Automatic source-to-source error compensation of floating-point programs

Laurent Thévenoux\textsuperscript{1}  Philippe Langlois\textsuperscript{2}  Matthieu Martel\textsuperscript{2}

\textsuperscript{1}LIP, ENS de Lyon, INRIA, France
\textsuperscript{2}University of Perpignan Via Domitia, France
Context and motivation

Context

- Numerical computations can be **innaccurate**: rounding errors
- **Techniques** are available for programmers to improve their numerical programs: expansions, software libraries, . . .
- These techniques are costly: improving **accuracy** impacts **execution-time**
- **Error compensation** technique allows a good tradeoff between **accuracy** and **execution-time** but reserved to experts

Motivation

- Accuracy and **execution-time** are two major concerns of software developers
- Critical in many systems (from automotive to aerospace industry)

**Automate** compensation to allow non-expert users to use it:

source-to-source error compensation
Context and motivation

Context

- Numerical computations can be **inaccurate**: rounding errors
- **Techniques** are available for programmers to improve their numerical programs: expansions, software libraries, . . .
- These techniques are costly: improving **accuracy** impacts **execution-time**
- **Error compensation** technique allows a good tradeoff between **accuracy** and **execution-time** but reserved to experts

Motivation

- **Accuracy** and **execution-time** are two major concerns of software developers
- Critical in many systems (from automotive to aerospace industry)

Automate compensation to allow non-expert users to use it:

**source-to-source error compensation**
Context and motivation

Context

- Numerical computations can be **innaccurate**: rounding **errors**
- **Techniques** are available for programmers to **improve** their numerical programs: expansions, software libraries, . . .
- These techniques are costly: improving **accuracy** impacts **execution-time**
- **Error compensation** technique allows a good tradeoff between **accuracy** and **execution-time** but reserved to experts

Motivation

- **Accuracy** and **execution-time** are two major concerns of software developers
- Critical in many systems (from automotive to aerospace industry)

**Automate** compensation to allow non-expert users to use it:

**source-to-source error compensation**
Outline

Background on floating-point arithmetic
   Error-free transformations
   Double-double expansions and compensated algorithms

Automatic program transformation
   Improving accuracy: methodology
   Experimental results

Conclusion and perspectives
IEEE 754 floating-point arithmetic [IEEE754]
A standard to represent real numbers since 1985

- $\mathbb{F}$, the finite floating-point (FP) numbers following one of the formats of IEEE 754
- This set is defined by a precision $p$, and an exponent range $[e_{\text{min}}, e_{\text{max}}]$ such that

$$p = 53, \quad e_{\text{max}} = 1 - e_{\text{min}} = 1023$$

in binary64 format.
- Has several rounding modes: to nearest (RN), to zero (RZ), to infinities (RU, RD)

\[
x = 0.1 \approx 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ ...
\]

$\hat{x} = \text{binary32 representation of } x \text{ (before rounding)}$

$\leftrightarrow$ A way of estimating the accuracy of $\hat{x} = R^*(x)$ is through the number of significant bits $0 \leq \#_{\text{sig}} \leq p$ shared by $x$ and $\hat{x}$:

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

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

$$E_{\text{rel}}(\hat{x}) = |x - \hat{x}|/|x|, \quad x \neq 0.$$
IEEE 754 floating-point arithmetic [IEEE754]
A standard to represent real numbers since 1985

- \( \mathbb{F} \), the **finite** floating-point (FP) numbers following one of the formats of IEEE 754
- This set is defined by a **precision** \( p \), and an **exponent range** \([e_{min}, e_{max}]\) such that
  \[
p = 53, \quad e_{max} = 1 - e_{min} = 1023 \quad \text{in binary64 format}.
\]
- Has several **rounding modes**: to nearest (RN), to zero (RZ), to infinities (RU, RD)

\[
x = 0.1 \approx 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ ...
\]

\( \hat{x} = \text{binary32 representation of } x \) (before rounding)

\[\rightarrow\] A way of estimating the **accuracy** of \( \hat{x} = R^*(x) \) is through the number of significant bits \( 0 \leq \#_{sig} \leq p \) shared by \( x \) and \( \hat{x} \):

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

where \( E_{ref}(\hat{x}) \) is the relative error defined by: \( E_{rel}(\hat{x}) = |x - \hat{x}|/|x|, \quad x \neq 0 \).
IEEE 754 floating-point arithmetic [IEEE754]
A standard to represent real numbers since 1985

- \( \mathbb{F} \), the **finite** floating-point (FP) numbers following one of the formats of IEEE 754
- This set is defined by a **precision** \( p \), and an **exponent range** \([e_{\text{min}}, e_{\text{max}}]\) such that
  \[
  p = 53, \quad e_{\text{max}} = 1 - e_{\text{min}} = 1023 \quad \text{in binary64 format}.
  \]
- Has several **rounding modes**: to nearest (RN), to zero (RZ), to infinities (RU, RD)

\[
\begin{align*}
x & = 0.1 \approx \overline{0} 0 1 1 1 1 1 0 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 1 0 0 1 1 0 \ldots \\
\hat{x} & = \text{binary32 representation of } x \text{ (before rounding)} \\
\end{align*}
\]

\( \text{lost bits after rounding-to-nearest RN}(x) \)

\( \leftrightarrow \) A way of estimating the **accuracy** of \( \hat{x} = R^*(x) \) is through the number of significant bits \( 0 \leq \#_{\text{sig}} \leq p \) shared by \( x \) and \( \hat{x} \):

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

where \( E_{\text{ref}}(\hat{x}) \) is the relative error defined by: \( E_{\text{rel}}(\hat{x}) = |x - \hat{x}|/|x|, \quad x \neq 0 \).
IEEE 754 floating-point arithmetic [IEEE754]
A standard to represent real numbers since 1985

- \( \mathbb{F} \), the **finite** floating-point (FP) numbers following one of the formats of IEEE 754
- This set is defined by a **precision** \( p \), and an **exponent range** \([e_{\text{min}}, e_{\text{max}}]\) such that

\[
p = 53, \quad e_{\text{max}} = 1 - e_{\text{min}} = 1023 \quad \text{in binary64 format.}
\]

- Has several **rounding modes**: to nearest (RN), to zero (RZ), to infinities (RU, RD)

\[
x = 0.1 \approx \begin{array}{ccccccccccccccccccccccc} 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & \cdots
\end{array}
\]

\[
\hat{x} = \text{binary32 representation of } x \text{ (before rounding)}
\]

\[
\text{lost bits after rounding-to-nearest RN}(x), \ #_{\text{sig}} = 23
\]

\[\leftrightarrow \text{A way of estimating the **accuracy** of } \hat{x} = R^*(x) \text{ is through the number of significant bits } 0 \leq #_{\text{sig}} \leq p \text{ shared by } x \text{ and } \hat{x}:\]

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

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

\[E_{\text{rel}}(\hat{x}) = |x - \hat{x}|/|x|, \quad x \neq 0.\]
Error-free transformations (EFTs)
Allow to compute the error generated by a floating-point addition or multiplication

Principle [MBdD10]
Let \( \circ \in \{+,-,\times\} \), if \( x = \text{RN}(a \circ b) \), then the floating-point error \( y = \text{RN}(a \circ b) - x \) is exactly representable in \( \mathbb{F} \): EFTs allow to compute \( y \) with floating-point arithmetic!

For the sum...

\[
\text{function FastTwoSum}(a, b) \triangleright [\text{Dekker}, 71] \\
x \leftarrow \text{RN}(a + b) \quad \triangleright |a| \geq |b| \\
y \leftarrow \text{RN}((a - x) + b) \\
\text{return } (x, y) \\
\text{end function}
\]

\[
\text{function TwoSum}(a, b) \triangleright [\text{Knuth}, 69] \\
x \leftarrow \text{RN}(a + b) \\
z \leftarrow \text{RN}(x - a) \\
y \leftarrow \text{RN}((a - (x - z)) + (b - z)) \\
\text{return } (x, y) \\
\text{end function}
\]

…and the product

\[
\text{function TwoProduct}(a, b) \triangleright [\text{Dekker}, 71] \\
x \leftarrow \text{RN}(a \times b) \\
[a_H, a_L] = \text{Split}(a) \\
[b_H, b_L] = \text{Split}(b) \\
y \leftarrow \text{RN}(a_L \times b_L - (((x - a_H \times b_H) - a_L \times b_H) - a_H \times b_L)) \\
\text{return } (x, y) \\
\text{end function}
\]

\[
\text{function Split}(a) \triangleright [\text{Veltkamp}, 68] \\
c \leftarrow \text{RN}(f \times a) \quad \triangleright f = 2[^{p/2}] + 1 \\
a_H \leftarrow \text{RN}(c - (c - a)), \quad a_L \leftarrow \text{RN}(a - a_H) \\
\text{return } (a_H, a_L) \\
\text{end function}
\]

- EFTs are costly: 3, 6, and 17 FP operations for FastTwoSum, TwoSum, and TwoProduct
- \textit{fused-multiply-add} instruction can \textbf{reduce} the cost of TwoProduct to 2
Double-double expansions and compensated algorithms
Improving accuracy using EFTs

Two methods based on EFTs

- **Double-double (DD) expansions**: introduced in the 1970’s [Dek71]
- **Compensated algorithms**: popularized in the 2000’s [ROO05, GLL09]

**Double-double**

- Dekker, 1971
- Bailey+, QD Lib, 2000
- Saito, Scilab Toolbox: QuPAT, 2010
- generic method, algorithms applied to each elementary operations
- easy automatic application (overloading)

**Compensated algorithms**

- Rump+, Sum2, Dot2, 2005
- Louvet, CompHorner, 2007
- Graillat+, CompHornerDer, 2013
- specific, expert work: a thesis or research paper per algorithm
- today: sum, dot product, polynomial evaluations

They provide roughly the same accuracy...

- **Double-double** is generic but has a strong impact on performance
- Compensation allows better performance: more instruction level parallelism [LL07] but it is very specific
Double-double expansions and compensated algorithms
Improving accuracy using EFTs

Two methods based on EFTs

▶ **Double-double (DD) expansions**: introduced in the 1970’s [Dek71]
▶ **Compensated algorithms**: popularized in the 2000’s [ROO05, GLL09]

**Double-double**

▶ Dekker, 1971
▶ Bailey+, QD Lib, 2000
▶ Saito, Scilab Toolbox: QuPAT, 2010
▶ generic method, algorithms applied to each elementary operations
▶ easy automatic application (overloading)

**Compensated algorithms**

▶ Rump+, Sum2, Dot2, 2005
▶ Louvet, CompHorner, 2007
▶ Graillat+, CompHornerDer, 2013
▶ specific, expert work: a thesis or research paper per algorithm
▶ today: sum, dot product, polynomial evaluations

They provide roughly the same accuracy…

▶ **Double-double** is generic but has a strong impact on performance
▶ **Compensation** allows better performance: more instruction level parallelism [LL07]
  but it is very specific
Expansions vs. compensated algorithms
Why compensated algorithms expose more Instruction Level Parallelism (ILP) than expansions based ones?

```plaintext
function Sum(a₁, a₂, ..., aₙ)
    s₁ ← a₁
    for i = 2 : n do
        sᵢ ← RN(sᵢ⁻¹ + aᵢ)
    end for
    return sₙ
end function

function SumDD(a₁, a₂, ..., aₙ)
    s_H⁻¹ ← a₁
    s_L⁻¹ ← 0
    for i = 2 : n do
        [s_Hᵢ, s_Lᵢ] = QD_TwoSum(s_Hᵢ⁻¹, s_Lᵢ⁻¹, aᵢ, ∅)
    end for
    return s_Hₙ
end function

function Sum2(a₁, a₂, ..., aₙ)
    s¹ ← a₁
    e¹ ← 0
    for i = 2 : n do
        [sᵢ, ε] = TwoSum(sᵢ⁻¹, aᵢ)
        eᵢ ← RN(eᵢ⁻¹ + ε)
    end for
    return RN(sₙ + eₙ)
end function
```
Expansions vs. compensated algorithms
Why compensated algorithms expose more Instruction Level Parallelism (ILP) than expansions based ones?

function Sum(a_1, a_2, ..., a_n)
    s_1 ← a_1
    for i = 2 : n do
        s_i ← RN(s_{i-1} + a_i)
    end for
    return s_n
end function

function SumDD(a_1, a_2, ..., a_n)
    s_{1H} ← a_1
    s_{1L} ← 0
    for i = 2 : n do
        [s_{iH}, s_{iL}] = QD_TwoSum(s_{i-1H}, s_{i-1L}, a_i, ∅)
    end for
    return s_{nH}
end function

function Sum2(a_1, a_2, ..., a_n)
    s_1 ← a_1
    e_1 ← 0
    for i = 2 : n do
        [s_i, e_i] = TwoSum(s_{i-1}, a_i)
        e_i ← RN(e_{i-1} + e_i)
    end for
    return RN(s_n + e_n)
end function
Expansions vs. compensated algorithms

Why compensated algorithms expose more Instruction Level Parallelism (ILP) than expansions based ones?

function Sum(a₁, a₂, ..., aₙ)
  s₁ ← a₁
  for i = 2 : n do
    sᵢ ← RN(sᵢ₋₁ + aᵢ)
  end for
  return sₙ
end function

function SumDD(a₁, a₂, ..., aₙ)
  s₁ᴴ ← a₁
  s₁ᴸ ← 0
  for i = 2 : n do
    [sᵢᴴ, sᵢᴸ] = QD_TwoSum(sᵢ₋₁ᴴ, sᵢ₋₁ᴸ, aᵢ, ∅)
  end for
  return sₙᴴ
end function

function Sum2(a₁, a₂, ..., aₙ)
  s₁ ← a₁
  e₁ ← 0
  for i = 2 : n do
    [sᵢ, eᵢ] = TwoSum(sᵢ₋₁, aᵢ)
    eᵢ ← RN(eᵢ₋₁ + eᵢ)
  end for
  return RN(sₙ + eₙ)
end function
Methodology of accuracy improvement
Benefit from the good ILP of compensation automatically: detect FP sequences

Detect floating-point sequences

- A sequence is the set $S$ of all dependent operations required to obtain one or several results
- CoHD tool performs this step after transform original code in three-address form
- In this example one sequence of two operations is detected

```c
double Horner(double *P, uint n, double x) {
    double r;
    uint i;
    r = P[n];
    for (i = n - 1; i >= 0; i--) {
        r = r * x + P[i];
    }
    return r;
}
```
Detect floating-point sequences

- A sequence is the set $\mathcal{S}$ of all dependent operations required to obtain one or several results
- CoHD tool performs this step after transform original code in three-address form
- In this example one sequence of two operations is detected
Methodology of accuracy improvement
Benefit from the good ILP of compensation automatically: detect FP sequences

Detect floating-point sequences

- A sequence is the set \( S \) of all dependent operations required to obtain one or several results
- CoHD tool performs this step after transform original code in three-address form
- In this example one sequence of two operations is detected

```c
double Horner(double *P, uint n, double x) {
    double r, tmp;
    uint i;
    r = P[n];
    for (i = n-1; i >= 0; i--) {
        \[
        \begin{align*}
        \text{tmp} &= r * x; \\
        r &= \text{tmp} + P[i];
        \end{align*}
        \]
    }
    return r;
}
```
Methodology of accuracy improvement
Benefit from the good ILP of compensation automatically: replace FP computations with EFTs

Compute error terms and accumulate them

- For each $s \in S$:
  - replace floating-point operations by EFTs,
  - and accumulate errors (inherited, generated),

with the following algorithms:

```c
double Horner(double *P, uint n, double x) {
    double r, tmp;
    uint i;
    r = P[n];
    for (i = n-1; i >= 0; i--) {
        tmp = r * x;
        r = tmp + P[i];
    }
    return r;
}
```

- Every FP number $n \in s$ becomes a compensated number $\langle n, \delta_n \rangle$ where $\delta_n$ is the accumulated error attached to the computed result $n$
Methodology of accuracy improvement
Benefit from the good ILP of compensation automatically: replace FP computations with EFTs

Compute error terms and accumulate them

- For each \( s \in S \):
  - replace floating-point operations by EFTs,
  - and accumulate errors (inherited, generated),

with the following algorithms:

\[
\text{AutoComp}_\text{TwoSum}(a, \delta_a), (b, \delta_b) \\
\quad \begin{align*}
[s, \delta_c] &= \text{TwoSum}(a, b) \\
\delta_s &\leftarrow \text{RN}((\delta_a + \delta_b) + \delta_c) \\
\text{return} \ (s, \delta_s)
\end{align*}
\]

\[
\text{AutoComp}_\text{TwoProduct}(a, \delta_a), (b, \delta_b) \\
\quad \begin{align*}
[s, \delta_x] &= \text{TwoProduct}(a, b) \\
\delta_s &\leftarrow \text{RN}(((a \times \delta_b) + (b \times \delta_a)) + \delta_x) \\
\text{return} \ (s, \delta_s)
\end{align*}
\]

- Every FP number \( n \in s \) becomes a compensated number \( \langle n, \delta_n \rangle \) where \( \delta_n \) is the accumulated error attached to the computed result \( n \)

```c
double Horner(double *P, uint n, double x) {
    double r, tmp, d_tmp, d_r;
    uint i;
    r = P[n];
    d_r = 0.0;
    for(i = n-1; i >= 0; i--)
        tmp = r * x;
        r = tmp + P[i];
}
return r;
```
Methodology of accuracy improvement
Benefit from the good ILP of compensation automatically: replace FP computations with EFTs

Compute error terms and accumulate them

- For each \( s \in \mathcal{S} \):
  - replace floating-point operations by EFTs,
  - and accumulate errors (inherited, generated),

with the following algorithms:

\[
\text{AutoComp\_TwoSum}(a, \delta_a), (b, \delta_b) \\
\quad \left[ s, \delta_+ \right] = \text{TwoSum}(a, b) \\
\quad \delta_s \leftarrow \text{RN}(\delta_a + \delta_b + \delta_+) \\
\quad \text{return } \langle s, \delta_s \rangle \\
\]

\[
\text{AutoComp\_TwoProduct}(a, \delta_a), (b, \delta_b) \\
\quad \left[ s, \delta_x \right] = \text{TwoProduct}(a, b) \\
\quad \delta_s \leftarrow \text{RN}((a \times \delta_b) + (b \times \delta_a) + \delta_x) \\
\quad \text{return } \langle s, \delta_s \rangle \\
\]

- Every FP number \( n \in s \) becomes a compensated number \( \langle n, \delta_n \rangle \) where \( \delta_n \) is the accumulated error attached to the computed result \( n \)

```c
double Horner(double *P, uint n, double x) {
    double r, tmp, d_tmp, d_r;
    uint i;
    r = P[n];
    d_r = 0.0;
    for (i = n - 1; i >= 0; i--) {
        tmp = r * x;
        r = tmp + P[i];
    }
    return r;
}
```

Methodology of accuracy improvement
Benefit from the good ILP of compensation automatically: replace FP computations with EFTs

Compute error terms and accumulate them

- For each $s \in S$:
  - replace floating-point operations by EFTs,
  - and accumulate errors (inherited, generated),

with the following algorithms:

\[
\text{AutoComp\_TwoSum}(a, \delta_a), (b, \delta_b)
\]

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

\[
\text{AutoComp\_TwoProduct}(a, \delta_a), (b, \delta_b)
\]

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

- Every FP number $n \in s$ becomes a compensated number $\langle n, \delta_n \rangle$ where $\delta_n$ is the accumulated error attached to the computed result $n$

```c
#include <math.h>

double Horner(double *P, uint n, double x) {
    double r, tmp, d_tmp, d_r, c, rh, rl,
    xh, xl, d_2p;
    uint i;
    r = P[n];
    d_r = 0.0;
    for (i = n-1; i >= 0; i--) {
        tmp = r * x;
        c = r * 134217729;
        rh = c - (c - r);
        rl = r - rh;
        c = x * 134217729;
        xh = c - (c - x);
        xl = x - xh;
        d_2p = rl * xl - (((x - rh * xh) - rl * xh) - rh * xl);
        d_tmp = d_2p + d_r * x;
        r = tmp + P[i];
    }
    return r;
}
```
Methodology of accuracy improvement
Benefit from the good ILP of compensation automatically: replace FP computations with EFTs

Compute error terms and accumulate them

- For each \( s \in \mathcal{S} \):
  - replace floating-point operations by EFTs,
  - and accumulate errors (inherited, generated),

with the following algorithms:

\[
\text{AutoComp\textunderscore TwoSum}(a, \delta_a, b, \delta_b) \]

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

\[
\text{AutoComp\textunderscore TwoProduct}(a, \delta_a, b, \delta_b) \]

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

- Every FP number \( n \in s \) becomes a compensated number \( \langle n, \delta_n \rangle \) where \( \delta_n \) is the accumulated error attached to the computed result \( n \)

```c
double Horner(double *P, uint n, double x) {
    double r, tmp, d_tmp, d_r, c, rh, rl, xh, xl, d_2p;
    uint i;
    r = P[n];
    d_r = 0.0;
    for(i = n-1; i >= 0; i --) {
        tmp = r * x;
        c = r * 134217729; // 2^24
        rh = c - (c - r);
        rl = r - rh;
        c = x * 134217729; // 2^24
        xh = c - (c - x);
        xl = x - xh;
        d_2p = rl * xl - ((( x - rh * xh) - rl * xh) - rh * xl);
        d_tmp = d_2p + d_r * x;
        r = tmp + P[i];
    }
    return r;
}
```
Methodology of accuracy improvement
Benefit from the good ILP of compensation automatically: replace FP computations with EFTs

Compute error terms and accumulate them

- For each \( s \in \mathcal{S} \):
  - replace floating-point operations by EFTs,
  - and accumulate errors (inherited, generated),

with the following algorithms:

\[
\text{AutoComp\_TwoSum}(a, \delta_a), (b, \delta_b) = \begin{cases} 
[s, \delta_+] = \text{TwoSum}(a, b) \\
\delta_s &\leftarrow \text{RN}((\delta_a + \delta_b) + \delta_+) \\
\end{cases}
\]

\[
\text{AutoComp\_TwoProduct}(a, \delta_a), (b, \delta_b) = \begin{cases} 
[s, \delta_x] = \text{TwoProduct}(a, b) \\
\delta_s &\leftarrow \text{RN}(((a \times \delta_b) + (b \times \delta_a)) + \delta_x) \\
\end{cases}
\]

- Every FP number \( n \in s \) becomes a compensated number \( (n, \delta_n) \) where \( \delta_n \) is the accumulated error attached to the computed result \( n \)

```c
double Horner(double *P, uint n, double x) {
    double r, tmp, d_tmp, d_r, c, rh, rl, xh, xl, d_2p, u, d_2s;
    uint i;
    r = P[n];
    d_r = 0.0;
    for (i = n - 1; i >= 0; i--) {
        tmp = r * x;
        c = r * 134217729;
        rh = c - (c - r);
        rl = r - rh;
        c = x * 134217729;
        xh = c - (c - x);
        xl = x - xh;
        d_2p = rl * xl - (((x - rh * xh) - rl * xh) - rh * xl);
        d_tmp = d_2p + d_r * x;
        r = tmp + P[i];
        u = r - tmp;
        d_2s = (tmp - (r - u)) + (P[i] - u);
        d_r = d_2s + d_tmp;
    }
    return r;
}
```
Compensate errors: close sequences

- For each \( s \in \mathcal{S} \), \( \text{close}(s) \) means computing

\[
n \leftarrow \text{RN}(n + \delta_n)
\]

for \( n \) being a result of \( s \)

```c
double Horner(double *P, uint n, double x) {
    double r, tmp, d_tmp, d_r, c, rh, rl,
             xh, xl, d_2p, u, d_2s;
    uint i;
    r = P[n];
    d_r = 0.0;
    for(i = n-1; i >= 0; i--) {
        tmp = r * x;
        c = r * 134217729;
        rh = c - (c - r);
        rl = r - rh;
        c = x * 134217729;
        xh = c - (c - x);
        xl = x - xh;
        d_2p = rl * xl - ((( x - rh * xh)
                          - rl * xh) - rh * xl);
        d_tmp = d_2p + d_r * x;
        r = tmp + P[i];
        u = r - tmp;
        d_2s = (tmp - (r - u)) + (P[i] - u);
        d_r = d_2s + d_tmp;
    }
    return r;
}
```
Methodology of accuracy improvement
Benefit from the good ILP of compensation automatically: close sequences (error compensation)

Compensate errors: close sequences

- For each $s \in \mathcal{S}$, close$(s)$ means computing

  $$n \leftarrow \text{RN}(n + \delta_n)$$

for $n$ being a result of $s$

```c
double Horner(double *P, uint n, double x) {
    double r, tmp, d_tmp, d_r, c, rh, rl,
        xh, xl, d_2p, u, d_2s;
    uint i;
    r = P[n];
    d_r = 0.0;
    for (i = n-1; i >= 0; i--) {
        tmp = r * x;
        c = r * 134217729;
        rh = c - (c - r);
        rl = r - rh;
        c = x * 134217729;
        xh = c - (c - x);
        xl = x - xh;
        d_2p = rl * xl - (((x - rh * xh) - rl * xh) - rh * xl);
        d_tmp = d_2p + d_r * x;
        r = tmp + P[i];
        u = r - tmp;
        d_2s = (tmp - (r - u)) + (P[i] - u);
        d_r = d_2s + d_tmp;
    }
    return r + d_r;
}
```
Experimental results: case studies from compared works

Our method results compared to existing double-double expansions and compensated ones

1. **Sum2** for the recursive summation of $n$ values [ROO05]

<table>
<thead>
<tr>
<th>Data</th>
<th># values</th>
<th>condition number</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_1$</td>
<td>$32 \times 10^4$</td>
<td>$10^8$</td>
</tr>
<tr>
<td>$d_2$</td>
<td>$32 \times 10^5$</td>
<td>$10^8$</td>
</tr>
<tr>
<td>$d_3$</td>
<td>$32 \times 10^6$</td>
<td>$10^8$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data</th>
<th># values</th>
<th>condition number</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_4$</td>
<td>$32 \times 10^4$</td>
<td>$10^{16}$</td>
</tr>
<tr>
<td>$d_5$</td>
<td>$32 \times 10^5$</td>
<td>$10^{16}$</td>
</tr>
<tr>
<td>$d_6$</td>
<td>$32 \times 10^6$</td>
<td>$10^{16}$</td>
</tr>
</tbody>
</table>

2. **CompHorner** [GLL09] and **CompHornerDer** [JGH13] for Horner’s evaluation of $p_H(x) = (x - 0.75)^5(x - 1)^{11}$ and its derivative

3. **CompdeCasteljau** and **CompdeCasteljauDer** [JLCS10] for evaluating $p_D(x) = (x - 0.75)^7(x - 1)$ and its derivative, written in the Bernstein basis (deCasteljau’s scheme)

4. **CompClenshawI** and **CompClenshawII** [JBL11] for evaluating $p_C(x) = (x - 0.75)^7(x - 1)^{10}$ written in the Chebyshev basis (Clenshaw’s scheme)

<table>
<thead>
<tr>
<th>Data</th>
<th># $x$</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>256</td>
<td>${0.85 : 0.95}$ (uniform dist.)</td>
</tr>
<tr>
<td>$x_2$</td>
<td>256</td>
<td>${1.05 : 1.15}$ (uniform dist.)</td>
</tr>
</tbody>
</table>
Experimental results: performance comparison

Our method results compared to existing double-double expansions and compensated ones

- Compensated algorithms, \texttt{Comp} (manuay implemented), and \texttt{AC} (automatically generated) present similar performance.
- \texttt{Comp} and \texttt{AC} algorithms present more ILP than \texttt{DD} (double-double) ones.

\textarrow{measurements done with \texttt{papi} tool (API of performance counters) on an \texttt{intel Core i5, 2.53GHz} over Linux kernel 3.2 and \texttt{gcc4.6.3 -O2}}

\textarrow{ideal measurements (not shown here) validate experimental measures}
Experimental results: performance comparison

Our method results compared to existing double-double expansions and compensated ones

Compensated algorithms, Comp (manually implemented), and AC (automatically generated) present similar performance.

Comp and AC algorithms present more ILP than DD (double-double) ones.

Measurements done with papi tool (API of performance counters) on an Intel Core i5, 2.53GHz over Linux kernel 3.2 and gcc 4.6.3 -O2. Ideal measurements (not shown here) validate experimental measures.
Experimental results: accuracy comparison

Our method results compared to existing double-double expansions and compensated ones

- Compensated algorithms, **Comp** (manually implemented), and **AC** (automatically generated) present **similar accuracy**

→ DD (double-double) algorithms are more accurate when data are too ill-conditioned (three leftmost cases)
Conclusions and perspectives

Summary

- A new method for automatically compensating the FP error of the computations
- Similar results to those of manual implementations of compensated algorithms
  - better accuracy and ILP exposure

Perspectives

- Support $\div$, $\sqrt{}$ and elementary functions
- Support all C and validate our approach on other case studies
- Unbounded compensation: SumK, HornerK
- Integrate of our code transformation into gcc
  - ask me if you want to see a demonstration of our actual prototype (CoHD)

Related works

- 6 months of knowledge transfer in a startup
- First step toward multi-criteria optimization (accuracy and execution time)
  - https://hal.archives-ouvertes.fr/hal-01157509
Conclusions and perspectives

Summary

- A new method for automatically compensating the FP error of the computations
- Similar results to those of manual implementations of compensated algorithms
  - better accuracy and ILP exposure

Perspectives

- Support \( \div, \sqrt{\ } \) and elementary functions
- Support all C and validate our approach on other case studies
- Unbounded compensation: SumK, HornerK
- Integrate of our code transformation into gcc
  - ask me if you want to see a demonstration of our actual prototype (CoHD)

Related works

- 6 months of knowledge transfer in a startup
- First step toward multi-criteria optimization (accuracy and execution time)
  - https://hal.archives-ouvertes.fr/hal-01157509
References

[Dek71]  T. J. Dekker
   A Floating-Point Technique for Extending the Available Precision, 1971

[GLL09]  S. Graillat, P. Langlois, N. Louvet

   Algorithms for Quad-Double Precision Floating Point Arithmetic, 2001

[IEEE754]  IEEE Standard for Floating-Point Arithmetic
   Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, 2008

   Accurate Evaluation of a Polynomial in Chebyshev Form, 2011

   Accurate Evaluation of the k-th Derivative of a Polynomial and its Application, 2013

[JLCS10]  H. Jiang, S. Li, L. Cheng, F. Su
   Accurate Evaluation of a Polynomial and its Derivative in Bernstein Form, 2010


[LMT10a]  P. Langlois, M. Martel, L. Thévenoux
   Trade-off Between Accuracy and Time for Automatically Generated Summation Algorithms, 2010

[LMT12]  P. Langlois, M. Martel, L. Thévenoux
   Automatic Code Transformation to Optimize Accuracy and Speed in Floating-Point Arithmetic, 2012

   Handbook of Floating-Point Arithmetic, 2010

   Accurate Sum and Dot Product, 2005

[She97]  J.R. Shewchuk
   Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates, 1997