The data in the column AD with AA is often worse than the column GIA, because AA does not always generate tighter bound than IA if no strong cancelation occurs. All the methods can obtain a sensitivity which is not far away from optimal, because there are no strong cancelations in the expressions.

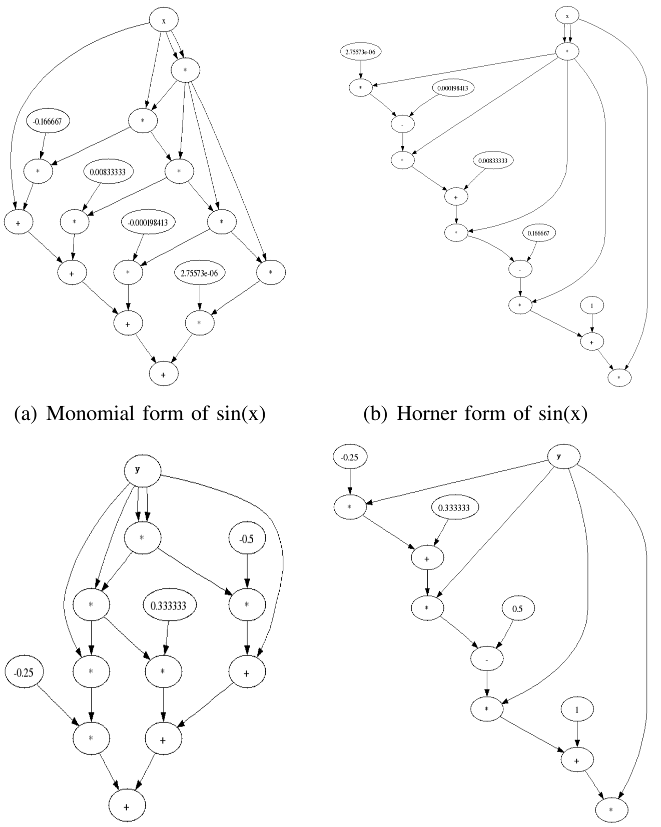
As the static sensitivity computation is sensitive to the form of expressions, it is useful to have certain compiler transform which can make use of this fact and generate a form that is more friendly to the analysis. Note in practice, designer shall prefer a horner scheme as it uses fewer multiplications.

VII. FURTHER DISCUSSIONS AND CONCLUSIONS

This paper so far focuses on the error coming from the truncations of one input variable. Similar techniques can be used to capture the propagation of errors from truncation of constants and intermediate variables. Their sensitivities on the final error can also be computed in a similar fashion. However, even if we can get the exact sensitivities, the correlation between the sensitivities is still not captured in the static analysis.

AA and GIA can support the enclosure operation, which can be used to analyze the code with control flow. However, it is not easy to get an exact symbolic expression through automatic differentiation and thus it is difficult to get an optimal static analysis.

This paper presents a comparative study for three static analysis arithmetics and their use in precision analysis. For programs with higher order cancelations, a symbolic or higher order method is preferred. For programs without strong cancelations, any method can estimate the sensitivity relatively well and GIA slightly outperforms others. Also, our limited experiments show that Horner scheme is a preferred form over the monomial form for sensitivity computation using these static analysis arithmetics.



(a) Monomial form of sin(x) (b) Horner form of sin(x)

(c) Monomial form of log(1+y) (d) Horner form of log(1+y)

Fig. 1. DFG of polynomials

ACKNOWLEDGMENT

This work is partially funded by Natural Science Foundation grant 0306682 and also a grant from Altera Corporation and Magma Design Automation under the California MICRO program.

REFERENCES

[1] http://www.llvm.org

[2] A. A. Gaffar, O. Mencer, W. Luk, and P. Cheung, “Unifying bit-width optimisation for fixed-point and floating-point designs,” In *Proc. FCCM*, pages 79–88, 2004.

[3] C. F. Fang, R. A. Rutenbar and T. Chen, “Fast, accurate static analysis for fixed-point finite-precision effects in DSP designs,” In *Proc. ICCAD*, pages 275–282, 2003.

[4] D-U. Lee, A. A. Gaffar, O. Mencer and W. Luk, “MiniBit: bitwidth optimization via affine arithmetic,” In *Proc. DAC*, pages 837–840, 2005.

[5] Y. Pu and Y. Ha, “An automated, efficient and static bit-width optimization methodology towards maximum bit-width-to-error tradeoff with affine arithmetic model,” In *Proc. ASPDAC*, pages 886–891, 2006.

[6] J. Stolfi and L. H. de Figueiredo, “Self-validated numerical nethods and applications,” Brazilian Mathematics Colloquium Monograph, IMPA, Rio De Janeiro, Brazil, 1997.

[7] R. E. Moore, “Methods and applications of interval analysis,” SIAM, 1979.

[8] E. R. Hansen, “A generalized interval arithmetic,” Interval mathematics (K. Nickel, ed.): LNCS 29, 7–18 (Springer-Verlag, 1975).

[9] N. Doi, T. Horiyama, M. Nakanishi, and S. Kimura, “An optimization method in floating-point to fixed-point conversion using positive and negative error analysis and sharing of operations,” In *Proc. SASIMI*, pages 466–471, 2004.

[10] R.Seroul, “Evaluation of Polynomials: Horner’s Method,” In *Programming for Mathematicians. Berlin: Springer-Verlag*, pp. 216-262, 2000.