Automatic source-to-source error compensation of floating-point programs: code synthesis to optimize accuracy and time

Laurent Thévenoux, Philippe Langlois, Matthieu Martel

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–23
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Automatic source-to-source error compensation of floating-point programs: code synthesis to optimize accuracy and time

Laurent Thévenoux[1]*, Philippe Langlois[2] and Matthieu Martel[3]

[1]*Inria – Laboratoire LIP, (CNRS, ENS de Lyon, Inria, UCBL), Univ. de Lyon, France*
[2]*UPVD – Laboratoire LIRMM, (CNRS, UPVD, UM2), Univ. de Perpignan, France*
[3]*UPVD – Laboratoire LAMPS (EA4217), Univ. de Perpignan, France*

SUMMARY

Numerical programs with IEEE 754 floating-point computations may suffer from inaccuracies, since finite precision arithmetic is an approximation of real arithmetic. Solutions that reduce the loss of accuracy are available, such as compensated algorithms or double-double precision floating-point arithmetic. Our goal is to automatically improve the numerical quality of a numerical program with the smallest impact on its performance. We define and implement source code transformations in order to derive automatically compensated programs. We present several experimental results to compare the transformed programs and existing solutions. The transformed programs are as accurate and efficient as the implementations of compensated algorithms when the latter exist. Furthermore, we propose some transformation strategies allowing us to partially improve the accuracy of programs and to tune the impact on execution time. Trade-offs between accuracy and performance are assured by code synthesis. Experimental results show that user-defined trade-offs are achievable in a reasonable amount of time, with the help of the tools we present in the paper.

Received ...

KEY WORDS: code synthesis; compensation; execution-time performance; floating-point arithmetic; multi-criteria optimization; numerical accuracy

1. INTRODUCTION

In this paper, we focus on numerical programs using IEEE 754 floating-point arithmetic. Several techniques have been introduced to improve the accuracy of numerical algorithms, such as expansions [1, 2], compensations [3, 4], differentiation methods [5] or extended precision arithmetic using multiple-precision libraries [6, 7]. Nevertheless, there are numerous and clearly identified numerical failures [8, 9]. This illustrates that these improvement techniques are perhaps not so widely used when writing numerical software, or not sufficiently automated to be applied more systematically. For example, the programmer has to modify the source code by overloading floating-point types with double-double arithmetic [7] or, less easily, by compensating the floating-point operations with error-free transformations (EFT) [3]. The latter transformations are difficult to implement without the preliminary step of designing the modified algorithm, which is typically done manually.

As much as the numerical accuracy, execution time is a critical matter for programs, especially in embedded systems where reducing the execution time may lead to less energy consumption

*Correspondence to: Laboratoire de l'informatique du parallélisme, ENS de Lyon, 46 allée d'Italie, 69364 Lyon, France.
E-mail: laurent.thevenoux@inria.fr.

or better reactivity. Taking into account both accuracy and execution time simultaneously is a difficult problem because these two criteria do not cohabit well: improving accuracy may be costly and execution-time improvement can impact accuracy as well. A solution is to provide trade-offs between accuracy and time [10].

We present a method that allows a usual developer to automatically improve the numerical accuracy of his programs without increasing execution times significantly. Although we do not provide error bounds on the processed algorithms, we automatically insert at compile-time some compensation steps based on EFTs. More precisely, we introduce a tool named CoHD [11] to parse C programs and generate a new C code with a compensated computation: floating-point operations $\pm$ and $\times$ are replaced by their respective error-free algorithms TwoSum and TwoProduct [12, Chap. 4]. Since using a generic solution (such as operator overloading) is not compatible with such transformations, CoHD generates a specific inline code.

We also present the SyHD software to perform transformations aiming to improve accuracy with an execution time requirement. SyHD synthesizes C source code for both accuracy and execution-time criteria. It uses CoHD transformations to improve accuracy, and transformation strategies to reduce how this accuracy improvement impacts the execution time. To improve execution time, we define two kinds of partial compensation strategies: some compensate the floating-point computations that generate the largest errors, and others do loop transformations, such as loop fission [13]. Then, SyHD synthesizes a new program by automatically analyzing the set of transformed programs for a given environment (defined by a target, some data, as well as some accuracy and execution-time constraints).

To demonstrate the efficiency of this approach, we compare the automatically transformed algorithms to existing compensated ones such as for floating-point summation [14] and polynomial evaluation [3, 4]. Our previous work CoHD [11] led to automatically recovering the same level of accuracy and execution time. Compensation is the right choice to benefit from a high instruction level parallelism (ILP) compared to the solutions derived using fixed-length expansions such as double-double or quad-double [7, 15]. The automatically transformed algorithms are shown to be very close, both in terms of accuracy and execution time, to the compensated algorithms we consider here. Now, our synthesizer SyHD applies transformation strategies to generate programs with accuracy and execution time trade-offs. We successfully generate programs with accuracy and execution-time trade-offs in a reasonable amount of time. We describe how this approach applies to a significant set of examples, ranging from recursive summation to polynomial evaluations and iterative refinement for linear system solving.

1.1. Related work

Several program transformation techniques or tools that improve the accuracy of floating-point computations have been proposed in the last decades. Nevertheless, most of them do not particularly take care of the execution time. One recent exception is Precimonius [16] that attempts to decrease the precision of intermediate operations to improve run-time and memory use.

Extending the precision is a commonly used technique. Libraries that implement expansions, like the QD Library [7] or the Scilab toolbox [17], easily improve accuracy in simple cases. MPFR [6] is an arbitrary-precision floating-point library with correct rounding, to accurately evaluate floating-point computations with the all necessary precision. These techniques introduce serious overheads for runtime and memory, as shown in [18] for example.

Rewriting expressions. In [19], Ioualalen and Martel propose the Sardana tool. It performs a bounded exhaustive search for algebraically-equivalent programs for which a better accuracy bound could be proven statically. This approach is based on abstract interpretation to bound rounding errors using a reasonable over-approximation. A set of equivalent programs is generated over the real numbers, and the one with the smallest rounding error is chosen. Panchekha *et al.* [20] propose the Herbie tool, which improves the accuracy of expressions by randomly sampling inputs, localizing error, generating candidate rewrites, and merging rewrites with complementary effects. Herbie's results show that it can effectively discover transformations that substantially improve accuracy while imposing a median runtime overhead of 40% on their numerous selected case studies. A more

DOI: 10.1002/cpe

aggressive solution, like the one used in GCC when using the `-fast-math` flag, speeds up floating-point programs by rewriting floating-point computations even if the numerical results may change. GCC gives no guarantee about the resulting accuracy but warns the user of potential numerical failures.

All these approaches tackle the question of accuracy improvement from different angles but share the tendency to combine this improvement with other criteria, especially with execution-time. Our tools can be used as a numerical analysis assistant but do not include the verification of numerical codes. Assisting and certifying numerical programs is for instance the goal of Gappa [21], a tool based on the Coq proof assistant [22], or Fluctuat [23], used to track floating-point errors with abstract interpretation.

1.2. Outline

This article is organized as follows. Section 2 provides the notations and some background material on floating-point arithmetic, EFTs, and accuracy improvement techniques like double-double arithmetic or compensation. The core of this article is Section 3 and Section 4, where we present our automatic code transformations to optimize the accuracy of floating-point computations with a small execution time overhead. Figure 1 gives an overview of this process. The four bottom modules compose the CoHD tool and the four top ones describe SyHD. Most of our contributions are implemented in the module "Transformation library," which is described in both Section 3 and Section 4. Section 4 also describes the modules "Strategies set research space," "Measurements & result exploration," and "Strategy selection." Other modules are classical compilation or code transformation blocks. In Section 5, some experimental results illustrate the practical interest of our approach compared to existing ones. Conclusions and perspectives are given in Section 6.

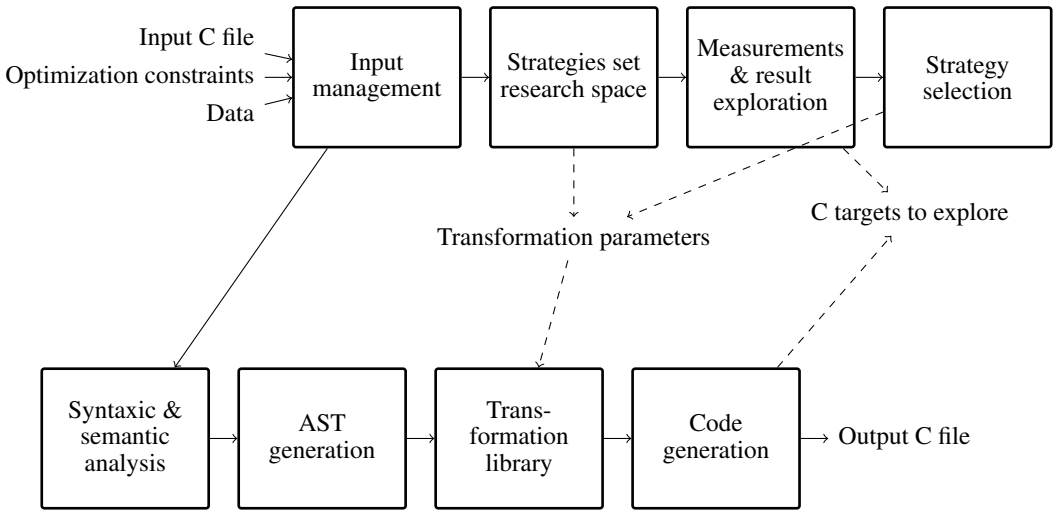


Figure 1. CoHD (bottom) and SyHD (top) tools detailed in Section 3 and Section 4.

2. PRELIMINARIES AND NOTATION

In this section we use classical notations to address IEEE floating-point arithmetic, basic methods to analyze the accuracy of floating-point computations, and EFTs of the basic operations $\pm$ and $\times$. We also demonstrate how to exploit these EFTs within expansions and compensations.

DOI: 10.1002/cpe

2.1. IEEE floating-point arithmetic

In base 2 and precision p, IEEE floating-point numbers have the form

$$f = (-1)^s \cdot m \cdot 2^e,$$

where $s \in \{0, 1\}$ is the sign, $m = \sum_{i=0}^{p-1} d_i 2^{-i} = (d_0.d_1 d_2 \cdots d_{p-1})_2$ is the significand (with $d_i \in \{0, 1\}$), and e is the exponent. The IEEE 754-2008 standard [24] defines such numbers for several formats, rounding modes, and the semantics of the five basic operations $\pm, \times, \div, \sqrt{\cdot}$.

Notation and assumptions. Throughout the paper, all computations are performed in the *binary64* format, with the *round-to-nearest* mode. We assume that neither overflow nor underflow occurs during the computations. We use the following notations.

- $\mathbb{F}$ is the set of all normalized floating-point numbers. For example, the *binary64* format, over 64 bits, includes $p - 1 = 52$ bits for the fractional part of the significand, 11 bits for the exponent e, and 1 bit for the sign s.
- $fl(\cdot)$ denotes the result of a floating-point computation where every operation inside the parenthesis is performed in precision p and with rounding to nearest.
- Given $x \in \mathbb{F}$, we write $ulp(x)$ to denote the unit in the last place of x, defined by $ulp(x) = 2^e \cdot 2^{1-p}$ if $x \neq 0$, and by 0 otherwise. Let $\widehat{x} = fl(x)$ for a real number x. We have $|x - \widehat{x}| \leqslant ulp(\widehat{x})/2$.

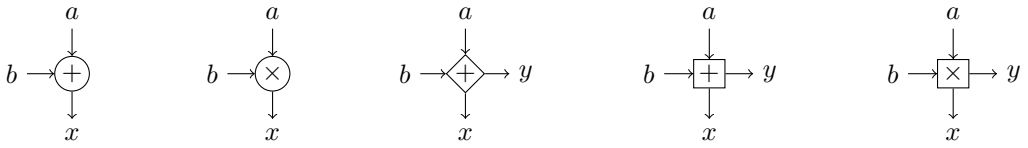
Accuracy analysis. One way to measure the accuracy of $\widehat{x} = fl(x)$ is the number of significant bits $\#_{sig}$ shared by x and $\widehat{x}$:

$$\#_{sig}(\widehat{x}) = -log_2\big(|x - \widehat{x}|/|x|\big), \quad x \neq 0. \tag{1}$$

2.2. Error-free transformations

In this section, we briefly introduce the error-free transformations (EFT) previously discussed in [11] (we refer to the latter for a more detailed description). EFTs provide lossless transformations of basic floating-point operations $\circ \in \{+, -, \times\}$. The practical interest of EFTs comes from FASTTWOSUM, TWOSUM (introduced by Dekker [1], Knuth [25, Chap. 4] and Møller [26]), and TWOPRODUCT [1] algorithms which exactly compute in floating-point arithmetic the error term of the sum and the product.

Figure 2 defines diagrams for floating-point operations $\pm$ and $\times$, and for their EFTs. In what follows, we graphically represent transformation algorithms as basic computational blocks.



(a) SUM (b) PRODUCT (c) FASTTWOSUM (d) TWOSUM (e) TWOPRODUCT

Figure 2. Diagrams for basic floating-point operations (a), (b) and EFT algorithms (c), (d), and (e).

2.3. Double-double and compensated algorithms

We will now focus on two methods using these EFTs to double the accuracy: double-double expansions and compensations. We will then recall why compensated algorithms run faster than double-double algorithms (we refer to [11] for a complete discussion on these two approaches).

DOI: 10.1002/cpe

Double-double expansions. We consider here the Briggs, Kahan, and Bailey algorithms used in the QD library [7]. Double-double arithmetic simulates computations with twice precision. Proofs are detailed in [27]. In practice, double-double algorithms can be simply derived by substituting the basic operations.

Compensated algorithms. As double-double algorithms, compensated algorithms can double the accuracy but they are not easy to derive. Compensated algorithms have been, up to now, defined on a case by case basis and by experts of rounding error analysis [3, 28, 4, 29, 14]. For example, the compensated Algorithm SUM2 [14] returns a sum that is proven to be twice as accurate.

In this paper, we refer to a compensated algorithm by using the prefix "*Comp*" or the suffix "*2*," and to a double-double algorithm with the prefix or suffix "*DD*."

Double-double versus compensation. Double-double and compensated algorithms can be used to obtain similar accuracy. How do they compare in terms of computing time? A detailed analysis has been presented in [30] concerning the classical summation algorithm. The authors show that SUM2 algorithm benefits from an ILP seven times higher than that of SUMDD.

This fact is measurable in practice, and the compensated algorithms exploit this low-level parallelism much better than double-double ones. The example of HORNER's polynomial evaluation is detailed in [15]. The latter shows that the compensated HORNER's algorithm runs at least twice as fast as its double-double counterpart with the same output accuracy. This efficiency motivates us to automatically generate compensated algorithms.

3. COHD: AUTOMATIC ERROR COMPENSATION

We present solutions for improving accuracy thanks to code transformation, in order to benefit from the efficiency of compensation. This code transformation provides trade-off between accuracy and time compared to double-double expansions for example. More advanced trade-offs obtained with code synthesis will be discussed in Section 4.

CoHD is a source-to-source transformer written in OCaml and built as a compiler. The *front-end* reads input C files and comes from a previous development by Casse [31]. The *middle-end* implements some passes of optimization, from classical compiler passes such as operand renaming or three-address code conversion [13, Chap. 19]. It also implements one pass of floating-point error compensation we define in next Section 3.1. Then, the *back-end* translates the intermediate representation into C code.

3.1. Improving accuracy: methodology

Our code transformation automatically compensates programs following three steps.

1. Detection of sequences of floating-point computations. A sequence is the set $\mathscr{S}$ of dataflow-dependent operations required to obtain one or several results.
2. For each sequence $\mathscr{S}_i$, computation and accumulation of the error terms. This is performed besides the original sequence by (a) replacing floating-point operations by the corresponding EFTs, and (b) accumulating error terms following Algorithms 1 and 2 given hereafter. At this stage, every floating-point number $x \in \mathscr{S}_i$ becomes *a compensated number*, denoted $\langle x, \delta_x \rangle$, *where* $\delta_x \in \mathbb{F}$ *is the accumulated error term attached to the computed result* x.
3. Closing of the sequences. Closing is the compensation step itself, so that $close(\mathscr{S}_i)$ means computing $x \leftarrow fl(x + \delta_x)$ for x being any result of $\mathscr{S}_i$.

3.2. Compensated operators for compensated numbers

Next Algorithms 1 and 2 automatically compensate the error of basic floating-point operations when inputs are compensated numbers. Two different sources of errors have to be considered.

DOI: 10.1002/cpe

First, the error generated by the elementary operation itself, which is computed with an EFT. This computation corresponds to $\delta_{\pm}$ and $\delta_{\times}$ in the first line of Algorithms 1 and 2. Second, the errors inherited from the two operands, denoted by δ_a and δ_b, are accumulated with the previous generated error. This accumulation corresponds to the second line of Algorithms 1 and 2. These inherited errors come from previous floating-point calculations. Operands with no inherited error are processed differently to minimize added compensations.

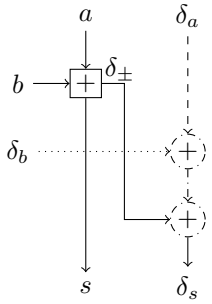
$[s, \delta_{\pm}] = \text{TwoSum}(a, b)$ ▷ Elementary operation replaced by the EFT
$\delta_s \leftarrow fl((\delta_a + \delta_b) + \delta_{\pm})$ ▷ Accumulation of generated and inherited errors
return $\langle s, \delta_s \rangle$

Algorithm 1: AC_TwoSum($\langle a, \delta_a \rangle, \langle b, \delta_b \rangle$), automatically compensated sum of two *compensated numbers*.

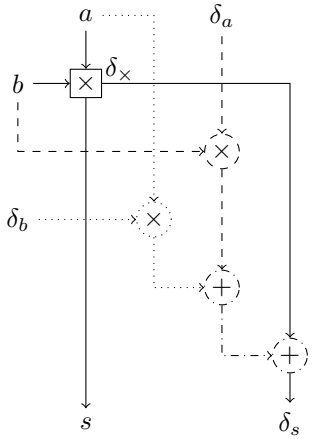
$[s, \delta_{\times}] = \text{TwoProduct}(a,\ b)$ ▷ Elementary operation replaced by the EFT
$\delta_s \leftarrow fl(((a \times \delta_b) + (b \times \delta_a)) + \delta_{\times})$ ▷ Accumulation of generated and inherited errors
return $\langle s, \delta_s \rangle$

Algorithm 2: AC_TwoProduct($\langle a, \delta_a \rangle, \langle b, \delta_b \rangle$), automatically compensated product of two *compensated numbers*.

Figure 3 shows such variants which can be obtained by removing the dashed or dotted lines.



(a) AC_TwoSum.



(b) AC_TwoProduct.

Figure 3. Diagrams for Algorithms 1 and 2 for the automatic compensation of the sum (a), and the product (b). The dashed or dotted lines are removed when a or b is a standard floating-point number.

3.3. Application to computation sequence

In this section, we illustrate how our methodology transforms Listing 1 to Listing 2.

Listing 1: Original code computing the sequence $a = b + c \times d$.

```
double foo() {
  double a, b, c, d ;
  [...]
  a = b + c * d ; /* computation sequence */
  [...]
  return a ;
}
```


DOI: 10.1002/cpe

First, CoHD detects the sequence a = b + c ∗ d (Listing 1, line 4). This sequence is preliminarly converted in three-address form (Listing 2, lines 10 and 22). Then, computations are replaced by EFTs, and errors are accumulated following Algorithm 2 for the multiplication at line 10 (lines 11 to 21), and Algorithm 1 for the addition at line 22 (lines 23 to 27). Finally, the last step closes the sequence here at line 30 where the error term delta_a is added to its corresponding variable.

Listing 2: Transformed code computing the sequence $a = b + c \times d$ with error compensation.

```
double foo() {
  double a, b, c, d ;
  [...]
  /* variables introduced by step 1 */
  double t, c_H, c_L, d_L, d_H, tmp_L, a_L ;
  /* variables introduced by step 2 */
  double delta_tmp, delta_a ;
  [...]
  /* first part of the sequence detected at step 1 */
  tmp = c * d ;
  /* step 2a: adding 16 flops with TwoProduct(c,d) */
  t = 134217729.0 * c ; /* 2^ceil(53/2) + 1 */
  c_H = t - (t - c) ;
  c_L = c - c_H ;
  t = 134217729.0 * d ;
  d_H = t - (t - d) ;
  d_L = d - d_H ;
  tmp_L = c_L * d_L - ((( tmp - c_H * d_H) - c_L * d_H) - c_H * d_L) ;
  /* step 2b: accumulation of TwoProduct error term result and inherited errors from c and d */
  delta_tmp = tmp_L ;
  /* second part of the sequence detected at step 1 */
  a = b + tmp ;
  /* step 2a: adding 5 flops with TwoSum(b,tmp) */
  t = a - b ;
  a_L = (b - (a - t)) + (tmp - t)
  /* step 2b: accumulation of TwoSum error term result and inherited errors from b and tmp */
  delta_a = a_L + delta_tmp;
  [...]
  /* step 3: close sequence */
  return a + delta_a ;
}
```

4. SYHD: CODE SYNTHESIS TO OPTIMIZE ACCURACY AND TIME

Section 3 explains how to generate a fully compensated program. Here we introduce strategies of partial transformation that yield programs with different patterns of transformation. Hence we aim to select between them the best trade-off for accuracy and runtime constraints defined by the user.

SyHD is a code synthesizer also written in OCaml. As described in Figure 1, it uses CoHD to perform multiple code transformations from some inputs being the C program file to optimize, optimization constraints, and data. SyHD explores a set of transformed programs generated with CoHD following several transformation strategies and finds the program with the best accuracy and execution time properties corresponding to the input parameters. Here, we describe how SyHD builds this transformed program set and how it selects the one with the best accuracy and time tradeoff.

4.1. Automatic compensation

The automatic compensation process described in Section 3 improves accuracy with a minimal performance overhead compared to other techniques. Our SyHD tool here goes further to minimize execution-time overhead by partially applying the compensation of floating-point computations. Hence we reduce the performance overhead of the transformed program by sacrificing some

DOI: 10.1002/cpe

accuracy. When partially compensating a program, new alternatives occur. Indeed, after a compensated part, the accumulated errors can be propagated through the next floating-point computations instead of performing a closure. For addition and multiplication, this propagation is achieved by Algorithms 3 and 4 given below.

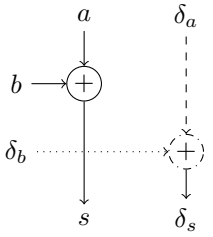
$s \leftarrow RN(a + b)$ ▷ Elementary operation unchanged
$\delta_s \leftarrow RN(\delta_a + \delta_b)$ ▷ Propagation of inherited errors
return $\langle s, \delta_s \rangle$

Algorithm 3: AP_TWOSUM($\langle a, \delta_a \rangle, \langle b, \delta_b \rangle$), sum with automatic error propagation of two *compensated numbers*.

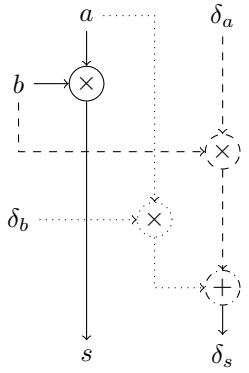
$s \leftarrow RN(a \times b)$ ▷ Elementary operation unchanged
$\delta_s \leftarrow RN(a \times \delta_b + b \times \delta_a)$ ▷ Propagation of inherited errors
return $\langle s, \delta_s \rangle$

Algorithm 4: AP_TWOPRODUCT($\langle a, \delta_a \rangle, \langle b, \delta_b \rangle$), product with automatic error propagation of two *compensated numbers*.

Again, operands with no inherited errors are processed to minimize added operations. Figure 4 shows such variants which can be obtained by removing the dashed or dotted lines.



(a) AP_TWOSUM.



(b) AP_TWOPRODUCT.

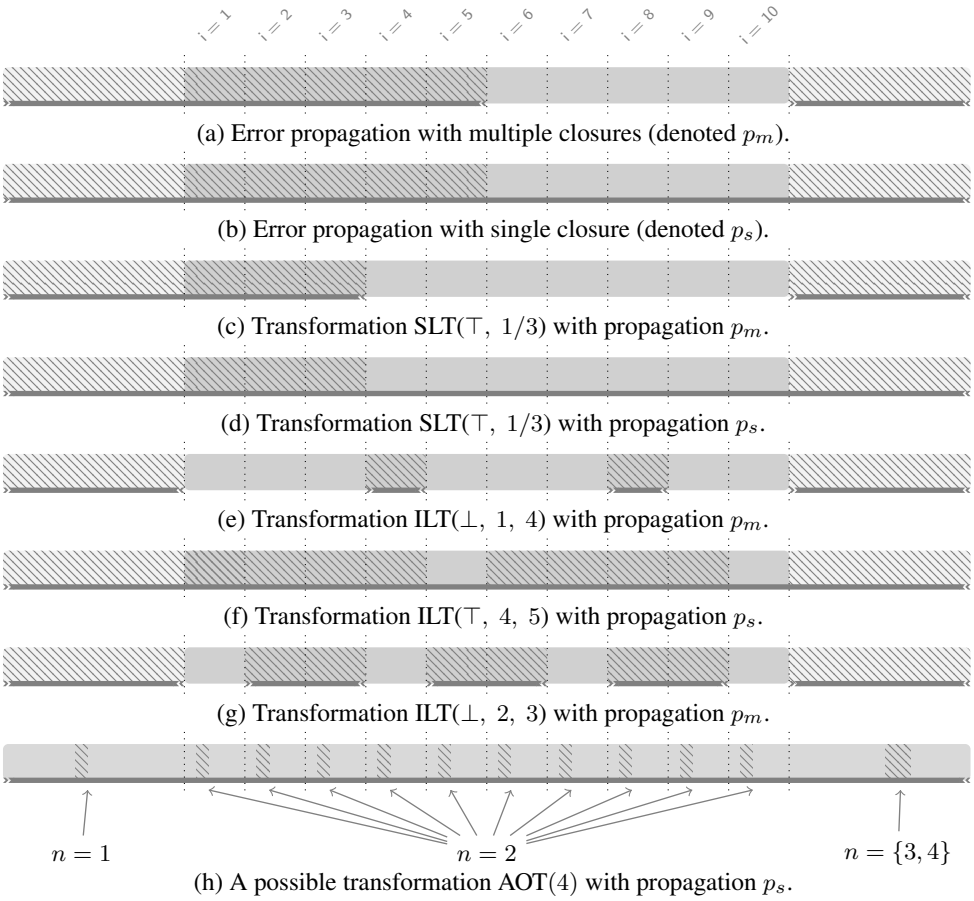
Figure 4. Diagrams for Algorithms 3 and 4 for the automatic error propagation of the sum (a), and the product (b). Here the elementary operations are not replaced by an EFT as for Algorithms 1 and 2 of Figure 3, so only inherited errors are processed (remove dashed or dotted lines for standard inputs).

One source of error remains now compared to previous Algorithms 1 and 2. Here, we only consider the inherited errors of the previous computations and ignore the generated error. Error propagation corresponds to the second lines of Algorithms 3 and 4 (the first lines are the unchanged elementary operations). This class of propagation is depicted in Figure 5b and it is denoted by p_s which means "propagation with single closure." On the contrary, the closures performed at the end of each compensated part do not require the use of these additional algorithms. Computations not affected by compensation remain unchanged. We denote this case as p_m, meaning "propagation with multiple closures." It is represented in Figure 5a.

4.2. Optimization strategies

In this section, we describe how programs are partially compensated. We propose two kinds of optimization strategies. A first pair of transformation operates on loops, while a third one operates on the floating-point computations which generate the largest errors. We use the minimal

DOI: 10.1002/cpe



(a) Error propagation with multiple closures (denoted p_m).

(b) Error propagation with single closure (denoted p_s).

(c) Transformation SLT($\top$, $1/3$) with propagation p_m.

(d) Transformation SLT($\top$, $1/3$) with propagation p_s.

(e) Transformation ILT($\bot$, 1, 4) with propagation p_m.

(f) Transformation ILT($\top$, 4, 5) with propagation p_s.

(g) Transformation ILT($\bot$, 2, 3) with propagation p_m.

(h) A possible transformation AOT(4) with propagation p_s.

Figure 5. Representation of transformed program traces (sequential execution). Figures 5a and 5b define the basics of the representation: compensated parts are shaded and error propagation is represented with a sub bold line. Figures 5c to 5h show several examples of program transformation.

program model of Listing 3 in order to clearly express the program transformations introduced on Algorithms 5, 6, and 7. These latter are respectively described in the following paragraphs as single loop, intermittent loop, and accuracy-oriented transformations.

Listing 3: Minimal model of program representation

```
statement := sequence of statement list
           | loop of id * min * max * statement
           | computation of expression
expression := binary of expression * operator * expression
            | unary of operator * expression
            | id
operator := '+' | '-' | '=' | '?' | ':' | '>'
```

We also define some basic procedures needed in Algorithms 5, 6, and 7 to manipulate program representation described in Listing 3.

- The loop_stmt, expr_stmt, and sequence procedures return a statement as ensued by the program representation of Listing 3.
- The compensate and no_compensate procedures return a statement resulting of the compensation (or the no-compensation) of a given statement. The compensate procedure applies automatic error compensation as described in Section 3. The no_compensate

DOI: 10.1002/cpe

procedure has no effect if the propagation p_m is in use, or applies automatic error propagation algorithms on floating-point sequences as explained in Section 4.1.

Simple loop transformation. SLT splits a loop into two loops to partially compensate one of them. Algorithm 5 defines the code transformation of this strategy. SLT depends on parameters (ρ, r), where r is the ratio of loop iterations to compensate, and $\rho \in \{\top, \bot\}$ is the position of these loop iterations. For example, SLT$(\top, 1/2)$ compensates the first half iterations of the original loop. Figures 5c and 5d illustrate it and an example of such transformation is given in Listing 5.

Require: A loop statement $\mathsf{id} * \mathsf{min} * \mathsf{max} * \mathsf{body}$ as defined in Section 4, and the strategy parameters: the ratio r of iterations to compensate, and the position $\rho \in \{\top, \bot\}$
Ensure: A sequence statement replacing the required input statement
  if $\rho == \top$ **then**
    $e = floor((\mathsf{max} - \mathsf{min}) \times r)$
    $l1 = \mathsf{loop\_stmt}(\mathsf{id}, \mathsf{min}, e + \mathsf{min}, \mathsf{compensate}(body))$
    $l2 = \mathsf{loop\_stmt}(\mathsf{id}, e + \mathsf{min} + 1, \mathsf{max}, \mathsf{no\_compensate}(body))$
  else
    $e = floor((max - min) \times (1 - r))$
    $l1 = \mathsf{loop\_stmt}(\mathsf{id}, \mathsf{min}, e + \mathsf{min}, \mathsf{no\_compensate}(body))$
    $l2 = \mathsf{loop\_stmt}(\mathsf{id}, e + \mathsf{min} + 1, \mathsf{max}, \mathsf{compensate}(body))$
  end if
  return $\mathsf{sequence}(l1, l2)$

Algorithm 5: Simple loop transformation (SLT).

Intermittent loop transformation. ILT splits a loop into $2k$ blocks in order to partially compensate one block every two blocks. The resulting program transformation consists in two loops nested into another one. Algorithm 6 describes this code transformation. ILT depends on three parameters (ρ, t, f), where t is the number of loop iterations to compensate, f is the frequency of repetition (in number of loop iterations), and $\rho \in \{\top, \bot\}$ specifies the position of the loop iterations to compensate: the first t iterations of f if $\rho = \top$, the last t iterations otherwise. For example, ILT$(\top, 1, 2)$ compensates the first iteration for each block of 2 iterations. Figures 5e, 5f, and 5g illustrate this kind of transformations and an example is given in Listing 6.

Require: A loop statement $\mathsf{id} * \mathsf{min} * \mathsf{max} * \mathsf{body}$ as defined in Section 4, and the strategy parameters: the number t of loop iterations to compensate, the frequency f (in iterations) of compensation, and the position $\rho \in \{\top, \bot\}$
Ensure: A loop statement replacing the required input statement
  $m1 = \mathsf{expr\_stmt}(j + t - 1 > \mathsf{max}\ ?\ j + t - 1 : \mathsf{max})$
  $m2 = \mathsf{expr\_stmt}(j + f - 1 > \mathsf{max}\ ?\ j + f - 1 : \mathsf{max})$
  if $\rho == \top$ **then**
    replace t by $f - t$ in $m1, m2$
    $l1 = \mathsf{loop\_stmt}(\mathsf{id}, j, m1, \mathsf{compensate}(body))$
    $l2 = \mathsf{loop\_stmt}(\mathsf{id}, m1 + 1, m2, \mathsf{no\_compensate}(body))$
  else
    $l1 = \mathsf{loop\_stmt}(\mathsf{id}, j, m1, \mathsf{no\_compensate}(body))$
    $l2 = \mathsf{loop\_stmt}(\mathsf{id}, m1 + 1, m2, \mathsf{compensate}(body))$
  end if
  $e = \mathsf{expr\_stmt}(j = j + f)$
  return $\mathsf{loop\_stmt}(j, \mathsf{min}, \mathsf{max}, \mathsf{sequence}(l1, l2, e))$

Algorithm 6: Intermittent loop transformation (ILT).

DOI: 10.1002/cpe

Listing 4. Recursive summation in C.

```
int main(int argc, char** argv) {
  int i, n=atoi(argv[1]); double s, tab[n];
  FILE *file;
  file = fopen (argv[2], "rb");
  fread(&tab, sizeof(double), n, file);
  fclose (file);

  s = tab[0];
  for(i=1; i<n; i++)
    s = s + tab[i];

  printf("%a\n", s);
  return 0;
}
```

Listing 5. Listing 4 after transformation $\mathrm{SLT}(\top, 1/2)$ with propagation p_m.

```
int main(int argc, char** argv) {
  int i, n = atoi(argv[1]); double s, tab[n] ;
  int ac_end;
  double s_dt, __ac__tmp1, __ac__tmp1_dt,
       s_twosum, s_ts1, s_ts2;
  FILE * file;
  file = fopen(argv[2], "rb");
  fread(&tab, sizeof( double ), n, file);
  fclose(file);

  s = tab[0];
  s_dt = 0.0;
  ac_end = (int) n * 0.5;
  for(i = 1; i < ac_end; i++) {
    __ac__tmp1 = s;
    __ac__tmp1_dt = s_dt;
    s = __ac__tmp1 + tab[i];
    s_ts1 = s - tab[i];
    s_ts2 = s - s_ts1;
    s_twosum = (__ac__tmp1 - s_ts1)
      + (tab[i] - s_ts2);
    s_dt = s_twosum + __ac__tmp1_dt;
  }
  s += s_dt;
  s_dt = 0.0;
  for(i = ac_end; i < n; i++) {
    s = s + tab[i];
  }

  printf("%a\n", s + s_dt);
  return 0;
}
```

Listing 6. Listing 4, after transformation $\mathrm{ILT}(\top, 1, 2)$ with propagation p_s.

```
int main(int argc, char** argv) {
  int i, n = atoi(argv[1]); double s, tab[n] ;
  int ac_i, ac_end2, ac_end3;
  double s_dt, __ac__tmp1, __ac__tmp1_dt,
       s_twosum, s_ts1, s_ts2;
  FILE * file;
  file = fopen(argv[2], "rb");
  fread(&tab, sizeof( double ), n, file);
  fclose(file);

  s = tab[0];
  s_dt = 0.0;
  for(ac_i = 1; ac_i < n; ac_i = ac_i + 2) {
    ac_end2 = ac_i + 1 < n ? ac_i + 1 : n;
    ac_end3 = ac_i + 2 < n ? ac_i + 2 : n;
    for(i = ac_i; i < ac_end2; i++) {
      __ac__tmp1 = s;
      __ac__tmp1_dt = s_dt;
      s = __ac__tmp1 + tab[i];
      s_ts1 = s - tab[i];
      s_ts2 = s - s_ts1;
      s_twosum = (__ac__tmp1 - s_ts1)
        + (tab[i] - s_ts2);
      s_dt = s_twosum + __ac__tmp1_dt;
    }
    for(i = ac_end2; i < ac_end3; i++) {
      s = s + tab[i];
      s_dt = s_dt;
    }
  }

  printf("%a\n", s + s_dt);
  return 0;
}
```

Accuracy oriented transformation. AOT selects the floating-point operations that generate the largest errors. Errors can be measured through different techniques. We propose to consider the absolute errors of the transformed code results compared to some reference values computed in higher precision. The principle of this transformation is explained in Algorithm 7 and Figure 5h illustrates it.

DOI: 10.1002/cpe

Require: A program p to transform, a set D of dataset, and the strategy parameter: the number n of expression statements to compensate
Ensure: The transformed input program which compensates the n expressions of p generating the maximum absolute errors compared to its higher-precision counterpart
for all $d \in D$ **do**
execute the program p and its counterpart in higher precision with the data d
$\widehat{P}_d \leftarrow$ all the intermediate results of the floating-point computations of p
$P_d \leftarrow$ all the intermediate results of the floating-point computations of p in higher precision (reference results)
for all $(\widehat{i}, i) \in (\widehat{P}_d, P_d)$ **do**
$e^d_{\widehat{i}} \leftarrow E_{\text{abs}}(\widehat{i}, i)$ ▷ the absolute error $\widehat{i} - i$
end for
end for ▷ vector e contain all the absolute errors of all the intermediate floating-point computations of p for each dataset in D
$e \leftarrow mean(e)$ ▷ component-wise
$e \leftarrow sort(e)$ ▷ decrease order
for all $i \in \{1, n\}$ **do**
compensate(get_expr($pop(e)$)) ▷ get_expr returns the expression statement of the computation which has generated the absolute error returned by pop
end for
return the input sequence with n compensated floating-point computations

Algorithm 7: Accuracy oriented transformation (AOT).

4.3. Strategies

Our loop transformation have to be sized with respect to the number of loop iterations. One additional parameter ν is introduced to be set by the user to adapt the transformations (ILT and AOT) to the program size. Figure 6 presents the default parameters for SLT, ILT and AOT. By default $\nu = 1$. So, for example, if ILT operates on a loop performing one million iterations, then the default transformation splits the loop into blocks of size ranging from two iterations to five. By setting ν to 100 or 100,000, ILT splits the loops into blocks of sizes ranging from 200 to 500 or 200,000 to 500,000, and so it adapts the transformation to the number of loop iterations.

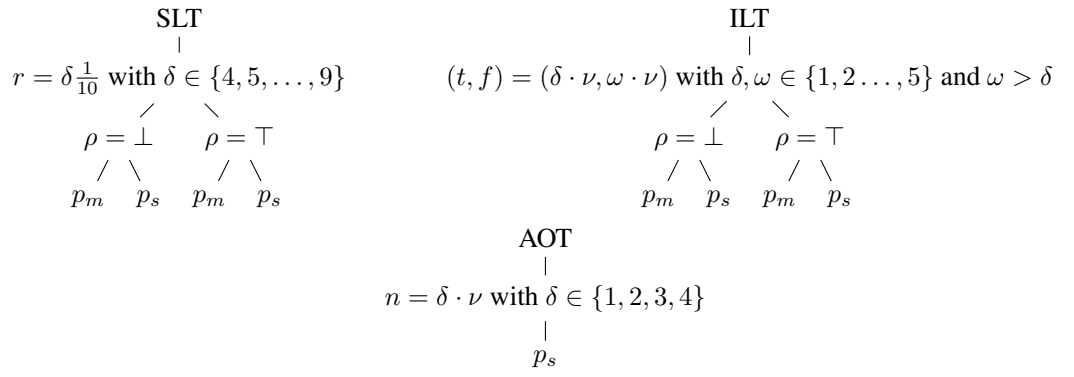


Figure 6. Default parameters of strategy set E for code synthesis. Parameter ν can be chosen by the user to adapt the transformations to the program size ($\nu = 1$ by default).

Let us note that, where computational sequence begin after entering the loop, propagations p_s and p_m are equivalent for the SLT transformation when $\rho = \bot$ (when appropriate the propagation is denoted by $p_{m|s}$). Then, the set E of strategies contains more than sixty different programs and

DOI: 10.1002/cpe

hence covers a large set of possible partially compensated programs. Section 4.4 describes how to select the best program into E to match a required accuracy and time trade-off.

4.4. Code synthesis: accuracy and execution time trade-offs

We propose two ways to select the transformed program corresponding to accuracy and time constraints. $R_{\text{acc}}(E)$ selects the faster program among the most accurate ones and $R_{\text{exe}}(E)$ selects a less accurate program (compared to the one given by R_{acc}) to reduce the execution time.

We introduce $r_{\text{sig. bits}}$ and r_{cycles} respectively as the ratios of significant bits and cycles numbers and defined as:

$$r_* = \frac{\textit{automatically transformed} \text{ program measure}}{\textit{high-precision} \text{ program measure}}. \tag{2}$$

We have:

$$R_{\text{acc}}(E) = \min_{s \in \{\max_{s \in E} r_{\text{sig. bits}}(s)\}} r_{\text{cycles}}(s), \text{ and,} \tag{3}$$

$$R_{\text{exe}}(E) = \min_{s \in E} \frac{\alpha \cdot r_{\text{cycles}}(s) + \beta \cdot (1 - r_{\text{sig. bits}}(s))}{\alpha + \beta}, \quad \forall \alpha, \beta > 0. \tag{4}$$

R_{exe} can be tuned to select a more or less accurate or fast program according to α and β. Moreover, to strictly satisfy the success of the selection, the transformed program must verifies the following points.

1. For each datum of a given dataset, the transformed program improves accuracy compared to the original program.
2. For the R_{acc} criterion, the transformed program runtime must be smaller than the *higher-precision* program.
3. For the R_{exe} criterion, the transformed program runtime must be smaller than the *fully compensated* program.

So, for a given program p and from: the set of strategies E, a set of data D, and an execution-time versus accuracy criterion $R \in \{R_{\text{acc}}, R_{\text{exe}}\}$, the program selection defined by Algorithm 8 returns a transformed program p_ω satisfying the R criterion. Algorithm 8 introduces two procedures, get_cycle_number and get_significant_bit_number, which return the number of cycles to execute a given program, and its accuracy expressed in the numbers of significant bits. Several solutions could implement these measures. We describe our preferred solutions in Section 5. Moreover, Algorithm 8 needs to compute some reference values such as $n_{\text{sig. bits}}$ the average number of significant bits obtained with a higher-precision version of the input program. In order to obtain the most efficient optimization strategy, data must be provided by the user. Although our tools do not support missing datasets, we could process our analysis on a substantial randomly chosen subset of the corresponding floating-point numbers set.

Algorithm 8 implies that the user has to preprocess the program to transform it by adding directives specifying which results have to be considered for accuracy improvement (typically, the result of a function). The user also has the option to specify the piece of code where time is measured.

5. EXPERIMENTAL RESULTS

We now describe the CoHD and SyHD tools that implement our code transformations. We apply them to several case studies described in Section 5.2 and chosen such that compensated versions are available to compare with and to validate the complete program transformation. We show in Section 5.3 that we automatically recover the accuracy and the performance of the existing compensated algorithms. We also add comparisons with the double-double versions. In Section 5.4 we explore the partial program transformation strategies in order to provide execution time versus accuracy tradeoffs. We show that we can generate transformed programs dealing with such

DOI: 10.1002/cpe

Require: A strategy set E, a set of data D, and an execution-time versus accuracy criterion R
Ensure: A transformed program p_ω satisfying the criterion R
function EVALUATE-PERFORMANCE($p,\ d,\ n_{\text{ref}}$) ▷ evaluate performance of program p with data d and n_{ref} as the cycle number of the reference program
$n \leftarrow \mathsf{get\_cycle\_number}(p)$
return $r_{\text{cycles}} \leftarrow n/n_{\text{ref}}$ ▷ see relation (2)
end function
function EVALUATE-ACCURACY($p,\ d,\ n_{\text{ref}}$) ▷ evaluate the accuracy of program p with data d and n_{ref} as the number of significant bits of the reference program
$o \leftarrow \mathsf{get\_significant\_bit\_number}(p)$
$n \leftarrow \#_{sig}(o)$ ▷ see relation (1)
return $r_{\text{significant bits}} \leftarrow n/n_{\text{ref}}$ ▷ see relation (2)
end function
▷ Compute reference values
$p_{\text{ref}} \leftarrow$ the "high-precision" version of program p
$n_{\text{cycles}} \leftarrow \mathsf{get\text{-}cycle\text{-}number}(p_{\text{ref}})$
$n_{\text{sig. bits}} \leftarrow \mathsf{get\text{-}significant\text{-}bit\text{-}number}(p_{\text{ref}})$
▷ Evaluate the accuracy and performance of each program generated with the strategies of E
$\Omega \leftarrow \emptyset$
for all $i \in E$ **do**
$p^i \leftarrow$ program generated with strategy i
$r^i_{\text{cycles}} = \text{EVALUATE-PERFORMANCE}(p^i, \mathsf{head}(D),\ n_{\text{cycles}})$
for all $d \in D$ **do**
$r^{i_d}_{\text{sig. bits}} = \text{EVALUATE-ACCURACY}(p^i,\ d,\ n_{\text{sig. bits}})$
end for
$r^i_{\text{sig. bits}} \leftarrow \mathsf{mean}\left(r^{i_d}_{\text{sig. bits}}\right)$
$\Omega \leftarrow \Omega \cup \left(i, r^i_{\text{cycles}}, r^i_{\text{sig. bits}}\right)$
end for
▷ Apply criterion selection
$s \leftarrow R(\Omega)$ ▷ see Relations (3) and (4)
return $p_\omega \leftarrow$ the transformed program p with strategy s

Algorithm 8: Code synthesis search.

constraints, and this in a reasonable amount of time. Test cases include summation, polynomial and derivative evaluations with HORNER, CLENSHAW, and DECASTELJAU algorithms for which compensated versions have been published. A last test case presented in Section 5.5 is the iterative refinement algorithm applied to several ill-conditioned linear systems.

We recall that compensated and double-double algorithms are respectively prefixed with "Comp" and "DD." Similarly, we use the "AC" prefix for those generated by our automatic compensation method. We start to introduce how we perform time and accuracy measurements as required by Algorithm 8.

5.1. Experimental environments and measure methods.

We perform accuracy and execution time measurements to compare programs automatically generated from our method to hand-coded compensation programs and double-double algorithms. All measurements are done within the following experimental environments.

Environment 1. Intel® Core™i5 CPU M540: 2.53GHz, Linux 3.2.0.51-generic-pae i686 i386, gcc v4.6.3 with -O2 -mfpmath=sse -msse4, PAPI v5.1.0.2 and PERPI (pilp5 version).

DOI: 10.1002/cpe

Environment 2. AMD Athlon^{TM}64 X2 Dual Core Processor 4000+, Linux 3.2.0.51-generic-pae i686 athlon i386, gcc v4.6.3 with -O2 -mfpmath=sse -msse2, PAPI v5.1.1 and PERPI (pilp5 version).

In the following, Environment 1 is the default.

Accuracy is measured as the number of significant bits $\#_{sig}$ (see Relation (1)) in the floating-point significand. So, 53 is the maximum value we can expect from the *binary64* format. For this study, we determine the accuracy with higher-precision reference values precomputed by MPFR within our tools.

A reliable measure of the execution time is more difficult to obtain. Such measurements are not always reproducible because of many unwanted side effects (operating system, executing programs...). The most significant possible measures are provided here using two tools. The first one is the well known PAPI library (Performance Application Programming Interface) [32] which reads the hardware counters of cycles or instructions of an actual execution. The second software, PERPI [33], measures the numbers of cycles and instructions of one *ideal execution*, that is, one execution by a machine with infinite resources. This measure is more related to a performance potential than to the actual measure provided by PAPI. Using both tools allow us to present confident and complementary results. Others measures are also presented in [11].

5.2. Case studies

The first case study corresponds to existing compensated algorithms and is summarized with Table I and Table II. Table I gives data used for the summation of n values (case 1). Table II gives representative data used for the polynomial evaluation algorithms (cases 2, 3 and 4).

Case 1. SUM2 for the recursive summation of n values [14].
Case 2. COMPHORNER [3] and COMPHORNERDER [4] for the Horner's evaluation of $p_H(x) = (x - 0.75)^5(x - 1)^{11}$ and its derivative.
Case 3. COMPDECASTELJAU and COMPDECASTELJAUDER [29] for evaluating with de Casteljau's scheme $p_D(x) = (x - 0.75)^7(x - 1)$ and its derivative, written in the Bernstein basis.
Case 4. COMPCLENSHAWI and COMPCLENSHAWII [28] for evaluating with Clenshaw's scheme $p_C(x) = (x - 0.75)^7(x - 1)^{10}$ written in the Chebyshev basis.

Table I. Case studies and data for the summation (case 1).

Summation		
Data	# values	condition number
d_1	32×10^4	10^8
d_2	32×10^5	10^8
d_3	32×10^6	10^8
d_4	32×10^4	10^{16}
d_5	32×10^5	10^{16}
d_6	32×10^6	10^{16}
d_7	10^5	irrelevant
d_8	10^6	irrelevant

Table II. Case studies and data for polynomial evaluation with HORNER, CLENSHAW, and DECASTELJAU (cases 2, 3, 4).

Polynomial evaluations		
Data	# x	range
x_1	256	$\{0.85 : 0.95\}$ (uniform dist.)
x_2	256	$\{1.05 : 1.15\}$ (uniform dist.)
x_3	256	$\{0.35 : 0.45\}$ (uniform dist.)
x_4	256	$\{0.6 : 0.7\}$ (uniform dist.)
x_5	256	$\{1.8 : 1.9\}$ (uniform dist.)
x_6	256	$\{1.2 : 1.3\}$ (uniform dist.)
x_7	256	$\{0.73 : 0.74\}$ (uniform dist.)
x_8	256	$\{0.755 : 0.8\}$ (uniform dist.)
x_9	512	$\{0.68 : 1.15\}$ (uniform dist.)
x	1	irrelevant

5.3. Complete program transformation

We detail the complete program transformation of the Horner evaluation (case 2). Others cases will be summarized further.

DOI: 10.1002/cpe

Horner's polynomial evaluation case. We automatically compensate the Horner scheme and compare it with DDHORNER and COMPHORNER. The compensated algorithm and the data come from [3]. Let $p_H(x) = (x - 0.75)^5(x - 1)^{11}$ be evaluated with HORNER's scheme, where $x \in x_9$. Figure 7 shows the accuracy of this evaluation using HORNER (original), DDHORNER, COMPHORNER, and our automatically generated ACHORNER algorithm.

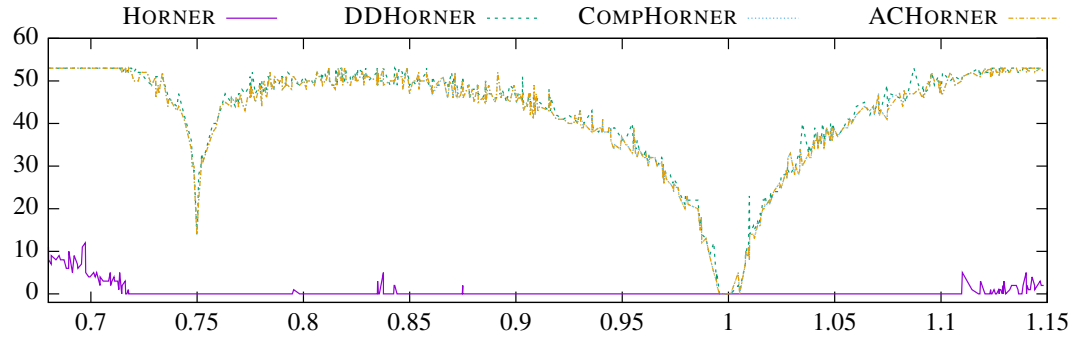


Figure 7. Number of significant bits $\#_{sig}$ (y axis) when evaluating $p_H(x) = (x - 0.75)^5(x - 1)^{11}$, where $x \in x_9$ (x axis) for HORNER, DDHORNER, COMPHORNER, and ACHORNER.

In each case, we measure the number of significant bits $\#_{sig}$. The original HORNER algorithm's accuracy is low since the evaluation is performed in the neighborhood of multiple roots: most of the time, there is no significant bit. The other algorithms yield better accuracy. Our automatically generated algorithm exhibits the same accuracy behavior as DDHORNER and COMPHORNER, which are twice as accurate as the original algorithm.

Table III. Performance measurements of the algorithms COMPHORNER, DDHORNER, and ACHORNER. Real values (PAPI) are the means of 10^6 measures. Ideal values are obtained with PERPI.

	PAPI			PERPI		
	instructions	cycles	IPC	instructions	cycles	IPC
COMPHORNER	532	277	1.99	566	62	9.12
DDHORNER	658	920	0.72	676	325	2.08
ACHORNER	553	303	1.82	581	77	7.54
AC/COMP	1.04	1.09	0.95	1.01	1.24	0.82
AC/DD	0.84	0.33	2.55	0.85	0.23	3.62

Table III shows the actual and ideal performances of the algorithms in terms of numbers of instructions, cycles, and instructions per cycle (IPC). The automatically generated algorithm has almost the same number of instructions and cycles as the compensated one. Moreover, Table III confirms that compensated algorithms benefit from more ILP than double-double ones. Even if the code of our generated algorithm is slightly different from the existing one [3], it appears here to be as accurate and efficient. Moreover, the multi-criteria optimization introduced in Section 4 will produce quite different algorithms, by trading off accuracy against speed.

Others cases. We now summarize our results for the case studies of Table I and Table II. They have been chosen such that the original algorithm returns no significant digit at the working precision, while all of them are recovered by the twice more accurate ones.

Figure 8 presents the differences, in terms of number of significant bits, between the automatically compensated algorithms (AC) and the compensated ones (COMP) or the double-double ones (DD). For example, the eleventh case concerns HORNER's evaluation of p_H for data x_1. The difference between the numbers of significant bits of ACHORNER and COMPHORNER is zero. The AC algorithm is as accurate as the existing compensated one. The difference with the DD algorithm is of one bit. Most of the other results exhibit a similar good behavior. The slight differences of

DOI: 10.1002/cpe

the last three data sets for the summation are due to the different effects of the sum length onto the compensated and double-double solutions: the accuracy bound of the former is quadratically affected by the length while being only linearly dependent in the double-double case. This appears only when the condition number and the sum length are large enough.

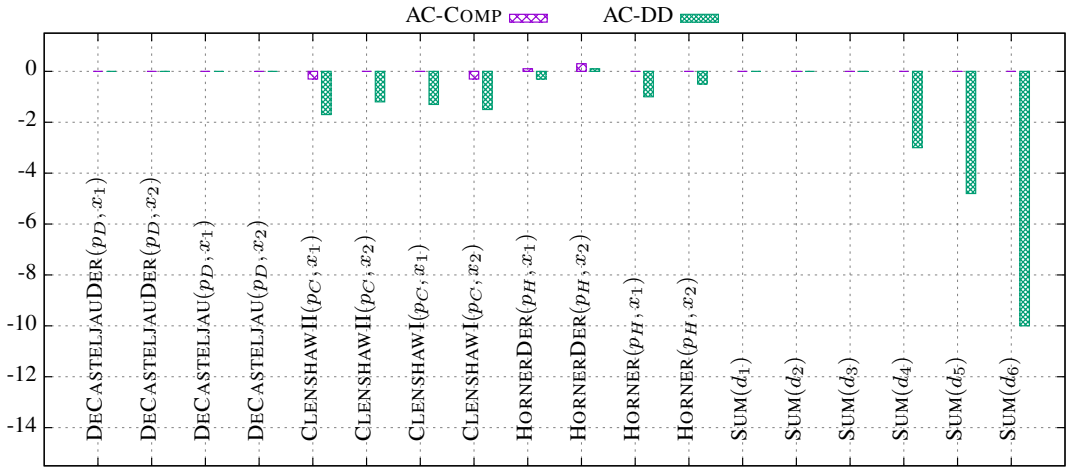
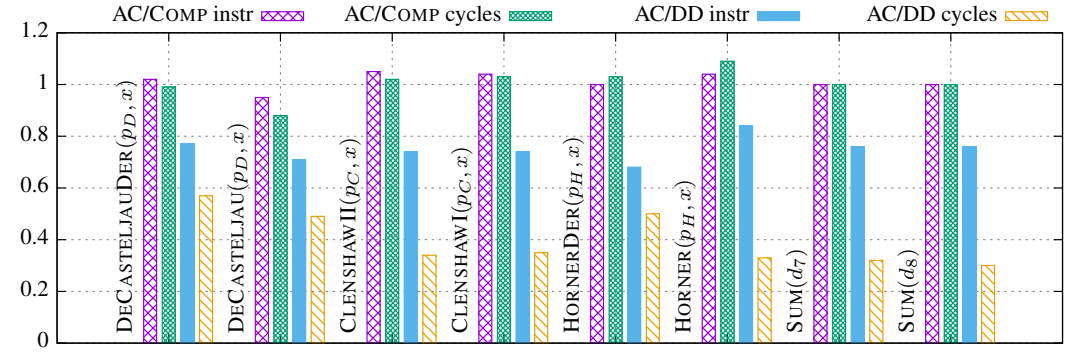


Figure 8. Differences of the numbers of significant bits $\#_{sig}$ for the automatically compensated (AC) algorithm versus the existing compensated algorithms (COMP), and the double-double (DD) ones.

Figure 9 presents the performance ratios between AC algorithms and existing COMP or DD ones. Here, PAPI measures the actual mean of 10^6 executions. PERPI ideal measures are presented in [11].

(a) Cycles and instruction ratios (the lower the better)



(b) IPC ratios (the higher the better)

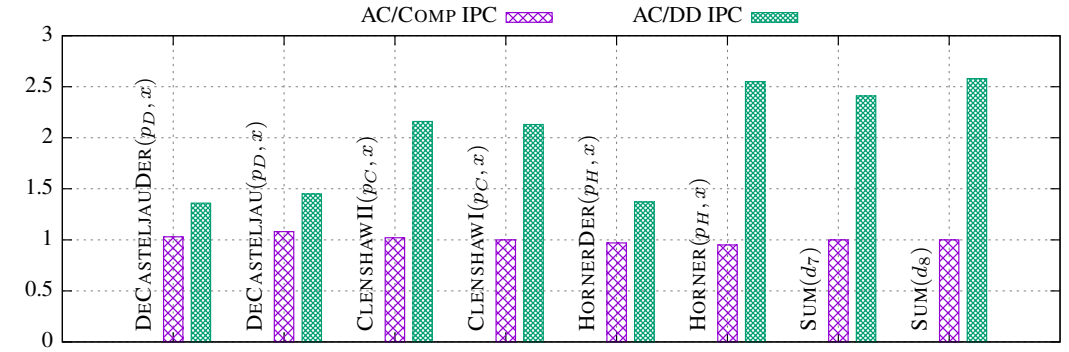


Figure 9. Performance ratios between automatically compensated algorithms (AC) and existing compensated (COMP) or double-double (DD) ones (mean of 10^6 values measured with PAPI).

DOI: 10.1002/cpe

Figure 9a shows the cycle and instruction ratios. For example, the rightmost plot is SUM(d_8). We see that the instruction ratio AC/COMP $= 1$. Compensated SUM2 and our automatically generated one share the same number of instructions. The instruction ratio compared to DD is 0.8 which means that 20% fewer instructions have been generated while returning the same accuracy. The compensated algorithms, original and generated, present a similar number of cycles that remarkably represents 30% of the DD execution ones. Figure 9b shows the ratio of the number of instructions per cycle. We observe that AC algorithms have the same interesting properties as the original compensated ones. Measurements also confirm the interest of compensated algorithms, exhibiting a better ILP potential than DD algorithms. Finally, we note that the ILP potential is not fully exploited in our experimental environment [11].

5.4. Partial program transformations

We now consider the partial program transformations from Section 4.2 and 4.3. We begin with a detailed example of code synthesis using the HORNER algorithm. Then we summarize the results for the case studies of Table I and Table II.

Horner's polynomial evaluation. Figure 10 summarizes the accuracy and performance improvements measured for all the transformed programs generated by the synthesis tool SyHD.

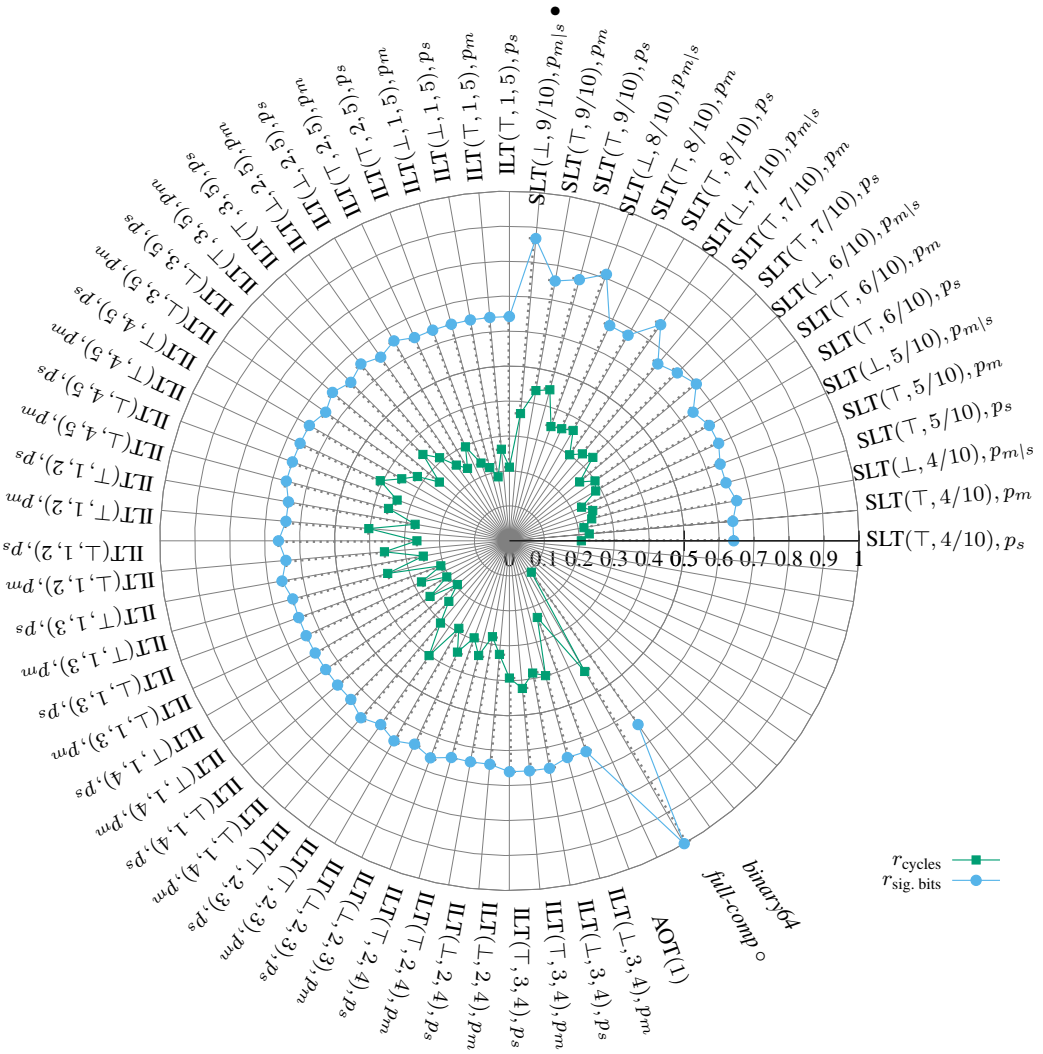


Figure 10. Results exploration in the case of HORNER's algorithm with data x_3 (r_{cycles} closer to 0 the better, $r_{\text{sig. bits}}$ closer to 1 the better, ∘ denotes the program returned with R_{acc}, and • the one returned with R_{exe}).

DOI: 10.1002/cpe

Figure 10 corresponds to the HORNER's algorithm running on data x_3 (256 values in $\{0.35 : 0.45\}$). Our transformation strategies (SLT, ILT, AOT) generate here 62 different programs. Figure 10 also displays how the accuracy and performance ratios ($r_{\text{sig. bits}}$ and r_{cycles}) vary according to the automatized transformations. We also add the original *binary64* program and the completely compensated *full-comp* generated by CoHD. We see that the 62 strategies of partial transformation manage to provide an equivalent significant bits ratio ($r_{\text{sig. bits}}$, closer to 1 the better) included from 0.6 to 0.7 and fail to achieve significant accuracy improvement compared to the original program (*binary64*), excepted for 5 SLTs strategies. The latter compensates for at least 70% (up to 90% for the best ratio) of the original program's computations, allowing to obtain from 77% to 88% of the accuracy of the completely compensated program *full-comp*. The cycles ratio (r_{cycles} closer to 0 the better) distribution is quite different. We recover what we would intuitively expect: the more the computations are compensated, the more the number of cycles increases.

Finally, the figure also points out the programs selected by the two trade-off selectors R_{acc} and R_{exe} (see relations (3) and (4)). Here SyHD default parameters apply (see Figure 6 and $\alpha = \beta$ for R_{exe}). R_{acc} selects the fully compensated program (*full-comp*, depicted by $\circ$) because it is the only one which achieves the highest level of accuracy. This is due to the ill-condition of our data as explained for Table IV. R_{exe} selects a program transformed following an SLT strategy: $\text{SLT}(\perp, 9/10), p_{m|s}$ (depicted by $\bullet$). This transformation was selected because it allows the highest level of accuracy in the shortest execution time than the fully compensated one. As expected, an other program can be selected if R_{exe} parameters was set differently.

Figure 11 gives the accuracy of the two selected programs from the Figure 10. It presents the number of significant bits when evaluating $p(x) = (x - 0.75)^5(x - 1)^{11}$ with the Horner's scheme and x in the x_3 dataset.

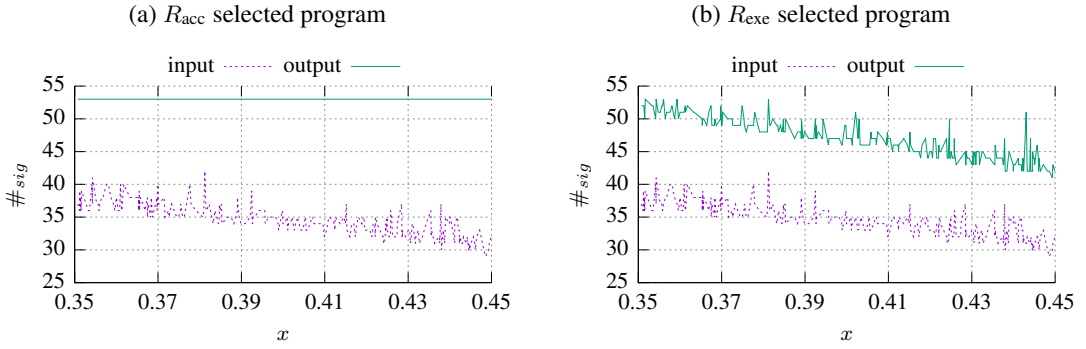


Figure 11. Accuracy (significant bits numbers) for (a) R_{acc} and (b) R_{exe} selection of the transformed HORNER's evaluation for the x_3 data (Table II, x axis).

Figure 11a compares the accuracy of the original and transformed programs when the R_{acc} selector is desired. Full accuracy is recovered for approximately 289 cycles which is more than 3 times faster than the double-double version. Figure 11b shows the accuracy behavior when the R_{exe} selector is required. By sacrificing accuracy (from 0 to 12 significant bits as shown by the "output" line in Figure 11b), we can save at least 13% of cycles for this polynomial evaluation (which now takes 251 cycles).

This example illustrates the potential of our approach. From Figure 11, we can consider the generation of programs presenting accuracy between the same level of accuracy as the original and the *full-comp* (completely compensated) programs. These programs could be generated by tweaking the selector parameters by requesting some different accuracy and execution time trade-offs.

Other cases. Table IV summarizes the code synthesis results obtained for all the case studies of Table I and Table II. Default SyHD parameters apply except for $\text{SUM}(d_1)$ and $\text{SUM}(d_2)$ where ν is respectively set to 1,000 and 10,000. The bold results correspond to the ones depicted in Figures 11a

DOI: 10.1002/cpe

(*full-comp*, also depicted by ∘ in Figure 10) and 11b (SLT$(\perp, 9/10)$ with $p_{m|s}$, also depicted by • in Figure 10).

Table IV. Results summarize of code synthesis on all the case studies (Table I and Table II). The bold results correspond to the ones depicted in Figures 11a (*full-comp*, also depicted by ∘ in Figure 10) and 11b (SLT$(\perp, 9/10)$ with $p_{m|s}$, also depicted by • in Figure 10).

Algorithms and data	Optimization criteria Environment 1		Optimization criteria Environment 2	
	R_{acc}	R_{exe}	R_{acc}	R_{exe}
SUM(d_1)	*full-comp*	fail †	*full-comp*	fail †
SUM(d_2)	*full-comp*	fail †	*full-comp*	fail †
HORNER(p_H, x_3)	**full-comp**	**SLT**$(\perp, \mathbf{9/10}), \mathbf{p_{m\|s}}$	*full-comp*	SLT$(\top, 9/10), p_m$
HORNER(p_H, x_4)	*full-comp*	SLT$(\top, 9/10), p_s$	*full-comp*	SLT$(\top, 9/10), p_m$
HORNERDER(p_H, x_3)	*full-comp*	fail ⋆†	*full-comp*	fail †
HORNERDER(p_H, x_4)	*full-comp*	SLT$(\top, 9/10), p_s$	*full-comp*	SLT$(\top, 9/10), p_s$
CLENSHAWI(p_C, x_3)	*full-comp*	SLT$(\perp, 8/10), p_{m\|s}$	*full-comp*	SLT$(\perp, 7/10), p_{m\|s}$
CLENSHAWI(p_C, x_5)	*full-comp*	SLT$(\top, 8/10), p_s$	*full-comp*	SLT$(\perp, 8/10), p_{m\|s}$
CLENSHAWII(p_C, x_3)	*full-comp*	SLT$(\perp, 8/10), p_{m\|s}$	*full-comp*	SLT$(\perp, 8/10), p_{m\|s}$
CLENSHAWII(p_C, x_5)	*full-comp*	SLT$(\top, 8/10), p_s$	*full-comp*	SLT$(\perp, 8/10), p_{m\|s}$
DECASTELJAU(p_D, x_7)	*full-comp*	fail ‡	*full-comp*	fail ‡
DECASTELJAU(p_D, x_8)	*full-comp*	fail ‡	*full-comp*	fail ‡
DECASTELJAUDER(p_D, x_7)	*full-comp*	fail ‡	*full-comp*	fail ‡
DECASTELJAUDER(p_D, x_8)	*full-comp*	fail ‡	*full-comp*	fail ‡

For all these cases, the R_{acc} selector returns the completely compensated algorithm as the transformed solution. This case is explained by the ill-condition of the tested case: no transformation except the fully compensated one yields a fully accurate compensation. Less ill-conditioned data would be transformed differently by R_{acc}. The R_{exe} selector only yields partially compensated programs with SLT transformation mainly because of the ill-condition of the data. Nevertheless, other partial transformations could give an appropriate result if R_{exe} was set differently or if data was less ill-conditioned.

Table IV also displays some failures, denoted by † or ‡ when selecting a program with R_{exe}. The following notes explain them.

- † This failure is raised because the accuracy requirement is not reached. As defined in the requirements of the code synthesis success (Section 4), the transformed program must improve the accuracy for each desired datum. This requirement is very strong and we could modify it (to make it customizable) to accept that the optimization as no effect on accuracy for some data. We could also customize the R_{exe} selector in order to accept a less accurate trade-off by setting $\alpha = \beta/2$ for example. By doing so, the case denoted by a ⋆ returns the transformed program obtained with the strategy SLT$(\perp, 7/10), p_{m|s}$.
- ‡ This failure is raised because of a lack of performance. DECASTELJAU's algorithms are tiny codes (a loop of 8 iterations containing a few floating-point operations), and this explains that every partially compensated program doesn't perform better than the fully compensated one. In such cases, failure can be avoided by returning this latter.

Finally, we claim that this code synthesis is processed in a reasonable amount of time. It takes from about 1 minute (Environment 1) to 1.5 minute (Environment 2) for the polynomial evaluation cases. In these cases, the code synthesis has to explore more than 60 different programs, over 256 data to measure accuracy, and between 100,000 and 1,000,000 executions to measure performance. For the summation cases, with larger dataset and longer execution time, the code synthesis can take up to 2 minutes (Environment 2) to perform the program selection.

5.5. Iterative refinement for linear system solving

Finally, we consider the classical iterative refinement technique used for improving the accuracy of the solution of a linear system $Ax = b$. Algorithm 9 performs such computation [34]. When

DOI: 10.1002/cpe

computing the residual in twice the working precision (line 4), iterative refinement returns the fully accurate solution of not too ill conditioned linear systems [35, Chap. 12]. The BLAS [36] (Basic Linear Algebra Subroutines) routines are convenient here since they are ready to use and well-tuned for performance. The XBLAS library [37] (extended and mixed precision BLAS) relies on BLAS and also provides some routines performing double precision inner computation, e.g. iterative refinement. The goal of this experiment is to compare our transformed code computing the residual to the one of the XBLAS.

Require: An $n \times n$ matrix A and an $n \times 1$ vector b
Ensure: A solution vector $x^{(i)}$ approximating x in $Ax = b$
1: Solve $Ax^{(1)} = b$
2: $i = 1$
3: **repeat**
4: Compute residual $r^{(i)} = Ax^{(i)} - b$
5: Solve $Ax^{(i+1)} = r^{(i)}$
6: Update $x^{(i+1)} = x^{(i)} - x^{(i+1)}$
7: $i = i + 1$
8: **until** $x^{(i)}$ is "accurate enough"
9: **return** $x^{(i)}$

Algorithm 9: Classical iterative refinement.

Figure 12 shows the iteration number and the accuracy of Algorithm 9 when the matrix A is $\texttt{orthog}(25)$ or $\texttt{gfpp}(50)$ for three variants of the algorithm: the residual is computed with BLAS, XBLAS, or the AC generated code. The elements of the vector b are uniformly distributed in $[0, 1]$. The stopping condition (line 8) is satisfied when $||x^{(i+1)} - x^{(i)}||/||x^{(i+1)}|| \leqslant 4u$, where $u = 2^{-53}$. The data of this experiment are obtained with the Matrix Computation Toolbox [38] and chosen similarly to [35, pp. 239–240].

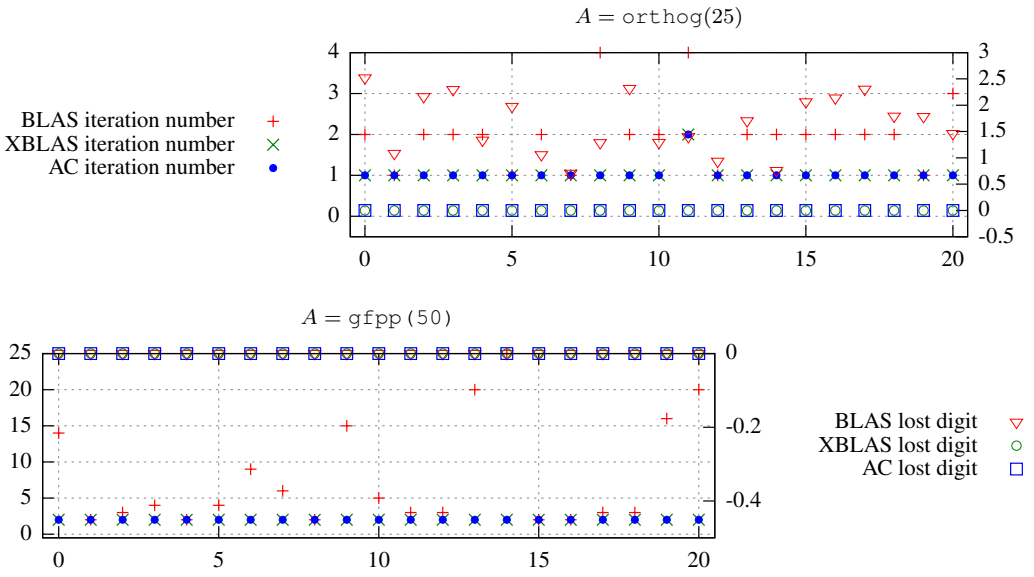


Figure 12. 20 executions (x axis) of Algorithm 9 where A is matrix $\texttt{orthog}(25)$ or $\texttt{gfpp}(50)$. The y axis shows the iteration number until termination (left) and the number of lost digits (right).

Figure 12 exhibits that the XBLAS implementation and AC generated code share both the same performance and accuracy. Table V presents the average cycle measure of 20 executions of Algorithm 9. It confirms that the XBLAS and AC codes present equivalent performance and perform

DOI: 10.1002/cpe

refinement with a similar number of cycles. In order to beat XBLAS performance, we will focus on a better code transformation from CoHD as discussed in Section 6.

Table V. Minimal, average and maximal kilocycles (PAPI) of 20 executions of Algorithm 9.

	XBLAS			AC		
	min.	**ave.**	max.	min.	**ave.**	max.
`orthog`(25)	483	**552**	790	496	**568**	816
`gfpp`(50)	109	**114**	180	112	**116**	186

6. CONCLUSIONS AND PERSPECTIVES

In this article we discussed the automated transformation of programs using floating-point arithmetic. We propose a new method to compensate automatically the floating-point errors of the computations, and to improve the accuracy without greatly impacting execution time. The automatic transformation produces some compensated algorithms which are as accurate and efficient as the ones derived by hand on a case-by-case basis. Moreover, we propose a partial compensation to reduce further the execution time overhead by trading off performance and accuracy. Trade-offs are ensured by code synthesis and multi-criteria program optimization. We developed strategies to generate partially transformed programs, as well as a method to find the best transformation satisfying an execution time or accuracy constraints. The efficiency of our approach has been illustrated in various case studies.

We now need to validate this approach and the tools by testing them on real and more sophisticated programs. To achieve this, we have to support floating-point divisions, square-roots, and the elementary functions. Possible solutions for improving our tools are numerous. First, we would like to add more transformation strategies in order to generate new partially compensated program patterns, or other optimization constraints such as code size or energy consumption. Then, we aim to develop a dynamic method for Algorithm 8 to find a program that satisfies the optimization constraints, for example, by using a genetic algorithm. We would also like to yield accuracy by adding an unbounded compensation step applying the step proposed in this paper several times, or by using expansions and compensation. Finally, in order to facilitate these perspectives, we plan to implement the CoHD transformations into the GCC compiler.

REFERENCES

1. Dekker TJ. A floating-point technique for extending the available precision. *Numer. Math.* 1971; **18**:224–242.
2. Shewchuk JR. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Springer Disc. & Comp. Geometry* 1997; **18**(3):305–363.
3. Graillat S, Langlois P, Louvet N. Algorithms for accurate, validated and fast polynomial evaluation. *Japan Journal of Industrial and Applied Mathematics* 2009; **26**(2-3):191–214, doi:10.1007/BF03186531.
4. Jiang H, Graillat S, Hu C, Li S, Liao X, Chang L, Su F. Accurate evaluation of the k-th derivative of a polynomial and its application. *Journal of Computational and Applied Mathematics* 2013; **243**:28–47, doi:10.1016/j.cam.2012.11.008. URL `http://www.sciencedirect.com/science/article/pii/S0377042712005018`.
5. Langlois P. Automatic linear correction of rounding errors. *BIT* 2001; **41**(3):515–539, doi:10.1023/A:1021919329342.
6. Fousse L, Hanrot G, Lefèvre V, Pélissier P, Zimmermann P. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Software* 2007; **33**(2).
7. Hida Y, Li XS, Bailey DH. Algorithms for quad-double precision floating point arithmetic. *Proc. of the 15th IEEE Symposium on Computer Arithmetic*, IEEE Computer Society Press, Los Alamitos, CA, USA, 2001; 155–162.
8. Blair M, Obenski S, Bridickas P. Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia. *Report GAO/IMTEC-92-26*, Information Management and Technology Division, United States General Accounting Office, Washington, D.C. feb 1992.
9. Lions JL, Hergott R, Humbert B, Lefort E. Ariane 5 flight 501 failure, report by the inquiry board. *Technical Report*, European Space Agency 1996.
10. Langlois P, Martel M, Thévenoux L. Automatic code transformation to optimize accuracy and speed in floating-point arithmetic. *Proc. of the 15th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, 2012.

DOI: 10.1002/cpe

11. Thévenoux L, Langlois P, Martel M. Automatic source-to-source error compensation of floating-point programs. *Proc. of the 18th IEEE International Conference on Computational Science and Engineering*, 2015. URL `https://hal.archives-ouvertes.fr/hal-01158399`.
12. Muller JM, Brisebarre N, de Dinechin F, Jeannerod CP, Lefèvre V, Melquiond G, Revol N, Stehlé D, Torres S. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.
13. Appel AW. *Modern Compiler Implementation: In ML*. Cambridge University Press: New York, NY, USA, 1998.
14. Ogita T, Rump SM, Oishi S. Accurate sum and dot product. *SIAM J. Sci. Comput.* Jun 2005; **26**(6):1955–1988, doi:10.1137/030601818.
15. Langlois P, Louvet N. More instruction level parallelism explains the actual efficiency of compensated algorithms 2007. URL `http://hal.archives-ouvertes.fr/hal-00165020`.
16. Rubio-González C, Nguyen C, Nguyen D, Demmel J, Kahan W, Sen K, Bailey DH, Iancu C, Hough D. Precimonius: Tuning assistant for floating-point precision. *Proc. of the SC13's International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, Colorado, USA, 2013.
17. Saito T, Ishiwata E, Hasegawa H. Development of quadruple precision arithmetic toolbox qupat on scilab. *Computational Science and Its Applications, ICCSA 2010*, *Lecture Notes in Computer Science*, vol. 6017. Springer Berlin Heidelberg, 2010; 60–70, doi:10.1007/978-3-642-12165-4_5.
18. Priest DM. On properties of floating point arithmetics: Numerical stability and the cost of accurate computations. PhD Thesis, University of California at Berkeley, Berkeley, CA, USA 1992. UMI Order No. GAX93-30692.
19. Ioualalen A, Martel M. Synthesizing accurate floating-point formulas. *Proc. of the 24th IEEE International Conference on Application-specific Systems, Architectures and Processors*, IEEE: Washington D.C., United States, 2013. URL `https://hal.archives-ouvertes.fr/hal-00835736`.
20. Panchekha P, Sanchez-Stern A, Wilcox JR, Tatlock Z. Automatically improving accuracy for floating point expressions. *Proc. of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, ACM: New York, NY, USA, 2015; 1–11, doi:10.1145/2737924.2737959.
21. de Dinechin F, Lauter CQ, Melquiond G. Certifying floating-point implementations using gappa. *CoRR* 2008; **abs/0801.0523**. URL `http://arxiv.org/abs/0801.0523`.
22. LogiCal Project. *The Coq proof assistant reference manual* 2004. URL `http://coq.inria.fr`, version 8.0.
23. Goubault E, Putot S. Static analysis of numerical algorithms. *Proc. of the 13th International Conference on Static Analysis*, SAS'06, Springer-Verlag: Berlin, Heidelberg, 2006; 18–34, doi:10.1007/11823230_3.
24. Society IC. *IEEE 754 Standard for Floating-Point Arithmetic*. The Institute of Electrical and Electronics Engineers, Inc., 2008, doi:10.1109/IEEESTD.2008.4610935.
25. Knuth DE. *The Art of Computer Programming: Seminumerical Algorithms*, vol. 2. 3rd edn., Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1997.
26. Møller O. Quasi double-precision in floating-point addition. *BIT* 1965; **5**:37–50, doi:10.1007/BF01975722.
27. Lauter CQ. Basic building blocks for a triple-double intermediate format. *Technical Report RR-5702*, Inria 2005. URL `http://hal.inria.fr/inria-00070314`.
28. Jiang H, Barrio R, Li H, Liao X, Cheng L, Su F. Accurate evaluation of a polynomial in Chebyshev form. *Appl. Math. Comput.* 2011; **217**(23):9702–9716, doi:10.1016/j.amc.2011.04.054. URL `http://www.sciencedirect.com/science/article/pii/S0096300311006242`.
29. Jiang H, Li S, Cheng L, Su F. Accurate evaluation of a polynomial and its derivative in Bernstein form. *Computers Math. Applic.* 2010; **60**(3):744–755, doi:10.1016/j.camwa.2010.05.021.
30. Goossens B, Langlois P, Parello D, Porada K. Computing time for summation algorithm: Less hazard and more scientific research. *Numerical Sofware: Design, Analysis and Verification*, Santander, Spain, 2012. URL `http://hal-lirmm.ccsd.cnrs.fr/lirmm-00835508`.
31. Casse H. Frontc 3.4: an OCaml C parser and pretty-printer 2000. TRACES Research Group, Institut de Recherche en Informatique de Toulouse (IRIT), France.
32. Mucci PJ, Browne S, Deane C, Ho G. PAPI: A portable interface to hardware performance counters. *Proc. of the Department of Defense HPCMP Users Group Conference*, 1999; 7–10.
33. Goossens B, Langlois P, Parello D, Petit E. PerPI: A tool to measure instruction level parallelism. *Applied Parallel and Scientific Computing*, *Lecture Notes in Computer Science*, vol. 7133, Jónasson K (ed.). Springer Berlin Heidelberg, 2012; 270–281, doi:10.1007/978-3-642-28151-8_27.
34. Demmel JW, Hida Y, Kahan W, Li S, Mukherjee S, Riedy EJ. Error bounds from extra-precise iterative refinement. *ACM Trans. Math. Software* Jun 2006; **32**(2):325–351.
35. Higham NJ. *Accuracy and Stability of Numerical Algorithms*. 2nd edn., Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2002.
36. Blackford SL, Demmel J, Dongarra J, Duff I, Hammarling S, Henry G, Heroux M, Kaufman L, Lumsdaine A, Petitet A, *et al.*. An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Software* Jun 2002; **28**(2):135–151.
37. Li XS, Demmel JW, Bailey DH, Henry G, Hida Y, Iskandar J, Kahan W, Kang SY, Kapur A, Martin MC, *et al.*. Design, implementation and testing of extended and mixed precision blas. *ACM Trans. Math. Softw.* Jun 2002; **28**(2):152–205.
38. Higham NJ. Matrix Computation Toolbox. URL `http://www.ma.man.ac.uk/~higham/mctoolbox`.