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

Laurent Thévenoux[1] Philippe Langlois[2] Matthieu Martel[2]

[1]LIP, ENS de Lyon, INRIA, France

[2]University of Perpignan Via Domitia, France

Context and motivation

Context

- Numerical computations can be innacurate: rounding errors
- **Techniques** are available for programmers to improve their numerical programs: expansions, software librairies,. . .
- 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 developpers
- 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 innacurate: rounding errors
- **Techniques** are available for programmers to improve their numerical programs: expansions, software librairies,...
- 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 developpers
- 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 innacurate: rounding errors
- **Techniques** are available for programmers to improve their numerical programs: expansions, software librairies,. . .
- 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 developpers
- 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

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)

$$x = 0.1 \approx \underbrace{\overbrace{0}^{s}\ \overbrace{0\,1\,1\,1\,1\,0\,1\,1}^{e}\ \overbrace{1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0}^{m}}_{\widehat{x} = \textit{binary32} \text{ representation of } x \text{ (before rounding)}}\ 1\,1\,0\,0\,1\,1\,0\,\cdots$$

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

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

where $E_{ref}(\widehat{x})$ is the relative error defined by: $E_{rel}(\widehat{x}) = |x - \widehat{x}|/|x|, \quad x \neq 0.$

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)

$$x = 0.1 \approx \underbrace{\overset{s}{0}\,\overbrace{0\,1\,1\,1\,1\,0\,1\,1}^{e}\,\overbrace{1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0}^{m}}_{\widehat{x} = \textit{binary32} \text{ representation of } x \text{ (before rounding)}}\,1\,1\,0\,0\,1\,1\,0\,\cdots$$

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

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

where $E_{ref}(\widehat{x})$ is the relative error defined by: $E_{rel}(\widehat{x}) = |x - \widehat{x}|/|x|, \quad x \neq 0.$

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)

$$x = 0.1 \approx \underbrace{\overset{s}{0}\ \overbrace{0\,1\,1\,1\,1\,0\,1\,1}^{e}\ \overbrace{1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,1}^{m}}_{\widehat{x} = \textit{binary32} \text{ representation of } x \text{ (before rounding)}}\ \underbrace{1\,1\,0\,0\,1\,1\,0\ \cdots}_{\text{lost bits after rounding-to-nearest RN}(x)}$$

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

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

where $E_{ref}(\widehat{x})$ is the relative error defined by: $E_{rel}(\widehat{x}) = |x - \widehat{x}|/|x|, \quad x \neq 0.$

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)

$$x = 0.1 \approx \underbrace{\overset{s}{0}\ \overbrace{0\,1\,1\,1\,1\,0\,1\,1}^{e}\ \overbrace{1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,1}^{m}}_{\widehat{x} = \textit{binary32} \text{ representation of } x \text{ (before rounding)}}\ \underbrace{1\,1\,0\,0\,1\,1\,0\,\cdots}_{\text{lost bits after rounding-to-nearest RN}(x),\ \#_{sig} = 23}$$

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

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

where $E_{ref}(\widehat{x})$ is the relative error defined by: $E_{rel}(\widehat{x}) = |x - \widehat{x}|/|x|, \quad x \neq 0.$

Error-free transformations (EFTs)

Allow to compute the error generated by a floating-point addition or multiplication

Principle [MBdD10]

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

For the sum...

```
function FastTwoSum(a, b)                ▷ [Dekker, 71]
  x ← RN(a + b)                          ▷ |a| ≥ |b|
  y ← RN((a − x) + b)
  return (x, y)
end function
```

```
function TwoSum(a, b)                    ▷ [Knuth, 69]
  x ← RN(a + b)
  z ← RN(x − a)
  y ← RN((a − (x − z)) + (b − z))
  return (x, y)
end function
```

...and the product

```
function TwoProduct(a, b)                ▷ [Dekker, 71]
  x ← RN(a × b)
  [a_H, a_L] = Split(a)
  [b_H, b_L] = Split(b)
  y ← RN(a_L × b_L − (((x − a_H × b_H) − a_L × b_H) − a_H × b_L))
  return (x, y)
end function
```

```
function Split(a)                        ▷ [Veltkamp, 68]
  c ← RN(f × a)                          ▷ f = 2^⌈p/2⌉ + 1
  a_H ← RN(c − (c − a)),   a_L ← RN(a − a_H)
  return (a_H, a_L)
end function
```

- EFTs are **costly**: 3, 6, and 17 FP operations for `FastTwoSum`, `TwoSum`, and `TwoProduct`
- *fused-multiply-add* instruction can **reduce** the cost of `TwoProduct` to 2

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

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

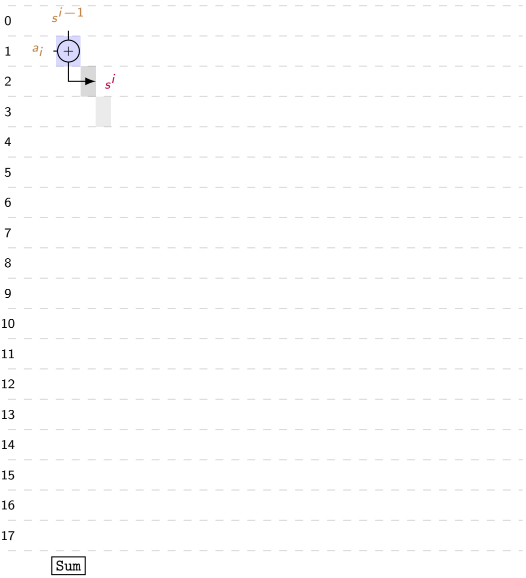
Expansions vs. compensated algorithms

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

function Sum($a_1, a_2, \ldots, a_n$)
  $s^1 \leftarrow a_1$
  for $i = 2 : n$ **do**
    $s^i \leftarrow \mathrm{RN}(s^{i-1} + a_i)$
  end for
  return s^n
end function

function SumDD($a_1, a_2, \ldots, a_n$)
  $s_H^1 \leftarrow a_1$
  $s_L^1 \leftarrow 0$
  for $i = 2 : n$ **do**
    $[s_H^i, s_L^i] = \mathtt{QD\_TwoSum}(s_H^{i-1}, s_L^{i-1}, a_i, \varnothing)$
  end for
  return s_H^n
end function

function Sum2($a_1, a_2, \ldots, a_n$)
  $s^1 \leftarrow a_1$
  $e^1 \leftarrow 0$
  for $i = 2 : n$ **do**
    $[s^i, \epsilon] = \mathtt{TwoSum}(s^{i-1}, a_i)$
    $e^i \leftarrow \mathrm{RN}(e^{i-1} + \epsilon)$
  end for
  return $\mathrm{RN}(s^n + e^n)$
end function



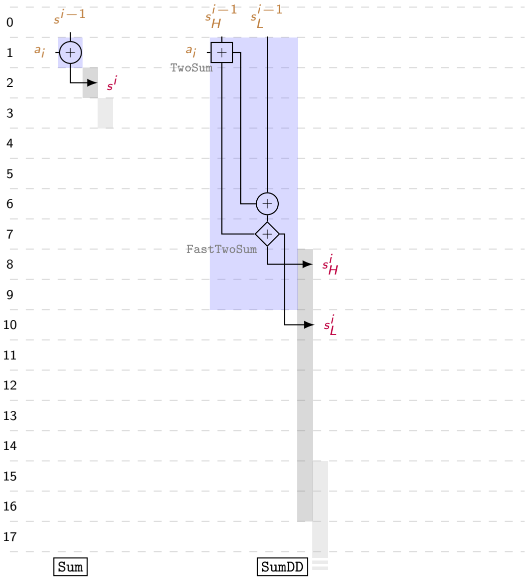
Expansions vs. compensated algorithms

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

function Sum$(a_1, a_2, \ldots, a_n)$
  $s^1 \leftarrow a_1$
  for $i = 2 : n$ **do**
    $s^i \leftarrow \mathrm{RN}(s^{i-1} + a_i)$
  end for
  return s^n
end function

function SumDD$(a_1, a_2, \ldots, a_n)$
  $s_H^1 \leftarrow a_1$
  $s_L^1 \leftarrow 0$
  for $i = 2 : n$ **do**
    $[s_H^i, s_L^i] = \mathtt{QD\_TwoSum}(s_H^{i-1}, s_L^{i-1}, a_i, \varnothing)$
  end for
  return s_H^n
end function

function Sum2$(a_1, a_2, \ldots, a_n)$
  $s^1 \leftarrow a_1$
  $e^1 \leftarrow 0$
  for $i = 2 : n$ **do**
    $[s^i, \epsilon] = \mathtt{TwoSum}(s^{i-1}, a_i)$
    $e^i \leftarrow \mathrm{RN}(e^{i-1} + \epsilon)$
  end for
  return $\mathrm{RN}(s^n + e^n)$
end function



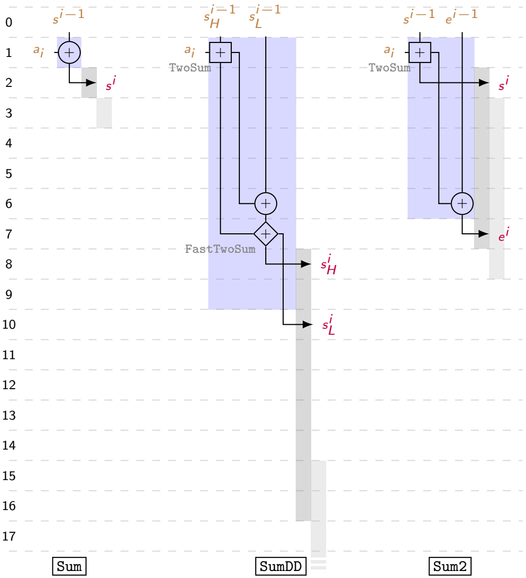
Expansions vs. compensated algorithms

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

function Sum($a_1, a_2, \ldots, a_n$)
  $s^1 \leftarrow a_1$
  for $i = 2 : n$ **do**
    $s^i \leftarrow \mathrm{RN}(s^{i-1} + a_i)$
  end for
  return s^n
end function

function SumDD($a_1, a_2, \ldots, a_n$)
  $s_H^1 \leftarrow a_1$
  $s_L^1 \leftarrow 0$
  for $i = 2 : n$ **do**
    $[s_H^i, s_L^i] = \texttt{QD\_TwoSum}(s_H^{i-1}, s_L^{i-1}, a_i, \varnothing)$
  end for
  return s_H^n
end function

function Sum2($a_1, a_2, \ldots, a_n$)
  $s^1 \leftarrow a_1$
  $e^1 \leftarrow 0$
  for $i = 2 : n$ **do**
    $[s^i, \epsilon] = \texttt{TwoSum}(s^{i-1}, a_i)$
    $e^i \leftarrow \mathrm{RN}(e^{i-1} + \epsilon)$
  end for
  return $\mathrm{RN}(s^n + e^n)$
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 $\mathscr{S}$ 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

```
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;
}
```

Methodology of accuracy improvement

Benefit from the good ILP of compensation automatically: detect FP sequences

Detect floating-point **sequences**

- A sequence is the set $\mathscr{S}$ 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

```
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;
}
```

Methodology of accuracy improvement

Benefit from the good ILP of compensation automatically: detect FP sequences

Detect floating-point **sequences**

- A sequence is the set $\mathscr{S}$ 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

```
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;
}
```

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

  with the following algorithms:

AutoComp_TwoSum$\langle a, \delta_a\rangle, \langle b, \delta_b\rangle$

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

AutoComp_TwoProduct$\langle a, \delta_a\rangle, \langle b, \delta_b\rangle$

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

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

```
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;
}
```

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

  with the following algorithms:

AutoComp_TwoSum$\langle a, \delta_a \rangle, \langle b, \delta_b \rangle$

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

AutoComp_TwoProduct$\langle a, \delta_a \rangle, \langle b, \delta_b \rangle$

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

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

```
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 $s \in \mathscr{S}$:
  - replace floating-point operations by EFTs,
  - and accumulate errors (inherited, generated),

  with the following algorithms:

  $\texttt{AutoComp\_TwoSum}\langle a, \delta_a\rangle, \langle b, \delta_b\rangle$

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

  $\texttt{AutoComp\_TwoProduct}\langle a, \delta_a\rangle, \langle b, \delta_b\rangle$

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

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

```
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 $s \in \mathscr{S}$:
  - replace floating-point operations by EFTs,
  - and accumulate errors (inherited, generated),

  with the following algorithms:

$\texttt{AutoComp\_TwoSum}\langle a, \delta_a \rangle, \langle b, \delta_b \rangle$
$[s, \delta_+] = \texttt{TwoSum}(a, b)$ $\delta_s \leftarrow \mathrm{RN}((\delta_a + \delta_b) + \delta_+)$ **return** $\langle s, \delta_s \rangle$

$\texttt{AutoComp\_TwoProduct}\langle a, \delta_a \rangle, \langle b, \delta_b \rangle$
$[s, \delta_\times] = \texttt{TwoProduct}(a, b)$ $\delta_s \leftarrow \mathrm{RN}(((a \times \delta_b) + (b \times \delta_a)) + \delta_\times)$ **return** $\langle s, \delta_s \rangle$

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

```
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 $s \in \mathscr{S}$:
  - replace floating-point operations by EFTs,
  - and accumulate errors (inherited, generated),

  with the following algorithms:

  $\texttt{AutoComp\_TwoSum}\langle a, \delta_a\rangle, \langle b, \delta_b\rangle$

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

  $\texttt{AutoComp\_TwoProduct}\langle a, \delta_a\rangle, \langle b, \delta_b\rangle$

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

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

```
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 $s \in \mathscr{S}$:
  - replace floating-point operations by EFTs,
  - and accumulate errors (inherited, generated),

  with the following algorithms:

  $\texttt{AutoComp\_TwoSum}\langle a, \delta_a\rangle, \langle b, \delta_b\rangle$

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

  $\texttt{AutoComp\_TwoProduct}\langle a, \delta_a\rangle, \langle b, \delta_b\rangle$

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

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

```
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 $s \in \mathscr{S}$, close(s) means computing

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

  for n being a **result** of s

```
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 $s \in \mathscr{S}$, close(s) means computing

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

  for n being a **result** of s

```
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]

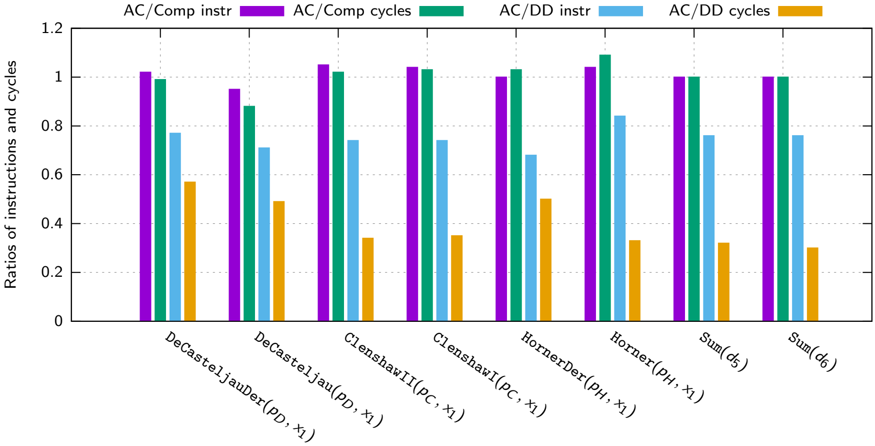
Data	# values	condition number	Data	# values	condition number
d_1	32×10^4	10^8	d_4	32×10^4	10^{16}
d_2	32×10^5	10^8	d_5	32×10^5	10^{16}
d_3	32×10^6	10^8	d_6	32×10^6	10^{16}

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)

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

Experimental results: performance comparison

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

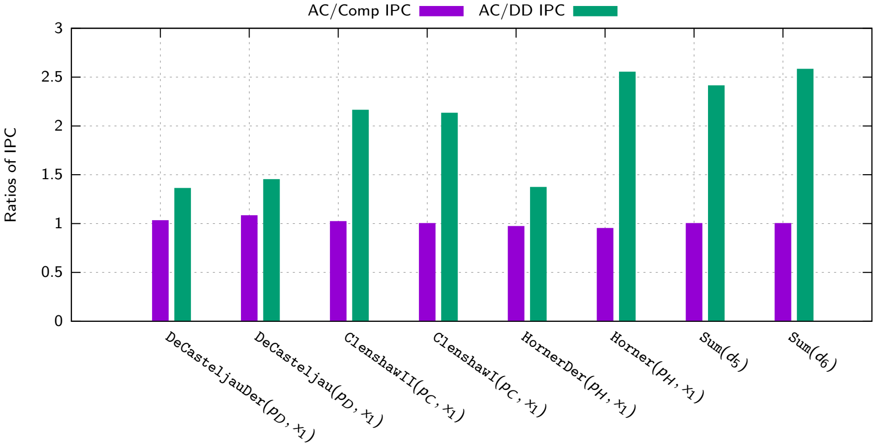


- Compensated algorithms, Comp (manualy implemented), and AC (automatically generated) present **similar performance**
- Comp and AC algorithms present **more ILP** than DD (double-double) ones

↪ measurments done with `papi` tool (API of performance counters) on an `intel Core i5, 2.53GHz` over Linux kernel 3.2 and `gcc4.6.3 -O2`
↪ *ideal measurements (not shown here) validate experimental measures*

Experimental results: performance comparison

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

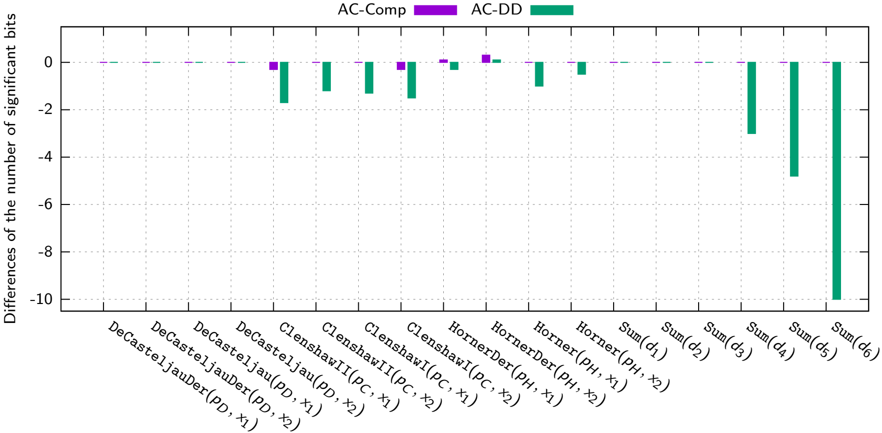


- Compensated algorithms, Comp (manualy implemented), and AC (automatically generated) present **similar performance**
- Comp and AC algorithms present **more ILP** than DD (double-double) ones

↪ measurments done with `papi` tool (API of performance counters) on an `intel Core i5, 2.53GHz` over Linux kernel 3.2 and `gcc4.6.3 -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 (manualy 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

  $\hookrightarrow$ 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

  $\hookrightarrow$ *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**)

  $\hookrightarrow$ `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

  $\hookrightarrow$ 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

  $\hookrightarrow$ *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**)

  $\hookrightarrow$ `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 Accurate, Validated and Fast Polynomial Evaluation, 2009

[HBL01] Y. Hida, X.S. Li, D.H. Bailey.
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

[JBL11] H. Jiang, R. Barrio, H. Li, X. Liao, L. Cheng, F. Su
Accurate Evaluation of a Polynomial in Chebyshev Form, 2011

[JGH13] H. Jiang, S. Graillat, C. Hu, S. Li, X. Liao, L. Chang, F. Su
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

[LL07] P. Langlois, N. Louvet.
More Instruction Level Parallelism Explains the Actual Efficiency of Compensated Algorithms, 2007

[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

[MBdD10] J.M. Muller, N. Brisebarre, F. de Dinechin, C.P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, S. Torres.
Handbook of Floating-Point Arithmetic, 2010

[ROO05] S.M. Rump, T. Ogita, S.Oishi.
Accurate Sum and Dot Product, 2005

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