Constrained Precision Tuning

Dorra Ben Khalifa\textsuperscript{1} and Matthieu Martel\textsuperscript{1,2}

Abstract—Precision tuning or customized precision number representations is emerging, in these recent years, as one of the most promising techniques that has a positive impact on the footprint of programs concerning energy consumption, bandwidth usage and computation time of numerical programs. In contrast to the uniform precision, mixed precision tuning assigns different finite-precision types to each variable and arithmetic operation of a program and offers many additional optimization opportunities. However, this technique introduces new challenges related to the cost of operations or type conversions which can overload the program execution after tuning. In this article, we extend our tool POP (Precision OPtimizer), with efficient ways to limit the number of drawbacks of mixed precision and to achieve best compromise between performance and memory consumption. On a popular set of tests from the FPBench suite, we discuss the results obtained by POP.

Index Terms—Floating-point arithmetic, numerical accuracy, static analysis, code optimization.

I. INTRODUCTION

In recent years, precision tuning to improve the performance metrics is emerging as a new trend to save the resources on the available processors, especially when new error-tolerant applications are considered [1]. By way of illustration, many applications can tolerate some loss in quality during computation, as in the case of media processing (audio, video and image), data mining, machine learning, etc. In addition, as almost all numerical computations are performed using floating-point operations to represent real numbers [2], the precision of the related data types should be adapted in order to guarantee the desired overall rounding error and to strengthen the performance of programs. For instance, using FP32 single precision formats is often at least twice as fast as the FP64 double precision ones on most modern processors [3]. Consequently, the natural question that arises is how to obtain the best precision/performance trade-off by allocating some program variables in low precision (e.g. FP16 and FP32) and by using high precision (e.g. FP64 and FP128) selectively. This process is also called, mixed-precision tuning.

Let us precise that precision tuning is not a simple task limited to changing the data type in the source code with the Find-and-Replace button of any text editor. It is a more complex technique which analyzes the semantics of the programs and presents several challenges both architectural and algorithmic. For this reason, various tools [4], [5], [6], [7], [8], [9] have been proposed to help developers select the most appropriate data representations. Such tools may integrate different approaches but their common goal is still to automatically or semi-automatically adapt an original code given in higher precision to the selected lower precision type. However, the common point to all the existing techniques is that they follow a trial-and-error strategy: they change the data types of some variables of the program and evaluate the accuracy of the result and depending on what is obtained they change more or less data types and repeat the process. At this level, we present 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 that 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 [10]: a bound on the absolute error $|\hat{x} - x|$ between the represented $\hat{x}$ value and the exact value $x$ that we would have in the exact arithmetic.

The POP tool [11], [12] proposes a novel static technique based on a semantic modelling of the propagation of the numerical errors throughout the code formulated with two methods. The first method consists of generating an Integer Linear Problem (ILP) from the program. Basically, this is done by reasoning on the most significant bit and the number of significant bits of the values which are integer quantities. The integer solution to this problem, computed in polynomial-time by a classical linear programming solver, gives the optimal data types at bit-level. The second method proposes a finer set of semantic equations which does not reduce directly to an ILP problem. So, we use the policy iteration (PI) technique to find a solution. Let us note that the originality of both methods is to find directly the minimal number of bits needed, known as bit-level precision tuning, at each control point of the variables of the program.

In this article, we focus on improving the efficiency of POP by experimenting several optimization criteria to our system of constraints in order to achieve the best compromise between performance and memory saving. The first criteria is related to minimize the number of formats in the tuned programs. The second criteria correponds to minimize the number of bits needed for each operation performed in programs. The third optimization function aims to avoid type conversions of the same variables in the same program.

The remainder of this article is as follows. Section II describes the static approach inside the POP tool. Section III point out the new optimization criteria related to the type...
II. POP: PRECISION OPTIMIZER

In this section, we start by describing the necessary background to understand the technique behind our tool POP (Section II-A). Next, we highlight the main architecture of POP in Section II-B. Section II-C presents a motivating example of tuning with POP.

A. Background

POP manipulates numbers for which we know their unit in the first place denoted by $ufp$ (Definition 1), their number of significant bits denoted by $nsb$ (Definition 2) and their unit in the last place denoted by $ulp$ (Definition 3). We also assume that the constants occurring in the source codes are exact and we bound the errors introduced by the finite precision (Definition 4).

Definition 1 (Unit in the First Place): The unit in the first place of a real number $x$ denoted by $ufp(x)$, and possibly encoded up to some rounding mode by a floating-point or a fixed-point number is given in Equation (1). This function is independent of the representation of $x$:

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

Definition 2 (Number of Significant Bits): Intuitively, $nsb(x)$ is the number of significant bits of $x$. Let $\hat{x}$ the approximation of $x$ in finite precision and let $\epsilon(x) = |x - \hat{x}|$ be the absolute error. Following Parker [13], if $nsb(x) = k$, for $x \neq 0$, then

$$\epsilon(x) \leq 2^{ufp(x)-k+1}. \quad (2)$$

Definition 3 (Unit in the Last Place): The unit in the last place of a number $x$ denoted by $ulp(x)$ is defined below in Equation (3). It depends on the unit in the first place $ufp(x)$ and the number of significant bits $nsb(x)$:

$$ulp(x) = ufp(x) - nsb(x) + 1. \quad (3)$$

For example, if $x = 2.75$ then, following equations (1) to (3) we have $ufp(x) = 1$, $nsb(x) = 4$ and consequently $ulp(x) = -2$. Note that we define in the same manner $ulp_e(x)$ and $nsb_e(x)$ respectively the $ulp$ and $nsb$ of the error on a number $x$ (see Figure 1) which are used to describe the error propagation through the computations and to compute either a carry bit can be propagated through a computation or not. In practice, POP implements an optimized carry bit function, denoted by $\xi$, that adds an extra bit in some arithmetic operation only if the errors of the operands can overlap. Definition 4 highlights this function.

Definition 4 (Carry Bit Function): Let $x$ and $y$ two operands of some operation which result is $z$. The optimized $\xi$ function is given as shown in Equation (4): if the $ulp_e$ of one of the two operands error $\epsilon(x)$ or $\epsilon(y)$ is greater than the $ulp_e$ of the other one (or conversely) then the numbers $x$ and $y$ are not aligned and consequently $\xi = 0$ (otherwise $\xi = 1$). The optimized $\xi$ function is given by

$$\xi(z)(x,y) = \begin{cases} 0 & \text{ulp}_e(x) \geq \text{ulp}_e(y), \\ 0 & \text{ulp}_e(y) \geq \text{ulp}_e(x), \\ 1 & \text{otherwise}. \end{cases} \quad (4)$$

Let us state that the carry bit function has an effect on the linearity of the constraints that we will highlight in the next section.

B. POP Outline

From a grammar, POP parses the input source codes using the ANTLR framework [14]. Next, it performs a range determination consisting in launching the execution of the program a certain number of times in order to determine dynamically the range of variables. More precisely, this is done by reasoning on the weight of the most significant bit already defined in Equation (1).

The new features of POP, in contrast to its former introduction in [15], [16], [17], [18], is to solve an ILP problem generated from the program source code. Next, the optimal solution computed by a classical LP solver (we use GLPK\footnote{https://www.gnu.org/software/glpk/} in practice) gives the optimized data types that satisfy the user accuracy requirement in a polynomial-time and with respect to some optimization function. We remind the reader that the goal of this article is to introduce several optimization functions and to evaluate the performance of tuning in terms of mixed-precision and quality of analysis. However, we must precise that we have over-approximated the carry bit propagation throughout the computations in the ILP formulation because of the min and max operators that arise when solving Equation (4). For this purpose, POP implements an optimization of the previous ILP method by introducing a second set of semantic equations. These new equations make it possible to tune even more the precision by being less pessimistic on the propagation of carries in arithmetic operations. So, we use the policy iteration (PI) method [19] to find efficiently a solution. We refer the reader to [12] for a detailed explanation about our method.

Concerning the complexity of the analysis performed by POP, in practice, the number of variables and constraints is linear in the size of the program and consequently the complexity to analyze a program of size $n$ is equivalent to that of solving a system of $n$ constraints in our language of constraints. In addition, POP is able to analyze large programs ($\approx$ KLOC) but the only limitation is related to the size of the problem accepted by the solver.

C. Motivating Example
To better explain the ILP nature of the precision tuning problem in POP, we consider the PID controller program [20] of Figure 2. This algorithm is widely used in embedded and critical systems e.g. aeronautic and avionic systems. The main feature of this program is to keep a physical parameter \( m \) at a specific value known as the setpoint. In other words, it tries to correct a measure by maintaining it at a defined value. The original PID program is depicted in the left hand side of Figure 2 whereas the optimized program by POP is given in the right hand side corner.

Some points can be highlighted about this example. For instance, the variables \( dt \) and \( invdt \) are both initialized respectively to the value 0.5, annotated with their control points thanks to the following annotations \( dt^{1/2} = 0.5^{1/2} \) and \( invdt^{1/2} = 0.5^{1/2} \) in the left hand side of Figure 2. Also, the statement \( require_{nsb}(m1,12)^{1/2} \) at Line 19 informs POP that the user wants to get 12 significant bits on variable \( m1 \) and so, \( nsb(m1) = 12 \). Next, POP reduces the problem to an ILP formulation by generating a set of constraints on all the labels of programs. Finally, the optimal solution computed by a GLPK solver [21] gives the optimized data types that satisfy the user accuracy requirement in a polynomial-time as shown in the right hand side of Figure 2. By way of illustration, the results obtained on Line 10 of the PID program says that for \( nsb(m1) = 12 \) bits, the number of significant bits needed for variable \( e \) is 12 bits, the variable \( c \) is computed with 14 bits whereas the operator \( − \) is computed with 12 bits, etc.

### III. GUIDING Tuning With Constraints

In this section, we introduce the different cost functions used by POP to obtain the best compromise between performance and memory saving. Recall that POP generates a set of constraints whose solution gives the tuning (See Section [1B]). Our new optimization criteria are then expressed as cost functions that the solver has to optimize. Below we propose three optimization criteria related to the largest data type, the number of bits needed for each operation and the prohibition of type conversions.

Let us remark that one may combine the cost functions introduced in this article and try to optimize the accuracies at any control point globally. While encompassing the cost functions presented in previous work [12] in terms of tuning, this method suffers from several drawbacks. First, it increases the number of constraints and variables of the system. Second, it over-constraints the system which makes the solver fail more often and finally it slows down the tuning time.

Let us also remark that, compared to other approaches such as the ones based on delta-debugging (see Section [V]), POP approach which is based on solving a system of constraints allows one to define easily many optimization criteria such as the ones introduced below without modifying significantly the tool and without increasing the combinatorial complexity of the problem.

#### A. Minimize the Number of the Largest Data Type

The purpose of this cost function is to answer the following question: What is the minimal number of bits of the greatest format needed in the program in order to ensure some accuracy on the result? For example, by this technique one may answer to questions such as is it possible to obtain a result with 18 significant bits using only, e.g. single precision numbers (FP32)?

More formally, let \( Lab \) denotes the set of labels of the program and let \( T : \ Lab \rightarrow \mathbb{N} \) be a tuning assigning to each control point \( \ell \in Lab \) an integer precision. A tuning is correct if it satisfies the system of constraints generated by POP [12] including the accuracy requirement on the result and we denote \( \mathcal{T} \) the set of correct tunings. The cost function for maximal precision \( MP \) that we aim to compute is given as
shown in Equation (5).

\[ MP = \min_{T \in \mathcal{F}} \left\{ \max_{\ell \in \mathcal{L}} T(\ell) \right\} . \]  

Minimizing the largest format may enable one to use a processor with limited formats (e.g. only single precision). In former work, our cost function was to minimize the sum of the precision of the assigned variables in the program, i.e. \( \sum_{\ell \in \mathcal{L}} T(\ell) \). However, this may lead to cases where some variables have large formats and others small ones (e.g. from FP16 half precision to FP64 double) which makes difficult hardware optimizations [15].

B. Minimize of the Operations Number of Bits

Our second cost function \( M\text{Op} \) focuses on the operators instead of the variables of the program. We aim at minimizing only the number of bits used in the arithmetic operations, without considering what is used for variables. The interest is to minimize the hardware needed to run the programs. Also, this optimization is particularly relevant for circuit implementations, e.g. using FPGAs [22].

Formally, let \( \mathcal{O} \subseteq \mathcal{L} \) be the subset of labels attached to operators (additions, multiplications, elementary functions, etc.) Here, we aim at computing

\[ M\text{Op} = \min_{T \in \mathcal{F}} \left\{ \sum_{\ell \in \mathcal{O}} T(\ell) \right\} . \]  

In the present work, we assign the same weight to each operation (i.e. its number of bits). However, it would be interesting to assign different weights, for instance to take into account that a multiplication is more costly than an addition at the hardware level (same for elementary functions.)

C. Avoid Type Conversions

Mixed precision tuning, as done by POP, offers the advantage of optimizing the precision of a variable at each of its occurrences. However, from a performance point of view, this introduces type conversions which may slow down the programs. Let \( \gamma : \mathcal{V} \rightarrow \mathcal{P}(\mathcal{L}) \) be a function mapping each variable \( x \) of a program to the set of labels corresponding to the occurrences of \( x \) and let \( \text{Dom}(\gamma) \) denote the definition domain of \( \gamma \). We add a mode in POP which enforces it to produce an uniform tuning by adding the constraints

\[ \forall x \in \text{Dom}(\gamma), \forall \ell_1, \ell_2 \in \gamma(x), \quad T(\ell_1) = T(\ell_2) . \]  

Let us remark that, in this mode, POP still achieve bit-level precision tuning. However this tuning is uniform and only one precision is returned for each variable which avoids type conversions. As previously mentioned, let us also underline the fact that since POP is based on a system of constraints, assigning to it new optimization objectives can be done easily, without a deep reactoring of the tool.

| Prog. | Require 8 bits | Require 12 bits |
|-------|---------------|---------------|
| Acc.  | 9 18 0 0 | 13 3 15 0 |
| Odo.  | 13 22 7 0 | 17 0 29 0 |
| Pnd.  | 12 17 1 0 | 21 0 13 0 |
| PID   | 10 22 0 0 | 14 0 22 0 |
| Trap. | 14 5 10 0 | 18 0 15 0 |

| Require 16 bits | Require 32 bits |
|-----------------|-----------------|
| Acc.  | 17 0 18 0 | 25 0 15 3 |
| Odo.  | 21 0 29 0 | 37 0 29 0 |
| Pnd.  | 25 0 12 1 | 33 0 10 3 |
| PID   | 20 0 19 0 | 28 0 10 9 |
| Trap. | 18 0 22 0 | 26 0 22 4 |
| PID   | 22 0 15 0 | 30 0 15 12 |

TABLE I

MINIMIZATION OF THE WORST PRECISION IN FUNCTION OF THE ACCURACY REQUIREMENT. MAXIMAL PRECISION (MP) COMPUTED BY POP FOLLOWED BY THE NUMBER OF FP16, FP32 AND FP64 VARIABLES.

IV. EXPERIMENTAL EVALUATION

We evaluate the performance of POP on a standard benchmark set for floating-point analysis (Section IV-A), and compare the results of precision tuning using the different cost functions already defined in Section III. Note that POP and all the data and results presented in this article (Section IV-B), are publicly available under GPL-3.0 License. Also, the experiments are ran on an Intel Core i5-8350U 1.7GHz Linux machine with 8 GB RAM.

Fig. 3. Percentage of reduction after tuning by POP of the total number of bits for the operators occurring in our test programs in function of the accuracy requirement on the result.

https://github.com/benkhelifadorra/POP-v2.0
systems, IoT and numerical analysis fields. We have selected programs from the embedded floating-point benchmarks and for measuring their accuracy [23]. FPBench develops standards for describing benchmarks. A. Benchmarks

Our cost functions are experimented on several FPBench benchmarks. FPBench develops standards for describing floating-point benchmarks and for measuring their accuracy [23]. We have selected programs from the embedded systems, IoT and numerical analysis fields.

- **Accelerometer** [16]: this program comes from the IoT field and measures the angle of inclination of an object.
- **Odometry** [24]: an example taken from robotics which concerns the computation of the position $(x,y)$ of a two wheeled robot by odometry.
- **Pendulum** [12]: models the movement of a simple pendulum without damping.
- **PID Controller** [20]: is a widely used algorithm in embedded and critical systems e.g. aeronautic and avionic systems. This program was highlighted in Section III-C.
- **Runge Kutta method** [25]: is an effective and widely used method for solving the initial-value problems of differential equations.
- **The trapezoidal rule** [25]: a well known algorithm in numerical analysis which approximates the definite integral \(\int_a^b f(x)\,dx\).

B. Evaluation

In this section, we present the results obtained with POP on the sample programs described in Section IV-A for the cost functions introduced in Section III.

Table II presents the results of the cost function concerning the minimization of the largest data type already presented in Section III-A. In this experiment, we consider four accuracy requirements for the results of our test programs: 8, 12, 16 and 32 significant bits. For each program and requirement, we display the maximal precision $MP$ as defined in Equation 5, as well as the number of variable in half, single and double precision (FP16, FP32 and FP64 respectively). For example, for the accelerometer program with an accuracy requirement of 12 bits, we find that $MP = 13$, meaning that 3 variables may be set in half precision and that the remaining 15 variables may be set in single precision. A first observation is that the $MP$ are not intuitive and should be difficult to obtain without POP, either by hand or by tuning tools based on delta-debugging (because of the theoretical complexity which makes the bit-level precision tuning untracktable for this class of tools, see related work in Section V). A second observation is that POP often finds $MP$s close to the accuracy requirement of the result, which means that the optimization is important. Not surprisingly, the $MP$ increases as the required number of significant bits on the result increases.

Our second experiment concerns the cost function of Section III-B related to the size of the arithmetic operators. Our results are summarized in Figure 3. The histogram gives percentages of optimization for the total number of bits needed for the operations. The percentage is computed with respect to an initial number of bits corresponding to all the operations done in FP64 double precision (which corresponds to 100%). The first four bars are for the accelerometer program, the next four bars are for the odometry program, etc. The four bars dedicated to a program correspond to the gains obtained for the accuracy requirements set on the results: 8, 16, 32 and 48 bits. For example, for the accelerometer program, the total number of bits for the operations is reduced by 85% and 70% for requirements of 8 and 16 bits respectively.

A first observation is that POP is able to reduce very significantly the number of bits needed when small nsb are required on the outputs (around 80% and 70% of reduction for nsb = 8 or nsb = 16 bits). Also, we observe that when large nsb quantities are required (e.g. 48 bits is close to the 53 bits of the double precision), POP is able to reduce the size of the operators by around 10%.

Finally, our last experiment, summed up in Table III is for uniform precision as defined in Section III-C. We consider four accuracy requirements for the results of our test programs: 8, 16, 24 and 32 bits. For each of these requirements, we run POP in mixed-precision mode and then in uniform precision mode. We display the percentages of optimization on the total number of bits used by the variables. Again, 100% corresponds to all the variables in double precision. In Table II two consecutive values are for the same requirement, in mixed and uniform precision respectively. For example, for the odometry program and for a requirement of 8 bits, the savings in number of bits obtained by POP are of 86% in mixed precision and of 81% in uniform precision. Similarly, for the same program and for a requirement of 32 bits, the saving are of 54% and 51% respectively. From Table III we may observe that POP still optimizes well the programs in uniform precision mode. This is mainly due to the fact that our sample programs do not use the same variables many times (see, for example the code of the PID program in Figure 2). A second remark is that bit-level precision tuning makes it possible to obtain important memory savings, even in uniform mode.

V. RELATED WORK

The last few years have seen a wealth of precision tuning tools. In this section, we discuss the strengths and shortcomings of each tool. Besides, we classify these approaches into.

![https://fpbench.org/](https://fpbench.org/)
two categories: static methods that extract additional knowledge from the program source code without executing it with input data and dynamic methods that involve the profiling of the target application to extract pieces of information by running the original version of the program.

A. Static Analysis Tools

Rosa [26] is a source-to-source compiler which takes as input a real-valued program with error specifications and synthesizes code over an appropriate floating-point (FP32, FP64, FP128, and an extended format with 256 bit width) or fixed-point data type (8, 16, 32 bit) which fulfills the specification. Unlike POP which is able to ensure mixed precision tuning on programs containing expressions, loops, conditionals and even arrays, Rosa handles conditional statements soundly and assigns only uniform precision to the variables of their programs. In addition, FPTuner [4] exposes a user-defined threshold for the amount of type casts that the tool may insert into the code. Let us state that the approach deployed by FPTuner is close the static technique of POP, especially in the constraint generation step. However, it relies on a local optimization procedure by solving quadratic problems for a given set of candidate data types. Contrarily to POP, FPTuner is limited to straight-line programs. Moreover, the TAFFO tool [27] is a LLVM-based tool-chain. Its strategy is to collect statically annotations from the source code and it converts them into LLVM-IR metadata with the goal to replace floating-point operations with fixed-point operations to the extent possible. This analysis is used to project on the output the error introduced by each fixed-point instruction. In contrast to TAFFO, POP is able to return solutions at bit-level suitable for the IEEE754 floating-point arithmetic, the fixed-point arithmetic and the MPFR library for non-standard precision. Nevertheless, the static tool Daisy [5] is able to provide a mixed precision solution that considers both floating-point and fixed-point data making it generally applicable to both scientific computing and embedded applications. However, Daisy does not address conditional-based programs.

B. Dynamic Analysis Tools

Precimonious [9] is a dynamic automated search-based tool that leverages the LLVM framework to tweak variable declarations to build and prototype mixed-precision configurations within a given error threshold. It is based on the delta-debugging algorithm search [28] which guarantees to find a local 1-minimum if one exists. A configuration is said to be 1-minimal if lowering any additional variable (or function call) leads to a configuration that produces an inaccurate result, or is not faster than the original program. Unlike POP which optimizes all the variables of the program, Precimonious optimizes only the precision of declared variables. It uses external description files (JSON or XML) to declare which variables in the source code should be explored and which data types have to be investigated. Moreover, it estimates round-off errors by dynamically evaluating the program on several random inputs. For this reason, the Blame Analysis technique [29] aims at reducing the space of variables of Precimonious. It performs shadow execution to identify variables that are numerically insensitive and which can consequently be excluded from the search space before tuning. The analysis finds a set of variables that can be in single precision, while the rest of the variables are in double precision. However, the output configurations may or may not improve performance, so to use the analysis in practice one must perform runs of the program to determine which configurations actually improve performance. Another dynamic tool sharing some methodologies of Precimonious is called PROMISE [30]. It modifies automatically the precision of variables taking into account an accuracy requirement on the computed result. Based on the delta-debugging search algorithm which reduces the search space of the possible variables to be converted, it provides a subset of the program variables which can be converted from FP64 to FP32 only. Meanwhile, PROMISE is able to tune programs only in FP32 single precision and it remains a time-intensive tool. HiFPTuner [31] is another extension of Precimonious which uses a hierarchical search approach. It combines a static analysis to create the hierarchical structure in order to minimize the number of type cast operations whereas the dynamic profiling highlights the hottest dependencies. A major limitation is that HiFPTuner’s configurations are dependent on the tuning inputs, and no accuracy guarantee is provided for untested inputs.

In addition, the CRAFT tool [32] is a framework that performs an automated search of a program’s instruction space, determining the level of precision necessary in the result of each instruction to pass a user-provided verification routine assuming all other operations are done in high precision such as FP64 double precision. However, it can be very time consuming even for very small programs. Let us note that there are other tools oriented to GPU applications [33], [34], [35] which combine static analysis for casting-aware performance modeling with dynamic analysis for enforcing precision constraints. For an in-depth description and a theoretical description between these tools, we refer the reader to our survey in [36].

VI. CONCLUSION AND FUTURE WORK

In this article, we have extended our tool POP relying on a modeling of the propagation of the errors throughout the code with new optimization criteria in order to obtain a trade-off between, precision, analysis time and memory consumption. To our knowledge, this is the first work interested in optimizing the results obtained after the precision tuning phase. The results discussed show that our tool succeeded in limiting the number of formats, the number of bits of operations and the number of type conversions between the variables, for the majority of our benchmarks and with respect to the accuracy requirements given by the user. We shed the light that these results are helpful in the hardware level, especially for some processors that are limited to specific formats. In furture work, we aim to adapt our precision tuning tool to generate code in the fixed-point arithmetic. In practice,
the information provided by POP may be used to generate computations in the fixed-point arithmetic with an accuracy guaranty on the results. Also, we are interested in combining POP with other tools performing error analysis [37], [38] and code transformation [20], [39] tasks in order to better improve the accuracy of the programs. Our technique is also generalizable to Deep Neural Networks for which it is important to save memory usage and computational resources.

REFERENCES

[1] S. Cherubin and G. Agosta, “Tools for reduced precision computation: A survey,” ACM Comput. Surv., vol. 53, no. 2, 2020.
[2] IEEE Standard for Binary Floating-Point Arithmetic, Std 754-2008 ed., ANSI/IEEE, 2008.
[3] M. Baboulin, A. Buttari, J. J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, “Accelerating scientific computations with mixed precision algorithms,” Comput. Phys. Commun., vol. 180, no. 12, pp. 2526–2533, 2009.
[4] W. 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, POPL 2017, G. Castagna and A. D. Gordon, Eds. ACM, 2017, pp. 300–315.
[5] E. Darulova, E. Horn, and S. Sharma, “Sound mixed-precision optimization with rewriting,” in Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2018, C. Gilli, B. Sinopoli, X. Liu, and P. Tabuada, Eds. IEEE Computer Society / ACM, 2018, pp. 208–219.
[6] H. Guo and C. Rubio-González, “Exploiting community structure for floating-point precision tuning,” in Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018. ACM, 2018, pp. 333–343.
[7] P. P. Kotipalli, R. Singh, P. Wood, I. Laguna, and S. Bagchi, “AMPTGA: automatic mixed precision floating point tuning for GPU applications,” in Proceedings of the ACM International Conference on Supercomputing, ICS . ACM, 2019, pp. 160–170.
[8] Eds. A. O. Makhorin, “Glpk (gnu linear programming kit),” in Encyclopedia of Operations Research and Management Science, 180, G. Castagna and A. D. Gordon, Eds. Springer, 2005, pp. 69–78.
[9] D. Ben Khalifa and M. Martel, “Precision tuning of an accelerometer-based pedometer algorithm for iot devices,” in International Conference on Internet of Things and Intelligence System, IOTAIS 2020. IEEE, 2020, pp. 113–119.
[10] D. Ben Khalifa and M. Martel, “An evaluation of POP performance for tuning numerical programs in floating-point arithmetic,” in 4th International Conference on Information and Computer Technologies, ICICT 2021, Kahului, HI, USA, March 11-14, 2021. IEEE, 2021, pp. 69–78.
[11] A. Costan, S. Gaubert, E. Goubault, M. Martel, and S. Putot, “A policy iteration algorithm for computing fixed points in static analysis of programs,” in Computer Aided Verification, 17th International Conference, CAV 2005, ser. Lecture Notes in Computer Science, vol. 3576. Springer, 2005, pp. 462–475.
[12] N. Damouche, M. Martel, and A. Chapoutot, “Transformation of a PID controller for numerical accuracy,” Electron. Notes Theor. Comput. Sci., vol. 317, pp. 47–54, 2015.
[13] A. O. Makkon, “Glpk (gnu linear programming kit),” Available at https://www.gnu.org/software/glpk/glpk.html.
[14] X. Gao and G. A. Constantinides, “Numerical program optimization for high-level synthesis,” in Proceedings of the 2015 ACM/IEEE International Symposium on Field-Programmable Gate Arrays, G. A. Constantinides and D. Chen, Eds. ACM, 2015, pp. 210–213.
[15] N. Damouche, M. Martel, P. Panchehka, J. Qiu, A. Sanchez-Stern, and Z. Tatlock, “Toward a standard benchmark format and suite for floating-point analysis,” 2016.
[16] N. Damouche, M. Martel, and A. Chapoutot, “Improving the numerical accuracy of programs by automatic transformation,” Int. J. Softw. Tools Technol., vol. 19, no. 4, pp. 427–448, 2017.
[17] K. E. Atkinson, “An introduction to numerical analysis.” 1989, New York, vol. 528, p. 38, 1991.
[18] E. Darulova and V. Kuncak, “Towards a compiler for reals,” ACM Trans. Program. Lang. Syst., vol. 39, no. 2, pp. 8:1–8:28, 2017.
[19] S. Cherubin, D. Cattaneo, M. Chiari, A. D. Bello, and G. Agosta, “TaFFO: tuning assistant for floating to fixed point optimization,” IEEE Embed. Syst. Lett., vol. 12, no. 1, pp. 5–8, 2020.
[20] A. Zeller and R. Hildebrandt, “Simplifying and isolating failure-inducing input,” IEEE Trans. Softw. Eng., vol. 28, no. 2, pp. 183–200, 2002.
[21] D. Ben Khalifa and Z. Rakamaric, “Rigorous floating-point mixed-precision tuning,” in Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 1074–1085.
[22] S. Graillat, F. Jézéquel, R. Picot, F. Fétrové, and B. Lathuilière, “Auto-tuning for floating-point precision with discrete stochastic arithmetic,” J. Comput. Sci., vol. 36, 2019.
[23] H. Guo and C. Rubio-González, “Exploiting community structure for floating-point precision tuning,” in Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018. ACM, 2018, pp. 333–343.
[24] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. LeGendre, “Automatically adapting programs for mixed-precision floating-point computation,” in International Conference on Supercomputing, ICS’13. ACM, 2013, pp. 369–378.
[25] R. Singh, P. Wood, I. Laguna, and S. Bagchi, “AMPTGA: automatic mixed precision floating point tuning for GPU applications,” in Proceedings of the ACM International Conference on Supercomputing, ICS’13. ACM, 2013, pp. 27:1–27:12.
[26] M. Martel, “Floating-point format inference in mixed-precision,” in NASA Formal Methods - 9th International Symposium, NFM, ser. Lecture Notes in Computer Science, vol. 10227, 2017, pp. 230–246.
[27] D. Ben Khalifa and M. Martel, “A study of the floating-point tuning behaviour on the n-body problem,” in Computational Science and Its Applications - ICCSA 2021 - 21st International Conference, Cagliari, Italy, September 13-16, 2021, Proceedings, Part V, ser. Lecture Notes in Computer Science, O. Gervasi, B. Murgante, S. Misra, C. Garau, I. Bleicic, D. Taniar, B. O. Apduhan, A. M. A. C. Rocha, E. Tarantino, and C. M. Torre, Eds., vol. 12953. Springer, 2021, pp. 176–190.
[28] A. Adje, D. Ben Khalifa, and M. Martel, “Fast and efficient bit-level precision tuning,” in Static Analysis - 29th International Symposium, SAS 2021, ser. Lecture Notes in Computer Science. Springer, 2021.
[29] D. S. Parker, “Monte carlo arithmetic: exploiting randomness in floating-point arithmetic,” University of California (Los Angeles), Tech. Rep. CSD-970002, 1997.
[30] T. Parr, The Definitive ANTLR 4 Reference, 2nd ed. Pragmatic Bookshelf, 2013.
[31] D. Ben Khalifa, M. Martel, and A. Adje, “POP: A tuning assistant for mixed-precision floating-point computations,” in Formal Techniques for Safety-Critical Systems - 7th International Workshop, FTSCS 2019, ser. Communications in Computer and Information Science, vol. 1165. Springer, 2019, pp. 77–94.
[32] D. Ben Khalifa and M. Martel, “Precision tuning and internet of things,” in International Conference on Internet of Things, Embedded Systems and Communications, INTETEC 2019. IEEE, 2019, pp. 80–85.
[37] A. Das, S. Krishnamoorthy, I. Briggs, G. Gopalakrishnan, and R. Tipireddy, “Fpdetect: Efficient reasoning about stencil programs using selective direct evaluation,” ACM Trans. Archit. Code Optim., vol. 17, no. 3, pp. 19:1–19:27, 2020.

[38] A. Das, I. Briggs, G. Gopalakrishnan, S. Krishnamoorthy, and P. Panchekha, “Scalable yet rigorous floating-point error analysis,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, C. Cuicchi, I. Qualters, and W. T. Kramer, Eds. IEEE/ACM, 2020, p. 51.

[39] B. Saiki, O. Flatt, C. Nandi, P. Panchekha, and Z. Tatlock, “Combining precision tuning and rewriting,” in 28th IEEE Symposium on Computer Arithmetic, ARITH 2021. IEEE, 2021, pp. 1–8.