Fast and Efficient Bit-Level Precision Tuning

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

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

Abstract. In this article, we introduce a new technique for precision tuning. This problem consists of finding the least data types for numerical values such that the result of the computation satisfies some accuracy requirement. State of the art techniques for precision tuning use a trial-and-error approach. They change the data types of some variables of the program and evaluate the accuracy of the result. Depending on what is obtained, they change more or less data types and repeat the process. Our technique is radically different. Based on semantic equations, we generate an Integer Linear Problem (ILP) from the program source code. Basically, this is done by reasoning on the most significant bit and the number of significant bits of the values which are integer quantities. The integer solution to this problem, computed in polynomial time by a classical linear programming solver, gives the optimal data types at the bit level. A finer set of semantic equations is also proposed which does not reduce directly to an ILP problem. So we use policy iteration to find the solution. Both techniques have been implemented and we show that our results encompass the results of state-of-the-art tools.

Keywords: Static analysis, computer arithmetic, integer linear problems, numerical accuracy, policy iteration.

1 Introduction

Let us consider a program P computing some numerical result R, typically but not necessarily in the IEEE754 floating-point arithmetic [1]. Precision tuning then consists of finding the smallest data types for all the variables and expressions of P such that the result R has some desired accuracy. These last years, much attention has been paid to this problem [8,11,14,16,17,24]. Indeed, precision tuning makes it possible to save memory and, by way of consequence, it has a positive impact on the footprint of programs concerning energy consumption, bandwidth usage, computation time, etc.

A common point to all the techniques cited previously is that they follow a trial-and-error approach. Roughly speaking, one chooses a subset S of the variables of P, assigns to them smaller data types (e.g. `binary32` instead of `binary64` [1]) and evaluates the accuracy of the tuned program P'. If the accuracy of the result returned by P' is satisfying then new variables are included in S or even smaller data types are assigned to certain variables already in S

(e.g. `binary16`). Otherwise, if the accuracy of the result of P' is not satisfying, then some variables are removed from S. This process is applied repeatedly, until a stable state is found. Existing techniques differ in their way to evaluate the accuracy of programs, done by dynamic analysis [14,16,17,24] or by static analysis [8,11] of P and P'. They may also differ in the algorithm used to define S, delta debugging being the most widespread method [24]. A notable exception is FPTuner [8] which relies on a local optimization procedure by solving quadratic problems for a given set of candidate datatypes. A more exhaustive state-of-the-art about precision tuning techniques is given in [7].

Anyway all these techniques suffer from the same combinatorial limitation: If P has n variables and if the method tries k different data types then the search space contains k^n configurations. They scale neither in the number n of variables (even if heuristics such as delta debugging [24] or branch and bound [8] reduce the search space at the price of optimality) or in the number k of data types which can be tried. In particular, bit level precision tuning, which consists of finding the minimal number of bits needed for each variable to reach the desired accuracy, independently of a limited number k of data types, is not an option.

So the method introduced in this article for precision tuning of programs is radically different. Here, no trial-and-error method is employed. Instead, the accuracy of the arithmetic expressions assigned to variables is determined by semantic equations, in function of the accuracy of the operands. By reasoning on the number of significant bits of the variables of P and knowing the weight of their most significant bit thanks to a range analysis performed before the tuning phase (see Section 3), we are able to reduce the problem to an Integer Linear Problem (ILP) which can be optimally solved in one shot by a classical linear programming solver (no iteration). Concerning the number n of variables, the method scales up to the solver limitations and the solutions are naturally found at the bit level, making the parameter k irrelevant. An important point is that the optimal solution to the continuous linear programming relaxation of our ILP is a vector of integers, as demonstrated in Section 4.2. By consequence, we may use a linear solver among real numbers whose complexity is polynomial [25] (contrarily to the linear solvers among integers whose complexity is NP-Hard [22]). This makes our precision tuning method solvable in polynomial-time, contrarily to the existing exponential methods. Next, we go one step further by introducing a second set of semantic equations. These new equations make it possible to tune even more the precision by being less pessimistic on the propagation of carries in arithmetic operations. However the problem does not reduce any longer to an ILP problem (min and max operators are needed). Then we use policy iteration (PI) [9] to find efficiently the solution.

Both methods have been implemented inside a tool for precision tuning named POP. Formerly, POP was expressing the precision tuning problem as a set of first order logical propositions among relations between linear integer expressions [2,3,4,6]. An SMT solver (Z3 in practice [21]) was used repeatedly to find the existence of a solution with a certain weight expressing the number of significant bits (`nsb`) of variables. In the present article, we compare experimen-

tally our new methods to the SMT based method previously used by POP and to the Precimonious tool [14,24]. These experiments on programs coming from mathematical libraries or other applicative domains such as IoT [2,3] show that the technique introduced in this article for precision tuning clearly encompasses the state of the art techniques.

The rest of this article is organized as follows. In the next section, we provide a motivating example. We then present in Section 3 some essential background on the functions needed for the constraint generation and also we detail the set of constraints for both ILP and PI methods. Section 4 presents the proofs of correctness. We end up in Section 5 by showing that our new technique exhibits very good results in practice before concluding in Section 6.

2 Running Example

A motivating example to better explain our method is given by the code snippet of Figure 1. In this example, we aim at modeling the movement of a simple pendulum without damping. Let $l = 0.5\ m$ be the length of this pendulum, $m = 1\ kg$ its mass and $g = 9.81\ m \cdot s^{-2}$ Newton's gravitational constant. We denote by θ the tilt angle in radians as shown in Figure 1 (initially $\theta = \frac{\pi}{4}$). The Equation describing the movement of the pendulum is given in Equation (1).

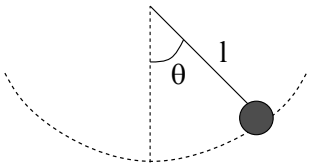
$$m \cdot l \cdot \frac{d^2\theta}{dt^2} = -m \cdot g \cdot \sin\theta \tag{1}$$

Equation (1) being a second order differential equation. We need to transform it into a system of two first order differential equations for resolution. We obtain $y_1 = \theta$ and $y_2 = \frac{d\theta}{dt}$. By applying Euler's method to these last equations, we obtain Equation (2) implemented in Figure 1.

$$\frac{dy_1}{dt} = y_2 \quad \text{and} \quad \frac{dy_2}{dt} = -\frac{g}{l} \cdot \sin y_1 \tag{2}$$

The key point of our technique is to generate a set of constraints for each statement of our imperative language introduced further in Section 3. For our example, we suppose that all variables, before POP analysis, are in double precision (source program in the top left corner of Figure 1) and that a range determination is performed by dynamic analysis on the program variables (we plan to use a static analyzer in the future). POP assigns to each node of the program's syntactic tree a unique control point in order to determine easily the number of significant bits of the result as mentioned in the bottom corner of Figure 1. Some notations can be highlighted about the structure of POP source code. For instance, the annotation $\mathtt{g}^{\ell_1} = 9.81^{\ell_0}$ denotes that this instance of `g` has the unique control point ℓ_1. As well, we have the statement `require_nsb(y2,20)` which informs the tool that the user wants to get on variable `y2` only 20 significant bits (we consider that a result has n significants if the relative error between the exact and approximated results is less than 2^{-n}). Finally, the minimal precision needed for the inputs and intermediary results satisfying the user

```
1  g = 9.81; l = 0.5;
2  y1 = 0.785398; y2 = 0.785398;
3  h = 0.1; t = 0.0;
4  while (t<10.0) {
5   y1new = y1 + y2 * h ;
6   aux1 = sin(y1) ;
7   aux2 = aux1 * h * g / l;
8   y2new = y2 - aux2;
9   t = t + h;
10  y1 = y1new; y2 = y2new;
11 };
12 require_nsb(y2,20);
```




```
1  g^ℓ1 = 9.81^ℓ0; l^ℓ3 = 0.5^ℓ2;
2  y1^ℓ5 = 0.785398^ℓ4; y2^ℓ7 = 0.785398^ℓ6;
3  h^ℓ9 = 0.1^ℓ8; t^ℓ11 = 0.0^ℓ10;
4  while (t^ℓ13 <^ℓ15 10.0^ℓ14)^ℓ59 {
5   y1new^ℓ24 = y1^ℓ17 +^ℓ23 y2^ℓ19 *^ℓ22 h^ℓ21;
6   aux1^ℓ28 = sin(y1^ℓ26)^ℓ27;
7   aux2^ℓ40 = aux1^ℓ30 *^ℓ39 h^ℓ32
8     *^ℓ38 g^ℓ34 /^ℓ37 l^ℓ36;
9   y2new^ℓ46 = y2^ℓ42 -^ℓ45 aux2^ℓ44;
10  t^ℓ52 = t^ℓ48 +^ℓ51 h^ℓ50;
11  y1^ℓ55 = y1new^ℓ54; y2^ℓ58 = y2new^ℓ57;
12 };
13 require_nsb(y2,20)^ℓ61;
```

```
1  g|20| = 9.81|20|; l|20| = 1.5|20|;
2  y1|29| = 0.785398|29|; y2|21| =
       0.0|21|;
3  h|21| = 0.1|21|; t|21| = 0.0|21|;
4  while (t<1.0) {
5    y1new|20| = y1|21| +|20| y2|21|
6      *|22| h|21|;
7    aux1|20| = sin(y1|29|)|20|;
8    aux2|20| = aux1|19| *|20| h|18|
9      *|19| g|17| /|18|l|17|;
10   y2new|20| = y2|21| -|20| aux2|18|;
11   t|20| = t|21| +|20| h|17|;
12   y1|20| = y1new|20|; y2|20|=y2new
         |20|;
13 };
14 require_nsb(y2,20);
```

Fig. 1: Top left: source program. Top right: pendulum movement for $\theta = \frac{\pi}{4}$. Bottom left: program annotated with labels and with inferred accuracies (right).

assertion is observed on the bottom right corner of Figure 1. In this code, if we consider for instance lines 5 and 6, then `y1new|20|` means that the variable needs 20 significant bits at this point. Similarly, `y1` and `y2` need 21 bits each and the addition requires 20 bits.

In the next section, we detail the ILP and PI formulations for precision tuning implemented in POP. Also, we show the nature of constraints generated for the pendulum example and consequently the new data types already presented in Figure 1. Note that our tool achieves precision tuning only. The inputs are the program and the ranges over the variables of the program. We insist on the fact that POP is not able to produce those ranges or to verify the correctness of the input ranges. Those ranges are understood as intervals. This range inference is completely external to our tool and has to be performed by an invariant generator or an analyzer. To simplify the implementation, we use a dynamic analysis which produces an under-approximation under the form of intervals. Static analyzers with sophisticated abstract domains could be used such as [10]. In particular the efficiency of our techniques for loops depends on the precision of the range analysis for loops. The sensibility of our precision tuning to the estimate of the ranges depends on the following point. We use in the tuning phase the `ufp` of the values. So we are sensible to the order of magnitude of the ranges but not to the exact values. For example, we will obtain the same tuning

with the ranges $[3.4, 6.1]$ and $[2.5, 7.8]$. But, obviously we get a worst tuning if we use $[0.0, 1000.0]$.

3 Generation of Constraints for Bit-Level Tuning

In this section, we start with providing essential definitions for understanding the rest of the article. Also, we define a simple imperative language from which we generate semantic equations in order to determine the least precision needed for the program numerical values. Then, we will focus on the two sets of constraints obtained when using the simple ILP and the more complex PI formulation which optimizes the carry bits that propagate throughout computations.

3.1 Preliminary Notations and Definitions

Our technique is independent of a particular computer arithmetic (e.g. IEEE754 [1] and POSIT [15]). In fact, we manipulate numbers for which we know the unit in the first place (ufp) and the number of significant digits (nsb). We also assume that the constants occurring in the source codes are exact and we bound the errors introduced by the finite precision computations. Then, in the following, $\mathsf{ufp_e}(x)$ and $\mathsf{nsb_e}(x)$ denote the ufp and nsb of the error on x (note that $\mathsf{nsb_e}(x)$ may be infinite in some cases). These functions are defined hereafter and a more intuitive presentation is given in Figure 2.

Unit in the First Place The unit in the first place of a real number x (possibly encoded up to some rounding mode by a floating-point or a fixpoint number) is given in Equation (3). This function is independent of the representation of x.

$$\mathsf{ufp}(x) = \begin{cases} \min\{i \in \mathbb{Z} : 2^{i+1} > |x|\} = \lfloor \log_2(|x|) \rfloor & \text{if } x \neq 0, \\ 0 & \text{if } x = 0. \end{cases} \tag{3}$$

Number of Significant Bits Intuitively, $\mathsf{nsb}(x)$ is the number of significant bits of x. Let $\hat{x}$ the approximation of x in finite precision and let $\varepsilon(x) = |x - \hat{x}|$ be the absolute error. Following Parker [23], if $\mathsf{nsb}(x) = k$, for $x \neq 0$, then

$$.\varepsilon(x) \leq 2^{\mathsf{ufp}(x)-k+1} \tag{4}$$

In addition, if $x = 0$ then $\mathsf{nsb}(x) = 0$. For example, if the exact binary value 1.0101 is approximated by either $x = 1.010$ or $x = 1.011$ then $\mathsf{nsb}(x) = 3$.

Unit in the Last Place The unit in the last place ulp of x is defined by

$$\mathsf{ulp}(x) = \mathsf{ufp}(x) - \mathsf{nsb}(x) + 1 \ . \tag{5}$$

Computation Errors The unit in the first place of the error on x is $\mathsf{ufp_e}(x) = \mathsf{ufp}(x) - \mathsf{nsb}(x)$. The number of significant bits of the computation error on x is denoted $\mathsf{nsb_e}(x)$. It is used to optimize the function ξ defined in Equation (6). As mentioned earlier, we assume that there is no error on any constant c arising in programs, i.e. $\mathsf{nsb_e}(c) = 0$. Nevertheless, the $\mathsf{nsb_e}$ of the results of elementary operations may be greater than 0. For instance, if we add two constants c_1, c_2 in x such that $\mathsf{ufp_e}(c_1) \geq \mathsf{ufp_e}(c_2)$ then $\mathsf{nsb_e}(x) = \mathsf{ufp_e}(c_1) - (\mathsf{ufp_e}(c_2) - \mathsf{nsb_e}(c_2))$ which corresponds to the nsb of the resulting error (see Figure 2). The unit in the last place of the computation error on x is denoted $\mathsf{ulp_e}(x)$ and we have $\mathsf{ulp_e}(x) = \mathsf{ufp_e}(x) - \mathsf{nsb_e}(x) + 1$.

Carry Bit During an operation between two numbers c_1 and c_2, a carry bit can be propagated through the operation. We model the carry bit by a function denoted ξ computed as shown in Figure 2:

If the ulp of one of the two operands c_1 or c_2 is greater than the ufp of the other one (or conversely) then c_1 and c_2 are not aligned and $\xi = 0$ (otherwise $\xi = 1$). Recall that, for a number x, we have $\mathsf{ufp_e}(x) = \mathsf{ufp}(x) - \mathsf{nsb}(x)$. In Section 3.3, we will use the ξ function to optimize the error terms. The over-approximation of ξ by supposing that it is always equal to 1 leads to the analysis of Section 3.2. However, when many operations are done in a program which has to compute with some tens of nsb, adding one bit is far from being negligible. Consequently, a refined analysis is presented in Section 3.3 where ξ is formulated by min and max operators. Let c_1 and c_2 be the operands of some operation whose result is x. The optimized ξ function of Section 3.3 is given by

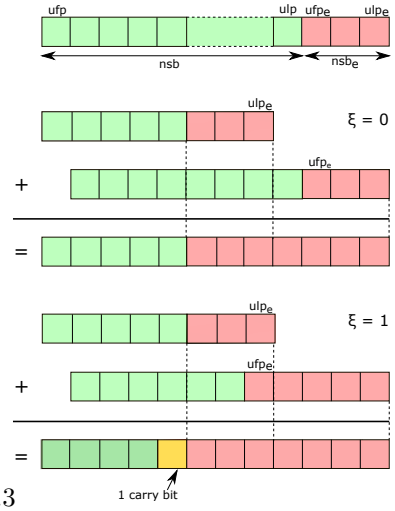


Fig. 2: Schematic representation of ufp, nsb and ulp for values and errors. Representation of the carry bit function ξ.

$$\xi(x)(c_1, c_2) = \begin{cases} 0 & \mathsf{ulp_e}(c_1) \geq \mathsf{ufp_e}(c_2), \\ 0 & \mathsf{ulp_e}(c_2) \geq \mathsf{ufp_e}(c_1), \\ 1 & \text{otherwise.} \end{cases}$$

In Figure 5, an equivalent yet less intuitive definition of ξ is used which corresponds to Equation (6) in Lemma 1.

Lemma 1 *Let c_1 and c_2 be the operands of some operation whose result is x.*

$$\xi(x)(c_1, c_2) = \begin{cases} 0 & \mathsf{ufp_e}(c_1) - \mathsf{nsb_e}(c_1) \geq \mathsf{ufp}(c_2) - \mathsf{nsb}(c_2) \quad \textit{or} \quad \textit{conversely}, \\ 1 & \textit{otherwise.} \end{cases} \tag{6}$$

$$\ell \in Lab \quad x \in Id \quad \odot \in \{+, -, \times, \div\} \quad math \in \{\sin, \cos, \tan, \arcsin, \log, \ldots\}$$
$$\mathbf{Expr} \ni \mathbf{e} : \mathrm{e} ::= \mathrm{c}\#p^{\ell} \mid x^{\ell} \mid e_1^{\ell_1} \odot^{\ell} e_2^{\ell_2} \mid math(e^{\ell_1})^{\ell} \mid sqrt(e^{\ell_1})^{\ell}$$
$$\mathbf{Cmd} \ni \mathbf{c} : c ::= c_1^{\ell_1}; c_2^{\ell_2} \mid x =^{\ell} e^{\ell_1} \mid \textbf{\textit{while}}^{\ell} b^{\ell_0} \textbf{\textit{ do }} c_1^{\ell_1} \mid \textbf{\textit{if}}^{\ell} b^{\ell_0} \textbf{\textit{ then }} c_1^{\ell_1} \textbf{\textit{ else }} c^{\ell_2} \mid \textbf{\textit{require\_nsb}}(x, n)^{\ell}$$

Fig. 3: Language of input programs.

3.2 Integer Linear Problem Formulation

First, we define in Figure 3 the simple imperative language in which our input programs are written.

We denote by Id the set of identifiers and by Lab the set of control points of the program as a means to assign to each element $e \in Expr$ and $c \in Cmd$ a unique control point $\ell \in Lab$. First, in c#p, p indicates the initial number of significant bits of the constant c in the source code. Next, the statement require_nsb(x,n)$^{\ell}$ indicates the minimal number of significant bits n that a variable x must have at a control point ℓ. The rest of the grammar is standard.

As we have mentioned, we are able to reduce the problem of determining the lowest precision on variables and intermediary values in programs to an Integer Linear Problem (ILP) by reasoning on their unit in the first place (ufp) and the number of significant bits (nsb). In addition, we assign to each control point ℓ an integer variable $\mathsf{nsb}(\ell)$ corresponding to the nsb of the arithmetic expressions. Note that $\mathsf{nsb}(\ell)$ is determined by solving the ILP generated by the rules of Figure 4. Let us also mention that, in order to avoid cumbersome notations, the constraints introduced hereafter assume that the programs handle scalar values instead of the intervals given by the range analysis. A generalisation to intervals is introduced in [18] for a comparable (yet not linear) set of constraints.

Let us now focus on the rules of Figure 4 where $\varrho : \mathrm{Id} \to \mathrm{Id} \times \mathrm{Lab}$ is an environment which relates each identifier x to its last assignment x^{ℓ}: Assuming that $x :=^{\ell} e^{\ell_1}$ is the last assignment of x, the environment ϱ maps x to x^{ℓ}. Then, $\mathcal{E}[\![e]\!]\ \varrho$ generates the set of constraints for an expression $e \in Expr$ in the environment ϱ. In the sequel, we formally define these constraints for each element of our language. No constraint is generated for a constant c#p as mentioned in Rule (Const) of Figure 4. For Rule (Id) of a variable x^{ℓ}, we require that the nsb at control point ℓ is less than its nsb in the last assignment of x given in $\varrho(x)$. For a binary operator $\odot \in \{+, -, \times, \div\}$, we first generate the set of constraints $\mathcal{E}[\![e_1^{\ell_1}]\!]\varrho$ and $\mathcal{E}[\![e_2^{\ell_2}]\!]\varrho$ for the operands at control points ℓ_1 and ℓ_2. Considering Rule (Add), the result of the addition of two numbers is stored in control point ℓ. Recall that a range determination is performed before the accuracy analysis, $\mathsf{ufp}(\ell)$, $\mathsf{ufp}(\ell_1)$ and $\mathsf{ufp}(\ell_2)$ are known at constraint generation time.

In the present ILP of Figure 4, we over-approximate the function ξ by $\xi(\ell)(\ell_1, \ell_2) = 1$ for all ℓ, ℓ_1 and ℓ_2. To wrap up, for the addition (Rule (Add)), we have the $\mathsf{nsb}(\ell) = \mathsf{ufp}(\ell) - \mathsf{ufp}_{\mathsf{e}}(\ell)$. More precisely, let us consider the addition $c_1^{\ell_1} +^{\ell} c_2^{\ell_2}$ and let us assume that $\mathsf{prec}(\ell)$ denotes the precision of this operation.

$$\mathcal{E}[c\#\mathtt{p}^{\ell}]\varrho = \emptyset \quad \text{(Const)} \qquad \mathcal{E}[x^{\ell}]\varrho = \{\mathsf{nsb}(\varrho(x)) \geq \mathsf{nsb}(\ell)\} \quad \text{(Id)}$$

$$\begin{array}{c}\mathcal{E}[e_1^{\ell_1} +^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}[e_1^{\ell_1}]\varrho \ \cup \ \mathcal{E}[e_2^{\ell_2}]\varrho \\ \cup \\ \{\mathsf{nsb}(\ell_1) \geq \mathsf{nsb}(\ell) + \mathsf{ufp}(\ell_1) - \mathsf{ufp}(\ell) + \xi(\ell)(\ell_1,\ell_2), \\ \mathsf{nsb}(\ell_2) \geq \mathsf{nsb}(\ell) + \mathsf{ufp}(\ell_2) - \mathsf{ufp}(\ell) + \xi(\ell)(\ell_1,\ell_2)\}\end{array} \quad \text{(Add)}$$

$$\begin{array}{c}\mathcal{E}[e_1^{\ell_1} -^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}[e_1^{\ell_1}]\varrho \ \cup \ \mathcal{E}[e_2^{\ell_2}]\varrho \\ \cup \\ \{\mathsf{nsb}(\ell_1) \geq \mathsf{nsb}(\ell) + \mathsf{ufp}(\ell_1) - \mathsf{ufp}(\ell) + \xi(\ell)(\ell_1,\ell_2), \\ \mathsf{nsb}(\ell_2) \geq \mathsf{nsb}(\ell) + \mathsf{ufp}(\ell_2) - \mathsf{ufp}(\ell) + \xi(\ell)(\ell_1,\ell_2)\}\end{array} \quad \text{(Sub)}$$

$$\begin{array}{c}\mathcal{E}[e_1^{\ell_1} \times^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}[e_1^{\ell_1}]\varrho \ \cup \ \mathcal{E}[e_2^{\ell_2}]\varrho \\ \cup \\ \{\mathsf{nsb}(\ell_1) \geq \mathsf{nsb}(\ell) + \xi(\ell)(\ell_1,\ell_2) - 1, \quad \mathsf{nsb}(\ell_2) \geq \mathsf{nsb}(\ell) + \xi(\ell)(\ell_1,\ell_2) - 1\}\end{array} \quad \text{(Mult)}$$

$$\begin{array}{c}\mathcal{E}[e_1^{\ell_1} \div^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}[e_1^{\ell_1}]\varrho \ \cup \ \mathcal{E}[e_2^{\ell_2}]\varrho \\ \cup \\ \{\mathsf{nsb}(\ell_1) \geq \mathsf{nsb}(\ell) + \xi(\ell)(\ell_1,\ell_2) - 1, \quad \mathsf{nsb}(\ell_2) \geq \mathsf{nsb}(\ell) + \xi(\ell)(\ell_1,\ell_2) - 1\}\end{array} \quad \text{(Div)}$$

$$\mathcal{E}\left[\sqrt{e^{\ell_1}}^{\ell}\right]\varrho = \mathcal{E}[e_1^{\ell_1}]\varrho \ \cup \ \{\mathsf{nsb}(\ell_1) \geq \mathsf{nsb}(\ell)\} \quad \text{(Sqrt)}$$

$$\mathcal{E}\left[\phi\left(e^{\ell_1}\right)^{\ell}\right]\varrho = \mathcal{E}[e_1^{\ell_1}]\varrho \cup \{\mathsf{nsb}(\ell_1) \geq \mathsf{nsb}(\ell) + \varphi\} \text{ with } \phi \in \{\sin, \cos, \tan, \log, \ldots\} \quad \text{(Math)}$$

$$\mathcal{C}\left[x :=^{\ell} e^{\ell_1}\right]\varrho = \left(C, \varrho\left[x \mapsto \ell\right]\right) \text{ where } C = \mathcal{E}[e_1^{\ell_1}]\varrho \cup \{\mathsf{nsb}(\ell_1) \geq \mathsf{nsb}(\ell)\} \quad \text{(Assign)}$$

$$\begin{array}{c}\mathcal{C}\left[c_1^{\ell_1}\,;c_2^{\ell_2}\right]\varrho = \left(C_1 \cup C_2, \varrho_2\right) \\ \text{where } \left(C_1,\varrho_1\right) = \mathcal{C}\left[c_1^{\ell_1}\right]\varrho \text{ and } \left(C_2,\varrho_2\right) = \mathcal{C}\left[c_2^{\ell_2}\right]\varrho_1\end{array} \quad \text{(Seq)}$$

$$\begin{array}{c}\mathcal{C}[\mathtt{if}^{\ell}\ e^{\ell_0}\ \mathtt{then}\ c^{\ell_1}\ \mathtt{else}\ c^{\ell_2}]\ \varrho = (C_1 \cup C_2 \cup C_3, \varrho') \\ \text{where } \left|\begin{array}{l}\forall x \in \mathrm{Id},\ \varrho'(x) = \ell,\ (C_1,\varrho_1) = \mathcal{C}[c_1^{\ell_1}]\ \varrho,\ (C_2,\varrho_2) = \mathcal{C}[c_2^{\ell_2}]\ \varrho, \\ C_3 = \bigcup\limits_{x\in\mathrm{Id}} \{\mathsf{nsb}(\varrho_1(x)) \geq \mathsf{nsb}(\ell),\ \mathsf{nsb}(\varrho_2(x)) \geq \mathsf{nsb}(\ell)\}\end{array}\right.\end{array} \quad \text{(Cond)}$$

$$\begin{array}{c}\mathcal{C}[\mathtt{while}^{\ell}\ e^{\ell_0}\ \mathtt{do}\ c^{\ell_1}]\ \varrho = (C_1 \cup C_2, \varrho') \\ \text{where } \left|\begin{array}{l}\forall x \in \mathrm{Id},\ \varrho'(x) = \ell,\ (C_1,\varrho_1) = \mathcal{C}[c_1^{\ell_1}]\ \varrho' \\ C_2 = \bigcup\limits_{x\in\mathrm{Id}} \{\mathsf{nsb}(\varrho(x)) \geq \mathsf{nsb}(\ell),\ \mathsf{nsb}(\varrho_1(x)) \geq \mathsf{nsb}(\ell)\}\end{array}\right.\end{array} \quad \text{(While)}$$

$$\mathcal{C}[\mathtt{require\_nsb}(x,\mathtt{p})^{\ell}]\varrho = \{\mathsf{nsb}(\varrho(x)) \geq \mathtt{p}\} \quad \text{(Req)}$$

$$\xi(\ell)(\ell_1,\ell_2) = 1$$

Fig. 4: ILP constraints with pessimistic carry bit propagation $\xi = 1$.

The error $\varepsilon(\ell)$ is bound by $\varepsilon(c_1^{\ell_1} +^{\ell} c_2^{\ell_2}) \leq \varepsilon(c_1^{\ell_1}) + \varepsilon(c_2^{\ell_2}) + 2^{\mathsf{ufp}(c_1+c_2)-\mathsf{prec}(\ell)}$and

$$\mathsf{ufp_e}(\ell) = \max\left(\mathsf{ufp}(\ell_1) - \mathsf{nsb}(\ell_1), \mathsf{ufp}(\ell_2) - \mathsf{nsb}(\ell_2), \mathsf{ufp}(l) - \mathsf{prec}(\ell)\right) + \xi(\ell)(\ell_1,\ell_2) \tag{7}$$

Since $\mathsf{nsb}(\ell) \leq \mathsf{prec}(\ell)$, we may get rid of the last term in Equation (7) and the two constraints generated for Rule (ADD) are derived from Equation (8).

$$\mathsf{nsb}(\ell) \leq \mathsf{ufp}(\ell) - \max\big(\mathsf{ufp}(\ell_1) - \mathsf{nsb}(\ell_1), \mathsf{ufp}(\ell_2) - \mathsf{nsb}(\ell_2)\big) - \xi(\ell)(\ell_1, \ell_2) \tag{8}$$

Rule (Sub) for the subtraction is obtained similarly to the addition case. For Rule (Mult) of multiplication (and in the same manner Rule(Div)), the reasoning mimics the one of the addition. Let c_1 and c_2 be two numbers and c the result of their product, $c = c_1^{\ell_1} \times^{\ell} c_2^{\ell_2}$. We denote by $\varepsilon(c_1)$, $\varepsilon(c_2)$ and $\varepsilon(c)$ the errors on c_1, c_2 and c, respectively. The error $\varepsilon(c)$ of this multiplication is $\varepsilon(c) = c_1 \cdot \varepsilon(c_2) + c_2 \cdot \varepsilon(c_1) + \varepsilon(c_1) \cdot \varepsilon(c_2)$. These numbers are bounded by

$$\begin{array}{c} 2^{\mathsf{ufp}(c_1)} \leq c_1 \leq 2^{\mathsf{ufp}(c_1)+1} \quad \text{and} \quad 2^{\mathsf{ufp}(c_1)-\mathsf{nsb}(c_1)} \leq \varepsilon(c_1) \leq 2^{\mathsf{ufp}(c_1)-\mathsf{nsb}(c_1)+1} \\ 2^{\mathsf{ufp}(c_2)} \leq c_2 \leq 2^{\mathsf{ufp}(c_2)+1} \quad \text{and} \quad 2^{\mathsf{ufp}(c_2)-\mathsf{nsb}(c_2)} \leq \varepsilon(c_2) \leq 2^{\mathsf{ufp}(c_2)-\mathsf{nsb}(c_2)+1} \\ 2^{\mathsf{ufp}(c_1)+\mathsf{ufp}(c_2)-\mathsf{nsb}(c_2)} + 2^{\mathsf{ufp}(c_2)+\mathsf{ufp}(c_1)-\mathsf{nsb}(c_1)} \leq \varepsilon(c) \leq 2^{\mathsf{ufp}(c)-\mathsf{nsb}(c)+1} \\ +2^{\mathsf{ufp}(c_1)+\mathsf{ufp}(c_2)-\mathsf{nsb}(c_1)-\mathsf{nsb}(c_2)} \end{array} \tag{9}$$

We get rid of the last term $2^{\mathsf{ufp}(c_1)+\mathsf{ufp}(c_2)-\mathsf{nsb}(c_1)-\mathsf{nsb}(c_2)}$ of the error $\varepsilon(c)$ which is strictly less than the former two ones. By assuming that $\mathsf{ufp}(c_1 + c_2) = \mathsf{ufp}(c)$ and by reasoning on the exponents, we obtain the equations of Rule (Mult).

$$\mathsf{nsb}(\ell_1) \geq \mathsf{nsb}(\ell) + \xi(\ell)(\ell_1, \ell_2) - 1 \quad \text{and} \quad \mathsf{nsb}(\ell_2) \geq \mathsf{nsb}(\ell) + \xi(\ell)(\ell_1, \ell_2) - 1 \ .$$

The accuracy of math functions depends on each implementation (for example this is not in the IEEE754 Standard). It is then difficult to propose something independent of the user's library. Then, for the elementary functions such as logarithm, exponential and the hyperbolic and trigonometric functions gathered in Rule (Math), each implementation has its own nsb which we have to know to model the propagation of errors in our analyses. To cope with this limitation, we consider that each elementary function introduces a loss of precision of φ bits, where $\varphi \in \mathbb{N}$ is a parameter of the analysis and consequently of our tool, POP. In future work, we plan to reverse the question and to let the tool find the minimal accuracy needed for functions by including their accuracy in the constraint systems.

The rules of commands are rather classical, we use control points to distinguish many assignments of the same variable and also to implement joins in conditions and loops. Given a command c and an environment ϱ, $\mathcal{C}[c]\ \varrho$ returns a pair (C, ϱ') made of a set C of constraints and of a new environment ϱ'. The function $\mathcal{C}$ is defined by induction on the structure of commands in figures 4 and 5. For conditionals, we generate the constraints for the `then` and `else` branches plus additional constraints to join the results of both branches. Currently, we do not take care of the guards. As a result, we analyze both the `then` and `else` branches of the `if` statement with the whole environment. This is correct but it is a source of imprecision. Concerning loops, we relate the number of significants bits at the end of the `body` to the nsb of the same variables and the beginning of the loop as shown in Rule (While).

Back to Line 5 of the pendulum program of Section 2, we generate seven constraints as shown in Equation (10).

$$C_1 = \left\{ \begin{array}{l} \mathsf{nsb}(\ell_{17}) \geq \mathsf{nsb}(\ell_{23}) + (-1) + \xi(\ell_{23})(\ell_{17}, \ell_{22}) - (-1), \\ \mathsf{nsb}(\ell_{22} \geq \mathsf{nsb}(\ell_{23} + 0 + \xi(\ell_{23})(\ell_{17}, \ell_{22}) - (1), \\ \mathsf{nsb}(\ell_{19}) \geq \mathsf{nsb}(\ell_{22}) + \xi(\ell_{22})(\ell_{19}, \ell_{21}) - 1, \\ \mathsf{nsb}(\ell_{21}) \geq \mathsf{nsb}(\ell_{22}) + \xi(\ell_{22})(\ell_{19}, \ell_{21}) - 1, \\ \mathsf{nsb}(\ell_{23}) \geq \mathsf{nsb}(\ell_{24}),\ \xi(\ell_{23})(\ell_{17}, \ell_{22}) \geq 1,\ \ \xi(\ell_{22})(\ell_{19}, \ell_{21}) \geq 1 \end{array} \right\} \tag{10}$$

The first two constraints are for the addition. As mentioned previously, the ufp are computed by a prior range analysis. Then, at constraint generation time, they are constants. For our example, $\mathsf{ufp}(\ell_{17}) = -1$. This quantity occurs in the first constraints. The next two constraints are for the multiplication. The fifth constraint $\mathsf{nsb}(\ell_{23}) \geq \mathsf{nsb}(\ell_{24})$ is for the assignment and the last two constraints are for the constant functions $\xi(\ell_{23})(\ell_{17}, \ell_{22})$ and $\xi(\ell_{22})(\ell_{19}, \ell_{21})$, respectively for the addition and multiplication. For a user requirement of 20 bits on the variable `y2` (all variables are in double precision initially), POP succeeds in tuning the majority of variables of the pendulum program into single precision with a total number of bits at bit level equivalent to 274 (originally the program used 689 bits). The new mixed precision formats obtained are: `y1new|20| = y1|21| +|20| y2|22| ×|22| h|22|`.

3.3 Policy Iteration for Optimized Carry Bit Propagation

The policy iteration algorithm is used to solve nonlinear fixpoint equations when the function is written as the infimum of functions for which a fixpoint can be easily computed. The infimum formulation makes the function not being differentiable in the classical sense. The one proposed in [9] to solve smallest fixpoint equations in static analysis requires the fact that the function is order-preserving to ensure the decrease of the intermediate solutions provided by the algorithm. In this article, because of the nature of the semantics, we propose a policy iterations algorithm for a non order-preserving function.

More precisely, let F be a map from a complete lattice L to itself such that $F = \inf_{\pi \in \Pi} f^{\pi}$. Classical policy iterations solve $F(\mathbf{x}) = \mathbf{x}$ by generating a sequence $(\mathbf{x}^k)_k$ such that $f^{\pi^k}(\mathbf{x}^k) = \mathbf{x}^k$ and $\mathbf{x}^{k+1} < \mathbf{x}^k$. The set Π is called the set of policies and f^{π} a policy map (associated to π). The set of policy maps has to satisfy the selection property meaning that for all $\mathbf{x} \in L$, there exists $\pi \in \Pi$ such that $F(\mathbf{x}) = f^{\pi}(\mathbf{x})$. This is exactly the same as for each $\mathbf{x} \in L$, the minimization problem $\mathrm{Min}_{\pi \in \Pi} f^{\pi}(\mathbf{x})$ has an optimal solution. If Π is finite and F is order-preserving, policy iterations converge in finite time to a fixpoint of F. The number of iterations is bounded from above by the number of policies. Indeed, a policy cannot be selected twice in the running of the algorithm. This is implied by the fact that the smallest fixpoint of a policy map is computed. In this article, we adapt policy iterations to the problem of precision tuning. The function F here is constructed from inequalities depicted in Figure 4 and Figure 5. We thus have naturally constraints of the form $F(\mathbf{x}) \leq \mathbf{x}$. We will

$$\mathcal{E}'[c\#\mathtt{p}^{\ell}]\varrho = \{\mathsf{nsb_e}(\ell) = 0\} \quad (\textsc{Const}') \qquad \mathcal{E}'[x^{\ell}]\varrho = \{\mathsf{nsb_e}(\varrho(x)) \geq \mathsf{nsb_e}(\ell)\} \quad (\textsc{Id}')$$

$$\mathcal{E}'[e_1^{\ell_1} +^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}'[e_1^{\ell_1}]\varrho \ \cup\ \mathcal{E}'[e_2^{\ell_2}]\varrho \quad (\textsc{Add}')$$
$$\cup$$
$$\left\{ \begin{array}{c} \mathsf{nsb_e}(\ell) \geq \mathsf{nsb_e}(\ell_1),\ \mathsf{nsb_e}(\ell) \geq \mathsf{nsb_e}(\ell_2), \\ \mathsf{nsb_e}(\ell) \geq \mathsf{ufp}(\ell_1) - \mathsf{ufp}(\ell_2) + \mathsf{nsb}(\ell_2) - \mathsf{nsb}(\ell_1) + \mathsf{nsb_e}(\ell_2) + \xi(\ell)(\ell_1,\ell_2), \\ \mathsf{nsb_e}(\ell) \geq \mathsf{ufp}(\ell_2) - \mathsf{ufp}(\ell_1) + \mathsf{nsb}(\ell_1) - \mathsf{nsb}(\ell_2) + \mathsf{nsb_e}(\ell_1) + \xi(\ell)(\ell_1,\ell_2) \end{array} \right\}$$

$$\mathcal{E}'[e_1^{\ell_1} -^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}'[e_1^{\ell_1}]\varrho \ \cup\ \mathcal{E}'[e_2^{\ell_2}]\varrho \quad (\textsc{Sub}')$$
$$\cup$$
$$\left\{ \begin{array}{c} \mathsf{nsb_e}(\ell) \geq \mathsf{nsb_e}(\ell_1),\ \mathsf{nsb_e}(\ell) \geq \mathsf{nsb_e}(\ell_2), \\ \mathsf{nsb_e}(\ell) \geq \mathsf{ufp}(\ell_1) - \mathsf{ufp}(\ell_2) + \mathsf{nsb}(\ell_2) - \mathsf{nsb}(\ell_1) + \mathsf{nsb_e}(\ell_2) + \xi(\ell)(\ell_1,\ell_2), \\ \mathsf{nsb_e}(\ell) \geq \mathsf{ufp}(\ell_2) - \mathsf{ufp}(\ell_1) + \mathsf{nsb}(\ell_1) - \mathsf{nsb}(\ell_2) + \mathsf{nsb_e}(\ell_1) + \xi(\ell)(\ell_1,\ell_2) \end{array} \right\}$$

$$\mathcal{E}'[e_1^{\ell_1} \times^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}'[e_1^{\ell_1}]\varrho \ \cup\ \mathcal{E}'[e_2^{\ell_2}]\varrho \quad (\textsc{Mult}')$$
$$\cup$$
$$\{\ \mathsf{nsb_e}(\ell) \geq \mathsf{nsb}(\ell_1) + \mathsf{nsb_e}(\ell_1) + \mathsf{nsb_e}(\ell_2) - 2,\ \mathsf{nsb_e}(\ell) \geq \mathsf{nsb}(\ell_2) + \mathsf{nsb_e}(\ell_2) + \mathsf{nsb_e}(\ell_1) - 2\ \}$$

$$\mathcal{E}'[e_1^{\ell_1} \div^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}'[e_1^{\ell_1}]\varrho \ \cup\ \mathcal{E}'[e_2^{\ell_2}]\varrho \quad (\textsc{Div}')$$
$$\cup$$
$$\{\ \mathsf{nsb_e}(\ell) \geq \mathsf{nsb}(\ell_1) + \mathsf{nsb_e}(\ell_1) + \mathsf{nsb_e}(\ell_2) - 2,\ \mathsf{nsb_e}(\ell) \geq \mathsf{nsb}(\ell_2) + \mathsf{nsb_e}(\ell_2) + \mathsf{nsb_e}(\ell_1) - 2\ \}$$

$$\mathcal{E}'\left[\sqrt{e^{\ell_1}}^{\ell}\right]\varrho = \mathcal{E}'[e_1^{\ell_1}]\varrho \ \cup\ \{\mathsf{nsb_e}(\ell) \geq \mathsf{nsb_e}(\ell_1)\} \quad (\textsc{Sqrt}')$$

$$\mathcal{E}'\left[\phi\left(e^{\ell_1}\right)^{\ell}\right]\varrho = \mathcal{E}'[e_1^{\ell_1}]\varrho \cup \{\mathsf{nsb_e}(\ell) \geq +\infty\} \text{ with } \phi \in \{\sin, \cos, \tan, \log, \ldots\} \quad (\textsc{Math}')$$

$$\mathcal{C}'\left[x :=^{\ell} e^{\ell_1}\right]\varrho = \left(C, \varrho\left[x \mapsto \ell\right]\right) \text{ where } C = \mathcal{E}'[e_1^{\ell_1}]\varrho \cup \{\mathsf{nsb_e}(\ell_1) \geq \mathsf{nsb_e}(\ell)\} \quad (\textsc{Assign}')$$

$$\mathcal{C}'\left[c_1^{\ell_1}\,;c_2^{\ell_2}\right]\varrho = \left(C_1 \cup C_2, \varrho_2\right) \text{ with } \left(C_1, \varrho_1\right) = \mathcal{C}'\left[c_1^{\ell_1}\right]\varrho \text{ and } \left(C_2, \varrho_2\right) = \mathcal{C}'\left[c_2^{\ell_2}\right]\varrho_1 \quad (\textsc{Seq}')$$

$$\mathcal{C}'[\mathtt{if}^{\ell}\ e^{\ell_0}\ \mathtt{then}\ c^{\ell_1}\ \mathtt{else}\ c^{\ell_2}]\ \varrho = (C_1 \cup C_2 \cup C_3, \varrho')$$
$$\text{where} \left| \begin{array}{l} \forall x \in \mathrm{Id},\ \varrho'(x) = \ell,\ (C_1, \varrho_1) = \mathcal{C}'[c_1^{\ell_1}]\ \varrho,\ (C_2, \varrho_2) = \mathcal{C}'[c_2^{\ell_2}]\ \varrho, \\ C_3 = \bigcup\limits_{x \in \mathrm{Id}} \{\mathsf{nsb_e}(\varrho_1(x)) \geq \mathsf{nsb_e}(\ell),\ \mathsf{nsb_e}(\varrho_2(x)) \geq \mathsf{nsb_e}(\ell)\} \end{array} \right. \quad (\textsc{Cond}')$$

$$\mathcal{C}'[\mathtt{while}^{\ell}\ e^{\ell_0}\ \mathtt{do}\ c^{\ell_1}]\ \varrho = (C_1 \cup C_2, \varrho')$$
$$\text{where} \left| \begin{array}{l} \forall x \in \mathrm{Id},\ \varrho'(x) = \ell,\ (C_1, \varrho_1) = \mathcal{C}'[c_1^{\ell_1}]\ \varrho' \\ C_2 = \bigcup\limits_{x \in \mathrm{Id}} \{\mathsf{nsb_e}(\varrho(x)) \geq \mathsf{nsb_e}(\ell),\ \mathsf{nsb_e}(\varrho_1(x)) \geq \mathsf{nsb_e}(\ell)\} \end{array} \right. \quad (\textsc{While}')$$

$$\mathcal{C}'[\mathtt{require\_nsb}(x, \mathtt{p})^{\ell}]\varrho = \emptyset \quad (\textsc{Req}')$$

$$\xi(\ell)(\ell_1,\ell_2) = \min \left(\begin{array}{l} \max\left(\mathsf{ufp}(\ell_2) - \mathsf{ufp}(\ell_1) + \mathsf{nsb}(\ell_1) - \mathsf{nsb}(\ell_2) - \mathsf{nsb_e}(\ell_2), 0\right), \\ \max\left(\mathsf{ufp}(\ell_1) - \mathsf{ufp}(\ell_2) + \mathsf{nsb}(\ell_2) - \mathsf{nsb}(\ell_1) - \mathsf{nsb_e}(\ell_1), 0\right), 1 \end{array} \right)$$

Fig. 5: Constraints solved by PI with min and max carry bit formulation.

give details about the construction of F at Proposition 1. Consequently, we are

interested in solving:

$$\underset{\mathsf{nsb},\mathsf{nsb_e}}{\text{Min}} \sum_{\ell} \mathsf{nsb}(\ell) \,\text{s.t.}\, F\begin{pmatrix}\mathsf{nsb}\\ \mathsf{nsb_e}\end{pmatrix} \leq \begin{pmatrix}\mathsf{nsb}\\ \mathsf{nsb_e}\end{pmatrix} \quad \mathsf{nsb} \in \mathbb{N}^{Lab},\ \mathsf{nsb_e} \in \mathbb{N}^{Lab} \tag{11}$$

Let $\xi : Lab \to \{0,1\}$. We write S^1_ξ the system of inequalities depicted in Figure 4 and S^2_ξ the system of inequalities presented at Figure 5. Note that the final system of inequalities is $S_\xi = S^1_\xi \cup S^2_\xi$ meaning that we add new constraints to S^1_ξ. If the system S^1_ξ is used alone, ξ is the constant function equal to 1. Otherwise, ξ is defined by the formula at the end of Figure 5.

Proposition 1. *The following results hold:*

1. *Let ξ the constant function equal to 1. The system S^1_ξ can be rewritten as $\{\mathsf{nsb} \in \mathbb{N}^{Lab} \mid F(\mathsf{nsb}) \leq (\mathsf{nsb})\}$ where F maps $\mathbb{R}^{Lab}$ to itself, $F(\mathbb{N}^{Lab}) \subseteq (\mathbb{N}^{Lab})$ and has coordinates which are the maximum of a finite family of affine order-preserving functions.*
2. *Let ξ the function such that $\xi(\ell)$ equals the function of Figure 5. The system S_ξ can be rewritten as $\{(\mathsf{nsb},\mathsf{nsb_e}) \in \mathbb{N}^{Lab}\times\mathbb{N}^{Lab} \mid F(\mathsf{nsb},\mathsf{nsb_e}) \leq (\mathsf{nsb},\mathsf{nsb_e})\}$ where F maps $\mathbb{R}^{Lab} \times \mathbb{R}^{Lab}$ to itself, $F(\mathbb{N}^{Lab} \times \mathbb{N}^{Lab}) \subseteq (\mathbb{N}^{Lab} \times \mathbb{N}^{Lab})$ and all its coordinates are the min-max of a finite family of affine functions.*

Note that, in the first case, F does not map from $\mathbb{R}^{Lab} \times \mathbb{R}^{Lab}$ to itself. It is easy to extend F as a map from $\mathbb{R}^{Lab} \times \mathbb{R}^{Lab}$ to itself without affecting its intrinsic behaviour. From Proposition 1, when S_ξ is used, we can write F as $F = \min_{\pi\in\Pi} f^\pi$, where f^π is the maximum of a finite family of affine functions and thus used a modified policy iterations algorithm. The set of policies here is a map $\pi : Lab \mapsto \{0,1\}$. A choice is thus a vector of 0 or 1. A policy map f^π is a function $\mathbb{N}^{Lab}$ to itself such that the coordinates are $f^\pi_\ell(\ell)$. If the coordinate $f^\pi_\ell(\ell)$ depends on ξ then $\xi(\ell) = \pi(\ell)$. Otherwise, the function is the maximum of affine functions and a choice is not required.

Corollary 1. *Any feasible solution of Problem* (11) *satisfies our ILP constraints of Figure 4 (or Figure 5 if ξ is not fixed to 1).*

Proposition 2 (Algorithm correctness). *The sequence $(\sum_{\ell\in Lab} \mathsf{nsb}^k(\ell))_{0\leq k\leq K}$ generated by Algorithm 1 satisfies the following properties:*

1. *$K < +\infty$ i.e. the sequence is of finite length;*
2. *each term of the sequence furnishes a feasible solution for Problem* (11)*;*
3. *$\sum_{\ell\in Lab} \mathsf{nsb}^{k+1}(\ell) < \sum_{\ell\in Lab} \mathsf{nsb}^k(\ell)$ if $k < K-1$ and $\sum_{\ell\in Lab} \mathsf{nsb}^K(\ell) = \sum_{\ell\in Lab} \mathsf{nsb}^{K-1}(\ell)$;*
4. *the number k is smaller than the number of policies.*

Figure 5 displays the new rules that we add to the global system of constraints in which the only difference is to activate the optimized function ξ instead of its over-approximation in Figure 4. As mentioned in Equation (6), to compute

Algorithm 1: Non-monotone Policy Iterations Algorithm

Result: An over-approximation of an optimal solution of Equation (11)

Let $k := 0$, $S := +\infty$;

Choose $\pi^0 \in \Pi$;

Select an optimal solution of $(\mathsf{nsb}^k, \mathsf{nsb_e}^k)$ the integer linear program:

$$\text{Min} \left\{ \sum_{\ell \in Lab} \mathsf{nsb}(\ell) \mid f^{\pi^k}(\mathsf{nsb}, \mathsf{nsb_e}) \leq (\mathsf{nsb}, \mathsf{nsb_e}),\ \mathsf{nsb} \in \mathbb{N}^{Lab},\ \mathsf{nsb_e} \in \mathbb{N}^{Lab} \right\} ;$$

if $\sum_{\ell \in Lab} \mathsf{nsb}^k(\ell) < S$ **then**

  $S := \sum_{\ell \in Lab} \mathsf{nsb}^k(\ell)$;

  Choose $\pi^{k+1} \in \Pi$ such that $F(\mathsf{nsb}^k, \mathsf{nsb_e}^k) = f^{\pi^{k+1}}(\mathsf{nsb}^k, \mathsf{nsb_e}^k)$;

  $k := k + 1$ and go to 3;

else

  Return S and nsb^k.

end

the ulp of the errors on the operands, we need to estimate the number of bits of the error $\mathsf{nsb_e}$ for each operand on which all the rules of Figure 5 are based. By applying this reasoning, the problem do not remain an ILP any longer. Let us concentrate on the rules of Figure 5. The function $\mathcal{E}'[e]\ \varrho$ generates the new set of constraints for an expression $e \in Expr$ in the environment ϱ. For Rule (Const′), the number of significant bits of the error $\mathsf{nsb_e} = 0$ whereas we impose that the $\mathsf{nsb_e}$ of a variable x at control point ℓ is less than the last assignment of $\mathsf{nsb_e}$ in $\varrho(x)$ as shown in Rule (Id′) of Figure 5. Considering Rule (Add′), we start by generating the new set of constraints $\mathcal{E}'[e_1^{\ell_1}]\varrho$ and $\mathcal{E}'[e_2^{\ell_2}]\varrho$ on the operands at control points ℓ_1 and ℓ_2. Then, we require that $\mathsf{nsb_e}(\ell) \geq \mathsf{nsb_e}(\ell_1)$ and $\mathsf{nsb_e}(\ell) \geq \mathsf{nsb_e}(\ell_2)$ where the result of the addition is stored at control point ℓ. Additionally, $\mathsf{nsb_e}(\ell)$ is computed as shown hereafter.

$$\mathsf{nsb_e}(\ell) \geq \max \begin{pmatrix} \mathsf{ufp}(\ell_1) - \mathsf{nsb}(\ell_1) \\ \mathsf{ufp}(\ell_2) - \mathsf{nsb}(\ell_2) \end{pmatrix} - \min \begin{pmatrix} \mathsf{ufp}(\ell_1) - \mathsf{nsb}(\ell_1) - \mathsf{nsb_e}(\ell_1) \\ \mathsf{ufp}(\ell_2) - \mathsf{nsb}(\ell_2) - \mathsf{nsb_e}(\ell_2) \end{pmatrix} + \xi(\ell)(\ell_1, \ell_2)$$

By breaking the min and max operators, we obtain the constraints on $\mathsf{nsb_e}(\ell)$ of Rule (Add′). For the subtraction, the constraints generated are similar to the addition case. Considering now Rule (Mult′), as we have defined in Section 3.2, $\varepsilon(c) = c_1 \cdot \varepsilon(c_2) + c_2 \cdot \varepsilon(c_1) + \varepsilon(c_1) \cdot \varepsilon(c_2)$ where $c = c_1^{\ell_1} \times^{\ell} c_2^{\ell_2}$. By reasoning on $\mathsf{ulp_e}$, we bound $\varepsilon(c)$ by

$$\varepsilon(c) \leq 2^{\mathsf{ufp}(c_1)} \cdot 2^{\mathsf{ufp}(c_2) - \mathsf{nsb}(c_2) - \mathsf{nsb_e}(c_2) + 1} + 2^{\mathsf{ufp}(c_2)} \cdot 2^{\mathsf{ufp}(c_1) - \mathsf{nsb}(c_1) - \mathsf{nsb_e}(c_1) + 1} + 2^{\mathsf{ufp}(c_2) + \mathsf{ufp}(c_1) - \mathsf{nsb}(c_1) - \mathsf{nsb}(c_2) - \mathsf{nsb_e}(c_1) - \mathsf{nsb_e}(c_2) + 2}$$

By selecting the smallest term $\mathsf{ufp}(c_2) + \mathsf{ufp}(c_1) - \mathsf{nsb}(c_1) - \mathsf{nsb}(c_2) - \mathsf{nsb_e}(c_1) - \mathsf{nsb_e}(c_2) + 2$, we obtain that

$$\mathsf{nsb_e}(\ell) \geq \max \begin{pmatrix} \mathsf{ufp}(\ell_1) + \mathsf{ufp}(\ell_2) - \mathsf{nsb}(\ell_1) \\ \mathsf{ufp}(\ell_1) + \mathsf{ufp}(\ell_2) - \mathsf{nsb}(\ell_2) \end{pmatrix} - \begin{array}{l} \mathsf{ufp}(\ell_1) + \mathsf{ufp}(\ell_2) - \mathsf{nsb}(\ell_1) - \\ \mathsf{nsb}(\ell_2) - \mathsf{nsb_e}(\ell_1) - \mathsf{nsb_e}(\ell_2) + 2 \end{array}$$

Finally, by simplifying the equation above we found the constraints of Rule (Mult′) in Figure 5 (same for Rule (Div′)). For Rule (Sqrt′), we generate

the constraints on the expression $\mathcal{E}'[e_1^{\ell_1}]\varrho$ and we require that $\mathsf{nsb_e}$ of the result stored at control point ℓ is greater than the $\mathsf{nsb_e}$ of the expression a control point ℓ_1. For Rule (MATH$'$) , we assume that $\mathsf{nsb_e}(\ell)$ is unbounded. Concerning the commands, we define the set $\mathcal{C}'[c]\ \varrho$ which has the same function as $\mathcal{C}$ defined in Figure 4. The reasoning on the commands also remains similar except that this time we reason on the number of bits of the errors $\mathsf{nsb_e}$. The only difference is in Rule (REQ$'$) where the set of constraints is empty. Let us recall that the constraints C_2 of Figure 5 are added to the former constraints C_1 of Figure 4 and are sent to a linear solver (GLPK in practice).

Now, let us take again the pendulum program of Figure 1. By analyzing Line 5 of our program, we have to add the following set of constraints C_2 of Equation (12), along with the former set C_1 of Equation (10). In fact, policy iteration makes it possible to break the min in the $\xi(\ell_{23})(\ell_{17},\ell_{22})$ function by choosing the max between $\mathsf{ufp}(\ell_{22}) - \mathsf{ufp}(\ell_{17}) - \mathsf{nsb}(\ell_{17}) - \mathsf{nsb}(\ell_{22}) - \mathsf{nsb_e}(\ell_{17})$ and 0, the max between $\mathsf{ufp}(\ell_{17}) - \mathsf{ufp}(\ell_{22}) + \mathsf{nsb}(\ell_{22}) - \mathsf{nsb}(\ell_{17}) - \mathsf{nsb_e}(\ell_{22})$ and 0 and the constant 1. Next, it becomes possible to solve the corresponding ILP. If no fixed point is reached, POP iterates until a solution is found. By applying this optimization, the new formats are presented in lines 5 and 6 of the bottom right corner of Figure 1: `y1new|20| = y1|21| +|20| y2|21| ×|22| h|21|`. By comparing with the formats obtained with the ILP formulation, a gain of precision of 1 bit is observed on variables `y2` and `h` (total of 272 bits at bit level for the optimized program).

$$C_2 = \left\{ \begin{array}{l} \mathsf{nsb_e}(\ell_{23}) \geq \mathsf{nsb_e}(\ell_{17}), \mathsf{nsb_e}(\ell_{23}) \geq \mathsf{nsb_e}(\ell_{22}), \\ \mathsf{nsb}(\ell_{23}) \geq -1 - 0 + \mathsf{nsb}(\ell_{22}) - \mathsf{nsb}(\ell_{17}) + \mathsf{nsb_e}(\ell_{22}) + \xi(\ell_{23},\ell_{17},\ell_{22}), \\ \mathsf{nsb_e}(\ell_{23}) \geq 0 - (-1) + \mathsf{nsb}(\ell_{17}) - \mathsf{nsb}(\ell_{22}) + \mathsf{nsb_e}(\ell_{17}) + \xi(\ell_{23},\ell_{17},\ell_{22}), \\ \mathsf{nsb_e}(\ell_{23}) \geq \mathsf{nsb_e}(\ell_{24}), \mathsf{nsb_e}(\ell_{22}) \geq \mathsf{nsb}(\ell_{19}) + \mathsf{nsb_e}(\ell_{19}) + \mathsf{nsb_e}(\ell_{21}) - 2, \\ \mathsf{nsb_e}(\ell_{22}) \geq \mathsf{nsb}(\ell_{21}) + \mathsf{nsb_e}(\ell_{21}) + \mathsf{nsb_e}(\ell_{19}) - 2, \\ \xi(\ell_{23})(\ell_{17},\ell_{22}) = \min \begin{pmatrix} \max\big(0 - 6 + \mathsf{nsb}(\ell_{17}) - \mathsf{nsb}(\ell_{22}) - \mathsf{nsb_e}(\ell_{17}), 0\big), \\ \max\big(6 - 0 + \mathsf{nsb}(\ell_{22}) - \mathsf{nsb}(\ell_{17}) - \mathsf{nsb_e}(\ell_{22}), 0\big), 1 \end{pmatrix} \end{array} \right\} \tag{12}$$

4 Correctness

4.1 Soundness of the Constraint System

Let $\equiv$ denote the syntactic equivalence and let $e^\ell \in Expr$ be an expression. We write $\mathsf{Const}(e^\ell)$ the set of constants occurring in the expression e^ℓ. For example, $\mathsf{Const}(18.0^{\ell_1} \times^{\ell_2} x^{\ell_3} +^{\ell_4} 12.0^{\ell_5} \times^{\ell_6} y^{\ell_7} +^{\ell_8} z^{\ell_9}) = \{18.0^{\ell_1}, 12.0^{\ell_5}\}$. Also, we denote by $\tau\ :\ \mathrm{Lab} \to \mathbb{N}$ a function mapping the labels of an expression to a nsb. The notation $\tau \models \mathcal{E}[e^\ell]\varrho$ means that τ is the minimal solution to the ILP $\mathcal{E}[e^\ell]\varrho$. We write $\varrho_\perp$ the empty environment ($dom(\varrho_\perp) = \emptyset$). The small step operational semantics of our language is displayed in Figure 6. It is standard, the only originality being to indicate explicitly the nsb of constants. For the result of an elementary operation, this nsb is computed in function of the nsb of the operands. Lemma 2 asses the soundness of the constraints for one step of the semantics.

$$\frac{\varrho(x) = c\#\mathtt{p}}{\langle x^{\ell}, \varrho\rangle \longrightarrow \langle c^{\ell}\#\mathtt{p}, \varrho\rangle}$$

$$\frac{c = c_1 \odot c_2,\ \mathtt{p} = \mathsf{ufp}(c) - \mathsf{ufp_e}\big(c^{\ell}\#\mathtt{p}\big)}{\langle c_1^{\ell_1}\#p_1 \odot^{\ell} c_2^{\ell_2}\#p_2, \varrho\rangle \longrightarrow \langle c\#\mathtt{p}, \varrho\rangle} \quad \odot \in \{+, -, \times, \div\}$$

$$\frac{\langle e_1^{\ell_1}, \varrho\rangle \longrightarrow \langle e_1'^{\ell_1}, \varrho\rangle}{\langle e_1^{\ell_1} \odot^{\ell} e_2^{\ell_2}, \varrho\rangle \longrightarrow \langle e_1'^{\ell_1} \odot^{\ell} e_2^{\ell_2}, \varrho\rangle} \qquad \frac{\langle e_2^{\ell_2}, \varrho\rangle \longrightarrow \langle e_2'^{\ell_2}, \varrho\rangle}{\langle c_1^{\ell_1}\#\mathtt{p} \odot^{\ell} e_2^{\ell_2}, \varrho\rangle \longrightarrow \langle c_1^{\ell_1}\sharp\mathtt{p} \odot^{\ell} e_2'^{\ell_2}, \varrho\rangle}$$

Fig. 6: Small Step Operational semantics of arithmetic expressions.

Lemma 2 *Given an expression $e^{\ell} \in Expr$, if $e^{\ell} \to e'^{\ell}$ and $\tau \models \mathcal{E}[e^{\ell}]\varrho_{\perp}$ then for all $c^{\ell_c}\#\boldsymbol{p} \in \mathsf{Const}(e'^{\ell})$ we have $\boldsymbol{p} = \tau(\ell_c)$.*

Proof. By case examination of the rules of Figure 4. Hereafter, we focus on the most interesting case of addition of two constants. Recall that $\mathsf{ufp_e}(\ell) = \mathsf{ufp}(\ell) - \mathsf{nsb}(\ell)$ for any control point ℓ. Assuming that $e^{\ell} \equiv c_1^{\ell_1} +^{\ell} c_2^{\ell_2}$ then by following the reduction rule of Figure 6, we have $e^{\ell} \to c^{\ell}\#\mathtt{p}$ with $\mathtt{p} = \mathsf{ufp}(c) - \mathsf{ufp_e}\big(c\big)$. On the other side, by following the set of constraints of Rule (ADD) in Figure 4 we have $\mathcal{E}[e^{\ell}]\varrho = \{\mathsf{nsb}(\ell_1) \geq \mathsf{nsb}(\ell) + \mathsf{ufp}(\ell_1) - \mathsf{ufp}(\ell) + \xi(\ell)(\ell_1, \ell_2),\ \mathsf{nsb}(\ell_2) \geq \mathsf{nsb}(\ell) + \mathsf{ufp}(\ell_2) - \mathsf{ufp}(\ell) + \xi(\ell)(\ell_1, \ell_2)\}$. These constraints can be written as

$$\mathsf{nsb}(\ell) \leq \mathsf{ufp}(\ell) - \mathsf{ufp}(\ell_1) + \mathsf{nsb}(\ell_1) - \xi(\ell)(\ell_1, \ell_2)$$
$$\mathsf{nsb}(\ell) \leq \mathsf{ufp}(\ell) - \mathsf{ufp}(\ell_2) + \mathsf{nsb}(\ell_2) - \xi(\ell)(\ell_1, \ell_2)$$

and may themselves be rewritten as Equation (8), i.e.

$$\mathsf{nsb}(\ell) \leq \mathsf{ufp}(\ell) - \max\big(\mathsf{ufp}(\ell_1) - \mathsf{nsb}(\ell_1), \mathsf{ufp}(\ell_2) - \mathsf{nsb}(\ell_2)\big) - \xi(\ell)(\ell_1, \ell_2)\ .$$

Since, obviously, $\mathsf{ufp}(c) = \mathsf{ufp}(\ell)$ and since the solver finds the minimal solution to the ILP, it remains to show that

$$\mathsf{ufp_e}(\ell) = \max\big(\mathsf{ufp}(\ell_1) - \mathsf{nsb}(\ell_1), \mathsf{ufp}(\ell_2) - \mathsf{nsb}(\ell_2), \mathsf{ufp}(\ell) - \mathsf{prec}(\ell)\big) + \xi(\ell)(\ell_1, \ell_2)$$

which corresponds to the assertion of Equation(7). Consequently, $\mathsf{nsb}(\ell) = \mathtt{p}$ as required, for this case, in Figure 6. □

Theorem 1. *Given an expression $e^{\ell} \to e'^{\ell}$. If $e^{\ell} \to^* e'^{\ell}$ and if $\tau \models \mathcal{E}[e^{\ell}]\varrho_{\perp}$, then $\forall\ c^{\ell_c}\#\boldsymbol{p} \in \mathsf{Const}(e'^{\ell})$ we have $\boldsymbol{p} = \tau(\ell_c)$.*

4.2 ILP Nature of the Problem

In this section, we give insights about the complexity of the problem. The computation relies on integer linear programming. Integer linear programming is known to belong to the class of NP-Hard problems. A lower bound of the optimal value in a minimization problem can be furnished by the continuous linear programming relaxation. This relaxation is obtained by removing the integrity constraint. Recall that a (classical) linear program can be solved in polynomial-time. Then, we can solve our problem in polynomial-time if we can show that

the continuous linear programming relaxation of our ILP has an unique optimal solution with integral coordinates. Proposition 3 presents a situation where a linear program has a unique optimal solution which is a vector of integers.

Proposition 3. *Let $G : [0,+\infty)^d \mapsto [0,+\infty)^d$ be an order-preserving function such that $G(\mathbb{N}^d) \subseteq \mathbb{N}^d$. Suppose that the set $\{y \in \mathbb{N}^d \mid G(y) \leq y\}$ is non-empty. Let $\varphi : \mathbb{R}^d \mapsto \mathbb{R}$ a strictly monotone function such that $\varphi(\mathbb{N}^d) \subseteq \mathbb{N}$. Then, the minimization problem below has an unique optimal solution which is integral.*

$$\underset{y \in [0,+\infty)^d}{\text{Min}} \ \varphi(y) \ \text{ s. t. } G(y) \leq y$$

Theorem 2. *Assume that the system S of inequalities depicted in Figure 4 has a solution. The smallest amount of memory $\sum_{\ell \in Lab} \mathsf{nsb}(\ell)$ for S can be computed in polynomial-time by linear programming.*

Proof. The function $\sum_{\ell \in Lab} \mathsf{nsb}(\ell)$ is strictly monotone and stable on integers. From the first statement of Proposition 1, the system of constraints is of the form $F(\mathsf{nsb}) \leq \mathsf{nsb}$ where F is order-preserving and stable on integers. By assumption, there exists a vector of integers nsb s.t. $F(\mathsf{nsb}) \leq \mathsf{nsb}$. We conclude from Proposition 3. □

For the second system, in practice, we get integral solutions to the continuous linear programming relaxation of our ILP of Equation (11). However, because of the lack of monotonicity of the functions for the rules (ADD) and (SUB) of Figure 4, we cannot exploit Proposition 4 to prove the polynomial-time solvability.

5 Experimental Results

In this section, we aim at evaluating the performance of our tool POP implementing the techniques of Section 3. We have evaluated POP on several numerical programs. Two of them were used as a benchmark for precision tuning in prior work [24] and are coming from the GNU scientific library (GSL): **arclength** and **simpson** program which corresponds to an implementation of the widely used Simpson's rule [19]. The next three programs were used as benchmarks for POP in its former version [2,3,6]. The **rotation** program performs a matrix-vector product to rotate a vector around the z axis by an angle of θ [6]. The **accelerometer** program measures an inclination angle [2]. The **lowPassFilter** program [3] is taken from a pedometer application [20]. These last two programs come from the IoT field. We also experiment POP on a **2-Body** problem program and the **pendulum** program already introduced in Section 2.

The experiments shown in Table 1 present the tuning results produced by POP for each error threshold 10^{-4}, 10^{-6}, 10^{-8} and 10^{-10}. This is for compatibility with Precimonious which uses decimal thresholds. Technically, we translate these error thresholds into nsb. In Table 1, we represent by "TH" the error threshold given by the user. "BL" is the percentage of optimization at bit level. "IEEE" denotes the percentage of optimized variables in IEEE754 formats (`binary16`, `binary32`, etc.) In IEEE mode, the nsb obtained at bit level is approximated by the upper number of bits corresponding to a IEEE754 format. "ILP-time"

Program	TH	BL	IEEE	ILP-time	BL	IEEE	PI-time	H	S	D	LD
	10^{-4}	61%	43%	0.9s	62%	45%	1.5s	8	88	25	0
	10^{-6}	50%	21%	0.9s	51%	21%	1.4s	2	45	74	0
arclength	10^{-8}	37%	3%	0.8s	38%	4%	1.6s	2	6	113	0
	10^{-10}	24%	-1%	1.0s	25%	-1%	1.7s	2	0	116	3
	10^{-12}	12%	-17%	0.3s	14%	-8%	1.5s	2	0	109	10
	10^{-4}	64%	45%	0.1s	67%	56%	0.5s	6	42	1	0
	10^{-6}	53%	30%	0.2s	56%	31%	0.5s	1	27	21	0
simpson	10^{-8}	40%	4%	0.1s	43%	7%	0.3s	1	5	43	0
	10^{-10}	27%	1%	0.1s	28%	1%	0.4s	1	0	48	0
	10^{-12}	16%	1%	0.1s	16%	1%	0.3s	0	1	48	0
	10^{-4}	73%	61%	0.2s	76%	62%	1.0s	53	69	0	0
	10^{-6}	62%	55%	0.2s	65%	55%	1.0s	2	102	0	0
accelerometer	10^{-8}	49%	15%	0.2s	52%	18%	1.0s	2	33	69	0
	10^{-10}	36%	1%	0.2s	39%	1%	1.0s	2	0	102	0
	10^{-12}	25%	1%	0.2s	28%	1%	1.0s	2	0	102	0
	10^{-4}	78%	66%	0.08s	79%	68%	1.3s	46	38	0	0
	10^{-6}	67%	53%	0.08s	68%	56%	0.5s	12	70	2	0
rotation	10^{-8}	53%	29%	0.07s	54%	29%	0.4s	0	46	38	0
	10^{-10}	40%	0%	0.1s	41%	0%	0.5s	0	0	84	0
	10^{-12}	29%	0%	0.09s	30%	0%	0.5s	0	0	48	0
	10^{-4}	68%	46%	1.8s	69%	46%	10.7s	260	581	0	0
	10^{-6}	57%	38%	1.8s	58%	45%	11.0s	258	580	3	0
lowPassFilter	10^{-8}	44%	-7%	2.0s	45%	-7%	11.4s	258	2	581	0
	10^{-10}	31%	-7%	1.7s	32%	-7%	10.9s	258	0	583	0
	10^{-12}	20%	-7%	1.8s	21%	-7%	11.3s	258	0	583	0
	10^{-4}	41%	51%	0.81s	41%	51%	0.82s	5	39	5	0
	10^{-6}	18%	49%	0.78s	18%	49%	0.9s	0	44	5	0
2-Body	10^{-8}	-7%	5%	0.8s	-7%	5%	0.78s	0	5	44	0
	10^{-10}	-34%	-2%	0.8s	-34%	-2%	0.9	0	0	48	1
	10^{-12}	-57%	-11%	0.9s	-57%	-11%	1.0s	0	0	44	0
	10^{-4}	71%	54%	0.15s	71%	54%	0.4s	0	13	0	0
	10^{-6}	60%	50%	0.2s	60%	50%	0.5s	0	12	1	0
Pendulum	10^{-8}	47%	0%	0.12s	47%	0%	0.4s	0	0	13	0
	10^{-10}	33%	0%	0.16s	34%	0%	0.5	0	0	13	0
	10^{-12}	22%	0%	0.11s	22%	0%	0.4s	0	0	13	0

Table 1: Precision tuning results for POP for the ILP and PI methods.

is the total analysis time of POP in the case of ILP formulation. We have also "PI-time" to represent the time passed by POP to find the right policy and to resolve the precision tuning problem. "H", "S", "D" and "LD" denote respectively the number of variables obtained in, half, single, double and long-double precision when using the PI formulation that clearly displays better results.

Let us focus on the first "TH", "BL", "IEEE" and "ILP-time" columns of Table 1. We compute the improvements compared to the case where all variables are in double precision before tuning. For the **arclength** program, the optimization reaches 61% at bit-level while it achieves 43% in IEEE mode (100% is the percentage of all variables initially in double precision, 121 variables for the original **arclength** program that used 7744 bits). This is obtained in only 0.9 second by applying the ILP formulation. When we refine the solution by applying the policy iteration method (from the sixth column), POP attains 62% at bit-level and 43% for the IEEE mode. Although POP needs more analysis time to find

Program	Tool	#Bits saved - Time in seconds			
		Threshold 10^{-4}	Threshold 10^{-6}	Threshold 10^{-8}	Threshold 10^{-10}
arclength	POP(ILP) (28)	**2464b.** - 1.8s.	**2144b.** - 1.5s.	**1792b.** - 1.7s.	**1728b.** - 1.8s.
	POP(SMT) (22)	1488b. - 4.7s.	1472b. - 3.04s.	864b. - 3.09s.	384b. - 2.9s.
	Precimonious (9)	576b. - 146.4s.	576b. - 156.0s.	576b. - 145.8s.	576b. - 215.0s.
simpson	POP(ILP) (14)	**1344b.** - 0.4s.	**1152b.** - 0.5s.	**896b.** - 0.4s.	**896b.** - 0.4s.
	POP(SMT) (11)	896b. - 2.9s.	896b. - 1.9s.	704b. - 1.7s.	704b. - 1.8s.
	Precimonious (10)	704b. - 208.1s.	704b. - 213.7s.	704b. - 207.5s.	704b. - 200.3s.
rotation	POP(ILP) (25)	**2624b.** - 0.47s.	2464b. - 0.47s.	2048b. - 0.54s.	1600b. - 0.48s.
	POP(SMT) (22)	1584b. - 1.85s.	2208b. - 1.7s.	1776b. - 1.6s.	1600b. - 1.7s.
	Precimonious (27)	2400b. - 9.53s.	**2592b.** - 12.2s.	**2464b.** - 10.7s.	**2464b.** - 7.4s.
accel.	POP(ILP) (18)	**1776b.** - 1.05s.	**1728b.** - 1.05s.	**1248b.** - 1.04s.	**1152b.** - 1.03s.
	POP(SMT) (15)	1488b. - 2.6s.	1440b. - 2.6s.	1056 - 2.4s.	960b. - 2.4s.
	Precimonious (0)	-	-	-	-

Table 2: Comparison between POP(ILP), POP(SMT) and Precimonious: number of bits saved by the tool and time in seconds for analyzing the programs.

and iterate between policies, the time of analysis remain negligible, not exceeding 1.5 seconds. For a total of 121 variables for the **arclength** original program, POP succeeds in tuning 8 variables to half precision (H), 88 variables passes to single precision (S) whereas 25 variables remain in double precision (D) for an error threshold of 10^{-4}. We remark that our second method displays better results also for the other user error thresholds. For the **simpson**, **accelerometer**, **rotation** and **lowPassFilter**, the improvement is also more important when using the PI technique than when using the ILP formulation. For instance, for an error threshold of 10^{-6} for the **simpson** program, only one variable passes to half precision, 27 variables turns to single precision while 21 remains in double precision with 56% of percentage of total number of bits at bit level using the policy iteration method. Concerning the **2-Body** and the **pendulum** codes, the two techniques return the same percentage at bit level and IEEE mode for the majority of error thresholds except for the **pendulum** program where POP reaches 34% at bit level when using the PI method for a threshold of 10^{-10}.

Now, we stress on the negative percentages that we obtain in Table 1, especially for the **arclength** program with 10^{-10} and 10^{-12} for the columns IEEE, the **lowPassFilter** program for errors of 10^{-8}, 10^{-10} and 10^{-12} and finally for the **2-Body** for almost all the error thresholds. In fact, POP is able to return new formats for any threshold required by the user without additional cost nor by increasing the complexity even if it fails to have a significant improvement on the program output. To be specific, taking again the **arclength** program, for an error of 10^{-12}, POP fulfills this requirement by informing the user that this precision is achievable only if 10 variables passes to the long double precision (LD) which is more than the original program whose variables are all in double precision. By doing so, the percentage of IEEE formats for both ILP and PI formulations reaches -17% and -8%, respectively. Same reasoning is adopted for the **lowPassFilter** which spends more time, nearly 12 seconds, with the policy iteration technique to find the optimized formats (total of 841 variables). For the **2-Body** program, for an error threshold of 10^{-8}, the number of bits after

optimization attains 2452 bits where the original program used only 2597 bits which corresponds to a percentage of -7% at bit level. Note that in these cases, other tools like Precimonious [24] fail to propose formats.

Table 2 shows a comparison between the new version of POP combining both ILP and PI formulations called POP(ILP), the former version of POP that uses the Z3 SMT solver coupled to binary search to find optimal solution [6], called POP(SMT) and the prior state-of the-art Precimonious [24]. The results of the mixed precision tuning are shown for the **arclength**, **simpson**, **rotation** and **accelerometer** programs. Let us mention that some examples used in Precimonious benchmarks [24] cannot be analyzed as-is by POP for implementation reasons (calls to external libraries or use of syntactic forms not yet implemented in our tool). Conversely, let us also mention that Precimonious fails to tune (with zero improvement) some examples handled by POP, e.g. **lowPassFilter**. Since POP (in its both versions) and Precimonious implement two different techniques, we have adjusted the criteria of comparison in several points. First, we mention that POP optimizes much more variables than Precimonious. While it disadvantages POP, we only consider in the experiments of Table 2 the variables optimized by Precimonious to estimate the quality of the optimization. Second, let us note that the error thresholds are expressed in base 2 in POP and in base 10 in Precimonious. For the relevance of comparisons, all the thresholds are expressed in base 10 in tables 1 and 2. In practice, POP will use the base 2 threshold immediately lower than the required base 10 threshold. In Table 2, we indicate in bold the tool that exhibits better results for each error threshold and each program. Starting with the **arclength** program, POP(ILP) displays better results than the other tools by optimizing 28 variables. For an error threshold of 10^{-4}, 2464 bits are saved by POP(ILP) in 1.8 seconds while POP(SMT) saved only 1488 bits in more time (11 seconds). Precimonious were the slowest tool on this example, more than 2 minutes with 576 bits for only 9 variables optimized. For the **simpson** program, POP(ILP) do also better than both other tools. However, for the **rotation** program, POP(ILP) saves more bits than the other tools only for an error of 10^{-4} while Precimonious do well for this program for the rest of error thresholds. Finally, Precimonious fails to tune the **accelerometer** program (0 variables) at the time that POP(ILP) do faster (only 1 second) to save much more bits than POP(SMT) for any given error threshold.

In [4,5], we show how POP generates MPFR code [13] with the precision returned by the tuning and we run the programs. We also run a MPFR version with high precision (e.g. 300 bits) and compute the error that we compare to the threshold. The results show that the thresholds are respected.

6 Conclusion and Perspectives

In this article, we have introduced a new technique for precision tuning, clearly different from the existing ones. Instead of changing more or less randomly the data types of the numerical variables and running the programs to see what happens, we propose a semantical modelling of the propagation of the numerical

errors throughout the code. This yields a system of constraints whose minimal solution gives the best tuning of the program, furthermore, in polynomial time. Two variants of this system are proposed. The first one corresponds to a pure ILP. The second one, which optimizes the propagation of carries in the elementary operations can be solved using policy iterations [9]. Proofs of correctness concerning the soundness of the analysis and the integer nature of the solutions have been presented in Section 4 and experimental results showing the efficiency of our method have been introduced in Section 5.

Compared to other approaches, the strength of our method is to find directly the minimal number of bits needed at each control point to get a certain accuracy on the results. Consequently, it is not dependant of a certain number of data types (e.g. the IEEE754 formats) and its complexity does not increase as the number of data types increases. The information provided may also be used to generate computations in the fixpoint arithmetic with an accuracy guaranty on the results. Concerning scalability, we generate a linear number of constraints and variables in the size of the analyzed program. The only limitation is the size of the problem accepted by the solver. Note that the number of variables could be reduced by assigning the same precision to a whole piece of code (for example an arithmetic expression, a line of code, a function, etc.) Code synthesis for the fixpoint arithmetic and assigning the same precision to pieces of code are perspectives we aim at explore at short term.

At longer term, other developments of the present work are planned. First we wish to adapt the techniques developed in this article to the special case of Deep Neural Networks for which it is important to save memory usage and computational resources. Second, we aim at using our precision tuning method to guide lossy compression techniques for floating-point datasets [12]. In this case, the bit-level accuracy inferred by our method would determine the compression rate of the lossy technique.

References

1. ANSI/IEEE: IEEE Standard for Binary Floating-point Arithmetic, std 754-2008 edn. (2008)
2. Ben Khalifa, D., Martel, M.: Precision tuning and internet of things. In: International Conference on Internet of Things, Embedded Systems and Communications, IINTEC 2019. pp. 80–85. IEEE (2019)
3. Ben Khalifa, D., Martel, M.: Precision tuning of an accelerometer-based pedometer algorithm for iot devices. In: International Conference on Internet of Things and Intelligence System, IOTAIS 2020. pp. 113–119. IEEE (2020)
4. Ben Khalifa, D., Martel, M.: An evaluation of POP performance for tuning numerical programs in floating-point arithmetic. In: 4th International Conference on Information and Computer Technologies, ICICT 2021, Kahului, HI, USA, March 11-14, 2021. pp. 69–78. IEEE (2021)
5. Ben Khalifa, D., Martel, M.: A study of the floating-point tuning behaviour on the n-body problem. In: The 2021 International Conference on Computational Science and Its Applications. Springer (2021)

6. Ben Khalifa, D., Martel, M., Adjé, A.: POP: A tuning assistant for mixed-precision floating-point computations. In: Formal Techniques for Safety-Critical Systems - 7th International Workshop, FTSCS 2019. Communications in Computer and Information Science, vol. 1165, pp. 77–94. Springer (2019)
7. Cherubin, S., Agosta, G.: Tools for reduced precision computation: A survey. ACM Comput. Surv. **53**(2) (2020)
8. Chiang, W., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., Rakamaric, Z.: Rigorous floating-point mixed-precision tuning. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL. pp. 300–315. ACM (2017)
9. Costan, A., Gaubert, S., Goubault, E., Martel, M., Putot, S.: A policy iteration algorithm for computing fixed points in static analysis of programs. In: Computer Aided Verification, 17th International Conference, CAV 2005. Lecture Notes in Computer Science, vol. 3576, pp. 462–475. Springer (2005)
10. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astreé analyzer. In: Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005. Lecture Notes in Computer Science, vol. 3444, pp. 21–30. Springer (2005)
11. Darulova, E., Horn, E., Sharma, S.: Sound mixed-precision optimization with rewriting. In: Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS. pp. 208–219. IEEE Computer Society / ACM (2018)
12. Diffenderfer, J., Fox, A., Hittinger, J.A.F., Sanders, G., Lindstrom, P.G.: Error analysis of ZFP compression for floating-point data. SIAM J. Sci. Comput. **41**(3), A1867–A1898 (2019)
13. Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P.: Mpfr: A multiple-precision binary floating-point library with correct rounding. ACM Trans. Math. Softw. **33** (2007)
14. Guo, H., Rubio-González, C.: Exploiting community structure for floating-point precision tuning. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018. pp. 333–343. ACM (2018)
15. Gustafson, Yonemoto: Beating floating point at its own game: Posit arithmetic. Supercomput. Front. Innov.: Int. J. **4**(2), 71–86 (2017). https://doi.org/10.14529/jsfi170206
16. Kotipalli, P.V., Singh, R., Wood, P., Laguna, I., Bagchi, S.: AMPT-GA: automatic mixed precision floating point tuning for GPU applications. In: Proceedings of the ACM International Conference on Supercomputing, ICS. pp. 160–170. ACM (2019)
17. Lam, M.O., Hollingsworth, J.K., de Supinski, B.R., LeGendre, M.P.: Automatically adapting programs for mixed-precision floating-point computation. In: International Conference on Supercomputing, ICS'13. pp. 369–378. ACM (2013)
18. Martel, M.: Floating-point format inference in mixed-precision. In: NASA Formal Methods - 9th International Symposium, NFM. Lecture Notes in Computer Science, vol. 10227, pp. 230–246 (2017)
19. McKeeman, W.M.: Algorithm 145: Adaptive numerical integration by simpson's rule. Commun. ACM **5**(12), 604 (1962)
20. Morris, D., Saponas, T., Guillory, A., Kelner, I.: Recofit: Using a wearable sensor to find, recognize, and count repetitive exercises. Conference on Human Factors in Computing Systems (2014)
21. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 4963, pp. 337–340. Springer (2008)

22. Papadimitriou, C.H.: On the complexity of integer programming. Journal of the ACM (JACM) **28**(4), 765–768 (1981)
23. Parker, D.S.: Monte carlo arithmetic: exploiting randomness in floating-point arithmetic. Tech. Rep. CSD-970002, University of California (Los Angeles) (1997)
24. Rubio-González, C., Nguyen, C., Nguyen, H.D., Demmel, J., Kahan, W., Sen, K., Bailey, D.H., Iancu, C., Hough, D.: Precimonious: tuning assistant for floating-point precision. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13. pp. 27:1–27:12. ACM (2013)
25. Schrijver, A.: Theory of linear and integer programming. John Wiley & Sons (1998)

7 Appendix

We need a lemma on some algebraic operations stable on the set of functions written as the min-max of a finite family of affine functions. The functions are defined on $\mathbb{R}^d$.

Lemma 1. *The following statements hold:*

- *The sum of two min-max of a finite family of affine functions is a min-max of a finite family of affine functions.*
- *The maximum of two min-max of a finite family of affine functions is a min-max of a finite family of affine functions.*

Proof. Let g and h be two min-max of a finite family of affine functions and $f = g + h$. We have $g = \min_i \max_j g^{ij}$ and $h = \min_k \max_l h^{kl}$. Let $x \in \mathbb{R}^d$. There exist i, k such that $f(x) \geq \max_j g^{ij}(x) + \max_l h^{kl}(x) = \max_{j,l} g^{ij}(x) + h^{kl}(x)$. We have also, for all i, k, $f(x) \leq \max_j g^{ij}(x) + \max_l h^{kl}(x) = \max_{j,l} g^{ij}(x) + h^{kl}(x)$. We conclude that $f(x) = \min_{i,k} \max_{j,l} g^{ij}(x) + h^{kl}(x)$ for all x. We use the same argument for the max. □

Proposition 2. *The following results hold:*

1. *Let ξ the constant function equal to 1. The system S^1_ξ can be rewritten as $\{\mathsf{nsb} \in \mathbb{N}^{Lab} \mid F(\mathsf{nsb}) \leq (\mathsf{nsb})\}$ where F maps $\mathbb{R}^{Lab} \times \mathbb{R}^{Lab}$ to itself, $F(\mathbb{N}^{Lab} \times \mathbb{N}^{Lab}) \subseteq (\mathbb{N}^{Lab} \times \mathbb{N}^{Lab})$ and has coordinates which are the maximum of a finite family of affine order-preserving functions.*
2. *Let ξ the function such that $\xi(\ell)$ equals the function defined at Fig. 5. The system S_ξ can be rewritten as $\{(\mathsf{nsb}, \mathsf{nsb_e}) \in \mathbb{N}^{Lab} \times \mathbb{N}^{Lab} \mid F(\mathsf{nsb}, \mathsf{nsb_e}) \leq (\mathsf{nsb}, \mathsf{nsb_e})\}$ where F maps $\mathbb{R}^{Lab} \times \mathbb{R}^{Lab}$ to itself, $F(\mathbb{N}^{Lab} \times \mathbb{N}^{Lab}) \subseteq (\mathbb{N}^{Lab} \times \mathbb{N}^{Lab})$ and all its coordinates are the min-max of a finite family of affine functions.*

Proof. We only give details about the system S^1_ξ (Figure 4). By induction on the rules. We write $L = \{\ell \in Lab \mid F_\ell \text{ is constructed }\}$. This set is used in the proof to construct F inductively.

For the rule (CONST), there is nothing to do. For the rule (ID), if the label $\ell' = \rho(x) \in L$ then we define $F_{\ell'}(\mathsf{nsb}) = \max(F_{\ell'}(\mathsf{nsb}), \mathsf{nsb}(\ell))$. Otherwise $F_{\ell'}(\mathsf{nsb}) = \mathsf{nsb}(\ell)$. As $\mathsf{nsb} \mapsto \mathsf{nsb}(\ell)$ is order-preseving and the maximum of one affine function, $F_{\ell'}$ is the maximum of a finite family of order-preserving affine functions since max preserves order-preservation.

For the rules (Add), (Sub), (Mult), (Div), (Math) and (Assign), by induction, it suffices to focus on the new set of inequalities. If $\ell_1 \in L$, we define F_{ℓ_1} as the max with old definition and $RHS(\mathsf{nsb})$ i.e. $F_{\ell_1}(\mathsf{nsb}) = \max(RHS(\mathsf{nsb}), F_{\ell_1}(\mathsf{nsb}))$ where $RHS(\mathsf{nsb})$ is the right-hand side part of the new inequality. If $\ell_1 \notin L$, we define $F_{\ell_1}(\mathsf{nsb}) = RHS(\mathsf{nsb})$. In the latter rules, $RHS(\mathsf{nsb})$ are order-preserving affine functions. It follows that F_{ℓ_1} is the maximum of a finite family of order-preserving affine functions.

The result follows by induction for the rule (SEQ).

The rules (Cond) and (While) are treated as the rules (Add), (Sub), (Mult), (Div), (Math) and (Assign), by induction and the consideration of the new set of inequalities.

The last rule (Req) constructs $F_{\rho(x)}$ either as the constant function equal to p at label $\rho(x)$ or the maximum of the old definition of $F_{\rho(x)}$ and p if $\rho(x) \in L$. The proof for the system S_ξ uses the same arguments and Lemma 1. □

Proposition 3 (Algorithm correctness). *The sequence* $(\sum_{\ell \in Lab} \mathsf{nsb}^k(\ell))_{0 \leq k \leq K}$ *generated by Algorithm 1 satisfies the following properties:*

1. $K < +\infty$ *i.e. the sequence is of finite length;*
2. *each term of the sequence furnishes a feasible solution for Problem* (11)*;*
3. $\sum_{\ell \in Lab} \mathsf{nsb}^{k+1}(\ell) < \sum_{\ell \in Lab} \mathsf{nsb}^k(\ell)$ *if* $k < K-1$ *and* $\sum_{\ell \in Lab} \mathsf{nsb}^K(\ell) = \sum_{\ell \in Lab} \mathsf{nsb}^{K-1}(\ell)$*;*
4. *the number* k *is smaller than the number of policies.*

Proof. Let $\sum_{\ell \in Lab} \mathsf{nsb}^k(\ell)$ be a term of the sequence and $(\mathsf{nsb}^k, \mathsf{nsb_e}^k)$ be the optimal solution of $\mathrm{Min}\{\sum_{\ell \in Lab} \mathsf{nsb}(\ell) \mid f^{\pi^k}(\mathsf{nsb}, \mathsf{nsb_e}) \leq (\mathsf{nsb}, \mathsf{nsb_e}),\ \mathsf{nsb} \in \mathbb{N}^{Lab},\ \mathsf{nsb_e} \in \mathbb{N}^{Lab}\}$. Then $F(\mathsf{nsb}^k, \mathsf{nsb_e}^k) \leq f^{\pi^k}(\mathsf{nsb}^k, \mathsf{nsb_e}^k)$ by definition of F. Moreover, $F(\mathsf{nsb}^k, \mathsf{nsb_e}^k) = f^{\pi^{k+1}}(\mathsf{nsb}^k, \mathsf{nsb_e}^k)$ and $f^{\pi^k}(\mathsf{nsb}^k, \mathsf{nsb_e}^k) \leq (\mathsf{nsb}^k, \mathsf{nsb_e}^k)$. This proves the second statement. Furthermore, it follows that $f^{\pi^{k+1}}(\mathsf{nsb}^k, \mathsf{nsb_e}^k) \leq (\mathsf{nsb}^k, \mathsf{nsb_e}^k)$ and $(\mathsf{nsb}^k, \mathsf{nsb_e}^k)$ is feasible for the minimisation problem for which $(\mathsf{nsb}^{k+1}, \mathsf{nsb_e}^{k+1})$ is an optimal solution. We conclude that $\sum_{\ell \in Lab} \mathsf{nsb}^{k+1}(\ell) \leq \sum_{\ell \in Lab} \mathsf{nsb}^k(\ell)$ and the Algorithm terminates if the equality holds or continues as the criterion strictly decreases. Finally, from the strict decrease, a policy cannot be selected twice without terminating the algorithm. In conclusion, the number of iterations is smaller than the number of policies. □

Proposition 4. *Let* $G : [0, +\infty)^d \mapsto [0, +\infty)^d$ *be an order-preserving function such that* $G(\mathbb{N}^d) \subseteq \mathbb{N}^d$*. Suppose that the set* $\{y \in \mathbb{N}^d \mid G(y) \leq y\}$ *is non-empty. Let* $\varphi : \mathbb{R}^d \mapsto \mathbb{R}$ *a strictly monotone function such that* $\varphi(\mathbb{N}^d) \subseteq \mathbb{N}$*. Then, the minimization problem below has an unique optimal solution which is integral.*

$$\min_{y \in [0,+\infty)^d} \varphi(y) \ \text{ s.t. } G(y) \leq y$$

Proof. Let $L := \{x \in [0, +\infty)^d \mid G(x) \leq x\}$ and $u = \inf L$. It suffices to prove that $u \in \mathbb{N}^d$. Indeed, as φ is stricly monotone then $\varphi(u) < \varphi(x)$ for all $x \in [0, +\infty)^d$ s.t. $G(x) \leq x$ and $x \neq u$. The optimal solution is thus u. If $u = 0$, the result holds. Now suppose that $0 < u$, then $0 \leq G(0)$. Let $M := \{y \in \mathbb{N}^d \mid y \leq G(y), y \leq u\}$. Then $0 \in M$ and we write $v := \sup M$. As M is a complete lattice s.t. $G(M) \subseteq M$, from Tarski's theorem, v satisfies $G(v) = v$ and $v \leq u$. Moreover, $v \in \mathbb{N}^d$ and $v \leq u$. Again, from Tarski's theorem, u is the smallest fixpoint of G, it coincides with v. Then $u \in \mathbb{N}^d$. □