Sardana: an Automatic Tool for Numerical Accuracy Optimization

Arnault Ioualalen, Matthieu Martel

Synthesizing Accurate Formulas for the Floating-Point Arithmetic

Arnault Ioualalen[1,2,3] Matthieu Martel[1,2,3]

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

[2]Univ. Montpellier II, Laboratoire d'Informatique Robotique et de Microelectronique de Montpellier, UMR 5506, F-34095, Montpellier, France

[3]CNRS, Laboratoire d'Informatique Robotique et de Microelectronique de Montpellier, UMR 5506, F-34095, Montpellier, France

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

Abstract

Many critical embedded systems perform floating-point computations yet their accuracy is difficult to assert and strongly depending on how formulas are written in programs. In this article, we focus on the synthesis of accurate formulas mathematically equal to the original formulas occurring in source codes. In general, an expression may be rewritten in many ways using associativity, distributivity, etc. To consider 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. In this article, we specifically address the problem of selecting an accurate formula among all the expressions of an APEG. This is necessary to synthesize an expression which minimizes the roundoff errors in the worst case. To validate our approach, we present experimental results showing how APEGs, combined with profitability analysis, make it possible to improve significantly the accuracy of floating-point computations.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Program Verification—Validation; D.3.4 [*Programming Languages*]: Processors—Optimization; F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages—Program Analysis; G.1.0 [*Mathematics of Computing*]: Numerical Analysis—Computer Arithmetic

General Terms Algorithms, Languages, Theory, Verification

Keywords Program Transformation, Numerical Precision, Abstract Interpretation, Compiler Optimizations, Floating-Point Numbers

1. Introduction

The most critical control systems of recent planes, spacial vehicules or nuclear powerplants rely on floating-point computations [1] yet this arithmetic is not intuitive [18, 21]. Indeed, it is very difficult (and almost impossible) to predict by hand the accuracy of the evaluation of a formula, given certain ranges for the inputs. Recently, static analysis techniques have been developed to infer safe error bounds for the computations arising in critical programs written in C [9–11]. However these techniques do not indicate how to improve the accuracy if the infered error bounds are not satisfying. Furthermore, given several mathematically equivalent expressions, it is still a mess to guess which one is the most accurate for our ranges of inputs.

Our work concerns the synthesis at compile-time of accurate formulas, for given input ranges, to replace the expressions written by the programmers in source codes [16, 17]. We consider that a program would return an exact result if the computations were carried out using real numbers. In practice, roundoff errors arise during the execution and these errors are closely related to the way formulas are written. Our approach is based on abstract interpretation [6, 7]. We build Abstract Program Equivalence Graphs (APEGs) to represent in polynomial size an exponential number of mathematically equivalent expressions [].APEGs are abstractions of the Equivalence Program Expression Graphs introduced in [24, 25]. The concretization of an APEG yields expressions of very different shapes and accuracies. Then, to synthesize expressions from APEGs, we perform a profitability analysis which consists of searching the most accurate concrete expression among the set of expressions represented in an APEG.

This article mainly focuses on our profitability analysis. An APEG is an abstraction of an exponential number of expressions and the profitability has to extract an accurate formula. We compute safe error bounds using established static analysis techniques for numerical accuracy [10, 15] and we use a limited depth search algorithm with memoization to explore the APEG structure. In addition, APEGs contain abstraction boxes representing any parsing of a sequence of operations defined by a set of operands and an unique commutative operator (e.g. $\boxed{+(a_1,\ldots,a_n)}$ denotes any way to compute $\sum_{i=1}^{n} a_i$). We define a way to synthesize an accurate formula for an abstraction box.

For tractability reasons, we require our profitability analysis to be polynomial in the size of the APEGs. The APEGs representing an exponential number of expressions, our profitability is then a heuristic and we present experimental results to assert its efficiency. Our techniques have been implemented in a tool, named Sardana which takes as entry synchronous programs written in Lustre [5, 12]. Inputs are represented by abstract streams which indicate, at each instant, a range for the values of the variables and a range for the roundoff errors on these variables. We present experi-

mental results showing how our techniques improve the numerical formulas arising in two avionic applications: the first one is an autopilot open-source software, Paparazzi, developed by the DRONES laboratory at the French civil aviation university (ENAC). The second case study is a sequence of relevant pieces of codes extracted from an industrial critical software coming from aeronautics.

To our knowledge, there is no other work which compares directly to ours, concerning the synthesis of numerically accurate expressions for the floating-point arithmetic, at compile-time, for specified input ranges. 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 [3, 22]. 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 fixed-point arithmetic as discussed in [].

The rest of this article is organized as follows. In Section 2, we briefly recall elements of the floating-point arithmetic and presents EPEGs [24]. Section 3 introduces our abstract representation, called APEG. Section 4 concerns the exploration of APEGs in order to synthesize a new program which lowers the roundoff errors. Sections 5 summarizes the experiments performed with our tool Sardana. Finally, Section 6 concludes.

2. Background

This section briefly surveys the aspects of floating-point arithmetic useful to the comprehension of the rest of this article. Then we introduce PEGs [24, 25] which are a new kind of intermediate representation for compilers. PEGs are compared to the SSA form. Abstract PEGs are defined in Section 3.

2.1 Floating-Point Arithmetic

The IEEE754 Standard specifies the representation of numbers and the semantics of the elementary operations for floating-point arithmetic [1, 18, 21]. First of all, a floating-point number x in base β is defined by

$$x = s \cdot (d_0.d_1 \dots d_{p-1}) \cdot \beta^e = s \cdot m \cdot \beta^{e-p+1} \tag{1}$$

where $s \in \{-1, 1\}$ is the sign, $m = d_0 d_1 \dots d_{p-1}$ is the mantissa with digits $0 \leq d_i < \beta$, $0 \leq i \leq p-1$, p is the precision and e is the exponent, $e_{min} \leq e \leq e_{max}$. A floating-point number x is normalized whenever $d_0 \neq 0$. Normalization avoids multiple representations of the same number. IEEE754 Standard introduces a few values for p, e_{min} and e_{max}. For example, single precision numbers are defined by $\beta = 2$, $p = 23$, $e_{min} = -126$ and $e_{max} = +127$. The IEEE754 Standard also specifies special values (denormalized numbers, infinites and NaN) which are not used in this article.

Let $\uparrow_\circ : \mathbb{R} \to \mathbb{F}$ be the function which returns the roundoff of a real number following the rounding mode $\circ \in \{\circ_{-\infty}, \circ_{+\infty}, \circ_0, \circ_\sim\}$ (towards $\pm\infty$, 0 or to the nearest). $\uparrow_\circ$ is fully specified by the IEEE754 Standard which also requires, for any elementary operation $\Diamond$, that:

$$x_1 \Diamond_{\mathbb{F},\circ} x_2 = \uparrow_\circ (x_1 \Diamond_{\mathbb{R}} x_2) \tag{2}$$

Equation (2) states that the result of an operation between floating-point numbers is the roundoff of the exact result of this operation. In this article, we also use the function $\downarrow_\circ : \mathbb{R} \to \mathbb{R}$ which returns the roundoff error. We have:

$$\downarrow_\circ (r) = r - \uparrow_\circ (r) \tag{3}$$

Enhancing the quality of the implementation of a formula $f(\mathbf{x})$ then consists of minimizing the roundoff error on the result. In other words, using the notation of Equation (3), we aim at minimizing $\downarrow_\circ (f(\mathbf{x}))$, for all the possible vectors of inputs $\mathbf{x}$.

2.2 EPEGs and Phase Ordering

Nowadays, most compilers implement a lot of optimizations such as loop unfolding, vectorization, dead code elimination, etc. Most of these algorithms are easily to implemented if the compiler first transforms its intermediate representation (IR) of the programs into one of the SSA forms [8]. However, the order of application of these optimizations is often unclear, the user only deciding which optimizations having to be applied by setting compiler flags. In practice, the optimizations are applied in a sequential order which is determined statically by some internal heuristics of 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 [26].
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 [24], 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. In the following, we will show how EPEGs are defined and built directly from the control flow graph. We will also show how EPEGs allow the compilers to select a new version of a program while considering all its optimizations together.

2.3 EPEG Construction

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 1). 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 it is impossible to use any of the equalities, i.e. when 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

$P^n(x) = \sum_{k=0}^{n}(-1)^k \times \binom{n}{k} \times x^k$, 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 [19, §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 [23].

To summarize, EPEGs are able to cope with the phase ordering problem as they merge all optimizations, but they raise new issues concerning tractability that our new IR, called APEG, addresses (see Section 3).

Next, 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 [14], etc.) This step is referred to as the *profitability* phase and is it performed with a global profitability heuristic. The profitability phase has two major issues: (i) it must extract a well-formed program, and (ii) it must rely on an efficient and tractable cost model to meet the user requirements. In [24], the first issue is solved by associating to each node n and each equivalence class a variable $B(n)$ which takes only the value 0 (if the node is not in the extracted program) or 1 (if the node belongs to the extracted program). These variables encode the constraints that must hold to ensure that the extracted program is well-formed. For example, when the variable of a node is set to 1 then all the variable of its child nodes must also be set to 1. The second issue is addressed by the authors with the following cost model. The cost $C(n)$ of a node n is defined as $\mathit{basic\_cost}(n) \times k^{\mathit{depth}(n)}$, where $\mathit{basic\_cost}(n)$ accounts for how expensive is the node n itself, and $k^{\mathit{depth}(n)}$ 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. In this article, we present in Section 4 a completely different profitability that we use to improve the numerical accuracy of programs. This profitability performs a limited depth search to combine multiple equivalence classes in order to evaluate more precisely the worst accuracy of the expression starting at a given node.

3. APEG Construction

In this section, we present our new intermediate representation of programs, called Abstract Program Expansion Graph (APEG). This intermediate representation is inspired from the EPEGs introduced in Section 2. However, as an EPEG is not necessarily complete, 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 [19, §6.3], where n in the number of operands of the abstraction box.

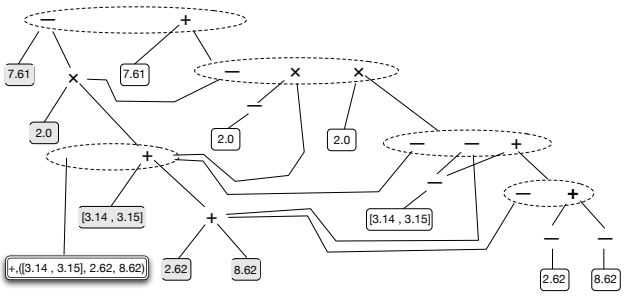

Figure 1. *Example of an APEG, gray nodes and gray rectangles are the original PEG nodes, dashed circles are equivalence classes, the rectangle with a double outline on the left is an abstraction box. Except for the abstraction box present, this APEG illustrates how the propagation of the minus operator is done.*

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 1 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:

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 2 our algorithm adds the following nodes into the APEG :

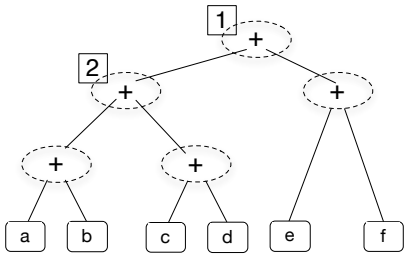

Figure 2. *Example of an homogeneous APEG.*

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

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.

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 2, our algorithm adds the following nodes into the equivalence class 1 :

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

4. Profitability Analysis

The profitability corresponds to the process of synthesizing a new version of a program from an APEG. Our tool, Sardana, takes as inputs a Lustre program together with ranges for the inputs and it synthesizes a new program with better numerical accuracy. This section is then organized as follows. Section 4.1 introduces how we compute the propagatation of roundoff errors in floating-point computations. Section 4.2 describes our abstract representation of the streams of values manipulated by the Lustre programs. Finally, in Section 4.3, we present our method to synthesize from an APEG a new program whose accuracy is better than the original one.

4.1 Analysis of Floating-Point Computations

To compute safe bounds on the numerical accuracy of arithmetic expressions, we use abstract values $(x^\sharp, \mu^\sharp) \in \mathbb{E}^\sharp$ where $x^\sharp$ and $\mu^\sharp$ are intervals whose bounds are floating-point numbers, and where $x^\sharp$ represents the interval of values of the input and $\mu^\sharp$ represents the intervals of errors on the input [17]. A value $(x^\sharp, \mu^\sharp)$ abstracts a set of concrete values $\{(x_i, \mu_i),\ i \in I\}$ by intervals in a component-wise way.

For an arithmetic expression, the propagation of roundoff errors corresponds to the semantics of [15, 17] and is given in Figure 3. The abstract function $\uparrow^\sharp_\circ$ corresponds to the concrete function $\uparrow_\circ$ introduced in Section 2.1. We have:

$$\uparrow^\sharp_\circ ([\underline{x}, \overline{x}]) = [\uparrow_{-\infty} (\underline{x}), \uparrow_{+\infty} (\overline{x})] \tag{7}$$

The function $\downarrow^\sharp_\circ$ is a safe abstraction of $\downarrow_\circ$, i.e. $\forall x \in [\underline{x}, \overline{x}]$, $\downarrow_\circ (x) \in \downarrow^\sharp_\circ ([\underline{x}, \overline{x}])$. For example, if the current rounding mode $\circ$ is to the nearest, one may choose

$$\downarrow^\sharp_\circ ([\underline{x}, \overline{x}]) = [-y, y] \quad \text{with } y = \frac{1}{2}\text{ulp}\big(\max(|\underline{x}|, |\overline{x}|)\big) \tag{8}$$

where the *u*nit in the *l*ast *p*lace ulp(x) is the weight of the least significant digit of the floating-point number x [20]. For an addition, the errors on the operands are added to the error due to the roundoff of the result, as specified by Equation (2). For a subtraction, the errors on the operands are subtracted. Finally, the semantics of the multiplication comes from the development of $(x_1^\sharp + \mu_1^\sharp) \times (x_2^\sharp + \mu_2^\sharp)$. The semantics of other operations is described in [15, 17]. We use the former arithmetic for the elementary operations between floating-point values. In Lustre, values are streams and the operations have to be extended to streams, as developed in the following section.

4.2 Abstract Streams of Values

As a synchronous language, Lustre [5, 12] handles streams of values recording the values of a program point at each time instant. For example, let us consider the following program:

```
a = 1.0 / 3.0;
b = 1.0 -> a * pre(b);
```

A constant stream mapping each instant to the internal representation of $\frac{1}{3}$ is associated to `a`. The stream `b` is the stream whose value is $b(0) = 1.0$ at time 0 and whose value at time i is $b(i) = a(i) \times b(i-1)$.

Following the notations of [4], we denote $s = \langle m, t_0 : x_0 \wedge t_1 : x_1 \wedge \ldots \wedge t_N : x_N \rangle$ the stream s such that, at each instant $i \mod m$, $s(i) = x_k$ if $t_k \leq i \mod m < t_{k+1}$ for some $k \in [0, N-1]$ or $s(i) = x_N$ if $t_N \leq i \mod m < m$. By extension, we consider that if $m = 0$ then no modulo holds and $\forall i \geq t_N$, $s(i) = x_N$. We also assume that always $t_0 = 0$.

Our streams associate at each instant a value in the domain $\mathbb{E}$ of intervals of floating-point numbers with errors introduced in Section 4.1. Hence, a stream $s \in \mathbb{S}$ is a mapping $s : \mathbb{N} \to \mathbb{E}$ where $\mathbb{N}$ denotes the set of non-negative integers. Coming back to our former example, the stream associated to `a` is

$$\begin{array}{lll} \langle 0, & 0: & ([3.33333333333325E-1, 3.33333333333334E-1], \\ & & [1.85037170770855E-17, 1.85037170770861E-17])\rangle \end{array}$$

and the stream associated to `b` is

$$\begin{array}{lll} \langle 5, & 0: & ([2.09075158128704E-7, 1.0], \\ & & [-6.49870810424242E-23, 3.89955157246792E-22])\wedge \\ & 1: & [2.09075158128704E-7, 3.33333333333334E-1], \\ & & [-2.77556273678202E-17, 4.62596578069798E-17])\wedge \\ & 2: & [2.09075158128704E-7, 1.11111111111112E-1], \\ & & [-1.61907865481218E-17, 2.85268996889166E-17])\wedge \\ & 3: & [2.09075158128704E-7, 3.70370370370373E-2], \\ & & [-8.86638677409539E-18, 1.50345826165255E-17])\wedge \\ & 4: & [2.09075158128704E-7, 1.23456790123458E-2], \\ & & [-3.82282320444157E-18, 6.56439316733447E-18]) \end{array}$$

Elementary operations are applied at each instant, i.e if $s = \langle m, 0 : x_0 \wedge t_1 : x_1 \wedge \ldots \wedge t_N : x_N \rangle$ and $s' = \langle m', 0 : x'_0 \wedge t'_1 : x'_1 \wedge \ldots \wedge t'_{N'} : x'_N \rangle$ then, for an operation $* \in \{+, -, \times\}$, we have $(s * s')(i \mod k) = s(i \mod m) * s'(i \mod m')$ where k is the least common multiple of m and m'.

$$(x_1^\sharp, \mu_1^\sharp) + (x_2^\sharp, \mu_2^\sharp) = \left(\uparrow_\circ^\sharp (x_1^\sharp + x_2^\sharp), \mu_1^\sharp + \mu_2^\sharp + \downarrow_\circ^\sharp (x_1^\sharp + x_2^\sharp)\right) \quad (4)$$

$$(x_1^\sharp, \mu_1^\sharp) - (x_2^\sharp, \mu_2^\sharp) = \left(\uparrow_\circ^\sharp (x_1^\sharp - x_2^\sharp), \mu_1^\sharp - \mu_2^\sharp + \downarrow_\circ^\sharp (x_1^\sharp - x_2^\sharp)\right) \quad (5)$$

$$(x_1^\sharp, \mu_1^\sharp) \times (x_2^\sharp, \mu_2^\sharp) = \left(\uparrow_\circ^\sharp (x_1^\sharp \times x_2^\sharp), x_1^\sharp \times \mu_2^\sharp + x_2^\sharp \times \mu_1^\sharp + \mu_1^\sharp \times \mu_2^\sharp + \downarrow_\circ^\sharp (x_1^\sharp \times x_2^\sharp)\right) \quad (6)$$

Figure 3. *Abstract semantics of the elementary operations for the floating-point arithmetic.*

The profitability has to search an expression which minimizes the roundoff errors. Our values being streams of abstract floating-point values with errors, we have to define in what sense we aim at minimizing the errors. First, let us introduce some notations. For a value $v = (x^\sharp, \mu^\sharp) \in \mathbb{E}$, let $\mathcal{E}(v) = \mu^\sharp$. The function $\mathcal{E}$ gives the error term of an abstract value. Next, if $\mathcal{E}(v) = \mu^\sharp = [\underline{\mu}, \overline{\mu}]$, then $M_\mathcal{E}^+(v) = \overline{\mu}$, $M_\mathcal{E}^-(v) = \underline{\mu}$, and $M_\mathcal{E}(v) = \max(|\underline{\mu}|, |\overline{\mu}|)$. In other words, $M_\mathcal{E}^+(v)$, $M_\mathcal{E}^-(v)$ and $M_\mathcal{E}(v)$ denote the upper and lower bounds of the error and the maximal absolute error bound, respectively. We consider several orders:

- Strict order: $s \prec_s s'$ if $\forall i \in \mathbb{N},\ \mathcal{E}(s(i)) \subseteq \mathcal{E}(s'(i))$. This order requires that, if $s \prec_s s'$ then at any time the error bound on s is less than the error bound on s'. The accuracy is improved at each instant.
- Max order: $s \prec_m s'$ if $\max_{i\in\mathbb{N}} M_\mathcal{E}(s(i)) \leq M_\mathcal{E}(s'(i))$. This order only considers the worst errors. We have $s \prec_m s'$ if the worst intensity of the error possibly associated to a value at a given instant is always smaller in s than is s',. Elsewhere, the errors may be greater in s' than in s.
- Integral order: $s \prec_i s'$ if $\sum_{i=0}^{i=m-1} M_\mathcal{E}^+(s(i)) - M_\mathcal{E}^-(s(i)) \leq \sum_{i=0}^{i=m-1} M_\mathcal{E}^+(s'(i)) - M_\mathcal{E}^-(s'(i))$. This order compares the integrals of the error functions $\mathcal{E}(s)$ and $\mathcal{E}(s')$. We have $s \prec_i s'$ if the sums of the errors at each instant is smaller in s than in s'. If $s \prec_i s'$ then at some instant i, the error $s(i)$ may be greater than the error $s'(i)$. If $m = 0$ or $m' = 0$, the integral is computed up to $\max(m, m') - 1$. If $m = 0$ and $m' = 0$, the integral is computed up to $max(t_N, t'_{N'})$.

The Strict order $\prec_s$ would require to synthesize a program which always improves the error bounds on the computed values. We consider that this order is too restrictive and we do not use it in practice. The Max order$\prec_m$ may be interesting in certain applicative contexts where the main objective is to lower the worst error bound and where the average error is not relevant. The Integral order $\prec_i$ gives a measure of the average error. We consider this order as the most interesting. In practice, our experiments confirm that the integral order is the order for which me may optimize the most the programs.

4.3 Profitability of APEGs

To synthesize an optimized program, the profitability has to be performed on the APEGs as defined in Section 3. A main difficulty is that, thanks to equivalence classes, an APEG may represent an exponential number of expressions whose accuracy should be individually evaluated. 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 should 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 equivalence classes again. For example, we may have $p'_1 = *(q_1, q_2)$ with $q_1 = \langle q'_1, \dots q'_r \rangle$ and $q_2 = \langle q''_1, \dots q''_s \rangle$ and we should consider all the expressions $*(*(q_u, q_v), p''_j)$ for $1 \leq i \leq n$, $1 \leq j \leq m$, $1 \leq u \leq r$ and $1 \leq v \leq s$. Hence, for tractability reasons, we shall not require the profitability to be exhaustive.

To cope with this combinatorial explosion, we use a limited depth search strategy with memoization. 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 former example, for a depth equal to 1, this consists of considering only the optimal expression for p'_1 (let us say $*(q'_3, q''_5)$) when examining $*(p_1, p_2)$. The other ways to evaluate p'_1, i.e. $*(q'_1, q''_1)$, $*(q'_1, q''_2)$, ... are no longer considered even if, once combined to p_2, they could yield a better result. At depth 2, we would consider all the terms $*(q'_i, q''_j)$ when synthesizing the expression $*(p_1, p_2)$ but not their sub-expressions.

Algorithm 4.3 illustrates how we perform the profitability when the depth is set to 1. The operator :: appends a value to a list. Algorithm 4.3 considers, for each node in the given equivalence class, the cartesian product of the elements of the node. It uses the equivalence classes of the node parameters and evaluates the roundoff error generated by the operator using the semantics introduced in Section 4.2. Then it returns the minimal roundoff error for the desired order ($\prec_s$, $\prec_m$ or $\prec_i$ as discussed in Section 4.2) after memoization of the accuracies of the expressions encountered.

For the sake of conciseness ,Algorithm 4.3 does not detail how to snthesize the final expression once we have found the minimal stream. This step is a simple propagation of the expressions along with the streams.

The next point concerns the synthesis of an expression for an abstraction box $B = \boxed{*, (p_1, \dots, p_n)}$. In this case, the profitability applies an heuristic which generates an accurate expression (yet not always optimal). This heuristic is a greedy algorithm which searches at each step the pair p_i and p_j such that $\downarrow_\circ^\sharp (p_i * p_j)$ is minimal. Once it founds p_i and p_j it replaces both terms in the box by a new term p_{ij} whose accuracy is equal to $\downarrow_\circ^\sharp (p_i * p_j)$. This heuristic performs at most n^2 pairing at each step, and is repeated exactly $n - 1$ times. Thus it computes in $O(n^3)$ iterations.

5. Case studies

In this section we present experimental results obtained on several cases studies with our tool, Sardana, which implements APEGs and the profitability heuristic described in Section 4. We illustrate our approach on Taylor series and two real case studies: the first one is an altitude estimator from an open-source project called Paparazzi, the second study concerns several pieces of code taken from a real industrial avionic embedded systems.

5.1 Transformation of Taylor Series

In several contexts programmers approximate elementary functions such as logarithms or trigonometric functions by polynomials obtained by means of a Taylor series development. However the evaluation of a Taylor series in the floating-point arithmetic is subject to accuracy losses around one of the roots of the approximant polynomial [13] and because the exact coefficients cannot be represented exactly in the floating-point arithmetic. Interestingly, if we evaluate a Taylor series near a value that anneals the approximated function,

Algorithm 1 Profitability of an equivalence class $\langle p_1, \ldots, p_n \rangle$, depth of search is set to to 1

$E \leftarrow [\,]$
for $p_i \in \langle p1, \ldots, p_n \rangle$ **do**
  if $p_i = \langle p'_1, \ldots, p\prime_k \rangle * \langle q''_1, \ldots, q''_m \rangle$ **then**
    for each $p'_j \in \langle p'_1, \ldots, p\prime_k \rangle$ **do**
      for each $p''_k \in \langle q''_1, \ldots, q''_m \rangle$ **do**
        $E \leftarrow (\downarrow^\sharp_\circ (p'_j * p''_k)) :: E$
      end for
    end for
  else if $p_i = l * \langle q_1, \ldots, q_m \rangle$ **or** $p_i = \langle q_1, \ldots, q_m \rangle * l$ **then**
    for $q_j \in \langle q_1, \ldots, q_m \rangle$ **do**
      $E \leftarrow (\downarrow^\sharp_\circ (l * q_j)) :: E$
    end for
  else
    //$p_i = l_1 * l_2$
    $E \leftarrow (\downarrow^\sharp_\circ (l_1 * l_2)) :: E$
  end if
end for
return Min(E)

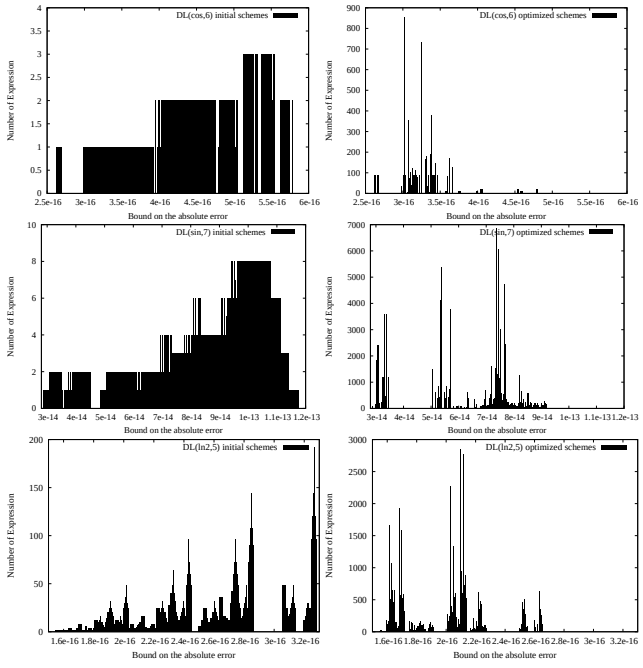

Figure 4. *Results for the expansion in Taylor series of cosine (at 6^{th} order), sine (at 7^{th} order) and $ln(2+x)$ (at 5^{th} order). Left part is the initial error distributions, right part is for the optimized schemes.*

less numerical errors arise for a lower order expansion than for a higher order one as, in the latter case, the polynomial resulting from the Taylor expansion has a root closer to the value that anneals the function. Therefore there is an implicit tradeoff between having a better accuracy everywhere except near the roots or been more accurate close to the roots.

We intend to show that our techniques are able to improve the numerical accuracy close to the roots of the approximated function even when the order of the expansion increases. We have performed benchmarks on the expansion into Taylor series of the following functions: cosine, sine[1] and $ln(2+x)^2$ for a few orders of expansion. All the experimental results presented in this section have been achieved using the IEEE-754 binary 64 format. Our benchmarks are done exhaustively, by generating all the possible evaluation schemes for each Taylor expansion according to [19, §6.2.2].

Figure 4 presents the error bounds found for the evaluation near a root of the Taylor expansion of cosine at the 6^{th} order (which corresponds to 15.924 parsings), the Taylor expansion of sine at the 7^{th} order (235.270 parsings) and the Taylor expansion of $ln(2+x)$ at the 5^{th} order (323.810 parsings). For each of these functions, Figure 4 summarizes on the left-hand-side both the initial error bounds and, on the right-hand-side, the optimized error bounds.

First, let us examine the results of the Taylor expansion of cosine (first line of Figure 4). We can see that the initial error bounds of cosine are almost linearly scattered between the values $2.5 \cdot 10^{-16}$ and $6.0 \cdot 10^{-16}$, i.e. the number of expressions with the same error bound grows linearly with the error. However our transformation is able to shift all of the error bounds between the values $2.5 \cdot 10^{-16}$ and $4.8 \cdot 10^{-16}$, with most of below $3.7 \cdot 10^{-16}$. This corresponds to an improvement of the worst case accuracy by 20% at least. If we look to the improvement of each expression we obtained an average improvement of 17%.

For the Taylor expansion of the sine function (second line of Figure 4), we can see that the initial error bounds are between $3 \cdot 10^{-14}$ and $1.2 \cdot 10^{-13}$, and most of them are near the worst error bound. We can see that our analysis has regrouped the initial error bounds near three areas: the optimal value, the value $5.5 \cdot 10^{-14}$ and the value $7.5 \cdot 10^{-14}$. Also, the optimized error bound are all inferior to $1 \cdot 10^{-13}$, which corresponds to an improvement of the worst case accuracy by 17% at least. If we look to the improvement of each expression we obtained an average improvement of 28%.

Finally we present the results obtained for the Taylor expansion of $ln(2+x)$ (third line of Figure 4). We can see that initially the number of expressions with the same value of error bound grows linearly with the errors, and are all between the values $1.6 \cdot 10^{-16}$ and $3.2 \cdot 10^{-16}$. Here our analysis has regrouped the initial error bounds near three areas: the value $1.7 \cdot 10^{-16}$, the value $2.1 \cdot 10^{-14}$ and the value $2.7 \cdot 10^{-14}$. Therefor we have improved the worst case accuracy by 16% at least. If we look to the improvement of each expression we obtained an average improvement of 23%.

5.2 Embedded Altitude Estimator

In this section we illustrate our approach on a Kalman filtering code used on an unmanned airborne system. This filter is used in the Paparazzi project[3] developed by the DRONES laboratory at the French civil aviation university (ENAC). Paparazzi is an open-source project which intends to create an auto-pilot system for fixed-wing aircrafts as well as for multicopters. This filter is used to predict the altitude estimation produced by the aircraft using GPS coordinates.

Contrarily to a GPS used on grounded vehicles, which only operate with longitude and latitude, aircrafts also uses the altitude provided by the GPS. However this last coordinate is usually less precise than the former. Combined to the slow rate of values produced GPS, a real-time auto-pilot must constantly predict and then correct its predictions when the actual values arrive.

The filter uses a covariance matrix P of size 3, constructed by means of Algorithm 2, where initially $P[i][j] = 10$ if $i = j$ and $P[i][j] = 0$ otherwise. The terms $\Delta t, C_1, C_2, C_3$ are constants.

[1] $\cos x = \sum_{n=0}^{+\infty} (-1)^n \frac{x^{2\,n}}{(2\,n)!}$ and $\sin x = \sum_{n=0}^{+\infty} (-1)^n \frac{x^{2\,n+1}}{(2\,n+1)!}$

[2] $ln(2+x) = \sum_{n=1}^{+\infty} (-1)^{n-1} \frac{x^n}{n \times 2^n}$.

[3] http://paparazzi.enac.fr

Intuitively the first row represents the confidence on the current altitude, the second row represents the confidence on the current vertical speed of the aircraft, and the third row represents the confidence on the current vertical acceleration of the aircraft. This covariance matrix is defined such as its terms slowly grow, in order to model the fact that the confidence decreases over time. The model used to infer this confidence assumes that: the vertical acceleration of an aircraft is linear, thus it deduces the vertical speed by integration of the acceleration and the altitude by integration of the speed. Algorithm 2 describes how this covariance matrix is updated at each step using the previous state of the matrix. Note that the calculation performed in Algorithm 2 has been already manually transformed by us to its simplest form.

Our purpose is to illustrate how the choice of the order Strict, Max or Integral on the streams of value, as introduced in section 4.2, impacts the result of our optimization process. We have used the three orders on a scenario where the covariance matrix is updated 35 times using Algorithm 2. We consider this scenario as realistic because the covariance matrix is usually updated dozens of times before a new value is produced by the GPS. Note that each of these results have been obtained in less than $20s$ on a laptop computer, and all the calculations are carried out in the IEEE-754 binary 32 format [1] accordingly to the specification of the Paparazzi project. In addition, let us remark that only the cells $P[0][0]$, $P[0][1]$, $P[1][0]$ and $P[1][1]$ updated by Algorithm 2 are candidate to significant optimizations as their expression is complex enough to allow some transformations.

In this case study, the choice of the order has two consequences on the optimized expression of the filter. First, depending on the order the significant cells are not always optimized. Second, even if a cell is optimized by two different orders, they are not optimized in the same way and consequently the errors bounds are not the same. Our results show that the Strict order is able to optimize all the significant cells ($P[0][0]$, $P[0][1]$, $P[1][0]$ and $P[1][1]$), the Max order optimizes only the cells $P[0][0]$, $P[1][0]$ and $P[1][1]$, and the Integral order optimizes only the cells $P[0][0]$, $P[0][1]$ and $P[1][0]$.

As $P[0][0]$ is optimized for all orders, let us show how the optimizations are different from one order to another. Figures 5 and 6 show the evolution of the error bound on $P[0][0]$ along the scenario using respectively the orders Max, Strict and Integral. These figures represent on the x-axis the instants where the error stream is defined, and on the y-axis the values of both the upper and lower bounds of the error attached to the concerned expression. Note that, in general, these two bounds are close and seem to coincide on the figures. The different orders produce the following expressions from the definition of $P[0][0]$:

- Strict order:

  $P[0][0]+((\Delta t\times((P[1][1]\times\Delta t)+(P[0][1]+P[1][0])))+C_1)$

- Max order:

  $(P[0][0]+(\Delta t\times((P[1][1]\times\Delta t)+(P[0][1]+P[1][0]))))+C_1$

- Integral order:

  $(P[0][0]+(\Delta t\times(P[0][1]+((P[1][1]\times\Delta t)+P[1][0]))))+C_1$

We can observe that for the Strict order the term $P[0][0]$ is added to the rest of the expression at the very end but that for the Max and Strict orders it is the constant C_1 which is add at the end of the calculation. Also, with the Max order Sardana recommends to first add together the terms $P[0][1]$ and $P[1][0]$, and then to add the result to the term $P[1][1]\times\Delta t$. On the contrary, with the Integral order, Sardana recommends to add the terms $P[1][1]\times\Delta t$ and $P[1][0]$ first and then to add the result to $P[1][0]$. This well illustrates how the floating-point arithmetic is not intuitive, because slight modifications, that an human expert may not foresee easily, lead to different improvements of the evolution of the error bounds, as it is illustrated by Figures 5 and 6.

Algorithm 2 Kalman filter for the altitude prediction. ($\overset{+}{\leftarrow}$ operator corresponds to the "add and store" operation.)

$P[0][0] \overset{+}{\leftarrow} (\Delta t \times (P[1][0] + P[0][1] + \Delta t \times P[1][1])) + C_1$
$P[0][1] \overset{+}{\leftarrow} \Delta t \times (P[1][1] - P[0][2] - \Delta t \times P[1][2])$
$P[0][2] \overset{+}{\leftarrow} \Delta t \times (P[1][2])$
$P[1][0] \overset{+}{\leftarrow} \Delta t \times (-P[2][0] + P[1][1] - \Delta t \times P[2][1])$
$P[1][1] \overset{+}{\leftarrow} (\Delta t \times (-P[2][1] - P[1][2] + \Delta t \times P[2][2])) + C_2$
$P[1][2] \overset{+}{\leftarrow} \Delta t \times (-P[2][2])$
$P[2][0] \overset{+}{\leftarrow} \Delta t \times (P[2][1])$
$P[2][1] \overset{+}{\leftarrow} \Delta t \times (-P[2][2])$
$P[2][2] \overset{+}{\leftarrow} C_3$

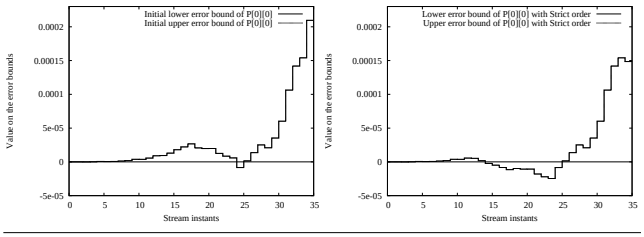

Figure 5. *On the left hand side, the evolution of the initial error bounds of $P[0][0]$. On the right hand side the evolution of the error bounds of the optimized expression of $P[0][0]$ using Strict order.*

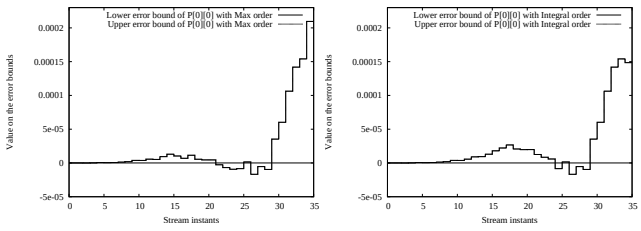

Figure 6. *On the left hand side, the evolution of the error bounds of the optimized expression of $P[0][0]$ using Max order. On the right hand side, the evolution of the error bounds of the optimized expression of $P[0][0]$ using Integral order.*

5.3 Avionic Benchmarks

We present in this section several experimental results obtained on pieces of code extracted from embedded critical industrial avionic codes. These experimental results concern in one hand programs using the IEEE-754 binary 32 format (Table 1), and, on the other hand, programs using the IEEE-754 binary 64 format (Table 2). For both floating-point formats, we have tested each program for many contexts, each having its own input values. Again, input values are described by means of streams of intervals of floating-point numbers. All the contexts we used have been kindly provided by the ASTRÉE team [2] who currently analyzes these programs and have access to realistic simulations of them. For industrial property reasons, we denote the programs used by the following generic terms:

- Interpol, stands for a first order interpolation,
- L-P filter, stands for a low-pass filter of the first order,
- H-P filter, stands for a high-pass filter of the first order,
- Transfer, stands for a transfer function of the second order between two inputs,
- Sqrt, stands for a polynomial interpolation of the square root.

Note that even if some programs appear in both tables 1 and 2, these programs have different codes and performances, yet they aim at achieving a similar task.
Note also that, in this case study, only the Integral order is used.

To perform our benchmarks, we used for each program and each context three identical scenarii in which Sardana computes streams of size 5, 15 or 35 instants. For all the contexts of each program, we present in Table 1 and 2 the improvement of the integral value of the errors for these scenarii.

We can draw two conclusions with these experiments. First, in many cases, the gain in accuracy decreases as the length of the streams increases. This can be observed on all low-pass filters described here. We explain this phenomenon by the fact that in most of these cases the values generated by the program tend towards zero and the errors on these values then decreases as well over time. Thus, the gain at each step of the integral tends to become smaller and smaller leading the integral gain to decrease as well. This conclusion is even more accurate to us, that in the other cases where the values are growing without limit (like with the high-pass filter in Table 1 or the Transfer program in Table 2) the gain on the integral value increases as the stream length increases.
The second conclusion is that our approach is able to improve the overall numerical accuracy of very different programs in many different contexts. This gain is for most of the considered programs between 2% and 20% for 50% to 90% of the contexts. For the few programs we are unable to improve such as Interpol in Table 1, we may argue that the way these programs are written does not allow much transformation on it. Therefor we are not able to synthesize new programs with significant difference in numerical accuracy, as there is not much to transform available.

Program	#Contexts	%Optimized	#Integral from		
			0 to 5	0 to 15	0 to 35
Interpol	2135	3.5%	0.4%	0.4%	0.4%
L-P filter 1	32	96.9%	13.5%	6.7%	3.1%
L-P filter 2	501	57%	6.6%	3.1%	1.3%
H-P filter	414	80.9%	11.7%	23%	25.6%
Transfer	477	100%	15.9%	18.4%	20.2%
			% accuracy gain		

Table 1. Results on programs using IEEE-754 binary 32 format.

Program	#Contexts	%Optimized	#Integral from		
			0 to 5	0 to 15	0 to 35
Interpol	7817	4.9%	7.1%	7.1%	7.1%
L-P filter 1	44	85,2%	9,2%	8,3%	7%
L-P filter 2	618	50,3%	7,5%	5,2%	4,1%
H-P filter 1	42	61,9%	14%	9,5%	2,5%
H-P filter 2	125	52%	7,4%	6,1%	3,9%
Transfer	364	98,5%	14,4%	17,7%	19,4%
Sqrt	76	80.2%	6.3%	6.3%	6.3%
			% accuracy gain		

Table 2. Results on programs using IEEE-754 binary 64 format.

6. Conclusion

In this article we have presented a new profitability heuristic which allows us to extract a more accurate version of a program from our intermediate representation called APEG. This article briefly describes how APEGs are constructed, and how they are able to represent many equivalent versions of a program in order to find one with better numerical accuracy.

We have designed a profitability analysis which runs in polynomial time and synthesize a new, but yet mathematically equivalent, version of a program. This heuristic is applied recursively into the APEG in order to synthesize a well-formed program using all the expressivity of our intermediate representation. We have presented extensive tests of our approach on many cases, both real case studies and exhaustive benchmarks. Our experimental results show that we are able to improve the accuracy of complex polynomial expressions by 20%.

Also our experimental results show significant improvement for real case example such as the embedded altitude estimator or real avionic code. We believe that our approach could be extended in many ways, also we already think about defining new propagation and expansion algorithms, also we are confident that the profitability heuristic we currently use could be improve in order to synthesize even more accurate programs.

References

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

[2] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Min, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace 2010*, Atlanta, Georgia, 20–22 April 2010. American Institute of Aeronautics and Astronautics.

[3] David Boland and George A. Constantinides. A scalable approach for automated precision analysis. In *FPGA*, pages 185–194, 2012.

[4] Olivier Bouissou and Matthieu Martel. Abstract interpretation of the physical inputs of embedded programs. In *VMCAI*, pages 37–51, 2008.

[5] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[6] 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*, pages 238–252. ACM Press, 1977.

[7] P. Cousot and R. Cousot. 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*, pages 178–190, Portland, Oregon, 2002. ACM Press, New York, NY.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[9] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Vedrine. Towards an industrial use of fluctuat on safety-critical avionics software. In *Formal Methods for Industrial Critical Systems, 14th International Workshop, FMICS'09*, 2009.

[10] E. Goubault, M. Martel, and S. Putot. Static analysis-based validation of floating-point computations. In *Numerical Software with Result Verification*, number 2991 in LNCS, 2004.

[11] E. Goubault and S. Putot. Static analysis of finite precision computations. In *VMCAI'11*, number 6538 in LNCS, pages 232–247, 2011.

[12] Nicolas Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[13] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[14] Philippe Langlois, Matthieu Martel, and Laurent Thévenoux. Accuracy Versus Time: A Case Study with Summation Algorithms. In *PASCO '10: Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, pages 121–130, New York, NY, USA, 2010. ACM.

[15] M. Martel. Semantics of roundoff error propagation in finite precision calculations. *Journal of Higher Order and Symbolic Computation*, 19:7–30, 2006.

[16] M. Martel. Semantics-based transformation of arithmetic expressions. In *Static Analysis Symposium, SAS'07*, number 4634 in LNCS. Springer-Verlag, 2007.

[17] Matthieu Martel. Enhancing the implementation of mathematical formulas for fixed-point and floating-point arithmetics. *Journal of Formal Methods in System Design*, 35:265–278, 2009.

[18] D. Monniaux. The pitfalls of verifying floating-point computations. *TOPLAS*, 30(3), May 2008.

[19] Christophe Mouilleron. *Efficient computation with structured matrices and arithmetic expressions*. PhD thesis, Université de Lyon – ENS de Lyon, November 2011. Manuscrit de Thèse, écrit en vue de l'obtention du titre de docteur de l'Université de Lyon - ENS de Lyon, spécialité informatique. Encadrant : Gilles Villard Co-encadrant : Claude-Pierre Jeannerod.

[20] J.-M. Muller. On the definition of ulp(x). Technical Report 5504, INRIA, 2005.

[21] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[22] Karthick Parashar, Romuald Rocher, Daniel Menard, and Olivier Sentieys. A hierarchical methodology for word-length optimization of signal processing systems. In *VLSI Design*, pages 318–323, 2010.

[23] Fred S. Roberts and Barry A. Tesman. *Applied combinatorics (2. ed.)*. Prentice Hall, 2005.

[24] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin 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*, pages 264–276, New York, NY, USA, 2009. ACM.

[25] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. *Logical Methods in Computer Science*, 7(1), 2011.

[26] Sid-Ahmed-Ali Touati and Denis Barthou. On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the 3rd conference on Computing frontiers*, CF '06, pages 147–156, New York, NY, USA, 2006. ACM.