Static Analysis-Based Validation of Floating-Point Computations

Sylvie Putot, Eric Goubault, Matthieu Martel

CEA Saclay, F91191 Gif-sur-Yvette Cedex, France
{sputot,egoubault,mmartel}@cea.fr

Abstract. Finite precision computations can severely affect the accuracy of computed solutions. We present a static analysis, and a prototype implementing this analysis for C codes, for studying the propagation of rounding errors occurring at every intermediary step in floating-point computations. The analysis presented relies on abstract interpretation by interval values and series of interval error terms. Considering all errors possibly introduced by floating-point numbers, it aims at identifying the operations responsible for the main losses of accuracy. We believe this approach is for now specially appropriate for numerically simple programs which results must be verified, such as critical instrumentation software.

Keywords. Static analysis, floating-point computations, intervals

1 Introduction

The manipulation of real numbers by computers is carried on using floating-point arithmetic, which relies on a finite representation of numbers. Although this approximation is accurate enough for most applications, in some cases results become irrelevant. And in critical software, these cases may not be acceptable.

Some work has already been done towards tools for evaluating the accuracy of computations in software. The most widely used, Cadna, relies on statistical methods, and gives most of the time a very sharp estimation of the relevance of computed results. But some errors can be underestimated, which is not satisfying for the verification of critical applications, which accuracy must be certified. Moreover, this method allows to study the result of a particular execution, and not for infinite sets of input values as is most of the time needed. Alternatively, most existing interval-based techniques, which are sure and consider sets of executions, aim at estimating tight bounds for the result of computations in infinite precision. This often supposes a rewriting of the code to be analyzed, and moreover does not address the problem of verifying the accuracy of existing software.

On the contrary, we are not interested in computing bounds for the real result of a given problem, but for the error committed using finite precision computations instead of real numbers computations[1]. Moreover, the origin of the main losses of precision is most of the time very localized, and we aim at pointing out

[1] This presentation follows earlier work by the authors, see [3],[6],[4]

which parts of the code are responsible for these losses. For that, we decompose the error between the results of the same computation achieved respectively with floating-point and real numbers in a sum of error terms corresponding to the elementary operations of this computation. This modelisation of the propagation of errors, called concrete semantics, is the topic of section 2.

This semantics can not be used in an analyzer, because the errors are real numbers, that can not always be represented by floating-point numbers, even with higher precision. Thus we derive an abstract semantics, which is the implementable version of the concrete semantics : over-approximations of the values and errors are computed using intervals. These intervals also allow to consider sets of input values. Static analysis consists in computing all possible values of the variables on the nodes of a program without executing it. A considerable issue is the fixed point computation in loops. This is presented in section 3.

A prototype implements this model, it is intended to cope with real problems. Special care was attached to the design of a graphic interface, that makes the large amount of information computed easily exploitable. The user can make sure that the floating-point computations are accurate enough, and identify the operations responsible for the main losses of accuracy.

2 Concrete Semantics to Interpret Arithmetic Operations

Let us first examine an introductory example in which we consider a simplified set $\mathbb{F}$ of floating-point numbers composed of a mantissa of four digits written in base 10. We consider a and b two intermediate computation results that are not computed exactly, and we note

$$a = 621.3 + 0.055\varepsilon_{\ell_1}, \;\; b = 1.287 + 0.00055\varepsilon_{\ell_2} \; .$$

In this definition, $a \in \mathbb{R}$ and $b \in \mathbb{R}$ are the values that would be got from an infinite precision computation. The floating-point execution of the same computation gives $a_{\mathbb{F}} = 621.3 \in \mathbb{F}$ and $b_{\mathbb{F}} = 1.287 \in \mathbb{F}$, and an error of 0.055 was committed at point ℓ_1 on the computation of a, and an error of 0.00055 was committed at point ℓ_2 on the computation of b. The symbols ε_{ℓ_1} and ε_{ℓ_2} are formal variables related to the control points ℓ_1 and ℓ_2.

We now consider the product $c = a \times b$, at point l_3. The exact result of the product of the floating-point numbers is $a_{\mathbb{F}} \times b_{\mathbb{F}} = 799.6131$, but the nearest floating-point number, supposing the current rounding mode is to the nearest, is $c_{\mathbb{F}} = 799.6$. A rounding error, defined by $a_{\mathbb{F}} \times b_{\mathbb{F}} - c_{\mathbb{F}} = 0.0131$, is thus committed. The computation $a \times b$ in real numbers intended by the programmer, is then

$$a \times^{\ell_3} b = c_{\mathbb{F}} + 0.0131\varepsilon_{\ell_3} + 0.055 \times 1.287\varepsilon_{\ell_1} + 0.00055 \times 621.3\varepsilon_{\ell_2} + 0.055 \times 0.00055\varepsilon_{\ell_1}\varepsilon_{\ell_2} \; .$$

We keep only one term gathering the errors of order higher than one, and rewrite

$$a \times^{\ell_3} b = c_{\mathbb{F}} + 0.070785\varepsilon_{\ell_1} + 0.341715\varepsilon_{\ell_2} + 0.0131\varepsilon_{\ell_3} + 0.00003025\varepsilon_{hi} \; .$$

The initial errors are amplified or reduced by further computations, thus the error on c is mainly due to the initial error on b. This result is quite obvious on this very simple example, but would be much more difficult and tedious to establish by hand on larger programs. We aim at designing an automatic tool providing this kind of information.

We now introduce formally this semantics [6], that details the contribution to the global error of the first order error terms, and globally computes the higher order errors, which are most of the time negligible. Let $\mathbb{F}$ be either the set of simple or double precision floating-point numbers. Let $\uparrow_\circ : \mathbb{R} \to \mathbb{F}$ be the function that returns the rounded value of a real number r, with respect to the rounding mode $\circ$. The function $\downarrow_\circ : \mathbb{R} \to \mathbb{F}$ that returns the roundoff error is defined by

$$\forall f \in \mathbb{R},\ \downarrow_\circ (f) = f - \uparrow_\circ (f) \ . \tag{1}$$

Assume that the control points of a program are annotated by unique labels $\ell \in L$, and that $\mathcal{L}$ denotes the union of L and the special word *hi* used to denote all terms of order higher or equal to 2. A number x is represented by

$$x = f^x + \sum_{\ell \in \mathcal{L}} \omega_\ell^x \varepsilon_\ell \ . \tag{2}$$

In equation (2), f^x is the floating-point number approximating the value of x. A term $\omega_\ell^x \varepsilon_\ell$ denotes the contribution to the global error of the first-order error introduced by the operation labeled ℓ, $\omega_\ell^x \in \mathbb{R}$ being the value of this error term and ε_ℓ a formal variable labelling the operation ℓ.

The result of an arithmetic operation $\Diamond^{\ell_i}$ contains the combination of existing errors on the operands, plus a new roundoff error term $\downarrow_\circ (f^x \Diamond f^y)\varepsilon_{\ell_i}$. For addition and subtraction, the errors are added or subtracted componentwise :

$$x +^{\ell_i} y = \uparrow_\circ (f^x + f^y) + \sum_{\ell \in \mathcal{L}} (\omega_\ell^x + \omega_\ell^y)\varepsilon_\ell + \downarrow_\circ (f^x + f^y)\varepsilon_{\ell_i} \ .$$

The multiplication introduces higher order errors, we write :

$$x \times^{\ell_i} y = \uparrow_\circ (f^x f^y) + \sum_{\ell \in \mathcal{L}} (f^x \omega_\ell^y + f^y \omega_\ell^x)\, \varepsilon_\ell + \sum_{\ell_1 \in \mathcal{L},\ \ell_2 \in \mathcal{L}} \omega_{\ell_1}^x \omega_{\ell_2}^y \, \varepsilon_{hi} + \downarrow_\circ (f^x f^y)\varepsilon_{\ell_i} \ .$$

The semantics for the division is obtained by a power series development :

$$(y)^{-1^{l_i}} = \uparrow_\circ \left(\frac{1}{f^y}\right) - \frac{1}{(f^y)^2} \sum_{\ell \in \mathcal{L}} \omega_\ell^y \varepsilon_\ell + \frac{1}{f^y} \sum_{n \geq 2} (-1)^n \left(\sum_{\ell \in \mathcal{L}} \frac{\omega_\ell^y}{f^y} \right)^n \varepsilon_{hi} + \downarrow_\circ \left(\frac{1}{f^y}\right) \varepsilon_{\ell_i} \ .$$

3 Static Analysis and Abstract Interpretation

Static analysis consists in computing some properties of a program without executing it, and for possibly large or infinite sets of inputs. We want here to compute all possible values f and errors ω_ℓ for each variable, valid for any iteration of the loops, on the nodes of the programs to analyze. Interval computations [8] are used to get computable supersets of these coefficients, in an abstract interpretation framework [2]. They allow on one hand to consider sets of execution, and on the other hand to include the rounding errors committed by the analysis.

3.1 Abstract Semantics to Interpret Arithmetic Operations

Let us consider again the multiplication introduced in section 2. The errors are real numbers, they are not always representable by floating-point numbers. Thus we define the abstract semantics for the operation, that implements the concrete semantics, using intervals as computable supersets of the real coefficients. We suppose the numbers used for the analysis have a mantissa of five digits in base 10, then the multiplication of a and b with the abstract semantics writes :

$$\begin{aligned} a \times^{\ell_3} b = {} & [c_{\mathbb{F}}, c_{\mathbb{F}}] + [0.070785, 0.070785]\varepsilon_{\ell_1} + [0.34171, 0.34172]\varepsilon_{\ell_2} \\ & + [0.0131, 0.0131]\varepsilon_{\ell_3} + [0.00003025, 0.00003025]\varepsilon_{hi} \ . \end{aligned}$$

The floating-point result is still the result $c_{\mathbb{F}}$ of the multiplication $a_{\mathbb{F}} \times b_{\mathbb{F}}$, rounded to the nearest floating-point number, with the precision of the floating-point number analyzed. This results simulates the floating-point execution.
The errors are computed using classical interval arithmetic, that is with outward rounding, to include the errors coming from the analysis which uses itself finite precision numbers. Using a higher precision for the computation of these error intervals allows to estimate them more tightly. Here, an extended precision to six digits would be enough to compute exactly the error, without the use of intervals. But it would be too costly to extend the precision for each additional operation. Moreover, some errors can not be represented by extended precision floating-point numbers, for example in some cases of divisions.

Now consider the same multiplication where the floating-point value of a is no longer a single value, but any possible value in an interval : for example we take $a' = [610, 630] + [0.055, 0.055]\varepsilon_{\ell_1}$. We get :

$$\begin{aligned} a' \times^{\ell_3} b = {} & [785.1, 810.8] + [0.070785, 0.070785]\varepsilon_{\ell_1} + [0.3355, 0.3465]\varepsilon_{\ell_2} \\ & + [-0.05, 0.05]\varepsilon_{\ell_3} + [0.00003025, 0.00003025]\varepsilon_{hi} \ . \end{aligned}$$

Indeed, $610 \times 1.287 = 785.07$, rounded to the nearest gives 785.1, and $630 \times 1.287 = 810.81$, rounded to the nearest gives 810.8. Thus the floating-point part of the result can be any floating-point value in the interval $[785.1, 810.8]$. The error coming from point ℓ_1 and the error of order higher than 1 are unchanged. The error coming from point ℓ_2 belongs to the result of the interval multiplication (with outward rounding) of $a'_{\mathbb{F}}$ and the error 0.00055, that is $[0.3355, 0.3465]$. The roundoff error introduced by the multiplication can only be bounded by the largest set of values which added to the floating-point result, do not affect its value in floating-point arithmetic, that is the interval $[-0.05, 0.05]$.

In the general case, we get the abstract semantics by interpreting the operation over error series with interval coefficients, using rounding to the nearest for the computation of the floating-point part, and outward rounding and possibly more precision for the propagation of the existing errors. For the division, we must compute an over-approximation of the sum of the terms of order higher or equal to two. For that, we note that the error committed by approximating

$(1+u)^{-1}$ by the first-order development $1-u$ is $g(u) = (1+u)^{-1}u^2$. And we can easily bound $g(u)$ for $u = (f^y)^{-1} \sum_{\ell \in \mathcal{L}} \omega_\ell^y$ by studying function g.

Most of the time, the new roundoff error introduced by an operation can only be bounded. Suppose the floating-point result of an operation is in the interval $[a, b]$, and note $r = max(|a|, |b|)$. The roundoff error due to this operation is bounded by $[-ulp(r)/2, ulp(r)/2]$, where $ulp(r)$ is the unit in the last place of r, that is the smallest number which, added to the floating-point number r, does affect its value. If the floating-point parts of the operands are intervals reduced to points ($a = b$), the error can be bounded more accurately using (1), by the difference of the floating-point result, and the interval result of the same operation achieved with outward rounding and the precision of the analysis.

3.2 Computations in Loops

When encountering a loop, the analyzer will try to produce an *invariant*, i.e. a property which holds true before or after some instruction in the body of the loop, regardless of the number of loops already executed. As an example, look at the program:

```
int i=1;
while (i<100)
  (1): i++;
(2):
```

suitably annotated with labels `(1)` (respectively `(2)`), locating the control point at the beginning of the body of the loop, just before `i++` takes place (respectively, after the loop). A correct invariant at `(1)` is `i` in $S = \{1, 2, \ldots, 99\}$, because each time the control flow goes through `(1)`, `i` takes its value in S. Notice that $S' = [0, 100]$ is also an invariant, but less precise. The most precise invariant at `(2)` is `i` equals 100.

If we represent the "effect" of one iteration of the loop on variables' values by a function f (its "semantics"), then calculating `(1)` amounts to finding the least fixed point of f, above some initial set of values X_0. Equivalently (when f is "continuous"), the invariant $i_{(1)}$ at `(1)` - only concerning variable `i` here - is given by Kleene's theorem:

$$i_{(1)} = X_0 \cup f(X_0) \cup f^2(X_0) \cup \ldots \cup f^n(X_0) \cup \ldots$$

This gives an immediate algorithm for computing the invariant, called the *fixed point iteration sequence*, in which we start with $i_{(1)}^0 = X_0 = [1, 1]$ and carry on by defining *(it):* $i_{(1)}^{n+1} = i_{(1)}^n \cup f\,(i_{(1)}^n)$, the limit of which being the least fixed point in question.

In our example, $f(S) = ([1, 1] \cup (S + [1, 1])) \cap]-\infty, 99]$. The iteration sequence is then $i_{(1)}^0 = [1, 1]$, $i_{(1)}^1 = [1, 2]$, ..., $i_{(1)}^j = [1, j+1]$ and finally, $i_{(1)}^{99} = i_{(1)}^{98} = [1, 99]$ (the fixed point). This algorithm is not very efficient in general. One may like to extrapolate the iteration sequence, by replacing the union operator in equation *(it)* by a so-called *widening* operator ∇. It can be defined axiomatically as an operator which always over-approximates the union, such that there is no infinite increasing sequence in such iterations. This ensures finite time response of a static analyzer in practice.

A very simple and classical widening operator on intervals of values is the one for which $[a,b]\nabla[c,d]$ is $[e,f]$ with

$$e = \begin{cases} a & \text{if } c \geq a \\ -\infty & \text{otherwise} \end{cases} \qquad f = \begin{cases} b & \text{if } b \geq d \\ \infty & \text{otherwise} \end{cases}$$

This operator extrapolates the max bound by ∞ if the max bound seems to increase from one iteration to the other (respectively, the min bound by $-\infty$ if the min bound seems to decrease from one iteration to the other). In our example, applying the widening operator in place of the union after step 1 of the iteration sequence, we get $i^2_{(1)} = [1,\infty[$ which is a correct invariant, although overapproximated, since $f(i^2_{(1)}) = [1,99] \subseteq [1,\infty[$. One more iteration gets us to the least fixed point indeed.

Static analysis is interesting for computing efficiently some properties over sets of executions. Consider the toy example

```
void f(int n) {
  float x = 2;
  for (i=0 ; i<n ; i++)
   (1): x = x/(n+1) + 1;  }
```

Static analysis allows to tell in two iterations, that for all possible value of $n \in [0,\infty]$, the value of x at point (1) in the loop belongs to $[1,2]$. Indeed, $x_0 = 2$; $x_1 = 2/(n+1) + 1 \cup 2 \in [1,2]$; $x_2 \in [1,2]$, the fixed point is reached.

The case of numerical computations in loops requires particular care : the classical fixed point iteration carried out without precaution will underline possibly infinite errors for most stable loops. We have had to design some special fixed point iteration strategies in order to get tighter estimations, but these are beyond the scope of this paper.

3.3 Other Semantics

The interpretation of the results for large programs can be facilitated by choosing different levels of error points (C lines, blocks of lines, functions, etc), and refining locally the result in the functions that have the most important errors. Grouping error points can also be used during the computation to reduce the memory and computation time of the analysis [6].

Other variations lead to "relational" analyses : an idea is to use the linear correlations between variables in order to reduce the over-estimation of errors, somehow like what is done in affine interval arithmetics [1]. Suppose there is one ε_i per node of the control flow graph of the program, a variable x can be written

$$x = f^x + \sum_{\ell \in L} t^x_\ell . \gamma_\ell \varepsilon_\ell + \omega^x_{os} \varepsilon_{os} \ , \tag{3}$$

where $f^x \in \mathbb{F}$ is the computed floating-point value, $\gamma_\ell \in \mathbb{R}$ is the error committed at point ℓ, and $t^x_\ell \in \mathbb{R}$ expresses the propagation of this error on variable x. When abstracting the coefficients γ_ℓ and t^x_ℓ by intervals, the linear correlations between variables are expressed in the t^x_ℓ, and allow some error balancing, which was not possible with only an interval error that lost a part of these correlations. The error γ_ℓ, which value is a priori unknown but can be bounded, is represented by an interval, but is seen as a formal variable that takes one particular value in this interval. And we can write for example the addition in the following way :

$$z = x +^{\ell_i} y = \uparrow_\circ (f^x + f^y) + \sum_{\ell \in L} (t^x_\ell + t^y_\ell).\gamma_\ell \varepsilon_\ell + (\omega^x_{os} + \omega^y_{os})\varepsilon_{os} + \downarrow_\circ (f^x + f^y)\varepsilon_{\ell_i} \; .$$

In this expression, the error γ_{ℓ_i} is $\downarrow_\circ (f^x + f^y)\varepsilon_{\ell_i}$, and, at point ℓ_i, the propagation coefficient is $t^z_{\ell_i} = 1$. Other variations using correlations between values and errors, for example by means of relative error, could also be used.

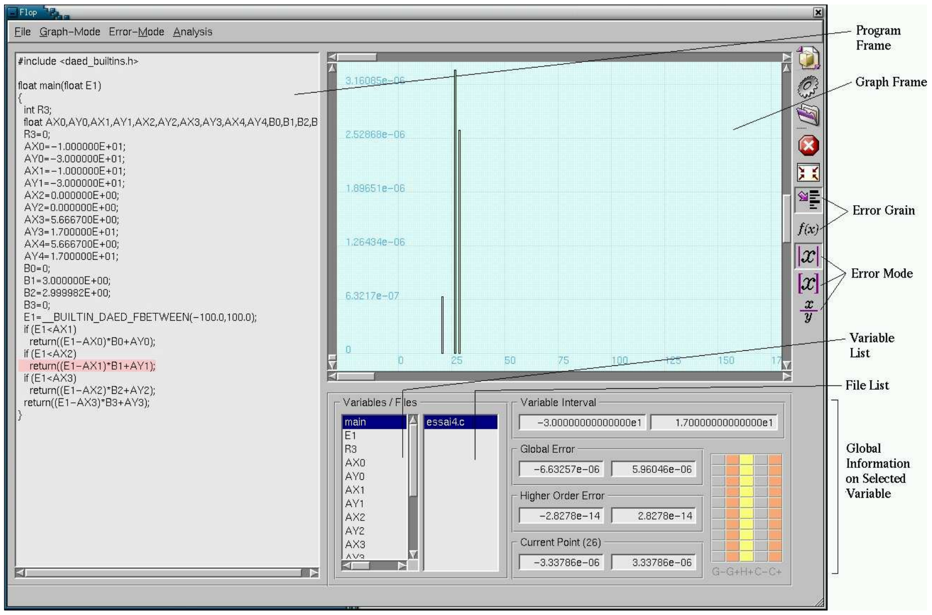
4 The Fluctuat Tool

A prototype [4] implements this abstract interpretation, for the analysis of C programs. The multi-precision library MPFR [5] (based on GMP) is used to compute tight bounds on the errors. As shown in Fig. 1, the main window of the analyzer displays the code of the program being analyzed, the list of variables in the program, and a graph representation of the error series related to the selected variable, at the last control point of the program. The operations are identified with their line number in the program, displayed on the X-axis. The bars indicate the maximum of the absolute values of the interval bounds. Clicking on an error bar makes the code frame emphasize the related program line and conversely.

In the example of Fig. 1, a program typical of an instrumentation software is being analyzed. It consists basically in an interpolation function with thresholds. One can see from the graph that the sources of imprecision for the return result of the function are (variable `main` selected): the floating-point approximation of constant B2 = 2.999982, the 2nd `return`, and the 3rd `return`, the last two being the more important. Using the assertion `__BUILTIN_DAED_FBETWEEN`, we imposed that `E1` is between -100 and 100. Then the control flow can go through all `return`. But in the first and last `return`, there is no imprecision committed. Thus, to improve the result, we can improve the computation of the 2nd and 3rd `return`. One way is to use `double E1` to improve the accuracy of the subtractions.

5 Conclusion and Future Work

We have presented some ideas about what static analysis can do for programs using floating-point numbers. A part of the work consists in modeling the results and the losses of accuracy using finite precision computations. The model used, looks like affine interval arithmetics, but is used with a very different intention : the coefficients have a meaning (floating-point value, influence of a part of the



```
#include <daed_builtins.h>

float main(float E1)
{
  int R3;
  float AX0,AY0,AX1,AY1,AX2,AY2,AX3,AY3,AX4,AY4,B0,B1,B2,B
  R3=0;
  AX0=-1.000000E+01;
  AY0=-3.000000E+01;
  AX1=-1.000000E+01;
  AY1=-3.000000E+01;
  AX2=0.000000E+00;
  AY2=0.000000E+00;
  AX3=5.666700E+00;
  AY3=1.700000E+01;
  AX4=5.666700E+00;
  AY4=1.700000E+01;
  B0=0;
  B1=3.000000E+00;
  B2=2.999982E+00;
  B3=0;
  E1=__BUILTIN_DAED_FBETWEEN(-100.0,100.0);
  if (E1<AX1)
    return((E1-AX0)*B0+AY0);
  if (E1<AX2)
    return((E1-AX1)*B1+AY1);
  if (E1<AX3)
    return((E1-AX2)*B2+AY2);
  return((E1-AX3)*B3+AY3);
}
```

Variable Interval: -3.00000000000000e1 1.70000000000000e1

Global Error: -6.63257e-06 5.96046e-06

Higher Order Error: -2.8278e-14 2.8278e-14

Current Point (26): -3.33786e-06 3.33786e-06

Fig. 1. Main window of the analyzer.

program on the global error), and are not used only to improve the precision like in affine arithmetics. Some work can still be done to improve the accuracy of this modelisation. But a consequent and difficult part is related to static analysis : efficient algorithms for fixed point computations in loops must be designed, and implementing a static analyzer for real problems is a heavy work. Our first concern is the analysis of instrumentation software, but we hope to be able to go slowly towards numerically more complex programs.

References

1. J. L. D. Comba and J. Stolfi. Affine arithmetic and its applications to computer graphics. In SIBGRAPI'93, Recife, PE (Brazil), October 20-22, 1993.
2. P. Cousot and R. Cousot. Abstract interpretation frameworks. Journal of Logic and Symbolic Computation, 2(4):511–547, 1992.
3. E. Goubault. Static analyses of the precision of floating-point operations. In Static Analysis Symposium, SAS'01, number 2126 in LNCS, Springer-Verlag, 2001.
4. E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations : a simple abstract interpreter. In ESOP'02, LNCS, Springer 2002.
5. G. Hanrot, V. Lefevre, F. Rouillier, P. Zimmermann. MPFR library. INRIA, 2001.
6. M. Martel. Propagation of roundoff errors in finite precision computations : a semantics approach. In ESOP'02, number 2305 in LNCS, Springer-Verlag, 2002.
7. M. Martel. Static Analysis of the Numerical Stability of Loops. In SAS'02, number 2477 in LNCS, Springer-Verlag, 2002.
8. R. E. Moore. Interval Analysis. Prentice-Hall, Englewood Cliffs, NJ, 1963.