POP: A Tuning Assistant for Mixed-Precision Floating-Point Computations

Dorra Ben Khalifa[1(✉)], Matthieu Martel[1,2], and Assalé Adjé[1]

[1] LAMPS Laboratory, University of Perpignan, 52 Av. P. Alduy, Perpignan, France
{dorra.ben-khalifa,matthieu.martel,assale.adje}@univ-perp.fr
[2] Numalis, Cap Omega, Rond-point Benjamin Franklin, Montpellier, France

Abstract. In this article, we describe a static program analysis to determine the lowest floating-point precisions on inputs and intermediate results that guarantees a desired accuracy of the output values. A common practice used by developers without advanced training in computer arithmetic consists in using the highest precision available in hardware (double precision on most CPU's) which can be exorbitant in terms of energy consumption, memory traffic, and bandwidth capacity. To overcome this difficulty, we propose a new precision tuning tool for the floating-point programs integrating a static forward and backward analysis, done by abstract interpretation. Next, our analysis will be expressed as a set of linear constraints easily checked by an SMT solver.

Keywords: Floating-point arithmetic · Mixed precision · Forward and backward error analysis · Constraints generation · SMT solver

1 Introduction

With the wide availability of processors with hardware floating-point units, many current critical applications, such as the critical control command systems for automotive, aeronautic, space, etc., which have stringent correctness requirements and whose failures have catastrophic consequences that endanger human life [1,9], rely heavily on floating-point operations. Without any extensive background in numerical accuracy and computer arithmetic, developers tend to use the highest precision available in hardware (usually double precision). Despite the fact that the results will be more accurate, this increases significantly the application runtime, bandwidth capacity and the memory and energy consumption of the system. In fact, we denote by the term *precision* the amount of information used to represent a value while the term *accuracy* denotes how close a floating-point computation comes to the real value. The challenge is to use no more precision than needed wherever possible without compromising overall accuracy (using a too low precision for a given algorithm and data set leads to inaccurate results). To overcome the problem of determining the accuracy of floating-point computations, many efforts have been done in automating

O. Hasan and F. Mallet (Eds.): FTSCS 2019, CCIS 1165, pp. 77–94, 2020.
https://doi.org/10.1007/978-3-030-46902-3_5

the choice of the best precision by dynamic or static methods [5,10,15,16] but they differ strongly in their way of accuracy determination. In this article, we are interested in the problem of determining the minimal precision on the inputs and the intermediary results of a program performing floating-point computations in order to get a desired accuracy on the outputs. Often in these programs, it is possible to reduce the floating-point precision of certain variables in order to increase performance, for example, the throughput of single-precision floating-point operations is twice that of double-precision operations. Also, the proposed tool in this article aims to apply the mixed-precision on the floating-point programs formats. Mixed-precision computing [10] is an approach to combine different precisions for different floating-point variables (contrarily to the uniform precision). Our approach combines a forward and a backward error analysis which are two popular paradigms of error analysis, done by abstract interpretations [3]. In fact, the forward analysis is classical. It examines how errors are magnified by each operation aiming to determine the accuracy on the results [11]. Next, a user requirement is given denoting the final accuracy wanted on some control points of the outputs. By taking in consideration the user assertions and the results of the forward analysis, the backward analysis is a complementary approach that starts with the computed answer to determine the exact floating-point input that would produce it in order to satisfy the desired accuracy. As could be expected, the forward and backward analysis can be handled iteratively to refine the results until a fixed-point is reached. Next, these forward and backward transfer functions are expressed as a set of linear constraints made of propositional logic formulas and relations between integer elements only. After, these constraints will be easily checked by an SMT solver (`Z3` is used in practice [7]).

The main contributions of this article are the following. First, we introduce refinements of the automated approach based on a static forward and backward analysis done in [11]. This approach will be explained in details specially for the cases of addition, the multiplication and the subtraction arithmetic expressions. Furthermore, our contribution revolves around the definition of the function ι, defined in [11] and redefined further in this work (see Fig. 2). The function ι is equivalent to the carry bit that can occur throughout floating-point computations (generally $\iota = 1$). Intuitively, a too conservative static analysis would consider that a carry can be propagated at each operation, which corresponds to $\iota = 1$. This function becomes very costly if we perform several computations at a time and therefore the errors would be considerable. It is then crucial to use the most precise function ι. This is why, we reexamine in this work this function by sorting out the different cases where this function might be equal to 1 or 0: difference in magnitude of two floating-point numbers and the superposition of the *ulp* and the *ufp*, defined in Sect. 3.1, of these two numbers relative to each other. After that, the previous analysis will be expressed as a set of propositional formulas on linear constraints between integer variables only (checked by `Z3`). The transformed program is guaranteed to use variables of lower precision with a minimal number of bits than the original program. Second, we present the steps of construction of our new tool, `POP`, which executes and evaluates any

kind of programs with respect to our grammar of a simple imperative language and including the implementation of the proposed approach. Also, we present some experimental results showing the efficiency of our mixed-precision tool in determining the minimal precision required.

The rest of this article is organized as follows. Section 2 introduces briefly some basic concepts related to the floating-point arithmetic and the related work of some existing precision tuning tools and we finish by introducing the overview of our approach. Section 3 deals with the forward and backward static error analysis by constraints generation with some examples. The implementation of our tool and the constraints resolution are presented in Sect. 4 and an experimental results are given in Sect. 5 before concluding in Sect. 6.

2 Overview

To better explain what POP does, a motivating example of a floating-point program is given in Fig. 1 which implements a simple scalar product of two vectors x and y presented with different magnitude of small and large floating-point values. For the vectors x and y, the variable values belong to $[1.0, 2.0]$, $[10.0, 15.0]$ and $[100.0, 110.0]$ for vector x and $[100.0, 110.0]$, $[5.0, 10.0]$ and $[450.0, 500.0]$ for vector y, respectively. In this example, we suppose that all variables are in double precision before analysis (original program in the left hand side of Fig. 1) and that a range determination is performed by dynamic analysis on these variables to make sure that no overflow can arise. We generate at each node of our program syntactic tree a unique control point in order to determine easily the final accuracy, after the forward and backward analysis, as shown on the left side of Fig. 1. It is conceivable that our program contains several annotations. First, for example on the left hand side of Fig. 1, the variables $\mathtt{x}_1$ and $\mathtt{y}_1$ are initialized to the abstract values $[1.0, 2.0]$ and $[10.0, 15.0]$ (in double precision) respectively, annotated with their control points thanks to the following annotations $\mathtt{x}_1^{|1|} = [1.0, 2.0]^{|0|}$ and $\mathtt{y}_1^{|3|} = [10.0, 15.0]^{|2|}$. As well, we have the statement

$$\mathtt{require\_accuracy}(\mathtt{v}, 23)^{|40|}$$

which informs the system that the user wants to turn on variable v to the simple precision at this control point. As a consequence, the minimal precision needed for the inputs and intermediary results satisfying the user assertion is observed on the right side of Fig. 1. For example, the variables $\mathtt{x}_1$ passed from the double into float precision thanks to the annotation $\mathtt{x}_1^{\sharp 21} = [1.0, 2.0]^{\sharp 22}$ (a floating-point number in single precision has 22 accurate digits). The results obtained show that POP, for present, automates precision tuning and propagates the user requirement along the program inputs and intermediary results.

3 Preliminary Notions

This section provides some background on the IEEE754 Standard of floating-point arithmetic, formats, rounding modes, errors and the *ufp* and *ulp* functions. Noting that several definitions of *ulp* exist in literature [12].

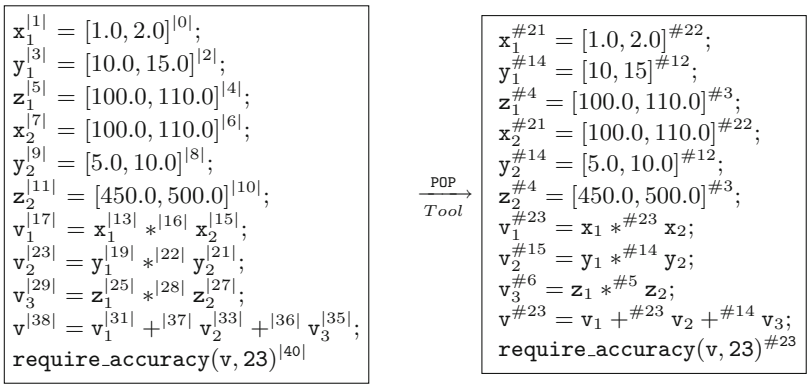


Fig. 1. Simple scalar product of two vectors program. The program on the left designs the initial program in double precision annotated with labels. On the right, the program after analysis annotated with the final accuracies at each label referring to the user requirement.

3.1 Basics on Floating-Point Arithmetic

The IEEE754 Standard formalizes a binary floating-point number x in base β (generally $\beta = 2$) as a triplet made of a sign, a mantissa and an exponent as shown in Eq. (1), where $s \in \{-1,1\}$ is the sign, m represents the mantissa, $m = d_0.d_1...d_{p-1}$, with the digits $0 \leq d_i < \beta$, $0 \leq i \leq p-1$, p is the precision (length of the mantissa) and the exponent $e \in [e_{min}, e_{max}]$.

$$x = s.m.\beta^{e-p+1} \tag{1}$$

Table 1. Parameters defining basic format floating-point numbers

Format	Name	Mantissa size (p - 1)	Size of e	e_{min}	e_{max}
Binary16	Half precision	10	5	−14	+15
Binary32	Single precision	23	8	−126	+127
Binary64	Double precision	52	11	−1122	+1223
Binary128	Quadruple precision	112	15	−16382	+16383

The IEEE754 Standard specifies some particular values for p, e_{min} and e_{max} [4]. Also, this standard defines binary formats (with $\beta = 2$) which are described in Table 1. Hence, the IEEE754 standard distinguishes between normalized and denormalized numbers. Indeed, the normalization of a floating-point number ensuring $d_0 \neq 0$ guarantees the uniqueness of its representation. Denormalized numbers make underflow gradual [13]. The IEEE754 standard defines also some special numbers. All these numbers are summarized in Table 2 (in Binary64).

Moreover, the IEEE754 Standard defines four rounding modes for elementary operations over floating-point numbers which are: towards $+\infty$, towards $-\infty$, towards zero and towards the nearest denoted by $\uparrow_{+\infty}$, $\uparrow_{-\infty}$, $\uparrow_0$ and $\uparrow_\sim$, respectively. Henceforth, we present the *ufp* (unit in the first place) and *ulp* (unit in the last place) functions which express the *weight of the most significant bit* and the *weight of the least significant bit*, respectively. In practice, these functions will be used further in this article to describe the error propagation across the computations. The definition of these functions is given in Eqs. (2) and (3) defined in [11].

$$ufp(x) = \min\{i \in \mathbb{Z} : 2^{i+1} > x\} = \lfloor \log_2(x) \rfloor \tag{2}$$

Let p be the size of the significand, the *ulp* of a floating-point number can be expressed as shown:

$$ulp(x) = ufp(x) - p + 1. \tag{3}$$

Table 2. Numbers in double precision

x	Exponent e	Mantissa m
$x = 0$ (if s = 0) $x = -0$ (if s = 1)	$e = 0$	$m = 0$
Normalized numbers $x = (-1)^s \times 2^{e-1023} \times 1.m$	$0 < e < 2047$	any
Denormalized numbers $x = (-1)^s \times 2^{e-1022} \times 0.m$	$e = 0$	$m \neq 0$
$x = +\infty$ (if s = 0) $x = -\infty$ (if s=0)	$e = 2047$	$m = 0$
$x = NaN$ (Not a Number)	$e = 2047$	$m \neq 0$

3.2 Related Work

There have been many efforts to automate the process of determining the best floating-point formats. Darulova and Kuncak [5] proposed a static analysis method to compute errors propagation. If their computed bound on the accuracy satisfies the post-conditions then the analysis is run again with a smaller format and it stops until finding the best format. Contrarily to our proposed tool, all their values have the same format (uniform-precision). Other methods rely on dynamic analysis. By way of illustration, Precimonious is considered as a dynamic automated search based tool that evaluates and executes different mixed-precision configurations of the program to identify the best configuration that satisfies the error threshold [15]. Also, we mention the Blame Analysis [16], a novel dynamic method that speeds up precision tuning by combining concrete and shadow program execution. The analysis determines the precision of all operands such that a given precision is achieved in the final result. So as to be more efficient with significant reduction in analysis time than used by itself,

Blame Analysis and Precimonious has been consolidated together and this combined approach has shown better results in term of program speedup compared to using Blame Analysis alone. Nonetheless, floating-point tuning of entire applications is not feasible yet, in this moment, by this method. Moreover, Lam et al. [10] instrument binary codes aiming to modify their precision without modifying the source codes. They also propose a dynamic search method to identify the parts of code where the precision should be modified. The major drawback of this tools is that the state space is exponential in the number of variables and exploring even a subset is very time-intensive.

Finally, there are various rigorous static analysis approaches that use interval and affine arithmetic or Taylor series approximations to analyze stability and to provide rigorous bounds on rounding errors. However, they do not scale very well and therefore have not been applied to high precision computing workloads. In this context, Chiang et al. [2] has proposed an approach which allocate a precision to the terms of only arithmetic expressions. Whereas they need to solve a quadratically constrained quadratic program to obtain their annotations. Also, Solovyev et al. [17] have proposed the FP-Taylor tool that implements a method to estimate round-off errors of floating-point computations called Symbolic Taylor Expansions.

4 Static Analysis by Constraints Generation

In this section, we refine the computations of the forward and backward transfer functions used by the `POP` tool for the cases of addition, product and subtraction done in [11]. These functions are defined using the *unit in the first* and *last places* introduced in Eqs. (2) and (3). Next, these functions will be formalized as a set of constraints made of propositional logic formulas and affine expressions among integers.

4.1 Forward and Backward Error Analysis

Forward Addition, Multiplication and Subtraction. Consequently, we introduce the forward transfer functions corresponding to the addition $\overrightarrow{\oplus}$, product $\overrightarrow{\otimes}$ and subtraction $\overrightarrow{\ominus}$ of two floating-point numbers $x \in \mathbb{F}_p$ and $y \in \mathbb{F}_q$ where $\mathbb{F}_p$ and $\mathbb{F}_q$ denote two sets of floating-point numbers in accuracy p and q, respectively. In Eq. (4), the operands $x_{p_{p'}}$ and $y_{q_{q'}}$ and their results $z_{r_{r'}}$ have respectively two parameters p, p', q, q' and r, r' which denote the correct precision of the result and of the error, respectively. Other than that, in distinction to [11], we introduce the truncation errors in order to be more precise through our computations. We denote the truncation errors by ε_+, $\varepsilon_\times$ and ε_- for the addition, product and subtraction operations respectively.

Definition 1. *The forward addition* $\overrightarrow{\oplus}$ *is given as shown in Eq. (4):*

$$\overrightarrow{\oplus}(x_{p_{p'}}, y_{q_{q'}}) = z_{r_{r'}} \quad where \quad r = ufp(x_{p_{p'}} + y_{q_{q'}}) - ufp(2^{ufp(x_{p_{p'}})-p+1} + 2^{ufp(y_{q_{q'}})-q+1} + 2^{ufp(z_{r_{r'}})-\sigma_+}) \tag{4}$$

In the sequel, we assume $x_{p_{p'}} = x$, $y_{q_{q'}} = y$ and $z_{r_{r'}} = z$. Let v be an exact value computed in infinite precision and the floating-point value is such that $\widehat{v} = d_0.d_1...d_{p-1}.2^e$ of $\mathbb{F}_p$. The comparison of these two values is $|v - \widehat{v}| \leq 2^{e-p+1}$. So, taking into account the definition of the function *ufp* in Eq. 2, we have for any $x \in \mathbb{F}_p$ and $y \in \mathbb{F}_q$ the error ε_x on x is bounded by:

$$\varepsilon_x < 2^{ulp(x)} = 2^{ufp(x)-p+1} \quad \text{and} \quad \varepsilon_y < 2^{ulp(y)} = 2^{ufp(y)-q+1} \tag{5}$$

The truncation error for the rounding mode towards the nearest $\uparrow_\sim$ defined by the IEEE754 Standard for the addition of x and y whose result is z is given by $\varepsilon_+ \leq 2^{\frac{1}{2}ulp(z)}$ and we have $ulp(z) = ufp(z) - \sigma_+ + 1$ where σ_+ presents the precision of the operator $+$. Thus, the truncation error is shown in Eq. 6:

$$\varepsilon_+ \leq 2^{ufp(z)-\sigma_+} \tag{6}$$

Definition 2. *The forward product* $\overrightarrow{\otimes}$ *is given as shown in Eq. (7):*

$$\overrightarrow{\otimes}(x_{p_{p'}}, y_{q_{q'}}) = z_{r_{r'}} \quad where \quad r = ufp(x_{p_{p'}} \times y_{q_{q'}}) - ufp(2^{ufp(x)+1}.2^{ufp(y)-q+1} + 2^{ufp(y)+1}.2^{ufp(x)-p+1} + 2^{ufp(x)-p+1}.2^{ufp(y)-q+1} + 2^{ufp(z)-\sigma_\times}) \tag{7}$$

We assume that the error $\varepsilon_{z_\times}$ of the multiplication of two floating-point numbers x and y whose result is z is $\varepsilon_{z_\times} = y \,.\, \varepsilon_x + x \,.\, \varepsilon_y + \varepsilon_x \,.\, \varepsilon_y + \varepsilon_\times$ where $\varepsilon_\times$ is the truncation error for the product and is equal to $\varepsilon_\times \leq 2^{ufp(z)-\sigma_\times}$ (for the rounding mode towards $\uparrow_\sim$) and where $\sigma_\times$ represents the precision of the operator $\times$. So, the error $\varepsilon_{z_\times}$ could be bounded as shown in Eq. 8:

$$2^{ufp(x)} \leq x < 2^{ufp(x)+1} \quad \text{and} \quad 2^{ufp(y)} \leq y < 2^{ufp(y)+1}$$

and consequently,

$$\begin{aligned}\varepsilon_{z_\times} &< 2^{ufp(x)+1}.2^{ufp(y)-q+1} + 2^{ufp(y+1}.2^{ufp(x)-p+1} + 2^{ufp(x)-p+1}.2^{ufp(y)-q+1} + 2^{ufp(z)-\sigma_\times} \\ &< 2^{ufp(x)+ufp(y)-q+2} + 2^{ufp(x)+ufp(y)-p+2} + 2^{ufp(x)+ufp(y)-p-q+2} + 2^{ufp(z)-\sigma_\times}\end{aligned}$$

thus,

$$\varepsilon_{z_\times} \leq 2^{ufp(x)+ufp(y)-q+1} + 2^{ufp(x)+ufp(y)-p+1} + 2^{ufp(x)+ufp(y)-p-q+1} + 2^{ufp(z)-\sigma_\times}. \tag{8}$$

Definition 3. *The forward subtraction* $\overrightarrow{\ominus}$ *is given as shown in Eq. (9):*

$$\overrightarrow{\ominus}(x_{p_{p'}}, y_{q_{q'}}) = z_{r_{r'}} \quad where \quad r = ufp(x_{p_{p'}} - y_{q_{q'}}) - ufp(2^{ufp(x)-p+1} - 2^{ufp(y)-q+1} - 2^{ufp(z)-\sigma_-}) \tag{9}$$

Using the same approach in the addition case, we have $2^{ufp(x)} \leq x < 2^{ufp(x)+1}$ and $2^{ufp(y)} \leq y < 2^{ufp(y)+1}$ and the truncation error $\varepsilon_- \leq 2^{ufp(z)-\sigma_-}$ where σ_- is the precision of the operator -. The subtraction error between x and y is bounded as mentioned in Eq. (9).

Backward Addition, Subtraction and Multiplication. Equivalently, we introduce the backward transfer functions $\overleftarrow{\oplus}$, $\overleftarrow{\otimes}$ and $\overleftarrow{\ominus}$ which take advantage of the forward transfer functions and of the accuracy requirement on the results and by combining these two findings it is then possible to lower the number of bits needed for one of the operands. We consider that x is unknown where the result z and the operand y are known. The backward functions for the proposed arithmetic functions are given in the following properties.

Definition 4. *The backward transfer function for the addition* $\overleftarrow{\oplus}$ *is given as shown:*

$$\overleftarrow{\oplus}(z,y) = (z-y)_{p_{p'}} \quad with \quad p = ufp(z-y) - ufp(2^{ufp(z)-r+1} - 2^{ufp(y)-q+1} - 2^{ufp(x)-\sigma_+}) \tag{10}$$

To apply the backward analysis, we assume that one of the operands is unknown (x in our case) while the result z is known. Then, we compute the precision p of the operand x with respect to the user accuracy requirement and the forward analysis result. As we said, the result and the operand errors can be bounded by $\varepsilon_{z_+} < 2^{ufp(z)-r+1}$ and $\varepsilon_y < 2^{ufp(y)-q+1}$ and for the truncation error is given as $\varepsilon_+ \leq 2^{ufp(x)-\sigma_+}$.

Definition 5. *We present the backward transfer function for the multiplication* $\overleftarrow{\otimes}$ *as shown:*

$$\overleftarrow{\otimes}(z,y) = (z \div y)_{p_{p'}} \quad with$$
$$p = ufp(z \div y) - ufp\Big(\frac{2^{ufp(y)+1}.2^{ufp(z)-r+1} - 2^{ufp(z)+1}.2^{ufp(y)-q+1}}{2^{ufp(y)+1}(2^{ufp(y)+1} + 2^{ufp(y)-q+1}} - 2^{ufp(x)-\sigma_\times}\Big) \tag{11}$$

In the case of product, we know that $\overleftarrow{\otimes}(z,y) = (z \div y)_{p_{p'}}$ with $p = ufp(z \div y) - ufp(\varepsilon_{z_\times})$ and where the truncation error $\varepsilon_\times \leq 2^{ufp(x)-\sigma_-}$ and the error $\varepsilon_{z_\times}$ is bounded as it is shown in Eq. (11).

Definition 6.

$$\overleftarrow{\ominus}(z,y) = (z+y)_{p_{p'}} \quad with \quad p = ufp(z+y) - ufp(2^{ufp(z)-r+1} + 2^{ufp(y)-q+1} + 2^{ufp(x)-\sigma_-}) \tag{12}$$

We know that the roundoff errors are bounded as $\varepsilon_z < 2^{ufp(z)-r+1}$ and $\varepsilon_y < 2^{ufp(y)-q+1}$ and the truncation error $\varepsilon_- \leq 2^{ufp(x)-\sigma_-}$ where σ_- denotes the precision of the operator - and the error in Eq. (12) is given as $\varepsilon_{z_-} = \varepsilon_x - \varepsilon_y - \varepsilon_-$.

Obviously, our static analysis does not work on scalar values as in Eqs. (4) to (12) but on intervals instead. As described in [11], we abstract sets of values of $\mathbb{F}_p$

using the following connection in Eq. (13) where an element $i^\sharp \in \mathbb{I}_p$ correspond to $i^\sharp = [\underline{f}, \overline{f}]_p$ is defined by two floating-point numbers and an accuracy p.

$$\mathbb{I}_p \ni [\underline{f}, \overline{f}]_p = \{f \in \mathbb{F}_p : \underline{f} \leq f \leq \overline{f}\} \quad with \quad \mathbb{I} = \bigcup_{p \in \mathbb{N}} \mathbb{I}_p. \tag{13}$$

The operations $\overrightarrow{\oplus}^\sharp$, $\overleftarrow{\oplus}^\sharp$, $\overrightarrow{\otimes}^\sharp$ and $\overleftarrow{\otimes}^\sharp$ among values of $\mathbb{I}_p$ are defined in [11] in function of $\overrightarrow{\oplus}$, $\overleftarrow{\oplus}$, $\overrightarrow{\otimes}$ and $\overleftarrow{\otimes}$. For the rest of the article, we deal with the generation of constraints only for the addition and the product.

4.2 Constraints Generation

In this section, we describe how to generate constraints to determine the lowest precision on variables and intermediary values in programs. An important definition of the function ι, computed on floating-point numbers, is given in this section. By this definition, we attempt to be far more efficient in the way we propagate errors across the arithmetic operations. The methodical difference between the function $\iota(u, v)$ proposed in [11] and our new definition $\iota(t, u, v, w)$ is that we take in consideration the *ufp* and *ulp* of the two operands in order to compare the two floating-point number errors α and β and we add an extra bit only if we are certain that $ulp(\alpha)$ is lesser than the $ufp(\beta)$ (0 otherwise). Compared to the former definition of [11], our new definition improves significantly the accuracy of the static analysis by being less pessimistic. As mentioned earlier, the transfer functions previously seen in Sect. 4.1 are not translated directly into constraints because the resulting system would be too difficult to solve and contain non-linear constraints. Therefore, we reduce the problem to a constraint system consisting in propositional formulas on linear relations between integer elements only. In what follows, we introduce the constraints that we generate for the arithmetic expressions in which we are interested.

Forward Operations. Back to Eqs. (4) to (12), our goal is to compute the correct precision r and the precision r' of the result error (ε_{z_+} for the addition and $\varepsilon_{z_\times}$ for the product) for the floating-point number z. Intuitively, we compute $z = x + y$ with related errors ε_x and ε_y and ε_{z_+} and we want to compute $ufp(\varepsilon_{z_+})$ in function of the errors on the operands.

Proposition 1. *Let x in $\mathbb{F}_p$ and y in $\mathbb{F}_q$ and let z the result of the addition operation between these two floating-point numbers. We have in the worst case a carry bit that can occur through this operation as it has been proven in [12].*

$$ufp(z) \leq max(ufp(x), ufp(y)) + 1 \tag{14}$$

As a matter of fact, the previous Eq. (14) is considered as correct but pessimistic (too large over-approximation) due to the fact that adding an extra bit specially for cases we would not to, becomes very costly if we perform several computations. In previous work [11], a new function ι was presented in order to refine Eq. (14): they compare the unit in the first places of the operands and they add

an extra bit only if they are equal which is correct but it misses exactness. In this work, we present our new definition of function ι. In fact, let x in $\mathbb{F}_p$ and y in $\mathbb{F}_q$, our strategy is to compare $ulp(x)$ with $ufp(y)$ and conversely ($ulp(y)$ with $ufp(x)$). In Definition 7, we present function ι and in Fig. 2 we present an example of cases of function ι where an extra bit can occur ($\iota = 1$) or not ($\iota = 0$).

Definition 7. *We introduce the function $\iota(t, u, v, w)$ as the exceeding of 1 bit that can occur in operations between the floating-point numbers.*

$$\iota(t, u, v, w) = \begin{cases} 0 & u > t \quad or \quad w > u, \\ 1 & otherwise. \end{cases} \tag{15}$$

Proposition 2. *In order to compute the function ι, we need to compute the unit in the last places $ulp(\alpha)$ and $ulp(\beta)$. Considering p' the precision of α, from Eq. (3) we have $ulp(\alpha) = ufp(\alpha) - p' + 1$ where $ufp(\alpha) = ufp(x) - p$. Consequently, we obtain that:*

$$ulp(\alpha) = ufp(x) - p - p' + 1 \tag{16}$$

We know from Eq. (3) that for $x \in \mathbb{F}_p$ the unit in the last place is $ulp(x) = ufp(x) - p + 1$. This definition is also valid for $ulp(\varepsilon_x)$ with p' the precision of ε_x and also we deduce that if $ulp(x) = ufp(x) - p + 1$ than $ufp(\varepsilon_x) = ufp(x) - p$ and than we obtain the result in Eq. (16).

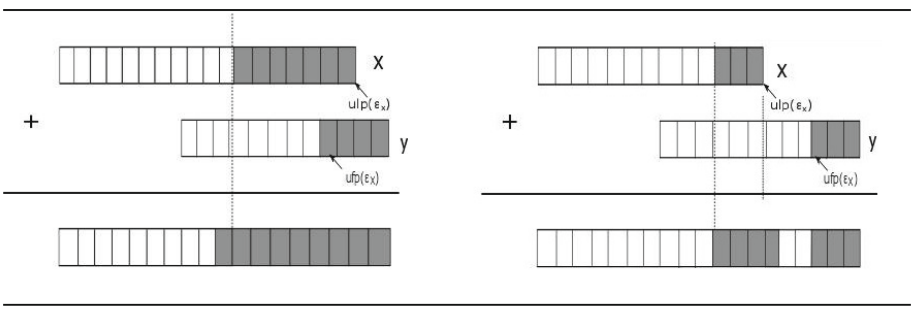


Fig. 2. Definition of function ι. The figure on the left represents the case of $\iota(\alpha, \beta) = 1$ and so an exceeding bit can occur throughout computations. The figure on the right is equivalent to $\iota(\alpha, \beta) = 0$

Forward Addition: From Definition 7, Eq. (5) and Eq. (14), we present Proposition (3). As we said before, if we sum $z = x + y$ the error is equal to $\varepsilon_{z_+} = \varepsilon_x + \varepsilon_y + \varepsilon_+$. Now, in order to apply the definition of the function ι, we will disassociate the total error ε_{z_+} into two errors: the roundoff error $\varepsilon_{xy} = \varepsilon_x + \varepsilon_y$ and the truncation error ε_+. Also, we will manage by presenting one case of the ι function ($u > t$).

Proposition 3. *Let $a = ufp(x)$, $b = ufp(y)$ and $c = ufp(z)$,*

$$ufp(\varepsilon_{xy}) < \max(a - p + 1, b - q + 1) + \iota(a - p - p' + 1, b - q) \tag{17}$$

Taking into account Eq. (17) above, $ufp(\varepsilon_{z_+})$ is then bounded by:

$$ufp(\varepsilon_{z_+}) < \max\left(\max\left((a-p+1, b-q+1) + \iota(a-p-p'+1, b-q), c-\sigma_+\right)\right) + \iota(a-p-p'+1, b-q) \tag{18}$$

which implies that the precision of the result z in this addition is

$$r = ufp(x+y) - \max\left(\max\left((a-p+1, b-q+1) + \iota(a-p, b-q), c-\sigma_+\right)\right) - \iota(a-p-p'+1, b-q). \tag{19}$$

Proof. Formally, let $\alpha = \sum_{i=n_0}^{n_1} \alpha_i 2^i$ and $\beta = \sum_{i=m_0}^{m_1} \beta_i 2^i$ two floating-point numbers. Let us assume that $n_1 < m_0$. From Definition 7, we have $ufp(\alpha) = n_1$ and $ulp(\beta) = n_0$ then:

$$\alpha + \beta = \sum_{i=n_1}^{m_0} \gamma_i 2^i \quad where \quad \gamma_i = \begin{cases} \alpha_i & \text{if} \quad i \in [n_0, n_1], \\ \beta_i & \text{if} \quad i \in [m_0, m_1] \\ 0 & \text{otherwise.} \end{cases}$$

Finally, we conclude that $ufp(\varepsilon_{z_+}) = m_1$. In the case where $n_0 > m_1$, we deduce that $ufp(\varepsilon_{z_+}) = n_1$. After, from Eq. (18), we substitute the new refinement over-approximation of the total error ε_{z_+} and consequently we deduce the precision r in Eq. (19).

Now, what remains to be done is to determine the precision of the error r' of the addition. That's why, we need to compute $ulp(\epsilon_{z_+})$ as it is shown in Eq. (20). In the case of addition, we present $ulp(\varepsilon_{z_+})$ as the smallest *ulp* between the two operands errors ($ulp(\varepsilon_x)$ and $ulp(\varepsilon_y)$) and we conclude finally that the precision of the error $r' = ufp(\epsilon_{z_+}) - ulp(\epsilon_{z_+})$.

$$ulp(\varepsilon_{z_+}) = min\big(ulp(\varepsilon_x), ulp(\varepsilon_y)\big) \tag{20}$$

Forward Multiplication. For the multiplication case, we apply our new Definition 7 and Eq. (8) and we present Proposition 4.

Proposition 4. *Let a and b and c three integers with $a = ufp(x)$, $b = ufp(y)$ and $c = ufp(z)$. We apply the same proceeding as in the forward addition, we dissociate the total error $\varepsilon_\times$ into the roundoff error $\varepsilon_{xy} = \varepsilon_x + \varepsilon_y$ and the truncation error $\varepsilon_\times$. So, we have:*

$$ufp(\varepsilon_{xy}) < \max(a + b - p + 1, a + b - q + 1) + \iota(a - p - p' + 1, b - q) \tag{21}$$

and then the total error $ufp(\varepsilon_{z\times})$ is given as

$$ufp(\varepsilon_{z_\times}) < \max\left(\max\left((a-p+1, b-q+1) + \iota(a-p-p'+1, b-q), c-\sigma_+\right)\right) + \iota(a-p-p'+1, b-q) \tag{22}$$

and then we deduce that

$$r = ufp(x \times y) - \max\left(\max\left((a-p+1, b-q+1) + \iota(a-p, b-q), c-\sigma_+\right)\right) - \iota(a-p-p'+1, b-q). \tag{23}$$

Next, like we have proceed in Eq. (20) in the case of addition we may say that the unit in the last place of $\varepsilon_{z_\times}$ is defined by

$$ulp(\varepsilon_{z_\times}) = ulp(\epsilon_x) + ulp(\epsilon_y) \tag{24}$$

By reasoning in the same way, we linearize the computations for the backward operations (addition and multiplication).

Backward Addition: We consider now the backward transfer functions, depending on Eq. (10) for the addition case. We know that $p = ufp(z-y) - ufp(\varepsilon_z - \varepsilon_y - \varepsilon_+)$. So, again let $c = ufp(z)$ we can over-approximate ε_z thanks to the relations $\varepsilon_z < 2^{c-r+1}$, $\varepsilon_y \geq 0$ and $\varepsilon_+ \geq 0$ and consequently

$$p = ufp(z-y) - c + r. \tag{25}$$

Backward Multiplication: Again, we take $a = ufp(x)$, $b = ufp(y)$ and $c = ufp(z)$. From Eq. (11), we know that $2^c \leq z < 2^{c+1}$, $2^b \leq y < 2^{b+1}$ and $\varepsilon_{z_\times} < 2^{c-r+1}$, $\varepsilon_y < 2^{b-q+1}$ which implies that $y.\varepsilon_{z_\times} - z.\varepsilon_y < 2^{c+b-r+2} - 2^{b+c-q+2}$ and that

$$\frac{1}{y.(y+\varepsilon_y)} < 2^{-2b}.$$

Consequently,

$$\varepsilon_{z_\times} \leq 2^{-2b}.(2^{c+b-r+2} - 2^{b+c-q+2}) - 2^{a-\sigma_\times} \leq 2^{c-b-r+1} - 2^{c-b-q+1} - 2^{a-\sigma_\times}$$

and finally,

$$p = ufp(z \div y) - \max(\max(c-b-r+1, c-b-q+1), a-\sigma_\times). \tag{26}$$

5 The POP Tool

In this section, we present our tool, `POP`: Precision OPtimizer. We present its architecture, its input including the program file annotated with the developer accuracy expectation, parameters and its outputs. Also, we illustrate the mechanism followed by `POP` to lower the precision of the floating-point programs.

5.1 Architecture

At this stage, we present the main architecture of `POP` also described in Fig. 3. `POP` is written in `JAVA` while each expression, boolean and statement presented in Fig. 4 are represented as packages gathering the different classes of their definition. We can illustrate the tool hierarchy as follows:

- **Parser:** It takes a file of a floating-point program referring to our simple imperative language. Before evaluating our program, we call the `ANTLR`: (ANother Tool for Language Recognition) [14] framework in order to generate, from a grammar file, a parser that can build and walk parse tree.
- **Range determination:** Consists in launching the execution of the program a certain number of times in order to determine dynamically the range of variables (we plan to use a static analyzer in the future).
- **Constraints generation:** It implements the forward and backward error analysis transfer function seen in Sect. 4 where the main semantics are detailed in [11]. In addition to the variables of accuracy assigned to each label ℓ which are $acc_F(\ell)$, $acc_B(\ell)$ and $acc(\ell)$ (defined in Sect. 5.2), we add new constraints relative to the *ulp* and the precision of the error in order to compute correctly the function ι discussed in Sect. 4.2.
- **Constraints resolution:** Firstly, we call the `Z3` SMT solver [6] to find a solution for our constraints and we implement a cost function (see Sect. 6) to refine the solutions obtained in term of optimality. In future work, we will explore a new resolution method based on policy iterations [8]. Concerning the complexity of the analysis performed by `POP`, in practice, the analysis is carried out by the SMT solver which solves the constraints. The number of variables and constraints is linear in the size of the program. The complexity to analyze a program of size n is then equivalent to that of solving a system of n constraints in our language of constraints (by the solver).

5.2 Simple Imperative Language of Constraints

In order to explain the constraints generation, we introduce the following simple imperative language. As it is mentioned in Fig. 4, we assign to each element of our language (expression, boolean and statement) a unique label $\ell \in lab$ with the intention of identifying without ambiguity each node of the syntactic tree. The same strategy as in [11] is adopted, the statement $\texttt{require\_accuracy(x,n)}^{\ell}$ denotes the accuracy that x must have at the control point ℓ. Therefore, we assign to each control point ℓ three integer variables corresponding to the forward, the backward and the final accuracies so that the inequality in Eq. (27) is verified. Hence, we notice that in the forward mode, the accuracy decreases contrarily to the backward mode when we strengthen the post-conditions (accuracy increases).

$$0 \leq acc_B(\ell) \leq acc(\ell) \leq acc_F(\ell) \tag{27}$$

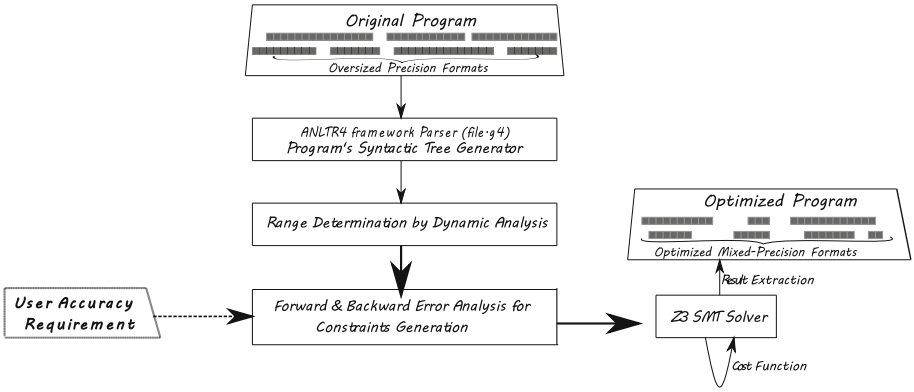


Fig. 3. POP mixed-precision analysis architecture

Expression : e ::= $c^{\sharp}_{\sharp}\mathbf{p}^{\ell}$ | id^{ℓ} | $e_1^{\ell_1} +^{\ell} e_2^{\ell_2}$ | $e_1^{\ell_1} -^{\ell} e_2^{\ell_2}$ | $e_1^{\ell_1} \times^{\ell} e_2^{\ell_2}$ | $e_1^{\ell_1} \div^{\ell} e_2^{\ell_2}$
Boolean : b ::= true | false | $e_1^{\ell_1} <^{\ell} e_2^{\ell_2}$ | $e_1^{\ell_1} >^{\ell} e_2^{\ell_2}$ | $e_1^{\ell_1} =^{\ell} e_2^{\ell_2}$
Statement : $c ::= c_1^{\ell_1}; c_2^{\ell_2}$ |$id =^{\ell} e^{\ell_1}$ | $\textbf{\textit{while}}^{\ell} b^{\ell_0}$ **do** $c_1^{\ell_1}$ | $\textbf{\textit{if}}^{\ell} b^{\ell_0}$ **then** $c_1^{\ell_1}$ **else** c | ***require_accuracy(x,n)***$^{\ell}$

Fig. 4. Simple imperative language of constraints

6 Experimental Results

In this section, we aim at evaluating the performance of POP which generates the constraints defined in Sect. 4.2 and calls the Z3 SMT solver in order to obtain a solution. The solutions returned by Z3 are not unique due to the fact that it is not an optimizer but a solver. To surpass this limitation, we add to our global system of constraints an additional constraint related to a cost function ϕ (we take the same definition in [11]). The purpose of a cost function $\phi(c)$ of a given program c is to compute the sum of the accuracies of all the variables and the intermediary values collected in each label of the arithmetic expressions as it is shown in Eq. (28).

$$\phi(c) = \sum_{x \in Id, \ell \in Lab} acc(x^{\ell}) + \sum_{\ell \in Lab} acc(\ell) \tag{28}$$

After, our tool searches the smallest integer P such that our system of constraints admits a solution. Consequently, we start the binary search with $P \in [0{,}52 \times n]$ where all the values are in double precision and where n is the number of terms in Eq. (28). While a solution is found for a given value of P, a new iteration of the binary search is run with a smaller value of P. When the solver fails for some P, a new iteration of the binary search is run with a larger P and we

continue this process until convergence. We ran our precision-tuning analysis on programs that perform sum and product operations only (for now) to show the performances of our forward and backward analysis described in Sect. 4.2. Noting that these operations are widely used in embedded systems, graphic processing, finance, etc. We take into consideration two examples which consist in a rotation matrix-vector multiplication and the computation of the determinant of 3×3 matrices and we present in Fig. 5 some measures of the efficiency of our analysis on these two examples. We assume that in the original programs of our examples all the variables are in double precision.

Rotation Matrix-Vector Multiplication

Our first example consists in a rotation matrix R which is used in the rotation of vectors and tensors while the coordinate system remains fixed. For instance, we want to rotate a vector around the z axis by angle θ. The rotation matrix and the rotated column vectors are given by:

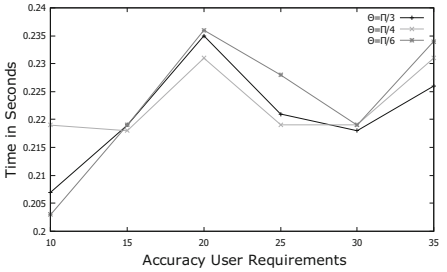
$$\begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

We aim from this experimentation to compute the performance of our `POP` tool from different angles of rotation $\frac{\pi}{3}$, $\frac{\pi}{4}$ and $\frac{\pi}{6}$, a variety of input vectors chosen with difference in magnitude $A = [1.0, 2.0, 3.0]$, $B = [10.0, 100.0, 500.0]$, $C = [100.0, 500.0, 1000.0]$, $D = [-100.0, -10.0, 1000.0]$, $E = [1.0, 2.0, 500.0]$ and $F = [1.0, 500.0, 10000.0]$ and for different user accuracy requirements 10, 15, 20, 25, 30 and 35. This example generates 858 constraints and 642 variables which are very manageable by the `Z3` solver. Initially starting with 10335 bits for the original program (only variables in double precision), Fig. 5c shows that the improvement, in the number of bits needed to realize the user requirements, compared to the initial number of bits, ranges from 38 % to 87 % which confirms the usefulness of our analysis. Also, we can observe in Fig. 5e that the majority of variables fits in single precision format for an accuracy ≤ 35 and that no double precision variables are noticed for vectors A, B, C, D and E for an accuracy 15. For this example, we found that the variation of the angles of rotation do not have impact on the number of double precision variables after analysis that's why we choose only the angle $\frac{\pi}{4}$ in Fig. 5e and by modifying the magnitude of the vectors at every turn. Besides, `POP` assigns zeros to the accuracies of the variables that are not used by the program.

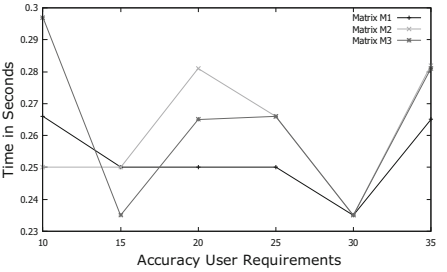
Determinant of 3 × 3 Matrices

Our second example computes the determinant $det(M)$ of a 3×3 matrices $M1$, $M2$ and $M3$ as shown:

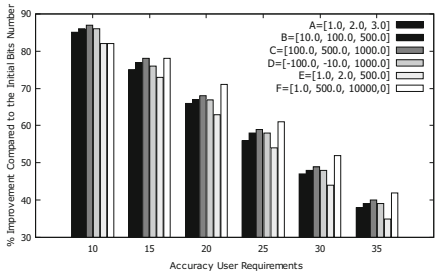
$$M = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \rightarrow det(M) = (a.e.i + d.h.c + g.b.f) - (g.e.c + a.h.f + d.b.i)$$



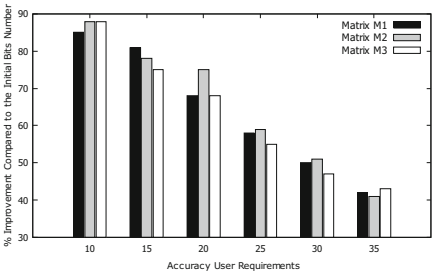
(a) Precision tuning tool execution time for the rotation matrix-vector multiplication



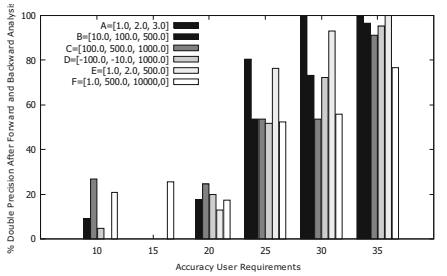
(b) Precision tuning tool execution time for the a 3×3 matrix determinant



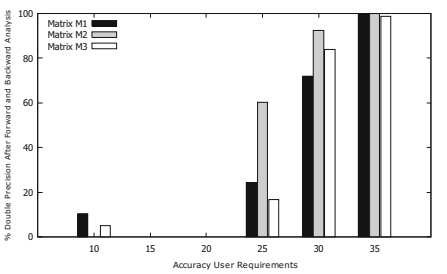
(c) Optimization of the number of bits compared to the original rotation matrix-vector multiplication program



(d) Optimization of the number of bits compared to the original 3×3 determinant program



(e) The percentage of the double precision variables after the forward and backward analysis for the first example for $\theta = \frac{\pi}{4}$



(f) The percentage of the double precision variables after the forward and backward analysis for the second example

Fig. 5. Measures of the efficiency of the analysis on the two input examples: the time execution measure, the optimization of the number of bits of the transformed programs compared to the original ones and the percentage of the double precision variables after analysis.

The matrices coefficients belong to multiple magnitude ranges: $M1 = \begin{bmatrix} [-50.1,50.1] & [-50.1,50.1] & [-50.1,50.1] \\ [-10.1,10.1] & [-10.1,10.1] & [-10.1,10.1] \\ [-5.1,5.1] & [-5.1,5.1] & [-5.1,5.1] \end{bmatrix}$, $M2 = \begin{bmatrix} [-100.1,100.1] & [-100.1,100.1] & [-100.1,100.1] \\ [-10.1,10.1] & [-10.1,10.1] & [-10.1,10.1] \\ [-2.1,2.1] & [-2.1,2.1] & [-2.1,2.1] \end{bmatrix}$ and

$M3 = \begin{bmatrix} [-10.1,10.1] & [-10.1,10.1] & [-10.1,10.1] \\ [-20.1,20.1] & [-20.1,20.1] & [-20.1,20.1] \\ [-5.1,5.1] & [-5.1,5.1] & [-5.1,5.1] \end{bmatrix}$. With 686 number of variables and 993 generated constraints, `POP` finds the minimal precision of the inputs and intermediary results for this example in less than 0.3 s as it is observed in Fig. 5b (time only for the resolution of the system of constraints and the calls of the `Z3` SMT solver done by binary search) for different requirements of accuracy. Hence, as viewed in our first example, the final number of bits of the transformed program compared to 9964 initial bits is considerable as shown in Fig. 5d. Finally, we notice that our analysis succeeded in turning off almost the double precision variables to a fairly rounded single precision ones for an accuracy ≤ 20.

7 Conclusions and Future Work

In this article, we have introduced `POP`, an automated tuning tool for floating-point precision that computes the minimal number of bits needed for the variables and intermediary results in order to accomplish the user requirement of accuracy. Also, we have explained in details our forward and backward static analysis, done by abstract interpretation. Moreover, we have shown that we can express our analysis as a set of constraints made of propositional logic formulas and relations between affine expressions over integers which can be easily checked by an SMT solver. Obviously, our approach can be extended to other language structures in particular arrays and functions. Besides, we have considered that a range determination is performed by dynamic analysis on the variables of our programs and that no overflow arises during our analysis but from this time on we would like to adopt a static analyzer in order to infer safe ranges on our variables.

In future work, we would like to explore the policy iteration method [8] as a replacement for the non-optimizing solver (`Z3`) coupled to a binary search used in this article. In fact, we aim to apply the policy iteration method to improve the accuracy. The principle consists in transforming all the generated constraints to the form of min-max of discrete affine maps. Further, it will be interesting to feed the policy iteration with the `Z3` solution as an initial policy and consequently comparing the solutions of these two methods in term of execution time and optimality. Nevertheless, our goal is to validate experimentally our tool on codes from various fields including safety-critical systems such as control systems for vehicles, medical equipment and industrial plants. Also, we are currently working on exploring the precision tuning in a new unexplored domain, Internet of Things. In fact, the type of problems of energy consumption and memory saving are widespread in this area that is why we are working on tuning the precision of the basic buildings of common IoT items such as accelerometers and gyroscopes. Conclusively, comparing our tool to other existing tools in the matter of analysis time and speed and the quality of the solution is a tremendous challenge to examine.

References

1. Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia. Technical report GAO/IMTEC-92-26, General Accounting Office (1992)
2. Chiang, W.F., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., Rakamarić, Z.: Rigorous floating-point mixed-precision tuning. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL). ACM (2017)
3. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977, pp. 238–252 (1977)
4. Damouche, N., Martel, M.: Salsa: an automatic tool to improve the numerical accuracy of programs. In: Shankar, N., Dutertre, B. (eds.) Automated Formal Methods. Kalpa Publications in Computing, EasyChair (2018)
5. Darulova, E., Kuncak, V.: Sound compilation of reals. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014. ACM (2014)
6. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
7. De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Commun. ACM **54**(9), 69–77 (2011)
8. Gaubert, S., Goubault, E., Taly, A., Zennou, S.: Static analysis by policy iteration on relational domains. In: De Nicola, R. (ed.) Programming Languages and Systems. Springer, Heidelberg (2007)
9. Halfhill, T.R.: The truth behind the Pentium bug: how often do the five empty cells in the Pentium's FPU lookup table spell miscalculation? (1995)
10. Lam, M.O., Hollingsworth, J.K., de Supinski, B.R., Legendre, M.P.: Automatically adapting programs for mixed-precision floating-point computation. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS 2013. ACM (2013)
11. Martel, M.: Floating-point format inference in mixed-precision. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 230–246. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_16
12. Muller, J.M.: On the definition of ULP(x). Research Report RR-5504, LIP RR-2005-09, INRIA, LIP, February 2005. https://hal.inria.fr/inria-00070503
13. Muller, J.M., et al.: Handbook of Floating-Point Arithmetic, 1st edn. Birkhäuser, Boston (2009)
14. Parr, T.: The Definitive ANTLR 4 Reference, 2nd edn. Pragmatic Bookshelf, Raleigh (2013)
15. Rubio-González, C., et al.: Precimonious: tuning assistant for floating-point precision. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2013, Denver, CO, USA, 17–21 November 2013 (2013)
16. Rubio-González, C., et al.: Floating-point precision tuning using blame analysis. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE) (2016)
17. Solovyev, A., Jacobsen, C., Rakamarić, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 532–550. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19249-9_33