Data-Types Optimization for Floating-Point Formats by Program Transformation*

Nasrine Damouche
University of Perpignan
LAboratory of Mathematics and PhysicS
52 Avenue Paul Alduy
66860 Perpignan, France
Email: nasrine.damouche@univ-perp.fr

Matthieu Martel
University of Perpignan
LAboratory of Mathematics and PhysicS
52 Avenue Paul Alduy
66860 Perpignan, France
Email: matthieu.martel@univ-perp.fr

Alexandre Chapoutot
U2IS, ENSTA ParisTech
Paris-Saclay University
828 bd des Maréchaux
91762 Palaiseau, France
alexandre.chapoutot@ensta-paristech.fr

***Abstract*—In floating-point arithmetic, a desirable property of computations is to be accurate, since in many industrial context small or large perturbations due to round-off errors may cause considerable damages. To cope with this matter of fact, we have developed a tool which corrects these errors by automatically transforming programs in a source to source manner. Our transformation, relying on static analysis by abstract abstraction, concerns pieces of code with assignments, conditionals and loops. By transforming programs, we can significantly optimize the numerical accuracy of computations by minimizing the error relatively to the exact result. An interesting side-effect of our technique is that more accurate computations may make it possible to use smaller data-types. In this article, we show that our transformed programs, executed in single precision, may compete with not transformed codes executed in double precision.**

I. INTRODUCTION

The floating-point numbers described by IEEE754 Standard [1], [16] are more and more used in many industrial applications, including critical embedded software. In these computations, a recurrent problem appears because of round-off errors that reduce the accuracy of the results. This perturbation becomes particularly crucial when accumulated errors cause damages whose gravity varies depending on the context of the application. Obviously, embedded software is not the only applicative domain concerned by these accuracy problems which are ubiquitous in all numerical scientific computations. In the light of this problem, we correct these errors by automatically transforming programs in a source to source manner. Basically, we transform not only arithmetic expressions but also pieces of code containing assignments, conditionals, loops and sequences of commands. Our transformation significantly improves the numerical accuracy of the program considered. To optimize programs, we generate large arithmetic expressions corresponding to the computations of the original program and further, we consider many expressions mathematically equivalent to the original ones in order to, finally, choose a more accurate among them in polynomial time.

There exists several methods for validating [3], [7], [8], [10], [20] and improving [12], [17] the accuracy of numerical codes in order to avoid failures. Here, as in our previous work, we rely on static analysis by abstract interpretation [4] to compute variable ranges and error bounds. We use a set of transformation rules for arithmetic expressions and commands [6]. These rules, which are applied in a deterministic order, allow one to obtain a more accurate code among all these which are considered. We have shown that the numerical accuracy of our case study programs is significantly improved. In most cases it is about of 20%.

In this paper, we propose a new experiment, that compares the optimized programs obtained by our tool and executed in single precision to the initial programs executed in both single and double precision. Our main contribution is to show that the transformed programs in single precision are close to the original programs in double precision. This offers to the programmer the possibility to degrade to the single precision data-type without loosing much information. To ensure that our tool is useful in practice, we write the source and transformed program in `C` and compile them by using GCC Compiler.

This article is organized as follows. We first review in Section II the basics of our transformation method based on floating-point arithmetic. We explain briefly how to transform arithmetic expressions. We then introduce in Section III the different transformation rules that allow us to automatically transform programs. We present in Section IV our main contribution which demonstrates the interest of the comparison between single, double precision and optimized codes. We illustrate this with some experiments concerning the computation of the integral of a polynomial using Simpson's Rule [2]. Related work is discussed in Section V. Finally, we give some concluding remarks and perspectives in Section VI.

II. ANALYSIS AND TRANSFORMATION OF EXPRESSIONS

In this section we introduce the background needed to understand our approach to improve the numerical accuracy of programs. We recall the key notions on our way of computing the rounding errors on floating-point arithmetic expressions. Next, we briefly explain how to transform arithmetic expressions using an intermediary representation called APEGs.

A. Static Analysis of Arithmetic Expressions

Floating-point numbers are used to represent real numbers. Because of their finite representation, round-off errors arise

*This work was supported by the ANR Project ANR-12-INSE-0007 ”CAFEIN”.

978-1-5090-2188-8/16/$31.00 ©2016 IEEE

during the computations which may cause damages in critical contexts. IEEE754 Standard formalizes a binary floating-point number as a triplet of sign (s $\in \{0, 1\}$), significand and exponent. We consider that a number x is written:

$$x = s \cdot (d_0.d_1 \dots d_{p-1}) \cdot b^e = s \cdot m \cdot b^{e-p+1} \ , \qquad (1)$$

where, s is the sign $\in \{-1, 1\}$, m is the mantissa, $m = d_0.d_1 \dots d_{p-1}$ with digits $0 \le d_i < b$, $0 \le i \le p-1$, p is the precision, and e is the exponent $\in [e_{min}, e_{max}]$.

A floating-point number x is *normalized* whenever $d_0 \neq 0$. Normalization avoids multiple representations of the same number. IEEE754 Standard specifies some particular values for p, e_{min} and e_{max} which are summarized in Figure 1 as well as *denormalized numbers* which are floating-point numbers with $d_0 = d_1 = \ldots = d_k = 0$, $k < p-1$ and $e = e_{min}$. Denormalized numbers make underflow gradual [9]. Finally, the following special values also are defined:

- `NaN` (Not a Number) result of an invalid operation,
- the values $\pm\infty$ corresponding to overflows,
- the values $+0$ and -0 (signed zeros).

Format	Name	p	e bits	e_{min}	e_{max}
Binary16	Half precision	11	5	-14	$+15$
Binary32	Single precision	24	8	-126	$+127$
Binary64	Double precision	53	11	-1122	$+1223$
Binary128	Quadratic precision	113	15	-16382	$+16383$

Fig. 1. Basic IEEE754 formats.

IEEE754 Standard defines four rounding modes for elementary operations over floating-point numbers. These modes are towards $-\infty$, towards $+\infty$, towards zero and to the nearest respectively denoted by $\uparrow_{+\infty}$, $\uparrow_{-\infty}$, $\uparrow_0$ and $\uparrow_\sim$. The semantics of the elementary operations specified by IEEE754 Standard is given by Equation (2).

$$x \circledast_r y = \uparrow_r (x * y) \ , \text{ with } \uparrow_r : \mathbb{R} \to \mathbb{F} \qquad (2)$$

where a floating-point operation, denoted by $\circledast_r \in \{+, -, \times, \div\}$, is computed using the rounding mode r and $*$ denotes an exact operation. Obviously, the results of the computations are not exact because of the round-off errors. This is why, we use also the function $\downarrow_r : \mathbb{R} \to \mathbb{R}$ that returns the round-off errors. We have

$$\downarrow_r (x) = x - \uparrow_r (x) \ . \qquad (3)$$

In order to compute the errors during the evaluation of arithmetic expressions [14], we use values which are pairs $(x, \mu) \in \mathbb{F} \times \mathbb{R} = \mathbb{E}$ where x denotes the floating point number used by the machine and μ denotes the exact error attached to $\mathbb{F}$, i.e., the exact difference between the real and floating-point numbers as defined in Equation (3). For example, the real number $\frac{1}{3}$ is represented by the value $v = \left(\uparrow_\sim \left(\frac{1}{3}\right), \downarrow_\sim \left(\frac{1}{3}\right)\right) = (0.333333, (\frac{1}{3} - 0.333333))$. The semantics of the elementary operations on $\mathbb{E}$ is defined in [14].

Our tool uses an abstract semantics [4] based on $\mathbb{E}$. The abstract values are represented by a pair of intervals. The first interval contains the range of the floating-point values of the program and the second one contains the range of the errors obtained by subtracting the floating-point values from the exact ones. In the abstract value denoted by $(x^\sharp, \mu^\sharp) \in \mathbb{E}^\sharp$, we have $x^\sharp$ the interval corresponding to the range of the values and $\mu^\sharp$ the interval of errors on $x^\sharp$. This value abstracts a set of concrete values $\{(x, \mu) \ : \ x \in x^\sharp \text{ and } \mu \in \mu^\sharp\}$ by intervals in a component-wise way. We now introduce the semantics of arithmetic expressions on $\mathbb{E}^\sharp$. We approximate an interval $x^\sharp$ with real bounds by an interval based on floating-point bounds, denoted by $\uparrow^\sharp (x^\sharp)$. Here bounds are rounded to the nearest, see Equation (4).

$$\uparrow^\sharp ([\underline{x}, \overline{x}]) = [\uparrow (\underline{x}), \uparrow (\overline{x})] \ . \qquad (4)$$

We denote by $\downarrow^\sharp$ the function that abstracts the concrete function $\downarrow$. It over-approximates the set of exact values of the error $\downarrow (x) = x - \uparrow (x)$. Every error associated to $x \in [\underline{x}, \overline{x}]$ is included in $\downarrow^\sharp ([\underline{x}, \overline{x}])$. We also have for a rounding mode to the nearest

$$\downarrow^\sharp ([\underline{x}, \overline{x}]) = [-y, y] \quad \text{with} \quad y = \frac{1}{2}\text{ulp}\big(\max(|\underline{x}|, |\overline{x}|)\big) \ . \qquad (5)$$

Formally, the *unit in the last place*, denoted by ulp(x), consists of the weight of the least significant digit of the floating-point number x. Equations (6) to (7) give the semantics of the addition and multiplication over $\mathbb{E}^\sharp$, for other operations see [14]. If we sum two numbers, we must add errors on the operands to the error produced by the round-off of the result. When multiplying two numbers, the semantics is given by the development of $(x_1^\sharp + \mu_1^\sharp) \times (x_2^\sharp + \mu_2^\sharp)$.

$$(x_1^\sharp, \mu_1^\sharp) + (x_2^\sharp, \mu_2^\sharp) = \Big(\uparrow^\sharp (x_1^\sharp + x_2^\sharp), \mu_1^\sharp + \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp + x_2^\sharp) \Big) \ , \qquad (6)$$

$$\begin{gathered} (x_1^\sharp, \mu_1^\sharp) \times (x_2^\sharp, \mu_2^\sharp) \\ = \\ (\uparrow^\sharp (x_1^\sharp \times x_2^\sharp), x_2^\sharp \times \mu_1^\sharp + x_1^\sharp \times \mu_2^\sharp + \mu_1^\sharp \times \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp \times x_2^\sharp)) \ . \end{gathered} \qquad (7)$$

B. Accuracy Improvement

The work in [12] concerns the definition of a new intermediary representation called APEG for Abstract Program Expression Graph. This approach represents in a polynomial size an exponential number of equivalent arithmetic expressions. Because APEGs hold in abstraction boxes many equivalent expressions up to associativity and commutativity, hence, it prevents the combinatorial problem. A box containing n operands can represent up to $1 \times 3 \times 5 ... \times (2n-3)$ possible formulas. In order to build large APEGs, two algorithms are used (*propagation* and *expansion* algorithm). The first one searches recursively in the APEG where a symmetric binary operator is repeated and introduces abstraction boxes. Then, the second algorithm finds a homogeneous part and inserts a polynomial number of boxes. In order to add new shapes of expressions in an APEG, one propagates recursively subtractions and divisions into the concerned operands, propagate products, and factorizing common factors. Finally, an accurate formula is searched among all the equivalent formulas represented in an APEG using the abstract semantics of Section II-A. The APEGs are an extension of the Equivalence Program Expression Graphs (EPEGs) introduced by R. Tate *et al.* [21]. An APEG is defined inductively as follows:

1) A constant cst or an identifier id is an APEG,
2) An expression $p_1 * p_2$ is an APEG, where p_1 and p_2 are APEGs and $*$ is a binary operator among $\{+, -, \times, \div\}$,

3) A box $\boxed{*(p_1, \dots, p_n)}$ is an APEG, where $* \in \{+, \times\}$ is a commutative and associative operator and the $p_{i, 1 \leq i \leq n}$, are APEGs,
4) A non-empty set $\{p_1, \dots, p_n\}$ of APEGs consists in an APEG where $p_{i, 1 \leq i \leq n}$, is not a set of APEGs itself. We call the set $\{p_1, \dots, p_n\}$ the equivalence class.

An example of APEG is given in Figure 2, it represents all the following expressions:

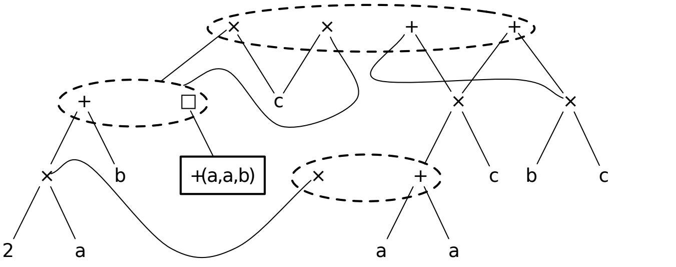


Fig. 2. APEG for the expression $e = \big((a + a) + b\big) \times c$.

$$\mathcal{A}(p) = \left\{ \begin{array}{c} \big((a+a)+b\big) \times c,\ \big((a+b)+a\big) \times c, \\ \big((b+a)+a\big) \times c,\ \big((2 \times a)+b\big) \times c, \\ c \times \big((a+a)+b\big),\ c \times \big((a+b)+a\big), \\ c \times \big((b+a)+a\big),\ c \times \big((2 \times a)+b\big), \\ (a+a) \times c + b \times c,\ (2 \times a) \times c + b \times c, \\ b \times c + (a+a) \times c,\ b \times c + (2 \times a) \times c \end{array} \right\} . \tag{8}$$

R. Tate *et al.* in their article [21], use rewriting rules to extend the structure up to saturation. In our context, such rules would consist of performing some pattern matching in an existing APEG p and then adding new nodes in p, once a pattern has been recognized.

III. Transformation of Commands

In this section, we introduce the transformation rules, implemented in our tool and used to improve the numerical accuracy of programs. The syntax of commands is given by:

$$\begin{array}{rl} \text{Com} \ni c ::= & id = e \mid c_1 ; c_2 \mid \text{if}_\Phi\ e \text{ then } c_1 \text{ else } c_2 \\ & \mid \text{while}_\Phi\ e \text{ do } c \mid \text{nop} \ , \end{array} \tag{9}$$

where e is an expression in $Expr$ made of arithmetic operations and comparison. The principle of the transformation of commands relies on a set of hypotheses:

- Programs are defined by a tuple
$$\langle c, \delta, C, \beta \rangle \to_\nu \langle c', \delta, C, \beta \rangle \quad :$$
  - c is the program at optimizing,
  - δ the formal environment that maps variables to expressions (δ: $\mathcal{V} \mapsto Expr$),
  - C is the context, i.e., the program enclosing the command to transform,
  - β is the black list that contains variables that must not be removed from the program,
  - ν is the reference variable at optimizing,
- Programs are written in SSA form (single static assignments),
- Transformation rules are applied deterministically,
- The best program, the most accurate, is obtained by comparing the reference variable of the original and transformed program.

Here, we survey briefly the different kinds of transformation rules. We refer the interested reader to [6] to see the details of these transformation rules based on the syntax and semantics. We start with the assignments. We have two rules, the first consists in removing the assignment from the program and saving it in the memory δ if the conditions below are verified:

a) The variables of the expression e does not appear in δ,
b) The variable v does not belong to the black list β,
c) The variable v is different of the reference variable ν.

Otherwise, we build a large expression by substituting in it the formal expressions memorized previously by the first rule in δ. Remark that by inlining expressions in variables when transforming programs, we create large formulas. In our implementation, in order to facilitate their manipulation, we slice these formulas, at a defined level of the syntactic tree, in several sub-expressions, and we assign them to intermediary variables. Finally, we inject these new assignments into the main program.

Example 3.1: To explain the use of the transformation rules of assignments, let us consider Equation (10) in which three variables x, y and z are assigned. In this example, ν consists of the variable z that we aim at optimizing, and $a = 0.1$, $b = 0.01$, $c = 0.001$ and $d = 0.0001$ are constants.

$$\begin{array}{rl} & \langle \mathtt{x = a + b; y = c + d; z = x + y},\ \delta,\ [\,],\ \{\mathtt{z}\} \rangle \\ \Longrightarrow_\nu & \langle \text{nop}; \mathtt{y = c + d; z = x + y},\ \delta' = \delta[\mathtt{x} \mapsto \mathtt{a + b}],\ [\,],\ \{\mathtt{z}\} \rangle \\ \Longrightarrow_\nu & \langle \mathtt{y = c + d; z = x + y},\ \delta' = \delta[\mathtt{x} \mapsto \mathtt{a + b}],\ [\,],\ \{\mathtt{z}\} \rangle \\ \Longrightarrow_\nu & \langle \text{nop}; \mathtt{z = x + y},\ \delta'' = \delta'[\mathtt{y} \mapsto \mathtt{c + d}],\ [\,],\ \{\mathtt{z}\} \rangle \\ \Longrightarrow_\nu & \langle \mathtt{z = x + y},\ \delta'' = \delta'[\mathtt{y} \mapsto \mathtt{c + d}],\ [\,],\ \{\mathtt{z}\} \rangle \\ \Longrightarrow_\nu & \langle \mathtt{z = ((d + c) + b) + a}, \delta'',\ [\,], \{\mathtt{z}\} \rangle \end{array} \tag{10}$$

In Equation (10), initially, the environment δ is empty, the black list contains z and the reference variable ν at optimizing is z. If we apply the first rule of assignment, we may remove the variable x and memorize it in δ. So, the line corresponding to the variable discarded is replaced by `nop` and the new environment is $\delta = [x \mapsto a + b]$. We then repeat the same process, we remove the variable y and saved it in δ. Next, we apply a rule for sequences which discards the `nop` statement. For the last step, we may not discard z because the condition is not satisfied ($z = \nu$). Then, we substitute x and y by their value in δ and we transform the expression. ■

Another kind of transformation rules is for the sequences of commands. If one member of the sequence is `nop`, then we transform only the other member, else, we transform both of them. Our implementation transforms also conditionals. If the condition is statically known, then we keep just the evaluated branch and we transform it, else, we transform both branches of the conditional. In some cases, we deal with undefined variables because they have been discarded from the program and saved in the environment δ as indicated in the first transformation rule for assignments. We then re-inject them into the program and we do the necessary transformations. The next transformation rules concern the `while` loop. We transform the body of the loop ensuring that the variables of the condition do not belong to the environment δ. Otherwise, we have to re-insert the variables memorized in the environment into the program as doing for the last rule of conditionals.

At the end of this section, we deal with complexity considerations. As underlined previously, only one rule may be selected at each step of the transformation of a program p. Consequently, the transformation would be linear in the size n, i.e., the number of lines, of p if we would not re-inject assignments. However, a given assignment cannot be removed twice, so the transformation is quadratic. Finally, the entire transformation of a program p is repeated until nothing changes, that is at most n times. Hence, the global complexity for a program transformation of size n is $\mathcal{O}(n^3)$.

IV. Experiments

We have implemented a tool based on the rules of Section III to improve the numerical accuracy of the floating-point computations. This tool finds a more accurate program among all those equivalent. In this section, we emphasize the efficiency of our implementation in terms of improving the data-types used by the programs. More precisely, we show that by using our tool, we approximate the results to be ever accurate and close to the results obtained under the double precision while using single precision.

In the light of these ideas, let us confirm our claims by means of a small examples such as Simpson's method. We start by briefly describing what our program computes, and then we give their listing before and after being optimized with our tool. Their accuracy is then discussed. Note that our programs are written in SSA form [5] to avoid any problem dealing with the reading and writing variables.

A. Problem Description and Simpson's Method

Simpson's method consists in a technique for numerical integration that approximates the computation of $\displaystyle\int_a^b f(x)\,\mathrm{d}x$. It uses a second order approximation of the function f by a quadratic polynomial P that takes three abscissa points a, b and m with $m = (a+b)/2$. When integrating the polynomial, we approximate the integral of f on the interval $[x, x+h]$ ($h \in \mathbb{R}$ small) with the integral of P on the same interval. Formally, the smaller the interval is, the better the integral approximation is. Consequently, we divide the interval $[a, b]$ into subintervals $[a, a+h]$, $[a+h, a+2h]$, $\ldots$ and then we sum the obtained values for each interval. We write:

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{h}{3}\Big[f(x_0) + 2\sum_{j=1}^{\frac{n}{2}-1} f(x_{2j}) + 4\sum_{j=1}^{\frac{n}{2}} f(x_{2j-1}) + f(x_n)\Big] \quad (11)$$

where

- n is the number of subintervals of [a,b] with n is even,
- $h = (b-a)/n$ is the length of the subintervals,
- $x_i = a + i \times h$ for $i = 0, 1, \ldots, n-1, n$.

In our case, we have chosen the polynomial given in Equation (12). It is well-known by the specialists of floating-point arithmetic that the developed form of the polynomial evaluates very poorly close to a multiple root. This motivates our choice of the function f below for our experiments.

$$\begin{array}{rcl} f & = & (x-2.)^7 \\ & = & x^7 - 14. \times x^6 + 84. \times x^5 - 280. \times x^4 \\ & & +\, 560. \times x^3 - 672. \times x^2 + 448. \times x - 128. \end{array} \quad . \quad (12)$$

The listing corresponding of the implementation of the Simpson's method is described on Figure 3.

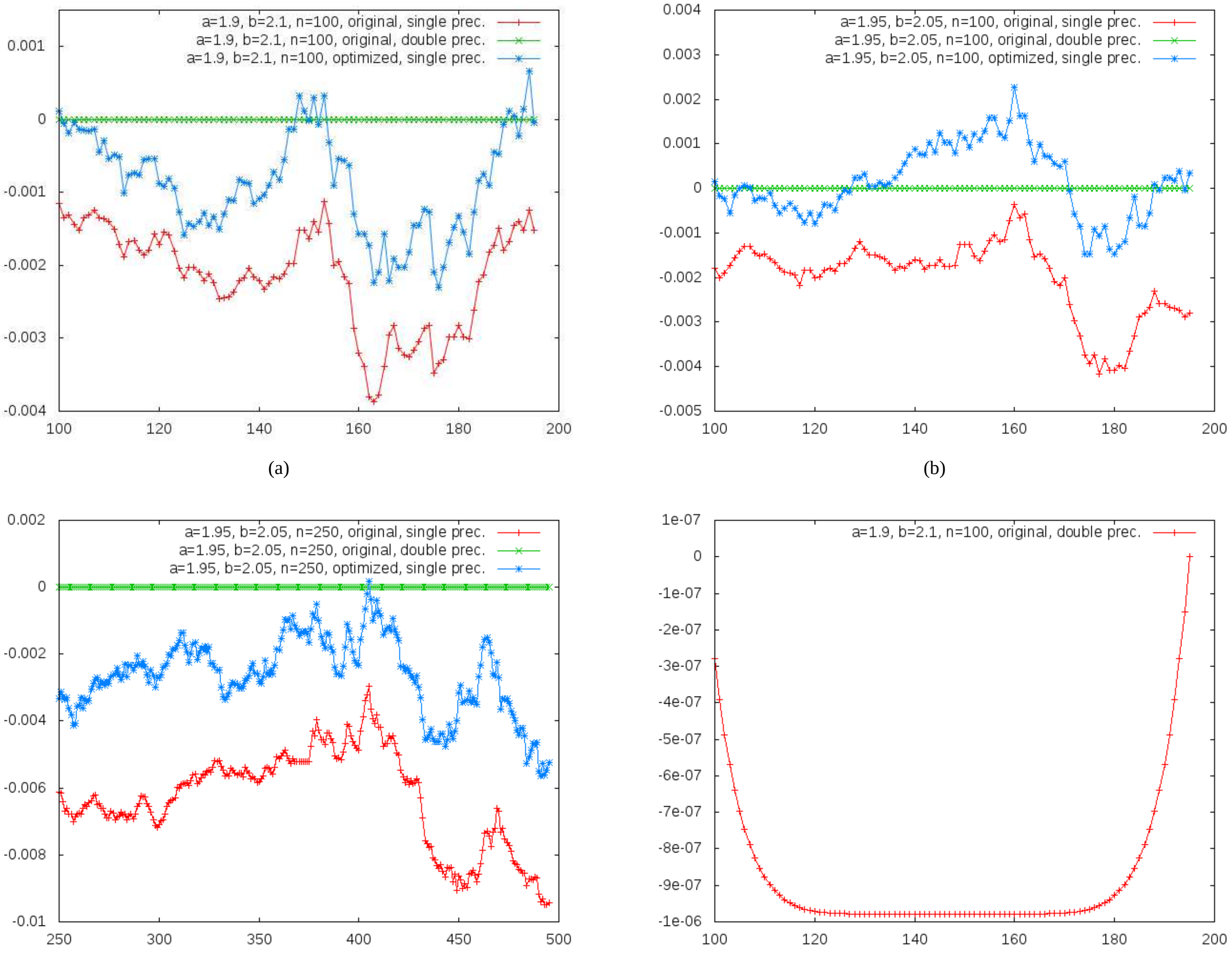
```
int main() {
 a = 1.9; b = 2.1; n = 100.0; i = 1.0; x = a; h = (b - a)/n;
 f = ((x*x*x*x*x*x*x) - 14.0 * (x*x*x*x*x*x) + 84.0 * (x*x*x
   * x*x) - 280.0 * (x*x*x*x) + 560.0 * (x*x*x) - 672.0 * (x
   * x) + 448.0 * x - 128.0);
 x = b ;
 g = ((x*x*x*x*x*x*x) - 14.0 * (x*x*x*x*x*x) + 84.0 * (x*x*x
   * x*x) - 280.0 * (x*x*x*x) + 560.0 * (x*x*x) - 672.0 * (x
   * x) + 448.0 * x - 128.0);
 s = f + g;
 while (i < n) {
  x = a + (i * h);
  f = ((x*x*x*x*x*x*x) - 14.0 * (x*x*x*x*x*x) + 84.0 * (x*x
    * x*x*x) - 280.0 * (x*x*x*x) + 560.0 * (x*x*x) - 672.0
    * (x*x) + 448.0 * x - 128.0);
  s = s + 4.0 * f;
  i = i + 1.0;
 };
 i = 2.0 ;
 while (i < n-1) {
  x = a + (i * h) ;
  f = ((x*x*x*x*x*x*x) - 14.0 * (x*x*x*x*x*x) + 84.0 * (x*x
    * x*x*x) - 280.0 * (x*x*x*x) + 560.0 * (x*x*x) - 672.0
    * (x*x) + 448.0 * x - 128.0);
  s = s + 2.0 * f;
  i = i + 1.0;
 };
 s = s * (h / 3.0);
}
```

Fig. 3. Listing of the initial Simpson's method.

```
int main() {
 TMP_3 = 6.859 ;
 TMP_1 = ((((((TMP_3 * 1.9) * 1.9) * 1.9) * 1.9) - (14.
       * (((TMP_3 * 1.9) * 1.9) * 1.9))) + (84. * ((6.859
       * 1.9) * 1.9)));
 TMP_2 = (1.9 * (280. * TMP_3));
 TMP_15 = 9.261000000000001;
 TMP_13 = ((((((TMP_15 * 2.1) * 2.1) * 2.1) * 2.1) - (14.
        * (((TMP_15 * 2.1) * 2.1) * 2.1))) + (84.
        * ((9.261000000000001 * 2.1) * 2.1)));
 TMP_14 = (2.1 * (280. * TMP_15));
 TMP_27 = 3.61;
 TMP_25 = ((((((TMP_27 * 1.9) * 1.9) * 1.9) * 1.9) * 1.9)
        - (14. * ((((TMP_27 * 1.9) * 1.9) * 1.9) * 1.9)));
 TMP_26 = (1.9 * (159.599999999999994 * TMP_3));
 TMP_32 = (TMP_3 * 1.9);
 i = 1.;
 s = ((((((TMP_1 - TMP_2) + (560. * TMP_3))
   - 2425.920000000000073) + 851.199999999999932) - 128.)
   + (((((TMP_13 - TMP_14) + (560. * TMP_15))
   - 2963.520000000000437) + 940.800000000000068) - 128.));
 f = ((((((TMP_25 + TMP_26) - (280. * TMP_32)) + (560.
   * (TMP_27 * 1.9))) - (672. * TMP_27))
   + 851.199999999999932) - 128.);
 x = 2.1;
 while (i < 100.) {
  x = (1.9 + (i * 0.002));
  TMP_37 = (x * x);
  TMP_35 = ((((((TMP_37 * x) * x) * x) * x) * x) - (14.
         * ((((TMP_37 * x) * x) * x) * x)));
  TMP_36 = (84. * (((TMP_37 * x) * x) * x));
  TMP_42 = (x * (TMP_37 * x));
  f = ((((((TMP_35 + TMP_36) - (280. * TMP_42)) + (560.
    * (TMP_37 * x)))-(672. * TMP_37)) + (448. * x)) - 128.);
  s = (s + (4. * f));
  i = (i + 1.);
 } ;  [ ... ]
 s = (0.000666666666667 * s) ;
}
```

Fig. 4. Listing of the transformed Simpson method.

When given the initial program described in Figure 3 to our tool, it improves its numerical accuracy by up to 99% depending on the entries. The transformed program is given in Figure 4. As detailed in Section III, our tool has:



(a)

(b)

(c)

(d)

Fig. 5. Simulation results of the Simpson's method with single, double precision and optimized program using our tool. The values of x and y axes correspond respectively to the value of n and s in Equation 13.

- created large expressions,
- transformed them into more accurate expressions,
- performed partial evaluation of the expressions,
- split the transformed expressions,
- assigned the transformed expressions to TMP variables.

This process has been applied inside and outside the loop.

B. Experimental results

The experimental results described hereafter compare the numerical accuracy of programs using single and double precision with a program transformed with our tool and running single precision only. Note that the codes presented in this article are written in `C`, compiled with the `GCC` compiler version `4.2.1` and executed by an `Intel Core i7` processor under `Ubuntu 15.04`. In addition, programs are compiled with the optimization level $-o0$ to avoid any optimization done by the compiler and additionally, we enforce the copy in the memory at each execution step by declaring all the variables as `volatile` (this avoids that values are kept in registers using more than 64 bits).

The results observed on Figure 5 when executing the initial program in single and double precision and the transformed program in single precision, demonstrate that our approach succeeds well to improve the accuracy. If we interest in the result of computations around the multiple root 2.0, we can see that the behavior of the optimized code is far closer to the original program executed with a double precision than the single precision original program. This shows that single precision may suffices in many contexts.

In Figure 5, one can see the difference, for different values of the step $h > 0$, in the computation of

$$s = \int_{a}^{a+nh} f(x)\,\mathrm{d}x, \quad 0 \le n < \frac{b-a}{h} \tag{13}$$

between the original program with both single and double precision and the transformed program (in single precision) in terms of numerical accuracy of computations. Obviously, the accuracy of the computations of the polynomial $f(x)$ depends on the values of x. The more the value of x is close to the multiple root, the worst the result is. For this reason, we make the interval $[a, b]$ vary by choosing $[a, b] \in \{[1.9, 2.1], [1.95, 2.05], [1.95, 2.05]\}$ and by choosing to split them in $n = 100$, $n = 100$ and $n = 250$ slices respectively for the application of Simpson's rule. Concerning the results obtained, our tool states that the percentage of the optimization computed by the abstract semantics of Section II-A is up to 99.39%. This means that the bound (obtained by the technique of Section II-A) on the numerical error of the computed values of the polynomial at any iteration is reduced by 99.39%.

Curve (d) of Figure 5 displays the function $f(x)$ at points $n = 1.9{+}0.02{\times}i$, $100 \leq i \leq 200$. Next, if we take for example Curve (b) of Figure 5, we observe that our implementation is as accurate as the initial program in double precision for many values of n since it gives results very close to the double precision while working in single precision. Note that, for the $x - axis$ of Figure 5, we have chosen to observe only the interesting interval of n which is around the multiple root 2.0.

V. Related work

Other research work tries to optimize the numerical datatypes or to increase the precision of computations. Duralova and Kuncak use an SMT solver to determine the minimal datatype needed to reach a certain accuracy [7]. For the fixed-point arithmetic [18], an alternative to the floating-point arithmetic, many approaches have been proposed to optimize the format of the numbers [13], [15]. Another kind of work aims at increasing the accuracy of computations by means of additional calculations which significantly slow down the applications. Double-double floating-point numbers emulate by software numbers with twice the precision of the double hardware format [11]. Finally, compensation techniques use error-free transformations to capture the exact errors of floating-point computations in order to re-inject the accumulated error in the result [19], [22].

VI. Conclusion

In this article, we have shown the usefulness of our implementation to improve the accuracy of programs. This allows one to work in a lower precision and obtain results close to the higher precision when transforming programs using our tool. Our examples compare the result of transformed program with the initial programs executed in single and double precision. We believe that this approach is very promising according to the different experiments results obtained.

Another research direction consists in the interprocedural programs transformation, i.e., we aim at generalizing our techniques to cover other kinds of programming language patterns like pointers, arrays and, specially functions.

We hope also in a future work to optimize several reference variables simultaneously. One difficulty is that the optimization of one variable may decrease the accuracy of other variables. Compromises have to be done. Another perspective consists at studying the impact of the accuracy optimization on the convergence time of distributed systems and also handle with the important issue that concerns the reproducibility of the results: different runs of the same application yield different results due to the variations in the order of evaluation of the mathematical expression. It will be very interesting to study how our technique could improve reproducibility.

References

[1] ANSI/IEEE. *IEEE Standard for Binary Floating-point Arithmetic*, std 754-2008 edition, 2008.

[2] K.-E. Atkinson. *An Introduction to Numerical Analysis*. Second Edition, 1988.

[3] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis by abstract interpretation of embedded critical software. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, 2011.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*, pages 238–252. ACM, 1977.

[5] R. Cytron and R. Gershbein. Efficient accomodation of may-alias information in SSA form. In *PLDI'93*, pages 36–45. ACM, 1993.

[6] N. Damouche, M. Martel, and A. Chapoutot. Intra-procedural optimization of the numerical accuracy of programs. In *FMICS'15*, volume 9128 of *LNCS*, pages 31–46. Springer, 2015.

[7] E. Darulova and V. Kuncak. Sound compilation of reals. In *POPL'14*, pages 235–248. ACM, 2014.

[8] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *FMICS'09*, pages 53–69, 2009.

[9] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[10] E. Goubault. Static analysis by abstract interpretation of numerical programs and systems, and FLUCTUAT. In *SAS'13*, volume 7935 of *LNCS*, pages 1–3. Springer, 2013.

[11] Y. Hida, X.-S. Li, and D.-H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *ARITH-15*, pages 155–162. IEEE Computer Society, 2001.

[12] A. Ioualalen and M. Martel. A new abstract domain for the representation of mathematically equivalent expressions. In *SAS'12*, volume 7460 of *LNCS*, pages 75–93. Springer, 2012.

[13] J.-L. Jerez, G.-A. Constantinides, and E.-C. Kerrigan. A low complexity scaling method for the lanczos kernel in fixed-point arithmetic. *IEEE Trans. Computers*, 64(2):303–315, 2015.

[14] M. Martel. Semantics of roundoff error propagation in finite precision calculations. *Higher-Order and Symbolic Comput.*, 19(1):7–30, 2006.

[15] D. Menard, N. Hervé, O. Sentieys, and H. Nguyen. High-level synthesis under fixed-point accuracy constraint. *J. Electrical and Computer Engineering*, 2012:906350:1–906350:14, 2012.

[16] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[17] J.-R. Wilcox P. Panchekha, A. Sanchez-Stern and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *PLDI'15*, pages 1–11. ACM, 2015.

[18] R. Yates. *Fixed-point Arithmetic: An Introduction.*, digital signal labs edition, 2009.

[19] S.-M. Rump. Ultimately fast accurate summation. *SIAM J. Scientific Computing*, 31(5):3466–3502, 2009.

[20] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *FM'15*, volume 9109 of *LNCS*, pages 532–550. Springer, 2015.

[21] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: A new approach to optimization. *Log. Meth. in Comp. Sci.*, 7(1), 2011.

[22] L. Thévenoux, P. Langlois, and M. Martel. Automatic source-to-source error compensation of floating-point programs. In *CSE'15*, 2015.