Automatic Source-to-Source Error Compensation of Floating-Point Programs

Laurent Thévenoux
Inria – Laboratoire LIP
(CNRS, ENS de Lyon, Inria, UCBL)
Univ. de Lyon, France
Email: laurent.thevenoux@inria.fr

Philippe Langlois and Matthieu Martel
Univ. Perpignan Via Domitia, DALI, F-66860, Perpignan, France
Univ. Montpellier II, LIRMM, UMR 5506, F-34095, Montpellier, France
CNRS, LIRMM, UMR 5506, F-34095, Montpellier, France
Email: {langlois, matthieu.martel}@univ-perp.fr

***Abstract*—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 as, for instance, compensated algorithms, more precise computation with double-double or similar libraries. Our objective is to automatically improve the numerical quality of a numerical program with the smallest impact on its performances. We define and implement source code transformation 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 than the implementations of compensated algorithms when the latter exist.**

I. 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, as for instance expansions [4], [23], compensations [7], [10], differential methods [14] or extended precision arithmetic using multiple-precision libraries [5], [8]. Nevertheless, bugs from numerical failures are numerous and well known [2], [18]. This illustrates that these improvement techniques are not known enough outside the floating-point arithmetic community, 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 [8] or, less easily, by compensating the floating-point operations with error-free transformations (EFT) [7]. The latter transformations are difficult to implement without a preliminary manual step to define the modified algorithm.

We present a method that allows a non floating-point expert to improve the numerical accuracy of his program without impacting too much the execution time. Our approach facilitates the numerical accuracy improvement by automating compensation process. Even if we not provide error bounds on the processed algorithms, our approach takes advantage of a fast program transformation, available for a large community of developers. So, we propose to automatically introduce at compile-time a compensation step by using error-free transformations. We have developed a tool to parse C programs and generate a new C code with a compensated treatment: floating-point operations $\pm$ and $\times$ are replaced by their respective error-free TwoSum and TwoProduct algorithms [21, Chap. 4]. The main advantage of this method compared to operator overloading is to benefit from code optimizations and an efficient code generation. Program transformation is strongly motivated by our perspectives for the multi-criteria optimization of programs [25]. These optimizations will allow to trade-off between accuracy and execution time. Programs will be partially compensated by using transformation strategies to meet some time and accuracy constraints which could be difficult to reach with operator overloading.

To demonstrate the efficiency of this approach, we compare our automatically transformed algorithms to existing compensated ones such as floating-point summation [22] and polynomial evaluation [7], [10]. The goal of this demonstration is to recover automatically the same results in terms of accuracy and execution time. Compensation is known to be a good choice to benefit from the good instruction level parallelism (ILP) of compensated algorithms compared to the ones derived using fixed-length expansions such as double-double or quad-double [8], [15]. Results for the automatically transformed algorithms, both in terms of accuracy and execution time, are shown to be very close to the results for the implementation of the studied compensated algorithms.

This article is organized as follows. Section II introduces background material on floating-point arithmetic, error-free transformations, and accuracy improvement techniques like double-double arithmetic and compensation. The core of this article is Section III, where we present our automatic code transformation to optimize the accuracy of floating-point computations with the smallest execution time overhead. In Section IV, we present some experimental results to illustrate the interesting behavior of our approach compared to existing ones. Conclusion and perspectives are proposed in Section V.

II. Preliminaries

In this section we recall classical notations to deal with 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 present how to exploit these EFTs with expansions and compensations.

A. IEEE Floating-Point Arithmetic

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

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

where $s \in \{0,1\}$ is the sign, $m = \sum_{i=0}^{p-1} d_i \beta^{-i} = (d_0.d_1 d_2 \cdots d_{p-1})_\beta$ is the mantissa (with $d_i \in \{0,1,\ldots,\beta -$

$1\}$), and e is the exponent. The IEEE 754-2008 standard [24] defines such numbers for several formats, that is, for various pairs (β, p). It also defines rounding modes, and the semantics of the basic operations $\pm, \times, \div, \sqrt{\cdot}$.

Notation and assumptions. Throughout the paper, all computations are performed in *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, in the *binary64* format floating-point numbers are expressed with $\beta = 2$ over 64 bits including $p = 53$ bits, 11 for the exponent e, and 1 for the sign s.
- $fl(\cdot)$ denotes the result of a floating-point computation where every operation inside the parenthesis is performed in the working precision and the *round-to-nearest* mode.
- $ulp(x)$ is the floating-point value of the unit in the last place of x defined by $ulp(x) = 2^e \cdot 2^{1-p}$. Let $\widehat{x} = fl(x)$ for a real number x. We have $|x - \widehat{x}| \leqslant ulp(\widehat{x})/2$.

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

$$\#_{sig}(\widehat{x}) = -log_2(E_{rel}(\widehat{x})),$$

where $E_{rel}(\widehat{x})$ is the relative error defined by:

$$E_{rel}(\widehat{x}) = \frac{|x - \widehat{x}|}{|x|}, \quad x \neq 0.$$

B. Error-Free Transformations

Error-free transformations (EFT) provide lossless transformations of basic floating-point operations $\circ \in \{+, -, \times\}$. Let $a, b \in \mathbb{F}$ and $\widehat{x} = fl(a \circ b)$. There exists a floating-point value $y = a \circ b - \widehat{x}$ such that $a \circ b = \widehat{x} + y$. We have $|y| \leqslant ulp(\widehat{x})/2$. Hence $\widehat{x}$ (resp. y) is the upper (resp. lower) part of $a \circ b$ and no digit of $\widehat{x}$ overlaps with y. The practical interest of EFTs comes from Algorithms 1, 2, 4, and 5 which exactly compute in floating-point arithmetic the error term y for the sum and the product.

$x \leftarrow fl(a + b)$ ▷ $|a| \geqslant |b|$
$y \leftarrow fl((a - x) + b)$
return $[x,\ y]$

Algorithm 1: FASTTWOSUM($a,\ b$) [Dekker, 1971].

$x \leftarrow fl(a + b)$
$z \leftarrow fl(x - a)$
$y \leftarrow fl((a - (x - z)) + (b - z))$
return $[x,\ y]$

Algorithm 2: TWOSUM($a,\ b$) [Møller, 1965 and Knuth, 1969].

Algorithms 1 and 2, respectively introduced by Dekker [4] and Knuth [12, Chap. 4] and Møller [19], provide the error of floating-point addition. The TWOSUM algorithm requires 6 floating-point operations (flop) instead of 3 for FASTTWOSUM, but does not require a preliminary comparison of a and b.

$c \leftarrow fl(f \times a)$ ▷ $f = 2^{\lceil p/2 \rceil} + 1$
$a_H \leftarrow fl(c - (c - a))$
$a_L \leftarrow fl(a - a_H)$
return $[a_H,\ a_L]$

Algorithm 3: SPLIT(a) [Veltkamp, 1968].

$x \leftarrow fl(a \times b)$
$[a_H,\ a_L] = \text{SPLIT}(a)$
$[b_H,\ b_L] = \text{SPLIT}(b)$
$y \leftarrow fl(a_L \times b_L - (((x - a_H \times b_H) - a_L \times b_H) - a_H \times b_L))$
return $[x,\ y]$

Algorithm 4: TWOPRODUCT($a,\ b$) [Dekker, 1971].

Algorithm 3, due to Veltkamp [4], splits a binary floating-point number into two floating-point numbers containing the upper and lower parts. It is used in Algorithm 4, introduced by Dekker [4], to compute the EFT of a product for the cost of 17 flops.

$x \leftarrow fl(a \times b)$
$y \leftarrow fl(\text{FMA}(a, b, -x))$
return $[x,\ y]$

Algorithm 5: TWOPRODUCTFMA($a,\ b$).

Some processors have a *fused multiply-add* (FMA) instruction which evaluates expressions such as $a \times b \pm c$ with a single rounding error. Algorithm 5 takes advantage of this instruction to compute the exact product of two floating-point numbers much faster, namely with 2 flops instead of 17 flops with TWOPRODUCT.

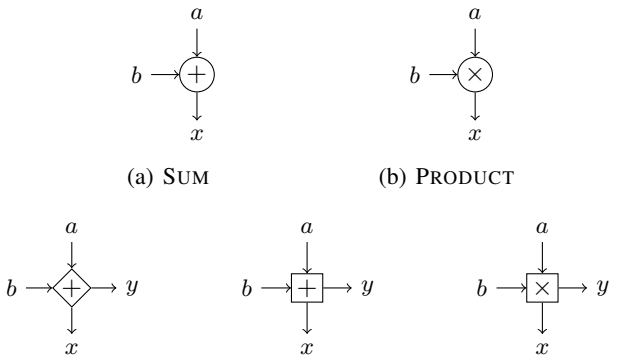
Table I presents the number of operations and the depth of the dependency graph (that is, the critical path in the EFT data flow graph) for each of the output values x and y of these algorithms. It is shown in [13] that TWOSUM is optimal, both in terms of the number of operations and the depth of the dependency graph.

EFT algorithm	flop		depth	
	x	y	x	y
FASTTWOSUM	1	3	1	3
TWOSUM	1	6	1	5
TWOPRODUCT	1	17	1	8
TWOPRODUCTFMA	1	2	1	2

TABLE I: Floating-point operation count and dependency graph depth for various EFTs.

The result $x = fl(a \circ b)$ is computed and available after only one floating-point operation. Moreover, the computation of y exposes some parallelism which can be exploited and, therefore, explains the efficiency of the algorithms [15].

Figure 1 defines diagrams for floating-point operations $\pm$ and $\times$, and for their EFTs. It allows us to graphically represent transformation algorithms as basic computational blocks.



(a) SUM (b) PRODUCT

(c) FASTTWOSUM (d) TWOSUM (e) TWOPRODUCT

Fig. 1: Diagrams for basic floating-point operations (a), (b) and EFT algorithms (c), (d), and (e).

C. Double-Double and Compensated Algorithms

We focus now on two methods using these EFTs to double the accuracy: double-double expansions and compensations. Then we recall why compensated algorithms are more efficient than double-double algorithms.

Double-double expansions. We present here the algorithms by Briggs, Kahan, and Bailey used in the QD library [8]. Let a, a_H and a_L be floating-point numbers of precision p. The corresponding double-double number of a is the unevaluated sum $a_H + a_L$ where a_H, and a_L do not overlap: $|a_L| \leqslant ulp(a_H)/2$. Double-double arithmetic simulates computations with precision $2p$. Proofs are detailed in [17].

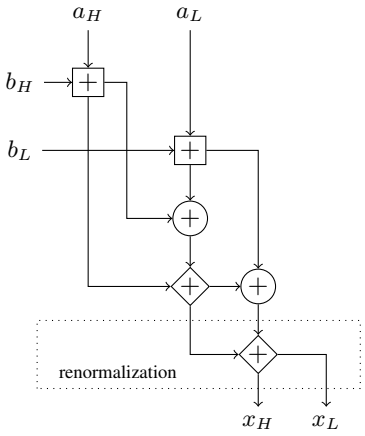
Algorithms 6 and 7 compute the sum and the product of two double-double numbers more accurately than Dekker's algorithms [4]. Double-double algorithms need a step of renormalization to guarantee $|x_l| \leqslant ulp(x_H)/2$. This step is insured by a FASTTWOSUM EFT and is represented by dotted boxes in Figure 2.

$[r_H,\ r_L] = \text{TWOSUM}(a_H,\ b_H)$
$[s_H,\ s_L] = \text{TWOSUM}(a_L,\ b_L)$
$c \leftarrow fl(r_L + s_H)$
$[u_H,\ u_L] = \text{FASTTWOSUM}(r_H,\ c)$
$w \leftarrow fl(s_L + u_L)$
return $[x_H,\ x_L] = \text{FASTTWOSUM}(u_H,\ w)$

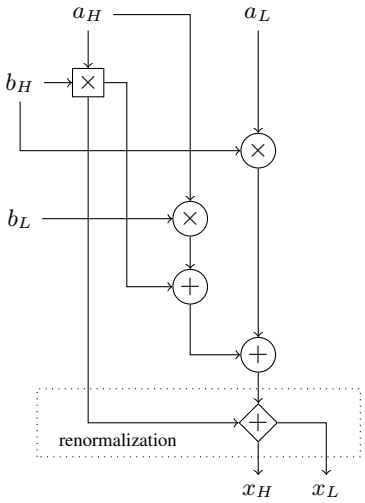
Algorithm 6: DD_TWOSUM($a_H,\ a_L,\ b_H,\ b_L$), double-double sum of two DD numbers [QD library, 2000].

$[r_H,\ r_L] = \text{TWOPRODUCT}(a_H,\ b_H)$
$r_L \leftarrow fl(r_L + (a_H \times b_L))$
$r_L \leftarrow fl(r_L + (a_L \times b_H))$
return $[x_H,\ x_L] = \text{FASTTWOSUM}(r_H,\ r_L)$

Algorithm 7: DD_TWOPRODUCT($a_H,\ a_L,\ b_H,\ b_L$), double-double product of two DD numbers [QD library, 2000].



(a) DD_TWOSUM.



(b) DD_TWOPRODUCT.

Fig. 2: Diagrams for Algorithms 6 and 7.

In practice, double-double algorithms can be simply used by overloading the basic operations as for example in Algorithm 8, which is the double-double version of SUM, the classical recursive algorithm to evaluate $a_1 + a_2 + \cdots + a_n$.

Compensated algorithms. As double-double algorithms, compensated algorithms can double the accuracy. We focus here on this class of algorithms. We already mentioned that double-double algorithms are easy to derive. On the contrary, compensated algorithms have been, up to now, defined case by case and by experts of rounding error analysis [7], [9], [10], [11], [22]. For example the compensated Algorithm 9, SUM2 [22], returns a twice more accurate sum.

Double-double versus compensation. Previous double-double and compensated sums provide roughly the same accuracy. How do they compare in terms of computing time? Algorithm 9 needs $7n - 6$ flops compared to $n - 1$ for the original SUM algorithm. The double-double summation implementation by Algorithm 8 needs $10n - 9$ flops, that is, almost 1.43 more floating-point operations than the compensated algorithm.

$s_H \leftarrow a_1$
$s_L \leftarrow 0$
for $i = 2 : n$ **do**
  $[s_H,\ s_L] = \text{DD\_TwoSum}(s_H,\ s_L,\ a_i,\ 0)$
end for
return s_H

Algorithm 8: SumDD($a_1,\ a_2,\ \ldots,\ a_n$), double-double classical recursive summation.

$s \leftarrow a_1$
$e \leftarrow 0$
for $i = 2 : n$ **do**
  $[s,\ \epsilon] = \text{TwoSum}(s,\ a_i)$
  $e \leftarrow fl(e + \epsilon)$
end for
return $fl(s + e)$

Algorithm 9: Sum2($a_1,\ a_2,\ \ldots,\ a_n$), compensated classical recursive summation [Rump, Ogita, and Oishi, 2005].

Now, let us consider the instruction level parallelism by inspecting the number of instructions which could be simultaneously executed per one cycle (IPC). In the case of the classical Sum algorithm, each iteration performs one floating-point operation. Each iteration can be followed immediately by the next iteration, so IPC(Sum) $= (n-1)/n \approx 1$. With the SumDD algorithm, each iteration of the loop contains 10 operations versus 7 for the Sum2 algorithm. Nevertheless, the main difference between both algorithms is in the parallelization of the loop iterations. The SumDD algorithm suffers from renormalization, and one iteration may only be followed by the next one with the latency of 7 floating-point operations, so IPC(SumDD) $= (10n-9)/(7n-5) \approx 1.42$. The Sum2 algorithm does not suffer from such drawbacks and iterations can be executed with a latency of only one flop: IPC(Sum2) $= (7n-6)/(n+5) \approx 7$. So Sum2 benefits from a seven times higher ILP. A detailed analysis has been presented in [16].

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

III. Automatic Code Transformation

We present how to improve accuracy thanks to code transformation. Experimental results are presented in Section IV.

A. Improving Accuracy: Methodology

Our code transformation automatically compensates programs and follows the next three steps.

1) First, detect floating-point computations sequences. A sequence is the set $\mathscr{S}$ of dataflow dependent operations required to obtain one or several results.
2) Then for each sequence $\mathscr{S}_i$ compute the error terms and accumulate them beside the original computation sequence by (a) replacing floating-point operations by the corresponding EFTs, and (b) accumulating error terms following Algorithms 10 and 11 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) Finally close 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 a result of $\mathscr{S}_i$.

B. Compensated operators

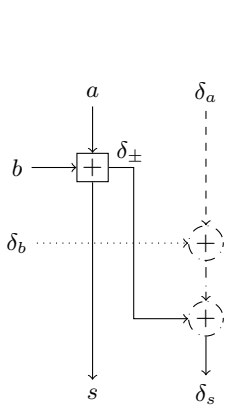
Algorithms 10 and 11 allow us to automatically compensate for the error of basic floating-point operations. Inputs are now compensated numbers.

$[s,\ \delta_\pm] = \text{TwoSum}(a,\ b)$
$\delta_s \leftarrow fl((\delta_a + \delta_b) + \delta_\pm)$
return $\langle s,\ \delta_s \rangle$

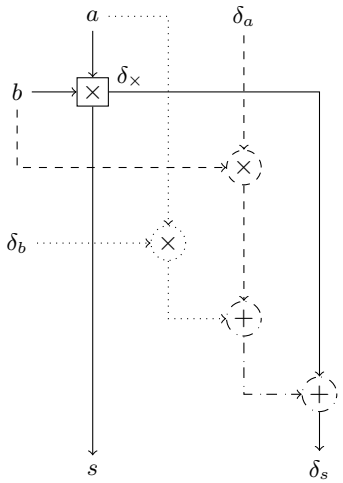
Algorithm 10: AC_TwoSum($\langle a,\ \delta_a \rangle,\ \langle b,\ \delta_b \rangle$), automatically compensated sum of two *compensated numbers*.

$[s,\ \delta_\times] = \text{TwoProduct}(a,\ b)$
$\delta_s \leftarrow fl(((a \times \delta_b) + (b \times \delta_a)) + \delta_\times)$
return $\langle s,\ \delta_s \rangle$

Algorithm 11: AC_TwoProduct($\langle a,\ \delta_a \rangle,\ \langle b,\ \delta_b \rangle$), automatically compensated product of two *compensated numbers*.



(a) AC_TwoSum.



(b) AC_TwoProduct.

Fig. 3: Diagrams for Algorithms 10 and 11 for the automatic compensation of the sum (a), and the product (b). The dashed or dotted lines are removed when a or b are not compensated numbers but standard floating-point numbers.

Two different sources of errors have to be considered. 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 10 and 11. 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 10 and 11. These inherited errors come from previous floating-point calculations. Operands with no inherited error are processed to minimize added compensations. Figure 3 shows such variants which can be obtained by removing the dashed or dotted lines.

Listing 2 illustrates what provides the code transformation of the sequence $a = b + c \times d$ of Listing 1.

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 ;
}
```

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 ;
}
```

IV. Experimental Results

We now describe our CoHD tool that implements this code transformation. We apply it to several case studies chosen such that there exist compensated versions to compare with. We also add comparisons with the corresponding double-double versions.

A. The CoHD Tool

CoHD is a source-to-source transformer written in OCaml and built as a compiler. The *front-end*, which reads input C files, comes from a previous development by Casse [3]. The *middle-end* implements some passes of optimization, from classical compiler passes such as operand renaming or three-address code conversion [1, Chap. 19]. It also implements one pass of floating-point error compensation. This pass uses our methodology and the algorithms defined in Section III. Then, the *back-end* translates the intermediate representation into C code.

B. Case Studies

We study here the cases described in Table II which are representative of existing compensated algorithms.

Case studies: compensated algorithms of reference

1) Sum2 for the recursive summation of n values [22].
2) CompHorner [7] and CompHornerDer [10] for Horner's evaluation of $p_H(x) = (x - 0.75)^5(x-1)^{11}$ and its derivative.
3) CompdeCasteljau and CompdeCasteljauDer [11] for evaluating $p_D(x) = (x-0.75)^7(x-1)$ and its derivative, written in the Bernstein basis, by means of deCasteljau's scheme.
4) CompClenshawI and CompClenshawII [9] for evaluating $p_C(x) = (x - 0.75)^7(x-1)^{10}$ written in the Chebyshev basis, by means of Clenshaw's scheme.

Summation (case 1 above)

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	random
d_8	10^6	random

Polynomial evaluations (cases 2, 3, 4)

Data	# x	range
x_1	256	$\{0.85 : 0.95\}$ (uniform dist.)
x_2	256	$\{1.05 : 1.15\}$ (uniform dist.)
x	1	random

TABLE II: Case studies and data for Sum and polynomial evaluation with Horner, Clenshaw, and DeCasteljau.

This section presents how we perform accuracy and execution time measurements to compare programs generated automatically by our method with programs, written by hand, that implement compensated and double-double algorithms. It also presents a study of the HORNER's algorithm, and summarizes other test study results: summation, polynomial and derivative evaluation with CLENSHAW or DECASTELJAU algorithms. All measurements are done with the following experimental environment: 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).

Accuracy and execution time measurements. Accuracy is measured as the number of significant bits in the floating-point mantissa. So, 53 is the maximum value we can expect from the *binary64* format.

A reliable measure of the execution time is more difficult to obtain. Such measurements are not always reproducible because of many side effects (operating system, executing programs,...). Significant measures are provided here using two software tools. First, PAPI (Performance Application Programming Interface) [20] allows us to read the physical counters of cycles or instructions that correspond to an actual execution. The second software, PERPI [6], measures the numbers of cycles and instructions of one *ideal execution*, that is, one execution by a machine with infinite resources. The latter measure is more related to a performance potential than to the actual one as provided by PAPI. Using both tools provides confident and complementary results.

Horner's polynomial evaluation. We automatically compensate HORNER's scheme and compare it with DDHORNER (a double-double HORNER evaluation) and COMPHORNER (a compensated HORNER algorithm). The compensated algorithm and the data come from [7].

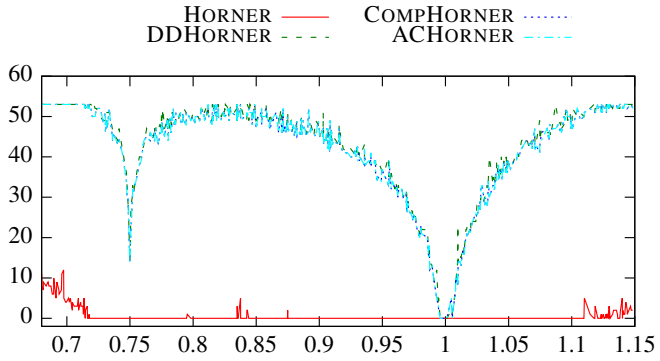


Fig. 4: Number of significant bits $\#_{sig}$ when evaluating $p_H(x) = (x - 0.75)^5(x - 1)^{11}$, where $x \in [0.68, 1.15]$ for HORNER, DDHORNER, COMPHORNER, and ACHORNER.

Let $p_H(x) = (x - 0.75)^5(x - 1)^{11}$ be evaluated with HORNER's scheme, for 512 $x \in \mathbb{F} \cap [0.68, 1.15]$. Figure 4 shows the accuracy of this evaluation using HORNER (original), DDHORNER, COMPHORNER, and our automatically generated ACHORNER algorithm. In each case, we measure the number of significant bits $\#_{sig}$. The original HORNER's accuracy is low since the evaluation is processed 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 has the same accuracy behavior as the twice more accurate DDHORNER and COMPHORNER.

	PAPI (PERPI)		
	instructions	cycles	IPC
COMPHORNER	532 (566)	277 (62)	1.99 (9.12)
DDHORNER	658 (676)	920 (325)	0.72 (2.08)
ACHORNER	553 (581)	303 (77)	1.82 (7.54)
AC/COMP	1.04 (1.02)	1.09 (1.24)	0.95 (0.82)
AC/DD	0.84 (0.85)	0.33 (0.23)	2.55 (3.62)

TABLE III: Performance measurements of the algorithms: COMPHORNER, DDHORNER, and ACHORNER. Real values (PAPI) are the mean of 10^6 measures. Ideal values (PERPI) are displayed within parentheses.

Table III shows the real 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 than the compensated one. Moreover, Table III confirms that compensated algorithms expose more ILP than double-double ones. Even if the code of the algorithm generated by our approach is slightly different from the code of the existing algorithm, it appears to be here as accurate and efficient as the one of [7]. Moreover, our future multi-criteria optimizations will produce algorithms quite different, by trading off accuracy against speed [25].

Further results. We now synthesize the case studies of algorithms and data presented in Table II. They are chosen such that the algorithm returns no significant digit at the working precision while all of them are recovered by the twice more accurate ones.

Table IV presents the differences of the number of significant bits, between the automatically compensated algorithms AC and the compensated (COMP) ones, or the double-double (DD) ones. For example, the first line concerns HORNER's scheme of p_H for data x_1. The difference between the number 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 similarly good behavior. The slight differences of 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 dependant in the double-double case. This appears only when the condition number is large enough.

Figure 5 presents the performance ratios between automatically compensated algorithms (AC) and existing compensated (COMP) or double-double (DD) ones.

Here again, PAPI real measures (the mean of 10^6 executions) and PERPI ideal measures are proposed (plain or dotted

Algorithm	Data	AC-COMP	AC-DD
HORNER	p_H, x_1	0	-1
HORNER	p_H, x_2	0	-0.5
HORNERDER	p_H, x_1	$+0.1$	-0.3
HORNERDER	p_H, x_2	$+0.3$	$+0.1$
CLENSHAWI	p_C, x_1	0	-1.3
CLENSHAWI	p_C, x_2	-0.3	-1.5
CLENSHAWII	p_C, x_1	-0.3	-1.7
CLENSHAWII	p_C, x_2	0	-1.2
DECASTELJAU	p_D, x_1	0	0
DECASTELJAU	p_D, x_2	0	0
DECASTELJAUDER	p_D, x_1	0	0
DECASTELJAUDER	p_D, x_2	0	0
SUM	d_1	0	0
SUM	d_2	0	0
SUM	d_3	0	0
SUM	d_4	0	-3
SUM	d_5	0	-4.8
SUM	d_6	0	-10

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

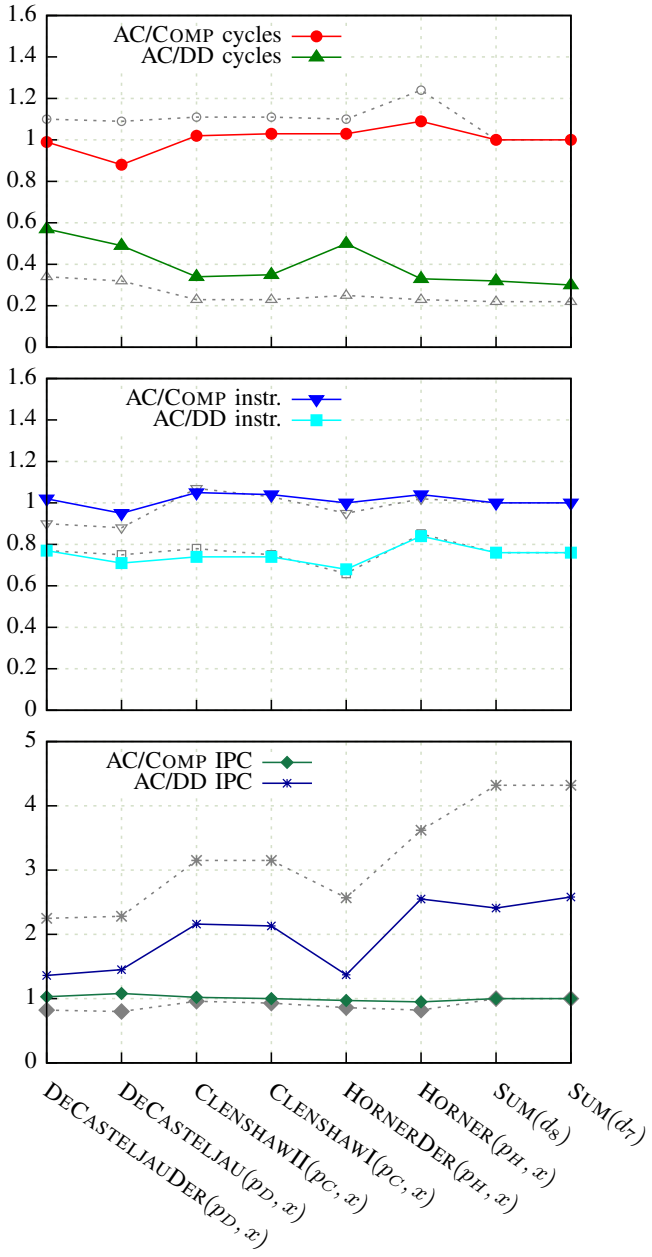


Fig. 5: Performance ratios between automatically compensated algorithms (AC) and existing compensated (COMP) or double-double (DD) ones. Line drawings are real measurements done with PAPI (mean of 10^6 values) while dotted ones are ideal measures done with PERPI.

lines, respectively). The top and middle parts of Figure 5 show respectively the ratios of the number of cycles and of the number of instructions. For example, the rightmost plot is SUM(d_7). We see that the instruction ratio AC/COMP $= 1$. The original SUM2 algorithm and our automatically generated one share the same number of instructions (both for the real and ideal measurements). The instruction ratio compared to the DD one is 0.8. This means 20% less generated instructions. Similarly, compensated algorithms (original and generated) present only 30% of the total cycles number of the DD one. The bottom part of Figure 5 shows the ratio of the number of instructions per cycle. We observe that AC algorithms have the same features as the original compensated ones. Measurements confirm also the interest of compensated algorithms, which have a better ILP potential than DD ones.

Finally, we note that the ILP potential (shown as dotted lines in Figure 5) is not fully exploited in our experimental environment. In a more favorable environment, where the hardware could exploit much more ILP, even better results for compensated algorithms are expected.

V. CONCLUSIONS AND PERSPECTIVES

In this article we discussed the automated transformation of programs using floating-point arithmetic. We propose a new method for automatically compensating the floating-point errors of the computations, which improves the accuracy without impacting execution time too much. The automatic transformation produces some compensated algorithms which are as accurate and efficient as the ones derived case by case. The efficiency of our approach has been illustrated on various case studies.

It remains now to validate this approach (and the CoHD tool) on real and more sophisticated programs. To achieve this, we have to add the support of floating-point division, square-root, and the elementary functions. Moreover, this work is actually a first step toward the automatic generation of multi-criteria program optimizations (with respect to accuracy and execution time). It will allow us to apply partial error compensation and optimize for the execution time overhead. Strategies of partial transformations assured by code synthesis will be the subject of another paper (which we currently write), whose abstract is given in [25].

REFERENCES

[1] A. W. Appel. *Modern Compiler Implementation: In ML*. Cambridge University Press, New York, NY, USA, 1998.

[2] M. Blair, S. Obenski, and P. Bridickas. 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., February 1992.

[3] H. Casse. frontc 3.4: an OCAML C parser and pretty-printer. 2000. TRACES Research Group, Institut de Recherche en Informatique de Toulouse (IRIT), France.

[4] T. J. Dekker. A floating-point technique for extending the available precision. *Numer. Math.*, 18:224–242, 1971.

[5] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Software*, 33(2), 2007.

[6] B. Goossens, P. Langlois, D. Parello, and E. Petit. PerPI: A tool to measure instruction level parallelism. In Kristján Jónasson, editor, *Applied Parallel and Scientific Computing*, volume 7133 of *Lecture Notes in Computer Science*, pages 270–281. Springer Berlin Heidelberg, 2012.

[7] S. Graillat, P. Langlois, and N. Louvet. Algorithms for accurate, validated and fast polynomial evaluation. *Japan Journal of Industrial and Applied Mathematics*, 26(2-3):191–214, 2009.

[8] Y. Hida, X. S. Li, and D. H. Bailey. Algorithms for quad-double precision floating point arithmetic. In *Proc. 15th IEEE Symposium on Computer Arithmetic*, pages 155–162. IEEE Computer Society Press, Los Alamitos, CA, USA, 2001.

[9] H. Jiang, R. Barrio, H. Li, X. Liao, L. Cheng, and F. Su. Accurate evaluation of a polynomial in Chebyshev form. *Appl. Math. Comput.*, 217(23):9702–9716, 2011.

[10] H. Jiang, S. Graillat, C. Hu, S. Li, X. Liao, L. Chang, and F. Su. Accurate evaluation of the k-th derivative of a polynomial and its application. *Computers Math. Applic.*, 243:28–47, 2013.

[11] H. Jiang, S. Li, L. Cheng, and F. Su. Accurate evaluation of a polynomial and its derivative in Bernstein form. *Computers Math. Applic.*, 60(3):744–755, 2010.

[12] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1997.

[13] P. Kornerup, V. Lefèvre, N. Louvet, and J.-M. Muller. On the computation of correctly rounded sums. *IEEE Trans. Comput.*, 61(3):289–298, 2012.

[14] P. Langlois. Automatic linear correction of rounding errors. *BIT*, 41(3):515–539, 2001.

[15] P. Langlois and N. Louvet. More instruction level parallelism explains the actual efficiency of compensated algorithms. http://hal.archives-ouvertes.fr/hal-00165020, 2007. Laboratoire de Physique Appliquée et d'Automatique, Perpignan, France.

[16] P. Langlois, D. Parello, and B. Goossens. Towards a reliable performance evaluation of accurate summation algorithms. *In SIAM Conference on Computational Science & Engineering (CSE13)*, 2013.

[17] C. Q. Lauter. Basic building blocks for a triple-double intermediate format. Technical Report RR-5702, INRIA, 2005.

[18] J.-L. Lions, R. Hergott, B. Humbert, and E. Lefort. Ariane 5 flight 501 failure, report by the inquiry board. Technical report, European Space Agency, 1996.

[19] O. Møller. Quasi double-precision in floating-point addition. *BIT*, 5:37–50, 1965.

[20] P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[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] S. M. Rump, T. Ogita, and S. Oishi. Accurate sum and dot product. *SIAM J. Sci. Comput*, 2005.

[23] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Springer Disc. & Comp. Geometry*, 18(3):305–363, 1997.

[24] IEEE Computer Society. *IEEE 754 Standard for Floating-Point Arithmetic*. The Institute of Electrical and Electronics Engineers, Inc., 2008.

[25] L. Thévenoux, M. Martel, and P. Langlois. Code synthesis to optimize accuracy and execution time of floating-point programs. https://hal.archives-ouvertes.fr/hal-01157509v1, 2015.