

A Study of the Floating-Point Tuning Behaviour on the N-body Problem

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

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

Abstract. In this article, we apply a new methodology for precision tuning to the N-body problem. Our technique, implemented in a tool named POP, makes it possible to optimize the numerical data types of a program performing floating-point computations by taking into account the requested accuracy on the results. POP reduces the problem of finding the minimal number of bits needed for each variable of the program to an Integer Linear Problem (ILP) which can be optimally solved in one shot by a classical linear programming solver. The POP tool has been successfully tested on programs implementing several numerical algorithms coming from mathematical libraries and other applicative domains such as IoT. In this work, we demonstrate the efficiency of POP to tune the classical gravitational N-body problem by considering five bodies that interact under gravitational force from one another, subject to Newton's laws of motion. Results on the effect of POP in term of mixed-precision tuning of the N-body example are discussed.

Keywords: Computer arithmetic · Precision tuning · Integer linear problems · N-body problem · Numerical accuracy

1 Introduction

Reducing the precision of floating-point data, also called precision tuning, provides for many practitioners in High-Performance Computing (HPC) and related fields an opportunity for exploring trade-offs in accuracy and performance [16]. Compared to higher-precision data formats (e.g. `binary64` and `binary128`), lower-precision data formats result in higher performance for computational intensive applications such as lower resource cost, reduced memory bandwidth requirements and energy consumption: the performance of `binary32` operations on modern architectures is often at least twice as fast as the performance of `binary64` operations [1].

A number of tools has been developed to assist developers in exploring the trade-off between floating-point accuracy and performance [6,9,13–15,21]. A common purpose of these techniques is that they follow a try and fail strategy to reduce the precision with respect to an accuracy constraint: they create a program search

© Springer Nature Switzerland AG 2021
O. Gervasi et al. (Eds.): ICCSA 2021, LNCS 12953, pp. 176–190, 2021.
https://doi.org/10.1007/978-3-030-86976-2_12

space then tune either program variables or assembly instructions while the optimized data formats are dependent to the tuning inputs. If the accuracy of the results are not satisfying, some variables or instructions are removed from the search space and this process is applied repeatedly until a (locally) optimal solution is returned. In the best of cases of recent tools, the program is no longer treated as a black-box but an analysis of the source code and the runtime behaviour is performed in order to provide a customized search space classification and then to identify dependencies among floating-point variables.

Unlike existing approaches, POP [2–5] implements a static technique based on a semantical modelling of the propagation of the numerical errors throughout the code. This results in generating a system of constraints whose minimal solution gives the best tuning of the program, furthermore, in polynomial time. The key feature of our method is to find directly the minimal number of bits needed, known as bit-level precision tuning, at each control point to get a certain accuracy on the results. Hence, it is not dependant of a certain number of data types (e.g. the IEEE754 formats [1]) and its complexity does not increase as the number of data types increases. In practical terms, by reasoning on the number of significant bits of the program variables and knowing the weight of their most significant bit thanks to a range analysis performed before the tuning phase (details in Sect. 4), POP is able to reduce the problem to an Integer Linear Problem (ILP) which can be optimally solved in one shot by a classical linear programming solver (no iteration) − we use GLPK [17] in practice. The method scales up to the solver limitations and the solutions are naturally found at the bit level. Furthermore, POP implements an optimization to the previous ILP method. The purpose of this new method is to handle carry bits by being less pessimistic on their propagation throughout arithmetic expressions. By doing so, a second finer set of constraints is generated by POP and the problem does not remain any longer to a pure ILP problem (min and max operators are needed). Then we use policy iteration (PI) technique [7] to find optimal solutions.

Proposed Contributions. In this article, we validate the efficiency of our approach on one of the oldest problem of modern physics, the N-body problem [12]. An N-body simulation numerically approximates the evolution of a system of bodies that interact with one another through some type of physical forces, where N presents the number of bodies in the system ($N = 5$). The program implements a second order differential equation which needs to be solved to get a location of the bodies for a given timevalue. By varying the required accuracy by the user, we show experimentally that POP succeeds in tuning the N-body program (original program $\simeq$ 330 LOCs). As a result, the transformed program is guaranteed to use variables of lower precision with a minimal number of bits than the original one. Prior work on the precision of N-body simulations have been carried out for a long time [18]. Compared to other experiments carried out with POP [2,3], the N-body example presents new difficulties, mainly more complex computations and a wide range of values with different magnitudes.

The different experimental evaluations presented in this article are the following. First, we measure the distance between the exact position of the bodies,

`Jupiter`, `Saturn`, `Uranus` and `Neptune` (the `Sun` position is fixed), computed with 500 bits and the position computed with n bits where $n = 11, 18, 24, 34, 43$ and 53 bits. These distances are given for each body with different time of simulation (10 and 30 years). Second, we demonstrate on this example the ability of POP to generate an MPFR code [11] with the new data types returned by the solver. Furthermore, we measure the global analysis time taken by POP and the execution time of the MPFR generated code and we prove that POP returns solutions in a few of a seconds. The global analysis time includes the time of the program evaluation, the range analysis determination, the constraint generation and their resolution by the solver.

Roadmap. The rest of this article is structured as follows. Section 2 describes the existing approaches for precision tuning. In Sect. 3, we discuss the technique behind POP on the illustrative example of N-Body problem. In Sect. 4, we point out the key contributions made by POP to tune floating-point programs. Section 5 presents a comprehensive evaluation and the experimental results of POP on the N-body problem and concluding remarks are discussed in Sect. 6.

2 Related Work

As we have discussed, the general areas for floating-point precision tuning have been receiving a lot of attention. In order to attain optimally lowered precision, such approaches are divided into two categories: formulating precision tuning as an optimization problem using a static performance and accuracy model, and dynamically searching through different precisions to find a local optimum.

Static Performance and Accuracy Model. Prior work in static error analysis provides a foundation for rigorously determining what precisions are required to meet error constraints for particular closed form equations. In this context, the FP-Taylor tool [6] has proposed a method called Symbolic Taylor Expansions in order to estimate round-off errors of floating-point computations. Unlike dynamic tools, the precision allocation guarantees to meet the error target across all program inputs in an interval. Even so, FP-Taylor is not designed to be a tool for complete analysis of floating-point programs: conditionals and loops can not be handled directly. More recently, they have extended their work by performing a broad comparison of many error bounding analyses to ensure the mixed-precision tuning technique [24]. Darulova et al. [9] 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. Basically, all the above methods suffer from scalability limitations and do not leverage community structure to guide the search. On the other hand, concerning scalability, POP generates a linear number of constraints and variables in the size of the analyzed program.

Dynamic Searching Applications. Precimonious [21] is a dynamic automated search based tool that leverages the LLVM framework to tweak variable declarations to build and prototype mixed-precision configurations. It aims at

finding 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. Lately, a new solution called HiFPTuner [13], which is an extension Of Precimnious, uses dependence analysis and edge profiling to enable a more efficient hierarchical search algorithm for mixed-precision configurations. As with other dynamic tuners, HiFPTuner's configurations are dependent on the tuning inputs, and no accuracy guarantee is provided for untested inputs. Craft [15] is a framework that uses binary instrumentation and modification to build mixed-precision configurations of existing binaries that were originally developed to use only double-precision. Stoke [23] is a general stochastic optimization and program synthesis tool to handle floating-point computation. Their algorithm applies a variety of program transformations, trading bit-wise precision for performance to enhance compiler optimization on floating-point binaries. Another dynamic tool is Adapt [19]. It uses the reverse mode of algorithmic differentiation to determine how much precision is needed in a program inputs and intermediate results in order to achieve a desired accuracy in its output, converting this information into precision recommendations. Following the idea of program transformation, the Ampt-Ga tool [14] 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. POP only focuses on the precision tuning problem. Hence, the input codes are taken as-is and we do not modify them. However, POP is compatible with other tools for program transformation for numerical accuracy [8,22]. Typically, these tools reorder the computations to make them more accurate in the computer arithmetic. For example, for sums, numbers will be added in increasing order of magnitude.

3 Running Example

Our tool, POP, automates precision tuning of floating-point programs. We reduce the problem of precision tuning to determining which program variables, if any, can have their types changed to a lower precision while satisfying the user accuracy assertions. In this section, we introduce the gravitational planetary problem simulation code. For this example, we aim at modelling the simulation of a dynamical system describing

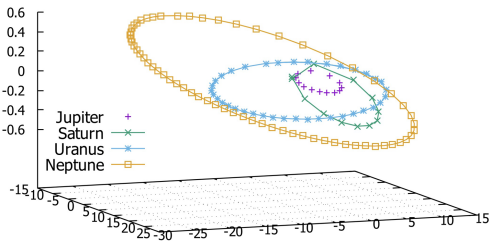


Fig. 1. Simulated movement of the bodies.

```
1  days_per_year^ℓ11 = 365.24^ℓ10;
2  dt^ℓ13 = 0.01^ℓ12;
3  t^ℓ15 = 0.0^ℓ14;
4  t_max^ℓ17 = 1000.0^ℓ16;
5                [...]
6  xJupiter^ℓ39 = 4.8414316^ℓ38;
7  vxJupiter^ℓ48 = 0.0016600767^ℓ44
8   *^ℓ47 days_per_year^ℓ46;
9  massJupiter^ℓ63 = 9.5479196E-4^ℓ59
10 *^ℓ62 solar_mass^ℓ61;
11 xSaturn^ℓ65 = 8.343367^ℓ64;
12               [...]
13 vxSaturn^ℓ74 = -0.002767425^ℓ70
14 *^ℓ73 days_per_year^ℓ72;
15 massSaturn^ℓ89 = 2.8588597E-4^ℓ85
16 *^ℓ88 solar_mass^ℓ87;
17               [...]
18 while (t^ℓ143 <^ℓ146 t_max^ℓ145) {
19   dx^ℓ757 = xJupiter^ℓ753 -^ℓ756 xSaturn^ℓ755;
20   dy^ℓ763 = yJupiter^ℓ759 -^ℓ762 ySaturn^ℓ761;
21   dz^ℓ769 = zJupiter^ℓ765 -^ℓ768 zSaturn^ℓ767;
22   distance^ℓ788 = sqrt(dx^ℓ771 *^ℓ774 dx^ℓ773
23   +^ℓ780 dy^ℓ776 *^ℓ779 dy^ℓ778 +^ℓ786 dz^ℓ782
24   *^ℓ785 dz^ℓ784)^ℓ787;
25   mag^ℓ800 = dt^ℓ790 /^ℓ799 distance^ℓ792 *^ℓ795
26   distance^ℓ794 *^ℓ798 distance^ℓ797;
27   vxJupiter^ℓ812 = vxJupiter^ℓ802 -^ℓ811
28   dx^ℓ804 *^ℓ807 massSaturn^ℓ806 *^ℓ810 mag^ℓ809;
29               [...]
30   vxSaturn^ℓ848 = vxSaturn^ℓ838 +^ℓ847 dx^ℓ840
31   *^ℓ843 massJupiter^ℓ842 *^ℓ846 mag^ℓ845;
32               [...]
33   xJupiter^ℓ2602 = xJupiter^ℓ2595 +^ℓ2601
34    dt^ℓ2597 *^ℓ2600 vxJupiter^ℓ2599;
35   xSaturn^ℓ2683 = xSaturn^ℓ2676 +^ℓ2682 dt^ℓ2678
36    *^ℓ2681 vxSaturn^ℓ2680;
37               [...]
38   t^ℓ2707 = t^ℓ2703 +^ℓ2706 dt^ℓ2705;} ;
39 require_nsb(xJupiter, 11)^ℓ2710;
40 require_nsb(xSaturn, 11)^ℓ2716;
41               [...]
```

```
1  days_per_year|56| = 365.24|56|;
2  dt|56| = 0.01|56|;
3  t|54| = 0.0|54|;
4  t_max|53| = 1000.0|53|;
5                [...]
6  xJupiter|59| = 4.8414316|59|;
7  vxJupiter|61| = 0.0016600767|61|
8  *|61| days_per_year|61|;
9  massJupiter|55| = 9.5479196E-4|55|
10 *|55| solar_mass|55|;
11 xSaturn|58| = 8.343367|58|;
12               [...]
13 vxSaturn|61| = -0.002767425|61|
14 *|61| days_per_year|61|;
15 massSaturn|53| = 2.8588597E-4|53|
16 *|53| solar_mass|53|;
17               [...]
18 while (t < t_max) {
19   dx|46| = xJupiter|46| -|46| xSaturn|47|;
20   dy|45| = yJupiter|45| -|45| ySaturn|46|;
21   dz|44| = zJupiter|44| -|44| zSaturn|45|;
22   distance|44| = sqrt(dx|46| *|46|
23   dx|46| +|45| dy|45| *|45|
24   dy|45| +|44| dz|35| *|35| dz|35|)|44|;
25   mag|44| = dt|44| /|44| distance|44| *|44|
26   distance|44| *|44| distance|44|;
27   vxJupiter|58| = vxJupiter|59| -|58|
28   dx|41| *|41| massSaturn|41| *|41| mag|41|;
29               [...]
30   vxSaturn|59| = vxSaturn|60| +|59|
31   dx|43| *|43| massJupiter|43| *|43| mag|43|;
32               [...]
33   xJupiter|53| = xJupiter|54| +|53|
34   dt|47| *|47| vxJupiter|47|;
35   xSaturn|53| = xSaturn|54| +|53|
36   dt|46| *|46| vxSaturn|46|;
37               [...]
38   t|53| = t|54| +|53| dt|38|;};
39 require_nsb(xJupiter, 11);
40 require_nsb(xSaturn, 11);
41               [...]
```

Fig. 2. Left: source program annotated with labels. Right: program with POP generated data types with ILP formulation.

the orbits of planets in the solar system interacting with each other gravitationally as shown in Fig. 1 (note that to for the sake of clarity of the graphic, Fig. 1 uses different simulation times for each body.)

We present, in Fig. 2, excerpts of code that measure the distance between the two planets `Jupiter` and `Saturn`. We assume that each body has its own mass (e.g. `massJupiter`, `massSaturn`), position (e.g. [`xJupiter`, `yJupiter`, `zJupiter`], [`xSaturn`, `ySaturn`, `zSaturn`]) and velocity (e.g. [`vxJupiter`, `vyJupiter`, `zyJupiter`], [`vxSaturn`, `vySaturn`, `vzSaturn`]). Moreover, we suppose that all variables, before POP analysis, are in double precision and that a range determination is performed by dynamic analysis on the program variables

(we plan to use a static analyzer in the future). POP assigns to each node of the program's syntactic tree a unique control point in order to determine easily the number of significant bits of the result as mentioned in the left hand side corner of Fig. 2. Some notations can be stressed about the structure of POP code. We annotate each variable with its unique control point as we can observe in the left hand side program of Fig. 2, e.g. $\texttt{xJupiter}^{\ell_{39}} = \texttt{4.8414316}^{\ell_{38}}$ denotes that the variable `xJupiter` has the control point ℓ_{39} and assigned to the value 4.8414316 at control point ℓ_{38}. Considering that `nsb` denotes the number of significant bits of the variables, the statements $\texttt{require_nsb(xJupiter,11)}^{\ell_{2710}}$ and $\texttt{require_nsb(xSaturn,11)}^{\ell_{2716}}$ on the last two lines of the code inform the system that POP user wants to have 11 accurate binary digits on variables `xJupiter` and `xSaturn` at their control points ℓ_{2710} and ℓ_{2716}, respectively. Note that a result has n significants if the relative error between the exact and approximated results is less than 2^n.

The key feature of our approach is to generate a set of constraints for each statement of our program (more details in Sect. 4). In other words, the accuracy of the arithmetic expressions assigned to variables is determined by semantic equations, in function of the accuracy of the operands. Consider the program of the right hand side of Fig. 2. We display the POP output N-body program coupled with the generated data types. For a user requirements of 11 bits on variables `xJupiter` and `xSaturn`, POP tunes successfully a large part of the variables of the program (number of constraints solved by GLPK $\simeq$ 3160 with 2468 variables). For instance, the result of the measured distance between *Jupiter* and *Saturn*, on line 22 of the right hand side code of Fig. 2, is computed with 44 bits at bit level. Note that the full code contains other `nsb` requirements for the other bodies. The `nsb` given in the right hand side of Fig. 2 are greater than the `nsb` required on the final results since they have been skewed to ensure the precision of the whole code at any iteration. Let us also note that even if the computed `nsb` do not correspond to IEEE74 formats [1], one may either take the IEEE754 format immediately above the computed `nsb` or choose a multiple precision library such as MPFR [11] or POSIT [25] (we will discuss more about this point later in this article).

In the next section, we detail the ILP and the PI formulations of the precision tuning problem implemented in POP. Also, we present the nature of the constraints generated for the N-body problem and consequently the new data types already discussed.

4 Overview of POP

POP has been extended in several ways since its first introduction in [2–5]. It supports the four elementary operations, trigonometric functions, the square root function, loops, conditionals and arrays. Originally, the analysis in POP were expressed as a set of first order logical propositions among relations between linear integer expressions. Next, these constraints are checked by the Z3 SMT solver [20] in order to return a solution with a certain weight expressing the

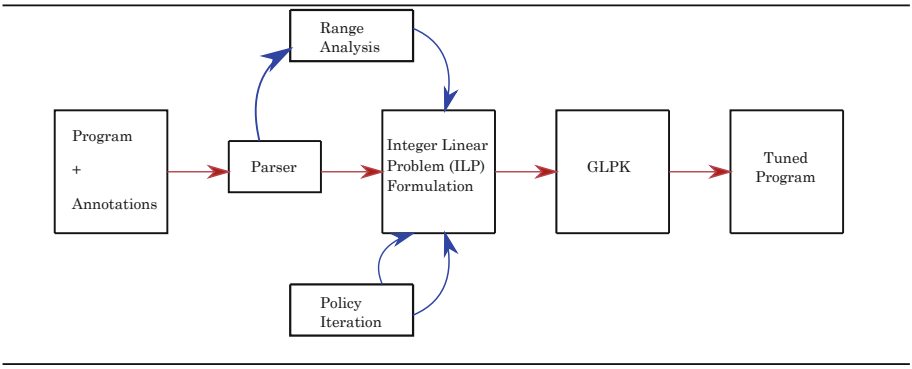


Fig. 3. POP overview.

number of significant bits of the variables. In the most recent version of POP, the intuition is to use no longer the non-optimizing Z3 SMT solver coupled to a binary search. By that means, we reduce the problem of determining the lowest precision on variables and intermediary values in programs to an Integer Linear Problem (ILP) which can be optimally solved in one breath by a classical linear programming solver as depicted in Fig. 3.

Our technique is independent of a particular computer arithmetic. In practice, we handle numbers for which we know their unit in the first place ufp and their number of significant bits nsb defined as follows. Recall that nsb stands for the number of significant bits of a number. The ufp of a number x is given in Eq. (1). This function is used further to describe the way roundoff errors are propagated across computations.

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

Noting that the ufp of the variable values are pre-computed by a prior range analysis. More precisely, the current version of POP performs a dynamic analysis giving an under-approximation of the ranges. Our precision tuning is sensible to the ufp of the values. In other words, it is sensible to the order of magnitude of the ranges but not to the exact values. For example, we will obtain the same tuning with the ranges $[3.4, 6.1]$ and $[2.5, 7.8]$. But, obviously we get a worst tuning if we use a much larger interval, e.g. $[0.0, 1000.0]$. In particular, the efficiency of our techniques for loops depends on the precision of the range analysis for loops.

Integer Linear Problem Formulation. In order to explain the obtained data types of our N-body program already illustrated in the right hand corner of Fig. 2, we present the system of constraints that corresponds to a pure ILP formulation as shown in Eq. (2). For the sake of conciseness, we will focus on lines 22 to 24 that measure the distance between the `Jupiter` and `Saturn` bodies (tuned program in the right hand corner of Fig. 2). To make it easier to follow our reasoning, we rewrite hereafter the statement under discussion annotated with the control points.

$$\begin{aligned}\texttt{distance}^{\ell_{788}} = \texttt{sqrt}(\ & \texttt{dx}^{\ell_{771}} \times^{\ell_{774}} \texttt{dx}^{\ell_{773}} +^{\ell_{780}} \texttt{dy}^{\ell_{776}} \times^{\ell_{779}} \texttt{dy}^{\ell_{778}} \\ & +^{\ell_{786}} \texttt{dz}^{\ell_{782}} \times^{\ell_{785}} \texttt{dz}^{\ell_{784}})^{\ell_{787}}\end{aligned}$$

$$C_1 = \left\{ \begin{array}{l} \mathsf{nsb}(\ell_{780}) \geq \mathsf{nsb}(\ell_{786}) + (-7) + \xi(\ell_{786})(\ell_{780}, \ell_{785}) - 7, \\ \mathsf{nsb}(\ell_{786}) \geq \mathsf{nsb}(\ell_{787}) + (-1) + 0, \\ \mathsf{nsb}(\ell_{774}) \geq \mathsf{nsb}(\ell_{780}) + 7 + \xi(\ell_{780})(\ell_{774}, \ell_{779}) - 7, \\ \mathsf{nsb}(\ell_{779}) \geq \mathsf{nsb}(\ell_{780}) + 6 + \xi(\ell_{780})(\ell_{774}, \ell_{779}) - 7, \\ \mathsf{nsb}(\ell_{785}) \geq \mathsf{nsb}(\ell_{786}) + (-3) + \xi(\ell_{786})(\ell_{780}, \ell_{785}) - 7, \\ \mathsf{nsb}(\ell_{787}) \geq \mathsf{nsb}(\ell_{788}), \mathsf{nsb}(\ell_{771}) \geq \mathsf{nsb}(\ell_{774}) + \xi(\ell_{774})(\ell_{771}, \ell_{773}) - 1, \\ \ldots \\ \xi(\ell_{780})(\ell_{774}, \ell_{779}) \geq 1, \quad \xi(\ell_{786})(\ell_{780}, \ell_{785}) \geq 1 \\ \xi(\ell_{774})(\ell_{771}, \ell_{773}) \geq 1, \quad \xi(\ell_{779})(\ell_{776}, \ell_{778}) \geq 1 \\ \xi(\ell_{785})(\ell_{782}, \ell_{784}) \geq 1 \end{array} \right\} \quad (2)$$

For this statement, POP generates 17 constraints as shown in system C_1 of Eq. (2). We assign to each control point (here 771 to 787) the integer variable nsb which are determined by solving the system C_1. The first two constraints of Eq. (2) are relative to the nsb of the additions stored at control points ℓ_{780} and ℓ_{786} respectively. The numbers computed corresponds to the ufp of the variable values e.g. $\mathsf{ufp}(\ell_{780}) = -1$. The following constraints that compute respectively $\mathsf{nsb}(\ell_{774})$, $\mathsf{nsb}(\ell_{779})$ and $\mathsf{nsb}(\ell_{785})$ are generated for the multiplication. The constraint $\mathsf{nsb}(\ell_{787}) \geq \mathsf{nsb}(\ell_{788})$ is for the square root function. Moreover, the constraint generated for $\mathsf{nsb}(\ell_{771})$ is relative to variable dx (same reasoning for variables dy and dz on their control points). Note that POP generates such constraints for all the statements of the N-body program.

Let us now focus on the last five constraints of system C_1. We introduce a constant function ξ. In the present ILP of Eq. (2), we assume that ξ is a constant function equal to 1. This function corresponds to the carry bit that can be propagated at each operation which is expressed by adding a supplementary bit on the elementary operation result. For instance, the constraint $\xi(\ell_{780})(\ell_{774}, \ell_{779}) \geq 1$ indicates that a carry bit is propagated on the result of the addition stored at control point ℓ_{780} which is correct but pessimistic (we highlight more the utility of ξ in the next paragraph). Finally, for a user accuracy requirement of 11 bits as displayed in Fig. 2. POP calls the GLPK [17] solver and consequently finds the least precision needed for all the N-body problem variables as we can observe hereafter (14636 is the total number of bits of the whole program after POP optimization).

$$\begin{aligned}\texttt{distance}|44| = \texttt{sqrt}(\ & \texttt{dx}|46| * |46|\texttt{dx}|46| + |45|\texttt{dy}|45| * |45|\texttt{dy}|45| \\ & +|44|\texttt{dz}|35| * |35|\texttt{dz}|35|)\end{aligned}$$

Policy Iteration to Refine Carry Bit Propagation. In the ILP formulation in the above paragraph, we have over-approximated the carry bit function by $\xi = 1$. In contrast, this function becomes very costly in large codes if we perform a lot of computations and therefore the errors would be considerable. For example,

if two operands and their errors do not overlap then adding a carry bit is useless. In what follows, we propose an optimization to use a more precise ξ function. Accordingly, when we model this optimization the problem will not remain an ILP any longer, with min and max operators that arise, as shown in the refined system of constraints C_2 of Eq. (3). Thus, we use the policy iteration method [7] to find an optimal solution.

$$C_2 = \left\{ \begin{array}{l} \mathsf{nsb_e}(\ell_{780}) \geq \mathsf{nsb_e}(\ell_{774}), \\ \mathsf{nsb_e}(\ell_{780}) \geq \mathsf{nsb_e}(\ell_{779}), \\ \mathsf{nsb}(\ell_{780}) \geq 7 - 6 + \mathsf{nsb}(\ell_{779}) - \mathsf{nsb}(\ell_{774}) + \mathsf{nsb_e}(\ell_{779}) + \xi(\ell_{780}, \ell_{774}, \ell_{779}), \\ \mathsf{nsb_e}(\ell_{780}) \geq 6 - 7 + \mathsf{nsb}(\ell_{774}) - \mathsf{nsb}(\ell_{779}) + \mathsf{nsb_e}(\ell_{774}) + \xi(\ell_{780}, \ell_{774}, \ell_{779}), \\ \mathsf{nsb_e}(\ell_{786}) \geq \mathsf{nsb_e}(\ell_{780}), \\ \mathsf{nsb_e}(\ell_{786}) \geq \mathsf{nsb_e}(\ell_{785}), \\ \mathsf{nsb}(\ell_{786}) \geq 7 - (-3) + \mathsf{nsb}(\ell_{785}) - \mathsf{nsb}(\ell_{780}) + \mathsf{nsb_e}(\ell_{785}) + \xi(\ell_{786}, \ell_{780}, \ell_{785}), \\ \mathsf{nsb_e}(\ell_{786}) \geq 3 - 7 + \mathsf{nsb}(\ell_{780}) - \mathsf{nsb}(\ell_{785}) + \mathsf{nsb_e}(\ell_{780}) + \xi(\ell_{786}, \ell_{780}, \ell_{785}), \\ \mathsf{nsb_e}(\ell_{774}) \geq \mathsf{nsb}(\ell_{771}) + \mathsf{nsb_e}(\ell_{771}) + \mathsf{nsb_e}(\ell_{773}) - 2, \\ \ldots \\ \xi(\ell_{780})(\ell_{774}, \ell_{779}) = \min \left(\begin{array}{l} \max\big(6 - 7 + \mathsf{nsb}(\ell_{774}) + \mathsf{nsb_e}(\ell_{774}), 0\big), \\ \max\big(7 - 6 + \mathsf{nsb}(\ell_{779}) + \mathsf{nsb_e}(\ell_{779}), 0\big), 1 \end{array} \right) \\ \xi(\ell_{786})(\ell_{780}, \ell_{785}) = \min \left(\begin{array}{l} \max\big(-3 - 7 + \mathsf{nsb}(\ell_{780}) + \mathsf{nsb_e}(\ell_{780}), 0\big), \\ \max\big(7 - (-3) + \mathsf{nsb}(\ell_{785}) + \mathsf{nsb_e}(\ell_{785}), 0\big), 1 \end{array} \right) \end{array} \right\} \tag{3}$$

Equation (3) displays the new constraints that we add to the global system of constraints in the case where we optimize the carry bit of the elementary operations. Before introducing these constraints, we define in Eq. (4) the unit in the last place ulp of a number x.

$$\mathsf{ulp}(x) = \mathsf{ufp}(x) - \mathsf{nsb}(x) + 1 \ . \tag{4}$$

The principle of the new ξ function is as follows: if the ulp of one of the two operands (or errors) is greater than the ufp (see Eq. (1)) of the other one (or conversely) then the two numbers are not aligned and no carry bit can be propagated through the operation (otherwise $\xi = 1$). Not surprisingly, our new system of constraint C_2 introduces a new integer quantity $\mathsf{nsb_e}$ which corresponds to the number of significant bits of the error which needs to be estimated. Formerly, let a number x, we define $\mathsf{ufp_e}(x)$ and $\mathsf{ulp_e}(x)$ as the unit in the first place and in the last place respectively of the error on x. From Eqs. (1) and (4), we have $\mathsf{ufp_e}(x) = \mathsf{ufp}(x) - \mathsf{nsb}(x)$ and $\mathsf{ulp_e}(x) = \mathsf{ufp_e}(x) - \mathsf{nsb_e}(x) + 1$ and consequently we can compute $\mathsf{nsb_e}(x)$.

In practice, policy iteration makes it possible to break the min in the $\xi(\ell_{780})(\ell_{774}, \ell_{779})$ and $\xi(\ell_{786})(\ell_{780}, \ell_{785})$ functions of the two additions as shown in Eq. (3) by choosing the max between the terms. Next, it becomes possible to solve the corresponding ILP. If no fixed point is reached, POP iterates until a solution is found. By applying this optimization, the new data types of the statement of lines 22 to 24 in Fig. 2 are given as follows.

$$\begin{array}{l} \texttt{distance}|41| = \texttt{sqrt}(\ \texttt{dx}|42| * |42|\texttt{dx}|42| + |42|\texttt{dy}|42| * |42|\texttt{dy}|42| \\ \qquad\qquad + |41|\texttt{dz}|31| * |31|\texttt{dz}|31|) \end{array}$$

By comparing with the formats already presented with the ILP method, it is obvious the gain of precision that we obtain on each variable and operation of

Table 1. Distances between the exact position (computed with 500 bits) and the position computed with n bits. Distances given for each body after 10 and 30 years of simulation. Followed by POP analysis time and the execution time of the MPFR generated code.

nsb	11	18	24	34	43	53
Simulation time: 10 years						
Jupiter	$5.542 \cdot 10^{-4}$	$1.650 \cdot 10^{-6}$	$1.577 \cdot 10^{-7}$	$4.998 \cdot 10^{-10}$	$5.077 \cdot 10^{-10}$	$5.076 \cdot 10^{-10}$
Saturn	$1.571 \cdot 10^{-3}$	$2.111 \cdot 10^{-5}$	$1.326 \cdot 10^{-7}$	$4.427 \cdot 10^{-10}$	$3.119 \cdot 10^{-10}$	$3.117 \cdot 10^{-10}$
Uranus	$2.952 \cdot 10^{-3}$	$2.364 \cdot 10^{-5}$	$1.140 \cdot 10^{-7}$	$3.072 \cdot 10^{-10}$	$7.212 \cdot 10^{-11}$	$7.236 \cdot 10^{-11}$
Neptune	$2.360 \cdot 10^{-3}$	$3.807 \cdot 10^{-5}$	$2.206 \cdot 10^{-7}$	$5.578 \cdot 10^{-10}$	$1.751 \cdot 10^{-10}$	$1.757 \cdot 10^{-10}$
Runtime	2'59	2'52	2'57	2'56	3'10	2'59
POP Time	25"	22"	22"	24"	23"	24"
Simulation time: 30 years						
Jupiter	$7.851 \cdot 10^{-4}$	$1.282 \cdot 10^{-5}$	$3.194 \cdot 10^{-8}$	$1.066 \cdot 10^{-8}$	$1.064 \cdot 10^{-8}$	$1.064 \cdot 10^{-8}$
Saturn	$3.009 \cdot 10^{-3}$	$1.934 \cdot 10^{-5}$	$2.694 \cdot 10^{-7}$	$1.7477 \cdot 10^{-8}$	$1.777 \cdot 10^{-8}$	$1.777 \cdot 10^{-8}$
Uranus	$6.839 \cdot 10^{-4}$	$6.132 \cdot 10^{-5}$	$8.901 \cdot 10^{-7}$	$5.105 \cdot 10^{-10}$	$1.464 \cdot 10^{-10}$	$1.457 \cdot 10^{-10}$
Neptune	$2.971 \cdot 10^{-3}$	$2.0227 \cdot 10^{-5}$	$2.469 \cdot 10^{-7}$	$3.869 \cdot 10^{-10}$	$4.775 \cdot 10^{-10}$	$4.779 \cdot 10^{-10}$
Runtime	2'39	2'45	2'43	2'56	2'48	2'40
POP Time	38"	39"	41"	37"	37"	37"

this statement. With the PI method, the total number of bits of the optimized N-body program is $\simeq$ 14335 (a gain of more than 300 bits compared to the ILP formulation). In term of complexity, for both ILP and PI methods, POP generates a linear number of constraints and variables in the size of the analyzed program and finds the best tuning of the variables in polynomial time.

5 Experimental Results

In this section, our goal is to evaluate the performances of POP in tuning the code simulating the behaviour of the different bodies of our example. We note that the N-body program has been excerpted (not fully) from [10] which relies on a second order differential equation solved by Euler's method. Now, we shed some light on the POP tool outline already depicted in Fig. 3. POP has been developed in JAVA. It uses the ANLTR v4.7.1[1] framework to parse the different input programs. As mentioned in Sect. 4, we reduce the precision tuning problem to an ILP by generating a set of semantical equations which can be solved by a linear solver. The integer solution to this problem, computed in polynomial time by a (real) linear programming solver, we use GLPK v4.65 [17], gives the optimal data types at the bit level.

[1] https://www.antlr.org/.

```
xJupiter = mpfr(4.841431617736816,59)
yJupiter = mpfr(-1.1603200435638428,60)
zJupiter = mpfr(-0.10362204164266586,57)
vxJupiter = mpfr(mpfr(0.001660076668485999,61)*mpfr(days_per_year,61)
    ,61)
vyJupiter = mpfr(mpfr(0.007699011359363794,61)*mpfr(days_per_year,61)
    ,61)
vzJupiter = mpfr(mpfr(-6.904600013513118E-5,61)*mpfr(days_per_year,61)
    ,61)
massJupiter = mpfr(mpfr(9.547919617034495E-4,55)*mpfr(solar_mass,55),55)
[...]
while( t<t_max):
    dx = mpfr(mpfr(xSun,57)-mpfr(xJupiter,59),58)
    dy = mpfr(mpfr(ySun,60)-mpfr(yJupiter,60),57)
    dz = mpfr(mpfr(zSun,60)-mpfr(zJupiter,57),55)
    distance = gmpy2.sqrt(mpfr(mpfr(mpfr(mpfr(dx,58)*mpfr(dx,58),58)
+mpfr(mpfr(dy,57)*mpfr(dy,57),57),57)+mpfr(mpfr(dz,46)
                *mpfr(dz,46),46),56))
    mag = mpfr(mpfr(dt,56)/mpfr(mpfr(mpfr(distance,56)
*mpfr(distance,56),56)*mpfr(distance,56),56),56)
    vxJupiter = mpfr(mpfr(vxJupiter,61)+mpfr(mpfr(mpfr(dx,56)
*mpfr(massSun,56),56)*mpfr(mag,56),56),60)
    vyJupiter = mpfr(mpfr(vyJupiter,61)+mpfr(mpfr(mpfr(dy,54)
*mpfr(massSun,54),54)*mpfr(mag,54),54),60)
    vzJupiter = mpfr(mpfr(vzJupiter,61)+mpfr(mpfr(mpfr(dz,55)
*mpfr(massSun,55),55)*mpfr(mag,55),55),60)
    [...]
    xJupiter = mpfr(mpfr(xJupiter,54)+mpfr(mpfr(dt,47)
                *mpfr(vxJupiter,47),47),53)
    yJupiter = mpfr(mpfr(yJupiter,54)+mpfr(mpfr(dt,47)
                *mpfr(vyJupiter,47),47),53)
    zJupiter = mpfr(mpfr(zJupiter,54)+mpfr(mpfr(dt,47)
                *mpfr(vzJupiter,47),47),53)
    [...]
}
```

Fig. 4. Python MPFR code automatically generated by POP for the N-body problem for a nsb requirement of 18 bits on the positions of the planets at the end of the simulation.

We ran our precision tuning analysis on the N-body problem with different nsb requirements on the program variables: 11, 18, 24, 34, 43 and 53 bits. This shows the ability of POP to tune programs in function of the IEEE754 formats (11, 24, 53) [1] as well as for arbitrary word length which can be encoded using libraries such as MPFR [11] or POSIT [25]. We test the efficiency of POP analysis in several ways. The experiments shown in Table 1 seek to measure the distances between the exact position of each of the bodies of our planetary system and the position computed with an nsb of 11, 18, 24, 34, 43 and 53 bits. The distances presented in Table 1 are given for a single position on the planets which follow the orbits previously presented in Fig. 1. The positions are taken after of 10 and 30 years of simulation time.

More precisely, for this experimentation, we generate the N-body program with all computations done on 500 bits (we assume that this gives the exact solution) and we also generate by the same manner an MPFR [11] code with the optimized data types returned by POP. For example, as we can observe in Table 1, for an nsb = 11, the distance measured for `Jupiter` is of the order

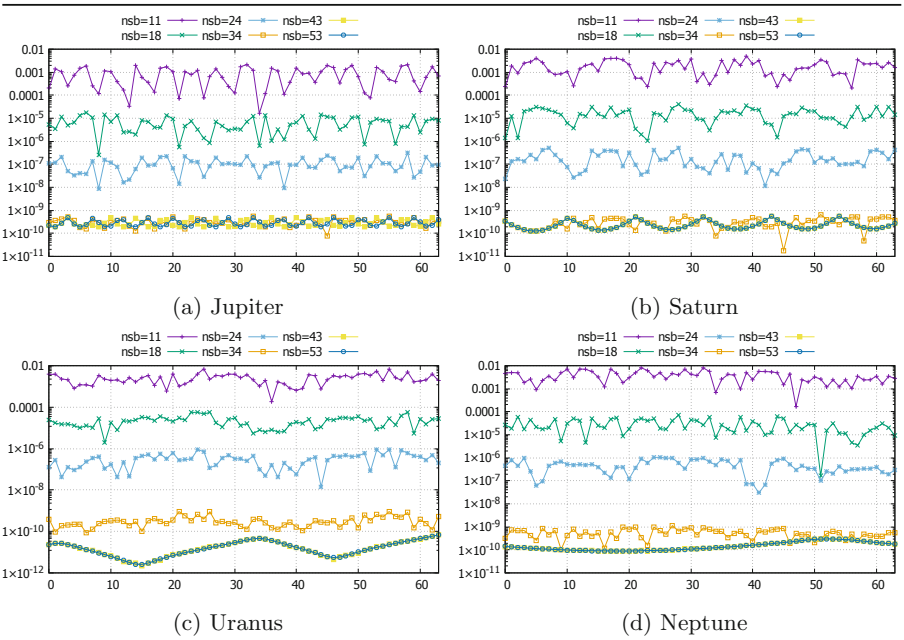


Fig. 5. Distance between the exact and the computed position for the 5 bodies with 11, 18, 24, 34, 43 and 53 bits.

of 10^{-4} for 10 years of simulation which confirms the usefulness of our analysis: desirable results (also for the remaining planets) that respects the user nsb requirement where the worst error is of 2^{-11} for nsb = 11. For a simulation of 10 and 30 years, the runtime spent to measure these distances reaches maximally 2 minutes 59 seconds for an nsb = 53. Concerning the POP time, our analysis took as little as 25 seconds (nsb = 11) to find that we can lower the precision of the majority of variables of the N-body program for a simulation time of 10 years and does not exceed 41 seconds for a simulation time of 30 years (nsb = 24). With this speed, we believe that for large codes POP achieves its best tuning in a minimal time. Figure 4 depicts the capability of POP to generate automatically a Python MPFR version of the N-body program on the position of the planets at the end of the simulation. The MPFR code is annotated with the optimized formats returned by POP after analysis for nsb = 18. In the future, we plan to also generate code for libraries based on the POSIT number system[2].

We end this section by focusing on the curves of Fig. 5. For this experiment, we plot the distance between the exact and the computed position for each body at each instant of the simulation. This extends the results of Table 1 to all instants and not to specific ones. Consequently, we deduce from these observations that the measured error is controlled for the different planets at each iteration of the simulation.

[2] https://github.com/stillwater-sc/universal.

6 Concluding Remarks

The primary goal of our work was to provide a new approach for mixed-precision tuning, totally different from the existing ones. The novelty of our technique is to propose a semantical modelling of the propagation of the numerical errors throughout the code expressed as a set of constraints. We have defined two variants of methods. The first one corresponds to a pure ILP with an over-approximation of the carries in the elementary operations. The second one aims to use a more precise carry bit function and is solved by the policy iteration technique [7]. Both two methods have been implemented in our tool POP. We believe this static analysis performed by our automated tool is unique. The effectiveness of POP has already been demonstrated on a variety of programs coming from different fields.

In this article, we have shown that POP is able to tune the N-body program according to different number of significant bit required by the user. The results presented are promising in term of the analysis technique, speed and efficiency. The only limitation we can face is the size of the problem accepted by the solver. In addition, we have also shown that POP is able to generate code for multiple precision libraries, MPFR in practice, and we plan to integrate POSIT libraries in the near future.

Broadly speaking, our important future directions include handling Deep Neural Network's (DNNs) for which saving resources is essential. Also, code synthesis for the fixed-point arithmetic and assigning the same precision to pieces of code are perspectives we aim at explore at short term.

References

1. ANSI/IEEE: IEEE Standard for Binary Floating-point Arithmetic, STD 754-2008 edn. (2008)
2. Ben Khalifa, D., Martel, M.: Precision tuning and internet of things. In: International Conference on Internet of Things, Embedded Systems and Communications, IINTEC 2019, pp. 80–85. IEEE (2019)
3. Ben Khalifa, D., Martel, M.: Precision tuning of an accelerometer-based pedometer algorithm for IoT devices. In: IEEE International Conference on Internet of Things and Intelligence System, IoTaIS 2020, Bali, Indonesia, 27–28 January 2021, pp. 116–122. IEEE (2020)
4. Ben Khalifa, D., Martel, M.: An evaluation of pop performance for tuning numerical programs in floating-point arithmetic. In: International Conference on Information and Computer Technologies, ICICT 2021. IEEE (2021)
5. Ben Khalifa, D., Martel, M., Adjé, A.: POP: a tuning assistant for mixed-precision floating-point computations. In: Hasan, O., Mallet, F. (eds.) FTSCS 2019. CCIS, vol. 1165, pp. 77–94. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-46902-3_5
6. Chiang, W., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., Rakamaric, Z.: Rigorous floating-point mixed-precision tuning. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL, pp. 300–315. ACM (2017)

7. Costan, A., Gaubert, S., Goubault, E., Martel, M., Putot, S.: A policy iteration algorithm for computing fixed points in static analysis of programs. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 462–475. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_46
8. Damouche, N., Martel, M.: Mixed precision tuning with salsa. In: Proceedings of the 8th International Joint Conference on Pervasive and Embedded Computing and Communication Systems, PECCS 2018, Porto, Portugal, 29–30 July 2018, pp. 185–194. SciTePress (2018)
9. Darulova, E., Horn, E., Sharma, S.: Sound mixed-precision optimization with rewriting. In: Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS, pp. 208–219. IEEE Computer Society/ACM (2018)
10. Demeure, N.: Compromis entre précision et performance dans le calcul haute performance. Ph.D. thesis, Université Paris-Saclay, January 2021
11. Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P.: MPFR: a multiple-precision binary floating-point library with correct rounding. ACM Trans. Math. Softw. **33** (2007)
12. Gardarsson, M., Kjartan, K.: Some theoretical and numerical aspects of the n-body problem (2013, student Paper)
13. Guo, H., Rubio-González, C.: Exploiting community structure for floating-point precision tuning. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, pp. 333–343. ACM (2018)
14. Kotipalli, P.V., Singh, R., Wood, P., Laguna, I., Bagchi, S.: AMPT-GA: automatic mixed precision floating point tuning for GPU applications. In: Proceedings of the ACM International Conference on Supercomputing, ICS, pp. 160–170. ACM (2019)
15. Lam, M.O., Hollingsworth, J.K., de Supinski, B.R., LeGendre, M.P.: Automatically adapting programs for mixed-precision floating-point computation. In: International Conference on Supercomputing, ICS 2013, pp. 369–378. ACM (2013)
16. Lam, M.O., Vanderbruggen, T., Menon, H., Schordan, M.: Tool integration for source-level mixed precision. In: 2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness), pp. 27–35 (2019)
17. Makhorin, A.O.: GLPK (GNU linear programming kit). http://www.gnu.org/software/glpk/glpk.html
18. Makino, J., Kokubo, E., Fukushige, T.: Performance evaluation and tuning of GRAPE-6 - towards 40 "real" Tflops. In: Proceedings of the ACM/IEEE SC2003 Conference on High Performance Networking and Computing, Phoenix, AZ, USA, 15–21 November 2003, CD-Rom, p. 2. ACM (2003)
19. Menon, H., et al.: Adapt: algorithmic differentiation applied to floating-point precision tuning. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018. IEEE Press (2018)
20. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
21. Rubio-González, C., et al.: Precimonious: tuning assistant for floating-point precision. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2013, pp. 27:1–27:12. ACM (2013)
22. Saiki, B., Flatt, O., Nandi, C., Panchekha, P., Tatlock, Z.: Combining precision tuning and rewriting. In: IEEE Symposium on Computer Arithmetic (ARITH) 2021. IEEE (2021)

23. Schkufza, E., Sharma, R., Aiken, A.: Stochastic optimization of floating-point programs with tunable precision. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, pp. 53–64. Association for Computing Machinery (2014)
24. Solovyev, A., Baranowski, M.S., Briggs, I., Jacobsen, C., Rakamarić, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. ACM Trans. Program. Lang. Syst. **41**(1), 1–39 (2018)
25. Uguen, Y., Forget, L., de Dinechin, F.: Evaluating the hardware cost of the posit number system. In: Sourdis, I., Bouganis, C., Álvarez, C., Díaz, L.A.T., Valero-Lara, P., Martorell, X. (eds.) 29th International Conference on Field Programmable Logic and Applications, FPL 2019, Barcelona, Spain, 8–12 September 2019, pp. 106–113. IEEE (2019)