

Fast and efficient bit-level precision tuning

Dorra Ben Khalifa

▶ **To cite this version:**

Dorra Ben Khalifa. Fast and efficient bit-level precision tuning. Computer Arithmetic. Université de Perpignan, 2021. English. NNT : 2021PERP0026 . tel-03509266

HAL Id: tel-03509266
https://theses.hal.science/tel-03509266

Submitted on 4 Jan 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

Pour obtenir le grade de

Docteur

Délivré par

UNIVERSITE DE PERPIGNAN VIA DOMITIA

Préparée au sein de l'école doctorale
Énergie et Environnement ED305
Et de l'unité de recherche **LAMPS EA 4217**

Spécialité : **Informatique**

Présentée par

Dorra BEN KHALIFA

Fast and Efficient Bit-Level Precision Tuning

Soutenue le 29/11/2021 devant le jury composé de :

Assalé ADJE, Maître de conférences Université de Perpignan Via Domitia, France	Co-Directeur de thèse
Eva DARULOVA, Professeur associé Université d'Uppsala, Suède	Examinatrice
Ganesh GOPALAKRISHNAN, Professeur Université de l'Utah, États-Unis	Rapporteur
Eric GOUBAULT, Professeur École Polytechnique, France	Rapporteur
Philippe LANGLOIS, Professeur Université de Perpignan Via Domitia, France	Examinateur
Matthieu MARTEL, Professeur Université de Perpignan Via Domitia, France	Directeur de thèse
David MONNIAUX, Directeur de recherche CNRS et Université de Grenoble Alpes, France	Rapporteur
Laura TITOLO. Chargée de recherche NASA, États-Unis	Examinatrice

Dedicated to everyone who shared a path on my journey to this point...

Acknowledgements

***

This work would have never seen the light of day if Matthieu MARTEL, who was so my internship director did not suggest that I pursue a thesis, when, true to myself, it was a dream to come true. I want to thank him today, because those three (and a half) years were for me an exciting and fruitful adventure, and this in large part thanks to him. I learned a lot from you Matthieu, *moltes gràcies*!

I'm grateful to Assalé ADJÉ, for having co-directed this thesis. Throughout this work, he was able to provide me with constant support, listening and trust. His knowledge, his criticism, and his constructive advice allowed me to carry out this work. Both have always been there when I felt the need. They always provided me with interesting challenges and pushed towards better results while giving me the freedom to choose my own way. Without their belief and support, this thesis would not be possible.

My sincere gratitude goes to the reviewers of this manuscript, Ganesh GOPALAKRISHNAN, Eric GOUBAULT and David MONNIAUX for having devoted their time to judge my work. I am very honored by their interest in this manuscript, and I am grateful to them for their pertinent remarks and recommendations that helped in improving the quality of my manuscript. I would also like to thank Eva DARULOVA, Philippe LANGLOIS and Laura TITOLO for agreeing to be members of the jury.

My warm thanks go to the members of the LAMPS laboratory, for providing me with a wonderful working environment full of diversity of research topics and with a friendly atmosphere. In particular, my gratitude goes to Sylvia MUNOZ for her invaluable help with the administration and for sometimes making the impossible possible.

This thesis would be impossible without my parents and sisters. Throughout my career, they have always supported, encouraged and helped me. They were able to give me every chance to succeed. May they find, in the accomplishment of this work, the culmination of their efforts and the expression of my most affectionate gratitude. Finally, I thank my little nieces Yasmine and Farah for all the affection and nonsense we share.

Contents

***

List of Figures **xi**

List of Tables **xv**

List of Listings **xv**

List of Abbreviations **xix**

1 Introduction **1**

1.1 Context 1
1.2 Problem Statement 4
1.3 Contributions 6
1.3.1 Contribution 1: Forward and Backward Error Analysis Paradigms . 6
1.3.2 Contribution 2: Integer Linear Programming (ILP) Formulation . . . 6
1.3.3 Contribution 3: Policy Iteration (PI) Optimization 7
1.3.4 Contribution 4: Automated Tool for Precision Tuning 7
1.3.5 Contribution 5: Precision Tuning and Internet of Things 7
1.3.6 Contribution 6: Evaluation of `POP` Performance 8
1.3.7 Contribution 7: Generation of Multiple Precision Code 8
1.3.8 Contribution 8: Comparison Against the State-of-the-Art Tools . . . 8
1.4 Organization of the Thesis 9
1.5 List of Published Work 10
1.5.1 In International Conference Proceedings 10
1.5.2 Long Abstracts 11
1.5.3 Posters 11

I Background and Related Work **13**

2 Definitions and Basic Notions **15**

2.1 Computer Arithmetic 16

2.1.1 Floating-Point Arithmetic . . . 16
2.1.2 Fixed-Point Arithmetic . . . 18
2.1.3 Interval Arithmetic . . . 19
2.1.4 Definitions and Properties . . . 20
2.2 Static Analysis by Abstract Interpretation . . . 21
2.2.1 Order Theory . . . 22
2.2.2 Fixpoints . . . 24
2.2.3 Concrete and Abstract Semantics . . . 25
2.2.4 Widening and Narrowing . . . 28
2.2.5 Policy Iteration . . . 30
2.3 Solvers . . . 32
2.3.1 Satisfiability Modulo Theory . . . 32
2.3.2 Linear Programming . . . 34
2.4 Summary . . . 37

3 Background on Precision Tuning 39
3.1 Analysis and Verification Tools . . . 41
3.1.1 Static Analyzers by Abstract Interpretation . . . 41
3.1.2 General Static Analyzers . . . 42
3.1.3 Dynamic Analyzers . . . 43
3.2 Rewriting-Based Optimization Tools . . . 44
3.3 Precision Tuning Tools . . . 45
3.3.1 Static Analysis Tools . . . 46
3.3.2 Dynamic Analysis Tools . . . 47
3.4 Combining Tools . . . 55
3.4.1 Combining Tools for Analysis and Optimization . . . 55
3.4.2 Combining Tools for Rewriting and Tuning . . . 56
3.5 Summary . . . 57

II A Static Precision Tuning Approach Based on Constraint Generation 59

4 Transfer Functions 61
4.1 Language and Overview . . . 63
4.2 Arithmetic Expressions . . . 64
4.2.1 Forward Analysis . . . 65
4.2.2 Backward Analysis . . . 68
4.3 Elementary Functions . . . 69
4.4 Carry Bit Function . . . 71
4.5 Summary . . . 72

5 Generation of Constraints for Bit-Level Tuning **75**
5.1 SMT-Based Method Constraint Generation . . . 76
5.1.1 Forward Analysis . . . 77
5.1.2 Backward Analysis . . . 78
5.2 ILP-Based Method Constraint Generation . . . 80
5.2.1 Pure ILP Formulation . . . 80
5.2.2 Policy Iteration for Optimized Carry Bit Propagation . . . 84
5.3 Correctness . . . 89
5.3.1 Soundness of the Constraint System . . . 90
5.3.2 ILP Nature of the Problem . . . 91
5.4 Summary . . . 92

6 The POP Tool **93**
6.1 Generalities . . . 94
6.2 POP(SMT) . . . 95
6.2.1 Outline . . . 95
6.2.2 Benchmarks . . . 97
6.2.3 First Results . . . 97
6.3 POP(ILP) . . . 99
6.3.1 Outline . . . 99
6.3.2 Benchmarks . . . 100
6.4 Summary . . . 101

III Internet of Things: A Field of Interest **103**

7 Precision Tuning and Internet of Things **105**
7.1 Tilt Angle Detection by an Accelerometer . . . 106
7.2 Accelerometer-Based Pedometer Algorithm . . . 108
7.2.1 Step Counting Algorithm for Embedded Applications . . . 109
7.2.2 Experimental Evaluation of the Pedometer Program . . . 111
7.3 Summary . . . 116

IV Evaluation of POP Performance for Tuning Numerical Programs **117**

8 Evaluation of POP(SMT) Performance **119**
8.1 Different Cost Functions for the Z3 SMT Solver . . . 120
8.1.1 Assigned Variables vs. All Control Points Cost Functions . . . 121
8.1.2 Numerical Results . . . 121
8.2 MPFR Code Generation . . . 123

8.3 Comparison between POP(SMT) and Precimonious . . . 124
8.3.1 Experimental Setup . . . 124
8.3.2 POP(SMT) vs. Precimonious Comparison Results . . . 126
8.4 Summary . . . 129

9 Fast and Efficient Bit-Level Precision Tuning with POP(ILP) 131
9.1 Numerical Results for the ILP-based Method . . . 132
9.2 POP(ILP) vs. the Prior State-of-the-Art Techniques . . . 133
9.3 Precision Tuning of the N-Body Problem . . . 136
9.3.1 Experimental Study . . . 138
9.3.2 Distance between the Exact and the Computed Positions of the Bodies 140
9.4 Summary . . . 143

10 Conclusion and Perspectives 145
10.1 Summary of Contributions . . . 146
10.2 Short-Term Perspectives . . . 148
10.2.1 Scalability of POP . . . 148
10.2.2 Extension of POP . . . 149
10.2.3 License . . . 149
10.2.4 Handling of Loops . . . 149
10.2.5 Fixed-Point Code synthesis . . . 150
10.2.6 Applications . . . 150
10.3 Long-Term Perspectives . . . 151
10.3.1 Combining Frameworks . . . 151
10.3.2 Applications to Neural Networks (NNs) . . . 151
10.3.3 GPU Oriented Approach . . . 152
10.4 Final Words . . . 153

V Résumé Étendu à L'intention du Lecteur Francophone 155

11 Analyse Statique pour le Réglage de la Précision Numérique 157
11.1 Notions Préliminaires et Langage de l'Outil POP . . . 159
11.1.1 Notions Préliminaires . . . 159
11.1.2 Langage Impératif . . . 160
11.2 Génération des Contraintes par Analyse Statique . . . 161
11.2.1 Analyse en Avant et en Arrière . . . 161
11.2.2 Programmation Linéaire en Nombres Entiers (ILP) . . . 162
11.2.3 Itération sur les Politiques (PI) . . . 163
11.3 L'outil POP . . . 165
11.3.1 Implémentation . . . 165

11.3.2 Réglage de la Précision et Internet des Objets (IoT) 167
11.4 Résultats Expérimentaux 169
11.4.1 Évaluation des Performances de `POP(SMT)` et `POP(ILP)` 169
11.4.2 Comparaison avec l'Outil `Precimonious` 170
11.4.3 Génération du Code de Précision Multiple 171
11.5 Synthèse de l'Existant 174

Bibliography **177**

List of Figures

***

1.1 The validity of Moore's law until 2020 (photo credits from https://www.ncbi.nlm.nih.gov/books/NBK321721/). 2
1.2 IoT connected devices from 2015 to 2025 (in billions) (photo credits from [MPN+19]). 3

2.1 Fixed-point representation of a signed number. 19
2.2 Schematic representation of ufp, nsb and ulp for values and errors. 20
2.3 Hasse diagram for the partial order set of $(\mathcal{P}(a,b,c),\subseteq)$. 23
2.4 A simple integer loop and its semantic equations. 25
2.5 Kleene's iterations of the program of Figure 2.4. 26
2.6 Widening and narrowing steps of the example of Figure 2.4 30
2.7 Policy iteration method on the program of Figure 2.4. 32

3.1 Summary of the precision tuning tools. 46
3.2 Illustration of the *delta-debugging* algorithm. 48

4.1 Language of input programs. 63
4.2 Example of forward addition: $3.0\,|53|\,+\,1.0\,|53|\,=\,4.0\,|54|$. 66
4.3 Schematic representation of the optimized carry bit function ξ. 72

5.1 ILP constraints with pessimistic carry bit propagation $\xi = 1$. 81
5.2 Constraints solved by PI with min and max carry bit formulation. 88
5.3 Small-step operational semantics of arithmetic expressions. 90

6.1 Functional architecture of POP. 94
6.2 POP(SMT) overview. 96
6.3 Measures of the efficiency of the analysis of POP(SMT) on the two input examples: the time execution measure, the optimization of the number of bits of the transformed programs compared to the original ones and the percentage of the double precision variables after analysis. 98
6.4 POP(ILP) overview. 100

7.1 Outline of tilt angle measurement by a 3-axis accelerometer. 107
7.2 Efficiency of POP(SMT) on the accelerometer tilt measure application. Top: the percentage of half and single precision for different nsb assertions after analysis. Bottom: The percentage of float and double variables for nsb = 24, 28, 30, 32 and 36 bits. 108
7.3 Top: Improvement of the number of bits compared to the initial bits number in the original program. Bottom: Memory size (in Bytes) for different user accuracy requirements. 109
7.4 Overview of the footstep counter algorithm. 110
7.5 Top: Number of bits optimized for different user accuracy requirements. Bottom: Improvement compared to the initial total number of bits of the original program. 112
7.6 Comparing the execution time of POP(SMT) analysis and the time spent by Z3 to find a solution to our system of constraints. 113
7.7 POP(SMT) optimization of the total number of bits for different magnitude of ranges of x, y and z. 114
7.8 The percentage of program variables in FP8, FP16, FP32 and FP64 after POP(SMT) analysis. 114

8.1 Mixed-precision tuning results in FP8, FP16, FP32, FP64 and FP128 for the all control points and variables accuracies cost function (top) and for the optimized cost function considering only the variables assigned (bottom). . 122
8.2 Measured error between the exact results with multiple precision and the results obtained with POP(SMT) for the **simpson**, **arclength** and **low pass filter** programs. 124

9.1 Simulated movement of the bodies. 136
9.2 Distance between the exact and the computed position for the 5 bodies with nsb = 11, 18, 24, 34, 43 and 53 bits. 143

10.1 Left: Initial program. Right: Annotations after precision tuning with POP. . 152

11.1 Représentation de ufp, nsb et ulp des nombres et des erreurs. 159
11.2 Représentation de la fonction ξ du bit de retenue. 160
11.3 Langage des programmes de POP. 161
11.4 Contraintes ILP avec une propagation pessimiste du bits de retenues $\xi = 1$. 164
11.5 Contraintes de la méthode d'itératon sur les politiques avec une nouvelle formulation du bit de retenue. 166
11.6 L'architecture de notre outil POP. 167

11.7 Efficacité de `POP(SMT)` sur l'application de l'accéléromètre. En haut : le pourcentage de FP16 et FP32 pour différentes assertions de pécision de l'utilisateur. En bas : Le pourcentage des variables en FP32 et FP64 pour $\mathsf{nsb} = 24, 28, 30, 32$ et 36 bits. 168
11.8 Optimisation du nombre total des bits par `POP(SMT)` pour différents intervalles utilisées pour les variables x, y et z. 170

List of Tables

***

2.1 Parameters defining the IEEE754 floating-point formats. 16
2.2 Numbers in IEEE754 double precision. 17

3.1 Hardware and software properties of the systems used in [BBD+09]. 40
3.2 Precision tuning tools properties. 54

8.1 Differences between `POP(SMT)` and `Precimonious`. 125
8.2 Number of optimized variables by both tools for each program. 126
8.3 Precision tuning results of `POP(SMT)` and `Precimonious` for error thresholds of 10^{-6} and 10^{-4}. 127
8.4 Precision tuning results of `POP(SMT)` and `Precimonious` for error thresholds of 10^{-10} and 10^{-8}. 128

9.1 Precision tuning results for `POP(ILP)` for the ILP and PI methods. 134
9.2 Comparison between `POP(ILP)`, `POP(SMT)` and `Precimonious`: number of bits saved by the tool and time in seconds for analyzing the programs. . . . 135
9.3 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(ILP)` analysis time and the execution time of the `MPFR` generated code. 141

10.1 The short and long-term perspectives of our thesis. 153

11.1 Comparaison entre `POP(ILP)`, `POP(SMT)` et `Precimonious` : nombre de bits économisés par chaque outil et temps d'analyse en secondes. 171
11.2 Distances entre la position exacte (calculée avec 500 bits) et la position calculée avec n bits. Distances données pour chaque corps après 10 et 30 ans de simulation. Suivi du temps d'analyse de `POP(ILP)` et du temps d'exécution du code généré par `MPFR`. 172

List of Listings

***

4.1 A simple example to show the nature of constraints generated. Left: source program. Center: program annotated with labels. Right: source program with inferred accuracies. 65

5.1 Top left: source program. Top right: pendulum movement for $\theta = \frac{\pi}{4}$. Bottom left: program annotated with labels. Bottom right: program with inferred accuracies. 83

7.1 Example of mixed-precision inference. Left: source program with inferred accuracies. Right: Program formats. 110

7.2 Left: Low pass filter FIR, remove mean and autocorrelation functions of the original program. Right: source program with inferred accuracies. 115

9.1 Left: source program annotated with labels. Right: program with `POP` generated data types with ILP formulation. 137

9.2 Python `MPFR` code automatically generated by `POP(ILP)` for the N-body problem for a `nsb` $= 18$ bits on the positions of the planets at the end of the simulation. 142

11.1 Gauche: programme source annoté avec les labels. Droite: Programme optimisé généré par `POP(ILP)`. 173

List of Abbreviations

***

`POP`	Precision `OP`timizer
SMT	**S**atisfiability **M**odulo **T**heories
LP	**L**inear **P**rogramming
ILP	**I**nteger **L**inear **P**rogramming
PI	**P**olicy **I**teration
FP	**F**loating-**P**oint
FLOPS	**FL**oating-Point **O**perations **P**er **S**econds
HPC	**H**igh **P**erformance **C**omputing
IoT	**I**nternet **o**f **T**hings
GPU	**G**eneral **P**rocessing **U**nit
DNN	**D**eep **N**eural **N**etwork
`POP(SMT)`	`POP` based forward and backward analysis solved by the `Z3` SMT solver.
`POP(ILP)`	`POP` based pure **ILP** with an over-approximation of the carry functions and **PI** technique for more precise carry bit function, solved and by the `GLPK` solver.
`nsb`	number of significant bits of a number.
$\texttt{nsb}_F$	number of significant bits in the forward mode.
$\texttt{nsb}_B$	number of significant bits in the backward mode.
$\texttt{nsb}_\texttt{e}$	number of significant bits of the error of a number.
`require_nsb(x,n)`	Statement that informs `POP` that the programmer wants n number of significant bits on variable x.

Chapter 1

"We have seen that computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially, because it produces objects of beauty."

— Donald E. Knuth[Knu74]

Introduction

***

1.1 Context

End of Moore's law: what is next for the future of computing?

A decade after the Dennard scaling [DGY+74], which argued that one could continue to decrease the transistor feature size and voltage while keeping the power density constant, has ended, Moore's law [Moo65] is also reaching its end. This law predicted that the number of transistors on an integrated circuit was doubling at a steady pace since 1965, bringing exponential increases to computing power. However, the end of scaling will not only have an effect on the processor but also on communications and storage. Over the last years, data is growing faster than Moore's law as illustrated in Figure 1.1, and the cost involved in developing new technology nodes is skyrocketing day after day. Not surprisingly, the increasing communication bandwidth entails more sophisticated protocols and higher speeds which require more processing power in the network nodes. Consequently, this will translate in increased power consumption up to the point where it becomes technically and economically unfeasible to further increase the communication bandwidth [DDBC+19].

Power consumption has become a critical aspect in the evolution of High Performance Computing systems (HPC). Nowadays, modern supercomputers run on huge amounts of electrical power. For instance, the Top500[1] project report shows that supercomputer performance is approximately doubling every year, whilst power consumption is also rising. As of June 2020, the most powerful supercomputer Fugaku, number 1 of the Top500, is nowadays able to reach around 400 petaFLOPS (400×10^{15} FLOPS) in terms of computation power. This equates to assembling tens of millions of laptops together against the 148.6 petaFLOPS of the predecessor supercomputer, Summit. At the same time, these supercomputers are in the top ten of the Green500[2], with around 14.7 gigaFLOPS per

[1] The Top500 is a statistical ranking showing the characteristics and performance of the 500 most powerful machines in the world (https://www.top500.org/).

[2] The Green500 provides rankings of the most energy-efficient supercomputers in the world (https://www.

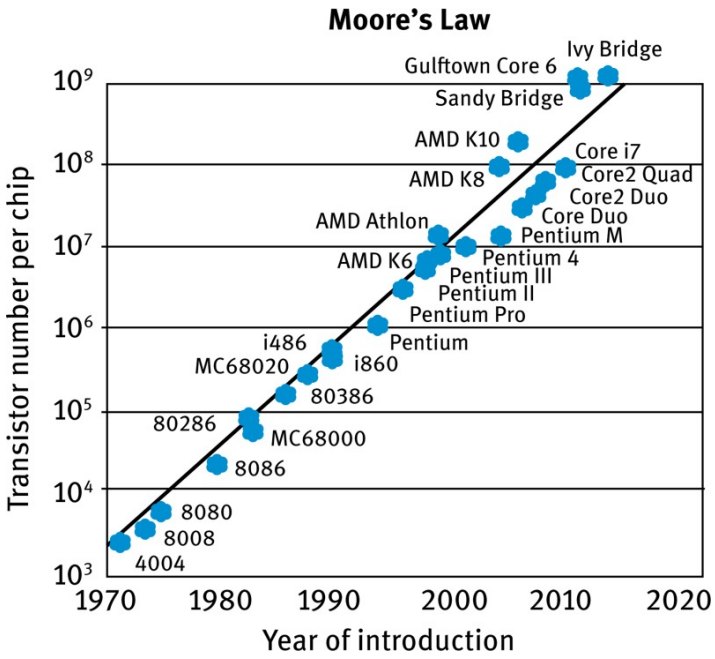


Figure 1.1: The validity of Moore's law until 2020 (photo credits from https://www.ncbi.nlm.nih.gov/books/NBK321721/).

Watt. However, energy consumption continues to increase; while the Summit computer has a consumption of 10 MW, Fugaku goes up to 28 MW. In brief, controlling the energy consumption of HPC platforms has become a necessity. It is not considered only as a way to control the cost but also as a step forward on the road towards exaflops.

From megascale to exascale computers...

As science tries to answer bigger and deeper questions, the need for more computational power continues to grow. To address this need, the current roadmap for HPC systems aims at reaching the exascale level, i.e. 10^{18} FLOPS within the time frame 2023 -2024, allowing them to run hundreds of fine-grained scientific applications simultaneously on a huge number of computing nodes [CAL+17]. Hence, the leap up to exascale computing will not only be a milestone in the history of computer science, but it will also bring the prospect of transforming progress in many areas, including energy, life sciences, material chemistry, national security, etc. Meanwhile, the challenges associated with exascale computing extend far beyond a certain floating-point throughput [VEL+17]. Specifically, an exascale supercomputer is envisioned to comprise of on the order of 100,000 interconnected servers or nodes in a target power envelope of $\approx$ 20 MW. This target can be achieved only with a sufficient memory bandwidth to feed the massive compute throughput, a sufficient memory capacity to execute meaningful problem sizes, and with user intervention due to hardware or system faults limited to the order of a week or more on average. Let us

top500.org/lists/green500/).

state that there is a 7× gap in FLOPS per Watt between the current most energy-efficient supercomputers and the exascale target where in this latter each node is supposed to deliver more than 10 teraFLOPS with less than 200W.

... to Internet of Things devices

The internet of things (IoT) has risen in use and popularity since it was first introduced at the beginning of the 21^{st} century, pointing toward productive and exciting new directions for a whole generation of information devices [BM19]. The term IoT refers to a paradigm where physical objects such as sensors, actuators, home appliances and so forth, are connected to the internet. The expected number of IoT devices in the near future are estimated to be in billions and are continuously increasing in number [RBA+21], e.g. this number will reach more than 75 billions by 2025 as depicted in Figure 1.2. Thus, these devices will produce a lot of electronic waste and will also consume a significant amount of energy in order to execute different tasks. This will eventually pose a challenge to reduce the energy consumption and will also demand for new ways of developing a green communication across the network.

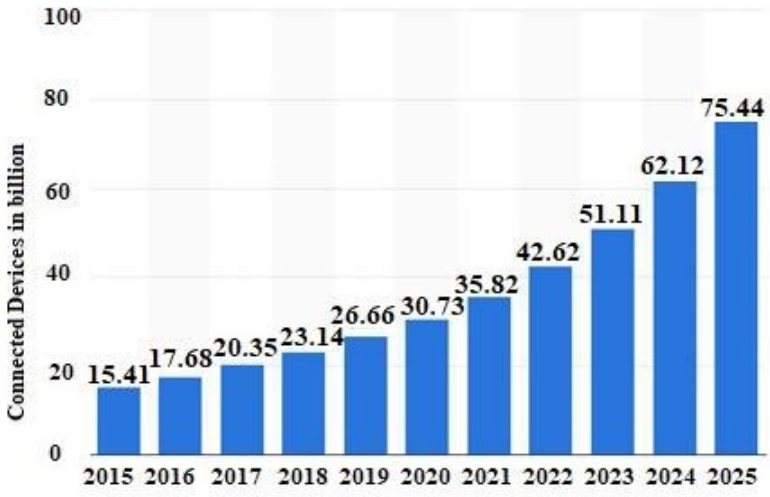


Figure 1.2: IoT connected devices from 2015 to 2025 (in billions) (photo credits from [MPN+19]).

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 [BM20]. 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.

... and towards a new era of innovation.

To put in a nutshell, our previous discussion has showed that energy consumption has become a major problem in many fields as we live now in an energy-scarce world. To tackle this problem, designing scalable, reliable, and energy efficient applications remains a real challenge to explore. In other words, designers seeking to reduce the energy usage should be helped in choosing adequate protocols, services and the best implementations of their applications with regards to the targeted infrastructure.

For instance, to fulfill the 20 MW target needed for exascale supercomputers and to improve the performance of HPC systems, energy-efficient heterogeneous supercomputers need to be coupled with software stacks able to exploit a range of techniques to trade-off between power, performance, and other metrics of quality to achieve the desired goals without exceeding the power envelope. With this solution, we attain the objective of this work.

The goal of this thesis is to propose an approach that guides developers and designers to achieve the best compromise between power and performance to attain the necessary energy efficiency. This approach consists of using a reduced (or customized) precision number representations which has been widely recognized, in these recent years, as one of the promising tools in the designer's arsenal. This process, called *precision tuning*, can make it possible to save memory and, by way of consequence, it has a positive impact on the footprint of programs concerning energy consumption, bandwidth usage and computation time.

1.2 Problem Statement

In this dissertation, we are interested in presenting a new technique for precision tuning. Let us consider a program P computing some numerical result R, typically but not necessarily in the IEEE754 floating-point arithmetic [ANS08]. Precision tuning then consists of finding the smallest data types for all the variables and expressions of P such that the result R has some desired accuracy. This problem originates from two different facts. The first one is that many applications can tolerate some loss of quality during computation, as in the case of media processing (audio, video and image), data mining, machine learning, etc [CA20]. The second fact is that the mantissa bits of floating-point arithmetic data, for example, are often wasted (they represent only the irrelevant parts of the computational error) and yet have to be hauled and stored across the memory hierarchy. More precisely, developers without any extensive background in numerical accuracy and computer arithmetic, tend to use the highest precision available in hardware (usually FP64 double precision). Despite the fact that the results will be more accurate, this increases significantly the application run-time, bandwidth capacity and the memory and energy consumption of the system.

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 [Mar17]: a bound on the absolute error $|x - \hat{x}|$ between the represented $\hat{x}$ value and the exact value x that we would have in the exact arithmetic. Therefore, the challenge is to use no more precision than needed wherever possible without compromising overall accuracy: using a too low precision for a given algorithm and data set leads to inaccurate results. Indeed, we aim in this thesis to apply *mixed-precision tuning* on the program formats [BMA19]. Contrarily to the uniform precision, we denote by mixed-precision tuning the fact of combining different precision for different variables in the same program, e.g. FP16, FP32, FP64, etc.

These last years, much attention has been paid to the precision tuning problem [CBB+17, DHS18, GR18a, KSW+19a, LHdSL13a, RNN+13]. A common point to all the techniques cited previously is that they follow a trial-and-error strategy. Roughly speaking, one chooses a subset S of the variables of P, assigns to them smaller data types (e.g. FP32 instead of FP64 [ANS08]) and evaluates the accuracy of the tuned program P'. If the accuracy of the result returned by P' is satisfying then new variables are included in S or even smaller data types are assigned to certain variables already in S (e.g. FP16). Otherwise, if the accuracy of the result of P' is not satisfying, then some variables are removed from S. This process is applied repeatedly, until a stable state is found. Existing techniques differ in their way to evaluate the accuracy of programs, done by dynamic analysis [GR18a, KSW+19a, LHdSL13a, RNN+13] or by static analysis [CBB+17, DHS18] of P and P'. They may also differ in the algorithm used to define S, *delta-debugging* being the most widespread method [RNN+13]. A notable exception is the `FPTuner` tool [CBB+17] which relies on a local optimization procedure by solving quadratic problems for a given set of candidate data types. A more exhaustive state-of-the-art about precision tuning techniques will be presented in Chapter 3 of this dissertation.

Anyway all these techniques suffer from the same combinatorial limitation: If P has n variables and if the method tries k different data types then the search space contains k^n configurations. They scale neither in the number n of variables (even if heuristics such as *delta-debugging* [RNN+13] or *branch and bound* [CBB+17] reduce the search space at the price of optimality) or in the number k of data types which can be tried. In particular, bit-level precision tuning, which consists of finding the minimal number of bits needed for each variable to reach the desired accuracy, independently of a limited number k of data types, is not an option.

So, the method introduced in this thesis for precision tuning of programs is radically different. Here, no trial-and-error method is employed. Instead, the accuracy of the arithmetic expressions assigned to variables is determined by semantic equations, in function of the accuracy of the operands. In practical terms, we propose a novel static technique based on a semantic 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 [ANS08]) and its complexity does not increase as the number of data types increases.

1.3 Contributions

To address the precision tuning problem, we conduct in this thesis static analysis methods that contribute to a fast and efficient determination of the minimal precision on the inputs and the intermediary results of numerical programs. We summarize these contributions in the following subsections.

1.3.1 Contribution 1: Forward and Backward Error Analysis Paradigms

The first contribution of this thesis is to combine a forward and a backward error analysis which are two popular paradigms of error analysis, done by abstract interpretation. The forward analysis is classical. It examines how errors are magnified by each operation aiming to determine the accuracy on the results. Next, a user requirement is given denoting the final accuracy wanted on some control points of the outputs. This accuracy designs the number of significant bits required by the user on a variable of the program. By taking into consideration the user assertions and the results of the forward analysis, the backward analysis is a complementary approach that starts with the computed answer to determine the exact input that would produce it in order to satisfy the desired accuracy, independently of any computer arithmetic.

As could be expected, the forward and backward analysis can be handled iteratively to refine the results until a fixpoint is reached. Next, these forward and backward transfer functions are expressed as a set of linear constraints made of propositional logic formulas and relations between integer elements only. After, a Satisfiability Modulo Theories (SMT) solver is used repeatedly to find the existence of a solution with a certain weight expressing the number of significant bits of variables.

1.3.2 Contribution 2: Integer Linear Programming (ILP) Formulation

The second contribution is a relaxation of the first one. Here, we reduce the problem to an Integer Linear Problem (ILP) which can be optimally solved in one shot by a classical linear programming solver (LP) with no iteration. Concerning the number n of program variables, the method scales up to the solver limitations and the solutions are naturally found at the bit-level, making the number of data types which can be tried irrelevant. An important point is that the optimal solution to the continuous linear programming relaxation of our ILP is a vector of integers. As a consequence, we may use a linear programming solver among real numbers whose complexity is polynomial [Sch98], contrarily to the linear

solvers among integers whose complexity is NP-Hard [Pap81]. This makes our precision tuning method solvable in polynomial-time, contrarily to the existing exponential methods.

1.3.3 Contribution 3: Policy Iteration (PI) Optimization

The third contribution consists of implementing an optimization for 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, we go one step further 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. However, the problem does not reduce any longer to an ILP problem (min and max operators are needed). Then we use policy iteration (PI) [CGG+05] to find efficiently the solution. Moreover, we present an evaluation of the performance of the ILP and PI methods on several benchmarks from different domains.

1.3.4 Contribution 4: Automated Tool for Precision Tuning

The methods described in sections 1.3.1, 1.3.2 and 1.3.3 above have been implemented inside a tool for precision tuning [BMA19, BM19, BM20, ABM21, BM21a, BM21b] named `POP`, short for Precision `OP`timizer. As a consequence, our tool contains two sub-tools: the first one corresponds to the forward and backward error analysis technique expressed as a set of constraints and solved by a SMT solver (the method described in Section 1.3.1). The second one corresponds to the ILP formulation which is solved by a LP solver and its optimization later using the PI method (the methods described in sections 1.3.2 and 1.3.3). The main hierarchy of our tool can be summarized in four basic steps:

- `POP` parses the program and generates its syntactic tree.
- While `POP` achieves only precision tuning, it uses a dynamic analysis which produces an under-approximation of the ranges of the variables for inputs taken randomly in user defined ranges.
- Based on a semantic modeling on the errors propagation, `POP` generates three variants of semantic equations. Each one refers to one of the contributions already presented.
- Once the semantic equations are generated, our tool calls two kinds of solvers: a SMT solver and a LP solver, to find an optimal solution to these equations. The solution corresponds to the minimal number of bits needed for each variable in the program.

1.3.5 Contribution 5: Precision Tuning and Internet of Things

To demonstrate the usefulness of our tool, we apply precision tuning on applications related to IoT because of the serious problems about the memory and energy consumption that are pervasive nowadays on the majority of IoT devices. In this context, we present in this

dissertation the measurements of performance of POP, in terms of quality of analysis and precision improvement, on two representative applications coming from the IoT field. The experiments are made by both versions of the tool. Also, we highlight a real comparison between the behaviours of these two versions on tuning these applications.

1.3.6 Contribution 6: Evaluation of POP Performance

Alongside the IoT field, we experiment the performance of our tool on several numerical programs coming from mathematical libraries or other application domains such as scientific computing, signal processing, physics, etc. The first experimentation concerns the contribution highlighted in Section 1.3.1. In particular, we compare two ways of optimizing programs in our tool throughout the definition of different cost functions given to the SMT solver. Second, we measure the run-time errors between the exact results given by an execution in multiple precision and the results of tuned programs by both versions of our tool. Third, we compare POP against prior state-of-the-art tools by taking into account several metrics of comparison. The last two experiments are detailed in the subsections hereafter.

1.3.7 Contribution 7: Generation of Multiple Precision Code

In the present contribution, we measure the error between the exact results given by an execution in multiple precision, using the MPFR library [FHL+07], and the results of the optimized programs returned by POP. Next, this error is compared to the error threshold set by the user. We will show that the measured error is always less than the threshold given by the user for several benchmarks analyzed by our tool. This is a way of assessing experimentally the correctness of our tool.

1.3.8 Contribution 8: Comparison Against the State-of-the-Art Tools

The last contribution of our thesis is a detailed comparison between POP and the prior state-the-art tool Precimonious [RNN+13]. Practically, the Precimonious tool will undergo a double comparison. Each one corresponds to a version of our tool POP. The experiments of this comparison are made on benchmarks coming from both tools and the results are evaluated in terms of analysis time, speed and the quality of tuning. Even though both tools use different techniques, we adjust the comparison criteria in order to make a closer comparison of the real behavior of these tools. Hence, we will demonstrate that the technique proposed in this thesis for precision tuning clearly encompasses the state-of-the-art techniques.

1.4 Organization of the Thesis

The remainder of this dissertation is divided into four parts, each subdivided into chapters as follows:

Part I: Background and Related Work

This part is composed of chapters 2 and 3. Chapter 2 describes all relevant background knowledge, which is necessary to understand and follow the concepts that we use in this thesis. We start by presenting some basic concepts related to the finite-precision arithmetic. Next, we provide necessary notions on the static analysis by abstract interpretation method. Also, we introduce the PI method and the prior work that have already used this technique. We end up this chapter by discussing briefly the SMT and ILP theories and presenting the differences between SMT and LP solvers Chapter 3 deals with an exhaustive survey on the existent techniques and tools concerning precision tuning. Besides, we extend our survey to present tools for error analysis and code transformation.

Part II: A Static Precision Tuning Approach Based Constraints Generation

This part is organized in three chapters and tackles the static method for precision tuning proposed in this thesis. We recall that our approach is based on a semantic modelling of the propagation of the numerical errors throughout the program source. As a result, this analysis is expressed as a set of constraints whose minimal solution gives the best tuning of the program. In Chapter 4, we define the transfer functions for each element of our imperative language. Furthermore, we introduce an optimization of the carry bit function that can propagate throughout the programs. Once the transfer functions are defined, Chapter 5 deals with the different set of constraints generated by each method embodied in `POP`. The first set of constraints expresses the precision tuning problem as a set of first order logical propositions among relations between linear integer expressions solved by a SMT solver. The second set of constraints corresponds to a pure ILP solved in one breath by a LP solver. The third set of constraints optimizes the propagation of carries in the elementary operations and it can be solved using the PI method. Chapter 6 incorporates the main architecture of our tool for precision tuning and its different steps of implementation. Chapters 4, 5 and 6 are revised versions of the following articles:

- Dorra Ben Khalifa, Matthieu Martel and Assalé Adjé. POP: A Tuning Assistant for Mixed-Precision Floating-Point Computations. *In the 7th International Workshop on Formal Techniques for Safety-Critical Systems, FTSCS'19, Revised Selected Papers.*
- Assalé Adjé, Dorra Ben Khalifa and Matthieu Martel. Fast and Efficient Bit-Level Precision Tuning. *In the 28th Static Analysis Symposium, SAS'21.*

Part III: Internet of Things: A Field of Interest

This part is made only of Chapter 7. It deals with experimenting our tool in a new application domain, namely for IoT. We present the different experimental results in terms of precision improvement, execution time, optimization in term of number of bits and mixed-precision configurations obtained after analysis, for two representative examples of this field. This chapter is a revised version of the following articles:

- Dorra Ben Khalifa and Matthieu Martel. Precision Tuning of an Accelerometer-Based Pedometer Algorithm for IoT Devices. *In the IEEE International Conference on Internet of Things and Intelligence System, IoTaIS'*20.
- Dorra Ben Khalifa and Matthieu Martel. Precision Tuning and Internet of Things. *In the IEEE International Conference on Internet of Things, Embedded Systems and Communications, IINTEC'*19.

Part IV: Evaluation of the Tool Performance for Tuning Numerical Programs

The final part is devoted to the evaluation of performance of our tool. Chapter 8 describes the performance of the forward and backward error analysis method in several manners on our benchmarks whilst Chapter 9 incorporates the evaluation of the ILP and PI formulations. Also, we validate the efficiency of our approach on one of the oldest problem of modern physics, the N-body problem. Chapters 8 and 9 are revised versions of the following articles:

- Dorra Ben Khalifa and Matthieu Martel. An Evaluation of POP Performance for Tuning Numerical Programs in Floating-Point Arithmetic. *In the* 4*th International Conference on Information and Computer Technologies, ICICT'*21.
- Assalé Adjé, Dorra Ben Khalifa and Matthieu Martel. Fast and Efficient Bit-Level Precision Tuning. *In the* 28*th Static Analysis Symposium, SAS'*21.
- Dorra Ben Khalifa and Matthieu Martel. A Study of the Floating-Point Tuning Behaviour on the N-body Problem. *In the* 21 *International Conference on Computational Science and Its Applications, ICCSA'*21.

1.5 List of Published Work

This dissertation is drawn from the following publications:

1.5.1 In International Conference Proceedings

- Assalé Adjé, Dorra Ben Khalifa and Matthieu Martel. **Fast and Efficient Bit-Level Precision Tuning**. *In the* 28*th Static Analysis Symposium, SAS'*21.
- Dorra Ben Khalifa and Matthieu Martel. **A Study of the Floating-Point Tuning Behaviour on the N-body Problem**. *In the* 21 *International Conference on Computational Science and Its Applications, ICCSA'*21.

- Dorra Ben Khalifa and Matthieu Martel. **An Evaluation of POP Performance for Tuning Numerical Programs in Floating-Point Arithmetic**. *In the 4th International Conference on Information and Computer Technologies, ICICT'21.*
- Dorra Ben Khalifa and Matthieu Martel. **Precision Tuning of an Accelerometer-Based Pedometer Algorithm for IoT Devices**. *In the IEEE International Conference on Internet of Things and Intelligence System, IoTaIS'20.*
- Dorra Ben Khalifa and Matthieu Martel. **Precision Tuning and Internet of Things**. *In the IEEE International Conference on Internet of Things, Embedded Systems and Communications, IINTEC'19.*
- Dorra Ben Khalifa, Matthieu Martel and Assalé Adjé. **POP: A Tuning Assistant for Mixed-Precision Floating-Point Computations**. *In the 7th International Workshop on Formal Techniques for Safety-Critical Systems, FTSCS'19, Revised Selected Papers.*

1.5.2 Long Abstracts

- Dorra Ben Khalifa and Matthieu Martel. **Managing Performance vs. Accuracy Trade-offs with an Improved Bit-Level Precision Tuning**. *In the 21th International CMMSE Conference and the 1st Conference on High Performance Computing (CHPC), 2021.*

1.5.3 Posters

- Dorra Ben Khalifa. **Floating-Point Precision Tuning by Static Analysis**. *In the Formal and Exhaustive Analysis of Numerical Intensive Control Software for Embedded Systems FEANICSES, ISAE-SUPAERO Toulouse*, 2019.

Part I

Background and Related Work

Chapter 2

Definitions and Basic Notions

***

2.1 Computer Arithmetic . . . 16
2.1.1 Floating-Point Arithmetic . . . 16
2.1.2 Fixed-Point Arithmetic . . . 18
2.1.3 Interval Arithmetic . . . 19
2.1.4 Definitions and Properties . . . 20
2.2 Static Analysis by Abstract Interpretation . . . 21
2.2.1 Order Theory . . . 22
2.2.2 Fixpoints . . . 24
2.2.3 Concrete and Abstract Semantics . . . 25
2.2.4 Widening and Narrowing . . . 28
2.2.5 Policy Iteration . . . 30
2.3 Solvers . . . 32
2.3.1 Satisfiability Modulo Theory . . . 32
2.3.2 Linear Programming . . . 34
2.4 Summary . . . 37

As we have mentioned in the general introduction, we are interested in the problem of determining the minimal precision on the inputs and intermediary results of numerical programs in order to get a desired accuracy on the outputs. Although our technique is independent of a particular computer arithmetic, this allows compilers to select the most appropriate formats, for example IEEE754 half (FP16), single (FP32), double (FP128) or quad (FP128) formats [ANS08, MBdD+10] for each variable. It is then possible to save memory, reduce CPU usage and use less bandwidth for communications whenever distributed applications are concerned. Our approach is also easily generalizable to fixed-point arithmetic for which it is mandatory to determine the formats of the data, for example for FPGA implementations [GC15].

In this chapter, we describe all relevant background, which is necessary to understand and follow the concepts that we use in our thesis. We will be brief in presenting some techniques whereas a complete study on the existing tools and approaches that address the precision tuning will be presented in Chapter 3.

This chapter is organized as follows. We first review in Section 2.1 necessary background about finite-precision arithmetic which is an important building block for precision tuning, in particular, the IEEE754 Standard of floating-point arithmetic and the fixed-point arithmetic. We later highlight in Section 2.2 the static analysis by abstract interpretation method used to model the propagation of the errors across the programs. As our analysis will be expressed by two kinds of constraints, Section 2.3 presents basic notions about the Satisfiability Modulo Theories (SMT) solvers used to solve constraints made of propositional logic formulas and relations between affine expressions among integers, and Linear programming (LP) solvers used when we formulate the problem as an Integer Linear Programming problem (ILP). Section 2.4 concludes.

2.1 Computer Arithmetic

Representing and manipulating real numbers efficiently by computers is required in many fields of science, engineering, finance and more. There exists several representations for approximating real numbers. Out of these representations, we will focus in this section on the floating-point (FP) arithmetic, fixed-point arithmetic and interval arithmetic. Also, we present essential definitions helpful for understanding our technique of precision tuning.

2.1.1 Floating-Point Arithmetic

Currently, FP numbers are the most common representation used in numeric applications. Therefore, optimizing the use of FP formats is often a key to obtain high performance. In the following, we will focus on these formats, and in particular on the IEEE754 ones. We refer the reader to the Handbook of Floating-Point Arithmetic by Muller et al. [MBdD+10] for a detailed and formal reference on the general subject of FP arithmetic.

Normalized IEEE754 Binary Floating-Point

Format	Name	p	e bits	e_{min}	e_{max}
FP16	Half precision	11	5	-14	$+15$
FP32	Single precision	24	8	-126	$+127$
FP64	Double precision	53	11	-1122	$+1223$
FP128	Quadruple precision	113	15	-16382	$+16383$
FP256	Octuple precision	237	19	-262142	$+262143$

Table 2.1: Parameters defining the IEEE754 floating-point formats.

The IEEE754 Standard [ANS08] for FP arithmetic is a technical standard for FP computation created by the Institute of Electrical and Electronic Engineers (IEEE). The standard formalizes a binary FP number x in base (or radix) β, generally $\beta = 2$, as a triplet made of a sign, a mantissa and an exponent as shown in Equation (2.1), where $s \in$ {-1,1} is the sign, m represents the mantissa, $m = d_0.d_1...d_{p-1}$, with the digits $0 \leqslant d_i < \beta$, $0 \leqslant i \leqslant p-1$, p is the precision (length of the mantissa) and the exponent $e \in [e_{min}, e_{max}]$.

$$x = s.m.\beta^{e-p+1} \tag{2.1}$$

The IEEE754 Standard specifies some particular values for p, e_{min} and e_{max}. Also, it defines binary formats (with $\beta = 2$) which are described in Table 2.1. Hence, the IEEE754 Standard distinguishes between normalized and denormalized numbers. Indeed, the normalization of a FP number ensuring $d_0 \neq 0$ guarantees the uniqueness of its representation. For the lowest value of the exponent, denormalized numbers allow the first digits $d_0, \cdots, d_k$ with $k \leqslant p$ to be equal to zero. Doing so, denormalized numbers make underflow gradual [MBdD+10]. The IEEE754 Standard defines also some special numbers[1]. All these numbers are summarized in Table 2.2 (in double precision). Moreover, the IEEE754 Standard defines four rounding modes for elementary operations among FP numbers which are: towards $+\infty$, towards $-\infty$, towards zero and towards the nearest denoted by $\uparrow_{+\infty}$, $\uparrow_{-\infty}$, $\uparrow_0$ and $\uparrow_\sim$, respectively.

x	Exponent e	Mantissa m
$x = 0$ (if s = 0) $x = -0$ (if s = 1)	$e = 0$	$m = 0$
Normalized numbers $x = (-1)^s \times 2^{e-1023} \times 1.m$	$0 < e < 2047$	any
Denormalized numbers $x = (-1)^s \times 2^{e-1022} \times 0.m$	$e = 0$	$m \neq 0$
$x = +\infty$ (if s = 0) $x = -\infty$ (if s=0)	$e = 2047$	$m = 0$
$x = NaN$ (Not a Number)	$e = 2047$	$m \neq 0$

Table 2.2: Numbers in IEEE754 double precision.

Round-off Errors

Any result of a finite precision computation is subject to rounding errors. Consequently, this result that appears to be reasonable may therefore contain errors, and it may be difficult to judge how large the error is. We show a simple example of round-off error below.

Example 2.1. If we have to compute the following operations on a typical calculator. First,

[1] Let us note that this thesis does not tackle these kind of numbers and thus we omit most details on them.

$x = \sqrt{2}$, then $y = x^2$ and finally $z = y - 2$, i.e, the result should be $z = (\sqrt{2})^2 - 2$, which obviously is 0. The result reported by the calculator is

$$z = -1.38032020120975 \times 10^{-16} \ .$$

Let us note that when operands of arithmetic operations have themselves been subject to previous rounding, catastrophic loss of significant digits may happen and consequently the result may be completely false. While these errors are individually small, they propagate through a computation and can make its results meaningless [Hal95, Pat92]. To estimate the round-off error of floating-point computations, many analysis techniques and tools have been proposed in the bibliography. We discuss these tools in Chapter 3 Section 3.1.

Verification using a Higher-Precision Arithmetic

We recall that our goal is to automatically compute the minimal number of bits needed for the variables and intermediary results of programs in order to accomplish the user requirement of accuracy. The most obvious and maybe the simplest way to validate that our output precision satisfy the user defined error constraints is to perform a comparison with the result of an equivalent computation performed with a higher precision arithmetic. Although the IEEE754 Standard specifies fixed formats only, several software tools exist for multiple-precision floating-point arithmetic, for example, `MP` [Bre78], `GMP` [GT15], `Pari-GP` [BBB+98], etc. For our comparisons, we are going to generate programs that use arbitrary-precision version with the GNU `MPFR`[2] library, short for Multiple Precision Floating-Point Routines [FHL+07]. Chapters 8 and 9 highlight our experimentation with the `MPFR` library.

2.1.2 Fixed-Point Arithmetic

Our tool and technique are also generalisable to support fixed-point arithmetic. In fact, one limitation of the FP arithmetic is that it requires dedicated support, either in hardware or in software, and depending on the application, this support may be too costly. However, the fixed-point arithmetic is an alternative which can be implemented with integers only.

A fixed number of digits is assigned to the sign, integer and fractional parts of the number within the data type format. As integer data types can be signed or unsigned, the sign field can be omitted also in fixed-point numbers. This is the case of unsigned fixed-point numbers, which represent the absolute value of the real number defined in Equation (2.2). Note that the binary point in fixed-point representation and the number of bits of each part are fixed. Thus, the scale factor of the associated data is constant and the range of the values that can be represented does not change during the computation.

$$(-1)^{sign} \times integer \cdot fractional \tag{2.2}$$

Let us also mention that many implementations of the fixed-point arithmetic use a two's complement representation instead of Equation (2.2).

[2] https://www.mpfr.org/

To be brief, Figure 2.1 presents the general representation of a number in fixed-point format composed of a sign bit s (the most significant bit) and $b-1$ bits divided between the integer and the fractional parts. m and n represent the position of the radix point respectively to the most significant bit (MSB) and to the least significant bit (LSB).

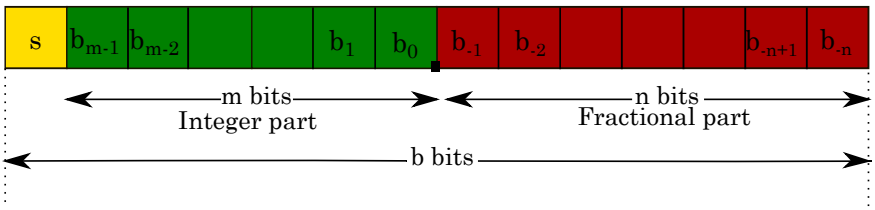


Figure 2.1: Fixed-point representation of a signed number.

Let us mention that there exists tools [DYSD14, LHD+10, MRS08] that apply the conversion from floating-point to fixed-point. Such methods offer the ability to trade-off the algorithm exactness for a more efficient implementation, providing either analytic or profile-based methods to obtain tight bounds on the numerical precision of fixed-point implementations. Compared to these approaches, the strength of our method is to find directly the minimal number of bits needed at each control point to get a certain accuracy on the results. Consequently, applying our method directly in hardware implementation on FGPAs and for implementation on processors with no floating-point hardware units is a very feasible task to explore in the near future.

2.1.3 Interval Arithmetic

Traditionally, guaranteed computations have been performed with *interval arithmetic*. The term "interval arithmetic", also known as interval analysis was formalized by Ramon E. Moore in the 1960s [Moo79] to bound rounding errors in mathematical computations. The theory of interval analysis emerged considering the computation of both the exact solution and the error term as a single entity, i .e. the interval.

In our thesis, we do not work only on scalar values but on intervals instead. An interval, denoted by $[x]$, is supposed to be closed and bounded non-empty set as shown in Equation (2.3) where $\underline{x}$ and $\overline{x}$ are called lower and upper bounds respectively. We have:

$$[x] = [\underline{x}, \overline{x}] = \{y \in \mathbb{R} | \underline{x} \leqslant y \leqslant \overline{x}\} \ . \tag{2.3}$$

Those intervals are added, subtracted, multiplied, etc., in such a way that each computed interval $[\underline{x}, \overline{x}]$ is guaranteed to contain the real value x of the corresponding variable in the exact computation which is being approximated. For example, if x and y are known to lie in the intervals $[\underline{x}, \overline{x}] = [2, 4]$ and $[\underline{y}, \overline{y}] = [-3, 2]$, then the sum $[\underline{x}, \overline{x}] + [\underline{y}, \overline{y}] = [2-3, 4+2] = [-1, 6]$ and the product $[\underline{x}, \overline{x}] \times [\underline{y}, \overline{y}] = [4 \times (-3), 4 \times 2] = [-12, 8]$.

A limitation of the interval arithmetic is that it introduces over-approximations for longer computations because it cannot track correlations between variables. For instance, if

we compute $x - x$ with $x \in [\underline{x}, \overline{x}] = [2, 5]$, the result produces $[2-5, 5-2] = [-3, +3]$ instead of $[0, 0]$. This loss of correlation was partially addressed by the affine arithmetic [dFS04].

In fact, affine forms [dFS04, GGP09], whose geometrical representation are zonotopes, are widely used in software verification [GMP02, GPV12] (more details will be provided in Chapter 3). Like other relational domains [CH78, Min06], they make it possible to obtain more precise results than the usual interval arithmetic by recording linear relations between variables.

2.1.4 Definitions and Properties

We remind the reader that our technique is independent of any of the particular computer arithmetic described in Section 2.1. In fact, we manipulate numbers for which we know the unit in the first place denoted by ufp, and the number of significant digits, denoted by nsb. We also assume that the constants occurring in the source codes are exact and we bound the errors introduced by the finite precision computations. In the following, we denote by $\mathsf{ufp}_e(x)$ and $\mathsf{nsb}_e(x)$ respectively the ufp and nsb of the error on a number x (note that $\mathsf{nsb}_e(x)$ may be infinite in some cases). These functions are defined hereafter and a more intuitive presentation is given in Figure 2.2.

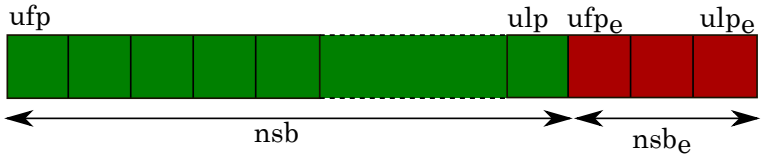


Figure 2.2: Schematic representation of ufp, nsb and ulp for values and errors.

Definition 2.1 (Unit in the First Place)**.** The unit in the first place of a real number x, denoted by $\mathsf{ufp}(x)$, and possibly encoded up to some rounding mode by a floating-point or a fixed-point number is given in Equation (2.4). This function is independent of the representation of x:

$$\mathsf{ufp}(x) = \begin{cases} \min\{i \in \mathbb{Z} : 2^{i+1} > |x|\} = \lfloor \log_2(|x|) \rfloor & \text{if } x \neq 0, \\ 0 & \text{if } x = 0. \end{cases} \tag{2.4}$$

■

Remark 2.1. We remark in Equation (2.4) that if $x = 0$ we have $\mathsf{ufp}(x) = 0$. In fact, a less pessimistic formula would be $\mathsf{ufp}(x) = e_{min}$ if it is defined or $\mathsf{ufp}(x) = -\infty$ if we assume that the range of exponents is unbound.

Definition 2.2 (Number of Significant Bits)**.** Intuitively, $\mathsf{nsb}(x)$ is the number of significant bits of x. Let $\hat{x}$ the approximation of x in finite precision and let $\varepsilon(x) = |x - \hat{x}|$ be the absolute error. Following Parker [Par97], if $\mathsf{nsb}(x) = k$, for $x \neq 0$, then

$$\varepsilon(x) \leqslant 2^{\mathsf{ufp}(x)-k+1} \ . \tag{2.5}$$

In addition, if $x = 0$ then $\mathsf{nsb}(x) = 0$. For example, if the exact binary value 1.0101 is approximated by either $x = 1.010$ or $x = 1.011$ then $\mathsf{nsb}(x) = 3$.

■

Definition 2.3 (Unit in the Last Place)**.** The unit in the last place of a number x denoted by $\mathsf{ulp}(x)$ is defined below in Equation (2.6). It depends on the unit in the first place $\mathsf{ufp}(x)$ and the number of significant bits $\mathsf{nsb}(x)$:

$$\mathsf{ulp}(x) = \mathsf{ufp}(x) - \mathsf{nsb}(x) + 1 \ . \tag{2.6}$$

■

Definition 2.4 (Computation Errors)**.** The number of significant bits of the computation error on x is denoted by $\mathsf{nsb}_\mathsf{e}(x)$. It is used to optimize the carry bit function denoted by ξ which can propagate throughout the computations. In order to compute this quantity, we need to compute the unit in the first place of the error on x which is given by

$$\mathsf{ufp}_\mathsf{e}(x) = \mathsf{ufp}(x) - \mathsf{nsb}(x) \ . \tag{2.7}$$

We assume that there is no error on any constant c arising in programs, i.e. $\mathsf{nsb}_\mathsf{e}(c) = 0$. Nevertheless, the nsb_e of the results of elementary operations may be greater than 0. For instance, if we add two constants c_1, c_2 in x such that $\mathsf{ufp}_\mathsf{e}(c_1) \geqslant \mathsf{ufp}_\mathsf{e}(c_2)$ then $\mathsf{nsb}_\mathsf{e}(x) = \mathsf{ufp}_\mathsf{e}(c_1) - (\mathsf{ufp}_\mathsf{e}(c_2) - \mathsf{nsb}_\mathsf{e}(c_2))$ which corresponds to the nsb of the resulting error (see Figure 2.2). Therefore, we denote by $\mathsf{ulp}_\mathsf{e}(x)$ the unit in the last place of the computation error on x and it is defined by

$$\mathsf{ulp}_\mathsf{e}(x) = \mathsf{ufp}_\mathsf{e}(x) - \mathsf{nsb}_\mathsf{e}(x) + 1 \ . \tag{2.8}$$

■

Let us note that these functions (equations (2.4), (2.6), (2.7) and (2.8)) will be used to describe the error propagation through the computations. This error modelling is performed by abstract interpretation which we highlight in the next section.

2.2 Static Analysis by Abstract Interpretation

Static program analysis aims at automatically determining whether a program satisfies some particular properties such as "the program never dereferences a null pointer", "the program never divides by zero", "the user-specified assertions are never violated", etc. Abstract interpretation [CC77a, CH78, CC92] provides the mathematical theory to design such analysis. It consists of a general theory for approximating the behavior of programs, developed by Patrick Cousot and Radhia Cousot in the late 1970s, as a unifying framework for static program analysis. Abstract interpretation gathers the concepts necessary to build an approximate static analysis.

In the past decade, abstract interpretation-based static analyzers began to have an impact in real-world software development. This is the case, for instance, of the static analyzer `Astrée` [CCF+05] which is used daily by industrial end-users in order to prove the absence of run-time errors in embedded synchronous C programs. We shed light on these tools later in Chapter 3.

In this section, we introduce the notions required by abstract interpretation, such as partial order, lattices but also the notion of fixpoint. Then, we define more precisely the concept of semantics of a program and abstract domains. Finally, we present mechanisms of acceleration of convergence such as the widening and narrowing before introducing the policy iteration technique which we will use in Chapter 9 to find a solution to our refined system of constraints.

A full description of the theory of abstract interpretation is available in [CC77a] and in Miné's tutorial [Min17].

2.2.1 Order Theory

In the following, we briefly recall well-known mathematical concepts in order to describe abstract interpretation as proposed in [CC77a, CC92]. First, we need to introduce standard definitions related to relations, partial order sets, lattices and functions.

Definition 2.5 (Relation)**.** A binary relation $\mathcal{R}$ between two sets $\mathcal{A}$ and $\mathcal{B}$ is a subset of the Cartesian product $\mathcal{A} \times \mathcal{B}$. We often write $x \mathcal{R} y$ for $(x, y) \in \mathcal{R}$. We present hereafter some important properties which may hold for a binary relation $\mathcal{R}$ over a set $\mathcal{S}$:

- $\forall x \in \mathcal{S} : x \mathcal{R} x$ (reflexivity),
- $\forall x \in \mathcal{S} : \neg(x \mathcal{R} x)$ (irreflexivity),
- $\forall x, y \in \mathcal{S} : x \mathcal{R} y \Rightarrow y \mathcal{R} x$ (symmetry),
- $\forall x, y \in \mathcal{S} : x \mathcal{R} y \wedge y \mathcal{R} x \Rightarrow x = y$ (anti-symmetry),
- $\forall x, y, z \in \mathcal{S} : x \mathcal{R} y \wedge y \mathcal{R} z \Rightarrow x \mathcal{R} z$ (transitivity),
- $\forall x, y \in \mathcal{S} : x \mathcal{R} y \vee y \mathcal{R} x$ (totality).

■

Definition 2.6 (Partial Order, Poset)**.** A Partial order $\sqsubseteq$ on a set $\mathcal{P}$ is a relation $\sqsubseteq \in \mathcal{P} \times \mathcal{P}$ that is reflexive, anti-symmetric and transitive. A partial order set, or poset, $(\mathcal{P}, \sqsubseteq)$ is a set $\mathcal{P}$ equipped by a partial order $\sqsubseteq$.

■

For instance, for any set of elements $\mathcal{S}$, the set of its parts, $\mathcal{P}(\mathcal{S})$ with the inclusion partial order $\subseteq$ is a poset. A convenient way of representing a poset is by using a Hasse diagram. In such a diagram, elements of $\mathcal{P}$ are nodes organized such that the greater elements are higher whereas the edges represent the order relation between element.

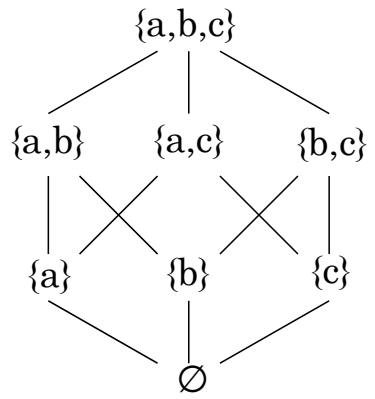


Figure 2.3: Hasse diagram for the partial order set of $(\mathcal{P}(a,b,c),\subseteq)$.

Example 2.2. Figure 2.3 depicts a Hasse diagram of $(\mathcal{P}(X),\subseteq)$ where $X = \{a,b,c\}$.

Definition 2.7 (Lower and Upper Bounds)**.** Let $(\mathcal{P},\sqsubseteq)$ be a poset, and $\mathcal{S} \subseteq \mathcal{P}$. An element $u \in \mathcal{P}$ is an *upper bound* of $\mathcal{S}$ if $\forall s \in \mathcal{S}$, $s \sqsubseteq u$. The element u is the *least upper bound*, or *join*, of $\mathcal{S}$, denoted by $\sqcup\mathcal{S}$, if $u \sqsubseteq u'$ for each upper bound u' of $\mathcal{S}$. Similarly, the element $l \in \mathcal{P}$ is a *lower bound* of $\mathcal{S}$ if $\forall s \in \mathcal{S}$, $u \sqsubseteq s$. The element l is the *greatest lower bound*, or *meet*, of $\mathcal{S}$, denoted by $\sqcap\mathcal{S}$, if $l' \sqsubseteq l$ for each lower bound l' of $\mathcal{S}$.

■

Example 2.3. In the poset of Figure 2.3, $\{a,b,c\}$ is an upper bound and $\varnothing$ is a lower bound for all the other subsets. Since the elements $\{a,b,c\}, \{a,b\}, \{a,c\}$ and $\{a\}$ are upper bounds of $\{a\}$ and $\{a\}$ is in relation with all of them then we deduce that $\{a\}$ is the least upper bound and the greatest lower bound at the same time.

Definition 2.8 (Chain and Complete Partial Order)**.** Let $(\mathcal{P},\sqsubseteq)$ be a poset. A *chain* $\mathcal{C} = (x_i)_{i\in\mathbb{N}}$ is a monotone sequence of elements of $\mathcal{P}$: $x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq x_{n-1} \sqsubseteq \cdots$. A *complete partial order (CPO)* is a poset $(\mathcal{P},\sqsubseteq)$ such that $\mathcal{P}$ has a least element $\bot$ and every chain $\mathcal{C}$ has a least upper bound $\sqcup\mathcal{C}$.

■

Definition 2.9 (Lattice)**.** A lattice $(\mathcal{P},\sqsubseteq,\sqcup,\sqcap)$ is a poset where each pair of elements $a, b \in \mathcal{P}$, has a least upper bound denoted by $a \sqcup b$, and a greatest lower bound denoted by $a \sqcap b$. If it exists, the least element $\sqcup\mathcal{P}$ is denoted $\bot$, called *bottom*. The greatest element $\sqcap\mathcal{P}$ is denoted $\top$, called *top*.

■

Example 2.4. An integer interval lattice can be constructed as follows:

$$(\{[a,b]|a,b \in \mathbb{Z}, a \leqslant b\} \cup \{\bot\}, \subseteq, \sqcup, \cap) \ .$$

Here, the set of integers $[a,b]$ is comprised between a and b with $a \leqslant b$. The smallest element $\bot$ represents the empty set $\varnothing$. The partial order is $\subseteq$ and $\cap$ is the greatest lower bound, as intervals are closed under intersection. However, they are not closed under set union, hence, the least upper bound can be defined as $[a,b] \sqcup [a',b'] = [\min(a,a'), \max(b,b')]$, while $\exists x : x \sqcup \bot = \bot \sqcup x = x$. Indeed, $[a,b] \sqcup [a',b']$ computes the smallest interval containing intervals $[a,b]$ and $[a',b']$.

Definition 2.10 (Complete Lattice)**.** A complete lattice $(\mathcal{P}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is a poset such that for any subset of $\mathcal{P}$, possibly infinite, the least upper bound and the greatest lower bound are defined. Noting that in a complete lattice $\top$ and $\bot$ always exist.

■

Example 2.5. The integer interval lattice of Example 2.4 is not a complete lattice as the infinite family of intervals $\{[0,i] \mid i \geqslant 0\}$ has no least upper bound.

Definition 2.11 (Monotonic Function)**.** Let $(\mathcal{P}_1, \sqsubseteq_1)$ and $(\mathcal{P}_2, \sqsubseteq_2)$ be two posets. A function $f : \mathcal{P}_1 \to \mathcal{P}_2$ is said to be *monotonic* if and only if:

$$\forall x, y \in \mathcal{P}_1, x \sqsubseteq_1 y \Rightarrow f(x) \sqsubseteq_2 f(y) \ . \tag{2.9}$$

■

Definition 2.12 (Continuous Function)**.** Let $(\mathcal{P}_1, \sqsubseteq_1)$ and $(\mathcal{P}_2, \sqsubseteq_2)$ be two posets. A function $f : \mathcal{P}_1 \to \mathcal{P}_2$ is said to be *continuous* if it preserves existing least upper bounds of chains that is, for each chain $\mathcal{C} \subseteq \mathcal{P}_1$, if $\sqcup\mathcal{C}$ exists then we have:

$$f(\sqcup\mathcal{C}) = \sqcup\{f(x)|x \in \mathcal{C}\} \ . \tag{2.10}$$

■

2.2.2 Fixpoints

Once a language includes loops, the analysis needs to build a loop invariant, holding before entering the loop and after each iteration, upon re-entering the body. This invariant corresponds to a *fixpoint*. In the following, we recall the fundamental theorems due to Alfred Tarski [Tar55] and Stephen Cole Kleene [Bir67].

Definition 2.13 (Fixpoint Computation)**.** Let $(\mathcal{P}, \sqsubseteq, \sqcup, \sqcap)$ be a lattice and let $F : \mathcal{P} \to \mathcal{P}$. An element $x \in \mathcal{P}$ is called a *fixpoint* of F if $F(x) = x$. Similarly, it is called a *pre-fixpoint* if $x \sqsubseteq F(x)$, and a *post-fixpoint* if $F(x) \sqsubseteq x$. If there exists, the *least fixpoint* of F, denoted by $\mathsf{lfp}(F)$, is a fixpoint of F such that, for every fixpoint $x \in \mathcal{P}$ of F, $\mathsf{lfp}(F) \sqsubseteq x$. The *greatest fixpoint* of F denoted by $\mathsf{gfp}(F)$ is defined similarly.

■

Theorem 2.1 (Tarski's Fixpoint Theorem)**.** The set of fixpoints of a monotonic function $F : \mathcal{P} \to \mathcal{P}$ over a complete lattice is also a complete lattice.

Proof. By Tarski in [Tar55]. ■

In particular, Tarski's theorem implies that a monotonic function among a complete lattice has a least fixpoint lfp.

Theorem 2.2 (Kleene's Fixpoint Theorem)**.** Let $(\mathcal{P}, \sqsubseteq)$ be a CPO and let $F : \mathcal{P} \to \mathcal{P}$ be a continuous function. Then F has a least fixpoint such that

$$\mathsf{lfp}(F) = \sqcup\{F^i(\bot) \mid i \in \mathbb{N}\} \ . \tag{2.11}$$

Proof. Found in [Cou78]. ■

```
void main(){
   int x = 0;          // 1
   while(x < 100) {    // 2
     x = x + 1;        // 3
   }                   // 4
}
```

$x_1 = [0, 0]$
$x_2 =]-\infty, 99] \cap (x_1 \cup x_3)$
$x_3 = x_2 + [1, 1]$
$x_4 = [100, +\infty[\cap (x_1 \cup x_3)$

Figure 2.4: A simple integer loop and its semantic equations.

Example 2.6. Considering the program in the left hand side of Figure 2.4. The corresponding semantic equations in the lattice of intervals are given in the right hand side of the figure. The intervals $x1, ..., x4$ correspond to the control points 1 to 4 indicated as comments in the C code. The purpose is to look for a fixpoint of the function F given by the right hand side of these semantic equations. The standard Kleene iteration sequence is eventually constant after 100 iterations, reaching the least fixpoint as depicted in Figure 2.5.

2.2.3 Concrete and Abstract Semantics

Abstract interpretation is a fundamental theory and a practical framework for the realistic approximation of the semantics of programs. This section explains how to ensure that an approximated semantics is correct according to a concrete semantics.

Concrete Semantics

Verifying whether some property holds on a program requires explaining what the program actually does. This is what we call the *semantics* of the program. A language semantics,

iteration 1:

$$\begin{aligned} x_2^1 &=]-\infty, 99] \cap ([0,0] \cup \bot) \\ &= [0,0] \\ x_3^1 &= [0,0] + [1,1] \\ &= [1,1] \\ x_4^1 &= [100, +\infty[\cap ([0,0] \cup [1,1]) \\ &= \bot \end{aligned}$$

iteration 2:

$$\begin{aligned} x_2^2 &=]-\infty, 99] \cap ([0,0] \cup [1,1]) \\ &= [0,1] \\ x_3^2 &= [0,1] + [1,1] \\ &= [1,2] \\ x_4^2 &= [100, +\infty[\cap ([0,0] \cup [1,2]) \\ &= \bot \end{aligned}$$

iteration $i+1 (i < 100)$**:**

$$\begin{aligned} x_2^{i+1} &=]-\infty, 99] \cap ([0,0] \cup [1,i]) \\ &= [0,i] \\ x_3^{i+1} &= [0,i] + [1,1] \\ &= [1,i+1] \\ x_4^{i+1} &= [100, +\infty[\cap ([0,0] \cup [1,i]) \\ &= \bot \end{aligned}$$

Figure 2.5: Kleene's iterations of the program of Figure 2.4.

defined as a precise mathematical characterization of program executions, is generally not computable. Such semantics is called *concrete* semantics. Let us note that the correctness of the static analysis of the program is expressed with respect to the concrete semantics.

In fact, there are many mathematical models for describing the behavior of a program. They fall broadly into three categories: *denotational*, *axiomatic*, and *operational*. A denotational semantics formalizes the meaning of the language syntax through mathematical objects, called denotations. An axiomatic semantics establishes logical implications between assertions valid before a statement and assertions valid after it. The assertions are logical predicates describing the program states. An operational semantics closely describes the behavior of a construct execution through a transition system between the program states. Transitions can be defined as atomic execution steps (in small-step semantics), or inductively as sequences of computational steps (in big-step semantics). However, it would be wrong to view these three categories of semantics as in opposition of each other: they are equivalent and have each their uses.

This thesis uses the operational semantics to prove the correctness concerning the soundness of our analysis. Chapter 4 describes the language of the input programs from which we generate semantic equations in order to determine the least precision needed for the program numerical values. The small-step operational semantics of our programs is highlighted in Chapter 5.

For an in-depth description and a comparison between these semantics, we refer the reader to The Formal Semantics of Programming Languages by Winskel [Win93].

Abstract Semantics and Domains

The gist of abstract interpretation is to reason on an over-approximation of the semantics, also called *abstract* semantics. It is defined as an approximated characterization of programs executions and determined thanks to an *abstract domain*. An abstract domain approximates sets of invariants, so that they are representable and computable. The function that links each abstract state to the set of concrete states it represents is called the *concretization* of the domain. Similarly, the function that links each concrete state to the set of abstract states it represents is called the *abstraction* of the domain.

Broadly speaking, we distinguish two families of abstract domains:

1. *Non-relational* abstract domains which refer to domains where relations (comparisons) between variables are forgotten. This includes the sign domain, the constant and constant set domains, the interval domain and finally the congruence domain.

2. *Relational* abstract domains which are able to discover relationships between variables. We cite the polyhedra [CH78], zonotopes [GGP09], the octagons [Min06] as relational domains.

For the sake of conciseness, we omit the details on these domains in this thesis. However, we refer the reader to [CH78, Min06] for a good survey. In the following, we present the principle definitions essential to establish the relationships between concrete and abstract worlds.

Definition 2.14 (Abstract Domain)**.** Let $\mathcal{D}$ be the set of elements to be abstracted. $\mathcal{D}$ is called *concrete domain*. An *abstract domain* over the set $\mathcal{D}$ is the pair $(\mathcal{D}^\sharp, \gamma)$ where

- $\mathcal{D}^\sharp$ is an ordered set of elements called *abstract elements*,
- The *concretization function* $\gamma : (\mathcal{D}^\sharp, \sqsubseteq^\sharp) \to (\mathcal{D}, \sqsubseteq)$ is a monotonic function associating to each abstract element of $\mathcal{D}^\sharp$ a concrete element of $\mathcal{D}$.

■

Example 2.7. The abstraction of the interval arithmetic that we have presented in Section 2.1.3 is considered as the most popular abstract domain. The interval domain abstracts the set of possible values of a variable $v \in Var$, where Var is the set of variables, as an interval. It is denoted by $\mathcal{D}^\sharp = Var \to \mathbb{I}^\sharp$ where $\mathbb{I}^\sharp$ denotes the set of intervals and the abstract values are either non-empty intervals with finite or infinite bounds, or $\bot$:

$$\mathbb{I}^\sharp = \{[a,b] : a,b \in \mathbb{Z} \cup \{-\infty\} \cup \{+\infty\},\ a \leqslant b\} \cup \{\bot\}\ . \tag{2.12}$$

The concretization function of the abstract domain of intervals given above is:

$$\gamma([a,b]) \ = \ \{x \in \mathbb{Z} \mid a \leqslant x \leqslant b\}$$

$$\gamma(\bot) \ = \ \varnothing$$

As defined in [CC77a], we assume additionally the existence of a monotonic function called *abstraction function* $\alpha : (\mathcal{D}, \sqsubseteq) \rightarrow (\mathcal{D}^\sharp, \sqsubseteq^\sharp)$ that associates to each concrete element an abstract one such that (α, γ) forms a *Galois connection*:

Definition 2.15 (Galois Connection)**.** Let ($\mathcal{D}$,$\sqsubseteq$) and ($\mathcal{D}^\sharp$,$\sqsubseteq^\sharp$) be two posets. Let ($\mathcal{D}$,$\sqsubseteq$) be the concrete domain and let ($\mathcal{D}^\sharp$,$\sqsubseteq^\sharp$) be the abstract domain. A *Galois connection* denoted by $(\mathcal{D}, \sqsubseteq) \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} (\mathcal{D}^\sharp, \sqsubseteq^\sharp)$, is a pair of monotonic functions $\alpha : \mathcal{D} \rightarrow \mathcal{D}^\sharp$ and $\gamma : \mathcal{D}^\sharp \rightarrow \mathcal{D}$ such that:

$$\forall d \in \mathcal{D}, d^\sharp \in \mathcal{D}^\sharp : \alpha(d) \sqsubseteq^\sharp d^\sharp \iff d \sqsubseteq \gamma(d^\sharp) \ . \tag{2.13}$$

■

Example 2.8. Following up on Example 2.7, in order to define the Galois connection, we present hereafter the abstraction function alongside the concretization function already defined. we have:

$$\alpha(X) = \begin{cases} \bot & \text{if X= } \varnothing, \\ [\min X, \max X] & \text{otherwise.} \end{cases}$$

Let us note that an abstract semantics on an abstract domain can be defined through functions that over-approximate the concrete semantics of the language statements. Such functions are called *transfer functions*. We introduce the transfer functions of the language of our input programs in Chapter 4.

2.2.4 Widening and Narrowing

We have seen that the abstract interpretation theory has the objective of automatically proving properties of computer programs, by computing invariants that over-approximate the program behaviors [BSC12]. These invariants are defined as the least fixpoint of a system of semantic equations and the most famous approach for computing it is Kleene fixpoint computation already defined in Theorem 2.2. However, these equations may require a large number of iterations to be solved. Especially when the abstract domain is infinite or simply disproportionate, the convergence may be extremely slow. Moreover, some abstract domains used in real-world analyzers do not have a complete lattice structure such as the affine forms [dFS04, GGP09]. To accelerate the convergence, Cousot and Cousot [CC77b] have introduced the *widening* and *narrowing* mechanisms. The iteration mechanism starts from the least fixpoint of the abstract domain, then it performs a sequence of computations using the abstract transfer functions of the program. Widening operators are then used while computing the iterates to enforce or accelerate the convergence of increasing iteration sequences over abstract domains with infinite or very long strictly ascending chains, or even over finite but very large abstract domains. A widening operator is defined as follows:

Definition 2.16 (Widening)**.** Let ($\mathcal{D}$,$\sqsubseteq$) be a poset. A widening operator $\triangledown : (\mathcal{D} \times \mathcal{D}) \rightarrow \mathcal{D}$ is such that:

1. $\forall x, y \in \mathcal{D}$, we have $x \sqsubseteq x \bigtriangledown y$ and $y \sqsubseteq x \bigtriangledown y$;
2. for all increasing chains $x_0 \sqsubseteq x_1 \sqsubseteq \ldots \sqsubseteq x_n \sqsubseteq \ldots$, the increasing chain

$$y_0 = x_0$$

$$\forall n \in \mathbb{N}, y_{n+1} = y_n \bigtriangledown x_{n+1}$$

is ultimately stationary, $\exists l \geqslant 0 : \forall j \geqslant l : y_j = y_l$.

■

Example 2.9. Taking again Example 2.7, the widening operator $\bigtriangledown$ is defined by:

$$[a,b] \bigtriangledown [a',b'] = [c,d] \quad with \quad c = \begin{cases} -\infty & \text{if } a' \leqslant a \\ a & \text{otherwise} \end{cases} et \quad d = \begin{cases} +\infty & \text{if } b \leqslant b' \\ b & \text{otherwise} \end{cases}.$$

Narrowing helps to recover precision lost by widening steps. It is used to enforce or accelerate the convergence of decreasing iteration sequences. It is defined as follows:

Definition 2.17 (Narrowing)**.** Let ($\mathcal{D}$,$\sqsubseteq$) be a poset. A narrowing operator $\bigtriangleup : (\mathcal{D} \times \mathcal{D}) \to \mathcal{D}$ is such that:

1. for all element $x, y \in \mathcal{D}$, if $x \sqsupseteq y$ we have $x \sqsupseteq (x \bigtriangleup y) \sqsupseteq y$;
2. for all decreasing chains $x_0 \sqsupseteq x_1 \sqsupseteq \ldots \sqsupseteq x_n \sqsupseteq \ldots$, the decreasing chain

$$y_0 = x_0$$

$$\forall n \in \mathbb{N}, y_{n+1} = y_n \bigtriangleup x_{n+1}$$

is ultimately stationary, $\exists l \geqslant 0 : \forall j \geqslant l : y_j = y_l$.

■

Example 2.10. The narrowing operator $\bigtriangleup$ on the interval domain of Example 2.7 is:

$$[a,b] \bigtriangleup [a',b'] = [c,d] \quad with \quad c = \begin{cases} a' & \text{if } a = -\infty \\ a & \text{otherwise} \end{cases} et \quad d = \begin{cases} b' & \text{if } b = +\infty \\ b & \text{otherwise} \end{cases}.$$

Example 2.11. Following up on Example 2.6. Instead of the iterations of Kleene, the fixpoint can be obtained in a faster way by using the classical widening and narrowing operators already defined. In practice, when the parameter of the least fixpoint solver is equal to 10, the iteration sequence using widenings and narrowings takes 12 iterations as shown in Figure 2.6.

Widening steps:

$$\begin{aligned} x_2^{10} &= [0,9] \triangledown [0,10] \\ &= [0,\infty[\\ x_3^{10} &= [0,\infty[+[1,1] \\ &= [1,\infty[\\ x_4^{10} &= [100,\infty[\cap([0,0] \cup [1,\infty[) \\ &= [100,\infty[\end{aligned}$$

Narrowing steps:

$$\begin{aligned} x_2^{11} &= [0,\infty[\triangle[0,99] \\ &= [0,99] \\ x_3^{11} &= [0,99] + [1,1] \\ &= [1,100] \\ x_4^{11} &= [100,\infty[\cap([0,0] \cup [1,100]) \\ &= [100,100] \end{aligned}$$

Figure 2.6: Widening and narrowing steps of the example of Figure 2.4

.

2.2.5 Policy Iteration

Since a decade, another approach was introduced in the static analysis community to perform over-approximation and to achieve better precision than widening-based tools, called *policy iteration* [AGG12a, CGG+05, GGTZ07, RG15]. The idea of policy iteration was first introduced by Howard [How60] to solve stochastic control problems with finite state and action space. Next, this method was extended to stochastic games by Hoffman and Karp [HK66]. Policy iterations basically perform iterations with two phases:

- *Compute a policy*, that is a locally simplified version of the fixpoint problem;
- *Solve the policy* with efficient tools specialized for this simpler problem.

Noting that these two phases are alternatively performed until a good result is reached. To compute the least fixpoints, two different approaches have been proposed: the *min-policy iteration* [AGG12a, CGG+05] and the *max-policy iteration* [GS07a, GS07b]. To some extent, the min-policy iteration [AGG12a] can be seen as a very efficient narrowing, since they perform descending iterations from a post-fixpoint towards some fixpoint, working in a way similar to the Newton-Raphson numerical method [Atk91]. Iterations are not guaranteed to reach a fixpoint but can be stopped at any time leaving an over-approximation thereof. Moreover, convergence is usually fast. The max-policy iteration [GS10] work in the opposite direction compared to min-policy iteration. They start from bottom and iterate computations of greatest fixpoints on a set of max-policies until a global fixpoint is reached. Unlike the previous approach, this terminates with a theoretically precise fixpoint, but the user has to wait until the end since intermediate results are not over-approximations of a fixpoint.

In the context of static analysis, the use of policy iteration, to compute the least fixpoint of a self-map F was introduced in [CGG+05]. We state in Algorithm 1 a very general policy iteration algorithm. The first step is to describe F as the minimum (or the maximum) of a set Π of a simpler maps as shown in Equation (2.14) hereafter.

$$F = \inf_{\pi \in \Pi} f^{\pi} \tag{2.14}$$

Algorithm 1: Policy Iteration Algorithm in Static Analysis

Let $k = 0$, select $\pi^k \in \Pi$;
while $x \neq F(x)$ **do**
  Compute the least fixpoint x_k of (f^{π^k}) s. t. $f^{\pi^k}(x_k) = x_k$;
  Evaluate $F(x_k)$;
  if $F(x_k) = x_k$ **then**
    return x_k;
  else
    Select π^{k+1} s. t. $F(x_k) = f^{\pi^{k+1}}(x_k)$;
    $k = k + 1$.
  end
end

A *policy* (or *strategy*) is a selection of an element of Π . Next, the least fixpoint of this element is computed: the algorithm terminates if this fixpoint is a fixpoint of F, in other words $x_k = F(x_k)$. Otherwise, the algorithm iterates and a new strategy which reaches x_k is selected. The algorithm correctness stems for the principle of *selection property*. In fact, the set of policy maps need to satisfy a selection property which ensures that the minimal fixpoint of the original system of equations is the minimum of the fixpoints of the policies.

Definition 2.18 (Selection Property)**.** Let Π be the set of policies, let $(\mathcal{P}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ be a complete lattice and let F be a monotone self map. We say that F satisfies the selection property if:

$$\forall x \in \mathcal{P}, \exists \pi \in \Pi \, \text{s. t.} \, F(x) = f^{\pi}(x) \ .$$

■

In intervals (see [CGG+05]), policies are of four types: ll, rr, lr and rl where l is for "left" and r for "right". For an intersection between two intervals $I = [-a, b]$ and $J = [-c, d]$, we associate the following policies:

- $ll(I, J)$: we choose $-a$ as the lower bound and b as the upper bound,
- $lr(I, J)$: we choose $-a$ as lower bound and d as upper bound,
- $rl(I, J)$: we choose $-c$ as lower bound and b as upper bound,
- $rr(I, J)$: we choose $-c$ as lower bound and d as upper bound.

Thus, the intersection of the two intervals I and J is given by the following formula:

$$I \cap J = ll(I, J) \cap rr(I, J) \cap lr(I, J) \cap rl(I, J) \ . \tag{2.15}$$

Let us note that an important issue in the case of policy iteration algorithms in static analysis is the choice of the initial policy, since some choices may lead eventually to a fixpoint which is not minimal. Therefore, there is a number of heuristics that one might choose concerning

the initial policy as mentioned in [CGG+05]: if a finite constant bound appears in I or J, this bound is selected. Moreover, if a $+\infty$ upper bound or $-\infty$ lower bound appears in I or J, then, this bound is not selected, unless no other choice is available (choices that give no information are avoided). In the case where the application of these rules is not enough to determine the initial policy, the bound arising from the left hand side term is selected, meaning that if $I = [-a, \infty[$ then the initial policy of $I \cap J = lr(I, J)$ which keeps the lower bound of I and the upper bound of J.

Example 2.12. Let us take again Example 2.6 of the program depicted in Figure 2.4. The heuristics for the initial policy are presented in the left hand side of Figure 2.7. The least fixpoint is reached after one Kleene iteration as we can observe in the right hand side of the figure.

$$\begin{array}{l} x_1 = [0,0] \\ x_2 =]-\infty, 99] \cap (x_1 \cup x_3) \;\rightarrow\; [\min(x_1 \cup x_3),\ 99] \\ x_3 = x_2 + [1,1] \\ x_4 = [100, +\infty[\cap (x_1 \cup x_3) \;\rightarrow\; [100,\ \max(x_1 \cup x_3)] \end{array} \xrightarrow{\texttt{1 iteration}} \begin{array}{l} x_1 = [0,0] \\ x_2 = [0,99] \\ x_3 = [1,100] \\ x_4 = [100,100] \end{array}$$

Figure 2.7: Policy iteration method on the program of Figure 2.4.

In our thesis, we will present a new policy iteration algorithm for the problem of precision tuning. In Chapter 5, we will show that finding the minimal solution (number of bits) to our system of constraints is equivalent to compute the least fixpoint of these equations. In particular, the policy iteration will be used as an optimization approach to solve a refined system of constraints. To compute the least fixpoint, we proceed similarly as in [GGTZ07] by using Linear Programming (LP) for computation. In the next section, we highlight the different methods that we use to solve the different systems of constraints generated by our tool.

2.3 Solvers

Once the semantic equations are generated, they need to be solved. In our work, we are interested in two kinds of solvers: *Satisfiability Modulo Theories (SMT)* and *Linear Programming (LP)*. Let us recall that our goal is not to present an exhaustive list of the existing solvers but to introduce the basic terminologies of these theories.

2.3.1 Satisfiability Modulo Theory

Satisfiability is one of the most fundamental of problems in theoretical computer science, namely the problem of determining whether a formula expressing a constraint has a solution [BHvMW09]. In addition to static program analysis, constraint satisfaction problems

arise in many other diverse areas notably software and hardware verification, puzzles such as Sudoku, type inference, test-case generation, graph problems, etc. Many of these problems can be encoded by Boolean formulas and solved using *Boolean satisfiability (SAT)* solvers, where the goal is to decide whether a formula over Boolean variables, formed using logical connectives, can be made "true" by choosing "true/false" values for its variables. Other problems are more naturally described in more richer logic for example *first-order logic*, including the theory of equalities and uninterpreted functions, array theory, bit-vector and floating-point arithmetic, difference logic, and linear and non-linear arithmetic. Such problems can be handled by solvers for *theory satisfiability* or *Satisfiability Modulo Theories (SMT)*. It is well-known that SAT is considered as a NP-complete problem and that and certain classes of formulas accepted by SMT solvers belong to higher complexity classes or are even undecidable. This has not refrained researchers from looking for algorithms that, in practice, solve many relevant instances at reasonable cost [Mon16].

Principle In practice, SMT solvers (e.g., `Z3` [dMB08][3], `MathSAT`[4] [CGSS13], `CVC` [BCD+11] [5], etc.) combine the ability of SAT solvers to find solutions for complex propositional formulas with the ability of specialized theory solvers to find solutions to systems of constraints with respect to specific first order theories. A first-order formula may contain negations ($\neg$), conjunctions ($\wedge$), disjunctions ($\vee$) and quantifiers ($\exists, \forall$). If a formula is satisfiable, the SMT solver may provide a *model* of this satisfaction: an interpretation for the variable, function and predicate symbols that makes the formula "true". In the following, we present some definitions helpful to understand how SMT solvers work.

Definition 2.19 (Propositional Logic)**.** A propositional formula ϕ can be a propositional variable p or a negation ϕ_0, a conjunction $\phi_0 \wedge \phi_1$, a disjunction $\phi_0 \vee \phi_1$, or an implication $\phi_0 \Rightarrow \phi_1$ of smaller formulas ϕ_0, ϕ_1. A truth assignment M for a formula ϕ, denoted $M \models \phi$, maps the propositional variables in ϕ to $\{\bot, \top\}$. A given formula ϕ is *satisfiable* if there is a truth assignment M such that $M \models \phi$ under the usual truth table interpretation of the connectives. If $M \models \phi$ for every truth assignment M, then ϕ is valid. A propositional formula is either valid or its negation is satisfiable.

■

Example 2.13. $\forall\, x, y \in \mathbb{R}$, the formula given hereafter has no solution:

$$(x \leqslant 0 \vee x + y \leqslant 0) \wedge y \geqslant 1 \wedge x \geqslant 1 \ .$$

If we omit $x \geqslant 1$ then the solutions include the values $x = 0$ and $y = 1$.

[3] https://github.com/Z3Prover/z3
[4] https://mathsat.fbk.eu/
[5] https://cvc4.github.io/

Definition 2.20 (Litteral)**.** A *literal* is either a propositional variable p or its negation $\neg p$. The negation of a literal p is $\neg p$, and the negation of $\neg p$ is just p. A *formula* is a clause if it is the iterated disjunction of literals of the form $l_1 \vee \cdots \vee l_n$ for literals l_i, where $1 \leqslant i \leqslant n$. A formula is in *conjunctive normal form* (CNF) if it is the iterated conjunction of clauses $\tau_1 \wedge \cdots \wedge \tau_m$ for clauses τ_i, where $1 \leqslant i \leqslant m$.

■

SAT solving The principles of modern SAT solving have their origin in the 1960 procedure of Davis and Putnam [DP60], as simplified in 1962 by Davis, Logemann, and Loveland [DLL62]. Hence, most of the SMT solvers follow the *DPLL(T)* architecture.

The first step in the Davis—Putnam—Logemann—Loveland (DPLL) procedure is to convert the formula to CNF by introducing new variables to label the subformulas. A formula can be converted to clausal form by introducing fresh variables for each compound subformula and adding suitable clauses, e.g., in converting $\neg p \vee (\neg q \wedge r)$, we label $\neg q \wedge r$ as b and $\neg p \vee b$ as a to obtain the clauses $a, a \vee p, a \vee \neg b, \neg a \vee \neg p \vee b, b \vee q \vee \neg r, \neg b \vee \neg q, \neg b \vee r$. The DPLL algorithm tries to build a satisfying truth assignment using three main operations: **decide**, **propagate** and **backtrack**:

- The operation **decide** chooses an unassigned propositional variable and assigns it to "true" or "false".
- The operation **propagate** deduces the consequences of a partial truth assignment using deduction rules.
- The operation **backtrack** exists if there are conflicting clauses: given a partial truth assignment M and a clause C in the CNF formula ϕ such as all literals of C are assigned to false in M, then there is no way to extend M to a complete truth assignment M' that satisfies ϕ. We say this is a *conflict*, and then C is the *conflicting clause*. A conflict indicates that some of the earlier decisions cannot lead to a truth assignment that satisfies ϕ and the DPLL procedure must backtrack and try a different branch value. If a conflict is detected and there are no decisions to backtrack, then the formula ϕ is unsatisfiable (UNSAT).

We refer the reader to the Handbook of Satisfiability [BHvMW09] by Biere et al. and the survey done in [Mon16] for further information.

2.3.2 Linear Programming

Optimization problems are concerned with the efficient use or allocation of limited resources to meet desired objectives. These problems are characterized by the large number of solutions that satisfy the basic conditions of each problem [Gas85].

Depending on an objective that is implied in the statement of the problem, a particular solution to the problem considered as the best one is selected. As an illustration, a

manufacturing company must determine what combination of available resources will enable it to manufacture products in a way which not only satisfies its production schedule but also maximizes its profit. The basic conditions of this problem are the limitations of the available resources and the requirements of the production schedule. The objective of the problem, is the desire of a company to maximize the gain. Formally, a solution that satisfies all the constraints is called a *feasible solution*. Hence, a feasible solution that achieves the minimum value of the cost functional is said to be *an optimal feasible solution*. To connect with our work, the optimal feasible solution is the minimal number of bits required for each variable of the input programs.

Definition 2.21 (Linear Programming Problem)**.** A *Linear Programming problem* (LP) is the problem of minimizing (or maximizing) a linear function subject to a finite number of linear constraints. Given variables $\lambda = [\lambda_1, \cdots, \lambda_n]^T$, an LP problem in standard form can be given as shown

$$\begin{aligned} &\text{minimize (or maximize)} \quad Z(\lambda) = \sum_{j=1}^{n} c_j \lambda_j \\ &\text{subject to} \quad \sum_{j=1}^{n} a_{ij} \lambda_j = b_i \quad \forall i \in \{1, \cdots, m\} \\ &\qquad\qquad\qquad\qquad\qquad\qquad \text{and} \quad \lambda \geqslant 0 \ . \end{aligned} \tag{2.16}$$

where $Z(\lambda)$ is the linear objective function, $\sum_{j=1}^{n} a_{ij}\lambda_j = b_i \ (\forall i \in \{1, \cdots, m\})$ is the set of linear constraints, and a_{ij} and c_j are real coefficients. The linear constraints combined with a linear objective are called *Linear Programs* (LPs), and systems that solve them are called *LP solvers*. Note that if a constraint is not in the form of equality then we can add a non-negative variable, also called *slack variable*.

■

Integer Linear Programming (ILP) is a branch of LP that restricts all the variables of the model to be only integers. Similarly, the objective function in ILP problems is maximized or minimized subject to inequality and equality constraints and integrality restrictions on some or all of the integer variables. The computation of an optimal solution of an ILP is NP-hard; yet many large instances of such problems can be solved [CWM94].

In our thesis, we will explain in Chapter 5 how to formulate the problem of determining the lowest precision on variables and intermediary values in programs as an ILP problem by reasoning on their unit in the first place (ufp) (see Equation 2.4) and the number of significant bits (nsb) (see Definition 2.2) which are integer quantities. Also, we will prove that the integer solution to this problem is computed in polynomial-time by a classical LP solver.

LP solving There are many available methods for solving LP problems that are widely used in practice. Namely, *simplex method*, *interior point method* [Dik67] and *ellipsoid method* [Sho77]. The one used in this thesis is the simplex method which was proposed by Dantzig

in 1947 [Dan63, Dan90]. We distinguish two types of this algorithm: the *primal Simplex* and the *dual Simplex*. In our thesis, we are interested only in the primal simplex algorithm.

The simplex method has exponential-time complexity in the worst case, while both the interior point and ellipsoid algorithm are polynomial-time solvable. But in practice, the simplex algorithm is found be remarkably efficient and the run-time is often polynomial. Hence, it is widely used to solve LPs. The standard simplex algorithm contains two phases: the first phase looks for a feasible solution and the second searches the optimal one according to some criteria. This is made mathematically explicit by adding a linear objective function that is to be maximized. On the whole, the simplex method performs iteratively row operations on the simplex table. At each iteration, the method moves from a current basic feasible solution to another basic feasible solution which improves the objective function value. The method terminates when it cannot decrease the objective function value any more.

Difference between SMT solvers and LP solvers As we have mentioned in Section 2.3.1, SMT solvers combine SAT reasoning with specialized theory solvers either to find a feasible solution to a set of constraints or to prove that no such solution exists. In the other hand, LP solvers come from the tradition of optimization, and are designed to find feasible solutions that are optimal with respect to some optimization function. Let us note that many LP solvers are available. We cite in particular, `GLPK`[6] [Mak], `LP_SOLVE`[7] [BEN04], `CLP`[8] [LH03], `SCIP`[9] [Ach09] as open source solvers and `CPLEX`[10] [Cpl09], `Xpress`[11] [BBN16] and `Gurobi`[12] [Gur21] as commercial ones. Simplex-based LP solvers differ from SMT solvers in several important ways, including the following:

- LP solvers solve only conjunctions of constraints and thus they cannot handle arbitrary boolean combinations.
- LP solvers focus on both feasibility and optimization rather than just feasibility.
- LP solvers (generally) use floating-point rather than exact precision arithmetic internally.

Let us state that modern LP solvers incorporate highly sophisticated techniques, making them very efficient in practice [KBT14]. The techniques used in LP solvers have been extended to the optimization problems where all or some of the variables are required to be integers (*Integer Programming (IP)* and *Mixed Integer Programming (MIP)*). For an

[6] https://www.gnu.org/software/glpk/
[7] http://lpsolve.sourceforge.net/5.5/
[8] https://www.coin-or.org/
[9] https://scipopt.org/
[10] https://www.ibm.com/products/software
[11] https://www.fico.com/en/products/fico-xpress-optimization
[12] https://www.gurobi.com/

exhaustive presentation of the LP problems, we refer the reader to Linear Programming: Methods and Applications by Gass [Gas85].

2.4 Summary

In this chapter, we have presented an essential background helpful for understanding our technique for precision tuning. To sum up, we have started by presenting the IEEE754 Standard of floating-point arithmetic and the fixed-point arithmetic. Second, we have underlined the abstract interpretation theory on which our precision tuning approach is based. Finally, we have presented the SMT and LP paradigms that we use further to find the minimal number of bits of each variable with respect to the user accuracy requirement to our precision tuning problem.

We continue our survey in the upcoming chapter by presenting a complete study on the older and the recent trends of precision tuning techniques. We will also examine their strengths and shortcomings when comparing with our method.

Chapter 3

Background on Precision Tuning

***

3.1 Analysis and Verification Tools . . . 41
3.1.1 Static Analyzers by Abstract Interpretation . . . 41
3.1.2 General Static Analyzers . . . 42
3.1.3 Dynamic Analyzers . . . 43
3.2 Rewriting-Based Optimization Tools . . . 44
3.3 Precision Tuning Tools . . . 45
3.3.1 Static Analysis Tools . . . 46
3.3.2 Dynamic Analysis Tools . . . 47
3.4 Combining Tools . . . 55
3.4.1 Combining Tools for Analysis and Optimization . . . 55
3.4.2 Combining Tools for Rewriting and Tuning . . . 56
3.5 Summary . . . 57

Although users of High Performance Computing (HPC) are most interested in raw performance, both storage costs and power consumption have become critical concerns. This is due to several technological issues such as the power limitations of processors and the massive cost of communications which arise while executing applications on such architectures.

In recent years, precision tuning or customized precision 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 [CA20]. 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,

Architecture	Clock (GHz)	Peak FP32/ Peak FP64	Memory (MB)	Compiler
AMD Opteron 246	2.0	2	2048	Intel-9.1
IBM PowerPC 970	2.5	2	2048	IBM-8.1
Intel Xeon 5100	3.0	2	4096	Intel-9.1
STI Cell BE	3.2	14	512	Cell SDK-1.1

Table 3.1: Hardware and software properties of the systems used in [BBD+09].

as almost all numerical computations are performed using floating-point operations to represent real numbers [ANS08], 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 [BBD+09]. 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. Table 3.1 shows how the single precision is faster than double precision on some examples of hardware supports [BBD+09]. On AMD Opteron 246, IBM PowerPC 970, and Intel Xeon 5100, the single precision peak is twice the double precision peak. On the STI Cell BE, the single precision peak is fourteen times the double precision peak.

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 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.

In 2020, researchers from the Polytechnic University of Milan have presented a detailed survey on the existing state-of-the-art reduced-precision tools. Their survey has underlined the major advantages and drawbacks of the existing tools for precision tuning (our tool POP is newer than this survey) [CA20]. As a result, they have deduced that there exists very few tools in the state-of-the-art that process an efficient tuning of the programs. Meanwhile, the proposed tool for precision tuning in this thesis fills the majority of the gaps of the existing work. We point up the main features of POP in chapters 4 and 5.

The purpose of this chapter is to review the existing literature on techniques and tools concerning precision tuning. In addition, we extend our discussion to analysis, verification,

and transformation tools. The motivation of this discussion is that some precision tuning tools are extended from tools for error estimation, program rewriting and program verification. What makes our review different from exiting ones such as the floating-point analysis research community FPBench who maintains an online survey located at their webpage[1], and the survey done in [CA20], is that we go further by incorporating the most recent tools, at the time of writing this dissertation, and also examining the strengths and shortcomings of each tool in comparison to the newly developed tool in this thesis.

The remainder of this chapter is as follows. Section 3.1 discusses the state-of-the-art tools for error analysis and verification. Section 3.2 presents the optimizing tools by code transformation. An exhaustive survey about the precision tuning tools is given in Section 3.3. Section 3.4 discusses recent work that combine analysis, optimization and precision tuning tools. Section 3.5 concludes the survey.

3.1 Analysis and Verification Tools

As we have previously mentioned in Chapter 2, one significant problem of floating-point arithmetic is the presence of round-off errors that can make a numerical computation notably different from the actual real arithmetic computation. To deal with this issue, various analysis techniques and tools to estimate the round-off error of floating-point computations have been proposed in the literature. Although their focus is not on precision tuning, we mainly describe in this section tools that were used later by precision tuning tools and we will be brief on the other analysis tools that are not intended for reducing the precision of the program variables.

After taking a closer look at the behaviour of each tool, we divide these tools into three categories: static analyzers based on abstract interpretation, general static analyzers, and dynamic analyzers.

3.1.1 Static Analyzers by Abstract Interpretation

Many tools use abstract interpretation and semantics based approaches for the problem of analyzing floating-point programs. Astrée [CCF+05] is a commercial static analyzer tool[2] that proves the absence of runtime errors and invalid concurrent behavior in safety-critical software written or generated in C or C++. Moreover, let us mention Fluctuat [GMP02] another commercial static analyzer based on abstract interpretation. This latter tool uses a zonotopic abstract domain [GP11] that is based on affine arithmetic [dFS04]. Fluctuat accepts as input a C (or ADA) program with annotations about input ranges and uncertainties, and produces bounds for the round-off error of the program expressions decomposed with respect to its provenance. It has been used to verify safety-critical

[1] https://fpbench.org/community.html
[2] Astrée: https://www.absint.com/astree/index.htm

embedded systems and it is available on their official webpage[3]. More recently, researchers from the NIA[4] and NASA proposed the verification tool PRECiSA (Program Round-off Error Certifier via Static Analysis) [TFMM18]. It generates PVS[5] certificates that guarantee the correctness of the error bounds with respect to the floating-point IEEE754 Standard. Furthermore, given concrete ranges for the input variables of a program, the numerical estimations computed by PRECiSA are provably sound over-approximations of the possible round-off error that can occur in the program. The source code is available online at https://github.com/nasa/PRECiSA or via webpage[6]. An updated version of PRECiSA was recently presented in [STF+19].

3.1.2 General Static Analyzers

Prior work in static error analysis provides a foundation for rigorously determining what precision are required to meet error constraints for particular closed form equations. In this context, the Gappa tool [dDLM11] computes enclosures for floating-point expressions via interval arithmetic. This enclosure method enables a quick computation of the bounds, but sometimes it may result in pessimistic error estimations. This tool[7] also generates a proof of the results that can be checked in the Coq proof assistant[8]. Later, a framework called Real2Float [MCD17] was proposed. It aims at providing upper bounds on absolute round-off errors using semi-definite programming and sums of squares certificates. As in Gappa, the results can be checked using the Coq theorem prover. The source code of Real2Float is available at https://github.com/afd/real2float.

Additionally, a group at the University of Utah has proposed the FPTaylor tool [SJRG15]. This tool uses a method called *Symbolic Taylor Expansions* in order to estimate round-off errors of floating-point computations. It applies a global optimization technique to obtain tight bounds for round-off errors. 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 cannot be handled directly. Instead, it can be used as an external decision procedure for program verification tools such as [CKK+12]. The source code of the tool is available at https://github.com/soarlab/FPTaylor. Later, they have extended their work by performing a broad comparison of many error bounding analyses to ensure the mixed-precision tuning technique in a tool called FPtuner [CBB+17]. More highlights on this tool will be given in Section 3.3.

Other interesting static tools from the same group of the university of Utah are SATIRE [DBG+20] and FPDetect [DKB+20]. Not so far from the approach implemented in FPTaylor,

[3] Fluctuat: http://www.lix.polytechnique.fr/Labo/Sylvie.Putot/fluctuat.html
[4] National Institute of Aerospace: https://www.nianet.org/
[5] https://pvs.csl.sri.com/
[6] PRECiSA webpage: http://precisa.nianet.org/
[7] Gappa webpage: http://gappa.gforge.inria.fr/
[8] Coq webpage: https://coq.inria.fr/

SATIRE [DBG+20] (Scalable Abstraction-guided Technique for Incremental Rigorous analysis of round-off Errors) [DBG+20] sheds light on how scalability and bound-tightness can be attained through a combination of incremental analysis, abstraction, and judicious use of concrete and symbolic evaluation. It also supports scalable mixed-precision analysis and multi-output estimation of straight-line floating-point codes[9]. What distinguishes SATIRE from other tools is that its analysis scales to hundreds of thousands of operations(≈ 200K operators). The official repository is available at arnabd88/Satire. The second tool FPDetect [DKB+20] performs analytical error analysis for stencil codes with a view to bound the maximum observable error and then attribute any violations of this bound to either bit-flips or polyhedral compilation errors.

Unfortunately, most of these static analysis tools do not address conditional-based programs. For this reason, we believe that integrating our methods with these tools seems a promising task to explore. In fact, in future work we like to extend loops in the SATIRE tool by means of the policy iteration (PI) technique that can help to compute fixpoints of loops with conditionals [ABM21]. More highlights on the policy iteration technique will be given in Chapter 5.

3.1.3 Dynamic Analyzers

Researchers have proposed probabilistic methodologies to calculate the error bounds with a certain degree of confidence. In this context, The CADNA library[10](Control of Accuracy and Debugging for Numerical Applications) [JC08] enables one to control the numerical quality of any scientific code written in Fortran, C or C++. It estimates round-off errors by replacing floating-point arithmetic with *Discrete Stochastic Arithmetic* (DSA) [Vig04], essentially modeling error using randomized rounding (up or down) instead of round-to-nearest-even. Moreover, the Verificarlo [DdOCP16] and the Verrou [FL17] tools are based on probabilistic analysis. These two tools are based on *Monte Carlo Arithmetic* [Par97]. The Verificarlo tool, which source code available at github.com/verificarlo/verificarlo/tree/veritracer, instruments the LLVM-IR (intermediate representation) [LA04] of the program to substitute the IEEE754 floating-point arithmetic operations with equivalent instructions. However, the Verrou tool is based on the popular debugging tool Valgrind [NS07] and it works at binary level. The equivalent source code is available online at github.com/edf-hpc/verrou.

Other tools rely on *shadow value analysis*. The main goal of SHVAL (SHadow Value Analysis Library) is to empirically measure the error due to the data types used in the program to represent the real values using a larger-precision data type as reference. More precisely, SHVAL is able to instrument the executable code of the application to be analyzed. By doing so, it can trace the evolution of the variables at run-time. In particular, this shadow execution is used as a reference to compare the result of the original program. The official repository of SHVAL is available at https://github.com/crafthpc/shval.

[9] A straight-line code is a code without conditionals and loops

[10] https://www-pequan.lip6.fr/cadna/

The list of dynamic analyzers expands to include the Herbgrind[11] [SPLT18] tool, whose main strategy consists of inspecting floating-point operations and comparing them against a shadow execution based on the MPFR library [FHL+07] to identify the point in the program where floating-point errors start to spread.

Recent work: The recent dissertation work in [Dem21] proposes the Shaman library that uses operator overloading and error-free transformations to track numerical error through computations by replacing floating-point numbers with instrumented alternatives. Furthermore, the work done in [LJS+21] proposes a two-phase analysis that combines different program analyses to conditionally verify the absence of special values and cancellation errors in numerical kernels in large programs. The objective of the first approach is to infer the ranges of the kernel inputs automatically. The second phase utilizes a slightly adapted existing static and sound round-off error analysis already implemented in their tool Daisy [DIN+18] (detailed in Section 3.4) to verify the kernels.

The reader may have noticed our ambition to work on tools designed for error analysis, alongside our basic subject of precision tuning. Unfortunately, today's error analysis methods that are designed to span entire input ranges and produce tight error bounds are unable to handle expressions with more than a few hundred nodes (except for SATIRE [DBG+20]). They also cannot handle complex loop structures with nested conditional expressions owing to their inability to arrive at tight-enough loop invariants. For all these reasons, we believe that integrating our methods mainly the policy iteration technique (PI) can help to cover these limitations.

3.2 Rewriting-Based Optimization Tools

The main insight of the optimizing tools by code transformations (or rewriting) is to improve the accuracy of the floating-point computations done in numerical codes. Unlike the vast literature available on the error estimation tools, the optimizing tools by rewriting are not numerous. More precisely, the state-of-the-art covers Xfp [DKMS13], Herbie [PSWT15] and Salsa [Dam16].

In particular, Xfp [DKMS13] is a tool for fixed-point optimization. It selects the fixed-point implementation which minimizes the rounding error with respect to the floating-point one. Based on code rewriting, Xfp first rewrites the input expression into one which is equivalent under a real-valued semantics, but one which has a smaller round-off error when implemented in finite precision and which does not increase the number of arithmetic operations. It relies on a *genetic algorithm*[12] to search the vast space of possible evaluation orders efficiently.

Herbie [PSWT15] is a dynamic tool which automatically improves the accuracy of floating-point expressions. Its heuristic search estimates and localizes rounding errors using

[11] Herbgrind webpage: http://herbgrind.ucsd.edu/

[12] Genetic algorithms are heuristic search algorithms inspired by natural evolution

sampled points (rather than static error analysis), applies a database of rules to generate improvements, takes series expansions, and combines improvements for different input regions. The official repository of `Herbie` is accessible at `https://github.com/uwplse/herbie`.

`Salsa` is a static tool [Dam16] which was developed in a former dissertation work within the university of Perpignan (Laboratory of Mathematics and Physics) and it is a source-to-source optimizer written in Ocaml. Let us note that source-to-source optimization means that the tools emit the same programming language they accept as input. Based on abstract interpretation, `Salsa` features both intra-procedural and inter-procedural code rewriting optimizations for the C programming language.

Although these tools do not always provide precision tuning via precision allocation, combining the code rewriting and precision tuning could be a promising approach that can improve both the accuracy and speed of floating-point expressions. We shed light on this idea in Section 3.4.2.

3.3 Precision Tuning Tools

The last few years have seen a wealth of precision tuning tools as depicted in Figure 3.1. In this section, we discuss the strengths and shortcomings of each tool. Besides, the originality of `POP` will be cleared up by showing to what extent it can fill many gaps of the state-of-the-art work. After taking a closer look on the behaviour of each tool, we deduce the following classification:

- **Static analysis tools:** that extract additional knowledge from the program source code **without executing it with input data**.

- **Dynamic analysis tools:** that involve the profiling of the target application to extract pieces of information by running the original version of the program. Let us mention that the majority of the tools of this category are based on a **trial-and-error strategy**: they deduce knowledge on the program from the effects of the changes they apply during the precision allocation. On this basis, this category can be classified into:

  - **Search algorithm-based tools:** that implement search algorithms to explore the space of possible data types more efficiently.

  - **GPU applications-based tools:** that operate on GPU kernels by using programming languages such as OpenCL or CUDA.

  - **Other dynamic tools:** other general tools based on dynamic analysis using different algorithms.

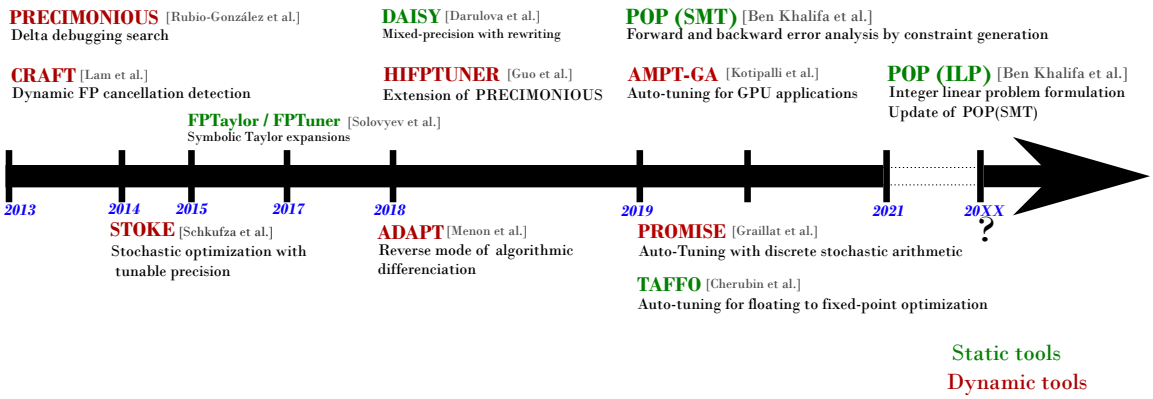


Figure 3.1: Summary of the precision tuning tools.

3.3.1 Static Analysis Tools

`Rosa` [DK17] 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. `Rosa` operates on a subset of the Scala programming language. In particular, the programmer writes the program in a real-valued specification language and makes numerical errors explicit in pre- and post-conditions. It is then up to `Rosa` to determine an appropriate data type which fulfills the specification and to generate the corresponding code. In addition, the common point between `Rosa` and the first version of `POP`[13] is that both internally exploit the `Z3` SMT solver [dMB08] to process the precision constraints derived from the program accuracy specifications. Also, `Rosa` handles conditional statements soundly but it assigns only uniform preicision to the variables of their programs. The source code of `Rosa` is available online at `https://github.com/malyzajko/rosa`.

Based on the *Symbolic Taylor Expansion* embodied in the `FPTaylor` [SJRG15] tool already described in Section 3.1.2, `FPTuner` [CBB+17] 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 to the one we will present in our thesis, 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. Unfortunately, it also requires a certain user skill for choosing which mixed-precision variants are more efficient and is thus not fully automated. Furthermore, its tuning time can be prohibitively large whereas we will show that the method embodied in our tool `POP` is more fast and efficient even in the case of large codes. `FPTuner` is available online at `github.com/soarlab/FPTaylor`.

In the context of precision tuning tool-chains, the `TAFFO` tool (Tuning Assistant for

[13] `POP` was initially based on an SMT-based method. Chapters 4, 5 and 6 highlight the tool.

Floating-point to Fixed-point Optimization) [CCC+20] is a LLVM-based tool-chain [LA04], which is packaged as a set of plugins for the `Clang` compiler[14]. 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. `TAFFO` is based on affine arithmetic [SF03]. This analysis is used to project on the output the error introduced by each fixed-point instruction. The advantage of `TAFFO` is that it supports both C and C++ programs and it can be provided as a plugin for LLVM which are feasible to be extended in `POP` in future work. 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. `TAFFO` is available online at `https://github.com/HEAPLab/TAFFO`.

3.3.2 Dynamic Analysis Tools

A considerable share of precision tuning tools are based on dynamic analysis. The main insight of these techniques is to lower the precision of the values in the program and observe the error on the output of a testing run. Consequently, their majority apply a trial-and-error paradigm to precision tuning. We distinguish tools which are based on search algorithms, tools oriented for GPU kernels and other general dynamic tools.

Search Algorithm-Based Tools

The first precision tuning reference tool based on search algorithm is the `Precimonious` tool [RNN+13]. Noting that `Precimonious` is the tool that interest us the most and we will present further in chapters 8 and 9 a full comparison on several benchmarks from both tools. For this reason, the next paragraph highlights the study of the `Precimonious` tool and its several extensions. The source code of `Precimonious` that we have installed to make our comparisons is available online at `https://github.com/corvette-berkeley/precimonious`.

Tool study: the Precimonious tool `Precimonious` [RNN+13] 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. `Precimonious` is based on the *delta-debugging* algorithm search [ZH02] 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. The *delta-debugging* algorithm has been originally conceived in the context of software testing for identifying the smallest failing test case. The input ingredients of this algorithm are the following:

- A list Δ of variables to be tuned as well as a list Δ' of all other variables of the program with their constant precision assignments.

[14] `https://clang.llvm.org/`

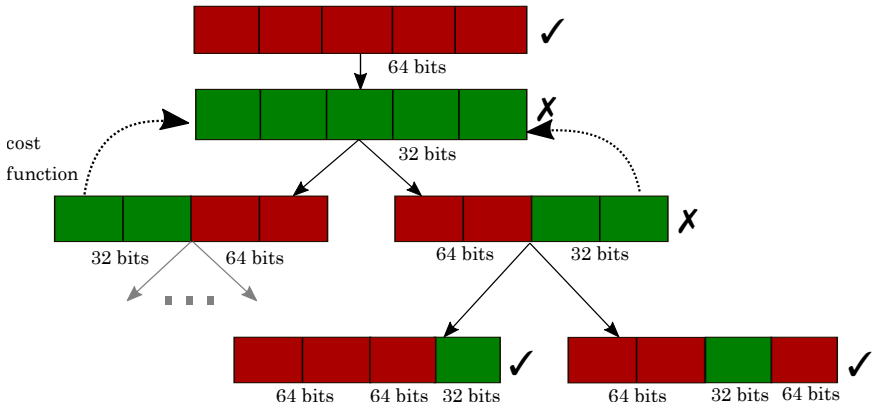


Figure 3.2: Illustration of the *delta-debugging* algorithm.

- An error function which bounds the round-off error of a given precision assignment.
- A cost function approximating the expected performance.
- An error bound $\varepsilon_{\max}$ to be satisfied.

The output of this algorithm is a precision assignment for variables in the set Δ. Let us note that the algorithm exhibits an $\mathcal{O}(n \log n)$ average complexity and an $\mathcal{O}(n^2)$ worst case complexity, where n is the number of elements to be tuned. Now, we explain the principle of the *delta-debugging* algorithm in the following example.

Example 3.1. As an illustration, we take the example depicted in Figure 3.2. In this example, we consider the case where variables can be in FP32 single precision and FP64 double precision. First of all, the *delta-debugging* algorithm starts by assigning all variables in Δ to the highest precision available, i.e. FP64. Next, it uses the error function to check whether the round-off error is below the error bound $\varepsilon_{\max}$. If it is not, then the optimisation is unsuccessful. Otherwise, the algorithm tries to lower all variables in Δ by assigning them to FP32 single precision. As before, it computes the maximum round-off error. If it is below $\varepsilon_{\max}$, the search stops as single precision is sufficient. If the error check does not succeed, the algorithm splits the change set Δ into almost equal size subsets which are pairwise disjoint. In our example, two equally sized lists Δ_1 and Δ_2 are obtained and then the algorithm recurses on each separately. When recursing on Δ_1, the new list of variables to lower becomes $\Delta_{new} = \Delta_1$ and the list of constant variables becomes $\Delta'_{new} = \Delta' + \Delta_2$. The case for recursing on Δ_2 is symmetric. Once a type assignment satisfies the error bound $\varepsilon_{\max}$, the recursion stops. Since several valid type assignments can be found, a cost function is used to select the one with lowest cost. Let us mention that the algorithm is generalized to several precision by first running it with the highest two precision. In the second iteration, the variables which have remained in the highest precision become constant and move to Δ'. The optimization is then performed on the new Δ considering the second and third

highest precision. While this algorithm helps in speeding up the search, this can still lead to a high number of builds and runs of the program.

Despite the fact that Precimonious can handle any program, including programs with loops, it presents several gaps. 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. By doing so, we can deduce that its approach is not sound and also in general inefficient, because of the large number of program executions needed for a reasonably confident error bound. Comparing to POP which takes several seconds per benchmark (not exceeding few minutes for large ones), Precimonious uses dynamic evaluation to estimate the expected running time. However, this approach is not entirely reliable as running times can vary substantially between runs. Last of all, Precimonious does not use any knowledge on the structure of the program to identify potential variables of interest. This latter limitation has made the subject of several work which extended Precimonious in several manners.

Blame Analysis [RNM+16] is a dynamic technique which 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. The source code of Blame Analysis can be found at https://github.com/corvette-berkeley/shadow-execution.

Another dynamic tool sharing some objectives and methodologies of Precimonious is called PROMISE[15] (PRecision OptiMISE) [GJP+19]. It is written in Python and it relies on the CADNA software [JC08] to implement the *Discrete Stochastic Arithmetic* (DSA) verification in C, C++, and Fortran program source code. PROMISE automatically modifies 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 [GR18b] is another extension of the Precimonious tool 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

[15] http://promise.lip6.fr/

inputs. Besides, It can be used to tune medium-sized programs only. HiFPtuner is available online at https://github.com/ucd-plse/HiFPTuner.

Similar search-based tools CRAFT(Configurable Run-time Analysis for Floating-point Tuning) [LHdSL13b, LHS13] 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. CRAFT relies on the well-established Dyninst binary analysis toolkit[16] to provide instrumented and mixed precision code. The original implementation of the CRAFT framework known as a binary mode version, relies on binary instrumentation and considers the whole program as its scope. Recently, its newer version is able to process the source code of the program and focus only on user-defined variables known as the variable mode version. Like Precimonious, CRAFT uses external description files (JSON or XML) to declare the variables and the data types to explore. While it uses heuristics to sample a fraction of the search space, it can be very time consuming even for very small programs. CRAFT source code is available at github.com/crafthpc/craft. Finally, a tool called fpPrecisionTuning [HMWA17] performs a search over the mixed-precision search space using a user-given error bound, but this tool uses MPFR [FHL+07] and source code modification to simulate non-standard precision. The source code of is available at github.com/minhhn2910/fpPrecisionTuning.

In summary, all the search-based approaches used for identifying valid mixed-precision configurations are time-intensive. In addition, their approaches does not scale with the number of possible number representations that can be used. Furthermore, discontinuities in the program can trick a greedy algorithm[17] into a local optimum, which may be considerably distant from the global optimum.

GPU Applications-Based Tools

Angerd et al. [ASS17] have described a framework for precision tuning for GPU applications. They investigate an approximation of floating-point values in computer graphics kernels using three different low-precision formats: the IEEE754 formats (specifically FP16 and FP32), the mantissa truncation in which the data types are obtained by truncating mantissa bits from the basic IEEE754 formats, and finally a dynamically selected exponent and mantissa width, which are data types with variable bit width but constant ratio between the number of mantissa bits and that of exponent bits. Angerd et al. [ASS17] extend LLVM with the custom defined data types and transparently converts the floating-point values. While these custom defined data types are not guaranteed to be supported by the target hardware, the proposed approach entails wrapping every memory access instruction to unpack and to pack the data from and to such data types. Consequently, this work is particularly pertinent

[16] https://github.com/dyninst/dyninst

[17] A greedy algorithm always takes the best immediate, or local, solution while finding an answer.

for architectures where the cache and the memory size are critical, e.g. HPC accelerators. Although the main focus of this framework operates on hardware-heterogeneity-aware programming languages, such as OpenCL, they process the whole computational kernel and do not satisfy any user accuracy requirement on the output.

Not too far from this work concept, the work described by Nobre et al. [NRB+18] presents a LARA-based approach [CCC+12] for precision-tuning. It takes an OpenCL kernel as input and it generates and evaluates multiple versions of the input kernel. Those versions exploit mixed-precision data types to achieve performance improvements over the original kernel, while they satisfy a user-defined constraint on the quality of the output. A similar work proposed by Rojek [Roj19] presents a machine-learning based method for the dynamic selection of the precision level for GPU computation. It implements a modified version of the random forest algorithm to decide whether a variable type should be in FP32 or FP64 floating-point (no other data types are considered). Broadly speaking, the tools proposed by Nobre et al. [NRB+18] and by Rojek [Roj19] operate on GPU kernels by using OpenCL/CUDA programming languages whereas Angerd et al. [ASS17] propose a GPU-oriented approach while working within the intermediate representation of the compiler.

Other tools perform a static data flow analysis with a dynamic profiling on the source code. In this setting, `AMPT-GA` [KSW+19b] is a tool oriented to GPU applications which combines static analysis for casting-aware performance modeling with dynamic analysis for enforcing precision constraints. Particularly, it performs a profile run of the application to identify the hottest computational kernels which may especially benefit from the precision reduction and a static analysis that only aims at identifying strongly connected variables in the dependency graph to attempt to assign the same data type to group of variables instead of acting on single variables.

`GPUMixer` [LWSB19] is one more tool designed for GPU kernels. This tool accepts CUDA kernels as input in the form of NVVM-IR intermediate representation[18], which can be generated by the `clang` compiler front-end, and replaces the FP64 floating-point operations with FP32. It decides whether to apply the conversion to a code region or not depending on a tunable metric, which is based on the ratio between the number of affected arithmetic instructions and the number of type cast instructions. However, the tool is limited by the fact that the NVIDIA CUDA C programming guide does not specify the cost of all GPU operations.

Other Notable Dynamic Tools

`Autoscaler for C` [KKS00] is a source-to-source compiler that complies with the ANSI C programming language. Its purpose is to convert every variable to fixed-point by using a data size which guarantees the absence of overflow. It performs an exploratory run over the original floating-point code to obtain an estimation of the dynamic range for each variable. This range estimation analysis is built upon the `Stanford University`

[18] NNVM is a reusable graph IR stack for deep learning systems

Intermediate Format (SUIF) compiler system [WFW+94]. Since the input and output of this translator are ANSI C compliant programs, it can be used for any fixed-point Digital Signal Processors (DSP) that supports ANSI C compiler. Another work not particularly different from the one implemented in [KKS00] is described in [CAL+17]. The proposed method automates the floating-to-fixed point conversion by re-targeting the existing source-to-source compiler GeCos framework (Generic Compiler Suite)[19], designed for use with hardware implementations, to produce code suitable for execution in HPC environments.

ADAPT [MLO+18] (Algorithmic Differentiation Applied to Precision Tuning) uses the reverse mode of algorithmic differentiation [Nau12] 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. As the algorithmic differentiation approach views a computer program as a composition of a sequence of arithmetic operations, ADAPT uses this data to capture the propagation of errors through the data flow graph of the computation. It performs aggregation and analysis on this data along with the original computation to determine the floating-point sensitivity of all the variables and operations in the program. Although the fact that ADAPT considers only IEEE754-compliant formats, it provides mixed-precision recommendations that satisfy a specified error threshold without requiring any search-based strategies. The official repository of ADAPT is available at github.com/llnl/adapt-fp. Later, ADAPT alongside CRAFT [LHdSL13b, LHS13] and another tool called Typeforge[20] have been incorporated in a framework named FloatSmith [LVMS19]. Broadly speaking, ADAPT provides the dynamic analysis of the code, CRAFT looks for the best precision tuning configuration, and Typeforge implements the source-to-source conversion for C/C++ code. FloatSmith is available online at github.com/crafthpc/floatsmith.

AMP [SKNS13, NAL+14] (Automated Mixed Precision) is a profile-driven tool that profiles applications to measure undesirable numerical behavior at the floating-point operation level. AMP takes a single precision application as input and its output is a mixed-precision application in which precision have been chosen to improve accuracy. The limitation of this tool is that it accepts only applications in which all operations are at the minimum precision. Otherwise, they should downgrade all the operations in higher precision (e.g. double precision) to single precision before applying their profiling. Indeed, although AMP monitor and locate numerical faults, it is infeasible to trace and quantify error propagation through every computational sequence of operations.

STOKE [SSA14] is a general stochastic optimization and program synthesis tool to handle floating-point computation. Beginning from floating-point binaries produced either by a production compiler or written by hand, the tool[21] shows that through repeated application of random transformations it is possible to produce high performance optimizations

[19] http://gecos.gforge.inria.fr

[20] Typeforge is a tool for type refactoring in C/C++ programs. It enables users to transform the type of any variables or operations. It guarantees the syntactic and semantic correctness of the generated code.

[21] STOKE webpage: http://stoke.stanford.edu/

that are specialized both to the range of live-inputs of a code sequence and the desired precision of its live-outputs.

More recently, a tool called PyFloaT [BII+20] has presented a methodology for tuning the precision of full fledged scientific applications written using multiple programming languages: Python, C++, CUDA and Fortran. It uses an instruction-centric analysis that uses call stack information and temporal locality to address the large scale of HPC scientific programs. The tool is available at https://github.com/hbrunie/PyFloT.

We end up the list of the precision dynamic tools by the approach proposed by Yesil et al. [YAK18] which is based on a proof concept for DPS (Dynamic Precision Scaling). The purpose of DPS is to run the program on reduced precision floating-point functional units whenever the data can tolerate the degradation, and to dynamically switch to the original floating-point data types when there is the need to preserve the accuracy. Moreover, we cite FloPoCo [dDP11] an open source C++ framework at http://flopoco.gforge.inria.fr/. It is a framework written in C++ that generates VHDL code to design custom arithmetic data path of floating-point cores. Also, it generates a synthetizable hardware description according to the parameters specified via C++ code.

The most challenging aspects of the precision tuning tools described in this section are outlined in Table 3.2. In particular, we report the implementation features of each tool such as:

- **Input language**: corresponds to the language supported by the tool (C, C++, CUDA, LLVM-IR, etc.) POP handles its own imperative language that we highlight in Chapter 4.

- **Output language**: corresponds to the language returned by the tool after finishing the tuning. We denote by "description" the tools that returns either files (JSON or XML) incorporating the reduced data types (if any) to the declared variables as done in Precimonious [RNN+13] and CRAFT [LHdSL13b, LHS13] or precision recommendations for variables that their data types should be optimized so as in ADAPT [MLO+18] and other tools.

- **Supported data types**: corresponds to the formats considered by each tool. For instance, "fixed" denotes the capability to deal with fixed-point representations, "IEE754" denotes generally the FP32, FP64 and FP128 floating-point formats. "custom" denotes the user-defined data types which are not guaranteed to be supported by a target hardware. Let us recall that our tool POP produces precision at bit-level. The advantage that we have in comparison with other tools is that with the returned number of bits, we are able to represent numbers in any computer arithmetic as well as with libraries implementing arbitrary precision arithmetic such as MPFR [FHL+07].

- **Framework**: corresponds to the architectures or bases each tool is built upon. For example, ADAPT [MLO+18] depends on CoDiPack (Code Differentiation Package)

Tool/Approach	Input Lang.	Output Lang.	Data Types	Framework	Analysis	Licensing
ADAPT [MLO+18]	C/C++, Fortran	description	FP32, FP64	CodiPack	dynamic	GNU GPL v3.0
AMPT-GA [KSW+19b]	C/C++	description	IEEE754	LLVM	dynamic	proprietary
Angerd et al. [ASS17]	LLVM-IR	description	FP32, custom	LLVM 3.5	dynamic	proprietary
AMP [SKNS13, NAL+14]	LLVM-IR	LLVM-IR	IEEE754	LLVM 3.4	dynamic	proprietary
Autoscaler for C [KKS00]	ANSI C	C++	fixed	SUIF	dynamic	proprietary
CRAFT [LHdSL13b, LHS13]	x86 binary, C/C++	description, C/C++	FP32, FP64	Dyninst	dynamic	GNU LGPL v3.0
FPTuner [CBB+17]	FPCore	FPCore	IEEE754	Gurobi 6.5	static	MIT
FloatSmith [LVMS19]	C/C++	description	FP32, FP64	CRAFT, ADAPT, TypeForge	dynamic	GNU GPL v3.0
fpPrecisionTuning [HMWA17]	C	MPFR	IEEE754, fixed	C2mpfr	dynamic	BSD, MIT
FloPoCo [dDP11]	C++	VHDL	custom	-	dynamic	proprietary
GPUMixer [LWSB19]	NVVM-IR	NVVM-IR	FP32, FP64	LLVM 4.0	dynamic	proprietary
HiFPTuner [GR18b]	LLVM-IR	description	FP32, FP64, FP128	Precimonious, LLVM 3.8	dynamic	BSD-3 Clause
Precimonious [RNN+13]	LLVM-IR	description	FP32, FP64, FP128	LLVM 3.0	dynamic	BSD-3 Clause
PROMISE [GJP+19]	C/C++	C/C++	FP32 and FP64	CADNA for C/C++	dynamic	GNU LGPL v3.0
PyFloaT [BII+20]	Python, C++ CUDA, fortran	description	IEEE754	GOTCHA	dynamic	MIT
Rojek [Roj19]	CUDA	CUDA	FP32 and FP64	–	dynamic	proprietary
Rosa [DK17]	Scala	Scala	IEEE754, fixed	Z3	static	BSD-2 Clause
STOKE [SSA14]	x86 binary	fixed x86 — 64	IEEE754, fixed	JIT assembler	dynamic	Apache2.0
TAFFO [CCC+20]	LLVM-IR	LLVM-IR	fixed	LLVM 8.0	static	MIT
POP [BMA19, ABM21]	**IMP**	**IMP+precision**	**bit-level**, **IEEE**754, **fixed** and **MPFR**	**ANTLR**, **Z3**, **GLPK**	**static**	**proprietary**

Table 3.2: Precision tuning tools properties.

which is s a tool for gradient evaluation in computer programs[22]. `FPTuner` [CBB+17] depends on `Gurobi v6.5`[23] which is a commercial optimization solver for linear programming (LP). The dash symbol "-" means that the tool is not based on any framework.

- **Type of analysis**: which are generally divided into static or dynamic analysis.
- **Licence**: corresponds to the rights under which each tool is released. For instance, `ADAPT` [MLO+18] is licensed under the GNU General Public License v3.0, `FPTuner` [CBB+17] is a free software released under the Massachusetts Institute of Technology (MIT) license and `HiFPTuner` [GR18b] is under the Berkeley Software Distribution License.

3.4 Combining Tools

The automated tools that we have discussed are typically complementary, each focusing on a distinct aspect of numerical reliability: error analysis tools, rewriting-based optimization and mixed-precision tuning. As a result, users will need to compose several to meet their development needs. In this section, we illustrate the benefits of composing complementary floating-point tools to achieve results neither tool provides in isolation. Our study reports two sorts of combining tools: **combining tools for analysis and optimization** and **combining tools for rewriting and tuning**.

3.4.1 Combining Tools for Analysis and Optimization

The first work that was interested in the combinations of tools is undoubtedly the `Daisy` tool [ID17, DIN+18, DHS18, DV19]. It provides in a single tool the main building blocks for accuracy analysis of floating-point and fixed-point computations which have emerged from recent related work. In particular, `Daisy` extends the approach implemented in `Rosa` [DK17] by integrating the rewriting capabilities of `Xfp` [DKMS13]. Additionally, it improves with respect to `Rosa` [DK17] by using a different SMT solver, dReal [GKC13] instead of `Z3`. `Daisy` integrates several techniques for sound analysis and optimization of finite-precision computations such as:

- Static dataflow analysis for finite-precision round-off error [DK14].
- `FPTaylor`'s optimization-based absolute error analysis [SJRG15].
- Support for mixed-precision and transcendental functions [DK11].
- Rewriting optimization [DKMS13].
- Interfaces to several SMT solvers (instead of the `Z3` SMT solver [dMB08] used with `Rosa` [DK17]) and code generation in Scala and C languages.

[22] CoDiPack library: `https://github.com/SciCompKL/CoDiPack`
[23] https://www.gurobi.com/

Let us note that Daisy 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. Unlike POP which is able to tune programs with expressions, loops, conditionals and even arrays, Daisy does not address conditional-based programs. The source code of Daisy is available open-source at https://github.com/malyzajko/daisy.

The authors of Daisy and Herbie have worked together to combine their tools in [BPDT18]. While Herbie optimizes the accuracy of straight-line floating-point expression, it employs a dynamic round-off error analysis and thus cannot provide sound guarantees on the results. Consequently, its combination with Daisy can help to check whether its unsound optimizations improved the worst-case round-off error or not. Meanwhile, this method do not handle loops and conditionals yet. This combination of tools is implemented as a script in the FPBench repository https://github.com/FPBench/FPBench.

3.4.2 Combining Tools for Rewriting and Tuning

The combining tools that interest us the most and may have a direct impact on our work are certainly those which combine mixed-precision tuning and rewriting. We recall that the main insight of rewriting techniques is to search through different evaluation orders to find one which minimizes the round-off error at no additional run-time cost. The mixed-precision tuning techniques aim to choose the smallest data type which still provides sufficient accuracy in order to save valuable resources like time, memory or energy. In this context, Anton is the first fully automated tool [DHS18] that combined these two techniques in one single tool. The rewriting step is inspired from the xfp tool [DKMS13]. For the mixed-precision tuning step, Anton uses a variation of the delta-debugging algorithm used by Precimonious [RNN+13]. It starts with all variables in the highest available precision and attempts to lower variables in a systematic way until it finds that no further lowering is possible while still satisfying the given error bound. Although Anton tried to reduce its search space by using a static sound error analysis as well as a static performance cost function, the technique is limited to rather small programs that can be verified statically.

Another study to find an efficient precision with a better accuracy of variables of programs was presented in [DM18]. It consists of the former work of Martel [Mar17] combined with the Salsa optimizing tool [Dam16]. The principle of this study is to apply the forward and backward error analysis by abstract interpretation approach [Mar17] to compute the least floating-point formats on the benchmarks of Salsa. Similarly to the Anton [DHS18] tool, their rewriting technique is performed before the mixed-precision tuning. Their results showed that the precision of the programs has been minimized by an average factor of 60% and also by optimizing the precision and the accuracy, an improvement of the execution time was also observed.

Only a few months ago, the developers of the optimizing tool Herbie have introduced

the Pherbie tool [SFN+21], short for Pareto Herbie[24]. The originality of this new tool is that it performs precision tuning and rewriting at the same time. Also, it adapts and extends techniques from Herbie to automatically generate a set of candidate implementations, and derive a Pareto-optimal accuracy versus speed trade-off, for a given floating-point expression. Pherbie implements precision tuning by introducing rewrites that cast candidate sub-expressions to different precision. As Herbie only considers implementations over a single uniform precision, Pherbie provides multi-precision implementations e.g. instead of a single sqrt operator, it provides $\texttt{sqrt}_{float32}$, $\texttt{sqrt}_{float64}$, $\texttt{sqrt}_{posit16}$, etc. Unlike our work, the analysis time of this tool can be exponential in the case of large programs containing a lot of expressions to rewrite which results in many new candidate implementations to manage and many calls to the Herbie tool.

3.5 Summary

We reviewed in this chapter research work related to the precision tuning tools. We have extended our review to the analysis and optimizing tools by code transformations. By and large, existing methods for precision tuning suffer from several limitations. As we have described in this chapter, the major drawback of the static analysis tools is fundamentally their incapacity to tune large codes with conditionals and loops. Nevertheless, a large amount of the dynamic tools follow a trial-and-error strategy by reducing the precision of arbitrary chosen variables and executing or analyzing statically the program to see the new accuracy. Besides, these tools are time and memory consuming.

In our thesis, we propose a new tool radically different from what have been presented before. POP implements a static analysis method relying on a modeling of the propagation of the errors throughout the code. Besides, we went further in the implementation task by integrating two variants of this method, each variant corresponds to a version of POP. The upcoming chapters 4 and 5 introduces these methods whereas Chapter 6 outlines the steps of construction POP.

[24] Pherbie is publicly available at https://herbie.uwplse.org.

Part II

A Static Precision Tuning Approach Based on Constraint Generation

Chapter 4

Transfer Functions

***

4.1 Language and Overview 63
4.2 Arithmetic Expressions 64
4.2.1 Forward Analysis 65
4.2.2 Backward Analysis 68
4.3 Elementary Functions 69
4.4 Carry Bit Function 71
4.5 Summary 72

In our review of the state of the art, we showed that the major limitation of the existing techniques is to use a trial-and-error approach, while they differ in the way of evaluating the accuracy of programs and reducing the exponential search space. The objective of this chapter is to introduce a new method for precision tuning, radically different from the state-of-the-art techniques and which belongs to the family of tools based on static analysis.

Our approach is based on a semantic modelling of the propagation of the numerical errors throughout the program source. This yields a system of constraints whose minimal solution gives the best tuning of the program. Based on a static analysis approach, we formulate the problem of precision tuning with two different methods. Each method expresses differently the set of constraints generated by our tool `POP` and consequently describes a contribution of this thesis. In this chapter, we note the first method as **SMT-based method** and Integer Linear Problem method or **ILP-based method** for the second one. These notations will be used all along this dissertation.

The SMT-based method combines a forward and a backward error analysis which are two popular paradigms of error analysis. The forward analysis is classical and it examines how errors are magnified by each operation in the program in order to determine the accuracy on the results. Next, a user accuracy requirement is given denoting the number of significant bits (`nsb`, see Definition 2.2) wanted at some control points for the

outputs. By taking into consideration the user assertions and the results of the forward analysis, the backward analysis is a complementary approach that starts with the computed answer to determine the exact precision on the inputs with respect to the desired accuracy. As could be expected, the forward and backward analysis can be handled iteratively to refine the results until a fixpoint is reached. Finally, these forward and backward transfer functions are expressed as a set of linear constraints made of propositional logic formulas and relations between integer elements only, checked by a SMT solver [DMB11]. The generation of constraints will be highlighted in Chapter 5. As a result, the tuned program is guaranteed to use variables of lower precision with a lower number of bits than the original program. This approach is implemented in the first version of our tool which is introduced further in Chapter 6. We note that the idea of the SMT-based method was first introduced by Martel in [Mar17].

The ILP-based method presents a relaxation of the system of constraints generated by the SMT-based method. It comes from the idea of changing the SMT-solver in order to solve the problem of precision tuning more efficiently. The originality of this technique is to generate an Integer Linear Problem (ILP) from the program source code. Basically, this is done by reasoning on the most significant bit (ufp given in Equation (2.4)) 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 the bit-level. We note that the ILP-based method relies on the same transfer functions (concrete semantics) defined in the present chapter. The only difference is that we apply a one-way analysis, mainly the backward analysis. With this approach we refine the number of constraints and variables generated. The ILP-based method is also embodied in our tool `POP` which will be described later in Chapter 6, while a detailed comparison between the two methods will be presented in Chapter 9. The main contributions of this chapter are:

- First, we define the transfer functions for the operations of the source program. What sets our tool apart from Martel's work is the extension of our language to the trigonometric functions, the square root function alongside to the four elementary operations for which we give the correctness proofs. As another extension of our tool, we accept programs with arrays and matrices.

- Second, we refine the carry bit function that can occur throughout the program computations. Before this thesis, previous methods [Mul05, Mar17] were too conservative in their analysis by assuming that a carry bit can be propagated at each operation of the program. Consequently, this function becomes very costly in large codes that perform a lot of computations and therefore the errors would be considerable. In this work, we define a more precise carry bit function in order to refine the obtained data types. This step is considered as a major contribution of our analysis.

We must admit that this chapter is extremely theoretical containing precise technical

definitions in order to describe the forward and backward transfer functions. The remainder of this chapter is organized as follows. Section 4.1 introduces the imperative language of our input programs. We define in Section 4.2 the forward and backward transfer functions for the arithmetic expressions, and, in Section 4.3, the elementary functions. Section 4.4 reports the proposed optimization on the carry bit that can occurs through the program computations. Finally, Section 4.5 concludes.

The content of this chapter is a revised version of the article [BMA19] published at the 7^{th} International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS) in 2019.

4.1 Language and Overview

In this section, we describe the language of the input programs from which we generate semantic equations in order to determine the least precision needed for the numerical values of programs. Let us state that `POP` handles loops, conditionals and arrays in contrast to the existing static tools for precision tuning already highlighted in Chapter 3.

Figure 4.1 defines the simple imperative language in which our input programs are written. We denote by Id the set of identifiers and by Lab the set of control points of the program used to assign to each element $e \in Expr$ and $c \in Cmd$ a unique control point $\ell \in Lab$ (an example is given in Listing 4.1). In Figure 4.1, in `c#p`, `p` indicates the initial number of significant bits of the constant c in the source code. The declaration of vectors is expressed by the statement `create_vector`(v,s)$^\ell$, while s denotes the size of the vector v. The declaration of a matrix m is expressed by the statement `create_matrix`(m,r,c)$^\ell$, while r and c denote respectively the number of rows and columns of the matrix. Next, the statement `require_nsb`(x,n)$^\ell$ indicates the minimal number of significant bits n that a variable x must have at a control point ℓ ($\mathsf{nsb}(x) = n$). The rest of the grammar is standard.

$$\ell \in Lab \quad x \in Id \quad \odot \in \{+, -, \times, \div\} \quad math \in \{\sin, \cos, \tan, \arcsin, \log, \ldots\}$$
$$\textbf{Expr} \ni \mathbf{e} : \mathrm{e} ::= \mathrm{c\#}p^\ell \mid x^\ell \mid e_1^{\ell_1} \odot^\ell e_2^{\ell_2} \mid math(e^{\ell_1})^\ell \mid sqrt(e^{\ell_1})^\ell$$
$$\textbf{Cmd} \ni \mathbf{c} : c ::= c_1^{\ell_1}; c_2^{\ell_2} \mid x =^\ell e^{\ell_1} \mid \texttt{while}^\ell\ b^{\ell_0}\ \texttt{do}\ c_1^{\ell_1} \mid \texttt{if}^\ell\ b^{\ell_0}\ \texttt{then}\ c_1^{\ell_1}\ \textbf{\textit{else}}\ c^{\ell_2} \mid$$
$$\texttt{create_vector}(v,s)^\ell \mid \texttt{create_matrix}(m,r,c)^\ell \mid \texttt{require_nsb}(x,n)^\ell$$

Figure 4.1: Language of input programs.

Now, we move to an overview of how we perform the tuning of a program written using the grammar of Figure 4.1. First of all, we assign to each control point ℓ two kinds of integer parameters: the *ufp* of the values and three integer variables corresponding to the forward, the backward and the final accuracies, denoted respectively by $\mathsf{nsb}_F(\ell)$, $\mathsf{nsb}_B(\ell)$ and $\mathsf{nsb}(\ell)$, so that the inequality in Equation (4.1) is always verified. Hence, we notice that in the forward mode, the accuracy decreases contrarily to the backward mode when we strengthen the post-conditions (accuracy increases).

$$0 \leqslant \mathsf{nsb}_B(\ell) \leqslant \mathsf{nsb}(\ell) \leqslant \mathsf{nsb}_F(\ell) \tag{4.1}$$

Remark 4.1. We must precise that Equation (4.1) is added only to the global system of constraints of the SMT-based method in which we combine a forward and a backward error analysis. We recall to the reader that when we formulate the problem as an ILP, we rather apply a one-way analysis instead of the forward and backward ones and consequently we will assign at each control point only the integer variable nsb. More details will be given in Chapter 5.

Example 4.1. Let us consider the simple C program of Listing 4.1. In this example, we suppose that all variables are in double precision before analysis (FP64). The original program is depicted in the left hand side of Listing 4.1.

Some points can be highlighted about this example. For instance, the variables `a` and `x` are initialized respectively to the values 1.0 and 0.0, annotated with their control points thanks to the following annotations $\mathtt{a}^{\ell_1} = \mathtt{1.0}^{\ell_0}$ and $\mathtt{x}^{\ell_5} = \mathtt{0.0}^{\ell_4}$ in the center part of Listing 4.1. As well, we have the statement $\mathtt{require_nsb(x,20)}^{\ell_{28}}$ which informs the tool that the user wants to get on variable `x` only 20 significant bits. We remind the reader that we consider that a result has n significants if the relative error between the exact and approximated results is less than 2^{-n}. As a consequence, the minimal precision needed for the inputs and intermediary results satisfying the user assertion is given on the right hand side of Listing 4.1.

Since, in this example, 20 bits only are required for `x`, the result of the addition $\mathtt{x} + \mathtt{a}$ also needs 20 accurate bits only. By combining this information with the result of the forward analysis, it is then possible to lower the number of bits needed for one of the operands. The tuned program is depicted in the right hand side of Listing 4.1.

In the sequel, we present the transfer functions of the different statements of our language that allow us to obtain the new optimized data types.

4.2 Arithmetic Expressions

In this section we refine the computations of the forward and backward transfer functions for the cases of the addition and the multiplication done in [Mar17]. The novelty of this work is that we are more precise in the propagation of the carry bits. We also present the forward and backward transfer functions for the subtraction and the division operations as well as for the trigonometric functions and the square root function. These functions are defined using the quantities ufp, ulp, nsb, nsb_e and ulp_e already defined in Chapter 2. Next, these functions will be formalized as a set of constraints made of propositional logic formulas and affine expressions among integers.

Simple C program:

```
1 a = 1.0;
2 i = 1.0;
3 x = 0.0;
4 while(i < 10.0){
5   a = a + 1.0;
6   x = x + a;
7   i = i + 1.0;
8 } ;
9   require_nsb(x, 20);
```

Program with labels:

```
1 a^ℓ1 = 1.0^ℓ0;
2 i^ℓ3 = 1.0^ℓ2;
3 x^ℓ5 = 0.0^ℓ4;
4 while (i^ℓ7 <^ℓ9 10.0^ℓ8){
5   a^ℓ14 = a^ℓ11 +^ℓ13 1.0^ℓ12;
6   x^ℓ20 = x^ℓ16 +^ℓ19 a^ℓ18;
7   i^ℓ25 = i^ℓ22 +^ℓ24 1.0^ℓ23;
8 }^ℓ26 ;
9   require_nsb(x, 20)^ℓ28;
```

Program annotated with precision:

```
1 a|19| = 1.0|19|;
2 i = 1.0;
3 x|20| = 0.0|20|;
4 while (i <10.0){
5   a|20| = a|19| +|20| 1.0
        |20|;
6   x|20| = x|20| +|20| a|20|
        ;
7   i = i + 1.0;
8 };
9 require_nsb(x, 20);
```

Listing 4.1: A simple example to show the nature of constraints generated. Left: source program. Center: program annotated with labels. Right: source program with inferred accuracies.

We remind that our technique is independent of a particular computer arithmetic. In fact, we manipulate numbers for which we know the unit in the first place (ufp), thanks to a range determination performed by dynamic analysis on the variables of our programs, and the number of significant digits (nsb) of the result which is given by the user of our tool.

4.2.1 Forward Analysis

We introduce the forward transfer functions corresponding to the addition $\overrightarrow{\oplus}$, subtraction $\overrightarrow{\ominus}$, multiplication $\overrightarrow{\otimes}$ and division $\overrightarrow{\oslash}$ of two numbers x and y. Each operation between these numbers whose result is the number z have respectively two parameters $\mathsf{nsb}(x)$, $\mathsf{nsb_e}(x)$, $\mathsf{nsb}(y)$, $\mathsf{nsb_e}(y)$, $\mathsf{nsb}(z)$ and $\mathsf{nsb_e}(z)$. We recall that the parameters $\mathsf{nsb}(x)$, $\mathsf{nsb}(y)$ and $\mathsf{nsb}(z)$ denote the number of significant bits for x, y and z respectively. For $\mathsf{nsb_e}(x)$, $\mathsf{nsb_e}(y)$ and $\mathsf{nsb_e}(z)$, we denote by these parameters the number of significant bits of the computation errors on x and y and their result z respectively (definitions previously detailed in Chapter 2). All these parameters will allow us to add a carry bit only when necessary (details are given in Section 4.4).

In addition, in distinction to [Mar17], we take into consideration in our analysis the truncation errors in order to be more precise through the computations due to the rounding of the operations [BMA19]. We denote the truncation errors by ε_+, ε_-, $\varepsilon_\times$ and $\varepsilon_\div$ for the addition, the subtraction, the multiplication and the division operations respectively. These errors depend on the precision of the operations. For example, if we perform an addition in the FP64 double precision, we obtain that $\varepsilon_+ \leqslant 2^{-53}$. The truncation error is defined hereafter in Definition 4.1.

Definition 4.1 (Truncation Error)**.** Let us consider the addition of two numbers x and y whose result of addition is z. As in the IEEE754 standard, the truncation error ε_+ for the rounding mode towards the nearest $\uparrow_\sim$ is given by $\varepsilon_+ \leqslant 2^{\frac{1}{2}\mathsf{ulp}(z)}$. Thus, we deduce from Equation (2.6) that $\mathsf{ulp}(z) = \mathsf{ufp}(z) - \mathsf{prec}(+) + 1$ where $\mathsf{prec}(+)$ presents the precision of

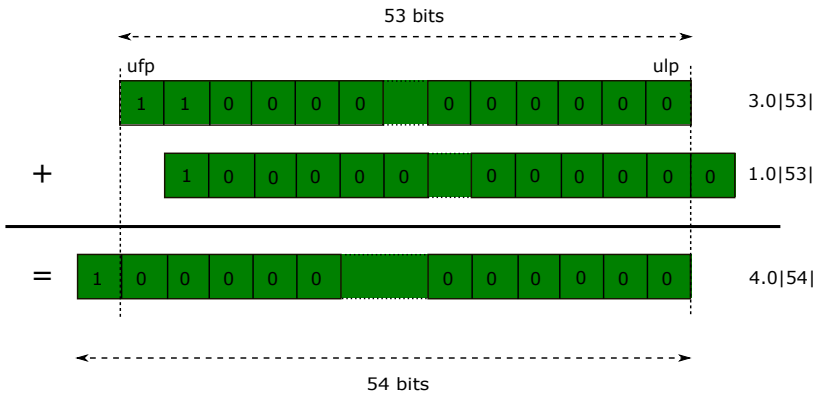


Figure 4.2: Example of forward addition: $3.0\,|\,53\,|\,+\,1.0\,|\,53\,|\,=\,4.0\,|\,54\,|$.

the operator $+$ given as shown:

$$\varepsilon_+ \leqslant 2^{\mathsf{ufp}(z)-\mathsf{prec}(+)} \ . \tag{4.2}$$

■

Now, we move to define the forward transfer functions for the addition, subtraction, multiplication and division respectively in definitions 4.2, 4.3, 4.4 and 4.5.

Definition 4.2 (Forward Addition)**.** Let x and y be two numbers whose result of addition is z. The forward addition $\overrightarrow{\oplus}$ is given as shown in Equation (4.3).

$$\overrightarrow{\oplus}(x,y) = z \ \text{ where } \ \mathsf{nsb}(z) = \mathsf{ufp}(x+y) - \mathsf{ufp}(2^{\mathsf{ufp}(x)-\mathsf{nsb}(x)+1} + 2^{\mathsf{ufp}(y)-\mathsf{nsb}(y)+1} + 2^{\mathsf{ufp}(z)-\mathsf{prec}(+)}) \ . \tag{4.3}$$

■

Intuitively, $\mathsf{nsb}(z)$ is the number of bits between $\mathsf{ufp}(z)$ and $\mathsf{ufp}_\mathsf{e}(z)$. Since $\mathsf{ufp}(z) = \mathsf{ufp}(x+y)$ and $\mathsf{ufp}_\mathsf{e}(z) = \mathsf{ufp}(2^{\mathsf{ufp}(x)-\mathsf{nsb}(x)+1} + 2^{\mathsf{ufp}(y)-\mathsf{nsb}(y)+1} + 2^{\mathsf{ufp}(z)-\mathsf{prec}(+)})$, we obtain Equation (4.3). In the forward addition case, we have $2^{\mathsf{ufp}(x)} \leqslant x < 2^{\mathsf{ufp}(x)+1}$ and $2^{\mathsf{ufp}(y)} \leqslant y < 2^{\mathsf{ufp}(y)+1}$ and the truncation error $\varepsilon_+ \leqslant 2^{\mathsf{ufp}(z)-\mathsf{prec}(+)}$. Thus, we assume that the total error on z given by ε_{z_+} is computed as $\varepsilon_{z_+} = \varepsilon_x + \varepsilon_y + \varepsilon_+$.

Example 4.2. Let $x = 3.0$, $y = 1.0$ and z the result of their addition. We assume that these variables are in FP64 double precision before analysis and that `require_nsb(z,23)`. The result of the forward addition has $\mathsf{nsb}_F = 54$ bits at bit-level since $\overrightarrow{\oplus}(3.0|53|, 1.0|53|) = 4.0|54|$. Figure 4.2 illustrates this example.

Definition 4.3 (Forward Subtraction)**.** Let x and y two numbers whose result of subtraction is z. The forward subtraction $\overrightarrow{\ominus}$ is given as shown in Equation (4.4).

$$\overrightarrow{\ominus}(x,y) = z \ \text{ where } \ \mathsf{nsb}(z) = \mathsf{ufp}(x-y) - \mathsf{ufp}(2^{\mathsf{ufp}(x)-\mathsf{nsb}(x)+1} - 2^{\mathsf{ufp}(y)-\mathsf{nsb}(y)+1} - 2^{\mathsf{ufp}(z)-\mathsf{prec}(-)}) \ . \tag{4.4}$$

■

Again, Equation (4.4) is obtained by computing $\mathsf{nsb}(z) = \mathsf{ufp}(z) - \mathsf{ufp}_{\mathsf{e}}(z)$. The same technique is also employed in equations (4.5) and (4.8).

Definition 4.4 (Forward Multiplication)**.** Let us assume a multiplication between two numbers x and y whose result is z. The forward multiplication $\overrightarrow{\otimes}$ is given as shown in Equation (4.5):

$$\begin{aligned}\overrightarrow{\otimes}(x,y) = z \text{ where } \mathsf{nsb}(z) = \mathsf{ufp}(x \times y) - \mathsf{ufp}(2^{\mathsf{ufp}(x)+1} \cdot 2^{\mathsf{ufp}(y)-\mathsf{nsb}(y)+1} \\ +2^{\mathsf{ufp}(y)+1} \cdot 2^{\mathsf{ufp}(x)-\mathsf{nsb}(x)+1} + 2^{\mathsf{ufp}(x)-\mathsf{nsb}(x)+1} \cdot 2^{\mathsf{ufp}(y)-\mathsf{nsb}(y)+1} + 2^{\mathsf{ufp}(z)-\mathsf{prec}(\times)}) \ .\end{aligned} \tag{4.5}$$

■

We assume that the error $\varepsilon_{z_\times}$ of the multiplication of two numbers x and y whose result is z is $\varepsilon_{z_\times} = y \ . \ \varepsilon_x + x \ . \ \varepsilon_y + \varepsilon_x \ . \ \varepsilon_y + \varepsilon_\times$ where $\varepsilon_\times$ is the truncation error for the multiplication and is equal to $\varepsilon_\times \leqslant 2^{\mathsf{ufp}(z)-\mathsf{prec}(\times)}$ (for the rounding mode towards $\uparrow_\sim$) and where $\mathsf{prec}(\times)$ represents the precision of the operator $\times$. So, the error $\varepsilon_{z_\times}$ can be bounded as shown in Equation (4.6). We have:

$$2^{ufp(x)} \leqslant x < 2^{ufp(x)+1} \quad \text{and} \quad 2^{ufp(y)} \leqslant y < 2^{ufp(y)+1}$$

and consequently,

$$\begin{aligned}\varepsilon_{z_\times} < 2^{\mathsf{ufp}(x)+1} \cdot 2^{\mathsf{ufp}(y)-\mathsf{nsb}(y)+1} + 2^{\mathsf{ufp}(y)+1} \cdot 2^{\mathsf{ufp}(x)-\mathsf{nsb}(x)+1} + 2^{\mathsf{ufp}(x)-\mathsf{nsb}(x)+1} \cdot \\ 2^{\mathsf{ufp}(y)-\mathsf{nsb}(y)+1} + 2^{\mathsf{ufp}(z)-\mathsf{prec}(\times)}\end{aligned}$$

thus,

$$\begin{aligned}\varepsilon_{z_\times} < 2^{\mathsf{ufp}(x)+\mathsf{ufp}(y)-\mathsf{nsb}(y)+2} + 2^{\mathsf{ufp}(x)+\mathsf{ufp}(y)-\mathsf{nsb}(x)+2} + 2^{\mathsf{ufp}(x)+\mathsf{ufp}(y)-\mathsf{nsb}(x)-\mathsf{nsb}(y)+2} \\ +2^{\mathsf{ufp}(z)-\mathsf{prec}(\times)} \ .\end{aligned}$$

Since $\mathsf{ufp}(x) + \mathsf{ufp}(y) - \mathsf{nsb}(x) - \mathsf{nsb}(y) + 2 < \mathsf{ufp}(x) + \mathsf{ufp}(y) - \mathsf{nsb}(y) + 2$ and $\mathsf{ufp}(x) + \mathsf{ufp}(y) - \mathsf{nsb}(x) - \mathsf{nsb}(y) + 2 < \mathsf{ufp}(x) + \mathsf{ufp}(y) - \mathsf{nsb}(x) + 2$, we may get rid of the penultimate term of the former equation and finally we obtain

$$\varepsilon_{z_\times} < 2^{\mathsf{ufp}(x)+\mathsf{ufp}(y)-\mathsf{nsb}(y)+2} + 2^{\mathsf{ufp}(x)+\mathsf{ufp}(y)-\mathsf{nsb}(x)+2} + 2^{\mathsf{ufp}(z)-\mathsf{prec}(\times)} \ . \tag{4.6}$$

Remark 4.2. If we assume that our analysis can make underflow in the multiplication between two numbers x and y whose result is z, which means that $z = 0$ while $x \neq 0$ and $y \neq 0$ then we can use the the truncation error $\varepsilon_\times$ defined in Equation (4.7) and thus Definition 4.4 remains valid.

$$\varepsilon_\times \leqslant \begin{cases} 2^{\mathsf{ufp}(z)-\mathsf{prec}(\times)} & \text{if } x \times y \neq 0, \\ 2^{\mathsf{ufp}(z)} & \text{otherwise.} \end{cases} \tag{4.7}$$

Example 4.3. Let $x = 4.0$, $y = 1.0$ and let z the result of their multiplication. By assuming that the variables are in double precision before analysis, we obtain $\overrightarrow{\otimes}(4.0|53|, 1.0|53|) = 4.0|53|$. Thus, $\mathsf{nsb}_F(z) = 53$.

Definition 4.5 (Forward Division)**.** The forward division $\overrightarrow{\oslash}$ between two numbers x and y whose result is z is given below as shown in Equation (4.8).

$$\overrightarrow{\oslash}(x,y) = z = \overrightarrow{\otimes}(x,y') \text{ with } y' = y^{-1},\ \mathsf{nsb}(y') = \mathsf{nsb}(y) \text{ and } \mathsf{nsb_e}(y') = \mathsf{nsb_e}(y). \tag{4.8}$$

■

For the division $\frac{x}{y}$, the approach consists in computing the reciprocal of the divisor y (with rounding to nearest) where no loss of precision is considered as shown in Equation (4.8). After, we proceed by multiplying the obtained result y' by x (same principle as in Equation (4.5)). The reason why no round-off is done in the inverse operation is that we consider, as in the IEEE754 Standard, that the division only performs one round-off error. This latter is taken into account in the multiplication and we wish to avoid a double rounding.

4.2.2 Backward Analysis

In the following we introduce the backward transfer functions $\overleftarrow{\oplus}$, $\overleftarrow{\ominus}$, $\overleftarrow{\otimes}$ and $\overleftarrow{\oslash}$ respectively for the addition, subtraction, multiplication and division. These functions take advantage of the results of the forward analysis and the accuracy requirement on the results. By combining these two data, it is then possible to lower the number of bits needed for one of the operands. In the sequel, we consider that the operand x is unknown where the result z and the operand y are known. The backward functions for the proposed elementary operations are given by the following definitions.

Definition 4.6 (Backward Addition)**.** The backward transfer function for addition $\overleftarrow{\oplus}$ between two numbers x and y whose result is z is given by Equation (4.9).

$$\begin{aligned}\overleftarrow{\oplus}(z,y) = (z-y) \text{ with } \mathsf{nsb}(x) = \mathsf{ufp}(z-y) - \mathsf{ufp}(2^{\mathsf{ufp}(z)-\mathsf{nsb}(z)+1} - 2^{\mathsf{ufp}(y)-\mathsf{nsb}(y)+1} \\ -2^{\mathsf{ufp}(x)-\mathsf{prec}(+)}) \ .\end{aligned} \tag{4.9}$$

■

To obtain Equation (4.9), we assumed that x is unknown while the result z is known. As mentioned earlier, the error of the result and the operand errors can be bounded by $\varepsilon_z < 2^{\mathsf{ufp}(z)-\mathsf{nsb}(z)+1}$ and that $\varepsilon_y < 2^{\mathsf{ufp}(y)-\mathsf{nsb}(y)+1}$. Finally, we compute the number of significant bits $\mathsf{nsb}(x)$ of the operand x with respect to the user accuracy requirement and the forward analysis result.

Now, we take again Example 4.2. We have two data available: the forward accuracy of the result z given as $\mathsf{nsb}_F(z) = 54$ bits and the user accuracy requirement on z given by $\mathsf{nsb} = 23$ bits. By combining these two knowledge, we obtain that $\overleftarrow{\oplus}(4.0|23|, 1.0|53|) = 3.0|25|$.

Definition 4.7 (Backward Subtraction)**.** The backward transfer function for subtraction $\overleftarrow{\ominus}$ between two numbers x and y whose result is z is given by Equation (4.10).

$$\begin{aligned}\overleftarrow{\ominus}(z,y) = (z+y) \text{ with } \mathsf{nsb}(x) = \mathsf{ufp}(z+y) - \mathsf{ufp}(2^{\mathsf{ufp}(z)-\mathsf{nsb}(z)+1} + 2^{\mathsf{ufp}(y)-\mathsf{nsb}(y)+1} \\ +2^{\mathsf{ufp}(x)-\mathsf{prec}(-)}) \ .\end{aligned} \tag{4.10}$$

■

We know that the round-off errors are bounded as $\varepsilon_z < 2^{\mathsf{ufp}(z)-\mathsf{nsb}(z)+1}$ and $\varepsilon_y < 2^{\mathsf{ufp}(y)-\mathsf{nsb}(y)+1}$ and the truncation error $\varepsilon_- \leqslant 2^{\mathsf{ufp}(x)-\mathsf{prec}(-)}$ where $\mathsf{prec}(-)$ denotes the precision of the operator $-$. Consequently, the total error on z is $\varepsilon_{z_-} = \varepsilon_x - \varepsilon_y - \varepsilon_-$.

Definition 4.8 (Backward Multiplication)**.** Let us assume the multiplication of two numbers x and y and the result is z. The backward transfer function for multiplication $\overleftarrow{\otimes}$ is given in Equation (4.11).

$$\overleftarrow{\otimes}(z,y) = (z \div y) \quad \text{with} \quad \mathsf{nsb}(x) = \mathsf{ufp}(z \div y) - \mathsf{ufp}\Big(\frac{2^{\mathsf{ufp}(y)+1} \cdot 2^{\mathsf{ufp}(z)-\mathsf{nsb}(z)+1} - 2^{\mathsf{ufp}(z)+1} \cdot 2^{\mathsf{ufp}(y)-\mathsf{nsb}(y)+1}}{2^{\mathsf{ufp}(y)+1} \cdot (2^{\mathsf{ufp}(y)+1} + 2^{\mathsf{ufp}(y)-\mathsf{nsb}(y)+1})} - 2^{\mathsf{ufp}(x)-\mathsf{prec}(\times)}\Big) \ . \tag{4.11}$$

■

In the case of multiplication, we know that $\overleftarrow{\otimes}(z,y) = (z \div y)$ with $\mathsf{nsb}(x) = \mathsf{ufp}(z \div y) - \mathsf{ufp}(\varepsilon_{z_\times})$ and where the truncation error $\varepsilon_\times \leqslant 2^{\mathsf{ufp}(x)-\mathsf{prec}(\times)}$ and thus the error $\varepsilon_{z_\times}$ is bounded as it is shown in Equation (4.11).

Let us go back to Example 4.3. After the forward analysis, we found that $\mathsf{nsb}_F(z) = 53$. If we have `require_nsb(z,23)`, we can obtain the following accuracy $\overleftarrow{\otimes}(4.0|23|, 1.0|53|) = 4.0|25|$.

Definition 4.9 (Backward Division)**.** The backward transfer function for division $\overleftarrow{\oslash}$ between two numbers x and y whose result is z is shown in Equation (4.12).

$$\overleftarrow{\oslash}(z,y) = \overrightarrow{\otimes}(z,y) \tag{4.12}$$

■

In case of division, we assume that the dividend x is unknown where the quotient z and the divisor y are known. Equation (4.12) shows that the backward division is equal to the forward multiplication between z and y as defined in Equation (4.5).

4.3 Elementary Functions

The elementary functions such as the natural logarithm, the exponential functions and the hyperbolic and trigonometric functions are not included in any arithmetic Standard when compared to the square root function which is included in the IEEE754 Standard. For this reason, each implementation of these functions has its own accuracy which we have to know in order to model the propagation of errors in our forward and backward analysis. To cope with this limitation, we consider that each elementary function introduces a loss of precision of φ bits, where $\varphi \in \mathbb{N}$ is a parameter of the analysis. For instance, we define in equations (4.13) and (4.14) the forward and backward transfer functions for the sine function.

Definition 4.10 (Forward Sine Function)**.** Let $\overrightarrow{\sin}(x)$ denotes the forward transfer function for the sine function. Let $\varphi \in \mathbb{N}$ denotes the number of bits to lose, given by the user. Then, we have:

$$\overrightarrow{\sin}(x) = z \ \text{ where } \ \mathsf{nsb}(z) = \mathsf{nsb}(x) - \varphi \ . \tag{4.13}$$

■

Definition 4.11 (Backward Sine Function)**.** Let $\overleftarrow{\sin}(x)$ denotes the backward transfer function for the sine function and let $\varphi \in \mathbb{N}$ the number of bits to lose. Then, we have:

$$\overleftarrow{\sin}(z) = x \text{ where } \mathsf{nsb}(x) = \mathsf{nsb}(z) + \varphi \ . \tag{4.14}$$

■

Example 4.4. Let $x = 3.0$, $\varphi = 9$ and let `require_nsb(sin(`x`),26)`. By applying equation (4.13) and (4.14) we obtain that

$$\overrightarrow{\sin}(x) = \sin(3.0|35|)|26| \ .$$

which means that $\mathsf{nsb}(3.0) \geqslant \mathsf{nsb}(\sin) + \varphi$.

For the remaining functions: cos, tan, arcsin, arccos, arctan, log and exp, we proceed with the same manner as for the sine function by assuming that φ bits can be lost through computations. Currently, we have a new idea to handle these elementary functions. In practice, we will reverse the problem by assuming that the parameter φ is another unknown of our system of constraints. In other words, the number of bits to lose will be computed in function to the user accuracy requirement on the outputs of the program.

Remark 4.3. The principle of analysis for the trigonometric function will remain the same when we translate the precision tuning problem into an integer linear programming problem in Chapter 5.

In the sequel, we present in equations (4.15) and (4.16) the forward and backward transfer functions for square root function.

Definition 4.12 (Forward Square Root)**.** Let x and z two numbers and let $\mathsf{prec}(\sqrt{})$ the precision of the operation. The forward transfer function for the square root of x is denoted by $\overrightarrow{\sqrt{x}}$ and is given by

$$\overrightarrow{\sqrt{x}} = z \text{ where } \mathsf{nsb}(z) = \mathsf{ufp}(z) - \mathsf{ufp}(2^{\mathsf{ufp}(x)-\mathsf{nsb}(x)+1} + 2^{\mathsf{ufp}(z)-\mathsf{prec}(\sqrt{})}) \ . \tag{4.15}$$

■

Definition 4.13 (Backward Square Root)**.** Let x and z two numbers and let prec the precision of the operation. The backward transfer function denoted by $\overleftarrow{\sqrt{x}}$ of a number x is given by

$$\overleftarrow{\sqrt{z}} = x \text{ where } \mathsf{nsb}(x) = \mathsf{ufp}(z^2) - \mathsf{ufp}(2^{\mathsf{ufp}(z^2)-\mathsf{nsb}(z)+1} - 2^{\mathsf{ufp}(x)-\mathsf{prec}(\sqrt{})}) \ . \tag{4.16}$$

■

Obviously, our static analysis does not work only on scalar values but on intervals instead. As described in [Mar17], we abstract sets of values by intervals. An element $i^\sharp \in \mathbb{I}_{\mathsf{nsb}}$

corresponds to $i^\sharp = [\underline{f}, \overline{f}]_{\mathsf{nsb}}$ and is defined by two numbers and a number of significant bits nsb.

$$\mathbb{I}_{\mathsf{nsb}} \ni [\underline{f}, \overline{f}]_{\mathsf{nsb}} = \{\underline{f} \leqslant f \leqslant \overline{f}\} \ . \tag{4.17}$$

In addition, we write $\mathbb{I} = \bigcup_{\mathsf{nsb} \in \mathbb{N}} \mathbb{I}_{\mathsf{nsb}}$. The operations $\overrightarrow{\oplus}^\sharp$, $\overleftarrow{\oplus}^\sharp$, $\overrightarrow{\ominus}^\sharp$, $\overleftarrow{\ominus}^\sharp$, $\overrightarrow{\otimes}^\sharp$, $\overleftarrow{\otimes}^\sharp$, $\overrightarrow{\oslash}^\sharp$ and $\overleftarrow{\oslash}^\sharp$ among values of $\mathbb{I}_{\mathsf{nsb}}$ are already defined in [Mar17] in function of $\overrightarrow{\oplus}$, $\overleftarrow{\oplus}$, $\overrightarrow{\ominus}$, $\overleftarrow{\ominus}$, $\overrightarrow{\otimes}$, $\overleftarrow{\otimes}$, $\overrightarrow{\oslash}$ and $\overleftarrow{\oslash}$ already defined in this chapter. We refer the reader to [Mar17] for a full description and details.

In the next section, we introduce the function which computes the carry bit that can occur throughout a program operations.

4.4 Carry Bit Function

A definition of the carry bit function denoted by ξ is given in this section. By this definition, we attempt to be more efficient in the way we propagate errors across the arithmetic operations than in previous work [Mul05, Mar17].

In practice, during an operation between two numbers x and y, a carry bit can be propagated through the operation. Proposition 4.1, already presented and proved in [Mul05], is considered as correct but pessimistic (too large over-approximation) due to the fact that adding an extra bit specially for cases we should not to, becomes very costly if we perform many computations.

Proposition 4.1 (Over-approximated Carry Bit)**.** Let x and y two numbers and let z be the result of their addition. We have in the worst case a carry bit that can occur through this operation.

$$\mathsf{ufp}(z) \leqslant \max(\mathsf{ufp}(x), \mathsf{ufp}(y)) + 1 \ . \tag{4.18}$$

■

In Martel's work [Mar17], a new carry bit function ξ was presented in order to refine Equation (4.18). His method compares the unit in the first place of the operands of the operation and adds an extra bit only if they are equal, in other words, when the two numbers are aligned. While this optimization is correct but it misses precision. The methodical difference between the former definition and our new definition of function ξ depicted in Figure 4.3 is that we take into consideration the unit in the first places as well as the unit in the last places of the two errors of the operands and we add an extra bit only if we are certain that the $\mathsf{ulp_e}$ (defined in Equation (2.8)) of one of the operands is lesser than the $\mathsf{ufp_e}$ (defined in Equation (2.7)) of the other operand or conversely.

In this dissertation and specially in Chapter 5 we will prove that our new definition improves significantly the accuracy of the static analysis by being less pessimistic.

Definition 4.14 (Optimized Carry Bit Function)**.** Let x and y two operands of some operation which result is z. The optimized ξ function is given as shown in Equation (4.19): if the $\mathsf{ulp_e}$ of one of the two operands error $\varepsilon(x)$ or $\varepsilon(y)$ is greater than the $\mathsf{ufp_e}$ of the other one (or

conversely) then the numbers x and y are not aligned and consequently $\xi = 0$ (otherwise $\xi = 1$). The optimized ξ function is given by

$$\xi(z)(x,y) = \begin{cases} 0 & \mathsf{ulp_e}(x) \geqslant \mathsf{ufp_e}(y), \\ 0 & \mathsf{ulp_e}(y) \geqslant \mathsf{ufp_e}(x), \\ 1 & \text{otherwise.} \end{cases} \tag{4.19}$$

■

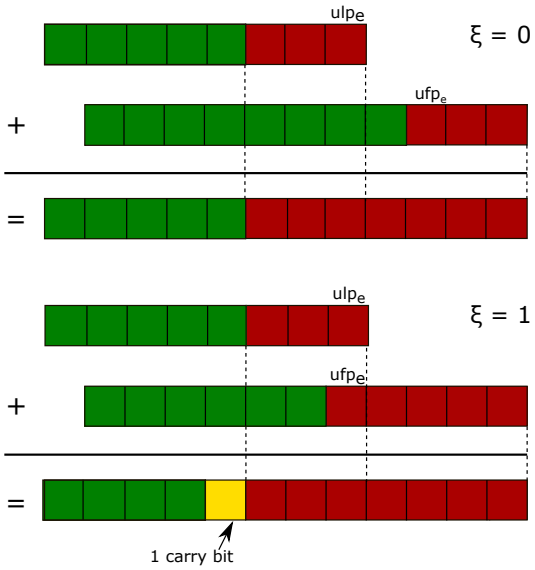


Figure 4.3: Schematic representation of the optimized carry bit function ξ.

Note that the SMT-based method and the ILP-based method use the same optimized carry bit function of Definition 4.14. Hence, an equivalent yet less intuitive definition of ξ is used in Chapter 5 Section 5.2.2 which corresponds to Equation (4.20) in Lemma 4.1.

Lemma 4.1. Let x and y be the operands of some operation whose result is z. We assume that Equation (4.20) is equivalent to Equation (4.19).

$$\xi(z)(x,y) = \begin{cases} 0 & \mathsf{ufp_e}(x) - \mathsf{nsb_e}(x) \geqslant \mathsf{ufp}(y) - \mathsf{nsb}(y) \ \textit{or} \ \text{conversely}, \\ 1 & \text{otherwise.} \end{cases} \tag{4.20}$$

■

We must precise that the basic contribution of the ILP-based method relies on this definition of ξ function. We will show in the next chapter a detailed comparison between the over-approximated and the optimized carry bit functions and how this function has an effect on the linearity of the constraints and on the new data types returned by the analysis.

4.5 Summary

The contribution of this chapter revolves around the forward and backward transfer functions. The information collected by our analysis allows us to compute the minimal

number of bits needed for the variables and intermediary results of programs in order to fulfill the requirements specified by the user. Indeed, we have extended this methodology to other language structures in particular loops[1], conditions, arrays, elementary functions and other arithmetic expressions. Besides, we have considered that a range determination is performed by dynamic analysis on the variables of our programs and that no overflow arises during our analysis.

In the next chapter we will express our analysis as a set of constraints. The first set of constraints is made of first order predicates and affine integer relations only, even if the analyzed programs contain non-linear computations. These constraints are fully based on the presented transfer functions. Finally, these constraints can be easily checked by a SMT solver.

The second set of constraints is different from the one generated by the SMT-based method. The precision tuning will be expressed as an ILP formulation. The transfer functions remain valid but in only the backward analysis. The new ILP-based method will be easier to solve in an efficient way.

[1]The commands will be discussed in Chapter 5

Chapter 5

Generation of Constraints for Bit-Level Tuning

***

5.1 SMT-Based Method Constraint Generation 76
5.1.1 Forward Analysis 77
5.1.2 Backward Analysis 78
5.2 ILP-Based Method Constraint Generation 80
5.2.1 Pure ILP Formulation 80
5.2.2 Policy Iteration for Optimized Carry Bit Propagation 84
5.3 Correctness 89
5.3.1 Soundness of the Constraint System 90
5.3.2 ILP Nature of the Problem 91
5.4 Summary 92

We introduce in the present chapter the constraints generated by our tool to determine the precision of the variables and intermediary values of the input programs. In fact, the transfer functions defined in the previous chapter are not translated directly into constraints because the resulting system would be too difficult to solve: it contains floating-point numbers and non-linear constraints. To cope with this limitation, we propose in this chapter two constraint systems. The first set of constraints corresponds to the SMT-based method. It is made of first order predicates and affine integer relations only, even if the analyzed programs contain non-linear computations (e.g. min and max functions). As a consequence, these constraints are easy to solve by a SMT solver repeatedly to find the existence of a solution with a certain weight expressing the number of significant bits (nsb) of variables. Chapters 7 and 8 evaluate this method implemented in our tool POP on several benchmarks.

As we have mentioned, our goal is to make our analysis simpler in order to make solvers work efficiently. For this reason, the second set of constraints reduces the precision tuning problem into an Integer Linear Programming (ILP) problem. By reasoning on the number of significant bits of the variables of the programs and knowing the weight of their most significant bit thanks to a range analysis performed before the tuning phase, we are able to reduce the precision tuning problem to an ILP which can be optimally solved in one shot by a classical linear programming (LP) solver (no iteration). An important point is that the optimal solution to the continuous linear programming relaxation of our ILP is a vector of integers, as demonstrated in Section 5.3.2. By consequence, we may use a linear solver among real numbers whose complexity is polynomial [Sch98] (contrarily to the linear solvers among integers whose complexity is NP-hard [Pap81]). This makes our ILP-based method solvable in polynomial-time, contrarily to the existing exponential methods and the SMT-based method. Next, we go one step further 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. However, the problem does not reduce any longer to an ILP problem (min and max operators are needed). Then we use the policy iteration (PI) technique [CGG+05] to find efficiently the solution.

The remainder of this chapter is organized as follows. Section 5.1 presents the constraints for the SMT-based method. Section 5.2 deals with the ILP-based method and the policy iteration method for an optimized carry bit propagation. The proofs of correctness are reported in Section 5.3, while Section 5.4 concludes.

This work is mostly based on the articles published respectively at the 7^{th} International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS) in 2019 [BMA19] (Section 5.1) and the 28^{th} Static Analysis Symposium (SAS) in 2021 [ABM21] (Section 5.2).

5.1 SMT-Based Method Constraint Generation

In this section, we introduce the constraints generated by our tool for the SMT-based method. In our analysis, we assume that a range analysis is performed before the accuracy analysis and that a bounding interval is given for each variable and each value at any control point of the input programs. For the sake of clarity, we insist on the fact that our tool achieves precision tuning only. The inputs are the program and the ranges of the variables of the program. Those ranges are understood as intervals. This range inference is completely external to our tool and has to be performed by an invariant generator or an analyzer. Nevertheless, our tool `POP` offers a dynamic analysis which produces an under-approximation under the form of intervals. Static analyzers with sophisticated abstract domains could be used instead such as [CCF+05]. In particular the efficiency of our techniques for loops depends on the precision of the range analysis for loops. Using a static analyzer to determine ranges of the variables shall be left to future work.

Let us also mention that, in order to avoid cumbersome notations, the constraints

introduced hereafter assume that the programs handle scalar values instead of the intervals given by the range analysis. A generalization to intervals is introduced in [Mar17] for a comparable set of constraints. In the sequel, we start explaining the constraints for the forward operations. We are interested in proving the constraints for the addition and the multiplication. The principle remains the same for all the other arithmetic expressions.

5.1.1 Forward Analysis

Forward addition If we have the following addition $z = x + y$ then the total error is equal to $\varepsilon_{z_+} = \varepsilon_x + \varepsilon_y + \varepsilon_+$. Now, in order to apply Definition 4.14, we will decompose the total error ε_{z_+} into two errors:

- **Round-off error** $\varepsilon_{xy} = \varepsilon_x + \varepsilon_y$ given by

$$\begin{aligned}\mathsf{ufp}(\varepsilon_{xy}) < \max(\mathsf{ufp}(x) - \mathsf{nsb}(x) + 1, \mathsf{ufp}(y) - \mathsf{nsb}(y) + 1) + \xi(\mathsf{ufp}(x) - \mathsf{nsb_e}(x) + 1,\\ \mathsf{ufp}(y) - \mathsf{nsb}(y))\ ,\end{aligned} \tag{5.1}$$

- **Truncation error** already defined in Equation (4.2).

By adding these two errors, we obtain that

$$\begin{aligned}\mathsf{ufp}(\varepsilon_{z_+}) < \max\left(\mathsf{ufp}(x) - \mathsf{nsb}(x) + 1, \mathsf{ufp}(y) - \mathsf{nsb}(y) + 1, \mathsf{ufp}(z) - \sigma_+\right)\\ +\xi(\mathsf{ufp}_e(x) - \mathsf{nsb_e}(x) + 1, \mathsf{ufp}(y) - \mathsf{nsb}(y))\ .\end{aligned} \tag{5.2}$$

Note that, since we assume that a range analysis has been performed before the accuracy analysis, $\mathsf{ufp}(x+y)$, $\mathsf{ufp}(x)$ and $\mathsf{ufp}(y)$ are known at constraint generation time. Consequently, we conclude from Equations (5.2) that the number of significant bits $\mathsf{nsb}(z)$ can be bounded as shown in Equation (5.3).

$$\begin{aligned}\mathsf{nsb}(z) = \mathsf{ufp}(x+y) - \max\left(\mathsf{ufp}(x) - \mathsf{nsb}(x) + 1, \mathsf{ufp}(y) - \mathsf{nsb}(y) + 1, \mathsf{ufp}(z) - \sigma_+\right)\\ -\xi(\mathsf{ufp}_e(x) - \mathsf{nsb_e}(x) + 1, \mathsf{ufp}(y) - \mathsf{nsb}(y))\ .\end{aligned} \tag{5.3}$$

Now, what remains to be done is to determine $\mathsf{nsb_e}(z)$ of the addition. This is why, we need to compute $\mathsf{ulp}(\epsilon_{z_+})$ as shown in Definition (5.1).

Definition 5.1 (Unit in The Last Place of an Addition Operation)**.** Let x, y and z three numbers such that $z = x + y$. Let ε_{z_+} the total error of the addition. We define $\mathsf{ulp}(\varepsilon_{z_+})$ as the smallest ulp between the two operand errors $\mathsf{ulp}(\varepsilon_x)$ and $\mathsf{ulp}(\varepsilon_y)$. This definition is given below in Equation (5.4).

$$\mathsf{ulp}(\varepsilon_{z_+}) = \min\left(\mathsf{ulp}(\varepsilon_x), \mathsf{ulp}(\varepsilon_y)\right)\ . \tag{5.4}$$

■

Forward multiplication We proceed by describing the constraint generated for the multiplication. Let us take again Equation (4.6). We proceed as we did in the case of addition by decomposing the total error $\varepsilon_\times$ into the round-off error $\varepsilon_{xy} = \varepsilon_x + \varepsilon_y$ and the truncation error $\varepsilon_\times$.

We have the round-off error ε_{xy} can be bounded by

$$\begin{aligned}\mathsf{ufp}(\varepsilon_{xy}) < \max(\mathsf{ufp}(x) + \mathsf{ufp}(y) - \mathsf{nsb}(x) + 2, \mathsf{ufp}(x) + \mathsf{ufp}(y) - \mathsf{nsb}(y) + 2)\\ +\xi(\mathsf{ufp}(x) - \mathsf{nsb_e}(x) + 1, \mathsf{ufp}(y) - \mathsf{nsb}(y))\ ,\end{aligned}$$

which can also be written as

$$\begin{aligned}\mathsf{ufp}(\varepsilon_{xy}) \leqslant \max(\mathsf{ufp}(x) + \mathsf{ufp}(y) - \mathsf{nsb}(x) + 1, \mathsf{ufp}(x) + \mathsf{ufp}(y) - \mathsf{nsb}(y) + 1) +\\ \xi(\mathsf{ufp}(x) - \mathsf{nsb_e}(x) + 1, \mathsf{ufp}(y) - \mathsf{nsb}(y))\ .\end{aligned} \tag{5.5}$$

Then the total error $\varepsilon_{z_\times}$ may be bound by

$$\begin{aligned}\mathsf{ufp}(\varepsilon_{z_\times}) \leqslant \max\left(\mathsf{ufp}(x) - \mathsf{nsb}(x) + 1, \mathsf{ufp}(y) - \mathsf{nsb}(y) + 1, \mathsf{ufp}(z) - \sigma_+\right)\\ +\xi(\mathsf{ufp}(x) - \mathsf{nsb_e}(x) + 1, \mathsf{ufp}(y) - \mathsf{nsb}(y))\ .\end{aligned} \tag{5.6}$$

By assuming that $\mathsf{ufp}(x \times y)$, $\mathsf{ufp}(x)$, $\mathsf{ufp}(y)$ and $\mathsf{ufp}(z)$ are known at constraint generation time thanks to a range analysis, we present the number of significant bits on z in Equation (5.7).

$$\begin{aligned}\mathsf{nsb}(z) = \mathsf{ufp}(x \times y) - \max\left(\mathsf{ufp}(x) - \mathsf{nsb}(x) + 1, \mathsf{ufp}(y) - \mathsf{nsb}(y) + 1, \mathsf{ufp}(z) - \sigma_+\right) -\\ \xi(\mathsf{ufp}_e(x) - \mathsf{nsb_e}(x) + 1, \mathsf{ufp}(y) - \mathsf{nsb}(y))\ .\end{aligned} \tag{5.7}$$

Note that, by reasoning on the exponents of the values, the constraints resulting from a multiplication operation become linear. Next, like we have done in Equation (5.4) we define the unit in the last place of $\varepsilon_{z_\times}$ in Definition (5.2).

Definition 5.2 (Unit in the Last Place of a Multiplication Operation)**.** Let x, y and z three numbers such that $z = x \times y$. Let $\varepsilon_{z_\times}$ the total error of the multiplication. Thus, we define $\mathsf{ulp}(\varepsilon_{z_\times})$ as

$$\mathsf{ulp}(\varepsilon_{z_\times}) = \mathsf{ulp}(\epsilon_x) + \mathsf{ulp}(\epsilon_y)\ . \tag{5.8}$$

■

5.1.2 Backward Analysis

By reasoning in the same way, we linearize the computations for the backward addition and multiplication operations.

Backward addition We consider now the backward transfer functions, depending on Equation (4.9) for the addition case. We know that $\mathsf{nsb}(x) = \mathsf{ufp}(z - y) - \mathsf{ufp}(\varepsilon_z - \varepsilon_y - \varepsilon_+)$. So, we can over-approximate ε_z thanks to the relations $\varepsilon_z < 2^{\mathsf{ufp}(z)-\mathsf{nsb}(z)+1}$, $\varepsilon_y \geqslant 0$ and $\varepsilon_+ \geqslant 0$ and consequently

$$\mathsf{nsb}(x) = \mathsf{ufp}(z - y) - \mathsf{ufp}(z) + \mathsf{nsb}(z)\ . \tag{5.9}$$

Backward multiplication From Equation (4.11), we know that $2^{\mathsf{ufp}(z)} \leqslant z < 2^{\mathsf{ufp}(z)+1}$, $2^{\mathsf{ufp}(y)} \leqslant y < 2^{\mathsf{ufp}(y)+1}$ and $\varepsilon_{z_\times} < 2^{\mathsf{ufp}(z)-\mathsf{nsb}(z)+1}$, $\varepsilon_y < 2^{\mathsf{ufp}(y)-\mathsf{nsb}(y)+1}$ which implies that $y.\varepsilon_{z_\times} - z.\varepsilon_y < 2^{\mathsf{ufp}(z)+\mathsf{ufp}(y)-\mathsf{nsb}(z)+2} - 2^{\mathsf{ufp}(y)+\mathsf{ufp}(z)-\mathsf{nsb}(y)+2}$ and that

$$\frac{1}{y \cdot (y+\varepsilon_y)} < 2^{-2\mathsf{ufp}(y)} \ .$$

Consequently,

$$\begin{aligned}\varepsilon_{z_\times} &\leqslant 2^{-2\mathsf{ufp}(y)} \cdot (2^{\mathsf{ufp}(z)+\mathsf{ufp}(y)-\mathsf{nsb}(z)+2} - 2^{\mathsf{ufp}(y)+\mathsf{ufp}(z)-\mathsf{nsb}(y)+2}) - 2^{\mathsf{ufp}(x)-\sigma_\times} \\ &\leqslant 2^{\mathsf{ufp}(z)-\mathsf{ufp}(y)-\mathsf{nsb}(z)+1} - 2^{\mathsf{ufp}(z)-\mathsf{ufp}(y)-\mathsf{nsb}(y)+1} - 2^{\mathsf{ufp}(x)-\sigma_\times} \ ,\end{aligned}$$

and finally,

$$\begin{aligned}\mathsf{nsb}(x) = \mathsf{ufp}(z \div y) - \max(&\mathsf{ufp}(c) - \mathsf{ufp}(y) - \mathsf{nsb}(z) + 1, \\ &\mathsf{ufp}(z) - \mathsf{ufp}(y) - \mathsf{nsb}(y) + 1, \mathsf{ufp}(x) - \sigma_\times) \ .\end{aligned} \tag{5.10}$$

Note that POP does not generate constraints only for arithmetic expressions but also for commands and arrays. The rules of commands are classical [NNH10], we use control points to distinguish many assignments of the same variable and also to implement joins in conditions and loops. A complete presentation of the commands will be provided in Section 5.2.

Example 5.1. In this example we show how to generate constraints for Example 4.1 depicted in Listing 4.1 of Chapter 4. The constraint system with control points ℓ_0 to ℓ_{28}, is made of constraints for the whole program including the while loop, assignments, additions and the counter. These numbers of variables and constraints are linear in the size of the program. Although the number of constraints generated is linear, it is too large to be fully displayed hereafter. Equation (5.11) shows some constraints without taking into account the while loop and by taking into consideration the operations of assignments a := 1.0 , $i = 1.0$ and x = 0.0 and only one more complex operation a := a + 1.0.

$$C = \left\{ \begin{array}{l} \mathsf{nsb}_F(\ell_0) = 53,\ \mathsf{nsb}_F(\ell_2) = 53,\ \mathsf{nsb}_F(\ell_4) = 53,\ \mathsf{nsb}_F(\ell_{12}) = 53, \\ \mathsf{nsb}_B(a^{\ell_1}) = \mathsf{nsb}_B(\ell_0),\ \mathsf{nsb}_F(i^{\ell_3}) = \mathsf{nsb}_F(\ell_2),\ \mathsf{nsb}_F(x^{\ell_5}) = \mathsf{nsb}_b(\ell_4), \\ \mathsf{nsb}(\overline{\ell_{13}}) = \mathsf{nsb}(\underline{\ell_{13}}) = 1 - \max(0 - \mathsf{nsb}_F(\ell_{11}), 0 - \mathsf{nsb}_F(\ell_{12})), \\ (0 - \mathsf{nsb}_F(\ell_{11})) = 0 - \mathsf{nsb}_F(\ell_{12}) \Rightarrow \xi(\ell_{13}) = \xi(\ell_{13}) = 1,\ \mathsf{nsb}_F(a^{\ell_1}) = \mathsf{nsb}_F(\ell_0), \\ \mathsf{nsb}_F(\ell_{13}) = \min(\mathsf{nsb}(\ell_{13}) - \xi(\ell_{13}), \mathsf{nsb}(\overline{\ell_{13}}) - \overline{\xi}(\ell_{13})) = 53,\ \mathsf{nsb}_B(\ell_{13}) = 20, \\ \mathsf{nsb}_B(\ell_{11}) = 0 - (1 - \mathsf{nsb}_B(\ell_{13})),\ \mathsf{nsb}_B(\ell_{12}) = 0 - (1 - \mathsf{nsb}_B(\ell_{13})), \\ \mathsf{nsb}_F(a^{\ell_{14}}) = \mathsf{nsb}_F(\ell_{13}),\ \mathsf{nsb}_F(x^{\ell_{20}}) = \mathsf{nsb}_F(\ell_{19}),\ \mathsf{nsb}_B(a^{\ell_{14}}) = \mathsf{nsb}_B(\ell_{13}), \\ \mathsf{nsb}_B(x^{\ell_{20}}) = \mathsf{nsb}_B(\ell_{19}), \end{array} \right\}. \tag{5.11}$$

Basically, the constraints of Equation (5.11) require that the forward accuracy at points ℓ_0, ℓ_2, ℓ_4 and ℓ_{12} is 53 bits (e.g. FP64 double precision). Next, the forward accuracy of a^{ℓ_1} is equal to 53 ($\mathsf{nsb}_F(\ell_0) = 53$) which is the precision of the value affected and it works the same for its backward accuracy. The constraints of the second line of our system works similarly. As our approach is also generalisable for intervals, we have $\mathsf{nsb}(\underline{\ell})$ and $\mathsf{nsb}(\overline{\ell})$ respectively for the lower and upper bound of the number of significant bits nsb of the result. To be brief,

we explain the constraint of $\mathsf{nsb}(\overline{\ell})$ and it will be equivalent for the other bound $\mathsf{nsb}(\underline{\ell})$. We have the precision $\mathsf{nsb}(\overline{\ell_{13}})$ which is equal to the number of bits between the $\mathsf{ufp}(\ell_{13})$ and the maximum precision of one of the two operands which are also computed as the difference of their ufp and their precision respectively. While we have $\mathsf{nsb}_F(\ell_{13}) = 53$, then by applying Equation (5.2) we obtain that

$$\varepsilon(\ell_{13}) \leqslant \max\left(\max\left((\mathsf{ufp}(1.0) - \mathsf{nsb}_F(\ell_{11}), \mathsf{ufp}(1.0) - \mathsf{nsb}_F(\ell_{12})\right) + \xi, \mathsf{ufp}(\ell_{13}) - \sigma_+\right)\right) + \xi \ .$$

After taking a closer look at the SMT-based method, we have remarked that these constraints can be formulated in a simpler way making it possible to use an optimizing solver which returns the solution in polynomial-time, instead of the non-optimizing SMT solver coupled to a binary search used to solve the previous constraints. For all these reasons, we present the ILP-based method in the next section.

5.2 ILP-Based Method Constraint Generation

In this section, we detail the ILP-based method. Also, we show the nature of constraints obtained when using a pure ILP with an over-approximated carry bit and the more complex PI formulation which optimizes the carry bits that propagate throughout computations.

5.2.1 Pure ILP Formulation

In this section, we show that we are able to reduce the problem of determining the lowest precision on variables and intermediary values in programs to an ILP by reasoning on their unit in the first place (ufp) and the number of significant bits (nsb).

In contrast to the SMT-based method, we assign to each control point ℓ a unique integer variable $\mathsf{nsb}(\ell)$ (instead of three variables as shown in Equation (4.1) corresponding to the nsb of the arithmetic expression. Note that $\mathsf{nsb}(\ell)$ is determined by solving the ILP problem generated by the rules of Figure 5.1. We remind the reader that the constraints introduced hereafter handle only scalar values.

Let us now focus on the rules of Figure 5.1. These rules are designed for the grammar already presented in Figure 4.1. We have $\varrho : \mathrm{Id} \to \mathrm{Id} \times \mathrm{Lab}$ an environment which relates each identifier x to its last assignment x^ℓ: Assuming that $x :=^\ell e^{\ell_1}$ is the last assignment of x, the environment ϱ maps x to x^ℓ. Then, $\mathcal{E}[\![e]\!]\ \varrho$ generates the set of constraints for an expression $e \in Expr$ in the environment ϱ.

In the sequel, we formally define these constraints for each element of our language. No constraint is generated for a constant $c\#\mathsf{p}$ as mentioned in Rule (CONST) of Figure 5.1. For Rule (ID) of a variable x^ℓ, we require that the nsb at control point ℓ is less than its nsb in the last assignment of x given in $\varrho(x)$. For a binary operator $\odot \in \{+, -, \times, \div\}$, we first generate the set of constraints $\mathcal{E}[\![e_1^{\ell_1}]\!]\varrho$ and $\mathcal{E}[\![e_2^{\ell_2}]\!]\varrho$ for the operands at control points ℓ_1 and ℓ_2. Considering Rule (ADD), the result of the addition of two numbers is stored in control point ℓ. Recall that a range determination is performed before the accuracy analysis, $\mathsf{ufp}(\ell)$, $\mathsf{ufp}(\ell_1)$ and $\mathsf{ufp}(\ell_2)$ are known at constraint generation time.

$$\mathcal{E}[c\#\mathtt{p}^{\ell}]\varrho = \varnothing \quad (\textsc{Const}) \qquad \mathcal{E}[x^{\ell}]\varrho = \{\mathsf{nsb}(\varrho(x)) \geqslant \mathsf{nsb}(\ell)\} \quad (\textsc{Id})$$

$$\begin{gathered}\mathcal{E}[e_1^{\ell_1} +^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}[e_1^{\ell_1}]\varrho \cup \mathcal{E}[e_2^{\ell_2}]\varrho \\ \cup \\ \{\mathsf{nsb}(\ell_1) \geqslant \mathsf{nsb}(\ell) + \mathsf{ufp}(\ell_1) - \mathsf{ufp}(\ell) + \xi(\ell)(\ell_1,\ell_2), \\ \mathsf{nsb}(\ell_2) \geqslant \mathsf{nsb}(\ell) + \mathsf{ufp}(\ell_2) - \mathsf{ufp}(\ell) + \xi(\ell)(\ell_1,\ell_2)\}\end{gathered} \quad (\textsc{Add})$$

$$\begin{gathered}\mathcal{E}[e_1^{\ell_1} -^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}[e_1^{\ell_1}]\varrho \cup \mathcal{E}[e_2^{\ell_2}]\varrho \\ \cup \\ \{\mathsf{nsb}(\ell_1) \geqslant \mathsf{nsb}(\ell) + \mathsf{ufp}(\ell_1) - \mathsf{ufp}(\ell) + \xi(\ell)(\ell_1,\ell_2), \\ \mathsf{nsb}(\ell_2) \geqslant \mathsf{nsb}(\ell) + \mathsf{ufp}(\ell_2) - \mathsf{ufp}(\ell) + \xi(\ell)(\ell_1,\ell_2)\}\end{gathered} \quad (\textsc{Sub})$$

$$\begin{gathered}\mathcal{E}[e_1^{\ell_1} \times^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}[e_1^{\ell_1}]\varrho \cup \mathcal{E}[e_2^{\ell_2}]\varrho \\ \cup \\ \{\mathsf{nsb}(\ell_1) \geqslant \mathsf{nsb}(\ell) + \xi(\ell)(\ell_1,\ell_2) - 1, \quad \mathsf{nsb}(\ell_2) \geqslant \mathsf{nsb}(\ell) + \xi(\ell)(\ell_1,\ell_2) - 1\}\end{gathered} \quad (\textsc{Mult})$$

$$\begin{gathered}\mathcal{E}[e_1^{\ell_1} \div^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}[e_1^{\ell_1}]\varrho \cup \mathcal{E}[e_2^{\ell_2}]\varrho \\ \cup \\ \{\mathsf{nsb}(\ell_1) \geqslant \mathsf{nsb}(\ell) + \xi(\ell)(\ell_1,\ell_2) - 1, \quad \mathsf{nsb}(\ell_2) \geqslant \mathsf{nsb}(\ell) + \xi(\ell)(\ell_1,\ell_2) - 1\}\end{gathered} \quad (\textsc{Div})$$

$$\mathcal{E}\left[\sqrt{e^{\ell_1}}^{\ell}\right]\varrho = \mathcal{E}[e_1^{\ell_1}]\varrho \cup \{\mathsf{nsb}(\ell_1) \geqslant \mathsf{nsb}(\ell)\} \quad (\textsc{Sqrt})$$

$$\mathcal{E}\left[\phi\left(e^{\ell_1}\right)^{\ell}\right]\varrho = \mathcal{E}[e_1^{\ell_1}]\varrho \cup \{\mathsf{nsb}(\ell_1) \geqslant \mathsf{nsb}(\ell) + \varphi\} \text{ with } \phi \in \{\sin, \cos, \tan, \log, \ldots\} \quad (\textsc{Math})$$

$$\mathcal{C}\left[x :=^{\ell} e^{\ell_1}\right]\varrho = \left(C, \varrho\left[x \mapsto \ell\right]\right) \text{ where } C = \mathcal{E}[e_1^{\ell_1}]\varrho \cup \{\mathsf{nsb}(\ell_1) \geqslant \mathsf{nsb}(\ell)\} \quad (\textsc{Assign})$$

$$\begin{gathered}\mathcal{C}\left[c_1^{\ell_1}; c_2^{\ell_2}\right]\varrho = (C_1 \cup C_2, \varrho_2) \\ \text{where } (C_1, \varrho_1) = \mathcal{C}\left[c_1^{\ell_1}\right]\varrho \text{ and } (C_2, \varrho_2) = \mathcal{C}\left[c_2^{\ell_2}\right]\varrho_1\end{gathered} \quad (\textsc{Seq})$$

$$\text{where} \left| \begin{array}{l} \mathcal{C}[\mathtt{if}^{\ell}\ e^{\ell_0}\ \mathtt{then}\ c^{\ell_1}\ \mathtt{else}\ c^{\ell_2}]\ \varrho = (C_1 \cup C_2 \cup C_3, \varrho') \\ \forall x \in \mathrm{Id},\ \varrho'(x) = \ell,\ (C_1, \varrho_1) = \mathcal{C}[c_1^{\ell_1}]\ \varrho,\ (C_2, \varrho_2) = \mathcal{C}[c_2^{\ell_2}]\ \varrho, \\ C_3 = \bigcup\limits_{x \in \mathrm{Id}} \{\mathsf{nsb}(\varrho_1(x)) \geqslant \mathsf{nsb}(\ell),\ \mathsf{nsb}(\varrho_2(x)) \geqslant \mathsf{nsb}(\ell)\} \end{array} \right. \quad (\textsc{Cond})$$

$$\text{where} \left| \begin{array}{l} \mathcal{C}[\mathtt{while}^{\ell}\ e^{\ell_0}\ \mathtt{do}\ c^{\ell_1}]\ \varrho = (C_1 \cup C_2, \varrho') \\ \forall x \in \mathrm{Id},\ \varrho'(x) = \ell,\ (C_1, \varrho_1) = \mathcal{C}[c_1^{\ell_1}]\ \varrho' \\ C_2 = \bigcup\limits_{x \in \mathrm{Id}} \{\mathsf{nsb}(\varrho(x)) \geqslant \mathsf{nsb}(\ell),\ \mathsf{nsb}(\varrho_1(x)) \geqslant \mathsf{nsb}(\ell)\} \end{array} \right. \quad (\textsc{While})$$

$$\mathcal{C}[\mathtt{require_nsb}(x, \mathtt{p})^{\ell}]\varrho = \{\mathsf{nsb}(\varrho(x)) \geqslant \mathtt{p}\} \quad (\textsc{Req})$$

$$\xi(\ell)(\ell_1, \ell_2) = 1$$

Figure 5.1: ILP constraints with pessimistic carry bit propagation $\xi = 1$.

In the present ILP of Figure 5.1, we over-approximate the function ξ by $\xi(\ell)(\ell_1,\ell_2)=1$ for all ℓ, ℓ_1 and ℓ_2. To wrap up, for the addition (Rule (ADD)), we have the $\mathsf{nsb}(\ell) = \mathsf{ufp}(\ell) - \mathsf{ufp_e}(\ell)$. More precisely, let us consider the addition $c_1^{\ell_1} +^{\ell} c_2^{\ell_2}$ and let us assume that $\mathsf{prec}(\ell)$ denotes the precision of this operation. The error $\varepsilon(\ell)$ is bound by $\varepsilon(c_1^{\ell_1} +^{\ell} c_2^{\ell_2}) \leqslant \varepsilon(c_1^{\ell_1}) + \varepsilon(c_2^{\ell_2}) + 2^{\mathsf{ufp}(c_1+c_2)-\mathsf{prec}(\ell)}$ and

$$\mathsf{ufp_e}(\ell) = \max\big(\mathsf{ufp}(\ell_1)-\mathsf{nsb}(\ell_1), \mathsf{ufp}(\ell_2)-\mathsf{nsb}(\ell_2), \mathsf{ufp}(l)-\mathsf{prec}(\ell)\big) + \xi(\ell)(\ell_1,\ell_2)\ . \tag{5.12}$$

Since $\mathsf{nsb}(\ell) \leqslant \mathsf{prec}(\ell)$, we may get rid of the last term in Equation (5.12) and the two constraints generated for Rule (ADD) are derived from Equation (5.13).

$$\mathsf{nsb}(\ell) \leqslant \mathsf{ufp}(\ell) - \max\big(\mathsf{ufp}(\ell_1)-\mathsf{nsb}(\ell_1), \mathsf{ufp}(\ell_2)-\mathsf{nsb}(\ell_2)\big) - \xi(\ell)(\ell_1,\ell_2) \tag{5.13}$$

Rule (SUB) for the subtraction is obtained similarly to the addition case. For Rule (MULT) of multiplication (and in the same manner Rule(DIV)), the reasoning mimics the one of the addition. Let c_1 and c_2 be two numbers and c the result of their product, $c = c_1^{\ell_1} \times^{\ell} c_2^{\ell_2}$. We denote by $\varepsilon(c_1)$, $\varepsilon(c_2)$ and $\varepsilon(c)$ the errors on c_1, c_2 and c, respectively. The error $\varepsilon(c)$ of this multiplication is $\varepsilon(c) = c_1\cdot\varepsilon(c_2) + c_2\cdot\varepsilon(c_1) + \varepsilon(c_1)\cdot\varepsilon(c_2)$. These numbers are bounded by

$$\begin{array}{rcl}
2^{\mathsf{ufp}(c_1)} \leqslant c_1 \leqslant 2^{\mathsf{ufp}(c_1)+1} \quad \text{and} \quad 2^{\mathsf{ufp}(c_1)-\mathsf{nsb}(c_1)} & \leqslant & \varepsilon(c_1) \leqslant 2^{\mathsf{ufp}(c_1)-\mathsf{nsb}(c_1)+1} \\
2^{\mathsf{ufp}(c_2)} \leqslant c_2 \leqslant 2^{\mathsf{ufp}(c_2)+1} \quad \text{and} \quad 2^{\mathsf{ufp}(c_2)-\mathsf{nsb}(c_2)} & \leqslant & \varepsilon(c_2) \leqslant 2^{\mathsf{ufp}(c_2)-\mathsf{nsb}(c_2)+1} \\
2^{\mathsf{ufp}(c_1)+\mathsf{ufp}(c_2)-\mathsf{nsb}(c_2)} + 2^{\mathsf{ufp}(c_2)+\mathsf{ufp}(c_1)-\mathsf{nsb}(c_1)} & \leqslant & \varepsilon(c) \leqslant 2^{\mathsf{ufp}(c)-\mathsf{nsb}(c)+1} \\
+2^{\mathsf{ufp}(c_1)+\mathsf{ufp}(c_2)-\mathsf{nsb}(c_1)-\mathsf{nsb}(c_2)} & &
\end{array} \tag{5.14}$$

We get rid of the last term $2^{\mathsf{ufp}(c_1)+\mathsf{ufp}(c_2)-\mathsf{nsb}(c_1)-\mathsf{nsb}(c_2)}$ of the error $\varepsilon(c)$ which is strictly less than the former two ones. By assuming that $\mathsf{ufp}(c_1+c_2) = \mathsf{ufp}(c)$ and by reasoning on the exponents, we obtain the equations of Rule (MULT):

$$\mathsf{nsb}(\ell_1) \geqslant \mathsf{nsb}(\ell) + \xi(\ell)(\ell_1,\ell_2) - 1 \quad \text{and} \quad \mathsf{nsb}(\ell_2) \geqslant \mathsf{nsb}(\ell) + \xi(\ell)(\ell_1,\ell_2) - 1\ .$$

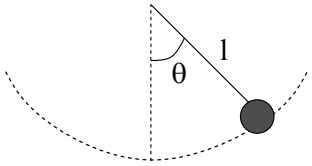
For the elementary functions such as logarithm, exponential, the hyperbolic and trigonometric functions gathered in Rule (MATH), the principle remains the same as for the SMT-based method. We consider that each elementary function introduces a loss of precision of φ bits, where $\varphi \in \mathbb{N}$ is a parameter of the analysis.

The rules of commands are rather classical, we use control points to distinguish many assignments of the same variable and also to implement joins in conditions and loops. Given a command c and an environment ϱ, $\mathcal{C}[\![c]\!]\ \varrho$ returns a pair (C,ϱ') made of a set C of constraints and of a new environment ϱ'. The function $\mathcal{C}$ is defined by induction on the structure of commands in figures 5.1 and 5.2. For conditionals, we generate the constraints for the `then` and `else` branches plus additional constraints to join the results of both branches. Currently, we do not take care of the guards. As a result, we analyze both the `then` and `else` branches of the `if` statement without reducing the range of the variables and consequently their ufp. This is correct but it is a source of imprecision. Let us mention that we do not address the problem that can arise on the variables of the guard expressions when we reduce their precision. Consequently, the range of the variables

can change and this can affect the choice of the branch `then` or `else` to explore. Work in [CGL10] has addressed this kind of problem.

For loops, we relate the number of significant bits at the end of the `body` to the `nsb` of the same variables and the beginning of the loop. More precisely, our tool manages correctly the loops because a range analysis on the variable values has been performed before the tuning phase. However, performing a precise range analysis for loops is not a simple task and this can explain why the majority of the state-of-the-art static tools do not handle loops.

```
1  g = 9.81; l = 0.5;
2  y1 = 0.785398; y2 = 0.785398;
3  h = 0.1; t = 0.0;
4  while (t<10.0) {
5   y1new = y1 + y2 * h ;
6   aux1 = sin(y1) ;
7   aux2 = aux1 * h * g / l;
8   y2new = y2 - aux2;
9   t = t + h;
10  y1 = y1new; y2 = y2new;
11 };
12 require_nsb(y2,20);
```




```
1  g^ℓ1 = 9.81^ℓ0; l^ℓ3 = 0.5^ℓ2;
2  y1^ℓ5 = 0.785398^ℓ4; y2^ℓ7 = 0.785398^ℓ6;
3  h^ℓ9 = 0.1^ℓ8; t^ℓ11 = 0.0^ℓ10;
4  while (t^ℓ13 <^ℓ15 10.0^ℓ14)^ℓ59 {
5   y1new^ℓ24 = y1^ℓ17 +^ℓ23 y2^ℓ19 *^ℓ22 h^ℓ21;
6   aux1^ℓ28 = sin(y1^ℓ26)^ℓ27;
7   aux2^ℓ40 = aux1^ℓ30 *^ℓ39 h^ℓ32 *^ℓ38 g^ℓ34 /^ℓ37
        l^ℓ36;
8   y2new^ℓ46 = y2^ℓ42 -^ℓ45 aux2^ℓ44;
9   t^ℓ52 = t^ℓ48 +^ℓ51 h^ℓ50;
10  y1^ℓ55 = y1new^ℓ54; y2^ℓ58 = y2new^ℓ57;
11 };
12 require_nsb(y2,20)^ℓ61;
```

```
1  g|20| = 9.81|20|; l|20| = 1.5|20|;
2  y1|29| = 0.785398|29|; y2|21| = 0.0|21|;
3  h|21| = 0.1|21|; t|21| = 0.0|21|;
4  while (t<1.0) {
5    y1new|20| = y1|21| +|20| y2|21| *|22| h|21|;
6    aux1|20| = sin(y1|29|)|20|;
7    aux2|20| = aux1|19| *|20| h|18|*|19| g|17| /|18|
         l|17|;
8    y2new|20| = y2|21| -|20| aux2|18|;
9    t|20| = t|21| +|20| h|17|;
10   y1|20| = y1new|20|; y2|20|=y2new|20|;
11 };
12 require_nsb(y2,20);
```

Listing 5.1: Top left: source program. Top right: pendulum movement for $\theta = \frac{\pi}{4}$. Bottom left: program annotated with labels. Bottom right: program with inferred accuracies.

Example 5.2. A motivating example to better explain our method is given by the code snippet of Listing 5.1. In this example, we aim at modeling the movement of a simple pendulum without damping. Let $l = 0.5\ m$ be the length of this pendulum, $m = 1\ kg$ its mass and $g = 9.81\ m \cdot s^{-2}$ Newton's gravitational constant. We denote by θ the tilt angle in radians as shown in Listing 5.1 (initially $\theta = \frac{\pi}{4}$). The Equation describing the movement of the pendulum is given in Equation (5.15).

$$m \cdot l \cdot \frac{d^2\theta}{dt^2} = -m \cdot g \cdot \sin\theta \tag{5.15}$$

Equation (5.15) being a second order differential equation. We need to transform it into a system of two first order differential equations for resolution. We obtain $y_1 = \theta$ and $y_2 = \frac{d\theta}{dt}$.

By applying Euler's method to these last equations, we obtain Equation (5.16) implemented in Listing 5.1.

$$\frac{dy_1}{dt} = y_2 \quad \text{and} \quad \frac{dy_2}{dt} = -\frac{g}{l} \cdot \sin y_1 \tag{5.16}$$

Before analysis, we suppose that all variables of the source program in the top left corner of Figure 5.1 are in FP64 double precision and that a range determination is performed by dynamic analysis on the program variables. The minimal precision needed for the inputs and intermediary results satisfying the user assertion is observed on the bottom right corner of Listing 5.1. For instance, in Line 5 of this code, `y1new|20|` means that the variable needs 20 significant bits at this point. Similarly, `y1` and `y2` need 21 bits each and the addition requires 20 bits.

Let us take for example Line 5 of the pendulum program to analyze. By applying Rule (ASSIGN), Rule (ADD) and Rule (MULT) of Figure 5.1, we generate seven constraints as shown in Equation (5.17).

$$C_1 = \left\{ \begin{array}{l} \mathsf{nsb}(\ell_{17}) \geqslant \mathsf{nsb}(\ell_{23}) + (-1) + \xi(\ell_{23})(\ell_{17}, \ell_{22}) - (-1), \\ \mathsf{nsb}(\ell_{22} \geqslant \mathsf{nsb}(\ell_{23} + 0 + \xi(\ell_{23})(\ell_{17}, \ell_{22}) - (1), \\ \mathsf{nsb}(\ell_{19}) \geqslant \mathsf{nsb}(\ell_{22}) + \xi(\ell_{22})(\ell_{19}, \ell_{21}) - 1, \\ \mathsf{nsb}(\ell_{21}) \geqslant \mathsf{nsb}(\ell_{22}) + \xi(\ell_{22})(\ell_{19}, \ell_{21}) - 1, \\ \mathsf{nsb}(\ell_{23}) \geqslant \mathsf{nsb}(\ell_{24}),\ \xi(\ell_{23})(\ell_{17}, \ell_{22}) \geqslant 1,\ \ \xi(\ell_{22})(\ell_{19}, \ell_{21}) \geqslant 1 \end{array} \right\} \tag{5.17}$$

The first two constraints are for the addition. As mentioned previously, the ufp are computed by a prior range analysis. Then, at constraint generation time, they are constants. For our example, $\mathsf{ufp}(\ell_{17}) = -1$. This quantity occurs in the first constraints. The next two constraints are for the multiplication. The fifth constraint $\mathsf{nsb}(\ell_{23}) \geqslant \mathsf{nsb}(\ell_{24})$ is for the assignment and the last two constraints are for the constant functions $\xi(\ell_{23})(\ell_{17}, \ell_{22})$ and $\xi(\ell_{22})(\ell_{19}, \ell_{21})$, respectively for the addition and multiplication. For a user requirement of 20 bits on the variable `y2`, our tool succeeds in tuning the majority of variables of the pendulum program into FP32 single precision with a total number of bits at bit-level equivalent to 274 (originally the program used 689 bits). As a result, the new mixed precision formats obtained for Line 5 under discussion are:

```
y1new|20|=y1|21| +|20| y2|22| × |22| h|22|;
```

5.2.2 Policy Iteration for Optimized Carry Bit Propagation

The policy iteration algorithm discussed in Chapter 2 Section 2.2.5, is used to solve nonlinear fixpoint equations when the function is written as the infimum of functions for which a fixpoint can be easily computed. The infimum formulation makes the function not being differentiable in the classical sense. The one proposed in [CGG+05] to solve smallest fixpoint equations in static analysis requires the fact that the function is order-preserving to ensure the decrease of the intermediate solutions provided by the algorithm. In this thesis, because of the nature of the semantics, we propose a policy iteration algorithm for a non order-preserving function.

More precisely, let F be a map from a complete lattice L to itself (Equation (2.14)). The classical policy iteration solves $F(\mathbf{x}) = \mathbf{x}$ by generating a sequence $(\mathbf{x}^k)_k$ such that $f^{\pi^k}(\mathbf{x}^k) = \mathbf{x}^k$ and $\mathbf{x}^{k+1} < \mathbf{x}^k$. We denoted the set Π by the set of policies and by f^π a policy map (associated to π). We recall that the set of policy maps has to satisfy the selection property meaning that for all $\mathbf{x} \in L$, there exists $\pi \in \Pi$ such that $F(\mathbf{x}) = f^\pi(\mathbf{x})$ (see Definition 2.18). This is exactly the same as for each $\mathbf{x} \in L$, the minimization problem $\mathrm{Min}_{\pi\in\Pi} f^\pi(\mathbf{x})$ has an optimal solution. If Π is finite and F is order-preserving, policy iteration converge in finite time to a fixpoint of F. The number of iterations is bounded from above by the number of policies. Indeed, a policy cannot be selected twice in the running of the algorithm. This is implied by the fact that the smallest fixpoint of a policy map is computed.

In our thesis, we adapt policy iteration to the problem of precision tuning. The function F here is constructed from inequalities depicted in Figure 5.1 and Figure 5.2. We thus have naturally constraints of the form $F(\mathbf{x}) \leqslant \mathbf{x}$. We will give details about the construction of F at Proposition 5.1. Consequently, we are interested in solving:

$$\underset{\mathsf{nsb},\mathsf{nsb_e}}{\mathrm{Min}} \quad \sum_\ell \mathsf{nsb}(\ell) \text{ s. t. } F\begin{pmatrix}\mathsf{nsb}\\ \mathsf{nsb_e}\end{pmatrix} \leqslant \begin{pmatrix}\mathsf{nsb}\\ \mathsf{nsb_e}\end{pmatrix} \quad \mathsf{nsb} \in \mathbb{N}^{Lab},\ \mathsf{nsb_e} \in \mathbb{N}^{Lab} \ . \tag{5.18}$$

Let $\xi : Lab \to \{0, 1\}$. We write S^1_ξ the system of inequalities depicted in Figure 5.1 and S^2_ξ the system of inequalities presented at Figure 5.2. Note that the final system of inequalities is $S_\xi = S^1_\xi \cup S^2_\xi$ meaning that we add new constraints to S^1_ξ. If the system S^1_ξ is used alone, ξ is the constant function equal to 1. Otherwise, ξ is defined by the formula at the end of Figure 5.2.

Proposition 5.1. The following results hold:

1. Let ξ the constant function equal to 1. The system S^1_ξ can be rewritten as $\{\mathsf{nsb} \in \mathbb{N}^{Lab} : F(\mathsf{nsb}) \leqslant (\mathsf{nsb})\}$ where F maps $\mathbb{R}^{Lab}$ to itself, $F(\mathbb{N}^{Lab}) \subseteq (\mathbb{N}^{Lab})$ and has coordinates which are the maximum of a finite family of affine order-preserving functions.

2. Let ξ the function such that $\xi(\ell)$ equals the function of Figure 5.2. The system S_ξ can be rewritten as $\{(\mathsf{nsb}, \mathsf{nsb_e}) \in \mathbb{N}^{Lab} \times \mathbb{N}^{Lab} : F(\mathsf{nsb}, \mathsf{nsb_e}) \leqslant (\mathsf{nsb}, \mathsf{nsb_e})\}$ where F maps $\mathbb{R}^{Lab} \times \mathbb{R}^{Lab}$ to itself, $F(\mathbb{N}^{Lab} \times \mathbb{N}^{Lab}) \subseteq (\mathbb{N}^{Lab} \times \mathbb{N}^{Lab})$ and all its coordinates are the min-max of a finite family of affine functions.

Proof. We only give details about the system S^1_ξ (Figure 5.1). By induction on the rules. We write $L = \{\ell \in Lab : F_\ell \text{ is constructed }\}$. This set is used in the proof to construct F inductively. For Rule (CONST), there is nothing to do. For Rule (ID), if the label $\ell' = \rho(x) \in L$ then we define $F_{\ell'}(\mathsf{nsb}) = \max(F_{\ell'}(\mathsf{nsb}), \mathsf{nsb}(\ell))$. Otherwise, $F_{\ell'}(\mathsf{nsb}) = \mathsf{nsb}(\ell)$. As $\mathsf{nsb} \mapsto \mathsf{nsb}(\ell)$ is order-preserving and the maximum of one affine function, $F_{\ell'}$ is the maximum of a finite family of order-preserving affine functions since max preserves order-preservation. For rules (ADD), (SUB), (MULT), (DIV), (MATH) and (ASSIGN), by induction, it suffices to focus on the new set of inequalities. If $\ell_1 \in L$, we define F_{ℓ_1} as the max with old definition and $RHS(\mathsf{nsb})$ i.e. $F_{\ell_1}(\mathsf{nsb}) = \max(RHS(\mathsf{nsb}), F_{\ell_1}(\mathsf{nsb}))$ where $RHS(\mathsf{nsb})$ is the right-hand

side part of the new inequality. If $\ell_1 \notin L$, we define $F_{\ell_1}(\mathsf{nsb}) = RHS(\mathsf{nsb})$. In the latter rules, $RHS(\mathsf{nsb})$ are order-preserving affine functions. It follows that F_{ℓ_1} is the maximum of a finite family of order-preserving affine functions. The result follows by induction for Rule (SEQ). Rules (COND) and (WHILE) are treated as rules (ADD), (SUB), (MULT), (DIV), (MATH) and (ASSIGN), by induction and the consideration of the new set of inequalities. Rule (REQ) constructs $F_{\rho(x)}$ either as the constant function equal to p at label $\rho(x)$ or the maximum of the old definition of $F_{\rho(x)}$ and p if $\rho(x) \in L$. ■

Note that, in the first case, F does not map from $\mathbb{R}^{Lab} \times \mathbb{R}^{Lab}$ to itself. It is easy to extend F as a map from $\mathbb{R}^{Lab} \times \mathbb{R}^{Lab}$ to itself without affecting its intrinsic behaviour. From Proposition 5.1, when S_{ξ} is used, we can write F as $F = \min_{\pi \in \Pi} f^{\pi}$, where f^{π} is the maximum of a finite family of affine functions and thus used a modified policy iteration algorithm. The set of policies here is a map $\pi : Lab \mapsto \{0,1\}$. A choice is thus a vector of 0 or 1. A policy map f^{π} is a function $\mathbb{N}^{Lab}$ to itself such that the coordinates are $f^{\pi}_{\ell}(\ell)$. If the coordinate $f^{\pi}_{\ell}(\ell)$ depends on ξ then $\xi(\ell) = \pi(\ell)$. Otherwise, the function is the maximum of affine functions and a choice is not required. Algorithm 2 depicts a modified policy iteration algorithm to solve our problem of precision tuning.

Corollary 5.1. Any feasible solution of Problem (5.18) satisfies our ILP constraints of Figure 5.1 (or Figure 5.2 if ξ is not fixed to 1).

Algorithm 2: Non-monotone Policy Iteration Algorithm

Result: An over-approximation of an optimal solution of Equation (5.18)
Let $k := 0$, $S := +\infty$;
Choose $\pi^0 \in \Pi$;
Select an optimal solution of $(\mathsf{nsb}^k, {\mathsf{nsb}_{\mathsf{e}}}^k)$ the integer linear program:

$$\text{Min} \left\{ \sum_{\ell \in Lab} \mathsf{nsb}(\ell) : f^{\pi^k}(\mathsf{nsb}, \mathsf{nsb}_{\mathsf{e}}) \leqslant (\mathsf{nsb}, \mathsf{nsb}_{\mathsf{e}}),\ \mathsf{nsb} \in \mathbb{N}^{Lab},\ \mathsf{nsb}_{\mathsf{e}} \in \mathbb{N}^{Lab} \right\} ;$$

if $\sum_{\ell \in Lab} \mathsf{nsb}^k(\ell) < S$ **then**
  $S := \sum_{\ell \in Lab} \mathsf{nsb}^k(\ell)$;
  Choose $\pi^{k+1} \in \Pi$ such that $F(\mathsf{nsb}^k, {\mathsf{nsb}_{\mathsf{e}}}^k) = f^{\pi^{k+1}}(\mathsf{nsb}^k, {\mathsf{nsb}_{\mathsf{e}}}^k)$;
  $k := k + 1$ and goto 3;
else
  Return S and nsb^k.
end

Proposition 5.2 (Algorithm correctness)**.** The sequence $(\sum_{\ell \in Lab} \mathsf{nsb}^k(\ell))_{0 \leqslant k \leqslant K}$ generated by Algorithm 2 satisfies the following properties:

1. $K < +\infty$ i.e. the sequence is of finite length,
2. Each term of the sequence furnishes a feasible solution for Problem (5.18),
3. $\sum_{\ell \in Lab} \mathsf{nsb}^{k+1}(\ell) < \sum_{\ell \in Lab} \mathsf{nsb}^k(\ell)$ if $k < K - 1$ and $\sum_{\ell \in Lab} \mathsf{nsb}^K(\ell) = \sum_{\ell \in Lab} \mathsf{nsb}^{K-1}(\ell)$,

4. The number k is smaller than the number of policies.

Proof. Let $\sum_{\ell\in Lab} \mathsf{nsb}^k(\ell)$ be a term of the sequence and $(\mathsf{nsb}^k, {\mathsf{nsb}_\mathsf{e}}^k)$ be the optimal solution of $\mathrm{Min}\{\sum_{\ell\in Lab} \mathsf{nsb}(\ell) : f^{\pi^k}(\mathsf{nsb}, \mathsf{nsb}_\mathsf{e}) \leqslant (\mathsf{nsb}, \mathsf{nsb}_\mathsf{e}),\ \mathsf{nsb} \in \mathbb{N}^{Lab},\ \mathsf{nsb}_\mathsf{e} \in \mathbb{N}^{Lab}\}$. Then, $F(\mathsf{nsb}^k, {\mathsf{nsb}_\mathsf{e}}^k) \leqslant f^{\pi^k}(\mathsf{nsb}^k, {\mathsf{nsb}_\mathsf{e}}^k)$ by definition of F. Moreover, $F(\mathsf{nsb}^k, {\mathsf{nsb}_\mathsf{e}}^k) = f^{\pi^{k+1}}(\mathsf{nsb}^k, {\mathsf{nsb}_\mathsf{e}}^k)$ and $f^{\pi^k}(\mathsf{nsb}^k, {\mathsf{nsb}_\mathsf{e}}^k) \leqslant (\mathsf{nsb}^k, {\mathsf{nsb}_\mathsf{e}}^k)$. This proves the second statement. Furthermore, it follows that $f^{\pi^{k+1}}(\mathsf{nsb}^k, {\mathsf{nsb}_\mathsf{e}}^k) \leqslant (\mathsf{nsb}^k, {\mathsf{nsb}_\mathsf{e}}^k)$ and $(\mathsf{nsb}^k, {\mathsf{nsb}_\mathsf{e}}^k)$ is feasible for the minimisation problem for which $(\mathsf{nsb}^{k+1}, {\mathsf{nsb}_\mathsf{e}}^{k+1})$ is an optimal solution. We conclude that $\sum_{\ell\in Lab} \mathsf{nsb}^{k+1}(\ell) \leqslant \sum_{\ell\in Lab} \mathsf{nsb}^k(\ell)$ and the algorithm terminates if the equality holds or continues as the criterion strictly decreases. Finally, from the strict decrease, a policy cannot be selected twice without terminating the algorithm. In conclusion, the number of iterations is smaller than the number of policies. ∎

Figure 5.2 displays the new rules that we add to the global system of constraints in which the only difference is to activate the optimized function ξ instead of its over-approximation in Figure 5.1. As mentioned in Definition 4.14 and Lemma 4.1, to compute the ulp of the errors on the operands, we need to estimate the number of bits of the error nsb_e for each operand on which all the rules of Figure 5.2 are based. By applying this reasoning, the problem do not remain an ILP any longer.

Let us concentrate on the rules of Figure 5.2. The function $\mathcal{E}'[e]\ \varrho$ generates the new set of constraints for an expression $e \in Expr$ in the environment ϱ. For Rule (Const′), the number of significant bits of the error $\mathsf{nsb}_\mathsf{e} = 0$ whereas we impose that the nsb_e of a variable x at control point ℓ is less than the last assignment of nsb_e in $\varrho(x)$ as shown in Rule (Id′) of Figure 5.2. Considering Rule (Add′), we start by generating the new set of constraints $\mathcal{E}'[e_1^{\ell_1}]\varrho$ and $\mathcal{E}'[e_2^{\ell_2}]\varrho$ on the operands at control points ℓ_1 and ℓ_2. Then, we require that $\mathsf{nsb}_\mathsf{e}(\ell) \geqslant \mathsf{nsb}_\mathsf{e}(\ell_1)$ and $\mathsf{nsb}_\mathsf{e}(\ell) \geqslant \mathsf{nsb}_\mathsf{e}(\ell_2)$ where the result of the addition is stored at control point ℓ. Additionally, $\mathsf{nsb}_\mathsf{e}(\ell)$ is computed as shown:

$$\mathsf{nsb}_\mathsf{e}(\ell) \geqslant \max\begin{pmatrix} \mathsf{ufp}(\ell_1) - \mathsf{nsb}(\ell_1) \\ \mathsf{ufp}(\ell_2) - \mathsf{nsb}(\ell_2) \end{pmatrix} - \min\begin{pmatrix} \mathsf{ufp}(\ell_1) - \mathsf{nsb}(\ell_1) - \mathsf{nsb}_\mathsf{e}(\ell_1) \\ \mathsf{ufp}(\ell_2) - \mathsf{nsb}(\ell_2) - \mathsf{nsb}_\mathsf{e}(\ell_2) \end{pmatrix} + \xi(\ell)(\ell_1, \ell_2)\ . \tag{5.19}$$

By breaking the min and max operators, we obtain the constraints on $\mathsf{nsb}_\mathsf{e}(\ell)$ of Rule (Add′). For the subtraction, the constraints generated are similar to the addition case. Considering now Rule (Mult′), as we have defined in Section 5.2.1, $\varepsilon(c) = c_1 \cdot \varepsilon(c_2) + c_2 \cdot \varepsilon(c_1) + \varepsilon(c_1) \cdot \varepsilon(c_2)$ where $c = c_1^{\ell_1} \times^\ell c_2^{\ell_2}$. By reasoning on ulp_e, we bound $\varepsilon(c)$ by

$$\varepsilon(c) \leqslant 2^{\mathsf{ufp}(c_1)} \cdot 2^{\mathsf{ufp}(c_2)-\mathsf{nsb}(c_2)-\mathsf{nsb}_\mathsf{e}(c_2)+1} + 2^{\mathsf{ufp}(c_2)} \cdot 2^{\mathsf{ufp}(c_1)-\mathsf{nsb}(c_1)-\mathsf{nsb}_\mathsf{e}(c_1)+1} + 2^{\mathsf{ufp}(c_2)+\mathsf{ufp}(c_1)-\mathsf{nsb}(c_1)-\mathsf{nsb}(c_2)-\mathsf{nsb}_\mathsf{e}(c_1)-\mathsf{nsb}_\mathsf{e}(c_2)+2}\ .$$

By selecting the smallest term $\mathsf{ufp}(c_2) + \mathsf{ufp}(c_1) - \mathsf{nsb}(c_1) - \mathsf{nsb}(c_2) - \mathsf{nsb}_\mathsf{e}(c_1) - \mathsf{nsb}_\mathsf{e}(c_2) + 2$, we obtain that

$$\mathsf{nsb}_\mathsf{e}(\ell) \geqslant \max\begin{pmatrix} \mathsf{ufp}(\ell_1) + \mathsf{ufp}(\ell_2) - \mathsf{nsb}(\ell_1) \\ \mathsf{ufp}(\ell_1) + \mathsf{ufp}(\ell_2) - \mathsf{nsb}(\ell_2) \end{pmatrix} - \begin{array}{l} \mathsf{ufp}(\ell_1) + \mathsf{ufp}(\ell_2) - \mathsf{nsb}(\ell_1) - \\ \mathsf{nsb}(\ell_2) - \mathsf{nsb}_\mathsf{e}(\ell_1) - \mathsf{nsb}_\mathsf{e}(\ell_2) + 2 \end{array}\ .$$

$$\mathcal{E}'[c\#\mathrm{p}^{\ell}]\varrho = \{\mathsf{nsb_e}(\ell) = 0\} \quad (\textsc{Const}') \qquad \mathcal{E}'[x^{\ell}]\varrho = \{\mathsf{nsb_e}(\varrho(x)) \geqslant \mathsf{nsb_e}(\ell)\} \quad (\textsc{Id}')$$

$$\mathcal{E}'[e_1^{\ell_1} +^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}'[e_1^{\ell_1}]\varrho \;\cup\; \mathcal{E}'[e_2^{\ell_2}]\varrho \quad (\textsc{Add}')$$
$$\cup$$
$$\left\{ \begin{array}{c} \mathsf{nsb_e}(\ell) \geqslant \mathsf{nsb_e}(\ell_1),\ \mathsf{nsb_e}(\ell) \geqslant \mathsf{nsb_e}(\ell_2), \\ \mathsf{nsb_e}(\ell) \geqslant \mathsf{ufp}(\ell_1) - \mathsf{ufp}(\ell_2) + \mathsf{nsb}(\ell_2) - \mathsf{nsb}(\ell_1) + \mathsf{nsb_e}(\ell_2) + \xi(\ell)(\ell_1,\ell_2), \\ \mathsf{nsb_e}(\ell) \geqslant \mathsf{ufp}(\ell_2) - \mathsf{ufp}(\ell_1) + \mathsf{nsb}(\ell_1) - \mathsf{nsb}(\ell_2) + \mathsf{nsb_e}(\ell_1) + \xi(\ell)(\ell_1,\ell_2) \end{array} \right\}$$

$$\mathcal{E}'[e_1^{\ell_1} -^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}'[e_1^{\ell_1}]\varrho \;\cup\; \mathcal{E}'[e_2^{\ell_2}]\varrho \quad (\textsc{Sub}')$$
$$\cup$$
$$\left\{ \begin{array}{c} \mathsf{nsb_e}(\ell) \geqslant \mathsf{nsb_e}(\ell_1),\ \mathsf{nsb_e}(\ell) \geqslant \mathsf{nsb_e}(\ell_2), \\ \mathsf{nsb_e}(\ell) \geqslant \mathsf{ufp}(\ell_1) - \mathsf{ufp}(\ell_2) + \mathsf{nsb}(\ell_2) - \mathsf{nsb}(\ell_1) + \mathsf{nsb_e}(\ell_2) + \xi(\ell)(\ell_1,\ell_2), \\ \mathsf{nsb_e}(\ell) \geqslant \mathsf{ufp}(\ell_2) - \mathsf{ufp}(\ell_1) + \mathsf{nsb}(\ell_1) - \mathsf{nsb}(\ell_2) + \mathsf{nsb_e}(\ell_1) + \xi(\ell)(\ell_1,\ell_2) \end{array} \right\}$$

$$\mathcal{E}'[e_1^{\ell_1} \times^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}'[e_1^{\ell_1}]\varrho \;\cup\; \mathcal{E}'[e_2^{\ell_2}]\varrho \quad (\textsc{Mult}')$$
$$\cup$$
$$\left\{ \mathsf{nsb_e}(\ell) \geqslant \mathsf{nsb}(\ell_1) + \mathsf{nsb_e}(\ell_1) + \mathsf{nsb_e}(\ell_2) - 2,\ \mathsf{nsb_e}(\ell) \geqslant \mathsf{nsb}(\ell_2) + \mathsf{nsb_e}(\ell_2) + \mathsf{nsb_e}(\ell_1) - 2 \right\}$$

$$\mathcal{E}'[e_1^{\ell_1} \div^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}'[e_1^{\ell_1}]\varrho \;\cup\; \mathcal{E}'[e_2^{\ell_2}]\varrho \quad (\textsc{Div}')$$
$$\cup$$
$$\left\{ \mathsf{nsb_e}(\ell) \geqslant \mathsf{nsb}(\ell_1) + \mathsf{nsb_e}(\ell_1) + \mathsf{nsb_e}(\ell_2) - 2,\ \mathsf{nsb_e}(\ell) \geqslant \mathsf{nsb}(\ell_2) + \mathsf{nsb_e}(\ell_2) + \mathsf{nsb_e}(\ell_1) - 2 \right\}$$

$$\mathcal{E}'\left[\sqrt{e^{\ell_1}}^{\ell}\right]\varrho = \mathcal{E}'[e_1^{\ell_1}]\varrho \;\cup\; \{\mathsf{nsb_e}(\ell) \geqslant \mathsf{nsb_e}(\ell_1)\} \quad (\textsc{Sqrt}')$$

$$\mathcal{E}'\left[\phi\left(e^{\ell_1}\right)^{\ell}\right]\varrho = \mathcal{E}'[e_1^{\ell_1}]\varrho \;\cup\; \{\mathsf{nsb_e}(\ell) \geqslant +\infty\} \text{ with } \phi \in \{\sin, \cos, \tan, \log, \ldots\} \quad (\textsc{Math}')$$

$$\mathcal{C}'\left[x :=^{\ell} e^{\ell_1}\right]\varrho = \left(C, \varrho\,[x \mapsto \ell]\right) \text{ where } C = \mathcal{E}'[e_1^{\ell_1}]\varrho \cup \{\mathsf{nsb_e}(\ell_1) \geqslant \mathsf{nsb_e}(\ell)\} \quad (\textsc{Assign}')$$

$$\mathcal{C}'\left[c_1^{\ell_1}; c_2^{\ell_2}\right]\varrho = \left(C_1 \cup C_2, \varrho_2\right) \text{ with } \left(C_1, \varrho_1\right) = \mathcal{C}'\left[c_1^{\ell_1}\right]\varrho \text{ and } \left(C_2, \varrho_2\right) = \mathcal{C}'\left[c_2^{\ell_2}\right]\varrho_1 \quad (\textsc{Seq}')$$

$$\mathcal{C}'[\texttt{if}^{\ell}\ e^{\ell_0}\ \texttt{then}\ c^{\ell_1}\ \texttt{else}\ c^{\ell_2}]\ \varrho = (C_1 \cup C_2 \cup C_3, \varrho')$$
$$\text{where} \left| \begin{array}{l} \forall x \in \mathrm{Id},\ \varrho'(x) = \ell,\ (C_1, \varrho_1) = \mathcal{C}'[c_1^{\ell_1}]\ \varrho,\ (C_2, \varrho_2) = \mathcal{C}'[c_2^{\ell_2}]\ \varrho, \\ C_3 = \bigcup\limits_{x \in \mathrm{Id}} \{\mathsf{nsb_e}(\varrho_1(x)) \geqslant \mathsf{nsb_e}(\ell),\ \mathsf{nsb_e}(\varrho_2(x)) \geqslant \mathsf{nsb_e}(\ell)\} \end{array} \right. \quad (\textsc{Cond}')$$

$$\mathcal{C}'[\texttt{while}^{\ell}\ e^{\ell_0}\ \texttt{do}\ c^{\ell_1}]\ \varrho = (C_1 \cup C_2, \varrho')$$
$$\text{where} \left| \begin{array}{l} \forall x \in \mathrm{Id},\ \varrho'(x) = \ell,\ (C_1, \varrho_1) = \mathcal{C}'[c_1^{\ell_1}]\ \varrho' \\ C_2 = \bigcup\limits_{x \in \mathrm{Id}} \{\mathsf{nsb_e}(\varrho(x)) \geqslant \mathsf{nsb_e}(\ell),\ \mathsf{nsb_e}(\varrho_1(x)) \geqslant \mathsf{nsb_e}(\ell)\} \end{array} \right. \quad (\textsc{While}')$$

$$\mathcal{C}'[\texttt{require_nsb}(x, \mathrm{p})^{\ell}]\varrho = \varnothing \quad (\textsc{Req}')$$

$$\xi(\ell)(\ell_1,\ell_2) = \min \left(\begin{array}{l} \max\left(\mathsf{ufp}(\ell_2) - \mathsf{ufp}(\ell_1) + \mathsf{nsb}(\ell_1) - \mathsf{nsb}(\ell_2) - \mathsf{nsb_e}(\ell_2), 0\right), \\ \max\left(\mathsf{ufp}(\ell_1) - \mathsf{ufp}(\ell_2) + \mathsf{nsb}(\ell_2) - \mathsf{nsb}(\ell_1) - \mathsf{nsb_e}(\ell_1), 0\right), 1 \end{array} \right)$$

Figure 5.2: Constraints solved by PI with min and max carry bit formulation.

Finally, by simplifying the equation above we found the constraints of Rule (MULT′) in Figure 5.2 (same for Rule (DIV′)). For Rule (SQRT′), we generate the constraints on the expression $\mathcal{E}'[e_1^{\ell_1}]\varrho$ and we require that $\mathsf{nsb_e}$ of the result stored at control point ℓ is greater than the $\mathsf{nsb_e}$ of the expression a control point ℓ_1. For Rule (MATH′) , we assume that $\mathsf{nsb_e}(\ell)$ is unbounded.

Concerning the commands, we define the set $\mathcal{C}'[c]\ \varrho$ which has the same function as $\mathcal{C}$ defined in Figure 5.1. The reasoning on the commands also remains similar except that this time we reason on the number of bits of the errors $\mathsf{nsb_e}$. The only difference is in Rule (REQ′) where the set of constraints is empty. Let us recall that the constraints of Figure 5.2 are added to the former constraints of Figure 5.1 and are sent to a linear solver. We highlight the resolution of constraints in the next chapter.

Now, let us take again the pendulum program of Listing 5.1. By analyzing Line 5 of our program, we have to add the following set of constraints C_2 of Equation (5.20), along with the former set C_1 of Equation (5.17). In fact, policy iteration makes it possible to break the min in the $\xi(\ell_{23})(\ell_{17},\ell_{22})$ function by choosing the max between $\mathsf{ufp}(\ell_{22}) - \mathsf{ufp}(\ell_{17}) - \mathsf{nsb}(\ell_{17}) - \mathsf{nsb}(\ell_{22}) - \mathsf{nsb_e}(\ell_{17})$ and 0, the max between $\mathsf{ufp}(\ell_{17}) - \mathsf{ufp}(\ell_{22}) + \mathsf{nsb}(\ell_{22}) - \mathsf{nsb}(\ell_{17}) - \mathsf{nsb_e}(\ell_{22})$ and 0, and the constant 1. Next, it becomes possible to solve the corresponding ILP. If no fixpoint is reached, our tool iterates until a solution is found. By applying this optimization, the new formats are presented in Line 5 of the bottom right corner of Figure 5.1:

```
y1new|20|=y1|21| +|20| y2|21| × |22| h|21|;
```

By comparing the formats obtained with the pure ILP formulation, a gain of precision of 1 bit is observed on variables `y2` and `h` (total of 272 bits at bit level for the optimized program).

$$C_2 = \left\{ \begin{array}{l} \mathsf{nsb_e}(\ell_{23}) \geqslant \mathsf{nsb_e}(\ell_{17}), \mathsf{nsb_e}(\ell_{23}) \geqslant \mathsf{nsb_e}(\ell_{22}), \\ \mathsf{nsb}(\ell_{23}) \geqslant -1 - 0 + \mathsf{nsb}(\ell_{22}) - \mathsf{nsb}(\ell_{17}) + \mathsf{nsb_e}(\ell_{22}) + \xi(\ell_{23},\ell_{17},\ell_{22}), \\ \mathsf{nsb_e}(\ell_{23}) \geqslant 0 - (-1) + \mathsf{nsb}(\ell_{17}) - \mathsf{nsb}(\ell_{22}) + \mathsf{nsb_e}(\ell_{17}) + \xi(\ell_{23},\ell_{17},\ell_{22}), \\ \mathsf{nsb_e}(\ell_{23}) \geqslant \mathsf{nsb_e}(\ell_{24}), \mathsf{nsb_e}(\ell_{22}) \geqslant \mathsf{nsb}(\ell_{19}) + \mathsf{nsb_e}(\ell_{19}) + \mathsf{nsb_e}(\ell_{21}) - 2, \\ \mathsf{nsb_e}(\ell_{22}) \geqslant \mathsf{nsb}(\ell_{21}) + \mathsf{nsb_e}(\ell_{21}) + \mathsf{nsb_e}(\ell_{19}) - 2, \\ \xi(\ell_{23})(\ell_{17},\ell_{22}) = \min \left(\begin{array}{l} \max\left(0 - 6 + \mathsf{nsb}(\ell_{17}) - \mathsf{nsb}(\ell_{22}) - \mathsf{nsb_e}(\ell_{17}), 0\right), \\ \max\left(6 - 0 + \mathsf{nsb}(\ell_{22}) - \mathsf{nsb}(\ell_{17}) - \mathsf{nsb_e}(\ell_{22}), 0\right), 1 \end{array} \right) \end{array} \right\} \quad (5.20)$$

Indeed, we believe that the PI method exhibits better results if we analyze large codes with a lot of operations which has to compute with some tens of nsb. In this case, adding one bit is far from being negligible. Chapter 9 confirms these results on several benchmarks (with more than 400 LOCs).

5.3 Correctness

In this section, we present proofs of correctness concerning the soundness of the analysis (Section 5.3.1) and the integer nature of the solutions (Section 5.3.2).

$$\frac{\varrho(x) = c\#\mathsf{p}}{\langle x^{\ell}, \varrho\rangle \longrightarrow \langle c^{\ell}\#\mathsf{p}, \varrho\rangle}$$

$$\frac{c = c_1 \odot c_2,\ \mathsf{p} = \mathsf{ufp}(c) - \mathsf{ufp_e}\big(c^{\ell}\#\mathsf{p}\big)}{\langle c_1^{\ell_1}\#p_1 \odot^{\ell} c_2^{\ell_2}\#p_2, \varrho\rangle \longrightarrow \langle c\#\mathsf{p}, \varrho\rangle} \quad \odot \in \{+, -, \times, \div\}$$

$$\frac{\langle e_1^{\ell_1}, \varrho\rangle \longrightarrow \langle e_1'^{\ell_1}, \varrho\rangle}{\langle e_1^{\ell_1} \odot^{\ell} e_2^{\ell_2}, \varrho\rangle \longrightarrow \langle e_1'^{\ell_1} \odot^{\ell} e_2^{\ell_2}, \varrho\rangle} \qquad \frac{\langle e_2^{\ell_2}, \varrho\rangle \longrightarrow \langle e_2'^{\ell_2}, \varrho\rangle}{\langle c_1^{\ell_1}\#\mathsf{p} \odot^{\ell} e_2^{\ell_2}, \varrho\rangle \longrightarrow \langle c_1^{\ell_1}\sharp\mathsf{p} \odot^{\ell} e_2'^{\ell_2}, \varrho\rangle}$$

Figure 5.3: Small-step operational semantics of arithmetic expressions.

5.3.1 Soundness of the Constraint System

Let $\equiv$ denote the syntactic equivalence and let $e^{\ell} \in Expr$ be an expression. We write $\mathsf{Const}(e^{\ell})$ the set of constants occurring in the expression e^{ℓ}. For example, $\mathsf{Const}(18.0^{\ell_1} \times^{\ell_2} x^{\ell_3} +^{\ell_4} 12.0^{\ell_5} \times^{\ell_6} y^{\ell_7} +^{\ell_8} z^{\ell_9}) = \{18.0^{\ell_1}, 12.0^{\ell_5}\}$. Also, we denote by $\tau : \mathrm{Lab} \to \mathbb{N}$ a function mapping the labels of an expression to a nsb. The notation $\tau \models \mathcal{E}[e^{\ell}]\varrho$ means that τ is the minimal solution to the ILP $\mathcal{E}[e^{\ell}]\varrho$. We write $\varrho_{\perp}$ the empty environment ($dom(\varrho_{\perp}) = \varnothing$). The small-step operational semantics of our language is displayed in Figure 5.3. It is standard, the only originality being to indicate explicitly the nsb of constants. For the result of an elementary operation, this nsb is computed in function of the nsb of the operands. Lemma 5.1 asses the soundness of the constraints for one step of the semantics.

Lemma 5.1. Given an expression $e^{\ell} \in Expr$, if $e^{\ell} \to e'^{\ell}$ and $\tau \models \mathcal{E}[e^{\ell}]\varrho_{\perp}$ then for all $c^{\ell_c}\#\mathsf{p} \in \mathsf{Const}(e'^{\ell})$ we have $\mathsf{p} = \tau(\ell_c)$.

Proof. By case examination of the rules of Figure 5.1. Hereafter, we focus on the most interesting case of addition of two constants. Recall that $\mathsf{ufp_e}(\ell) = \mathsf{ufp}(\ell) - \mathsf{nsb}(\ell)$ for any control point ℓ. Assuming that $e^{\ell} \equiv c_1^{\ell_1} +^{\ell} c_2^{\ell_2}$ then by following the reduction rule of Figure 5.3, we have $e^{\ell} \to c^{\ell}\#\mathsf{p}$ with $\mathsf{p} = \mathsf{ufp}(c) - \mathsf{ufp_e}\big(c\big)$. On the other side, by following the set of constraints of Rule (ADD) in Figure 5.1 we have $\mathcal{E}[e^{\ell}]\varrho = \{\mathsf{nsb}(\ell_1) \geqslant \mathsf{nsb}(\ell) + \mathsf{ufp}(\ell_1) - \mathsf{ufp}(\ell) + \xi(\ell)(\ell_1, \ell_2),\ \mathsf{nsb}(\ell_2) \geqslant \mathsf{nsb}(\ell) + \mathsf{ufp}(\ell_2) - \mathsf{ufp}(\ell) + \xi(\ell)(\ell_1, \ell_2)\}$. These constraints can be written as

$$\begin{aligned} \mathsf{nsb}(\ell) &\leqslant \mathsf{ufp}(\ell) - \mathsf{ufp}(\ell_1) + \mathsf{nsb}(\ell_1) - \xi(\ell)(\ell_1, \ell_2) \\ \mathsf{nsb}(\ell) &\leqslant \mathsf{ufp}(\ell) - \mathsf{ufp}(\ell_2) + \mathsf{nsb}(\ell_2) - \xi(\ell)(\ell_1, \ell_2) \end{aligned}$$

and may themselves be rewritten as in Equation (5.13), i.e.

$$\mathsf{nsb}(\ell) \leqslant \mathsf{ufp}(\ell) - \max\big(\mathsf{ufp}(\ell_1) - \mathsf{nsb}(\ell_1), \mathsf{ufp}(\ell_2) - \mathsf{nsb}(\ell_2)\big) - \xi(\ell)(\ell_1, \ell_2)\ .$$

Since, obviously, $\mathsf{ufp}(c) = \mathsf{ufp}(\ell)$ and since the solver finds the minimal solution to the ILP problem, it remains to show that

$$\mathsf{ufp_e}(\ell) = \max\big(\mathsf{ufp}(\ell_1) - \mathsf{nsb}(\ell_1), \mathsf{ufp}(\ell_2) - \mathsf{nsb}(\ell_2), \mathsf{ufp}(\ell) - \mathsf{prec}(\ell)\big) + \xi(\ell)(\ell_1, \ell_2)$$

which corresponds to the assertion of Equation(2.7). Consequently, $\mathsf{nsb}(\ell) = \mathsf{p}$ as required, for this case, in Figure 5.3. ∎

Theorem 5.1. Given an expression $e^{\ell} \rightarrow e'^{\ell}$. If $e^{\ell} \rightarrow^{*} e'^{\ell}$ and if $\tau \models \mathcal{E}[e^{\ell}]\varrho_{\perp}$, then $\forall\, c^{\ell_c}\#\mathtt{p} \in \mathsf{Const}(e'^{\ell})$ we have $\mathtt{p} = \tau(\ell_c)$. ■

5.3.2 ILP Nature of the Problem

In this section, we give insights about the complexity of the problem. The computation relies on ILP which is known to belong to the class of NP-hard problems. A lower bound of the optimal value in a minimization problem can be furnished by the continuous linear programming relaxation. This relaxation is obtained by removing the integrity constraint. Recall that a (classical) linear program can be solved in polynomial-time. Then, we can solve our problem in polynomial-time if we can show that the continuous linear programming relaxation of our ILP has an unique optimal solution with integer coordinates. Proposition 5.3 presents a situation where a linear program has a unique optimal solution which is a vector of integers.

Proposition 5.3. Let $G : [0,+\infty)^d \mapsto [0,+\infty)^d$ be an order-preserving function such that $G(\mathbb{N}^d) \subseteq \mathbb{N}^d$. Suppose that the set $\{y \in \mathbb{N}^d \mid G(y) \leqslant y\}$ is non-empty. Let $\varphi : \mathbb{R}^d \mapsto \mathbb{R}$ a strictly monotone function such that $\varphi(\mathbb{N}^d) \subseteq \mathbb{N}$. Then, the minimization problem below has an unique optimal solution which is a vector of integers.

$$\underset{y \in [0,+\infty)^d}{\mathrm{Min}}\ \varphi(y)\ \text{ s.t. } G(y) \leqslant y$$

Proof. Let $L := \{x \in [0,+\infty)^d \mid G(x) \leqslant x\}$ and $u = \inf L$. It suffices to prove that $u \in \mathbb{N}^d$. Indeed, as φ is strictly monotone then $\varphi(u) < \varphi(x)$ for all $x \in [0,+\infty)^d$ s.t. $G(x) \leqslant x$ and $x \neq u$. The optimal solution is thus u. If $u = 0$, the result holds. Now suppose that $0 < u$, then $0 \leqslant G(0)$. Let $M := \{y \in \mathbb{N}^d \mid y \leqslant G(y), y \leqslant u\}$. Then $0 \in M$ and we write $v := \sup M$. As M is a complete lattice s.t. $G(M) \subseteq M$, from Theorem 2.1 (Tarski), v satisfies $G(v) = v$ and $v \leqslant u$. Moreover, $v \in \mathbb{N}^d$ and $v \leqslant u$. Again, from Theorem 2.1, u is the smallest fixpoint of G, it coincides with v. Then $u \in \mathbb{N}^d$. ■

Theorem 5.2. Assume that the system S of inequalities depicted in Figure 5.1 has a solution. The smallest amount of memory $\sum_{\ell \in Lab} \mathsf{nsb}(\ell)$ for S can be computed in polynomial-time by linear programming.

Proof. The function $\sum_{\ell \in Lab} \mathsf{nsb}(\ell)$ is strictly monotone and stable on integers. From the first statement of Proposition 5.1, the system of constraints is of the form $F(\mathsf{nsb}) \leqslant \mathsf{nsb}$ where F is order-preserving and stable on integers. By assumption, there exists a vector of integers nsb s.t. $F(\mathsf{nsb}) \leqslant \mathsf{nsb}$. We conclude from Proposition 5.3. ■

For the system of Figure 5.2, in practice, we get integer solutions to the continuous linear programming relaxation of our ILP of Equation (5.18). However, because of the lack of monotonicity of the functions for rules (ADD) and (SUB) of Figure 5.1, we cannot exploit Proposition 5.3 to prove the polynomial-time solvability.

5.4 Summary

We have presented in this chapter a new technique for precision tuning, clearly different from the existing ones. Instead of changing more or less randomly the data types of the numerical variables and running the programs to see what happens, we propose a semantic modelling of the propagation of the numerical errors throughout the code. This yields a system of constraints whose minimal solution gives the best tuning of the program. Two variants of constraints have been proposed:

- The first one corresponds to the SMT-based method.
- The second one corresponds to ILP-based method which are divided into two systems: a pure ILP for the over-approximation of the carry bit function and a refined constraints which optimizes the propagation of carries in the elementary operations and which can be solved using the policy iteration method (PI).

Proofs of correctness concerning the soundness of the analysis and the integer nature of the solutions have been presented. Compared to other approaches, the strength of our method is to find directly the minimal number of bits needed at each control point to get a certain accuracy on the results. Consequently, it is not dependant of a certain number of data types (e.g. the IEEE754 formats) and its complexity does not increase as the number of data types increases. The information provided may also be used to generate computations in the fixed-point arithmetic with an accuracy guaranty on the results. Concerning scalability, we generate a linear number of constraints and variables in the size of the analyzed program. The only limitation is the size of the problem accepted by the solver. We dedicate the next chapter for introducing our tool for precision tuning.

Chapter 6

The POP Tool

***

6.1 Generalities . . . 94
6.2 POP(SMT) . . . 95
6.2.1 Outline . . . 95
6.2.2 Benchmarks . . . 97
6.2.3 First Results . . . 97
6.3 POP(ILP) . . . 99
6.3.1 Outline . . . 99
6.3.2 Benchmarks . . . 100
6.4 Summary . . . 101

In this chapter, we present POP, short for Precision OPtimizer, the tool implemented during this thesis to assist precision tuning. In 2019, we have published the first version of POP [BMA19] based on the SMT-based method. We note this version POP(SMT). In 2021, we have implemented the ILP-based method in POP [ABM21] as a replacement of the first method. We call this second version POP(ILP). As a result, POP integrates these two sub-tools as depicted in Figure 6.1.

Notation: if we do not specify the method, we use the notation POP by default. We recall that POP(SMT) and POP(ILP) do not generate the same constraints but the technique remains the same. It is just the formulation of the problem which differs from one method to another.

The remainder of this chapter is as follows. Section 6.1 presents generalities of the POP tool. Section 6.2 gives an overview of POP(SMT), while Section 6.3 highlights the steps of analysis by POP(ILP). The summary is given in Section 6.4.

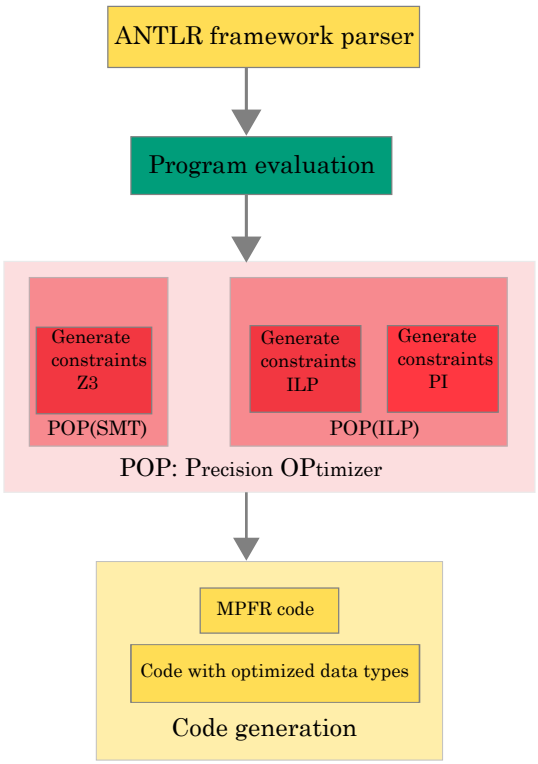


Figure 6.1: Functional architecture of POP.

6.1 Generalities

POP has been developed in JAVA. The different elements of the imperative language presented in Figure 4.1 are represented as packages gathering the different classes of their definitions. We illustrate the main tool hierarchy as follows:

- **Parser:** To parse the input source codes, POP uses the ANTLR tool v4.7.1 (ANother Tool for Language Recognition)[1]. From a grammar, ANTLR generates a parser that can build parse trees and also generates a listener interface (or visitor) that makes it easy to respond to the recognition of phrases of interest [Par13].

- **Range determination:** It consists in launching the execution of the program a certain number of times in order to determine dynamically the range of variables. More precisely, what we use in the tuning is the ufp of the values of the program. So we are 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 $[0.0, 1000.0]$. In future work, we plan to use a static analyzer.

- **Constraints generation:** It is the basis of our static approach for precision tuning. We recall that two kinds of system of constraints are generated. The first system models

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

the numerical errors propagation for the SMT-based method presented in Chapter 5 Section 5.1. The second system of constraints is generated for the ILP-based method in POP(ILP) introduced in Chapter 5 Section 5.2 . These two methods are based on the same transfer functions already discussed in Chapter 4.

- **Constraints resolution:** To find a solution for the constraints, the SMT-based method calls an SMT solver. Generally, SMT solvers combine SAT reasoning with specialized theory solvers either to find a feasible solution to a set of constraints or to prove that no such solution exists. The ILP-based method uses an LP solver to solve the constraints. The LP solvers come from the tradition of optimization, and are designed to find feasible solutions that are optimal with respect to some optimization function. We remind the reader that the difference between SMT solvers and LP solvers were discussed in Chapter 2 Section 2.3.

- **Code generation:** As output, POP generates an optimized version of the input source program annotated with the new nsb at each control point. We recall that our technique is independent of a particular computer arithmetic and the optimized formats are given in bit-level. However if we want these precision in the IEEE754 mode, the nsb obtained at bit-level is approximated by the upper number of bits corresponding to a IEEE754 format. For example, if a variable x has $\mathtt{nsb}(x) = 18$ bits, then x is tuned to the FP32 single precision. Besides, POP is able to generate a python MPFR [FHL+07] version of the input program. The interest of the MPFR codes is to measure the difference between the two programs (high precision assimilable to exact precision e.g. 500 bits and tuned) and to plot the curve of the difference in function of the theoretical error given by the user.

6.2 POP(SMT)

6.2.1 Outline

Figure 6.2 gives an architectural overview of POP(SMT) tool. To find a solution for our constraints, POP(SMT) calls the Z3 SMT solver[2]. In practice, Z3 is a state-of-the art theorem prover from Microsoft Research. It is targeted at solving problems that arise in software verification and software analysis [dMB08]. Consequently, it integrates support for a variety of theories. However, it is a low level tool that is often used as a component in the context of other tools that require solving logical formulas. In addition, the solutions returned by Z3 are not unique. In order to refine the solutions obtained in term of optimality, we add to our global system of constraints an additional constraint related to a cost function denoted by ϕ. We use the same definition in [Mar17]. The purpose of a cost function $\phi(c)$ of a given program c is to compute the sum of the accuracies (nsb) of all the variables

[2] https://github.com/Z3Prover/z3

and the intermediary values collected in each label of the arithmetic expressions as it is shown in Equation (6.1):

$$\phi(c) = \sum_{x \in Id, \ell \in Lab} \mathsf{nsb}(x^{\ell}) + \sum_{\ell \in Lab} \mathsf{nsb}(\ell) \ . \tag{6.1}$$

Next, POP(SMT) searches the smallest integer P such that our system of constraints admits a solution. Consequently, we start the binary search with $P \in [0{,}53 \times n]$ where all the values are in double precision and where n is the number of terms in Equation (6.1). While a solution is found for a given value of P, a new iteration of the binary search is run with a smaller value of P. When the solver fails for some P, a new iteration of the binary search is run with a larger P and we continue this process until convergence.

Concerning the complexity of the analysis performed by POP(SMT), in practice, the analysis is carried out by the SMT solver which solves the constraints. 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.

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

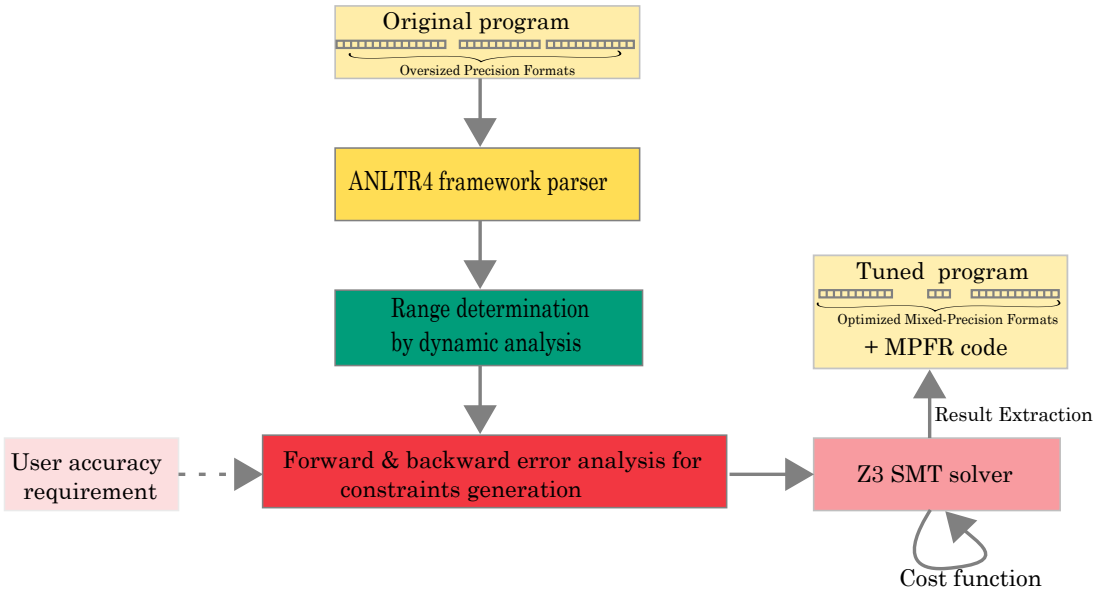


Figure 6.2: POP(SMT) overview.

6.2.2 Benchmarks

We evaluate POP(SMT) on several programs coming from different fields such as the GNU scientific library (GSL)[3] and the IoT domain. The benchmarks of POP(SMT) are the following:

- **Rotation** program [BMA19]: a rotation matrix-vector multiplication program.
- **Matrix determinant** program [BMA19]: computes the determinant $det(M)$ of 3×3 matrices.
- **Accelerometer** program [BM19]: measures the angle of inclination of an object. This program comes from the IoT field and will be highlighted in Chapter 7.
- **Pedometer** program:[BM20] counts the number of footsteps. This program derives also from the IoT field and will be detailed in Chapter 7.
- **Arclength** program: was first introduced in [Bai08] in which the task is to estimate the arc length of a given function.
- **Simpson** method: which corresponds to an implementation of the widely used Simpson's rule for integration in numerical analysis [McK62].

Chapters 7 and 8 presents a full evaluation of the performance of POP(SMT). The **rotation** and **matrix determinant** programs are highlighted in the next section.

6.2.3 First Results

We present in this section two simple examples which consist in a rotation matrix-vector multiplication and the computation of the determinant of 3×3 matrices and we present in Figure 6.3 some measures of the efficiency of our analysis on these two examples.

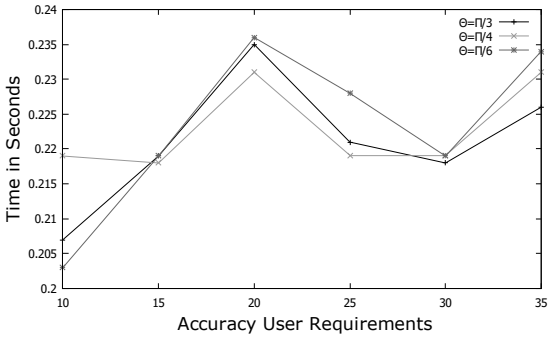
Example 1: Rotation Matrix-Vector Multiplication

Our first example consists in a rotation matrix R which is used in the rotation of vectors and tensors while the coordinate system remains fixed. For instance, we want to rotate a vector around the z axis by angle θ. The rotation matrix and the rotated column vectors are given by:

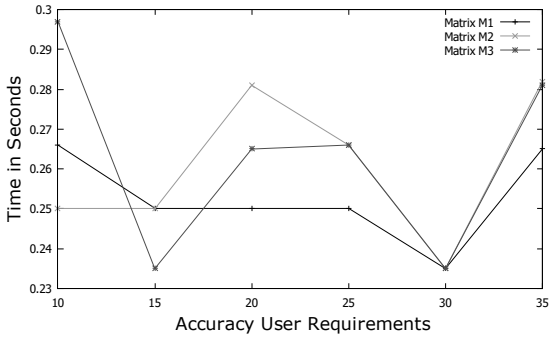
$$\begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} .$$

We aim from this experimentation to compute the performance of POP(SMT) from different angles of rotation $\frac{\pi}{3}$, $\frac{\pi}{4}$ and $\frac{\pi}{6}$, a variety of input vectors chosen with difference in magnitude $A = [1.0, 2.0, 3.0]$, $B = [10.0, 100.0, 500.0]$, $C = [100.0, 500.0, 1000.0]$, $D =$

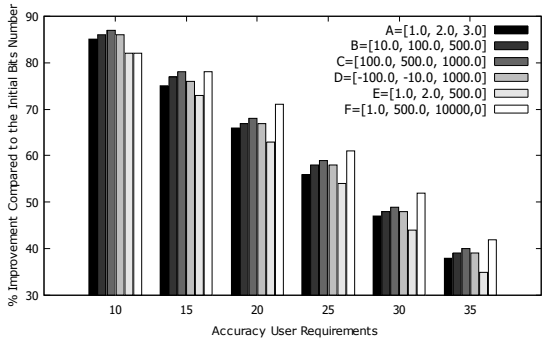
[3] https://www.gnu.org/software/gsl/



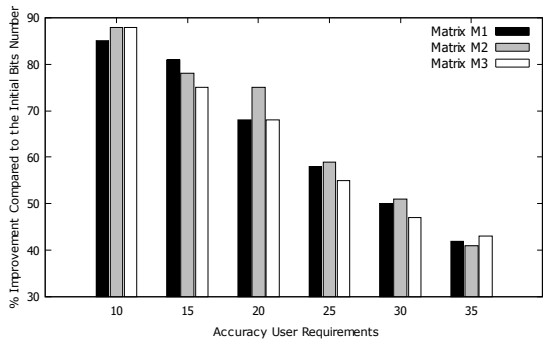
(a) POP(SMT) execution time for the rotation matrix-vector multiplication.



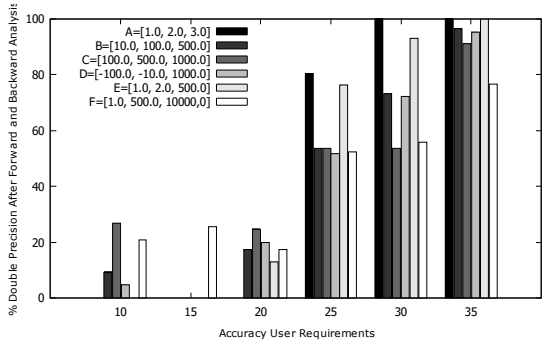
(b) POP(SMT) execution time for the a 3×3 matrix determinant.



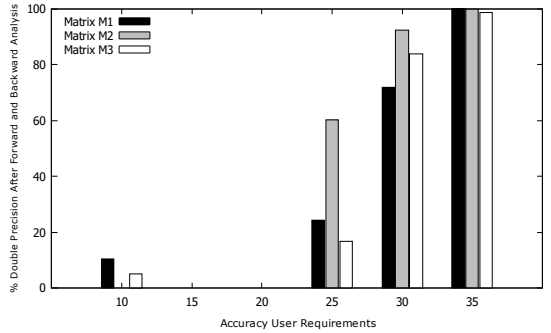
(c) Optimization of the number of bits compared to the original rotation matrix-vector multiplication program.



(d) Optimization of the number of bits compared to the original 3×3 determinant program.



(e) The percentage of the double precision variables after the forward and backward analysis for the first example for $\theta = \frac{\pi}{4}$.



(f) The percentage of the double precision variables after the forward and backward analysis for the second example.

Figure 6.3: Measures of the efficiency of the analysis of POP(SMT) on the two input examples: the time execution measure, the optimization of the number of bits of the transformed programs compared to the original ones and the percentage of the double precision variables after analysis.

$[-100.0, -10.0, 1000.0]$, $E = [1.0, 2.0, 500.0]$ and $F = [1.0, 500.0, 10000.0]$ and for nsb $= 10, 15, 20, 25, 30$ and 35 bits.

This example generates 858 constraints and 642 variables which are very manageable by the Z3 solver. Initially starting with 10335 bits for the original program (only variables in double precision), Figure 6.3c shows that the improvement in the number of bits needed to realize the user requirements ranges from 38% to 87% which confirms the usefulness of our analysis. Also, we can observe in Figure 6.3e that the majority of variables fits in single precision format for an nsb $\leqslant 35$ bits and that no double precision variables are noticed for vectors A, B, C, D and E for an nsb $= 15$ bits. For this example, we found that the variation of the angles of rotation do not have impact on the number of double precision variables after analysis that is why we choose only the angle $\frac{\pi}{4}$ in Figure 6.3e and by modifying the magnitude of the vectors at every turn. Besides, POP(SMT) assigns zeros to the accuracies of the variables that are not used by the program.

Example 2: Determinant of 3 × 3 Matrices

Our second example computes the determinant $det(M)$ of a 3×3 matrices $M1$, $M2$ and $M3$ as shown:

$$M = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \rightarrow det(M) = (a.e.i + d.h.c + g.b.f) - (g.e.c + a.h.f + d.b.i) \quad .$$

The matrices coefficients belong to multiple magnitude ranges: $M1 = \begin{bmatrix} [-50.1,50.1] & [-50.1,50.1] & [-50.1,50.1] \\ [-10.1,10.1] & [-10.1,10.1] & [-10.1,10.1] \\ [-5.1,5.1] & [-5.1,5.1] & [-5.1,5.1] \end{bmatrix}$, $M2 = \begin{bmatrix} [-100.1,100.1] & [-100.1,100.1] & [-100.1,100.1] \\ [-10.1,10.1] & [-10.1,10.1] & [-10.1,10.1] \\ [-2.1,2.1] & [-2.1,2.1] & [-2.1,2.1] \end{bmatrix}$ and $M3 = \begin{bmatrix} [-10.1,10.1] & [-10.1,10.1] & [-10.1,10.1] \\ [-20.1,20.1] & [-20.1,20.1] & [-20.1,20.1] \\ [-5.1,5.1] & [-5.1,5.1] & [-5.1,5.1] \end{bmatrix}$. With 686 number of variables and 993 generated constraints, POP(SMT) finds the minimal precision of the inputs and intermediary results for this example in less than 0.3 seconds as it is observed in Figure 6.3b. This time concerns the resolution of the system of constraints and the calls of the Z3 SMT solver done by binary search for different requirements of accuracy.

Hence, the final number of bits of the transformed program compared to 9964 initial bits is considerable as shown in Figure 6.3d. Finally, we notice that our analysis succeeded in turning off almost the double precision variables to a fairly rounded single precision ones for nsb $\leqslant 20$ bits.

6.3 POP(ILP)

6.3.1 Outline

We recall that in the most recent version of POP, the idea is to use no longer the non-optimizing Z3 SMT solver coupled to a binary search. By that means, we reduce the precision tuning problem to an ILP which can be optimally solved in one breath by a classical linear programming solver. In practice, We use the GLPK v4.65 solver [Mak] (GNU

Linear Programming Kit). GLPK[4] is a set of algorithms for solving different problems ranging from linear programming (LP) to integer linear programming (ILP). The outline of POP(ILP) is depicted in Figure 6.4. Concerning scalability, we generate a linear number of constraints and variables in the size of the analyzed program.

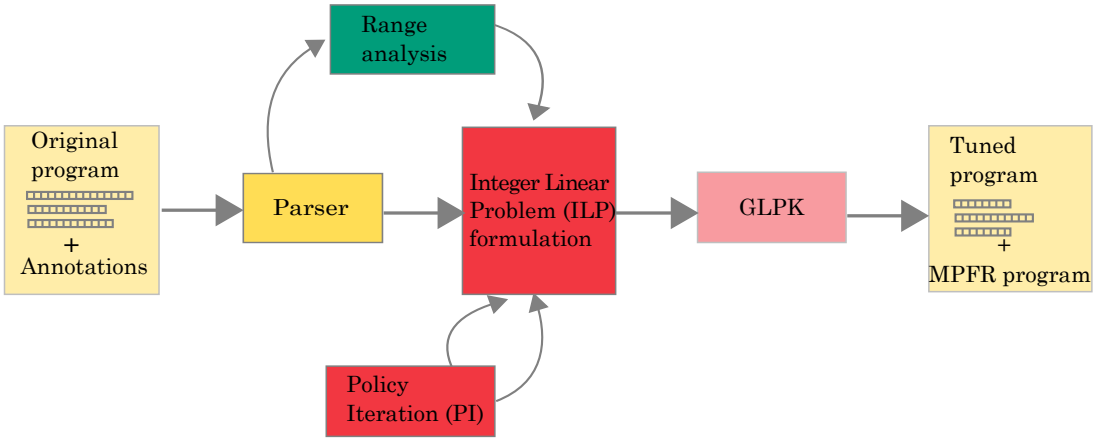


Figure 6.4: POP(ILP) overview.

6.3.2 Benchmarks

As in POP(SMT), we will evaluate the performance of POP(ILP) on several examples. We will use the POP(SMT) benchmarks already presented in Section 6.2.2 in order to compare the results of precision tuning by the two tools. In addition, we experiment POP(ILP) on new benchmarks coming from the FPBench[5] community. FPBench develops standards for describing floating-point benchmarks and for measuring their accuracy [DMP+16]. Here, we shed light on the new benchmarks of POP(ILP).

- **The N-body problem** [Dem21]: models the simulation of a dynamical system describing the orbits of planets in the solar system interacting with each other gravitationally. This application will be highlighted in Chapter 9 Section 9.3.
- **Pendulum** program [ABM21]: models the movement of a simple pendulum without damping. We recall that this program was introduced in Chapter 5.
- **Newton-Raphson method** [Atk91]: is a numerical method used to compute the successive approximations of the zeros of a real-valued function.
- **Odometry** [DMC17]: an example taken from robotics which concerns the computation of the position(x,y) of a two wheeled robot by odometry.
- **PID Controller** [DMC15]: is a widely used algorithm in embedded and critical systems e.g. aeronautic and avionic systems. The main feature of this program is to

[4] https://www.gnu.org/software/glpk/
[5] https://fpbench.org/

keep a physical parameter m at a specific value known as the `setpoint`(C). In other words, it tries to correct a measure by maintaining it at a defined value.

- **Runge Kutta method** [Atk91]: is an effective and widely used method for solving the initial-value problems of differential equations.
- **The trapezoidal rule** [Atk91]: a well known algorithm in numerical analysis which approximates the definite integral $\int_a^b f(x)\,\mathrm{d}x$.

Chapter 9 reports the results of precision tuning of `POP(ILP)` for these benchmarks.

6.4 Summary

In this chapter, we introduced our tool `POP`. We presented its architecture, its inputs including the source programs, its parameters and its outputs. In addition, we illustrated the algorithm followed by `POP` to lower the precision of the numerical programs. Two variants of `POP` were presented. `POP(SMT)` which implements the SMT-based method and `POP(ILP)` which implements the ILP-based method.

We dedicate the rest of this dissertation to present our experimental results. In the next chapter, we present the results of experimenting `POP(SMT)` in a new application domain, IoT. Besides, chapters 8 and 9 are devoted for the evaluation of performance for both `POP(SMT)` and `POP(ILP)` in term of analysis time, quality of solutions and efficiency as well as for comparing the two variants of `POP` and the state-of-the-art tools.

Part III

Internet of Things: A Field of Interest

Chapter 7

Precision Tuning and Internet of Things

***

7.1 Tilt Angle Detection by an Accelerometer 106
7.2 Accelerometer-Based Pedometer Algorithm 108
7.2.1 Step Counting Algorithm for Embedded Applications 109
7.2.2 Experimental Evaluation of the Pedometer Program 111
7.3 Summary 116

Precision tuning already has applications in many domains and, in the present chapter, we show its usefulness for IoT applications. In practice, 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 [KLKS18]. 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. More precisely, IoT devices are powered by an independent power supply like battery and energy harvester, which provide limited energy. Consequently, batteries require changing and replacement due to their short lifetime. Also, the memory of the IoT devices is used to store data and performance tasks, therefore, consistent memory access occurs during the operations of the IoT device. Thus, the energy savings associated with memory access reduce the average power consumption of IoT 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 and that is why we believe it possible and promising to adapt the precision tuning technique

to these kind of applications.

In this chapter, we focus on experimenting POP(SMT) on two representative examples coming from the IoT field and finding a trade-off between precision and energy. Guided by industrial demands [IMN17], we take as example the code of an accelerometer to convert pressure into movement. Secondly, we evaluate POP(SMT) on a significantly more complex example, an accelerometer-based pedometer for embedded applications [MSGK14]. We ran the experiments of the present chapter on an Intel Core i5-4200U 1.6GHz Windows machine with 6 GB RAM.

The remainder of this chapter is organized as follows. Section 7.1 presents the results of tuning the tilt angle detection application by an accelerometer. Section 7.2 evaluates POP(SMT) on a more complex example, an accelerometer-based pedometer for embedded applications. Finally, Section 7.3 concludes.

This work is based on the articles published respectively at the IEEE International Conference on Internet of Things, Embedded Systems and Communications (IINTEC) in 2019 [BM19] and the IEEE International Conference on Internet of Things and Intelligence System (IoTaIS) in 2020 [BM20].

Notation: we recall that the numerical results of this chapter belong to the first implementation of our tool POP already called POP(SMT). Once the ILP-based method has been implemented in POP(ILP), we re-analyzed these examples with this technique. The results will be highlighted in Chapter 9.

7.1 Tilt Angle Detection by an Accelerometer

To study the usefulness of POP(SMT), we choose to experiment our analysis on an example that measures an inclination angle with an accelerometer. An accelerometer is a sensor capable of measuring, in three dimensions, the linear accelerations of an object as well as vibrations [BM19]. In practice, there are accelerometers in many everyday objects use, such as smartphones, cars, sports watches and other devices. For instance, the accelerometer of a phone is able to give you the orientation of the phone but also, as indicated by its name, the acceleration undergone by the phone. Furthermore, an accelerometer usually breaks down into two parts: a mechanical part which is responsible for detecting the accelerations of a mass contained in the device and an electronic part having for mission to interpret this signal. In this context, we have analyzed an application that describes how often accelerometers are used to measure a tilt of an object.

Tilt detection is a simple application of an accelerometer where a change in angular position of the system in any direction is detected and indicated the corresponding angle scaled from microcontroller output and in order to have more accurate measurements of tilt in the x and y planes, we therefore need a 3 axis accelerometer as represented in Figure 7.1.

We aim from the accelerometer experimentation to measure the usefulness of our analysis and how POP(SMT) is capable to optimize the precision of the program variables.

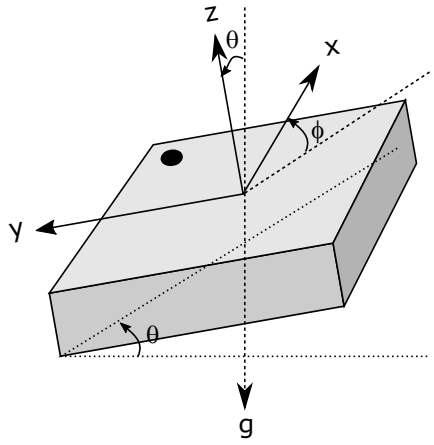


Figure 7.1: Outline of tilt angle measurement by a 3-axis accelerometer.

For this example, POP(SMT) generates 1179 variables and 1767 constraints which are very manageable by the Z3 solver. Indeed, it takes only 0.64 seconds to POP(SMT) to achieve the analysis, including constraints generation and the calls to Z3. Assuming that all the variables of the original program are in double precision, POP(SMT) succeeded in turning off variables into FP16 (half precision) and FP32 (single precision) as shown in the top side of Figure 7.2. For example, for nsb $= 20$ bits, the percentage of variables tuned into FP32 is large compared to the variables tuned in FP16: 93.13% for single precision and only 5.88% for half precision. Also, for an nsb $>$ 22 bits, POP(SMT) manages correctly the precision tuning approach by finding a single and double precision compromise.

As we show in the bottom of Figure 7.2, for nsb $= 30$ bits and nsb $= 32$ bits, the mixed-precision between single and double is obtained. Also, we can say that for an nsb $= 24$ bits there is as much single as double precision variables. Moreover, we notice that for an nsb $= 36$ bits all the variables remain in double precision and thus finding the minimal precision is only possible for accuracies lesser than 36 bits.

The top of Figure 7.3 describes the improvement of the number of bits compared to the original program. In fact, the original program starts with 15105 bits at bit-level, where all the variables are in double precision, and after POP(SMT) analysis we found that the improvement compared to the initial number of bits, ranges from 39% to 84% for an nsb starting from 12 to 36 bits which confirms the efficiency of our analysis.

As the subject of saving memory is challenging to us, in the bottom of Figure 7.3 we measure the memory used in Bytes by the program variables according to the precision requests. Initially, the memory size of the accelerometer program is equivalent to 816 Bytes and we measure in this experiment the size of the memory with each precision inserted. The bottom side of Figure 7.3 depicts that the memory used increases when we increase the accuracy requirements. For example, the program starts with 230 Bytes for an nsb $= 12$ bits and reaches nearly 816 Bytes for an nsb $= 32$ bits.

In addition, an important observation on the behavior of POP(SMT) is that it assigns

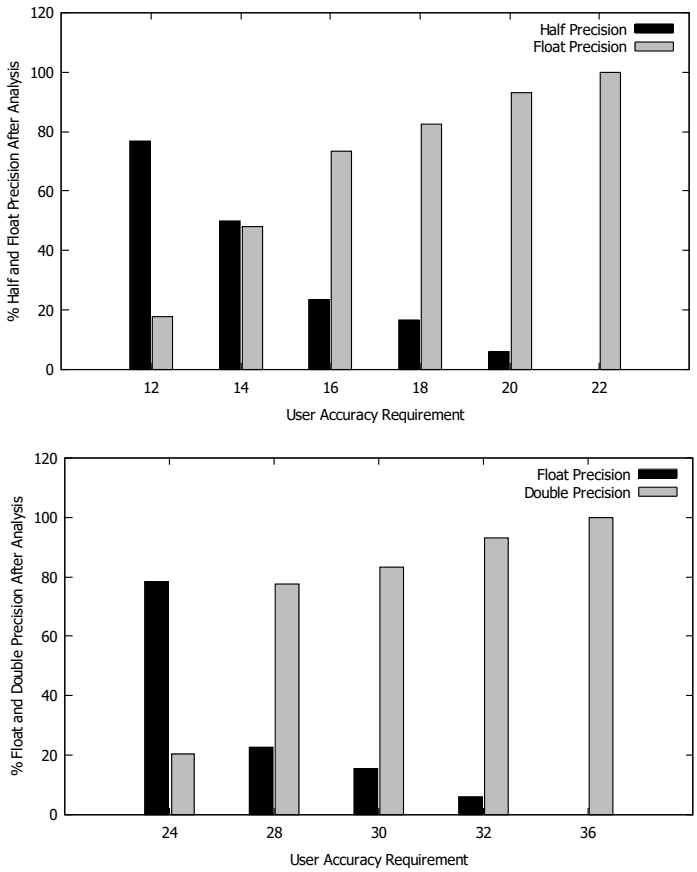


Figure 7.2: Efficiency of POP(SMT) on the accelerometer tilt measure application. Top: the percentage of half and single precision for different nsb assertions after analysis. Bottom: The percentage of float and double variables for nsb $= 24, 28, 30, 32$ and 36 bits.

zeros to the accuracies of the variables which are declared in the programs but which do not intervene in the computations.

Consequently, we present in Listing 7.1 the optimized tilt measure program where in the left hand side we describe the source program with the inferred accuracies after POP(SMT) analysis and the program annotated with the new optimized formats in the right hand side of Listing 7.1.

7.2 Accelerometer-Based Pedometer Algorithm

In the previous section, 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(SMT) 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 pedome-

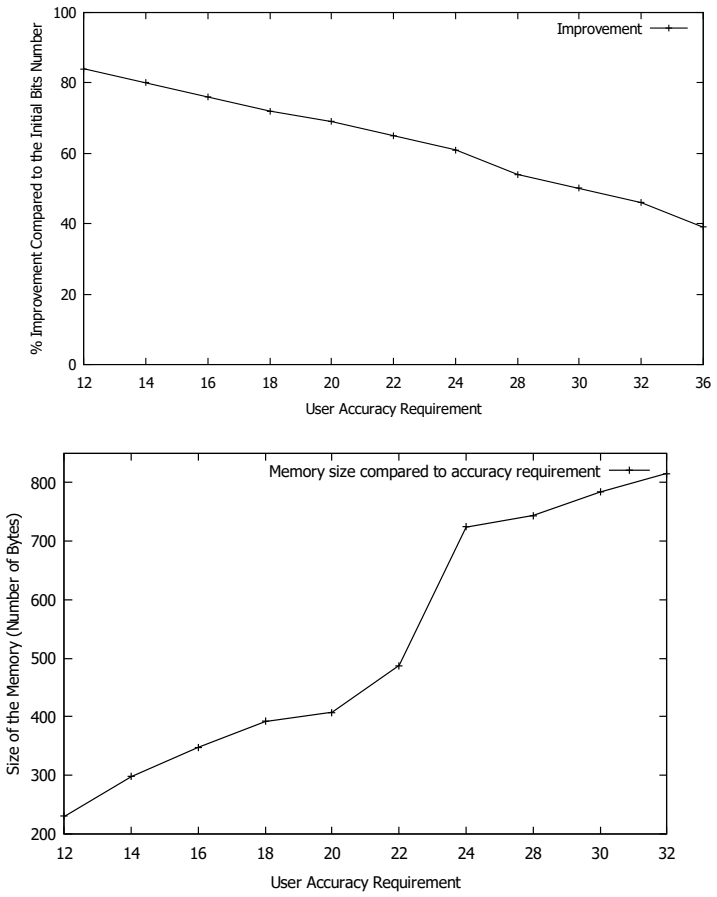


Figure 7.3: Top: Improvement of the number of bits compared to the initial bits number in the original program. Bottom: Memory size (in Bytes) for different user accuracy requirements.

ter algorithm and the experimental results of our method are illustrated in Section 7.2.1.

7.2.1 Step Counting Algorithm for Embedded Applications

A pedometer is a small device that counts the number of footsteps [Aho10]. It is also called a footstep counter. 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 [Zha10]. The present pedometer algorithm uses an accelerometer similar to the one that we have already described in Section 7.1 to count the number of footsteps.

Since the pedometer program is complex in its structures, containing nested loops, arrays and conditions, there are several stages in footstep detection [Lib08] as summarized in Figure 7.4. We outline each of these as following:

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.

```
1 xVal|20|= [-2.0, 2.0]|20| ; yVal|20| =
      [-2.0, 2.0]|20| ;
2 zVal|20| = [-2.0, 2.0]|20|;
3 /* total value of the acceleration*/
4 sos = xVal * xVal + yVal * yVal +
      zVal * zVal ;
5 total = sqrt(sos);
6 /*Angles computation and conversion
      to degrees*/
7 /*x Axis*/
8 u|21| = xVal|20| /|21| total|20|;
9 angleX|22| = u + u * u * u / 6.0 + u *
       u *
10  u * u * u * 3.0 /40.0 * 180.0 /
       3.1416 ;
11 require_nsb(angleX, 24);
12 /*y axis*/
13 v|21| = yVal|20| /|21| total|20|;
14 angleY|22| = v + v * v * v / 6.0 + v *
       v *
15  v * v * v * 3.0 /40.0 * 180.0 /
       3.1416;
16  require_nsb(angleY, 24)
17 /*z axis*/
18 w|21| = zVal|20| /|21| total|20|;
19 angleZ|22| = 1.570796 - (w + w * w * w
       / 6.0
20   + w * w * w * w * w * 3.0 /40.0 *
        180.0 / 3.1416);
21 require_nsb(angleZ,24);
```

```
1 float xVal, yVal, zVal, sos;
2 double angleX, angleY, angleZ;
3 /*init sos, total*/
4             [...]
5 /*init angleX*/
6 angleX = u + u * u * u / 6.0
7  + u * u * u * u * u* 3.0
8  /40.0 * 180.0 / 3.1416;
9             [...]
10 /*init angleY*/
11 angleY = v + v * v * v / 6.0 +
12  v * v * v * v * v * 3.0 /40.0
13 * 180.0 / 3.1416;
14             [...]
15 /*init angleZ*/
16 angleZ = acos(zVal / total )
17 * 180.0 / 3.1416;
18             [...]
```

Listing 7.1: Example of mixed-precision inference. Left: source program with inferred accuracies. Right: Program formats.

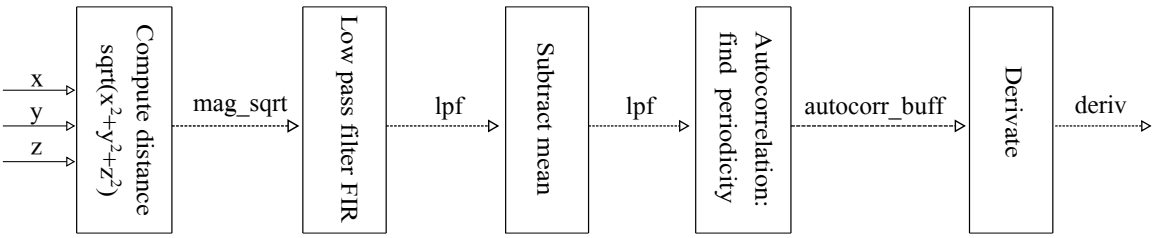


Figure 7.4: Overview of the footstep counter algorithm.

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 [Aho10]. 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.

Autocorrelation to find the signal periodicity The autocorrelation function is the core of the step counter algorithm [MSGK14]. 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.

Footstep detection using derivative In this step, the algorithm computes 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.

`POP(SMT)` 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 Section 7.2.2, we make vary the size of the window, as well as the accuracy requirements, and see its impact on the performance of `POP(SMT)`. Next, mag_sqrt holds the magnitude of data x, y and z. The annotation $require_nsb(mag_sqrt, 23)$ indicates to `POP(SMT)` 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 ($\texttt{nsb} = 23$ bits). 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(SMT)` analysis. After the explanation of the different steps of the footstep counter algorithm given to `POP(SMT)`, we measure in the next section the usefulness of our precision tuning analysis.

7.2.2 Experimental Evaluation of the Pedometer Program

The main results of running `POP(SMT)` on the footstep counter algorithm for the data set of [MSGK14] are presented in figures 7.5, 7.6, 7.7 and 7.8 which correspond respectively to:

1. The measurement of performance of `POP(SMT)` 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 control points, 53 being the size of mantissas in double precision (FP64).

2. The analysis time of the program in seconds.
3. The optimisation in term of number of bits for different intervals around the x, y and z average values.
4. The mixed-precision configurations obtained after `POP(SMT)` analysis in terms of number of variables and operations that we may tune into IEEE754 double (FP64), simple (FP32), half (FP16) and mini-float precision (FP8).

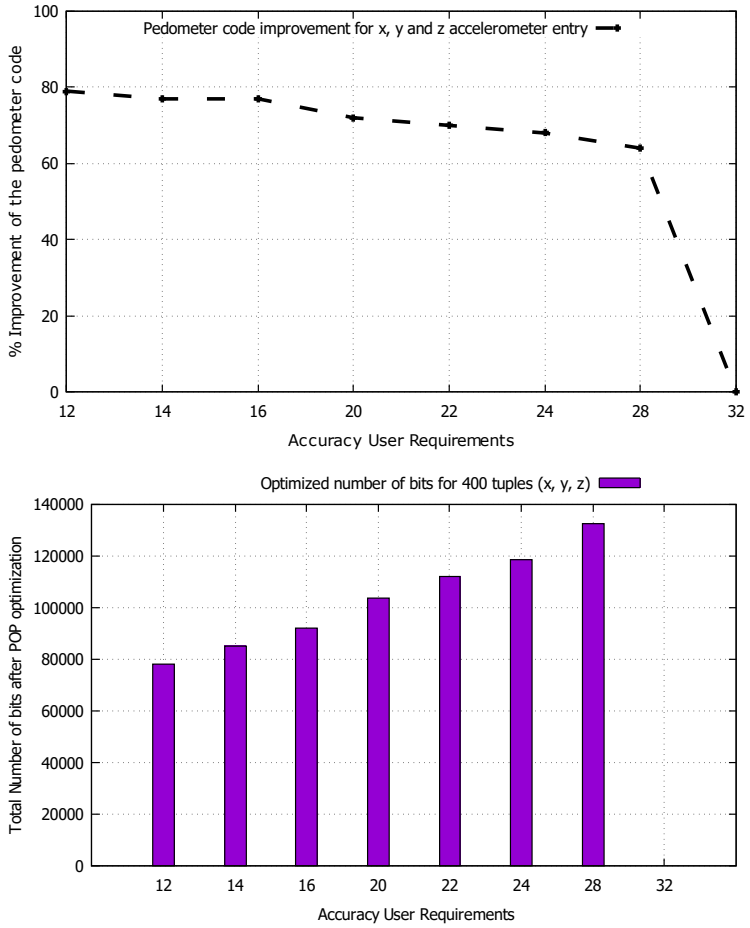


Figure 7.5: Top: Number of bits optimized for different user accuracy requirements. Bottom: Improvement compared to the initial total number of bits of the original program.

Of the eight accuracy requirements given to `POP(SMT)` (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 in the top of Figure 7.5, the improvement varies between 64% for `nsb` $=$ 28 bits to 79% for `nsb` $=$ 12 bits. In addition, let us note that `POP(SMT)` remains all variables in double precision for an `nsb` $=$ 32 bits and so no mixed-precision tuning is achieved beyond this accuracy for this example. Likewise, these results are confirmed in the bottom of Figure 7.5. Initially starting with 364905 bits at bit-level in the

original pedometer program, the total number of bits is optimized for more than 50% for a nsb $=$ 28 bits giving a new total of 132797 bits at bit-level.

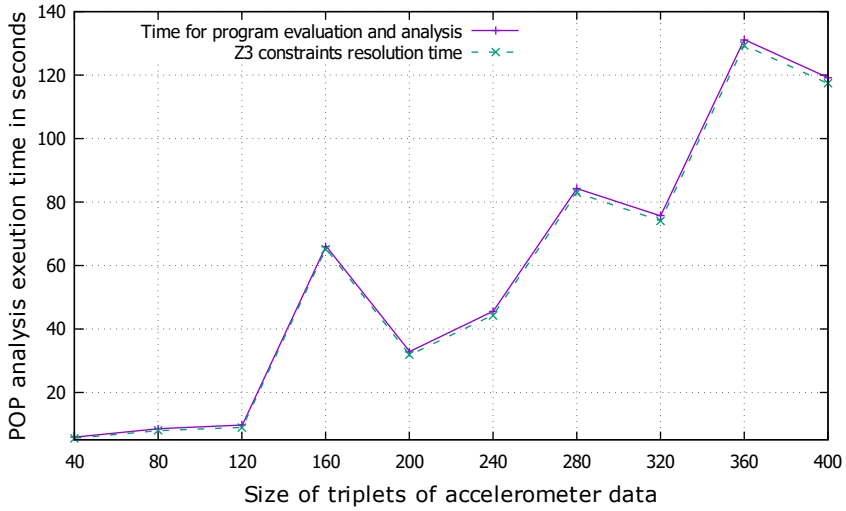


Figure 7.6: Comparing the execution time of POP(SMT) analysis and the time spent by Z3 to find a solution to our system of constraints.

Clearly, the various measurements of POP(SMT) execution time illustrated in Figure 7.6 are reasonable and only take a few minutes for the whole pedometer code with a window of 400 data sets. Accordingly to Figure 7.6, we observe that the execution time increases when modifying the size of the accelerometer data set (or in other words, the size of the window used in the autocorrelation stage). Indeed, we observe that almost all of POP(SMT) 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(SMT) (same for POP(ILP)) being a static analysis tool admitting sets for its inputs, Figure 7.7 shows how POP(SMT) 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 histograms of Figure 7.7 show that POP(SMT) succeeded in optimizing the initial number of bits for the original pedometer code, starting with 238977 bits at bit-level, for nsb $>$ 36 bits while for the case of the scalar values of the bottom of Figure 7.5 there were no precision tuning beyond an accuracy of 32 bits.

Assuming that in the original pedometer code, all the variables are in double precision, POP(SMT) succeeded in turning off variables into half and float precision as shown in Figure 7.8 and other variables remain in double precision for some accuracy assertions. The various diagrams of Figure 7.8 show the percentage of variables, after POP(SMT) analysis, in FP8, FP16, FP32 and FP64. To mention a few, 23% of variables are tuned to FP32 while the majority remains in FP64 for nsb $=$ 16 bits. Also, 50% of the program variables are turned off in single precision for an nsb $=$ 28 bits although 48% still in double precision

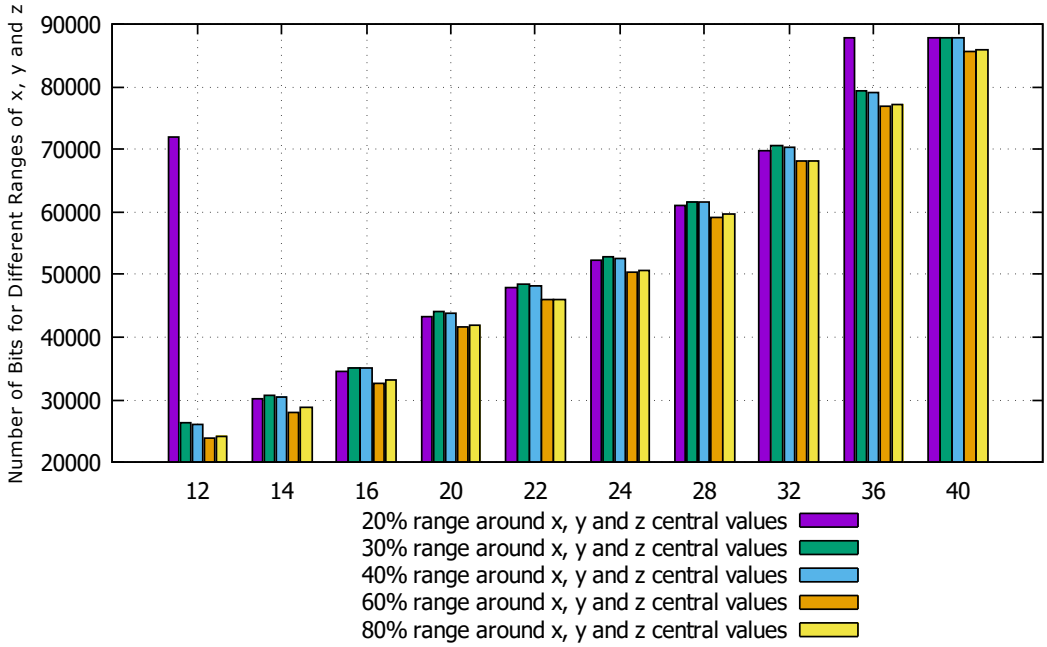


Figure 7.7: `POP(SMT)` optimization of the total number of bits for different magnitude of ranges of x, y and z.

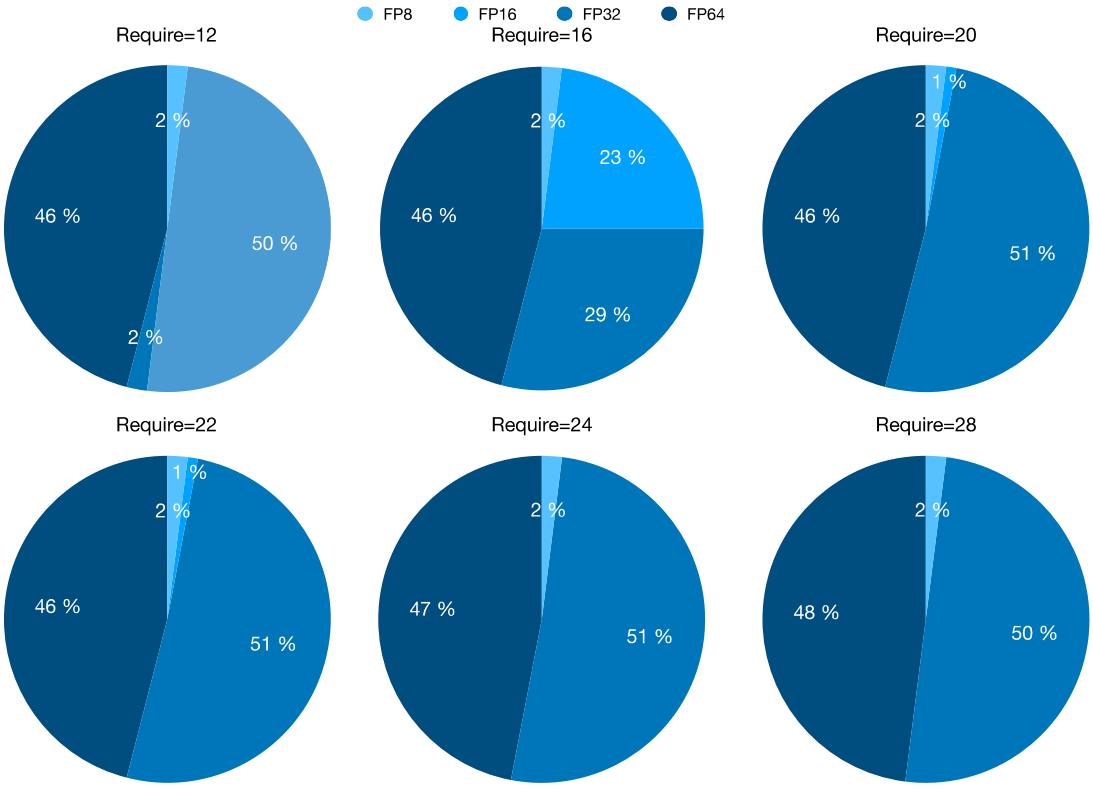


Figure 7.8: The percentage of program variables in FP8, FP16, FP32 and FP64 after `POP(SMT)` analysis.

and no variables tuned to FP8 mini-float precision. We present in Listing 7.2 the new mixed-precision formats of three steps of the footstep detection algorithm already detailed in Figure 7.4 which are: the low pass filter, remove of the mean and the autocorrelation function after POP(SMT) analysis. The left hand side of Listing 7.2 shows the original code for these functions where all the variables are in double precision and on the right hand side of Listing 7.2, we annotate the code with the new optimized formats for a given accuracy of nsb $=$ 20 bits for each function output.

```
1  /*lowPassfilt function*/
2  aux0 = 0.0;
3  for(n = 0; n < numTuples; n++) {
4   tempLpf = 0.0;
5    for (i = 0; i < lpfFiltLen; i++){
6     if(n - i >= 0.0){
7       aux0 = lpfCoeffs[i] * magSqrt[n
            - i];
8       tempLpf = tempLpf + aux0;};
9   };
10 };
11  require_nsb(tempLpf,20);
12
13 /* remove_mean function*/
14 sum = 0.0;
15 for(i = 0, i < numTuples, i++){
16  sum = sum + lpf[i];};
17 sum =  sum / numTuples;
18 for (i = 0; i < numTuples; i++){
19  lpf[i] = lpf[i] - sum ; };
20  require_nsb(lpf,20);
21
22 /*Autocorrelation function*/
23 tempAc = 0.0;
24 for(lag =0, lag < numAutoCorrLags,
       lag++){
25  aux1 =0.0;
26  for(i=0; i<numTuples-lag; i++){
27   aux1 = lpf[i] * lpf[i+lag];
28   tempAc = tempAc +  aux1;};
29 autoCorrBuff[lag]= tempAc;
30 };
31  require_nsb(autoCorrBuff,20);
```

```
1  /*lowPassfilt function*/
2  aux0|20| = 0.0|20|;
3  for(n = 0; n < numTuples; n++) {
4   tempLpf|20| = 0.0|20|;
5    for (i = 0; i < lpfFiltLen; i++){
6     if(n - i >= 0.0){
7       aux0|20| = lpfCoeffs[i]|20| *|20|
8        magSqrt[n - i]|35|;
9       tempLpf|20| = tempLpf|20| +|20| aux0
            |20|;};
10  };
11 };
12  require_nsb(tempLpf,20)|20|;
13
14 /* remove_mean function*/
15 sum|23| = 0.0|23|;
16 for(i = 0; i < numTuples; i++){
17  sum|23| = sum|23| +|23| lpf[i]|23|;};
18 sum|23| = sum|23| /|23| numTuples|13|;
19 for (i = 0; i < numTuples; i++){
20  lpf[i]|23| = lpf[i]|23| -|23| sum|23| ;};
21  require_nsb(lpf,20)|20|;
22
23 /*Autocorrelation function*/
24 for(lag =0; lag < numAutoCorrLags;
       lag++){
25  tempAc|24| = 0.0|24|;
26  aux1|24| = 0.0|24|;
27 for(i=0; i<numTuples-lag; i++){
28    aux1|24| = lpf[i]|23| *|24| lpf[i+lag]
          |23|;
29    tempAc|24| = tempAc|24| +|24| aux1|24|;
30   };
31 autoCorrBuff[lag]|24| = tempAc|24|;
32 };
33  require_nsb(autoCorrBuff,20)|20|;
```

Listing 7.2: Left: Low pass filter FIR, remove mean and autocorrelation functions of the original program. Right: source program with inferred accuracies.

7.3 Summary

In this chapter we experimented POP(SMT), our static tuning assistant for mixed-precision, in a new domain of application IoT because the type of problem of energy consumption and memory are widespread in this area. Firstly, we have experimented POP(SMT) on the example of an accelerometer which can be used to measure the static angle of tilt or inclination. The results discussed show that our tool succeeded in computing the accuracy needed for each variable and intermediary results. We believe also that the new formats obtained by POP(SMT) can be very helpful at the system design level, to dimension the hardware. More precisely, these results allow the architect to choose which compromise he wants between accuracy and memory consumption, which is a key point for the performance of IoT devices in particular.

Secondly, We have evaluated POP(SMT) on a pedometer program that implements a step counting algorithm for embedded applications. This example is significantly more complex than the accelerometer program which can be used as input of the present pedometer. Our results show that the static approach embodied in POP(SMT) manages correctly the mixed precision tuning which is confirmed by the different formats of program variables obtained after the analysis. It is also noticeable that POP(SMT) execution time remains 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.

Finally, the reader may rightfully remark that the results in the present chapter were obtained before the implementation of the ILP-based method in POP(ILP). In Chapter 8, we will clearly show that for the accelerometer and pedometer programs, POP(ILP) do well and saves more bits and analysis time than POP(SMT).

Part IV

Evaluation of POP Performance for Tuning Numerical Programs

Chapter 8

Evaluation of POP(SMT) Performance

***

8.1 Different Cost Functions for the Z3 SMT Solver . . . 120
8.1.1 Assigned Variables vs. All Control Points Cost Functions . . . 121
8.1.2 Numerical Results . . . 121
8.2 MPFR Code Generation . . . 123
8.3 Comparison between POP(SMT) and Precimonious . . . 124
8.3.1 Experimental Setup . . . 124
8.3.2 POP(SMT) vs. Precimonious Comparison Results . . . 126
8.4 Summary . . . 129

We have defined in Chapter 4 the forward and backward transfer functions. Next, these constraints have been expressed as a set of propositional formulae on relations between integer variables only, easily checked by Z3 [BMA19]. In the previous chapter, we have evaluated POP(SMT) on two representative examples coming from the IoT domain. In this Chapter, we propose a detailed evaluation of POP(SMT) performance is several manners:

- First, we improve the efficiency of POP(SMT) by experimenting with several cost functions in order to optimize the solutions returned by the Z3 SMT solver. At the same time, we compare the performance of our tool for each of these functions on different programs coming from scientific computing, signal processing and the IoT domain.
- Second, we measure for each tuned program the error between the exact results given by an execution in multiple precision and the results of the programs optimized

by POP(SMT). To do so, we have used the Multiple-Precision Binary Floating-Point Library With Correct Rounding MPFR [FHL+07]. Finally, the measured error is compared to the user required accuracy already defined by the quantity nsb.

- Third, we provide a detailed comparison of POP(SMT) against the prior state-of-the-art tool Precimonious [RNN+13] in terms of analysis time, speed and the quality of the mixed-precision tuning. We remind the reader that a detailed presentation of the Precimonious tool was discussed in Chapter 3 Section 3.3.2. Even though both tools use different techniques, we have adjusted the comparison criteria in order to make a closer comparison of the real behavior of these tools (more details presented in Section 8.3 of the present Chapter).

For the experimental setup, we ran these experiments on an Intel Core i5-8350U CPU cadenced at 1.7GHz on a Linux machine with 8 GB RAM. Concerning Precimonious, we used the version published at [RNN+13][1].

The remainder of this chapter is organized as follows. We start with experimenting POP(SMT) with different cost functions in Section 8.1. Section 8.2 reports and discusses the generation of MPFR codes by our tool. Section 8.3 presents a complete description of the comparison between POP(SMT) and the prior state-of-the-art Precimonious, before concluding in Section 8.4

The content of this chapter is mostly based on the article published at the 4^{th} IEEE International Conference on Information and Computer Technologies (ICICT) in 2021 [BM21a].

8.1 Different Cost Functions for the Z3 SMT Solver

In this experiment, we aim at evaluating two kinds of cost functions and compare the mixed-precision obtained in the tuned programs for each of these functions. Recall that POP(SMT) generates a set of constraints made of propositional logic formulas and affine expressions among integers already presented in [BMA19] and calls the Z3 SMT solver in order to obtain a solution.

However, the solutions returned are not optimal due to the fact that Z3 is a SMT solver and not an optimizer. To cover this limitation, we add to our global system of constraints, as we have shown in Chapter 6, an additional constraint related to a cost function ϕ. The aim of the first cost function $\phi(c)$ (defined in Equation (6.1)) is to compute the sum of the accuracies $accd(x^{\ell})$ of all the variables plus the accuracies $accd(\ell)$ at each control point ℓ of the arithmetic expressions. Then we ask Z3 to find a solution of a given weight. This operation is done repeatedly in a binary search.

In the next section, we define a second cost function and we compare the quality of the mixed-precision data formats obtained in the tuned programs by both functions.

[1] https://github.com/HGuo15/vagrant_precimonious

8.1.1 Assigned Variables vs. All Control Points Cost Functions

The purpose of the second cost function given by $\phi'(c)$ in Equation (8.1), is to optimize $\phi(c)$ by only considering the sum of accuracies for the variables assigned in the program and to no longer count accuracies for the different control points as indicated in Equation (6.1). Equation (8.1) hereafter highlights the optimized cost function $\phi'(c)$.

Proposition 8.1 (Cost Function for Assigned Variables)**.** Let Id and Lab denote respectively the sets of identifiers and labels of the program. Let $accd(x^\ell)$ be a variable of the constraint system corresponding to the accuracy of a variable $x \in Id$ at a control point $\ell \in Lab$ and let $\mathsf{nsb}(\ell)$ be the accuracy of the operation done at control point ℓ. Then $\phi'(c)$ is defined by

$$\phi'(c) = \sum_{x \in Id, \ell \in Lab} \mathsf{nsb}(x^\ell) \ . \tag{8.1}$$

■

After adding the additional cost function constraint that we are interested to use in the global system of constraints, POP(SMT) searches the smallest integer $P = \phi(c)$ or $P = \phi'(c)$ such that our system of constraints admits a solution. Consequently, we start the binary search with $P \in [0{,}112 \times n]$ where all the values are in long double precision (binary128) and where n is the number of terms in Equation (6.1) or Equation (8.1) (depending on the function we add in our system of constraints). When a solution is found for a given value of P, a new iteration of the binary search is run with a smaller value of P. When the solver fails for some P, a new iteration of the binary search is run with a larger P and we continue this process until convergence. We recall that the cost functions are simplified when we deal with arrays.

8.1.2 Numerical Results

Figure 8.1 shows the results of mixed-precision tuning obtained by POP(SMT) on several programs. The **low pass filter** program and the **derivative** function are parts of the pedometer program that counts the number of footsteps whose algorithm contains several steps [BM20] highlighted in Chapter 7. We recall that these last two programs are coming from the IoT field [MSGK14]. The **arclength** and the **simpson** programs were already presented in Chapter 6. The results are perceived by experimenting two cost functions to our global system of constraints in order to optimize the solutions returned by the Z3 solver.

Of the six accuracy requirements given to POP(SMT) for the four input programs, from a $\mathsf{nsb} = 12$ bits to $\mathsf{nsb} = 38$ bits, POP(SMT) succeeded in turning off variables into FP8 mini-float precision, FP16 half precision, FP32 single precision, FP64 double precision and FP128 long double precision as shown in the top and the bottom of Figure 8.1 for the two cost functions $\phi(c)$ and $\phi'(c)$ respectively.

Besides, the top of Figure 8.1 shows that for the **low pass filter** program, 16% of variables are tuned to FP8 mini-float precision, 37% are tuned to FP16 half precision and nearly 50% of variables are turned off in FP64 double precision for $\mathsf{nsb} = 12$ bits. Although, for the

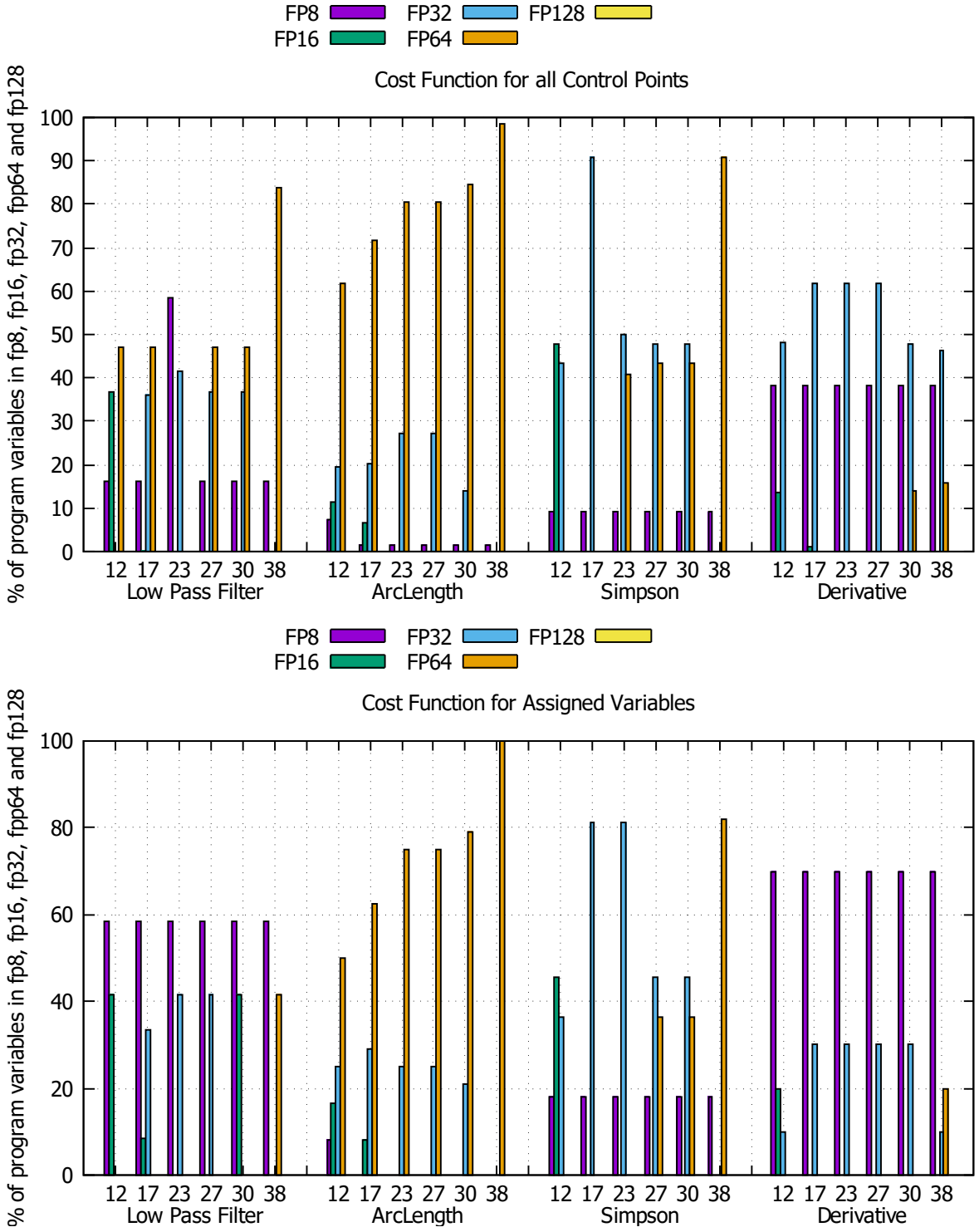


Figure 8.1: Mixed-precision tuning results in FP8, FP16, FP32, FP64 and FP128 for the all control points and variables accuracies cost function (top) and for the optimized cost function considering only the variables assigned (bottom).

same program, the majority of variables are transformed to FP64 for nsb = 38 bits. For the **arclength** program and for the different user accuracies, a large amount of variables are tuned to FP64 double precision starting with 62% for nsb = 12 bits until 98% for nsb = 38 bits. For the **simpson** program, we observe improvements but in less proportions than for the other examples. For the program implementing the **derivative** function of the pedometer code, we observe that the largest part of variables are transformed to FP32 single precision and FP8 mini-float precision compared with the few amount of variables tuned to double precision especially for accuracies of 30 and 38 bits.

The main results of running POP(SMT) with the new cost function ϕ' are illustrated in the bottom side of Figure 8.1. By comparing the results with the old cost function, we perceive the difference of the mixed-precision tuning results returned by POP(SMT). For the **low pass filter** program, we have more variables turned into FP8 mini-float precision with nearly 59% for almost every user accuracy requirements, compared to the number of variables in FP64 double precision obtained with the cost function illustrated in the top of Figure 8.1.

Furthermore, the majority of variables in program **arclength** are transformed into FP64 double precision, going up to 100% for nsb = 38 bits. Finally, for the derivative program, the percentage of variables in FP8 mini-float precision is greater than in FP32 single precision reaching a percentage of 70% for all user accuracies.

We bring the reader's attention to the fact that the second cost function $\phi'(c)$, which is a strict relaxation of $\phi(c)$, corresponds to what Precimonious actually optimizes. It is then fairer to use it in our comparisons in Section 8.3. In addition, all the original programs showed above are in FP128 long-double precision before POP(SMT) analysis for reasons of comparison with the Precimonious tool.

8.2 MPFR Code Generation

In this experiment, we seek to generate a MPFR code to create a program that gives an exact result by assuming that the original program is computed with a precision of 300 bits. After, the same code will be generated with the optimized precision returned by POP(SMT). The goal of this experiment is to measure the difference between the two programs and to plot the curve of the difference in function of the theoretical error which is the worst accepted error required by the user, for example this error is equal to 2^{-22} for nsb = 23 bits.

The experiment shown in Figure 8.2 examines the difference between the exact results computed with MPFR and the results returned by POP(SMT). Following this, we compare the difference with the worst error required by the user for different nsb values: 12, 17, 23, 27, 30 and 38 bits. Consequently, each program in this experiment corresponds to two curves and we can see in Figure 8.2 that the actual error measured is always less than the theoretical user error defined as $\beta^{-\mathsf{nsb}+1}$ where $\beta = 2$.

The curves of Figure 8.2 validate that POP(SMT) satisfies the user defined error constraints for the input programs: **arclength**, **simpson** and **low pass filter**. Also, we can see

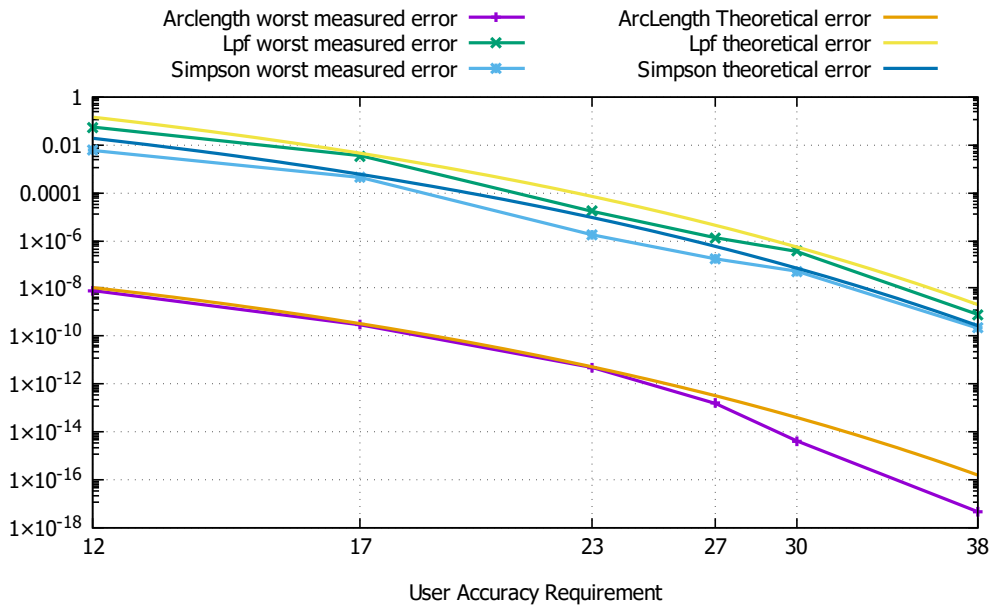


Figure 8.2: Measured error between the exact results with multiple precision and the results obtained with POP(SMT) for the **simpson**, **arclength** and **low pass filter** programs.

that the actual error for each program is always less than the theoretical error given by the user. For nsb = 17 bits and nsb = 23 bits, the curves of the **arclength** program can intersect but it always remain below the worst error tolerated by the user which are 2^{-16} and 2^{-22} respectively for these requirements. As an important observation, we remark that the measured errors are very close to the theoretical errors which is considered as a desirable property. In other cases, if the measured error were far smaller than the theoretical error this would probably mean that we use too many bits and consequently that the our method of precision tuning would be suboptimal.

8.3 Comparison between POP(SMT) and Precimonious

8.3.1 Experimental Setup

The main goal of this experimental evaluation is to compare our tool versus the dynamic-search based tool Precimonious [RNN+13]. Therefore, we evaluate POP(SMT) on two numerical programs used as benchmarks for precision tuning in Precimonious: the **simpson** and **arclength** programs. In addition, we evaluate Precimonious on three programs used as benchmarks for POP(SMT) which are the **rotation** program already highlighted in Chapter 6 Section 6.2.2, the **accelerometer** program and the **pedometer** program presented in Chapter 7.

The number of lines of code (LOCs) for each of the five programs differs from one tool to another because of the different instructions linked to the error constraints that

each tool adds. The differences between POP(SMT) and Precimonious are highlighted in Table 8.1. Since the tools implement two different techniques, we have adjusted some criteria including the number of variables that we optimize so that the comparison is as close as possible to the real behaviors of the two tools. This is done as follows:

Property	POP(SMT)	Precimonious
Kind of analysis	Forward and backward static analysis	Dynamic analysis by delta-debugging search
Output	Optimized formats given by Z3 (number of bits)	Type configurations rely on inputs tested **only**
Mixed-precision	FP8, FP16, FP32, FP64, FPxx	FP32 and FP64
Accuracy requirement	number of significant bits of the result	Error threshold (10^{-4}, 10^{-6}, 10^{-8}, ...)
Accepted language	Arrays, expressions, conditions, loops, ...	C program input

Table 8.1: Differences between POP(SMT) and Precimonious.

- POP(SMT) optimizes more variables than Precimonious. Consequently, for the sake of comparison, in the following we only consider the variables optimized by Precimonious to estimate the quality of the optimization. Let us note that by doing this adjustment, we disadvantage POP(SMT).

- For POP(SMT), the initial precision of the input programs is a parameter. This parameter was set to double precision in all the previous experiments in which we do not need to compare with other tools. To fit with Precimonious features in the comparisons, POP(SMT) initial precision is set to long double precision.

- Precimonious creates a search space for all variables which precision needs to be tuned [RNN+13] in FP32, FP64 or FP128 precision. According to POP(SMT), we take into account the variables that arise in the computation of the resulted variable on which the user requires a precision constraint and tunes their precision into FP8 and FP16 alongside the FP32, FP64 and FP128 precision. We do know that the FP8 minifloat precision is rarely used but our tool is able to find this format without additional cost nor to increase the complexity which depends on the number of formats. In contrast, Precimonious is not able to have this format without additional cost.

- In Precimonious, the user accuracy requirement is formulated as an error threshold. In practice, the error threshold represents the number of accuracy digits. For instance, the result is required to be correct up to 10 digits for an error threshold of 10^{-10}. Let us note that the error thresholds are expressed in base 2 in POP(SMT) and POP(ILP) and in base 10 in Precimonious.

  In this section, for the relevance of comparisons, all the thresholds are expressed in

base 10. In practice, POP(SMT) will use the base 2 threshold immediately lower than the required base 10 threshold.

In Table 8.2, we show the number of variables optimized for each program by both tools, while in tables 8.3 and 8.4 we present the number of variables tuned to FP8, FP16, FP32, FP64 and FP128 for error thresholds of 10^{-4}, 10^{-6}, 10^{-8} and 10^{-10}.

Program	POP(SMT)	Precimonious
arclength	25	9
simpson	10	10
rotation	27	27
accelerometer	19	0
pedometer	31	10

Table 8.2: Number of optimized variables by both tools for each program.

8.3.2 POP(SMT) vs. Precimonious Comparison Results

Table 8.3 and Table 8.4 show the results of the mixed-precision tuning for the five programs by both tools and for different error thresholds. We measure the running analysis time in seconds taken by the tools to find the new precision inferred in the optimized programs. In addition, we estimate the time by Z3 in POP(SMT) to solve the constraints. After the tools analysis, we compute the number of variables tuned into FP8, FP16, FP32, FP64 and those remained in FP128 and we present the total number of bits in the optimized programs. We denote by "S" the number of function prototypes which can be tuned to a lower precision. The dash symbol "-" denotes that a tool does not find any solution (configuration of types) that satisfies the user accuracy constraint. We show in bold the number of variables turned to FP8 and FP16 by POP(SMT) where Precimonious is only capable to tune the variables precision to FP32 and FP64 only.

For the running time measured for both tools, we can observe in Table 8.3 that the analysis are running faster with our tool for the different benchmarks and for error thresholds of 10^{-6} and 10^{-4}. However, POP(SMT) takes longer time to find the optimized precision for the pedometer program compared with Precimonious. By looking at the Z3 time spent to resolve the constraints, we can deduce that our tool consumes almost the entire time of analysis making calls to the Z3 solver. Globally, it remains fast and does not exceed a few minutes with only 4 minutes are spent to solve the constraints of the pedometer code for an error threshold of 10^{-6}.

Compared to Precimonious, POP(SMT) succeeds to tune two more programs. The reader may have remarked that by entering the **accelerometer** program as input to Precimonious, no valid configuration was found for this example with the different error thresholds inserted. In addition, Precimonious failed to tune the pedometer program for any error

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

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

Table 8.3: Precision tuning results of POP(SMT) and Precimonious for error thresholds of 10^{-6} and 10^{-4}.

Program	Tool	Threshold 10^{-8}								
		Running Time(s)	Z3 Time(s)	#Bits Optimized	#FP8	#FP16	#FP32	#FP64	#FP128	S
arclength	POP(SMT)	2.626	2.617	576	0	0	2	6	1	0
	Precimonious	145.8	-	448	0	0	2	4	1	2
simpson	POP(SMT)	0.594	0.541	272	0	**2**	0	8	0	1
	Precimonious	207.558	-	384	0	0	4	2	1	3
rotation	POP(SMT)	0.913	0.901	912	**2**	**4**	16	5	0	0
	Precimonious	10.697	-	992	0	0	25	1	1	0
accelermeter	POP(SMT)	2.917	2.897	608	0	0	19	0	0	0
	Precimonious	-	-	-	-	-	-	-	-	-
pedometer	POP(SMT)	102.885	102.751	352	0	0	7	2	0	1
	Precimonious	-	-	-	-	-	-	-	-	-
Program	Tool	Threshold 10^{-10}								
		Running Time(s)	Z3 Time(s)	#Bits Optimized	#FP8	#FP16	#FP32	#FP64	#FP128	S
arclength	POP(SMT)	2.442	2.435	608	0	0	1	7	1	0
	Precimonious	215.011	-	448	0	0	2	4	1	2
simpson	POP(SMT)	0.501	0.487	528	**2**	0	0	8	0	1
	Precimonius	200.388	-	384	0	0	4	2	1	3
rotation	POP(SMT)	0.927	0.923	1184	0	**4**	11	12	0	0
	Precimonious	7.409	-	992	0	0	25	1	1	0
accelermeter	POP(SMT)	2.761	2.735	992	0	0	7	12	0	0
	Precimonious	-	-	-	-	-	-	-	-	-
pedometer	POP(SMT)	177.348	177.161	576	0	0	0	9	0	1
	Precimonious	-	-	-	-	-	-	-	-	-

Table 8.4: Precision tuning results of POP(SMT) and Precimonious for error thresholds of 10^{-10} and 10^{-8}.

threshold lesser than 10^{-6}. Moreover, Table 8.3 lists the final optimized precision obtained in the form of FP8, FP16, FP32, FP64 and FP128 and counts the number of bits in "#Bits Optimized" of the tuned programs. Recall that "S" is the number of function calls to switch to lower precision. We took the same definition used in [RNN+13]. For example, a function whose prototype is tuned from `double -> double` to `float -> float` is counted as one switch in "S".

Moreover, we can observe from tables 8.3 and 8.4 that `POP(SMT)` is capable to optimize variables into FP8 and FP16 (see bold numbers under `POP(SMT)` "#FP8" and "#FP16") in addition to FP32, FP64 and FP128 which are the only three configurations inputs associated to each variable of the search space of `Precimonious`. By way of illustration, 13 variables of a total of 27 are tuned to FP8, 9 variables are optimized to FP16 and the rest are tuned to FP32 for the **rotation** program. We precise that the parameter "#Bits Optimized" is obtained by the sum of multiplying each variable by its precision. We remark that `POP(SMT)` succeeded in tuning 4 of 5 programs with less number of bits optimized than `Precimonious`, except for the **arclength** program.

At the same time, `Precimonious` do well for an error threshold of 10^{-10} for the **arclength**, **simpsons** and **rotation** programs as depicted in Table 8.4. We obtain in this experiment that the tuned **arclength** program by `Precimonious` uses less bits than our tool for the four given error thresholds.

8.4 Summary

In this chapter, a detailed evaluation of the efficiency of `POP(SMT)` has been discussed. First, we have tested two different cost functions to `POP(SMT)` global system of constraints in order to optimize the solutions returned by `Z3` solver. With this experiment, we have shown that this aspect can influence the total number of bits optimized in the tuned programs when using a function instead of the other. Second, we have measured the error between the exact results given by an execution in multiple precision and the results of optimized programs by `POP(SMT)` and we found that the measured error curve is always below the theoretical error required by the user.

As we consider that comparing our work to the existing state-of-the-art techniques is a tremendous challenge to examine, we have evaluated the performance of `POP(SMT)` by presenting a comparison against the `Precimonious` tool in terms of analysis time, speed and the quality of the solutions returned. We have demonstrated that the results achieved by `POP(SMT)` are promising. Hence, `POP(SMT)` was faster in the analysis time for the majority of the programs tested. In addition, we deduced that the `POP(SMT)` tool returned better mixed-precision results for different user accuracy requirements where the variables are tuned into FP8, FP16, FP32, FP64 and FP128 precision.

Chapter 9

Fast and Efficient Bit-Level Precision Tuning with POP(ILP)

***

9.1 Numerical Results for the ILP-based Method 132
9.2 POP(ILP) vs. the Prior State-of-the-Art Techniques 133
9.3 Precision Tuning of the N-Body Problem 136
9.3.1 Experimental Study 138
9.3.2 Distance between the Exact and the Computed Positions of the Bodies 140
9.4 Summary 143

After taking a closer look on how the precision tuning problem can be formulated as an ILP problem in Chapter 5, we evaluate in the present chapter the performance of POP(ILP) on several numerical programs coming from mathematical libraries and other application domains such as IoT and physics. Also, we present a full comparison between the behaviour of POP(ILP) against the first version POP(SMT) and the state-of-the-art Precimonious [RNN+13] and we show that our results encompass the results of state-of-the-art tools. Second, we demonstrate the efficiency of POP(ILP) 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(ILP) in term of mixed-precision tuning of the N-body example are discussed in this chapter. For the experimental setup of this chapter, we ran these experiments on an Intel Core i5-8350U CPU cadenced at 1.7GHz on a Linux machine with 8 GB RAM.

The remainder of this chapter is organized as follows. Section 9.1 shows that POP(ILP)

exhibits very good results when tuning a new variety of benchmarks. Section 9.2 reports the results of the comparison between `POP(ILP)`, `POP(SMT)` and `Precimonious`. We end up in Section 9.3 by explaining the precision tuning of the N-body problem before concluding in Section 9.4.

This work is a part of contribution that was published at the 28^{th} Static Analysis Symposium (SAS) in 2021 [ABM21] and the article [BM21b] published and presented at the 21^{st} International Conference on Computational Science and Applications (ICCSA) in 2021.

9.1 Numerical Results for the ILP-based Method

We evaluate in this section the performance of `POP(ILP)` on the benchmarks of `POP(SMT)` already presented in Chapter 6 Section 6.2.2 and on new benchmarks such as the **N-body problem**, the **pendulum** program, the **Newton-Raphson** method, the **odometry** program, the **PID Controller** program, the **Runge Kutta** method and the **trapezoidal rule** program. These programs were defined in Chapter 6 Section 6.3.2.

The experiments shown in Table 9.1 present the tuning results produced by `POP(ILP)` for each error threshold 10^{-4}, 10^{-6}, 10^{-8} and 10^{-10}. This is for compatibility with `Precimonious` which uses decimal thresholds. Technically, we translate these error thresholds into `nsb`. In Table 9.1, we represent by "TH" the error threshold given by the user. "BL" is the percentage of optimization at bit-level. "IEEE" denotes the percentage of optimized variables in IEEE754 formats (e.g. FP16, FP32 etc.) In IEEE mode, we recall that the `nsb` obtained at bit-level is approximated by the upper number of bits corresponding to a IEEE754 format. "ILP-time" is the total analysis time of `POP(ILP)` in the case of ILP formulation. We have also "PI-time" to represent the time passed by `POP(ILP)` to find the right policy and to resolve the precision tuning problem. "H", "S", "D" and "LD" denote respectively the number of variables obtained in, half, single, double and long-double precision when using the policy iteration (PI) formulation that clearly displays better results.

Let us focus on the first "TH", "BL", "IEEE" and "ILP-time" columns of Table 9.1. We compute the improvements compared to the case where all variables are in double precision before tuning. For the **arclength** program, the optimization reaches 61% at bit-level while it achieves 43% in the IEEE mode (100% is the percentage of all variables initially in double precision, 121 variables for the original **arclength** program that used 7744 bits). This is obtained in only 0.9 second by applying the ILP formulation. When we refine the solution by applying the policy iteration method (from the sixth column), `POP(ILP)` attains 62% at bit-level and 45% for the IEEE mode. Although POP(ILP) needs more analysis time to find and iterate between policies, the time of analysis remain negligible, not exceeding 1.5 seconds. For a total of 121 variables for the **arclength** original program, POP(ILP) succeeds in tuning 8 variables to half precision (H), 88 variables tuned to single precision (S) whereas 25 variables remain in double precision (D) for an error threshold of 10^{-4}. We remark that our second method displays better results also for the other user error

thresholds. For the **simpson**, **accelerometer**, **rotation** and **lowPassFilter**, the improvement is also more important when using the PI technique than when using the ILP formulation. For instance, for an error threshold of 10^{-6} for the **simpson** program, only one variable passes to half precision, 27 variables turn to single precision while 21 variables remain in double precision with 56% of percentage of total number of bits at bit-level using the policy iteration method. Concerning the **2-Body** and the **pendulum** codes, the two techniques return the same percentage at bit-level and IEEE mode for the majority of error thresholds except for the **pendulum** program where `POP(ILP)` reaches 34% at bit-level when using the PI method for a threshold of 10^{-10}.

Now, we stress on the negative percentages that we obtain in Table 9.1, especially for the **arclength** program with 10^{-10} and 10^{-12} for the columns IEEE, the **lowPassFilter** program for errors of 10^{-8}, 10^{-10} and 10^{-12} and finally for the **2-Body** for almost all the error thresholds. In fact, `POP(ILP)` is able to return new formats for any threshold required by the user without additional cost nor by increasing the complexity even if it fails to have a significant improvement on the program output. To be specific, taking again the **arclength** program, for an error of 10^{-12}, `POP(ILP)` fulfills this requirement by informing the user that this precision is achievable only if 10 variables passes to the long double precision (LD) which is more than the original program whose variables are all in double precision. By doing so, the percentage of IEEE formats for both ILP and PI formulations reaches -17% and -8%, respectively. Same reasoning is adopted for the **lowPassFilter** which spends more time, nearly 12 seconds, with the policy iteration technique to find the optimized formats (total of 841 variables). Also, the number of variables of the former program reaches 583 variables that are correctly optimized according to the different error thresholds. For instance, **2-Body** program, for an error threshold of 10^{-8}, the percentage of optimization attains -7% at bit-level. Note that in these cases, other tools like `Precimonious` [RNN+13] fail to propose formats.

Let us move to analyze the new benchmarks of `POP(ILP)` and start with the **Newton-Raphson** program. For this program, `POP(ILP)` succeeded to tune 7 variables to single precision (S) for an error threshold of 10^{-6} with the ILP method whereas there is no more optimization obtained when using the PI technique that optimizes the carry bits. For the remaining programs: **odometry**, **PID**, **Runge-Kutta** and **Trapeze**, the results of precision tuning are satisfactory. A common behaviour between these programs is that for an error threshold lesser than 10^{-8}, all the variables of these programs are tuned into double precision (D) in a maximum analysis time that does not exceed 2.67 seconds so as to the **Runge-Kutta** program.

9.2 `POP(ILP)` vs. the Prior State-of-the-Art Techniques

In the following experiment, we display the comparison between `POP(ILP)`, the former version of `POP` that uses the `Z3` SMT solver coupled to binary search to find the optimal

Program	TH	BL	IEEE	ILP-time	BL	IEEE	PI-time	H	S	D	LD
	10^{-4}	61%	43%	0.9s	62%	45%	1.5s	8	88	25	0
	10^{-6}	50%	21%	0.9s	51%	21%	1.4s	2	45	74	0
arclength	10^{-8}	37%	3%	0.8s	38%	4%	1.6s	2	6	113	0
	10^{-10}	24%	-1%	1.0s	25%	-1%	1.7s	2	0	116	3
	10^{-12}	12%	-17%	0.3s	14%	-8%	1.5s	2	0	109	10
	10^{-4}	64%	45%	0.1s	67%	56%	0.5s	6	42	1	0
	10^{-6}	53%	30%	0.2s	56%	31%	0.5s	1	27	21	0
simpson	10^{-8}	40%	4%	0.1s	43%	7%	0.3s	1	5	43	0
	10^{-10}	27%	1%	0.1s	28%	1%	0.4s	1	0	48	0
	10^{-12}	16%	1%	0.1s	16%	1%	0.3s	0	1	48	0
	10^{-4}	73%	61%	0.2s	76%	62%	1.0s	53	69	0	0
	10^{-6}	62%	55%	0.2s	65%	55%	1.0s	2	102	0	0
accelerometer	10^{-8}	49%	15%	0.2s	52%	18%	1.0s	2	33	69	0
	10^{-10}	36%	1%	0.2s	39%	1%	1.0s	2	0	102	0
	10^{-12}	25%	1%	0.2s	28%	1%	1.0s	2	0	102	0
	10^{-4}	78%	66%	0.08s	79%	68%	1.3s	46	38	0	0
	10^{-6}	67%	53%	0.08s	68%	56%	0.5s	12	70	2	0
rotation	10^{-8}	53%	29%	0.07s	54%	29%	0.4s	0	46	38	0
	10^{-10}	40%	0%	0.1s	41%	0%	0.5s	0	0	84	0
	10^{-12}	29%	0%	0.09s	30%	0%	0.5s	0	0	48	0
	10^{-4}	68%	46%	1.8s	69%	46%	10.7s	260	581	0	0
	10^{-6}	57%	38%	1.8s	58%	45%	11.0s	258	580	3	0
lowPassFilter	10^{-8}	44%	-7%	2.0s	45%	-7%	11.4s	258	2	581	0
	10^{-10}	31%	-7%	1.7s	32%	-7%	10.9s	258	0	583	0
	10^{-12}	20%	-7%	1.8s	21%	-7%	11.3s	258	0	583	0
	10^{-4}	41%	51%	0.81s	41%	51%	0.82s	5	39	5	0
	10^{-6}	18%	49%	0.78s	18%	49%	0.9s	0	44	5	0
2-Body	10^{-8}	-7%	5%	0.8s	-7%	5%	0.78s	0	5	44	0
	10^{-10}	-34%	-2%	0.8s	-34%	-2%	0.9	0	0	48	1
	10^{-12}	-57%	-11%	0.9s	-57%	-11%	1.0s	0	0	44	0
	10^{-4}	71%	54%	0.15s	71%	54%	0.4s	0	13	0	0
	10^{-6}	60%	50%	0.2s	60%	50%	0.5s	0	12	1	0
Pendulum	10^{-8}	47%	0%	0.12s	47%	0%	0.4s	0	0	13	0
	10^{-10}	33%	0%	0.16s	34%	0%	0.5	0	0	13	0
	10^{-12}	22%	0%	0.11s	22%	0%	0.4s	0	0	13	0
	10^{-4}	80%	68%	0.64s	80%	68%	4.5s	0	7	0	0
	10^{-6}	72%	68%	0.66s	72%	68%	4.3s	0	7	0	0
Newton-Raphson	10^{-8}	63%	30%	0.53s	63%	30%	4.8s	0	0	7	0
	10^{-10}	54%	30%	0.55s	54%	30%	5.9	0	0	7	0
	10^{-12}	46%	30%	0.61s	46%	30%	4.3s	0	0	7	0
	10^{-4}	78%	67%	1.5s	79%	67%	11.12s	0	29	0	0
	10^{-6}	70%	66%	1.0s	71%	67%	10.9s	0	29	0	0
Odometry	10^{-8}	61%	29%	1.92s	62%	29%	13.9s	0	0	29	0
	10^{-10}	52%	29%	1.20s	52%	29%	4.0s	0	0	29	0
	10^{-12}	44%	29%	0.33s	44%	29%	3.5s	0	0	29	0
	10^{-4}	76%	62%	0.13s	77%	62%	0.93s	0	19	0	0
	10^{-6}	67%	62%	0.12s	67%	62%	0.82s	0	19	0	0
PID	10^{-8}	56%	17%	0.13s	57%	17%	1.07s	0	0	19	0
	10^{-10}	45%	17%	0.14s	46%	17%	0.88s	0	0	19	0
	10^{-12}	36%	17%	0.13s	36%	17%	0.9s	0	0	19	0
	10^{-4}	75%	58%	0.16s	75%	58%	1.95s	0	22	0	0
	10^{-6}	65%	58%	0.18s	65%	58%	2.01s	0	22	0	0
Runge-Kutta	10^{-8}	52%	8%	0.20s	53%	8%	1.98s	0	0	22	0
	10^{-10}	40%	8%	0.24s	40%	8%	2.67s	0	0	22	0
	10^{-12}	30%	8%	0.17s	30%	8%	2.11s	0	0	22	0
	10^{-4}	70%	60%	0.15s	75%	60%	1.55s	0	15	0	0
	10^{-6}	60%	27%	0.16s	65%	60%	1.74s	0	15	0	0
Trapeze	10^{-8}	48%	11%	0.16s	53%	11%	1.64s	0	0	15	0
	10^{-10}	37%	11%	0.19s	42%	11%	1.96s	0	0	15	0
	10^{-12}	27%	11%	0.15s	31%	11%	1.81s	0	0	15	0

Table 9.1: Precision tuning results for POP(ILP) for the ILP and PI methods.

solution [BMA19] and the prior state-of the-art Precimonious [RNN+13] as demonstrated in Table 9.2. The goal of this comparison is to identify the tool that finds more precise data formats for the original programs analyzed as rapidly as possible. We remind the reader that we have adjusted some criteria including the number of variables that we optimize so that the comparison is as close as possible to the real behaviors of the two tools. These adjustments were clearly revealed in Chapter 8.

We recall that POP(ILP) combines both ILP and PI formulations. As long as the reader is not warned by switching to the more precise PI method, ILP mode is activated by default.

Program	Tool	#Bits saved - Time in seconds			
		Threshold 10^{-4}	Threshold 10^{-6}	Threshold 10^{-8}	Threshold 10^{-10}
arclength	POP(ILP) (28)	**2464b.** - 1.8s.	**2144b.** - 1.5s.	**1792b.** - 1.7s.	**1728b.** - 1.8s.
	POP(SMT) (22)	1488b. - 4.7s.	1472b. - 3.04s.	864b. - 3.09s.	384b. - 2.9s.
	Precimonious (9)	576b. - 146.4s.	576b. - 156.0s.	576b. - 145.8s.	576b. - 215.0s.
simpson	POP(ILP) (14)	**1344b.** - 0.4s.	**1152b.** - 0.5s.	**896b.** - 0.4s.	**896b.** - 0.4s.
	POP(SMT) (11)	896b. - 2.9s.	896b. - 1.9s.	704b. - 1.7s.	704b. - 1.8s.
	Precimonious (10)	704b. - 208.1s.	704b. - 213.7s.	704b. - 207.5s.	704b. - 200.3s.
rotation	POP(ILP) (25)	**2624b.** - 0.47s.	2464b. - 0.47s.	2048b. - 0.54s.	1600b. - 0.48s.
	POP(SMT) (22)	1584b. - 1.85s.	2208b. - 1.7s.	1776b. - 1.6s.	1600b. - 1.7s.
	Precimonious (27)	2400b. - 9.53s.	**2592b.** - 12.2s.	**2464b.** - 10.7s.	**2464b.** - 7.4s.
accel.	POP(ILP) (18)	**1776b.** - 1.05s.	**1728b.** - 1.05s.	**1248b.** - 1.04s.	**1152b.** - 1.03s.
	POP(SMT) (15)	1488b. - 2.6s.	1440b. - 2.6s.	1056 - 2.4s.	960b. - 2.4s.
	Precimonious (0)	-	-	-	-

Table 9.2: Comparison between POP(ILP), POP(SMT) and Precimonious: number of bits saved by the tool and time in seconds for analyzing the programs.

The results of the mixed-precision tuning are shown for the **arclength**, **simpson**, **rotation** and **accelerometer** programs, as depicted in Table 9.2. Let us state that some examples used in Precimonious benchmarks [RNN+13] cannot be analyzed as-is by POP (in its two versions) for implementation reasons such as the calls to external libraries or the use of syntactic forms not yet implemented in our tool. Conversely, let us also mention that Precimonious fails to tune (zero improvement) some examples handled by POP(ILP) and POP(SMT), e.g. the **lowPassFilter** program.

In Table 9.2, we indicate in bold the tool that exhibits better results for each error threshold and each program. Starting with the **arclength** program, POP(ILP) displays better results than the other tools by optimizing 28 variables. For an error threshold of 10^{-4}, 2464 bits are saved by POP(ILP) in 1.8 seconds while POP(SMT) saved only 1488 bits in more time (11 seconds). Precimonious were the slowest tool on this example with more than 2 minutes with 576 bits for only 9 variables optimized. For the **simpson** program, POP(ILP) do also better than both other tools. However, for the **rotation** program, POP(ILP) saves more bits than the other tools only for an error of 10^{-4} while Precimonious do well for this program for the rest of error thresholds. One possible explanation for this behavior is that POP(ILP) (same for POP(SMT)) is at a disadvantage by considering only the variables optimized by Precimonious when we adjust the comparison criteria.

Finally, Precimonious fails to tune the **accelerometer** program (0 variables) at the time that POP(ILP) do faster with only 1 second to save much more bits than POP(SMT) for any given error threshold.

9.3 Precision Tuning of the N-Body Problem

In the present section, we validate the efficiency of our approach on one of the oldest problem of modern physics, the N-body problem [GK13]. 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 ($N = 5$). We note that the N-body program has been excerpted (not fully) from [Dem21]. 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(ILP) 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 [MKF03]. Compared to other experiments carried out with POP(SMT) [BM19, BM20], 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

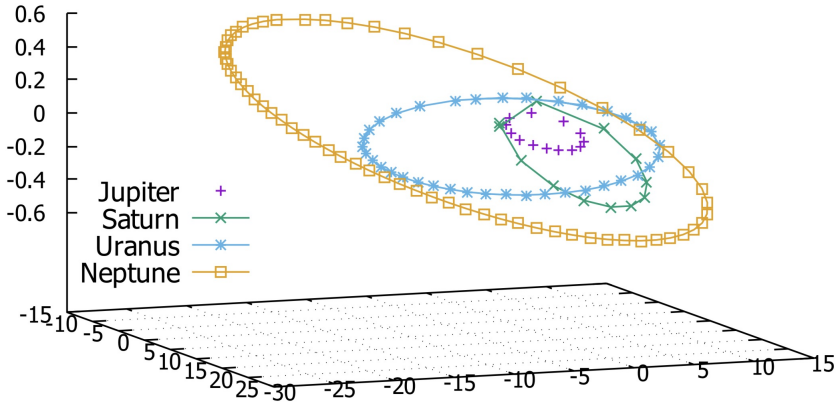


Figure 9.1: Simulated movement of the bodies.

for the N-body problem are the following. First, we measure the distance between the exact position of the bodies, Jupiter, Saturn, Uranus and Neptune by assuming that the Sun position is fixed, computed with 500 bits and the position computed with nsb bits where nsb $= 11, 18, 24, 34, 43$ 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(ILP) to generate a MPFR code with the new data types returned by our tool. Furthermore, we measure the global analysis time taken by POP(ILP) and the execution time of the MPFR generated code and we prove that POP(ILP) returns solutions in a few

seconds. We note that the global analysis time includes the time of the program evaluation, the range analysis determination, the constraint generation and the resolution by the solver.

```
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              [...]
```

Listing 9.1: Left: source program annotated with labels. Right: program with POP generated data types with ILP formulation.

As depicted in Figure 9.1, we aim at modelling the simulation of a dynamical system describing the orbits of planets in the solar system interacting with each other gravitationally. We note that to for the sake of clarity of the graphic, Figure 9.1 uses different simulation times for each body.

We present, in Listing 9.1, 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 assume that all the variables, before `POP(ILP)` analysis, are in double precision and we recall that a range determination is performed by dynamic analysis on the program variables. `POP(ILP)` assigns to each node of the program's syntactic tree a unique control point as mentioned in the left hand side corner of Listing 9.1. The statements `require_nsb(xJupiter,11)`$^{\ell_{2710}}$ and `require_nsb(xSaturn,11)`$^{\ell_{2716}}$ on the last two lines of the code inform the system that `POP(ILP)` user wants to have 11 accurate binary digits (`nsb`) on variables `xJupiter` and `xSaturn` at their control points ℓ_{2710} and ℓ_{2716}, respectively. We recall that a result has `nsb` significants if the relative error between the exact and approximated results is less than $2^{-\mathtt{nsb}+1}$.

Consider the program of the right hand side of Listing 9.1. We display the `POP(ILP)` output N-body program coupled with the generated data types. For `nsb` $=$ 11 bits on variables `xJupiter` and `xSaturn`, `POP(ILP)` 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 of Listing 9.1, 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 Listing 9.1 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 mention that even if the computed `nsb` do not correspond to IEEE74 formats [ANS08], one may either take the IEEE754 format immediately above the computed `nsb` or choose a multiple precision library such as `MPFR` or `POSIT` [UFdD19] (we will discuss more about this point later in this chapter).

9.3.1 Experimental Study

The key feature of our ILP approach, is to generate a set of constraints for each statement of our program. 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.

Integer Linear Problem Formulation with Pessimistic Carry Bit Propagation In order to explain the obtained data types of our N-body program already illustrated in the right hand corner of Listing 9.1, we present the system of constraints that corresponds to a pure ILP formulation as shown in Equation (9.1). 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 Listing 9.1). To make it easier to follow our reasoning, we rewrite hereafter the statement under discussion annotated with the control points.

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

$$C_1 = \left\{ \begin{array}{l} \mathsf{nsb}(\ell_{780}) \geqslant \mathsf{nsb}(\ell_{786}) + (-7) + \xi(\ell_{786})(\ell_{780}, \ell_{785}) - 7, \\ \mathsf{nsb}(\ell_{786}) \geqslant \mathsf{nsb}(\ell_{787}) + (-1) + 0, \\ \mathsf{nsb}(\ell_{774}) \geqslant \mathsf{nsb}(\ell_{780}) + 7 + \xi(\ell_{780})(\ell_{774}, \ell_{779}) - 7, \\ \mathsf{nsb}(\ell_{779}) \geqslant \mathsf{nsb}(\ell_{780}) + 6 + \xi(\ell_{780})(\ell_{774}, \ell_{779}) - 7, \\ \mathsf{nsb}(\ell_{785}) \geqslant \mathsf{nsb}(\ell_{786}) + (-3) + \xi(\ell_{786})(\ell_{780}, \ell_{785}) - 7, \\ \mathsf{nsb}(\ell_{787}) \geqslant \mathsf{nsb}(\ell_{788}), \mathsf{nsb}(\ell_{771}) \geqslant \mathsf{nsb}(\ell_{774}) + \xi(\ell_{774})(\ell_{771}, \ell_{773}) - 1, \\ \mathsf{nsb}(\ell_{773}) \geqslant \mathsf{nsb}(\ell_{774}) + \xi(\ell_{774})(\ell_{771}, \ell_{773}) - 1, \\ \mathsf{nsb}(\ell_{776}) \geqslant \mathsf{nsb}(\ell_{779}) + \xi(\ell_{779})(\ell_{776}, \ell_{778}) - 1, \\ \mathsf{nsb}(\ell_{778}) \geqslant \mathsf{nsb}(\ell_{779}) + \xi(\ell_{779})(\ell_{776}, \ell_{778}) - 1, \\ \mathsf{nsb}(\ell_{782}) \geqslant \mathsf{nsb}(\ell_{785}) + \xi(\ell_{785})(\ell_{782}, \ell_{784}) - 1, \\ \mathsf{nsb}(\ell_{784}) \geqslant \mathsf{nsb}(\ell_{785}) + \xi(\ell_{785})(\ell_{782}, \ell_{784}) - 1, \\ \xi(\ell_{780})(\ell_{774}, \ell_{779}) \geqslant 1, \;\; \xi(\ell_{786})(\ell_{780}, \ell_{785}) \geqslant 1 \\ \xi(\ell_{774})(\ell_{771}, \ell_{773}) \geqslant 1, \;\; \xi(\ell_{779})(\ell_{776}, \ell_{778}) \geqslant 1 \\ \xi(\ell_{785})(\ell_{782}, \ell_{784}) \geqslant 1 \end{array} \right\} \tag{9.1}$$

For this statement, POP(ILP) generates 17 constraints in global as shown in system C_1 of Equation (9.1). We assign to each control point (here 771 to 787) the integer variable nsb which are determined by solving the system C_1. To discuss some of them, the first two constraints of Equation (9.1) 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 $\mathsf{nsb}(\ell_{787}) \geqslant \mathsf{nsb}(\ell_{788})$ 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(ILP) generates such constraints for all the statements of the N-body program.

The last five constraints of system C_1 correspond to the constant carry bit function ξ. For instance, the constraint $\xi(\ell_{780})(\ell_{774}, \ell_{779}) \geqslant 1$ indicates the over-approximation of the carry bit propagation in the ILP approach on the result of the addition stored at control point ℓ_{780}. Finally, for nsb $= 11$ bits as displayed in Listing 9.1, POP(ILP) calls the GLPK solver and consequently finds the least precision needed for all the N-body problem variables as we can observe hereafter. We note that is the total number of bits of the whole program after POP(ILP) optimization is 14636 bits at bit-level:

```
distance|44| = sqrt(  dx|46| * |46|dx|46| + |45|dy|45| * |45|dy|45|
                    +|44|dz|35| * |35|dz|35|);
```

Policy Iteration to Refine Carry Bit Propagation With the PI method, 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 Equation (9.2). Thus, we use the policy iteration method [ABM21] to find an optimal solution as we have explained in Chapter 5. Equation (9.2) displays the new constraints that we add to the global system of constraints C_1. The purpose of these new constraints is to estimate the integer quantity nsb_e that helps to compute the new optimized ξ function already defined in Equation (4.19). In practice, policy iteration makes it possible to break the min and the max 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 Equation (9.2).

$$C_2 = \left\{ \begin{array}{l} \mathsf{nsb_e}(\ell_{780}) \geqslant \mathsf{nsb_e}(\ell_{774}), \\ \mathsf{nsb_e}(\ell_{780}) \geqslant \mathsf{nsb_e}(\ell_{779}), \\ \mathsf{nsb}(\ell_{780}) \geqslant 7 - 6 + \mathsf{nsb}(\ell_{779}) - \mathsf{nsb}(\ell_{774}) + \mathsf{nsb_e}(\ell_{779}) + \xi(\ell_{780}, \ell_{774}, \ell_{779}), \\ \mathsf{nsb_e}(\ell_{780}) \geqslant 6 - 7 + \mathsf{nsb}(\ell_{774}) - \mathsf{nsb}(\ell_{779}) + \mathsf{nsb_e}(\ell_{774}) + \xi(\ell_{780}, \ell_{774}, \ell_{779}), \\ \mathsf{nsb_e}(\ell_{786}) \geqslant \mathsf{nsb_e}(\ell_{780}), \\ \mathsf{nsb_e}(\ell_{786}) \geqslant \mathsf{nsb_e}(\ell_{785}), \\ \mathsf{nsb}(\ell_{786}) \geqslant 7 - (-3) + \mathsf{nsb}(\ell_{785}) - \mathsf{nsb}(\ell_{780}) + \mathsf{nsb_e}(\ell_{785}) + \xi(\ell_{786}, \ell_{780}, \ell_{785}), \\ \mathsf{nsb_e}(\ell_{786}) \geqslant 3 - 7 + \mathsf{nsb}(\ell_{780}) - \mathsf{nsb}(\ell_{785}) + \mathsf{nsb_e}(\ell_{780}) + \xi(\ell_{786}, \ell_{780}, \ell_{785}), \\ \mathsf{nsb_e}(\ell_{774}) \geqslant \mathsf{nsb}(\ell_{771}) + \mathsf{nsb_e}(\ell_{771}) + \mathsf{nsb_e}(\ell_{773}) - 2, \\ \ldots \\ \xi(\ell_{780})(\ell_{774}, \ell_{779}) = \min \left(\begin{array}{l} \max\left(6 - 7 + \mathsf{nsb}(\ell_{774}) + \mathsf{nsb_e}(\ell_{774}), 0\right), \\ \max\left(7 - 6 + \mathsf{nsb}(\ell_{779}) + \mathsf{nsb_e}(\ell_{779}), 0\right), 1 \end{array} \right) \\ \xi(\ell_{786})(\ell_{780}, \ell_{785}) = \min \left(\begin{array}{l} \max\left(-3 - 7 + \mathsf{nsb}(\ell_{780}) + \mathsf{nsb_e}(\ell_{780}), 0\right), \\ \max\left(7 - (-3) + \mathsf{nsb}(\ell_{785}) + \mathsf{nsb_e}(\ell_{785}), 0\right), 1 \end{array} \right) \end{array} \right\} \quad (9.2)$$

Next, it becomes possible to solve the corresponding ILP. If no fixpoint is reached, `POP(ILP)` iterates until a solution is found. By applying this optimization, the new data types of the statement of lines 22 to 24 in Listing 9.1 are given as follows:

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

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 this statement. With the PI method, the total number of bits of the optimized N-body program is $\simeq$ 14335 at bit-level forming a gain of more than 300 bits compared to the pure ILP formulation. In term of complexity, for both ILP and PI methods, `POP(ILP)` 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.

9.3.2 Distance between the Exact and the Computed Positions of the Bodies

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(ILP)` to tune programs in function of the IEEE754 formats (11, 24, 53) [ANS08] as well as for arbitrary word length which can be encoded using libraries such as `MPFR` or POSIT [GY17]. We test the efficiency of `POP(ILP)` analysis in several ways. The experiments shown in Table 9.3 seek to measure the distances between the exact position of each of the bodies of our planetary system and the position computed with `nsb` = 11, 18, 24, 34, 43 and 53 bits. The distances presented in Table 9.3 are given for a single position on the planets which follow the orbits previously presented in Figure 9.1. The positions are taken after 10 and 30 years of simulation time.

More precisely, for this experimentation, we generate the N-body program with all computations done with 500 bits by assuming that this gives the exact solution, and we also generate by the same manner an `MPFR` code with the optimized data types returned by `POP(ILP)`. For example, as we can observe in Table 9.3, for `nsb` = 11, the distance

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(ILP) 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(ILP) Time	38″	39″	41″	37″	37″	37″

Table 9.3: 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(ILP) analysis time and the execution time of the MPFR generated code.

measured for Jupiter is of the order of 10^{-4} for 10 years of simulation which confirms the usefulness of our analysis: desirable results that respects the user nsb requirement where the worst error is of 2^{-11} for nsb = 11 bits. The results are also satisfactory for the remaining planets. For a simulation of 10 and 30 years, the run-time spent to measure the distances reaches maximally 2 minutes 59 seconds for an nsb = 53. Concerning the POP(ILP) time, our analysis took as little as 25 seconds for nsb $= 11$ bits 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 for nsb = 24 bits . With this speed, we believe that for large codes POP(ILP) achieves its best tuning in a minimal time.

Figure 9.2 depicts the capability of POP(ILP) to generate automatically a Python MPFR

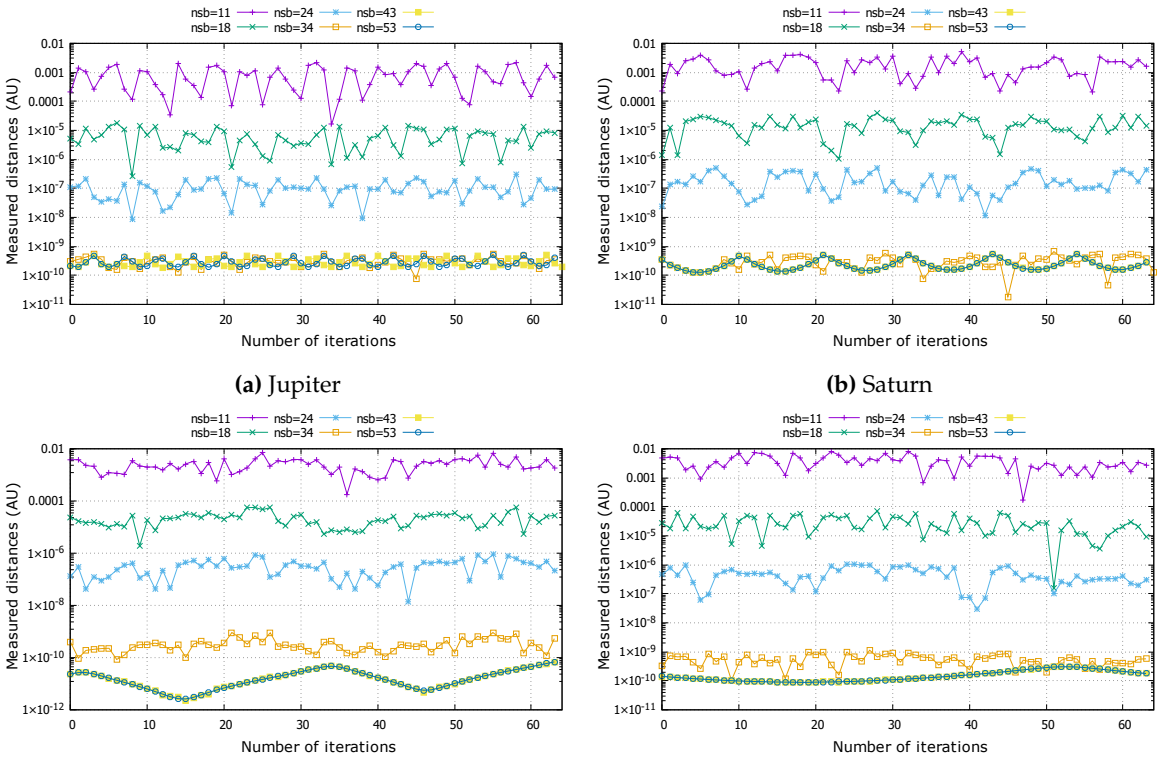
```
xJupiter = mpfr(4.841431617736816,59)
yJupiter = mpfr(-1.1603200435638428,60)
zJupiter = mpfr(-0.10362204164266586,57)
vxJupiter = mpfr(mpfr(0.001660076668485999,61)*mpfr(days_per_year,61),61)
vyJupiter = mpfr(mpfr(0.007699011359363794,61)*mpfr(days_per_year,61),61)
vzJupiter = mpfr(mpfr(-6.904600013513118E-5,61)*mpfr(days_per_year,61),61)
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)
                        [...]
}
```

Listing 9.2: Python MPFR code automatically generated by POP(ILP) for the N-body problem for a nsb $= 18$ bits on the positions of the planets at the end of the simulation.

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(ILP) after analysis for nsb $= 18$ bits. In the future, we plan to also generate code for libraries based on the POSIT number system[1].

We end the experiments by focusing on the curves of Figure 9.2. 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 9.3 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.

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



(a) Jupiter **(b)** Saturn

(c) Uranus **(d)** Neptune

Figure 9.2: Distance between the exact and the computed position for the 5 bodies with nsb = 11, 18, 24, 34, 43 and 53 bits.

9.4 Summary

As first contribution of this Chapter, we evaluated POP(ILP) on several numerical programs. In addition, we provided a detailed comparison of POP(ILP) against the previous version POP(SMT) [BMA19] and the prior state-of-the-art tool Precimonious [RNN+13] in terms of analysis time, speed and the quality of the mixed-precision tuning. With this contribution, we validate the performance of our tool POP(ILP) and we believe its efficiency in tuning more larger codes with more complex structures.

In the second contribution, we have also shown that POP(ILP) is able to tune the N-body program according to different number of significant bits 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. Besides, we have also shown that POP(ILP) is able to generate code for multiple precision libraries, MPFR in practice, and we plan to integrate POSIT libraries in the near future.

To conclude, in chapters 7, 8 and 9, the two implementations of POP have been evaluated. The new ILP approach proposed has exhibited more interesting results on the mixed-precision tuning of the programs. Consequently, we are considering several directions for

our future work. The first one includes 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.

Chapter 10

Conclusion and Perspectives

***

10.1 Summary of Contributions . . . 146
10.2 Short-Term Perspectives . . . 148
10.2.1 Scalability of POP . . . 148
10.2.2 Extension of POP . . . 149
10.2.3 License . . . 149
10.2.4 Handling of Loops . . . 149
10.2.5 Fixed-Point Code synthesis . . . 150
10.2.6 Applications . . . 150
10.3 Long-Term Perspectives . . . 151
10.3.1 Combining Frameworks . . . 151
10.3.2 Applications to Neural Networks (NNs) . . . 151
10.3.3 GPU Oriented Approach . . . 152
10.4 Final Words . . . 153

The work presented in this thesis addressed the problem of precision tuning which is emerging as a new trend in HPC, especially when new error-tolerant applications are considered, to improve performance metrics such as computation latency and power consumption. This work has led to the development of POP which is an automated tool for precision tuning. Based on static analysis, POP computes the minimal number of bits needed at bit-level for the variables and the intermediary results of programs in order to accomplish the user requirement of accuracy. In this chapter, we summarize the different techniques used by POP in order to obtain a fast and efficient bit-level precision tuning. Additionally, we will discuss the prospects of this thesis and suggest various research directions for future work, notably in short and long-term, that would leverage our results.

10.1 Summary of Contributions

We presented in this thesis eight contributions for the precision tuning problem. In Chapter 2, we started this dissertation by introducing a brief theoretical underpinnings of the different concepts related to the development of POP. The background was related to finite-precision standards to represent real numbers. Also, we highlighted the static analysis technique with a concise description of the abstract interpretation theory. Nevertheless, we outlined the PI method and its previous introduction in the context of static analysis. We closed the chapter by presenting the SMT and ILP theories exploited by our tool POP.

In Chapter 3, we discussed the tools and the approaches that have been proposed in the literature to deal with the precision tuning problem. The originality of this survey is that we extended our study to other tools for numerical accuracy analysis, code transformation as well as for their combinations (e.g. Daisy [ID17, DIN+18, DHS18, DV19], Anton [DHS18], etc.) We also focused on the differences between the approaches proposed in the bibliography and our own approach. Conclusively, we clearly showed that our approach relies on a very different paradigm avoiding the usual trial-and-error one.

In Chapter 4, we formulated the problem of precision tuning with two different methods: the **SMT-based method**, implemented in a version of POP called `POP(SMT)` and the **ILP-based method** implemented in a version called `POP(ILP)`. Consequently, each version of the POP tool expresses differently the set of constraints generated. We started the chapter by defining the transfer functions of the elements of POP language input programs. The SMT-based method combines a forward and a backward error analysis. This method is considered as the first contribution of this thesis. In fact, the forward analysis examines how errors are magnified by each operation, aiming to determine the accuracy of the results. The backward analysis takes as input the user requirement of accuracy and the results of the forward analysis in order to determine the precision of the inputs and intermediary results. Next, this analysis has been expressed as a set of linear constraints easily checked by a SMT solver. The details about the constraints generation were provided in Chapter 5. The first originality of the SMT-based method in comparison with Martel's work [Mar17] is the richness of the language handled by POP. It accepts programs with the four basic elementary functions, the trigonometric functions, commands, loops, arrays, matrices and the square root function. The second originality is the reexamination of the carry bit function that can occur throughout the program computations. More precisely, we underlined that it is crucial to modify this definition in order to improve the precision of our analysis.

The SMT-based method has permitted us to propose another new form of the precision tuning problem. The idea comes from several limitations that we have figured out when using the non-optimizing solver Z3 specially in term of complexity and analysis time. As a consequence, we proposed the ILP-based method which translates the precision tuning problem as ILP problem generated from the program source code. The ILP formulation is the second contribution of this thesis. This is done by reasoning on the most significant bit

(ufp) and the number of significant bits (nsb) of the values which are integer quantities. The optimal solution computed by a classical LP solver (we use GLPK [Mak] in practice) gives the optimized data types that satisfy the user accuracy requirement in a polynomial-time. Consequently, our tuning is not dependant of any particular computer arithmetic. However, we must say that we have over-approximated the carry bit propagation throughout the computations in the ILP formulation.

The third contribution consists of introducing a second set of semantic equations which make it possible to tune even more the precision by being less pessimistic on the propagation of carries in arithmetic operations. By doing so, the problem does not reduce any longer to an ILP problem. Then we used the policy iteration (PI) method to find efficiently the solution.

In Chapter 6, we described the fourth contribution of this thesis. It consists of the implementation of the SMT-based method and ILP-based method in POP. We started by presenting the stages of implementation and the different architectures of POP(SMT) and POP(ILP). As well, we have introduced the different benchmarks that we used to evaluate the performance of each version of the tool.

In Chapter 7, we experimented POP(SMT) on two examples coming from the IoT field. The first example is an accelerometer which can be used to measure the static angle of tilt or inclination. Our experimental results showed that POP(SMT) succeeded in computing the accuracy needed for each variable and intermediary results of the accelerometer program. For instance, we measured an improvement ranging from 65 % to 84 % for an accuracy lesser than 23 bits. The second example consists of the pedometer program that implements a step counting algorithm for embedded applications. This example is significantly more complex than the accelerometer one. Our results showed that POP(SMT) succeeded in tuning the majority of variables. For instance, we measured an improvement ranging from 64 % to 79 % for an accuracy lesser than 32 bits. In addition, POP(SMT) managed correctly the mixed-precision tuning which is confirmed by the different formats of program variables obtained after the analysis. It is also noticeable that POP(SMT) execution time remain very short, even for complex programs like the code of the pedometer. Thus, we believe that the results discussed in this chapter can be very helpful at system design level, to dimension the hardware. It will help the architect to choose which compromise he wants between accuracy and memory consumption, which is a key point for the performance of IoT devices in particular.

Chapters 8 and 9 deal with the evaluation of the performance of POP(SMT) and POP(ILP) in several manners.

In Chapter 8, we provided a detailed comparison of POP(SMT) over the prior state- of-the-art tool Precimonious [RNN+13] in terms of analysis time, speed and the quality of the solution. The experimental results showed that POP(SMT) succeeded to tune more programs than Precimonious. In addition, we deduced that our tool returned better mixed-precision results for different user accuracy requirements where the variables are tuned into FP8, FP16,

FP32, FP64 and FP128 precision. The second experimentation in this chapter concerned the use of two different cost functions to POP(SMT) global system of constraints in order to optimize the solutions returned by the Z3 solver. Next, we compared the performance of our tool for each of these functions on different programs coming from scientific computing, signal processing or IoT. The results have shown that this can influence the total number of bits optimized in the tuned programs by obtaining different optimized precision when using a function instead of the other. Moreover, we measured the error between the exact results given by an execution in multiple precision (using MPFR [FHL+07]) and the results of optimized programs by POP(SMT) and we found that the measured error curves are always below the theoretical errors required by the user for the different benchmarks.

In Chapter 9, we evaluated the performance of POP(ILP) on new benchmarks and we compared the mixed-precision tuning obtained when activating the pure ILP method and the optimizing PI method. Second, we presented a full comparison between the behaviour of POP(ILP) [ABM21, BM21b] against the first version POP(SMT) [BMA19, BM19, BM20, BM21a] and Precimonious [RNN+13] and we showed that our results encompass the results of these tools. Finally, we demonstrated the efficiency of POP(ILP) to tune the classical gravitational N-body problem ($\approx$ 400 LOCs) 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.

10.2 Short-Term Perspectives

In this section, we present the short-term perspectives of our thesis. These perspectives are related to the internal behaviour and functionality of POP. We mentioned many perspectives throughout our presentation of the contributions and their implications. The objective of this section is to develop these perspectives and aggregate them.

10.2.1 Scalability of POP

Concerning scalability, POP generates a linear number of constraints and variables in the size of the analyzed program. The only limitation is the size of the problem accepted by the solver. In future work, our objective is to address the scalability issue by at least two ideas. The first idea consists of studying how to tune a program partitioned into several parts such that the same precision is used for all the statements of the same part (e.g. an arithmetic expression, a line of code, a function, a loop, etc.) This will significantly reduce the size of our constraint systems and make the tuning scale up. In addition, concerning big arrays, we will study how to incorporate lossy compression techniques such as ZFP [FDH+20] or SZ [DC16] and how to determine the compression rate using our tuning tool POP.

The second idea consists of exploring commercial LP solvers that are less limited in the size of the ILP problem (so as the GLPK solver [Mak] used in this thesis) such as CPLEX [Cpl09] and Gurobi [Gur21]. With this solution, we may expect that even if the

ILP problem generated from the code source is large then the solver is able to return an optimal solution in a short time.

10.2.2 Extension of POP

In this future work, we manage to extend our software by following two directions. We recall that we have simplified the implementation of POP by considering that a range determination is performed by dynamic analysis on the variables of our programs and that no overflow arises during our analysis. From this time on, our first extension consists of adopting static analyzers with sophisticated abstract domains in order to infer safe ranges on our variables. Consequently, two possibilities of this extension are under study. The first one is to implement a new static analyzer from scratch. The implementation of this idea will probably take more time, but maybe we will overcome the shortcomings of the error analysis tools that we have already studied in Chapter 3. The second possibility is more relevant. It consists of using one of the today's error analysis method that is designed to span entire input ranges and produce tight error bounds. We come back on this point later in Section 10.3.1.

The second extension of POP is related to the language of programs. Currently, our tool handles programs with loops, arrays, matrices, conditionals, arithmetic expressions, trigonometric functions, square root function, etc. Honestly, we found some difficulties when we compared POP to other state-of-the-art tools and specially Precimonious [RNN+13]. One of these difficulties is the incapability of our tool to analyze some benchmarks of these tools that contain calls to external libraries or use of syntactic forms that we have not yet implemented. As a result, the comparison is no longer feasible. For these reasons, we will extend the POP language to deal with other structures such as functions. In fact, functions are also easy to manage since only one type per argument and returned value need, in general, to be inferred.

10.2.3 License

The current version of POP is stable whilst its licence is still proprietary. We plan to distribute POP online with some open source license within 1 to 2 months. Initially, the distribution will perhaps be done by taking into account the new extensions seen in sections 10.2.1 and 10.2.2.

10.2.4 Handling of Loops

Scalable error analysis is central to improve the precision tuning process. In this context, we are starting an international collaboration with the team of the university of Utah that developed the SATIRE [DBG+20] and FPDetect [DKB+20] tools (presented in Chapter 3). The SATIRE tool conducts error analysis for expressions containing over 4 million operator nodes situated in a straight-line program block. The FPDetect tool can perform analytical error analysis for stencil codes with a view to bound the maximum observable error. In

particular, the SATIRE and FPDetect tools typically take an HPC application, computing on each point of a grid some mathematical function using a stencil. In order to bound the accuracy on the results, formal expressions describing the propagation of errors throughout the computations are used. This is possible because no loop is present in the code. Otherwise, the results could depend on the computation of the fixpoint for programs with several execution paths and one would have to take the worst case for soundness reasons. In loops, this would make the fixpoint computation difficult to compute efficiently. Unfortunately, both tools can conduct this error analysis only for straight-line codes without conditionals or loops. As policy iteration exhibits very good results in practice when used in static analysis [AGG12b, BSC12, CGG+05, RG15, ABM21], we aim at using in this collaboration, this technique to develop loop-level error analysis in order to obtain tight loop fixpoints.

10.2.5 Fixed-Point Code synthesis

Fixed-point representations are an important resource in application development whenever the need to overcome computational resource limitations emerges [CCC+20]. Such representations are principally employed in embedded applications. Additionally, they are also exploited as a mean to data size tuning for HPC tasks. Our ambition from this work is 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. What makes this future direction achievable in the short term is the ability of POP to find the optimal precision needed at bit-level. As we have translated the precision given in bit-level into the floating-point arithmetic, our efforts will then focus on exploiting the fixed-point numerical representation by considering the fact that some architectures are more suited to fixed-point computations than others. The case studies for this point will belong to embedded and IoT applications.

10.2.6 Applications

While POP has been successfully evaluated on different benchmarks coming from various domains, it has not yet been tested on codes of similar sizes to what tools like SATIRE [DBG+20] can handle. Our efforts currently focus on porting the benefits of precision tuning to new classes of applications. Hence, it is important to have a suite of benchmarks spanning multiple application domains. To start with, we would like to analyze larger programs containing several complex structures such as nested loops, conditionals and big arrays. Also, we can start by the FPBench community which proposes on their webpage[1] more than 130 benchmarks from different sources (FPTaylor [SJRG15], Herbie [PSWT15], Rosa [DK17] and Salsa [Dam16]) covering a variety of application domains.

Nevertheless, we would like to strengthen the collaborations inside the LAMPS lab-

[1] https://fpbench.org/community.html

oratory since it links several multidisciplinary axes such as contact and fluid mechanic, statistical physics, optimization, etc. This will allow us to enlarge the catalog of benchmarks of POP. Furthermore, these collaborations may help us to brainstorm on new case studies that may address other interesting issues.

10.3 Long-Term Perspectives

After presenting the short-term perspectives of our thesis, we present in this section, the professional project that we want to acquire in the next short years. These projects are based on the miscellaneous prerequisites and the background that we learned during this thesis.

10.3.1 Combining Frameworks

In the literature review of Chapter 3, we have discussed automated tools based on error analysis methods, rewriting-based optimization and mixed-precision tuning. Also, we have shown that combining these tools will help users to meet their development needs. In this work, we are interested in combining our tool for precision tuning POP with other tools performing error analysis and code transformation tasks. Several tools are already inspiring us such as Daisy [DV19], Anton [DHS18] and more recently Pherbie (available at https://herbie.uwplse.org).

As part of our collaboration with the university of Utah, we plan to use their static error analysis tools such as FPTaylor [SJRG15], SATIRE [DBG+20] and FPDetect [DKB+20] instead of the dynamic range determination performed by our tool. Also, commercial tools including Astrée [CCF+05] and Fluctuat [GMP02] are worth a try. Likewise, we believe that coupling code rewriting and precision tuning could be a promising approach that can improve both the accuracy and speed of floating-point expressions. Therefore, combining POP with the optimizing tool Salsa [Dam16] will help us to obtain a best compromise between precision and accuracy of the program variables.

10.3.2 Applications to Neural Networks (NNs)

Given how research directions can evolve, we are leaving the possibility of work on Deep Neural Networks (DNNs) precision tuning open. Most of today's numerical computations are performed using floating-point data operations for representing real numbers. The precision of the related data types should be adapted in order to guarantee a desired overall rounding error and to strengthen the performance of programs. In the context of DNNs, this adaptation lies to two different techniques. First, as DNNs become larger and more complex, recent work has shown that they can be compressed and different strategies have been introduced such as weight pruning, low-rank factorization or quantization [JGM+20]. Second, the precision tuning technique is important for several applications from a wide variety of domains and, more recently, for NNs [IM19]. Intuitively, tuning the precision of a NN can be reduced to an ILP approach formulation as follows:

- We can deduce the number of significant digits of the result of some operation $z = x * y$ from the (integer) weight of the most significant bit and the number of significant digits of the operands x and y.
- The most significant bits of the manipulated values may be computed by a prior static analysis. The number of significant digits of the inputs are known, it corresponds to the precision of the inputs.
- The number of significant digits of the outputs is also known, it corresponds to a requirement imposed by the user on the desired accuracy of the results.

Gathering all these points, one may build an ILP problem which unknowns are the (integer) numbers of significant of the intermediary computations. For example, let us consider the code snippet on the left hand side of Figure 10.1, corresponding to a dot product similar to what NNs compute.

```
1  w1 = 0.1;
2  w2 = 2.3;
3  x1 = 0.4;
4  x2 = 0.6;
5  y = w1 × x1 + w2 × x2;
6  require_nsb(y, 18);
```

```
1  w1^|-4,17| = 0.1^|-4,17|;
2  w2^|1,17| = 2.3^|1,17|;
3  x1^|-2,16| = 0.4^|-2,16|;
4  x2^|-1,17| = 0.6^|-1,17|;
5  y^|0,18| = w1^|-4,17| ×^|-3,18| x1^|-2,16|
6           +^|0,17| w2^|1,17| ×^|0,18| x2^|-1,17|;
7  require_nsb(y, 18);
```

Figure 10.1: Left: Initial program. Right: Annotations after precision tuning with POP.

As shown in the right hand side of Figure 10.1, POP computes at each point of this program the pair $|m, s|$ where m is the weight of the most significant bits and s the number of significants such that these numbers satisfy the accuracy requirement that y has 18 significant bits at the end of the computation.

This research direction aims at developing a similar approach for DNN compression. In other words, the work planned for this perspective is twofold:

- **Theoretical**: reduction of compression techniques to an ILP problem and integration with precision tuning techniques for DNNs. In particular, we aim at applying this approach to weight pruning and quantization compression techniques.
- **Practical**: development of a tool based on the former theoretical framework. This tool will solve ILP problems combining constraints for compression and precision tuning.

10.3.3 GPU Oriented Approach

Nowadays, GPUs have been extensively used to accelerate scientific applications from a variety of domains [LWSB19, KSW+19b, GB20]: computational fluid dynamics, astronomy

and astrophysics, climate modeling, numerical analysis, neural networks, to name a few. As HPC scientific applications increasingly rely on GPU accelerators to perform floating-point arithmetic, tools to extract the maximum performance out of floating-point intensive computations are also becoming increasingly important. In recent NVIDIA GPUs, for example, the throughput of single precision floating-point operations is twice that of double precision operations. In this research axis, we seek to extend `POP` to tune floating-point mixed-precision scientific applications on GPUs. Unlike existing tools, our objective is to provide application-level guidance on precision level for entire GPU applications rather than localized kernels, functions, or instructions.

10.4 Final Words

At the end of this thesis, we believe that our work contributes effectively in developing the upcoming methods for precision tuning and helps in designing adequate tools that fill the gaps of the existing ones. We may say that our work also opens many perspectives for both researchers and tool makers. We summarize in Table 10.1 these perspectives.

Perspectives
Produce tight error bounds by a static analyzer
Reduce the number of variables and constraints
Extend `POP` language by including functions
Combining `POP` with rewriting and error estimation tools
Integrate loops by applying PI method in `Satire`-like tools
Set the `POP` code available in open source
Infer Fixed-point arithmetic code synthesis in `POP`
Observe how `POP` works on more complex use cases
Negociate precision tuning for DNNs and GPU applications

Table 10.1: The short and long-term perspectives of our thesis.

Part V

Résumé Étendu à L'intention du Lecteur Francophone

Chapitre 11

Analyse Statique pour le Réglage de la Précision Numérique

***

11.1 Notions Préliminaires et Langage de l'Outil POP . . . 159
11.1.1 Notions Préliminaires . . . 159
11.1.2 Langage Impératif . . . 160
11.2 Génération des Contraintes par Analyse Statique . . . 161
11.2.1 Analyse en Avant et en Arrière . . . 161
11.2.2 Programmation Linéaire en Nombres Entiers (ILP) . . . 162
11.2.3 Itération sur les Politiques (PI) . . . 163
11.3 L'outil POP . . . 165
11.3.1 Implémentation . . . 165
11.3.2 Réglage de la Précision et Internet des Objets (IoT) . . . 167
11.4 Résultats Expérimentaux . . . 169
11.4.1 Évaluation des Performances de POP(SMT) et POP(ILP) . . . 169
11.4.2 Comparaison avec l'Outil Precimonious . . . 170
11.4.3 Génération du Code de Précision Multiple . . . 171
11.5 Synthèse de l'Existant . . . 174

Contexte et Enjeu Majeur

De nos jours, la consommation d'énergie est devenue un aspect critique dans l'évolution des systèmes de calcul haute performance (HPC). En effet, les super-

calculateurs modernes fonctionnent avec d'énormes quantités d'énergie électrique. Par exemple, le rapport du projet des Top500[1] montre que les performances des super-calculateurs doublent approximativement chaque année, tandis que la consommation d'énergie augmente également. En juin 2020, le supercalculateur le plus puissant Fugaku, numéro 1 du Top500, est aujourd'hui capable d'atteindre environ 400 pétaFLOPS (400×10^{15} FLOPS) en terme de puissance de calcul. Cela équivaut à assembler des dizaines de millions d'ordinateurs portables contre les 148,6 pétaFLOPS du supercalculateur prédécesseur, Summit. Dans le même temps, ces super-calculateurs sont dans le top dix du Green500[2], avec environ 14,7 gigaFLOPS par watt. Cependant, la consommation d'énergie continue d'augmenter ; alors que l'ordinateur Summit a une consommation de 10 MW, Fugaku monte à 28 MW. En résumé, maîtriser la consommation énergétique des plateformes HPC est devenu une nécessité. Il est considéré non seulement comme un moyen de contrôler les coûts mais aussi comme un pas en avant sur la voie des exaflops. Pour résumer, notre discussion a montré que la consommation d'énergie est devenue un problème majeur dans de nombreux domaines car nous vivons maintenant dans un monde où l'énergie est rare. Pour s'attaquer à ce problème, la conception d'applications évolutives, fiables et économes en énergie reste un véritable défi à explorer. En d'autres termes, les concepteurs cherchant à réduire la consommation d'énergie doivent être aidés dans le choix des protocoles, des services adéquats et des meilleures implémentations de leurs applications par rapport à l'infrastructure ciblée.

Objectif de cette Thèse

L'objectif de cette thèse est de proposer une approche qui guide les développeurs et les concepteurs pour atteindre le meilleur compromis entre la puissance et la performance pour atteindre l'efficacité énergétique nécessaire. Cette approche consiste à utiliser des représentations numériques à précision réduite (ou personnalisée) qui a été largement reconnue, ces dernières années, comme l'un des outils prometteurs dans l'arsenal du concepteur. Ce procédé, également appelé *precision tuning* (réglage de la précision), peut permettre d'économiser de la mémoire et, par voie de conséquence, il a un impact positif sur l'empreinte des programmes concernant la consommation d'énergie, l'utilisation de la bande passante et le temps de calcul.

Pour répondre au problème de réglage de la précision, nous menons dans cette thèse des méthodes d'analyse statique qui contribuent à une détermination rapide et efficace de la précision minimale sur les entrées et les résultats intermédiaires des programmes numériques. Ces méthodes ont été intégrées dans notre outil appelé `POP`. Dans ce chapitre, nous présentons un résumé étendu de la thèse. Il est, du fait de sa taille réduite,

[1] LeTop500 est un classement statistique montrant les caractéristiques et les performances des 500 machines les plus puissantes au monde (https://www.top500.org/).

[2] Le Green500 fournit des classements des super-calculateurs les plus économes en énergie au monde (https://www.top500.org/lists/green500/).

nécessairement moins complet que le manuscrit anglophone. Néanmoins, nous avons préservé l'introduction, la conclusion et les contributions clefs des différentes parties pour le lecteur francophone intéressé.

11.1 Notions Préliminaires et Langage de l'Outil POP

11.1.1 Notions Préliminaires

Notre technique de réglage de précision est indépendante d'une arithmétique particulière (par exemple IEEE754 [ANS08] et POSIT [CGG+05]). En fait, nous manipulons des nombres dont nous connaissons leur ufp (*unit in the first place*) et le nombre de chiffres significatifs nsb (*number of significant bits*). Nous supposons aussi que les constantes produisant dans les codes sources de notre logiciel POP sont exactes et que nous bornons les erreurs introduites par les calculs en précision finie. Ensuite, nous désignons par $\mathsf{ufp}_e(x)$ et $\mathsf{nsb}_e(x)$, les ufp et nsb de l'erreur sur x . Ces fonctions sont définies ci-après et une présentation plus intuitive est donnée dans la Figure 11.1.

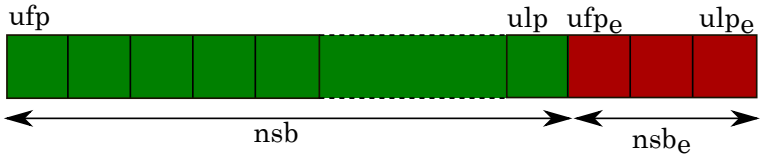


FIGURE 11.1 : Représentation de ufp, nsb et ulp des nombres et des erreurs.

Unit in the First Place Le poids du bit le plus fort d'un nombre réel x (éventuellement codé jusqu'à un certain mode d'arrondi par un nombre à virgule flottante ou un nombre à virgule fixe) est donné par l'équation (11.1). Cette fonction est indépendante de la représentation de x.

$$\mathsf{ufp}(x) = \begin{cases} \min\{i \in \mathbb{Z} : 2^{i+1} > |x|\} = \lfloor \log_2(|x|) \rfloor & \text{if } x \neq 0, \\ 0 & \text{if } x = 0. \end{cases} \tag{11.1}$$

Number of Significant Bits Intuitivement, $\mathsf{nsb}(x)$ est le nombre de bits significatifs de x. Soit $\hat{x}$ l'approximation de x en précision finie et soit $\varepsilon(x) = |x - \hat{x}|$ l'erreur absolue. D'après Parker [Par97], si $\mathsf{nsb}(x) = k$, pour $x \neq 0$, alors :

$$\varepsilon(x) \leqslant 2^{\mathsf{ufp}(x)-k+1} \ . \tag{11.2}$$

De plus, si $x = 0$ alors $\mathsf{nsb}(x) = 0$. Par exemple, si la valeur binaire exacte 1.0101 est approximée par $x = 1.010$ ou $x = 1,011$ alors $\mathsf{nsb}(x) = 3$.

Unit in the Last Place Le poids du bit le moins fort ulp d'un nombre x est défini par :

$$\mathsf{ulp}(x) = \mathsf{ufp}(x) - \mathsf{nsb}(x) + 1 \ . \tag{11.3}$$

Erreurs de Calculs Nous définissons le poids du bit le plus fort de l'erreur sur x par $\mathsf{ufp_e}(x) = \mathsf{ufp}(x) - \mathsf{nsb}(x)$. Le nombre de bits significatifs de l'erreur sur x est noté $\mathsf{nsb_e}(x)$. Il est utilisé pour optimiser la fonction du bit de retenue ξ définie ci-après. Comme mentionné précédemment, nous supposons qu'il n'y a pas d'erreur sur une constante c apparaissant dans les programmes, c'est-à-dire $\mathsf{nsb_e}(c) = 0$. Néanmoins, les $\mathsf{nsb_e}$ des résultats des opérations élémentaires peuvent être supérieurs à 0. De plus, le poids du bit le moins fort de l'erreur sur x est noté par $\mathsf{ulp_e}(x)$ et par conséquent, $\mathsf{ulp_e}(x) = \mathsf{ufp_e}(x) - \mathsf{nsb_e}(x) + 1$.

Bit de Retenue Lors d'une opération entre deux nombres c_1 et c_2, un bit de retenue peut se propager à travers l'opération. Nous modélisons le bit de retenue par une fonction notée ξ calculée comme suit : Si l' ulp de l'un des deux opérandes c_1 ou c_2 est supérieur à ufp de l'autre opérande (ou inversement) alors c_1 et c_2 ne sont pas alignés et $\xi = 0$ (sinon $\xi = 1$). Figure 11.2 illustre le principe de la fonction ξ.

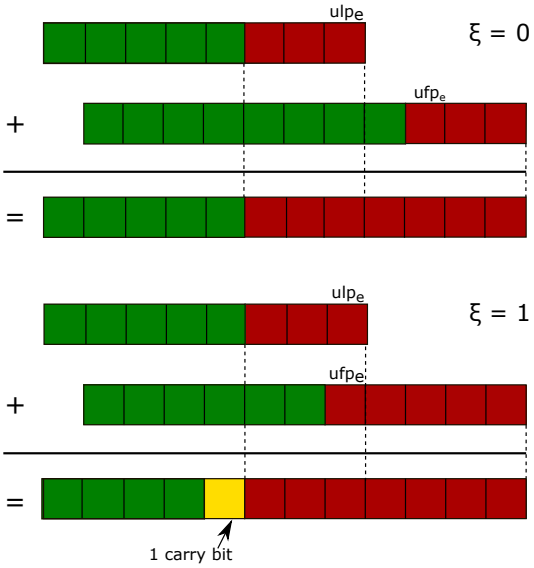


FIGURE 11.2 : Représentation de la fonction ξ du bit de retenue.

11.1.2 Langage Impératif

Dans cette section, nous décrivons le langage des programmes d'entrée de notre logiciel POP à partir duquel nous générons des équations sémantiques afin de déterminer la précision nécessaire pour les valeurs numériques du programme. La Figure 11.3 définit le langage impératif simple dans lequel nos programmes d'entrée sont écrits.

On note *Id* l'ensemble des identifiants et *Lab* l'ensemble des points de contrôle du programme comme moyen d'affecter à chaque élément $e \in Expr$ et $c \in Cmd$ un unique point de contrôle $\ell \in Lab$. Par exemple, dans `c#p`, `p` indique la précision initiale de la constante c dans le code source. La déclaration des vecteurs est exprimée par l'instruction `create_vector`(v,s)$^\ell$, tandis que s désigne la taille du vecteur v. Ensuite, l'instruction

`require_nsb`(x,n)$^{\ell}$ indique le nombre minimal de bits significatifs n qu'une variable x doit avoir à un point de contrôle ℓ ($\mathsf{nsb}(x) = n$). Le reste de la grammaire est standard.

$$\ell \in Lab \quad x \in Id \quad \odot \in \{+, -, \times, \div\} \quad math \in \{\sin, \cos, \tan, \arcsin, \log, \ldots\}$$
$$\textbf{Expr} \ni \mathbf{e} : \mathrm{e} ::= \mathrm{c}\#p^{\ell} \mid x^{\ell} \mid e_1^{\ell_1} \odot^{\ell} e_2^{\ell_2} \mid math(e^{\ell_1})^{\ell} \mid sqrt(e^{\ell_1})^{\ell}$$
$$\textbf{Cmd} \ni \mathbf{c} : c ::= c_1^{\ell_1}; c_2^{\ell_2} \mid x =^{\ell} e^{\ell_1} \mid \texttt{while}^{\ell}\ b^{\ell_0}\ \texttt{do}\ c_1^{\ell_1} \mid \texttt{if}^{\ell}\ b^{\ell_0}\ \texttt{then}\ c_1^{\ell_1}\ \textit{else}\ c^{\ell_2} \mid$$
$$\texttt{create_vector}(v, s)^{\ell} \mid \texttt{require_nsb}(x, n)^{\ell}$$

FIGURE 11.3 : Langage des programmes de POP.

11.2 Génération des Contraintes par Analyse Statique

Nous avons présenté dans cette thèse huit contributions pour le problème de réglage de précision. Nous avons commencé cette thèse en introduisant les différents concepts liés au développement de POP. La première partie était lié à la norme IEEE754 [ANS08] et à l'arithmétique à virgule fixe utilisées pour représenter les nombres réels. Aussi, nous avons mis en évidence la technique d'analyse statique avec une description concise de la théorie de l'interprétation abstraite. Nous avons aussi présenté la méthode d'itérations sur les politiques [CGG+05] et son utilisation dans le cadre de l'analyse statique. De plus, nous avons exposé les théories SMT et ILP exploitées par notre outil POP.

Dans l'état de l'art, nous avons discuté les outils et les approches qui ont été proposés dans la littérature pour traiter le problème de réglage de précision. L'originalité de cette enquête est que nous avons étendu notre étude à d'autres outils d'analyse de précision numérique, de transformation de code ainsi que pour leurs combinaisons. Nous nous sommes également concentrés sur les différences entre les approches proposées dans la bibliographie et notre propre approche intégrée dans POP. En conclusion, nous avons clairement montré que notre approche repose sur un paradigme différent de celui qui a été déjà présenté dans la littérature.

11.2.1 Analyse en Avant et en Arrière

La méthode d'analyse des erreurs en avant et en arrière est considérée comme la première contribution de cette thèse [BMA19]. Cette méthode, appelée la **méthode basée sur SMT**[3] a été implémentée dans une version de POP appelée `POP(SMT)`. En fait, l'analyse en avant examine comment les erreurs sont amplifiées par chaque opération, visant à déterminer la précision des résultats. L'analyse en arrière prend en entrée l'exigence de précision donnée par l'utilisateur et les résultats de l'analyse en avant afin de déterminer la précision des entrées et des résultats intermédiaires. Ensuite, cette analyse a été exprimée sous la forme d'un ensemble de contraintes linéaires facilement vérifiées par un solveur SMT tel que Z3[dMB08]. La première originalité de la méthode basée sur SMT par rapport aux travaux de

[3]Satisfiabilité modulo des théories

Martel [Mar17] est la richesse du langage manipulé par POP. Il accepte des programmes avec les quatre fonctions élémentaires de base, les fonctions trigonométriques, les commandes, les boucles, les tableaux, les matrices et la fonction racine carrée. La seconde originalité est que POP réexamine la propagation du bit de retenue (voir la Figure 11.2) qui peut intervenir tout au long des calculs du programme. Plus précisément, nous avons souligné qu'il est crucial de modifier cette définition afin d'améliorer la précision de notre analyse.

11.2.2 Programmation Linéaire en Nombres Entiers (ILP)

La méthode basée sur SMT, que nous avons évoqué dans la Section 11.2.1, nous a permis de proposer une nouvelle forme du problème de réglage de précision. L'idée vient de plusieurs limitations que nous avons découvert lors de l'utilisation du solveur non-optimisant Z3 spécialement en termes de complexité et de temps d'analyse. En conséquence, nous avons proposé la méthode basée sur ILP qui traduit le problème de réglage de précision en problème de programmation linéaire en nombres entiers généré à partir du code source du programme. La formulation ILP est la deuxième contribution de cette thèse [ABM21]. Cela se fait en raisonnant sur le poids du bit le plus fort (ufp) et le nombre de bits significatifs (nsb) des valeurs qui sont des quantités entières. Dans la Figure 11.4, nous montrons le système de contraintes que POP génère en utilisant la méthode basée ILP.

Les règles de cette figure sont basées sur notre grammaire déjà présentée dans la Figure 11.3. On a $\varrho : \mathrm{Id} \to \mathrm{Id} \times \mathrm{Lab}$ est un environnement qui relie chaque identifiant x à sa dernière affectation x^ℓ. En supposant que $x :=^\ell e^{\ell_1}$ est la dernière affectation de x, l'environnement ϱ relie x à x^ℓ. Ensuite, $\mathcal{E}[\![e]\!]\,\varrho$ génère l'ensemble des contraintes pour une expression $e \in Expr$ dans l'environnement ϱ. Dans la suite, nous définissons formellement ces contraintes pour chaque élément de notre langage. Aucune contrainte n'est générée pour une constante c#p comme mentionné dans la règle (CONST) de la Figure 11.4. Pour la règle (ID) d'une variable x^ℓ, nous exigeons que le nsb au point de contrôle ℓ soit inférieur à son nsb dans la dernière affectation de x donné en $\varrho(x)$. Pour un opérateur binaire $\odot \in \{+, -, \times, \div\}$, nous générons d'abord l'ensemble de contraintes $\mathcal{E}[\![e_1^{\ell_1}]\!]\varrho$ et $\mathcal{E}[\![e_2^{\ell_2}]\!]\varrho$ pour les opérandes aux points de contrôle ℓ_1 et ℓ_2. Compte tenu de la règle (ADD), le résultat de l'addition de deux nombres est stocké dans le point de contrôle ℓ. Rappelons qu'une analyse dynamique pour la détermination de intervalles des valeurs est effectuée avant l'analyse de précision et donc $\mathsf{ufp}(\ell)$, $\mathsf{ufp}(\ell_1)$ et $\mathsf{ufp}(\ell_2)$ sont connus au moment de la génération des contraintes. Les règles (SUB) pour la soustraction, (MULT) pour la multiplication et Rule(DIV)) pour la division sont obtenues de manière similaire au cas de l'addition.

Pour les fonctions élémentaires telles que logarithme, exponentielle, les fonctions hyperboliques et trigonométriques regroupées dans Rule (MATH), nous considèrons que chaque fonction élémentaire introduit une perte de précision de φ bits, où $\varphi \in \mathbb{N}$ est un paramètre de l'analyse.

Les règles de commandes sont assez classiques, on utilise des points de contrôle pour distinguer de nombreuses affectations d'une même variable et aussi pour implémenter

des jointures dans des conditions et des boucles. Étant donné une commande c et un environnement ϱ, $\mathcal{C}[c]\ \varrho$ renvoie un couple (C, ϱ') constitué d'un ensemble C de contraintes et d'un nouvel environnement ϱ'. La fonction $\mathcal{C}$ est définie par induction sur la structure des commandes dans les figures 11.4 et 11.5. Pour les conditions, nous générons les contraintes pour les branches `then` et `else` ainsi que des contraintes supplémentaires pour joindre les résultats des deux branches. Pour les boucles, nous relions le nombre de bits significatifs à la fin du `body` au `nsb` des mêmes variables et au début de la boucle.

Finalement, la solution optimale calculée par un solveur de programmation linéaire (LP) classique (nous utilisons `GLPK` [Mak] en pratique) donne les types de données optimisés qui satisfont l'exigence de précision de l'utilisateur et en plus, dans un temps polynomial. Par conséquent, notre réglage ne dépend d'aucune arithmétique particulière. Cependant, nous devons dire que nous avons sur-approximé la propagation du bit de retenue tout au long des calculs dans cette formulation ILP.

11.2.3 Itération sur les Politiques (PI)

La troisième contribution consiste à introduire un deuxième jeu d'équations sémantiques qui permettent d'affiner encore plus la précision en étant moins pessimiste sur la propagation des bits de retenues dans les opérations arithmétiques. Par conséquent, le problème ne se réduit plus à un problème ILP. Ensuite, nous avons utilisé la méthode d'itération sur les politiques pour trouver plus efficacement la solution. Nous montrons dans la Figure 11.5 les nouvelles règles que nous ajoutons au système global de contraintes dans lequel la seule différence est d'activer la fonction optimisée ξ au lieu de sa sur-approximation que nous avons évoqué dans la Figure 11.4. Pour calculer l'ulp des erreurs sur les opérandes, nous devons estimer le nombre de bits de l'erreur $\mathsf{nsb_e}$ pour chaque opérande sur lequel sont basées toutes les règles de la Figure 11.5.

En suivant cette méthode, la fonction $\mathcal{E}'[e]\ \varrho$ génére le nouvel ensemble de contraintes pour une expression $e \in Expr$ dans l'environnement ϱ. Pour la règle $(\textsc{Const}')$, le nombre de bits significatifs de l'erreur $\mathsf{nsb_e} = 0$ alors que nous imposons que le $\mathsf{nsb_e}$ d'une variable x au point de contrôle ℓ est inférieur à la dernière affectation de $\mathsf{nsb_e}$ dans $\varrho(x)$ comme indiqué dans la règle $(\textsc{Id}')$ de la Figure 11.5. En considérant la règle $(\textsc{Add}')$, nous commençons par générer le nouvel ensemble de contraintes $\mathcal{E}'[e_1^{\ell_1}]\varrho$ et $\mathcal{E}'[e_2^{\ell_2}]\varrho$ sur les opérandes aux points de contrôle ℓ_1 et ℓ_2. Ensuite, nous avons besoin de $\mathsf{nsb_e}(\ell) \geqslant \mathsf{nsb_e}(\ell_1)$ et $\mathsf{nsb_e}(\ell) \geqslant \mathsf{nsb_e}(\ell_2)$ où le résultat de l'addition est stocké au point de contrôle ℓ.

Pour la règle $(\textsc{Sqrt}')$, nous générons les contraintes sur l'expression $\mathcal{E}'[e_1^{\ell_1}]\varrho$ et nous demandons que $\mathsf{nsb_e}$ du résultat stocké au point de contrôle ℓ est supérieur au $\mathsf{nsb_e}$ de l'expression au point de contrôle ℓ_1. Pour la règle $(\textsc{Math}')$, nous supposons que $\mathsf{nsb_e}(\ell)$ n'a pas de borne supérieure.

Concernant les commandes, nous définissons l'ensemble $\mathcal{C}'[c]\ \varrho$ qui a la même fonction que $\mathcal{C}$ défini dans la Figure 11.4. Le raisonnement sur les commandes reste également similaire sauf que cette fois nous raisonnons sur le nombre de bits des erreurs $\mathsf{nsb_e}$. La seule

$$\mathcal{E}[c\#\mathtt{p}^{\ell}]\varrho = \varnothing \quad (\textsc{Const}) \qquad \mathcal{E}[x^{\ell}]\varrho = \{\mathsf{nsb}(\varrho(x)) \geqslant \mathsf{nsb}(\ell)\} \quad (\textsc{Id})$$

$$\begin{array}{c}\mathcal{E}[e_1^{\ell_1} +^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}[e_1^{\ell_1}]\varrho \ \cup\ \mathcal{E}[e_2^{\ell_2}]\varrho \\ \cup \\ \{\mathsf{nsb}(\ell_1) \geqslant \mathsf{nsb}(\ell) + \mathsf{ufp}(\ell_1) - \mathsf{ufp}(\ell) + \xi(\ell)(\ell_1,\ell_2), \\ \mathsf{nsb}(\ell_2) \geqslant \mathsf{nsb}(\ell) + \mathsf{ufp}(\ell_2) - \mathsf{ufp}(\ell) + \xi(\ell)(\ell_1,\ell_2)\}\end{array} \quad (\textsc{Add})$$

$$\begin{array}{c}\mathcal{E}[e_1^{\ell_1} -^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}[e_1^{\ell_1}]\varrho \ \cup\ \mathcal{E}[e_2^{\ell_2}]\varrho \\ \cup \\ \{\mathsf{nsb}(\ell_1) \geqslant \mathsf{nsb}(\ell) + \mathsf{ufp}(\ell_1) - \mathsf{ufp}(\ell) + \xi(\ell)(\ell_1,\ell_2), \\ \mathsf{nsb}(\ell_2) \geqslant \mathsf{nsb}(\ell) + \mathsf{ufp}(\ell_2) - \mathsf{ufp}(\ell) + \xi(\ell)(\ell_1,\ell_2)\}\end{array} \quad (\textsc{Sub})$$

$$\begin{array}{c}\mathcal{E}[e_1^{\ell_1} \times^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}[e_1^{\ell_1}]\varrho \ \cup\ \mathcal{E}[e_2^{\ell_2}]\varrho \\ \cup \\ \{\mathsf{nsb}(\ell_1) \geqslant \mathsf{nsb}(\ell) + \xi(\ell)(\ell_1,\ell_2) - 1,\ \ \mathsf{nsb}(\ell_2) \geqslant \mathsf{nsb}(\ell) + \xi(\ell)(\ell_1,\ell_2) - 1\}\end{array} \quad (\textsc{Mult})$$

$$\begin{array}{c}\mathcal{E}[e_1^{\ell_1} \div^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}[e_1^{\ell_1}]\varrho \ \cup\ \mathcal{E}[e_2^{\ell_2}]\varrho \\ \cup \\ \{\mathsf{nsb}(\ell_1) \geqslant \mathsf{nsb}(\ell) + \xi(\ell)(\ell_1,\ell_2) - 1,\ \ \mathsf{nsb}(\ell_2) \geqslant \mathsf{nsb}(\ell) + \xi(\ell)(\ell_1,\ell_2) - 1\}\end{array} \quad (\textsc{Div})$$

$$\mathcal{E}\left[\sqrt{e^{\ell_1}}^{\ell}\right]\varrho = \mathcal{E}[e_1^{\ell_1}]\varrho \ \cup\ \{\mathsf{nsb}(\ell_1) \geqslant \mathsf{nsb}(\ell)\} \quad (\textsc{Sqrt})$$

$$\mathcal{E}\left[\phi\left(e^{\ell_1}\right)^{\ell}\right]\varrho = \mathcal{E}[e_1^{\ell_1}]\varrho \ \cup\ \{\mathsf{nsb}(\ell_1) \geqslant \mathsf{nsb}(\ell) + \varphi\} \text{ with } \phi \in \{\sin, \cos, \tan, \log, \ldots\} \quad (\textsc{Math})$$

$$\mathcal{C}\left[x :=^{\ell} e^{\ell_1}\right]\varrho = \left(C, \varrho\,[x \mapsto \ell]\right) \text{ where } C = \mathcal{E}[e_1^{\ell_1}]\varrho \cup \{\mathsf{nsb}(\ell_1) \geqslant \mathsf{nsb}(\ell)\} \quad (\textsc{Assign})$$

$$\begin{array}{c}\mathcal{C}\left[c_1^{\ell_1}; c_2^{\ell_2}\right]\varrho = (C_1 \cup C_2, \varrho_2) \\ \text{where } (C_1,\varrho_1) = \mathcal{C}\left[c_1^{\ell_1}\right]\varrho \text{ and } (C_2,\varrho_2) = \mathcal{C}\left[c_2^{\ell_2}\right]\varrho_1\end{array} \quad (\textsc{Seq})$$

$$\mathcal{C}[\mathtt{if}^{\ell}\ e^{\ell_0}\ \mathtt{then}\ c^{\ell_1}\ \mathtt{else}\ c^{\ell_2}]\ \varrho = (C_1 \cup C_2 \cup C_3, \varrho')$$
$$\text{where} \left| \begin{array}{l} \forall x \in \mathrm{Id},\ \varrho'(x) = \ell,\ (C_1,\varrho_1) = \mathcal{C}[c_1^{\ell_1}]\ \varrho,\ (C_2,\varrho_2) = \mathcal{C}[c_2^{\ell_2}]\ \varrho, \\ C_3 = \bigcup\limits_{x\in\mathrm{Id}} \{\mathsf{nsb}(\varrho_1(x)) \geqslant \mathsf{nsb}(\ell),\ \mathsf{nsb}(\varrho_2(x)) \geqslant \mathsf{nsb}(\ell)\} \end{array} \right. \quad (\textsc{Cond})$$

$$\mathcal{C}[\mathtt{while}^{\ell}\ e^{\ell_0}\ \mathtt{do}\ c^{\ell_1}]\ \varrho = (C_1 \cup C_2, \varrho')$$
$$\text{where} \left| \begin{array}{l} \forall x \in \mathrm{Id},\ \varrho'(x) = \ell,\ (C_1,\varrho_1) = \mathcal{C}[c_1^{\ell_1}]\ \varrho' \\ C_2 = \bigcup\limits_{x\in\mathrm{Id}} \{\mathsf{nsb}(\varrho(x)) \geqslant \mathsf{nsb}(\ell),\ \mathsf{nsb}(\varrho_1(x)) \geqslant \mathsf{nsb}(\ell)\} \end{array} \right. \quad (\textsc{While})$$

$$\mathcal{C}[\mathtt{require_nsb}(x,\mathtt{p})^{\ell}]\varrho = \{\mathsf{nsb}(\varrho(x)) \geqslant \mathtt{p}\} \quad (\textsc{Req})$$

$$\xi(\ell)(\ell_1,\ell_2) = 1$$

FIGURE 11.4 : Contraintes ILP avec une propagation pessimiste du bits de retenues $\xi = 1$.

différence est dans la règle (REQ′) où l'ensemble de contraintes est vide. Nous rappelons que les contraintes de la Figure 11.5 s'ajoutent aux anciennes contraintes de la Figure 11.4 et sont envoyées au solveur linéaire GLPK [Mak].

11.3 L'outil POP

11.3.1 Implémentation

POP a été développé en JAVA. Les différents éléments du langage impératif présentés dans la Figure 11.3 sont représentés sous forme de packages regroupant ses différentes classes. Nous illustrons la hiérarchie principale de notre outil dans la Figure 11.6 et elle est définie comme suit :

- **Parser :** Pour analyser les codes sources d'entrée, POP utilise l'outil ANTLR v4.7.1 (ANother Tool for Language Recognition)[4]. A partir d'une grammaire, ANTLR est capable de générer un analyseur qui peut construire les arbres syntaxiques des programmes.
- **Analyse dynamique pour les valeurs des variables :** Elle consiste à lancer l'exécution du programme un certain nombre de fois afin de déterminer dynamiquement les intervalles de variables. Plus précisément, ce que nous utilisons dans le réglage est la quantité ufp des valeurs des variables du programme.
- **Génération de contraintes :** C'est la base de notre approche statique pour le réglage de précision numérique. Nous rappellons que deux types de système de contraintes sont générés. Le premier système modélise la propagation des erreurs numériques pour la méthode basée sur SMT dans POP(SMT). Le deuxième système de contraintes est généré pour la méthode basée sur ILP dans POP(ILP). Ces deux méthodes sont basées sur les mêmes fonctions de transfert.
- **Résolution des contraintes :** Pour trouver une solution aux contraintes, la méthode basée sur SMT appelle un solveur SMT. Généralement, les solveurs SMT combinent le raisonnement SAT avec des solveurs théoriques spécialisés soit pour trouver une solution réalisable à un ensemble de contraintes, soit pour prouver qu'une telle solution n'existe pas. La méthode basée sur ILP utilise un solveur LP pour résoudre les contraintes. Les solveurs LP proviennent de la tradition de l'optimisation et sont conçus pour trouver des solutions réalisables qui sont optimales par rapport à une fonction d'optimisation.
- **Génération du code :** À la fin de l'analyse, POP génère une version optimisée du programme source d'entrée annotée avec les nouveaux nsb à chaque point de contrôle. Nous rappelons que notre technique est indépendante d'une arithmétique particulière

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

$$\mathcal{E}'[c\#\mathtt{p}^{\ell}]\varrho = \{\mathsf{nsb_e}(\ell) = 0\} \quad (\textsc{Const}') \qquad \mathcal{E}'[x^{\ell}]\varrho = \{\mathsf{nsb_e}(\varrho(x)) \geqslant \mathsf{nsb_e}(\ell)\} \quad (\textsc{Id}')$$

$$\mathcal{E}'[e_1^{\ell_1} +^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}'[e_1^{\ell_1}]\varrho \ \cup \ \mathcal{E}'[e_2^{\ell_2}]\varrho \quad (\textsc{Add}')$$
$$\cup$$
$$\left\{ \begin{array}{c} \mathsf{nsb_e}(\ell) \geqslant \mathsf{nsb_e}(\ell_1),\ \mathsf{nsb_e}(\ell) \geqslant \mathsf{nsb_e}(\ell_2), \\ \mathsf{nsb_e}(\ell) \geqslant \mathsf{ufp}(\ell_1) - \mathsf{ufp}(\ell_2) + \mathsf{nsb}(\ell_2) - \mathsf{nsb}(\ell_1) + \mathsf{nsb_e}(\ell_2) + \xi(\ell)(\ell_1,\ell_2), \\ \mathsf{nsb_e}(\ell) \geqslant \mathsf{ufp}(\ell_2) - \mathsf{ufp}(\ell_1) + \mathsf{nsb}(\ell_1) - \mathsf{nsb}(\ell_2) + \mathsf{nsb_e}(\ell_1) + \xi(\ell)(\ell_1,\ell_2) \end{array} \right\}$$

$$\mathcal{E}'[e_1^{\ell_1} -^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}'[e_1^{\ell_1}]\varrho \ \cup \ \mathcal{E}'[e_2^{\ell_2}]\varrho \quad (\textsc{Sub}')$$
$$\cup$$
$$\left\{ \begin{array}{c} \mathsf{nsb_e}(\ell) \geqslant \mathsf{nsb_e}(\ell_1),\ \mathsf{nsb_e}(\ell) \geqslant \mathsf{nsb_e}(\ell_2), \\ \mathsf{nsb_e}(\ell) \geqslant \mathsf{ufp}(\ell_1) - \mathsf{ufp}(\ell_2) + \mathsf{nsb}(\ell_2) - \mathsf{nsb}(\ell_1) + \mathsf{nsb_e}(\ell_2) + \xi(\ell)(\ell_1,\ell_2), \\ \mathsf{nsb_e}(\ell) \geqslant \mathsf{ufp}(\ell_2) - \mathsf{ufp}(\ell_1) + \mathsf{nsb}(\ell_1) - \mathsf{nsb}(\ell_2) + \mathsf{nsb_e}(\ell_1) + \xi(\ell)(\ell_1,\ell_2) \end{array} \right\}$$

$$\mathcal{E}'[e_1^{\ell_1} \times^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}'[e_1^{\ell_1}]\varrho \ \cup \ \mathcal{E}'[e_2^{\ell_2}]\varrho \quad (\textsc{Mult}')$$
$$\cup$$
$$\left\{ \ \mathsf{nsb_e}(\ell) \geqslant \mathsf{nsb}(\ell_1) + \mathsf{nsb_e}(\ell_1) + \mathsf{nsb_e}(\ell_2) - 2,\ \mathsf{nsb_e}(\ell) \geqslant \mathsf{nsb}(\ell_2) + \mathsf{nsb_e}(\ell_2) + \mathsf{nsb_e}(\ell_1) - 2 \ \right\}$$

$$\mathcal{E}'[e_1^{\ell_1} \div^{\ell} e_2^{\ell_2}]\varrho = \mathcal{E}'[e_1^{\ell_1}]\varrho \ \cup \ \mathcal{E}'[e_2^{\ell_2}]\varrho \quad (\textsc{Div}')$$
$$\cup$$
$$\left\{ \ \mathsf{nsb_e}(\ell) \geqslant \mathsf{nsb}(\ell_1) + \mathsf{nsb_e}(\ell_1) + \mathsf{nsb_e}(\ell_2) - 2,\ \mathsf{nsb_e}(\ell) \geqslant \mathsf{nsb}(\ell_2) + \mathsf{nsb_e}(\ell_2) + \mathsf{nsb_e}(\ell_1) - 2 \ \right\}$$

$$\mathcal{E}'\left[\sqrt{e^{\ell_1}}^{\ell}\right]\varrho = \mathcal{E}'[e_1^{\ell_1}]\varrho \ \cup \ \{\mathsf{nsb_e}(\ell) \geqslant \mathsf{nsb_e}(\ell_1)\} \quad (\textsc{Sqrt}')$$

$$\mathcal{E}'\left[\phi\left(e^{\ell_1}\right)^{\ell}\right]\varrho = \mathcal{E}'[e_1^{\ell_1}]\varrho \ \cup \ \{\mathsf{nsb_e}(\ell) \geqslant +\infty\} \text{ with } \phi \in \{\sin, \cos, \tan, \log, \ldots\} \quad (\textsc{Math}')$$

$$\mathcal{C}'\left[x :=^{\ell} e^{\ell_1}\right]\varrho = \left(C, \varrho\,[x \mapsto \ell]\,\right) \text{ where } C = \mathcal{E}'[e_1^{\ell_1}]\varrho \cup \{\mathsf{nsb_e}(\ell_1) \geqslant \mathsf{nsb_e}(\ell)\} \quad (\textsc{Assign}')$$

$$\mathcal{C}'\left[c_1^{\ell_1}; c_2^{\ell_2}\right]\varrho = \left(C_1 \cup C_2, \varrho_2\right) \text{ with } \left(C_1, \varrho_1\right) = \mathcal{C}'\left[c_1^{\ell_1}\right]\varrho \text{ and } \left(C_2, \varrho_2\right) = \mathcal{C}'\left[c_2^{\ell_2}\right]\varrho_1 \quad (\textsc{Seq}')$$

$$\mathcal{C}'[\mathtt{if}^{\ell}\ e^{\ell_0}\ \mathtt{then}\ c^{\ell_1}\ \mathtt{else}\ c^{\ell_2}]\ \varrho = (C_1 \cup C_2 \cup C_3, \varrho')$$
$$\text{where} \left| \begin{array}{l} \forall x \in \mathrm{Id},\ \varrho'(x) = \ell,\ (C_1, \varrho_1) = \mathcal{C}'[c_1^{\ell_1}]\ \varrho,\ (C_2, \varrho_2) = \mathcal{C}'[c_2^{\ell_2}]\ \varrho, \\ C_3 = \bigcup\limits_{x \in \mathrm{Id}} \{\mathsf{nsb_e}(\varrho_1(x)) \geqslant \mathsf{nsb_e}(\ell),\ \mathsf{nsb_e}(\varrho_2(x)) \geqslant \mathsf{nsb_e}(\ell)\} \end{array} \right. \quad (\textsc{Cond}')$$

$$\mathcal{C}'[\mathtt{while}^{\ell}\ e^{\ell_0}\ \mathtt{do}\ c^{\ell_1}]\ \varrho = (C_1 \cup C_2, \varrho')$$
$$\text{where} \left| \begin{array}{l} \forall x \in \mathrm{Id},\ \varrho'(x) = \ell,\ (C_1, \varrho_1) = \mathcal{C}'[c_1^{\ell_1}]\ \varrho' \\ C_2 = \bigcup\limits_{x \in \mathrm{Id}} \{\mathsf{nsb_e}(\varrho(x)) \geqslant \mathsf{nsb_e}(\ell),\ \mathsf{nsb_e}(\varrho_1(x)) \geqslant \mathsf{nsb_e}(\ell)\} \end{array} \right. \quad (\textsc{While}')$$

$$\mathcal{C}'[\mathtt{require_nsb}(x, \mathtt{p})^{\ell}]\varrho = \varnothing \quad (\textsc{Req}')$$

$$\xi(\ell)(\ell_1,\ell_2) = \min \left(\begin{array}{l} \max\left(\mathsf{ufp}(\ell_2) - \mathsf{ufp}(\ell_1) + \mathsf{nsb}(\ell_1) - \mathsf{nsb}(\ell_2) - \mathsf{nsb_e}(\ell_2), 0\right), \\ \max\left(\mathsf{ufp}(\ell_1) - \mathsf{ufp}(\ell_2) + \mathsf{nsb}(\ell_2) - \mathsf{nsb}(\ell_1) - \mathsf{nsb_e}(\ell_1), 0\right), 1 \end{array} \right)$$

FIGURE 11.5 : Contraintes de la méthode d'itératon sur les politiques avec une nouvelle formulation du bit de retenue.

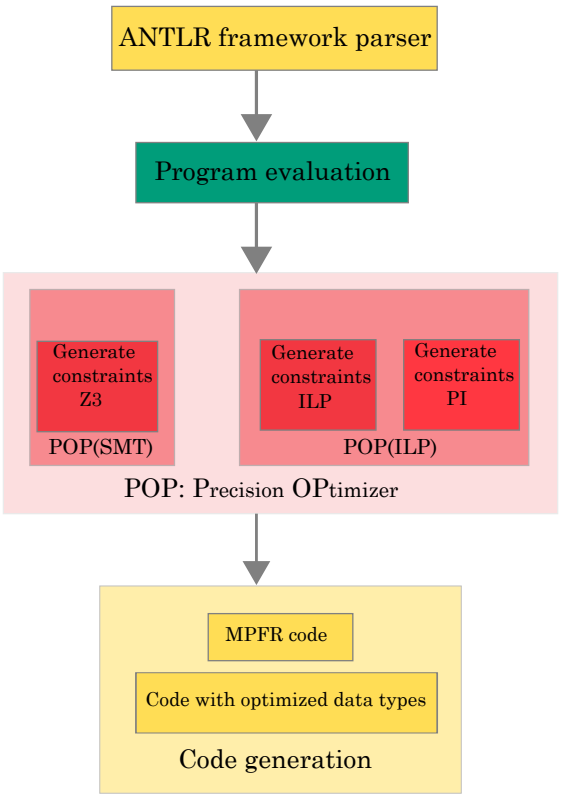


FIGURE 11.6 : L'architecture de notre outil POP.

et que les formats optimisés sont donnés en nombre de bits (*bit-level*). Cependant, si nous voulons ces précisions en mode IEEE754, le nsb obtenue en nombre de bits est approximé par le nombre supérieur de bits correspondant à un format IEEE754. Par exemple, si $\texttt{nsb}(x) = 18$ bits, alors x est réglé sur la simple précision FP32. De plus, POP est capable de générer une version python MPFR [FHL+07] du programme d'entrée. Le but à partir des codes MPFR est de mesurer la différence entre les deux programmes (original et optimisé) et de tracer la courbe de la différence en fonction de l'erreur théorique donnée par l'utilisateur.

11.3.2 Réglage de la Précision et Internet des Objets (IoT)

Le réglage de précision a déjà des applications dans de nombreux domaines et, dans cette section, nous résumons son utilité pour les applications IoT. Nous nous sommes concentrés sur l'expérimentation de POP(SMT) sur deux exemples représentatifs issus du domaine de l'IoT et sur la recherche d'un compromis entre la précision et l'énergie. Guidés par des demandes industrielles [IMN17], nous avons pris comme exemple le code d'un accéléromètre pour convertir la pression en mouvement. Deuxièmement, nous avons évalué POP(SMT) sur un exemple nettement plus complexe, un podomètre basé sur un accéléromètre pour les applications embarqués [MSGK14].

Détection d'angle d'inclinaison par un accéléromètre

Un accéléromètre est un capteur capable de mesurer, en trois dimensions, les accélérations linéaires d'un objet ainsi que les vibrations [BM19]. En pratique, il existe des accéléromètres dans de nombreux objets du quotidien, tels que les smartphones, les voitures, les montres de sport et autres appareils. Par exemple, l'accéléromètre d'un téléphone est capable de vous donner l'orientation du téléphone mais aussi, comme son nom l'indique, l'accélération subie par le téléphone.

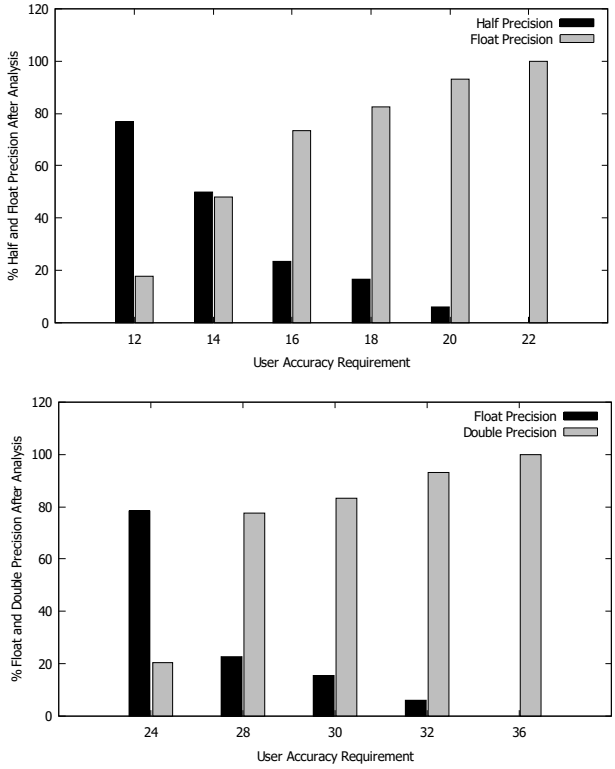


FIGURE 11.7 : Efficacité de `POP(SMT)` sur l'application de l'accéléromètre. En haut : le pourcentage de FP16 et FP32 pour différentes assertions de pécision de l'utilisateur. En bas : Le pourcentage des variables en FP32 et FP64 pour nsb $= 24, 28, 30, 32$ et 36 bits.

Nous visons à partir de l'expérimentation de l'accéléromètre pour mesurer l'utilité de notre analyse et comment `POP(SMT)` est capable d'optimiser la précision des variables du programme. Pour cet exemple, `POP(SMT)` génère 1179 variables et 1767 contraintes qui sont très gérables par le solveur `Z3`. En effet, il ne faut que 0.64 secondes à `POP(SMT)` pour réaliser l'analyse de la précision, y compris le temps pour la génération des contraintes et les appels au solveur `Z3`. En supposant que toutes les variables du programme original (avant analyse) sont en double précision (FP64), `POP(SMT)` a réussi à optimiser les précisions des variables en FP16 (half précision) et FP32 (simple précision) comme indiqué dans la partie supérieure de la Figure 11.7. Par exemple, pour nsb $= 20$ bits, le pourcentage des variables passées en FP32 est important par rapport aux variables passées en FP16 : 93, 13%

pour la simple précision et seulement 5,88% pour la half précision (FP16). Aussi, pour nsb $>$ 22 bits, POP(SMT) gère correctement l'approche statique de réglage de précision en trouvant un compromis entre les variables passéees en simple et en double précision.

Comme nous le montrons dans la partie inférieure de la Figure 11.7, pour nsb $=$ 30 bits et nsb $=$ 32 bits, la précision mixte entre la FP32 et FP64 est obtenue. Aussi, nous pouvons dire que, pour nsb $=$ 24 bits, il y a autant de variables en FP32 que en FP64. De plus, nous remarquons que pour un nsb $=$ 36 bits toutes les variables restent en FP64 et donc trouver la précision minimale n'est possible que pour des précisions inférieures à 36 bits pour cet exemple.

Évaluation expérimentale du programme du podomètre

Le deuxième exemple consiste à analyser le code d'un podomètre qui implémente un algorithme de comptage de pas pour les applications embarqués. Cet exemple est nettement plus complexe que celui de l'accéléromètre en termes des lignes des codes et des structures utilisées. Dans l'expérimentation ci-dessous, nous changeons la taille des fenêtres de 160, 280 et 360 pour les variables x, y et z dont leurs valeurs sont donnéés par un accéleromètre. En effet, nous avons remarqué que le solveur Z3 prend plus de temps pour résoudre les contraintes même si le temps d'exécution pour trouver les nouveaux formats pour les variables du programme reste court. De plus, POP(SMT) (pareil que pour POP(ILP)) étant un outil d'analyse statique admettant des intervalles pour ses entrées, la Figure 11.8 montre comment POP(SMT) se comporte s'il n'y a pas des variables scalaires mais des intervalles. En pratique, nous avons pris des intervalles autour des valeurs moyennes de x, y et z afin de nous baser sur un ensemble d'exécutions au lieu d'une seule. Les histogrammes de la figure 11.8 montrent que POP(SMT) a réussi à optimiser le nombre initial de bits pour le code du podomètre original, en commençant par 238977 bits en nombre total des bits, pour une précision égale à 36 bits.

11.4 Résultats Expérimentaux

11.4.1 Évaluation des Performances de POP(SMT) et POP(ILP)

Tout d'abord, nous avons évalué les performances de POP avec ces deux versions POP(SMT) and POP(ILP) suivant différentes manières. Dans un premier temps, nous avons comparé les deux versions POP(SMT) et POP(ILP) avec l'outil de l'état de l'art Precimonious [RNN+13] en termes du temps d'analyse, de rapidité et de qualité des solutions. Les résultats expérimentaux ont montré que POP(SMT) et POP(ILP) réussissaient à régler plus de programmes que Precimonious [RNN+13]. Pour la majorité des programmes évalués, nous allons montrer que notre outil renvoit de meilleurs résultats en précision mixte pour les différentes exigences de précision des utilisateurs.

La deuxième expérimentation concernait l'utilisation de deux fonctions de coût différentes au système global de contraintes de la version POP(SMT) afin d'optimiser les solutions

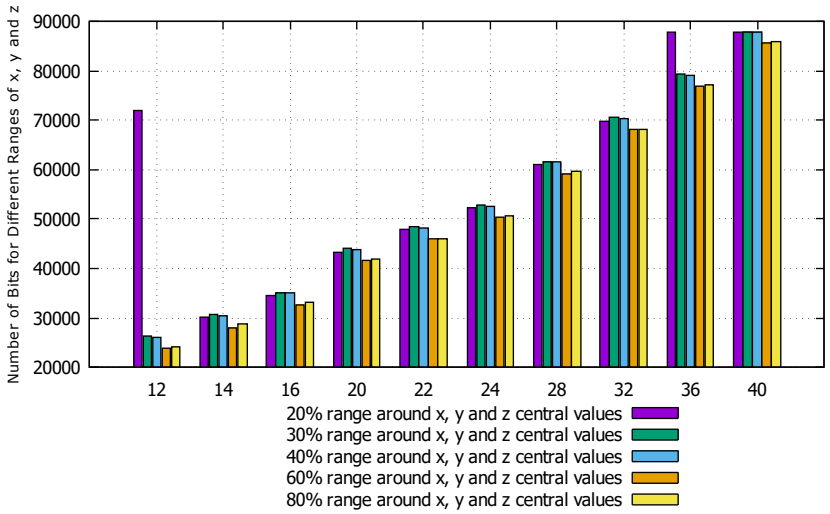


FIGURE 11.8 : Optimisation du nombre total des bits par `POP(SMT)` pour différents intervalles utilisées pour les variables x, y et z.

renvoyées par le solveur `Z3`. Ensuite, nous avons comparé les performances de chacune de ces fonctions sur différents programmes issus du calcul scientifique, du traitement du signal ou de l'IoT. Les résultats ont montré que cela peut influencer le nombre total de bits optimisés dans les programmes optimisés en obtenant différentes précisions lors de l'utilisation d'une fonction de coût au lieu d'une autre.

La troisième expérimentation, nous avons mesuré l'erreur entre les résultats exacts donnés par une exécution en précision multiple (en utilisant `MPFR` [FHL+07]) et les résultats de programmes optimisés par `POP(SMT)` et `POP(ILP)` et nous avons trouvé que les courbes d'erreurs mesurées sont toujours inférieures aux erreurs théoriques demandées par l'utilisateur pour les différents benchmarks.

Dans ce qui suit, nous allons montrer les évaluations que pour la version `POP(ILP)` qui est la version courrante de notre outil.

11.4.2 Comparaison avec l'Outil `Precimonious`

Le tableau 11.1 montre une comparaison entre la version de `POP(ILP)` combinant les formulations ILP et PI. Les résultats du réglage de précision mixte sont affichés pour les programmes **arclength**, **simpson**, **rotation** et **accelerometer**. Il faut mentionner que certains exemples utilisés par `Precimonious` [RNN+13] ne peuvent pas être analysés par `POP(ILP)` pour plusieurs raisons (appels à des bibliothèques externes ou utilisation de formes syntaxiques non encore implémentées dans notre outil). Réciproquement, `Precimonious` ne réussit pas à optimiser certains exemples gérés par `POP(ILP)`, par exemple le programme **lowPassFilter**.

Puisque `POP` (dans ses deux versions) et `Precimonious` mettent en œuvre deux techniques différentes, nous avons ajusté les critères de comparaison suivant plusieurs aspects. Tout d'abord, nous mentionnons que `POP` optimise beaucoup plus de variables

Program	Tool	#Bits saved - Time in seconds			
		Threshold 10^{-4}	Threshold 10^{-6}	Threshold 10^{-8}	Threshold 10^{-10}
arclength	POP(ILP) (28)	**2464b.** - 1.8s.	**2144b.** - 1.5s.	**1792b.** - 1.7s.	**1728b.** - 1.8s.
	POP(SMT) (22)	1488b. - 4.7s.	1472b. - 3.04s.	864b. - 3.09s.	384b. - 2.9s.
	Precimonious (9)	576b. - 146.4s.	576b. - 156.0s.	576b. - 145.8s.	576b. - 215.0s.
simpson	POP(ILP) (14)	**1344b.** - 0.4s.	**1152b.** - 0.5s.	**896b.** - 0.4s.	**896b.** - 0.4s.
	POP(SMT) (11)	896b. - 2.9s.	896b. - 1.9s.	704b. - 1.7s.	704b. - 1.8s.
	Precimonious (10)	704b. - 208.1s.	704b. - 213.7s.	704b. - 207.5s.	704b. - 200.3s.
rotation	POP(ILP) (25)	**2624b.** - 0.47s.	2464b. - 0.47s.	2048b. - 0.54s.	1600b. - 0.48s.
	POP(SMT) (22)	1584b. - 1.85s.	2208b. - 1.7s.	1776b. - 1.6s.	1600b. - 1.7s.
	Precimonious (27)	2400b. - 9.53s.	**2592b.** - 12.2s.	**2464b.** - 10.7s.	**2464b.** - 7.4s.
accel.	POP(ILP) (18)	**1776b.** - 1.05s.	**1728b.** - 1.05s.	**1248b.** - 1.04s.	**1152b.** - 1.03s.
	POP(SMT) (15)	1488b. - 2.6s.	1440b. - 2.6s.	1056 - 2.4s.	960b. - 2.4s.
	Precimonious (0)	-	-	-	-

TABLE 11.1 : Comparaison entre `POP(ILP)`, `POP(SMT)` et `Precimonious` : nombre de bits économisés par chaque outil et temps d'analyse en secondes.

que `Precimonious`. Bien que cela désavantage notre outil, nous ne considérons dans les expériences de Table 11.1 que les variables optimisées par `Precimonious` pour estimer la qualité de l'optimisation. Deuxièmement, notons que les seuils d'erreur s'expriment en base 2 pour `POP` et en base 10 pour `Precimonious`. Pour la pertinence des comparaisons, tous les seuils d'erreurs sont exprimés en base 10 dans cette expérimentation.

Dans le tableau 11.1, nous indiquons en gras l'outil qui présente les meilleurs résultats pour chaque seuil d'erreur et chaque programme. À partir du programme **arclength**, `POP(ILP)` affiche de meilleurs résultats que les autres outils en optimisant 28 variables. Pour un seuil d'erreur de 10^{-4}, 2464 bits sont économisés par `POP(ILP)` en 1.8 secondes alors que `POP(SMT)` n'a économisé que 1488 bits en plus de temps (11 secondes). `Precimonious` était l'outil le plus lent sur cet exemple avec plus de 2 minutes en économisant que 576 bits pour seulement 9 de variables optimisées. Pour le programme **simpson**, `POP(ILP)` fait également mieux que les deux autres outils. Cependant, pour le programme **rotation**, `POP(ILP)` n'économise plus de bits que pour une erreur de 10^{-4} tandis que `Precimonious` économise plus de bits pour ce programme pour le reste des seuils d'erreur. Enfin, `Precimonious` ne réussit pas à optimiser le programme **accelerometer** (0 variables) au moment où `POP(ILP)` économise beaucoup plus de bits en seulement une seule seconde.

11.4.3 Génération du Code de Précision Multiple

Dans cette contribution, nous avons effectué notre analyse de réglage de précision sur le problème à N-corps [BM21b] (N=5) avec différents exigences de précision sur les variables du programme : 11, 18, 24, 34, 43 et 53 bits. Pour cela, nous avons testé l'efficacité de `POP(ILP)` en plusieurs manières. Pour cette expérimentation, nous avons généré le programme N-corps avec tous les calculs effectués sur 500 bits (en supposant que cela donne la solution exacte) et un code `MPFR` [FHL+07] avec les types de données optimisés renvoyés par `POP(ILP)`. Par exemple, comme on peut l'observer dans la Table 11.2, pour

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(ILP) 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(ILP) Time	38″	39″	41″	37″	37″	37″

TABLE 11.2 : Distances entre la position exacte (calculée avec 500 bits) et la position calculée avec n bits. Distances données pour chaque corps après 10 et 30 ans de simulation. Suivi du temps d'analyse de POP(ILP) et du temps d'exécution du code généré par MPFR.

nsb $= 11$ bits, la distance mesurée pour Jupiter est de l'ordre de 10^{-4} pour 10 années de simulation ce qui confirme l'utilité de notre analyse : résultats souhaitables qui respectent l'exigence de précision donnée par l'utilisateur (nsb) où la pire erreur est de 2^{-11} pour nsb = 11 bits. Les résultats sont également satisfaisants pour les planètes restantes. Pour une simulation de 10 et 30 ans, le temps d'exécution passé à mesurer les distances atteint au maximum 2 minutes 59 secondes pour un nsb = 53 bits. Concernant le temps d'analyse, POP(ILP) a pris aussi peu que 25 secondes pour pour trouver les nouveaux types de données pour la majorité des variables du programme pour un temps de simulation de 10 années et ne dépasse pas 41 secondes pour 30 années de simulation (nsb $= 24$) bits . Avec cette rapidité d'analyse, nous pensons que pour les gros codes, POP(ILP) atteint son

meilleur réglage en un temps minimal. Nous présentons, dans la Figure 11.1, des extraits de code qui mesurent la distance entre les deux planètes `Jupiter` et `Saturne`; Le code donné dans la partie gauche de la figure illustre le programme annoté par les différents points de contrôle. Le programme après les analyses de précision par `POP(ILP)` est donné dans la partie droite de la même figure.

```
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              [...]
```

Listing 11.1 : Gauche : programme source annoté avec les labels. Droite : Programme optimisé généré par `POP(ILP)`.

11.5 Synthèse de l'Existant

Les travaux présentés dans cette thèse ont abordé le problème du réglage de précision qui émerge comme une nouvelle tendance en HPC, en particulier, lorsque de nouvelles applications tolérantes aux erreurs sont envisagées, pour améliorer les mesures de performances telles que la consommation d'énergie et de la mémoire. Ce travail a conduit au développement de `POP` qui est un logiciel automatisé pour le réglage de la précision numérique. Basée sur une analyse statique, `POP` calcule le nombre minimal de bits nécessaires pour les variables et les résultats intermédiaires des programmes afin d'accomplir l'exigence de précision de l'utilisateur. Nous avons résumé dans ce chapitre les différentes techniques utilisées par `POP`[5] afin d'obtenir un réglage de précision rapide et efficace.

En guise de perspectives, nous envisageons de suivre diverses pistes de recherche pour des futurs travaux, notamment à court et à long terme, qui permettront de valoriser nos résultats. La première perspective concerne la scalabilité de notre outil `POP` qui génère un nombre de contraintes et de variables linéaire à la taille du programme analysé. Cependant, la seule limitation affronté est la taille du problème accepté par le solveur. Dans les travaux futurs, notre objectif est d'aborder le problème de scalabilité par au moins deux idées. La première idée consiste à étudier comment la même précision soit utilisée pour toutes les instructions d'une même partie du programme (par exemple : une expression arithmétique, une ligne de code, une fonction, une boucle, etc.). Cela réduira considérablement la taille de nos systèmes de contraintes. De plus, concernant les grands tableaux, nous étudierons comment incorporer des techniques de compression avec perte telles que la technique ZFP [FDH+20] et comment déterminer le taux de compression à l'aide de notre outil `POP`. La seconde idée consiste à explorer des solveurs LP commerciaux qui ne se limitent pas à la taille du problème ILP (comme le solveur `GLPK` [Mak] utilisé dans cette thèse) tels que `CPLEX` [Cpl09] et `Gurobi` [Gur21]. Avec cette solution, nous sommes sûrs que même si le problème ILP généré à partir du code source est important, le solveur est capable de renvoyer une solution optimale dans un peu de temps.

Nous nous intéressons aussi à court terme à étendre `POP` en adoptant un analyseur statique avec des domaines abstraits sophistiqués pour inférer des intervalles sûres sur les variables. L'idée consiste à utiliser l'une des méthodes d'analyse d'erreur actuelles tels que l'outil `SATIRE` [DBG+20] et `FPDetect` [DKB+20]. La seconde extension de `POP` sera liée au langage des programmes. Actuellement, notre outil analyse des programmes avec des boucles, des tableaux, des matrices, des conditions, des expressions arithmétiques, des fonctions trigonométriques, la fonction racine carrée, etc. Cependant, nous avons rencontré quelques difficultés lorsque nous avons comparé `POP` à d'autres outils comme `Precimonious` [RNN+13]. L'une de ces difficultés est l'incapacité de notre outil à analyser certains exemples qui contiennent des appels à des bibliothèques externes ou qui utilisent des formes syntaxiques que nous n'avons pas encore implémentées. Pour pallier à ce

[5] `POP` n'est pas encore publiquement distribué. Nous comptons de le distribuer au bout de quelques mois.

problème, nous allons étendre le langage du `POP` pour traiter d'autres structures telles que les fonctions à titre d'exemple.

De plus, nos efforts se concentrent actuellement sur l'évaluation de notre logiciel dans des domaines d'application différents. Nous pouvons commencer par la communauté `FPBench` qui propose sur leur page web[6] plus de 130 de benchmarks couvrant une variété de domaines. Néanmoins, nous souhaitons renforcer les collaborations au sein du laboratoire LAMPS puisqu'il relie plusieurs axes multidisciplinaires tels que la mécanique des contacts et des fluides, la physique statistique, l'optimisation, etc. Ces collaborations peuvent nous aider à réfléchir à des nouvelles études de cas qui pourraient aborder d'autres problèmes intéressants. À long terme, nous nous intéressons à générer du code pour l'arithmétique à virgule fixe grâce aux informations fournies par `POP` qui garantissent la précision souhaitée par l'utilisateur sur les résultats. Une autre direction sera la possibilité d'appliquer nos méthodes pour optimiser la précision des réseaux de neurones.

[6] https://fpbench.org/community.html

Bibliography

***

[ABM21] Assalé Adjé, Dorra Ben Khalifa, and Matthieu Martel. Fast and efficient bit-level precision tuning. In *Static Analysis - 28th International Symposium, SAS 2021, Chicago, Illinois*, Lecture Notes in Computer Science. Springer, 2021.

[Ach09] Tobias Achterberg. Scip: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[AGG12a] Assalé Adjé, Stéphane Gaubert, and Eric Goubault. Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. *Log. Methods Comput. Sci.*, 8(1), 2012.

[AGG12b] Assalé Adjé, Stéphane Gaubert, and Eric Goubault. Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. *Log. Methods Comput. Sci.*, 8(1), 2012.

[Aho10] Tom Ahola. Pedometer for running activity using accelerometer sensors on the wrist. *Medical Equipment Insights*, 2010.

[ANS08] ANSI/IEEE. *IEEE Standard for Binary Floating-point Arithmetic*, std 754-2008 edition, 2008.

[ASS17] Alexandra Angerd, Erik Sintorn, and Per Stenström. A framework for automated and controlled floating-point accuracy reduction in graphics applications on gpus. *ACM Trans. Archit. Code Optim.*, 14(4):46:1–46:25, 2017.

[Atk91] Kendall E Atkinson. An introduction to numerical analysis. 1989. *New York*, 528:38, 1991.

[Bai08] David Bailey. Resolving numerical anomalies in scientific computation. 03 2008.

[BBB+98] C. Batut, K. Belabas, D. Bernardi, H. Cohen, and M. Olivier. *User's Guide to PARI-GP*. Laboratoire A2X, Université Bordeaux I, France, 1998.

[BBD+09] Marc Baboulin, Alfredo Buttari, Jack J. Dongarra, Jakub Kurzak, Julie Langou, Julien Langou, Piotr Luszczek, and Stanimire Tomov. Accelerating scientific computations with mixed precision algorithms. *Comput. Phys. Commun.*, 180(12):2526–2533, 2009.

[BBN16] Pietro Belotti, Timo Berthold, and Kelligton Neves. Algorithms for discrete nonlinear optimization in fico xpress. In *2016 IEEE Sensor Array and Multichannel Signal Processing Workshop (SAM)*, pages 1–5, 2016.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, page 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[BEN04] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. lp_solve 5.5, open source (mixed-integer) linear programming system. Software, 2004.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[BII+20] Hugo Brunie, Costin Iancu, Khaled Z. Ibrahim, Philip Brisk, and Brandon Cook. Tuning floating-point precision using dynamic program information and temporal locality. In Christine Cuicchi, Irene Qualters, and William T. Kramer, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 50. IEEE/ACM, 2020.

[Bir67] Garrett Birkhoff. *Lattice Theory*. American Mathematical Society, Providence, 3rd edition, 1967.

[BM19] Dorra Ben Khalifa and Matthieu Martel. Precision tuning and internet of things. In *International Conference on Internet of Things, Embedded Systems and Communications, IINTEC 2019*, pages 80–85. IEEE, 2019.

[BM20] Dorra Ben Khalifa and Matthieu Martel. Precision tuning of an accelerometer-based pedometer algorithm for iot devices. In *International Conference on Internet of Things and Intelligence System, IOTAIS 2020*, pages 113–119. IEEE, 2020.

[BM21a] Dorra Ben Khalifa and Matthieu 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*, pages 69–78. IEEE, 2021.

[BM21b] Dorra Ben Khalifa and Matthieu Martel. A study of the floating-point tuning behaviour on the n-body problem. In Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Chiara Garau, Ivan Blecic, David Taniar, Bernady O. Apduhan, Ana Maria A. C. Rocha, Eufemia Tarantino, and Carmelo Maria Torre, editors, *Computational Science and Its Applications - ICCSA 2021 - 21st International Conference, Cagliari, Italy, September 13-16, 2021, Proceedings, Part V*, volume 12953 of *Lecture Notes in Computer Science*, pages 176–190. Springer, 2021.

[BMA19] Dorra Ben Khalifa, Matthieu Martel, and Assalé Adjé. POP: A tuning assistant for mixed-precision floating-point computations. In *Formal Techniques for Safety-Critical Systems - 7th International Workshop, FTSCS 2019*, volume 1165 of *Communications in Computer and Information Science*, pages 77–94. Springer, 2019.

[BPDT18] Heiko Becker, Pavel Panchekha, Eva Darulova, and Zachary Tatlock. Combining tools for optimization and analysis of floating-point computations. In Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik P. de Vink, editors, *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*, volume 10951 of *Lecture Notes in Computer Science*, pages 355–363. Springer, 2018.

[Bre78] Richard P. Brent. A fortran multiple-precision arithmetic package. *ACM Trans. Math. Softw.*, 4(1):57–70, 1978.

[BSC12] Olivier Bouissou, Yassamine Seladji, and Alexandre Chapoutot. Acceleration of the abstract fixpoint computation in numerical program analysis. *J. Symb. Comput.*, 47(12):1479–1511, 2012.

[CA20] Stefano Cherubin and Giovanni Agosta. Tools for reduced precision computation: A survey. *ACM Comput. Surv.*, 53(2), 2020.

[CAL+17] Stefano Cherubin, Giovanni Agosta, Imane Lasri, Erven Rohou, and Olivier Sentieys. Implications of reduced-precision computations in HPC: performance, energy and error. In Sanzio Bassini, Marco Danelutto, Patrizio Dazzi, Gerhard R. Joubert, and Frans J. Peters, editors, *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing, ParCo 2017, 12-15 September 2017, Bologna, Italy*, volume 32 of *Advances in Parallel Computing*, pages 297–306. IOS Press, 2017.

[CBB+17] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamaric. Rigorous floating-point mixed-precision tuning. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming*

Languages, POPL 2017, Paris, France, January 18-20, 2017, pages 300–315. ACM, 2017.

[CC77a] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[CC77b] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of recursive procedures. In Erich J. Neuhold, editor, *Formal Description of Programming Concepts: Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts, St. Andrews, NB, Canada, August 1-5, 1977*, pages 237–278. North-Holland, 1977.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[CCC+12] João M. P. Cardoso, Tiago Carvalho, José Gabriel F. Coutinho, Wayne Luk, Ricardo Nobre, Pedro C. Diniz, and Zlatko Petrov. LARA: an aspect-oriented programming language for embedded systems. In Robert Hirschfeld, Éric Tanter, Kevin J. Sullivan, and Richard P. Gabriel, editors, *Proceedings of the 11th International Conference on Aspect-oriented Software Development, AOSD 2012, Potsdam, Germany, March 25-30, 2012*, pages 179–190. ACM, 2012.

[CCC+20] Stefano Cherubin, Daniele Cattaneo, Michele Chiari, Antonio Di Bello, and Giovanni Agosta. TAFFO: tuning assistant for floating to fixed point optimization. *IEEE Embed. Syst. Lett.*, 12(1):5–8, 2020.

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[CGG+05] Alexandru Costan, Stephane Gaubert, Eric Goubault, Matthieu Martel, and Sylvie Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In *Computer Aided Verification, 17th International Conference, CAV 2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 462–475. Springer, 2005.

[CGL10] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity analysis of programs. In *POPL*, pages 57–70. ACM, 2010.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, page 93–107, Berlin, Heidelberg, 2013. Springer-Verlag.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96. ACM Press, 1978.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c - A software analysis perspective. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*, volume 7504 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2012.

[Cou78] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes*. 1978.

[Cpl09] IBM ILOG Cplex. V12. 1: User's manual for cplex. *International Business Machines Corporation*, 46(53):157, 2009.

[CWM94] S. Chaudhuri, R.A. Walker, and J.E. Mitchell. Analyzing and exploiting the structure of the constraints in the ilp approach to the scheduling problem. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):456–471, 1994.

[Dam16] Nasrine Damouche. *Improving the Numerical Accuracy of Floating-Point Programs with Automatic Code Transformation Methods. (Amélioration de la précision numérique de programmes basés sur l'arithmétique flottante par les méthodes de transformation automatique)*. PhD thesis, University of Perpignan, France, 2016.

[Dan63] G. Dantzig. Linear programming and extensions. 1963.

[Dan90] George B. Dantzig. *Origins of the Simplex Method*, page 141–151. Association for Computing Machinery, New York, NY, USA, 1990.

[DBG+20] Arnab Das, Ian Briggs, Ganesh Gopalakrishnan, Sriram Krishnamoorthy, and Pavel Panchekha. Scalable yet rigorous floating-point error analysis. In Christine Cuicchi, Irene Qualters, and William T. Kramer, editors, *Proceedings of the International Conference for High Performance Computing, Networking,*

Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020, page 51. IEEE/ACM, 2020.

[DC16] Sheng Di and Franck Cappello. Fast error-bounded lossy HPC data compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pages 730–739. IEEE Computer Society, 2016.

[DDBC+19] Marc Duranton, Koen De Bosschere, Bart Coppens, Christian Gamrat, Madeleine Gray, Harm Munk, Emre Ozer, Tullio Vardanega, and Olivier Zendra. *The HiPEAC Vision 2019*. HiPEAC CSA, 2019.

[dDLM11] Florent de Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using gappa. *IEEE Trans. Computers*, 60(2):242–253, 2011.

[DdOCP16] Christophe Denis, Pablo de Oliveira Castro, and Eric Petit. Verificarlo: Checking floating point accuracy through monte carlo arithmetic. In Paolo Montuschi, Michael J. Schulte, Javier Hormigo, Stuart F. Oberman, and Nathalie Revol, editors, *23nd IEEE Symposium on Computer Arithmetic, ARITH 2016, Silicon Valley, CA, USA, July 10-13, 2016*, pages 55–62. IEEE Computer Society, 2016.

[dDP11] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with flopoco. *IEEE Des. Test Comput.*, 28(4):18–27, 2011.

[Dem21] Nestor Demeure. *Compromis entre précision et performance dans le calcul haute performance*. PhD thesis, Université Paris-Saclay, January 2021.

[dFS04] Luiz Henrique de Figueiredo and Jorge Stolfi. Affine arithmetic: Concepts and applications. *Numer. Algorithms*, 37(1-4):147–158, 2004.

[DGY+74] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[DHS18] Eva Darulova, Einar Horn, and Saksham Sharma. Sound mixed-precision optimization with rewriting. In Chris Gill, Bruno Sinopoli, Xue Liu, and Paulo Tabuada, editors, *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2018, Porto, Portugal, April 11-13, 2018*, pages 208–219. IEEE Computer Society / ACM, 2018.

[Dik67] II Dikin. Iterative solution of problems of linear and quadratic programming. In *Doklady Akademii Nauk*, volume 174, pages 747–748. Russian Academy of Sciences, 1967.

[DIN+18] Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. Daisy - framework for analysis and optimization of numerical programs (tool paper). In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*, volume 10805 of *Lecture Notes in Computer Science*, pages 270–287. Springer, 2018.

[DK11] Eva Darulova and Viktor Kuncak. Trustworthy numerical computation in scala. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 325–344. ACM, 2011.

[DK14] Eva Darulova and Viktor Kuncak. Sound compilation of reals. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 235–248. ACM, 2014.

[DK17] Eva Darulova and Viktor Kuncak. Towards a compiler for reals. *ACM Trans. Program. Lang. Syst.*, 39(2):8:1–8:28, 2017.

[DKB+20] Arnab Das, Sriram Krishnamoorthy, Ian Briggs, Ganesh Gopalakrishnan, and Ramakrishna Tipireddy. Fpdetect: Efficient reasoning about stencil programs using selective direct evaluation. *ACM Trans. Archit. Code Optim.*, 17(3):19:1–19:27, 2020.

[DKMS13] Eva Darulova, Viktor Kuncak, Rupak Majumdar, and Indranil Saha. Synthesis of fixed-point programs. In Rolf Ernst and Oleg Sokolsky, editors, *Proceedings of the International Conference on Embedded Software, EMSOFT 2013, Montreal, QC, Canada, September 29 - Oct. 4, 2013*, pages 22:1–22:10. IEEE, 2013.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[DM18] Nasrine Damouche and Matthieu Martel. Mixed precision tuning with salsa. In Luis Gomes, Andreas Ahrens, César Benavente-Peces, and Mohammad S. Obaidat, editors, *Proceedings of the 8th International Joint Conference on Pervasive and Embedded Computing and Communication Systems, PECCS 2018, Porto, Portugal, July 29-30, 2018*, pages 185–194. SciTePress, 2018.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[DMB11] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 2011.

[DMC15] Nasrine Damouche, Matthieu Martel, and Alexandre Chapoutot. Transformation of a PID controller for numerical accuracy. *Electron. Notes Theor. Comput. Sci.*, 317:47–54, 2015.

[DMC17] Nasrine Damouche, Matthieu Martel, and Alexandre Chapoutot. Improving the numerical accuracy of programs by automatic transformation. *Int. J. Softw. Tools Technol. Transf.*, 19(4):427–448, 2017.

[DMP+16] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Jason Qiu, Alex Sanchez-Stern, and Zachary Tatlock. Toward a standard benchmark format and suite for floating-point analysis. 2016.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[DV19] Eva Darulova and Anastasia Volkova. Sound approximation of programs with elementary functions. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, volume 11562 of *Lecture Notes in Computer Science*, pages 174–183. Springer, 2019.

[DYSD14] Gaël Deest, Tomofumi Yuki, Olivier Sentieys, and Steven Derrien. Toward scalable source level accuracy analysis for floating-point to fixed-point conversion. In Yao-Wen Chang, editor, *The IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2014, San Jose, CA, USA, November 3-6, 2014*, pages 726–733. IEEE, 2014.

[FDH+20] Alyson Fox, James Diffenderfer, Jeffrey Hittinger, Geoffrey Sanders, and Peter Lindstrom. Stability analysis of inline ZFP compression for floating-point data in iterative methods. *SIAM J. Sci. Comput.*, 42(5):A2701–A2730, 2020.

[FHL+07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33, 2007.

[FL17] François Févotte and Bruno Lathuilière. Studying the numerical quality of an industrial computing code: A case study on code_aster. In Alessandro Abate and Sylvie Boldo, editors, *Numerical Software Verification - 10th International Workshop, NSV 2017, Heidelberg, Germany, July 22-23, 2017, Proceedings [collocated with CAV 2017]*, volume 10381 of *Lecture Notes in Computer Science*, pages 61–80. Springer, 2017.

[Gas85] Saul I. Gass. *Linear Programming: Methods and Applications (5th Ed.)*. McGraw-Hill, Inc., USA, 1985.

[GB20] Ruidong Gu and Michela Becchi. Gpu-fptuner: Mixed-precision auto-tuning for floating-point applications on GPU. In *27th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2020, Pune, India, December 16-19, 2020*, pages 294–304. IEEE, 2020.

[GC15] Xitong Gao and George A. Constantinides. Numerical program optimization for high-level synthesis. In George A. Constantinides and Deming Chen, editors, *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 22-24, 2015*, pages 210–213. ACM, 2015.

[GGP09] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. The zonotope abstract domain taylor1+. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 627–633. Springer, 2009.

[GGTZ07] Stephane Gaubert, Eric Goubault, Ankur Taly, and Sarah Zennou. Static analysis by policy iteration on relational domains. In Rocco De Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2007.

[GJP+19] Stef Graillat, Fabienne Jézéquel, Romain Picot, François Févotte, and Bruno Lathuilière. Auto-tuning for floating-point precision with discrete stochastic arithmetic. *J. Comput. Sci.*, 36, 2019.

[GK13] Myrdal Gardarsson and Kari Kjartan. Some theoretical and numerical aspects of the n-body problem, 2013. Student Paper.

[GKC13] Sicun Gao, Soonho Kong, and Edmund M. Clarke. dreal: An SMT solver for nonlinear theories over the reals. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 208–214. Springer, 2013.

[GMP02] Eric Goubault, Matthieu Martel, and Sylvie Putot. Asserting the precision of floating-point computations: A simple abstract interpreter. In Daniel Le Métayer, editor, *Programming Languages and Systems, 11th European Symposium*

on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings, volume 2305 of *Lecture Notes in Computer Science*, pages 209–212. Springer, 2002.

[GP11] Eric Goubault and Sylvie Putot. Static analysis of finite precision computations. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 2011.

[GPV12] Eric Goubault, Sylvie Putot, and Franck Védrine. Modular static analysis with zonotopes. In *Static Analysis Symposium*, volume 7460 of *Lecture Notes in Computer Science*, pages 24–40. Springer, 2012.

[GR18a] Hui Guo and Cindy 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*, pages 333–343. ACM, 2018.

[GR18b] Hui Guo and Cindy Rubio-González. Exploiting community structure for floating-point precision tuning. In Frank Tip and Eric Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 333–343. ACM, 2018.

[GS07a] Thomas Gawlitza and Helmut Seidl. Precise fixpoint computation through strategy iteration. In Rocco De Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 300–315. Springer, 2007.

[GS07b] Thomas Gawlitza and Helmut Seidl. Precise relational invariants through strategy iteration. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 23–40. Springer, 2007.

[GS10] Thomas Martin Gawlitza and Helmut Seidl. Computing relaxed abstract semantics w.r.t. quadratic zones precisely. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2010.

[GT15] Torbjrn Granlund and Gmp Development Team. *GNU MP 6.0 Multiple Precision Arithmetic Library*. Samurai Media Limited, London, GBR, 2015.

[Gur21] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2021.

[GY17] Gustafson and Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomput. Front. Innov.: Int. J.*, 4(2):71–86, 2017.

[Hal95] Tom R. Halfhill. The truth behind the Pentium bug: How often do the five empty cells in the Pentium's FPU lookup table spell miscalculation? 1995.

[HK66] A. J. Hoffman and R. M. Karp. On nonterminating stochastic games. *Manage. Sci.*, 12(5):359–370, 1966.

[HMWA17] Nhut-Minh Ho, Elavarasi Manogaran, Weng-Fai Wong, and Asha Anoosheh. Efficient floating point precision tuning for approximate computing. In *22nd Asia and South Pacific Design Automation Conference, ASP-DAC 2017, Chiba, Japan, January 16-19, 2017*, pages 63–68. IEEE, 2017.

[How60] R. A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA, 1960.

[ID17] Anastasiia Izycheva and Eva Darulova. On sound relative error bounds for floating-point arithmetic. In Daryl Stewart and Georg Weissenbacher, editors, *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 15–22. IEEE, 2017.

[IM19] Arnault Ioualalen and Matthieu Martel. Neural network precision tuning. In David Parker and Verena Wolf, editors, *Quantitative Evaluation of Systems, 16th International Conference, QEST 2019, Glasgow, UK, September 10-12, 2019, Proceedings*, volume 11785 of *Lecture Notes in Computer Science*, pages 129–143. Springer, 2019.

[IMN17] Arnault Ioualalen, Matthieu Martel, and Nicolas Normand. An overview of numalis software suite for reliable numerical computation. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Toulouse, France, October 23-26, 2017*, pages 1–4. IEEE Computer Society, 2017.

[JC08] Fabienne Jézéquel and Jean Marie Chesneaux. CADNA: a library for estimating round-off error propagation. *Comput. Phys. Commun.*, 178(12):933–955, 2008.

[JGM+20] Vinu Joseph, Ganesh Gopalakrishnan, Saurav Muralidharan, Michael Garland, and Animesh Garg. A programmable approach to neural network compression. *IEEE Micro*, 40(5):17–25, 2020.

[KBT14] Tim King, Clark W. Barrett, and Cesare Tinelli. Leveraging linear and mixed integer programming for SMT. In Philipp Rümmer and Christoph M. Wintersteiger, editors, *Proceedings of the 12th International Workshop on Satisfiability Modulo Theories, SMT 2014, affiliated with the 26th International Conference on Computer Aided Verification (CAV 2014), the 7th International Joint Conference on Automated Reasoning (IJCAR 2014), and the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT 2014), Vienna, Austria, July 17-18, 2014*, volume 1163 of *CEUR Workshop Proceedings*, page 65. CEUR-WS.org, 2014.

[KKS00] Ki-Il Kum, Jiyang Kang, and Wonyong Sung. Autoscaler for c: an optimizing floating-point to integer c program converter for fixed-point digital signal processors. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 47(9):840–848, 2000.

[KLKS18] M. Kim, J. Lee, Y. Kim, and Y. H. Song. An analysis of energy consumption under various memory mappings for fram-based IoT devices. In *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, pages 574–579, Feb 2018.

[Knu74] Donald E. Knuth. Computer programming as an art. *Commun. ACM*, 17(12):667–673, 1974.

[KSW+19a] Pradeep V. Kotipalli, Ranvijay Singh, Paul Wood, Ignacio Laguna, and Saurabh Bagchi. AMPT-GA: automatic mixed precision floating point tuning for GPU applications. In *Proceedings of the ACM International Conference on Supercomputing, ICS*, pages 160–170. ACM, 2019.

[KSW+19b] Pradeep V. Kotipalli, Ranvijay Singh, Paul Wood, Ignacio Laguna, and Saurabh Bagchi. AMPT-GA: automatic mixed precision floating point tuning for GPU applications. In Rudolf Eigenmann, Chen Ding, and Sally A. McKee, editors, *Proceedings of the ACM International Conference on Supercomputing, ICS 2019, Phoenix, AZ, USA, June 26-28, 2019*, pages 160–170. ACM, 2019.

[LA04] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004.

[LH03] R. Lougee-Heimer. The common optimization interface for operations research: Promoting open-source software in the operations research community. *IBM J. Res. Dev.*, 47(1):57–66, January 2003.

[LHD+10] Michael D. Linderman, Matthew Ho, David L. Dill, Teresa H. Meng, and Garry P. Nolan. Towards program optimization through automated analysis

of numerical precision. In Andreas Moshovos, J. Gregory Steffan, Kim M. Hazelwood, and David R. Kaeli, editors, *Proceedings of the CGO 2010, The 8th International Symposium on Code Generation and Optimization, Toronto, Ontario, Canada, April 24-28, 2010*, pages 230–237. ACM, 2010.

[LHdSL13a] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In *International Conference on Supercomputing, ICS'13*, pages 369–378. ACM, 2013.

[LHdSL13b] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In Allen D. Malony, Mario Nemirovsky, and Samuel P. Midkiff, editors, *International Conference on Supercomputing, ICS'13, Eugene, OR, USA - June 10 - 14, 2013*, pages 369–378. ACM, 2013.

[LHS13] Michael O. Lam, Jeffrey K. Hollingsworth, and G. W. Stewart. Dynamic floating-point cancellation detection. *Parallel Comput.*, 39(3):146–155, 2013.

[Lib08] Ryan Libby. A simple method for reliable footstep detection on embedded sensor platforms. *Sensors (Peterborough, NH)*, 2008.

[LJS+21] Debasmita Lohar, Clothilde Jeangoudoux, Joshua Sobel, Eva Darulova, and Maria Christakis. A two-phase approach for conditional floating-point verification. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*, volume 12652 of *Lecture Notes in Computer Science*, pages 43–63. Springer, 2021.

[LVMS19] Michael O. Lam, Tristan Vanderbruggen, Harshitha Menon, and Markus Schordan. Tool integration for source-level mixed precision. In Ignacio Laguna and Cindy Rubio-González, editors, *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness), Denver, CO, USA, November 18, 2019*, pages 27–35. IEEE, 2019.

[LWSB19] Ignacio Laguna, Paul C. Wood, Ranvijay Singh, and Saurabh Bagchi. Gpumixer: Performance-driven floating-point tuning for GPU scientific applications. In Michèle Weiland, Guido Juckeland, Carsten Trinitis, and Ponnuswamy Sadayappan, editors, *High Performance Computing - 34th International Conference, ISC High Performance 2019, Frankfurt/Main, Germany, June 16-20, 2019, Proceedings*, volume 11501 of *Lecture Notes in Computer Science*, pages 227–246. Springer, 2019.

[Mak] Andrew O. Makhorin. Glpk (gnu linear programming kit). Available at `http://www.gnu.org/software/glpk/glpk.html`.

[Mar17] Matthieu Martel. Floating-point format inference in mixed-precision. In *NASA Formal Methods - 9th International Symposium, NFM*, volume 10227 of *Lecture Notes in Computer Science*, pages 230–246, 2017.

[MBdD+10] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.

[MCD17] Victor Magron, George A. Constantinides, and Alastair F. Donaldson. Certified roundoff error bounds using semidefinite programming. *ACM Trans. Math. Softw.*, 43(4):34:1–34:31, 2017.

[McK62] William Marshall McKeeman. Algorithm 145: Adaptive numerical integration by simpson's rule. *Commun. ACM*, 5(12):604, 1962.

[Min06] Antoine Miné. The octagon abstract domain. *High. Order Symb. Comput.*, 19(1):31–100, 2006.

[Min17] Antoine Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Found. Trends Program. Lang.*, 4:120–372, 2017.

[MKF03] Junichiro Makino, Eiichiro Kokubo, and Toshiyuki Fukushige. 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, 15-21 November 2003, Phoenix, AZ, USA, CD-Rom*, page 2. ACM, 2003.

[MLO+18] Harshitha Menon, Michael O. Lam, Daniel Osei-Kuffuor, Markus Schordan, Scott Lloyd, Kathryn Mohror, and Jeffrey Hittinger. 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, Dallas, TX, USA, November 11-16, 2018*, pages 48:1–48:13. IEEE / ACM, 2018.

[Mon16] David Monniaux. A survey of satisfiability modulo theory. In Vladimir P. Gerdt, Wolfram Koepf, Werner M. Seiler, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing - 18th International Workshop, CASC 2016, Bucharest, Romania, September 19-23, 2016, Proceedings*, volume 9890 of *Lecture Notes in Computer Science*, pages 401–425. Springer, 2016.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[Moo79] R. E. Moore. *Methods and Applications of Interval Analysis*. SIAM, 1979.

[MPN+19] Divya Priyadharshini Mohan, Ponnusamy Ponmurugan, M. Nishanthi, A. Judy, and Elzalet Jayaram. Efficient device for measuring and controlling various parameters in standalone solar/wind system incorporating internet of things. *Far East Journal of Electronics and Communications*, 19:119–132, 02 2019.

[MRS08] Daniel Ménard, Romuald Rocher, and Olivier Sentieys. Analytical fixed-point accuracy evaluation in linear time-invariant systems. *IEEE Trans. Circuits Syst. I Regul. Pap.*, 55-I(10):3197–3208, 2008.

[MSGK14] D. Morris, T. Scott Saponas, A. Guillory, and I. Kelner. Recofit: Using a wearable sensor to find, recognize, and count repetitive exercises. pages 3225–3234, 2014.

[Mul05] Jean-Michel Muller. On the definition of ulp(x). Research Report RR-5504, LIP RR-2005-09, INRIA, LIP, 2005.

[NAL+14] Ralph Nathan, Bryan Anthonio, Shih-Lien Lu, Helia Naeimi, Daniel J. Sorin, and Xiaobai Sun. Recycled error bits: Energy-efficient architectural support for floating point accuracy. In Trish Damkroger and Jack J. Dongarra, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, pages 117–127. IEEE Computer Society, 2014.

[Nau12] Uwe Naumann. *The Art of Differentiating Computer Programs - An Introduction to Algorithmic Differentiation*, volume 24 of *Software, environments, tools*. SIAM, 2012.

[NNH10] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010.

[NRB+18] Ricardo Nobre, Luís Reis, João Bispo, Tiago Carvalho, João M. P. Cardoso, Stefano Cherubin, and Giovanni Agosta. Aspect-driven mixed-precision tuning targeting gpus. In *Proceedings of the 9th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and 7th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms, PARMA-DITAM@HiPEAC 2018, Manchester, United Kingdom, January 23-23, 2018*, pages 26–31. ACM, 2018.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on*

Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007, pages 89–100. ACM, 2007.

[Pap81] Christos H Papadimitriou. On the complexity of integer programming. *Journal of the ACM (JACM)*, 28(4):765–768, 1981.

[Par97] D. S. Parker. Monte carlo arithmetic: exploiting randomness in floating-point arithmetic. Technical Report CSD-970002, University of California (Los Angeles), 1997.

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[Pat92] Patriot missile defense: Software problem led to system failure at dhahran, saudi arabia. Technical Report GAO/IMTEC-92-26, General Accounting office, 1992.

[PSWT15] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 1–11. ACM, 2015.

[RBA+21] Maulin Raval, Shubhendu Bhardwaj, Aparna Aravelli, Jaya Dofe, and Hardik A. Gohel. Smart energy optimization for massive iot using artificial intelligence. *Internet Things*, 13:100354, 2021.

[RG15] Pierre Roux and Pierre-Loïc Garoche. Practical policy iterations - A practical use of policy iterations for static analysis: the quadratic case. *Formal Methods Syst. Des.*, 46(2):163–196, 2015.

[RNM+16] Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H. Bailey, and David Hough. Floating-point precision tuning using blame analysis. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 1074–1085. ACM, 2016.

[RNN+13] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious: tuning assistant for floating-point precision. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13*, pages 27:1–27:12. ACM, 2013.

[Roj19] Krzysztof Rojek. Machine learning method for energy reduction by utilizing dynamic mixed precision on gpu-based supercomputers. *Concurr. Comput. Pract. Exp.*, 31(6), 2019.

[Sch98] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[SF03] J. Stolfi and L. H. D. Figueiredo. An introduction to affine arithmetic. *Trends in Applied and Computational Mathematics*, 4:297–312, 2003.

[SFN+21] Brett Saiki, Oliver Flatt, Chandrakana Nandi, Pavel Panchekha, and Zachary Tatlock. Combining precision tuning and rewriting. In *ARITH*, pages 1–8. IEEE, 2021.

[Sho77] N. Z. Shor. Cut-off method with space extension in convex programming problems. *Cybernetics*, 13:94–96, 1977.

[SJRG15] Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In Nikolaj Bjørner and Frank S. de Boer, editors, *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, volume 9109 of *Lecture Notes in Computer Science*, pages 532–550. Springer, 2015.

[SKNS13] Kushal Seetharam, Lance Ong-Siong Co Ting Keh, Ralph Nathan, and Daniel J. Sorin. Applying reduced precision arithmetic to detect errors in floating point multiplication. In *IEEE 19th Pacific Rim International Symposium on Dependable Computing, PRDC 2013, Vancouver, BC, Canada, December 2-4, 2013*, pages 232–235. IEEE Computer Society, 2013.

[SPLT18] Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. Finding root causes of floating point error. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 256–269. ACM, 2018.

[SSA14] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating-point programs with tunable precision. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 53–64. ACM, 2014.

[STF+19] Rocco Salvia, Laura Titolo, Marco A. Feliú, Mariano M. Moscato, César A. Muñoz, and Zvonimir Rakamaric. A mixed real and floating-point solver. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods -*

11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings, volume 11460 of *Lecture Notes in Computer Science*, pages 363–370. Springer, 2019.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[TFMM18] Laura Titolo, Marco A. Feliú, Mariano M. Moscato, and César A. Muñoz. An abstract interpretation framework for the round-off error analysis of floating-point programs. In Isil Dillig and Jens Palsberg, editors, *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*, volume 10747 of *Lecture Notes in Computer Science*, pages 516–537. Springer, 2018.

[UFdD19] Yohann Uguen, Luc Forget, and Florent de Dinechin. Evaluating the hardware cost of the posit number system. In Ioannis Sourdis, Christos-Savvas Bouganis, Carlos Álvarez, Leonel Antonio Toledo Díaz, Pedro Valero-Lara, and Xavier Martorell, editors, *29th International Conference on Field Programmable Logic and Applications, FPL 2019, Barcelona, Spain, September 8-12, 2019*, pages 106–113. IEEE, 2019.

[VEL+17] Thiruvengadam Vijayaraghavan, Yasuko Eckert, Gabriel H. Loh, Michael J. Schulte, Mike Ignatowski, Bradford M. Beckmann, William C. Brantley, Joseph L. Greathouse, Wei Huang, Arun Karunanithi, Onur Kayiran, Mitesh R. Meswani, Indrani Paul, Matthew Poremba, Steven Raasch, Steven K. Reinhardt, Greg Sadowski, and Vilas Sridharan. Design and analysis of an APU for exascale computing. In *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*, pages 85–96. IEEE Computer Society, 2017.

[Vig04] Jean Vignes. Discrete stochastic arithmetic for validating results of numerical software. *Numer. Algorithms*, 37(1-4):377–390, 2004.

[WFW+94] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. Suif: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 29(12):31–37, 1994.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993.

[YAK18] Serif Yesil, Ismail Akturk, and Ulya R. Karpuzcu. Toward dynamic precision scaling. *IEEE Micro*, 38(4):30–39, 2018.

[ZH02] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.

[Zha10] Neil Zhao. Full-featured pedometer design realized with 3-axis digital accelerometer. 2010.

Abstract

Although users of High Performance Computing (HPC) are most interested in raw performance, both storage costs and power consumption have become critical concerns. This is due to several technological issues such as the power limitation of processors and the massive cost of communications which arise when executing applications on such architectures. In recent years, the use of precision tuning to improve the performance metrics is emerging as a new trend to save the resources on the available processors. In this thesis, we introduce a new technique for precision tuning radically different from the existing ones. The main idea of our approach is based on a semantic modelling of the propagation of the numerical errors throughout the program source. This yields a system of constraints whose minimal solution gives the best tuning of the program. Based on a static analysis approach, we formulate the problem of precision tuning with two different methods. The first method combines a forward and a backward error analysis which are two popular paradigms of error analysis. Next, our analysis is expressed as a set of linear constraints, made of propositional logic formulas and relations between integer elements only, checked by a SMT solver. The second 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. A finer set of semantic equations is also proposed which does not reduce directly to an ILP problem. So, we use the policy iteration technique to find a solution. Both methods have been implemented in a tool named, POP. We provide in this thesis a detailed evaluation of the performance of POP on several benchmarks coming from various application domains such as embedded systems, Internet of Things (IoT), physics, etc. Also, we show that our results of precision tuning encompass the results of the state-of-the-art tools in several manners.

Keywords: Computer arithmetic, numerical accuracy, static analysis, constraint generation, SMT solver, LP solver, policy iteration.

Résumé

Bien que les utilisateurs de calcul haute performance (HPC) soient plus intéressés par les performances brutes, les coûts de stockage et la consommation d'énergie sont devenus des préoccupations importantes. Ces dernières années, l'utilisation du réglage de la précision pour améliorer les métriques de performance est devenu une nouvelle tendance pour économiser les ressources sur les processeurs disponibles. Ce processus est appelé réglage de précision (precision tuning). Dans cette thèse, nous introduisons une nouvelle technique de réglage de précision radicalement différente de celles existantes. Notre approche est basée sur une modélisation sémantique de la propagation des erreurs numériques à travers le programme. Cela génère un système de contraintes dont la solution minimale donne le meilleur réglage de précision du programme. En se basant sur une approche d'analyse statique, nous formulons le problème du réglage de précision avec deux méthodes différentes. La première méthode combine une analyse d'erreurs en avant et en arrière. Ensuite, nos analyses sont exprimées sous la forme d'un ensemble de contraintes linéaires vérifiées par un solveur SMT. La deuxième méthode consiste à générer un problème de programmation linéaire en nombres entiers (ILP) à partir du code source du programme. Cela se fait en raisonnant sur le bit de poids fort et le nombre de bits significatifs des valeurs des variables. La solution entière à ce problème, calculée en temps polynomial par un solveur de programmation linéaire classique, donne une optimisation des types de données en nombre de bits. Un ensemble plus fin d'équations sémantiques est également proposé dans cette thèse. Il utilise la méthode d'itération sur les politiques pour trouver les nouvelles précisions. Les deux méthodes ont été implémentées dans un outil appelé, POP. Nous proposons dans cette thèse une évaluation détaillée des performances de POP sur plusieurs exemples couvrant divers domaines d'application tels que les systèmes embarqués, l'Internet des objets (IoT), la physique, etc. De plus, nous proposons une comparaison détaillée entre POP et les outils de l'état de l'art.

Mots clefs : Arithmétique des ordinateurs, précision numérique, analyse statique, génération des contraintes, solveur SMT, solveur LP, itération sur les politiques.