Synthesis of Arithmetic Expressions for the Fixed-Point Arithmetic: The Sardana Approach

Arnault Ioualalen, Matthieu Martel

Synthesis of Arithmetic Expressions for the Fixed-Point Arithmetic: The Sardana Approach

Arnault Ioualalen and Matthieu Martel

Univ. Perpignan Via Domitia, Digits, Architectures et Logiciels Informatiques, F-66860, Perpignan, France

Univ. Montpellier II, Laboratoire d'Informatique Robotique et de Microélectronique de Montpellier
UMR 5506, F-34095, Montpellier, France

CNRS, Laboratoire d'Informatique Robotique et de Microélectronique de Montpellier
UMR 5506, F-34095, Montpellier, France

arnault.ioualalen@univ-perp.fr matthieu.martel@univ-perp.fr

***Abstract*—Sardana is a tool which optimizes the arithmetic expressions present in source codes. The optimization is done by synthesizing automatically new mathematically equal expressions, given ranges of values for the variables. In previous work, Sardana has been used to optimize the numerical accuracy of floating-point expressions, by minimizing the worst roundoff error on the result of the evaluation. In this article, we show how our tool can be used to synthesize arithmetic expressions optimized for the fixed-point arithmetic. In this context, Sardana minimizes the number of bits required to represent without overflow the integer parts of the fixed-point numbers possibly occurring at any stage of the evaluation of an expression. We present experimental results showing how our tool optimizes the implementation of digital filters commonly used in image processing.**

***Index Terms*—Code Synthesis, Fixed-Point Formats, Abstract Interpretation, Compilers.**

I. Introduction

Sardana [] is a tool which optimizes the arithmetic expressions present in source codes. We consider that the formulas written in programs are mathematically exact and that we would obtain exact results if the computations were carried out with real numbers. Then Sardana synthesizes automatically new expressions which are mathematically equal to the original ones while being better-suited to the computer arithmetic. In order to obtain interesting gains, the synthesis is performed for specific ranges of values of the variables. When Sardana is used to optimize the numerical accuracy of IEEE754 floating-point computations [1], it generates a new expression which lowers the worst roundoff error, i.e. the error on the result of the computation, in the worst case, for inputs taken from the specified ranges. In previous work, we have applied these techniques to optimize the numerical accuracy of floating-point computations occurring in critical avionic software [].

In this article, we show how Sardana can be used to optimize fixed-point formats [2]. We aim at minimizing the size of the integer parts of fixed-point numbers while ensuring the absence of overflow. Sardana may optimize either the maximal or the cumulated number of bits needed to represent the integer parts of the intermediary results, during the evaluation of an expression. In addition, the tool computes the roundoff error on the result of the evaluation of the new expression, for a given size of the fractional part of the numbers. To illustrate our techniques, we have carried out several experimentations concerning the optimization of the implementation of digital filters commonly used in image processing.

In general, an arithmetic expression may be rewritten in many ways using associativity, distributivity, etc. Our approach is based on abstract interpretation [3], [4]. To capture as many rewritings as possible while avoiding any combinatorial explosion, we use a new intermediate representation, called APEG, enabling to represent many equivalent expressions in the same structure []. APEGs are abstractions of the Equivalence Program Expression Graphs introduced in [5], [6] for the phase ordering problem. The concretization of an APEG yields expressions of very different shapes. Then, to synthesize expressions from APEGs, we perform a profitability analysis which consists of searching the expression which optimizes the implementation, i.e. which minimizes the size of the integer parts, among the set of expressions represented in an APEG.

The most comparable work concerns the synthesis of operators for the fixed-point arithmetic and the optimization of the size of the integer and fractionnal parts of fixed-point numbers [7], [8], [9]. In comparison, our method seems more general in the sense that it is not a domain specific optimization. Instead, our method consists of building a tractable representation of the set of candidate optimized programs, for a given set of equivalence rules, and then selecting a good candidate with respect to a non-specific optimality criterium. Indeed our techniques are also relevant for other arithmetics, like the filoating-point arithmetic, as discussed in [].

This article is organized as follows. Section II deals with the fixed-point arithmetic and our way of measuring the quality of some implementation of an expression. APEGs are introduced in Section III and Section IV presents the profitability analysis. Experimental results are described in Section V. Finally, Section VI concludes.

II. Fixed-Point Arithmetic

In this section, we briefly survey the aspects of fixed-point arithmetic useful to the comprehension of the rest of this article. We also define the measure that we aim at optimizing when synthesising new expressions.

A. Fixed-Point Arithmetic

There is no general standard for fixed-point arithmetic comparable to the IEEE754 Standard for floating-point arithmetic [1]. Following [10], [11], we consider that a number x is written:

$$x = s \cdot (d_{m-1} \dots d_0 . d_{-1} \dots d_{-n}) = s \cdot \sum_{i=-n}^{m-1} d_i \beta^i \quad (1)$$

In Equation (1), x is a number made of $m + n + 1$ digits. The m first digits represent the integer part while the n last digits represent the fractional part (see Figure 1). $s \in \{-1, 1\}$ is the sign of x. For the basis, we always assume that $\beta = 2$. In addition, for the sake of simplicity and without lost of generality, we do not consider numbers encoded in the two's complement format which is also common in the implementations of fixed-point arithmetic. As argued in the next paragraph, we assume that no overflow arises and that, whenever it is necessary, the results of elementary operations are truncated (rounding mode towards zero).

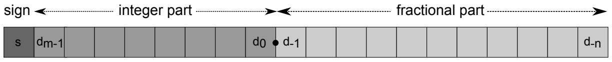

Fig. 1. *Representation of the fixed-point formats.*

Slightly different problems may be formulated concerning the enhancement of the implementation of a formula in fixed-point arithmetic. Following [10], [11], in this article, we are interested in minimizing the m parameter of Equation (1), that is in finding the minimal size for the integer part of numbers such that no overflow arise during the computation, for all the acceptable inputs. So, we introduce the function

$$\mathcal{I}(x) = \min \ \{m \in \mathbb{N} \ : \ \lfloor x \rceil \leq \beta^m\} \quad (2)$$

where $\lfloor x \rceil$ denotes the integer part of x. We consider two cases: In the first case, in the final implementation, all the values may be encoded in the same format (the one for which we determine m). This is usually the case when the program is targeted for a general purpose processor: the designer wants to know, for example, if 16 bits are enough to perform the computation or if all the data must be encoded in a 32 bits format. In this case, Sardana computes the maximal size of the integer part of the values possibly taken by the intermediary results, during the evaluation of the expressions. The second case concerns more specific processing units (e.g. FPGAs) for which one may wish to minimize the size of the circuit by using numbers of different formats (e.g. to use less bits when the values are smaller). In this case, Sardana computes the sum of the sizes of the integer parts of all the intermediary results possibly obtained during the evaluation of the expression, for any set of inputs taken from the ranges specified by the user.

B. Optimized Implementation in Fixed-Point Arithmetic

We consider a simple semantics for the arithmetic expressions whose syntax is given by:

$$e ::= v \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \quad (3)$$

A value is a pair (x, μ) where x denotes the fixed-point number, and μ measures the quality of the implementation. More precisely, assuming that (x, μ) is the result of the evaluation of some expression e, μ denotes the maximal or cumulated number of digits needed to encode the integer parts of the numbers during the evaluation of e and x is the result of this evaluation. In floating-point arithmetic, μ would be the distance between a real number $x_{\mathbb{R}}$ and the floating-point number x corresponding to the roundoff of $x_{\mathbb{R}}$ [].

In the rest of this article, we consider abstract values $(x^\sharp, \mu^\sharp)$ where $x^\sharp$ is an interval. A value $(x^\sharp, \mu^\sharp)$ abstracts a set of concrete values $\{(x_i, \mu_i), \ i \in I\}$. The

$$(x_1^\sharp, \mu_1^\sharp) + (x_2^\sharp, \mu_2^\sharp) = \Big(x_1^\sharp + x_2^\sharp, \max(\mu_1^\sharp, \mu_2^\sharp, \mathcal{I}^\sharp(x_1^\sharp + x_2^\sharp))\Big) \tag{4}$$

$$(x_1^\sharp, \mu_1^\sharp) - (x_2^\sharp, \mu_2^\sharp) = \Big(x_1^\sharp - x_2^\sharp, \max(\mu_1^\sharp, \mu_2^\sharp, \mathcal{I}^\sharp(x_1^\sharp - x_2^\sharp))\Big) \tag{5}$$

$$(x_1^\sharp, \mu_1^\sharp) \times (x_2^\sharp, \mu_2^\sharp) = \Big(x_1^\sharp \times x_2^\sharp, \max(\mu_1^\sharp, \mu_2^\sharp, \mathcal{I}^\sharp(x_1^\sharp \times x_2^\sharp))\Big) \tag{6}$$

$$(x_1^\sharp, \mu_1^\sharp) + (x_2^\sharp, \mu_2^\sharp) = \Big(x_1^\sharp + x_2^\sharp, \mu_1^\sharp + \mu_2^\sharp + \mathcal{I}^\sharp(x_1^\sharp + x_2^\sharp)\Big) \tag{7}$$

$$(x_1^\sharp, \mu_1^\sharp) - (x_2^\sharp, \mu_2^\sharp) = \Big(x_1^\sharp - x_2^\sharp, \mu_1^\sharp + \mu_2^\sharp + \mathcal{I}^\sharp(x_1^\sharp - x_2^\sharp)\Big) \tag{8}$$

$$(x_1^\sharp, \mu_1^\sharp) \times (x_2^\sharp, \mu_2^\sharp) = \Big(x_1^\sharp \times x_2^\sharp, \mu_1^\sharp + \mu_2^\sharp + \mathcal{I}^\sharp(x_1^\sharp \times x_2^\sharp)\Big) \tag{9}$$

Fig. 2. *Abstract semantics of the elementary operations for the fixed-point arithmetic.*

interval $x^\sharp$ abstracts all the values possibly arising during the evaluation and $\mu^\sharp$ indicates the maximal number of bits needed to encode a value somewhere in the computation. We do not need an interval for $\mu^\sharp$ since we only store the greatest value. So, in the fixed-point abstract semantics $\mu^\sharp$ is an integer.

The abstract semantics of arithmetic operations is given in Figure 2. In equations (4) to (6), to compute $\mu^\sharp$, we take the maximum of the measures $\mu_1^\sharp$ and $\mu_2^\sharp$ on the operands and of the measure $\mathcal{I}^\sharp(x^\sharp)$ on the result $x^\sharp$ of an operation. The term $\mathcal{I}^\sharp(x)$ denotes a safe abstraction of $\mathcal{I}(x)$:

$$\mathcal{I}^\sharp([\underline{x}, \overline{x}]) = \max\left\{\mathcal{I}(x) \ : \ x \in [\underline{x}, \overline{x}]\right\}. \tag{10}$$

Equations (7) to (9) correspond to the second optimization. They compute the cumulated number of digits needed to represent the integer parts of the intermediary results during the evaluation of the expression.

Let us conclude this section by considering the following example: Let

$$\mathtt{E} = (\mathtt{a} + (\mathtt{b} + (\mathtt{c} + \mathtt{d}))) \times \mathtt{e} \tag{11}$$

and let us assume that the variables belong to the ranges:

$$\begin{array}{lll} \mathtt{a} \in [-14, -13] & \mathtt{b} \in [-3, -2] & \\ \mathtt{c} \in [3, 3.5] & \mathtt{d} \in [12.5, 13.5] & \mathtt{e} = 2 \end{array} \tag{12}$$

Using the semantics of Figure 2, we obtain:

$$E_{max} = \big([-3, 4], 5\big) \qquad E_{sum} = \big([-3, 4], 28\big)$$

The term E_{max} is obtained using the semantics of equations (4) to (6) while E_{sum} corresponds to the semantics of equations (7) to (9). Under the assumptions of Equation (12), the result returned by the machine always belongs to the interval $[-3, 4]$. In addition, E_{max} states that 5 bits may be needed for the integer part, somewhere in the computation, and E_{sum} states that 28 bits are globally needed to represent the integer parts of the intermediary results. Note that only 3 bits are required for the integer part of the result. However, for instance, if c= 3.5 and d= 13.5 then c+d= 17 and $\mathcal{I}(17) = 5$. Sardana automatically computes that we can rewrite the expression E into

$$\mathtt{E}' = \mathtt{e} \times ((\mathtt{a} + \mathtt{d}) + (\mathtt{b} + \mathtt{c})) \tag{13}$$

and that:

$$E'_{max} = \big([-3, 4], 4\big) \qquad E'_{sum} = \big([-3, 4], 21\big)$$

In $\mathtt{E}'$, a and d are added first and the product is not distributed. These choices make it possible to store all the intermediate values on 4 bits only (in absolute value, the greatest number arising during the computation is 14). In addition, 21 bits are enought to represent the integer parts of all the intermediary results during the evaluation of E'. In the next sections, we aim at showing how Sardana synthesizes expressions like E', given the original expression E and ranges for the inputs, as done in Equation (12).

III. Expression Representation

A. Equilavence Program Expression Graphs

Compilers implement a lot of optimizations such as loop unfolding, vectorization, dead code elimination, etc. Most of these algorithms are easy to implement if the compiler first transforms its intermediate representation (IR) of the programs into one of the SSA forms [12]. In practice, the optimizations are applied in an order determined by the compiler and the order actually chosen may not allow to perform the best optimizations on the programs. Compilers being unable to take into account future transformations of the code, the best sequential order can only be found by solving a combinatorial problem usually known as the *phase ordering problem*, which is proven to be undecidable [13]. Obviously, a SSA form can only represent one version of the code at a time. If we would like to use a SSA form to represent multiple versions of a program through different optimizations, we would have to handle separated *definition-use* chains and this would cause duplications of the code and a significant growth of the IR. To cope with this problem, a new intermediate representation has been introduced in [5], called Equivalent-Program Expression Graph (EPEG). This IR is designed to represent in a single structure multiple versions of a program, each version corresponding to a possible transformation of the code.

EPEGs are based on Program Expression Graphs (PEGs) which share similitudes with the gated SSA representation in the sense that the value of an expression depends only on the value of its sub-expressions, without any side-effect. EPEGs are built by means of equality reasoning. Equalities correspond to possible optimizations and introduce in the source IR, called PEG, new nodes corresponding to new versions of the program, without doubling the size of the representation at each application of an equality. This is possible thanks to the introduction of equivalence classes which record in the same node of the PEG many semantically equivalent ways to implement an expression (see Figure III-B). Therefore, in a PEG, if two expressions are semantically equal then the root nodes of both expressions are in the same equivalence class, and we can select either of them to synthesize an executable code. Whenever the PEG is saturated, we refer to it as an EPEG.

A drawback of EPEGs is that, in some cases, the saturation process may not terminate due to the infinite application of some equalities. Then the user has to set a threshold to ensure the termination of the process. Even without the infinite development of some equalities, EPEGs are not necessarily tractable on a computer. For example, if we build an EPEG with usual algebraic laws like associativity, commutativity, distributivity and factorization, it is possible to generate an exponential number of different expressions. For example, if we consider all the polynomials defined by $P^n(x) = \sum_{k=0}^{n}(-1)^k \times \begin{pmatrix} n \\ k \end{pmatrix} \times x^k$, which corresponds to the developed form of the function $(x-1)^n$, assuming that x^n is written as the product $\prod_{i=1}^{n} x$, for $n = 5$ there are 2.3 million distinct parsings, and for $n = 6$ there are 1.3 billion parsings [14, §6.2.2]. Despite EPEGs are able to merge common parts of expressions, there are for example $\frac{2k!}{(k+1)! \times k!}$ different shapes of expressions that cannot be merged for a summation of k terms [15].

Once some EPEG is complete, the compiler has to synthesize a final version of the program that satisfies the requirements of the user (performance, accuracy, parallelism [16], etc.) This step is referred to as the *profitability* phase and is it performed with a global profitability heuristic (relying on an efficient and tractable cost model.The cost $C(n)$ of a node n is defined as $\textit{basic\_cost}(n) \times k^{\textit{depth}(n)}$, where $\textit{basic\_cost}(n)$ accounts for how expensive is the node n itself, and $k^{\textit{depth}(n)}$ for how often n is executed (*depth* represents the loop nesting factor of the node n and k is a constant).
This cost model avoids a combinatorial explosion by performing on each node a purely local evaluation, without considering the possible choices for the other nodes present in the equivalence classes below.

B. Abstract Program Expression Graphs

In this section, we present our new intermediate representation of programs, called Abstract Program Expansion Graph (APEG). We define the APEGs as an intermediate structure between the initial PEGs and the theoretical complete EPEGs which can be intractable or infinite. The main objective of APEGs is to use abstractions in order to remain polynomial in size while still representing the largest number of equivalent expressions. APEGs contain a compact representation of many transformations of expressions in abstraction boxes which allow one to represent very large sets of expressions in polynomial size, despite that these expressions are of very different shape. An abstraction box is defined by a commutative binary operator, such as $+$ or $\times$, and by a list of nodes which correspond to the operands. These nodes can be either constants, variable identifiers, sub-trees, equivalence classes or abstraction boxes. An abstraction box stands for all the parsings of the given leaves using the binary operator. For example, $\boxed{+, (a, b, c, d)}$ stands for all parsings of the sum $a + b + c + d$. Also, $\boxed{+, (a, b, c, \boxed{\times, (x_1, x_2, x_3, x_4)})}$ stands for all the possible summations of the sum $a + b + c + X$, where X stands for any parsings of the product $x_1 \times x_2 \times x_3 \times x_4$. So, an abstraction box is a very compact structure which is able to represent up to $(2n-1)!!$ possible evaluation schemes according to [14, §6.3], where n in the number of operands of the abstraction box.

Contrarily to EPEG construction, the APEG construction do not rely on equality saturation. Instead, we use several algorithms, each one being performed independently of the others and in polynomial time. We have designed two kinds of transformation algorithms: the propagation algorithms, and the expansion algorithms. Our approach consists of composing each of these algorithms together in order to produce the largest APEG, in the sense of the number of versions of a program it represents, while staying polynomial. We use the following propagation algorithms in order to introduce various shapes of expressions into the APEG:

- We propagate subtractions into the concerned operands. For example, from the expression $a-(b+(c-d))$ we introduce the expressions $a+(-b-(c-d))$ and $a+(-b+(-c+d))$,
- We propagate products into their operands. For example, from the expression $a \times ((b+c)+d)$ we introduces the expressions $(a \times (b+c)) + a \times d$ and $(a \times b + a \times c) + a \times d$,
- We factorize common factors. For example, from the expression $(a \times b + a \times c) + a$ we introduce the expressions $a \times (b+c) + a$ and $a \times ((b+c)+1)$.

Figure III-B illustrates how the propagation of the minus operator is applied to an APEG. The expansion algorithms are not designed to introduce new shapes of expressions. Instead, they only add abstraction boxes into the APEG. These algorithms search recursively in the APEG where a symmetric binary operator is repeated (we referred at these parts as *homogeneous parts*). As the propagation algorithms tend to generate homogeneous parts, the expansion algorithms are meant to be used after the propagation algorithms. When an expansion algorithm founds an homogeneous part it inserts a polynomial number of abstraction boxes into it, each of these abstraction boxes representing alternative versions of the homogeneous part. We have designed our expansion algorithms in order to add abstraction boxes which mainly represent new shapes of expressions. This allow us to manipulate these shapes without having to enumerate them. We have designed the two following expansion algorithms:

a) Horizontal expansion: This algorithm splits recursively an homogeneous part into a left sub-tree and a right sub-tree and insert for each one an abstraction box containing the other. On the example given in Figure III-B our algorithm adds the following nodes to the APEG:

- $\boxed{\text{+, (a,b,c,d)}} + (e+f)$ and $((a+b)+(c+d)) + \boxed{\text{+, (e,f)}}$ into equivalence class 1,
- $(a+b) + \boxed{\text{+, (c,d)}}$ and $\boxed{\text{+, (a,b)}} + (c+d)$ into equivalence class 2.

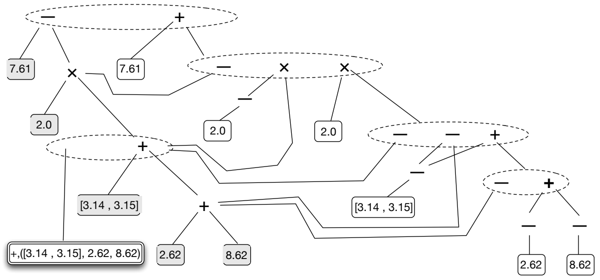

Fig. 3. *Example of an APEG, gray nodes and gray rectangles are the original PEG nodes, dashed circles are equivalence classes, the rectangle with a double outline on the left is an abstraction box. Except for the propagation of the minus operator is done.*

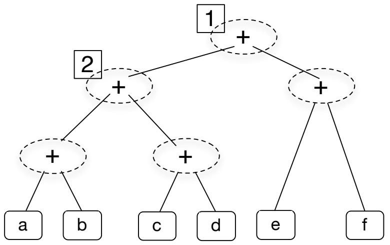

Fig. 4. *Example of an homogeneous APEG.*

The complexity of this algorithm is $O(n)$ where n is the number of leaves, as it only performs one walk through the APEG to find the homogeneous sub-trees. It also adds at most $2(n-1)$ abstraction boxes as it only adds at most 2 abstraction boxes at each equivalence class.

b) Vertical expansion: This algorithm recursively identify the outer nodes of an homogenous sub-tree and inserts an abstraction box containing them. On the example given in Figure III-B, our algorithm adds the following nodes into the equivalence class:

- $(a+b) + \boxed{\text{+, (c,d,e,f)}}$ and $(c+d) + \boxed{\text{+, (a,b,e,f)}}$,
- $a + \boxed{\text{+, (b,c,d,e,f)}}$, $b + \boxed{\text{+, (a,c,d,e,f)}}$, $c + \boxed{\text{+, (a,b,d,e,f)}}$,
  $d + \boxed{\text{+, (a,b,c,e,f)}}$, $e + \boxed{\text{+, (a,b,c,d,f)}}$, $f + \boxed{\text{+, (a,b,c,d,e)}}$.

This algorithm is implemented in $O(n)$ where n is the number of leaves, by performing a search of homogeneous sub-tree starting from the leaves. Then, when the sub-tree stops to be homogeneous it walks through the homogeneous sub-tree found to insert the abstraction boxes. This algorithm adds at most $2(n-1)$ abstraction boxes as it adds (i) one node linking each leaf to an abstraction box containing all the other leaves, and (ii) at most one node linking an abstraction box to each equivalence class. Note that we could add new algorithms to both propagation and expansion steps. The algorithms we use currently form a minimal base to extract an optimized program whose shape is significantly different from the original one.

IV. Profitability Analysis

In this section we give an overview of how our profitability analysis works. Our cost model relies on the number of bits required to encode the integer parts of the numbers, as defined in Section II-B. We use the semantics of Figure 2 to evaluate which expression in an APEG requires the least number of bits to encode the integer part (term $\mu^\sharp$ of the abstract values).

The main difficulty is that it is possible to extract an exponential number of expressions from an APEG. For example, let us consider an operator $*(p_1, p_2)$ where p_1 and p_2 are equivalence classes $p_1 = \langle p'_1, \dots p'_n \rangle$ and $p_2 = \langle p''_1, \dots p''_m \rangle$. Then we have to consider all the expressions $*(p'_i, p''_j)$ for $1 \leq i \leq n$ and $1 \leq j \leq m$. In general, the sub-APEGs contained in p_1 and p_2 may be operations whose operands are again equivalence classes. To cope with this combinatorial explosion, we use a limited depth search strategy. We select the way an expression is evaluated by considering only the best way to evaluate its sub-expressions. This corresponds to a local choice. In our example, synthesizing an expression for $*(p_1, p_2)$ consists of searching the expression $p'_i * p''_j$ whose error is minimal with respect to any $p'_i \in p_1$ and any $p''_j \in p_2$.

For a box $B = \boxed{*, (p_1, \dots, p_n)}$ we use an heuristic which synthesizes an expression limitating the number of digits for the integer part (yet not always optimal). This heuristic is defined as a greedy algorithm which searches at each step the pair p_i and p_j such that the error term carried out by the expression $p_i * p_j$ is minimal. Then p_i and p_j are removed from the box and a new term p_{ij} is added whose measure is equal to the measure of $p_i * p_j$ defined by the equations of Figure 2. This process is repeated until there is only one node left in the box. This last node corresponds to the root of the expression synthesized for the abstraction box.

V. Experimental Results

In this section, we present some experimental results obtained with Sardana and which concern the optimization of commonly used digital filters. These filters are widely used on embedded image processing as they are easy to implement and can also be parallelized easily. Sardana is used to reduce the number of digits required to represent the integer part of the fixed-point number, thus improving either the overall accuracy by allowing the allocation of more digits for the fractional part without any risk of overflow or by improving the size of a circuit in an embedded component.

Our case study concerns the minimization of the number of digits of the integer part of fixed-point numbers involved in the computation of the convolution product corresponding to a digital filter. Most digital filters are defined by a matrix $K = (k_{rl})$ with $1 \leq r \leq n$ and $1 \leq l \leq m$, called the kernel. Formally, the output o of a convolution matrix K of size $n \times m$ applied to a pixel at coordinates (i, j) of a 2-dimensional image I is defined by the following equation.

$$o = \sum_{r=1}^{n} \sum_{l=1}^{m} I(i + r - 1, j + l - 1) \times k_{rl} \tag{14}$$

Sardana minimizes the number of digits required for the integer parts of the numbers by changing the parsing of the sums of products between the kernel values and the pixels intensity of Equation (14). First, let us mention that each filter outputs a value between 0 and 255, therefor the integer part of the output value requires 8 bits. However the intermediary computations of the convolution product does not necessarily require the same sizes. For example, the value -2 requires 2 bits for the integer part, the value 4 requires 3 bits, but the sum of the two terms only requires 2 bits. Finally to perform this operation we need 5 bits for the inputs, and 2 bits for the output, so 7 bits in total. In addition most kernels of digital filter contain symmetries and sometimes opposite values, like the Sobel kernel presented in Table II. Consequently, in order to optimize the number of digits used for the integer part, Sardana may merge the opposite terms sooner in the calculation. As discussed in Section II-B, we denote $Filter_{sum}$ the global number of digits needed to represent all the integer parts of the intermediary results.

Table I presents the results obtained on many usual digital filters. The first column is the type of the filter on which we use the Sardana tool with two different sizes of kernel: 3×3 and 5×5. The $Filter_{sum}$ column presents the cumulated number of digits needed to represent the integer part of the calculation defined by Equation (14) for any values of the input pixels, i.e. when all the intensity of the pixels are defined by the interval $[0, 255]$. Next, the $Filter^{opt}_{sum}$ column is for the same filter but with the optimized equation synthesized by Sardana. Finally the $\%Gain$ column shows the difference between $Filter_{sum}$ and $Filter^{opt}_{sum}$ expressed as a percentage.

	size 3x3				size 5x5		
Filter	$Filter_{sum}$	$Filter^{opt}_{sum}$	$\%Gain$	Avg-IP	$Filter_{sum}$	$Filter^{opt}_{sum}$	$\%Gain$
Gaussian	183	176	3,8%	~ 139	442	393	11%
Laplacian 1	204	180	11,7%	~ 152	713	660	7,4%
Laplacian 2	248	246	0,8%	~ 204	779	723	7,1%
Prewit 1	178	163	8,4%	~ 133	638	569	10,8%
Prewit 2	184	169	8,1%	~ 136	644	571	11,3%
Rehauss 1	259	253	2,3%	~ 212	841	794	5,5%
Rehauss 2	249	242	2,8%	~ 201	724	678	6,3%
Robert 1	263	233	11,4%	~ 161	778	694	10,7%
Robert 2	266	233	12,4%	~ 153	781	694	11,1%
Sobel 1	198	176	11,1%	~ 145	769	678	11,8%
Sobel 2	194	176	9,2%	~ 144	752	678	10,6%

TABLE I
RESULTS ON DIGITAL FILTERS OF SIZE 3X3 AND 5X5.

Our experiments show a significant improvement on most filters by reducing by 10% the number of digits needed for the integer part.

We have also used the optimized expression of each filter on the image of Figure 5, in order to measure precisely how many digits are required to perform the calculation. Contrarily to the results described earlier, we compute here, for each pixel, how many digits are actually used given the optimized equation of a filter and the precise values of intensity of the concerned pixel and the one surrounding it. This corresponds to analyzing the optimized equation of the filter knowing the exact value of the terms $I(i + r - 1, j + l - 1)$ of Equation (14). Then we show in the Avg-IP column the average value for the whole image of the actual number of digits needed by each pixel. We found that the average sum of sizes is usually 25% smaller than the number required to avoid any overflow, for any value of pixel intensity between 0 and 255. These results raises two observations. Firstly, we have an estimation of the loss of precision introduced by the abstraction (i.e. the interval arithmetics). Secondly, these results could point out that the cases where the digits of the integer part are all needed all along the computation may be pathological cases as they do not occur much on this image. In this case, further optimization using Sardana will depend on (i) the definition of the filter and (ii) the knowledge of these pathological cases, in order to describe more accurately the values of intensity of the pixels. However it is important to note that when the value of intensity of the pixels are close to zero their integer part is necessarily smaller. In our first experiment we consider that the pixel intensity are all between 0 and 255, but in Figure 5 the intensity are more close to 0 than 255 because the image is rather dark. This could also explain why we have such difference between the results in the $Filter^{opt}_{sum}$ column and the Avg-IP column.

The results presented above show that Sardana is able to lower the number of digits required for the integer parts, for many convolution products. Nevertheless, the issue of precision is also important and this problem is also addressed by Sardana. For example, the kernel of a Gaussian filter used for a blur effect contains values that cannot be represented exactly neither in a floating-point representation nor in fixed-point representation. Sardana allows the user to set the size of the fractional parts of numbers and then computes the roundoff error on the results, in the worst case, assuming that the input belong to the ranges specified by the user. For several sizes of the fractional part of numbers, Sardana computes the roundoff errors given in Table III for the Gaussian filter mentioned earlier.

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

TABLE II
THE SOBEL OPERATORS USED FOR EDGE DETECTION.

Fig. 5. The image used to evaluate the optimized digital filters.

Fractional part size	Error bound generated
4	$1.2 \cdot 10^2$
6	$2.8 \cdot 10^1$
8	5.1
10	1.9
12	$3.4 \cdot 10^{-1}$
14	$3.1 \cdot 10^{-2}$
16	0.0

TABLE III
EVOLUTION OF THE ABSOLUTE BOUND OF ERROR OF A GAUSSIAN KERNEL OF SIZE 3.

VI. CONCLUSION

In this article we have presented a novel approach to minimize the size of the integer parts of fixed-point numbers while ensuring the absence of overflow. This approach is based on (i) a new representation of program called APEG, which is a tractable abstraction of an exponential number of programs which are all mathematically equivalent to the original one, and (ii) a local search heuristic able to synthesize a new version of the program which requires less digits for the integer parts. This approach has been concretize into a tool called Sardana, which was originally designed to improve the accuracy of floating-point expressions, and have been recently extended to handle the fixed-point arithmetic. The Sardana tool have been used to optimize many digital filters which are commonly used in the image processing field. For each one we have synthesized a new expression of their convolution product that requires less digits for the integer parts, independently of the image on which it is applied to. Our results show that for most filters we are able to reduce by 10% the number of digits used. Also our tool provide informations concerning the roundoff error introduced by each filter depending on the size of the fractional part that have been set.

We are confident that our tool can be useful in many other fields of application which rely on fixed-point arithmetic. And on this matter we intend to improve Sardana in order to perform multi criteria optimization, such as minimizing the overall number of bits used by the integer parts, while improving the accuracy or the latency of the program.

REFERENCES

[1] *IEEE Standard for Binary Floating-point Arithmetic*, Std 754-2008 ed., ANSI/IEEE, 2008.

[2] R. Yates, *Fixed-Point Arithmetic: An Introduction*, Digital Signal Labs, 2009.

[3] P. Cousot and R. Cousot, “Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixed points,” in *Principles of Programming Languages 4*. ACM Press, 1977, pp. 238–252.

[4] ——, “Systematic design of program transformation frameworks by abstract interpretation,” in *Conference Record of the Twentyninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002, pp. 178–190.

[5] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, “Equality saturation: A new approach to optimization,” in *POPL ’09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. ACM, 2009, pp. 264–276.

[6] ——, “Equality saturation: A new approach to optimization,” *Logical Methods in Computer Science*, vol. 7, no. 1, 2011.

[7] D. Boland and G. A. Constantinides, “A scalable approach for automated precision analysis,” in *FPGA*, 2012, pp. 185–194.

[8] T. Hilaire and P. Chevrel, “Sensitivity-based pole and input-output errors of linear filters as indicators of the implementation deterioration in fixed-point context,” *EURASIP Journal of Advanced Signal Processing*, vol. 2011, 2011.

[9] K. Parashar, R. Rocher, D. Menard, and O. Sentieys, “A hierarchical methodology for word-length optimization of signal processing systems,” in *VLSI Design*, 2010, pp. 318–323.

[10] R. Rocher, D. Menard, N. Herve, and Sentieys, “Fixed-point configurable hardware components,” *EURASIP Journal on Embedded Systems (JES)*, vol. 2006, 2006.

[11] R. Rocher, D. Menard, O. Sentieys, and P. Scalart, “Analytical accuracy evaluation of fixed-point systems,” in *EUSIPCO’07*, 2007.

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, 1991.

[13] S.-A.-A. Touati and D. Barthou, “On the decidability of phase ordering problem in optimizing compilation,” in *Proceedings of the 3rd conference on Computing frontiers*, ser. CF ’06. ACM, 2006, pp. 147–156.

[14] C. Mouilleron, “Efficient computation with structured matrices and arithmetic expressions,” Ph.D. dissertation, Université de Lyon – ENS de Lyon, Nov. 2011.

[15] F. S. Roberts and B. A. Tesman, *Applied combinatorics (2. ed.)*. Prentice Hall, 2005.

[16] P. Langlois, M. Martel, and L. Thévenoux, “Accuracy Versus Time: A Case Study with Summation Algorithms,” in *PASCO ’10: Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*. ACM, 2010, pp. 121–130.

[17] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein, “The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages,” in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, 1990, pp. 257–271.

[18] P. Tu and D. Padua, “Efficient building and placing of gating functions,” in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, 1995, pp. 47–55.

[19] J.-D. Choi, R. Cytron, and J. Ferrante, “Automatic construction of sparse data flow evaluation graphs,” in *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL ’91. ACM, 1991, pp. 55–66.

[20] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson, “Practical improvements to the construction and destruction of static single assignment form,” *Software - Practice and Experience*, vol. 28, no. 8, pp. 859–881, Jul 1998.