2018 5th International Conference on Control, Decision and Information Technologies (CoDIT'18)

Thessaloniki, Greece
April 10-13, 2018

On the Impact of Numerical Accuracy Optimization on General Performances of Programs

Nasrine Damouche[1] and Matthieu Martel[2]

***Abstract*— The floating-point numbers used in computer programs are a finite approximation of real numbers. In practice, this approximation may introduce round-off errors and this can lead to catastrophic results. In previous work, we have proposed intraprocedural and interprocedural program transformations for numerical accuracy optimization. All these transformations have been implemented in our tool, `Salsa`. The experimental results applied on various programs either coming from embedded systems or numerical methods, show the efficiency of the transformation in terms of numerical accuracy improvement but also in terms of other criteria such as execution time and code size. This article studies the impact of program transformations for numerical accuracy specially in embedded systems on other efficiency parameters such as execution time, code size and accuracy of the other variables (these which are not chosen for optimization).**

I. Introduction

The floating-point arithmetic described by IEEE754 Standard [1], [23] is well-known for its intricate behavior. For example, usual laws such as associativity and distributivity of the addition and product do not hold, operations are not invertible, etc. However, floating-point numbers are more and more used in many industrial applications including critical embedded systems. Unfortunately, floating-point computations can result in unexpected results, a recurrent problem arising because of round-off errors that perturb the accuracy of the results. These last years, techniques have been proposed to systematically validate [3], [13], [14], [16], [25] and improve [19], [24] the numerical accuracy of programs and avoid failures. To deal with this issue, we improve the numerical accuracy of computations by automatically transforming programs in a source to source manner. The transformation includes arithmetic expressions, intraprocedural (assignments, conditionals, loops, etc.) and interprocedural programs (functions). The evaluation of our tool on various floating-point programs coming from embedded systems or numerical algorithms shows that the numerical accuracy of the optimized programs can often be significantly improved. By transforming programs, we create large arithmetic expressions that we re-parse differently in order to find the best way to rewrite the computation considered and then return a more accurate expression among the mathematically equivalent expressions and in polynomial time.

[1]Nasrine Damouche is member of the LAboratory of Mathematics and PhysicS, University of Perpignan, 52 Avenue Paul Alduy, 66860 Perpignan, France. `nasrine.damouche@univ-perp.fr`

[2]Matthieu Martel is member of the LAboratory of Mathematics and PhysicS, University of Perpignan, 52 Avenue Paul Alduy, 66860 Perpignan, France. `matthieu.martel@univ-perp.fr`

In our tool, `Salsa`, the computations of variable ranges and error bounds are relying on static analysis methods by abstract interpretation [4]. We use a set of transformation rules for arithmetic expressions, commands [8] and functions [7]. By using these rules, a transformed program is more accurate in terms of accuracy in the sense that it returns a result which is closer to the exact result that we would obtain if we were using the arithmetic of real numbers. This is confirmed by better theoretical experimental results [11]. This article introduces several contributions.

- The first point concerns the evaluation of the accuracy of several variables. `Salsa` improves the accuracy of a single variable of a program at once. The idea consists of improving the accuracy of a given variable and then of taking the transformed program and re-transforming it by improving the accuracy of another variable. Doing this process several times, and measuring the improvement of the accuracy, we show that the transformations done for one variable do not alternate significantly the transformations done for another variable. It is then possible to optimize the accuracy of many variables using `Salsa`. This is a first multi-criteria optimization.
- Next, we have studied other ways of transforming programs to obtain compromise between numerical accuracy of programs and their execution time. Basically, to optimize an expression, `Salsa` builds a kind of abstract syntax tree called APEG [19] and containing many expressions mathematically equivalent to the original expression. It is then necessary to extract the final expression from this APEG. In general, `Salsa` searches the most accurate expression inside the APEG. Here we explore other alternatives such as:
  – reducing the number of operations of the program,
  – balancing the binary syntactic tree of the program to allow more instruction level parallelism, (ILP),
  – merging the two processes, i.e., reducing the number of operations and balancing its binary syntactic tree simultaneously.

  Finding a compromise between execution time and numerical accuracy is our second kind of multi-criteria optimization.
- A third point concerns the size of the generated code. This characteristic may be important in embedded systems with few memory. We show that the programs transformed by `Salsa` are not much larger that the original ones. In particular, when several variables are optimized, the compromise between accuracy optimiza-

```
%Salsa%
double main(){
e0 = 0.0; t = 0.0; m = 150.0;
i0 = 0.0; dt = 0.2; c = 0.0;
while (t < 10.0) {
  e1 = c − m ;
  p = prop(e1) ;
  i = integral(i0 ,m,c,dt) ;
  d = deriv(e1,e0) ;
  r = p + i + d ;
  m = m + r ∗ 0.01 ;
  t = t + dt ;
  e0 = e1 ;
  }
 return m ;
}
double prop(double e1){
  kp = 9.4514 ;
  res = kp ∗ e1 ;
  return res ;
}
double integral(double ii , double mm,double cc,
                 double ddtt){
  ki = 0.69006 ;
  res = ii + (ki ∗ ddtt ∗ (cc − mm)) ;
  return res ;
}
double deriv(double e1, double e0){
  kd = 2.8454 ;
  invdt = 5.0 ;
  res = kd ∗ (e1 − e0) ∗ invdt ;
  return res ;
}
```

Fig. 1. Original PID Controller program.

tion and code size in much better than if we duplicate the computations to improve optimally the accuracy of each variable separately.

All our experiments are made on an Intel Core i7 with 8 Go memory on Ubuntu 15.04 in IEEE754 single precision in order to emphasize the effect of the finite precision.

We illustrate what Salsa does when performing the transformations mentioned in the above items, on a significant program commonly used in embedded systems: a PID controller. The PID controller used here is far more complex than other PID used in other articles [10], [8]. It uses a finite length window to record the ten last values of errors to compute the integral term and functions to compute P, I and D. This also illustrates the kind of code that Salsa is able to transform. The program corresponding to the PID is given in Figure 1. Several strategies have been proposed in [7] to transform functions and, in this article, we also evaluate them on the PID controller in terms of accuracy and code size.

This article is organized as follows. In Section II, we explain briefly the basics of arithmetic expressions, intraprocedural and interprocedural transformations. In Section III, we present the behavior of numerical accuracy when optimizing many variables simultaneously. Section IV is dedicated to time and accuracy simultaneous optimization. In Section V, we give numbers on the code size of programs before and after being optimized and the execution time required. Related work is discussed in Section VI. Section VII gives perspectives and future work.

II. Background

In this section, we introduce background material concerning the estimation of errors due to the floating-point arithmetic and the optimization of arithmetic expressions, commands and functions. We also introduce the sample programs used in our experiments.

A. Sample Programs

In this section, we briefly present the sample programs used in our experiments in Sections III, IV and V. In addition to the PID controller introduced in Section I, we use five example programs:

- Odometry: It consists of computing the position (x, y) of a two wheeled robot by odometry [11],
- Rocket: It computes the positions of a rocket and a satellite in space. It consists of simulating their trajectories around the earth using the Cartesian and polar systems, in order to project the gravitational forces in the system composed of the earth, the rocket and the satellite [9],
- Runge-Kutta 4: It integrates an order 1 ordinary differential equation [11],
- Jacobi: It consists of an iterative computation that solves linear systems of the form $Ax = b$,
- PID: It keeps a physical parameter at a specific value known as the setpoint [11].

The last program, the PID, is a more complex implementation than in former articles. It uses functions to compute the three terms P, I and D and a finite size window for the integrator. Its implementation is given in Figure 1.

B. Floating-Point Arithmetic

In this section, we give a brief review on the IEEE754 Standard [1]. Then, we explain how the computations of the round-off errors are done. First of all, floating-point numbers are finite approximation of real numbers. This is why round-off errors that arise during the computations may cause damages in critical contexts. IEEE754 Standard formalizes a binary floating-point number as a triplet made of a sign, a mantissa and an exponent. The representation of a floating-point number x written in base b is given by Equation 1.

$$x = s \cdot m \cdot b^{e-p+1} \tag{1}$$

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 $\uparrow_\sim$ in this article. The semantics of the elementary operations specified by IEEE754 Standard is given by Equation (2).

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

where a floating-point operation, denoted by $\circledast_r$, is computed using the rounding mode r. The exact operation is denoted $*$, we have $* \in \{+, -, \times, \div\}$. 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

$$\downarrow_r (x) = x - \uparrow_r (x) \ . \tag{3}$$

In order to compute the errors during the evaluation of arithmetic expressions [21], 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).

Example 2.1: For instance, let us consider the real number $\frac{1}{3}$. Its representation by the value $v = (x, \mu)$, is $v = (\uparrow_\sim \left(\frac{1}{3}\right), \downarrow_\sim \left(\frac{1}{3}\right)) = (0.333333, (\frac{1}{3} - 0.333333))$. The semantics of the elementary operations on $\mathbb{E}$ is defined in [21].

■

In practice, our tool uses an abstract semantics [4] based on $\mathbb{E}$. The abstract values are represented by a pair of intervals. The first interval corresponds to the range of the floating-point values of the program and the second one corresponds to 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).

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

The function that abstracts the concrete function $\downarrow$ is denoted by $\downarrow^\sharp$. 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

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

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) and (7) give the semantics of the addition and multiplication over $\mathbb{E}^\sharp$, for other operations see [21]. 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 $(x_1^\sharp + \mu_1^\sharp) \times (x_2^\sharp + \mu_2^\sharp)$.

$$(x_1^\sharp, \mu_1^\sharp) + (x_2^\sharp, \mu_2^\sharp) = \big(\uparrow^\sharp (x_1^\sharp + x_2^\sharp), \mu_1^\sharp + \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp + x_2^\sharp) \big) \ , \tag{6}$$

$$(x_1^\sharp, \mu_1^\sharp) \times (x_2^\sharp, \mu_2^\sharp) =$$
$$\big(\uparrow^\sharp (x_1^\sharp \times x_2^\sharp), x_2^\sharp \times \mu_1^\sharp + x_1^\sharp \times \mu_2^\sharp + \mu_1^\sharp \times \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp \times x_2^\sharp) \big) \ . \tag{7}$$

C. Transformation for Numerical Accuracy

In this section, we give a brief review on the transformation of programs to improve their numerical accuracy. First, we present how to transform arithmetic expressions introduced by [19]. Then we give the principles of the transformation of both intraprocedural and interprocedural programs presented in [8], [7].

1) Transformation of Expressions: In this section, we present how to improve the numerical accuracy of arithmetic expression [19]. The work in [19] consists of defining a new intermediary representation called APEG for Abstract Program Expression Graph. This approach represents in polynomial size an exponential number of equivalent arithmetic expressions. To limit the combinatorial problems, the APEGs hold in abstraction boxes many equivalent expressions up to associativity and commutativity. 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 algorithm 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-B. The APEGs are an extension of the Equivalence Program Expression Graphs (EPEGs) introduced by R. Tate *et al.* [26]. 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 $\boxed{*(p_1, \ldots, p_n)}$ 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 $\{p_1, \ldots, p_n\}$ of APEGs consists of an APEG where $p_{i, 1 \leq i \leq n}$, is not a set of APEGs itself. We call the set $\{p_1, \ldots, p_n\}$ the equivalence class.

Example 2.2: An example of APEG is given in Figure 2, it represents all the following expressions:

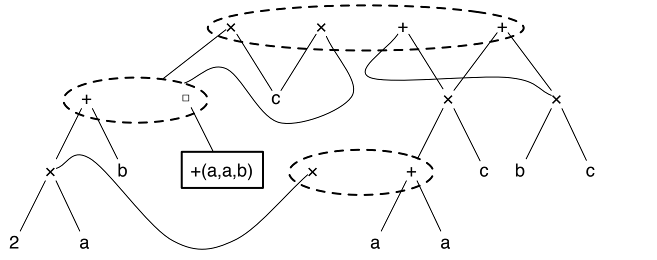

Fig. 2. APEG for the expression $e = \big((a + a) + b\big) \times c$.

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

■

In their article [26], R. Tate *et al.* 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.

2) Transformation of programs: In this section, we summarize our transformation rules introduced in our previous work [7], [8]. We start by presenting the intraprocedural transformation of programs. Then, we present the interprocedural transformation of programs. These rules used to improve the numerical accuracy of programs are implemented in our tool, `Salsa`.

a) Intraprocedural Transformation: The intraprocedural transformation rules that we present in this section includes assignments, conditionals, loops, etc. The principle of the transformation of commands relies on a set of hypotheses:

- Programs are defined by a tuple

  $$\langle c, \delta, C, \beta\rangle \rightarrow_\nu \langle c', \delta, C, \beta\rangle$$

  where c is the program to be optimized, δ the formal environment that maps variables to expressions, 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 and ν is the reference variable at optimizing.
- To avoid conflict between read and written variables at different control points, our parser puts the inputs programs in SSA form (single static assignments).
- The best program, i.e. the most accurate, is obtained by comparing the reference variable of the original and transformed programs.

We survey below briefly the different kinds of transformation rules. We refer the interested reader to [8] to see the details of these rules. We start with assignments. We have two rules, the first consists of removing the assignment from the program and saving it in the memory δ if some conditions are verified. 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 simplify their manipulation, we slice these formulas, at a defined level of the binary syntactic tree, in several sub-expressions, and we assign them to intermediary variables, `TMP`. Finally, we inject these new assignments into the main program.

Example 2.3: To explain the use of the transformation rules of assignments, let us consider Equation (9) 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 $a = 0.1$, $b = 0.0001$ are constants.

$$\begin{array}{ll} & \langle \mathtt{x = a + b;\ y = x + x;}\ ,\ \delta,\ [\,],\ \{\mathtt{y}\}\rangle \\ \Longrightarrow_\nu & \langle \mathsf{nop};\ \mathtt{y = x + x;}\ ,\ \delta' = \delta[\mathtt{x} \mapsto \mathtt{a + b}],\ [x = a + b;\ ],\ \{\mathtt{y}\}\rangle \\ \Longrightarrow_\nu & \langle \mathtt{y = (a + b) + (a + b)}, \delta' = \delta[\mathtt{x} \mapsto \mathtt{a + b}],\ [\,],\ \{\mathtt{y}\}\rangle \\ \Longrightarrow_\nu & \langle \mathtt{y = (((b + b) + a) + a)}, \delta',\ [\,], \{\mathtt{y}\}\rangle \end{array} \quad (9)$$

In Equation (9), initially, the environment δ is empty, the black list contains y and the reference variable ν at optimizing is y. 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 $\delta = [x \mapsto a+b]$. Next, we apply a rule for sequences which discards the `nop` statement. For the last step, we may not discard y because the condition is not satisfied ($y = \nu$). Then, we substitute x by their value in δ and we transform the expression.

■

Our implementation transforms also conditionals and loops. For 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. For the `while` loops, 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. The last transformation rules concerns 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.

b) Interprocedural Transformation: In this section, we briefly show how to transform functions using a set of transformation rules formally defined in [6], [7]. We assume that any program has a function named *main* which is the first function called at execution time, and the returned variable v is the *target variable* to be optimized $\nu = \{v\}$.

Basically, our interprocedural transformation follows the same objective as the intraprocedural one.

We aim at creating large arithmetic expressions which can be recombined into more accurate ones as explained in Section II-C.1. The larger the expressions are, the more opportunities we have to rewrite them. The first interprocedural transformation rule, (R1), consists of inlining the body of the function into the calling function. This makes it possible to create larger expressions in the caller. Then the new program can be more optimized by applying the intraprocedural transformation rules previously seen in Section II-C.2.a. Recall that, the use of rule (R1) depends of two aspects: the main function must contain fewer number of calls to the callee function and the body of the callee function must not be enough larger. The second transformation rule, (R2), is used when we deal with a small number of calls to a large function in the original program. The idea is

```
%Salsa%
double main() {
e0 = 0.0; t = 0.0; m = 0.150e3;
while (t < 10.0)  {
  e1 = -0.150e3 ;
  kp = 0.94514e1 ;
  res1 = (kp * -0.150e3) ;
  ii = 0.0 ;
  mm = 0.150e3 ;
  cc = 0.0 ;
  ddtt = 0.2 ;
  ki = 0.69006 ;
  res2 = (ii + ((ki * ddtt) * (cc - m))) ;
  kd = 0.28454e1 ;
  invdt = 0.5e1 ;
  res3 = ((kd * (-0.150e3 - e0)) * invdt) ;
  m = (((0.1e-1 * (-0.213405e4)) + ((0.1e-1
    * (-0.207018e2)) + (0.1e-1 * (-0.141771e4)))) + m);
  t = (0.2 + t) ;
  e0 = -0.150e3
  }
return m ;
}
```

Fig. 3. Optimized PID Controller program using R1 of Section II-C.2.b.

```
%Salsa%
double main() {
e0 = 0.0;
t = 0.0;
m = 0.150e3 ;
while (t < 10.0)  {
  e1 = (c - m) ;
  p = propTMP_2() ;
  i = integralTMP_7() ;
  d = derivTMP_10() ;
  m = (((+0.1e-1 * (-0.213405e4)) + ((+0.1e-1
    * (-0.207018e2)) + (+0.1e-1 * (-0.141771e4)))) + m);
  t = (0.2 + t) ;
  e0 = -0.150e3
 }
return m ;
}
double derivTMP_10() {
  TMP_8 = -0.150e3 ;
  res = (+0.28454e1 * (-0.150e3 * 0.5e1))
  return res ;
}
double integralTMP_7() {
  TMP_4 = 0.150e3 ;
  TMP_6 = 0.2 ;
  res = (TMP_6 * (TMP_4 * 0.69006))
  return res ;
}
double propTMP_2() {
  TMP_1 = -0.150e3 ;
  res = -0.141771e4
  return res ;
}
```

Fig. 4. Optimized PID Controller program using R2 of Section II-C.2.b.

to specialize the function with respect to the argument by passing the abstract value of the argument to the function when the variability of the interval is small (for example whenever it contains less than ω floating-point numbers). By variability, we mean that the distance between the lower bound and the upper bound of an interval is small. If the variability of the interval $i = [\underline{i}, \overline{i}]$ is smaller than a parameter ω then, we apply the second rule, we substitute the argument of the function by the abstract value of the parameter. In practice, we choose $\omega = 2^4 \times ulp(max(|\underline{i}|, |\overline{i}|)$ where ulp(x) is defined in Section II-B. Note that we conserve the original function in our code when the condition on the variability is not satisfied. The last rule, (R3), consists of substituting the formal expression of the parameters of a given function call to the formal parameters inside the body of the called function. It can be seen as a lazy evaluation of the parameters in the caller. By applying this rule, we obtain the new function whose parameters are the variables of the expressions of the arguments. Then we rewrite the new function by using the intraprocedural transformation rules to optimize the numerical accuracy of the computations. In Figure 1, we give the code of the original PID program and the optimized programs obtained by using respectively only the rule (R1), (R2) or (R3) previously presented.

III. Multi-variables Optimization

In this section, we compare the improvement of one single variable versus more than one variable and we study the interest in improving the same variable several times. The question addressed here is to know whether, by optimizing the numerical accuracy of a given variable, we decrease the accuracy of another variables within the program or not. Hence, we compare the improvement of numerical accuracy of a single variable to the accuracy obtained when optimizing this variable several times (in our case, we optimize it three times). We also, optimize different variables of the program. That means that we optimize the first variable, then we take the optimized program and we optimize it for the second variable, and so on. The results measured show that the transformations done for one variable do not alternate significantly the transformations done for other variables. In our experiments, we have evaluated the optimization of the accuracy of several variables on the programs `odometry`, `Jacobi` and `Rocket` introduced in Section II-A.

For instance, if we take the `odometry` program which computes the position (x, y) of a two wheeled robot, we remark that by optimizing the variable x twice, or when optimizing x and then y, we obtain the same result. In other words, the variable y is optimized in the same way alone or jointly with x. Figure 6 summarizes the results obtained for `odometry`, `Jacobi` and `Rocket`. For example, for `odometry`, in the first line of Figure 6, we optimize x once. Its accuracy passes from $1.9855e^{-13}$ to $6.5252e^{-14}$ and at the same time, the accuracy of y passes from $2.6080e^{-13}$ to $9.2143e^{-14}$. The variable y which has not been chosen for optimization is optimized anyway. We have also observed on the other examples that optimizing the accuracy of one variable increases the accuracy of the others. We could think that the transformation done for one variable decreases the accuracy of the others. Our experiments show that this is not the case and that several variables can be optimized for our sample programs.

IV. Time and Accuracy Optimization

The main idea of this experiment consists of finding a compromise between the numerical accuracy of programs and their execution time. To do this, we take the original program and we rewrite it in three different ways. In the first strategy, the program is rewritten in manner to obtain a

Code	Optimized value	Initial error on X	Error after Transformation on X	Initial error on Y	Error after Transformation on Y
Odometry	`X`	$1.9855e^{-13}$	$6.5252e^{-14}$	$2.6080e^{-13}$	$9.2143e^{-14}$
	`X → X`	$6.5252e^{-14}$	$5.3587e^{-14}$	$9.2143e^{-14}$	$6.1489e^{-14}$
	`X→X→X`	$5.3587e^{-14}$	$6.5252e^{-14}$	$6.1489e^{-14}$	$9.2143e^{-14}$
	`X→Y`	$6.5252e^{-14}$	$5.3587e^{-14}$	$9.2143e^{-14}$	$6.1489e^{-14}$

Code	Optimized value	Initial error on X	Error after Transformation on X	Initial error on Y	Error after Transformation on Y
Rocket	`X`	$1.2909e^{-14}$	$2.4145e^{-14}$	$2.8569e^{-14}$	$1.3977e^{-14}$
	`X→X`	$2.4145e^{-14}$	$2.4145e^{-14}$	$1.3977e^{-14}$	$1.3977e^{-14}$
	`Y`	$1.2909e^{-14}$	$2.4145e^{-14}$	$2.8569e^{-14}$	$1.3977e^{-14}$
	`Y→Y`	$2.4145e^{-14}$	$2.4145e^{-14}$	$1.3977e^{-14}$	$1.3977e^{-14}$

Code	Optimized value	Initial error on X1	Error after Transformation on X1	Initial error on X2	Error after Transformation on X2
Jacobi	`X1`	$2.4242e^{-16}$	$1.3759e^{-16}$	$2.5658e^{-16}$	$1.4595e^{-16}$
	`X1→X1`	$1.3759e^{-16}$	$1.3759e^{-16}$	$1.4595e^{-16}$	$1.4595e^{-16}$
	`X2`	$2.4242e^{-16}$	$1.3759e^{-16}$	$2.5658e^{-16}$	$1.4595e^{-16}$
	`X2→X2`	$1.3759e^{-16}$	$1.3759e^{-16}$	$1.4595e^{-16}$	$1.4595e^{-16}$

Fig. 6. Optimizing one vs. several reference variables of `Odomerty`, `Rocket` and `Jacobi` programs.

```
%Salsa%
double main() {
e0 = 0.0 ;
t = 0.0 ;
m = 0.150e3 ;
while (t < 10.0)  {
   e1 = (c − m) ;
   p = propTMP_1() ;
   i = integralTMP_2() ;
   d = derivTMP_3() ;
   m = (((+0.1e−1 ∗ (−0.213405e4)) + ((+0.1e−1
     ∗ (−0.207018e2)) + (+0.1e−1 ∗ (−0.141771e4)))) + m);
   t = (0.2 + t) ;
   e0 = −0.150e3
 }
return m ;
}
double derivTMP_3() {
   res = (+0.28454e1 ∗ (−0.750e3)) ;
   return res ;
}
double integralTMP_2() {
   res = (−0.207018e2) ;
   return res ;
}
double propTMP_1() {
   res = −0.141771e4 ;
   return res ;
}
```

Fig. 5. Optimized `PID` Controller program using R3 of Section II-C.2.b.

balanced expression in terms of its binary syntactic tree. By this, we mean that we change the order of the parentheses of the expressions in order to obtain a balanced syntactic tree while respecting the semantics of the original program. For example, $((a+b)+c)+d$ is less balanced than $(a+b)+(c+d)$. Balanced expressions are better suited than unbalanced ones with regard to the instruction level parallelism offered by modern processors. In other words, the balanced tree allows one to accelerate the computations on certain architectures.

In the second strategy, we rewrite the original program by reducing the number of operations in the given expressions of the program. In other words, this strategy consists of factorizing or simplifying the expressions of the program. For example, $a \times (b + c)$ contains less operations than $(a \times b) + (a \times c)$ and both expressions are mathematically equivalent.

For the last strategy, we merge the two former approaches, i.e., we reduce the number of operations of the program and then we balance its binary syntactic tree. For example, by reducing the number of operations, the expression $((x \times a + x \times b) + x \times c) + x \times d$ becomes $x \times (((a+b)+c)+d)$. Then this new expression is balanced and we obtain $x \times ((a+b)+(c+d))$.

For these three strategies, we measure the numerical accuracy of computations and the execution time needed for the execution of each of them. We have experimented these three strategies on four of the programs introduced previously in this article: `odometry`, `Jacobi`, `PID` and `Runge-Kutta` of order four. In each case, we take the original program and we transform it by `Salsa` and then, we take the rewritten program and we transform it also by `Salsa`. Then we observe the behavior of each of them in terms of accuracy and execution time. The first part of Figure 7 gives the results obtained when just balancing the binary syntactic tree of each program. We observe that the numerical accuracy of computations is improved and the execution time needed by the program is reduced. For instance, if we take the `PID` controller program, Figure 7 shows that the initial error of the original program which is $4.6680e^{-14}$, is reduced to $2.1346e^{-16}$ when using `Salsa`, where the initial error of the transformed program with balancing the binary syntactic tree which equals to $2.5637e^{-15}$ is improved to $1.9829e^{-16}$. Now, if we focus on the execution time required by both these programs, we find that the time is gone from $0.125s$ to $0.098s$, a significant improvement of 21.6%.

The results obtained with the second strategy also are summarized in Figure 7. The figure shows that by reducing the number of operation of the program, the numerical accuracy of programs and their execution time are improved. By example, the initial error on the `PID` program is passed from $2.8574e^{-15}$ to $1.5211e^{-15}$ and the program needs $0.1s$ to be transformed instead of $0.125s$.

In the last experimentation, we have combined both reducing the number of operations and then balancing the syntactic tree of programs. We give still in Figure 7 the results obtained by merging both approaches as explained previously. This seems very interesting since the results are more relevant. In case of the `PID` program, the optimized error is $1.9829e^{-16}$ where the initial error was $2.5622e^{-15}$.

Code	Original Program			Transformed Program (bal. tree)		
	Initial error	Transfo. error	Transfo. time(s)	Initial error	Transfo. error	Transfo. time(s)
Odometry	$4.9801e^{-13}$	$1.9829e^{-13}$	0.098	$2.5637e^{-14}$	$1.9829e^{-15}$	0.090
Runge-Kutta 4	$4.6666e^{-13}$	$2.9226e^{-13}$	0.098	$4.6047e^{-14}$	$2.9253e^{-15}$	0.067
Jacobi	$2.9142e^{-16}$	$1.7258e^{-16}$	1.098	$2.5658e^{-16}$	$1.5517e^{-16}$	0.125
PID	$4.6680e^{-14}$	$2.1346e^{-16}$	0.125	$2.5637e^{-15}$	$1.9829e^{-16}$	0.098

Code	Original Program			Transformed Program (nb. op)		
	Initial error	Transfo. error	Transfo. time(s)	Initial error	Transfo. error	Transfo. time(s)
Odometry	$4.9801e^{-13}$	$1.9829e^{-13}$	0.098	$2.4011e^{-14}$	$1.8574e^{-15}$	0.091
Runge-Kutta 4	$4.6666e^{-13}$	$2.9226e^{-13}$	0.098	$4.6805e^{-14}$	$2.9312e^{-16}$	0.751
Jacobi	$2.9142e^{-16}$	$1.7258e^{-16}$	1.098	$1.7176e^{-16}$	$4.4408e^{-17}$	0.115
PID	$4.6680e^{-14}$	$2.1346e^{-16}$	0.125	$2.8574e^{-15}$	$1.5211e^{-15}$	0.100

Code	Original Program			Transformed Program (bal. tree & nb. op.)		
	Initial error	Transfo. error	Transfo. time(s)	Initial error	Transfo. error	Transfo. time(s)
Odometry	$4.9801e^{-13}$	$1.9829e^{-13}$	0.098	$1.5847e^{-14}$	$1.9229e^{-15}$	0.080
Runge-Kutta 4	$4.6666e^{-13}$	$2.9226e^{-13}$	0.098	$3.6095e^{-14}$	$2.9253e^{-15}$	0.050
Jacobi	$2.9142e^{-16}$	$1.7258e^{-16}$	1.098	$1.5658e^{-16}$	$1.5017e^{-16}$	0.120
PID	$4.6680e^{-14}$	$2.1346e^{-16}$	0.125	$2.5622e^{-15}$	$1.9829e^{-16}$	0.070

Fig. 7. Reduction of the number of operations and/or balancing the binary syntactic tree of programs.

Same for the execution time, it is decreased from $0.125s$ into $0.070s$. This shows that improving accuracy and execution time simultaneously is possible on representative examples.

V. Code size

In this section, we aim at evaluating the performances of Salsa from another point of view: numerical accuracy and code size of programs. By transforming programs, we may create new variables when we deal with large expressions that we associate to TMP variables (see Section II). This explains the code size before and after transformation. The comparison between the original program code size and the transformed one is given in Figure 8. For instance, in the case of the PID Controller program, we remark that the size of the original program is 623 Bytes while its code after transformation is 1068 Bytes. In mean, the size of the optimized code is less than twice the size of the original.

Despite increasing the size of program, the numerical accuracy of each program is widely reduced (see Figure 8). More precisely, if we take the PID Controller program, we have that the numerical accuracy of the original program is $0.2059e^{-13}$ while its accuracy after being transformed is $0.2221e^{-13}$. It means that the error of computation of the PID Controller is reduced by 7.86%. This is mainly due to the fact that Salsa performs partial evaluation during the transformation and that computations with fewer operations often generate less errors and are often privileged in the choice of expressions in APEGs.

VI. Related Work

During the last fifteen years, several static analyses of the numerical accuracy of floating-point computations have been introduced. While these methods compute an over-approximation of the worst error arising during the executions of a program, they operate on final codes, during the verification phase and not at implementation time. Static analyses based on abstract interpretation [4], [5] have been proposed and implemented in the Fluctuat tool [17], [18] which has been used in several industrial contexts. A main advantage of this method is that it enables one to bound safely all the errors arising during a computation, for large ranges of inputs. It also provides hints on the sources of errors, that is on the operations which introduce the most important precision loss. This latter information is of great interest to improve the accuracy of the implementation. More recently, Darulova and Kuncak have proposed a tool, Rosa, which uses a static analysis coupled to a SMT solver to compute the propagation of errors [13]. None of the techniques mentioned above generate more accurate programs.

Other approaches rely on dynamic analysis. For instance, the Precimonious tool tries to decrease the precision of variables and checks whether the accuracy requirements are still full filled [2]. Lam et al. instrument binary codes in order to modify their precision without modifying the source codes [20]. They also propose a dynamic search method to identify the pieces of code where the precision should be modified. Again, these techniques do not transform the codes in order to improve the accuracy.

Finally, another related research axis concerns the compile-time optimization of programs to improve the accuracy of the floating-point computation in function of given ranges for the inputs, without modifying the formats of the numbers [15]. The Sardana tool takes arithmetic expressions and optimize them using a source-to-source transformation. Herbie optimizes the arithmetic expressions of Scala codes. While Sardana uses a static analysis to select the best expression, Herbie uses dynamic analysis (a set or random runs). A comparison of these tools is given in [12]. These techniques are limited to arithmetic expressions.

VII. Conclusion

In this article, we have shown the usefulness of our tool, Salsa to improve the accuracy of programs in conjunction with other criteria. These other criteria are often very important in embedded systems since they concern memory and speed. We have experimented our tool to optimize simultaneously the numerical accuracy and the execution time of programs by first balancing the syntactic tree, second reducing the number of operations of programs and finally by merging both approaches. We have also shown that

Code	Initial error	Error after transformation	% of improvement	Code size$_o$ (Bytes)	Code size$_t$ (Bytes)	$\frac{codesize_o}{codesize_t}$
`Odometry`	$0.5794e^{-14}$	$0.4788e^{-14}$	17.36	900	1439	0.62
`Rocket`	$2.8569e^{-14}$	$1.3977e^{-14}$	51.07	1911	1624	1.1
`Runge-Kutta 4`	$4.6666e^{-13}$	$2.9226e^{-13}$	37.37	678	1649	0.41
`Jacobi`	$2.9142e^{-16}$	$1.7258e^{-16}$	40.77	1032	1959	0.52
`PID`	$0.2059e^{-13}$	$0.2221e^{-13}$	7.86	623	1068	0.58

Fig. 8. Numerical accuracy and code size measurements of programs.

by optimizing a single variable the accuracy of the other variables is not decreased. Also, we have shown that by creating new variables while transforming programs, the code size of the optimized program is less than the size of programs if we duplicate their code.

A future work consists of studying the impact of the accuracy optimization of parallel programs and also to consider an 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 the parallel evaluation of the mathematical expression. It will be very interesting to study how our technique could improve reproducibility.

Another research direction concerns the optimization of the data-types in order to save memory [22]. This consists of finding the least formats (i.e., half, single or double precision) required in order to ensure the accuracy of results. In addition, it would be interesting to extend our program transformation to help this format optimization.

References

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

[2] F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In *ACM SIGPLAN PLDI'12*, pages 453–462. ACM, 2012.

[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] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Principles of Programming Languages*, pages 178–190. ACM, 2002.

[6] N. Damouche. *Improving the Numerical Accuracy of Floating-Point Programs with Automatic Code Transformation Methods*. PhD thesis, Université de Perpignan Via Domitia, 2016.

[7] N. Damouche, M. Martel, and A. Chapoutot. Numerical accuracy improvement by interprocedural program transformation. accepted.

[8] 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.

[9] N. Damouche, M. Martel, and A. Chapoutot. Optimizing the accuracy of a rocket trajectory simulation by program transformation. In *CF'15*, pages 40:1–40:2. ACM, 2015.

[10] N. Damouche, M. Martel, and A. Chapoutot. Transformation of a PID controller for numerical accuracy. *ENTCS*, 317:47–54, 2015.

[11] N. Damouche, M. Martel, and A. Chapoutot. Improving the numerical accuracy of programs by automatic transformation. In *International Journal on Software Tools for Technology Transfer*. Springer, 2016. DOI: 10.1007/s10009-016-0435-0.

[12] N. Damouche, M. Martel, P. Panchekha, C. Qiu, A. Sanchez-Stern, and Z. Tatlock. Toward a standard benchmark format and suite for floating-point analysis. In P. Prabhakar S. Bogomolov, M. Martel, editor, *NSV*, LNCS. Springer, 2016.

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

[14] 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.

[15] P-L. Garoche, F. Howar, T. Kahsai, and X. Thirioux. Testing-based compiler validation for synchronous languages. In J. M. Badger and K. Yvonne Rozier, editors, *NFM*, volume 8430 of *LNCS*, pages 246–251. Springer, 2014.

[16] 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.

[17] E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations: A simple abstract interpreter. In D. Le Métayer, editor, *ESOP*, volume 2305 of *LNCS*, pages 209–212. Springer, 2002.

[18] E. Goubault, M. Martel, and S. Putot. Some future challenges in the validation of control systems. In *ERTS*, 2006.

[19] 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.

[20] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In *Supercomputing, ICS'13*, pages 369–378. ACM, 2013.

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

[22] Matthieu Martel. Floating-point format inference in mixed-precision. In Clark Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal Methods - 9th International Symposium, NFM 2017*, volume 10227 of *Lecture Notes in Computer Science*, pages 230–246, 2017.

[23] 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.

[24] 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.

[25] 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.

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