Precision Tuning of an Accelerometer-Based Pedometer Algorithm for IoT Devices

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*—Internet of Things (IoT) is considered as one of the most evolving technologies. It impacts almost every aspect of life in the modern world. One thing that prevents this technology from reaching its full potential is the too high memory and energy consumption of IoT devices. An aspect of this problem is the numerical computations performed by IoT devices which are generally carried out in higher precision than needed, by lack of techniques to tune this precision in function of the actual needs of the application. To surpass this limitation, we present our tool POP based on a static program analysis, done by abstract interpretation. POP provides a mixed precision tuning that finds the instructions and variables that may use lower precision with respect to the user accuracy requirements on the results. We demonstrate the efficiency of POP to tune an accelerometer-based pedometer algorithm for embedded applications.**

***Index Terms*—Computer arithmetic, precision tuning, pedometer, forward and backward static analysis.**

I. Introduction

Initially designed for scientific computing, floating-point arithmetic [1] is more and more used in many embedded and safety critical systems including cars, planes, nuclear power plants, or medical devices. These applications rely tremendously on floating-point computations especially with the wide availability of processors with hardware floating-point units such as in smartphones and, more recently, in IoT devices. However, inaccuracies in these computations can cause bugs, which can lead to disasters [2] [3] [4].

The Internet of Things (IoT) is one of the most evolving technologies that contribute significantly to the improvement of our daily lives. However, the problem of energy and memory consumption are ubiquitous in this field. Indeed, many IoT devices require continuous power, and although the battery is an option, it is not always economical. Other devices are powered by an independent power supply like battery and energy harvesters, which provide limited energy [5]. Thus, batteries require frequent changing and replacement caused by their short life cycle. In addition, the IoT device memory is used to store data, therefore, many memory accesses occur during the execution of the devices.

Nevertheless, IoT applications usually do not require very accurate results and, consequently, it is very feasible to lower the average precision of the computations to cope with memory and energy issues without affecting the efficiency of the devices. We propose to find the least data formats needed to ensure a certain accuracy, also called precision tuning, by means of our tool POP (Precision OPtimiser). The present work addresses the problem of determining the minimal precision on the inputs and intermediary results of a pedometer code performing floating-point computations while guaranteeing a desired accuracy on the outputs. The challenge is then to use no more precision than needed wherever possible without compromising the overall accuracy (using a too low precision for this algorithm and its data leads to wrong results). Let us remark that tuning the precision of the input data may let the device designer choose less accurate sensors, usually cheaper and less resource consuming. Our precision tuning technique offers many advantages. First, it can be repositioned to the fixed-point arithmetic for the case of IoT devices without floating-point units. Second, the POP tool aims to apply mixed-precision tuning on the floating-point formats of the program which consists on associating different precisions for different floating-point variables [6].

POP combines a forward and backward static analysis, done by abstract interpretation [7]. The forward analysis examines how errors are magnified by each operation which contribute to the accuracy on the results [8] [9]. By considering the user accuracy requirements at some control points and the results of the forward analysis, the backward analysis is a complementary approach that starts with the computed answer to determine the least floating-point input that would produce it. Our analysis is expressed as a set of propositional formulas on constraints between integer variables only (checked by the SMT solver Z3 [10]). As a result, the transformed program is guaranteed to use variables of lower precision than the original program. For our experiments, we use an accelerometer-based pedometer algorithm for embedded applications [11] coming from the IoT field. In previous work [12], we showed the usefulness of our analysis on an accelerometer code which can be used to measure the static angle of tilt or inclination. Here, we evaluate POP on a significantly more complex example, a pedometer that implements a step counting algorithm. The step counter calculates the steps from the x, y and z axis of the accelerometer example, depending on which acceleration axis change is the largest one. The steps of the pedometer algorithm and the experimental results of our method are illustarted in Section III.

The remainder of the article is organized as follows. Section II introduces the floating-point arithmetic and details the approach integrated in our tool throughout an example. Section III is devoted to the analysis of the experimental results. Section IV deals with the related work and Section V concludes.

II. Background Material

This section provides essential background on floating-point arithmetic (Section II-A) and presents the main technique of constraints generation on an accelerometer sensor similar to what we can find in smartphones to convert brute sensor data into movements in Section II-B. This accelerometer may also feed the pedometer algorithm introduced in Section III.

A. Elements of Floating-Point Arithmetic

The floating-point arithmetic is specified by the IEEE754 Standard. In this arithmetic, a number $x = s.m.\beta^{e-p+1}$is expressed in base $\beta = 2$. Each number has a sign $s \in$ {-1,1}, a mantissa $m = d_0.d_1...d_{p-1}$, whose digits are $0 \leq d_i < \beta$, $0 \leq i \leq p-1$. The precision is given by p and the exponent is $e \in [e_{min}, e_{max}]$. Briefly speaking, we present in Equation (1) the *ufp* of a floating-point number x where its *ulp* is given in Equation (2) with p bits of precision. These two functions make it possible to compute the propagation of errors throughout a computation.

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

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

B. Constraints Generation on a Three Axis Accelerometer

We explain, in the present section, the forward and backward static analysis expressed as a system of constraints performed by POP on the program of an accelerometer. This program allows one to compute the static angle of inclination also called tilt angle. An accelerometer detect movement; when moved, it records acceleration. It is used, in smartphones, handheld game consoles and other devices, to measure the acceleration of the device from side to side (lateral), up and down (longitudinal), and front to back (vertical) [12]. It can also provide inputs to the pedometer of Section III. In order to improve the accuracy of the measure of inclination in the x and y planes, a 3 axis accelerometer is used. Figure 1 describes the improvement found by POP on the number of bits in function of the desired accuracy on x, y and z by comparison to the initial code. Initially, the code used 15105 bits (all in FP64). For the static analysis, POP generates 1767 constraints among 1179 variables which is quite tractable for the Z3 solver. As a result, the improvement given by our tool on the quantity of bits required to reach the user specification, ranges from 39% to 84 % for given accuracy in the range of 12 to 36 (an accuracy n means that the computed result $\widehat{x}$ has n correct bits w.r.t. the exact result x, i.e $|x - \widehat{x}| \leq 2^{-n}$). This confirms the efficiency of our tool as we can see in Figure 1. Figure 2 shows the different formats of precision obtained by POP for a code snippet of the accelerometer example for a user accuracy of 23 bits on variable $angleX$. The complete accelerometer code has been presented in [12]. Now, we move to explain where this gain in precision comes from and what is the technique behind our tool.

From the technical point of view, POP takes at first a program in an oversized precision, generally all the variables are in the IEEE754 double precision. POP is able to analyze programs supporting the four elementary arithmetic expressions, arrays, loops, conditions, trigonometric functions and

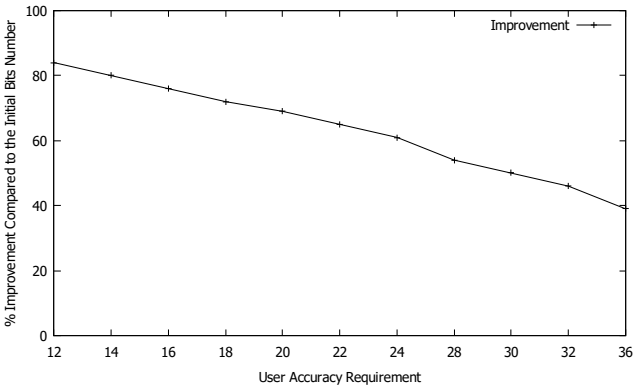

Fig. 1: Improvement with respect to the initial number of bits in the initial accelerometer code.

the square root function. A range determination by dynamic analysis is performed on all the program variables. In the future, we plan to use a static analyzer in the future. Next, a set of constraints is generated from the syntactic tree of the program. These constraints encode our forward and backward static analysis. The Z3 SMT solver finds the solution [10].

More precisely, the forward analysis propagates soundly the roundoff errors on the inputs of the program as well as on the intermediary results. Doing so, we can determine the accuracy of the results. Later on, using the user accuracy requirements plus the accuracies given by the forward analysis, our backward analysis infers the least precision required for the inputs and the results of the intermediary computations to fulfill the assertions [9] (our programs contain a `require_accuracy(x,p)`$|\ell|$ statement which indicates to POP the precision asked by the user: `p` bits for variable `x` at the control point ℓ). Hence, we assign to any control point ℓ two kinds of integer parameters: the *ufp* of the value (see Equation (1)) and 3 integer variables: one for the forward accuracy, one for the backward and one for the final accuracy, denoted by $acc_F(\ell)$, $acc_B(\ell)$ and $acc(\ell)$ respectively, such that Equation (3) always holds.

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

We encourage the reader to refer to [9] for more explanations about the forward and backward transfer functions for arithmetic expressions, statements, trigonometric functions and square root function.

The main point of our technique is to formulate the transfer functions (both forward and backward) as a set of constraints containing only propositional logic formulas and affine expressions between integer quantities, although the program contains non-linear floating-point computations. Once the constraints have been generated, POP uses Z3 [10] in order to find a solution to this system. The solutions returned by Z3 being not unique, we add an additional constraint to our system which corresponds to a cost function ϕ. Let Id and Lab denote the sets of variable identifiers and labels respectively. Let $acc(x^\ell)$ be a variable of the constraint system corresponding to the final accuracy of x at control point ℓ and let $acc(\ell)$ be the final accuracy of the operation done at control point ℓ. For the source code c, the $\phi(c)$ function aims at computing the sum of the $acc(x^\ell)$ quantities

```
xVal#21 = [2.0,2.0]#21;
yVal#21 = [2.0,2.0]#21;
zVal#21 = [2.0,2.0]#21;
u#22= xVal#21 *#22 [0.3,0.3]#21;
angleX0= 0#22;
angleX1#23 = u#22 *#23 u#22 *#22
 u#22 *#22[0.1666,0.1666]#22;
angleX2#20 = u#22 *#20 u#22 *#20
 u#22 *#19
u#22 *#19 u#22 *#19 4.2971#19;
angleX#23 = angleX0#22 +#23 angleX1#23
 +#23 angleX2#20 ;
require_accuracy(angleX,23)#23; [...]
```

Fig. 2: Mixed-precision tuning on the computation and conversion of $angleX$ of the acceleromter program.

for all variables and accuracies $acc(\ell)$ at each control point ℓ. This corresponds to the functions given in Equation (4).

$$\phi(c) = \sum_{x \in Id, \ell \in Lab} acc(x^{\ell}) + \sum_{\ell \in Lab} acc(\ell) \quad (4)$$

Next, POP looks for the minimal integer P such that the constraints given to Z3 admit a solution. As a consequence, we perform a binary search for $P \in$ [0,52 $\times$ n]. The value $52 \times n$ corresponds to the upper bound when all the accuracies are in double precision. When for some value of P, a solution is found by Z3, we reiterate with a smaller value of P. Conversely if Z3 fails for some P, we reiterate with a larger P and we continue this process up to convergence. Note that $\phi(c)$ becomes more complex when dealing with arrays. For the sake of conciseness, we omit these details in this article.

Our goal is to use POP on applications coming from the Internet of Things field because of memory and energy consumptions issues on many IoT devices. In the next section, we apply the optimization of the precision on an accelerometer-based pedometer for embedded applications.

III. Precision Tuning of a Pedometer Algorithm

We show, in this section, a classical pedometer code and how POP is able to tune the precision of this code. Section III-A introduces the algorithm and Section III-B presents some experimental results.

A. Step Counting Algorithm for Embedded Applications

A pedometer is a small device that counts the number of footsteps [13]. It is also called a footstep counter and it uses an accelerometer similar to the one of Section II-B to count the number of footsteps. Some pedometers also perform a few additionally tunable computations to detect how far a person walked in miles or how many calories have been burned [14]. Since the program is complex in its structures, containing nested loops, arrays and conditions, there are several stages in footstep detection [15] as summarized in Figure 3. We outline each of these in this section.

a) Extract 3D Vector: The algorithm, at the first stage, takes the magnitude of the entire acceleration vector i.e. $\sqrt{x^2 + y^2 + z^2}$, where x, y, and z are the outputs of the accelerometer along the three axis.

b) FIR low-pass filter: Sometimes, the pedometer vibrates very quickly or very slowly for a reason other than walking or running. The step counter will also take it as a footstep and this invalid information must be discarded. So, the second step consists of removing the noise and extracting the specific signal corresponding to walking. A simple solution is to use a low pass filter that keeps only frequencies related to walking and removes the rest [13]. In practice, a low pass filter is a circuit that modifies, reshapes or rejects any unwanted high frequencies of an electrical signal and accepts or transmits only the signal desired by the circuit designer. For instance, if a regular walking pace is under two steps a second, equivalent to 2 Hz, then all frequencies above this value may be removed by the filter and all other activities such as running or bicycling cannot be detected.

c) Autocorrelation to find the signal periodicity: The autocorrelation function [11] is the core of the step counter algorithm. It can be used to find the periodicity of a noisy signal in the time domain. Briefly speaking, we call a periodicity a pattern belonging inside a time series and which is repeated at regular time intervals. So, the autocorrelation function correlates the elements to others of the same series which are separated by a determined time interval.

d) Footstep detection using derivative: In this step, the algorithm calculates the derivative and finds the first zero crossing from positive to negative (or negative to positive), which corresponds to the first positive peak in the autocorrelation. Finally, by counting the number of times the derivative function has changed from positive to negative, the number of steps occurred is detected. Figure 3 illustrates the whole algorithm stages described above. For each output (which is also the input for the next algorithm). POP is annotated with assertions indicating which accuracy the user wants for the variables of interest. The main variables used in our example are the following. First, num_Tuples denotes the width of the window used to seek autocorrelation (400 tuples (x, y, z) in our example). In the next section, we will make vary the size of the window, as well as the accuracy requirements, and see its impact on the performance of POP. Next, mag_sqrt holds the magnitude of data x, y and z. The annotation $require\_accuracy(mag\_sqrt, 23)$ indicates to POP that all the values stored in this variable must have an accuracy of 23 bits corresponding to a single precision number rounded to the nearest. At the output of the FIR low-pass filter, the signal is given in variable lpf. After computing and removing the mean, the autocorrelation is applied and the results are holden in $autocorr\_buff$. Finally, the algorithm calculates the derivative and stores it in the $deriv$ variable. Noting that all theses variables are in double precision before the start of POP analysis. After the explanation of the different steps of the footstep counter algorithm given to POP, we measure in the next section the usefulness of our precision tuning analysis.

B. Experimental Evaluation

The main results of running POP on the footstep counter algorithm for the data set of [11] are presented in figures 4, 5, 6 and 7 which correspond to:

- The measurements of performance of POP in terms of precision improvement compared to the original program. The total number of bits is obtained by adding the number of bits needed to store all the variables and intermediary results at all the control points. In the original program, it is equal to 53 times the number of

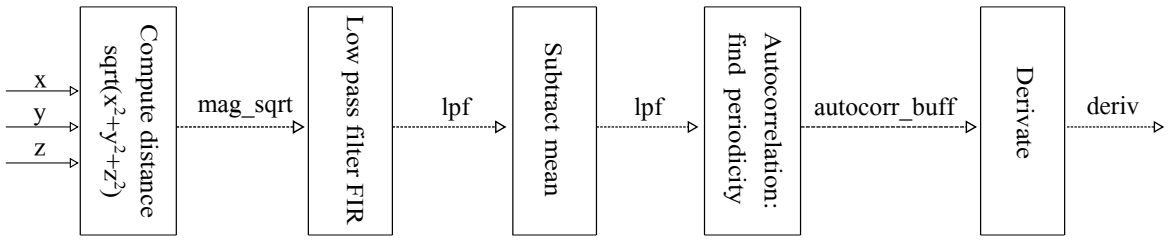

Fig. 3: Overview of the footstep counter algorithm.

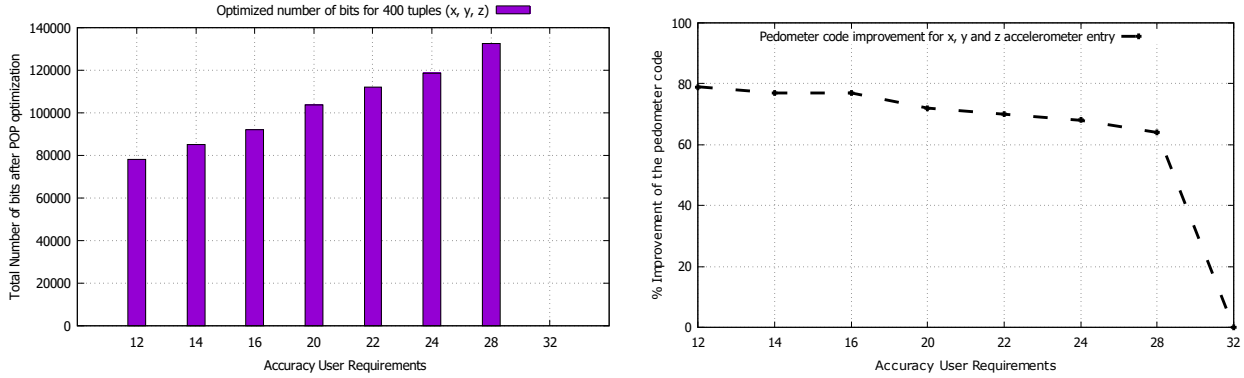

Fig. 4: Top left: Number of bits optimized for different user accuracy requirements. Top right: Improvement compared to the initial total of bits for the original program.

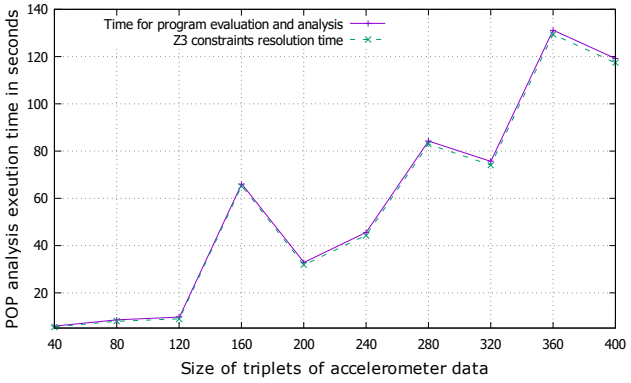

Fig. 5: Comparing the execution time of POP analysis and the time spent by Z3 to find a solution to our system of constraints.

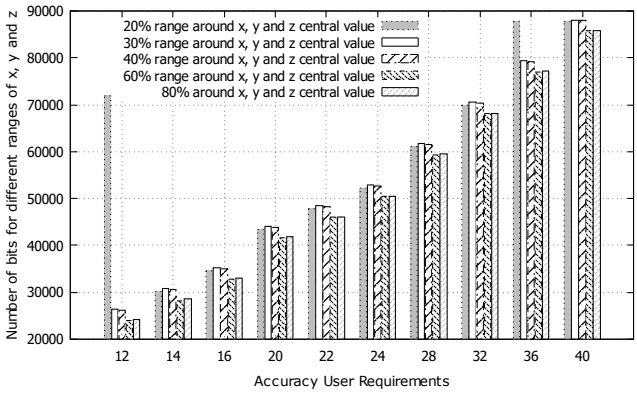

Fig. 6: POP optimization of the total number of bits for different magnitude of ranges of x, y and z.

control points, 53 being the size of mantissas in double precision,
- The execution time for the program analysis,
- The optimisation in term of number of bits for different intervals around the x, y and z average values,
- The mixed-precision configurations obtained after analysis in terms of number of variables or operations that we may tune into IEEE754 double (FP64), simple (FP32), precision (FP16) and minifloat precision (FP8).

We ran our experiments on an Intel Core i5-8350U 1.7GHz Linux machine with 8 GB RAM.

Of the eight accuracy requirements given to POP (from 8 to 32 bits of accuracy by step of 4), the improvement compared to the initial number of bits in the original pedometer code is considerable. As shown on the right hand side of Figure 4, the improvement varies between 64% for an accuracy equal to 28 to 79% for an accuracy of 12. In addition, let us note that POP remain all variables in double precision for an accuracy of 32 bits and so no mixed precision tuning is achieved beyond this accuracy for this example. Likewise, these results are confirmed in the left hand side of Figure 4. Initially starting with 364905 bits in the original program, the total number of bits is optimized for more than 50% for an accuracy of 28 (132797 bits).

Clearly, the various measurements of POP execution time illustrated in Figure 5 are reasonable and only take a few minutes for the whole pedometer code with a window of 400 data sets. Accordingly to Figure 5, we observe that the execution time increases when modifying the size of the accelerometer data set (or in other words, the size of

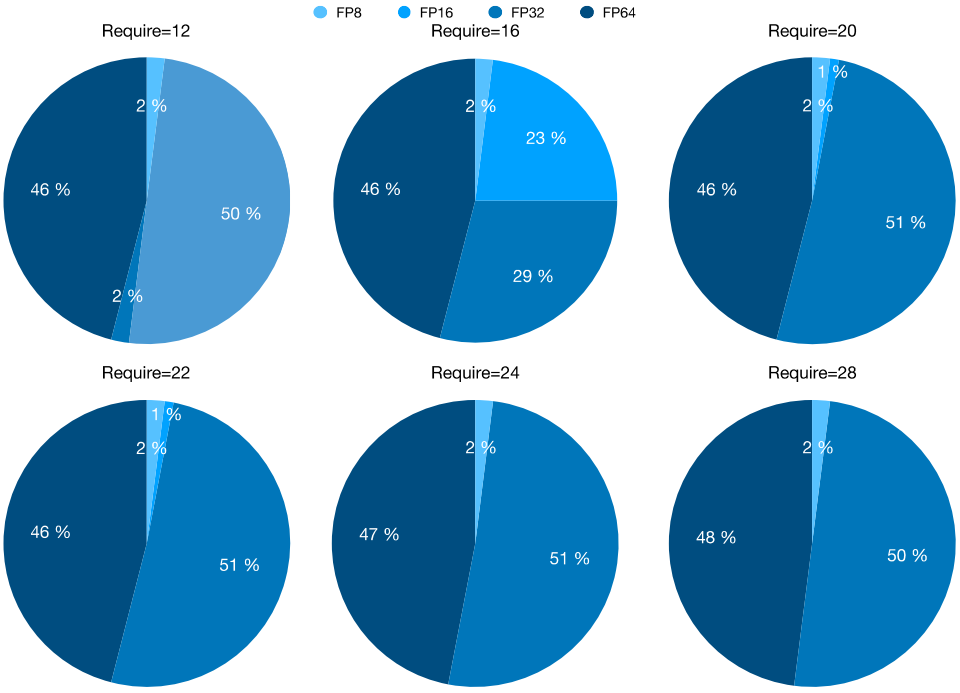

Fig. 7: The percentage of program variables in FP8, FP16, FP32 and FP64 after POP analysis.

the window used in the autocorrelation stage). Indeed, we observe that almost all of POP execution time is spent in calls to Z3.

For windows of 160, 280 and 360 for x, y and z, the Z3 solver takes more to solve the constraints even though the execution time to find the new formats for the program variables of interest remains practical in our working environment. In addition, POP being a static analysis tool admitting sets for its inputs, Figure 6 shows how POP behaves if there is no scalar inputs but intervals. In practice, we have taken intervals around average values for x, y and z so as to be based on a set of executions instead of a single one. The histogram of Figure 6 shows that POP succeeded in optimizing the initial number of bits for the original pedometer code, starting with 238977 bits, for a user accuracy more than 36 bits while for the case of the scalar values of the left hand side of Figure 4 there were no precision tuning beyond an accuracy of 32.

Assuming that in the original pedometer code, all the variables are in double precision, POP succeeded in turning off variables into half and float precision as shown in Figure 7 and other variables remain in double precision for some accuracy assertions. The various diagrams of Figure 7 show the percentage of variables, after POP analysis, in FP8, FP16, FP32 and FP64. To mention a few, 23% of variables are tuned to simple precision while the majority remains in FP64 for an accuracy of 16. Also, 50% of the program variables are turned off in simple precision for an accuracy of 28 although 48% still in double 64-bit floating-point precision and no variables tuned to FP8 precision. We present in Figure 8 the new mixed-precision formats of three steps of the footstep detection algorithm detailed in Section III-A which are: the low pass filter, remove of the mean and the autocorrelation function after POP analysis. The top left of Figure 8 shows the original code for these functions where all the variables are in double precision and on the top right we annotate the code with the new optimized formats for a user requirement of 20 bits for each function output.

IV. Related work

Many efforts have been done with the aim to find the minimal precision on program variables while satisfying the desired accuracy on the results. 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: Several rigorous static analysis approaches employed interval and affine arithmetic or Taylor series approximations to analyze numerical stability and to provide rigorous bounds on rounding errors. In this context, Solovyev et al. [16] have proposed the FP-Taylor tool which implements a method called Symbolic Taylor Expansions in order to estimate round-off errors of floating-point computations. More recently, work in FP-Taylor was developed to ensure the mixed-precision tuning technique. In practice, 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 unsuited for most programs. Darulova et al. [17] proposed a technique to rewrite programs by adjusting the evaluation order of arithmetic expressions prior to tuning. However, the technique is limited to rather small programs that can be verified statically.

b) Dynamic Searching Applications: Precimonious [18] is a dynamic automated search based tool. It aims at finding the *1-minimal* configuration, i.e., a configuration where

```
/*lowPassfilt function*/
aux0 = 0.0;
for(n = 0; n < numTuples; n++) {
 tempLpf = 0.0;
  for (i = 0; i < lpfFiltLen; i++){
   if(n - i >= 0.0){
    aux0 = lpfCoeffs[i] * magSqrt[n - i];
    tempLpf = tempLpf + aux0;};
 };
};
require_accuracy(tempLpf, 20);

/* remove_mean function*/
sum = 0.0;
 for(i = 0, i < numTuples, i++){
  sum = sum + lpf[i];};
sum =  sum / numTuples;
for (i = 0; i < numTuples; i++){
 lpf[i] = lpf[i] - sum ; };
require_accuracy(lpf, 20);

 /*Autocorrelation function*/
for(lag =0, lag < numAutoCorrLags, lag++){
 tempAc = 0.0;
 aux1 =0.0;
  for(i=0; i<numTuples-lag; i++){
   aux1 = lpf[i] * lpf[i+lag];
   tempAc = tempAc +  aux1;};
 autoCorrBuff[lag]= tempAc;
};
require_accuracy(autoCorrBuff, 20);
```

```
/*lowPassfilt function*/
aux0#20 = 0.0#20;
for(n = 0; n < numTuples; n++) {
 tempLpf#20 = 0.0#20;
  for (i = 0; i < lpfFiltLen; i++){
   if(n - i >= 0.0){
    aux0#20 = lpfCoeffs[i]#20 *#20 magSqrt[n - i]#35;
    tempLpf#20 = tempLpf#20 +#20 aux0#20;};
  };
};
require_accuracy(tempLpf, 20)#20;

/* remove_mean function*/
sum#23 = 0.0#23;
for(i = 0; i < numTuples; i++){
 sum#23 = sum#23 +#23 lpf[i]#23;};
sum#23 = sum#23 /#23 numTuples#13;
for (i = 0; i < numTuples; i++){
 lpf[i]#23 = lpf[i]#23 -#23 sum#23 ;};
require_accuracy(lpf, 20)#20;

/*Autocorrelation function*/
for(lag =0; lag < numAutoCorrLags; lag++){
  tempAc#24 = 0.0#24;
  aux1#24 = 0.0#24;
   for(i=0; i<numTuples-lag; i++){
    aux1#24 = lpf[i]#23 *#24 lpf[i+lag]#23;
    tempAc#24 = tempAc#24 +#24 aux1#24;
   };
 autoCorrBuff[lag]#24 = tempAc#24;
};
require_accuracy(autoCorrBuff, 20)#20;
```

Fig. 8: Left: Low pass filter FIR, remove mean and autocorrelation functions of the original program. Variable lpf holds the filtered signal where $lpfFiltLen$ is the length of the filter. $magSqrt$ denotes the square root of magnitude data of x, y and z. $numAutoCorrLags$ denotes the number of lags to calculate the autocorrelation and $autocorr\_buff$ holds the results. The $require\_accuracy$ statement states that the variables must have 20 bits. Right: source program with inferred accuracies.

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 [19] 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. Following with the idea of program transformation, the recent tool AMPT-GA [20] 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. Lately, a new solution called HiFPTuner [21] reduces 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 addition, Lam et al. [6] 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 major drawback is that it does not guarantee a reduction in the execution time. We mention, also, some models in [22] 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 error analysis, not for finding faster configurations.

V. Conclusion

We have evaluated, in this article, our tool POP in the IoT field on a complex example, a pedometer that implements a step counting algorithm for embedded applications. This example is significantly more complex than the accelerometer example of [12] which can be used as input of the present pedometer. The main technique of POP brings two novelties: a forward and backward static analysis and then to express the obtained transfer functions as a set of constraints made of first order predicates and affine integer relations only. These constraints are checked by the Z3 SMT solver. Our results show that POP achieve success when computing the accuracy desired by the user for each variable by an improvement ranging from 64 % to 79 % for an accuracy lesser than 32. In addition, our approach manages correctly the mixed-precision turning which is confirmed by the different formats of program variables obtained after the analysis. It is also noticeable that POP execution time remain very short, even for complex codes like the code of the pedometer.

In the future, we aim to go further in this domain application by applying our method to a real program integrated

in an IoT device to measure physically the gain in memory and energy.

Also, we aim at improving our technique for precision tuning by generating an Integer Linear Problem (ILP) from the program source code. Basically, this will be done by reasoning on the most significant bit and the number of significant bits of the values which are integer quantities. The solution to this problem, computed by a linear programming solver, will give the optimal data types at the bit level. A finer set of semantic equations can also be proposed which does not reduce directly to an ILP problem. So we aim at implementing these two techniques and comparing them to the current method.

REFERENCES

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

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

[3] A. D. Franco, H. Guo, and C. Rubio-González, "A comprehensive study of real-world numerical bug characteristics," in *ASE 2017*. IEEE Computer Society, 2017.

[4] D. Monniaux, "The pitfalls of verifying floating-point computations," *CoRR*, vol. abs/cs/0701192, 2007. [Online]. Available: http://arxiv.org/abs/cs/0701192

[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 *4th World Forum on Internet of Things (WF-IoT)*, 2018.

[6] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre, "Automatically adapting programs for mixed-precision floating-point computation," in *Supercomputing*. ACM, 2013.

[7] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*. ACM, 1977, pp. 238–252.

[8] M. Martel, "Floating-point format inference in mixed-precision," in *NASA Formal Methods, NFM 2017*, ser. LNCS, vol. 10227, 2017.

[9] 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*, ser. Communications in Computer and Information Science, vol. 1165. Springer, 2019.

[10] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.

[11] D. Morris, T. S. Saponas, A. Guillory, and I. Kelner, "Recofit: Using a wearable sensor to find, recognize, and count repetitive exercises," pp. 3225–3234, 2014.

[12] D. Ben Khalifa and M. Martel, "Precision tuning and internet of things," in *Internet of Things, Embedded Systems and Communications, IINTEC 2019*. IEEE, 2019, pp. 80–85.

[13] T. Ahola, "Pedometer for running activity using accelerometer sensors on the wrist," *Medical Equipment Insights*, 2010.

[14] N. Zhao, "Full-featured pedometer design realized with 3-axis digital accelerometer," 2010.

[15] R. Libby, "A simple method for reliable footstep detection on embedded sensor platforms," *Sensors (Peterborough, NH)*, 2008.

[16] A. Solovyev, M. S. Baranowski, I. Briggs, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan, "Rigorous estimation of floating-point round-off errors with symbolic taylor expansions," *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 1, pp. 2:1–2:39, 2019.

[17] E. Darulova, E. Horn, and S. Sharma, "Sound mixed-precision optimization with rewriting," *CoRR*, vol. abs/1707.02118, 2017.

[18] 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 *High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, pp. 27:1–27:12.

[19] 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 *International Conference on Software Engineering (ICSE)*, 2016, pp. 1074–1085.

[20] 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*. ACM, 2019, pp. 160–170.

[21] H. Guo and C. Rubio-González, "Exploiting community structure for floating-point precision tuning," in *Software Testing and Analysis*. ACM, 2018.

[22] M. O. Lam and B. L. Rountree, "Floating-point shadow value analysis," in *Extreme-Scale Programming Tools*. IEEE Press, 2016.