4ccurate 4lgorithms in Floating Point 4rithmetic

Philippe LANGLOIS[1]

[1]DALI Research Team, University of Perpignan, France

SCAN 2006 at Duisburg, Germany, 26-29 September 2006

Compensated algorithms in floating point arithmetic

Algorithm accuracy *vs.* condition number

For a given working precision $\mathtt{u}$,

- Backward stable algorithms:
  The solution accuracy is no worse than

  condition number $\times$ computing precision ($\times$ O(problem dimension))

  - Example: arithmetic operators, classic algorithms for summation, dot product, Horner's method, backward substitution, GEPP, ...
  - How to manage ill-conditionned cases, *i.e.*, when condition number $> 1/\mathtt{u}$?

Algorithm accuracy *vs.* condition number

For a given working precision $\mathtt{u}$,

- Backward stable algorithms:
  The solution accuracy is no worse than

  condition number $\times$ computing precision ($\times$ O(problem dimension))

  - Example: arithmetic operators, classic algorithms for summation, dot product, Horner's method, backward substitution, GEPP, ...
  - How to manage ill-conditionned cases, *i.e.*, when condition number $> 1/\mathtt{u}$?
- Highly accurate or even faithful algorithms:
  - The solution accuracy is independant of the condition number (for a range of it)
  - A faithful rounded result is sometimes available
  - Example: Priest's algorithm, distillation algorithms, Rump-Ogita-Oishi for summation and dot product, iterative refinement with extra precision, (polynomial evaluation of a canonical form –needs the roots!, ...).

Compensated algorithms

Compensated algorithm corrects the generated rounding error

- the solution accuracy is no worse than

$$\boxed{\text{computing precision} + \text{condition number} \times (\text{computing precision})^K;}$$

- the computation is performed only with the working precision $\mathtt{u}$;
- the performances are no worse than expansion libraries (double-double, quad-double) and even more general ones (arprec, MPFR, ...).
- It also returns a validated dynamic error bound. (bonus)

Compensated algorithms

Compensated algorithm corrects the generated rounding error

- the solution accuracy is no worse than

$$\boxed{\text{computing precision} + \text{condition number} \times (\text{computing precision})^K;}$$

- the computation is performed only with the working precision $\mathtt{u}$;
- the performances are no worse than expansion libraries (double-double, quad-double) and even more general ones (arprec, MPFR, ...).
- It also returns a validated dynamic error bound. (bonus)

- Accuracy as if computed in twice the working precision
  - alternative to double-double in *e.g.*, XBLAS
  - Compensated summation: Neumaier (74), Sum2 in Ogita-Rump-Oishi (05)
  - Compensated Horner algorithm and compensated backward substitution

Compensated algorithms

Compensated algorithm corrects the generated rounding error

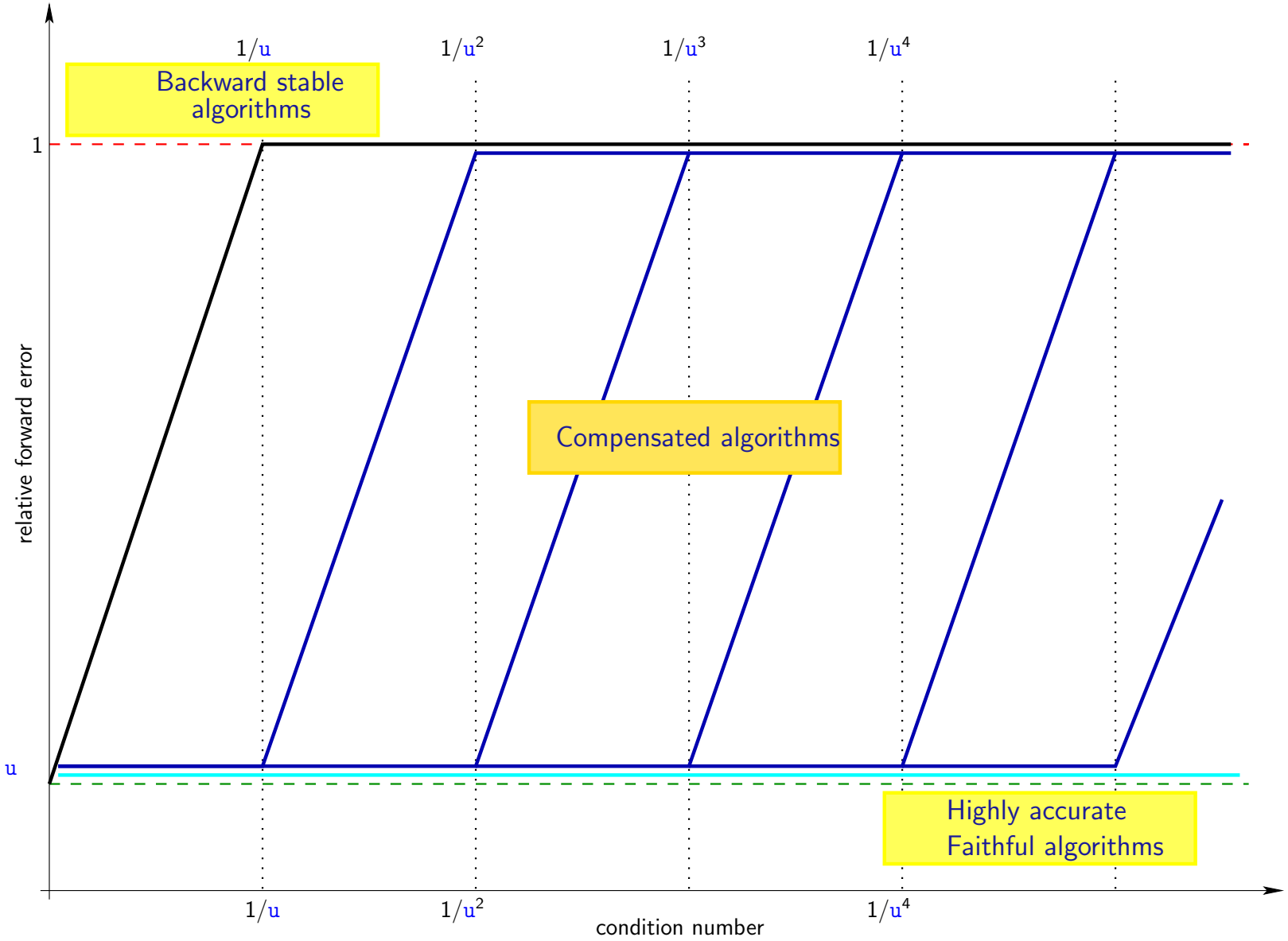
- the solution accuracy is no worse than

$$\boxed{\text{computing precision} + \text{condition number} \times (\text{computing precision})^K;}$$

- the computation is performed only with the working precision $\mathtt{u}$;
- the performances are no worse than expansion libraries (double-double, quad-double) and even more general ones (arprec, MPFR, ...).
- It also returns a validated dynamic error bound. (bonus)

- Accuracy as if computed in twice the working precision
  - alternative to double-double in *e.g.*, XBLAS
  - Compensated summation: Neumaier (74), Sum2 in Ogita-Rump-Oishi (05)
  - Compensated Horner algorithm and compensated backward substitution
- Accuracy as if computed in K times the working precision
  - Compensated summation: SumK Ogita-Rump-Oishi (05)
  - Compensated Horner algorithm

Algorithm accuracy *vs.* condition number



Introducing example: rounding errors when solving $Tx = b$

Classic substitution algorithm computes $x_i = \left(b_i - \sum_{j=1}^{i-1} t_{i,j} x_j\right) / t_{i,i}, i = 1 : n$

Algorithm (...and its generated rounding errors –inner loop)

$s_{i,0} = b_i$
for $j = 1 : i - 1$ *loop*
  $p_{i,j} = t_{i,j} \otimes \widehat{x}_j$ { *generates* $\pi_{i,j}$ }
  $s_{i,j} = s_{i,j-1} \ominus p_{i,j}$ { *generates* $\sigma_{i,j}$ }
end loop
$\widehat{x}_i = s_{i,i-1} \oslash t_{i,i}$ { *generates* δ_i }

The global forward error in computed $\widehat{x}$ is $(\Delta x_i)_{i=1:n}$,

$$\Delta x_i = x_i - \widehat{x}_i = \frac{1}{t_{i,i}} \left[\sum_{j=1}^{i-1} (\sigma_{i,j} - \pi_{i,j} - t_{i,j} \times \Delta x_j) \right] + \delta_i$$

Algorithm (Inlining the computation of the correcting term $\widehat{\Delta}x$)

for $i = 1 : n$ *loop*

$s_{i,0} = b_i$

$u_{i,0} = 0$

for $j = 1 : i - 1$ *loop*

$(p_{i,j}, \pi_{i,j}) = TwoProd(t_{i,j}, \widehat{x}_j)$

$(s_{i,j}, \sigma_{i,j}) = TwoSum(s_{i,j-1}, -p_{i,j})$

$u_{i,j} = u_{i,j-1} \ominus (t_{i,j} \otimes \widehat{\Delta}x_j \oplus \pi_{i,j} \ominus \sigma_{i,j})$

end loop

$(\widehat{x}_i, \delta_i) = ApproxTwoDiv(s_{i,i-1}, t_{i,i})$

$\widehat{\Delta}x_i = u_{i,i-1} \oslash t_{i,i} \oplus \delta_i$

end loop

Algorithm (Inlining the computation of the correcting term $\widehat{\Delta}x$)

for $i = 1 : n$ *loop*

$s_{i,0} = b_i$

$u_{i,0} = 0$

for $j = 1 : i - 1$ *loop*

$(p_{i,j}, \pi_{i,j}) = TwoProd(t_{i,j}, \widehat{x}_j)$

$(s_{i,j}, \sigma_{i,j}) = TwoSum(s_{i,j-1}, -p_{i,j})$

$u_{i,j} = u_{i,j-1} \ominus (t_{i,j} \otimes \widehat{\Delta}x_j \oplus \pi_{i,j} \ominus \sigma_{i,j})$

end loop

$(\widehat{x}_i, \delta_i) = ApproxTwoDiv(s_{i,i-1}, t_{i,i})$

$\widehat{\Delta}x_i = u_{i,i-1} \oslash t_{i,i} \oplus \delta_i$

end loop

Algorithm (Compensating $\widehat{x}$ adding the correction $\widehat{\Delta}x$)

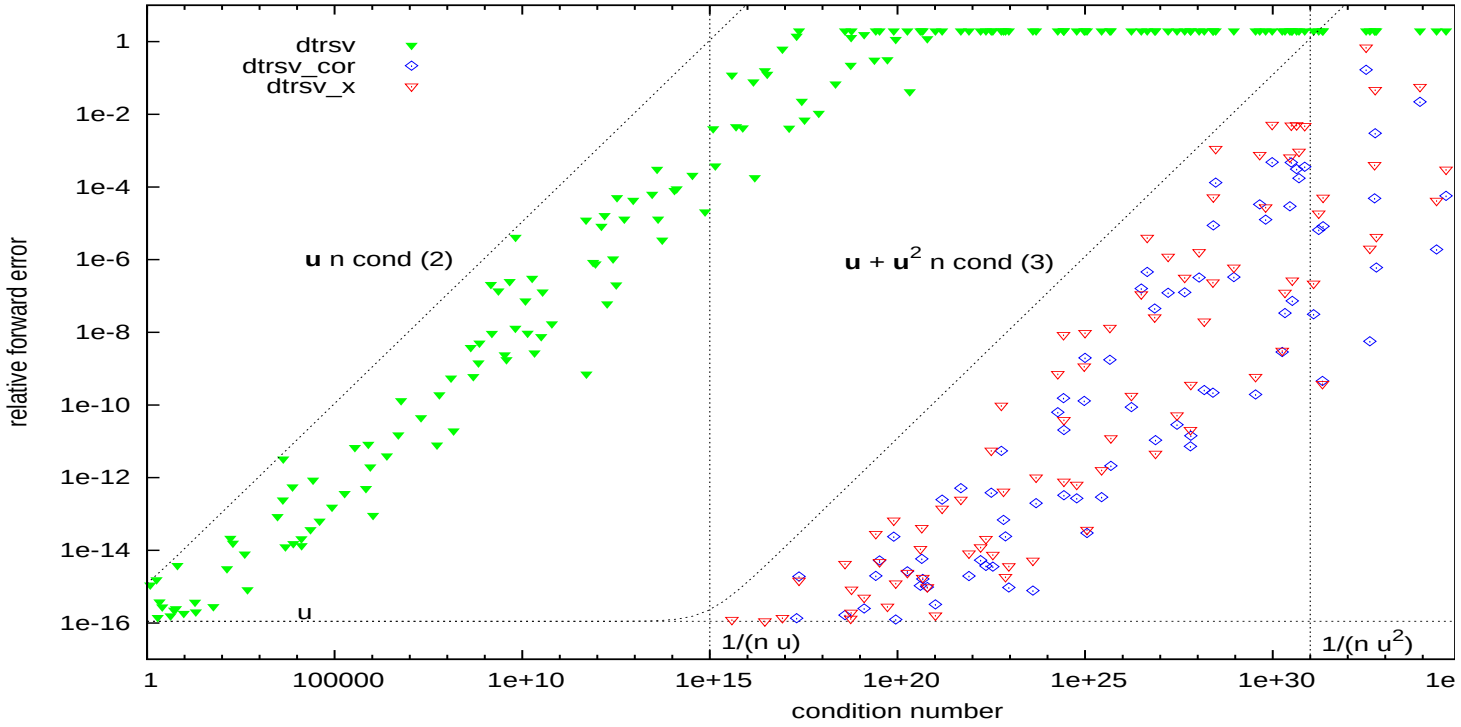
for $i = 1 : n$ *loop*

$\overline{x}_i = \widehat{x}_i \oplus \widehat{\Delta}x_i$

end loop

Accuracy as in doubled precision for $Tx = b$

dtrsv: IEEE-754 double precision substitution algorithm,
dtrsv_cor: compensated substitution algorithm,
dtrsv_x: XBLAS substitution with inner computation in double-double arithmetic



- Compensated substitution accuracy as if computed in doubled precision.
- Actual accuracy of compensated substitution is bounded as

$$\|x - \widehat{x}\|_\infty / \|x\|_\infty \lesssim \mathtt{u} + n \times \mathrm{cond}(T, x) \times \mathtt{u}^2.$$

Measured running times are interesting

The general behavior:

1. Overhead to double the precision: measured running times are significantly better than expected from the theoretical flops count.
2. Compensated versus double-double: compensated algorithm runs about two times faster than its double-double counterpart.

An example
Env. 1: Intel Celeron: 2.4GHz, 256kB L2 cache - GCC 3.4.1
Env. 2: Pentium 4: 3.0GHz, 1024kB L2 cache - GCC 3.4.1

ratio	env.	measured times min	mean	max	theoretical # flops
compensated / double	1	2.17	2.95	9.70	13.5
	2	4.50	8.14	9.95	
double-double / double	1	4.02	5.81	19.66	22.5
	2	8.82	16.10	19.60	
double-double/compensated	1	1.81	1.95	2.07	1.67
	2	1.87	1.98	2.03	

Our first step towards compensated algorithms (1998-2003)

CENA: a software for automatic linear correction of rounding errors

- Motivation: automatic improvement and verification of the accuracy of a result computed by a program considered as a black box.
- Principle: compute a linear approximation of the global absolute error w.r.t. the generated elementary rounding error and use it as a correcting term.
- CENA = algorithmic differentiation + computing elementary rounding error + running error analysis + overloaded facilities (*e.g.*, Fortran 90).

Our first step towards compensated algorithms (1998-2003)

CENA: a software for automatic linear correction of rounding errors

- Motivation: automatic improvement and verification of the accuracy of a result computed by a program considered as a black box.
- Principle: compute a linear approximation of the global absolute error w.r.t. the generated elementary rounding error and use it as a correcting term.
- CENA = algorithmic differentiation + computing elementary rounding error + running error analysis + overloaded facilities (*e.g.*, Fortran 90).
- Should be considered as a development tool: easy to use, help to identify where to use more precision.
- Step 2: inlining the correction!
- Much more details in BIT 00, JCAM 01, TCS 03.

Classic error free transformations

Error free transformations are properties and algorithms to compute the generated elementary rounding errors at the CWP.

$$a, b \text{ entries in } \mathbb{F}, \quad a \circ b = \mathtt{fl}\,(a \circ b) + \varepsilon, \quad \text{with } \varepsilon \text{ output in } \mathbb{F}$$

Related works

- EFT are so named by Ogita-Rump-Oishi (SISC, 05)
  sum-and-roundoff pair, product-and-roundoff pair in Priest (PhD, 92),
  no special name before.
- Recent and interesting discussion in Nievergelt (TOMS,04)

Algorithms to compute the error term are extremely dependent to properties of the floating point arithmetic *e.g.*, basis of representation, rounding mode ...

Classic error free transformations

Be careful: some of the following relations only apply for IEEE-754 arithmetic with rounding to the nearest.

- Summation: Møller (65), Kahan (65), Knuth (74), Priest(91).
  $(s, \sigma) = \mathrm{TwoSum}\,(a, b)$ is such that $a + b = s + \sigma$ and $s = a \oplus b$.
- Product: Veltkampt and Dekker (72), ...
  $(p, \pi) = \mathrm{TwoProd}\,(a, b)$ is such that $a \times b = p + \pi$ and $p = a \otimes b$.

Classic error free transformations

Be careful: some of the following relations only apply for IEEE-754 arithmetic with rounding to the nearest.

- Summation: Møller (65), Kahan (65), Knuth (74), Priest(91).
  $(s, \sigma) = \text{TwoSum}\,(a, b)$ is such that $a + b = s + \sigma$ and $s = a \oplus b$.
- Product: Veltkampt and Dekker (72), ...
  $(p, \pi) = \text{TwoProd}\,(a, b)$ is such that $a \times b = p + \pi$ and $p = a \otimes b$.
- FMA: Boldo+Muller (05)
  $(p, \delta_1, \delta_2) = \text{ErrFMA}(a, b, c)$ is such that $p = \text{FMA}(a, b, c)$ and $a \times b + c = p + \delta_1 + \delta_2$.

Classic error free transformations

Be careful: some of the following relations only apply for IEEE-754 arithmetic with rounding to the nearest.

- Summation: Møller (65), Kahan (65), Knuth (74), Priest(91).
  $(s, \sigma) = \mathrm{TwoSum}\,(a, b)$ is such that $a + b = s + \sigma$ and $s = a \oplus b$.
- Product: Veltkampt and Dekker (72), ...
  $(p, \pi) = \mathrm{TwoProd}\,(a, b)$ is such that $a \times b = p + \pi$ and $p = a \otimes b$.
- FMA: Boldo+Muller (05)
  $(p, \delta_1, \delta_2) = \mathrm{ErrFMA}(a, b, c)$ is such that $p = \mathrm{FMA}(a, b, c)$ and $a \times b + c = p + \delta_1 + \delta_2$.
- Division and square root: Pichat (77), Markstein (90) If $d = a \oslash b$, then $r = a - d \times b$ is a floating point value. EFT exist for the remainder and the square and are not useful hereafter.

Not error free transformations

- Approximate division: Langlois+Nativel (98)
  $(d, \widehat{\delta}) = \text{ApproxTwoDiv}\,(a, b)$ is such that $a/b = d + \delta$ and $d = a \oslash b$, $|\widehat{\delta} - \delta| \leq \mathtt{u}|\delta|$.
- Approximate square root: Langlois+Nativel (98)

Algorithms to compute EFT: two examples

Algorithm (fast-two-sum, Dekker (71))

function $s, \sigma = \text{FastTwoSum}\,(a, b)$
% $|a| \geq |b|$
$s := a \oplus b;\ b_{app} := s \ominus a;\ \sigma := b \ominus b_{app}$

cost: 3 flops and 1 test

Algorithm (algorithm without pre-ordering or two-sum, Knuth (74))

function $s, \sigma = \text{TwoSum}\,(a, b)$
$s := a \oplus b;$
$b_{app} := s \ominus a; \quad a_{app} := s \ominus b_{app};$
$\delta_b := b \ominus b_{app}; \quad \delta_a := a \ominus a_{app};$
$\sigma := \delta_a \oplus \delta_b;$

cost: 6 flops

Error Free Transformations: summary

- EFT for proving theorems on error bounds
  - Standard model: $a \circ b = \mathtt{fl}\,(a \circ b)\,(1+\eta)$ with $|\eta| \leq \mathtt{u}$;
  - EFT: $a \circ b = \mathtt{fl}\,(a \circ b) + \varepsilon$
  - Mathematical equalities between floating point entries
- Algorithms to compute EFT
  - yield generated rounding errors,
  - are performed at the working precision,
  - without change of rounding mode,
  - better benefit of hardware units: pipeline
- Cost in #flops

TwoSum	FastTwoSum	TwoProd	TwoProdFMA	ThreeFMA
6	3 (+1 test)	17	2	17

Accuracy of the Horner algorithm

We consider the polynomial

$$p(x) = \sum_{i=0}^{n} a_i x^i,$$

with $a_i \in \mathbb{F}$, $x \in \mathbb{F}$

Algorithm (Horner)

function $r_0 = \text{Horner}\,(p, x)$
$r_n = a_n$
for $i = n-1 : -1 : 0$
  $r_i = r_{i+1} \otimes x \oplus a_i$
end

- $\text{cond}(p, x) = \frac{\sum |a_i x^i|}{|p(x)|} \geq 1$
- Relative accuracy of the evaluation with the Horner scheme:

$$\frac{|\text{Horner}\,(p, x) - p(x)|}{|p(x)|} \leq \underbrace{\gamma_{2n}}_{\approx 2n\mathtt{u}} \text{cond}(p, x)$$

- $\mathtt{u}$ is the computing precision:
  IEEE-754 double, 53-bits mantissa, rounding to the nearest $\Rightarrow \mathtt{u} = 2^{-53}$.

EFT for the Horner scheme

Let p be a polynomial of degree n with coefficients in $\mathbb{F}$. If no underflow occurs,

$$\boxed{p(x) = \text{Horner}\,(p, x) + (p_\pi + p_\sigma)(x),}$$

with $p_\pi(X) = \sum_{i=0}^{n-1} \pi_i X^i, \quad p_\sigma(X) = \sum_{i=0}^{n-1} \sigma_i X^i$, and π_i and σ_i in $\mathbb{F}$.

EFT for the Horner scheme

Let p be a polynomial of degree n with coefficients in $\mathbb{F}$. If no underflow occurs,

$$\boxed{p(x) = \text{Horner}\,(p, x) + (p_\pi + p_\sigma)(x),}$$

with $p_\pi(X) = \sum_{i=0}^{n-1} \pi_i X^i$, $\quad p_\sigma(X) = \sum_{i=0}^{n-1} \sigma_i X^i$, and π_i and σ_i in $\mathbb{F}$.

Algorithm (Horner scheme)

function $r_0 = \text{Horner}\,(p, x)$
$r_n = a_n$
for $i = n-1 : -1 : 0$ *loop*
  $p_i = r_{i+1} \otimes x$
    % rounding error $\pi_i \in \mathbb{F}$
  $r_i = p_i \oplus a_i$
    % rounding error $\sigma_i \in \mathbb{F}$
end loop

Algorithm (EFT for Horner)

function $[r_0, p_\pi, p_\sigma] = \text{EFTHorner}\,(p, x)$
$r_n = a_n$
for $i = n-1 : -1 : 0$ *loop*
  $[p_i, \pi_i] = \text{TwoProd}\,(r_{i+1}, x)$
  $[r_i, \sigma_i] = \text{TwoSum}\,(p_i, a_i)$
  $p_\pi[i] = \pi_i$
  $p_\sigma[i] = \sigma_i$
end loop

The compensated Horner algorithm and its accuracy

$(p_\pi + p_\sigma)(x)$ is the global error affecting Horner (p, x).
$\Rightarrow$ we compute an approximate of $(p_\pi + p_\sigma)(x)$ as a correcting term.

Algorithm (Compensated Horner algorithm)

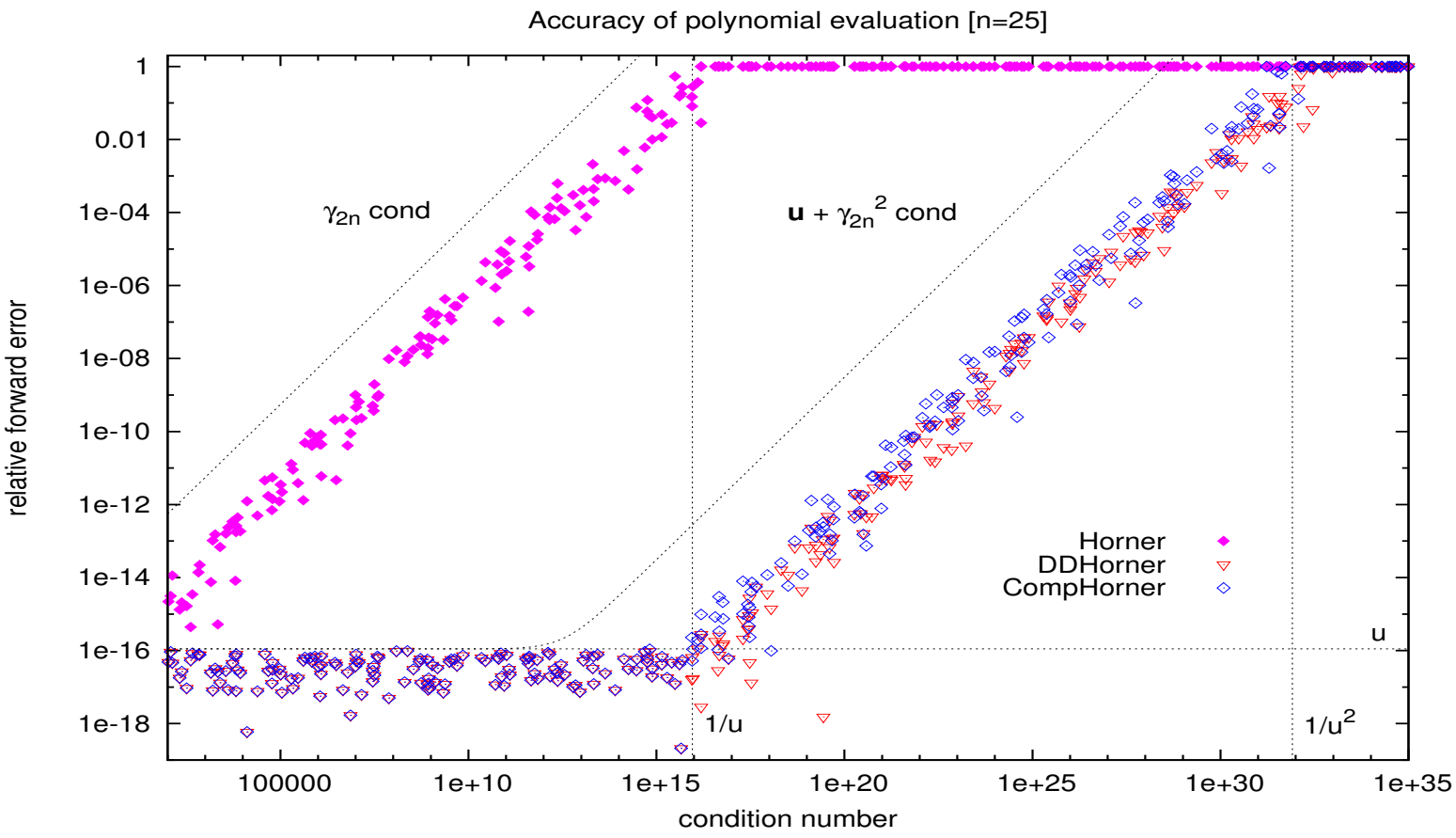
function $\overline{r} = \text{CompHorner}\,(p, x)$
$[\widehat{r}, p_\pi, p_\sigma] = \text{EFTHorner}\,(p, x)$ % $\widehat{r} = \text{Horner}\,(p, x)$
$\widehat{c} = \text{Horner}\,(p_\pi \oplus p_\sigma, x)$
$\overline{r} = \widehat{r} \oplus \widehat{c}$

Theorem

Given p a polynomial with floating point coefficients, and x a floating point value,

$$|\text{CompHorner}\,(p, x) - p(x)|/|p(x)| \leq \mathtt{u} + \underbrace{\gamma_{2n}^2}_{2n^2\mathtt{u}^2} \text{cond}(p, x).$$

. . . as if computed in doubled precision



- Expected accuracy improvement compared to Horner
- The *a priori* bound is correct but pessimistic
- Same accuracy is provided by CompensatedHorner and DDHorner (Horner with inner computation with double-doubles).

How to validate the accuracy of the computed evaluation?

Classic dynamic approaches:

- Compensated RoT or approximate
- Wilkinson's running error bound
- Interval arithmetic

A dynamic and validated error bound

Theorem

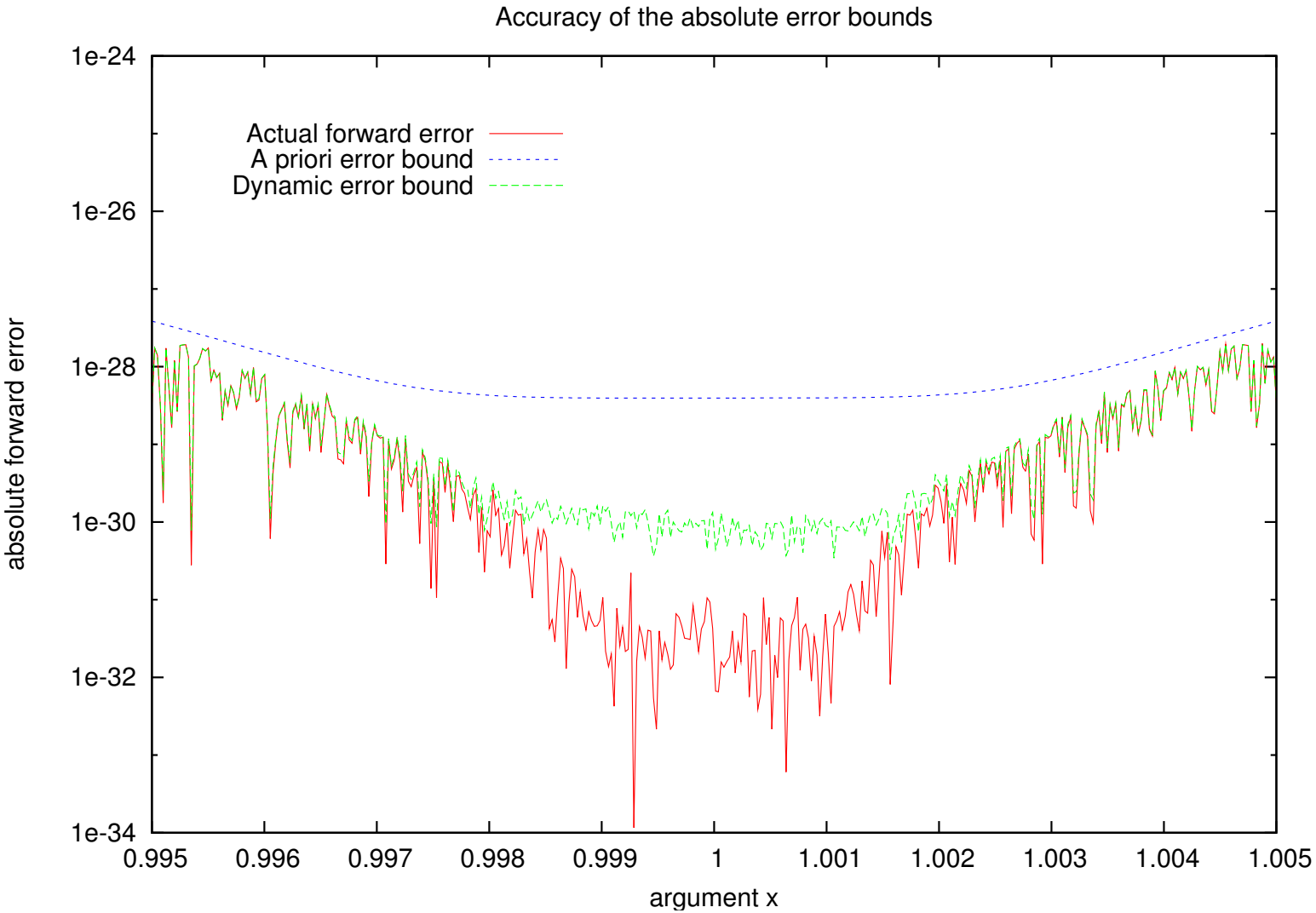
Given a polynomial p with floating point coefficients, and a floating point value x, we consider $\mathrm{res} = \mathrm{CompensatedHorner}\,(p, x)$.
The absolute forward error affecting the evaluation is bounded according to

$$|\mathrm{CompensatedHorner}\,(p, x) - p(x)| \leq$$
$$fl((\mathtt{u}|\mathrm{res}| + (\gamma_{4n+2}\mathrm{HornerSum}\,(|p_\pi|, |p_\sigma|, |x|) + 2\mathtt{u}^2|\mathrm{res}|))).$$

- The dynamic bound is computed in entire FPA with round to the nearest mode only.
- The dynamic bound improves the *a priori* bound.
- Proof uses EFT and the standard model of FPA (as in Ogita-Rump-Oishi paper)

The dynamic bound improves the *a priori* bound

Example: evaluating $p_5(x) = (x-1)^5$ for 1024 points around $x = 1$

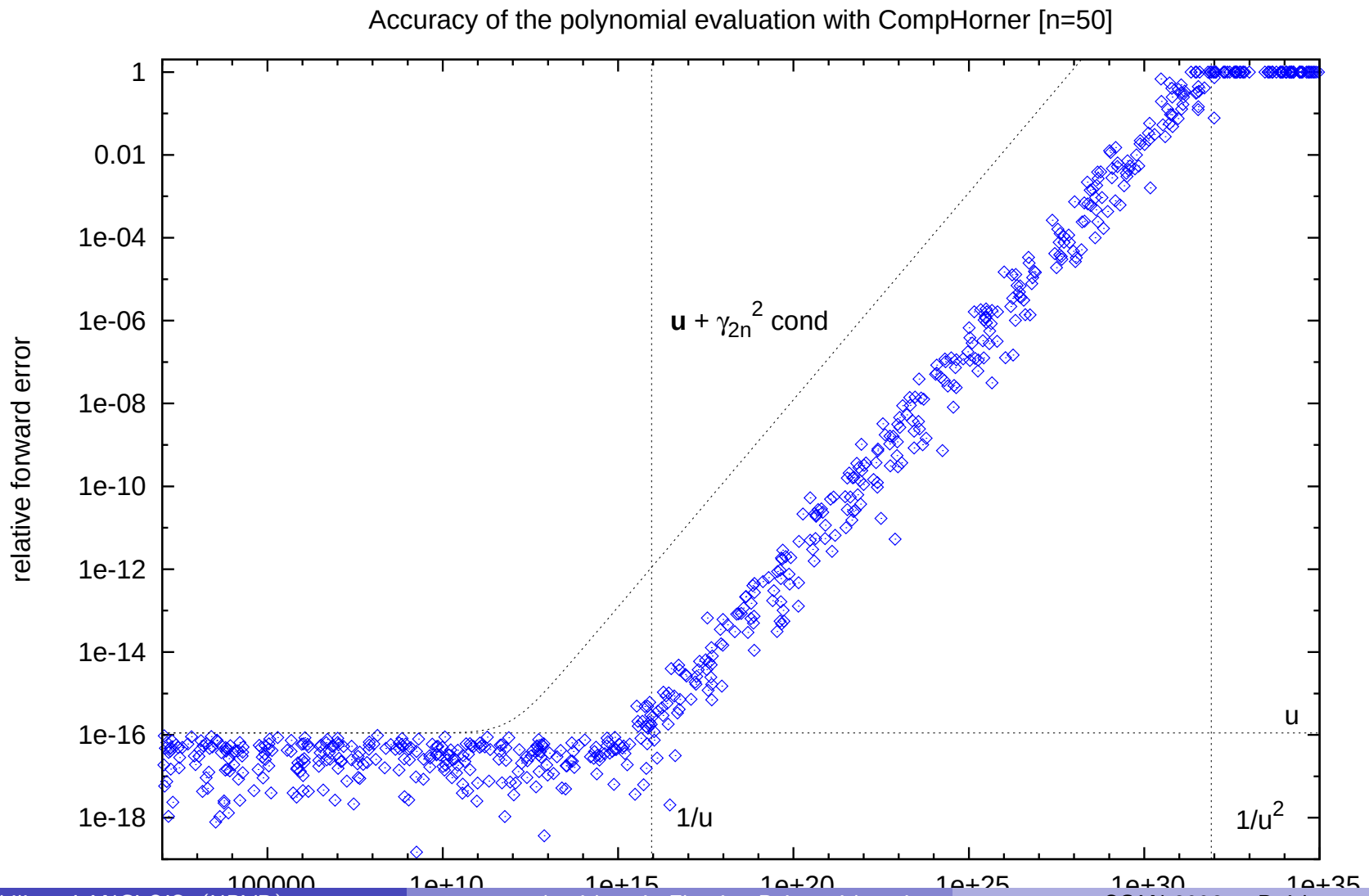


Motivation for faithfully rounding a real value

- A faithful rounded result = one of the two neighbouring floating point values
- Interest: sign determination, *e.g.*, for geometric predicates

Motivation for faithfully rounding a real value

- A faithful rounded result = one of the two neighbouring floating point values
- Interest: sign determination, *e.g.*, for geometric predicates



Two sufficient conditions for faithful rounding

Let p be a polynomial of degree n, x be a floating point value.

Theorem

CompHorner (p, x) *is a faithful rounding of* $p(x)$ *while one of the next two bounds is satisfied.*

An *a priori* bound for the condition number:

$$\mathrm{cond}(p, x) < \frac{1 - \mathtt{u}}{2 + \mathtt{u}} \frac{\mathtt{u}}{\gamma_{2n}^2} \approx 1/(8n^2\mathtt{u}).$$

A dynamic bound, validated for floating-point evaluation:

$$fl\left(\frac{\widehat{\gamma}_{2n-1}\mathrm{Horner}\left(|p_\pi| \oplus |p_\sigma|, |x|\right)}{1 - 2(n+1)\mathtt{u}}\right) < \frac{\mathtt{u}}{2}|\mathrm{CompHorner}\,(p, x)\,|.$$

Faithful rounding: explanations

- The previous error bound is too large to prove faithful rounding:

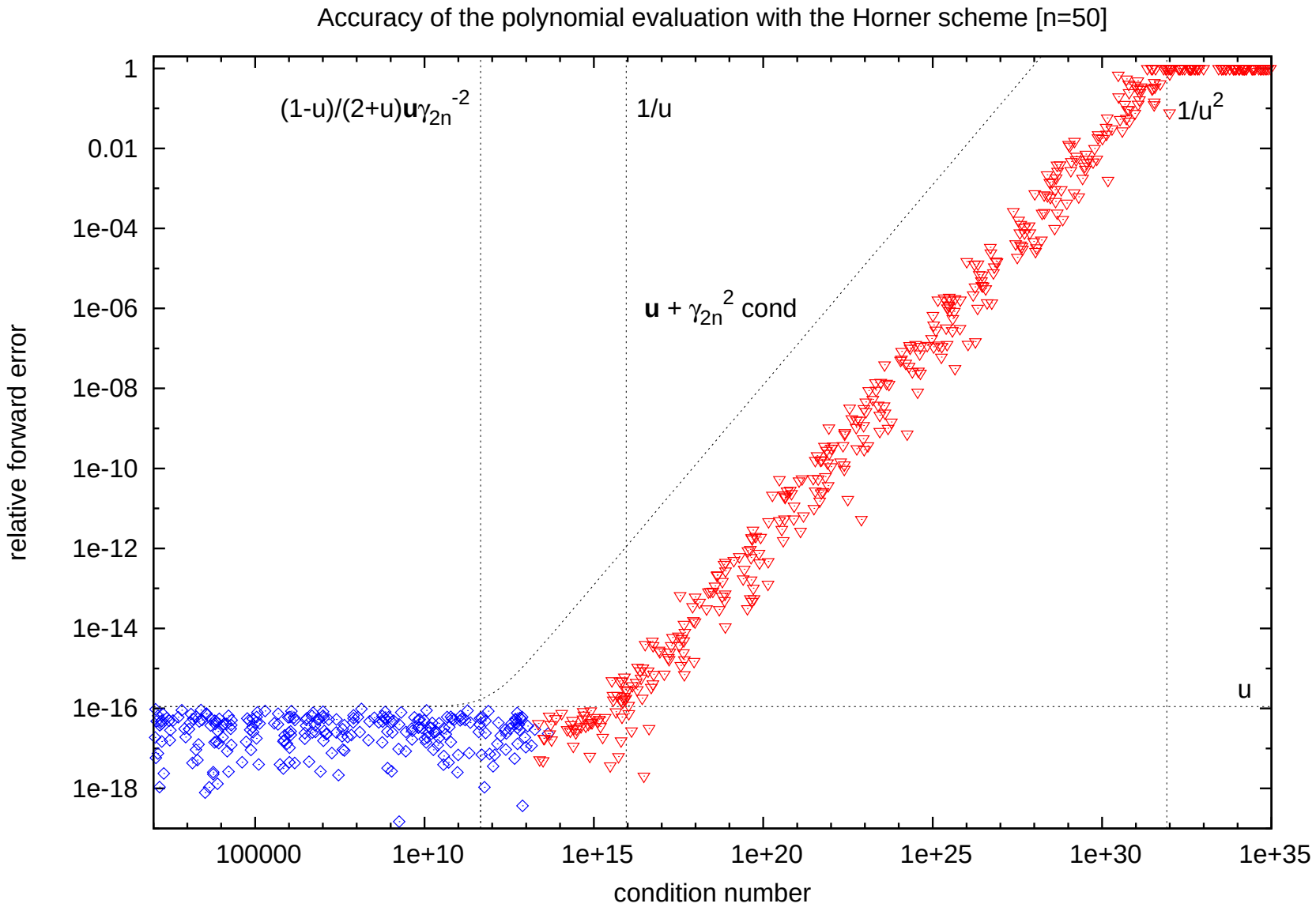
$$\frac{|\text{CompHorner}\,(p,x) - p(x)|}{|p(x)|} \leq \mathtt{u} + \underbrace{\gamma_{2n}^2}_{2n^2\mathtt{u}^2} \text{cond}(p,x)$$

- We use a result from Rump-Ogita-Oishi (nov. 2005).
- In practice for IEEE double precision

  A priori bound: cond up to 10^{12} for $n = 20$ and up to 10^{11} for $n = 100$

  Dynamic bound: previous bound is improved but still not optimal.

Checking for faithful Horner



Two compensated Horner algorithms with FMA

Left: Same EFT with FMA inside TwoProd:

$$p(x) = \text{Horner}\,(p, x) + (p_\pi + p_\sigma)(x)$$

Right: A variant of previous EFT when FMA is the inner operation of the Horner algorithm:

$$p(x) = \text{HornerFMA}\,(p, x) + (p_\varepsilon + p_\varphi)(x)$$

Algorithm (EFT with FMA)

$[h, p_\pi, p_\sigma] = \text{EFTHornerFMA}(p, x)$
$s_n = a_n$
for $i = n-1 : -1 : 0$ *loop*
  $[p_i, \pi_i] = \mathit{TwoProdFMA}(s_{i+1}, x)$
  $[s_i, \sigma_i] = \mathit{TwoSum}(p_i, a_i)$
  $p_\pi[i] = \pi_i$
  $p_\sigma[i] = \sigma_i$
end loop
$h = s_0$

Algorithm (EFT for FMA)

$[h, p_\varepsilon, p_\varphi] = \text{EFTHornerFMA}(p, x)$
$s_n = a_n$
for $i = n-1 : -1 : 0$ *loop*
  $[u_i, \varepsilon_i, \varphi_i] = \mathit{ThreeFMA}(u_{i+1}, x, a_i)$
  $p_\varepsilon[i] = \varepsilon_i$
  $p_\varphi[i] = \varphi_i$
end loop
$h = u_0$

Corresponding results

Theorem

Given p a polynomial with floating point coefficients, and x a floating point value, we have

$$\begin{aligned}\frac{|\mathrm{CompHornerFMA}\,(p,x)-p(x)|}{|p(x)|} &\leq \mathtt{u}+(1+\mathtt{u})\gamma_n\gamma_{2n}cond(p,x)\\ &\leq \mathtt{u}+2n^2\mathtt{u}^2cond(p,x)+O(\mathtt{u}^3).\end{aligned}$$

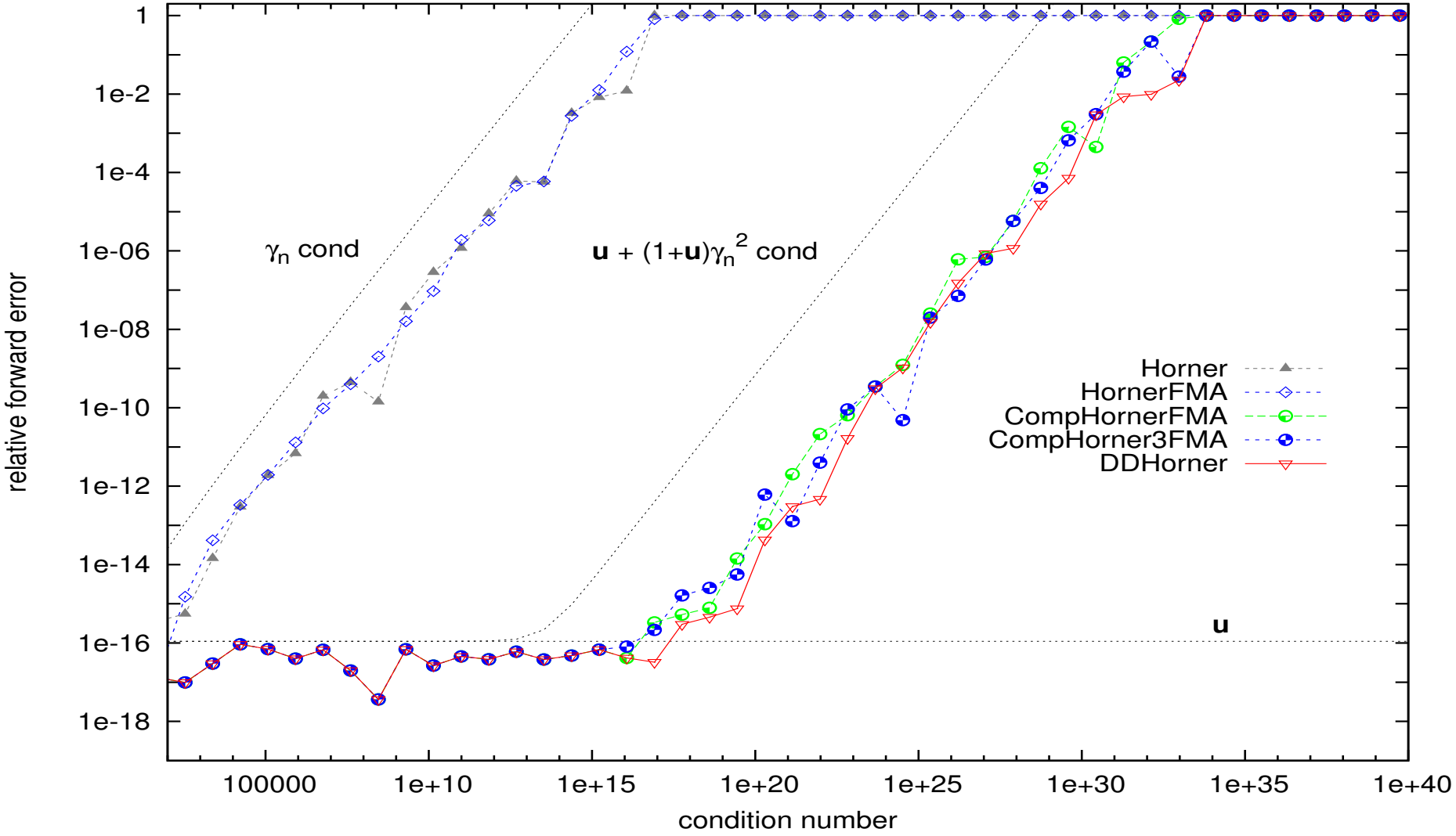
and CompHornerFMA *flops counts is* $10n-1$.

We have also

$$\begin{aligned}\frac{|\mathrm{CompHornerThreeFMA}\,(p,x)-p(x)|}{|p(x)|} &\leq \mathtt{u}+(1+\mathtt{u})\gamma_n^2cond(p,x)\\ &\leq \mathtt{u}+n^2\mathtt{u}^2cond(p,x)+O(u^3);.\end{aligned}$$

and CompHornerThreeFMA *flops counts is* $19n$.

Accuracy experiments: no surprise



For the same output accuracy, prefer FMA to compensate Horner without FMA

environment	$\frac{\text{CompHornerFMA}}{\text{HornerFMA}}$ mean	theo.	$\frac{\text{CompHorner3FMA}}{\text{HornerFMA}}$ mean	theo.	$\frac{\text{DDHorner}}{\text{HornerFMA}}$ mean	theo.
Itanium 1, 733MHz, GCC 2.96	2.6	10	4.8	19	6.5	20
Itanium 2, 900MHz, GCC 3.3.5	2.5	10	4.4	19	7.9	20
Itanium 2, 1.6GHz, GCC 3.4.4	3.8	10	6.0	19	7.3	20

How to explain measured performances?

How to explain measured performances?

- Theoretical ratios (flops):

$$\frac{\text{CompHorner}}{\text{Horner}} \sim 10.5 \qquad \frac{\text{DDHorner}}{\text{Horner}} \sim 14 \qquad \frac{\text{DDHorner}}{\text{CompHorner}} \sim 1.33$$

- Some practical ratios (average running times for degree 5 to 200):

		$\frac{\text{CompHorner}}{\text{Horner}}$	$\frac{\text{DDHorner}}{\text{Horner}}$	$\frac{\text{DDHorner}}{\text{CompHorner}}$
Pentium 4, 3.00 GHz	GCC 3.3.5	3.77	10.00	2.66
	ICC 9.1	3.06	8.88	2.92
Athlon 64, 2.00 GHz	GCC 4.0.1	3.89	10.48	2.70
Itanium 2, 1.4 GHz	GCC 3.4.6	3.64	5.50	1.51
	ICC 9.1	1.87	8.78	4.67
		$\sim 2-4$	$\sim 5-10$	$\sim 2-5$

How to explain measured performances?

- Theoretical ratios (flops):

$$\frac{\text{CompHorner}}{\text{Horner}} \sim 10.5 \qquad \frac{\text{DDHorner}}{\text{Horner}} \sim 14 \qquad \frac{\text{DDHorner}}{\text{CompHorner}} \sim 1.33$$

- Some practical ratios (average running times for degree 5 to 200):

		$\frac{\text{CompHorner}}{\text{Horner}}$	$\frac{\text{DDHorner}}{\text{Horner}}$	$\frac{\text{DDHorner}}{\text{CompHorner}}$
Pentium 4, 3.00 GHz	GCC 3.3.5	3.77	10.00	2.66
	ICC 9.1	3.06	8.88	2.92
Athlon 64, 2.00 GHz	GCC 4.0.1	3.89	10.48	2.70
Itanium 2, 1.4 GHz	GCC 3.4.6	3.64	5.50	1.51
	ICC 9.1	1.87	8.78	4.67
		$\sim 2-4$	$\sim 5-10$	$\sim 2-5$

- Why such difference between theoretical and measured running times?
- Why does CompHorner run between 2 and 5 times faster than DDHorner?

What is Instruction-Level Parallelism?

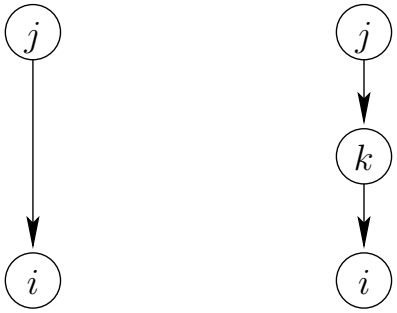
- *All processors since about 1985, including those in the embedded space, use pipelining to overlap the execution of instructions and improve performance. This potential overlap among instructions is called instruction-level parallelism (ILP) since the instruction can be evaluated in parallel.* (Hennessy & Patterson)

What is Instruction-Level Parallelism?

- *All processors since about 1985, including those in the embedded space, use pipelining to overlap the execution of instructions and improve performance. This potential overlap among instructions is called instruction-level parallelism (ILP) since the instruction can be evaluated in parallel.* (Hennessy & Patterson)
- A wide range of techniques exploit the parallelism among instructions: pipelining, superscalar architectures. . .
- More ILP thanks to less instruction dependancies
  - if two instructions are *parallel* they can execute simultaneously in a pipeline,
  - if two instructions are *dependent* they must be executed in order.
- Here data dependence is the main source of instruction dependences (no control dependencies nor name dependences).

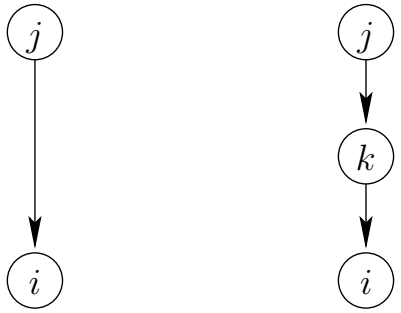
Data dependences

- An instruction i is data dependent on an instruction j if either
  - instruction j produce a result that may be used by instruction i,
  - their exist a chain of dependences of the first type between i and j.



Data dependences

- An instruction i is data dependent on an instruction j if either
  - instruction j produce a result that may be used by instruction i,
  - their exist a chain of dependences of the first type between i and j.



- If two instructions are data dependent, they cannot execute simultaneously.
- Dependences are properties of programs: the presence of a data dependence in an instruction sequence reflects a data dependence in the source code.
- What about the dependences in CompHorner and DDHorner?

Identification of the difference between CompHorner and DDHorner

```
function r = CompHorner(P, x)
  s_n = a_i ;  c_n = 0
  for i = n-1 : -1 : 0

    [p_i , π_i] = 2Prod(s_{i+1}, x)



    [s_i , σ_i] = 2Sum(p_i , a_i)
    c_i = c_{i+1} ⊗ x ⊕ (π_i ⊕ σ_i)

  end
  r = s_0 ⊕ c_0
```

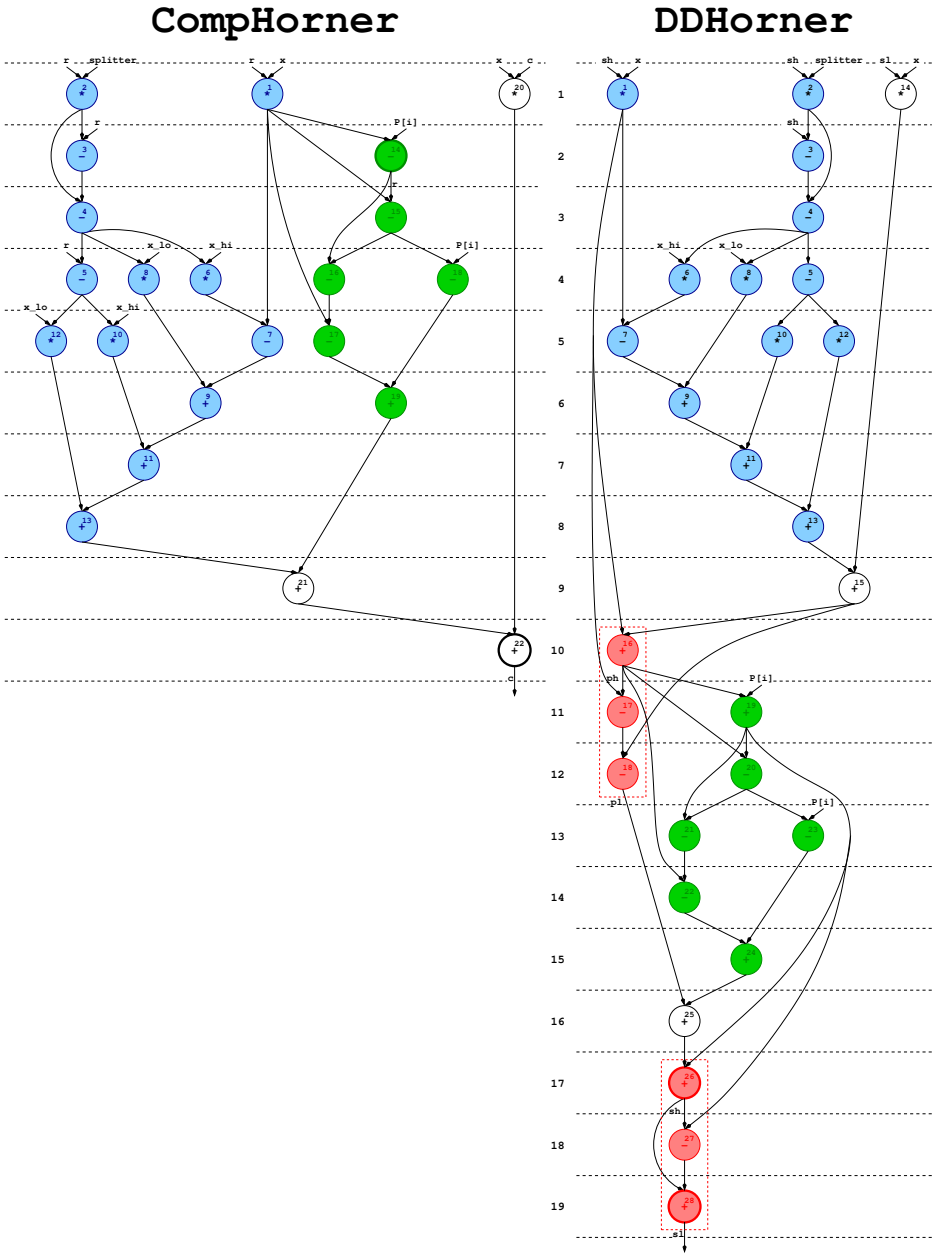
```
function r = DDHorner(P, x)
  sh_n = a_i ;  sl_n = 0
  for i = n-1 : -1 : 0
    %% [ph_i , pl_i]=[sh_{i+1} , sl_{i+1}] ⊗ x
    [th, tl] = 2Prod(sh_{i+1}, x)
    tl = sl_{i+1} ⊗ x ⊕ tl
    [ph_i , pl_i] = Fast2Sum(th, tl)

    %% [sh_i , sl_i]=[ph_i , pl_i] ⊕ a_i
    [th, tl] = 2Sum(ph_i , a_i)
    tl = tl ⊕ pl_i
    [sh_i , sl_i] = Fast2Sum(th, tl)
  end
  r = sh_0
```

In red the only differences between CompHorner and DDHorner

```
function r = CompHorner'(P, x)
  s_n = a_i ;  c_n = 0
  for i = n-1 : -1 : 0

    [p_i , π_i] = 2Prod(s_{i+1}, x)
    t_i = c_{i+1} ⊗ x ⊕ π_i



    [s_i , σ_i] = 2Sum(p_i , a_i)
    c_i = t_i ⊕ σ_i

  end
  r = s_0 ⊕ c_0
```

```
function r = DDHorner(P, x)
  sh_n = a_i ;  sl_n = 0
  for i = n-1 : -1 : 0
    %% [ph_i , pl_i]=[sh_{i+1} , sl_{i+1}] ⊗ x
    [th, tl] = 2Prod(sh_{i+1}, x)
    tl = sl_{i+1} ⊗ x ⊕ tl
    [ph_i , pl_i] = Fast2Sum(th, tl)

    %% [sh_i , sl_i]=[ph_i , pl_i] ⊕ a_i
    [th, tl] = 2Sum(ph_i , a_i)
    tl = tl ⊕ pl_i
    [sh_i , sl_i] = Fast2Sum(th, tl)
  end
  r = sh_0
```

Difference in the data flow graph



- We represent all data dependences in the inner loop of each algorithm.
- The critical path in floating point operations for CompHorner is twice shorter than in DDHorner.
- More parallelism among floating point operations in CompHorner than in DDHorner
  - TwoProd, TwoSum, next iteration can be pipelined.
  - Normalisation forbid such ILP
- Thus more potential ILP, and greater practical performance!

To finish...

Main message: Compensated algorithms are an interesting alternative to existing librairies as double or quad-double

- accurary as if computed with K times the available precision,
- efficiency in term of running-time without too much portability sacrifice since only working with IEEE-754 precisions: single, double;
- dynamic bound to control the accuracy of the compensated result.

Some work in progress:

- GPU and Sony-IBM Cell: new exotic processors are extremely powerful architectures but only in single precision!
- Highly accurate Horner schema thanks to compensation and faithful summation of Rump-Ogita-Horner.

Acknowledgements

SCAN'06 organizers and Professor W. Luther in particular
Stef Graillat and Nicolas Louvet – go to his talk this afternoon at 15:15
Professor B. Goossens and D. Parello for their expertise in micro-architecture and compiler optimisation
Rump-Ogita-Oishi recent papers for the motivation of these developments.