Precision Tuning and Internet of Things

Dorra BEN KHALIFA
Laboratory of Mathematics and Physics LAMPS
University of Perpignan Via Domitia
52 Av. P. Alduy, 66100, Perpignan, France
dorra.ben-khalifa@univ-perp.fr

Matthieu MARTEL
Laboratory of Mathematics and Physics LAMPS
University of Perpignan Via Domitia
52 Av. P. Alduy, 66100, Perpignan, France
matthieu.martel@univ-perp.fr

***Abstract*—Numerical precision / memory / energy trade-off is a more and more important concern in the design of future computing systems at all scales. In practice, programmers tend to use the highest precision available in hardware which is the IEEE754 double precision [1] on current processors. This approach can be too costly in terms of computing time, memory transfer and energy consumption. To overcome this difficulty, we present, in this article, a floating-point precision tuning tool called, POP: Precision OPtimizer, which integrates a static program analysis to determine the minimal precision on the inputs and the intermediary results of a program performing floating-point computations in order to ensure a desired accuracy on the outputs. Our approach combines a forward and a backward static analysis done by abstract interpretation. Next, our analysis is expressed as a set of linear constraints easily checked by an SMT solver. Precision tuning already has applications in many domains and, in this article, we show its usefulness in a new, unexplored domain, namely for Internet of Things applications. We show how it can be important in terms of memory and energy concerns. A prototype implementing our analysis has been realized and experimental results are presented on numerical computations performed binary an accelerometer comparable to what we can find in smartphones to convert brute sensor data into movements.**

***Index Terms*—Precision tuning, Floating-point arithmetic, Internet of Things, Applications, Static analysis**

I. Introduction

The use of floating-point arithmetic, whose specification is given by the IEEE754 Standard [1], to carry out real arithmetic on computers is almost standard practice [2]. Nowadays, with the wide availability of processors with hardware floating-point units, e.g. for smartphones or other internet of things (IoT) devices, many applications rely heavily on floating-point operations. In practice, these applications such as the critical command systems for automotive, aeronautic, space, etc., have stringent correctness requirements and their failure have catastrophic consequences on human life [3]. To minimize the chance of problems, developers are likely to use the highest precision available in hardware throughout the whole program without an extensive background in numerical accuracy. Despite the fact that the results will be more accurate, this increases significantly the application runtime, bandwidth capacity and the memory and energy consumption of the system. Furthermore, we will concentrate, in this work, on an important difference relating the terms precision and accuracy that are often confused, even though they have significantly different meanings. Here, we call *precision* a property of a number format and refers to the amount of information used to represent a number. Better or higher precision means more numbers can be represented, and also means a better resolution. Otherwise, the term *accuracy* denotes how close a floating-point computation comes to the real value [4]: a bound on the absolute error $|x - \widehat{x}|$ between the represented $\widehat{x}$ value and the exact value x that we would have in the exact arithmetic. This present work address the problem of determining the minimal precision on the inputs and intermediary results of a program performing a floating-point computations that guarantees a desired accuracy of the outputs values. Consequently, it is possible to save memory, reduce CPU usage and use less bandwidth for communications. In addition, our precision tuning technique could be easily transposed to the case of fixed-point arithmetic for the case of IoT devices which do not have floating-point units. We believe it possible and promising to adapt the precision tuning technique to applications of Internet of Things because the type of problem of energy consumption and memory are widespread in this area. In fact, The devices of Internet of things are powered by an independent power supply like battery and energy harvester, which provide limited energy [5]. Consequently, batteries require changing and replacement due to their short lifetime. Also, the IoT devices memory is used to store data and performance tasks, therefore, consistent memory access occurs during the operation of the IoT device. Thus, the energy savings associated with memory access reduce the average power consumption of IoT devices. Nevertheless, the POP tool applies the mixed-precision on the floating-point programs formats. We denote by mixed-precision computing the approach of combining different precisions for different floating-point variables (contrarily to the uniform precision). The common objective of existing approaches, which differ from each other in their way of determining accuracy either by dynamic or static methods [6] [7] [8] [9], is to compute approximations of the errors on the outputs of a program depending on the initial errors on the inputs and the roundoff of the arithmetic operations performed during the execution [4]. POP takes as input a program annotated by the user accuracy expectation and it implements a static forward and backward error analysis which are two popular paradigms of error analysis, done by abstract interpretation [10]. The forward analysis is classical. It examines how errors are magnified by each operation aiming to determine the accuracy on the results. Next, the backward analysis takes in consideration the user requirement denoting the final accuracy wanted on some control points of the outputs and the results of the forward analysis. Obviously, it is a complementary approach that starts with the computed answer to determine the exact floating-point input that would produce it in order to satisfy the desired accuracy. As could be expected, the forward and the backward analysis can be handled iteratively to refine the results until a fixed-point is reached. Then, the forward and backward transfer functions are expressed as a set of linear constraints made of propositional logic formulas and relations between integer elements only. After, these constraints will be solved by an SMT solver (we use `z3` in practice [11]). In previous work, we have described the improvement of our approach compared to [4]. The contribution revolved around reexamining a function ι considered as the carry bit that can occur throughout floating-point computations. A too conservative static analysis would consider that a carry bit can be propagated at each operation which corresponds to $\iota = 1$ which is very costly if we perform several computations at a time. That's why, we proved that it is crucial to use the most precise function ι.

This work focuses on experimenting the tool on examples issued from different fields specially the Internet of Things and finding a trade-off between precision and energy.

The reminder of the article is organized as follows. Section 2 introduces briefly some elements of floating-point arithmetic and also it deals with presenting related work of some existing tools. Section 3 introduces our tool POP and its acitecture. The experimental examples and results are detailed in Section 4 before concluding in Section 5.

II. Background

In this section, we provide some background on floating-point arithmetic helpful to understand the rest of the article in Section II-A. Also, we will discuss the importance of mixed-precision tuning and we will finish by presenting some related work in Section II-B.

A. Basics on Floating-point Arithmetic

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

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

The IEEE754 Standard specifies some particular values for p, e_{min} and e_{max} [12]. Also, this standard defines binary formats (with β = 2) which are described in Table I. Moreover, the IEEE754 Standard defines four rounding modes for elementary operations over floating-point numbers which are [12]: towards $+\infty$, towards $-\infty$, towards zero and towards the nearest denoted by $\uparrow_{+\infty}$, $\uparrow_{-\infty}$, $\uparrow_0$ and $\uparrow_\sim$, respectively. Henceforth, we introduce two functions *ufp* and *ulp* which

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

TABLE I: Basic binary IEEE754 formats

compute the *unit in the first place* and the *unit in the last place* respectively. These functions are used to describe the error propagation across the computations [4]. Equation (2) presents the *ufp* of a number x

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

Noting that several definitions of *ulp* exist in literature [13]. We present the *ulp* of a floating-point number with p significand size as shown

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

B. Mixed-Precision Tuning and Related Work

a) ***Mixed-Precision Tuning challenges and Related Work:*** Past research [14] [15] [8] has shown that by using a combination of 32-bit and 64-bit floating-point arithmetic, the performance of many dense and sparse linear algebra algorithms can be significantly enhanced while maintaining the 64-bit accuracy of the resulting solution. Also, the twice number of single precision data elements can be stored at each level of the memory hierarchy (register file, the set of caches and the principal memory) because of its compact representation and then it increases crucially the number of cache and consumes more bandwidth between the memory levels. For example, taking a simple arithmetic expression $(u+v) \div (w-y)$, then, carrying out the division expression in a single precision and the rest in double precision would be the best choice [8]. In order to obtain the best floating-point formats as a function of the expected accuracy on the results, many efforts must be presented. We will present these approaches according to their static or dynamic methods of analysis.

- **Static Anaysis** Darulova and Kuncak [6] proposed a static analysis method to compute errors propagation. If their computed bound on the accuracy satisfies the post-conditions then the analysis is run again with a smaller format and it stops until finding the best format. Contrarily to our proposed tool, all their values have the same format (uniform-precision). Also, there are some recent efforts in rigorous floating-point error estimation which are based on combinations of abstract interpretation and conservative range calculation. In this context, Chiang et al. [2] have proposed a new method which allocate precision to the terms of only arithmetic expressions. Whereas they need to solve a quadratically constrained quadratic program to obtain their annotations. Nevertheless, Solovyev et al. [16] have proposed the FP-Taylor tool that implements the Symbolic Taylor Expansions method to estimate roundoff errors of floating-point computations. However, these approaches do not scale very well and therefore have not been applied to high precision computing workloads.
- **Dynamic Anaysis** Rubio-González et al. [7] proposed the Precimonious tool which is a dynamic automated search bases tool. It tries to decrease the precision of variables and checks if the accuracy requirements are still fulfilled. Precimonious executes different mixed-precision configurations of the program

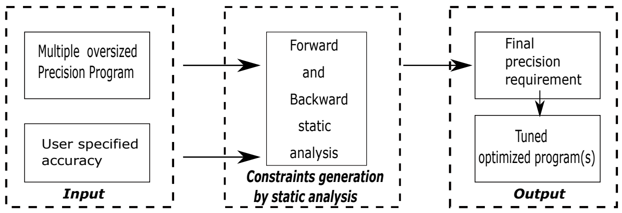

Multiple oversized Precision Program
User specified accuracy
Input
Forward and Backward static analysis
Constraints generation by static analysis
Final precision requirement
Tuned optimized program(s)
Output

Fig. 1: POP Mixed-Precision Analysis Approach

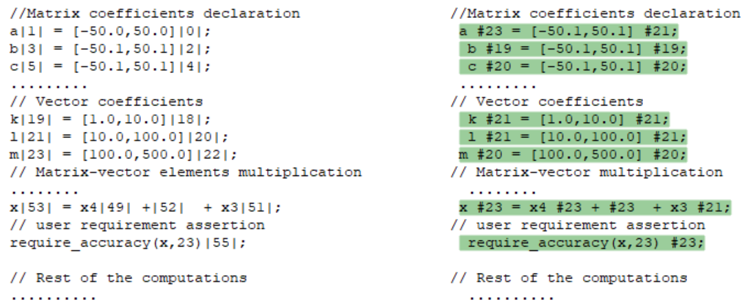
in order to identify the best configuration that satisfies the error threshold. Precimonious may be accelerated by a pre-processing blame analysis process [9]that empirically identifies variables that do not significantly affect program behavior, and thus are safe to be assigned lower bit-widths. This combined approach has shown better results in term of program speedup compared to using Precimonious alone. Moreover, Lam et al. instrument binary codes in a tool called CRAFT aiming to modify their precision without modifying the source codes. 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 main drawback of this tools is that the state space is exponential in term of number of variables and even exploring a subset is very time-intensive.

III. POP Tool: Precision OPtimizer

At this stage, we present our tool POP. As mentioned in Figure 1, POP takes as input a program performing floating-point computations in an oversized precision (generally double precision) referring to our simple imperative language [4]. Firstly, we call ANTLR (ANother Tool for Language Recognition) [17] in order to generate from a grammar file a parser that can build and walk parse tree. After, we proceed by a dynamic range determination phase which consists in launching the execution of the program a certain number of times and to make sure that no overflow can arises during our analysis (we manage to use a static analyzer in the future). Once the range of variables is given, POP starts the constraints generation thanks to the forward and backward transfer functions where the main semantic is detailed in [4]. The forward approach propagates safely the errors on the inputs and on the results of the intermediary operations aiming to determine the accuracy of the results. Next, based on the user accuracy requirement and the results of the forward analysis, the backward analysis computes the minimal precision needed for the inputs and the intermediate results in order to satisfy the assertions. Therefore, we assign to each control point three integer variables corresponding to the forward, the backward and the final accuracies so that the inequality in Equation 4 is verified. Hence, we notice that in the forward mode, the accuracy decreases contrarily to the backward mode when we strengthen the post-conditions (accuracy increases).

$$0 \leq acc_B(\ell) \leq acc(\ell) \leq acc_F(\ell) \tag{4}$$

Finally, POP resolves the previous constraints by calling the Z3 SMT solver [11] to find a solution for our constraints and it implements a cost function to refine the solutions obtained in term of optimality. In future work, we will explore the policy iteration technique as w new resolution method [18]. The transformed program is guaranteed to use variables of lower precision with an minimal number of bits than the original program. In order to better understand this approach, we present the following example. Considering a simple example of a matrix-vector multiplication. The coefficients of the matrix 3×3 and of the vector belong to multiple magnitude ranges. The vector coefficients are $x_1 = [1.0, 2.0, 3.0]$, $x_2 = [10.0, 100.0, 500.0]$ and $x_3 = [100.0, 500.0, 1000.0]$ and our matrix is defined by



```
//Matrix coefficients declaration
a|1| = [-50.0,50.0]|0|;
b|3| = [-50.1,50.1]|2|;
c|5| = [-50.1,50.1]|4|;
.........
// Vector coefficients
k|19| = [1.0,10.0]|18|;
l|21| = [10.0,100.0]|20|;
m|23| = [100.0,500.0]|22|;
// Matrix-vector elements multiplication
........
x|53| = x4|49| +|52|  + x3|51|;
// user requirement assertion
require_accuracy(x,23)|55|;

// Rest of the computations
..........
```

```
//Matrix coefficients declaration
a #23 = [-50.1,50.1] #21;
 b #19 = [-50.1,50.1] #19;
 c #20 = [-50.1,50.1] #20;
 .........
// Vector coefficients
 k #21 = [1.0,10.0] #21;
 l #21 = [10.0,100.0] #21;
m #20 = [100.0,500.0] #20;
// Matrix-vector multiplication
  ........
x #23 = x4 #23 + #23  + x3 #21;
// user requirement assertion
 require_accuracy(x,23) #23;

// Rest of the computations
  ..........
```

Fig. 2: Matrix-vector multiplication code snippet. The code on the left designs the initial program in double precision annotated with labels. On the right, the program after analysis annotated by the final accuracies at each label.

$M1 = \begin{bmatrix} [-50.1,50.1] & [-50.1,50.1] & [-50.1,50.1] \\ & [-10.1,10.1] & [-10.1,10.1] \\ [-5.1,5.1] & [-5.1,5.1] & [-5.1,5.1] \end{bmatrix}$ In this example, we suppose that all variables are in double precision before analysis (left hand side of Figure 2). Let $\overrightarrow{\oplus}$ denote the forward addition, $\overrightarrow{\otimes}$ for the forward multiplication and respectively $\overleftarrow{\oplus}$ and $\overleftarrow{\otimes}$ for the backward addition and multiplication. We define the forward addition as shown in Definition 1:

Definition 1: Let $\mathbb{F}_p$ and $\mathbb{F}_q$ denote two sets of floating-point numbers in accuracy p and q, respectively. The forward addition $\overrightarrow{\oplus}$ is given as shown in Equation (5) where $x \in \mathbb{F}_p$ and $y \in \mathbb{F}_q$. In Equation (5) and (6) , the operands $x_{p_{p'}}$ and $y_{q_{q'}}$ and their results $z_{r_{r'}}$ have respectively two parameters p, p', q, q' and r, r' which denote the correct precision of the result and of the error, respectively. ε_+ denotes the truncation error for the addition. We denote by σ_+ the precision of the operator $+$. More details of the static forward and backward transfer function are explained in [4]. The forward and backward transfer functions for the addition are given in equations (5) and (6).

$$\begin{aligned} \overrightarrow{\oplus}(x_{p_{p'}}, y_{q_{q'}}) = z_{r_{r'}} \quad where \quad r = \mathit{ufp}(x_{p_{p'}} + y_{q_{q'}}) - \\ \mathit{ufp}(2^{\mathit{ufp}(x_{p_{p'}})-p+1} + 2^{\mathit{ufp}(y_{q_{q'}})-q+1} + 2^{\mathit{ufp}(z_{r_{r'}})-\sigma_+}) \end{aligned} \tag{5}$$

$$\begin{aligned} \overleftarrow{\oplus}(z, y) = (z - y)_{p_{p'}} \quad with \quad p = \mathit{ufp}(z - y) - \mathit{ufp}(2^{\mathit{ufp}(z)-r+1} \\ -2^{\mathit{ufp}(y)-q+1} - 2^{\mathit{ufp}(x)-\sigma_+}) \end{aligned} \tag{6}$$

Now, we present in Figure 2 a code snippet of our example in order to present the important annotations that POP uses. It is obvious that our program contain several annotations. All the program variables are initialized to abstract values and annotated with their control points as shown in the left side of Figure 2 (i.e. $a|1| = [-50.0, 50.0]|0|;$). Also, we have the statement $\texttt{require\_accuracy}(\mathtt{x}, 23)^{|55|}$ which informs the system that the programmer wants to have 23 accurate binary digit on x at this control point. For 642 size of variables and 858 constraints, POP finds the minimal precision on the inputs and intermediary results satisfying the user assertion, as it is mentioned in the right side of Figure 2, in less than 0.5 seconds (time only for the resolution of constraints and the calls of the Z3 SMT solver). Moreover, the original program starts initially with 10335 bits and after program analysis the number of bits has been reduced to only 1813 bits needed to satisfy the assertions (an improvement of 83 % of the number of bits needed compared to the initial number).

After presenting the main objective of POP, we want at this stage to develop this tool for applications from several domains and especially the Internet of Things field. We had the idea to apply the optimization of the precision on applications related to Internet of the things because of the serious concerns about the memory and

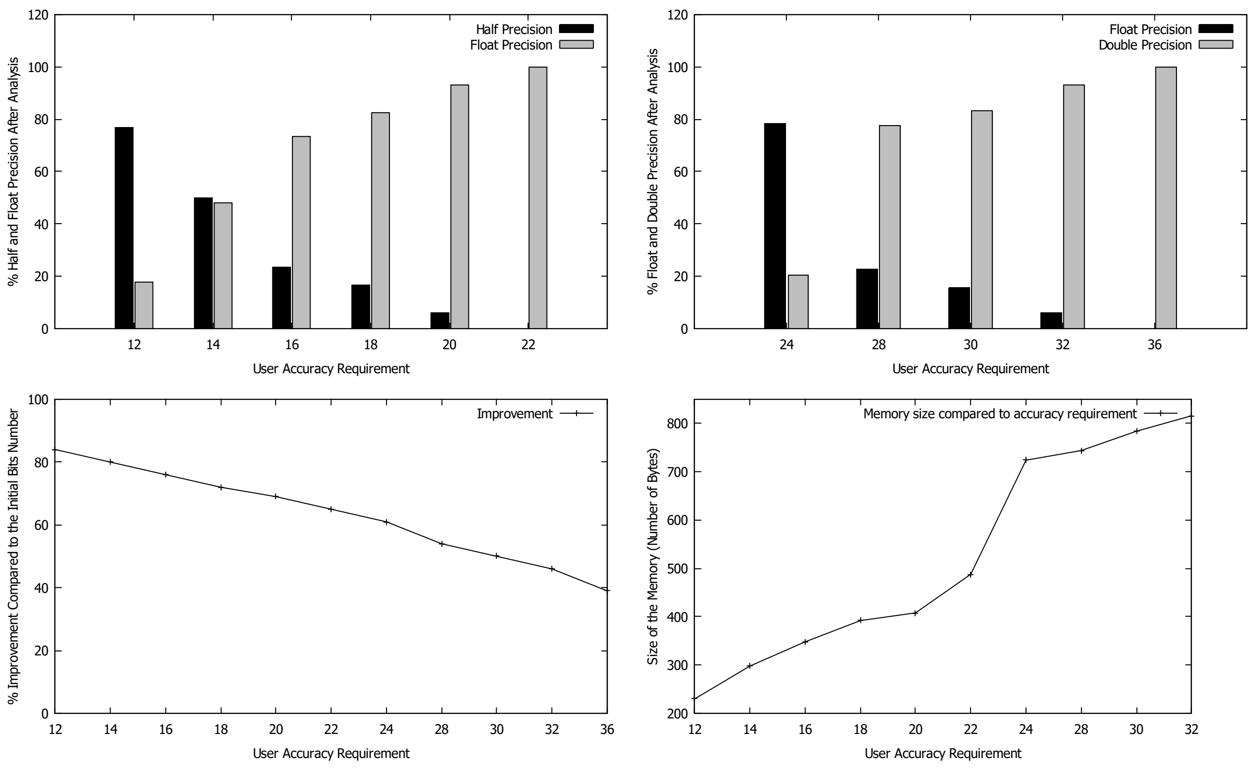


Fig. 3: Efficiency of POP on the accelerometer titlt measure application. Top left: the percentage of half and float precision for different accuracy assertions after analysis. Top right: The percentage of float and double variables for accuracies 24, 28, 30, 32 and 36. Bottom left: Improvement of the number of bits compared to the initial bits number in the original program. Bottom right: Memory size (in Bytes) in different requirements of accuracy

energy consumptions that are present nowadays on the majority of IoT devices.

IV. Experimental Results

a) ***Constraints Generation and Resolution:*** As we have explained above, POP generates linear constraints made of propositional logic formulas and relations between integer elements and calls Z3 SMT solver in order to obtain a solution. Also, an additional constraint related to a cost function ϕ (we take the same definition in [4]). The purpose of a cost function $\phi(c)$ of a given program c is to compute the sum of the accuracies of all the variables and the intermediary values collected in each label of the arithmetic expressions as it is shown in Equation (7).

$$\phi(c) = \sum_{x \in Id, \ell \in Lab} acc(x^{\ell}) + \sum_{\ell \in Lab} acc(\ell) \qquad (7)$$

After, our tool searches the smallest integer P such that our system of constraints admits a solution. Consequently, we start the binary search with $P \in [0{,}52 \times n]$ where all the values are in double precision and where n is the number of terms in Equation (7). While a solution is found for a given value of P, a new iteration of the binary search is run with a smaller value of P. When the solver fails for some P, a new iteration of the binary search is run with a larger P and we continue this process until convergence.

b) ***Measuring Tilt Angle using Three Axis:*** To study the usefulness of our tool, we choose to experiment our analysis on an example that measures an inclination angle with an accelerometer: a sensor capable of measuring, in three dimensions, the linear accelerations of an object as well as vibrations. As a matter of fact, there are accelerometers in many everyday objects use, such as smartphones, cars, sports watches and other devices: i.e. the accelerometer of a phone is able to give you the orientation of the phone but also ,as indicated by its name, the acceleration undergone by the phone. An accelerometer usually breaks down into two parts: a mechanical part which is responsible for detecting the accelerations of a mass contained in the device and an electronic part having for mission to interpret this signal. Our application describes how often accelerometers are used to measure a tilt of an object. Tilt detection is a simple application of an accelerometer where a change in angular position of the system in any direction is detected and indicated the corresponding angle scaled from microcontroller output and in order to have more accurate measurements of tilt in the x and y planes we therefore need a 3 axis accelerometer. We aim from the accelerometer experimentation to measure the usefulness of our analysis and how POP is capable to optimize the precision of our program variables. For this example, POP generates 1179 variables and 1767 constraints which is very manageable by the Z3 solver. Assuming that in the original program of our examples all the variables are in double precision, POP succeeded in turning off variables into half and float precision as shown in the top left side of Figure 3. For example, for an accuracy of 20, the percentage of variables passing in float is large compared to the variables in half precision (93.13% for float and 5.88% for half variables). Also, for accuracies greater than 22, POP manages correctly the precision turning approach by finding a float and double precisions compromise. As we show on the top right side of Figure 3, for a requirement accuracy of 30 and 32, the mixed precision between float and double is obtained. Also, we can say that of an accuracy of 24 there is as much float as double precision variables. Moreover, we notice that for an accuracy of 36 that all the variables remain in double precision and thus finding the minimal

```
xVal #20= [-2.0, 2.0] #20 ; yVal #20 = [-2.0, 2.0] #20 ; zVal #20  = [-2.0, 2.0] #20 ;
// total value of the acceleration
sos = xVal * xVal + yVal * yVal + zVal * zVal ;
total = sqrt(sos);
// Angles computation and conversion to degrees
//x Axis
u #21 = xVal #20 / #21 total #20 ;
angleX #22 = u + u * u * u / 6.0  + u * u * u * u * u
 * 3.0 /40.0 * 180.0 / 3.1416 ;
 require_accuracy(angleX, 24);
//y axis
v #21 = yVal #20 / #21 total #20 ;
angleY #22 = v + v * v * v / 6.0  + v * v * v * v * v
 * 3.0 /40.0 * 180.0 / 3.1416 ;
  require_accuracy(angleY, 24)
// z axis
w #21 = zVal #20 /#21 total #20 ;
angleZ #22 = 1.570796 - (w + w * w * w / 6.0  + w * w
 * w * w * w * 3.0 /40.0 * 180.0 / 3.1416) ;
 require_accuracy(angleZ, 24)
```

```
float xVal, yVal, zVal, sos;
double angleX, angleY, angleZ;
// init sos, total
.......
.......
//init angleX
angleX = u + u * u * u / 6.0  + u * u * u * u * u
 * 3.0 /40.0 * 180.0 / 3.1416 ;
 ......
//init angleY
angleY = v + v * v * v / 6.0  + v * v * v * v * v
 * 3.0 /40.0 * 180.0 / 3.1416 ;
 ......
//init angleZ
angleZ = acos(zVal / total ) * 180.0 / 3.1416;
.......
```

Fig. 4: Example of mixed-precision inference. Left: source program with inferred accuracies. Right : Program formats.

precision is only possible for accuracies lesser than 36. The bottom left of Figure 3 describes the improvement of the number of bits compared to the original program. In fact, the original program starts with 15105 bits (all in double precision) and after analysis we found that the improvement, in the number of bits needed to realize the user requirements, compared to the initial number of bits, ranges from 39% to 84 % for an accuracy starting from 12 to 36 which confirms the efficiency of our analysis. As the subject of saving memory is challenging to us, the bottom right of Figure 3 shows the memory used in Bytes by the program variables according to the precision requests. Initially, the memory size is equivalent to 816 Bytes and and we measure in this experiment the size of the memory with each precision inserted. An important observation on the behavior of our tool is that POP assigns zeros to the accuracies of the variables that are not used by the program. We present in Figure 4 our transformed tilt measure program where in the right side we describe the source program with the inferred accuracies after analysis and the program new optimized formats in the right side on the Figure 4.

V. Conclusion and Future Work

The main idea of this article is to experiment, POP, a tuning assistant for mixed-precision, in different domain of application. POP implements a static forward and backward analysis to determine the minimal precision on the inputs and intermediary results of a program that guarantees a desired accuracy of the outputs values. The originality of this article was to adapt the precision tuning technique in applications of Internet of Things because the type of problem of energy consumption and memory are widespread in this area. Our preliminary results shows that it is promising to explore the precision tuning in this unexplored domain. In this work, we have experimented POP on the example of an accelerometer which can be used to measure the static angle of tilt or inclination. Our preliminary results shows that POP succeeded in computing the accuracy needed for each variable and intermediary results by an improvement ranging from 65 % to 84 % for an accuracy lesser than 23. This work can be improved in several ways. Obviously, our approach can be extended to other applications from various fields including safety-critical systems such as control systems for vehicles, medical equipment and industrial plants. Also, we would like to explore the policy iteration method [18] as a replacement for the non-optimizing solver (Z3) coupled to a binary search used in this article. Finally, comparing our tool to other existing tools in the matter of analysis time and speed and the quality of the solution is a tremendous challenge to examine.

References

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

[2] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, "Rigorous floating-point mixed-precision tuning," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 2017. New York, NY, USA: ACM, 2017, pp. 300–315. [Online]. Available: http://doi.acm.org/10.1145/3009837.3009846

[3] "Patriot missile defense: Software problem led to system failure at dhahran, saudi arabia," General Accounting office, Tech. Rep. GAO/IMTEC-92-26, 1992.

[4] 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.

[5] M. Kim, J. Lee, Y. Kim, and Y. H. Song, "An analysis of energy consumption under various memory mappings for fram-based iot devices," in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, Feb 2018, pp. 574–579.

[6] E. Darulova and V. Kuncak, "Sound compilation of reals," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. ACM, 2014.

[7] 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*.

[8] 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.

[9] 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.

[10] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, 1977, pp. 238–252.

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

[12] 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.

[13] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*, 1st ed. Birkhäuser Boston, 2009.

[14] D. H. Bailey and J. M. Borwein, "High-precision arithmetic: Progress and challenges."

[15] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, "Exploiting mixed precision floating point hardware in scientific computations," 2007.

[16] 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.

[17] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed. Pragmatic Bookshelf, 2013.

[18] 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.