Automatic Source-to-Source Error Compensation of Floating-Point Programs
Laurent Thévenoux, Philippe Langlois, Matthieu Martel

To cite this version:
Laurent Thévenoux, Philippe Langlois, Matthieu Martel. Automatic Source-to-Source Error Compensation of Floating-Point Programs. Computational Science and Engineering (CSE), Oct 2015, Porto, Portugal. pp.9–16, 10.1109/CSE.2015.11. hal-01158399

HAL Id: hal-01158399
https://hal.archives-ouvertes.fr/hal-01158399
Submitted on 27 Oct 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License
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. 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 do 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 $±$ and $×$ 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 $±$ and $×$. 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:

$$f = (−1)^{s} \cdot m \cdot β^e,$$

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

In this article, 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 do 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 $±$ and $×$ 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.

A. IEEE Floating-Point Arithmetic

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

$$f = (−1)^{s} \cdot m \cdot β^e,$$

where $s \in \{0, 1\}$ is the sign, $m = \sum_{i=0}^{p-1} d_i β^{-i} = (d_0, d_1 d_2 \cdots d_{p-1})_β$ is the mantissa (with $d_i \in \{0, 1, \ldots, β−1\}$ for each digit $d_i$).
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 $\hat{x} = fl(x)$ for a real number $x$. We have $|x - \hat{x}| \leq ulp(\hat{x})/2$.

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

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

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

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

B. Error-Free Transformations

Error-free transformations (EFT) provide lossless transformations of basic floating-point operations $\circ \in \{+, -, \times, \div\}$. Let $a, b \in \mathbb{F}$ and $\hat{x} = fl(a \circ b)$. There exists a floating-point value $y = a \circ b - \hat{x}$ such that $a \circ b = \hat{x} + y$. We have $|y| \leq ulp(\hat{x})/2$. Hence $\hat{x}$ (resp. $y$) is the upper (resp. lower) part of $a \circ b$ and no digit of $\hat{x}$ 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.

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

$$x \leftarrow fl(a + b) \quad \triangleright \quad |a| \geq |b|$$
$$y \leftarrow fl((a - x) + b)$$
$$\text{return } [x, y]$$

Algorithm 2: TwoSUM$(a, b)$ [Moller, 1965 and Knuth, 1969].

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

Algorithms 1 and 2, respectively introduced by Dekker [4] and Knuth [12, Chap. 4] and Moller [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 $b$.

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

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

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

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

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.

<table>
<thead>
<tr>
<th>EFT algorithm</th>
<th>flop depth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
</tr>
<tr>
<td>FASTTwoSum</td>
<td>1</td>
</tr>
<tr>
<td>TwoSum</td>
<td>1</td>
</tr>
<tr>
<td>TwoProduct</td>
<td>1</td>
</tr>
<tr>
<td>TwoProductFMA</td>
<td>1</td>
</tr>
</tbody>
</table>

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: $|a_L| \leq ulp(a_H)/2$. 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 $|x| \leq ulp(x_H)/2$. This step is insured by a FastTwoSum EFT and is represented by dotted boxes in Figure 2.

<table>
<thead>
<tr>
<th>$r_H$, $r_L$</th>
<th>$r_H$, $r_L$ = TwoSum($a_H$, $b_H$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_H$, $s_L$</td>
<td>$s_H$, $s_L$ = TwoSum($a_L$, $b_L$)</td>
</tr>
<tr>
<td>c</td>
<td>$c \leftarrow fl(r_L + s_H)$</td>
</tr>
<tr>
<td>[u_H, u_L]</td>
<td>[u_H, u_L] = FastTwoSum($r_H$, c)</td>
</tr>
<tr>
<td>w</td>
<td>$w \leftarrow fl(s_L + u_L)$</td>
</tr>
<tr>
<td>return</td>
<td>$[x_H, x_L] = \text{FastTwoSum}(u_H, w)$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$r_H$, $r_L$</th>
<th>$r_H$, $r_L$ = TwoProduct($a_H$, $b_H$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_L$</td>
<td>$r_L \leftarrow fl(r_L + (a_H \times b_H))$</td>
</tr>
<tr>
<td>$r_L$</td>
<td>$r_L \leftarrow fl(r_L + (a_L \times b_H))$</td>
</tr>
<tr>
<td>return</td>
<td>$[x_H, x_L] = \text{FastTwoSum}(r_H, r_L)$</td>
</tr>
</tbody>
</table>

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

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.
The example of $H_O R N E R$ in [16].

A detailed analysis has been presented by the next iteration, so IPC($H_{\mathrm{DD}}$) algorithm, each iteration of the loop contains $10$ operations versus $7$ for the $S_{\mathrm{UM}}$ algorithm. Nevertheless, the main difference between both algorithms is in the parallelization of the loop iterations. The $S_{\mathrm{UM}}DD$ 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($S_{\mathrm{UM}}DD$) $= (10n-9)/(7n-5) \approx 1.42$. The $S_{\mathrm{UM}}2$ algorithm does not suffer from such drawbacks and iterations can be executed with a latency of only one flop: IPC($S_{\mathrm{UM}}2$) $= (7n-6)/(n+5) \approx 7$. So $S_{\mathrm{UM}}2$ 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 $H_O R N E R$’s polynomial transformation is detailed in [15]. This latter shows that the compensated $H_O R N E R$’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 $\mathcal{S}$ of dataflow dependent operations required to obtain one or several results.

2) Then for each sequence $\mathcal{S}_i$ 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 $x \in \mathcal{S}_i$ 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($\mathcal{S}_i$) means computing $x \leftarrow fl(x + \delta_x)$ for $x$ being a result of $\mathcal{S}_i$.

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.

Algorithm 8: $S_{\mathrm{UM}}DD(a_1, a_2, \ldots, a_n)$, double-double classical recursive summation.

\[
\begin{align*}
  s_H & \leftarrow a_1 \\
  s_L & \leftarrow 0 \\
  & \text{for } i = 2 : n \\
  & [s_H, s_L] = \text{DD}_TWO\text{SUM}(s_H, s_L, a_i, 0) \\
  & \text{return } s_H
\end{align*}
\]

Algorithm 9: $S_{\mathrm{UM}}2(a_1, a_2, \ldots, a_n)$, compensated classical recursive summation [Rump, Ogita, and Oishi, 2005].

\[
\begin{align*}
  s & \leftarrow a_1 \\
  e & \leftarrow 0 \\
  & \text{for } i = 2 : n \\
  & [s, e] = \text{TwoSUM}(s, a_i) \\
  & e \leftarrow fl(e + \epsilon) \\
  & \text{return } fl(s + e)
\end{align*}
\]

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 $S_{\mathrm{UM}}$ algorithm, each iteration performs one floating-point operation. Each iteration can be followed immediately by the next iteration, so IPC($S_{\mathrm{UM}}$) $= (n-1)/n \approx 1$. With the $S_{\mathrm{UM}}DD$ algorithm, each iteration of the loop contains $10$ operations versus $7$ for the $S_{\mathrm{UM}}2$ algorithm. Nevertheless, the main difference between both algorithms is in the parallelization of the loop iterations. The $S_{\mathrm{UM}}DD$ 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($S_{\mathrm{UM}}DD$) $= (10n-9)/(7n-5) \approx 1.42$.

The $S_{\mathrm{UM}}2$ algorithm does not suffer from such drawbacks and iterations can be executed with a latency of only one flop: IPC($S_{\mathrm{UM}}2$) $= (7n-6)/(n+5) \approx 7$. So $S_{\mathrm{UM}}2$ 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 $H_O R N E R$’s polynomial evaluation algorithm is detailed in [15]. This latter shows that the compensated $H_O R N E R$’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 $\mathcal{S}$ of dataflow dependent operations required to obtain one or several results.

2) Then for each sequence $\mathcal{S}_i$ compute the error terms and accumulate them beside the original computation

\[
\begin{align*}
  \delta_H & \leftarrow a_1 \\
  \delta_L & \leftarrow 0 \\
  & \text{for } i = 2 : n \\
  & [\delta_H, \delta_L] = \text{DD}_TWO\text{SUM}(\delta_H, \delta_L, a_i, 0) \\
  & \text{return } \delta_H
\end{align*}
\]

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 $x \in \mathcal{S}_i$ 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($\mathcal{S}_i$) means computing $x \leftarrow fl(x + \delta_x)$ for $x$ being a result of $\mathcal{S}_i$.

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.

\[
\begin{align*}
  [s, \delta_\pm] & = \text{TwoSUM}(a, b) \\
  \delta_s & \leftarrow fl((\delta_a + \delta_b) + \delta_\pm) \\
  & \text{return } \langle s, \delta_s \rangle
\end{align*}
\]

Algorithm 10: $AC_{\text{TwoSUM}}(a, \delta_a, (b, \delta_b))$, automatically compensated sum of two compensated numbers.

\[
\begin{align*}
  [s, \delta_x] & = \text{TwoPRODUCT}(a, b) \\
  \delta_s & \leftarrow fl(((\delta_a \times \delta_b) + (b \times \delta_a)) + \delta_x) \\
  & \text{return } \langle s, \delta_s \rangle
\end{align*}
\]

Algorithm 11: $AC_{\text{TwoPRODUCT}}(\langle a, \delta_a \rangle, (b, \delta_b))$, automatically compensated product of two compensated numbers.

This fact is measurable in practice, and compensated algorithms exploit this low level parallelism much better than double-double ones. The example of $H_O R N E R$’s polynomial transformation is detailed in [15]. This latter shows that the compensated $H_O R N E R$’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.
\( \delta_b \), 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 \( a = b + c \times d \).
```c
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.
```c
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^\lceil 53/2 \rceil + 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.

<table>
<thead>
<tr>
<th>Case studies: compensated algorithms of reference</th>
<th>Condition number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) \text{SUM2} for the recursive summation of ( n ) values \cite{22}.</td>
<td>( 10^8 )</td>
</tr>
<tr>
<td>2) \text{COMP_HORNER} \cite{7} and \text{COMP_HORNER_DER} \cite{10} for Horner’s evaluation of ( p_H(x) = (x - 0.75)^5(x - 1)^{11} ) and its derivative.</td>
<td>( 10^8 )</td>
</tr>
<tr>
<td>3) \text{COMP_DECASTELJAU} and \text{COMP_DECASTELJAU_DER} \cite{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.</td>
<td>( 10^{16} )</td>
</tr>
<tr>
<td>4) \text{COMP_CLENSHWI} and \text{COMP_CLENSHWII} \cite{9} for evaluating ( p_C(x) = (x - 0.75)^7(x - 1)^{10} ) written in the Chebyshev basis, by means of Clenshaw’s scheme.</td>
<td>( 10^{16} )</td>
</tr>
</tbody>
</table>

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textbf{Summation (case 1 above)} & \textbf{Data} & \textbf{# values} \textbf{condition number} \\
\hline
\text{SUM2} & \text{d_1} & \text{32 \times 10^4} & 10^8 \\
& \text{d_2} & \text{32 \times 10^5} & 10^8 \\
& \text{d_3} & \text{32 \times 10^6} & 10^8 \\
& \text{d_4} & \text{32 \times 10^4} & 10^{16} \\
& \text{d_5} & \text{32 \times 10^5} & 10^{16} \\
& \text{d_6} & \text{32 \times 10^6} & 10^{16} \\
& \text{d_7} & \text{10^6} & \text{random} \\
& \text{d_8} & \text{10^6} & \text{random} \\
\hline
\end{tabular}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\textbf{Polynomial evaluations (cases 2, 3, 4)} & \textbf{Data} & \textbf{# x} \textbf{range} \\
\hline
\text{COMP_HORNER} & \text{\{0.85 : 0.95\}} & \text{\text{uniform dist.}} \\
\text{\{1.05 : 1.15\}} & \text{\text{uniform dist.}} \\
\text{\{random\}} & \text{\text{random}} \\
\text{\{random\}} & \text{\text{random}} \\
\hline
\end{tabular}
\end{table}

TABLE II: Case studies and data for \text{SUM} and polynomial evaluation with \text{HORNER, CLENSHW}, and \text{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 results: summation, polynomial and derivative evaluation with CleShaw or DecaStel-Jau algorithms. All measurements are done with the following experimental environment: Intel® CoreTM 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].

![Graph](image)

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 $x \in \mathbb{R} \cap [0.68, 1.15]$. 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.

<table>
<thead>
<tr>
<th>Papi (PERPI)</th>
<th>instructions</th>
<th>cycles</th>
<th>IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPHorner</td>
<td>532 (566)</td>
<td>277 (62)</td>
<td>1.99 (9.12)</td>
</tr>
<tr>
<td>DDHorner</td>
<td>658 (676)</td>
<td>920 (325)</td>
<td>0.72 (2.08)</td>
</tr>
<tr>
<td>ACHorner</td>
<td>553 (581)</td>
<td>303 (77)</td>
<td>1.82 (7.54)</td>
</tr>
<tr>
<td>AC/COMP</td>
<td>1.04 (1.02)</td>
<td>1.09 (1.24)</td>
<td>0.95 (0.82)</td>
</tr>
<tr>
<td>AC/DD</td>
<td>0.84 (0.85)</td>
<td>0.33 (0.23)</td>
<td>2.55 (3.62)</td>
</tr>
</tbody>
</table>

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...
The efficiency of our approach has been illustrated on various case studies. The existing compensated algorithms (C\textsuperscript{d}) have a better ILP potential than DD ones. Measurements confirm also the interest of compensated algorithms, which are as accurate and efficient as the ones derived case by case. The transformation produces some 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


