# An Evaluation of POP Performance for Tuning Numerical Programs in Floating-Point Arithmetic

Dorra Ben Khalifa[1]
[1] *LAMPS - University of Perpignan Via Domitia*
52 Av. P. Alduy, 66100, Perpignan, France
dorra.ben-khalifa@univ-perp.fr

Matthieu Martel[1,2]
[2] *Numalis*
Cap Omega, Rond-point B. Franklin, Montpellier, France
matthieu.martel@univ-perp.fr

***Abstract*—Most of today's numerical computations are performed using floating-point data operations for representing real numbers. The precision of the related data types should be adapted in order to guarantee a desired overall rounding error and to strengthen the performance of programs. This adaptation lies to the precision tuning technique, which consists of finding the least floating-point formats enabling a program to compute some results with an accuracy requirement. Precision tuning is important for several applications from a wide variety of domains including embedded systems, Internet of Things (IoT), etc. In this work, we evaluate the performance of POP, our static analysis tool, in several manners. First, we compare two ways of optimizing programs in POP throughout the definition of different cost functions given to the solver. Second, we measure the runtime errors between the exact results given by an execution in multiple precision and the results of tuned programs by POP and show that the measured error is always less than the accuracy requirements given by the user for several examples. Third, we provide a detailed comparison of POP versus the prior state-of-the-art tool, Precimonious.**

***Index Terms*—Floating-point arithmetic, precision tuning, static program analysis, numerical accuracy.**

## I. Introduction

Nowadays, with the wide availability of processors with hardware floating-point units, e.g. for smartphones or other IoT devices, many applications rely on floating-point computations. In practice, these applications such as the critical control-command systems for aeronautic, automotive, etc., have stringent correctness requirements and limitations related to accuracy issues including extreme sensitivity to rounding errors and poorly handled numerical exceptions [1]. This has led to several bugs and catastrophic consequences [2]. Without any extensive background in numerical accuracy and computer arithmetic and to strengthen the accuracy of floating-point programs, developers tend to use the highest precision available in hardware (usually double precision). While more robust, this can degrade program performance significantly: the application runtime, bandwidth capacity and the memory and energy consumption of the system.

Besides, reducing the floating-point precision of certain variables, also called Precision Tuning, is a possible solution to speedup the execution or to save resources such as memory and energy. Precision tuning has resulted in the development of several tools based on static [3]–[7] or dynamic methods [8]–[12] to help programmers explore the trade-off between floating-point precision and performance. Their main objective is to improve performance by reducing accuracy while maintaining a precision constraint. In this article, we present an evaluation of the performance of our static analysis based tool named POP (short for "Precision OPtimizer") that, by minimizing the over-estimated precision of the inputs and intermediary results, guarantees a desired precision on the outputs of floating-point programs. Our tool takes as input a program annotated with the accuracy required by the user and it implements a forward and backward static program analysis as a set of constraints. The analysis is expressed as a set of propositional formulae on constraints between integer variables only (checked by Z3) [13]. The constraints are made of inequalities between linear arithmetic expressions possibly containing $\min$ and $\max$ operators. In practice, these constraints are easy to handle for Z3. As a result, the transformed program is guaranteed to use variables of lower precision with a minimal number of bits than the original program. The contributions introduced in this article are:

1) We improve the efficiency of POP by experimenting with several cost functions to our global system of constraints in order to optimize the solutions returned by the Z3 solver. We compare the performance of our tool for each of these functions on different programs coming from scientific computing, signal processing or the IoT domain.
2) For each tuned program, we measure the error between the exact results given by an execution in multiple precision (using MPFR [14]) and the results of optimized programs by POP. This error is compared to the user required accuracy.
3) We provide a detailed comparison of POP against the prior state-of-the-art tool Precimonious [8] in terms of analysis time, speed and the quality of the solution. Even though both tools use different techniques, we have adjusted the comparison criteria in order to make a closer comparison of the real behavior of these tools (see Section IV-B3). A detailed explanation of POP is given in Section II while the comparison versus Precimonious is presented in Section IV-B3.

The rest of this article is organized as follows. We present some preliminary definitions and an overview of our tool POP in Section II. Section III deals with the related work of the existing tools. Section IV is devoted to the analysis of the experimental results. Limitations and future work are discussed in Section V. Finally, Section VI concludes.

## II. Overview of POP Tool

This section starts with a brief background on the IEEE754 Standard for floating-point arithmetic and the definitions of the *ufp* and *ulp* functions. In addition, an overview of our tool POP is presented.

### A. Preliminary Definitions

The IEEE754 Standard [15] formalizes a binary floating-point number $x = s.m.\beta^{e-p+1}$, generally in base $\beta = 2$, as a triplet made of a sign $s$, a mantissa $m$ and an exponent $e$. The sign is $s \in \{-1, 1\}$, the mantissa is $m = d_0.d_1...d_{p-1}$, with the digits $0 \leq d_i < \beta$, $0 \leq i \leq p-1$, and $p$ is the precision (length of the mantissa). Finally, the exponent is $e \in [e_{min}, e_{max}]$.

The IEEE754 Standard specifies some particular values for $p$, $e_{min}$ and $e_{max}$ [16] and defines binary formats. Hence, the IEEE754 standard distinguishes between normalized and denormalized numbers. The normalization of a floating-point number ensuring $d_0 \neq 0$ guarantees the uniqueness of its representation. The IEEE754 standard defines also some special numbers (see [7] for details). Moreover, the IEEE754 Standard defines four rounding modes for elementary operations over floating-point numbers which are: towards $+\infty$, towards $-\infty$, towards zero and to 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 last significant bit*, respectively. In practice, our approach which describes the error propagation across the computations is based on these functions as already detailed in [7]. The definitions of these functions are given at equations (1) and (2) defined in [17] .

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

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

$$\mathit{ulp}(x) = \mathit{ufp}(x) - p + 1 \ . \qquad (2)$$

### B. POP Technical Approach

To better explain the forward and backward static analysis by constraint generation performed by POP, a straightforward C program used as motivating example is given in Figure 1. In this example, we suppose that all variables are in double precision before analysis (original program in the left hand side of Figure 1) and that a range determination is performed by dynamic analysis on these variables (we plan to use a static analyzer in the future). We generate at each node of the syntactic tree of our program a unique control point in order to determine easily the final accuracy, after the forward and backward analysis, as shown on the right hand side of Figure 1. Our program contains several annotations. First, for example, on the right hand side of Figure 1, the variables `a` and `x` are initialized respectively to the floating-point values 1.0 and 0.0 (in double precision). These values themselves are annotated with their control points $\ell_0$, $\ell_1$, ... thanks to the following annotations $\mathtt{a}^{\ell_1} = 1.0^{\ell_0}$ and $\mathtt{x}^{\ell_3} = 0.0^{\ell_2}$. In addition, we have the statement $\mathtt{require\_accuracy}(\mathtt{x}, 20)^{\ell_{28}}$ which informs the system that the user wants to get on variable `x` only 20 correct digits at its control point (we consider that a result has $n$ correct digits or significants if the relative error between the exact and approximated results is less than $2^{-n}$). As a consequence, the minimal precision needed for the inputs and intermediary results satisfying the user assertion is observed on the bottom center side of Figure 1. Since, in our example, 20 bits only are required for `x`, the result of the addition $\mathtt{x} + \mathtt{a}$ also needs 20 accurate bits only. By combining this information with the result of the forward analysis, it is then possible to lower the number of bits needed for one of the operands. The fundamental point of our technique is to generate simple constraints made of propositional logic formulae and of affine expressions among integers, even if the floating-point computations in the source code are non-linear [18]. Therefore, we assign to each control point $\ell$ two types of integer parameters: the *ufp* of the values (see Equation (1)) and three integer variables corresponding to the forward, the backward and the final accuracies, denoted $acc_F(\ell)$, $acc_B(\ell)$ and $acc(\ell)$ respectively, so that the inequality in Equation (3) is always 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).

$$0 \leq acc_B(\ell) \leq acc(\ell) \leq acc_F(\ell) \qquad (3)$$

The constraint system in this example, with control points $\ell_0$ to $\ell_{28}$, is made of constraints for the whole program including the while loop, assignments, additions and the counter. These numbers of variables and constraints are linear in the size of the program. Although the number of constraints generated is linear, it is too large to be written by hand yet in the same time it is easy to be checked by an SMT solver. In the sequel, we show at Equation (4) some constraints without taking into account the while loop and by taking into consideration the operations of assignments $\mathtt{a} := 1.0$ , $i = 1.0$ and $\mathtt{x} = 0.0$ and only one elementary operation $\mathtt{a} := \mathtt{a} + 1.0$ (see [17] for more details for the constraint generation of while loops and other commands.) The way to generate systematically these constraints from the source code of the program is explained in [17].

$$C = \left\{ \begin{array}{l} \mathtt{acc}_F(\ell_0) = 53,\ \mathtt{acc}_F(\ell_2) = 53,\ \mathtt{acc}_F(\ell_4) = 53, \\ \mathtt{acc}_F(\ell_{12}) = 53, \\ \mathtt{acc}_F(a^{\ell_1}) = \mathtt{acc}_F(\ell_0),\ \mathtt{acc}_B(a^{\ell_1}) = \mathtt{acc}_B(\ell_0), \\ \mathtt{acc}_F(i^{\ell_3}) = \mathtt{acc}_F(\ell_2),\ \mathtt{acc}_F(x^{\ell_5}) = \mathtt{acc}_B(\ell_4), \\ r^{\ell_{13}} = 1 - \max(0 - \mathtt{acc}_F(\ell_{11}), 0 - \mathtt{acc}_F(\ell_{12})), \\ (0 - \mathtt{acc}_F(\ell_{11})) = 0 - \mathtt{acc}_F(\ell_{12}) \Rightarrow i^{\ell_{13}} = 1, \\ \mathtt{acc}_F(\ell_{13}) = r^{\ell_{13}} - i^{\ell_{13}} = 53,\ \mathtt{acc}_B(\ell_{13}) = 20 \\ \mathtt{acc}_B(\ell_{11}) = 0 - (1 - \mathtt{acc}_B(\ell_{13})), \\ \mathtt{acc}_B(\ell_{12}) = 0 - (1 - \mathtt{acc}_B(\ell_{13})), \\ \mathtt{acc}_F(a^{\ell_{14}}) = \mathtt{acc}_F(\ell_{13}),\ \mathtt{acc}_F(x^{\ell_{20}}) = \mathtt{acc}_F(\ell_{19}), \\ \mathtt{acc}_B(a^{\ell_{14}}) = \mathtt{acc}_B(\ell_{13}),\ \mathtt{acc}_B(x^{\ell_{20}}) = \mathtt{acc}_B(\ell_{19}) \end{array} \right\}. \qquad (4)$$

Basically, the constraints of Equation (4) require that the forward accuracy at points $\ell_0$, $\ell_2$, $\ell_4$ and $\ell_{12}$ is 53 (IEEE754 double precision). Next, the forward accuracy of $a^{\ell_1}$ is 53 $(\mathtt{acc}_F(\ell_0))$ which is the precision of the value affected and it works the same for its backward accuracy. The constraints of the second line of our system works similarly. Also, we have the precision $r^{\ell_{13}}$ which is equal to the number of bits between the $\mathtt{ufp}(\ell_{13})$ (see Equation (1)) and the maximum precision of one of the two operands which are also computed as the difference of their `ufp` and their precisions respectively.

**Simple C program:**

```
a := 1.0;
i := 1.0;
x := 0.0;
while(i < 10.0){
    a := a + 1.0;
    x := x + a;
    i := i + 1.0;
} ;
require_accuracy(x,20);
```

**Program with labels:**

```
a^ℓ1 := 1.0^ℓ0;
i^ℓ3 := 1.0^ℓ2;
x^ℓ5 := 0.0^ℓ4;
while (i^ℓ7 <^ℓ9 10.0^ℓ8){
    a^ℓ14 := a^ℓ11 +^ℓ13 1.0^ℓ12;
    x^ℓ20 := x^ℓ16 +^ℓ19 a^ℓ18;
    i^ℓ25 := i^ℓ22 +^ℓ24 1.0^ℓ23;
}^ℓ26 ;
require_accuracy(x,20)^ℓ28;
```

**Program annotated with precisions:**

```
a#19 := 1.0#19;
i := 1.0;
x#20 := 0.0#20;
while (i <10.0){
    a#20 := a#19  +#20 1.0#20;
    x#20 := x#20  +#20 a#20 ;
    i := i + 1.0;
} ;
require_accuracy(x,20) #20;
```

Fig. 1: A simple example to show the nature of constraints generated by POP. Top left: source program. Top right: program annotated with labels. Bottom center : source program with inferred accuracies by POP.

The accuracy $\mathsf{acc}_F(\ell_{13})$ is equal to 53 which is the number of bits between $\mathsf{ufp}(1.0+1.0) = 1$ and the ufp $u$ of the error which is

$$u = \max\big(\max\big((\mathsf{ufp}(1.0) - \mathsf{acc}_F(\ell_{11}), \mathsf{ufp}(1.0) - \mathsf{acc}_F(\ell_{12})) + i, \mathsf{ufp}(\ell_{13}) - \sigma_+\big)\big) + i$$

We denote by $\sigma_+$ the precision of the operator $+$ and we have the function $i$ which takes 1 or 0 depending on the carry bit that can occur throughout the computations. To deal with intervals, variable $r^\ell$ and $i^\ell$, for any label $\ell$ in general, has to be replaced by two variables $[\underline{r}^\ell, \overline{r}^\ell]$ and $[\underline{i}^\ell, \overline{i}^\ell]$ respectively. We refer the reader to [7] for a detailed explanation of the forward and backward transfer functions for arithmetic expressions, statements, trigonometric functions and square root function.

POP has been extended in several ways since its first introduction in [7] and [18]. POP supports the subtraction and division arithmetic operations alongside the addition and multiplication already presented in [7]. In addition, the sine trigonometric function and the square root function are also implemented in POP with the appropriate forward and backward transfer functions expressed as a set of constraints. Although the square root is included in the IEEE754 Standard, it is not the case for the other elementary functions such as the natural logarithm, the exponential functions and the hyperbolic and trigonometric functions. Therefore each implementation of these functions has its own accuracy which we have to know to model the propagation of errors in our forward and backward analyses. In addition, POP now accepts arrays.

## III. Related work

In order to obtain the best floating-point formats as a function of the expected accuracy on the results, many efforts have been done. We classify these approaches into two categories: static methods that tune the precision to control the error and dynamic searching methods that take into consideration the performance of the tuned programs.

### A. Static Analysis Tools

There are various rigorous static analysis approaches that use interval and affine arithmetic [19] or Taylor series approximations to analyze stability and to provide rigorous bounds on rounding errors. In this context, Solovyev et al. [4] have proposed the FP-Taylor tool that implements a method to estimate round-off errors of floating-point computations called Symbolic Taylor Expansions. More recently, work in FP-Taylor was developed to ensure the mixed-precision tuning technique in a tool called FPTuner [6]. Actually, work in FPTuner provides *expression-level* precision guarantees by using Taylor series expansions. It formulates an error-constrained mixed integer optimization problem that attempts to find the lowest precision possible for a given error bound. The mathematical model obtains a rigorous error bound but is unable to deal with statements such as loops and conditionals, making it unsuitable for most programs. Darulova et al. [5] propose a technique to rewrite programs by adjusting the evaluation order of arithmetic expressions prior to tuning. However, the technique is limited to relatively small programs that can be verified statically.

### B. Dynamic Searching Applications

Other methods rely on dynamic analysis. Precimonious [8] is a dynamic automated search based tool. It aims to find the *1-minimal* configuration, i.e., a configuration where changing even a single variable from higher to lower precision would cause the configuration to cease to be valid. A valid configuration is defined as one in which the relative error in program output is within a given threshold and there is a performance improvement compared to the baseline version

of the program. However, it does not use any knowledge on the structure of the program to identify potential variables of interest. Also, we mention the Blame Analysis [12] which is another dynamic method that speeds up precision tuning by combining concrete and shadow program execution. It creates a blame set for each instruction, which comprises the variables whose precisions can be reduced to reach a given error threshold of the instruction under consideration. While this technique is considered as expansive, it does reduce the search space for Precimonious when these two techniques are used together. Pursuing with the idea of program transformation, the recent tool AMPT-GA [10] selects application-level data precisions to maximize performance while satisfying accuracy constraints. AMPT-GA combines static analysis for casting-aware performance modeling with dynamic analysis for modeling and enforcing precision constraints.

Recently, a new solution called HiFPTuner [9] soothes this problem to some extent by taking a white box approach. It analyzes the source code and its runtime behaviors to identify dependencies among floating-point variables, and to provide a customized hierarchical search for each program under analysis to limit the search space of Precimonious. In another work, Lam et al. [11] instrument binary code in a tool called CRAFT aiming to modify their precision without modifying the source code. Also, their tool is based on a dynamic search method in order to identify in which parts of code the precision should be modified. The major drawback is that it does not guarantee a reduction in the execution time. We mention, also, some models in [20] and [21] that perform *Shadow Value Analysis* by inserting low level instructions at runtime to simulate floating point instructions at another precision. However, this is only a tool for analysis with respect to error, not for finding faster configurations.

By and large, existing methods for precision tuning follow a try and fail approach, reducing the precision of arbitrary chosen variables and executing [8] or analyzing statically [4] the program to see the new accuracy. Conversely, our static analysis based method relies on a modeling of the propagation of the errors throughout the code implemented by a system of constraints. The solution to this system gives the optimized precision of all the variables without execution of the program. For instance, the differences between POP and Precimonious [8] are displayed in Table I.

## IV. Experimental Evaluation

After presenting the main objective of POP, we want to validate its performance through the following experimental evaluations:

- We evaluate the quality of solutions returned by Z3 with different cost functions given as additional constraints added to our global system of constraints.
- We measure the absolute error between the exact results obtained by running a MPFR version of the programs with a high precision [14] and the results of tuned programs by POP. We show that this error is less than the theoretical error required by POP user. This experimentally validates the correctness of the tool, in complement to the proofs given in [7].
- We compare POP against a dynamic search algorithm, as implemented in the tool Precimonious [8].

We ran our experiments on an Intel Core i5-8350U CPU cadenced at 1.7GHz on a Linux machine with 8 GB RAM. Concerning Precimonious, we used the version provided at `https://github.com/HGuo15/vagrant_precimonious`.

### A. Experimental Setup

Figure 2 shows the results of mixed-precision tuning obtained by POP on several programs. The **low pass filter** program and the **derivative** function are parts of the pedometer program that counts the number of footsteps whose algorithm contains several steps [22]. The last two programs come from the IoT field [23]. The **arclength** program was first introduced in [24] and the **simpson** program which corresponds to an implementation of the widely used Simpson's rule for integration in numerical analysis [25]. The results are perceived by experimenting two cost functions to our global system of constraints in order to optimize the solutions returned by the Z3 solver.

The experiment shown in Figure 3 examines the difference between the exact results computed using the MPFR library [14] in high precision (300 bits) and the results returned by our tool and compares this difference with the worst error required by the user for different accuracies: 12, 17, 23, 27, 30 and 38 correct digits. Each program in this experiment corresponds to two curves and we can see in Figure 3 that the actual error measured is always less than the theoretical user error defined as $\beta^{-x+1}$ where $\beta = 2$ and $x$ denotes the user accuracy requirement.

For the comparison against the state-of-art precision tuner Precimonious, we evaluate POP on two numerical programs, already introduced above, used as a benchmark for precision tuning in prior work [1], [8] and coming from the GNU Scientific Library (GSL): **simpson** and **arclength**. In addition, we evaluate Precimonious on three programs used as benchmarks for POP. The **rotation matrix-vector multiplication** to rotate a vector around the $z$ axis by angle $\theta$ [7]. Next, the program **accelerometer** which is an implementation of an application that measures an inclination angle with an accelerometer [18] and finally the **pedometer** program [23].

Table II shows the number of variables optimized by both tools after analysis for each program while in Table III we present the number of variables tuned to FP8 (mini-float precision), FP16 (IEEE754 half precision), FP32 (IEEE754 simple precision), FP64 (IEEE double precision) and FP128 (IEEE754 long-double precision) for each error on the results ($10^{-4}$, $10^{-6}$, $10^{-8}$ and $10^{-10}$). The error threshold represents the number of accuracy digits (for example, the result is required to be correct up to 10 digits for an error threshold of $10^{-10}$). Let us note that the error thresholds are expressed in base 2 in POP and in base 10 in Precimonious. In this section, for the relevance of comparisons, all the thresholds are expressed in base 10. In practice, POP will use the base 2 threshold immediately lower than the required base 10 threshold.

### B. Experimental Results

*1) Experiment 1: Assigned Variables vs. All Control Points Cost Functions:* In this experiment, we aim at evaluating two kinds of cost functions and compare the mixed-precision

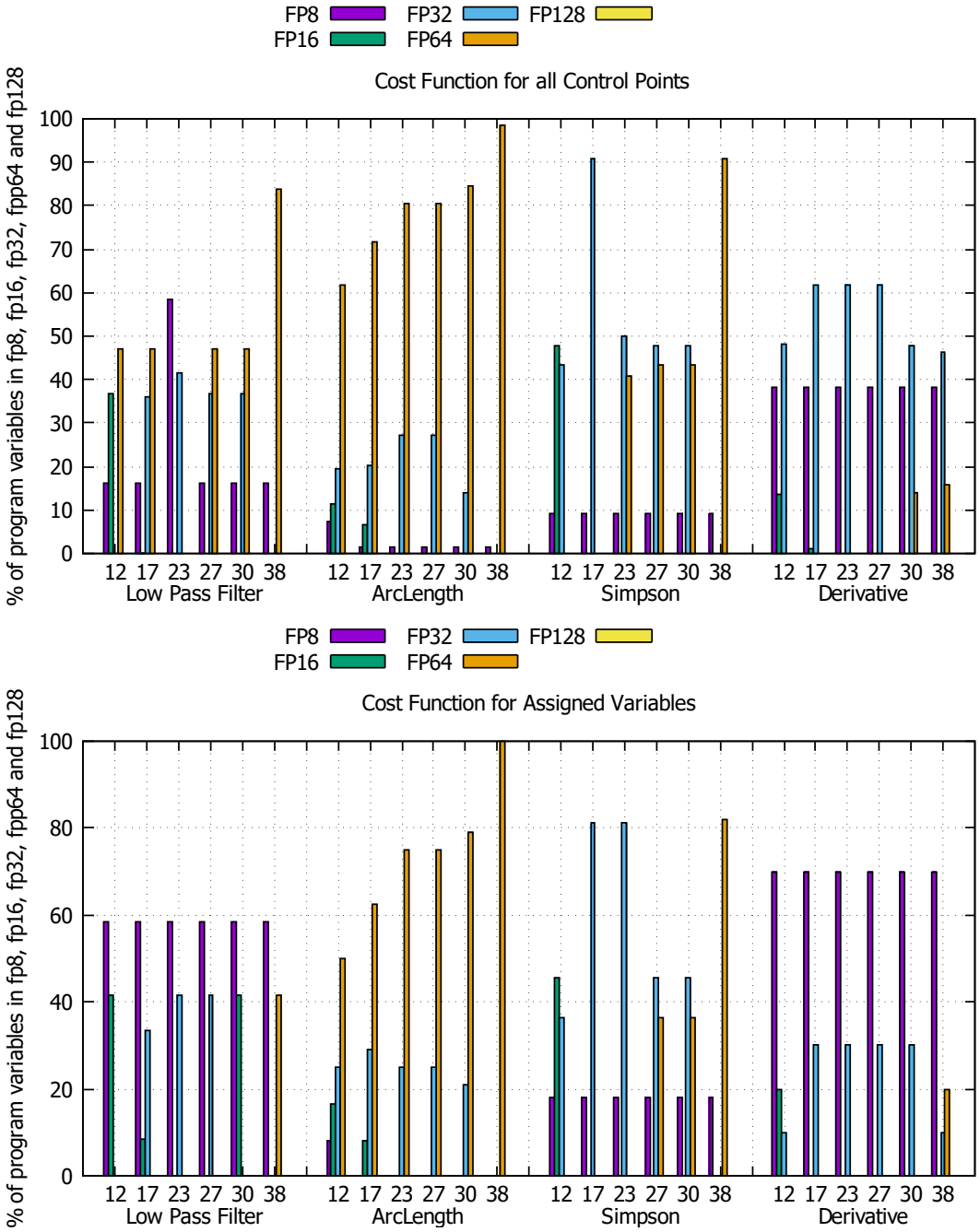


Fig. 2: Mixed-precision tuning results in FP8, FP16, FP32, FP64 and FP128 for the all control points and variables accuracies cost function (top) and for the optimized cost function considering only the variables assigned (bottom).

obtained in the tuned programs for each of these functions. Recall that POP generates a set of constraints made of propositional logic formulas and affine expressions among integers already presented in [7] and calls the Z3 SMT solver in order to obtain a solution. However, the solutions returned are not optimal due to the fact that Z3 is a solver and not an optimizer. To surpass this limitation, we add to our global system of constraints an additional constraint related to a cost function $\phi$ (we use the same definition as in [17]). Then we ask Z3 to find a solution of a given weight. This operation is done repeatedly in a dichotomic search.

Let $Id$ and $Lab$ denote respectively the sets of identifiers and labels of the program and let $acc(x^\ell)$ be a variable of the constraint system corresponding to the accuracy of a variable $x \in Id$ at a control point $\ell \in Lab$ and let $acc(\ell)$ be the accuracy of the operation done at control point $\ell$. The aim of the first cost function $\phi(c)$, that we want to optimize for a given program $c$ by dichotomy, is to compute the sum of the accuracies $acc(x^\ell)$ of all the variables plus the accuracies $acc(\ell)$ at each control point $\ell$ of the arithmetic expressions as it is shown in Equation (5).

$$\phi(c) = \sum_{x \in Id, \ell \in Lab} acc(x^\ell) + \sum_{\ell \in Lab} acc(\ell) \qquad (5)$$

The purpose of the second cost function $\phi'(c)$ in Equation (6) is to only consider the sum of accuracies for the variables

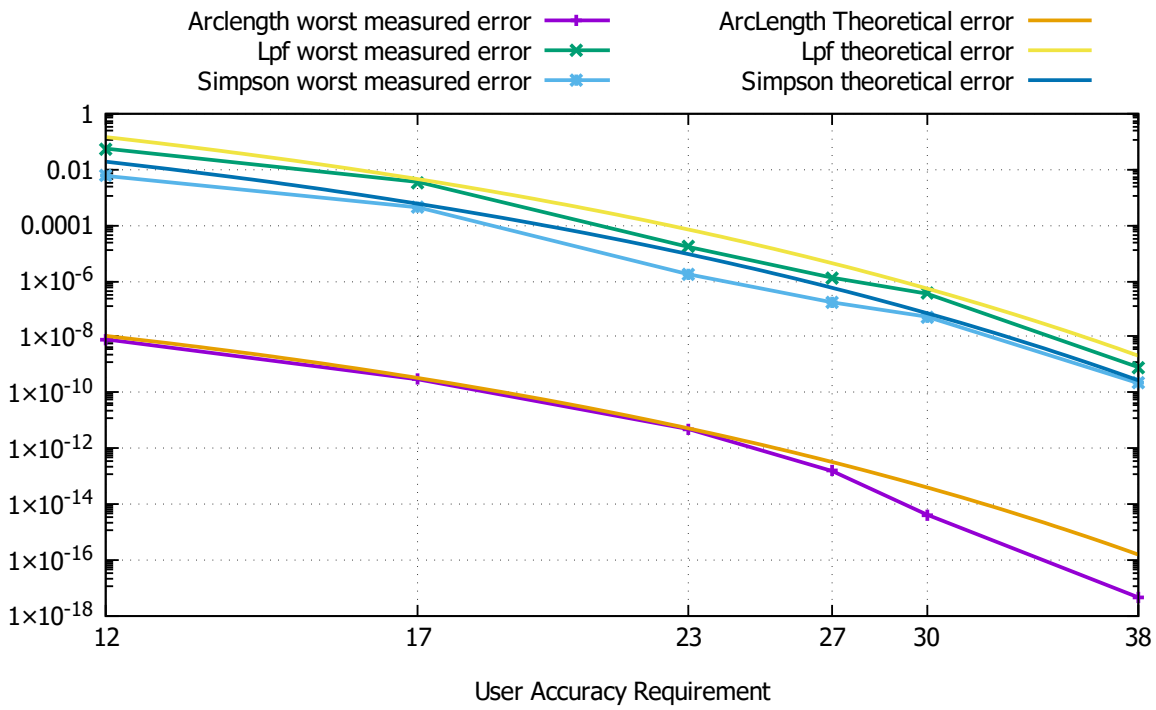


Fig. 3: Measured error between the exact results with multiple precision and the results obtained with POP for the **simpson**, **arclength** and **low pass filter** programs.

assigned in the program and to no longer count accuracies for the different control points as indicated in Equation (5).

$$\phi'(c) = \sum_{x \in Id, \ell \in Lab} acc(x^{\ell}) \tag{6}$$

Let us remark that this second cost function, which is a strict relaxation of the first one, corresponds to what Precimonious actually optimizes. It is then fairer to use it in our comparisons. Next, our tool searches the smallest integer $P = \phi(c)$ or $P = \phi'(c)$ such that our system of constraints admits a solution. Consequently, we start the binary search with $P \in [0{,}112 \times n]$ where all the values are in long double precision and where $n$ is the number of terms in Equation (5) or Equation (6) (depending on the function we add in our system of constraints). When 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.

Note that the cost functions that we use become more complex when dealing with arrays and then we consider that all elements have the same precision as in almost the programming languages. In order to compute the total number of bits, we have to multiply the precision by the number of elements. In addition, we have to do this process only once for each array instead of several times for each use of arrays. This implied to modify significantly the tool compared to what we had implemented for the other simple variables.

The main results of running POP with these two cost functions $\phi$ and $\phi'$ are presented in Figure 2. In addition to the **simpson** and **arclength** programs, we take the code of two functions implementing respectively a **low pass filter** and the **derivative** function. Recall that these functions are taken from the pedometer code.

Of the six accuracy requirements given to POP (from 12 to 38 bits of accuracy) for the four programs, POP succeeded in turning off variables into FP8, FP16, FP32, FP64 and FP128 as shown in the top and the bottom of Figure 2 for the two cost functions. The top of Figure 2 shows that for the **low pass filter** program, 16% of variables are tuned to FP8, 37% are tuned to FP16 and nearly 50% of variables are turned off in double precision (FP64) for a user accuracy of 12. Although, for the same program, the majority of variables are transformed to FP64 for an accuracy of 38. For the **arclength** program and for the different user accuracies, a large amount of variables are tuned to FP64 double precision starting with 62% for an accuracy of 12 until 98% for an accuracy of 38 bits. For the **simpson** program, we observe improvements but in less proportions than for the other examples. For the program implementing the **derivative** function of the pedometer code, we observe that the largest part of variables are transformed to FP32 simple precision and FP8 mini-float precision compared with the few amount of variables tuned to double precision especially for accuracies of 30 and 38 bits.

The main results of running POP with the new cost function $\phi'$ are illustrated in the bottom side of Figure 2. By comparing the results with the old cost function, we perceive the difference of the mixed-precision tuning results returned by POP. For the **low pass filter** program, we have more variables turned into FP8 mini-float precision (nearly 59% for almost every user accuracy requirements) compared to the number of variables in double precision obtained with the cost function illustrated in the top of Figure 2. The

TABLE I: Differences between POP and Precimonious.

| Property | POP | Precimonious |
|---|---|---|
| Kind of analysis | Forward & backward static analysis | Dynamic analysis by delta-debugging search |
| Output | Optimized formats given by Z3 (number of bits) | Type configurations rely on inputs tested **only** |
| Mixed-precision | FP8, FP16, FP32, FP64, FPxx | FP32 and FP64 |
| Accuracy requirement | number of significant bits of the result | Error threshold ($10^{-4}$, $10^{-6}$, $10^{-8}$, ...) |
| Accepted language | Arrays, conditions, loops, no function yet | C program input |

TABLE II: Number of optimized variables by both tools for each program.

| Program | #Variables optimized by POP | #Var. optim. by Precimonious |
|---|---|---|
| Arclength | 25 | 9 |
| Simpsons | 10 | 10 |
| Rotation-matrix | 27 | 27 |
| Accelerometer | 19 | 0 |
| Pedometer | 31 | 10 |

majority of variables in program **arclength** are transformed into double precision, going up to $100\%$ for an accuracy of $38$. Finally, for the derivative program, the percentage of variables in FP8 mini-float precision is greater than in FP32 simple-float precision reaching $70\%$ for all user accuracies.

*2) Experiment 2: Relative Error Measured between The Exact Results with Multiple Precision and the Results Obtained with POP:* In this experiment, we seek to generate an MPFR [14] code to create a program that gives an exact result (we assume that the original program is computed with a precision of $300$ bits). After the same code is generated with the optimized precisions returned by our tool. The goal of this experiment is to measure the difference between the two programs and to plot the curve of the difference in function of the theoretical error which is the worst accepted error required by the user, for example this error is equal to $2^{-22}$ for an accuracy of $23$.

The curves of Figure 3 validate that POP satisfies the user defined error constraints for the input programs: **arclength**, **simpson** and **low pass filter**. We can see that the actual error, for each program, is always less than the theoretical error given by the user. For a user accuracy requirements of $17$ and $23$ bits, the curves of the **arclength** program can intersect but it always remains below the worst error tolerated by the user ($2^{-16}$ and $2^{-22}$ respectively for these requirements). Let we remark that the measured errors are very close to the theoretical error. This is a desirable property: instead if the measured error where far smaller than the theoretical error this would probably mean that we use too many bits and consequently that the tuning would be suboptimal.

*3) Comparison between POP and Precimonious :* The main goal of this experimental evaluation is to compare our tool versus the dynamic-search based tool Precimonious. The number of lines of code (LOC) for each of the five programs differs from one tool to another because of the different instructions linked to the error constraints that each tool adds. In Table II, we show the number of variables optimized by both tools for each input program. Since the tools implement two different techniques, we have adjusted some criteria including the number of variables that we optimize so that the comparison is as close as possible to the real behaviors of the two tools. This is done as follows:

- POP optimizes more variables than Precimonious. Consequently, for the sake of comparison, in the following we only consider the variables optimized by Precimonious to estimate the quality of the optimization. Let us remark that this disadvantages our tool, POP.
- For POP, the initial precision of the input programs is a parameter. This parameter was set to double precision in Section II. To fit with Precimonious features in the comparisons, in the sequel, POP initial precision is set to long double.
- Precimonious creates a search space for all variables which precisions needs to be tuned [8] in float (FP32), double (FP64) or long-double (FP128) precisions whereas our tool takes into account the variables that arise in the computation of the resulted variable on which the user requires a precision constraint and tunes their precisions into mini-float precision (FP8) and half precision (FP16) alongside the float, double and long-double precisions. We do know that FP8 mini-float precision is rarely used but our tool is able to find this format without additional cost nor to increase the complexity which depends on the number of formats. In contrast, Precimonious is not able to have this format without additional cost.

Table III shows the results of the mixed-precision tuning for the five programs by both tools and for different error thresholds. We measure the running analysis time in seconds taken by the tools to find the new precisions inferred in the optimized programs. In addition, we estimate the time by Z3 in POP to solve the constraints. After the tools analysis, we compute the number of variables tuned into FP8, FP16, FP32, FP64 and those remained in FP128 and we present the total number of bits in the optimized programs. We denote by "S" the number of function prototypes which can be tuned to a lower precision and the symbol "-" denotes that a tool does not find any solution (configuration of types) that satisfies the user accuracy constraint. We show in bold the number of variables turned to FP8 and FP16 by POP where Precimonious is only capable to tune the variables precision to FP32 and FP64 only.

For the running time measured for both tools, we can observe (in Table III) that the analysis are running faster with our tool for the different benchmarks and error thresholds. However, our tool takes longer time to find the optimized precisions for the pedometer program compared with Precimonious. By looking at the Z3 time spent to resolve the constraints, we can deduce that our tool consumes almost the entire time of analysis making calls to the Z3 solver. Although it remains fast and does not exceed a few minutes (only $4$ minutes are spent to solve the constraints of the pedometer code for an error threshold of $10^{-6}$), we plan

TABLE III: Precision tuning results for POP and Precimonious for different error thresholds $10^{-10}$, $10^{-8}$, $10^{-6}$ and $10^{-4}$.

| Program | Tool | Threshold $10^{-4}$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Running Time(s) | Z3 Time(s) | #Bits Optimized | #FP8 | #FP16 | #FP32 | #FP64 | #FP128 | S |
| **arclength** | POP | 2.867 | 2.834 | 464 | 0 | **3** | 1 | 4 | 1 | 4 |
| | Precimonious | 146.4 | - | 448 | 0 | 0 | 2 | 4 | 1 | 2 |
| **simpson** | POP | 0.608 | 0.594 | 160 | **2** | **7** | 1 | 0 | 0 | 1 |
| | Precimonious | 208.092 | - | 384 | 0 | 0 | 4 | 2 | 1 | 3 |
| **rotation matrix-vector** | POP | 0.882 | 0.869 | 408 | **13** | **9** | 5 | 0 | 0 | 0 |
| | Precimonious | 9.536 | - | 1056 | 0 | 0 | 25 | 0 | 2 | 0 |
| **accelermeter** | POP | 2.819 | 2.801 | 280 | **3** | **16** | 0 | 0 | 0 | 0 |
| | Precimonious | - | - | - | - | - | - | - | - | - |
| **pedometer** | POP | 185.002 | 184.865 | 272 | 0 | **5** | 2 | 2 | 0 | 1 |
| | Precimonious | 21.100 | - | 672 | 0 | 0 | 5 | 0 | 4 | 0 |

| Program | Tool | Threshold $10^{-6}$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Running Time(s) | Z3 Time(s) | #Bits Optimized | #FP8 | #FP16 | #FP32 | #FP64 | #FP128 | S |
| **arclength** | POP | 2.300 | 2.278 | 528 | 0 | **1** | 2 | 5 | 1 | 3 |
| | Precimonious | 156.0 | - | 512 | 0 | 0 | 2 | 5 | 1 | 1 |
| **simpson** | POP | 0.499 | 0.492 | 272 | **2** | 0 | 8 | 0 | 0 | 1 |
| | Precimonious | 213.672 | - | 384 | 0 | 0 | 4 | 2 | 1 | 3 |
| **rotation matrix-vector** | POP | 0.920 | 0.915 | 688 | **6** | **8** | 10 | 3 | 0 | 0 |
| | Precimonious | 12.201 | - | 864 | 0 | 0 | 27 | 0 | 0 | 0 |
| **accelermeter** | POP | 2.756 | 2.736 | 560 | 0 | **3** | 16 | 0 | 0 | 0 |
| | Precimonious | - | - | - | - | - | - | - | - | - |
| **pedometer** | POP | 215.528 | 215.387 | 352 | 0 | 0 | 7 | 2 | 0 | 1 |
| | Precimonious | 22.275 | - | 672 | 0 | 0 | 5 | 0 | 4 | 1 |

| Program | Tool | Threshold $10^{-8}$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Running Time(s) | Z3 Time(s) | #Bits Optimized | #FP8 | #FP16 | #FP32 | #FP64 | #FP128 | S |
| **arclength** | POP | 2.626 | 2.617 | 576 | 0 | 0 | 2 | 6 | 1 | 0 |
| | Precimonious | 145.8 | - | 448 | 0 | 0 | 2 | 4 | 1 | 2 |
| **simpson** | POP | 0.594 | 0.541 | 272 | 0 | **2** | 0 | 8 | 0 | 1 |
| | Precimonious | 207.558 | - | 384 | 0 | 0 | 4 | 2 | 1 | 3 |
| **rotation matrix-vector** | POP | 0.913 | 0.901 | 912 | **2** | **4** | 16 | 5 | 0 | 0 |
| | Precimonious | 10.697 | - | 992 | 0 | 0 | 25 | 1 | 1 | 0 |
| **accelermeter** | POP | 2.917 | 2.897 | 608 | 0 | 0 | 19 | 0 | 0 | 0 |
| | Precimonious | - | - | - | - | - | - | - | - | - |
| **pedometer** | POP | 102.885 | 102.751 | 352 | 0 | 0 | 7 | 2 | 0 | 1 |
| | Precimonious | - | - | - | - | - | - | - | - | - |

| Program | Tool | Threshold $10^{-10}$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Running Time(s) | Z3 Time(s) | #Bits Optimized | #FP8 | #FP16 | #FP32 | #FP64 | #FP128 | S |
| **arclength** | POP | 2.442 | 2.435 | 608 | 0 | 0 | 1 | 7 | 1 | 0 |
| | Precimonious | 215.011 | - | 448 | 0 | 0 | 2 | 4 | 1 | 2 |
| **simpson** | POP | 0.501 | 0.487 | 528 | **2** | 0 | 0 | 8 | 0 | 1 |
| | Precimonius | 200.388 | - | 384 | 0 | 0 | 4 | 2 | 1 | 3 |
| **rotation matrix-vector** | POP | 0.927 | 0.923 | 1184 | 0 | **4** | 11 | 12 | 0 | 0 |
| | Precimonious | 7.409 | - | 992 | 0 | 0 | 25 | 1 | 1 | 0 |
| **accelermeter** | POP | 2.761 | 2.735 | 992 | 0 | 0 | 7 | 12 | 0 | 0 |
| | Precimonious | - | - | - | - | - | - | - | - | - |
| **pedometer** | POP | 177.348 | 177.161 | 576 | 0 | 0 | 0 | 9 | 0 | 1 |
| | Precimonious | - | - | - | - | - | - | - | - | - |

to replace the use of Z3 by the technique of policy iteration in the future [26].

Compared to Precimonious, our tool succeeds to tune two more programs. Next, let us remark that by entering the **accelerometer** program as input to Precimonious, no valid configuration was found for this example with error thresholds $10^{-4}$ to $10^{-10}$. In addition, Precimonious failed to tune the pedometer program for any error threshold lesser than $10^{-6}$. Moreover, Table III lists the final optimized precisions obtained in the form of FP8 mini-float precision, FP16 half precision, FP32 simple precision, FP64 double precision and FP128 long-double precision and counts the number of bits in "#Bits Optimized" of the tuned programs. Recall that "S" is the number of function calls to switch to lower precision (same parameter as in [1]). For example, a function whose prototype is tuned from `double -> double` to `float -> float` is counted as one switch in "S".

As we can observe from Table III, our tool POP is capable to optimize variables into FP8 and FP16 (see bold numbers under POP "#FP8" and "#FP16" in Table III) in addition to FP32, FP64 and FP128 which are the only three configurations inputs associated to each variable of the search space of Precimonious. By way of illustration, 13 variables of a total of 27 are tuned to FP8, 9 variables are optimized to FP16 and the rest are tuned to FP32 for the **rotation matrix-vector**. The "#Bits Optimized" is obtained by the sum of multiplying each variable by its precision. We remark that POP succeeded in tuning 4 of 5 programs with less number of bits optimized than precimonious (except the **arclength** program). At the same time, Precimonious do well for an error threshold of $10^{-10}$ for the **arclength**, **simpsons** and **rotation matrix-vector** programs (the tuned **arclength** program by Precimonious uses less bits than our tool for the four given error thresholds).

## V. Limitations and Future Work

As we have mentioned in Section II, a range determination is performed before the analysis is done. Obviously, the precision of this range analysis impacts the precision of the floating-point format determination and the computation of sharp ranges, by means of relational domains [27], [28] improves the quality of the final result. In future work, we would like to use a static analyzer in order to infer safe ranges on our variables. Besides, the implementation of functions

inside POP is desired at this stage especially as we plan to strengthen the comparisons with other existing tools and to evaluate their benchmarks as input of our tool.

Even if the execution time of our analysis are not prohibitive, we believe that using the policy iteration method [26] as a replacement for the non-optimizing solver (Z3) coupled to a binary search can provide an interesting improvement of the performance. Actually, we aim at applying the policy iteration method to improve the accuracy. For this reason, we need to show that finding the minimum precision with the generated constraints is equivalent to search the smallest fixed-point of a monotone function in such a way that the selection property on which policy iteration relies is satisfied. 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.

As we have experimented our tool in the IoT field, we have not yet applied the optimized programs obtained by POP on a real IoT device to measure physically the gain in memory and energy which is considered as a prior future work. Moreover, we manage to evaluate our tool on more complex programs coming from this application domain. Finally, our approach is also easily generalizable to the case of the fixed-point arithmetic for which it is mandatory to determine the formats of the data, for example for FPGA implementations [29].

## VI. Conclusion

In this article, we have evaluated the efficiency of our tool POP following several criteria. First, we have tested two different cost functions to POP global system of constraints in order to optimize the solutions returned by Z3 solver have shown that this can influence the total number of bits optimized in the tuned programs by obtaining different optimized precisions when using a function instead of the other. Second, we have measured the error between the exact results given by an execution in multiple precision and the results of optimized programs by our tool and found that the measured error curve is always below the theoretical error required by the user. Also, we have evaluated the performance of our automated tuning tool POP by presenting a comparison against the prior state-of-the-art tool Precimonious in terms of analysis time, speed and the quality of the solution and where our tool was faster in the analysis time for the majority of the programs tested. In addition, we deduced that our tool returns better mixed-precision results for different user accuracy requirements where the variables are tuned into FP8, Fp16, FP32, FP64 and FP128 precisions. The results achieved are promising for our tool so the next step is to strengthen the evaluation in several manners in future work by experimenting it with larger applications in various domains such as IoT and HPC.

## References


[1] H. Guo and C. Rubio-González, "Exploiting community structure for floating-point precision tuning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, 2018.

[2] A. D. Franco, H. Guo, and C. Rubio-González, "A comprehensive study of real-world numerical bug characteristics," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, G. Rosu, M. D. Penta, and T. N. Nguyen, Eds. IEEE Computer Society, 2017, pp. 509–519.

[3] A. Nötzli and F. Brown, "Lifejacket: Verifying precise floating-point optimizations in LLVM," *CoRR*, 2016.

[4] A. Solovyev, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, "Rigorous estimation of floating-point round-off errors with symbolic taylor expansions," in *FM 2015: Formal Methods*, N. Bjørner and F. de Boer, Eds. Springer International Publishing, 2015.

[5] E. Darulova, E. Horn, and S. Sharma, "Sound mixed-precision optimization with rewriting," 2017.

[6] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, "Rigorous floating-point mixed-precision tuning," *SIGPLAN Not.*, vol. 52, no. 1, p. 300315, 2017.

[7] D. Ben Khalifa, M. Martel, and A. Adjé, "POP: A tuning assistant for mixed-precision floating-point computations," in *Formal Techniques for Safety-Critical Systems - 7th International Workshop, FTSCS 2019, Shenzhen, China, November 9, 2019, Revised Selected Papers*.

[8] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: tuning assistant for floating-point precision," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*.

[9] H. Guo and C. Rubio-González, "Exploiting community structure for floating-point precision tuning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. Association for Computing Machinery, 2018.

[10] P. V. Kotipalli, R. P. Singh, P. Wood, I. Laguna, and S. Bagchi, "Ampt-ga: automatic mixed precision floating point tuning for gpu applications," in *ICS '19*, 2019.

[11] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre, "Automatically adapting programs for mixed-precision floating-point computation," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. ACM, 2013.

[12] C. Rubio-Gonzlez, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, D. H. Bailey, and D. Hough, "Floating-point precision tuning using blame analysis," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016.

[13] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, 2011.

[14] L. Fousse, G. Hanrot, V. Lefvre, P. Pélissier, and P. Zimmermann, "Mpfr: A multiple-precision binary floating-point library with correct rounding." *ACM Trans. Math. Softw.*, vol. 33, 2007.

[15] *IEEE Standard for Binary Floating-point Arithmetic*, ANSI/IEEE, 2008.

[16] N. Damouche and M. Martel, "Salsa: An automatic tool to improve the numerical accuracy of programs," in *Automated Formal Methods*, ser. Kalpa Publications in Computing, N. Shankar and B. Dutertre, Eds. EasyChair, 2018.

[17] M. Martel, "Floating-point format inference in mixed-precision," in *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, 2017, pp. 230–246.

[18] D. Ben Khalifa and M. Martel, "Precision tuning and internet of things," in *International Conference on Internet of Things, Embedded Systems and Communications, IINTEC 2019, Tunis, Tunisia, December 20-22, 2019*. IEEE, 2019, pp. 80–85.

[19] L. H. de Figueiredo and J. Stolfi, "Affine arithmetic: Concepts and applications," *Numerical Algorithms*, vol. 37, no. 1-4, pp. 147–158, 2004.

[20] M. O. Lam and J. K. Hollingsworth, "Fine-grained floating-point precision analysis," *The International Journal of High Performance Computing Applications*, 2018.

[21] M. O. Lam and B. L. Rountree, "Floating-point shadow value analysis," in *Proceedings of the 5th Workshop on Extreme-Scale Programming Tools*. IEEE Press, 2016.

[22] D. Ben Khalifa and M. Martel, "Precision tuning of an accelerometer-based pedometer algorithm for iot devices," in *International Conference on Internet of Things and Intelligence System, IoTaiS 2020*. IEEE, 2020, pp. 113–119.

[23] D. Morris, T. Saponas, A. Guillory, and I. Kelner, "Recofit: Using a wearable sensor to find, recognize, and count repetitive exercises," *Conference on Human Factors in Computing Systems - Proceedings*, 2014.

[24] D. Bailey, "Resolving numerical anomalies in scientific computation," 03 2008.
[25] W. M. McKeeman, "Algorithm 145: Adaptive numerical integration by simpson's rule," *Commun. ACM*, vol. 5, no. 12, p. 604, 1962.
[26] S. Gaubert, E. Goubault, A. Taly, and S. Zennou, "Static analysis by policy iteration on relational domains," in *Programming Languages and Systems*, R. De Nicola, Ed. Springer Berlin Heidelberg, 2007.
[27] A. Miné, "The octagon abstract domain," *High. Order Symb. Comput.*, vol. 19, no. 1, pp. 31–100, 2006.
[28] E. Goubault and S. Putot, "A zonotopic framework for functional abstractions," *Formal Methods Syst. Des.*, vol. 47, no. 3, pp. 302–360, 2015.
[29] X. Gao, S. Bayliss, and G. Constantinides, "Soap: Structural optimization of arithmetic expressions for high-level synthesis," 2013.