Synthesis of certified programs in fixed-point arithmetic, and its application to linear algebra basic blocks

Mohamed Amine Najahi

ACADÉMIE DE MONTPELLIER

UNIVERSITÉ DE **P**ERPIGNAN **V**IA **D**OMITIA

THÈSE

présentée à l'Université de Perpignan Via Domitia
dans l'équipe-projet DALI-LIRMM pour
obtenir le diplôme de doctorat

Spécialité	:	**Informatique**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Énergie et Environnement**

Synthesis of certified programs in fixed-point arithmetic, and its application to linear algebra basic blocks

par

Mohamed Amine NAJAHI

Soutenue le 10 décembre 2014, devant le jury composé de :

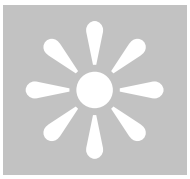
Rapporteurs
M. Laurent-Stéphane DIDIER, Professeur ..... Université du Sud Toulon-Var
M. Florent DE DINECHIN, Professeur ..... INSA de Lyon

Examinateurs
M. Ali MILI, Professeur ..... New Jersey Institute of Technology
Mme. Sylvie BOLDO, Chargée de recherche, HDR ..... INRIA Saclay

Directeur de thèse
M. Matthieu MARTEL, Maître de conférences, HDR ..... Université de Perpignan Via Domitia

Co-Directeur de thèse
M. Guillaume REVY, Maître de conférences ..... Université de Perpignan Via Domitia

À ma famille



REMERCIEMENTS

Cette thèse est l'aboutissement d'une aventure de trois années des plus agréables et enrichissantes. C'est donc tout naturellement que j'ai réservé mes premiers remerciements à mes directeurs de thèse, Matthieu MARTEL et Guillaume REVY. Je leur exprime ma gratitude pour la confiance qu'il m'ont accordée et pour avoir guidé, avec beaucoup de bienveillance, mes travaux de recherche.
Je remercie les rapporteurs, Laurent-Stéphane DIDIER et Florent DE DINECHIN, d'avoir accepté de relire et de valider mon travail. Je remercie également Ali MILI et Sylvie BOLDO d'avoir accepté de faire partie de mon jury de thèse, et de s'être intéressé à mes travaux.

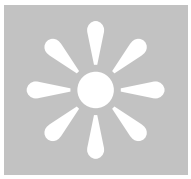
Durant ces trois ans à DALI, je n'ai cessé d'apprendre au contact des permanents de l'équipe et je souhaite les en remercier. Merci à : Bernard, David PARELLO, David DEFOUR, et Christophe. Je n'oublie pas Philippe LANGLOIS que je remercie chaleureusement d'avoir accepté de faire partie de mon jury de *CST* et de m'avoir ouvert les portes de ma prochaine aventure japonaise.
Je remercie également Sylvia qui fait un travail admirable pour nous simplifier les taches administratives. Je n'oublie pas non plus de remercier les anciens *ATERs* de l'équipe dont Michael le grand amateur de foot, ou encore Christophe MOUILLERON qui m'a beaucoup épaulé durant ma première année de thèse. Viennent ensuite mes collègues doctorants dont certains ont été mes co-bureaux, et que je souhaite remercier dans l'ordre chronologique: Arnault, Laurent, Manuel, Jean-Marc, Chems, Rafife, Nasrine, et Djallal. À ceux parmi eux qui n'ont pas encore soutenu leur thèse, je leur souhaite bon courage!

Les bons souvenirs me rappellent aussi avec insistances mes professeurs à l'*UPMC*. Je voudrais tous les remercier et en particulier Michèle SORIA pour ses conseils, Christian

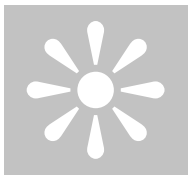
QUEINNEC pour avoir instillé en moi, dés la L1, une passion pour l'informatique et Lisp, et enfin Alain KRAUS dont les cours d'algèbre m'ont réconcilié avec les mathématiques.

Pour conclure, j'ai réservé mes derniers remerciements à ma famille. Je remercie mes parents et mon oncle qui m'ont fait confiance et qui m'ont sans cesse soutenu pendant mes années d'études. Je remercie mes frères, et ma soeur pour qui j'ai souvent été absent ces dernières années. Je remercie les autres membres de ma famille en France et en Tunisie et leur dédie cette thèse. Enfin, j'embrasse ma compagne Caroline et la remercie d'avoir toujours cru en moi et d'avoir accepté l'éloignement pendant ces trois dernières années.

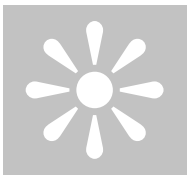
Des logiciels libres dont *GNU Emacs*, *LaTeX*, ou encore *Gnuplot* m'ont facilité la réalisation de ce document. Que soient remerciés leurs auteurs et ceux qui y ont contribué. Enfin, j'espère que cette thèse enrichira les connaissances de son lecteur en arithmétique des ordinateurs et saura répondre à certains de ses questionnements. Que soient donc remerciés tous ceux qui y ont contribué de prés ou de loin.



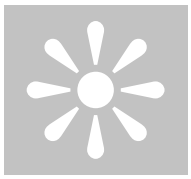
CONTENTS



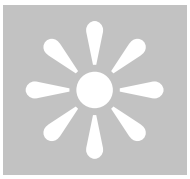
LIST OF FIGURES



LIST OF TABLES



LIST OF ALGORITHMS



INTRODUCTION

Context of this work

OVER the last years, embedded devices have gained widespread use in numerous markets that include the automotive, medical, and consumer electronics. This trend, according to market place observers, is not temporary and the embedded systems market is expected to grow thanks to their continually decreasing prices.

Typical embedded devices are built around a microprocessor or an FPGA that is dedicated to one or few tasks. And vendors tend to pick the cheapest hardware components that efficiently carry out the required computations without wasting too much energy. Such criteria are not satisfied by general purpose processors which are costly, greedy for resources, and tailored for multitasking and for dealing with large amounts of memory.

Rather, these criteria are fulfilled by Digital Signal Processors (DSPs), microcontrollers, fixed-point processors, and more recently FPGAs. To be cost effective, these architectures rarely provide any hardware support for floating-point computations. Yet, they are often used for computational intensive tasks such as signal and image processing. These tasks implement algorithmic basic blocks such as convolutions, fast Fourier transforms, digital filters, and linear algebra computations.

For the software developer, implementing these algorithms on embedded devices requires to adjust to the environmental constraints and to use the fixed-point arithmetic if necessary. Indeed, using floating-point computations or relying on operating systems facilities is not always affordable. To add to this is that programming embedded devices is often less interactive than desktop programming. When combined, these factors make fixed-point programming tedious and error prone. For instance, in [KWCM98], Willems reports that more than 50% of the design time of digital systems is spent on manually mapping

a floating-point prototype into a fixed-point program. Besides, fixed-point programmers cannot claim, like assembly language programmers, that their code has a better quality than the compilers'. Indeed, toy examples put aside, it is difficult to manually write correct fixed-point code.

The second perceived problem with fixed-point programming, besides its difficulty, is its low numerical quality. Indeed, compared to floating-point numbers, fixed-point numbers have a low dynamic range. This property led to the persistent belief that fixed-point numbers are inherently unsafe and should not be relied upon for critical applications.

In summary, to bring fixed-point programming to non-expert developers and make it a reasonable alternative to floating-point, the goals to be achieved are as follows:

1. simplifying fixed-point programming, and
2. asserting strong properties on the numerical quality of fixed-point programs.

To achieve these goals, the ANR project DEFIS[1] (Design of FIxed-point embedded Systems) was set up by researchers working on signal processing and computer arithmetic. This thesis took place in the context of this project which was intended to be a catalyst for new techniques and tools for fixed-point programming and aimed to unify the efforts of two separate research communities. DEFIS also involved industrial partners whose applications could not be implemented in fixed-point arithmetic without new breakthroughs, techniques, or tools.

Members of DEFIS tackled the above goals from different angles. While Ménard *et al.* (CAIRN team, IRISA at Rennes and Lannion) worked on converting floating-point programs to fixed-point ones [MCCS02], Didier *et al.* (PEQUAN team, Univ. Pierre et Marie Curie) concentrated on a specific area of applications, namely digital filters [LHD12]. Although these approaches are different, work also progressed on unifying them in a unique flow [MRS+12], exemplified by the DEFIS tool whose work-flow is shown in Figure 1.9 of Chapter 1.

Our approach: synthesis of certified fixed-point code

In this thesis, we tackled goals 1 and 2 as follows: First, to deal with the difficulty of fixed-point programming, we suggested to rely on code synthesis tools. For instance, Part I describes an arithmetic model and a tool, CGPE, that implements it. With such a tool, the non-expert programmer concentrates on algorithmic level issues and leaves the burden of fixed-point code generation to the tool. However, unlike the approach pioneered by Sung *et al.* [SK95] and Ménard *et al.* [MCCS02], ours do not proceed by floating-point to fixed-point conversion. Rather, our tools take a mathematical description of the problem as well

[1] ANR project DEFIS (*INS 2011*, ANR-11-INSE-0008). See http://defis.lip6.fr/.

as the range of the input variables and generate fixed-point code. This choice makes our tools less general, since the number of input problems supported is limited and, in order for the tool to generate code for a new kind of problems, support for it must be added. However, this design choice was made so that analytic methods could be used to determine the ranges of the variables and to bound the rounding errors.

Then, as for the claim that fixed-point computations are inherently inaccurate, some examples are presented that refute it such as the IIR filter implementation in Section 3.4.3 of Chapter 3. Indeed, depending on the input problem, fixed-point programs may yield more accurate results than floating-point ones. Yet, to further increase the user's confidence in the numerical quality of the codes generated by our tools, we suggested to generate accuracy certificates. The user runs a formal verification tool[2] on these scripts which reassures him that the rounding errors are indeed below the threshold returned by the tool.

Since no open source fixed-point generation tools were available, a part of the work that led to this thesis was allocated for software development and led to the following realizations:

- Substantial enhancement to the CGPE software tool. It was extended from a tool that supported only polynomial evaluation in unsigned arithmetic [MR11] to a modular library that handles many types of expressions and has multiple front and back-ends. These successive enhancement were the result of collaborative development with Guillaume REVY and Christophe MOUILLERON. CGPE is described in Chapter 3 and is now an open source project.
- A new tool, FPLA[3] (Fixed-Point Linear Algebra) to investigate the trade-offs of generating fixed-point code for higher level problems such as matrix multiplication, triangular matrix inversion, and Cholesky decomposition. FPLA deals with these algorithmic problems while entirely relying on CGPE for low level code synthesis. This tool was used to implement the approaches of Chapters 4 and 5, and is described in Chapter 6.

Organization of this work

Overall organization

The organization of this thesis is illustrated by the graph of Figure 1 below. It starts with an introductory chapter to fixed-point arithmetic and to the state of the art techniques to automate fixed-point programming. Then, the rest of the document is made of two parts of 2 and 3 chapters, respectively. The two parts correspond to the two levels tackled in this thesis: the arithmetic and algorithmic levels.

[2] The formal verification tool we rely on in this work is GAPPA.

[3] The licence of this tool is yet to be decided but is available upon request.

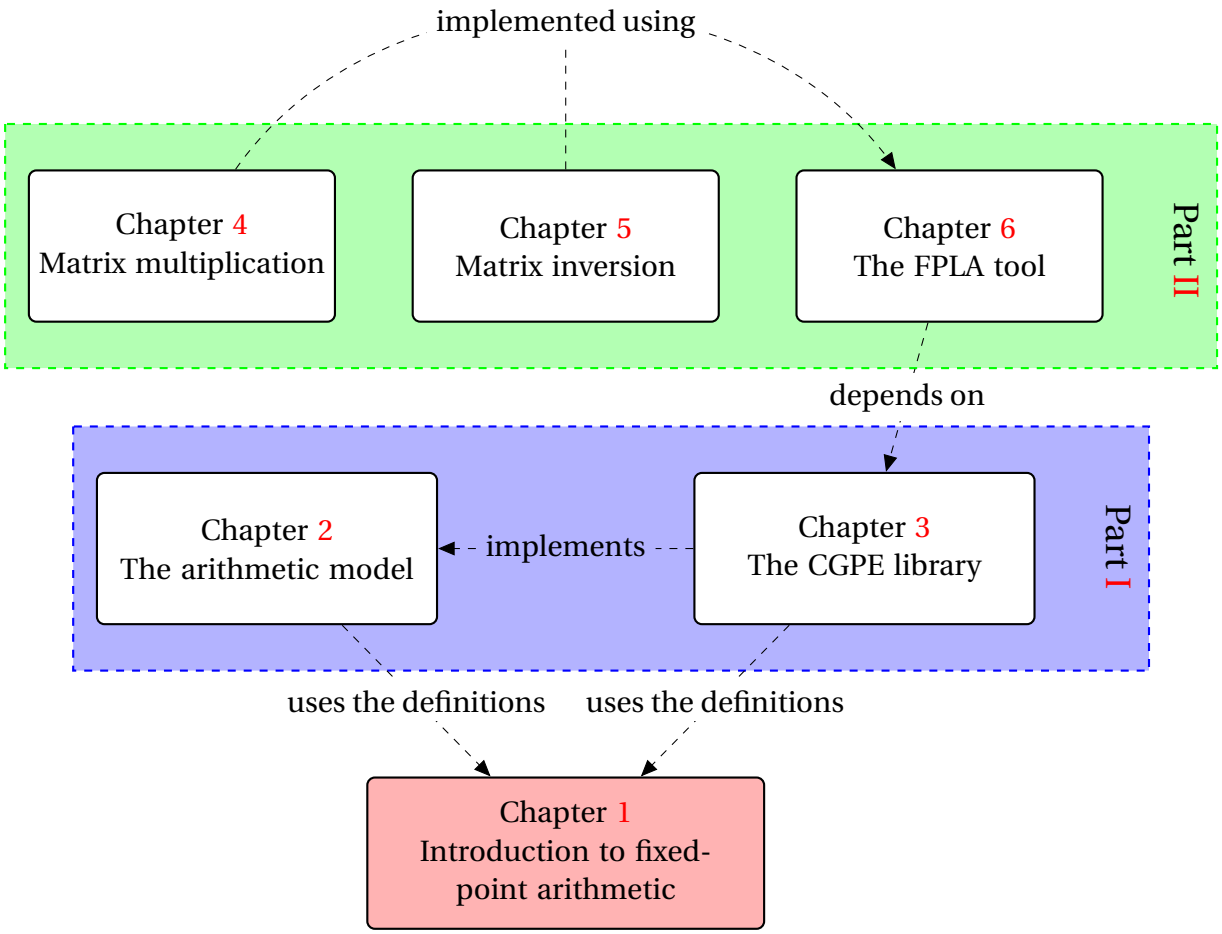


Figure 1: Organization of this document and relations between the chapters.

Part I deals with the arithmetic model and its implementation in the CGPE software tool. Chapter 2 explains how to implement and bound the rounding errors of the basic fixed-point operators. Chapter 3 exposes the design of the CGPE tool and shows some code samples generated for certain problems.

Part II deals with the more algorithmic problems of synthesizing code for matrix multiplication and inversion. Chapter 4 is a study of code size versus accuracy trade-offs when generating code for matrix multiplication. Chapter 5 presents a flow for matrix inversion and exposes the basic blocks needed which include Cholesky decomposition and triangular matrix inversion. Finally, Chapter 6 summarizes the design of the FPLA software tool and shows how to use it to synthesize code for basic linear algebra blocks.

Figure 1 justifies this organization. It shows that Part II is built on the abstraction layer provided by CGPE and the arithmetic model it implements.

Detailed description of the chapters

Chapter 1 - State of the art on fixed-point arithmetic. This chapter introduces the fixed-point number representation and presents the notation used throughout this work to denote fixed-point formats. Then, it clarifies the difference between the fixed-point and floating-point representations which stems from the implicit nature of the scaling factor. For this purpose, this chapter briefly describes the integer and floating-point arithmetics.

In addition, this chapter deals with the state of the art approaches to automate fixed-point programming. It starts by citing some well known methodologies and shows that they have in common the two steps of range and precision analyses. Finally, the rest of the chapter is a survey of methods for range and precision analyses. Each method is presented and tested on examples.

This chapter also mentions the general framework suggested along with our DEFIS partners and presented at DASIP 2012 [MRS+12].

> [MRS+12] Daniel Ménard, Romuald Rocher, Olivier Sentieys, Nicolas Simon, Laurent-Stéphane Didier, Thibault Hilaire, Benoît Lopez, Eric Goubault, Sylvie Putot, Franck Vedrine, Amine Najahi, Guillaume Revy, Laurent Fangain, Christian Samoyeau, Fabrice Lemonnier, and Christophe Clienti. Design of Fixed-Point Embedded Systems (defis) French ANR Project. In *Proceedings of the Conference on Design and Architectures for Signal and Image Processing, DASIP 2012, Karlsruhe, Germany, 23-25 October 2012*, pages 365–366, Karlsruhe, Allemagne, 2012.

Part I – A framework for the synthesis of certified fixed-point programs

This part deals with the low level details of an arithmetic model and its implementation. Its goal is to describe an arithmetic model and to build a tool that generates certified fixed-point code that adheres to it.

Chapter 2 - Implementation and error bounds of the basic fixed-point arithmetic operators. This chapter explains how the range of values and the error bounds of every fixed-point variable are computed at synthesis time. It uses an approach where the propagation rule of every fixed-point operator must be decided.

Then, each fixed-point operator such as addition, multiplication, shifts, square root, and division is studied. For each, we show how to determine the fixed-point format of its output, and a mean to implement it using integer operations.

The basic arithmetic model of this chapter was first described in an article presented at PECCS 2014 [MNR14a]. The square root and division operators were later formalized for the purpose of matrix inversion and were presented at DASIP 2014 [MNR14b].

> [MNR14a] Matthieu Martel, Amine Najahi, and Guillaume Revy. Code Size and Accuracy-Aware Synthesis of Fixed-Point Programs for Matrix Multiplication. In *Proc. of the 4th International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS 2014)*, pages 204–214. SciTePress, January 2014.

> [MNR14b] Matthieu Martel, Amine Najahi, and Guillaume Revy. Toward the synthesis of fixed-point code for matrix inversion based on cholesky decomposition. In *Proceedings of the 6th Conference on Design and Architectures for Signal and Image Processing (DASIP 2014)*, pages 73–80, Madrid, Spain, October 2014.

Chapter 3 - The CGPE software tool. This chapter presents our implementation of the arithmetic model of Chapter 2. After a review of the CGPE software tool, it describes a major enhancement based on instruction selection. This enhancement was the subject of an abstract submitted to SCAN 2012 [MNR12], and an article published and presented at SYNASC 2014 [MNR14c]. Finally, all the necessary steps to generate certified code for an infinite impulse response (IIR) filter using CGPE are shown.

> [MNR12] Christophe Mouilleron, Amine Najahi, and Guillaume Revy. Approach based on instruction selection for fast and certified code generation. In *Proceedings of the 15th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN 2012)*, Novosibirsk, Russia, September 2012.

> [MNR14c] Christophe Mouilleron, Amine Najahi, and Guillaume Revy. Automated Synthesis of Target-Dependent Programs for Polynomial Evaluation in Fixed-Point Arithmetic. In *Proceedings of the 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2014)*, Timisoara, Romania, September 2014.

Part II – Study of the trade-offs in the synthesis of fixed-point programs for linear algebra basic blocks

This part is organized in three chapters and tackles problems of higher level than those of Part I. Particularly, it is dedicated to code synthesis for linear algebra basic blocks that include matrix multiplication and matrix inversion.

Chapter 4 - Fixed-point code synthesis for matrix multiplication. This chapter first considers straightforward approaches to synthesize code for matrix multiplication. It shows that these approaches may yield either code with loose accuracy bounds or large size. It then presents a new method to find trade-offs between code size and accuracy. This approach tries to generate less code by merging together vectors that are close according to a certain distance. This technique is implemented in the FPLA tool and experimental data is given that shows how it can reduce the code size by 50% while maintaining good numerical properties.

The results of this chapter were published and presented at PECCS 2014 [MNR14a].

> [MNR14a] Matthieu Martel, Amine Najahi, and Guillaume Revy. Code Size and Accuracy-Aware Synthesis of Fixed-Point Programs for Matrix Multiplication. In *Proc. of the 4th International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS 2014)*, pages 204–214. SciTePress, January 2014.

Chapter 5 - Fixed-point synthesis for matrix decomposition and inversion. This chapter investigates code synthesis for matrix inversion. It proposes a flow that is based on code synthesis for Cholesky decomposition followed by code synthesis for triangular matrix inversion. These two building blocks involve code synthesis for division, and since the output format of division must be fixed by the user, various strategies are investigated on how to set the right output format.

Experimental data is provided for the different strategies and the error bounds are compared to experimental ones obtained by comparing to floating-point implementations.

The results of this chapter were also obtained using the FPLA tool, and were published and presented at DASIP 2014 [MNR14b].

> [MNR14b] Matthieu Martel, Amine Najahi, and Guillaume Revy. Toward the synthesis of fixed-point code for matrix inversion based on cholesky decomposition. In *Proceedings of the 6th Conference on Design and Architectures for Signal and Image Processing (DASIP 2014)*, pages 73–80, Madrid, Spain, October 2014.

Chapter 6 - FPLA: a software tool for fixed-point code synthesis. This chapter presents an overview of the FPLA software tool. This tool relies on CGPE for low level code synthesis and generates fixed-point code for matrix multiplication, Cholesky decomposition, and triangular matrix inversion. Then, examples are presented to show the quality of the code generated. Finally, some further improvements to the whole tool-chain are suggested as future works.

CHAPTER 1

STATE OF THE ART ON FIXED-POINT ARITHMETIC

In this chapter, we introduce the fixed-point arithmetic in two steps. First, we expose the fixed-point numbers representation and compare it with the representation of floating-point numbers. Next, we summarize the state of the art techniques for range and precision analyses, the two essential steps in the process of automated fixed-point programming.

1.1 Introduction

FIXED-POINT numerical programming dates back to the early days of computing. Indeed, in his 1945 seminal report describing the *EDVAC* [vN93], John von Neumann investigated the use of fixed-point arithmetic in an entire section entitled "The binary point". However, with the emergence of the software industry, more and more programmers lacked the arithmetic and numerical analysis expertise required to write fixed-point programs. Also, the difficulties of fixed-point programming were made worse by the absence of standards and code generation tools. These limits of fixed-point arithmetic, in conjunction to Moore's law which promised an enduring boost in the hardware's performance led to the widespread adoption of the standardized floating-point arithmetic [75408]. Indeed, programming with floating-point numbers is handy since all the

arithmetical details are transparent to the programmer and are provided for by the environment and hardware. However, with targets such as ASICs, FPGAs, and embedded systems dedicated to digital signal processing, floating-point hardware remains slow and costly, and fixed-point arithmetic is still an efficient and low footprint alternative.

This chapter is an introduction to the fixed-point arithmetic. It is organized as follows: Section 1.2 explains the representation of fixed-point numbers and introduces the notation used throughout this work to describe fixed-point formats. This section is concluded by a comparison between fixed-point and floating-point numbers. Section 1.3 describes the two categories of automated fixed-point programming which are fixed-point code generation and floating-point to fixed-point conversion. The second part of the section is a survey of range and precision analyses: the two main steps of automated fixed-point programming.

1.2 The fixed-point arithmetic

Contrarily to the floating-point arithmetic, virtually all processors have built-in support for integer arithmetic. Since fixed-point operations essentially rely on integer instructions, computing with fixed-point numbers is highly efficient. Indeed, some fixed-point operations such as addition and subtraction are identical to their integer counterparts. For this reason, we shall briefly describe the integer arithmetic as implemented in most digital computers.

1.2.1 Background on integer arithmetic

Modern processors embed an execution unit called **ALU** (Arithmetic Logic Unit) that performs elementary integer operations including $+$, $-$, and $\times$ [HP06]. Some architectures also provide hardware support for $\div$, %, and $\sqrt{}$, that is integer division, remainder, and square root operations. The high efficiency and error free nature of integer arithmetic makes it ideal for symbolic computations and for manipulating enumerable values.

Yet, a classical programming pitfall is to confuse computer integers with mathematical integers. Indeed, to speed up computations and reduce the memory footprint, computer arithmetic uses limited precision. Standard `C` [Int10] for instance, stipulates that the `int` data-type must provide at least all the integers in the interval $[-(2^{15}-1), 2^{15}-1]$. This allows a complying compiler to implement the `int` data-type using no more than 16 bits.[1] The same is true for Standard `C`'s `long` data-type for which a complying implementation may use a 32-bit integer representation [Int10].

The limited precision implies that the range of computer integers is bounded, and that overflow occurs when the result of any operation happens to be outside the representable

[1] The interval provided by the standard is symmetric and encloses $2^{16}-1$ values. This choice was made to allow both sign-and-magnitude and two's complement implementations to use 2-bytes per integer [HJ95].

range. In this case, the behavior of the program is highly dependant on the hardware and programming language. For instance, languages like `C` and `Java` ignore the issue of integer overflow, while `Ada` and `Fortran` expect the hardware to notify the program [HP06].

In practice, most ALUs implement integer operations using modular arithmetic. In this arithmetic, operations are performed modulo N and emulate the integers as long as the computed values are less than N. While the result of an intermediate computation T may grow larger than N using integer arithmetic, its modular counterpart computes the value (T mod N) whose magnitude is always inferior to N.

On binary machines, choosing N to be a power of 2 speeds up elementary operations and reduces the hardware cost. For instance, many low-end micro-controllers use $N = 2^8$. On desktops and some embedded system, $N = 2^{16}$ and $N = 2^{32}$ are widely deployed while $N = 2^{64}$ can be found mainly on high-end workstations.

Integer and modular arithmetics are useful and efficient for symbolic computations and cryptography, but are not suitable for numerical computations involving fractional numbers. For such applications, the fixed-point and floating-point numbers are better approximations of real numbers.

1.2.2 Computer representation of fixed-point numbers

The fixed-point arithmetic allows the programmer to compute with rational numbers, while using the integer representation and operators. Indeed, a fixed-point number x is defined by the coupling of two integers X and f as follows:

1. X the integer representation of x, and
2. f the implicit scaling factor of x.

The value of the fixed-point number x is given by the rational

$$x = \frac{X}{\beta^f}, \tag{1.1}$$

where β is the representation radix. Since the binary radix is widely used on modern processors, we implicitly consider $\beta = 2$ in the subsequent, and use the words bit and digit interchangeably. Figure 1.1 below shows a fixed-point number whose integer representation fits in 8 bits and having a scaling factor of 5.

Assuming X to be a k-bit unsigned integer, the value of the fixed-point number x is given by:

$$x = \frac{\sum_{\ell=0}^{k-1} X_\ell \cdot 2^\ell}{2^f} = \sum_{\ell=-f}^{k-1-f} X_{\ell+f} \cdot 2^\ell. \tag{1.2}$$

For instance, Figure 1.1 has $(k, f) = (8, 5)$ and $x = \sum_{\ell=-5}^{2} X_{\ell+5} \cdot 2^\ell$.

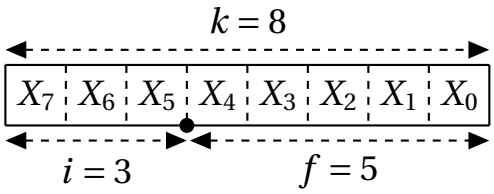


Figure 1.1: An 8-bit fixed-point variable with a scaling factor of 5.

Signed fixed-point numbers

Analogously to integer arithmetic, signed fixed-point numbers are obtained by using a signed integer representation. Indeed, if X were a two's complement encoded integer, the value of the corresponding fixed-point number would be

$$x = \frac{-X_{k-1}\cdot 2^{k-1} + \sum_{\ell=0}^{k-2} X_\ell \cdot 2^\ell}{2^f} = -X_{k-1}\cdot 2^{k-1-f} + \sum_{\ell=-f}^{k-2-f} X_{\ell+f}\cdot 2^\ell. \tag{1.3}$$

In that case, the most significant bit, i.e. X_{k-1}, is called the sign bit. The number is positive if its sign bit is equal to 0 and strictly negative otherwise.

Example 1.1. Let x be a fixed-point number with the 8-bit integer representation

$$X = (1010\,1100)_2$$

and a scaling factor f of 5.

- ◇ If x is an unsigned fixed-point number, then

$$X = (172)_{10} \text{ and } x = (101.01100)_2 = (5.375)_{10}.$$

- ◇ If x is a signed fixed-point number, then

$$X = (-84)_{10} \text{ and } x = -4 + (001.01100)_2 = (-2.625)_{10}.$$

Range of values of a fixed-point variable

Our definition of fixed-point numbers imposes no constraints on the integer representation X. However, for reasons ranging from the efficiency of computations to the better bounding of rounding errors, it is convenient to assume that the integer X belongs to a

bounded interval. Indeed, by allocating 8 bits to the fixed-point variable of Figure 1.1, we implicitly assumed that X were such that

$$X \in \mathbb{Z} \cap [0, 255]\,. \tag{1.4}$$

In such configuration, the fixed-point variable x, with a scaling factor of 5, is bounded as follows:

$$x \in \mathscr{I} = \left[0, 2^3 - 2^{-5}\right], \tag{1.5}$$

where $\mathscr{I}$ is a discrete interval whose elements are equidistantly separated by a step of 2^{-5}. In the signal processing terminology, this distance between two consecutive fixed-point numbers is called the quantization step [CC99]. In practical situations, one may have a more thorough information on the range of values of X. For instance, assume the values taken by x to be the result of measurements retrieved from a sensor and as such, are known to be in a narrower interval,

$$x \in \mathscr{I}' = \left[2^{-5}, 2^2 + 2^{-5}\right]. \tag{1.6}$$

This constraint on x would translate to the following sharper enclosure on its integer representation:

$$X \in [1, 129]. \tag{1.7}$$

Although $\mathscr{I}$ in (1.5) is a safe over-approximation of $\mathscr{I}'$ since $\mathscr{I}' \subset \mathscr{I}$, the enclosure (1.6) is less conservative than (1.5) and allows to compute sharper bounds on rounding errors.

This example stresses the importance of implementation details in fixed-point arithmetic. Indeed, despite being tedious and error prone, fixed-point programming is used for its efficiency and small footprint. To be efficient, a fixed-point framework needs to take into consideration implementation details such as the size of the integer representation and the sharp enclosures on the range of input variables.

The fixed-point code synthesis tools CGPE and FPLA described in Chapters 3 and 6, respectively, operate on fixed-point variables that have a fixed word-length and potentially an interval that narrows the range of their values.

The Q notation for fixed-point formats

The **Q** notation and its variants are widely used to describe fixed-point formats (See [Rev09],[Yat13], and [KiKS96]). This notation results from the following observation: applying the scaling factor of a fixed-point number to its integer representation splits it into two parts, the integer part and the fraction part.

In Figure 1.1, this splitting is signaled by a dot widely known as the radix-point. The integer part is composed of the digits of positive weight, that is, digits whose exponent in Equation (1.3) is positive. In Figure 1.1, this part contains the 3 bits to the left of the radix-point and its width is denoted by i. The fraction part contains the rest of the bits, whose weight is negative and whose contribution to the fixed-point number is fractional. The

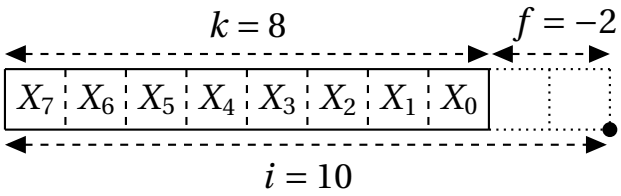


Figure 1.2: A fixed-point variable in the fixed-point format $\mathbf{Q}_{10,-2}$.

number of bits of the fraction part is the same as the scaling factor and is denoted by f in the same figure.

In the **Q** notation, a fixed-point number is said to be in the $\mathbf{Q}_{a,b}$ format if the width of its integer and fraction parts are a and b, respectively. For instance, the fixed-point number of Figure 1.1 is in the $\mathbf{Q}_{3,5}$ format.

The range of a Q format. The notions of fixed-point formats and discrete intervals are related. Indeed, a fixed-point format $\mathbf{Q}_{a,b}$ accommodates values in:

$$\mathscr{R}\text{ange}(\mathbf{Q}_{a,b}) = \begin{cases} \mathscr{I}_u = \left[0, 2^a - 2^{-b}\right], & \text{if unsigned arithmetic is used} \\ \mathscr{I}_s = \left[-2^{a-1}, 2^{a-1} - 2^{-b}\right], & \text{if signed arithmetic is used} \end{cases} \tag{1.8}$$

where $\mathscr{I}_u$ and $\mathscr{I}_s$ are both discrete intervals of step 2^{-b}. The analogous correspondence that deduces formats from ranges is exposed in Section 1.3.4.

Negative integer or fraction parts. The **Q** notation may seem inconsistent when the scaling factor is negative or is larger than the word-length. For instance, the format $\mathbf{Q}_{10,-2}$, shown in Figure 1.2 above implies that the number's representation still fits in 8 bits and has an integer part of 10 bits and a fraction part of -2 bits. The fixed-point numbers in the format $\mathbf{Q}_{10,-2}$ are integers that belong to the interval $[0, 1020]$ and which are equidistantly separated by a step of 4, i.e.,

$$\mathscr{J} = 4\mathbb{Z} \cap [0, 1020]. \tag{1.9}$$

In this case, the inconsistency is having to consider a fraction part whose size is negative.

A similar inconsistency occurs when the scaling factor is larger than the word-length. This is the case with the fixed-point format $\mathbf{Q}_{-2,10}$ shown in Figure 1.3 below. The fixed-point numbers in $\mathbf{Q}_{-2,10}$ fit in 8 bits, belong to the interval $[0, 2^{-2} - 2^{-10}]$, and are separated by a step of 2^{-10}. Again, one must cope with the notation that implies an integer part having a negative size.

Alternative notations. To deal with these inconsistencies, some authors suggested alternative notations for fixed-point formats. For instance, Lopez *et al.* [LHD14] use the nota-

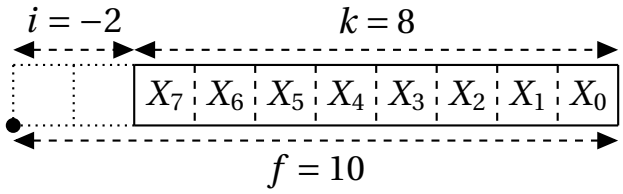


Figure 1.3: A fixed-point variable in the fixed-point format $\mathbf{Q}_{-2,10}$.

tion FPF(m,l), where m and l are the weights of the most and least significant bits[2] of the fixed-point format, respectively. In this notation, the format of the fixed-point number of Figure 1.2 is FPF$(10,2)$ and that of Figure 1.3 is FPF$(-2,-10)$.

The nature of the scaling factor

The implicit nature of the scaling factor of fixed-point numbers often confuses programmers who are used to work with floating-point codes. Indeed, there is a striking resemblance between associating an integer representation to a scaling factor to obtain a fixed-point number and associating a significand to an exponent to get a floating-point number. However, unlike the exponent that is encoded in a floating-point number representation, the fixed-point scaling factor is not encoded in the program's data. Its value is known only to the programmer who is writing the software.

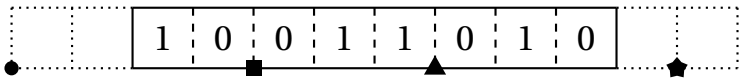


Figure 1.4: A fixed-point number in different fixed-point formats.

A consequence of this implicit nature is that every arithmetical detail, including alignments and overflow prevention must be statically handled and provided for by the programmer. Only when there is a need to retrieve a result, interpret it, or display it to a user, is the fixed-point programmer obliged to apply the scaling factor. To further clarify this interpretation process, consider the example of the integer representation shown in Figure 1.4, and suppose it is produced by a fixed-point program. Regardless of the format, the screen would display the value $(154)_{10}$. However, the programmer must interpret this value in light of the fixed-point format. For instance, Table 1.1 below gives 4 different interpretations that correspond to 4 different fixed-point formats. Viewing fixed-point numbers as simply a different mean to interpret data is the approach emphasized by Yates [Yat13].

On ordering fixed-point formats. Occasionally, one has to compare two fixed-point formats, or to promote a variable from a fixed-point format to another. In such cases, we will

[2] These are commonly called MSB and LSB, respectively.

Scaling factor symbol in Figure 1.4	**Q format**	**Value of X**	**Value of x**
●	$\mathbf{Q}_{-2,10}$	$(10011010)_2 = (154)_{10}$	$(0.0010011010)_2 = (0.150390625)_{10}$
■	$\mathbf{Q}_{2,6}$	$(10011010)_2 = (154)_{10}$	$(10.011010)_2 = (2.40625)_{10}$
▲	$\mathbf{Q}_{5,3}$	$(10011010)_2 = (154)_{10}$	$(10011.010)_2 = (19.25)_{10}$
⬟	$\mathbf{Q}_{9,-1}$	$(10011010)_2 = (154)_{10}$	$(100110100)_2 = (308)_{10}$

Table 1.1: Different interpretations of the same integer representation.

say that the format $\mathbf{Q}_{a_1,b_1}$ is larger than $\mathbf{Q}_{a_2,b_2}$ if $a_1 > a_2$ or if $a_1 = a_2$ and $b_1 > b_2$. For instance, the formats of Table 1.1 are presented in an increasing order, from the smallest to the largest format.

The set of fixed-point variables

When implementing a problem such as univariate polynomial evaluation in fixed-point arithmetic, the coefficients are fixed-point numbers, but the input is a fixed-point variable. Such variable is an element of the set of fixed-point variables we denote $\mathbb{F}\text{ix}$.

A variable $x \in \mathbb{F}\text{ix}$ has a fixed-point format $\mathbf{Q}_{a_x,b_x}$ and an interval $\mathscr{I}_X$ that bounds its integer representation. $\mathscr{I}_X$ is such that $\mathscr{I}_X \subseteq \left[-2^{k-1}, 2^{k-1}-1\right]$ if the variable is signed, and $\mathscr{I}_X \subseteq \left[0, 2^k-1\right]$ if the variable is unsigned, where $k = a_x + b_x$ is the word-length.

In the subsequent, we will use the notation $n \in x$ to say that a fixed-point number n belongs to the fixed-point variable x, if n is in the fixed-point format $\mathbf{Q}_{a_x,b_x}$ and the integer representation N of n belongs to $\mathscr{I}_X$.

Example 1.2. Consider the 32-bit signed fixed-point variables $(x, y) \in \mathbb{F}\text{ix}^2$ shown in Table 1.2.

	Format	**Interval of the Int. Repr.**	$\mathscr{R}$**ange in decimal**
x	$\mathbf{Q}_{4,28}$	$\mathscr{I}_X = \left[-2^{31}, 2^{31}-1\right]$	$[-8, 7.999999996274709701538085937 5]$
y	$\mathbf{Q}_{8,24}$	$\mathscr{I}_Y = \left[-2^{31}+2^{20}, 2^{22}\right]$	$[-127.9375, 0.25]$

Table 1.2: Two fixed-point variables $(x, y) \in \mathbb{F}\text{ix}^2$ and their range in decimal.

$\mathscr{I}_Y$ narrows the range of values taken by y, i.e.,

$$\mathscr{R}\text{ange}(y) = [-127.9375, 0.25] \subsetneq \mathscr{R}\text{ange}(\mathbf{Q}_{8,24}) = [-128, 127.99999940395355224609375]\,.$$

Writing fixed-point code for a problem like polynomial evaluation where the input variable is y implies that, at run-time, this code is guaranteed to behave correctly only when fed with integers that belong to $\mathscr{I}_Y$.

1.2.3 Comparison between fixed-point and floating-point numbers

The floating-point arithmetic

Similarly to fixed-point, floating-point arithmetic dates back to the early 1940's when it was used in the world's first operating computer, the Z3 [Ove01]. In this arithmetic, numbers are represented in standardized formats inspired by the classical scientific notation. With a careful specification of the arithmetic operations and rounding modes, floating-point numbers are an approximation of real numbers that offers some degree of portability to numerical software. Indeed, the IEEE-754 standard [75408], initially published in 1985 and revised in 2008, is followed by most hardware manufacturers and software implementations. Apart from specifying different representation formats and their respective encodings, this standard describes the numerical behavior of the basic operations as well as the rounding modes to be implemented.

As illustrated by Figure 1.5, a standard floating-point number x is represented by 3 values: a sign s, an exponent e, and a significand m. Its value is given by:

$$x = (-1)^s \cdot m \cdot \beta^e. \tag{1.10}$$

The standard specifies different constraints on s, e, and m for $\beta \in \{2, 10\}$:

- The sign $s \in \{0, 1\}$ is a binary value. The number is positive if its sign is zero, and negative otherwise.
- The exponent $e \in [e_{min}, e_{max}]$ where e_{min} and e_{max} are specified for each standardized format.[3]
- The significand $m = m_0.m_1 m_2 \cdots m_{p-1}$ with $m_i \in \{0, \ldots, \beta - 1\}$ and $0 \leq i < p$.

Table 1.3 shows the IEEE-754 specifications for three widely deployed binary formats.

The exact results of arithmetic operations such as addition and multiplication on floating-point numbers are not necessarily floating-point numbers. To deal with this,

[3] To be more specific, the standard specifies that, rather than the exponent itself, a biased version of it is stored.

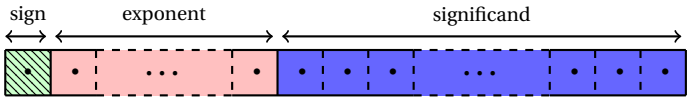


Figure 1.5: IEEE-754 floating-point number encoding.

Standard name	**Common name**	**Bits distribution (sign, exponent, mantissa)**	**Range of e** $[e_{min}, e_{max}]$
Binary32	Single precision	$(1,8,23)$	$[-126,127]$
Binary64	Double precision	$(1,11,52)$	$[-1022,1023]$
Binary128	Quadruple precision	$(1,15,112)$	$[-16382,16383]$

Table 1.3: Standard specifications for 3 IEEE-754 binary formats.

the standard specifies four rounding modes. These rounding modes, round toward zero, round toward plus and minus infinity, and round to nearest even uniquely specify the floating-point number to be returned when the result of an operation is not a floating-point number. Rounding to nearest is the default in most configurations while rounding towards both infinities is useful to implement interval and stochastic arithmetics [Moo66],[JC08]. Also, to enforce reproducibility and encourage formal verification of IEEE-754 programs, the standard requires that the operations $+, -, \times, \div$, and $\sqrt{}$ be correctly rounded [Gol91],[75408]. Finally, the recent development of the floating-point standard tends towards specifying elementary functions such as trigonometric and exponential functions.

The difference between fixed-point and floating-point numbers

The name "fixed-point" is due to the fact that the scaling factor of a fixed-point variable does not change at run-time. This contrasts with floating-point variables where the exponent grows and shrinks dynamically so as to keep a normalized mantissa.[4] A consequence of the dynamic nature of the exponent is that floating-point numbers are not equidistantly separated. Indeed, the gap between a binary floating-point number x and the next binary floating-point number larger than x is given by $ulp(x) = \epsilon \cdot 2^E$ where ϵ is a constant that depends on the word-length of the mantissa and E is the exponent of x [MBDD+10, § 2]. By comparison, fixed-point numbers of the same format $\mathbf{Q}_{a,b}$ are equidistantly separated by a step of 2^{-b}. This layout difference is shown in Figure 1.6 where the total of floating-point and fixed-point numbers is the same.

[4] Subnormal floating-point numbers are the exception to this rule since their mantissa is not normalized.

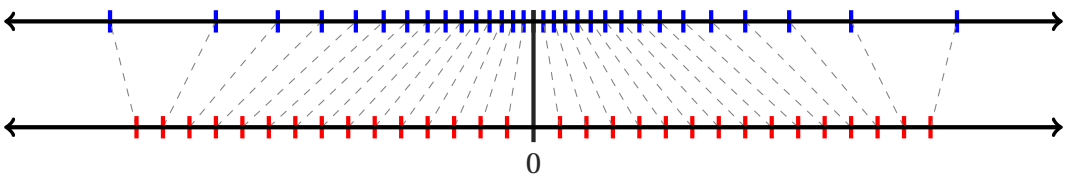


Figure 1.6: Typical distributions of floating-point (top) and fixed-point numbers (bottom).

Another consequence of the run-time evolving nature of the exponent is the possibility for floating-point variables to represent numbers having a large difference in magnitude. This capability is measured by the dynamic range which is the logarithm of the ratio of the largest absolute value representable and the minimum absolute value representable. For the binary32 format, a rough approximation[5] of the dynamic range gives

$$\log_2\left(\frac{2^{e_{max}}}{2^{e_{min}}}\right) = \log_2\left(2^{253}\right) = 253. \tag{1.11}$$

For N-bit signed fixed-point numbers in the $\mathbf{Q}_{a,b}$ format, this ratio is given by

$$\log_2\left(\frac{2^{a-1}}{2^{-b}}\right) = \log_2\left(2^{a+b-1}\right) = \log_2\left(2^{N-1}\right) = N-1, \tag{1.12}$$

which shows that the dynamic range of fixed-point numbers is linearly proportional to the word-length. In this example, the large dynamic range, 253 compared to 31, shows that 32-bit floating-point variables are more appropriate for holding values of different orders of magnitude than fixed-point variables. However, fixed-point variables provide more precision for values of the same or with a slightly varying order of magnitude. Figure 1.7 compares the dynamic range of fixed-point variables with the dynamic range of commonly used floating-point formats. The figure shows how the dynamic range of fixed-point variables evolves linearly with the word-length while the dynamic range of floating-point formats grows exponentially. It also suggests that the dynamic range of fixed-point variables may even be equal or larger to that of floating-point variables for small word-lengths, i.e., less than 10.

1.3 Automated implementation of fixed-point algorithms

Without proper tools, fixed-point programming is tedious and error prone. Willems *et al.* [KWCM98] report that in digital systems, more than 50% of the design time is spent on manually mapping the floating-point prototype into a fixed-point program. To overcome these issues, many researchers tackled the problem of automated implementation

[5] Subnormal numbers were ignored in this approximation.

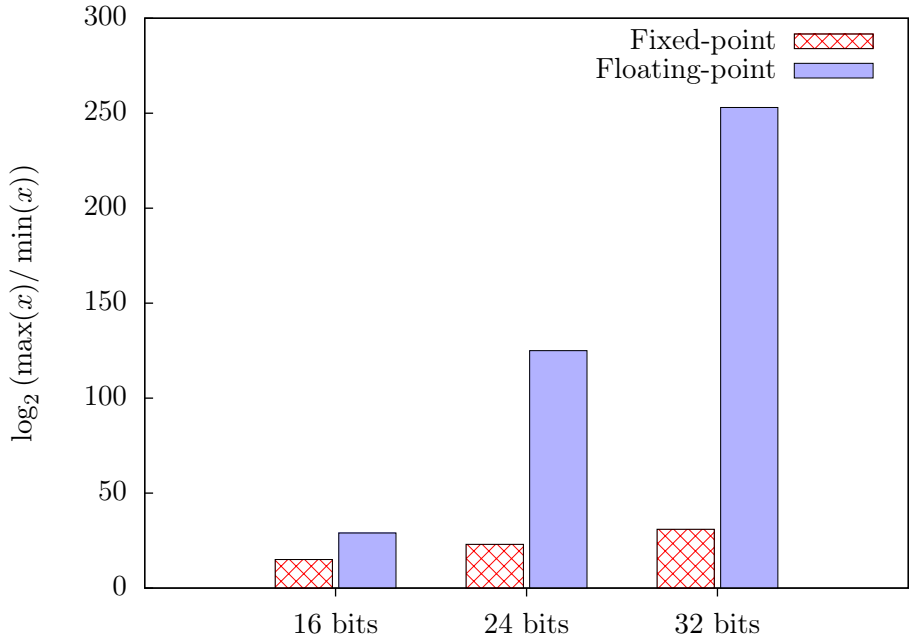


Figure 1.7: Comparison of the dynamic range of fixed-point numbers with common floating-point formats.

of fixed-point programs. For instance, the work of Sung *et al.* [KiKS96] is an early example of research where a methodology and a tool that generates fixed-point code are described. In recent years, the volume of research in this field increased and one can distinguish two broad categories of solutions:

1. solutions based on fixed-point code generation for certain blocks such as polynomial and dot-product evaluations, and
2. solutions based on floating-point to fixed-point design conversion.

1.3.1 Fixed-point code generation

In this category of solutions, a code synthesis tool is developed to tackle a particular mathematical problem such as polynomial evaluation, dot-products, or linear algebra primitives. The tool takes as input some information on the input variables, typically the ranges and/or the fixed-point formats, and eventually the expected output formats and error bounds. Then, the tool proceeds to generate fixed-point code. When fed with appropriate input, the generated code evaluates the particular problem.

Example 1.3. Consider a synthesis tool that generates fixed-point C code for summation. When given as input two 32-bit fixed-point variables v_1 and v_2 in the formats $\mathbf{Q}_{1,31}$ and $\mathbf{Q}_{2,30}$, the tool may generate the code shown in Listing 1.1. The synthesized code is essentially a function that takes two fixed-point numbers in the formats $\mathbf{Q}_{1,31}$ and $\mathbf{Q}_{2,30}$, and returns a fixed-point number in the $\mathbf{Q}_{3,29}$ format. Before adding the two arguments, the code converts them to the $\mathbf{Q}_{3,29}$ format to avoid overflow.

▷ **Listing 1.1:** Synthesized C code for the sum of v_1 and v_2.

```
1 int32_t sum(int32_t v1, int32_t v2){
2   int32_t tmp1 = v1 >> 2;       //tmp1 is in the format Q3.29
3   int32_t tmp2 = v2 >> 1;       //tmp2 is in the format Q3.29
4   return tmp1 + tmp2;     //the result is in the format Q3.29
5 }
```

Multiple research works have been published that describe tools dedicated to fixed-point code synthesis for particular problems. For instance, the works of Revy *et al.* [Rev09], [MR11] and Lee *et al.* [LV09], [LCLV08] treat the particular case of polynomial evaluation. Jeannerod *et al.* [JJL12] extend Revy's work to elementary functions. For FIR and IIR filters, Hilaire *et al.* [LHD12], [LHD14] describe the inner working of a fixed-point generation tool. Finally, the SPIRAL research group[6] set itself the objective of automating code generation for various signal processing primitives. The team provides on-line generators for such blocks as FFTs [VP04], DCTs, and Viterbi decoders.

1.3.2 Floating-point to fixed-point code conversion

This second type of methodologies is based on source code transformation. The tool takes as input the source code of a floating-point program. This code may be annotated by the programmer to point out extra information on some variables that cannot be inferred from simply parsing the code. The tool proceeds either by simulating the code or by interpolation [KWCM98] to determine the fixed-point formats that should be assigned to every variable in the computation.

At the end of this procedure, a new code is output that contains no references to floating-point data-types. Figure 1.8 shows the variant of this methodology suggested by Sung *et al.* [KiKS96]. The tools that automate this process, often called float-to-fix tools, use compilation techniques that include parsing, annotating intermediate representations, and code generation.

[6] Information on project SPIRAL is available at http://spiral.net/.

Figure 1.8: The float-to-fix methodology suggested by Sung in [KiKS96].

The early works of Sung *et al.* [KiKS96] and [SK95] suggest a float-to-fix framework based on statistical information deduced from floating-point simulations. In [KWCM98], Willems *et al.* suggest to enhance Sung's method by allowing the user to annotate the source code. These annotations compensate for the fact that source-to-source transformation has no information on the mathematical problem the code is supposed to solve. Although floating-point simulation gives an enclosure of the range of variables, the developer may be aware of corner cases that rarely appear during simulations. Finally, Ménard *et al.* [MCCS02] suggest a float-to-fix tool that parses a program into a Control Data Flow Graph and proceeds by annotating this intermediate representation with range and precision information.

1.3.3 Summary and comparison of the two methodologies

The code generation solution does not provide backward compatibility with any floating-point codes. It is not fully automated in the sense that the programmer must find an adequate tool for the problem at hand. However, since it has a more thorough knowledge of the mathematical problem, it usually relies on analytic methods instead of lengthy simulations. As such, it can provide strict error bounds and have a faster synthesis time. On the other hand, float-to-fix conversion has the advantage of being automatically applicable to a wide variety of problems. It also fits better in the classical design flow where the prototyping phase is carried out on a desktop machine using numerical computing environments such as Matlab®. These environments are able to quickly provide a floating-point implementation and by using float-to-fix tools, the design flow is completely automated. Finally, some researchers suggest to use the best of the two worlds. For instance, the code synthesis methodology for numerical linear algebra blocks suggested by Frantz *et al.* [NNF07] simulates a floating-point version but still uses some knowledge on the input problem.

Figure 1.9 shows the fixed-point synthesis flow suggested by the DEFIS (DEsign of FIxed-point embedded Systems) project and presented in [MRS+12]. DEFIS' architecture is flexible since it decides depending on user annotations whether to use fixed-point generation or float-to-fix C code conversion. As shown in Figure 1.9, both cases have major steps in common. These steps include IWL and FWL determination, that is, range and precision analyses. Indeed, the essential step in generating fixed-point code is determining the fixed-point formats. To do so, one must determine the integer and fraction parts of

each variable. The integer part is determined using range analysis while the fraction part is determined using precision analysis.

The rest of this chapter describes the state of the art methods for these two major steps.

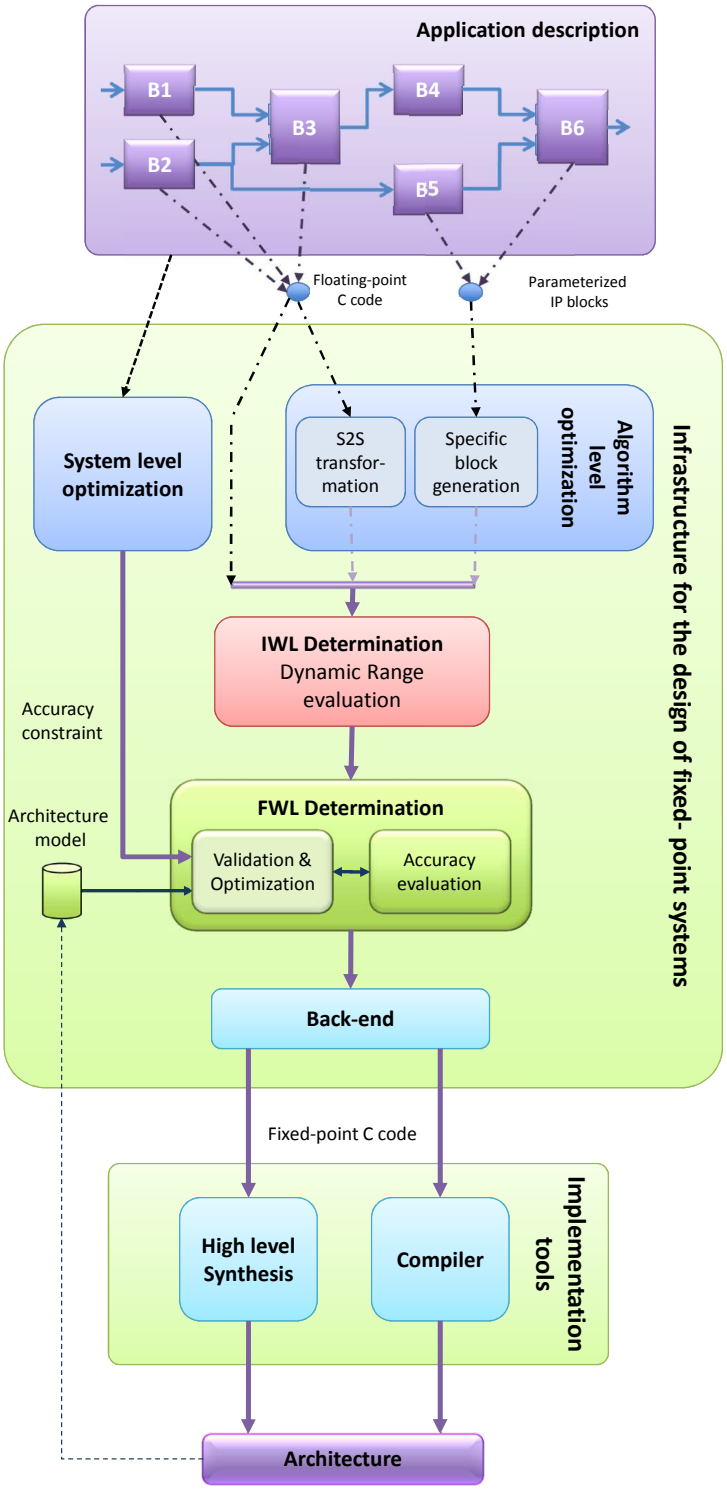


Figure 1.9: The flow of the automated design suggested by the DEFIS project.

1.3.4 Range analysis

Range analysis is the process of determining an enclosure of the values that every numeric variable in the program can hold. This phase is crucial in a fixed-point design since it allows to determine the width of the integer part of fixed-point variables.

Indeed, if as the result of range analysis, the enclosure of a signed variable v is determined to be $[a,b]$, then the width i of its integer part is given by:

$$i = \left\lceil \log_2 \left(\max \left(|a|, |b| \right) \right) \right\rceil + \alpha \tag{1.13}$$

where

$$\alpha = \begin{cases} 1, & \text{if} \quad \mod \left(\log_2(b), 1 \right) \neq 0, \\ 2, & \text{if} \quad \mod \left(\log_2(b), 1 \right) = 0. \end{cases} \tag{1.14}$$

These formulas derived by Lee *et al.* [LGC+06] take into account all the corner cases inherent to two's complement arithmetic.

With fixed word-length numbers, determining the width of the integer part imposes the width of the fraction part. In such contexts, the sharpness of the enclosure computed by the range analysis phase has a great impact on the numerical accuracy of the design. While a large integer part is desirable to avoid overflow, imposing a smaller fraction part negatively impacts the accuracy.

In the following, we present a representative set of techniques for range analysis and compare their behavior on the evaluation of a degree-3 polynomial function:

$$P(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 = a_0 + \Big(x \cdot \big(a_1 + x \cdot (a_2 + x \cdot a_3) \big) \Big) \tag{1.15}$$

where the input variable x belongs to the enclosure $[-1.75, 1.9]$ and where the value of the coefficients a_0, a_1, a_2, and a_3 is given by Table 1.4.

	x	a_0	a_1	a_2	a_3
Value	$[-1.75, 1.9]$	1.133003325617080	−1.949537446612336	−1.25869956	1
Format	$\mathbf{Q}_{2,30}$	$\mathbf{Q}_{2,30}$	$\mathbf{Q}_{2,30}$	$\mathbf{Q}_{2,30}$	$\mathbf{Q}_{2,30}$

Table 1.4: The range of the input and of the coefficients of the polynomial function (1.15) and their fixed-point formats.

The `C` program shown in Listing 1.2 corresponds to the Horner scheme evaluation of this function using binary64 floating-point numbers. This code is written in a three address coding style so that, as shown in Table 1.5, each intermediate variable v_0 through v_5 corresponds to a sub-expression of the polynomial function of Equation (1.15).

▷ **Listing 1.2:** C code for the Horner scheme evaluation of the expression (1.15)

```
double horner(double* a, double x){
  double v0 = x * a[3];
  double v1 = v0 + a[2];
  double v2 = x * v1;
  double v3 = v2 + a[1];
  double v4 = x * v3;
  double v5 = v4 + a[0];
  return v5;
}
```

Variable	Corresponding polynomial
v_0	$a_3 x$
v_1	$a_2 + a_3 x$
v_2	$a_2 x + a_3 x^2$
v_3	$a_1 + a_2 x + a_3 x^2$
v_4	$a_1 x + a_2 x^2 + a_3 x^3$
v_5	$a_0 + a_1 x + a_2 x^2 + a_3 x^3$

Table 1.5: The mapping between the intermediate variables of Listing 1.2 and the sub-expressions of the polynomial function of Equation (1.15).

Range analysis by fixed-point format propagation

Occasional fixed-point programmers who do not dispose of range analysis tools usually rely on format propagation. The method does not involve advanced numerical computations and is based on simple rules of thumb. It is described in detail by Yates [Yat13] and is used by Lopez *et al.* [LHD12] to implement linear filters. The idea is to start from the format of the input variables and to use propagation rules to determine the format of intermediate and output variables. Consider for instance the following propagation rules adapted to a design with 32-bit fixed-point variables:

$$\begin{aligned} \mathbf{Q}_{a,b} + \mathbf{Q}_{c,d} &\rightarrow \mathbf{Q}_{max(a,c)+1,32-(max(a,c)+1)} \\ \mathbf{Q}_{a,b} - \mathbf{Q}_{c,d} &\rightarrow \mathbf{Q}_{max(a,c)+1,32-(max(a,c)+1)} \\ \mathbf{Q}_{a,b} \times \mathbf{Q}_{c,d} &\rightarrow \mathbf{Q}_{a+c-1,32-(a+c-1)} \end{aligned} \tag{1.16}$$

The first and second rules state that the width of the integer part of the sum or difference of two fixed-point variables corresponds to the largest integer part of the operands augmented by 1, to avoid overflow. The third rule states that the width of the integer part of a product is the sum of the width of the integer part of the operands and that one bit should be removed from the resulting integer part. The removed bit is a redundant sign bit that results from the multiplication of two signed values. [7]

Example 1.4. Applied to the polynomial evaluation of Listing 1.2, the method yields the fixed-point formats shown by the tree structure of Figure 1.10.

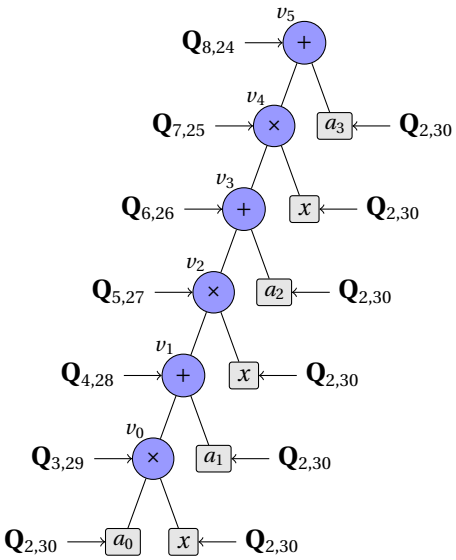


Figure 1.10: Fixed-point formats for the variables of Listing 1.2 obtained by applying the propagation rules (1.16).

The fixed-point formats obtained using this method are often conservative for the following reasons:

1. The fixed-point formats carry less information on the range of variables than intervals.
2. The propagation rules are not sensitive to data correlations.
3. The propagation rules are conservative.

[7] Section 2.3 further describes this detail.

Range analysis using interval arithmetic

Interval arithmetic is a method for bounding the values of computations that was formalized by Moore [Moo66, MKC09a] around the 1960's. An interval is defined[8] by a lower bound a and an upper bound b, and is generally denoted $[a,b]$: an element x is said to belong to $[a,b]$ if $a \leq x \leq b$. By properly defining operations on the endpoints, one can efficiently compute enclosures for the sum, product, and other operations on intervals. For instance, the rules for the basic operations are given by:

$$\begin{aligned}
x \in [a,b] &\rightarrow (-x) \in -[a,b] = [-b,-a] \\
x \in [a,b] \wedge y \in [c,d] &\rightarrow (x+y) \in [a,b]+[c,d] = [a+c,b+d] \\
x \in [a,b] \wedge y \in [c,d] &\rightarrow (x-y) \in [a,b]-[c,d] = [a-d,b-c] \\
x \in [a,b] \wedge y \in [c,d] &\rightarrow (x \times y) \in [a,b] \times [c,d] = [min(ac,bc,ad,bd), max(ac,bc,ad,bd)]
\end{aligned}$$

and are illustrated graphically on Figure 1.11.

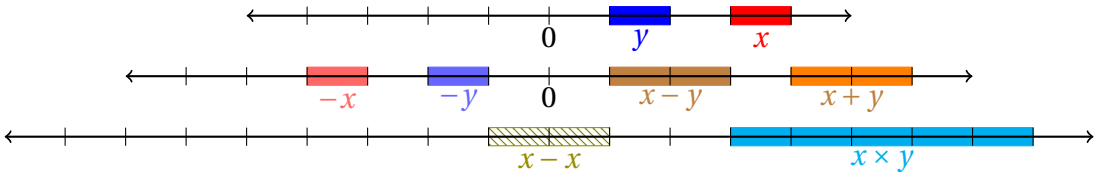


Figure 1.11: Illustration of the interval arithmetic rules.

Apart from being efficient, interval arithmetic is a non-intrusive range analysis method. By using object-oriented techniques such as polymorphism and operator overloading, transforming a program into an equivalent one that uses interval arithmetic requires minor modifications to the original source code.

However, the main drawback of this method is the data dependency problem [Han75]. Being entirely a numerical method, interval arithmetic is not sensitive to correlations between variables. This problem is exemplified by the computation of $x-x$. In interval arithmetic, if $x \in \mathcal{J} = [a,b]$ then $(x-x) \in \mathcal{J}-\mathcal{J} = [a-b,b-a]$ as shown in Figure 1.11. On designs involving multiple correlations, the data dependency problem leads to conservative bounds.

In theory, intervals can be defined on ordered sets such as the integers, rationals, and real numbers. But, practical implementations mostly use integer or floating-point numbers to store and compute with endpoints. In such case, care must be taken to use the appropriate rounding modes. For instance, directed rounding shall be used on the bounds to preserve the inclusion property.

[8] There is an alternative formalization that represents an interval using its center and radius. The major difference between the two representations resides in the efficiency of certain interval operations.

The Boost interval library [BMP06] uses C++ templates to provide generic intervals. For intervals with floating-point endpoints, the MPFI library [CLN+] (see also [RR05]) relies on the MPFR multiple precision floating-point package [FHL+07]. This library is used for range analysis in the fixed-point synthesis tool CGPE [Rev09], [MR11].

Example 1.5 below applies interval arithmetic to the polynomial evaluation of Listing 1.2.

Example 1.5. Apart from the obtained intervals, Table 1.6 also shows the fixed-point formats deduced from them using Equation (1.13).

Variable	Range obtained using interval arithmetic	Format
x	$[-1.75000000000000, 1.90000000000000]$	$\mathbf{Q}_{2,30}$
v_0	$[-1.75000000000000, 1.90000000000000]$	$\mathbf{Q}_{2,30}$
v_1	$[-3.00869956000001, 0.64130044000001]$	$\mathbf{Q}_{3,29}$
v_2	$[-5.71652916400001, 5.81272423000001]$	$\mathbf{Q}_{4,28}$
v_3	$[-4.71652916400001, 6.81272423000001]$	$\mathbf{Q}_{4,28}$
v_4	$[-14.56552656016345, 14.45586656857160]$	$\mathbf{Q}_{5,27}$
v_5	$[-13.43252323454637, 15.58886989418868]$	$\mathbf{Q}_{5,27}$

Table 1.6: Ranges and fixed-point formats of the intermediate variables of (1.2) computed using interval arithmetic.

These formats are less conservative than those obtained by format propagation. Indeed, for v_5, interval arithmetic yields a format with an integer part of 5 bits instead of 8 bits for format propagation. Yet, as illustrated by Figure 1.12, the enclosure computed using interval arithmetic for v_5 over-approximates the final range of the computation. This is due to the data dependency problem.

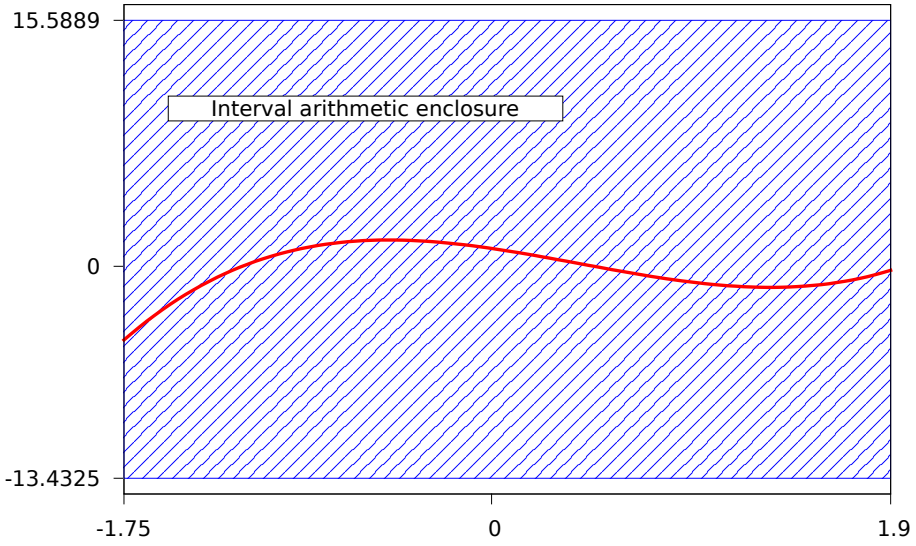


Figure 1.12: Computed enclosure on v_5 using interval arithmetic (in blue) and curve of the polynomial function (1.15) (in red).

Range analysis via affine arithmetic

Affine and generalized interval arithmetics [Han75],[CG+09] resulted from an effort to solve the problems due to the data-dependency issue inherent to interval arithmetic. It is used by Fang *et al.* [FRC03] to determine the range of variables in digital signal processing applications (FIR and IDCT). They report a slight improvement over interval arithmetic. In [LGC+06], Lee *et al.* use it for polynomial evaluation and RGB to YCbCr transformation.

To solve the data dependency problem, an affine variable x is represented as a combination of numeric values x_i and symbolic values ϵ_i as follows:

$$x = x_0 + x_1 \cdot \epsilon_1 + x_2 \cdot \epsilon_2 + \cdots + x_n \cdot \epsilon_n \tag{1.17}$$

where the variables ϵ_i, for $1 \leq i \leq n$ are in the interval $[-1,1]$. A variable $y \in \mathscr{I} = [a,b]$ is converted to the affine form $y = y_0 + y_1 \cdot \epsilon_1$ by computing $y_0 = \frac{a+b}{2}$ and $y_1 = \frac{b-a}{2}$. In this case, y_0 and y_1 are the center and radius of the interval $\mathscr{I}$, respectively, and ϵ_1 is a symbolic value that takes its values in $[-1,1]$. The opposite operation that converts an affine variable into an interval is called a reduction, and consists in replacing the ϵ_i, for $1 \leq i \leq n$, by $[-1,1]$ and applying basic interval arithmetic operations. The main feature of affine variables is that summing or subtracting them leads to affine forms where cancellations occur between symbolic variables.

Example 1.6. Consider the computation of a bound on $x - y$ where x and y are such that $y = x + 1$ and $x \in [-4, 5]$. In interval arithmetic, we have that $y \in [-3, 6]$ and $(x - y) \in [-10, 8]$. In affine arithmetic, the affine representation of x is $x = 0.5 + 4.5 \cdot \epsilon_x$ and that of y is $y = x + 1 = 1.5 + 4.5 \cdot \epsilon_x$. Then, we have $(x - y) = (0.5 - 1.5) + \epsilon_x \cdot (4.5 - 4.5) = -1 + 0 \cdot \epsilon_x$. Reducing this affine form yields the point interval $[-1, -1]$.

Example 1.6 shows how affine arithmetic solves the data dependency problem when working with summations. However, since the product of two affine forms is not an affine form, multiplication causes some correlations to be lost. This is obvious in Example 1.7 below which applies affine arithmetic to the polynomial evaluation of Listing 1.2. The example was implemented using the *libaffa* library developed by Gray and based on the work of Stolfi *et al.* [dFS04].

Example 1.7. The enclosures obtained by running affine arithmetic on the polynomial evaluation of Listing 1.2 are shown in Table 1.7. As illustrated in Figure 1.13, the improvement over interval arithmetic obtained for this example is not significant. No reduction in the format of any of the fixed-point variables v_0 through v_5 was obtained. This is due to the loss of correlations induced by multiplications in polynomial evaluation.

Variable	Range using affine arithmetic	Format
x	$[-1.75000000000000, 1.90000000000000]$	$\mathbf{Q}_{2,30}$
v_0	$[-1.75000000000000, 1.90000000000000]$	$\mathbf{Q}_{2,30}$
v_1	$[-3.00869956000000, 0.64130043999999]$	$\mathbf{Q}_{3,29}$
v_2	$[-5.44277916400000, 5.26522423000000]$	$\mathbf{Q}_{4,28}$
v_3	$[-7.39231661061233, 3.31568678338766]$	$\mathbf{Q}_{4,28}$
v_4	$[-14.04540156016343, 13.73965432312158]$	$\mathbf{Q}_{5,27}$
v_5	$[-12.91239823454636, 14.87265764873866]$	$\mathbf{Q}_{5,27}$

Table 1.7: Ranges and fixed-point formats of the intermediate variables of Listing 1.2 computed using affine arithmetic.

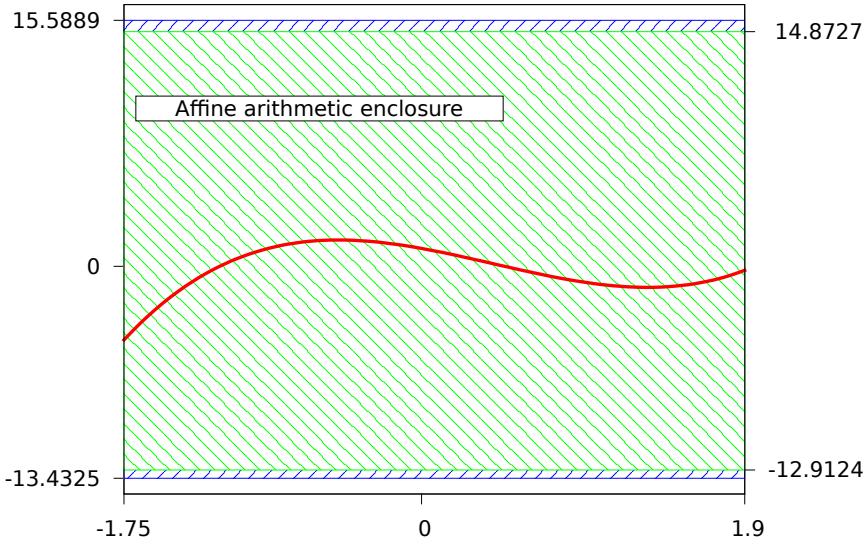


Figure 1.13: Computed enclosure on v_5 using affine arithmetic (in green) and curve of the polynomial function of Equation (1.15) (in red).

Range analysis of polynomial functions by differentiation

This analytic range analysis method is based on differentiation. It was suggested by Villasenor *et al.* [LV09] who used it to implement univariate polynomial evaluation. It relies on the following observation: when evaluating a polynomial, each intermediate variable in the design holds the value of a sub-expression of the final polynomial. These sub-expressions are themselves polynomials whose local minima and maxima can be determined by computing the roots of their derivatives. Indeed, Algorithm 1.1 shows how to compute the maximum and minimum of a polynomial function on a given interval.

This algorithm uses the classical process of locating the roots of the derivative polynomial to isolate the extrema. Indeed, these extrema either correspond to the endpoints of the input interval or to the roots of the derivative polynomial. For polynomials, this method yields the tightest possible certified range for the intermediate variables.

In Example 1.8 below, the differentiation method is applied to the polynomial evaluation of Listing 1.2.

Example 1.8. The differentiation technique yields the enclosures and fixed-point formats shown in Table 1.8. For each of the variables v_3, v_4 and v_5, the fixed-point format has an integer part that is one bit tighter than that obtained using interval and affine arithmetics.

Algorithm 1.1 Compute the extrema of a polynomial on an interval.

Require: a polynomial function P, an interval $[a,b]$.
Ensure: $[c,d]$ such that $P(x) \in [c,d]$ for $x \in [a,b]$.

```
P' ← the derivative of P
ℛ ← Roots(P')
ℛ ← insert(a, ℛ)
ℛ ← insert(b, ℛ)
𝒮 = ∅
for r ∈ ℛ do
    if r ∈ [a, b] then
        insert(P(r), 𝒮)
    end if
end for
c ← min(𝒮)
d ← max(𝒮)
```

Variable	Polynomial	Range	Format
x	x	$[-1.75000000000000, 1.90000000000000]$	$\mathbf{Q}_{2,30}$
v_0	$a_3 x$	$[-1.75000000000000, 1.90000000000000]$	$\mathbf{Q}_{2,30}$
v_1	$a_2 + a_3 x$	$[-3.00869956000001, 0.64130044000001]$	$\mathbf{Q}_{3,29}$
v_2	$a_2 x + a_3 x^2$	$[-0.39608114558605, 5.26522423000001]$	$\mathbf{Q}_{4,28}$
v_3	$a_1 + a_2 x + a_3 x^2$	$[-2.34561859219839, 3.31568678338767]$	$\mathbf{Q}_{3,29}$
v_4	$a_1 x + a_2 x^2 + a_3 x^3$	$[-5.80245187092842, 0.53541226965402]$	$\mathbf{Q}_{4,28}$
v_5	$a_0 + a_1 x + a_2 x^2 + a_3 x^3$	$[-4.66944854531134, 1.66841559527110]$	$\mathbf{Q}_{4,28}$

Table 1.8: The ranges obtained for the intermediate variables of Listing 1.2 using the differentiation method.

A comparison of the range of v_5 obtained by differentiation with those obtained by interval and affine arithmetics is shown in Figure 1.14. This figure also shows how the differentiation method returns the tightest possible enclosure.

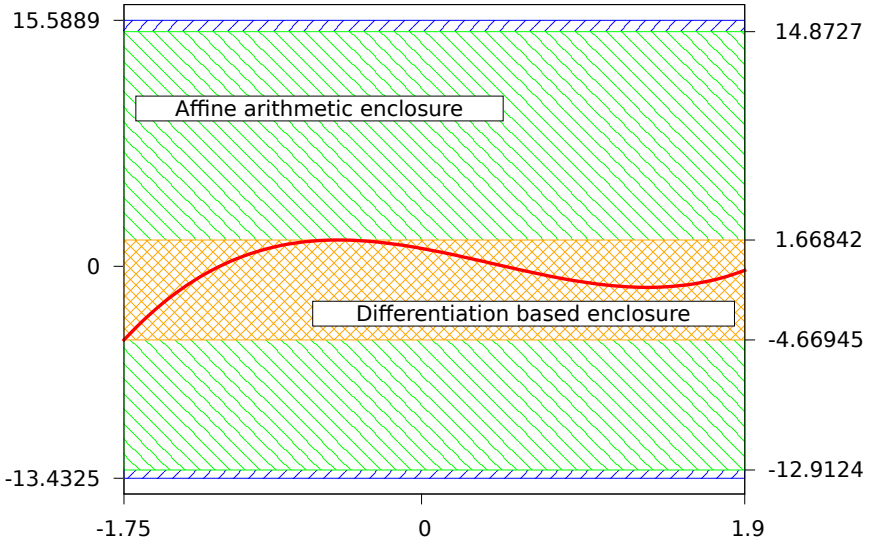


Figure 1.14: Comparison of the enclosure on v_5 obtained by differentiation (in orange) with the enclosures obtained by interval and affine arithmetic (in green and blue respectively) for the polynomial function (1.15).

Range analysis for linear filters using the ℓ_1-norm

Linear time invariant systems such as Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters are widely used as basic blocks in digital signal processing [PV08]. These filters take an impulse as input and produce another impulse as output for each time step, and are described by the following state equations:

$$\begin{aligned} x(k+1) &= A \cdot x(k) + \beta \cdot u[k], \quad x(0) = 0, \\ y[k] &= c \cdot x(k) + d \cdot u[k] \end{aligned} \tag{1.18}$$

where $u[k]$ and $y[k]$ are the input and output, and $x(k) \in \mathbb{R}^n$ is the state of the system at step k. The response of such system is determined by

$$h(k) = \begin{cases} d & k = 0, \\ c \cdot A^{k-1} \beta & k > 0, \end{cases} \tag{1.19}$$

and the largest possible value of the output impulse y is bounded as follows

$$\|y\|_\infty \leq \|H\|_{\ell_1} \cdot \|u\|_\infty \tag{1.20}$$

where $||H||_{\ell_1}$ is the ℓ^∞-gain of the system [BB92].

Boyd *et al.* [BB92] suggest an algorithm to compute a certified bound for

$$\|H\|_{\ell_1} = \sum_{k=0}^{\infty} |h(k)|.$$

However, this algorithm is tedious to implement and the bound is often approximated in practice by the finite sum $\sum_{k=0}^{N} |h(k)|$ where N is a large integer.

Example 1.9. Consider the following transfer function of a 4-taps filter with a cutoff frequency of $0.3 \cdot \pi$:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3}}{1 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3}}, \tag{1.21}$$

where the b_i and a_j coefficients are given in Table 1.9.

	Int. Repr.	**Format**	**Decimal value**
b_0	1701940795	$\mathbf{Q}_{-3,35}$	0.049532996345078572630882263183593750
b_1	1276455597	$\mathbf{Q}_{-1,33}$	0.148598989122547209262847900390625
b_2	1276455597	$\mathbf{Q}_{-1,33}$	0.148598989122547209262847900390625
b_3	1701940795	$\mathbf{Q}_{-3,35}$	0.049532996345078572630882263183593750
a_1	−1247599398	$\mathbf{Q}_{2,30}$	−1.161917483434081077575683593750
a_2	1494525688	$\mathbf{Q}_{1,31}$	0.695942755788564682006835937500
a_3	−1183360567	$\mathbf{Q}_{-1,33}$	−0.137761301244609057903289794921875

Table 1.9: The coefficients of the Butterworth filter of Equation (1.21).

This filter uses fixed-point coefficients and is an approximation of a 3rd order Butterworth filter. Its direct form realization shown in Figure 1.15 is one of many possible means to implement this filter using delay blocks, adders, and multipliers.

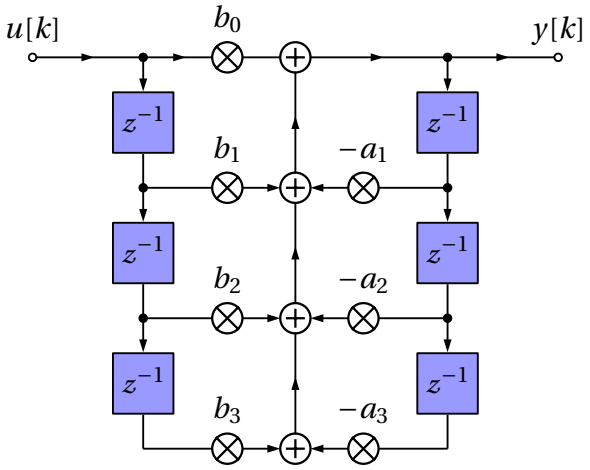


Figure 1.15: Block diagram of a direct form implementation of the Butterworth filter of Equation (1.21).

Converting this filter to the state equations in the form (1.18) yields the parameters shown in Table 1.10.

A	β	c	d
$\begin{pmatrix} a_1 & a_2 & a_3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \end{pmatrix}^T$	$\begin{pmatrix} a_1 - b_1 a_0 & a_2 - b_2 a_0 & a_3 - b_3 a_0 \end{pmatrix}$	b_0

Table 1.10: State space parameters of the linear filter (1.21).

By using the formula of Equation (1.19) and the coefficients of Table 1.10, the approximation

$$\|H\|_{\ell_1} \approx \sum_{k=0}^{1024} |h(k)|$$

yields the gain 1.2595080665573077. In this case, if the magnitude of the input pulse $u[k]$ is guaranteed to be in the enclosure $\mathscr{U} = [-15.5, 15.5]$, then the output $y[k]$ is also guaranteed to be in the interval

$$\mathscr{Y} = \|H\|_{\ell_1} \cdot \mathscr{U} = [-19.52237503163826936, 19.52237503163826936].$$

Using Equation (1.13), it is straightforward to deduce that the appropriate 32-bit format for $y[k]$ is $\mathbf{Q}_{6,26}$.

Simulation based range analysis

Simulation based methods [SK95], [NNF07] are used in commercial tools such as Matlab's Fixed-Point Designer®, Synopsys CoCentric Fixed-point Designer® or Mentor Graphics Catapult C®. These methods gather range information by extensively running a slightly modified floating-point design. The modifications change the type of floating-point variables to a class that stores its value each time a method is called on it. Examples of such class are `fSig` described by Sung [SK95] and `ti_float` [NNF07]. These classes take advantage of object-oriented techniques, particularly operators overloading of operations such as $+, -, \times$, and $\div$, to be minimally intrusive towards the original code. Once the data-types modified, the design is run with several input signals to obtain a collection of values for each traced variable. From these values, statistical information such as the minimum, maximum, mean, and standard deviation are computed and are used to estimate the range of the variable. For this last step, Didier *et al.* [CDV12] investigate the theoretical aspects of inferring the range from statistical information.

Simulation based methods are non-intrusive and yield tight ranges contrarily to interval arithmetic. However, among its drawback is the lengthy simulations needed to obtain representative statistical information and the absence of guaranties on the produced enclosures.

Example 1.10. With a simulation of the program of Listing 1.2 based on 1 000 runs, the minimum and maximum values for each intermediate variable are shown in Table 1.11.

Variable	Range	Format
v_0	$[-1.749204292241113, 1.899043597536648]$	$\mathbf{Q}_{2,30}$
v_1	$[-3.007903852241113, 0.640344037536648]$	$\mathbf{Q}_{3,29}$
v_2	$[-0.396079979860254, 5.261438328988734]$	$\mathbf{Q}_{4,28}$
v_3	$[-2.345617426472590, 3.311900882376398]$	$\mathbf{Q}_{3,29}$
v_4	$[-5.793191238929927, 0.535412061984584]$	$\mathbf{Q}_{4,28}$
v_5	$[-4.660187913312846, 1.668415387601664]$	$\mathbf{Q}_{4,28}$

Table 1.11: The minimum and maximum of the intermediate variables of Listing 1.2 obtained by a 1 000-runs simulation.

Figure 1.16 shows how the final enclosure v_5 gets closer to that obtained by differentiation when the number of simulation runs increases. The enclosures obtained by simulation for polynomial evaluation are quite sharp even with relatively few runs. Indeed,

since univariate polynomial evaluation consumes only one variable as input, it lends itself easily to range analysis by simulation. And it is even affordable to exhaustively test all of the input values.

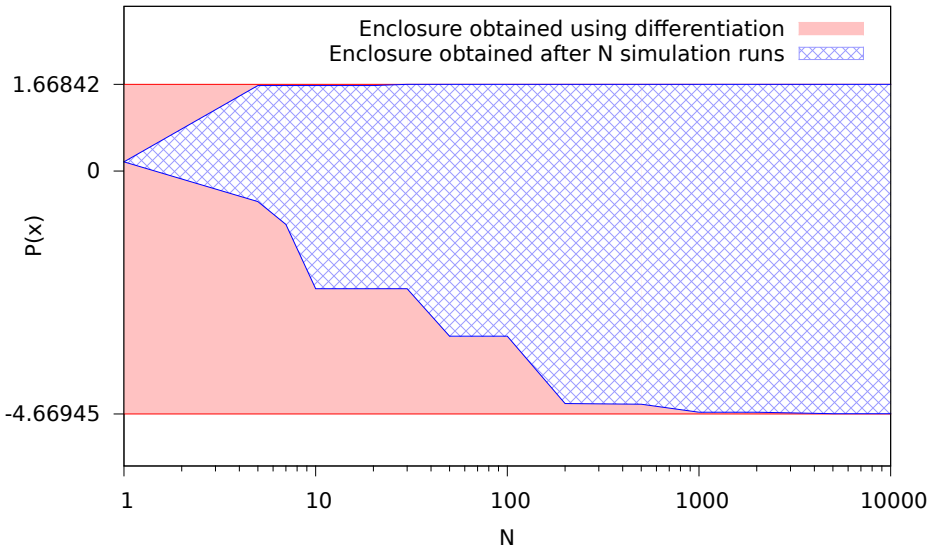


Figure 1.16: Evolution of the enclosure on v_5 with the number of simulation runs.

1.3.5 Precision analysis

Precision analysis is the second main step in automated fixed-point programming. Indeed, a fixed-point format is determined by the size of its integer and fraction parts. The goal of range analysis is to determine the size of the integer part and the goal of precision analysis is to determine the size of the fraction part.

Precision analysis with fixed-size word-length

If the target of the fixed-point design is a classical CPU or a fixed word-length DSP, then the word-length is imposed by the width of the data-path. In such case, setting the integer part of a fixed-point variable automatically determines the width of the fraction part. For instance, if v were a 32-bit fixed-point variable with an integer part of width 5, then the format will automatically be $\mathbf{Q}_{5,27}$. Although a meticulous precision analysis may prove that 20 bits of fraction part are sufficient, i.e., that the format $\mathbf{Q}_{5,20}$ is adequate, its mapping to software still imposes to use 32-bit integers and therefore, considering the format $\mathbf{Q}_{5,27}$ certainly does not minimize the fraction part but comes at no extra cost. Furthermore, the

experiments we shall present in Chapter 3 as well as in Part II assume a fixed size word-length and were conducted using 32 bit fixed-point numbers.

Precision analysis with custom word-length

When the target is an FPGA or an ASIC, the implementer has complete freedom over the word-length of every fixed-point variable in the design. In such case, minimizing the width of fraction parts saves resources. Indeed, one can anticipate narrower data-paths as well as compact operators (adders, multipliers, and dividers) which induce savings in the chip area, its power consumption, and therefore its cost. For this reason, precision analysis algorithms take a cost function and an accuracy criterion as input in addition to the detailed design. Their goal is to minimize the cost function while ensuring that the accuracy criterion is satisfied. This can be summarized by the following optimization problem:

- **Optimization problem:** find the vector $\mathbf{f}$ representing the combination of the widths of the fraction parts such that

$$\min \mathscr{C}(\mathbf{f}) \qquad \text{under the constraint} \qquad \mathscr{P}(\mathbf{f}) \geq \mathscr{B} \tag{1.22}$$

  where $\mathscr{C}$ is the cost function to minimize, $\mathscr{P}$ the accuracy of the design, and $\mathscr{B}$ is the accuracy bound, that is, the minimal accuracy allowed.

Depending on the design goals, the cost function $\mathscr{C}$ corresponds to energy consumption, the chip's area, or a combination of both.

The following of this section gives a brief survey of algorithms to solve this problem and starts by defining the sets $\mathbf{f}^{min}$ and $\mathbf{f}^{uni}$ that are used in many of these algorithms. These notations are borrowed from the work of Nguyen [Ngu11] which is an in-depth treatment of word-length optimization and precision analysis.

Definition 1.3.1. The distance between two vectors $\mathbf{f}^1$ and $\mathbf{f}^2$ representing the combination of the widths of the fraction parts is given by $d\left(\mathbf{f}^1,\mathbf{f}^2\right) = \sum_{i=0}^{n-1} \left|\mathbf{f}_i^1 - \mathbf{f}_i^2\right|$.

Definition 1.3.2. The minimum word-length set (MWS) is the vector $\mathbf{f}^{min} = \left(f_0^{min}, f_1^{min}, f_2^{min}, \ldots f_{n-1}^{min}\right)$ where f_k^{min} is obtained by minimizing only the fraction width of the k^{th} variable while allowing arbitrarily large word-lengths for the $n-1$ other variables. As such $\mathbf{f}^{min}$ rarely satisfies (1.22) but will often be used as a starting point.

Definition 1.3.3. The minimum uniform word-length set is the vector $\mathbf{f}^{uni} = \left(f^{uni}, f^{uni}, f^{uni}, \ldots f^{uni}\right)$ where f^{uni} is the smallest value such that $\mathscr{P}\left(\mathbf{f}^{uni}\right) \geq \mathscr{B}$ is satisfied.

Modified exhaustive search algorithms

Kum *et al.* [SK95] in one of the earliest float-to-fix conversion works suggest to solve the optimization problem in (1.22) using exhaustive search starting from the MWS. At iteration d of their algorithm, all vectors at distance d from MWS are tested. If a solution is found, the algorithm stops, otherwise iteration $d+1$ is executed and vectors at distance $d+1$ are considered. This algorithm has two drawbacks:

1. Depending on the cost function, it is not guaranteed to return the optimal solution. Indeed, if the found solution is at distance d from MWS, a solution that further minimizes $\mathscr{C}$ may exist at distance $d+1$. This is precisely the difference between this algorithm and classical exhaustive search.
2. The search is costly as soon as n and the distance d between MWS and the first acceptable solution are large. For instance, the number of times the procedure $\mathscr{P}$ must be tested is given by $\binom{n+d-1}{d-1}$. For the polynomial example of Listing 1.2 where $n=6$, if an acceptable solution is at distance $d_s=10$ from MWS, then $\binom{15}{9}=5005$ intermediate evaluations are needed.

Branch and bound algorithms. Branch and bound algorithms resemble exhaustive search, except that large classes of the search space are ignored depending on some criteria. Indeed, Burleson *et al.* [CB94] suggest to conduct a branch and bound algorithm between f^{min} and f^{uni}. The enhancement over exhaustive search consists in the following heuristic: as soon as a vector f^s does not satisfy the accuracy constraint, all the f^k that satisfy $f_i^k \leq f_i^s$ for $0 \leq i < n$ are discarded.

Greedy algorithms

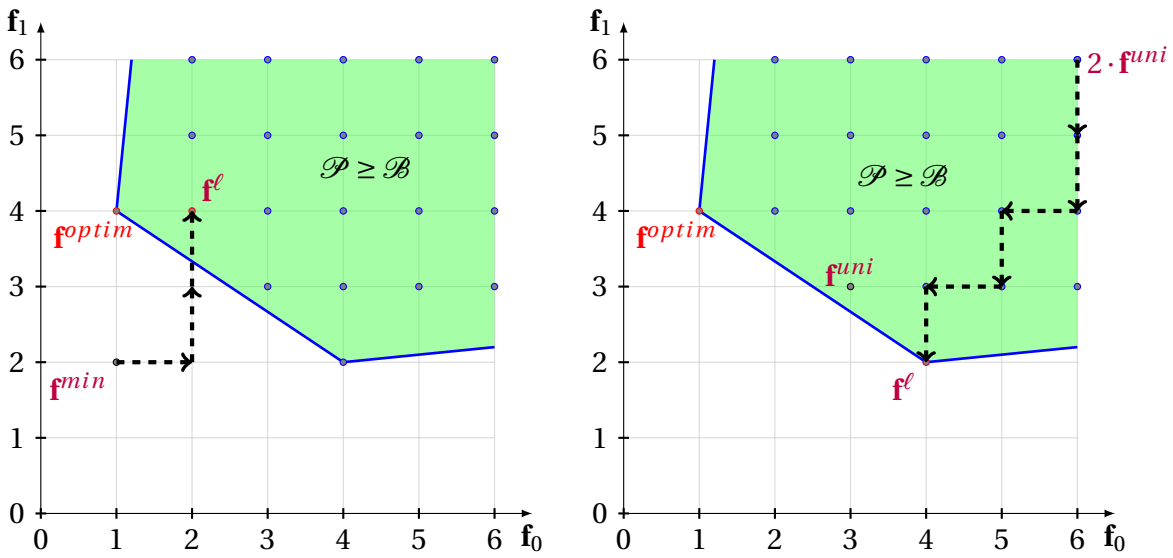
These algorithms perform faster than the exhaustive search by making local decisions to find a solution of the optimization problem.

Min+1 and max-1 algorithms. The min+1 algorithm [HEKC01] starts from MWS. At each step and as long as $\mathscr{P}(\mathbf{f}) < \mathscr{B}$, it temporarily increments each component of $\mathbf{f}$. The component that most augments the accuracy is permanently incremented, while the others are restored. The first solution $\mathbf{f}^\ell$ that verifies $\mathscr{P}(\mathbf{f}^\ell) \geq \mathscr{B}$ is returned.

The max−1 algorithm used by Constantinides [CCL01] to minimize the word-length in LTI systems is another greedy algorithm that starts from $k \cdot \mathbf{f}^{uni}$ for a well-chosen k. At each step, the components of $\mathbf{f}$ are decremented temporarily and the cost $\mathscr{C}$ is computed. Only the component that induces the largest decrease in $\mathscr{C}$ is permanently decremented.

Multiple variants of these algorithms have been suggested (See [Ngu11] for more details). They are faster than exhaustive search, however, they provide no guaranties on the optimality of the returned solution.

Example 1.11. Consider the cost function $\mathscr{C}(\mathbf{f}) = \mathbf{f}_0 + \mathbf{f}_1$ as well as the configuration shown in Figure 1.17. We have $\mathbf{f}^{\min} = (1,2)$ and $\mathbf{f}^{uni} = (3,3)$. The paths followed by the greedy algorithms to find a solution are shown by the dotted arrows and both of them miss the unique optimal solution since they do not use a backtracking or local search mechanism. For instance, in Figure 1.17a, once the step is taken from $\mathbf{f}^{min}$ to the position of coordinates $(2,2)$, there is no possibility to return back to a position of coordinates (n_1, n_2) with $n_1 < 2$ or $n_2 < 2$. The same applies to the $max-1$ algorithm shown in Figure 1.17b.



(a) Path followed by the min+1 algorithm.

(b) Path followed by the max−1 algorithm.

Figure 1.17: Word-length optimization in $\mathbb{Z}^2$ using the min+1 and max−1 algorithms.

Both algorithms end up finding a solution $\mathbf{f}^{\ell}$ such that $\mathscr{C}(\mathbf{f}^{\ell}) = 6$ while $\mathbf{f}^{optim}$ satisfies $\mathscr{C}(\mathbf{f}^{optim}) = 5$.

The GRASP algorithm. The GRASP (Greedy Randomized Adaptive Search Procedure) algorithm introduced by Ménard *et al.* [HNMS11] relies on a heuristic to correct the behav-

ior of greedy algorithms by adding a search step. The novelty of this algorithm is the Tabu search [Glo90] step conducted in the neighbourhood of each intermediate greedy solution to search for eventual better solutions. As shown in Figure 1.18, this algorithm follows closely the min+1 algorithm of Figure 1.17 except that the Tabu search conducted around each intermediate node ends up finding the optimal solution.

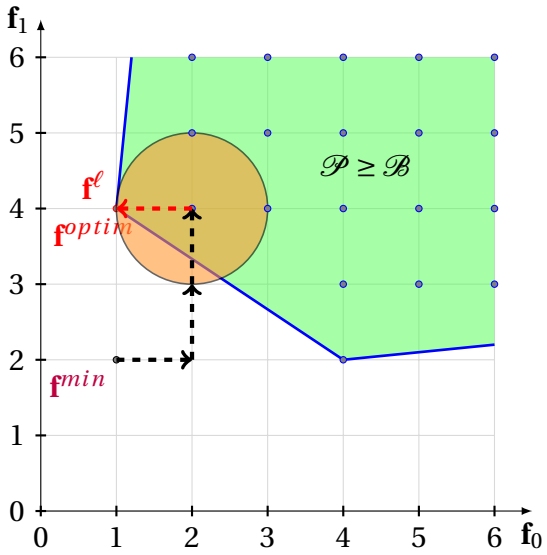


Figure 1.18: Word-length optimization in $\mathbb{Z}^2$ using the GRASP algorithm.

Stochastic algorithms

This category includes the Simulated Annealing algorithm (SA) and its variants such as Adaptive Simulated Annealing (ASA). These algorithms work by exploring the neighborhood of an intermediate solution to improve it while still allowing a probability to jump back to a worse solution. They are used for the precision analysis of polynomial evaluation by Cheung *et al.* [LGC+06] and for PID implementations by Chen *et al.* [CWIC99].

Finally, genetic algorithms are another variant of stochastic algorithms that have been suggested for precision analysis by Tang *et al.* [NTM92].

1.4 Conclusion

In this chapter, we introduced the fixed-point arithmetic. Compared to floating-point, fixed-point programming requires more expertise from the programmer who must keep

track of fixed-point formats and handle arithmetical details such as alignments and overflow prevention. To increase the quality of fixed-point codes and make fixed-point programming accessible to non-experts, automated techniques and tools are needed. These tools are built around two stages: range analysis and precision analysis, and a representative set of techniques for both stages was described in the second section of this chapter. In the next chapter, one of these techniques, namely interval arithmetic is discussed with great detail. The chapter shows how to use it to assert some properties on the generated code, such as its numerical quality.

Part I

A framework for the synthesis of certified fixed-point programs

CHAPTER 2

IMPLEMENTATION AND ERROR BOUNDS OF THE BASIC FIXED-POINT ARITHMETIC OPERATORS

In the previous chapter, we introduced the representation of fixed-point numbers and the state of the art methods for automating fixed-point programming. In this chapter, we present an interval arithmetic based framework whose goal is twofold: to determine the fixed-point formats through range analysis and to bound the rounding errors of fixed-point computations. Within this framework, we present the basic fixed-point arithmetic operators such as addition, multiplication, square root, and division. For each operator, we explain how to implement it using integer operations and how to bound its rounding errors using interval arithmetic.

2.1 Introduction

UNLIKE the floating-point arithmetic, there is no standard that governs most fixed-point implementations. Rather, it is customary for research articles on fixed-point arithmetic to start by a presentation of an arithmetic model. Here, we mean by arithmetic model the precise semantics of operations such as addition and multiplication,

and sometimes the mean to estimate the accuracy of these operators. Examples of arithmetic models include Fang *et al.*'s work [FRC03] which is based on affine arithmetic and Didier *et al.*'s [LHD12] which uses a probabilistic estimation of the propagation of noise.

Our arithmetic model is based on interval arithmetic which was described in Section 1.3.4. While introduced only as a range analysis technique in Chapter 1, we shall also use interval arithmetic to estimate the precision of our fixed-point implementations. Indeed, in this chapter, we show how to automate interval propagation for range analysis inside a fixed-point generation tool and how to use intervals to deduce strict bounds on the rounding errors that occur when evaluating arithmetic expressions.

To do so, for each operator $\diamond$ such that $\diamond \in \mathcal{O} = \{+, -, \times, \ll, \gg, \sqrt{}, /\}$, we first show how it propagates the ranges and rounding errors, then we describe its implementation by means of integer operations. To this end, we keep track of two intervals for every variable v in the design:

- $\mathbf{Val}(v)$ is an enclosure of the value of v, and
- $\mathbf{Err}(v)$ is an enclosure of the rounding errors entailed by the computation of v.

For each operator $\diamond$, we shall explicit the basic rules to compute $\mathbf{Val}(v)$ and $\mathbf{Err}(v)$ from $\mathbf{Val}(v_1), \mathbf{Val}(v_2), \mathbf{Err}(v_1)$ and $\mathbf{Err}(v_2)$, where v_1 is the first operand of $\diamond$ and v_2 the second operand if $\diamond$ is binary. To simplify our formulas, we also use the $\mathbf{Math}(v)$ interval which is related to $\mathbf{Val}(v)$ and $\mathbf{Err}(v)$ by the following equation:

$$\mathbf{Math}(v) = \mathbf{Val}(v) + \mathbf{Err}(v). \tag{2.1}$$

Conceptually, $\mathbf{Math}(v)$ is an enclosure one would obtain for v had all the computations been carried using infinite precision, i.e., without any rounding errors. Figure 2.1 illustrates our purpose: it shows the operator $\diamond$ as a node in a potentially large tree structure.

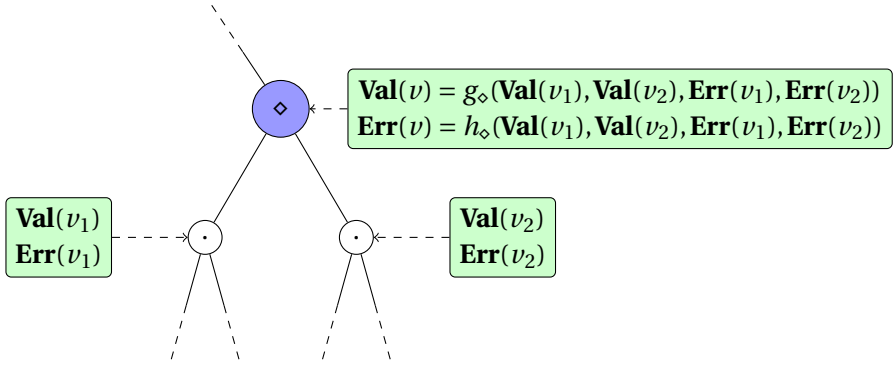


Figure 2.1: Node representation of the operator $\diamond$ and the relationships between $\mathbf{Val}(v), \mathbf{Err}(v)$ and $\mathbf{Val}(v_1), \mathbf{Val}(v_2), \mathbf{Err}(v_1), \mathbf{Err}(v_2)$.

The figure suggests that, once the functions $g_\diamond$ and $h_\diamond$ explicitly stated for all $\diamond \in \mathscr{O}$, determining $\mathbf{Val}(v)$ and $\mathbf{Err}(v)$ for every node in the tree representation of an arithmetic expression requires nothing more than propagating them by the means of a bottom up traversal. In the following sections, we give explicit formulas for $g_\diamond$ and $h_\diamond$ for the set of basic operators $\mathscr{O}$.

2.2 Addition and subtraction

Summation is ubiquitous in numerical programs. Linear functions, dot-products, average, and norm computations all involve computing additions. Subtraction is less common but when implemented in two's complement arithmetic, addition and subtraction use the same hardware, and subtracting a number reduces to adding its two's complement.

2.2.1 Ranges and error bounds for the addition/subtraction operator

In fixed-point arithmetic, two addends must be aligned in the same format $\mathbf{Q}_{i_s,f_s}$, that is, they must share the same integer and fraction part sizes. If this is not the case, then alignment operations should be applied to the addends before the addition node. These alignments are performed by means of the shift operators presented in Section 2.4. Authors that include Rutenbar *et al.* [FRC03] as well as some libraries such as Mentor Graphics'® `ac_fixed` [Bol08, §3] consider the alignment phase as part of fixed-point addition. Our choice not to follow this convention is motivated by keeping all the operations explicit and by making fixed-point addition coincide with integer addition.[1] Therefore, in our definition of addition and subtraction, alignments are not part of the operation.

It follows from this definition that addition and subtraction are error free and that we have for $\diamond \in \{+,-\}$:

$$\mathbf{Val}(v) = \mathbf{Val}(v_1) \diamond \mathbf{Val}(v_2). \tag{2.2}$$

As for the error bound, we have:

$$\begin{aligned} \mathbf{Err}(v) &= \mathbf{Math}(v) - \mathbf{Val}(v) \\ &= \big(\mathbf{Math}(v_1) \diamond \mathbf{Math}(v_2)\big) - \big(\mathbf{Val}(v_1) \diamond \mathbf{Val}(v_2)\big) \\ &= \big(\mathbf{Math}(v_1) - \mathbf{Val}(v_1)\big) \diamond \big(\mathbf{Math}(v_2) - \mathbf{Val}(v_2)\big) \\ &= \mathbf{Err}(v_1) \diamond \mathbf{Err}(v_2), \end{aligned} \tag{2.3}$$

that is the rounding error of a summation or subtraction node is nothing more than the sum or difference of the rounding errors on the operands. This is a consequence of the error free nature of these operations.

[1] Among others, this convention simplifies the instruction selection process detailed in Section 3.3.

2.2.2 Output format of the addition/subtraction operator

Since the operands of fixed-point summation/subtraction v_1 and v_2 are aligned, we consider without loss of generality that they are both in the $\mathbf{Q}_{i_1,f_1}$ format. Depending on the enclosure $\mathbf{Val}(v)$, v requires the following format:

1. $\mathbf{Q}_{i_1,f_1}$, if $\mathbf{Val}(v) \subseteq \mathscr{R}\text{ange}(\mathbf{Q}_{i_1,f_1})$,
2. $\mathbf{Q}_{i_1+1,f_1-1}$, if $\mathbf{Val}(v) \not\subseteq \mathscr{R}\text{ange}(\mathbf{Q}_{i_1,f_1})$, that is, if overflow occurs.

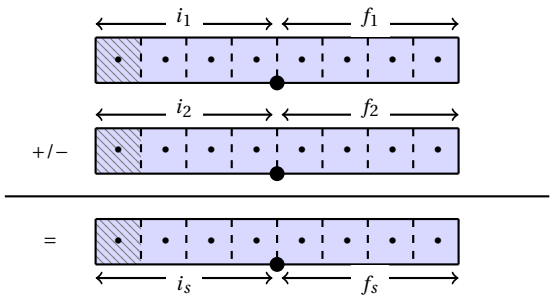


Figure 2.2: Addition or subtraction of two 8-bit aligned fixed-point variables.

When the output format is $\mathbf{Q}_{i_1,f_1}$

Case 1 occurs when the sum fits in $\mathscr{R}\text{ange}(\mathbf{Q}_{i_1,f_1})$ and is illustrated by Figure 2.2. A potential sub-case is when the addition or subtraction leads to cancellations as illustrated by Figure 2.3, where the three most significant bits of the result have the same value.

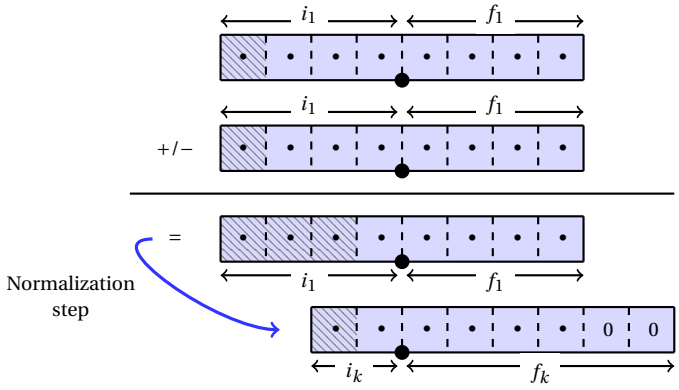


Figure 2.3: Sum or difference of two fixed-point variables involving cancellations and followed by a normalization step.

Clearly, the two most significant bits are redundant sign bits that can be removed by scaling the result to a smaller format than $\mathscr{R}\text{ange}(\mathbf{Q}_{i_1,f_1})$, that is, a format with an integer part of size i_k such that $i_k < i_1$. This alignment re-normalizes the result as shown in the bottom of Figure 2.3 and is handled by means of left shifting which is discussed in Section 2.4. Again, this normalization is not part of the operation of addition or subtraction itself.

When the output format is $\mathbf{Q}_{i_1+1,f_1-1}$

Case 2 happens when $\mathscr{R}\text{ange}(\mathbf{Q}_{i_1,f_1}) \subset \mathbf{Val}(v) \subset \mathscr{R}\text{ange}(\mathbf{Q}_{i_1+1,f_1-1})$. In this case, summing or subtracting the two variables causes overflow since the result is potentially too large to fit in a $\mathbf{Q}_{i_1,f_1}$ variable. The solution is to align the two operands before performing the addition or subtraction. That is, the addition or subtraction is transformed into an operation between two variables in the $\mathbf{Q}_{i_1+1,f_1-1}$ format by means of right shifting. This process is shown in Figure 2.4. Once again, aligning the two addends is not part of the addition/subtraction operation.

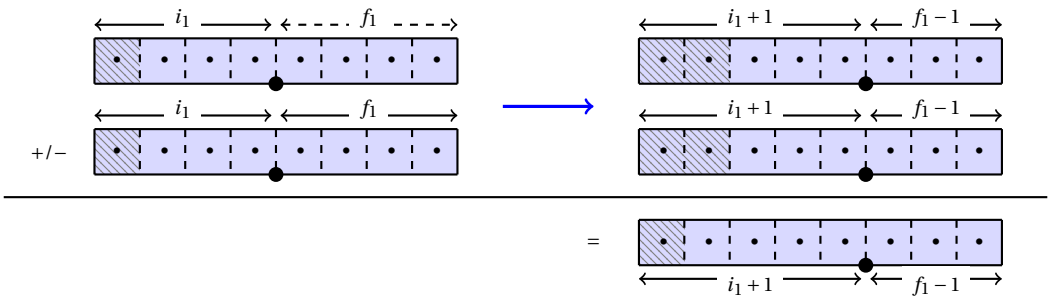


Figure 2.4: Sum or difference of two fixed-point variables involving alignments to avoid overflow.

2.2.3 Implementation of the addition/subtraction operator

As a result of our definition of fixed-point addition/subtraction as operations between two aligned operands, and of our caution to avoid the occurrence of overflow, computing these operations reduces to integer addition or subtraction of the integer representations. Indeed, we have that:

$$\begin{aligned} v_1 \diamond v_2 &= V_1 \cdot 2^{f_1} \diamond V_2 \cdot 2^{f_2} \\ &= V_1 \cdot 2^{f_1} \diamond V_2 \cdot 2^{f_1} \quad \text{since} \quad f_1 = f_2 \\ &= (V_1 \diamond V_2) \cdot 2^{f_1}. \end{aligned}$$

Listing 2.1 shows two `C` functions that emulate fixed-point addition and subtraction, respectively, for 32-bit signed fixed-point variables.

▷ **Listing 2.1:** C code for 32-bit fixed-point addition and subtraction.

```
int32_t add (int32_t v1, int32_t v2){
  return v1 + v2;
}

int32_t sub (int32_t v1, int32_t v2){
  return v1 - v2;
}
```

2.3 Multiplication

Unlike addition and subtraction, there are no constraints on the formats of multiplicands. Also, as a rule of thumb [Yat13], the product of v_1 and v_2 results in a fixed-point variable in the format $\mathbf{Q}_{i_r,f_r}$ with

$$i_r = i_1 + i_2 \quad \text{and} \quad f_r = f_1 + f_2, \tag{2.4}$$

as shown in Figure 2.5.

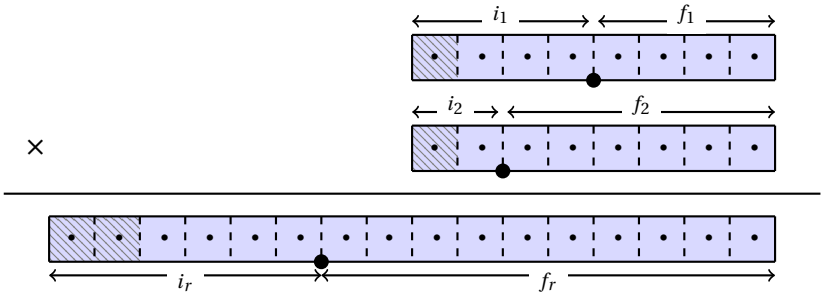


Figure 2.5: Output format of fixed-point multiplication.

This bit distribution is due to:

1. The largest possible multiplicands being -2^{i_1-1} and -2^{i_2-1}, the product could be as large as $2^{i_1+i_2-2}$.
2. The smallest multiplicands being 2^{-f_1} and 2^{-f_2}, the smallest product is $2^{-(f_1+f_2)}$.

The redundant sign bit

The MSB of Figure 2.5 is a redundant bit and can be discarded. Nevertheless an exception occurs in signed arithmetic when the operands are both equal to the smallest negative

value, that is, $v_1 = -2^{i_1-1}$ and $v_2 = -2^{i_2-1}$. In that case, only the most significant bit represents the sign of the result and it cannot be discarded. Also, in most practical cases, one cannot afford to double the word-length after every multiplication. Therefore, the multiplication we consider in the subsequent is truncated, i.e., only the upper half of the result is considered.

Special case of multiplication by powers of 2

Multiplication by a power of 2 is an interesting special case since it can be transformed into a virtual shift. A virtual shift just consists in changing the fixed-point format of the operand and does not imply any physical operation during run-time. Hence, once transformed into a virtual shift, multiplication by a power of 2 is cost-free, and does not impact the evaluation latency of the program. This optimization is implemented in the fixed-point synthesis tool CGPE, and is analogous to the constant folding facilities offered by modern compilers. As for the output format, the multiplication of v_1 by 2^p results in a fixed-point variable in the format $\mathbf{Q}_{i_r,f_r}$ with

$$i_r = i_1 + p \quad \text{and} \quad f_r = f_1 - p.$$

More in-depth treatment of virtual shifts is given in Section 2.4 which is dedicated to the description of the shifts operators.

2.3.1 Range and error bounds for the multiplication operator

For the output range of a multiplication we have the following:

$$\mathbf{Val}(v) = \mathbf{Val}(v_1) \cdot \mathbf{Val}(v_2) - \mathbf{Err}_\times, \tag{2.5}$$

where $\mathbf{Err}_\times$ accounts for the rounding error of truncating the product to fit the output format. As for the error term, we have:

$$\begin{aligned}
\mathbf{Err}(v) &= \mathbf{Math}(v) - \mathbf{Val}(v) \\
&= \big(\mathbf{Math}(v_1) \cdot \mathbf{Math}(v_2)\big) - \mathbf{Val}(v) \\
&= \Big(\big(\mathbf{Val}(v_1) + \mathbf{Err}(v_1)\big) \cdot \big(\mathbf{Val}(v_2) + \mathbf{Err}(v_2)\big)\Big) - \mathbf{Val}(v) \\
&= \mathbf{Val}(v_1) \cdot \mathbf{Val}(v_2) - \mathbf{Val}(v) + \mathbf{Val}(v_1) \cdot \mathbf{Err}(v_2) + \mathbf{Val}(v_2) \cdot \mathbf{Err}(v_1) + \mathbf{Err}(v_1) \cdot \mathbf{Err}(v_2) \\
&= \mathbf{Err}_\times + \mathbf{Val}(v_1) \cdot \mathbf{Err}(v_2) + \mathbf{Val}(v_2) \cdot \mathbf{Err}(v_1) + \mathbf{Err}(v_1) \cdot \mathbf{Err}(v_2). \qquad (2.6)
\end{aligned}$$

Notice how $\mathbf{Err}(v)$ depends not only on the errors of the operands, i.e., $\mathbf{Err}(v_1)$ and $\mathbf{Err}(v_2)$, but also on their range of values $\mathbf{Val}(v_1)$ and $\mathbf{Val}(v_2)$. This explains our assertion of Section 1.2.2 that maintaining sharp enclosures on the values yields tight error bounds.

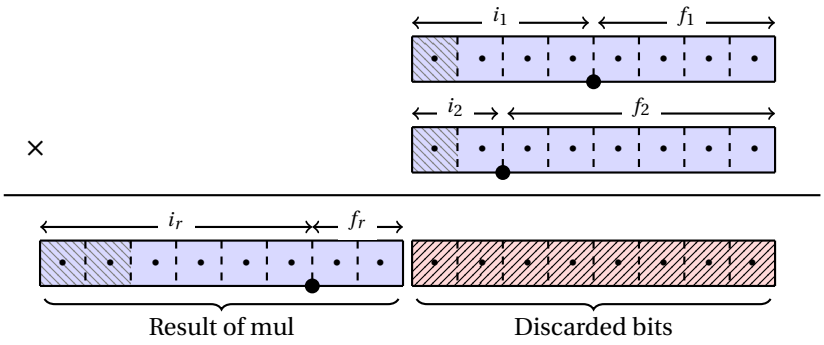


Figure 2.6: Fixed-point multiplication without word-length doubling.

2.3.2 Implementation of the multiplication operator

The implementation of fixed-point multiplication is highly dependent on the targeted hardware. When it comes to 32-bits arithmetic, targets such as STMicroelectronics integer processor `ST231` [ST208] are shipped with many variants of multiplication. Among these variants, `mul64h` for instance returns the upper half of the product of two 32-bits variables. This instruction is the one used by Revy [Rev09] for fixed-point multiplication when synthesizing code for polynomial evaluation.

Emulation of the instruction by doubling the word-length

In absence of a specific hardware target, the result of this instruction can be emulated using `C` code. Indeed, one of the possible 32-bit implementations is given by Listing 2.2. Its behavior is shown graphically by Figure 2.6 on 8-bit operands.

▷ **Listing 2.2:** `C` code for signed 32-bit fixed-point multiplication.

```
int32_t mul (int32_t v1, int32_t v2){
  int64_t prod = ((int64_t) v1) * ((int64_t) v2);
  return (int32_t) (prod >> 32);
}
```

The code of Listing 2.2 works by first computing a full product, that is, by doubling the word-length before truncating the lower half by the means of a right shift. With such a multiplier, the error term $\mathbf{Err}_\times$ of Equation (2.6) accounts for the discarded bits and is given by:

$$\mathbf{Err}_\times = \left[0, 2^{-f_r} - 2^{-(f_1+f_2)}\right]. \tag{2.7}$$

Emulation of the instruction without doubling the word-length

Listing 2.2 supposes that the hardware and programming environment are capable of computing the entire product, i.e., with word-length doubling. If no such feature is available or if it is too costly, Warren [Jr.02] describes an algorithm to obtain the same result with small multipliers. An implementation of this algorithm is shown in Listing 2.3 and allows to compute the upper part of a 32×32 multiplication using only multipliers that take 16-bit operands and yield 32-bit values.

▷ **Listing 2.3:** C code computing the upper half of 32-bit multiplication using only $16 \times 16 \rightarrow 32$ multipliers.

```
1  inline int32_t mul(int32_t a, int32_t b)
2  {
3    int16_t aLow = a & 0xFFFF; int16_t aHigh = a >> 16;
4    int16_t bLow = b & 0xFFFF; int16_t bHigh = b >> 16;
5    int32_t zLowLow = mul_16x16(aLow,bLow);
6    int32_t zLowHigh = mul_16x16(aLow,bHigh);
7    int32_t zHighLow = mul_16x16(aHigh,bLow);
8    int32_t zHighHigh = mul_16x16(aHigh,bHigh);
9    int32_t t = zHighLow + ( zLowLow >> 16 );
10   int32_t wLow = t & 0xFFFF;
11   int32_t wHigh = t >> 16;
12   int32_t w1 = zLowHigh + wLow;
13   return zHighHigh + wHigh + ( w1 >> 16 );
14 }
```

The different steps of Listing 2.3 are graphically depicted in Figure 2.7.

2.4 Physical and virtual shifts

Shifts are used in fixed-point programs for one of the following reasons:

1. to align the operands of addition or subtraction,
2. to normalize the result of an operation where cancellation occurred as shown in Figure 2.3, and
3. to replace a multiplication or a division by a power of 2 as discussed in Section 2.3.[2]

These 3 purposes imply 3 different kinds of shifts, namely right shifts, left shifts, and virtual shifts. Following Yates [Yat13], we classify these shifts into physical[3] and virtual shifts. Right and left shifts are considered physical shifts since they do require the run-time execution

[2] The remark is true when working with any radix. That is, to say that multiplying by a power of that radix comes down to a shift.

[3] Yates uses the term literal shift instead of physical shift.

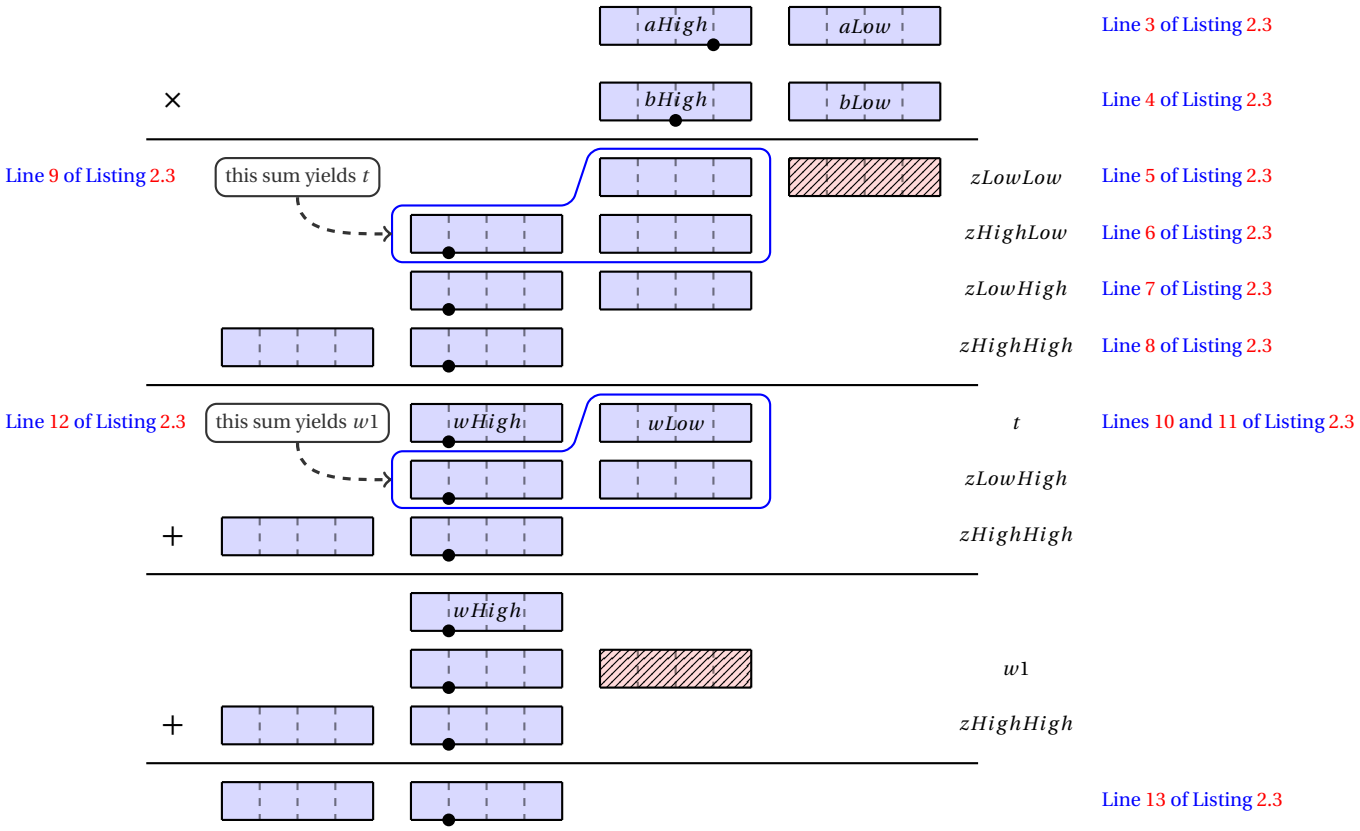


Figure 2.7: Detailed description of the steps involved in Warren's algorithm.

of a shifting instruction that may be coupled or not to a change of format. Virtual shifts on the other hand do not require any run-time operation, indeed they merely consist in a static change of the fixed-point format.

2.4.1 Range and error bounds for the shift operator

The 3 kinds of shifts have different range and error bounds, and are therefore treated separately in the following.

Right shift

Shifting the value v_1 by s positions to the right results in a fixed-point value in the format $\mathbf{Q}_{i_r,f_r}$ with

$$i_r = i_1 + s \quad \text{and} \quad f_r = f_1, \tag{2.8}$$

as shown in Figure 2.8 for $s = 2$.

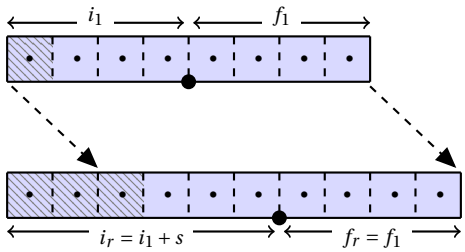


Figure 2.8: Error free right shift.

However, the shift shown in this figure is an error free variant where the word-length is also extended by s bits. In practice and similarly to the case of multiplication, one cannot always afford to extend the word-length. In such case, the leftmost excess bits allocated to the integer part by the right shift are reclaimed from the fraction part. A side effect of this strategy of keeping a fixed word-length is that right shift induces a rounding error as shown in Figure 2.9. Indeed, the rightmost two bits in this figure are "lost" as a side effect of the right shift.

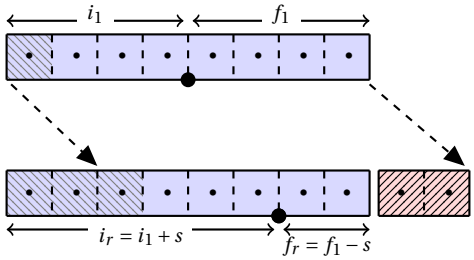


Figure 2.9: Right shift with fixed word-length.

This variant of right shifts produces the output format $\mathbf{Q}_{i_r,f_r}$ where

$$i_r = i_1 + s \quad \text{and} \quad f_r = f_1 - s. \tag{2.9}$$

Its value range is given by:

$$\mathbf{Val}(v) = \mathbf{Val}(v_1) - \mathbf{Err}_{\gg}, \tag{2.10}$$

where $\mathbf{Err}_{\gg}$ accounts for the rounding error of truncating the result and is bounded by the enclosure

$$\mathbf{Err}_{\gg} = \left[0, 2^{-f_r} - 2^{-f_1}\right] = \left[0, 2^{-f_1+s} - 2^{-f_1}\right]. \tag{2.11}$$

As for the bound on rounding error, it is given by:

$$\mathbf{Err}(v) = \mathbf{Math}(v) - \mathbf{Val}(v) = \mathbf{Math}(v_1) - \left(\mathbf{Val}(v_1) - \mathbf{Err}_{\gg}\right) = \mathbf{Err}(v_1) + \mathbf{Err}_{\gg}. \tag{2.12}$$

Left shift

Left shift is useful to get rid of redundant sign bits and its aim is to bring a variable into a smaller fixed-point format.

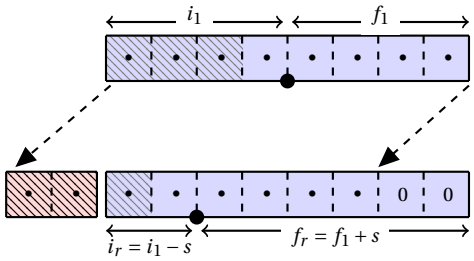


Figure 2.10: Left shift of 2 positions in presence of redundant sign bits.

As illustrated by Figure 2.10, left shift by s positions produces a variable with the same value but in the $\mathbf{Q}_{i_r,f_r}$ format where

$$i_r = i_1 - s \quad \text{and} \quad f_r = f_1 + s. \tag{2.13}$$

Since, in absence of overflow, left shift is error free and does not alter the value of a variable, we have:

$$\mathbf{Val}(v) = \mathbf{Val}(v_1). \tag{2.14}$$

and as a consequence, we have for the error bound:

$$\mathbf{Err}(v) = \mathbf{Math}(v) - \mathbf{Val}(v) = \mathbf{Math}(v_1) - \mathbf{Val}(v_1) = \mathbf{Err}(v_1). \tag{2.15}$$

Finally, to avoid altering the value of the variable, the new rightmost bits introduced by the left shift must be equal to 0. This is the case for the two LSB bits in Figure 2.10.

Virtual shifts

Virtual shift is merely a change in the format of a fixed-point variable. It is not only cost free but also error free. Right virtual shift $\gg_v$ replaces multiplication by powers of 2 while left virtual shift replaces division by powers of 2. Indeed, we have for $\diamond \in \{\gg_v, \ll_v\}$:

$$\mathbf{Val}(v) = \mathbf{Val}(v_1) \cdot 2^{(-1)^{\delta} \cdot s} \tag{2.16}$$

where δ is such that:

$$\delta = \begin{cases} 0, & \text{if} \quad \diamond = \ll_v, \\ 1, & \text{if} \quad \diamond = \gg_v . \end{cases}$$

For the error bound, we have:

$$\begin{aligned}\mathbf{Err}(v) &= \mathbf{Math}(v) - \mathbf{Val}(v)\\ &= \mathbf{Math}(v_1)\cdot 2^{(-1)^{\delta}\cdot s} - \mathbf{Val}(v_1)\cdot 2^{(-1)^{\delta}\cdot s}\\ &= \mathbf{Err}(v_1)\cdot 2^{(-1)^{\delta}\cdot s}. \qquad (2.17)\end{aligned}$$

Example 2.1. Consider the signed 8-bit fixed-point value v in the fixed-point format $\mathbf{Q}_{4,4}$ such that $V = (1010\,1101)_2$. The decimal value of v is $v = -83\cdot 2^{-4} = -5.1875$ and its graphical depiction is shown at the top of Figure 2.11.

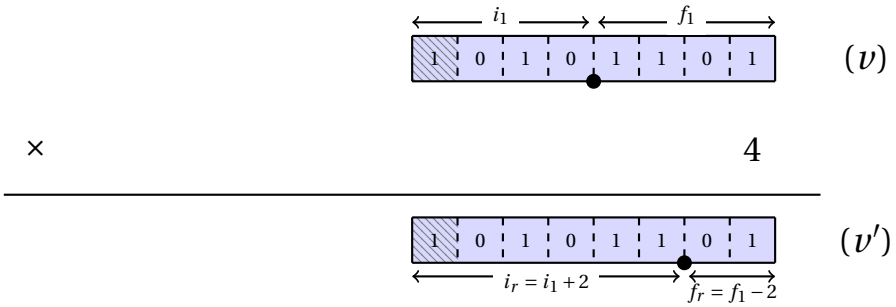


Figure 2.11: Virtual right shift by 2 positions replaces multiplication by 4.

Multiplying v by 4 yields v' such that $V' = V$ and such that the fixed-point format of v' is $\mathbf{Q}_{6,2}$. Hence, v' has the value $v' = V'\cdot 2^{-2} = -20.75$ as shown in Figure 2.11. This indeed corresponds to the value of v multiplied by 4.

2.4.2 Implementation of the shift operators

In code generation tools, if arithmetic expressions are represented as trees, then each intermediate node corresponds to a fixed-point variable. In such case, virtual shift do not involve any code generation and is just a matter of manipulating the fixed-point format of the intermediate variable.

On the other hand, left and right shifts involve generating code and care must be taken in making the generated code coincide with the semantics we exposed in Section 2.4.1. For instance, concerning left shift, standard C [Int10, § 6.5.7] states the following:

> The result of $E1 \ll E2$ is $E1$ left-shifted $E2$ bit positions; vacated bits are filled with zeros.

This is exactly the desired behaviour. Hence a 32-bit acceptable implementation of left shift is given by Listing 2.4.

▷ **Listing 2.4:** C code for 32-bit fixed-point left shifting.

```
int32_t left_shift (int32_t v1, int s){
  return v1 << s;
}
```

However, for right shift, the C standard states the following:

> The result of $E1 \gg E2$ is $E1$ right-shifted $E2$ bit positions. If $E1$ has an unsigned type or if $E1$ has a signed type and a non-negative value, the value of the result is the integral part of the quotient of $E1/2^{E2}$. If $E1$ has a signed type and a negative value, the resulting value is implementation-defined.

Clearly, the C standard avoids taking a two's complement approach and does not impose that "vacated bits" be sign extended. Therefore, C specifications are only satisfactory when working with unsigned fixed-point variables. The behaviour of right shift in presence of signed fixed-point variables is implementation defined. Indeed, about this issue, the manual of the widespread GCC compiler suite [Fou05, § 4.5] states the following:

> Bit-wise operators act on the representation of the value including both the sign and value bits, where the sign bit is considered immediately above the highest-value value bit. Signed $\gg$ acts on negative numbers by sign extension.

Contrarily to the standard, GCC follows a two's complement strategy and its users can therefore implement right shift for signed variables by the straightforward code shown in Listing 2.5.

▷ **Listing 2.5:** C code for 32-bit fixed-point right shifting.

```
int32_t right_shift (int32_t v1, int s){
  return v1 >> s;
}
```

2.5 Square root

Compared to the previous operators, square root received less attention from researchers working on fixed-point arithmetic. Yet, it is useful in many numerical basic blocks such as euclidean norms and Cholesky decomposition of symmetric positive-definite matrices. Indeed, it is essentially for the purpose of implementing code synthesis for Cholesky decomposition that we first formalized fixed-point square root in [MNR14b].

2.5.1 Output range and error bounds for square root

Square root is a unary operator which assumes that $v_1 \geq 0$. If this condition is satisfied, then the following equation that relates the ranges of input and output is satisfied:

$$\mathbf{Val}(v) = \sqrt{\mathbf{Val}(v_1)} - \mathbf{Err}_{\sqrt{}}. \tag{2.18}$$

In this equation, $\mathbf{Err}_{\sqrt{}}$ is the error induced by the computation of the square root itself and depends on the square root algorithm used. A description of the algorithms to implement fixed-point square root in terms of integer operations is given in Section 2.5.2.

As for the error bound, we have the following enclosure:

$$\begin{aligned}
\mathbf{Err}(v) &= \mathbf{Math}(v) - \mathbf{Val}(v) \\
&= \sqrt{\mathbf{Math}(v_1)} - \left(\sqrt{\mathbf{Val}(v_1)} - \mathbf{Err}_{\sqrt{}}\right) \\
&= \sqrt{\mathbf{Val}(v_1) + \mathbf{Err}(v_1)} - \sqrt{\mathbf{Val}(v_1)} + \mathbf{Err}_{\sqrt{}} && (2.19) \\
&= \sqrt{\mathbf{Val}(v_1)} \cdot \left(\sqrt{1 + \frac{\mathbf{Err}(v_1)}{\mathbf{Val}(v_1)}} - 1\right) + \mathbf{Err}_{\sqrt{}} && (2.20)
\end{aligned}$$

where the last factorization of Equation (2.20) is justified by the interval dependency problem mentioned in Section 1.3.4.

However, this formula does not yield tight error bounds as soon as $\mathbf{Val}(v_1)$ smallest elements are of the same order of magnitude than $\mathbf{Err}(v_1)$. To overcome this issue, one may use the sub-additivity property of the square root function which holds as long as x and $x+y$ are both positive:

$$\sqrt{x} - \sqrt{|y|} \leq \sqrt{x+y} \leq \sqrt{x} + \sqrt{|y|}. \tag{2.21}$$

Using this property in our case yields the following:

$$\sqrt{\mathbf{Val}(v_1)} - \sqrt{|\mathbf{Err}(v_1)|} \leq \sqrt{\mathbf{Val}(v_1) + \mathbf{Err}(v_1)} \leq \sqrt{\mathbf{Val}(v_1)} + \sqrt{|\mathbf{Err}(v_1)|}, \tag{2.22}$$

which induces the following bound on $\mathbf{Err}(v)$ when applied to (2.19):

$$\mathbf{Err}_{\sqrt{}} - \sqrt{|\mathbf{Err}(v_1)|} \leq \mathbf{Err}(v) \leq \mathbf{Err}_{\sqrt{}} + \sqrt{|\mathbf{Err}(v_1)|}. \tag{2.23}$$

In a practical implementation, since both (2.23) and (2.20) are strict bounds, one can compute their intersection. This is how the CGPE tool computes $\mathbf{Err}(v)$ for the square root operator.

2.5.2 Implementation of fixed-point square root

The algorithms we suggest in this section allow to compute the square root of fixed-point values and to deduce $\mathbf{Err}_{\sqrt{}}$.

Naive fixed-point square root

Given a fixed-point variable v_1 and according to the definition of fixed-point numbers given in Chapter 1, there exist two integers V_1 and f_1 such that $v_1 = V_1 \cdot 2^{-f_1}$. Therefore, a first approach is to consider the following rewriting:

$$\sqrt{v_1} = \begin{cases} \sqrt{V_1} \cdot 2^{-f_1/2}, & \text{if } f_1 \text{ is even,} \\ \sqrt{V_1/2} \cdot 2^{-(f_1-1)/2}, & \text{if } f_1 \text{ is odd.} \end{cases} \tag{2.24}$$

An implementation of this approach would compute $\sqrt{v_1}$ using one of the following depending on the parity of f_1:

$$\lfloor \sqrt{V_1} \rfloor \cdot 2^{-f_1/2} \quad \text{or} \quad \lfloor \sqrt{V_1/2} \rfloor \cdot 2^{-(f_1-1)/2} \tag{2.25}$$

where $\lfloor \sqrt{\cdot} \rfloor$ is the *integer square root* operation. This operation may be implemented in hardware or in software using multiple techniques such as digit-recurrence, and Newton-Raphson or Goldschmidt iteration [EIM+00],[EL04]. We deduce that the variable that holds $\sqrt{v_1}$ has $\approx f_1/2$ fraction bits, and that

$$\mathbf{Err}_{\sqrt{}} = \left[0, 2^{-\frac{f_1}{2}}\right] \quad \text{or} \quad \mathbf{Err}_{\sqrt{}} = \left[0, 2^{-\frac{f_1-1}{2}}\right], \tag{2.26}$$

depending on the parity of f_1. Compared to the error bounds of the previous operations, an error bound of $\approx 2^{-f_1/2}$ for the square root is not acceptable in practice.

Example 2.2. Consider the computation by the above algorithm of the square root of the 16-bit unsigned fixed-point value v_1 where $V_1 = 30270 = (01110110\,00111110)_2$ and $f_1 = 10$. The value of v_1 is given by $v_1 = V_1 \cdot 2^{-f_1} = \frac{30270}{1024} = (29.560546875)_{10}$. The input v_1 as well as its square root are illustrated by Figure 2.12.

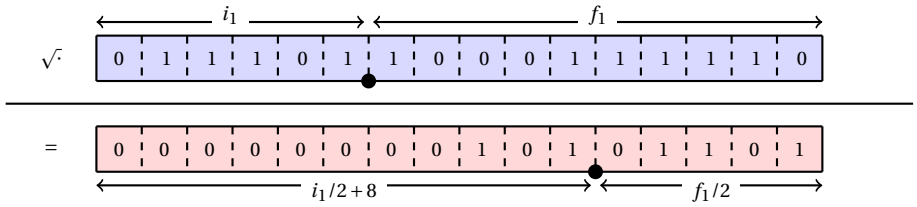


Figure 2.12: Square root of a 16-bit fixed-point value by the naive approach.

By following the approach detailed above, we have $\lfloor\sqrt{V_1}\rfloor = 173$ and

$$\sqrt{v_1} = 173 \cdot 2^{-5} = \frac{173}{32} = (5.40625)_{10} = (101.01101)_2.$$

However, the exact computation should yield $(5.4369611801998365\cdots)_{10}$. In this case $2^{-6} < \mathbf{Err}_{\sqrt{}} < 2^{-5}$ which indeed belongs to the enclosure (2.26).

Accurate fixed-point square root

To overcome the accuracy issue of naive fixed-point square root, we use the following rewriting of v_1:

$$v_1 = 2^{\eta} \cdot V_1 \cdot 2^{-(f_1+\eta)} \tag{2.27}$$

with the integer η being a parameter of the algorithm chosen at synthesis-time such as $f_1+\eta$ is even. Using this scaling factor, it follows that

$$\sqrt{v_1} = \sqrt{2^{\eta} \cdot V_1} \cdot 2^{\frac{-(f_1+\eta)}{2}}. \tag{2.28}$$

The cases $\eta = 0$ and $\eta = -1$ correspond to the even and odd cases of naive square root, respectively. An algorithm that exploits (2.28) shifts the integer representation V_1 of v_1 by η bits to the left and computes its integer square root. The result of this algorithm is a fixed-point variable with $(f_1+\eta)/2$ bits of fraction part. Hence using this approach, we conclude that

$$i_r = \lceil i_1/2 \rceil, \quad f_r = \frac{f_1+\eta}{2}, \text{ and } \sqrt{v_1} = \left\lfloor \sqrt{2^{\eta} \cdot V_1} \right\rfloor \cdot 2^{\frac{-(f_1+\eta)}{2}}, \tag{2.29}$$

where $\lfloor\sqrt{2^{\eta} \cdot V_1}\rfloor$ is computed using an integer square root operation. With such an algorithm, one controls the error $\mathbf{Err}_{\sqrt{}}$ induced by computing the square root and given by:

$$\mathbf{Err}_{\sqrt{}} = \left[0, 2^{-\frac{(f_1+\eta)}{2}}\right]. \tag{2.30}$$

Notice that it would not make sense to choose $\eta < 0$, since this would result in an increase of the error bound $\mathbf{Err}_{\sqrt{}}$.

Listing 2.6 gives a C implementation for a 32-bit square root that follows this approach. It is based on a digit-recurrence iteration and is suitable only when $0 \leq \eta < 32$. This is by far the most frequent case we treated in practice. This square root implementation is a slightly modified variant of a widely used integer square root algorithm [Jr.02]. The modifications consist in the shift of the input so as to multiply it by 2^{η}.

▷ **Listing 2.6:** C code for 32 → 32-bit fixed-point square root.

```
uint32_t isqrt32hu(uint32_t V1, uint16_t eta)
{
  uint64_t V1_extended = ((uint64_t)V1) << eta;
  uint64_t V = 0;
  int64_t one = 0x4000000000000000; // 2^(62)

  while (one != 0){
    if (V1_extended >= V + one){
       V1_extended = V1_extended - (V + one);
       V = V + (one << 1);
    }
    V >>= 1;
    one >>= 2;
  }
  return (uint32_t)V;
}
```

Example 2.3. Consider now the computation by this new algorithm of the square root of the 16-bit unsigned fixed-point value v_1 from Example 2.2 where $V_1 = 30270 = (01110110\,00111110)_2$ and $f_1 = 10$. And let us consider $\eta = 16$.

This second algorithm computes the integer square root of $V_1 \cdot 2^{16}$ which yields $44539 = (10101101\,11111011)_2$ and then sets the new scaling factor to $f_r = \dfrac{f_1+16}{2} = 13$.

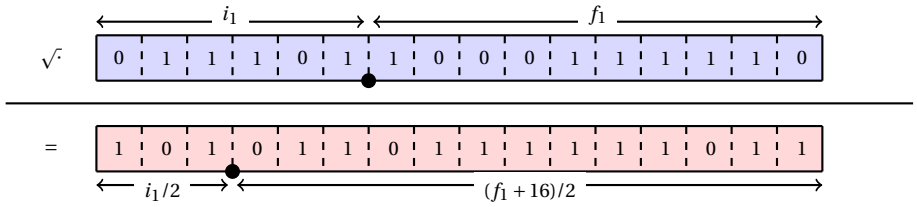


Figure 2.13: Square root of a 16-bit fixed-point value by the accurate approach.

We therefore have

$$\sqrt{v_1} = 44539 \cdot 2^{-13} = (5.4368896484375)_{10} = (101.0110111111011)_2.$$

In this case $2^{-14} < \mathbf{Err}_{\sqrt{}} < 2^{-13}$ which indeed belongs to the enclosure (2.30). This tight enclosure demonstrates that this algorithm is more accurate than naive fixed-point square root.

A slightly more accurate square root

When no adequate instruction is available on the target machine, integer square root must be implemented as a software routine such as in Listing 2.6. If one is to afford such a costly implementation, it may be interesting to rather compute the *nearest integer square root* function $\lfloor\sqrt{\cdot}\rceil$ which has almost the same efficiency penalty. This function returns the nearest integer to the square root of its argument.

Now, by computing $\lfloor\sqrt{2^{\eta}\cdot V_1}\rceil$, it is possible to gain 1 bit of precision on $\mathbf{Err}_{\sqrt{}}$ which would be given by the formula:

$$\mathbf{Err}_{\sqrt{}} = \left[-2^{-\frac{(f_1+\eta)}{2}-1}, 2^{-\frac{(f_1+\eta)}{2}-1}\right]. \tag{2.31}$$

Modifying Listing 2.6 to compute $\lfloor\sqrt{\cdot}\rceil$ instead of $\lfloor\sqrt{\cdot}\rfloor$ is just a matter of adding the snippet of Listing 2.7 between Lines 14 and 15.

▷ **Listing 2.7:** Snippet to add between Lines 14 and 15 of Listing 2.6 to compute $\lfloor\sqrt{\cdot}\rceil$ instead of $\lfloor\sqrt{\cdot}\rfloor$.

```
if (V1_extended > V)
  V++;
```

Indeed, the shift-and-subtract algorithm of Listing 2.6 runs in 32 steps and yields `V` such that `V1`$-$`V`^{2} $=$`V1_extended`. The snippet of Listing 2.7 takes advantage from the observation that if `(V1_extended > V)` then we have: which proves that `(V+1)` is the closest integer to the square root of `V1`.

2.6 Division

The division operator is even less studied in fixed-point arithmetic than square root and many programmers are advised to avoid it since it is costly and inefficient. Yet, for certain algorithms, one cannot get around division. Examples include computing the inverse of a matrix or solving linear systems. Indeed, it is with the aim of generating fixed-point code for matrix inversion that we first formalized division in [MNR14b], gave its error bounds and a mean to implement it.

In this section, we expose two approaches to division: an approach that doubles the word-length and an approach with custom word-length output. We show that the latter involves finding a trade-off between sharp error bounds and the risk of run-time overflows.

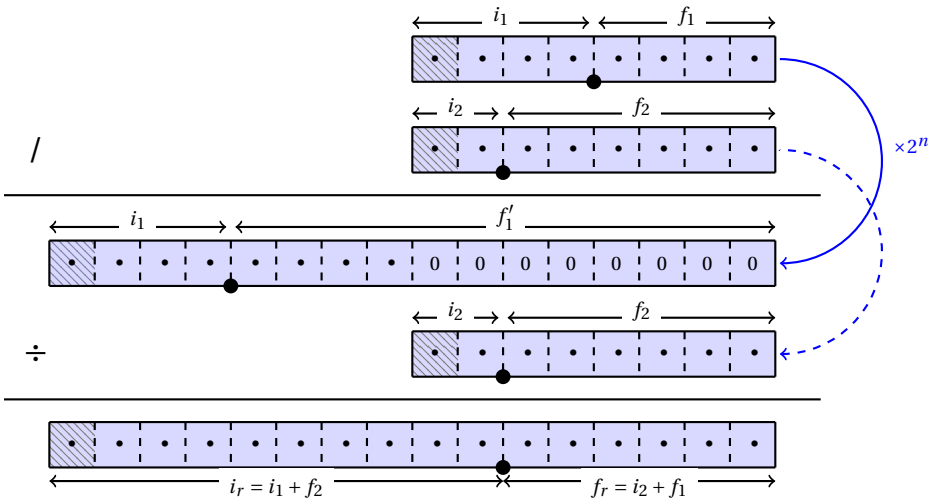


Figure 2.14: Output format of fixed-point division with word-length doubling.

2.6.1 Division with word-length doubling and its implementation

As pointed out by Yates [Yat13], the quotient v_1/v_2 when defined requires a fixed-point variable in the format $\mathbf{Q}_{i_r,f_r}$ with

$$i_r = i_1 + f_2 \quad \text{and} \quad f_r = f_1 + i_2 \tag{2.32}$$

since:

1. The largest possible dividend is -2^{i_1-1} while the smallest divisor is 2^{-f_2}. Thus the quotient could be as large as $-2^{i_1+f_2-1}$.
2. The smallest dividend is 2^{-f_1} while the largest divisor is -2^{i_2-1}. To be accurate, the fractional part must be at least of size $f_1 + i_2$.

Implementation of word-length doubling division

If one considers that both operands of division have the same word-length k, that is

$$k = i_1 + f_1 = i_2 + f_2,$$

then, the theoretical division presented above doubles the word-length since:

$$i_r + f_r = (i_1 + f_2) + (f_1 + i_2) = (i_1 + f_1) + (i_2 + f_2) = 2 \cdot k.$$

Figure 2.14 shows how to implement such a division for $k = 8$. In this figure, the symbols / and ÷ stand for fixed-point and integer division, respectively.

Listing 2.8 suggests a $32 \times 32 \rightarrow 64$ implementation for this fixed-point division.

▷ **Listing 2.8:** A C implementation of $32 \times 32 \rightarrow 62$ division.

```
1 int64_t div32_32_64(int32_t V1, int32_t V2)
2 {
3   int64_t t1 = ((int64_t)V1) << 32;
4   int64_t V = (t1 / V2);
5   return V;
6 }
```

2.6.2 Division without word-length doubling

Following a reasoning analogous to that for multiplication and shifting, it is costly to double the word-length with each division. In this section, we investigate other approaches to division where the output has the same word-length as the operands. We present two different algorithms and analyze their differences by exhibiting their error bounds. To present the two methods, let us recall that v_1 and v_2, the two fixed-point operands of division, can be written as $v_1 = V_1 \cdot 2^{-f_1}$ and $v_2 = V_2 \cdot 2^{-f_2}$ where V_1, V_2, f_1 and f_2 are integers.

First approach to non word-length doubling fixed-point division

This first variant of fixed-point division exploits the following rewriting:

$$\frac{v_1}{v_2} = \frac{V_1 \cdot 2^{-f_1}}{V_2 \cdot 2^{-f_2}} = \frac{V_1}{V_2} \cdot 2^{-(f_1 - f_2)}. \tag{2.33}$$

An algorithm inspired by Equation (2.33) would compute the quotient of the two integers V_1 and V_2, and then considers $f_1 - f_2$ as the implicit scaling factor of the result. Furthermore, the C standard requires integer division to be computed by discarding the fractional part of the exact division result, even for non-positive results, that is, by rounding the exact result toward zero [Int10, § 6.5.5]:

> When integers are divided, the result of the / operator is the algebraic quotient with any fractional part discarded (This is often called "truncation toward zero").

Therefore, if one is to denote by `trunc`$(\cdot)$ this behavior, the naive C compliant implementation of the division operator computes:

$$\frac{v_1}{v_2} = \text{trunc}\left(\frac{V_1}{V_2}\right) \cdot 2^{-(f_1 - f_2)}. \tag{2.34}$$

and the following property holds:

$$-1 < \frac{V_1}{V_2} - \text{trunc}\left(\frac{V_1}{V_2}\right) < 1. \tag{2.35}$$

Then, it is straightforward to deduce $\mathbf{Err}_/$ from the property (2.35):

$$\mathbf{Err}_/ = \left[-2^{-(f_1-f_2)}, 2^{-(f_1-f_2)}\right]. \tag{2.36}$$

Example 2.4. Compared to the division with word-length doubling shown in Figure 2.14, this variant yields only the upper part of the full result as shown in Figure 2.15, on a 8-bit example.

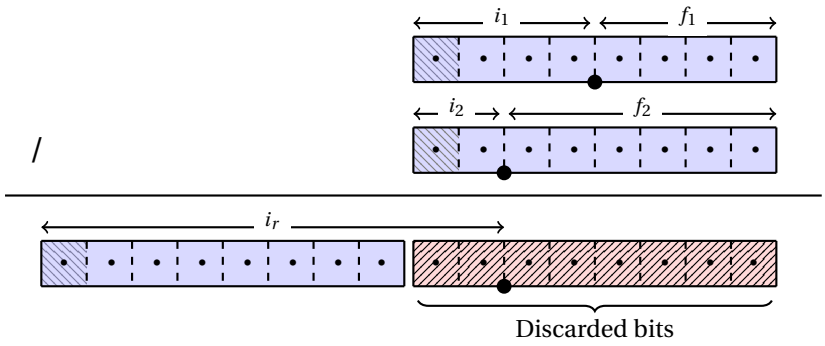


Figure 2.15: Example of one output format obtained using this first division approach.

In Figure 2.15, the dividend and the divisor are in the $\mathbf{Q}_{4,4}$ and $\mathbf{Q}_{2,6}$ formats, respectively. Computing a word-length doubling division leads to a quotient in the $\mathbf{Q}_{10,6}$ format. However, this first approach returns only the upper half of the result which has the format $\mathbf{Q}_{10,-2}$. For $\mathbf{Err}_/$, we have according to the formula of Equation (2.36):

$$\mathbf{Err}_/ = \left[-2^{-(f_1-f_2)}, 2^{-(f_1-f_2)}\right] = \left[-2^{-(4-6)}, 2^{-(4-6)}\right] = \left[-2^2, 2^2\right].$$

As expected, this error has the same weight as the LSB of the $\mathbf{Q}_{10,-2}$ result.

An advantage of this approach is that no shifting of the numerator is needed and fixed-point division corresponds to integer division.

▷ **Listing 2.9:** A C implementation for $32 \times 32 \rightarrow 32$ division according to the first approach.

```
int32_t div32_32_32h(int32_t V1, int32_t V2)
{
  return V1/V2;
}
```

A general approach to non word-length doubling fixed-point division

If the accuracy bound of the first approach is not satisfactory, one can achieve lower bounds but has to cope with the risk that overflow may occur at run-time. Example 2.5 illustrates the accuracy versus risk of overflow trade-off on the division of 8-bit variables.

Example 2.5. Suppose the end user to be dissatisfied with the error enclosure

$$\mathbf{Err}_{/} = \left[-2^{-(f_1-f_2)}, 2^{-(f_1-f_2)}\right] = \left[-2^2, 2^2\right],$$

obtained using the previous approach. He may conceive of a fixed-point division that returns 8 contiguous bits starting from the LSB or from any other position of the result. Such scenario is illustrated in Figure 2.16 where the 3 LSB and 5 MSB bits are discarded and the block of 8 bits in between is returned as the result of division.

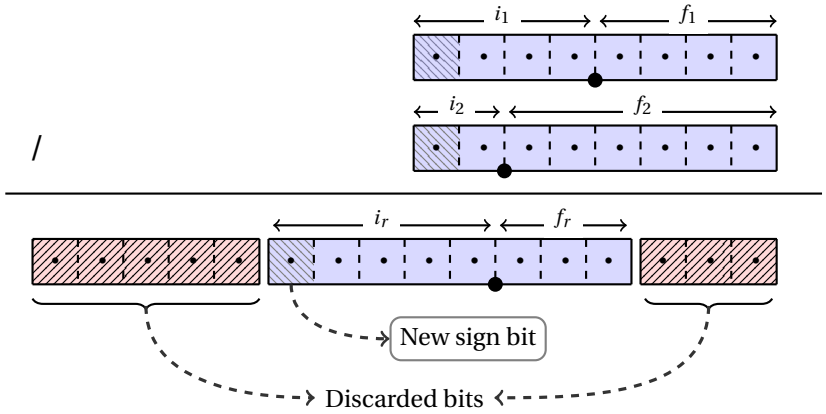


Figure 2.16: Example of the output format obtained using the general approach.

In this particular case, the user asked for a result in the $\mathbf{Q}_{5,3}$ format, and since the full result is a $\mathbf{Q}_{10,6}$ variable, 5 bits of the integer part and 3 bits of the fraction part were dropped.

Now, it is clear that choosing to drop the 5 MSB bits is a winning bet only if these bits are redundant sign bits and therefore, carry no information at run-time. Indeed, in Figure 2.16, if the value of any of the 5 discarded MSB bits happens to differ from that of the new sign bit, then overflow has occurred, and the returned result is wrong. To implement this general approach, suppose the user asks for the output format $\mathbf{Q}_{i_r,f_r}$, one can rewrite

the fixed-point division as follows:

$$\frac{v_1}{v_2} = \frac{V_1 \cdot 2^{-f_1}}{V_2 \cdot 2^{-f_2}} = \frac{V_1 \cdot 2^{f_r - f_1 + f_2}}{V_2} \cdot 2^{-f_r}. \tag{2.37}$$

And, an algorithm that exploits this rewriting computes

$$\frac{v_1}{v_2} = \text{trunc}\left(\frac{V_1 \cdot 2^{\eta}}{V_2}\right) \cdot 2^{-f_r}$$

where $\eta = f_r - f_1 + f_2$. Then we deduce that the error $\mathbf{Err}_/$ is as follows:

$$\mathbf{Err}_/ = \left[-2^{-f_r}, 2^{-f_r}\right].$$

Listing 2.10 gives an implementation of this $32 \times 32 \rightarrow 32$-bit division based on standard `C` integer division. If this option is not available or is too costly, this operation, just like square root, may be implemented in hardware or in software using digit-recurrence, and Newton-Raphson or Goldschmidt iteration.

▷ **Listing 2.10:** `C` code of $32 \times 32 \rightarrow 32$-bit fixed-point division operation with the general approach.

```
1 int32_t div32_32_32m(int32_t V1, int32_t V2, uint16_t eta)
2 {
3   int64_t t1 = ((int64_t)V1) << eta;
4   int64_t V = t1 / V2;
5   return (int32_t) V;
6 }
```

Figure 2.17 shows how this approach computes the result of the division of Example 2.5 where $n = 8$ and the output format asked for is $\mathbf{Q}_{5,3}$. In this case, the dividend must be left extended by $\eta = f_r - f_1 + f_2 = 3 - 4 + 6 = 5$ positions before the division is performed.

Final remarks on implementing the division operator

At first glance, it may seem as if division introduces a weakness in our methodology. Indeed, since the user must decide on the output fixed-point format of division, the synthesis flow is no longer fully automated. However, asking for an output format of division must rather be understood as a feature the synthesis tool provides to its user. Indeed, we could have fixed rigidly the output format of division, but this may reveal to be either too risky or too detrimental to accuracy. Allowing the user to set the level of accuracy versus risk of overflow trade-off is a more flexible approach.

Finally, notice that we could have gone through this discussion when we treated multiplication in Section 2.3. However, we chose to always keep the upper half of the full-length product. We did so based on the following reasoning:

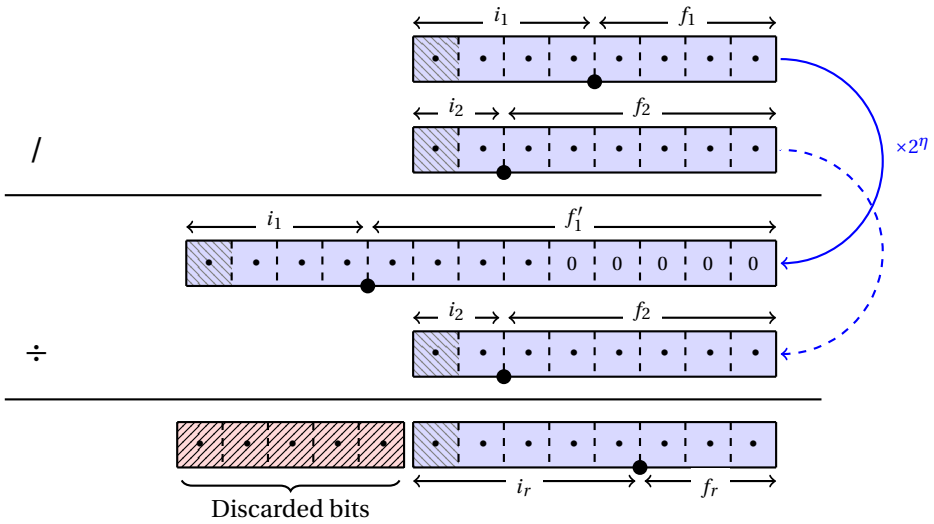


Figure 2.17: Division using a constant word-length and returning the $\mathbf{Q}_{5,3}$ result for the input of Example 2.5.

- ◇ Many processors such as the ST231 provide an instruction that returns the upper half of a multiplication,
- ◇ If one is to consider the two operands of multiplication as normally distributed n-bit random variables, then their product is normally distributed and is likely to require $2n-2$ bits.

This reasoning does not hold for division since the ratio of two normally distributed random variables is not normally distributed.

2.6.3 Output range and error bounds for division

The variants of division presented above fix the output format of division. In doing so, they make assumptions on the run-time values taken by the divisor v_2. Suppose for now that these assumptions are such that $v_2 \in \widehat{\mathbf{Val}(v_2)}$ where $\widehat{\mathbf{Val}(v_2)} \subset \mathbf{Val}(v_2)$ and $0 \notin \widehat{\mathbf{Val}(v_2)}$. For now, we will use $\widehat{\mathbf{Val}(v_2)}$ in our formulas to compute $\mathbf{Err}(v)$ and show later how to compute its value.

Since, $0 \notin \widehat{\mathbf{Val}(v_2)}$, the following interval divisions are not degenerate and we have the following for $\mathbf{Err}(v)$:

$$\mathbf{Err}(v) = \frac{\mathbf{Math}(v_1)}{\mathbf{Math}(v_2)} - \frac{\mathbf{Val}(v_1)}{\widehat{\mathbf{Val}(v_2)}} + \mathbf{Err}_/, \tag{2.38}$$

where $\mathbf{Err}_/$ is the error entailed by the division itself and which depends on the approach

used for implementing it. By further developing Equation (2.38), we have:

$$\begin{aligned}
\mathbf{Err}(v) &= \frac{\mathbf{Val}(v_1)+\mathbf{Err}(v_1)}{\widehat{\mathbf{Val}(v_2)}+\mathbf{Err}(v_2)} - \frac{\mathbf{Val}(v_1)}{\widehat{\mathbf{Val}(v_2)}} + \mathbf{Err}_{/} \\
&= \frac{\widehat{\mathbf{Val}(v_2)}\cdot\mathbf{Val}(v_1)+\widehat{\mathbf{Val}(v_2)}\cdot\mathbf{Err}(v_1)-\mathbf{Val}(v_1)\cdot\widehat{\mathbf{Val}(v_2)}-\mathbf{Val}(v_1)\cdot\mathbf{Err}(v_2)}{\widehat{\mathbf{Val}(v_2)}\cdot\left(\widehat{\mathbf{Val}(v_2)}+\mathbf{Err}(v_2)\right)} + \mathbf{Err}_{/} \\
&= \frac{\widehat{\mathbf{Val}(v_2)}\cdot\mathbf{Err}(v_1)-\mathbf{Val}(v_1)\cdot\mathbf{Err}(v_2)}{\widehat{\mathbf{Val}(v_2)}\cdot\left(\widehat{\mathbf{Val}(v_2)}+\mathbf{Err}(v_2)\right)} + \mathbf{Err}_{/}. \qquad (2.39)
\end{aligned}$$

2.6.4 How to compute the value of $\widehat{\mathbf{Val}(v_2)}$?

Since the output format of division is fixed to the format $\mathbf{Q}_{i_r,f_r}$, its value is supposed to be in the range $[-2^{i_r-1}, 2^{i_r-1}-2^{f_r}]$. Therefore, we have that

$$\mathbf{Val}(v) = \mathscr{R}\text{ange}(\mathbf{Q}_{i_r,f_r}) = [-2^{i_r-1}, 2^{i_r-1}-2^{f_r}]. \qquad (2.40)$$

Since $\mathbf{Val}(v)$, $\mathbf{Val}(v_1)$, and $\widehat{\mathbf{Val}(v_2)}$ are related by the formula:

$$\mathbf{Val}(v) = \frac{\mathbf{Val}(v_1)}{\widehat{\mathbf{Val}(v_2)}} - \mathbf{Err}_{/}, \qquad (2.41)$$

it is possible to deduce $\widehat{\mathbf{Val}(v_2)}$ by computing:

$$\widehat{\mathbf{Val}(v_2)} = \frac{\mathbf{Val}(v_1)}{\widetilde{\mathbf{Val}(v)}+\mathbf{Err}_{/}}, \qquad (2.42)$$

where $\widetilde{\mathbf{Val}(v)}$ corresponds to $\mathbf{Val}(v)\setminus\{0\}$, that is

$$\widetilde{\mathbf{Val}(v)} = [-2^{i_r-1}, -2^{-f_r}] \cup [2^{-f_r}, 2^{i_r-1}-2^{f_r}]. \qquad (2.43)$$

Furthermore, to make sure that $\widehat{\mathbf{Val}(v_2)} \subseteq \mathbf{Val}(v_2)$, the CGPE tool computes $\widehat{\mathbf{Val}(v_2)}$ as follows:

$$\widehat{\mathbf{Val}(v_2)} = \frac{\mathbf{Val}(v_1)}{\widetilde{\mathbf{Val}(v)}+\mathbf{Err}_{/}} \cap \mathbf{Val}(v_2).$$

2.7 Conclusion

In this chapter, we introduced a technique based on interval arithmetic to compute bounds on the range and rounding errors of every variable in a fixed-point design. This technique propagates the intervals $\mathbf{Val}(v)$ and $\mathbf{Err}(v)$ through the tree-like representation of an arithmetic expression. For this framework to be useful, propagation rules must be

derived for each arithmetic operator used in the design. Therefore, we made explicit these rules for each operator $\diamond \in \{+, -, \times, \ll, \gg, \sqrt{}, /\}$ and suggested a standard `C` implementation for each. Once implemented in a fixed-point synthesis tool, this framework provides range analysis and allows to generate certified code, i.e., code whose rounding errors are guaranteed to belong to a predetermined interval. Indeed, the next chapter discusses the CGPE tool, which is a particular implementation of this framework.

CHAPTER 3

THE CGPE SOFTWARE TOOL

The previous chapter introduced the arithmetic model we shall rely on for the rest of this work. In the first part of this chapter, its implementation in the CGPE software tool is described. This tool is dedicated to fixed-point synthesis for basic arithmetic expressions such as sums, dot-products, and polynomial evaluations. The second part of the chapter describes how the tool was enhanced by adding an instruction selection step. Finally, the last section walks through the steps involved in generating certified code for an infinite impulse response filter.

3.1 Introduction

As stated in the introduction to this work, the main obstacle towards the widespread use of fixed-point arithmetic is the absence of code generation tools. Indeed, many researchers such as Irturk [IMK09] and Fang [FRC03] provide feedback and results obtained using their own tool-chains, but the source code is seldom available. To remedy to this situation, we committed ourselves to enhancing the CGPE software tool, which generates fixed-point code for a variety of arithmetic expressions. Before our enhancements, CGPE handled only unsigned code generation for polynomial evaluation, and counted on the user to provide correctly aligned coefficients. It was since extended to support more expressions, to handle signed arithmetic, and to use alignment shifts when necessary.

Besides being an open source project, the originality of the current CGPE tool is that it is built to strictly enforce an arithmetic model. Indeed, it keeps track of expressions using a tree-like intermediate representation and uses the rules of Chapter 2 to propagate the enclosures $\mathbf{Val}(v)$ and $\mathbf{Err}(v)$. The interval $\mathbf{Val}(v)$ allows to deduce the fixed-point format of each intermediate node while $\mathbf{Err}(v)$ gives a bound on the rounding errors of the fixed-point implementation.

This chapter presents this tool and is organized into three sections as follows: Section 3.2 describes the tool, its input and output, and its architecture. Section 3.3 presents a recent enhancement to the tool: a module based on instruction selection to optimize the synthesized code. Finally, Section 3.4 describes the process of automated code synthesis for the Infinite Impulse Response (IIR) filter introduced in Example 1.9 of Chapter 1.

3.2 The CGPE software tool

3.2.1 Background and early objectives of CGPE

CGPE[1] is an acronym for *Code Generation for Polynomial Evaluation*. It is a `C++` tool that was developed in 2009 by Guillaume REVY who described its internals in his thesis [Rev09]. Revy's goal was to enhance the FLIP (Floating-point Library for Integer Processors) library [BBDdD+04] which provided IEEE-754 floating-point support to integer processors such as the ST00 family of VLIW processors. To adhere to the IEEE standard, FLIP had to provide correct rounding for the floating-point operators including addition, subtraction, multiplication, division, and square root as well as support for elementary functions. In the `0.3` version of FLIP (described in [Rai06]), support for division and square root relied on a unique iteration of Goldschmidt's algorithm to refine an approximation obtained by evaluating a small degree[2] polynomial.

REVY noticed that the ST231 was well suited for polynomial evaluation and that one can reduce the latency of division and square root by exploiting the instruction-level parallelism offered by custom evaluation schemes instead of Horner's rule. For instance, consider the following example which arises when implementing floating-point square root.

Example 3.1. To emulate floating-point square root, one must evaluate the following bivariate polynomial where the coefficients a_i, for $0 \le i \le 9$, as well as the variables s and t are represented in fixed-point:

$$P(s,t) = 2^{-25} + s \cdot \left(a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5 + a_6 t^6 + a_7 t^7 + a_8 t^8\right). \tag{3.1}$$

The Horner form of this polynomial is given by Equation (3.2), and its implementation requires 36 clock cycles on the ST231 where addition and multiplication cost 1 and 3

[1] See `http://cgpe.gforge.inria.fr` and [MR11], [Rev09].

[2] The degree is 5 for division and 3 for square root. (See [Rai06] for a description of the algorithms)

cycles, respectively.

$$P(s,t) = 2^{-25} + s\cdot\Big(a_0 + t\Big(a_1 + t\Big(a_2 + t\Big(a_3 + t\big(a_4 + t\big(a_5 + t\,(a_6 + t\,(a_7 + t a_8))\big)\big)\Big)\Big)\Big)\Big). \quad (3.2)$$

However, Revy was able to improve this latency by evaluating the same polynomial using the scheme of Equation (3.3).

$$\begin{aligned} P(s,t) \;=\; & \Big[\big(2^{-25} + (s\cdot(a_0 + t\cdot a_1))\big) + \big((s\cdot t^2)\cdot\big((a_2 + t\cdot a_3) + t^2\cdot a_4\big)\big)\Big] + \\ & \Big[\big((t\cdot t^2)\cdot(s\cdot t^2)\big)\cdot\big((a_6 + t\cdot a_7) + \big(t^2\cdot(a_8 + t\cdot a_9)\big)\big)\Big]. \end{aligned} \quad (3.3)$$

This scheme reduced the evaluation latency to only 13 cycles.

This example stresses the importance of a tool that finds evaluation schemes which reduce the latency of evaluation while still satisfying some accuracy constraints. Indeed, the evaluation scheme (3.3) was found automatically by CGPE and was later proved to be optimal in terms of latency on the ST231 [Rev09].

3.2.2 The architecture of CGPE

The input of CGPE

In its early versions, CGPE's input consisted in a description of the input polynomial that included its degree, its coefficients, the range of the variables, and whether its a univariate or a bivariate polynomial. In its recent versions, support was added for sums, dot-products, and fully parenthesized expressions. Regardless of the problem to solve, the details are described in an external `XML` file, that contains an interval of values and a fixed-point format for each coefficient or variable, as well as a maximum error bound allowed for the generated code. Each input may also be accompanied by an associated error bound.

Listing 3.1 shows the input `XML` file of a degree-5 polynomial approximant of the function $1/(1+x)$ over $[0,1]$. Since the polynomial is univariate,[3] the listing contains the hexadecimal value and the format of each of the 6 coefficients, as well as the range and format of the variable. Line 9 of the listing contains the maximum magnitude tolerated for rounding errors. Its value is

$$3213b - 26 = 3213\cdot 2^{-26} < 2^{-14}.$$

Besides the description of the problem to synthesize code for, CGPE takes as input a set of criteria and architectural constraints. These include the latency of each basic operator

[3] The `XML` attributes x and y uniquely identify each coefficient. For instance, x = "4" and y = "1" correspond to the coefficient of the monomial $x^4 y$. The polynomial is univariate if all the coefficients and variables have the attribute y = "0".

▷ **Listing 3.1:** A CGPE input XML file for a degree-5 polynomial evaluation.

```
<polynomial>
 <coefficient x="0" y="0" inf="0x7ffec8d0" sup="0x7ffec8d0" sign="0"
    integer_part="2" fraction_part="30"/>
 <coefficient x="1" y="0" inf="0x7f9bef55" sup="0x7f9bef55" sign="1"
    integer_part="2" fraction_part="30"/>
 <coefficient x="2" y="0" inf="0x7ab5c54b" sup="0x7ab5c54b" sign="0"
    integer_part="2" fraction_part="30"/>
 <coefficient x="3" y="0" inf="0x647d671d" sup="0x647d671d" sign="1"
    integer_part="2" fraction_part="30"/>
 <coefficient x="4" y="0" inf="0x379913e9" sup="0x379913e9" sign="0"
    integer_part="2" fraction_part="30"/>
 <coefficient x="5" y="0" inf="0x0e358cb5" sup="0x0e358cb5" sign="1"
    integer_part="2" fraction_part="30"/>
 <variable x="1" y="0" inf="0x00000000" sup="0xffe00000" sign="0"
    integer_part="0" fraction_part="32"/>
 <error value="3213b-26" strict="true" type="absolute"/>
</polynomial>
```

and a bound on the overall latency. And, since it was initially intended for use with VLIW processors, additional parameters are provided to set the available level of parallelism, that is, the number of issues on the target and the number of multipliers.

The output of CGPE

At the end of the synthesis process, CGPE produces a set of C codes evaluating the input problem on the given target. For each synthesized code, the tool ensures that the generated code satisfies the latency criterion and that the magnitude of the rounding error is inferior to the maximum error bound provided. To make the latter condition checkable, a GAPPA[4] certificate file is generated for each code. The GAPPA tool uses a combination of interval arithmetic and rewriting rules to prove that the evaluation error entailed in the C code is indeed below the given threshold. Listing 3.2 shows one of the automatically produced C codes for the evaluation of the degree-5 polynomial approximant of the function $1/(1+x)$ over $[0,1]$ whose XML input file was given by Listing 3.1. The rounding errors of the code of Listing 3.2 belong to the following interval:

$$\mathbf{Err}(\mathtt{r11}) = \left[-2^{-28.3536}, 2^{-28.4164}\right],$$

[4] See http://gappa.gforge.inria.fr and [Mel06].

▷ **Listing 3.2:** Example of code automatically generated using CGPE.

```
1  //   a0 = +0x7ffec8d0p-30      a1 = -0x7f9bef55p-30
2  //   a2 = +0x7ab5c54bp-30      a3 = -0x647d671dp-30
3  //   a4 = +0x379913e9p-30      a5 = -0x0e358cb5p-30
4  uint32_t func_d5( uint32_t x /* Q0.32 */) { // Formats
5    uint32_t r0  = mul(x, 0x7f9bef55);        // Q2.30
6    uint32_t r1  = 0x7ffec8d0 - r0;           // Q2.30
7    uint32_t r2  = mul(x, x);                 // Q0.32
8    uint32_t r3  = mul(x, 0x647d671d);        // Q2.30
9    uint32_t r4  = 0x7ab5c54b - r3;           // Q2.30
10   uint32_t r5  = mul(r2, r4);               // Q2.30
11   uint32_t r6  = r1 + r5;                   // Q2.30
12   uint32_t r7  = mul(r2, r2);               // Q0.32
13   uint32_t r8  = mul(x, 0x0e358cb5);        // Q2.30
14   uint32_t r9  = 0x379913e9 - r8;           // Q2.30
15   uint32_t r10 = mul(r7, r9);               // Q2.30
16   uint32_t r11 = r6 + r10;                  // Q2.30
17   return r11;
18 }
```

which is inferior in magnitude to the error bound requested by the user. This property can indeed be checked by running GAPPA on the certificate generated by CGPE and shown in Listing 3.3. Lines 12 to 23 of this certificate are just a transposition of the C code of Listing 3.2 to the syntax of GAPPA. In this syntax, the `r0` to `r11` variables are the results of finite word-length computations, and their error free counterparts are the `Mr0` to `Mr11` variables. The clauses between lines 32 and 46 are to be checked by the GAPPA tool. Among these, the clause of Line 45 is the most crucial since it checks that

$$|\texttt{r11} - \texttt{Mr11}| - \texttt{CertifiedBound} <= 0,$$

that is, that the magnitude of the rounding errors on the final result is less than the required error bound.

▷ **Listing 3.3:** GAPPA certificate produced by CGPE for the code of Listing 3.2.

```
1  a0 = fixed<-26,dn>(0x07ffec8dp-26);
2  a1 = fixed<-30,dn>(0x7f9bef55p-30);
3  a2 = fixed<-30,dn>(0x7ab5c54bp-30);
4  a3 = fixed<-30,dn>(0x647d671dp-30);
5  a4 = fixed<-30,dn>(0x379913e9p-30);
6  a5 = fixed<-30,dn>(0x0e358cb5p-30);
```

```
T = fixed<-32,dn>(var0);

CertifiedBound = 3213b-26 - 1b-300;

## Evaluation scheme
r0  fixed<-30,dn>= T * a1;   Mr0  = MT * a1;      ##Q2.30
r1  fixed<-30,dn>= a0 - r0;  Mr1  = a0 - Mr0;     ##Q2.30
r2  fixed<-32,dn>= T * T;    Mr2  = MT * MT;      ##Q0.32
r3  fixed<-30,dn>= T * a3;   Mr3  = MT * a3;      ##Q2.30
r4  fixed<-30,dn>= a2 - r3;  Mr4  = a2 - Mr3;     ##Q2.30
r5  fixed<-30,dn>= r2 * r4;  Mr5  = Mr2 * Mr4;    ##Q2.30
r6  fixed<-30,dn>= r1 + r5;  Mr6  = Mr1 + Mr5;    ##Q2.30
r7  fixed<-32,dn>= r2 * r2;  Mr7  = Mr2 * Mr2;    ##Q0.32
r8  fixed<-30,dn>= T * a5;   Mr8  = MT * a5;      ##Q2.30
r9  fixed<-30,dn>= a4 - r8;  Mr9  = a4 - Mr8;     ##Q2.30
r10 fixed<-30,dn>= r7 * r9;  Mr10 = Mr7 * Mr9;    ##Q2.30
r11 fixed<-30,dn>= r6 + r10; Mr11 = Mr6 + Mr10;   ##Q2.30

## Results
{
  (
       var0 in [0x00000000p-32,0xffe00000p-32]
    /\ T - MT in [0,0]
    ## Constraints
    ->
       r0 in [0,267485051b-27]
    /\ r0 - Mr0 in [-4294967295b-62,0]
    /\ r1 in [940447b-27,134212749b-26]
    /\ r1 - Mr1 in [0,4294967295b-62]
    /\ r2 in [0,4190209b-22]
    /\ r2 - Mr2 in [-4294967295b-64,0]
    /\ r3 in [0,1685116785b-30]
    /\ r3 - Mr3 in [-4294967295b-62,0]
    /\ r4 in [186808045b-29,2058732875b-30]
    /\ r4 - Mr4 in [0,4294967295b-62]
    /\ r5 in [0,2056722885b-30]
    ...
    /\ r11 in [940447b-27,5135085355b-30]
    /\ |r11 - Mr11| - CertifiedBound <= 0
    /\ CertifiedBound in ?
  )
}
```

The work-flow of CGPE

CGPE has a compiler-like architecture with three stages as illustrated by Figure 3.1.

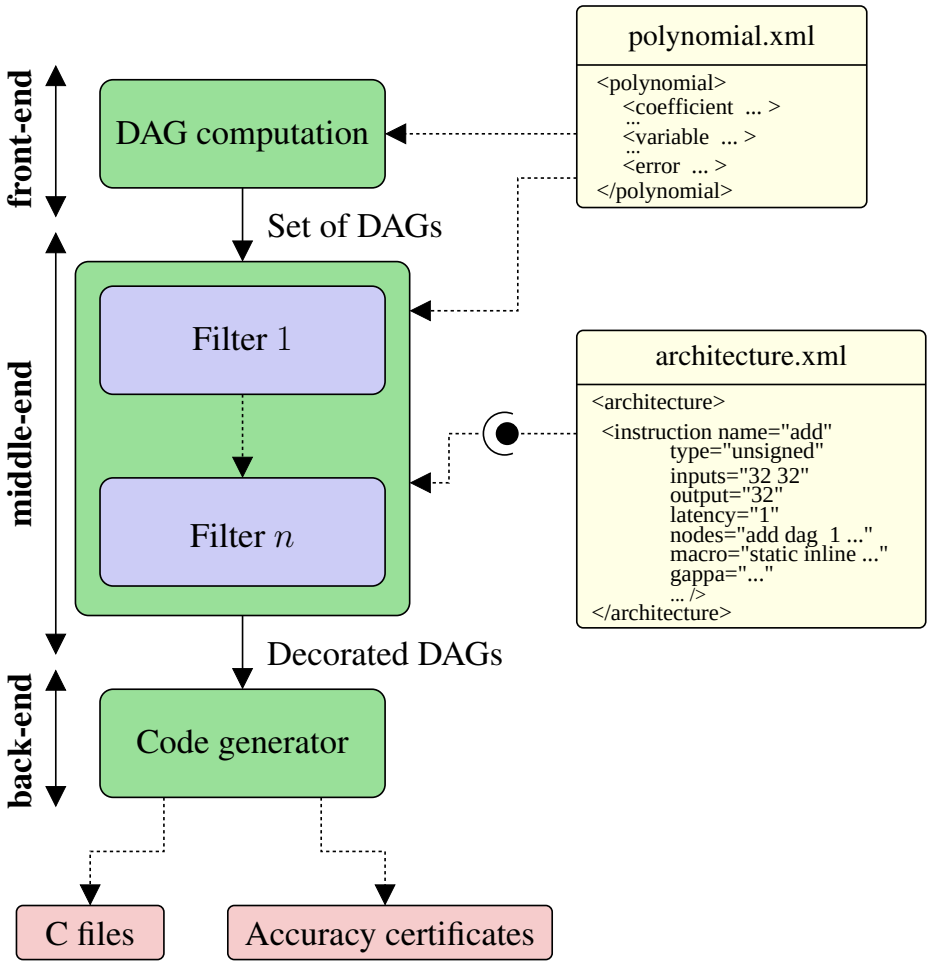
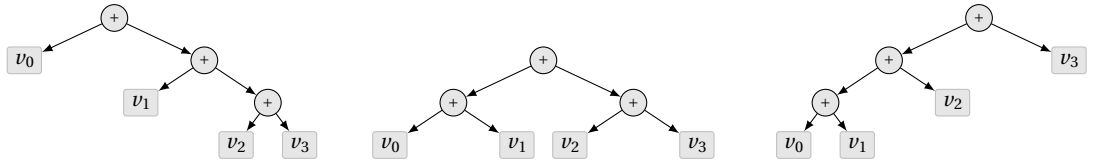


Figure 3.1: Data-flow path and the three stages of CGPE.

First, CGPE starts by a computation step that plays the role of a compiler's front-end. This step computes a set of DAGs (Directed Acyclic Graphs), each DAG being the intermediate representation of a given scheme.

Example 3.2. Given the problem of summing 4 variables v_0, v_1, v_2 and v_3, a possible output of CGPE's front-end is the collection of 3 DAGs shown in Figure 3.2. These DAGs represent 3 different evaluation schemes that correspond to the following fully parenthesized expressions:

1. $\Big(v_0 + \big(v_1 + (v_2 + v_3)\big)\Big)$,
2. $\big((v_0 + v_1) + (v_2 + v_3)\big)$,
3. $\Big(\big((v_0 + v_1) + v_2\big) + v_3\Big)$.



(a) DAG of $\Big(v_0 + \big(v_1 + (v_2 + v_3)\big)\Big)$. **(b)** DAG of $\big((v_0 + v_1) + (v_2 + v_3)\big)$. **(c)** DAG of $\Big(\big((v_0 + v_1) + v_2\big) + v_3\Big)$.

Figure 3.2: A sample of CGPE's front-end output for the sum of 4 variables.

Indeed, "scheme" is the terminology used to denote the fully parenthesized expressions for which a corresponding DAG is produced by the front-end.[5]

The time and memory critical task in this case is the generation of fast schemes in the front-end. For instance, $\prod_{i=1}^{n-2}(2i+1)$ different ways can be used to carry out the sum of n variables. This number sequence[6] grows exponentially: while there exists 945 different schemes to implement a size-6 sum or dot-product, this number exceeds $34 \cdot 10^6$ for size-10 sums.

Example 3.3. Figure 3.3 shows the DAG representing the evaluation code of Listing 3.2. It corresponds to the fully parenthesized expression:

$$P(x) = (((a_0 - (x \cdot a_1)) + ((x \cdot x) \cdot (a_2 - (x \cdot a_3)))) + (((x \cdot x) \cdot (x \cdot x)) \cdot (a_4 - (x \cdot a_5)))).$$

The figure explains the choice of using Directed Acyclic Graphs instead of trees. Indeed, in polynomial expressions, expressions may be factored and therefore be common to two sub-expressions. Using DAGs ensures that a unique code will be generated for these sub-expressions. For instance, in Figure 3.3, the intermediate variable $r_2 = x^2$ is used by both r_5 and r_7.

[5] Revy [Rev09] and Mouilleron [Mou11] use the word "parenthesization" instead of "scheme".
[6] See http://oeis.org/A001147.

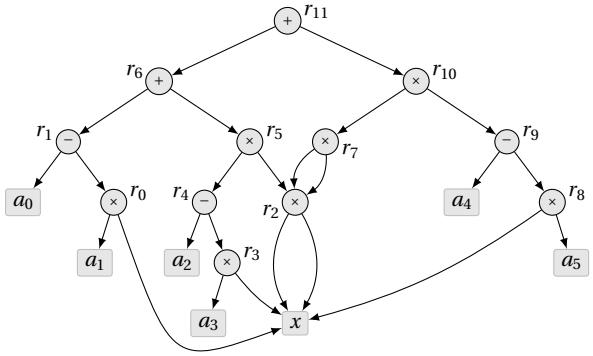


Figure 3.3: DAG representing the evaluation code in Listing 3.2.

In CGPE, the combinatorial explosion of evaluation schemes is tackled using different strategies and heuristics described in depth by Revy [Rev09, §6] and Mouilleron [Mou11, §8.1.2]. One such heuristic is to assume unbounded parallelism, to consider only the latency of each basic operator (adder, multiplier, ...), and to keep only the schemes that reduce the evaluation latency.

Second, CGPE goes to a filtering step that can be seen as a compiler's middle-end. At this step, each DAG undergoes a series of filters, each filter being dedicated to a criterion and deciding whether to keep the scheme or to prune it. DAGs that pass all the filters reach CGPE's back-end, which takes care of producing the code and the associated accuracy certificate.

3.3 Target-dependent synthesis with CGPE

Enhancing CGPE with instruction selection was motivated by the large number of patterns $(a \times b) + c$ appearing in polynomial evaluation as well as shift operations induced by the use of fixed-point arithmetic. Indeed many modern architectures are shipped with advanced instructions, allowing to fuse at least two classical operations in a single one. The `fused multiply-add` (FMA), computing $(a \times b) + c$ in one instruction and with only one final rounding is an example of such advanced instructions. Moreover, its support by floating-point processors is now required by the IEEE 754-2008 standard [75408]. As for integer arithmetic, an example of advanced instruction is the `mulacc`, available on certain architectures including the ARM processors [ARM09], and that computes $(a \times b) + c$ in a single instruction. Also, CGPE's initial target, the ST231 [ST208], is shipped with a `shift-and-add` instruction that performs a left shift of 1 up to 4 positions followed by an addition, that is, the pattern $(a \ll b) + c$ with $b \in \{1, \cdots, 4\}$.

To add support for advanced instructions in CGPE's code synthesis process, two strategies are conceivable:

1. adding support for these operations in the DAG computation algorithms in the front-end, or
2. adding an instruction selection module to the middle-end, that takes into account these instructions.

The current computation algorithms of the front-end are designed to build DAGs using only addition/subtraction, multiplication, and more recently square root and division. These algorithms are already complex, problem dependant, and lack scalability.

Indeed, handling new binary operations or operations with higher arity is tedious and imposes changes to all the DAG computation algorithms. For this reason, and since CGPE's middle-end had already a modular design, we preferred to implement target dependant code optimization in a middle-end instruction selection filter.

Instruction selection is a well-known process in compilation theory [ASU86, §8.9]. Given a set of instructions and an intermediate representation of an expression, possibly a DAG, an *instruction selection algorithm*, also called a *tiling algorithm*, produces a collection of instructions necessary to evaluate this expression. To optimize the generated code with respect to some criterion, a cost may be associated to each considered instruction, allowing thus to define a cost function used to evaluate the cost of a DAG. And a good selection is one that minimizes this cost function. In the case of DAGs, this problem was proven to be NP-complete [KG08], even for simple machines, but algorithms that perform well in practice have been designed to tackle this problem. Indeed, the technique presented in this section is inspired by the NOLTIS algorithm [KG08] and more generally by bottom-up rewrite systems that deal with instruction selection on DAGs.

3.3.1 Architecture description

For the sake of modularity, we chose a structure where the core of CGPE is not aware of the instructions available for use. Indeed, these instructions are described in an external `XML` architecture file. An entry in this file corresponds to a hardware instruction or to a basic block of instructions if one is to target FPGAs or to test the utility of exotic instructions. Listing 3.4 gives an example of the entry for $32\times32\rightarrow32$-bit unsigned addition.

Precisely, for each instruction, this file contains the following information: its *name*, its *type*, that is, signed or unsigned, the size of its *inputs* and *output*, and its *latency* in cycles. If the entry matches an instruction available in hardware, this is its latency on the target. Otherwise, if the entry represents a basic block, its latency is usually determined using unbounded parallelism, that is, without any resource constraint.

In addition to these parameters, the description contains a `C` macro to emulate the instruction if it is not available on the target, and a piece of GAPPA script.

The latter is useful for the generation of accuracy certificates and uses the semantics of GAPPA.

▷ **Listing 3.4:** The entry of 32-bit unsigned addition in the architecture file.

```
1 <instruction
2   name="add" type="unsigned" inputs="32 32" output="32"
3   nodes="add dag 1 dag 2" latency="1"
4   macro="static uint32_t __name__(uint32_t a, uint32_t b){
5            return (a + b);
6          }"
7   gappa="_r_ fixed<_Fr_,dn>= _1_ + _2_;
8          _Mr_ = _M1_ + _M2_;"
9 />
```

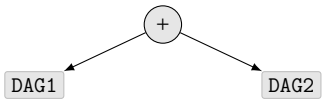
In Listing 3.4, Line 7 returns the value `_r_` computed in fixed-point arithmetic in the program. In this example, it corresponds to the computed value `_1_ + _2_` (addition between the first and second operands of the instruction) rounded downward (`dn`) with `_Fr_` fractional bits. Line 8 returns the mathematical value `_Mr_ = _M1_ + _M2_`, computed as if all the previous computations had been done exactly. Once these two values are computed, GAPPA deduces an enclosure of the rounding errors by simply subtracting `_Mr_` from `_r_`. Finally the attribute *nodes* gives the description of the pattern matched by the instruction in terms of atomic operations. The entry of Listing 3.4 is solely composed of an addition, whose children are both any DAGs, and corresponding to the first and second parameters of the instruction, respectively. The node description can be used to match any DAG of height at most 4 (a self imposed limit to ease instruction selection). It is determined by traversing this DAG in left-to-right breadth first order. For instance, the nodes attribute of the ST231 `shift-and-add` instruction is shown in Listing 3.5.

▷ **Listing 3.5:** Nodes description of ST231's `shift-and-add` instruction.

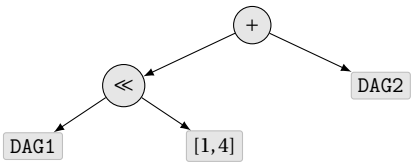
```
nodes="add shift dag 2 dag 1 value [-4,-1]"
```

Here "`value [-4, -1]`" corresponds to the right child of the shift operator, and indicates that it is a numerical value in $[-4,-1]$. Since we follow the convention that a negative value implies a left shift, this nodes attribute describes a left shift of 1 to 4 positions followed by an addition.

The node description of Listing 3.4's addition and Listing 3.5's shift-and-add are both illustrated graphically in Figure 3.4.



(a) The nodes description of Line 3 of Listing 3.4.



(b) The nodes description of Listing 3.5.

Figure 3.4: Node representation of the addition and the shift-and-add operators.

3.3.2 Instruction selection as a CGPE filter

Instruction selection is implemented as a middle-end filter and produces a tiling of the DAG that minimizes a cost function. By varying this cost function, the synthesized code can be optimized for different criteria.

The tiling algorithm used is an adaptation of the NOLTIS algorithm (Near-Optimal Linear Time Instruction Selection) introduced by Koes and Goldstein [KG08]. This algorithm is particularly well-suited for DAGs and proceeds in three major steps:

1. Step 1 traverses the DAG and assigns to each node the instruction that minimizes the cost function.
2. Step 2 handles the case of nodes that are covered by more than a tile: either it leaves the tiles as they are, or cuts the sub-DAG rooted at this shared node and marks it as resolved for the rest of the algorithm.
3. Step 3 consists in another round of instruction selection. It differs from Step 1 in the following: it does not try to tile sub-DAGs marked as resolved by step 2.

First, let us remark that the second step of NOLTIS may lead to an increase in the number of operators in the synthesized codes. Second, as shown for instance in [Mou11, § 7.1.1], minimizing the evaluation latency on unbounded parallelism relies on minimizing the maximum of this latency on all operands. Thus no improvement can be expected by running Step 2. Finally this step might be used only for marking shared nodes, that is, nodes computing powers of x. Assuming that an operation fusing several multiplications is at least as accurate as the combination of the single multiplication instructions, for accuracy purposes, this step seems to be useless. For all these reasons, the main expected benefits come from the first step of NOLTIS, even if the second part is necessary to correctly synthesize codes on architectures providing instructions like $(a \times b) \times c$. Hence we present an adaptation of NOLTIS based on its first step only. This is mainly what is done to tile trees, which is easier than tiling DAGs since there is no shared node handling.

Formally, let $\mathscr{G}$ be a node of the DAG and $\mathscr{T}$ be the set of tiles that match this node and that can be used to evaluate it. For a given tile $t \in \mathscr{T}$, children(t) denotes the set of nodes of

the DAG that are children of t. The sequel of this section presents the three cost functions we have implemented.

Minimize the number of instructions. The original NOLTIS algorithm uses a cost function that minimizes the latency on a sequential machine. This can be easily adapted to optimize the number of instructions in the output code. Hence the minimal operator count $\mathscr{C}(\mathscr{G})$ is:

$$\mathscr{C}(\mathscr{G}) = \min_{t\in\mathscr{T}}\Big(\mathscr{C}_t(\mathscr{G})\Big) \quad \text{with} \quad \mathscr{C}_t(\mathscr{G}) = 1 + \sum_{n\,\in\,\text{children}(t)} \mathscr{C}(n).$$

Reduce latency on unbounded parallelism. Our second adaptation of NOLTIS aims at optimizing the evaluation latency on unbounded parallelism. Reducing the evaluation latency of $\mathscr{G}$ relies on the reduction of the maximum latency of its children. Hence the minimal evaluation latency $\mathscr{L}(\mathscr{G})$ is as follows:

$$\mathscr{L}(\mathscr{G}) = \min_{t\in\mathscr{T}}\Big(\mathscr{L}_t(\mathscr{G})\Big)$$

$$\text{with} \quad \mathscr{L}_t(\mathscr{G}) = \text{latency}(t) \quad + \max_{n\,\in\,\text{children}(t)} \mathscr{L}(n).$$

Increase accuracy. Our third adaptation aims at reducing the evaluation error. Hence to decide which tile t leads to the tightest evaluation error, we generate the GAPPA script for the sub-DAG rooted at $\mathscr{G}$, and for each tile $t \in \mathscr{T}$, we compute the evaluation error $\mathscr{E}_t(\mathscr{G})$ and keep the best one.

3.3.3 Experimental results

In order to show the impact of enhancing CGPE with instruction selection, this section presents data obtained from 3 experiments. Each experiment shows the impact of one of the cost functions presented in the previous section.

Impact on the number of operations

In this first experiment, we consider a set of polynomials of degrees 5 up to 12 that approximate the functions $\cos(x)$ and $\sin(x)$ over $[-1,1]$, and $\log_2(1+x)$ over $[-0.5,0.5]$. These polynomials were computed using the `fpminimax` function [BC07] of the software tool Sollya.[7] For each polynomial, 50 programs were synthesized, with each being optimized for the use of a particular instruction among the following:

[7] See `http://sollya.gforge.inria.fr` and [Lau08].

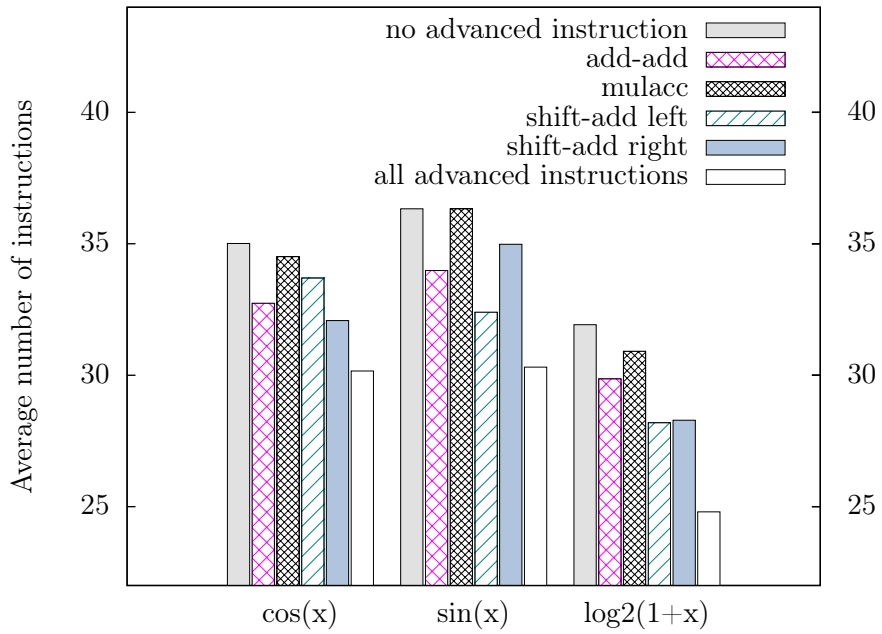


Figure 3.5: Average number of instructions in the synthesized codes, for the evaluation of polynomials of degree 5 up to 12 for various elementary functions.

- an `add-add` that computes `(a + b) + c`,
- a `mulacc` that computes `(a * b) + c`,
- a `shift-and-add` left that computes `(a ≪ b) + c` with $b \in [1,4]$,
- a `shift-and-add` right that computes `(a ≫ b) + c` with $b \in [1,4]$.

For each function and each advanced instruction above, Figure 3.5 shows the average number of operations in the output code. It also shows the number of operations when no advanced instructions are used as well as when all of them are available for use.

From these results, we can observe that thanks to our technique, we reduce the number of instructions in the generated codes of 13.9% up to 22.3% depending on the function when all the advanced instructions are considered, and of 8.3% up to 11.7% otherwise. For example, the evaluation of the $\sin(x)$ function illustrates the interest of the `shift-and-add` operator available on the ST200 family cores, since, in this case, it is the most valuable operator and leads to a gain of 10.8%.

Also, remark that our implementation of instruction selection allows us to test for unimplemented instructions and to eventually give feedback on the ones that would be valuable to have in hardware. For instance, Figure 3.5 shows that an operator similar to the ST231's `shift-and-add`, but with a right shift instead of a left shift, would be of great use for evaluating polynomials in signed fixed-point arithmetic. Indeed, in the examples

of $\cos(x)$ and $\log_2(1+x)$ of Figure 3.5, this is the most relevant instruction since it leads to a reduction of the operation count of 8.4% and 11.4%, respectively.

Impact on the accuracy for some functions

This second experiment illustrates the impact of the accuracy based selection on various functions. For this purpose, we consider a set of mathematical functions, and we approximate each function $f(x)$ by a degree-d polynomial over an interval $\mathscr{I}$ using the `fpminimax` function. For this experiment we fix the fixed-point format of odd and even coefficients to $\mathbf{Q}_{1,31}$ and $\mathbf{Q}_{3,29}$, respectively. Then for each polynomial, we generate two fixed-point codes: one is optimized for accuracy, and the other not. This generation process took no more than a few seconds. In order to observe the impact of the accuracy based selection, we also define a basic block called `fx_fma` behaving in fixed-point arithmetic like the floating-point FMA, and computing:

$$(a \times b) + (c \gg n) \quad \text{with} \quad n \in \{1, \cdots, 31\}$$

with only one final truncation. Table 3.1 summarizes the results, where the accuracy of the generated code is shown as $\log_2(|\epsilon|)$ with ϵ being the certified error bound computed using GAPPA.

$f(x)$	$\mathscr{I}$	d	Accuracy	
			not optimized	optimized
$\exp(x)-1$	$[-0.25, 0.25]$	7	-26.98	-27.34
$\exp(x)$	$[0, 1]$	7	-13.94	-14.90
$\sin(x)$	$[-0.5, 0.5]$	9	-18.95	-19.91
$\cos(x)$	$[-0.5, 0.25]$	5	-27.01	-27.26
$\tan(x)$	$[0.25, 0.5]$	9	-18.81	-19.64
$\log_2(1+x)/x$	$[2^{-23}, 1]$	7	-13.94	-14.89
$\sqrt{1+x}$	$[2^{-23}, 1]$	7	-13.94	-14.90

Table 3.1: Impact of the accuracy based selection step on the certified accuracy of the generated code for various functions.

We can observe that our technique allows to automatically improve the certified accuracy of the generated code by approximately 1 bit for most functions. For example, without optimization, CGPE produces a code with a certified error bound of $\approx 2^{-18.95}$ for the

degree-9 polynomial approximating $\sin(x)$. This bound is reduced to $2^{-19.91}$ when the accuracy based selection is used, that is, this technique divides the error bound by approximately 2.

Our multi-criteria instruction selection is a synthesis-time process, and it consists in using a specific instruction only when it improves the cost function, here the accuracy. And supporting a new instruction like the `fx_fma` requires nothing more than adding its description to the architecture XML file. This is much faster and less error prone than designing a new DAG computation algorithm.

Impact on the evaluation latency for $\cos(x)$

In this third experiment, we show the impact of latency based selection. Note that the function itself does not influence the latency of the generated code, but only the degree of its approximant polynomial. Hence, let us consider a degree-7 polynomial approximating the function $\cos(x)$ over $[0,2]$ without any constraint on the fixed-point format of its coefficients. Here we assume 1-cycle addition, subtraction, and shift-and-add operator (as specified in the ST231), and 3-cycle multiplication and `mulacc` operation. We have synthesized codes to evaluate $\cos(x)$ using Horner's and Estrin's rules, two classical evaluation schemes, as well as code generated automatically by CGPE. Table 3.2 gives the latency on unbounded parallelism of these codes, before and after tiling. We observe that the code after tiling has a lower latency than before tiling, and that the speed-up may be up to ≈ 17% for Horner's rule.

	Without tiling	**With tiling**	**Speed-up**
Horner's rule	41	34	≈ 17.1%
Estrin's rule	16	14	≈ 12.5%
Best scheme generated by CGPE	15	13	≈ 13.3%

Table 3.2: Latency in # cycles on unbounded parallelism, for various schemes, with and without tiling.

Obviously, we can remark that it is of great interest to provide hardware support for instructions fusing several operations, and having a lower latency than the sum of the latencies of all fused operations.

3.4 Case study: code synthesis for an IIR filter

The original CGPE tool was limited to polynomial evaluation in unsigned arithmetic. After extensive enhancements, it is now a library that can handle different types of input expres-

sions in signed and unsigned arithmetics. Indeed, in this section, we show how to use it to obtain a certified fixed-point implementation for an IIR filter. The low-pass filter we shall implement was first introduced in Example 1.9 of Chapter 1 and has a cutoff frequency of $0.3 \cdot \pi$. It is a fixed-point approximation of a 3rd order Butterworth filter whose transfer function H is given by:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3}}{1 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3}}, \tag{3.4}$$

where the coefficients b_i and a_j are described in Table 3.3.

	Int. Repr.	**Format**	**Decimal value**
b_0	1701940795	$\mathbf{Q}_{-3,35}$	0.049532996345078572630882263183593750
b_1	1276455597	$\mathbf{Q}_{-1,33}$	0.148598989122547209262847900390625
b_2	1276455597	$\mathbf{Q}_{-1,33}$	0.148598989122547209262847900390625
b_3	1701940795	$\mathbf{Q}_{-3,35}$	0.049532996345078572630882263183593750
a_1	−1247599398	$\mathbf{Q}_{2,30}$	−1.161917483434081077575683593750
a_2	1494525688	$\mathbf{Q}_{1,31}$	0.695942755788564682006835937500
a_3	−1183360567	$\mathbf{Q}_{-1,33}$	−0.137761301244609057903289794921875

Table 3.3: The coefficients and fixed-point formats of the Butterworth filter 3.4.

A block diagram of this filter is shown in Figure 3.6 and we shall realize it using Equation (3.5) that relates its output $y[k]$ at time step k to its input $u[k]$ and its state.

$$y[k] = \sum_{i=0}^{3} b_i \cdot u[k-i] - \sum_{i=1}^{3} a_i \cdot y[k-i], \tag{3.5}$$

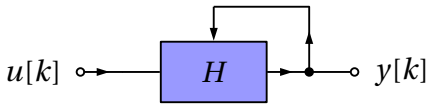


Figure 3.6: Transfer function of the Butterworth filter 3.4.

Next, we shall consider that the input $u[k]$ belongs to the enclosure $[-15.5, 15.5]$ and assign it to a $\mathbf{Q}_{5,27}$ variable. Finally, we showed in Example 1.9 of Chapter 1 that a range analysis based on the ℓ_1-norm of the filter yields the enclosure $[-19.52237503163826936, 19.52237503163826936]$ as a bound for the output of the filter. Therefore, we assign the $\mathbf{Q}_{6,26}$ format to the variable $y[k]$.

Table 3.4 summarizes the fixed-point and decimal enclosures on the values of $u[k]$ and $y[k]$.

	Enclosure of the Int. Repr.	**Format**	**Enclosure in decimal**
$u[k]$	$[-2080374784, 2080374784]$	$\mathbf{Q}_{5,27}$	$\mathscr{U} = [-15.5, 15.5]$
$y[k]$	$[-1310124411, 1310124411]$	$\mathbf{Q}_{6,26}$	$\mathscr{Y} = [-v, v]$ where $v = 19.522375032305717468261718 75$

Table 3.4: The enclosures and formats of the input and output of the filter 3.4.

3.4.1 Code generation for the IIR filter

Listing 3.6 shows the `XML` input file for this filter. In Line 1 of this listing, the type of expression is set to dot-product. Indeed, computing $y[k]$ requires a dot-product between the vector of coefficients and the vector of variables shown in Equation (3.6).

$$y[k] = \Big[\, b_0\, b_1\, b_2\, b_3\, -a_1\, -a_2\, -a_3 \,\Big] \cdot \Big[\, u[k]\; u[k-1]\; u[k-2]\; u[k-3]\; y[k-1]\; y[k-2]\; y[k-3] \,\Big]^T \tag{3.6}$$

The number of different evaluation schemes for a size-7 dot-product is 10395 and one can afford to test all of them. Indeed, CGPE generates code for all of the schemes in less than 3 minutes on an Intel Core i7-870 desktop machine running at 2.93 GHz. Among these 10395 schemes, only 3 have the lowest evaluation error. And among these 3 schemes, the `C` code of Listing 3.7 is the fastest on unbounded parallelism since it requires only 13 cycles.

By checking the GAPPA script generated by CGPE, one makes sure that the error bounds are indeed correct. In this case, the scheme used is the one shown in Figure 3.7 and we have that:

$$\mathbf{Err}(\texttt{r17}) = \left[\underline{e}, \overline{e}\right] = \left[-380104605495 \cdot 2^{-61}, 0\right] \approx \left[-2^{-22.5324}, 0\right]. \tag{3.7}$$

To differentiate it from the other schemes, we will call this evaluation scheme S_1. Notice that since intermediate computations are always truncated and never rounded, the enclosure on the error is not centered around 0. Hence, it is possible to reduce the magnitude of the error bound by adding the corrective term $\frac{-\underline{e}}{2}$ to `r17`. By doing so, the enclosure on the error is brought to $\approx \left[-\frac{\underline{e}}{2}, \frac{-\underline{e}}{2}\right]$. We will denote by S_1' this slightly corrected scheme.

Remark on the sign of the error bound computed by CGPE

To maintain backward compatibility, CGPE uses the following relationship between $\mathbf{Math}(v)$, $\mathbf{Val}(v)$, and $\mathbf{Err}(v)$:

$$\mathbf{Math}(v) = \mathbf{Val}(v) - \mathbf{Err}(v).$$

▷ **Listing 3.6:** The XML input file for the filter 3.4.

```
<dotproduct inf="0xb1e91685" sup="0x4e16e97b" integer_width="6"
    fraction_width="26" width="32">
  <coefficient name="b0" value="0x65718e3b" integer_width="-3"
      fraction_width="35" width="32"/>
  <coefficient name="b1" value="0x4c152aad" integer_width="-1"
      fraction_width="33" width="32"/>
  <coefficient name="b2" value="0x4c152aad" integer_width="-1"
      fraction_width="33" width="32"/>
  <coefficient name="b3" value="0x65718e3b" integer_width="-3"
      fraction_width="35" width="32"/>
  <coefficient name="na1" value="0x4a5cdb26" integer_width="2"
      fraction_width="30" width="32"/>
  <coefficient name="na2" value="0xa6eb5908" integer_width="1"
      fraction_width="31" width="32"/>
  <coefficient name="na3" value="0x4688a637" integer_width="-1"
      fraction_width="33" width="32"/>

  <variable name="u0" inf="0x84000000" sup="0x7c000000" integer_width="5"
      fraction_width="27" width="32"/>
  <variable name="u1" inf="0x84000000" sup="0x7c000000" integer_width="5"
      fraction_width="27" width="32"/>
  <variable name="u2" inf="0x84000000" sup="0x7c000000" integer_width="5"
      fraction_width="27" width="32"/>
  <variable name="u3" inf="0x84000000" sup="0x7c000000" integer_width="5"
      fraction_width="27" width="32"/>
  <variable name="y1" inf="0xb1e91685" sup="0x4e16e97b" integer_width="6"
      fraction_width="26" width="32"/>
  <variable name="y2" inf="0xb1e91685" sup="0x4e16e97b" integer_width="6"
      fraction_width="26" width="32"/>
  <variable name="y3" inf="0xb1e91685" sup="0x4e16e97b" integer_width="6"
      fraction_width="26" width="32"/>
</dotproduct>
```

This is different from the error model we presented in Chapter 2 where we used the formula:

$$\mathbf{Math}(v) = \mathbf{Val}(v) + \mathbf{Err}(v).$$

$\mathbf{Math}(v)$ and $\mathbf{Val}(v)$ are the same in the two models and the error bound produced by CGPE must only be negated to fit in the framework of Chapter 2. Furthermore, this difference is barely noticeable since one is usually more interested in the magnitude of the error.

▷ **Listing 3.7:** The C code generated by CGPE for the filter 3.4.

```
1  int32_t filter( int32_t u0    /*Q5.27*/ ,
2                  int32_t u1    /*Q5.27*/ ,
3                  int32_t u2    /*Q5.27*/ ,
4                  int32_t u3    /*Q5.27*/ ,
5                  int32_t y1    /*Q6.26*/ ,
6                  int32_t y2    /*Q6.26*/ ,
7                  int32_t y3    /*Q6.26*/ )
8  {                                          //Formats  Err
9    int32_t r0  = mul(0x4a5cdb26, y1);       //Q8.24   [-2^{-24},0]
10   int32_t r1  = mul(0xa6eb5908, y2);       //Q7.25   [-2^{-25},0]
11   int32_t r2  = mul(0x4688a637, y3);       //Q5.27   [-2^{-27},0]
12   int32_t r3  = mul(0x65718e3b, u0);       //Q2.30   [-2^{-30},0]
13   int32_t r4  = mul(0x65718e3b, u3);       //Q2.30   [-2^{-30},0]
14   int32_t r5  = r3 + r4;                   //Q2.30   [-2^{-29},0]
15   int32_t r6  = r5 >> 2;                   //Q4.28   [-2^{-27.6781},0]
16   int32_t r7  = mul(0x4c152aad, u1);       //Q4.28   [-2^{-28},0]
17   int32_t r8  = mul(0x4c152aad, u2);       //Q4.28   [-2^{-28},0]
18   int32_t r9  = r7 + r8;                   //Q4.28   [-2^{-27},0]
19   int32_t r10 = r6 + r9;                   //Q4.28   [-2^{-26.2996},0]
20   int32_t r11 = r10 >> 1;                  //Q5.27   [-2^{-25.9125},0]
21   int32_t r12 = r2 + r11;                  //Q5.27   [-2^{-25.3561},0]
22   int32_t r13 = r12 >> 2;                  //Q7.25   [-2^{-24.3853},0]
23   int32_t r14 = r1 + r13;                  //Q7.25   [-2^{-23.6601},0]
24   int32_t r15 = r14 >> 1;                  //Q8.24   [-2^{-23.1798},0]
25   int32_t r16 = r0 + r15;                  //Q8.24   [-2^{-22.5324},0]
26   int32_t r17 = r16 << 2;                  //Q6.26   [-2^{-22.5324},0]
27   return r17;
28 }
```

3.4.2 Comparison between our implementation and the ideal IIR filter

Rounding errors occur at each time step of our filter implementation. Therefore, its output differs from the perfect filter of Equation (3.4). Indeed, our implementation is described by the block diagram of Figure 3.8 and by the following equation that relates its output $y'[k]$ to its input $u[k]$ at time step k:

$$y'[k] = \sum_{i=0}^{3} b_i \cdot u[k-i] - \sum_{i=1}^{3} a_i \cdot y'[k-i] + \epsilon[k], \tag{3.8}$$

where $\epsilon[k] \in$ **Err**(r17).

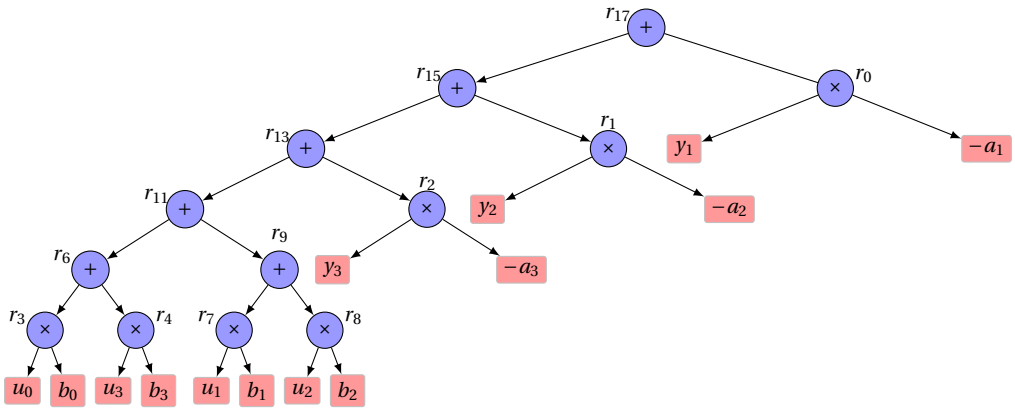


Figure 3.7: DAG of the scheme S_1 of Listing 3.7 without the shifting nodes.

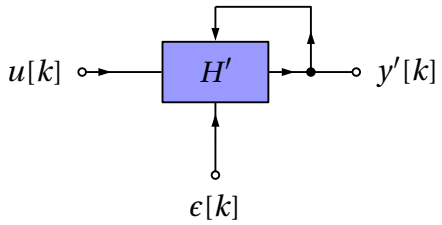


Figure 3.8: Block diagram of our implementation of the Butterworth filter 3.4.

However, as shown by Didier *et al.* [LHD14], the output $y'[k]$ at step k of our implementation is related to the output of the ideal filter $y[k]$ by the following equation:

$$\begin{aligned} \epsilon_y[k] &= y'[k] - y[k] \\ &= \epsilon[k] - \sum_{i=1}^{3} a_i \cdot (y'[k-i] - y[k-i]) \\ &= \epsilon[k] - \sum_{i=1}^{3} a_i \cdot (\epsilon_y[k-i]) \end{aligned} \tag{3.9}$$

And the key observation is that Equation (3.9) is the realization of a filter whose transfer function H'' is given by:

$$H''(z) = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3}}. \tag{3.10}$$

At time step k, the input of filter (3.10) is $\epsilon[k]$ and its output is $\epsilon_y[k]$.

Now the relationship between the ideal filter, our implementation, and the filter of Equation (3.10) is better described by the block diagram of Figure 3.9.

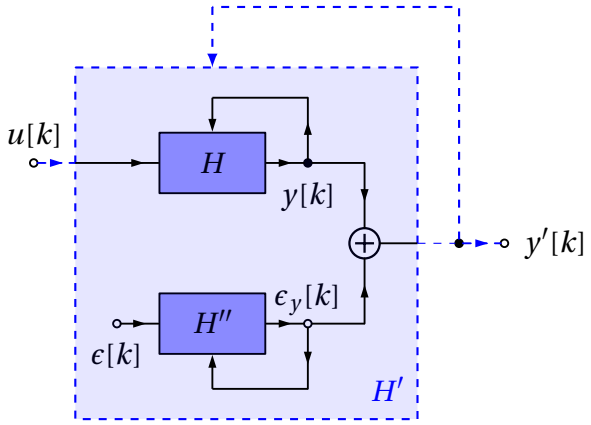


Figure 3.9: The relationship between the implemented filter and the ideal filter 3.4.

And estimating the absolute error of our implementation comes down to bounding the values of $\epsilon_y[k]$. As shown in Section 1.3.4 of Chapter 1, this is achievable by computing the ℓ_1-norm of the filter H'' and by deducing a bound on its output by the formula:

$$\|\epsilon_y\|_\infty \leq \|H''\|_{\ell_1} \cdot \|\epsilon\|_\infty. \tag{3.11}$$

By approximating the ℓ_1-norm of H'' using the expression

$$\|H''\|_{\ell_1} \approx \sum_{0}^{1024} |h''(k)|,$$

we obtain the value 54.2927105220054074 as the gain of this filter. Using Equation (3.11), we deduce that

$$\|\epsilon_y\|_\infty \leq -8.949832764746 \cdot 10^{-6} \leq 2^{-16.76}. \tag{3.12}$$

This bound also implies that

$$y'[k] \in \left(\mathscr{Y} + [-2^{-16.76}, 2^{-16.76}]\right) \subset [-20,20] \subset \mathscr{R}\text{ange}(\mathbf{Q}_{6,26}),$$

which reassures us that the initial choice of the format $\mathbf{Q}_{6,26}$ for y is correct and convinces us that y' won't overflow this format.

3.4.3 Experimental errors of the IIR filter implementation

CGPE's back-end is modular and has many code generators that read annotated DAGs and write code. This allows the tool to generate code that evaluates the same input problem using the floating-point binary32 and binary64 formats.

Hence, as a first experiment to assert the low pass properties of our filter, we generated a noisy signal composed by summing a low and a high frequency sinusoids and filtered

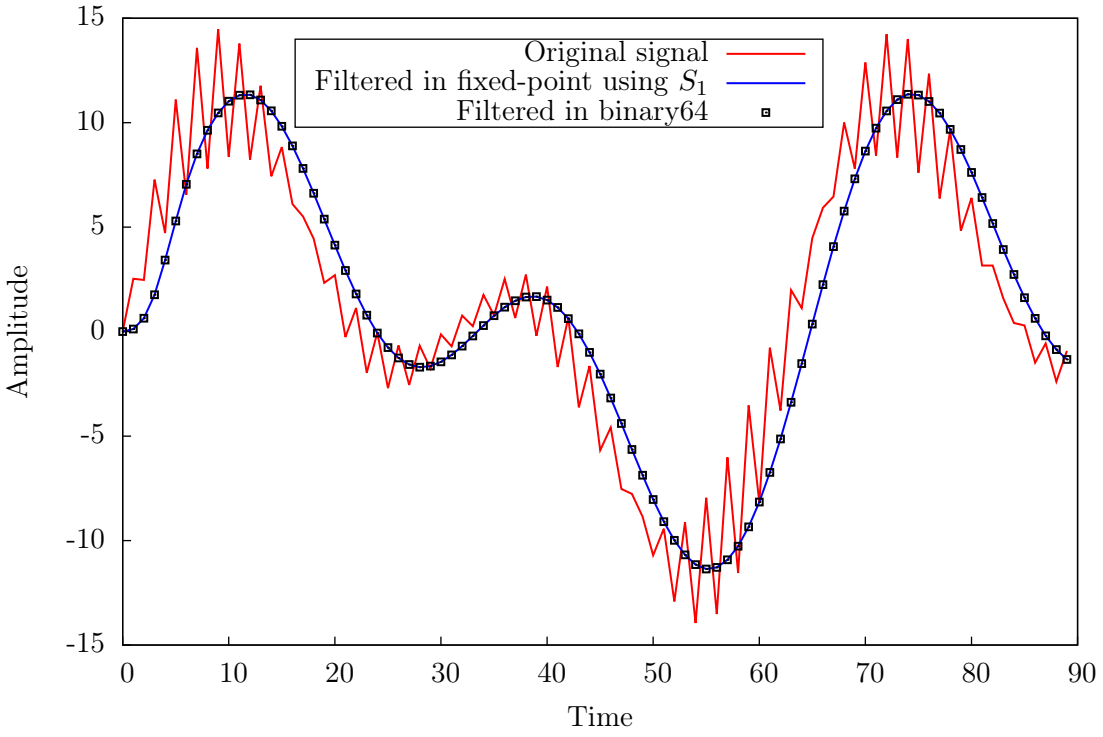


Figure 3.10: A noisy signal filtered in fixed-point and floating-point arithmetics using filter 3.4.

it both in fixed-point and using the binary64 floating-point arithmetic. As shown in Figure 3.10, both in fixed-point and floating-point, the output signal is cleaned from its high frequency components.

Figure 3.10 also shows that the output of the fixed-point implementation follows closely the floating-point one. The next experiment shows to what extent this observation holds.

Comparing the experimental errors of the filter

To have a reference implementation, we implemented the filter using the multi-precision library MPFR [FHL+07]. We used 512-bit numbers which guarantees that the implementation computes a highly accurate result for this filter.

Figure 3.11 shows the experimental errors, that is, the difference between the output of our synthesized implementations and the output of the MPFR variant. These errors were obtained by running these implementations on the sinusoidal signal of the previous experiment. Figure 3.11 shows that our fixed-point implementation is slightly more accurate than a binary32 implementation. It also assures us of the correctness of the accuracy

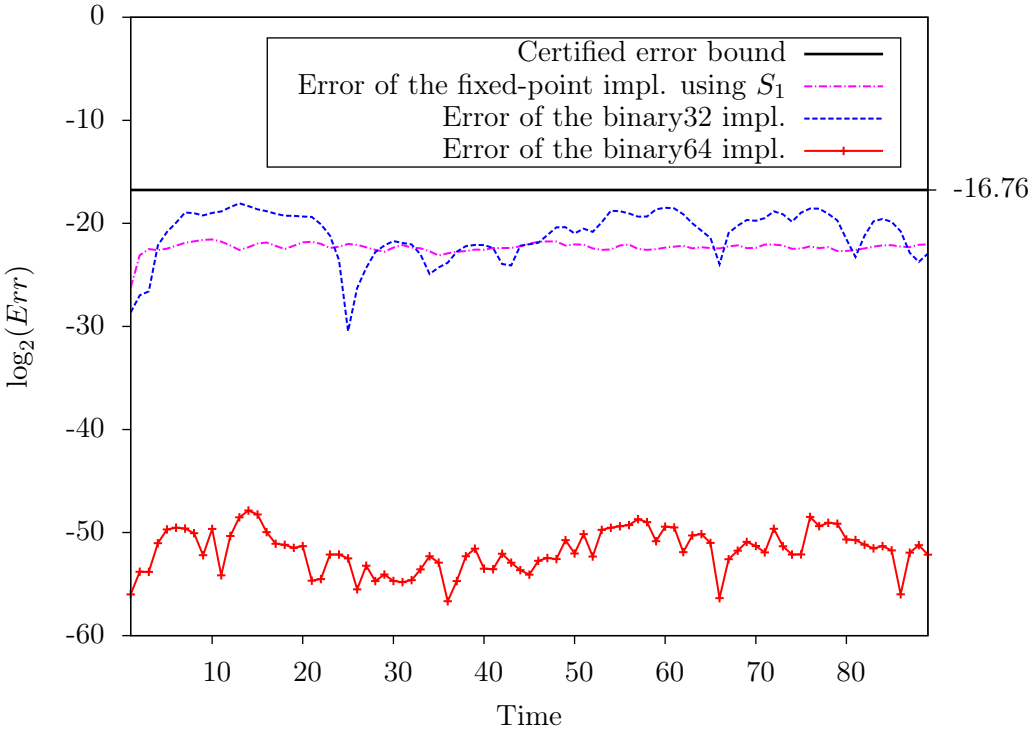


Figure 3.11: Experimental error of the fixed-point and floating-point implementations of filter 3.4 based on the MPFR reference implementation.

bound. Indeed, the experimental errors of the scheme S_1 are inferior in magnitude to the error bound given by the inequality of Equation (3.12).

Comparing different fixed-point evaluation schemes

In this last experiment, we compare the bounds and the experimental errors of 3 different schemes to evaluate the filter in fixed-point. Here, S_1 is the scheme obtained by CGPE and whose latency on unbounded parallelism is of 13 cycles assuming 1-cycle addition and 3-cycles multiplication, and S_1' is the corrected variant of S_1 scheme. Hence, it is more accurate but needs 14 cycles due to the extra addition. Finally, S_2 is a scheme that executes in only 11 cycles but has a larger error bound. Figure 3.12 shows that the precision hierarchy between the schemes obtained by theoretical means is indeed validated by the experiments. For example, the scheme S_1' which is a mere correction of scheme S_1 is always more accurate after a significant number of filter iterations. The figure also shows that the difference in accuracy between the accurate scheme, i.e., S_1' and the fast scheme S_2 are significant for such a small example. Indeed, as shown by Table 3.5 which summarizes the bounds and the maximum experimental errors of our implementations, S_1' is

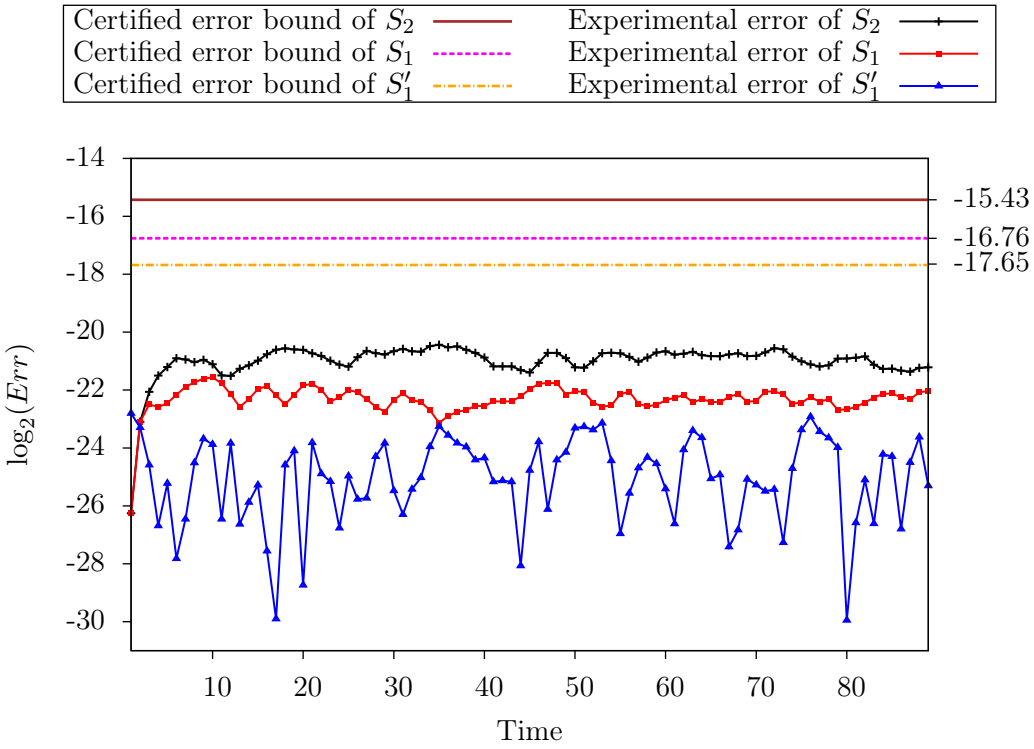


Figure 3.12: Bounds and experimental errors of 3 different fixed-point implementations of filter 3.4 based on the MPFR reference implementation.

two bits more accurate than S_2. This stresses the importance of relying on tools to find accuracy versus latency trade-offs when choosing an evaluation scheme. Finally, Table 3.5 also shows that, experimentally, all the fixed-point schemes are more accurate than the binary32 floating-point implementation.

	S_1	S'_1	S_2	binary32
Error bound	$2^{-16.76}$	$2^{-17.65}$	$2^{-15.43}$	–
Maximum experimental error	$2^{-21.56}$	$2^{-22.80}$	$2^{-20.43}$	$2^{-18.05}$
Average experimental error	$2^{-22.32}$	$2^{-25.07}$	$2^{-21.01}$	$2^{-21.07}$

Table 3.5: The bounds and experimental errors of our filter implementations.

3.4.4 Summary on code synthesis for IIR filters

Using CGPE, code synthesis for IIR filters is entirely automated. Indeed, the tool comes with helper scripts that take the description of a filter (cutoff frequency, order, ...) and generate the corresponding `XML` input file. Yet, further improvement can still be achieved as follows: as pointed out in Section 1.2.1 of Chapter 1, ALUs provide modular arithmetic instead of integer arithmetic. While this low level detail may seem inconvenient, one can use it to his advantage in some situations. Indeed, in our filter implementation, some of the summands require up to 8 integer part bits. However, the result of their sum was proved by range analysis to require no more than 6 bits. This implies that the MSBs of the operands are canceled by the sum and suggests that one may allocate only 6 integer bits for these variables and simply ignore overflows.

This optimization reduces the rounding errors since the discarded integer bits get allocated for the fraction part. It also removes some useless alignment shifts, thus saving few evaluation cycles. In the context of filters design, this property was first used by Jackson [Jac70]. A rigorous proof of its soundness is given by Didier *et al.* [LHD14] who use it in their fixed-point generation framework.

3.5 Conclusion

This chapter presents the CGPE software tool. This tool implements the arithmetic model of Chapter 2. It has a compiler-like architecture and internally represents arithmetic expressions using DAGs. Its first stage consists of generators that are dependent on the input problem. Its middle-end applies the arithmetic model to the DAGs generated by the first stage and decides of the DAGs to keep according to accuracy or latency criteria set by the user. The DAGs that survive this pruning process reach the back-end which traverses the DAG to generate codes and accuracy certificates. CGPE has back-ends that generate fixed-point and floating-point `C` code. Recently, thanks to the FloPoCo library [dDP11], support was added for VHDL code generation as well.

Contrarily to many fixed-point generation tools, CGPE is open-source and is also available in library form which allows it to be used as a low level fixed-point generation framework. Indeed, the algorithms presented in the next part concentrate on solving higher level problems while relying on CGPE to generate code for basic expressions such as dot-products.

Part II

Study of the trade-offs in the synthesis of fixed-point programs for linear algebra basic blocks

CHAPTER 4

FIXED-POINT CODE SYNTHESIS FOR MATRIX MULTIPLICATION

This chapter, like the following two chapters, builds on the arithmetic model introduced in Part I. It describes how to synthesize fixed-point code for matrix multiplication. To do so, its starts by a summary of straightforward approaches where it is shown that they either yield compact but inaccurate code, or very large and accurate codes. To tackle this problem, the rest of the chapter describes the means to achieve trade-offs between code size and accuracy when synthesizing fixed-point code for matrix multiplication.

4.1 Introduction

In the previous chapter, we showed that our arithmetic model and the CGPE tool that implements it allow us to generate certified code for expressions such as dot-products, sums, and linear filters. Naturally, the next step is to ask if this framework would scale, i.e., that is would be able to synthesize accurate code for larger problems. For instance, a problem that is just one level higher than dot-products is that of synthesizing code for matrix multiplication.[1] In practice, matrix multiplication is useful for applying linear transfor-

[1] We implicitly consider that code synthesis for matrix-vector products is a sub-case of matrix multiplication.

mations to input data derived from signal or image processing. It is also useful for solving linear systems by iterative methods [GVL96] and for matrix inversion. For instance, the Cholesky decomposition based matrix inversion presented in Chapter 5 involves multiplying the inverse of triangular matrices.

In floating-point, matrix multiplication that uses the direct definition below is straightforward to implement.

$$(AB)_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}. \tag{4.1}$$

Listing 4.1 shows how to do it in just three nested loops. In practice, matrix multiplication for scientific computations is provided through the GEMM routine in highly optimized BLAS libraries [ABB+99]. And these libraries go beyond Equation (4.1) and use asymptotically fast algorithms such as Strassen's [Hig02].

▷ **Listing 4.1:** A C implementation of floating-point matrix multiplication.

```
1 int main(void)
2 {
3   int i,j,k;
4   float A[N][N]={...}, B[N][N]={...}, C[N][N]={0,...,0};
5   for (i = 0; i < N ; i++)
6     for (j = 0; j < N ; j++)
7       for (k = 0; k < N ; k++)//computes the dot-product of row i and column j
8         C[i][j]+=A[i][k]*B[k][j];
9 }
```

In fixed-point arithmetic, however, works that treat matrix multiplications are scarce. Abdul Gaffar *et al.* [LGC+06] use a certified approach to generate fixed-point codes for small 2×2 matrices as well as size-8 DCT. However, the DCT matrix is structured and constant. On the other hand, Frantz *et al.* [NNF07] do not treat matrix multiplication explicitly but describe a tool that implements many linear algebra primitives such as SVD, LU, and QR decompositions. Their work is based on Sung's technique [KiKS96] to convert floating-point designs into fixed-point and presents the following drawbacks:

1. Simulation time is exponentially proportional to the number of input variables. It is therefore impractical for large input matrices.
2. It provides no strict guaranties on rounding errors.

Indeed, in our work [MNR14a], we were able to generate code for size-64 matrices, which are considered large in fixed-point arithmetic, and necessitate trade-offs between code size and accuracy. For such problem that involves 8192 input variables, simulation methods do not scale and only an analytic approach like ours is practical.

According to Equation (4.1), matrix multiplication can be decomposed into multiple dot-product computations. And since CGPE supports code generation for dot-products, a simple synthesis algorithm for matrix multiplication may consist in invoking CGPE as many times as required to generate code for each dot-product. This kind of straightforward algorithms is investigated in Section 4.2 of this chapter. In this section, it is shown that straightforward algorithms may result in codes whose size is large (too many dot-product codes generated) or whose error bound is not sufficiently tight to satisfy the numerical quality demanded by the programmer. Then, Section 4.3 suggests a strategy to find trade-offs between straightforward algorithms. This strategy is implemented and experimental data is presented to show its effectiveness on a set of benchmarks in Section 4.4.

4.2 Straightforward approaches for the synthesis of matrix multiplication programs

In this section, we give a statement of the problem of code synthesis for matrix multiplication and discuss two straightforward approaches to solve it. The second approach uses the notion of merging two fixed-point variables. We therefore start this section by introducing this notion. Towards the end of the section, we give some code size and accuracy estimates for both approaches. In the next section, we will add accuracy and code size constraints to the problem and try to solve it by finding a trade-off between the approaches shown in this section.

4.2.1 Merging two fixed-point variables

Let $(x, y) \in \mathbb{F}\text{ix}^2$ be two fixed-point variables. To compute their union, one must determine $z \in \mathbb{F}\text{ix}$ such that $\mathbf{Q}_{a_z,b_z}$ and $\mathscr{I}_Z$ are, respectively, the smallest format and enclosure that accommodate the values of x and y without overflow. To compute z, we use Algorithm 4.2 described below. Notice that this merging entails a rounding error when the two variables are in a different fixed-point format. Indeed, this error is the result of the run-time shift one must insert to align the variable that has the smaller fixed-point format.

Example 4.1. Consider the three 32-bit signed fixed-point variables $(x, y, t) \in \mathbb{F}\text{ix}^3$ shown in Table 4.1. The last two rows show the mergings $x \cup y$ and $x \cup t$.

Algorithm 4.2 Computing $z \in \mathbb{F}\text{ix}$ such that $z = x \cup y$ and $(x, y) \in \mathbb{F}\text{ix}^2$.

Input:

Two variables $(x, y) \in \mathbb{F}\text{ix}^2$ such that
$\mathscr{I}_X = \left[\underline{i_x}, \overline{i_x}\right]$ and $\mathscr{I}_Y = \left[\underline{i_y}, \overline{i_y}\right]$

Output:

$z \in \mathbb{F}\text{ix}$ such that $z = x \cup y$, and
Err(z), a bound on the error induced by this merge

Algorithm:

1: **if** $\mathbf{Q}_{a_x,b_x} \geq \mathbf{Q}_{a_y,b_y}$ **then**
2: $\quad \mathbf{Q}_{a_z,b_z} \leftarrow \mathbf{Q}_{a_x,b_x}$
3: $\quad \mathscr{I}_Z \leftarrow \left[\min\left(\underline{i_x}, \frac{\underline{i_y}}{a_x - a_y}\right), \max\left(\overline{i_x}, \frac{\overline{i_y}}{a_x - a_y}\right)\right]$
4: $\quad$ **Err**$(z) \leftarrow \left[0, 2^{-b_x} - 2^{-b_y}\right]$
5: **else**
6: $\quad \mathbf{Q}_{a_z,b_z} \leftarrow \mathbf{Q}_{a_y,b_y}$
7: $\quad \mathscr{I}_Z \leftarrow \left[\min\left(\frac{\underline{i_x}}{a_y - a_x}, \underline{i_y}\right), \max\left(\frac{\overline{i_x}}{a_y - a_x}, \overline{i_y}\right)\right]$
8: $\quad$ **Err**$(z) \leftarrow \left[0, 2^{-b_y} - 2^{-b_x}\right]$
9: **end if**

	Format	Interval of the Int. Repr.	Decimal enclosure	Error
x	$\mathbf{Q}_{3,29}$	$\mathscr{I}_X = \left[-2^{31}, 2^{28}\right]$	$[-4, 0.5]$	$[0, 0]$
y	$\mathbf{Q}_{3,29}$	$\mathscr{I}_Y = \left[-2^{26}, 2^{30}\right]$	$[-0.125, 2]$	$[0, 0]$
t	$\mathbf{Q}_{4,28}$	$\mathscr{I}_T = \left[-2^{27}, 2^{30}\right]$	$[-0.5, 4]$	$[0, 0]$
$z' = x \cup y$	$\mathbf{Q}_{3,29}$	$\mathscr{I}_{Z'} = \left[-2^{31}, 2^{30}\right]$	$[-4, 2]$	$[0, 0]$
$z'' = x \cup t$	$\mathbf{Q}_{4,28}$	$\mathscr{I}_{Z''} = \left[-2^{30}, 2^{30}\right]$	$[-4, 4]$	$\left[0, 2^{-29}\right]$

Table 4.1: The union of fixed-point variables, their resulting range in decimal, and their error.

Since x and y have the same fixed-point format $\mathbf{Q}_{3,29}$, $z' = x \cup y$ is also in the same fixed-point format. As for $\mathscr{I}_{Z'}$, we have $\mathscr{I}_{Z'} = \left[\min\left(-2^{31}, -2^{26}\right), \max\left(2^{28}, 2^{30}\right)\right] = \left[-2^{31}, 2^{30}\right]$. This merging comes at no cost in accuracy since both variables are in the same fixed-point format.

However, x and t are not in the same fixed-point format. In this case, the format of their union z'' is the largest among both formats, i.e., $\mathbf{Q}_{4,28}$. As for $\mathscr{I}_{Z''}$, one must be careful not to consider only $\mathscr{I}_T$, indeed, although x has a smaller fixed-point format, the upper bound on its enclosure, when normalized, is larger than that of t. In Algorithm 4.2,

the computations that lead to the correct enclosure $\mathscr{I}_{Z''}$ are carried out either in Line 3 or Line 7.

4.2.2 Problem statement

Let A and B be two matrices of fixed-point variables of size $m \times n$ and $n \times p$, respectively:

$$A \in \mathbb{F}\text{ix}^{m \times n} \quad \text{and} \quad B \in \mathbb{F}\text{ix}^{n \times p},$$

and let us denote by $A_{i,:}$ and $A_{:,j}$ the i^{th} row and j^{th} column of A, respectively, and $A_{i,j}$ the element of the i^{th} row and j^{th} column of A.

The objective is to generate fixed-point code to multiply A and B. And, since A and B are matrices of fixed-point variables, the generated code should be able to multiply at run-time any matrices A' and B', where A' and B' are two matrices that belong to A and B, that is, where the fixed-point numbers $A'_{i,k}$ and $B'_{k,j}$ belong to the fixed-point variables $A_{i,k}$ and $B_{k,j}$, respectively.

This consists in writing a program for computing $C = A \cdot B$, where $C \in \mathbb{F}\text{ix}^{m \times p}$. Therefore, $\forall i, j \in \{1, \cdots, m\} \times \{1, \cdots, p\}$, we have:

$$C_{i,j} = A_{i,:} \cdot B_{:,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}, \tag{4.2}$$

4.2.3 Accurate approach

Suppose we have access to a routine DPSynthesis(U, V) that synthesizes code for the dot-product of two vectors of fixed-point variables U and V.[2] Following Equation (4.2), a first straightforward approach to generate code for matrix multiplication consists in synthesizing a program for each dot-product $A_{i,:} \cdot B_{:,j}$. Algorithm 4.3 below implements this approach.

Algorithm 4.3 Accurate algorithm.

Input:

Two matrices $A \in \mathbb{F}\text{ix}^{m \times n}$ and $B \in \mathbb{F}\text{ix}^{n \times p}$

Output:

Code to compute the product $A \cdot B$

Algorithm:

```
1: for 1 ≤ i ≤ m do
2:     for 1 ≤ j ≤ p do
3:         DPSynthesis(A_{i,:}, B_{:,j})
4:     end for
5: end for
```

[2] This routine is provided by CGPE.

Algorithm 4.3 issues $m \times p$ queries to the DPSynthesis routine. At runtime, only one call to each generated code will be issued, for a total of $m \times p$ calls.

4.2.4 Compact approach

To significantly reduce the number of dot-product codes generated (we call them DPCodes in the following), some of them could be factored to evaluate more than one dot-product at run-time. Algorithm 4.4 whose input and output are the same as Algorithm 4.3, pushes this idea to the limits by merging element by element the matrices A and B into a unique row $\mathscr{U}$ and column $\mathscr{V}$, respectively.

Algorithm 4.4 Compact algorithm.

Input:

Two matrices $A \in \mathbb{F}\text{ix}^{m\times n}$ and $B \in \mathbb{F}\text{ix}^{n\times p}$

Output:

Code to compute the product $A \cdot B$

Algorithm:

1: $\mathscr{U} \leftarrow A_{1,:} \cup A_{2,:} \cup \cdots \cup A_{m,:}$, with $\mathscr{U} \in \mathbb{F}\text{ix}^{1\times n}$
2: $\mathscr{V} \leftarrow B_{:,1} \cup B_{:,2} \cup \cdots \cup B_{:,p}$, with $\mathscr{V} \in \mathbb{F}\text{ix}^{n\times 1}$
3: DPSynthesis($\mathscr{U}, \mathscr{V}$)

This approach issues a unique call to the DPSynthesis routine. However, at run-time, $m \times p$ calls to this synthesized code are still needed to evaluate the matrix product.

Example 4.2. Let us now illustrate the differences between these two algorithms by considering the problem of generating code for the product of the following two fixed-point matrices:

$$A = \begin{pmatrix} [-1000, 1000] & [-3000, 3000] \\ [-1, 1] & [-1, 1] \end{pmatrix}$$

and

$$B = \begin{pmatrix} [-2000, 2000] & [-2, 2] \\ [-4000, 4000] & [-10, 10] \end{pmatrix},$$

where $A_{1,1}$ and $B_{1,1}$ are in the format $\mathbf{Q}_{11,21}$, $A_{1,2}$ in $\mathbf{Q}_{12,20}$, $A_{2,1}$, $A_{2,2}$, $B_{2,1}$ in $\mathbf{Q}_{2,30}$, $B_{1,2}$ in $\mathbf{Q}_{3,29}$, and $B_{2,2}$ in $\mathbf{Q}_{5,27}$. Algorithm 4.3 produces 4 distinct codes, denoted by $\text{DPCode}_{1,1}$, $\text{DPCode}_{1,2}$, $\text{DPCode}_{2,1}$, and $\text{DPCode}_{2,2}$.

On the other hand, Algorithm 4.4 first computes $\mathscr{U}$ and $\mathscr{V}$ as follows:

$$\mathscr{U} = A_{1,:} \cup A_{2,:} = ([-1000, 1000][-3000, 3000])$$

and

$$\mathscr{V} = B_{:,1} \cup B_{:,2} = \begin{pmatrix} [-2000, 2000] \\ [-4000, 4000] \end{pmatrix}.$$

Then, $\text{DPCode}_{\mathscr{U},\mathscr{V}}$ is generated that evaluates the dot-product of $\mathscr{U}$ and $\mathscr{V}$. Tables 4.2 and 4.3 summarize the properties of the codes produced, respectively, by Algorithms 4.3 and 4.4 on this example.

Dot-product	$A_{1,:} \cdot B_{:,1}$	$A_{1,:} \cdot B_{:,2}$	$A_{2,:} \cdot B_{:,1}$	$A_{2,:} \cdot B_{:,2}$
Evaluated using	$\text{DPCode}_{1,1}$	$\text{DPCode}_{1,2}$	$\text{DPCode}_{2,1}$	$\text{DPCode}_{2,2}$
Output format	$Q_{26,6}$	$Q_{18,14}$	$Q_{15,17}$	$Q_{7,25}$
Certified error	$\approx 2^{-5}$	$\approx 2^{-14}$	$\approx 2^{-16}$	$\approx 2^{-24}$
Maximum error	$\approx 2^{-5}$			
Average error	$\approx 2^{-7}$			

Table 4.2: Numerical properties of the 4 codes generated by Algorithm 4.3 for $A \cdot B$.

Dot-product	$A_{1,:} \cdot B_{:,1}$	$A_{1,:} \cdot B_{:,2}$	$A_{2,:} \cdot B_{:,1}$	$A_{2,:} \cdot B_{:,2}$
Evaluated using	$\text{DPCode}_{\mathscr{U},\mathscr{V}}$			
Output format	$Q_{26,6}$			
Certified error	$\approx 2^{-5}$			
Maximum error	$\approx 2^{-5}$			
Average error	$\approx 2^{-5}$			

Table 4.3: Numerical properties of the code generated by Algorithm 4.4 for the $A \cdot B$.

On the first hand, Table 4.2 shows that Algorithm 4.3 produces codes optimized for the range of their entries: it is clearly superior in terms of accuracy since a dedicated code evaluates each run-time dot-product.

On the other hand, Table 4.3 shows that, as expected, Algorithm 4.4 produces far less code: it is is optimal in terms of code size since a unique dot-product code is generated, but remains a worst-case in terms of accuracy.

4.2.5 Code size and accuracy estimates

The dot-product is the basic block of classical matrix multiplication. In a size-n dot-product, regardless of the evaluation scheme used, n multiplications and $n-1$ additions are performed. Also, depending on the formats of their operands, additions frequently require alignment shifts. Thus the number of shifts is bounded by $2n$. Hence $4n-1$ is a worst case bound on the number of elementary operations (additions, multiplications, and shifts) needed to evaluate a size-n dot-product. Globally, $(4n-1)\cdot t$ is a bound on the total size of a matrix multiplication code, where $t \in \{1, \cdots, m \times p\}$ is the number of generated dot-product codes. On the previous example, this bound evaluates to 28 for Algorithm 4.3 vs. 7 for Algorithm 4.4, since $n = 2$, and $t = 4$ and 1, respectively.

As for accuracy estimates, given a matrix multiplication program composed of several DPCodes, the maximum of all error bounds can be considered as a measure of the numerical quality. However, examples can be found where this metric does not reflect the numerical accuracy of all the codes. Consider for instance Example 4.2 where both algorithms generate codes that have the same maximum error bound ($\approx 2^{-5}$), yet 3 of the 4 DPCodes generated by Algorithm 4.3 are by far more accurate than this bound. For this reason, one may also rely on the average error and consider it as a more faithful criterion to estimate the accuracy.

Fixed-point arithmetic is primarily used in embedded systems where the execution environment is usually constrained. Hence even tools that produce codes with guaranteed error bounds would be useless if the generated code size is excessively large. In the following section, we go further than Algorithms 4.3 and 4.4, and explore the possible means to achieve trade-offs between the two conflicting goals.

4.3 Dynamic closest pair algorithm for code size vs. accuracy trade-offs

In this section, we discuss how to achieve code size versus accuracy trade-offs, and the related combinatorics. We finally detail our new approach based on rows and columns merging and implemented in the *Dynamic Closest Pair algorithm*.

4.3.1 How to achieve trade-offs?

On the first hand, when developing a numerical application for embedded systems, the amount of program memory available imposes an upper bound on the code size. On the other hand, the nature of the application and its environment help in deciding on the required degree of accuracy.

	Case 1		**Case 2**		**Case 3**		**Case 4**		**Case 5**	
	Accuracy	Size	Accuracy	Size	Accuracy	Size	Accuracy	Size	Accuracy	Size
Accurate algorithm	✗	–	–	–	–	–	✓	✓	✓	✗
Compact algorithm	–	–	–	✗	✓	✓	✗	✓	✗	✓
Action	Fail		Fail		Use Algorithm 4.4		Use Algorithm 4.3		Find trade-offs	

Table 4.4: Summary of the possible cases when synthesizing matrix multiplication codes.

Once these parameters set, the programmer tries Algorithms 4.3 and 4.4 and checks the cases illustrated in Table 4.4.

Case 1 occurs when the accurate algorithm (Algorithm 4.3) is not accurate enough. Since Algorithm 4.3 produces the most accurate codes, a possible solution is to adapt the fixed-point computation word-lengths to reach the required accuracy, as in [LV09]. On the other hand, Case 2 occurs when the compact algorithm (Algorithm 4.4) does not satisfy the code size constraint. Again, since this algorithm produces the most compact code, other solutions must be considered such as adding more hardware resources. Finally, the only uncertainty that remains is described by Case 5 of Table 4.4. It happens when Algorithm 4.3 satisfies the accuracy constraint but has a large code size while Algorithm 4.4 satisfies the code size bound but is not accurate enough. This case appeals for code size versus accuracy trade-offs.

Recall that $m \times p$ dot-product calls are required at runtime. To evaluate them using less than $m \times p$ DPCodes, it is necessary to factor some DPCodes so that they would evaluate more than one run-time dot-product. This amounts to merging certain rows and/or columns of the input matrices together. Obviously, it is useless to go as far as compressing the left and right matrices into one row and column, respectively, since this corresponds to Algorithm 4.4. Our idea is illustrated by Figure 4.1 on a 4×4 matrix multiplication. In this example, the first matrix is compressed into a 2×4 matrix while the second matrix is compressed into a 4×2 matrix, as shown by the differently colored and shaped blocks. In this case, the number of required codes is reduced from 16 to 4. For example, DPCode_1 has been particularly optimized for the computation of $A_{1,:} \cdot B_{:,1}$ and $A_{1,:} \cdot B_{:,2}$, and will be used exclusively for these at run-time.

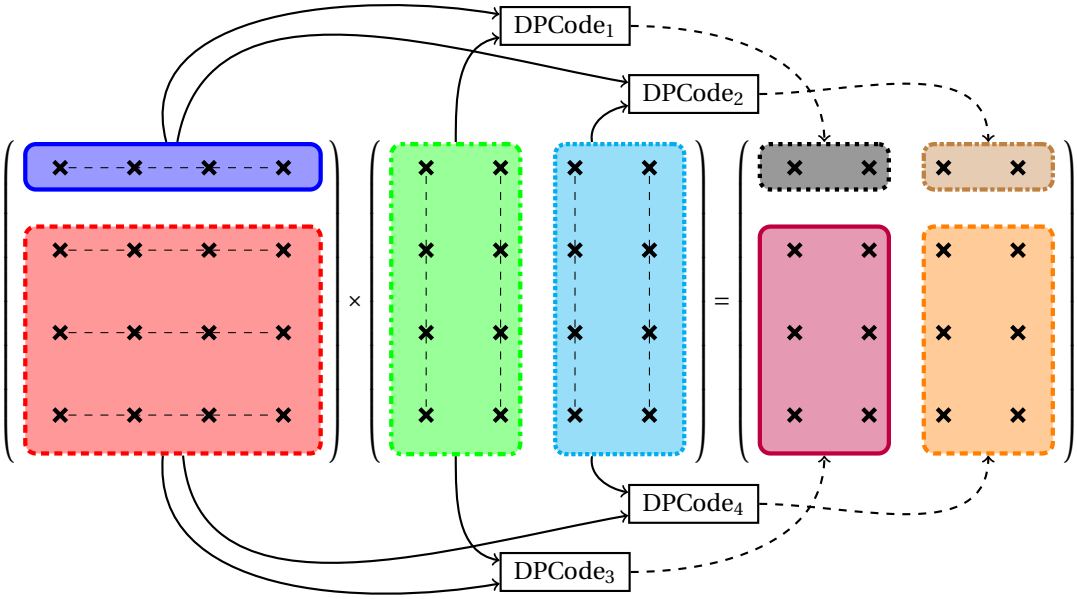


Figure 4.1: One merging strategy on a 4×4 matrix multiplication.

4.3.2 Combinatorial aspect of the merging strategy

Consider the two sets of vectors:

$$\mathscr{S}_A = \{A_{1,:}, \cdots, A_{m,:}\} \quad \text{and} \quad \mathscr{S}_B = \{B_{:,1}, \cdots, B_{:,p}\},$$

associated to the input matrices:

$$A \in \mathbb{F}\text{ix}^{m \times n} \quad \text{and} \quad B \in \mathbb{F}\text{ix}^{n \times p}.$$

In our case, the problem of finding an interesting code size versus accuracy trade-off reduces to finding partitions of the sets $\mathscr{S}_A$ and $\mathscr{S}_B$ into $k_A \leq m$ and $k_B \leq p$ subsets, respectively, such that both of the following conditions hold:

1. the code size bound σ is satisfied, that is:

$$(4n-1) \cdot k_A \cdot k_B < \sigma, \tag{4.3}$$

2. and the error bound ϵ is guaranteed, that is:

$$\epsilon_{\text{matrix}} < \epsilon, \tag{4.4}$$

where ϵ_{matrix} is either the minimal, the maximal, or the average computation error depending on the certification level required by the user.

Remark that, given the partitions of $\mathscr{S}_A$ and $\mathscr{S}_B$, the first condition is easy to check. However in order to guarantee the error condition, we must synthesize the DPCodes and deduce their error bounds using CGPE.

A benefit of formulating the refactoring strategy in terms of partitioning is the ability to give an upper bound on the number of possible dot-product mergings. Indeed, given a non-empty set $\mathscr{S}$ of k vectors, the number of different ways to partition $\mathscr{S}$ into $k' \leq k$ non-empty subsets of vectors is given by the *Stirling* number[3] of the second kind $\left\{ {k \atop k'} \right\}$, defined as follows:

$$\left\{ {k \atop k'} \right\} = \frac{1}{k'!} \sum_{j=0}^{k'} (-1)^{k'-j} \frac{k'!}{j!(k'-j)!} j^k.$$

However, k' is *a priori* unknown and can be $\in \{1, \cdots, k\}$. The total number of possible partitions of a set of k vectors is therefore given by the following sum, commonly referred to as the *Bell* number:[4]

$$B(k) = \sum_{k'=1}^{k} \left\{ {k \atop k'} \right\}.$$

Finally, in our case, the total number of partitionings is defined as follows:

$$\mathscr{P}(m,p) = B(m) \cdot B(p) - 2, \tag{4.5}$$

where $m \times p$ is the size of the resulting matrix. Notice that we exclude two partitions:

1. The partition of $\mathscr{S}_A$ and $\mathscr{S}_B$ into, respectively, m and p subsets which correspond to putting one and only one vector in each subset. This is the partitioning that leads to Algorithm 4.3.
2. The partition of $\mathscr{S}_A$ and $\mathscr{S}_B$ into one subset each. This partitioning leads to Algorithm 4.4.

Some values of the number $\mathscr{P}$ of Equation (4.5) are given in Table 4.5. Since this number is large, even for small matrix sizes, heuristics will be necessary to tackle this problem. In the following, we introduce a method based on finding closest pairs of vectors according to a certain metric. This allows to find partitions that achieve the required trade-off.

(m,p)	(5,5)	(6,6)	(10,10)	(16,16)	(25,25)	(64,64)
Number of algorithms $\mathscr{P}$	2704	41 209	$\approx 2^{34}$	$\approx 2^{66}$	$\approx 2^{124}$	$\approx 2^{433}$

Table 4.5: Some values of $\mathscr{P}$ for the multiplication of square matrices.

[3] See http://oeis.org/A008277.
[4] See http://oeis.org/A000110.

4.3.3 Dynamic Closest Pair Algorithm

A component-wise merging of two vectors $\mathscr{U}$ and $\mathscr{V}$ of fixed-point variables yields a vector whose ranges are larger than those of $\mathscr{U}$ and $\mathscr{V}$. This eventually leads to a degradation of the accuracy if the resulting vector is used to generate some DPCodes. In the extreme, this is illustrated by Algorithm 4.4 in Section 4.2.4. Therefore the underlying idea of our approach is that of putting together, in the same subset, row or column vectors that are close according to a given distance or criterion. Hence we ensure a reduction in code size while maintaining tight fixed-point formats, and thus guaranteeing a tight error bound.

Many metrics can be used to compute the distance between two vectors. Below, we cite two mathematically rigorous distances that are suitable for fixed-point arithmetic: the Hausdorff distance and the fixed-point distance. However, as our method does not use the mathematical properties of distances, any criterion that may discriminate between pairs of vectors of fixed-point variables may be used. For instance, although not a distance, the width criterion introduced below was used in our experiments.

Hausdorff distance. The range of a fixed-point variable corresponds to a rational discrete interval. It follows that the Hausdorff distance [MKC09b], widely used as a metric in interval arithmetic, can be applied to fixed-point variables. Given two fixed-point variables x and y and their ranges $\mathscr{R}\text{ange}(x) = \left[\underline{r_x}, \overline{r_x}\right]$ and $\mathscr{R}\text{ange}(y) = \left[\underline{r_y}, \overline{r_y}\right]$, this distance $d_H(x,y)$ is defined as follows:

$$d_H : \mathbb{F}\text{ix} \times \mathbb{F}\text{ix} \longrightarrow \mathbb{R}^+$$
$$d_H(x,y) = \max\left\{\left|\underline{r_x} - \underline{r_y}\right|, \left|\overline{r_x} - \overline{r_y}\right|\right\},$$

Roughly, this distance computes the maximum increase suffered by $\mathscr{R}\text{ange}(x)$ and $\mathscr{R}\text{ange}(y)$ when computing the union $x \cup y$, as illustrated on Figure 4.2.

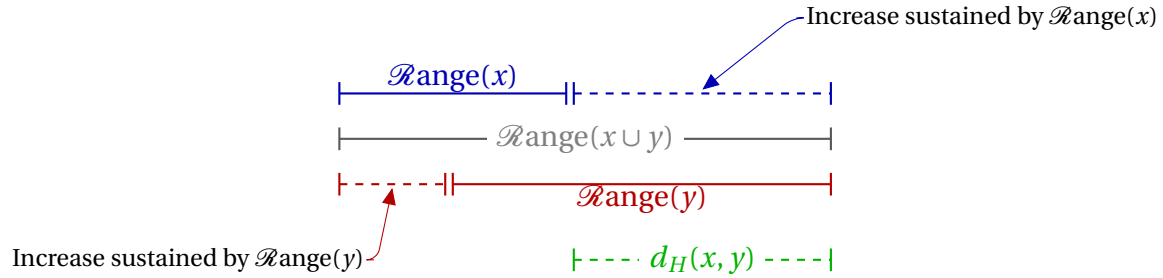


Figure 4.2: Illustration of the Hausdorff distance between two input variables $(x,y) \in \mathbb{F}\text{ix}^2$.

This distance illustrates our heuristic: by trying to merge only vectors of variables that minimize the Hausdorff distance, we make sure that this merging minimally impacts their range.

Fixed-point distance. Contrarily to the Hausdorff distance which reasons on the ranges defined by the fixed-point variables, the fixed-point distance uses only their fixed-point formats. As such, it is slightly faster to compute. Given two fixed-point variables x and y, this distance $d_F(x, y)$ is defined as follows:

$$d_F : \mathbb{F}\text{ix} \times \mathbb{F}\text{ix} \to \mathbb{N}$$
$$d_F(x, y) = |a_x - a_y|,$$

where $\mathbf{Q}_{a_x,b_x}$ and $\mathbf{Q}_{a_y,b_y}$ are the fixed-point formats of x and y, respectively. Analogously to Hausdorff distance, this distance computes the increase in the integer part suffered by x and y when computing their union $x \cup y$.

Width criterion. Let x, y, and z be three fixed-point variables such that $z = x \cup y$ where z is computed according to Algorithm 4.2. Our third metric consists in considering the width of $\mathscr{R}\text{ange}(z) = \left[\underline{r_z}, \overline{r_z}\right]$ as illustrated on Figure 4.3. Formally, it is defined as follows:

$$d_W : \mathbb{F}\text{ix} \times \mathbb{F}\text{ix} \to \mathbb{R}^+$$
$$d_W(x, y) = \left(\overline{r_z} - \underline{r_z}\right).$$

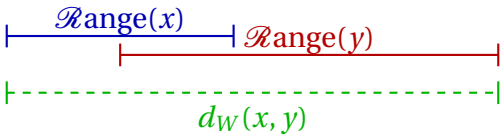


Figure 4.3: Illustration of the width criterion between two input variables $(x, y) \in \mathbb{F}\text{ix}$.

Notice that although the metrics are introduced as functions of two fixed-point intervals, we generalized them to vectors of fixed-point variables by considering either the component-wise max or average value.

Given one of the above metrics and a set $\mathscr{S}$ of vectors, it is straightforward to implement a `findClosestPair` routine that returns the closest pair of vectors in $\mathscr{S}$. There are several ways to implement such a routine. A $\mathscr{O}(n^2)$ naive approach such as the one shown in Algorithm 4.5 would compare all the possible pairs of vectors. But, depending on the distance used, optimized implementations may rely on the well established *fast closest pair of points* algorithms [SH75], [CLRS09, §33].

Algorithm 4.5 `findClosestPair` algorithm.

Input:

A set of vectors $\mathscr{S} = \{v_1, \ldots, v_n\}$ of cardinal n
A metric d

Output:

The closest pair of vectors (v_i, v_j) *s.t.* $v_i, v_j \in \mathscr{S}^2$ and $i \neq j$
The value of $d(v_i, v_j)$

Algorithm:

```
1: closest_pair ← (v_1, v_2)
2: d_min ← d(v_1, v_2)
3: for 1 ≤ i ≤ n do
4:     for i < j ≤ n do
5:         d_temp ← d(v_i, v_j)
6:         if d_temp < d_min then
7:             closest_pair ← (v_i, v_j)
8:             d_min ← d_temp
9:         end if
10:    end for
11: end for
```

Nevertheless, our contribution lies mainly in the design of Algorithm 4.6 which is based on a dynamic search of a code that satisfies both an accuracy bound $\mathscr{C}_1$ and a code size bound $\mathscr{C}_2$.

Here, following Case 5 of Table 4.4, we assume that Algorithm 4.3 satisfies the accuracy bound $\mathscr{C}_1$, otherwise, no smaller code satisfying $\mathscr{C}_1$ could be found. Therefore, Algorithm 4.6 starts with two sets of m and p vectors, respectively, corresponding to the rows of A and the columns of B. As long as the bound $\mathscr{C}_1$ is satisfied, each step of the while loop merges together the closest pair of rows or columns, and thus decrements the total number of vectors by 1. At the end of Algorithm 4.6, if the size of the generated code satisfies the code size bound $\mathscr{C}_2$, a trade-off solution has been found. Otherwise, Algorithm 4.6 failed to find a code that satisfies both bounds $\mathscr{C}_1$ and $\mathscr{C}_2$. This algorithm was implemented in the FPLA tool presented in Chapter 6. Section 4.4 studies its efficiency on a variety of fixed-point benchmarks.

4.4 Numerical experiments

In this section, we illustrate the efficiency of our heuristics, and the behaviour of Algorithm 4.6 as well as the impact of the distance and the matrix size through a set of numerical results.

Algorithm 4.6 Dynamic Closest Pair algorithm.

Input:

Two matrices $A \in \mathbb{F}\text{ix}^{m\times n}$ and $B \in \mathbb{F}\text{ix}^{n\times p}$
An accuracy bound $\mathscr{C}_1$ (ex. average error bound is $< \epsilon$)
A code size bound $\mathscr{C}_2$
A metric d

Output:

Code to compute $A \cdot B$ s.t. $\mathscr{C}_1$ and $\mathscr{C}_2$ are satisfied,
or no code otherwise

Algorithm:

1: $\mathscr{S}_A \leftarrow \{A_{1,:}, \ldots, A_{m,:}\}$
2: $\mathscr{S}_B \leftarrow \{B_{:,1}, \ldots, B_{:,p}\}$
3: **while** $\mathscr{C}_1$ is satisfied **do**
4: $(u_A, v_A), d_A \leftarrow findClosestPair(\mathscr{S}_A, d)$
5: $(u_B, v_B), d_B \leftarrow findClosestPair(\mathscr{S}_B, d)$
6: **if** $d_A \leq d_B$ **then**
7: $remove(u_A, v_A, \mathscr{S}_A)$
8: $insert(u_A \cup v_A, \mathscr{S}_A)$
9: **else**
10: $remove(u_B, v_B, \mathscr{S}_B)$
11: $insert(u_B \cup v_B, \mathscr{S}_B)$
12: **end if**
13: **for** $(A_i, B_j) \in \mathscr{S}_A \times \mathscr{S}_B$ **do**
14: DPSynthesis(A_i, B_j)
15: **end for**
16: **end while**
17: /* Revert the last merging step. */
18: /* Check the bound $\mathscr{C}_2$. */

4.4.1 Experimental environment

Experiments have been carried out with 32 bit fixed-point variables and using 3 structured and 1 unstructured benchmark. For structured benchmarks, the large coefficients distribution throughout the matrices follows different patterns. This is achieved through weight matrices, as shown in Table 4.6 where $W_{i,j}$ corresponds to the element of row i and column j of the considered weight matrix.

Notice, that the dynamic range defined as $max(W_{i,j})/min(W_{i,j})$ is the same for all benchmarks, and is equal to $2^{\lfloor n/2 \rfloor}$. The reason we did not directly use these matrices in our experiments is that the first three patterns correspond to structured matrices in the usual sense and that better algorithms to multiply structured matrices exist [Mou11]. To obtain random matrices where the large coefficients are still distributed according to

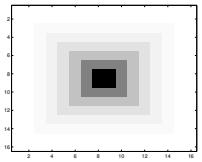
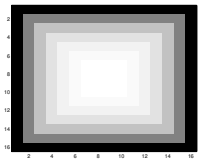
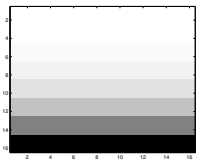
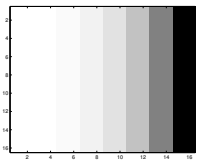
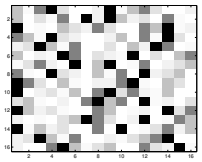
Name	$W_{i,j}$	Heat map
Center	$2^{\max(i,j,n-1-i,n-1-j)-\lfloor n/2 \rfloor}$	
Edges	$2^{\min(i,j,n-1-i,n-1-j)}$	
Rows / Columns	$2^{\lfloor i/2 \rfloor}$ $2^{\lfloor j/2 \rfloor}$	 
Random	$2^{\text{rand}(0,\lfloor n/2 \rfloor-1)}$	

Table 4.6: Weight matrices considered for the benchmarks.

the pattern described by the weight matrices, we computed the Hadamard product of Table 4.6 matrices with normally distributed matrices generated using Matlab®'s `randn` function. Figure 4.4 shows a sample matrix obtained by proceeding this way. Finally, notice that the matrices obtained this way have floating-point coefficients. In order to get fixed-point matrices, we first converted them to interval matrices by considering the radius 1 intervals centered at each coefficient. Next, the floating-point intervals are converted into fixed-point variables by considering the smallest fixed-point format that holds all the interval's values.

4.4.2 Efficiency of the distance based heuristic

As a first experiment, let us consider 2 of the benchmarks: *Center* and *Random* square matrices of size 6. For each, we build two matrices A and B, and observe the efficiency of our *closest pair* heuristic based approach by comparing the result of Algorithm 4.6 to all the possible codes. To do so, we compute all the possible row and column mergings:

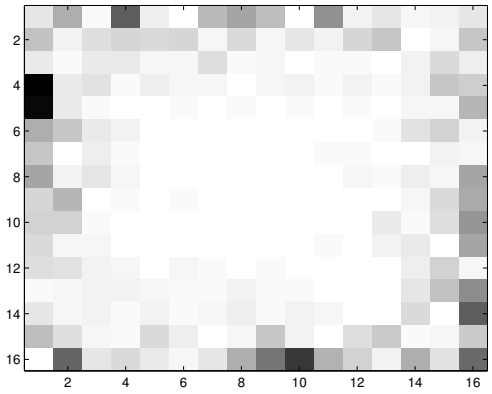
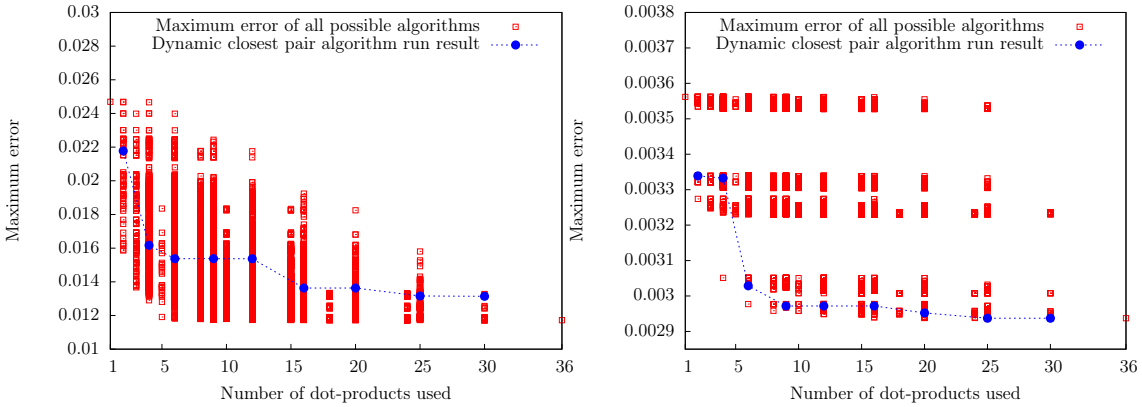


Figure 4.4: The heat map of a size-16 pondered matrix from the Edges benchmark.

Equation (4.5) assures that there are 41 209 such mergings for size-6 matrices. For each of these, we synthesized the codes for computing $A \cdot B$, and determined the maximum and average errors. This exhaustive experiment took approximately 2h15min per benchmark on an Intel Core i7-870 desktop machine running at 2.93 GHz. Figures 4.5 and 4.6 show the maximum and average errors of the produced codes according to the number of DPCodes involved. Next, we ran our tool with Hausdorff's distance and with the accuracy bound $\mathscr{C}_1$ set to a large value so as to see the behavior of Algorithm 4.6 on all the intermediate steps. This took less than 10 seconds for each benchmark and corresponds to the dark blue dots in Figures 4.5 and 4.6. Notice on both sides the accurate algorithm which produces 36 DPCodes and the compact algorithm which produces only one DPCode.

For the structured *Center* benchmark, Algorithm 4.6 behaves as expected. For both maximum and average error, it is able to drastically reduce the number of DPCodes without impacting the accuracy. Indeed, as shown in Figure 4.5b, for a maximum error of $\approx 3 \cdot 10^{-3}$ which is close to the most accurate algorithm, our algorithm is able to reduce the number of DPCodes from 36 to 9. For average error in Figure 4.6, Algorithm 4.6 even finds a merging that produces 9 DPCodes and has almost the same accuracy as Algorithm 4.3 which is the most accurate.

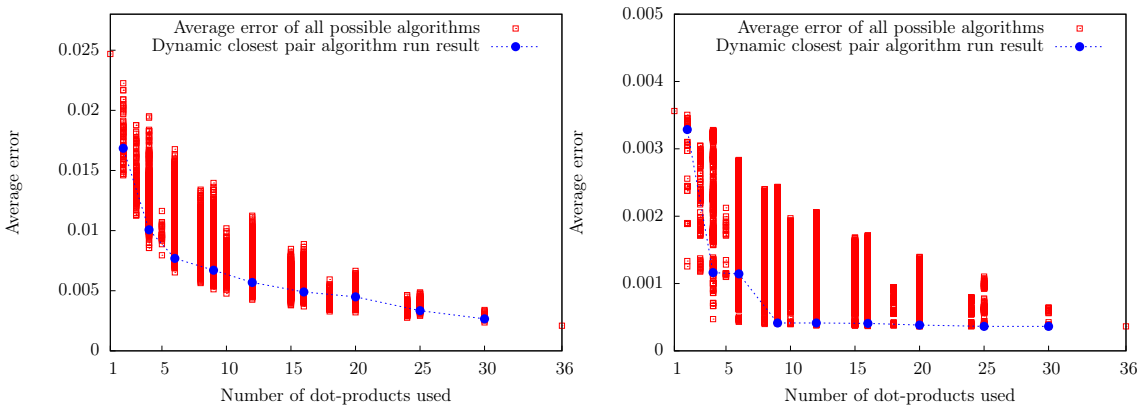
For the *Random* benchmark, the behavior of Algorithm 4.6 is less predictable. Indeed, in this benchmark, the elements of high dynamic range are spread over the matrix and do not follow a particular pattern. In this case, it is less obvious for Algorithm 4.6 to find the best codes in terms of accuracy. Indeed, Algorithm 4.6 follows a greedy approach in making the local decision to merge two vectors. And, once it goes in a wrong branch of the result space, this may lead to a code having an average or maximum error slightly larger than the best case. This can be observed on Figure 4.6b: the first 6 steps produce code with very tight average error, but step 7 results in a code with an average error of $\approx 10^{-3}$ while the best code has an error of $\approx 5 \cdot 10^{-4}$. As a consequence, the following of the algorithm gives a



(a) Random matrices. **(b)** Center matrices.

Figure 4.5: Maximum error according to the number of DPCodes.

code with an error of $\approx 3 \cdot 10^{-3}$ instead of $\approx 10^{-3}$ for the best case. The same phenomenon happens at step 1 Figure 4.5a.



(a) Random matrices.. **(b)** Center matrices.

Figure 4.6: Average error versus the number of DPCodes.

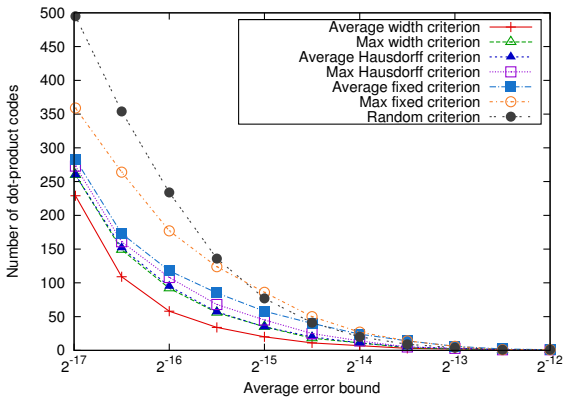
Despite this, these experiments show the interest of our approach. Indeed we may observe that, at each step, the heuristic merges together 2 rows of A or 2 columns of B to produce a code having in most cases an average error close to the best case. This is particularly the case on Figures 4.5b and 4.6b for *Center* benchmarks. Moreover, Algorithm 4.6 converges toward code having good numerical quality much faster than the exhaustive approach.

4.4.3 Impact of the metric on the trade-off strategy

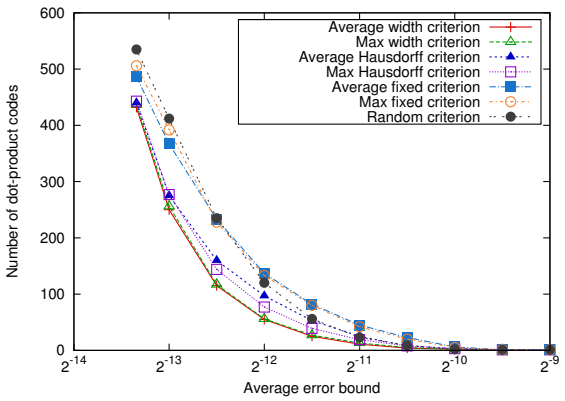
In this second experiment, we consider 25×25 matrices. For each benchmark introduced above, 50 different matrix products are generated, and the results exhibited are computed as the average on these 50 products. To compare the different distances, we consider the *average accuracy* bound: for each metric, we varied this bound and used Algorithm 4.6 to obtain the most compact codes that satisfy it. Here we ignored the code size bound $\mathscr{C}_2$ by setting it to a large enough value. Also, in order to show the efficiency of the *closest pair* strategy, we compare the codes generated using Algorithm 4.6 with those of an algorithm where the merging of rows and columns is carried out randomly. Figure 4.7 shows the results of running FPLA.

First notice that, as expected, large accuracy bounds yield the most compact codes. For instance, for all the benchmarks, no matter the distance used, if the target average accuracy

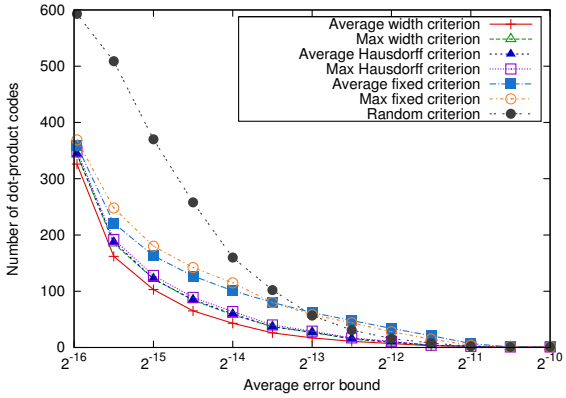
(a) Center matrices.



(b) Edges matrices.



(c) Rows/Columns matrices.



(d) Random matrices.

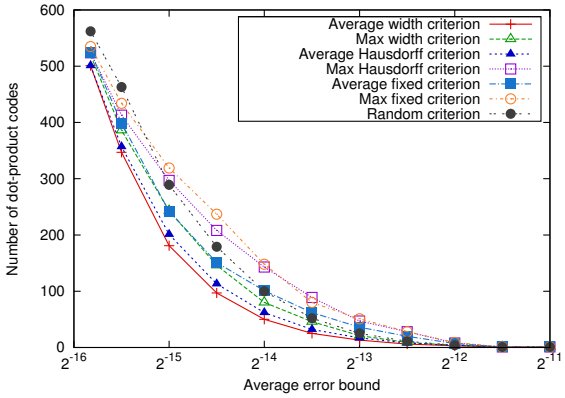


Figure 4.7: Number of dot-product codes generated by each algorithm for increasing average error bounds.

is $> 2^{-9.5}$, one DPCode suffices to evaluate the matrix multiplication. This indeed amounts to using Algorithm 4.4. Also as expected and except for few values, when used with one of the distances above, our algorithm produces less DPCodes than with the random function as a distance. Using the average width criterion, our algorithm is by far better than the random algorithm and yields on the *Center* and *Rows/Columns* benchmarks a significant reduction in code size, as shown on Figures 4.7a and 4.7c. For example, for the *Center* benchmark, when the average error bound is set to 2^{-16}, our algorithm satisfies it with only 58 DPCodes, while the random algorithm needs 234 DPCodes. This yields a code size reduction of up to 75%. Notice also that globally, the *Center* benchmark is the most accurate. This is due to the fact that few *Rows/Columns* have a high dynamic range. On Figures 4.7b and 4.7d, in the *Edges* as well as *Random* benchmarks, all of the rows and columns have a high dynamic range which explains in part why these benchmarks are less accurate than the *Center* benchmark. These experiments also suggest that average based distances yield tighter code than maximum based ones.

4.4.4 Impact of the matrix size

In this third experiment, we study the influence of the matrix sizes on the methodology presented above. To do so, we consider square matrices of the *Center* benchmark with sizes 8, 16, 32, and 64, where each element has been scaled so as these matrices have the same dynamic range. We run Algorithm 4.6 using the *average width* criterion as a metric with different *average error* bounds from 2^{-21} to 2^{-14}. Here the bound $\mathscr{C}_2$ has also been ignored. For each of these benchmarks, we determine the number of DPCodes used for each average error, as shown in Table 4.7 (where "–" means "no result has been found").

	Maximum error							
Matrix size	2^{-21}	2^{-20}	2^{-19}	2^{-18}	2^{-17}	2^{-16}	2^{-15}	2^{-14}
8	24	6	1	1	1	1	1	1
16	–	117	40	16	3	1	1	1
32	–	–	552	147	14	2	1	1
64	–	–	–	2303	931	225	48	1

Table 4.7: Number of DPCodes for various matrix sizes and error bounds.

This shows clearly that our method is extensible to large matrices, since it allows to reduce the size of the problem to be implemented, while maintaining a good numerical quality. For example, the 64×64 accurate matrix multiplication would require 4096 DPCodes. Using our heuristic, we produce a code with 2303 DPCodes having an average error bounded by 2^{-18}, that is, a reduction of about 45%. Remark that no code with

average error bound of 2^{-19} is found, which means that even the accurate algorithm (Algorithm 4.3) has an error no tighter than 2^{-19}: we can conclude that our heuristic converges towards code having an error close to the best case, but with half less DPCodes. Finally, if the user's accuracy expectations are low, i.e., if an error bound of 2^{-14} is acceptable, then only one DPCode is sufficient to implement matrix multiplication for all the sizes.

4.5 Conclusion

This chapter tackled the problem of generating fixed-point code for matrix multiplication. It started by exposing two straightforward algorithms to solve this problem. When constraints on accuracy and code size are added to the problem, these two approaches represent both ends of the spectrum. Indeed, one needs to find implementations of matrix multiplication that present a good trade-off between the most accurate and the most compact algorithms. In the second part of the chapter, we presented a new strategy based on the merging of row or column vectors of the input matrices. This strategy, implemented in the *dynamic closest pair algorithm*, allows to reduce the size of the generated code while guaranteeing that the error bound is satisfied. Finally, the efficiency of this approach has been illustrated on a set of benchmarks.

CHAPTER 5

FIXED-POINT SYNTHESIS FOR MATRIX DECOMPOSITION AND INVERSION

The previous chapter presented the higher level problem of matrix multiplication. This chapter tackles code generation for matrix inversion. While the current state of our tools do not support the complete flow for matrix inversion, most of the basic blocks are provided for. Contrarily to matrix multiplication, matrix inversion involves division and square root operations. Since the output format of division must be set by a user defined method, this chapter investigates different approaches to do so and gives experimental data for each method.

5.1 Introduction

MATRIX inversion is known to be numerically unstable. As a consequence, numerical analysts advise against using it for a large set of problems. Yet, as stated by Higham [Hig02, § 14], cases exist where the inverse conveys useful information. For instance, in wireless communications, matrix inversion is used in equalization algorithms [ZQZ05] as well as detection estimation algorithms in space-time coding [CDH03].

In the context of DEFIS, our industrial partners look forward to having an implementation of matrix inversion for a Space Time Adaptive Processing algorithm (STAP) that is

embedded in radar applications. Such applications require inverting a positive-definite covariance matrix [Gue03].

This chapter does not discuss these applications and leaves it to the designer to decide whether computing the inverse is indeed justified. Rather, its goal is to introduce a flow for fixed-point code synthesis for matrix inversion and to conduct experiments to show its behavior and numerical quality.

Only few research articles on fixed-point matrix inversion are available. One such work is the Gusto tool suggested by Irturk *et al.* and presented in a series of articles [IBMK10], [IMK09], and [IBMK08]. Its authors do provide benchmarks for the accuracy of their matrix inversion algorithms. These algorithms are essentially based on QR decomposition. However, the methodology and arithmetic model for generating the fixed-point implementation are not explicit in the articles and the treated matrices do not exceed size 8. Besides, the evaluation of the rounding error is based on a *a posteriori* simulation process. Therefore, no certified error bounds are provided. Another work that heavily relies on simulations is Frantz *et al.* [NNF07]. The authors use the simulation approach introduced by Sung *et al.* [SK95] to study the mapping of many linear algebra routines to Texas Instruments' C64x+ fixed-point DSP. These routines include Cholesky, LU, QR, SVD, and Gauss-Jordan decompositions. They treat matrices of sizes up to 30, but uniquely with simulation based methods and without providing certified error bounds. Indeed, in working with a rigorous error model, our work is analogous to [LGC+06] and [FRC03], where affine arithmetic is used to assert properties on the error bounds. However, both approaches target discrete transformation algorithms.

This chapter is organized as follows: Section 5.2 recalls the matrix inversion methods. Then, Section 5.3 details the work-flow we propose to map matrix inversion into the fixed-point arithmetic. Finally, experimental results are exhibited in Section 5.4.

5.2 A methodology for matrix inversion

A survey of floating-point matrix inversion and linear systems solving shows that there are many algorithms in use: Cramer's rule, LU and QR decompositions, ... [GVL96]. A common pattern to the efficient algorithms is the decomposition of the input matrix into a product of easy to invert matrices (triangular or orthogonal matrices). LU decomposition, for instance, proceeds by Gaussian elimination to decompose an input matrix A into two triangular matrices L and U such that $A = LU$. Inverting triangular matrices being straightforward, solving the associated linear system is equally simple and so is obtaining the inverse A^{-1} by the formula $A^{-1} = U^{-1}L^{-1}$. Almost the same chain of reasoning is applicable to QR decomposition. The common pattern of these algorithm is captured by the three steps of Figure 5.1, that is:

1. the decomposition of the input matrix A into easy to invert matrices (diagonal, triangular, or orthogonal matrices),
2. the computation of the inverse of the matrices obtained in step 1 , and
3. the combination of the inverses computed in step 2 to obtain A^{-1}.

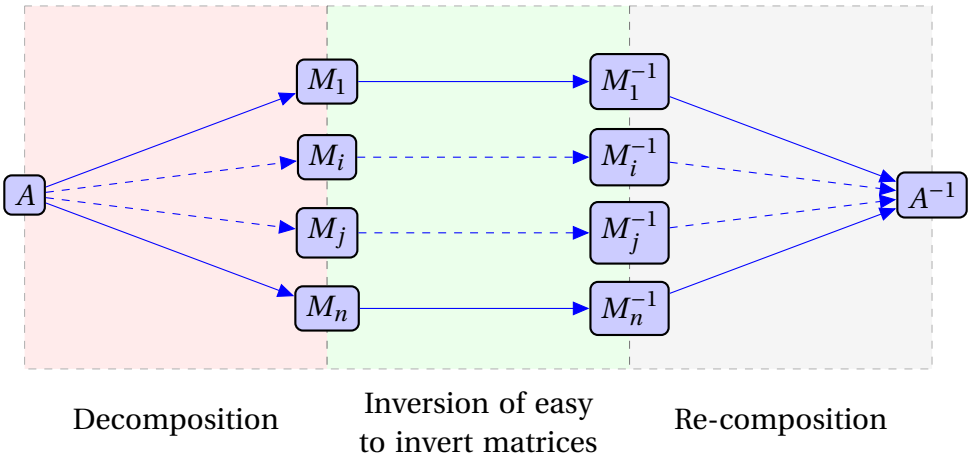


Figure 5.1: A typical flow for a decomposition based matrix inversion.

5.2.1 Matrix inversion using Cholesky decomposition

Motivated mainly by the application of our industrial partner, and the prevalence of symmetric positive-definite matrices in signal processing, we chose to tackle matrix inversion through Cholesky decomposition. Given a symmetric positive-definite matrix A, the method follows the three steps of Figure 5.1, that is:

1. matrix decomposition: computing a lower triangular matrix L such as $A = LL^T$,
2. triangular matrix inversion: computing L^{-1}, and
3. inverse re-composition through multiplication: $A^{-1} = L^{-T}L^{-1}$.

In floating-point arithmetic, the computationally intensive step of this flow is the decomposition part [GVL96]. This is the case as well for the LU and QR based methods. The decomposition step is also the missing link in fixed-point arithmetic. Indeed, we presented in Chapter 4 a methodology for fixed-point code synthesis for matrix multiplication that we first described in [MNR14a], but, to our knowledge, no published works suggest a rigorous methodology of code synthesis for matrix decomposition or triangular matrix inversion. In the following, we start by recalling the formulas to compute these two steps.

5.2.2 The triangular matrix inversion step

Using the so called backward and forward substitution techniques, inverting a triangular matrix is a straightforward process in floating-point arithmetic. Indeed, for a lower triangular matrix M, its inverse N is given by the following equation:

$$n_{i,j} = \begin{cases} 0 & \text{if } i < j \\ \dfrac{1}{m_{i,i}} & \text{if } i = j \\ \dfrac{-c_{i,j}}{m_{i,i}} & \text{if } i > j \end{cases} \qquad \text{where } c_{i,j} = \sum_{k=j}^{i-1} m_{i,k} \cdot n_{k,j}. \tag{5.1}$$

While in floating-point arithmetic, implementing these equations requires only three nested loops, it is more challenging in fixed-point arithmetic. Indeed, the coefficient $n_{i,j}$ depends on other coefficients of the inverse N, namely all the $n_{k,j}$ with $k \in \{j, \ldots, i-1\}$. This implies that the synthesis tool, when generating code that computes $n_{i,j}$ must know the ranges and formats of all the $n_{k,j}$ with $k \in \{j, \ldots, i-1\}$. It is clear that such a tool must follow a determined order in synthesizing code and that it must keep track of the formats and ranges of the computed coefficients so as to reuse them. Besides, similarly to the work in [MNR14a] presented in Chapter 4, this process may involve multiple trade-offs between code size and accuracy.

5.2.3 The Cholesky decomposition step

Finally, the remaining step in our flow is Cholesky decomposition. If A is a symmetric positive-definite matrix, its Cholesky decomposition is defined. Since this method exploits the structure of the matrix, it is more efficient than Gaussian elimination and is also known for its numerical stability [Hig02, § 10].

The aim of Cholesky's method is to find a lower triangular matrix L such that:

$$A = L \cdot L^T. \tag{5.2}$$

And by equating the coefficients in (5.2) and using the symmetry of A, the following formula for the general term of L is deduced:

$$\ell_{i,j} = \begin{cases} 0 & \text{if } i < j \\ \sqrt{c_{i,i}} & \text{if } i = j \\ \dfrac{c_{i,j}}{\ell_{j,j}} & \text{if } i \neq j \end{cases} \qquad \text{where } c_{i,j} = a_{i,j} - \sum_{k=0}^{j-1} \ell_{i,k} \cdot \ell_{j,k} \tag{5.3}$$

Again, before generating code for $\ell_{i,j}$, one must generate code for its dependencies. These include all the $\ell_{i,k}$ and $\ell_{j,k}$ with $k \in \{0,\ldots,j-1\}$ as well as $\ell_{j,j}$ if the coefficient is not diagonal. Also, synthesizing code that computes $\ell_{i,j}$ from $c_{i,j}$ involves the square root and division operators. Therefore, as explained in Section 2.6.2 of Chapter 2, the intervention of the user is needed to provide the fixed-point format of the output of division. By doing so, the user sets the appropriate trade-off between the sharpness of the accuracy bounds and the risk of run-time overflows.

We conclude this section with the following two remarks on this flow:

Using Cholesky based matrix inversion for general matrices. Remark that restraining to symmetric positive-definite matrices is not an overkill. Indeed, Cholesky decomposition can be used to invert any non-symmetric positive-definite matrix A by decomposing the following matrix: $M = AA^T$ which is guaranteed to be symmetric to obtain $M = LL^T$. From this decomposition, A^{-1} can be recovered using the formula:

$$A^{-1} = A^T L^{-T} L^{-1}. \tag{5.4}$$

Pre-processing and range reduction of the input matrices. In fixed-point arithmetic, we can frequently reduce the ranges of the inputs to a range included in $[-1,1]$. Indeed, for Cholesky decomposition, if a matrix A does not satisfy this condition, instead of decomposing A, we can decompose

$$B = 2^{-k} \cdot A \text{ with } k \in \mathbb{Z}, \tag{5.5}$$

to obtain $B = RR^T$, where B is a matrix with coefficients in $[-1,1]$. It follows that $A = 2^k \cdot RR^T = LL^T$ where

$$L = 2^{k/2} \cdot R.$$

Notice that k must be chosen even. For triangular matrix inversion, we can still compute A^{-1} as follows:

$$A^{-1} = 2^{-k} \cdot B^{-1}, \text{ where } B \text{ is as in Equation (5.5).}$$

In fixed-point arithmetic, these scalings are just a matter of manipulating the fixed-point format of the coefficients. These scaling justify our choice of Section 5.4 to target benchmarks with input coefficients that belong $[-1,1]$.

5.3 Code synthesis for triangular matrix inversion and Cholesky decomposition

The basic blocks introduced in the previous section were implemented in the FPLA tool which is presented in Chapter 6. This tool was developed with the aim of generating fixed-point code for the most frequently used linear algebra routines. It handles the aspects

peculiar to each class of input problems and relies on the CGPE library for the low-level code synthesis details. For triangular matrix inversion and Cholesky decomposition, FPLA internally keeps track of two matrices of fixed-point variables:

1. the input matrix, and
2. the resulting matrix.

The input matrix is not modified throughout the run. However, the resulting matrix is updated with the range, format, and error bound of each code returned by the CGPE. Therefore, FPLA must correctly handle the ordering of calls to CGPE in such a way that each coefficient's code is generated only after all the information on which it depends has been collected.

5.3.1 Order of code synthesis in FPLA

In triangular matrix inversion and according to Equation (5.1), the diagonal elements do not depend on any generated code. Therefore they may be computed in any order. The non-diagonal coefficients depend only on the coefficients that precede them on the same column. Therefore, FPLA can follow a row major, column major, or even a diagonal major approach. The latter consists in generating the elements on the diagonal, followed by those on the first sub-diagonal, and so on. The last code generated in this fashion would be that of the bottom left coefficient $\ell_{n-1,0}$. This is illustrated by Figure 5.2.

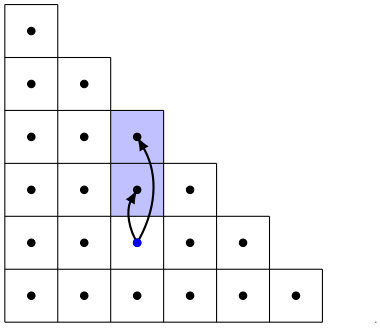


Figure 5.2: Dependencies of the coefficient $\ell_{5,3}$ (in blue) in the triangular matrix inversion of a 6×6 matrix.

For Cholesky decomposition, a diagonal element $\ell_{i,i}$ depends on the generated coefficients that precede it on row i. A non-diagonal element $\ell_{i,j}$ depends on the first j elements of row i as well as the first $j+1$ elements of row j. FPLA may satisfy these dependencies by following either a row major or column major synthesis strategy but not a diagonal major strategy. These dependencies are illustrated by Figure 5.3.

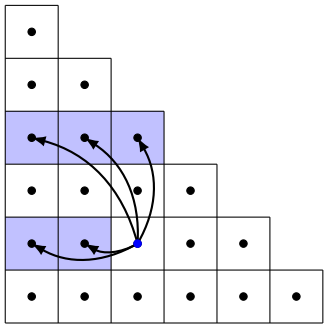


Figure 5.3: Dependencies of the coefficient $\ell_{5,3}$ (in blue) in the Cholesky's decomposition (right) of a 6×6 matrix.

Once all the coefficient codes of the resulting matrix are generated, FPLA generates the global `C` file where each code assigns its result to the correct matrix index. Listing 5.1 shows this global code for a size-3 triangular matrix inversion. The coefficients of the upper triangular part are explicitly set to zero. Then `compute_i_i` computes the coefficient $n_{i,i}$ as $n_{i,i} = 1/d$ while `compute_i_j` computes the coefficient $n_{i,j}$ as

$$n_{i,j} = (a_0 \cdot b_0 + \cdots + a_{i-j} \cdot b_{i-j})/d$$

where a_0 =`A[i][j]`, b_0 =`N[j][j]`, a_{i-j} =`A[i][i-1]`, b_{i-j} =`N[i-1][j]`, and d =`A[i][i]`. The six `compute_x_y` functions of this example are each generated by CGPE in a specific `C` file. Listing 5.2 shows the code of `compute_2_0` of Listing 5.1.

▷ **Listing 5.1:** FPLA output code for a 3×3 triangular matrix inversion.

```
1  // A: input matrix  --  N: inverse matrix of A
2  N[0][0] =  compute_0_0( A[0][0] );
3  N[0][1] =  0;
4  N[0][2] =  0;
5
6  N[1][0] = -compute_1_0( A[1][0], N[0][0], A[1][1] );
7  N[1][1] =  compute_1_1( A[1][1] );
8  N[1][2] =  0;
9
10 N[2][0] = -compute_2_0( A[2][0], A[2][1],
11                         N[0][0], N[1][0], A[2][2] );
12 N[2][1] = -compute_2_1( A[2][1], N[1][1], A[2][2] );
13 N[2][2] =  compute_2_2( A[2][2] );
```

▷ **Listing 5.2:** C code of the `compute_2_0` function.

```
1  int32_t compute_2_0( int32_t a0      /* Q1.31 in [-1,1]*/,
2                       int32_t a1      /* Q2.30 in [-2,2]*/,
3                       int32_t b0      /* Q1.31 in [-1,1]*/,
4                       int32_t b1      /* Q2.30 in [-2,2]*/,
5                       int32_t d       /* Q1.31 in [0.88,0.99]*/){
6    int32_t r0  = mul(a0, b0);         // Q2.30 in [-1,1]
7    int32_t r1  = r0 >> 2;             // Q4.28 in [-1,1]
8    int32_t r2  = mul(a1, b1);         // Q4.28 in [-4,4]
9    int32_t r3  = r1 + r2;             // Q4.28 in [-5,5]
10   int32_t r4  = div32hs(r3, d, 31); // Q4.28 in [-8,8]
11   return r4;
12 }
```

At run-time, the function `compute_2_0` is used to compute the value of the coefficient `N[2][0]` as follows:

5.3.2 How to use correctly fixed-point division?

Division, introduced in Chapter 2, is the trickiest among the fixed-point arithmetic operators. If we seek sharp error bounds, we must be careful when deciding of the fixed-point output format of each division. The integer part of this format can be set using multiple ways:

1. set to a constant, or
2. using a function that takes as input the formats of the dividend and divisor.

For instance, if we want division results to have an integer part two bits larger than the integer part of its left operand denoted i_1, we will use the function $f(i_1, i_2) = i_1 + 2$ to compute i.

At first sight, the first solution seems to be either too restrictive or too unsafe. Indeed with long chains of computations, the format of the intermediate operands tend to grow and choosing a small enough output integer part is a good idea to bring the results of divisions to a manageable range. Conversely choosing a small output integer part increases the chances that overflow occurs at run-time. In Section 5.4, we illustrate the interest of using the first solution on some cases, and we show experimental evidence of the problems caused by the different methods of deciding this output format.

5.4 Experimental results

In this section, we first explain how we generate our benchmarks. Then we investigate the impact of the output format of division and study the speed of the generation and the sharpness of the error bounds of our generated codes. Finally we show the impact of the matrix condition number on the accuracy of the generated code.

5.4.1 Generation of fixed-point test matrices

The input matrices for which FPLA generates code are made of fixed-point variables. After the synthesis process, in order to test the resulting code on a set of benchmarks, we need to generate matrices of fixed-point numbers that belong to these fixed-point variables. For Cholesky decomposition, the matrices of fixed-point numbers should be symmetric positive-definite. To obtain such matrices, we followed the 5 stages process below:

1. Generate a lower triangular random matrix M whose fixed-point coefficients belong to the input fixed-point variables. Determine A by filling the upper side of the matrix with M^T. Formally, this is equivalent to computing $A = M + (M^T - \text{diag}(M))$. At this point, A is a symmetric matrix that belongs to the input variable matrix.
2. Compute the smallest eigenvalue of A, denoted by $\lambda_{\min}$.
3. If $\lambda_{\min} > 0$, then the matrix A is positive-definite, and we are done.
4. Otherwise, compute $A' = A - (\lambda_{\min} + \delta)I$, for a small δ.
5. A' is guaranteed to be symmetric and positive-definite. This step, checks if all the coefficients of A' belong to their respective fixed-point variables. If it is the case, we are done, otherwise, either try to divide A' by a factor and retest Step 5, or restart from Step 1.

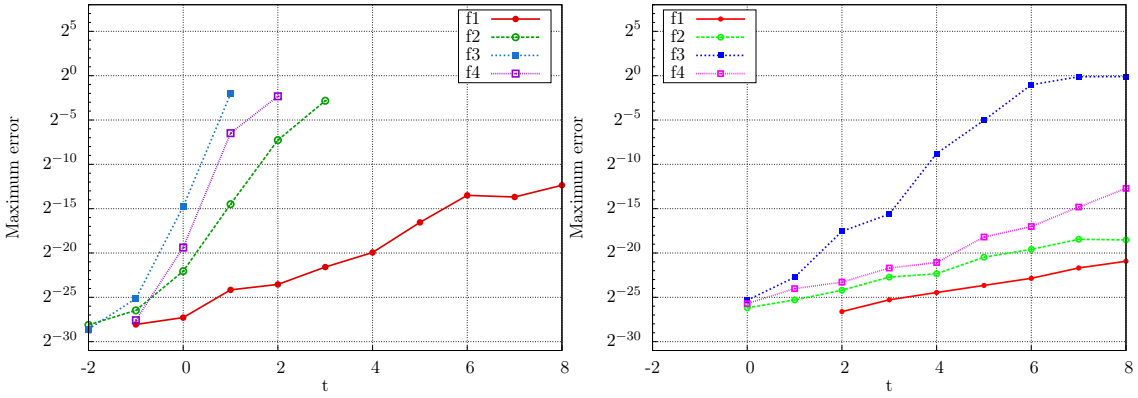
Generating triangular matrices is straightforward. For each coefficient, one must generate randomly a fixed-point number that belongs to the input fixed-point variable.

5.4.2 Impact of the output format of division on accuracy

As mentioned in Section 5.3.2, the output format of division must be explicitly set and has a great impact on the properties of the generated code. In this experiment, we use 4 different functions to set the integer part size of the result of a division. In each case, the output fraction part is determined so as each result fits on 32 bits. The functions used are the following:

1. $f_1(i_1, i_2) = t$,
2. $f_2(i_1, i_2) = \min(i_1, i_2) + t$,
3. $f_3(i_1, i_2) = \max(i_1, i_2) + t$,
4. $f_4(i_1, i_2) = \lfloor (i_1 + i_2)/2 \rfloor + t$,

where $t \in \mathbb{Z}$ is a user defined parameter, and i_1 and i_2 are the integer parts of the dividend and divisor, respectively. The function f_1 consists in fixing all the division results to the same fixed-point format. The experiment consists in computing the Cholesky decomposition and the triangular inversion of matrices of size 5 and 10, respectively. Using FPLA, we synthesize codes for each problem and each function in $\{f_1, f_2, f_3, f_4\}$, for t ranging from -2 to 8. Then for each synthesized code, 10000 example instances are generated and solved both in fixed and floating-point arithmetics. Each example input is a matrix having 32-bits coefficients in the range between -1 and 1. Then the error considered is obtained by comparing the results to floating-point computations and by considering the maximum errors among the 10000 samples. The results, for both basic blocks, are shown in Figures 5.4 and 5.5 for sizes 5 and 10, respectively, where the absence of the curve in some figures means that all of the examples overflew.



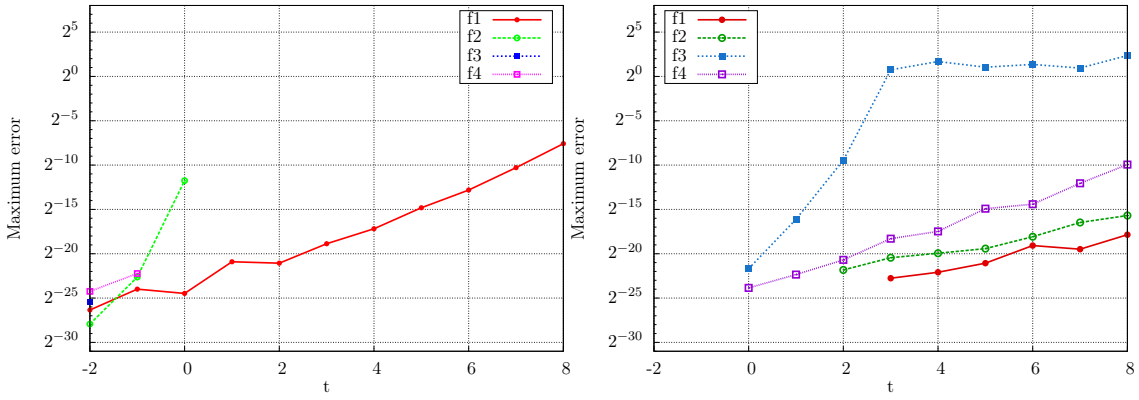
(a) Cholesky decomposition.

(b) Triangular matrix inversion.

Figure 5.4: Results obtained on the Cholesky decomposition and triangular matrix inversion of matrices of size 5×5 with different functions to set the format of division.

Obviously, one can observe that the function used to determine the output format of division has a great impact on the accuracy of the generated code. For example, if we consider the case $t = 0$ on 5×5 Cholesky decomposition on Figure 5.4a, using f_1 leads to an error of $\approx 2^{-28}$, while using f_3 gives an error $\approx 2^{-15}$, that is, twice larger than f_1. More particularly, we can observe that a good function choice is one that minimizes the output integer part but not too much. Indeed, as long as $t \geq -1$, using the function f_1 always leads to better maximum error than using the function f_3. In addition, surprisingly, as long as $t \geq -1$, the function that gives the best results is $f_1(i_1, i_2) = t$, namely the function that fixes explicitly all the division results of a resulting code to the same fixed-point format independently of the input formats.

Indeed the problem of using a function that depends on the input formats comes from the fact that it quickly leads to a growth of the integer part of each computed coefficient,



(a) Cholesky decomposition.

(b) Triangular matrix inversion.

Figure 5.5: Results obtained on the Cholesky decomposition and triangular matrix inversion of matrices of size 10×10 with different functions to set the format of division.

since it relies on the previously computed coefficient themselves. Hence the interest of f_1 is that it avoids this fast growth, and leads to result coefficients having a fixed and relatively small integer part, thus to tighter errors than the other functions. This remark is true for the four experiments of Figures 5.4 and 5.5 when $t \geq 0$ where f_1 is the only function that leads to successful results. This phenomenon becomes obvious as the matrix size increases.

However, one should not be too optimistic and set the value t of f_1 to a very low value when implementing triangular matrix inversion, and cases occur where f_1 leads to unsuccessful results. Indeed, the value of the diagonal coefficient of the inverse matrix is $1/a_{i,i}$ and since $a_{i,i}$ may be arbitrarily small, one way to fix the right t is to choose it such that no division overflows occur when computing the division of the diagonal elements.

These experiments may also be seen as simulations to find the right t. Indeed, suppose we need to generate fixed-point code for the inversion of size-10 triangular matrices. FPLA comes with helper scripts that generate tests matrices and produce the figures similar to Figures 5.4 and 5.5. A strategy then consists in using these figures to restrain the search for the adequate output format. In the cases of interest, $g(i_1, i_2) = 3$ and $h(i_1, i_2) = \lfloor (i_1 + i_2)/2 \rfloor$, except for size-10 Cholesky decomposition, seem to be the most promising functions to set the output of division, in terms of accuracy.

5.4.3 Sharpness of the error bounds and generation time

The originality of our approach is the automatic generation of certified error bounds along with the synthesized code. This enables the generated code to be used in critical and precision sensitive applications. However, it is equally important that these bounds be sharp, at least for a large class of matrices. To investigate their sharpness, we compare in this

second experiment the error bounds for the case of triangular matrix inversion with the experimental errors obtained from inverting 10000 sample matrices. This experiment is carried out using the function f_4 introduced in the previous experiment with $t = 1$. For each matrix size from 4 to 40, C code and error bounds are obtained by running FPLA. Figure 5.6 shows the evolution of the generation time when the size of the matrices grows. On an Intel Core i7-870 desktop machine running at 2.93 GHz, it does not exceed 14 seconds for 40×40 matrices. This is clearly an improvement of several orders of magnitude over a hand written fixed-point code.

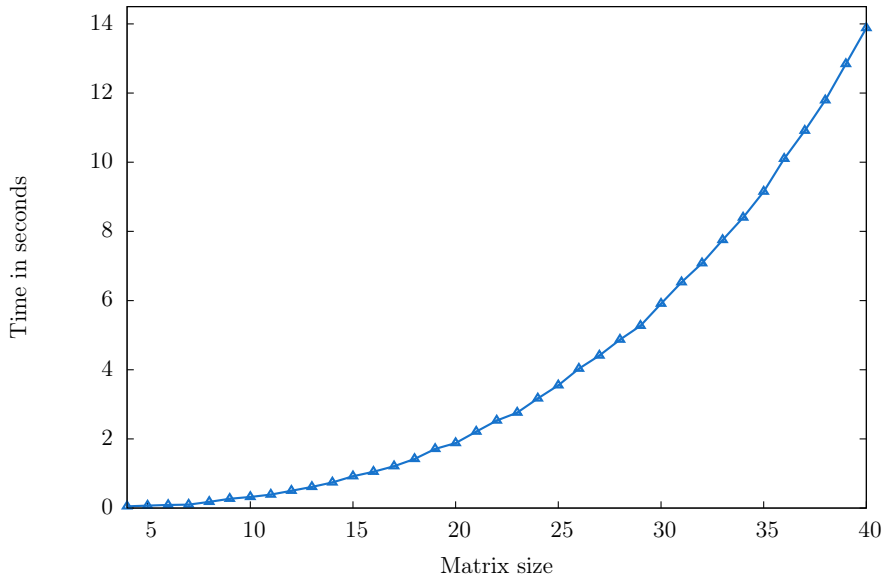


Figure 5.6: Generation time for the inversion of triangular matrices of size 4 to 40.

Besides being quickly generated, Figure 5.7 shows that these codes have low accuracy bounds, at least when the matrix size is less than 30. The bounds vary from 2^{-26} to 2^2 while the experimental errors vary from 2^{-28} to 2^{-6}. The difference between the error bounds and experimental errors is less than 2 bits for size-4 matrices and is inferior to 5 bits for size-15 matrices, and it grows as the size of the input matrices grows. Nevertheless, the two curves have the same overall shape, and the gap between them grows smoothly. And, although the bounds obtained for matrices of size larger than 35 are too large to be useful in practice, the experimental errors are still tight enough and do not exceed 2^{-6}. These issues may be tackled by considering other means to handle division that are more suited to large matrices. Indeed, our experiments tend to show that the output format of division impacts heavily the accuracy of the result and that there is no way to determine a format that is adapted to all matrix sizes. We also argue that a bound of 2^{-12} on the inversion of size-20 matrices is satisfying for a broad range of applications, and this is a large improvement over

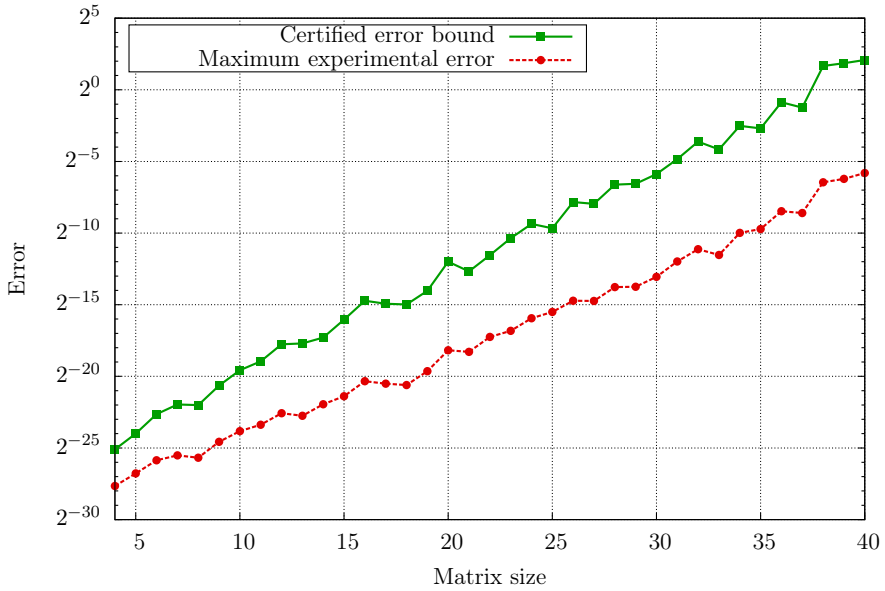


Figure 5.7: Comparison of the error bounds and experimental errors for the inversion of triangular matrices of size 4 to 40.

hand-written fixed-point codes or codes whose numerical quality is asserted uniquely by simulations and *a posteriori* processes.

5.4.4 Impact of the matrix condition number on accuracy

The sample matrices considered in the previous experiments were randomly drawn in the input intervals. In this third experiment, we consider the Cholesky decomposition of some standard matrices namely, KMS, Lehmer, Prolate, Hilbert, and Cauchy matrices. These symmetric positive-definite matrices have multiple properties and are often provided by numerical computing environments. Indeed, we generated them using MATLAB's `gallery('name', size)` command. Among these, Hilbert and Cauchy matrices and to a lower extent Prolate are ill-conditioned as shown in Figure 5.8.

Nonetheless, with a fixed-point code generated for matrices in the input format $\mathbf{Q}_{1,31}$, we were able to check that the fixed-point results, whenever computable, are accurate as shown in Figure 5.9. For sizes larger than 8 and 9, respectively, overflows occur when computing the decompositions of Cauchy and Hilbert matrices. But this fact does not invalidate our approach. Indeed, these matrices are very ill-conditioned and are difficult to decompose accurately even in floating-point arithmetic. On the other hand, KMS and Lehmer matrices have a linearly growing condition number and are therefore very well suited to our approach. As shown by the two bottom curves of Figure 5.9, the code generated by FPLA decomposes these matrices with a precision of up to 24 bits.

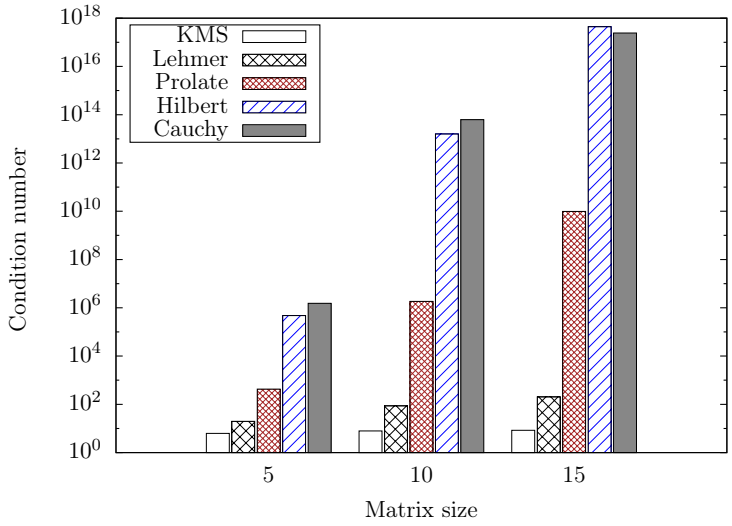


Figure 5.8: Evolution of the conditioning of the matrices of Figure 5.9.

In practice, nothing is noticeable on synthesis time, since the input matrices are made of fixed-point variables and that the same fixed-point code is generated for the 5 different classes of matrices. However, at run-time, ill-conditioned matrices tend to overflow more often and for smaller matrix sizes than well-conditioned matrices.

5.5 Conclusion

In this chapter, we presented an automated approach to help in writing codes in fixed-point arithmetic for the particular case of matrix inversion based on Cholesky decomposition. This was made possible by enhancing our tool-chain with the division and square root operators described in Chapter 2. We finally showed that accurate fixed-point codes accompanied by bounds on the rounding errors can be automatically generated in a few seconds to invert and to decompose matrices of sizes up to 40. The greatest difficulty of this process is related to fixing the output format of divisions. We tested various strategies to solve this issue, but it seems to be very dependent on the properties of the matrices to invert.

Finally, further research directions on matrix inversion may consist in investigating, similarly to the work done in [MNR14a] and presented in Chapter 4, the different trade-offs involved in the code synthesis process and especially the one between code size and accuracy. Indeed, this is shown in Figure 5.10 where the size of the computed dot-product is shown for each coefficient. In this Figure, the variables that can be merged share the same pattern, and one can evaluate each sub-diagonal using a unique code. This would reduce the code size, however, it will degrade the accuracy, since this code is no longer

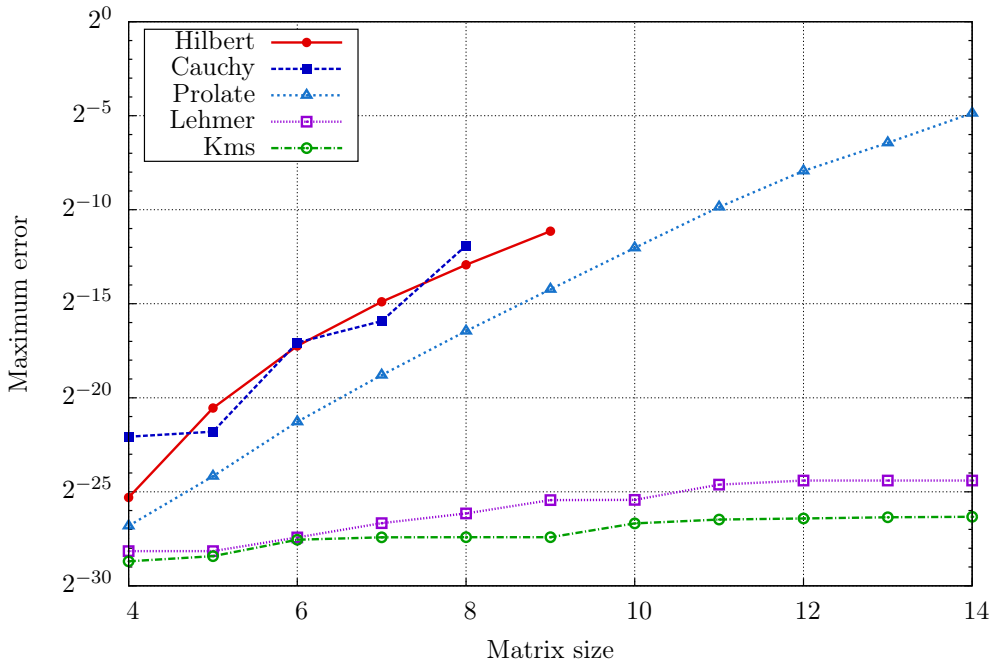


Figure 5.9: Maximum errors measured when computing the Cholesky decomposition of various kinds of matrices for sizes varying from 4 to 14.

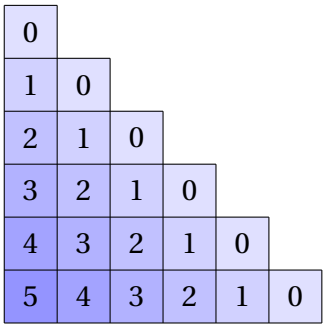


Figure 5.10: The variables that can be merged to reduce the code size of the triangular inversion of a 6×6 matrix.

optimized for the range of its inputs but takes as input the merging of many elements. Indeed, every sub-diagonal requires a dot-product computation of fixed size and one run-time DPCode could be used to evaluate it.

CHAPTER 6

FPLA: A SOFTWARE TOOL FOR FIXED-POINT CODE SYNTHESIS

This chapter presents the FPLA software tool. This tool generates fixed-point code for matrix multiplication, Cholesky decomposition, and triangular matrix inversion, and relies on CGPE for low level code synthesis. Its design makes it easy to add support for new basic blocks such as discrete transformations and convolutions. The chapter also summarizes the software development effort undertook during this thesis and suggests possible enhancements to the tool-chain.

6.1 Introduction

THE software development work done during this thesis reflects its organization. Initially, it was oriented towards making CGPE a versatile tool that generates fixed-point code for many types of expressions. Therefore, it was very entrenched in arithmetical details.

Then, we noticed that the learning curve to use CGPE was made steep by the low level details of the arithmetic model. Indeed, it was tedious to use CGPE directly to generate code for matrix multiplication. We therefore started working on FPLA (Fixed-Point Linear Algebra). This tool was a tentative to provide a user friendly interface to CGPE for the synthesis of matrix multiplication codes. For instance, the first version of FPLA took floating-

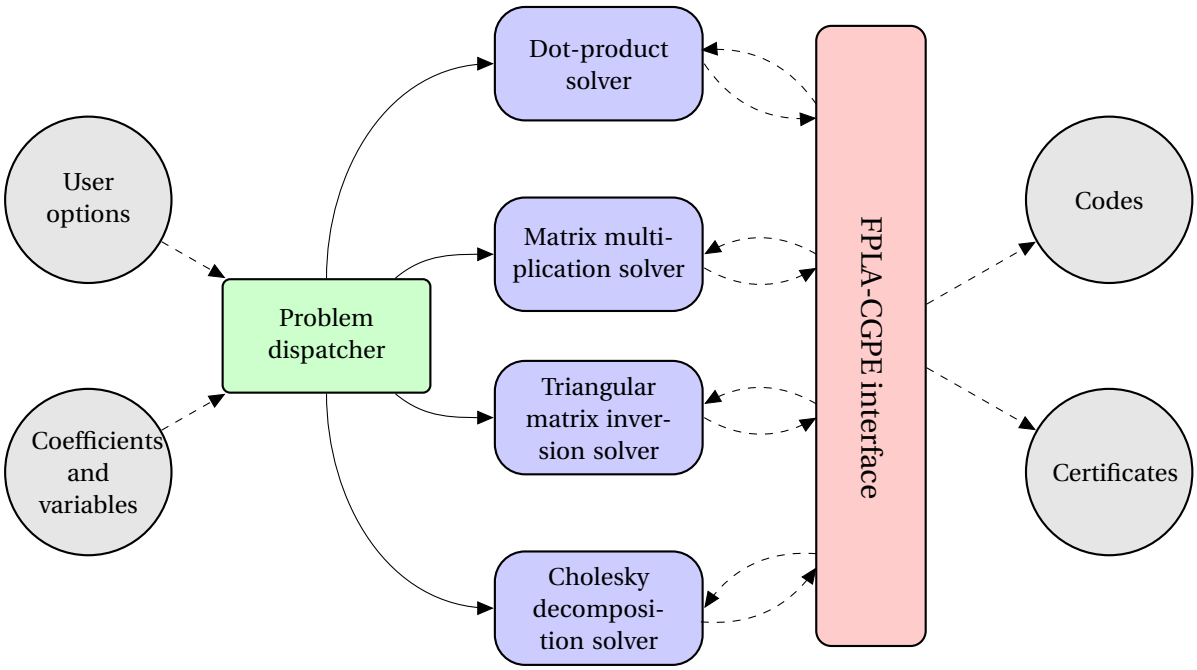


Figure 6.1: The current flow of FPLA.

point enclosures as input, transformed them to fixed-point, and generated an input script and `XML` file for each dot-product that needed to be synthesized. CGPE had to be run on each of these scripts to generate the entire matrix multiplication code. However, it was quickly clear to us that many trade-offs and algorithmic level optimizations were needed to generate efficient code for this kind of linear algebra problems. Not to mention matrix inversion, where as shown in Chapter 5, information needs to be passed in both ways between the two tools.

To adapt to this, CGPE was enhanced with a library version which allowed it to be used programmatically and to be directly interfaced with FPLA. From then on, FPLA evolved into a toolbox for code synthesis for linear algebra basic blocks. Indeed, the experiments of Chapters 4 and 5, and of our publications [MNR14a] and [MNR14b] were obtained using FPLA. In its current version, it generates code for the following problems:

1. dot-products,[1]
2. two dimensional discrete convolutions,
3. matrix multiplication: apart from the straightforward strategies, many strategies that look for trade-offs for matrix multiplication are implemented. The closest pair algorithm 4.6 is only one of 6 possible strategies. Also, many distances can be used along with these strategies.

[1] To this date, this is the only vector operation supported.

4. triangular matrix inversion: this basic block is implemented without support for trade-off strategies. However, since it involves divisions, it takes as input a function to set the output format of division,
5. Cholesky decomposition: like 4, it takes as input a function to set the output format of division.

6.2 Architecture of FPLA

The architecture of FPLA is shown in Figure 6.1 and is designed to be extensible. Indeed, to add a new solver, it suffices to inform the dispatcher, and to inherit from an abstract solver class. Solvers initialize an interface to the CGPE library and may allocate internal matrices to store intermediate variables, formats, and errors.

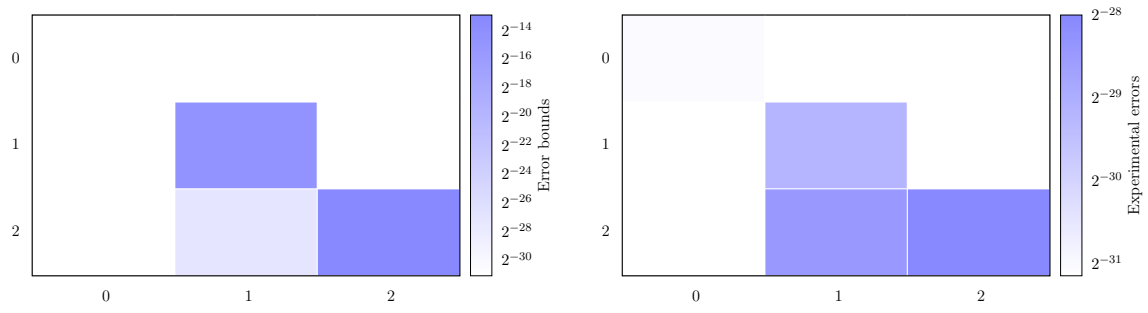
6.3 Example of code synthesis for Cholesky decomposition

In this section, we show the results of code synthesized by FPLA for Cholesky decomposition. As input, let us consider the matrix $A \in \mathbb{F}\text{ix}^{3\times 3}$ whose elements are described in Table 6.1.

We use the function $f(i_1, i_2) = 2$ to set the output format of divisions. FPLA generates the code for this example and produces the certified error bounds of Figure 6.2a. Next, 50 different matrices are generated according to the method presented in Section 5.4.1

Coefficient	Fixed-point format	Range of the integer representation
$A_{1,1}$	$\mathbf{Q}_{1,31}$	$[-1087801124, -516832474]$
$A_{1,2}$	$\mathbf{Q}_{1,31}$	$[-1761415931, -1653153893]$
$A_{1,3}$	$\mathbf{Q}_{1,31}$	$[-7579046, 1706840935]$
$A_{2,1}$	$\mathbf{Q}_{1,31}$	$[-1761415931, -1653153893]$
$A_{2,2}$	$\mathbf{Q}_{1,31}$	$[-860080423, -391158969]$
$A_{2,3}$	$\mathbf{Q}_{1,31}$	$[-323659740, 897800577]$
$A_{3,1}$	$\mathbf{Q}_{1,31}$	$[-7579046, 1706840935]$
$A_{3,2}$	$\mathbf{Q}_{1,31}$	$[-323659740, 897800577]$
$A_{3,3}$	$\mathbf{Q}_{1,31}$	$[-1315844890, 1276745965]$

Table 6.1: Fixed-point variables composing the matrix A.



(a) Error bounds obtained for the Cholesky decomposition of A.

(b) The maximum experimental errors obtained by comparing the synthesized fixed-point code with a binary64 implementation on 50 different matrices that belong to A.

Figure 6.2: Error bounds (left) and experimental errors (right) for the Cholesky decomposition of the size-3 matrix A.

of Chapter 5. The experimental errors of the fixed-point implementation are computed by considering the binary64 implementation as a reference and by taking the maximum errors on the 50 different example matrices. The results are shown in Figure 6.2b. Notice how the errors get larger for the rightmost coefficients. This could be explained by referring to the formula of Equation (5.3) for Cholesky decomposition. In this formula, the coefficients of column j require a size-j dot-product. Therefore, the rightmost coefficients require larger dot-products. And the arguments of these dot-products are themselves the results of previously generated dot-products which involve errors. This explains the pattern followed by both the experimental errors and the bounds. To further illustrate it, Figure 6.3 shows the experimental errors obtained for the Cholesky decomposition of matrices of size 20.

6.4 Example of code synthesis for triangular matrix inversion

We show in this section the quality of code synthesized for triangular matrix inversion by FPLA. Let the input matrix $A \in \mathbb{F}\text{ix}^{3\times 3}$ be such that its coefficients are described in Table 6.2.

We consider now the function $f(i_1, i_2) = 4$ to set the output format of divisions. FPLA produces for this example the certified error bounds shown in Figure 6.4a. Next, the experimental errors are obtained by considering the binary64 implementation as a reference.

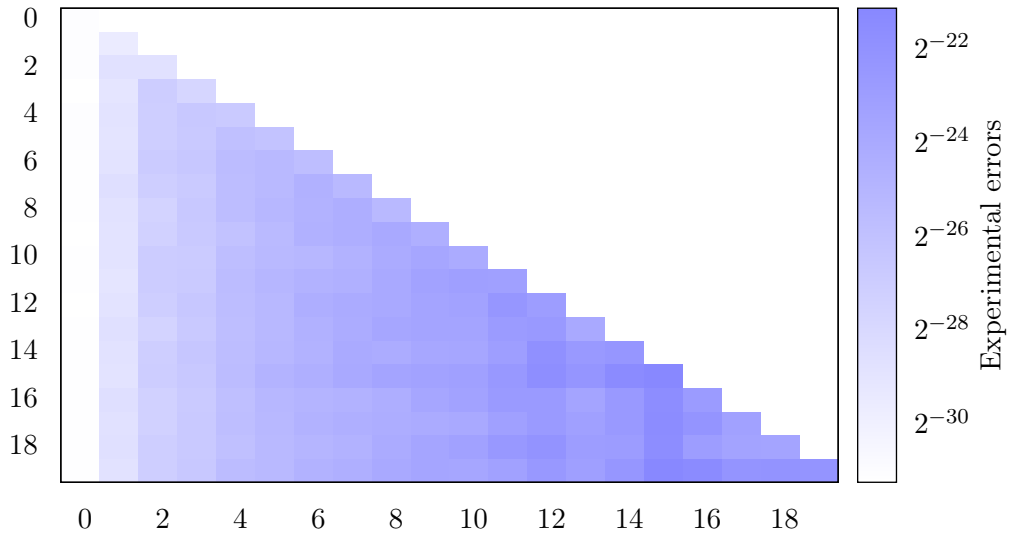


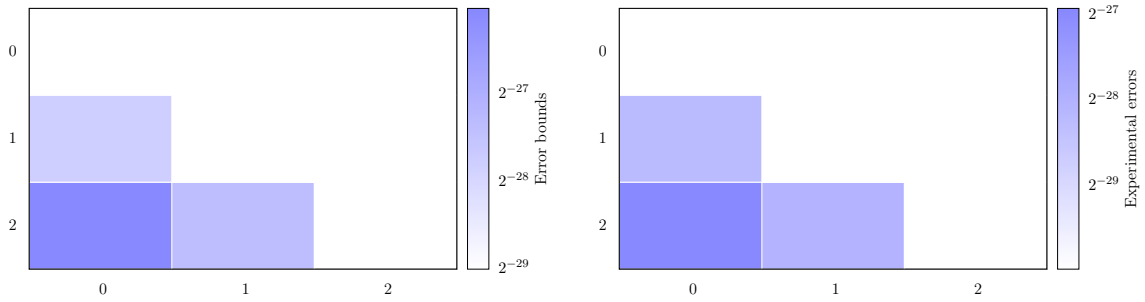
Figure 6.3: The maximum experimental errors obtained by comparing the synthesized fixed-point code for Cholesky decomposition with a binary64 implementation on 50 different matrices of size 20.

Again, these experimental errors, shown in Figure 6.4b, are the maximum errors on the inversion of 50 different sample matrices.

As a first observation, unlike for Cholesky decomposition, the least accurate coefficients seem to be concentrated on the lower left corner of the triangular matrix inverse. Again, this could be explained by examining the formula for triangular matrix inversion in Equation (5.1). The coefficients of the k^{th} sub-diagonal in this formula require a size-k dot-product. Therefore, the most accurate results are obtained for the diagonal elements and the least are those for the lower left corner element. This observation is even more noticeable for large matrices. Indeed, Figures 6.5a and 6.5b show, respectively, the bounds and experimental errors obtained from inverting a triangular matrix of size 20×20.

Coefficient	Fixed-point format	Range of the integer representation
$A_{1,1}$	$\mathbf{Q}_{2,30}$	$[1073737728, 1073741824]$
$A_{1,2}$	$\mathbf{Q}_{0,0}$	$[0,0]$
$A_{1,3}$	$\mathbf{Q}_{0,0}$	$[0,0]$
$A_{2,1}$	$\mathbf{Q}_{1,31}$	$[-1478062229, 1081282683]$
$A_{2,2}$	$\mathbf{Q}_{2,30}$	$[1073737728, 1073741824]$
$A_{2,3}$	$\mathbf{Q}_{0,0}$	$[0,0]$
$A_{3,1}$	$\mathbf{Q}_{1,31}$	$[-645717818, -117989189]$
$A_{3,2}$	$\mathbf{Q}_{1,31}$	$[-1440039578, 1463813698]$
$A_{2,2}$	$\mathbf{Q}_{2,30}$	$[1073737728, 1073741824]$

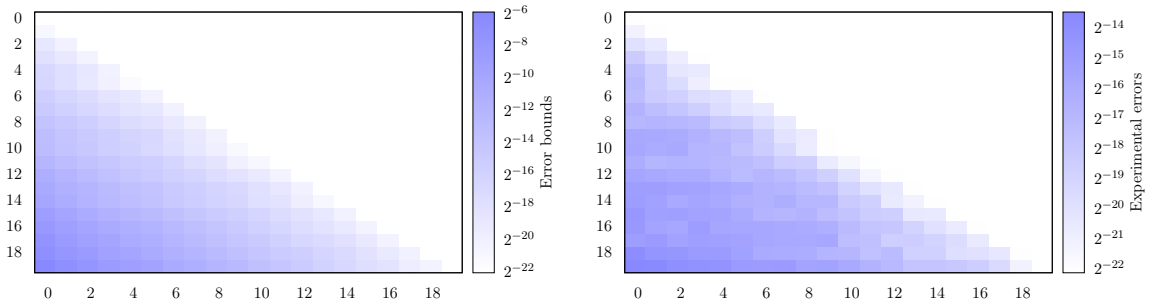
Table 6.2: Fixed-point variables composing the lower triangular matrix A.



(a) Error bounds obtained for the triangular inversion of A.

(b) The maximum experimental errors obtained by comparing the a binary64 implementation on 50 different matrices that belong to A.

Figure 6.4: Error bounds (left) and experimental errors (right) for the inversion of the size-3 triangular matrix A.



(a) Error bounds obtained for the inversion of a size-20 triangular matrix.

(b) The maximum experimental errors obtained by comparing the synthesized fixed-point code with a binary64 implementation on 50 different matrices of size 20.

Figure 6.5: Error bounds (left) and experimental errors (right) for the inversion of a size-20 triangular matrix.

6.5 Future development of the tool-chain

6.5.1 CGPE

The current version of CGPE is strongly tied to the arithmetic model presented in Chapter 2. One enhancement would be to separate the functioning of the tool from the arithmetical details. CGPE could then be tested with different arithmetic models that would be specific to each target. This would allow us to generate certified code for targets providing saturation upon overflow, guard bits, or wider accumulators without modifying the tool.

A second area to explore is high level synthesis: indeed, thanks to the FloPoCo library [dDP11], CGPE can generate VHDL. However, we did not study the quality of the synthesized architectures. For instance, we should investigate the difference between generating pipelined and combinatorial architectures. Also, FPGAs allow one to work with custom word-lengths. Although most experiments presented in this work assumed a fixed word-length of 32 bits, CGPE supports word-length optimization as presented in Section 1.3.5 of Chapter 1. Therefore, the savings in resources obtained from customizing word-lengths inside CGPE should be further studied.

6.5.2 FPLA

FPLA can essentially be enhanced by adding support for new basic blocks. For instance, for vector operations, FPLA may be enhanced with norms computations. For matrix in-

version, other work-flows such as those based on QR and LU decomposition may also be implemented. Also, transformations such as the discrete cosine transform (DCT) and convolutions are ubiquitous in signal processing and may be added to FPLA.

Finally, the trade-off strategy shown in Chapter 4 for matrix multiplication should be extended to matrix inversion.

6.5.3 FxPLib

FxPLib is an ongoing project started in collaboration with the PEQUAN team (Univ. Pierre et Marie Curie) to provide a library for fixed-point code simulation. It grew out of the frustrations induced by working with proprietary libraries such as Mentor Graphics®`ac_fixed`. These libraries are tailored for non-expert users and tend to hide some implementation details from the user.

FxPLib is intended to serve the following two purposes:

1. simulate the fixed-point codes generated by our tools,
2. enforce the use of a standardized notation for describing fixed-point computations, where all intermediate steps are made explicit.

In FxPLib, code is written in a three address style where each variable has a fixed-point format, and all the operations are explicit. For instance, computing the product of two variables starts by computing the full word-length doubling multiplication and then, either truncates the result or rounds it before assigning it to a variable with a smaller format. Listing 6.1 shows the FxPLib code generated by CGPE for the S_1 scheme of the filter of Equation (3.4) of Chapter 3.

▷ **Listing 6.1:** FxPLib code for the filter of Equation (3.4) .

```
FxPVar filter_fxp(FxPVar& u0, FxPVar& u1, FxPVar& u2, FxPVar& u3, FxPVar& y1,
    FxPVar& y2, FxPVar& y3){
  FxPVar b0(-3, -35, true); b0.set_integer(0x65718e3b);
  // ----------
  FxPVar b1(-1, -33, true); b1.set_integer(0x4c152aad);
  // ----------
  FxPVar b2(-1, -33, true); b2.set_integer(0x4c152aad);
  // ----------
  FxPVar b3(-3, -35, true); b3.set_integer(0x65718e3b);
  // ----------
  FxPVar a1(2, -29, true); a1.set_integer(0x252e6d93);
  // ----------
  FxPVar a2(1, -28, true); a2.set_integer(-0x0b2294df);
  // ----------
  FxPVar a3(-1, -33, true); a3.set_integer(0x4688a637);

  // ----------
  FxPVar r0(8, -24, true);
  FxPVar r0_full(8, -55, true); r0_full = a1 * y1;
  r0 = r0_full.shift_to_format(8, -24, fixed_rndd);
  // ----------
  FxPVar r1(7, -25, true);
  FxPVar r1_full(7, -54, true); r1_full = a2 * y2;
  r1 = r1_full.shift_to_format(7, -25, fixed_rndd);
  // ----------
  FxPVar r2(5, -27, true);
  FxPVar r2_full(5, -59, true); r2_full = a3 * y3;
  r2 = r2_full.shift_to_format(5, -27, fixed_rndd);
  // ----------
  FxPVar r3(2, -30, true);
  FxPVar r3_full(2, -62, true); r3_full = b0 * u0;
  r3 = r3_full.shift_to_format(2, -30, fixed_rndd);
  // ----------
  FxPVar r4(2, -30, true);
  FxPVar r4_full(2, -62, true); r4_full = b3 * u3;
  r4 = r4_full.shift_to_format(2, -30, fixed_rndd);
  // ----------
  FxPVar r5(2, -30, true);
  r5 = r3 + r4;
  // ----------
  FxPVar r6(4, -28, true);
  r6 = r5.shift_to_format(4, -28);
```

```
42    // ----------
43    FxPVar r7(4, -28, true);
44    FxPVar r7_full(4, -60, true); r7_full = b1 * u1;
45    r7 = r7_full.shift_to_format(4, -28, fixed_rndd);
46    // ----------
47    FxPVar r8(4, -28, true);
48    FxPVar r8_full(4, -60, true); r8_full = b2 * u2;
49    r8 = r8_full.shift_to_format(4, -28, fixed_rndd);
50    // ----------
51    FxPVar r9(4, -28, true);
52    r9 = r7 + r8;
53    // ----------
54    FxPVar r10(4, -28, true);
55    r10 = r6 + r9;
56    // ----------
57    FxPVar r11(5, -27, true);
58    r11 = r10.shift_to_format(5, -27);
59    // ----------
60    FxPVar r12(5, -27, true);
61    r12 = r2 + r11;
62    // ----------
63    FxPVar r13(7, -25, true);
64    r13 = r12.shift_to_format(7, -25);
65    // ----------
66    FxPVar r14(7, -25, true);
67    r14 = r1 + r13;
68    // ----------
69    FxPVar r15(8, -24, true);
70    r15 = r14.shift_to_format(8, -24);
71    // ----------
72    FxPVar r16(8, -24, true);
73    r16 = r0 + r15;
74    // ----------
75    FxPVar r17(6, -24, true);
76    r17 = r16.shift_to_format(6, -24);
77    return r17;
78 }
```

Lines 2 to 14 of this listing initialize the fixed-point coefficients of the filter and the rest of the listing is the computation of the evaluation scheme. For example, between Lines 17 and 19, a product is computed as follows:

1. the variable that will hold the output of multiplication is declared in Line 17,
2. an intermediate variable whose size is the sum of that of its operands is declared in Line 18,
3. the full product is computed and assigned to this variable in Line 18,
4. in Line 19, an explicit conversion is performed whereby only the upper half of the result is kept and truncation is performed.

6.6 Conclusion

For fixed-point programming to appeal to non expert programmers, code generation tools should be provided. Our contributions to reach this goal include the development of the CGPE and FPLA tools. This development resulted from an effort to provide a tool-chain built from the ground up on an arithmetic model. With these tools, we were able to generate in few seconds, code for large problems such as the triangular matrix inversion of size 40×40. Also, as a proof of concept, our tool-chain showed that one can generate code for large problems and still have error bounds that reassure the programmer. Finally, for each of the tools, we described some further improvements. These improvements push towards more research results on certified fixed-point code generation for high level problems but are not intended to provide industrial strength to our tools. To do so, more integration and better interactivity are needed.

CONCLUSION

THE work presented in this thesis addressed the automated synthesis of certified fixed-point programs and treated the particular cases of code generation for some linear algebra basic blocks. For this purpose, it tackled two recurrent issues encountered by embedded systems developers. These issues are the difficulty of fixed-point programming and the perceived low numerical quality of fixed-point computations. This work also led to the development of two software tools: CGPE and FPLA.

Tackling the difficulty of fixed-point programming

Writing fixed-point programs is tedious, time consuming, and requires arithmetic proficiency. To make it accessible to non-experts, this work suggests to build fixed-point synthesis tools instead of manually developing fixed-point programs. Moreover, to make it easy to perform range and error analyses, such tools must have a clear arithmetic model.

We presented an example of arithmetic model in Chapter 2 of Part I and provided a tool, CGPE, that implements it. While this error model is not suitable for all targets, the detailed steps of Chapter 2 should enable one to develop a model that is more suitable to his target.

Finally, our arithmetic model gives bounds on the rounding errors of fixed-point square root and division as well as the means to implement them. These operators are useful for linear algebra basic blocks and are often overlooked in research publications.

The CGPE library. CGPE implements the arithmetic model of Chapter 2 and was presented in Chapter 3. It was greatly enhanced during this thesis and support was added for signed arithmetic, shifts handling, new expressions such as dot-products, and instruction selection. Moreover, the current tool has an extensible architecture and comes with a li-

brary version. Thanks to CGPE, we showed in the case study of Chapter 3 that obtaining a fixed-point implementation of an Infinite Impulse Response filter is fast and automated. Our results also show that, besides having a low evaluation latency, the generated code is more accurate than a floating-point implementation using the same word-length. Finally, CGPE is now an open source research project, and can be seen as a proof of concept for a tool that strictly adheres to an arithmetic model.

Tackling the low numerical quality of fixed-point computations

The second issue with fixed-point numbers is their perceived lack of accuracy. This thesis tackles it by suggesting that the arithmetic model comes with strict bounds on the rounding errors of each operator. In Chapter 2, we showed how to derive such bounds. And by doing so, the tool that implements the model can propagate these bounds and provide the user with guarantied enclosures on rounding errors. For instance, Section 3.4 showed, step by step, how we obtained certified bounds on the rounding errors of our implementation of an IIR filter.

These error bounds serve two purposes: give feedback to the user to reassure him on the quality of the generated code and allow this code to be used in critical applications. Indeed, up until now, the accuracy of fixed-point implementations has been asserted *a posteriori* by using simulations and by comparing their result to that of floating-point implementations. Contrarily to our approach, such techniques do not provide guaranties and do not scale when the problem at hand has many input variables.

Tackling code synthesis for large linear algebra problems

Part II of this thesis relied on the arithmetic model and the tool presented in Part I to investigate code generation for higher level problems. Such problems include linear algebra basic blocks such as matrix multiplication and inversion. We showed in Chapter 4 that code synthesis for matrix multiplication can rely on straightforward approaches. But these approaches are on the edges of the accuracy versus code size spectrum. Therefore, we suggested, implemented, and provided experimental data for a novel approach that finds trade-offs between these algorithms.

Finally, we tackled matrix inversion in Chapter 5. We showed that the order of synthesis must be arranged to respect the dependencies between the coefficients. Matrix inversion also provided a test case for our square root and division operators. We showed through our experiments that the user must be careful in setting the output format of division. Indeed, small output formats lead to tight accuracy bounds but involve a high risk of run-time overflows.

The FPLA tool. FPLA is a toolbox built on top of CGPE to generate fixed-point code for linear algebra primitives. It is described in Chapter 6. This tool contains our implementa-

tions of the techniques presented in Chapters 4 and 5. In these Chapters, we discussed the numerical quality and accuracy bounds of codes generated using this tool. Finally, FPLA is meant to be extensible and we look forward to enhance it with support for other linear algebra basic blocks.

Future works

Using compilers and programming in high level languages is commonplace today. Yet, in the early 1960's, many software companies were skeptical about adopting these innovations. And, the shift towards high level languages was only finalized after compilers proved their effectiveness.

It seems that fixed-point code generation tools are in the same situation as early compilers. While many research works were published describing a variety of results and techniques for automated range and precision analyses, the absence of any open source or any widely used tools prevented reproducibility. However, the work of recent projects such as SPIRAL [VP04] and DEFIS [MRS+12] is encouraging. Indeed, these projects provide the missing link between academic research and industrial applications of fixed-point programming. In light of this and of the contributions of this thesis, we give three future research directions on fixed-point arithmetic:

Towards code synthesis for higher level problems. As of today, fixed-point programmers working on large applications have no choice but to implement part of them in floating-point arithmetic. For instance, an application like Space Time Adaptive Processing (STAP) for radars involves computing matrix inversion. Due to the large number of steps of matrix inversion, it is tedious to implement it without appropriate tools. The same applies to other frequently used basic blocks such as the Fast Fourier Transform, two dimensional convolutions, and advanced filters like Kalman's. This thesis tackled matrix multiplication and showed how to find accuracy versus size trade-off implementations. It also presented experimental data on matrix inversion. Future work would extend the trade-off strategy to matrix inversion as well as to the aforementioned basic blocks.

Towards high level synthesis. High level synthesis consists in generating hardware architectures instead of software implementations. Targeting FPGAs has two justifications: this type of hardware is becoming more and more popular today and it presents the advantage of allowing fully custom designs. For instance, a fixed-point FPGA implementation can use custom word-length numbers and therefore beat floating-point arithmetic in terms of chip area, energy consumption, memory bandwidth consumption, and latency.

Towards more standards and more versatile tools. More than tools, fixed-point programming lacks standards. Basic notions such as fixed-point formats, rounding errors, and

units in the last place are referred to differently among computer arithmeticians and signal processing researchers. Code generation itself is not standardized and is tedious due to the heterogeneity of the DSP architectures. For instance, a tool like CGPE would be considered attractive only as long as its underlying arithmetic model suits the target architecture. Therefore, as a long time objective, future work should be carried out to homogenize the notions coming from the two research communities. Also, future work on how to describe DSP architectures and modelize their arithmetic operations would provide more versatile tools. And the widespread use of these tools will lead to more reproducible research.

Part III

Appendices

APPENDIX A

SOFTWARE VERSIONS TO REPRODUCE THE EXPERIMENTS

For the sake of reproducibility, this appendix gives the software versions used for the experiments throughout this work.

Libaffa

- ***Version 0.9.6:*** available at http://sv.gnu.org/download/libaffa/libaffa-0.9.6.tar.gz.

MPFR

- ***Version 3.1.0:*** available at http://www.mpfr.org/mpfr-3.1.0/mpfr-3.1.0.tar.gz.

MPFI

- ***Version 1.5.1:*** available at https://gforge.inria.fr/frs/download.php/file/30130/mpfi-1.5.1.tar.gz.

GAPPA

- ***Version 0.17.1:*** available at https://gforge.inria.fr/frs/download.php/file/32292/gappa-0.17.1.tar.gz.

GNU Octave

- ***Version 3.6.2:*** available at ftp://ftp.octave.org/pub/gnu/octave/octave-3.6.2.tar.gz.

Matlab

- ***Version 7.11.0.584 (R2010b):*** See http://www.mathworks.com.

Sollya

- ***Version 4.0:*** available at https://gforge.inria.fr/frs/download.php/file/32549/sollya-4.0.tar.gz.

CGPE

Chapters 3, 5, and 6

- ***Revision 560:*** retrievable from CGPE's repository
  `svn checkout svn://scm.gforge.inria.fr/svnroot/cgpe/trunk@560`.

Chapter 4

- ***Revision 516:*** retrievable from CGPE's repository
  `svn checkout svn://scm.gforge.inria.fr/svnroot/cgpe/trunk@516`.

FPLA

Chapter 4

- ***Revision 79:*** retrievable from FPLA's repository
  `svn checkout https://gforge-lirmm.lirmm.fr/svn/fpp@79`.

Chapters 5 and 6

- ***Revision 107:*** retrievable from FPLA's repository
  `svn checkout https://gforge-lirmm.lirmm.fr/svn/fpp@107`.

BIBLIOGRAPHY

[75408] IEEE 754. IEEE Standard for Floating-Point Arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, August 2008. *[pages 9, 17, 18, and 83]*

[ABB+99] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999. *[page 104]*

[ARM09] *ARM Architecture Reference Manual*, 2009. *[page 83]*

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques and tools*. Addison Wesley, 1986. *[page 84]*

[BB92] V. Balakrishnan and S. Boyd. On computing the worst-case peak gain of linear systems. *Systems & Control Letters*, 19(4):265–269, 1992. *[pages 34 and 35]*

[BBDdD+04] Christian Bertin, Nicolas Brisebarre, Benoit Dupont de Dinechin, Claude-Pierre Jeannerod, Christophe Monat, Jean-Michel Muller, Saurabh-Kumar Raina, and Arnaud Tisserand. A floating-point library for integer processors, 2004. *[page 76]*

[BC07] Nicolas Brisebarre and Sylvain Chevillard. Efficient polynomial l-approximations. In *18th IEEE Symposium on Computer Arithmetic (ARITH-18 2007), 25-27 June 2007, Montpellier, France*, pages 169–176, 2007. *[page 87]*

[BMP06] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. The design of the boost interval arithmetic library. *Theor. Comput. Sci.*, 351(1):111–118, February 2006. *[page 29]*

[Bol08] Thomas Bollaert. Catapult synthesis: A practical introduction to interactive c synthesis. In Philippe Coussy and Adam Morawiec, editors, *High-Level Synthesis*, pages 29–52. Springer Netherlands, 2008. *[page 49]*

[CB94] H. Choi and W.P. Burleson. Search-based wordlength optimization for vlsi/dsp synthesis. In *VLSI Signal Processing, VII, 1994., [Workshop on]*, pages 198–207, 1994. *[page 40]*

[CC99] Andrea G. M. Cilio and Henk Corporaal. Floating point to fixed point conversion of c code. In *Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, pages 229–243, London, UK, 1999. Springer-Verlag. *[page 13]*

[CCL01] G.A Constantinides, P. Y K Cheung, and W. Luk. The multiple wordlength paradigm. In *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, pages 51–60, March 2001. *[page 40]*

[CDH03] Hangjun Chen, Xinmin Deng, and A. Haimovich. Layered turbo space-time coded mimo-ofdm systems for time varying channels. In *Proc. of the 2003 IEEE Global Telecommunications Conference (GLOBECOM'03)*, volume 4, pages 1831–1836 vol.4, 2003. *[page 125]*

[CDV12] Alexandre Chapoutot, Laurent-Stéphane Didier, and Fanny Villers. Range estimation of floating-point variables in simulink models. In *DASIP*, pages 1–8. IEEE, 2012. *[page 37]*

[CG+09] Jason Cong, Karthik Gururaj, Bin Liu 0006, Chunyue Liu, Zhiru Zhang, Sheng Zhou, and Yi Zou. Evaluation of static analysis techniques for fixed-point precision optimization. In Kenneth L. Pocek and Duncan A. Buell, editors, *FCCM*, pages 231–234. IEEE Computer Society, 2009. *[page 30]*

[CLN+] Sylvain Chevillard, Christoph Lauter, Hong Diep Nguyen, Nathalie Revol, and Fabrice Rouiller. Multiple precision floating-point interval library. Available at http://gforge.inria.fr/projects/mpfi/. *[page 29]*

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009. *[page 115]*

[CWIC99] S. Chen, J. Wu, R. H. Istepanian, and J. Chu. Optimizing stability bounds of finite-precision pid controller structures. *Automatic Control, IEEE Transactions on*, 44(11):2149–2153, Nov 1999. *[page 42]*

[dDP11] F. de Dinechin and B. Pasca. Designing custom arithmetic data paths with flopoco. *Design Test of Computers, IEEE*, 28(4):18–27, July 2011. *[pages 100 and 147]*

[dFS04] Luiz Henrique de Figueiredo and Jorge Stolfi. Affine arithmetic: Concepts and applications. *Numerical Algorithms*, 37(1-4):147–158, 2004. *[page 31]*

[EIM+00] Miloš D. Ercegovac, Laurent Imbert, David W. Matula, Jean-Michel Muller, and Guoheng Wei. Improving Goldschmidt division, square root, and square root reciprocal. *IEEE Trans. Computers*, 49(7):759–763, 2000. *[page 62]*

[EL04] Miloš D. Ercegovac and Tomas Lang. *Digital Arithmetic*. Morgan Kaufmann Publishers, 2004. *[page 62]*

[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 Transactions on Mathematical Software*, 33(2):13:1–13:15, June 2007. *[pages 29 and 97]*

[Fou05] Free S. Foundation. GNU GCC Manual. http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/, 2005. *[page 60]*

[FRC03] Claire F. Fang, Rob A. Rutenbar, and Tsuhan Chen. Fast, accurate static analysis for fixed-point finite-precision effects in dsp designs. In *Proceedings of the 2003 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '03, pages 275–, Washington, DC, USA, 2003. IEEE Computer Society. *[pages 30, 48, 49, 75, and 126]*

[Glo90] Fred Glover. Tabu Search: A Tutorial. *INTERFACES*, 20(4):74–94, July 1990. *[page 42]*

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991. *[page 18]*

[Gue03] Joseph R Guerci. *Space-time adaptive processing for radar*. Artech House, 2003. *[page 126]*

[GVL96] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996. *[pages 104, 126, and 127]*

[Han75] E.R. Hansen. A generalized interval arithmetic. In Karl Nickel, editor, *Interval Mathematics*, volume 29 of *Lecture Notes in Computer Science*, pages 7–18. Springer Berlin Heidelberg, 1975. *[pages 28 and 30]*

[HEKC01] Kyungtae Han, Iksu Eo, Kyungsu Kim, and Hanjin Cho. Numerical word-length optimization for cdma demodulator. In *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*, volume 4, pages 290–293 vol. 4, May 2001. *[page 40]*

[Hig02] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, second edition, 2002. *[pages 104, 125, and 128]*

[HJ95] Samuel P. Harbison and Guy L. Steele Jr. *C - a reference manual (4. ed.)*. Prentice Hall, 1995. *[page 10]*

[HNMS11] Nguyen Hai Nam, Daniel Ménard, and Olivier Sentieys. Novel Algorithms for Word-length Optimization. In *19th European Signal Processing Conference (EUSIPCO-2011)*, Barcelona, Espagne, September 2011. *[page 41]*

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. *[pages 10 and 11]*

[IBMK08] Ali Irturk, Bridget Benson, Shahnam Mirzaei, and Ryan Kastner. An fpga design space exploration tool for matrix inversion architectures. In *Proceedings of the 2008 Symposium on Application Specific Processors*, pages 42–47, Washington, DC, USA, 2008. IEEE Computer Society. *[page 126]*

[IBMK10] Ali Irturk, Bridget Benson, Shahnam Mirzaei, and Ryan Kastner. GUSTO: An Automatic Generation and Optimization Tool for Matrix Inversion Architectures. *ACM Trans. Embed. Comput. Syst.*, 9(4):32:1–32:21, 2010. *[page 126]*

[IMK09] Ali Irturk, Shahnam Mirzaei, and Ryan Kastner. An efficient fpga implementation of scalable matrix inversion core using qr decomposition. *Matrix*, pages 1–12, 2009. *[pages 75 and 126]*

[Int10] International Organization for Standardization. *Programming Languages – C*. ISO/IEC Standard 9899:2010, Geneva, Switzerland, 2010. *[pages 10, 59, and 67]*

[Jac70] L.B. Jackson. Roundoff-noise analysis for fixed-point digital filters realized in cascade or parallel form. *Audio and Electroacoustics, IEEE Transactions on*, 18(2):107–122, Jun 1970. *[page 100]*

[JC08] Fabienne Jézéquel and Jean Marie Chesneaux. Cadna: a library for estimating round-off error propagation. *Computer Physics Communications*, 178(12):933–955, 2008. *[page 18]*

[JJL12] Claude-Pierre Jeannerod and Jingyan Jourdan-Lu. Simultaneous floating-point sine and cosine for VLIW integer processors, February 2012. Manuscript submitted for publication. *[page 21]*

[Jr.02] Henry S. Warren Jr. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. *[pages 55 and 63]*

[KG08] David R. Koes and Seth C. Goldstein. Near-optimal instruction selection on DAGs. In *International Symposium on Code Generation and Optimization (CGO'08)*, Washington, DC, USA, 2008. IEEE Computer Society. *[pages 84 and 86]*

[KiKS96] Seehyun Kim, Ki il Kum, and Wonyong Sung. Fixed-point optimization utility for c and c++ based digital signal processing programs. In *IEEE Trans. Circuits and Systems II*, pages 1455–1464, 1996. *[pages ix, 13, 20, 21, 22, and 104]*

[KWCM98] H. Keding, M. Willems, M. Coors, and H. Meyr. Fridge: a fixed-point design and simulation environment. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '98, pages 429–435, Washington, DC, USA, 1998. IEEE Computer Society. *[pages 1, 19, 21, and 22]*

[Lau08] Christoph Lauter. *Arrondi correct de fonctions mathématiques - fonctions univariées et bivariées, certification et automatisation*. PhD thesis, Univ. de Lyon - ÉNS Lyon, 2008. *[page 87]*

[LCLV08] Dong-U Lee, Ray C. C. Cheung, Wayne Luk, and John D. Villasenor. Hardware implementation trade-offs of polynomial approximations and interpolations. *IEEE Trans. Computers*, 57(5):686–701, 2008. *[page 21]*

[LGC+06] Dong-U Lee, Altaf Abdul Gaffar, Ray C. C. Cheung, Oskar Mencer, Wayne Luk, and George A. Constantinides. Accuracy-guaranteed bit-width optimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(10):1990–2000, 2006. *[pages 25, 30, 42, 104, and 126]*

[LHD12] Benoit Lopez, Thibault Hilaire, and Laurent-Stéphane Didier. Sum-of-products evaluation schemes with fixed-point arithmetic, and their application to IIR filter implementation. In *Conf. on Design and Architectures for Signal and Image Processing (DASIP)*, 2012. *[pages 2, 21, 26, and 48]*

[LHD14] Benoit Lopez, Thibault Hilaire, and Laurent-Stéphane Didier. Formatting bits to better implement signal processing algorithms. In *PECCS*, pages 104–111, 2014. *[pages 14, 21, 95, and 100]*

[LV09] Dong-U Lee and John D. Villasenor. Optimized custom precision function evaluation for embedded processors. *IEEE Trans. Comput.*, 58(1):46–59, January 2009. *[pages 21, 32, and 111]*

[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*. Birkhauser Boston, 2010. *[page 18]*

[MCCS02] Daniel Menard, Daniel Chillet, François Charot, and Olivier Sentieys. Automatic floating-point to fixed-point conversion for DSP code generation. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems CASES '02*, pages 270–276, Grenoble, France, 2002. *[pages 2 and 22]*

[Mel06] Guillaume Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. PhD thesis, ÉNS Lyon, 2006. *[page 78]*

[MKC09a] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009. *[page 28]*

[MKC09b] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. SIAM, 2009. *[page 114]*

[MNR12] Christophe Mouilleron, Amine Najahi, and Guillaume Revy. Approach based on instruction selection for fast and certified code generation. In *Proceedings of the 15th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN 2012)*, Novosibirsk, Russia, September 2012. *[page 6]*

[MNR14a] Matthieu Martel, Amine Najahi, and Guillaume Revy. Code Size and Accuracy-Aware Synthesis of Fixed-Point Programs for Matrix Multiplication. In *Proc. of the 4th International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS 2014)*, pages 204–214. SciTePress, January 2014. *[pages 6, 7, 104, 127, 128, 138, and 142]*

[MNR14b] Matthieu Martel, Amine Najahi, and Guillaume Revy. Toward the synthesis of fixed-point code for matrix inversion based on cholesky decomposition. In *Proceedings of the 6th Conference on Design and Architectures for Signal and Image Processing (DASIP 2014)*, pages 73–80, Madrid, Spain, October 2014. *[pages 6, 7, 61, 65, and 142]*

[MNR14c] Christophe Mouilleron, Amine Najahi, and Guillaume Revy. Automated Synthesis of Target-Dependent Programs for Polynomial Evaluation in Fixed-Point Arithmetic. In *Proceedings of the 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2014)*, Timisoara, Romania, September 2014. *[page 6]*

[Moo66] Ramon E. Moore. *Interval analysis*. 1966. *[pages 18 and 28]*

[Mou11] Christophe Mouilleron. *Efficient computation with structured matrices and arithmetic expressions*. PhD thesis, Univ. de Lyon - ENS de Lyon, 2011. *[pages 82, 83, 86, and 117]*

[MR11] Christophe Mouilleron and Guillaume Revy. Automatic Generation of Fast and Certified Code for Polynomial Evaluation. In E. Antelo, D. Hough, and P. Ienne, editors, *Proceedings of the 20th IEEE Symposium on Computer Arithmetic (ARITH'20)*, pages 233–242, Tuebingen, Germany, July 2011. IEEE Computer Society. *[pages 3, 21, 29, and 76]*

[MRS+12] Daniel Ménard, Romuald Rocher, Olivier Sentieys, Nicolas Simon, Laurent-Stéphane Didier, Thibault Hilaire, Benoît Lopez, Eric Goubault, Sylvie Putot, Franck Vedrine, Amine Najahi, Guillaume Revy, Laurent Fangain, Christian Samoyeau, Fabrice Lemonnier, and Christophe Clienti. Design of Fixed-Point Embedded Systems (defis) French ANR Project. In *Proceedings of the Conference on Design and Architectures for Signal and Image Processing, DASIP 2012, Karlsruhe, Germany, 23-25 October 2012*, pages 365–366, Karlsruhe, Allemagne, 2012. *[pages 2, 5, 22, and 155]*

[Ngu11] Hai-Nam Nguyen. *Optimisation de la précision de calcul pour la réduction d'énergie des systèmes embarqués*. These, Université Rennes 1, December 2011. *[pages 39 and 41]*

[NNF07] Zoran Nikolic, Ha Thai Nguyen, and Gene Frantz. Design and implementation of numerical linear algebra algorithms on fixed point dsps. *EURASIP J. Adv. Signal Process*, 2007:13–13, June 2007. *[pages 22, 37, 104, and 126]*

[NTM92] R. Nambiar, C. K K Tang, and P. Mars. Genetic and learning automata algorithms for adaptive digital filters. In *Acoustics, Speech, and Signal Processing, 1992. ICASSP-92., 1992 IEEE International Conference on*, volume 4, pages 41–44 vol.4, Mar 1992. *[page 42]*

[Ove01] Michael L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. Society for Industrial and Applied Mathematics, Philadephia, PA, 2001. *[page 17]*

[PV08] P. Prandoni and M. Vetterli. *Signal Processing for Communications*. Communication and information sciences. CRC Press, 2008. *[page 34]*

[Rai06] Saurabh-Kumar Raina. *FLIP: a Floating-point Library for Integer Processors*. PhD thesis, Université de Lyon - École Normale Supérieure de Lyon, 46 allée d'Italie, F-69364 Lyon cedex 07, France, September 2006. *[page 76]*

[Rev09] Guillaume Revy. *Implementation of binary floating-point arithmetic on embedded integer processors - Polynomial evaluation-based algorithms and certified code generation*. PhD thesis, Université de Lyon - École Normale Supérieure de Lyon, 46 allée d'Italie, F-69364 Lyon cedex 07, France, December 2009. *[pages 13, 21, 29, 54, 76, 77, 82, and 83]*

[RR05] Nathalie Revol and Fabrice Rouillier. Motivations for an arbitrary precision interval arithmetic and the mpfi library. *Reliable Computing*, 11(4):275–290, 2005. *[page 29]*

[SH75] Michael Ian Shamos and Dan Hoey. Closest-point problems. In *FOCS*, pages 151–162, 1975. *[page 115]*

[SK95] Wonyong Sung and Ki-Il Kum. Simulation-based word-length optimization method for fixed-point digital signal processing systems. *Signal Processing, IEEE Transactions on*, 43(12):3087–3090, 1995. *[pages 2, 22, 37, 40, and 126]*

[ST208] *ST231 core and instruction set architecture – Reference manual*, 2008. *[pages 54 and 83]*

[vN93] John von Neumann. First draft of a report on the edvac. *IEEE Ann. Hist. Comput.*, 15(4):27–75, October 1993. *[page 9]*

[VP04] Yevgen Voronenko and Markus Püschel. Automatic generation of implementations for DSP transforms on fused multiply-add architectures. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 5, pages V–101, 2004. *[pages 21 and 155]*

[Yat13] Randy Yates. *Fixed-Point Arithmetic: An Introduction*. Digital Signal Labs, 2013. *[pages 13, 15, 26, 52, 55, and 66]*

[ZQZ05] Liang Zhou, Ling Qiu, and Jinkang Zhu. A novel adaptive equalization algorithm for MIMO communication system. In *Proc. of the IEEE 62nd Vehicular Technology Conference (VTC-2005-Fall)*, volume 4, pages 2408–2412, 2005. *[page 125]*

Abstract

To be cost effective, embedded systems are shipped with low-end micro-processors. These processors are dedicated to one or few tasks that are highly demanding on computational resources. Examples of widely deployed tasks include the fast Fourier transform, convolutions, and digital filters. For these tasks to run efficiently, embedded systems programmers favor fixed-point arithmetic over the standardized and costly floating-point arithmetic. However, they are faced with two difficulties: First, writing fixed-point codes is tedious and requires that the programmer must be in charge of every arithmetical detail. Second, because of the low dynamic range of fixed-point numbers compared to floating-point numbers, there is a persistent belief that fixed-point computations are inherently inaccurate. The first part of this thesis addresses these two limitations as follows: It shows how to design and implement tools to automatically synthesize fixed-point programs. Next, to strengthen the user's confidence in the synthesized codes, analytic methods are suggested to generate certificates. These certificates can be checked using a formal verification tool, and assert that the rounding errors of the generated codes are indeed below a given threshold. The second part of the thesis is a study of the trade-offs involved when generating fixed-point code for linear algebra basic blocks. It gives experimental data on fixed-point synthesis for matrix multiplication and matrix inversion through Cholesky decomposition.

Keywords: *Fixed-point arithmetic, automated code synthesis, certified numerical accuracy*

Résumé

Pour réduire les coûts des systèmes embarqués, ces derniers sont livrés avec des microprocesseurs peu puissants. Ces processeurs sont dédiés à l'exécution de tâches calculatoires dont certaines, comme la transformée de Fourier rapide, peuvent s'avérer exigeantes en termes de ressources de calcul. Afin que les implantations de ces algorithmes soient efficaces, les programmeurs utilisent l'arithmétique à virgule fixe qui est plus adaptée aux processeurs dépourvus d'unité flottante. Cependant, ils se retrouvent confrontés à deux difficultés: D'abord, coder en virgule fixe est fastidieux et exige que le programmeur gère tous les détails arithmétiques. Ensuite, et en raison de la faible dynamique des nombres à virgule fixe par rapport aux nombres flottants, les calculs en fixe sont souvent perçus comme intrinsèquement peu précis. La première partie de cette thèse propose une méthodologie pour dépasser ces deux limitations. Elle montre comment concevoir et mettre en œuvre des outils pour générer automatiquement des programmes en virgule fixe. Ensuite, afin de rassurer l'utilisateur quant à la qualité numérique des codes synthétisés, des certificats sont générés qui fournissent des bornes sur les erreurs d'arrondi. La deuxième partie de cette thèse est dédiée à l'étude des compromis lors de la génération de programmes en virgule fixe pour les briques d'algèbre linéaire. Des données expérimentales y sont fournies sur la synthèse de code pour la multiplication et l'inversion matricielles.

Mots clefs : *Arithmétique à virgule fixe, génération automatique de code, qualité numérique certifiée*

UPVD — 52 avenue Paul Alduy — 66860 Perpignan Cedex 9 — France