Kalpa Publications in Computing

Volume 5, 2018, Pages 63–76

Automated Formal Methods

kalpa Computing

Salsa: An Automatic Tool to Improve the Numerical Accuracy of Programs

Nasrine Damouche and Matthieu Martel

LAMPS Laboratory
University of Perpignan,
52 Avenue Paul Alduy,
Perpignan, France, 66860.
{nasrine.damouche,matthieu.martel}@univ-perp.fr

Abstract

This article describes Salsa, an automatic tool to improve the accuracy of the floating-point computations done in numerical codes. Based on static analysis methods by abstract interpretation, our tool takes as input an original program, applies to it a set of transformations and then generates an optimized program which is more accurate than the initial one. The original and the transformed programs are written in the same imperative language. This article is a concise description of former work on the techniques implemented in Salsa, extended with a presentation of the main software architecture, the inputs and outputs of the tool as well as experimental results obtained by applying our tool on a set of sample programs coming from embedded systems and numerical analysis.

1 Introduction

Floating-point numbers, whose specification is given by the IEEE754 Standard [1, 28], are more and more used in many industrial domains, including critical embedded software and, obviously, numerical simulations. However, floating-point arithmetic is prone to accuracy problems due to the round-off errors. The approximation becomes dangerous when accumulated errors cause damages whose gravity varies depending on the critical level of the application.

In this context, ensuring the correctness of the computations in embedded systems or numerical simulations is an important but extremely difficult task. Some useful tools and techniques have been developed for validating [4, 15, 16, 19, 30] and improving [24, 29] the accuracy of arithmetic expressions in order to avoid numerical failures (in this article, the accuracy refers to the difference between the exact and the floating-point result). One limitation of these tools, for example, `Sardana` [24] or `Herbie` [29], is that they are applicable on a single arithmetic expression. In other words, they cannot merge computations between the different lines of the code. For example, they can do nothing on a code written in 3-address style. To cope with this limitation, we propose an automatic tool, `Salsa`, which takes programs made of assignments, conditionals, loops, functions, etc. and generates another program numerically more accurate using a source-to-source transformation. This is possible thanks to the set of intraprocedural

and interprocedural transformation rules defined in previous work [13, 9]. `Salsa` relies on static analysis by abstract interpretation [5] to compute variable ranges and round-off error bounds.

In this article, we present a comprehensive summary of how `Salsa` works. An extended description is given in [8]. We give an overview of the formal intraprocedural [13] and interprocedural [9] rules used in our transformation as well as on how the transformation of expressions is done [24]. Then we present the software architecture of the tool and, finally, we give a set of benchmarks measuring the efficiency of `Salsa` from several points of view: numerical accuracy, code size, execution time, program transformation time.

The reminder of the article is organized as follows. Section 2 describes related work. Section 3 presents the IEEE754 Standard and how to compute the error bounds. Section 4 details the transformation of arithmetic expressions, the intraprocedural and the interprocedural transformation of programs. Section 5 introduces the main architecture of our tool, its inputs and outputs. Section 6 describes the implementation of our tool and its application to a number of example programs. This section also gives various experimental results obtained by applying our tool to our set of examples. We conclude in Section 7.

2 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 [5, 6] have been proposed and implemented in the `Fluctuat` tool [20, 21] 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 [15]. Solovyev et al. have proposed another tool, FP-Taylor based on symbolic Taylor expansions [30]. 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 [3]. Lam et al. instrument binary codes in order to modify their precision without modifying the source codes [26]. 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 [17]. 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 [14]. These techniques are limited to arithmetic expressions.

3 Floating-Point Arithmetic

In this section, we present briefly the IEEE754 Standard [1]. Next, we describe how the round-off errors are computed.

3.1 The IEEE754 Standard

Floating-point numbers are used to represent real numbers [18, 1]. Because of their finite representation, round-off errors arise during the computations and this may cause damages in critical contexts. The IEEE754 Standard formalizes a binary floating-point number as a triplet made of a sign, a mantissa and an exponent. We consider that a number x is written:

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

where, s is the sign $\in \{-1, 1\}$, b is the basis, $b = 2$, m is the mantissa, $m = d_0.d_1 \dots d_{p-1}$ with digits $0 \leq d_i < b$, $0 \leq i \leq p-1$, p is the precision and e is the exponent $e \in [e_{min}, e_{max}]$. The IEEE754 Standard specifies some particular values for p, e_{min} and e_{max}.

The 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 $\uparrow_\sim$. Let $\mathbb{R}$ be the set of real numbers and $\mathbb{F}$ be the set of floating-point numbers (we assume that only one format is used at the time, e.g. single or double precision). The semantics of the elementary operations specified by the 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 $r \in \{\uparrow_{+\infty}, \uparrow_{-\infty}, \uparrow_0, \uparrow_\sim\}$ and $* \in \{+, -, \times, \div\}$ is 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 error. We have

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

3.2 Error Bound Computation

In order to compute the errors during the evaluation of arithmetic expressions [27], we use values which are pairs $(x, \mu) \in \mathbb{F} \times \mathbb{R} \equiv \mathbb{E}$ where x is the floating-point number used by the machine and μ is 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 $v = (\uparrow_\sim \left(\frac{1}{3}\right), \downarrow_\sim \left(\frac{1}{3}\right)) = (0.333333, (\frac{1}{3} - 0.333333))$. The semantics of the elementary operations on $\mathbb{E}$ is defined in [27].

Our tool uses an abstract semantics [5] 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 $(x^\sharp, \mu^\sharp) \in \mathbb{E}^\sharp$, $x^\sharp$ is the interval corresponding to the range of the values and $\mu^\sharp$ is 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).

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

We denote by $\downarrow^\sharp$ the function that abstracts the concrete function $\downarrow_\sim$. Every error associated to $x \in [\underline{x}, \overline{x}]$ is included in $\downarrow^\sharp ([\underline{x}, \overline{x}])$. For a rounding mode to the nearest, we have

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

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 [27]. If we sum two numbers, we must add the 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 $(x_1^\sharp + \mu_1^\sharp) \times (x_2^\sharp + \mu_2^\sharp)$.

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

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

4 Transformation for Numerical Accuracy

In this section, we briefly present how to transform programs in order to improve their numerical accuracy. We start by presenting the transformation of arithmetic expressions introduced in [24]. Then we give the principles of the transformation of both intraprocedural and interprocedural programs.

4.1 Transformation of Expressions

We briefly introduce former work [24, 31] to semantically transform arithmetic expressions using Abstract Program Expression Graph (APEG). This data structure remains in polynomial size while dealing with an exponential number of equivalent expressions. To prevent any combinatorial problem, 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 algorithms). 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 factorizes common factors. Finally, an accurate formula is searched among all the equivalent formulas of the APEG using the abstract semantics of Section 3.2.

Example 4.1. *An example of APEG is given in Figure 1. When an equivalence class (denoted by a dotted ellipse) contains many sub-APEGs $p_1, \ldots, p_n$ then one of the p_i, $1 \leq i \leq n$, must be selected in order to build an expression. A box $\boxed{*(p_1, \ldots, p_n)}$ represents any parsing of the expression $p_1 * \ldots * p_n$. For instance, the APEG p of Figure 1 represents all the following expressions:*

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

In this example, the last step of the transformation consists of evaluating all the expressions in $\mathcal{A}(p)$ with the abstract semantics of Section 3.2 to select the most accurate one. ■

4.2 Intraprocedural Transformation

In this section, we focus on the transformation of commands which is done using a set of rewriting rules formally defined in [13]. Our language is made of assignments, conditionals, loops, etc. The transformation relies on several hypotheses. First of all, programs are assumed to be in static single assignment form (SSA form) [7]. The principle of this intermediary representation is that every variable may be assigned only once in the source code and must be assigned before its use. Our tool, Salsa puts programs in SSA form after passing them. The second hypothesis is that we optimize a *target variable* chosen by the user and denoted ν hereafter. Our transformation is defined by rules using states $\langle c, \delta, C, \beta \rangle$ where

- c is a command,
- δ is a formal environment $\delta : \mathcal{V} \rightarrow$ Expr which maps variables of $\mathcal{V}$ to expressions. Intuitively, this environment records the expressions assigned to variables in order to inline them later on in larger expressions,
- $C \in$ Ctx is a single hole context [23] that records the program enclosing the current command to be transformed,
- $\beta \subseteq \mathcal{V}$ is a list of assigned variables that should not be removed from the code. Initially, $\beta = \{\nu\}$, *i.e.*, the target variable ν must not be removed.

The environment δ is used to discard assignments from programs and to re-insert the expressions when the variables are read, in order to build larger expressions.

Let us consider first assignments. If (i) the variable v of some assignment $v = e$ does not exist in the domain of δ, if (ii) $v \notin \beta$ and if (iii) $v \neq \nu$ then we memorize e in δ and we remove the assignment from the program. Otherwise, if one of the conditions (i), (ii) or (iii) is not satisfied then we rewrite this assignment by inlining the variables saved in δ in the concerned expression. Note that, when transforming programs by inlining expressions in other expressions, we get larger formulas. In our implementation, when dealing with too large expressions, we create intermediary variables and we assign to them the sub-expressions obtained by slicing the global expression at a given level of the syntactic tree. The last step consists of re-inserting these intermediary variables into the main program.

Example 4.2. *For example, let us consider the program below in which three variables x, y and z are assigned. We assume that z is the variable that we aim at optimizing and $a = 0.1$,*

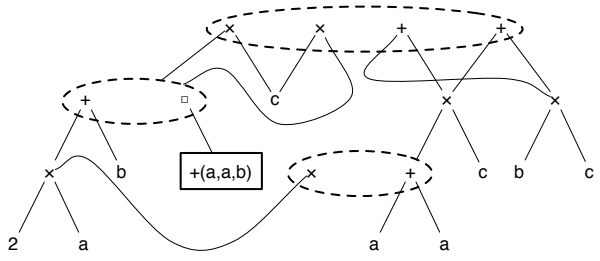

Figure 1: APEG for the expression $e = \big((a + a) + b\big) \times c$.

$b = 0.01$, $c = 0.001$ *and* $d = 0.0001$ *are constants.*

$$\begin{array}{l}\langle \mathtt{x} = \mathtt{a} + \mathtt{b}\ ;\ \mathtt{y} = \mathtt{c} + \mathtt{d}\ ;\ \mathtt{z} = \mathtt{x} + \mathtt{y}\ ,\ \delta,\ [],\ [z]\rangle \\ \rightarrow_\nu \langle \mathit{nop}\ ;\ \mathtt{y} = \mathtt{c} + \mathtt{d}\ ;\ \mathtt{z} = \mathtt{x} + \mathtt{y}, \delta' = \delta[\mathtt{x} \mapsto \mathtt{a} + \mathtt{b}],\ x = a + b; [],\ [z]\rangle \\ \rightarrow_\nu \langle \mathit{nop}\ ;\ \mathit{nop}\ ;\ \mathtt{z} = \mathtt{x} + \mathtt{y}, \delta'' = \delta'[\mathtt{y} \mapsto \mathtt{c} + \mathtt{d}],\ x = a + b; y = c + d; [],\ [z]\rangle \\ \rightarrow_\nu \langle \mathit{nop}\ ;\ \mathit{nop}\ ;\ \mathtt{z} = ((\mathtt{d} + \mathtt{c}) + \mathtt{b}) + \mathtt{a},\ \delta'',\ x = a + b; y = c + d; [],\ [z]\rangle \end{array} \quad (8)$$

In Equation (8), the environment δ and the context C are initially empty and the list β contains the target variable *$\mathtt{z}$. We 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 on the variable y. For the last step, we may not remove z because it is the* target variable. *Instead, we substitute, in z, x and y by their values in δ and we transform the expression using the technique described in Section 3.2.*

■

Our tool also transforms conditionals. If a certain condition is always true or false, then we keep only the right branch, otherwise, we transform both branches of the conditional. When it is necessary, we re-inject variables that have been discarded from the main program.

For a sequence $c_1;\ c_2$, the first command c_1 is transformed into c_1' in the current environment δ, C, ν and β and a new context C' is built which inserts c_1' inside C. Then c_2 is transformed into c_2' using the context $C[c_1'; []]$, the formal environments δ' and the list β' resulting from the transformation of c_1. Finally, we return the state $\langle c_1'\ ;\ c_2', \delta'', C, \beta''\rangle$.

Other transformations have been defined for while loops. A first rule makes it possible to transform the body of the loop assuming that the variables of the condition have not been stored in δ. In this case, the body is optimized in the context $C[\mathsf{while}_\Phi\ e\ \mathsf{do}\ []]$ where C is the context of the loop. A second rule builds the list $V = Var(e) \cup Var(\Phi)$ where $Var(\Phi)$ is the list of variables read and written in the Φ nodes of the loop. The set V is used to achieve two tasks: firstly, it is used to build a new command c' corresponding to the sequence of assignments that must be re-inserted. Secondly, the variables of V are removed from the domain of δ and added to β. The resulting command is obtained by transforming $c'; \mathsf{while}_\Phi\ e\ \mathsf{do}\ c$ with δ' and $\beta \cup V$.

4.3 Interprocedural Transformation

In this section, we show how we transform functions using a set of rewriting rules formally defined in [8, 9]. For the sake of simplicity, in our formal definitions, we only consider functions with one single argument only. However, in our implementation we support functions with many arguments. The generalization is straightforward. In addition, we assume that any program p has a function named *main* which is the first function called when executing p, and the returned variable v consists of the *target variable* at optimizing $\nu = \{v\}$ with $\nu \in \mathcal{V}$. Recall that $\mathcal{V}$ denotes the set of identifiers.

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 4.1. The larger the expressions are, the more opportunities we have to rewrite them. In the case of functions, we may either inline the body of a function into the caller or evaluate lazily the arguments, specially when they correspond to large expressions. We also use a specialization rule with respect to the arguments of the function since this also improves the accuracy in many contexts. In the rest of this section, we give a brief description of each of these three transformation rules.

The first rule 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 4.2.

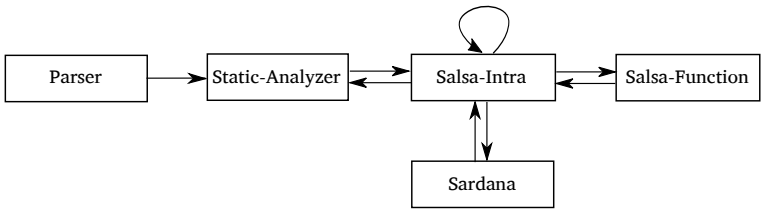

Figure 2: *Software architecture of* `Salsa`.

The second transformation rule is used when we deal with a small number of calls to a large function in the original program. The idea is 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 variable `u` of the function `f` 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 3.2. Note that we conserve the original function in our code when the condition on the variability is not satisfied.

The last rule 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 addition, in the calling site $z = f(e)$, we substitute $Var(e)$ to e in the arguments. For example, if $Var(e) = x$ the call becomes $z = f(x)$.

5 The Salsa Tool

In this section, we present our tool, `Salsa`. We detail its architecture, its inputs (files, parameters), its outputs and we describe the process followed by `Salsa` to improve the accuracy of programs given as inputs. After the presentation of `Salsa`, we give an example of application of our programs transformation.

5.1 Architecture

In this section, we describe the main architecture of our tool as shown in Figure 2. `Salsa` is based on the transformation described in Section 4. Written in `OCaml`, our tool is made of several modules. `Salsa` transforms the programs using the `Salsa-Intra`, `Salsa-Function` and `Sardana` modules.

- `Parser`: It takes programs in a C-like language with annotations (an example of program written in `Salsa` syntax is given in Figure 2), puts the program in SSA form and returns the syntactic tree,
- `Static-Analyzer`: It is based on abstract interpretation to compute the numerical accuracy of programs. The static analyzer infers safe ranges and computes error bounds for

each variable at each control point of the program for all the possible entries specified by intervals. To produce a more precise over-approximation of the maximum error, a work in progress consists of integrating the affine arithmetic [22] in our analyzer,

- `Sardana`: It improves the accuracy of arithmetic expressions [25]. `Sardana` takes as entry an arithmetic expression with free variables plus ranges for the free variables and returns a new expression more accurate than the original one and we assume that the variables belongs to the given ranges using the APEG described in Section 4.1,

- `Salsa-Intra`: It implements the intraprocedural transformation rules described in former work [13]. This module, takes a source program, calls the static analyzer to compute safe ranges for the variables at the current control point and then calls `Sardana` on the expression of the current control point with the ranges computed by the static analyzer,

- `Salsa-Function`: It implements the interprocedural transformation rules presented in Section 4.3 and detailed in [9]. This module transforms the functions as described in Section 4.3 (eg. by inlining, specialization, etc) and calls `Salsa-Intra` on the body of the transformed functions.

For each program transformed, `Salsa-Intra` re-starts the transformation while no more improvement of the accuracy is obtained. Finally, `Salsa` returns as output a file containing the transformed program with better accuracy and the accuracy improvement.

5.2 Inputs/Outputs

In this section, we specify the various inputs and outputs of `Salsa`. The tool takes as input a file that specifies ranges for the free variables of the program. These free variables are either introduced by the user to specialize the program for a class of executions or come from the sensors that transform physical measurements into digital data values in the case of embedded systems (typically these variables are declared `volatile` in `C` codes). These global variables are initialized thanks to the primitive `assert` at the beginning of each program (see Figure 3). In `assert` $id = [a, b, c, d]$, the interval used is of form $[a, b, c, d]$ where

- The first pair of values a, b consists of the range of floating-point value,

- The second pair c, d consists of the error range associated to the floating-point interval $[a, b]$.

Assertions are defined at the begining of the code. The end of the assertions section is marked by the keyword `%Salsa%`. Currently, `Salsa` optimizes one variable only, called the target variable. It consists of the variable returned by the main function of the program. `Salsa` depends on several parameters which tune its behavior. The main parameters are:

- `sliceSize`: This number corresponds to the height of the syntactic tree where we slice the arithmetic expressions. Basically `Salsa` builds very large expressions in order to have more opportunities to rewrite them into more accurate one. `sliceSize` is applied when we deal with very large expressions after inlining and substituting to avoid too large expressions in the optimized program. After slicing the expressions, we associate them to intermediary variables named `TMP`. For example if `sliceSize`$= 2$, the assignment $x = ((a+b)+c)+(d+e)$ is rewritten into `TMP1` $= (a+b)+c$; `TMP2` $= d+e$; $x =$ `TMP1`$+$`TMP2`,

- ω: This number gives the width of the intervals corresponding to the values of the arguments of a function. In our interprocedural transformation module, the intervals must contain at most $\omega = 2^4$ floating-point numbers. Our rule selector uses ω to determine if it performs function specialization or laziness. If the abstract values of the parameters have width less than ω then we use specialization rule otherwise we use laziness rule,

- `widen`: It consists of the number of iterations that we can do before applying the widening in the static analysis.

5.3 Example of Transformation

A `PID` Controller [2, 13] is a widely used algorithm in embedded and critical systems, like aeronautic and avionic systems. It keeps a physical parameter (m) at a specific value known as the *setpoint* (c). An original `PID` program is given in Figure 3. The function `integral` computes the integral of the error by the rectangle rule. The error being the difference between the *setpoint* and the measure, the controller computes a correction based on the integral i and derivative d of the error and also from a proportional error term p. We have

$$p = k_p \times e \ , \quad i = i + (k_i \times e \times dt) \ , \quad d = k_d \times (e - e_{\text{old}}) \times \frac{1}{dt} \ . \tag{9}$$

The weighted sum of these terms determines the reactivity, the robustness and the speed of the `PID` algorithm. If we take for example the `PID` program given in Figure 3, we have: the inputs are the measurement m and the setpoint c, the output the response r, the target variable is r that corresponds to the returned variable in the main function, the `sliceSize` parameter is equal to 5 and the `widen` parameter is equal to 10.

```
// Begin of assertions or global environment
assert m = [8.0,8.0, 0.0,0.0]
assert c = [5.0,5.0, 0.0,0.0]
// End of assertions
%Salsa%
// begin of Program
double main(){
  // Begin of initialization
  eold = 0.0 ; t = 0.0 ; i = 0.0 ; kp = 9.4514 ;
  kd = 2.8454 ; dt = 0.2 ; invdt = 5.0 ; m = 0.0 ;
  // End of initialization
  while (t < 100.0) {
    e = c - m ;
    p = kp * e ;
    i = integral(i,m,c,dt) ;
    d = kd * invdt * (e - eold) ;
    r = p + i + d ;
    eold = e ;
    m = m + r * 0.01 ;
    t = t + dt ;
  }
  return r ;              // Target variable
}
double integral(double ii, double mm,double cc,double ddt){
  ki = 0.69006 ;
  res = ii + (ki * ddt * (cc-mm)) ;
  return res ;
}
```

Figure 3: Salsa files.

Code	Original Abs. Error	Transformed Abs. Error	Original Rel. Error	Transformed Rel. Error	Transformation Time (s)
Odometry	5.55168e-14	2.13253e-14	7.7719e-16	3.7346e-16	0.047
PID Controller	4.66802e-14	2.13462e-16	1.6226e-15	7.4200e-17	0.121
Lead-Lag System	1.61083e-13	1.42663e-14	7.1038e-17	1.1145e-17	0.009
Runge-Kutta 4	1.26621e-15	2.77237e-16	7.1105e-16	1.6766e-16	0.022
Newton-Raphson	3.21484e-14	1.07444e-15	1.0937e-15	8.9536e-17	0.022
Simpson's Method	1.91875e-10	9.44669e-15	1.1605e-13	6.2320e-18	0.620
Rocket Trajectory	2.85693e-07	6.04012e-10	3.7310e-11	7.8882e-13	0.086

Table 1: *Improvement of the numerical accuracy of programs and their transformation time measurements.*

Code	Exec. time$_o$ (s)	Exec. time$_t$ (s)	%	Code size$_o$	Code size$_t$	$\frac{codesize_o}{codesize_t}$
Odometry	0.049	0.038	22.44	808	2.1 K	0.38
PID Controller	0.055	0.025	54.54	472	573	0.82
Lead-Lag System	0.049	0.046	6.12	618	752	0.82
Runge-Kutta 4	0.047	0.046	2.12	697	1.4 K	0.49
Newton-Raphson	0.059	0.047	20.33	465	1.1 K	0.42
Simpson's Method	0.082	0.051	37.80	648	4.2 K	0.15
Rocket Trajectory	0.055	0.047	14.54	1.6K	2.9 K	0.55

Table 2: *Execution time and code size measurements of programs.*

```
assert m = [8.0,8.0, 0.0,0.0]
assert c = [5.0,5.0, 0.0,0.0]
%Salsa%
double main() {
  t = 0.0 ; m = 0.0 ; eold = 0.0 ; i = 0.0 ;
  while (t < 100.0) {
    p = ((5.0 - m) * 9.4514) ;
    i = integral(i,m,5.0,0.2) ;
    d = (((5.0 - m) - eold) * 14.226999999999997) ;
    eold = (5.0 - m) ;
    r = ((p + i) + d) ;
    m = (m + (0.01 * r)) ;
    t = (t + 0.2) ;
  } ;
return r ;
}
double integral(double ii,double mm,double cc,double ddt) {
  res = (ii + ((0.69006 * (cc - mm)) * ddt)) ;
  return res ;
}
```

Figure 4: Transformed PID program.

After transformation, `Salsa` returns the program shown in Figure 4. `Salsa` asserts that the relative error of computation on r in the `PID` program is improved from $1.6226e^{-15}$ to $7.4200e^{-17}$ (see Table 1).

6 Experimentation

In this section, we aim at evaluating the performances of `Salsa` from various points of views. Obviously, we aim at evaluating how much the numerical accuracy is improved but also the impact on the code size, execution time and program transformation time. To do this, we have

taken a set of examples introduced in other articles and coming from different domains. We give hereafter, a brief description for each of them.

- `Odometry`: It consists of computing the position (x, y) of a two wheeled robot by odometry [13],
- `PID`: It keeps a physical parameter at a specific value known as the setpoint [13] as seen in Section 5.3,
- `Lead-Lag System`: It consists of a dynamical system including a mass and a spring which tries to move the mass from an initial position to a desired position [13],
- `Runge-Kutta 4`: It integrates an order 1 ordinary differential equation [13],
- `Newton-Raphson's method`: This numerical method is used to compute the successive approximations of the zeros of a real-valued function [10],
- `Simpson's method`: It is an improvement of the trapeze rule for numerical integration [12],
- `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 [11]. This code is larger than the former ones.

In Table 1, we give for each program:

- The upper bound of the original absolute error on the target variable obtained by static analysis as defined in Section 3.2,
- The upper bound of the absolute error on the target variable obtained after optimization,
- The original relative error obtained by dividing the absolute error by the upper bound of the interval of the value of the target variable,
- The relative error after transformation,
- The transformation time required to transform the programs.

The improvements of the relative errors on each code are also displayed in Figure 5. For example, if we take the `PID` controller previously presented in Section 5.3, its initial absolute error of computation is $4.66802e^{-14}$ while its absolute error of computation after transformation is equal to $2.13462e^{-16}$. We remark that the numerical accuracy of the `PID` Controller is improved by 99.54%. In other words, the absolute error of computation is reduced by 99.54%. If we are interested in measuring the time needed by `Salsa` to transform the `PID` Controller, we have that it takes 0.121 seconds. On a more complex example like the Rocket program, it takes 0.086 seconds to improve its accuracy for a total of more than 100 lines of code.

Beside accuracy measurements, we are also interested in examining the effects of `Salsa` on other criteria, related to performances of the transformed codes. In Table 2, we give for each program:

- The mean execution time in seconds obtained for `1000` runs of the original programs, $\texttt{Exec. time}_o$,
- The mean execution time after optimization obtained for `1000` runs for each program, $\texttt{Exec. time}_t$,

- The percentage of improvement of the execution time,
- The code size of the original program, `Code size`$_o$, in Bytes,
- The code size of the transformed program, `Code size`$_t$, in Bytes,
- The ratio between the code sizes of the original and transformed codes.

For execution time measurements, all the programs have been written in the `C` language and compiled with `GCC 4.9.2`, and executed on `Intel Core i7` in IEEE754 single precision in order to emphasize the effect of the finite precision. Programs are compiled with the default optimization level `-O2`. We have tried other levels of optimization without observing significant changes in our results. The results show that by transforming programs, we improve not only their accuracy but we reduce their execution time too. More precisely, if we take the `PID` Controller program, we shown in Table 2 that the execution time needed by the original program is 0.055 seconds while its execution time after being transformed is 0.025 seconds. It means that the execution time of the `PID` Controller is reduced by 54.54%. 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. By transforming programs, we may create new variables when we deal with large expressions that we associate to `TMP` variables. This explains the code size before, `code size`$_o$, and after transformation, `code size`$_t$ (in mean, the size of the optimized code is twice the size of the original). Despite increasing the size of program, the execution time of each program is widely reduced.

7 Conclusion

In this article, we have introduced `Salsa`, an automatic tool to improve the numerical accuracy of computations of intraprocedural and interprocedural programs by automatic transformation.

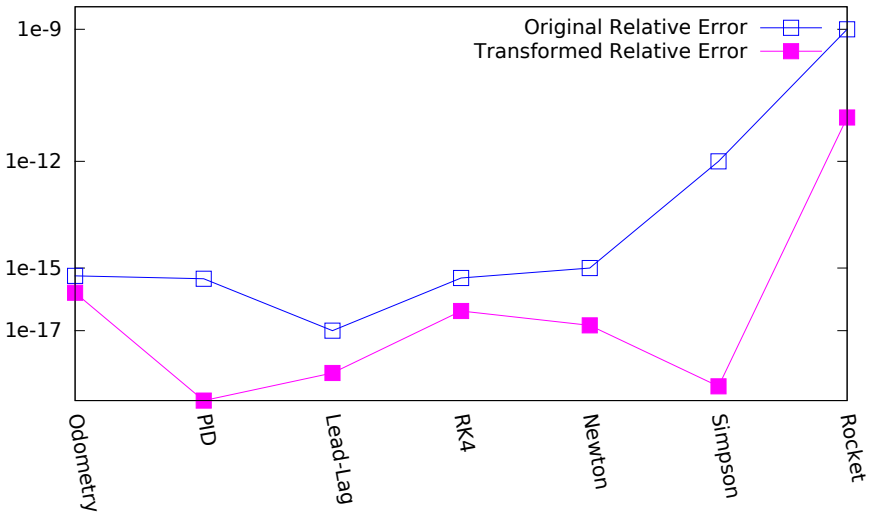

Figure 5: Improvements of the relative errors of the programs of Section 6.

We have detailed its architecture, and the different inputs and the outputs that it supports. We have tested `Salsa` across experimental results obtained on several examples coming from embedded systems and numerical algorithms. The results obtained show the efficiency of our tool on the numerical accuracy of computations which are improved by mean of 25.03%. Note that, this mean is a geometric mean. We have also shown that `Salsa` improves the execution time (by program specialization) but augments the code size (a factor 2 in mean). The time taken by `Salsa` to transform programs is also very short.

An interesting perspective consists of extending our work to optimize parallel programs. In this direction, we aim at solving new numerical accuracy problems like the order in an operation in a distributed system. We are interested also in studying the compromise between execution time, computation performances, numerical accuracy and the convergence acceleration of numerical methods. A key research direction is to study the impact of accuracy optimization on the convergence time of distributed numerical algorithms like the ones used usually for high performance computing. In addition, still about distributed systems, an important issue concerns the reproducibility of the results: Different runs of the same application yield different results on different machines due to the variations in the order of evaluation of the mathematical expression. We would like to study how our technique could improve reproducibility.

A significant interest would be to extend the current work with a case study concerning the optimization of several variables at the same time. In other words, we aim at improving the numerical accuracy of the program on several control points. That makes it possible to guess which variables are favorite to be optimized simultaneously without decreasing the accuracy of the other variables.

References

[1] ANSI/IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*. SIAM, 2008.

[2] K. J. Åström and T. Hagglund. *PID Controllers, 2nd ed.* Instrument Society of America, 1995.

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

[4] J. Bertrane, P. Cousot, R. Cousot, F. 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.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.

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

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

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

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

[10] N. Damouche, M. Martel, and A. Chapoutot. Impact of accuracy optimization on the convergence of numerical iterative methods. In M. Falaschi, editor, *LOPSTR*, volume 9527 of *LNCS*, pages 143–160. Springer, 2015.

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

[12] N. Damouche, M. Martel, and A. Chapoutot. Data-types optimization for floating-point formats by program transformation. In *CoDIT*. IEEE, 2016.

[13] 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.

[14] 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.

[15] E. Darulova and V. Kuncak. Sound compilation of reals. In S. Jagannathan and P. Sewell, editors, *POPL'14*, pages 235–248. ACM, 2014.

[16] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and V. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *FMICS*, volume 5825 of *LNCS*, pages 53–69. Springer, 2009.

[17] 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.

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

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

[20] 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.

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

[22] E. Goubault and S. Putot. Under-approximations of computations in real numbers based on generalized affine arithmetic. In Hanne Riis Nielson and Gilberto Filé, editors, *SAS*, volume 4634 of *LNCS*, pages 137–152. Springer, 2007.

[23] E. Hankin. *Lambda Calculi A Guide For Computer Scientists*. Clarendon Press, Oxford, 1994.

[24] 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.

[25] A. Ioualalen and M. Martel. Sardana: an automatic tool for numerical accuracy optimization. *SCAN: Scientifc Computing, Computer Arithmetic and Validated Numerics*, pages 1–4, 2012. al-00698619.

[26] 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.

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

[28] 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, 2010.

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

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

[31] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: A new approach to optimization. *Logical Methods in Computer Science*, 7(1), 2011.