BeBOP meeting (ParLab, EECS, UC Berkeley)

Berkeley, CA, USA - March 30, 2010

Implementation of binary floating-point arithmetic on embedded integer processors

Polynomial evaluation-based algorithms
and
certified code generation

Guillaume Revy

ParLab EECS University of California, Berkeley

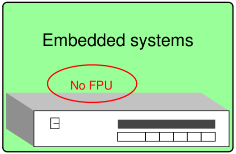
Ph.D. thesis' work done under the direction of Claude-Pierre Jeannerod and Gilles Villard
Arénaire INRIA project-team (LIP, Ens Lyon, France)

Motivation

- Embedded systems are ubiquitous
    - microprocessors dedicated to one or a few specific tasks
    - satisfy constraints: area, energy consumption, conception cost

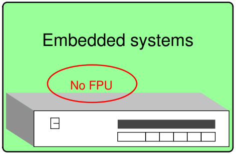
Motivation

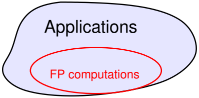
- Embedded systems are ubiquitous
  - microprocessors dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost
- Some embedded systems do not have any FPU (floating-point unit)



Motivation

- Embedded systems are ubiquitous
  - microprocessors dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost
- Some embedded systems do not have any FPU (floating-point unit)



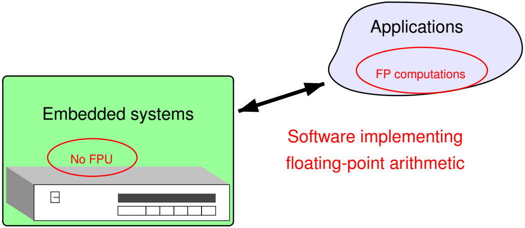


- Highly used in audio and video applications
  - demanding on floating-point computations

Motivation

- Embedded systems are ubiquitous
  - microprocessors dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost

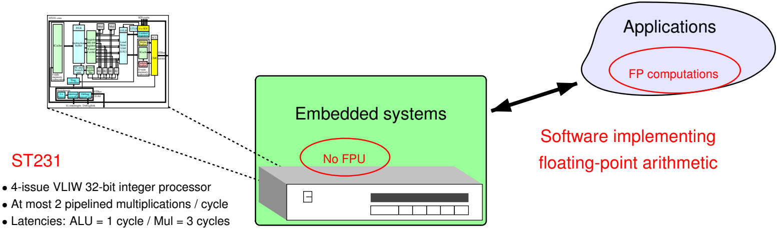
- Some embedded systems do not have any FPU (floating-point unit)



- Highly used in audio and video applications
  - demanding on floating-point computations

Motivation

- Embedded systems are ubiquitous
  - microprocessors dedicated to one or a few specific tasks
  - satisfy constraints: area, energy consumption, conception cost
- Some embedded systems do not have any FPU (floating-point unit)



- Highly used in audio and video applications
  - demanding on floating-point computations

How to emulate floating-point arithmetic in software?

Design and implementation of efficient software support for IEEE 754 floating-point arithmetic on integer processors

- Existing software for IEEE 754 floating-point arithmetic:
  - Software floating-point support of GCC, Glibc and μClibc, GoFast Floating-Point Library
  - SoftFloat ($\longrightarrow$ STlib)
  - FLIP (Floating-point Library for Integer Processors)
    - software support for *binary32* floating-point arithmetic on integer processors
    - correctly-rounded addition, subtraction, multiplication, division, square root, reciprocal, ...
    - handling subnormals, and handling special inputs

Towards the generation of fast and certified codes

- Underlying problem: development “by hand”
  - long and tedious, error prone
  - new target? new floating-point format?

Towards the generation of fast and certified codes

- Underlying problem: development "by hand"
  - long and tedious, error prone
  - new target? new floating-point format?

    $\Rightarrow$ need for automation and certification

Towards the generation of fast and certified codes

- Underlying problem: development “by hand”
  - long and tedious, error prone
  - new target? new floating-point format?

  $\Rightarrow$ need for automation and certification

- Current challenge: tools and methodologies for the automatic generation of efficient and certified programs
  - optimized for a given format, for the target architecture

Towards the generation of fast and certified codes

- Arénaire's developments: hardware (FloPoCo) and software (Sollya, Metalibm)

- Spiral project: hardware and software code generation for DSP algorithms

  Can we teach computers to write fast libraries?

Towards the generation of fast and certified codes

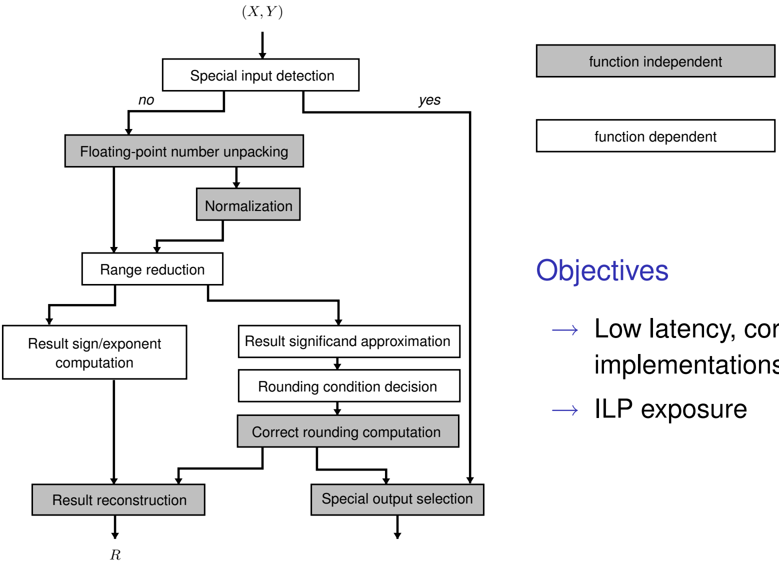
- Arénaire's developments: hardware (FloPoCo) and software (Sollya, Metalibm)
- Spiral project: hardware and software code generation for DSP algorithms

  Can we teach computers to write fast libraries?

- Our tool: CGPE (Code Generation for Polynomial Evaluation)

  *In the particular case of **polynomial evaluation**, we can teach computers to write **fast and certified** codes, for a given target and optimized for a given format.*

Basic blocks for implementing correctly-rounded operators

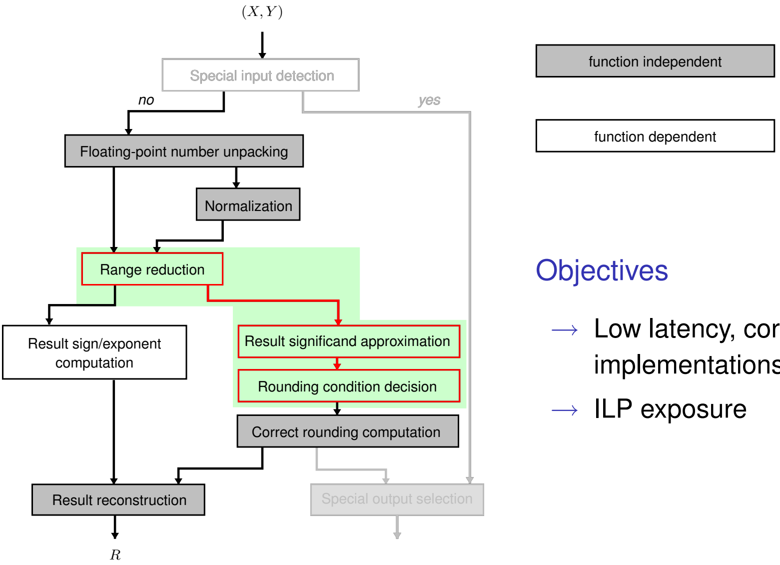


Objectives

→ Low latency, correctly-rounded implementations

→ ILP exposure

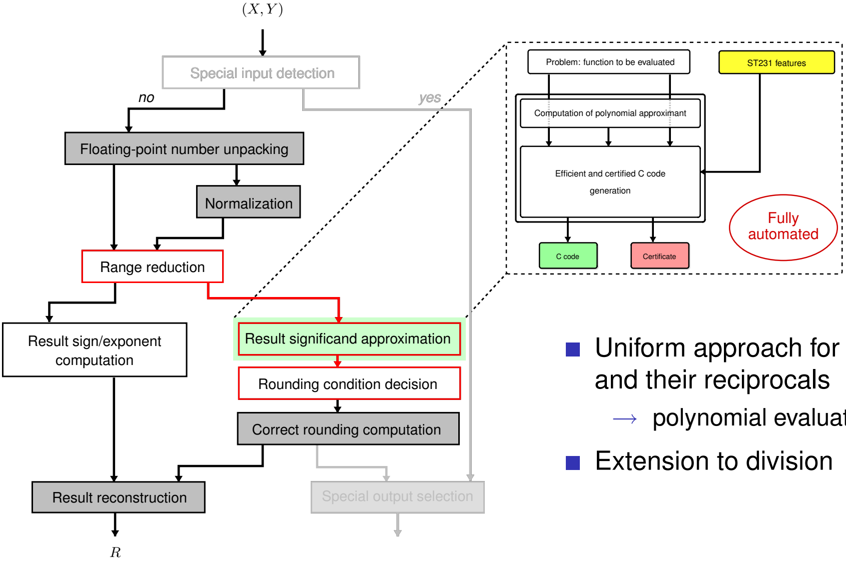
Basic blocks for implementing correctly-rounded operators



Objectives

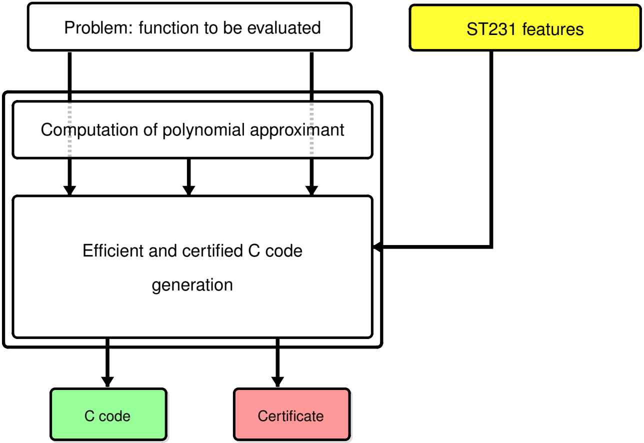
- $\longrightarrow$ Low latency, correctly-rounded implementations
- $\longrightarrow$ ILP exposure

Basic blocks for implementing correctly-rounded operators

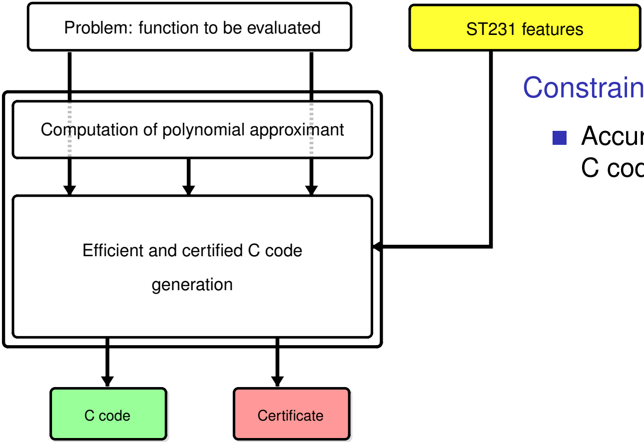


- Uniform approach for *n*th roots and their reciprocals
  - $\longrightarrow$ polynomial evaluation
- Extension to division

Flowchart for generating efficient and certified C codes



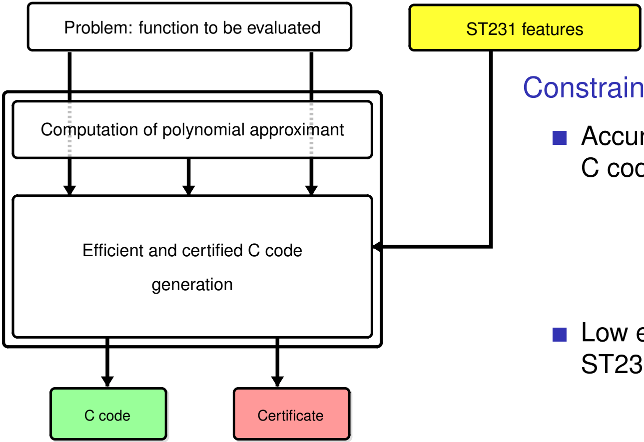
Flowchart for generating efficient and certified C codes



Constraints

- Accuracy of approximant and C code

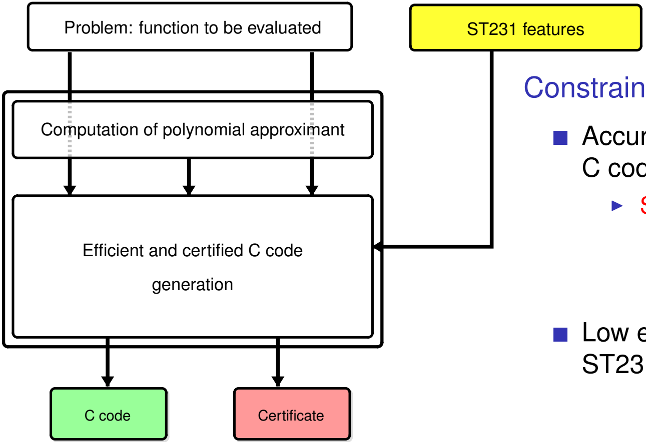
Flowchart for generating efficient and certified C codes



Constraints

- Accuracy of approximant and C code
- Low evaluation latency on ST231, ILP exposure

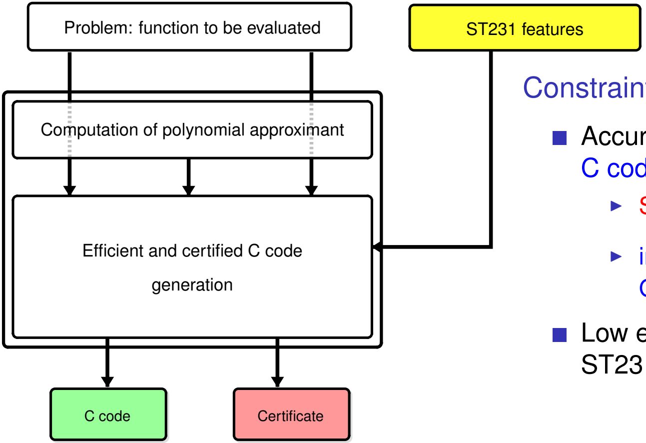
Flowchart for generating efficient and certified C codes



Constraints

- Accuracy of approximant and C code
  - Sollya
- Low evaluation latency on ST231, ILP exposure

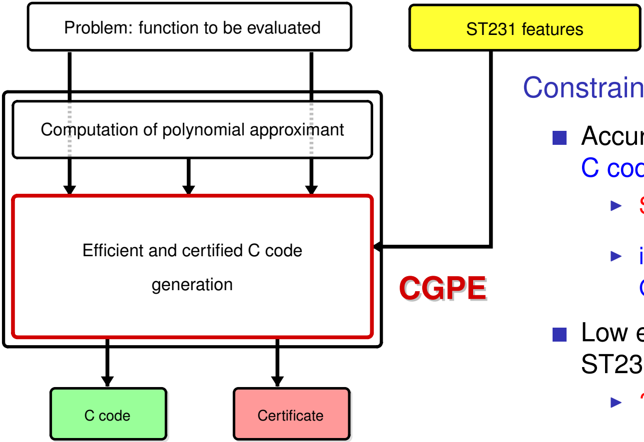
Flowchart for generating efficient and certified C codes



Constraints

- Accuracy of approximant and C code
  - Sollya
  - interval arithmetic (MPFI), Gappa
- Low evaluation latency on ST231, ILP exposure

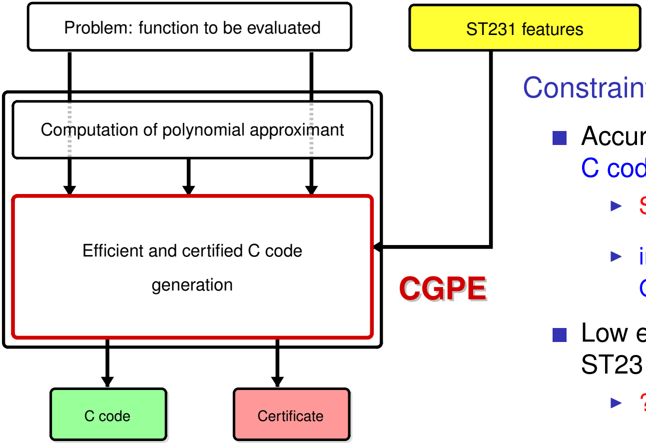
Flowchart for generating efficient and certified C codes



Constraints

- Accuracy of approximant and C code
  - Sollya
  - interval arithmetic (MPFI), Gappa
- Low evaluation latency on ST231, ILP exposure
  - ?

Flowchart for generating efficient and certified C codes



Constraints

- Accuracy of approximant and C code
  - Sollya
  - interval arithmetic (MPFI), Gappa
- Low evaluation latency on ST231, ILP exposure
  - ?
- Efficiency of the generation process

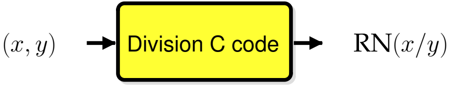
Outline of the talk

1. Design and implementation of floating-point operators
   - Bivariate polynomial evaluation-based approach
   - Implementation of correct rounding
2. Low latency parenthesization computation
   - Classical evaluation methods
   - Computation of all parenthesizations
   - Towards low evaluation latency
3. Selection of effective evaluation parenthesizations
   - General framework
   - Automatic certification of generated C codes
4. Numerical results
5. Conclusions
6. And now in ParLab: Debugging of floating-point programs

Outline of the talk

1. Design and implementation of floating-point operators
   - Bivariate polynomial evaluation-based approach
   - Implementation of correct rounding
2. Low latency parenthesization computation
3. Selection of effective evaluation parenthesizations
4. Numerical results
5. Conclusions
6. And now in ParLab: Debugging of floating-point programs

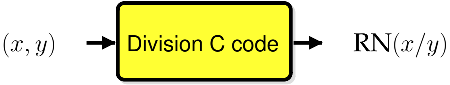
Notation and assumptions



- Input (x, y) and output $\mathrm{RN}(x/y)$: normal numbers
  - $\longrightarrow$ no underflow nor overflow
  - $\longrightarrow$ precision p, extremal exponents $e_{\min}$, $e_{\max}$

$$x = \pm 1.m_{x,1} \ldots m_{x,p-1} \cdot 2^{e_x} \quad \text{with} \quad e_x \in \{e_{\min}, \ldots, e_{\max}\}$$

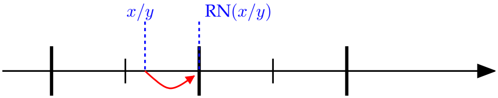
Notation and assumptions



- Input (x, y) and output $\mathrm{RN}(x/y)$: normal numbers
  - $\longrightarrow$ no underflow nor overflow
  - $\longrightarrow$ precision p, extremal exponents $e_{\min}$, $e_{\max}$

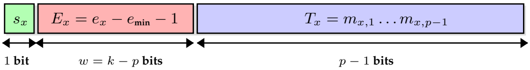
$$x = \pm 1.m_{x,1} \ldots m_{x,p-1} \cdot 2^{e_x} \quad \text{with} \quad e_x \in \{e_{\min}, \ldots, e_{\max}\}$$

  - $\longrightarrow$ RoundTiesToEven



Notation and assumptions

- Standard binary encoding: k-bit unsigned integer X encodes input x

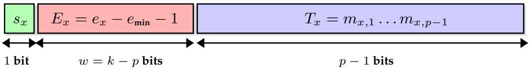


- Computation: k-bit unsigned integers

  $\longrightarrow$ integer and fixed-point arithmetic

Notation and assumptions

- Standard binary encoding: *k*-bit unsigned integer *X* encodes input *x*



- Computation: *k*-bit unsigned integers

  $\longrightarrow$ integer and fixed-point arithmetic

Range reduction of division

- Express the exact result $r = x/y$ as:

$$r = \ell \cdot 2^d \quad \Rightarrow \quad \mathrm{RN}(x/y) = \mathrm{RN}(\ell) \cdot 2^d$$

with

$$\ell \in [1,2) \quad \text{and} \quad d \in \{e_{\min}, \ldots, e_{\max}\}$$

Range reduction of division

- Express the exact result $r = x/y$ as:

$$r = \ell \cdot 2^d \quad \Rightarrow \quad \mathrm{RN}(x/y) = \mathrm{RN}(\ell) \cdot 2^d$$

with

$$\ell \in [1,2) \quad \text{and} \quad d \in \{e_{\min}, \ldots, e_{\max}\}$$

- Definition

$$c = 1 \quad \text{if} \quad m_x \geq m_y, \quad \text{and} \quad c = 0 \quad \text{otherwise}$$

Range reduction of division

- Express the exact result $r = x/y$ as:

$$r = \ell \cdot 2^d \quad \Rightarrow \quad \mathrm{RN}(x/y) = \mathrm{RN}(\ell) \cdot 2^d$$

with

$$\ell \in [1,2) \quad \text{and} \quad d \in \{e_{\min}, \ldots, e_{\max}\}$$

- Definition

$$c = 1 \quad \text{if} \quad m_x \geq m_y, \quad \text{and} \quad c = 0 \quad \text{otherwise}$$

- Range reduction

$$x/y = \underbrace{\left(2^{1-c} \cdot m_x/m_y\right)}_{:= \ell \in [1,2)} \cdot 2^d \quad \text{with} \quad d = e_x - e_y - 1 + c$$

Range reduction of division

- Express the exact result $r = x/y$ as:

$$r = \ell \cdot 2^d \quad \Rightarrow \quad \mathrm{RN}(x/y) = \mathrm{RN}(\ell) \cdot 2^d$$

with

$$\ell \in [1,2) \quad \text{and} \quad d \in \{e_{\min}, \ldots, e_{\max}\}$$

- Definition

$$c = 1 \quad \text{if} \quad m_x \geq m_y, \quad \text{and} \quad c = 0 \quad \text{otherwise}$$

- Range reduction

$$x/y = \underbrace{\left(2^{1-c} \cdot m_x/m_y\right)}_{:= \ell \in [1,2)} \cdot 2^d \quad \text{with} \quad d = e_x - e_y - 1 + c$$

How to compute the correctly-rounded significand $\mathrm{RN}(\ell)$?

Methods for computing the correctly-rounded significand

- Iterative methods: restoring, non-restoring, SRT, ...
  - Oberman and Flynn (1997)
  - minimal ILP exposure, sequential algorithm

Methods for computing the correctly-rounded significand

- Iterative methods: restoring, non-restoring, SRT, ...
  - Oberman and Flynn (1997)
  - minimal ILP exposure, sequential algorithm
- Multiplicative methods: Newton-Raphson, Goldschmidt
  - Piñeiro and Bruguera (2002) – Raina's Ph.D., FLIP 0.3 (2006)
  - exploit available multipliers, more ILP exposure

Methods for computing the correctly-rounded significand

- Iterative methods: restoring, non-restoring, SRT, ...
  - Oberman and Flynn (1997)
  - minimal ILP exposure, sequential algorithm
- Multiplicative methods: Newton-Raphson, Goldschmidt
  - Piñeiro and Bruguera (2002) – Raina's Ph.D., FLIP 0.3 (2006)
  - exploit available multipliers, more ILP exposure
- Polynomial-based methods
  - Agarwal, Gustavson and Schmookler (1999)
    $\longrightarrow$ univariate polynomial evaluation
  - Our approach
    $\longrightarrow$ bivariate polynomial evaluation: maximal ILP exposure

Correct rounding via truncated one-sided approximation

- How to compute $\mathrm{RN}(\ell)$, with $\ell = 2^{1-c} \cdot m_x / m_y$?

- Three steps for correct rounding computation

  1. compute $v = 1.v_1 \ldots v_{k-2}$ such that $-2^{-p} \leq \ell - v < 0$
     - $\longrightarrow$ implied by $|(\ell + 2^{-p-1}) - v| < 2^{-p-1}$
     - $\longrightarrow$ bivariate polynomial evaluation
  2. compute u as the truncation of v after p fraction bits
  3. determine $\mathrm{RN}(\ell)$ after possibly adding 2^{-p}

Correct rounding via truncated one-sided approximation

- How to compute $\mathrm{RN}(\ell)$, with $\ell = 2^{1-c} \cdot m_x / m_y$?

- Three steps for correct rounding computation

  1. compute $v = 1.v_1 \dots v_{k-2}$ such that $-2^{-p} \le \ell - v < 0$
     - $\longrightarrow$ implied by $|(\ell + 2^{-p-1}) - v| < 2^{-p-1}$
     - $\longrightarrow$ bivariate polynomial evaluation
  2. compute u as the truncation of v after p fraction bits
  3. determine $\mathrm{RN}(\ell)$ after possibly adding 2^{-p}

How to compute the one-sided approximation v and then deduce $\mathrm{RN}(\ell)$?

One-sided approximation via bivariate polynomials

1. Consider $\ell + 2^{-p-1}$ as the exact result of the function

$$F(s,t) = s/(1+t) + 2^{-p-1}$$

at the points $s^* = 2^{1-c} \cdot m_x$ and $t^* = m_y - 1$

One-sided approximation via bivariate polynomials

1. Consider $\ell + 2^{-p-1}$ as the exact result of the function

$$F(s,t) = s/(1+t) + 2^{-p-1}$$

at the points $s^* = 2^{1-c} \cdot m_x$ and $t^* = m_y - 1$

2. Approximate $F(s,t)$ by a bivariate polynomial $P(s,t)$

$$P(s,t) = s \cdot a(t) + 2^{-p-1}$$

$\longrightarrow$ $a(t)$: univariate polynomial approximant of $1/(1+t)$

$\longrightarrow$ approximation error E_{approx}

One-sided approximation via bivariate polynomials

1. Consider $\ell + 2^{-p-1}$ as the exact result of the function

$$F(s,t) = s/(1+t) + 2^{-p-1}$$

   at the points $s^* = 2^{1-c} \cdot m_x$ and $t^* = m_y - 1$

2. Approximate $F(s,t)$ by a bivariate polynomial $P(s,t)$

$$P(s,t) = s \cdot a(t) + 2^{-p-1}$$

   - $\rightarrow$ $a(t)$: univariate polynomial approximant of $1/(1+t)$
   - $\rightarrow$ approximation error E_{approx}

3. Evaluate $P(s,t)$ by a well-chosen efficient evaluation program $\mathcal{P}$

$$v = \mathcal{P}(s^*, t^*)$$

   - $\rightarrow$ evaluation error E_{eval}

One-sided approximation via bivariate polynomials

1. Consider $\ell + 2^{-p-1}$ as the exact result of the function

$$F(s,t) = s/(1+t) + 2^{-p-1}$$

   at the points $s^* = 2^{1-c} \cdot m_x$ and $t^* = m_y - 1$

2. Approximate $F(s,t)$ by a bivariate polynomial $P(s,t)$

$$P(s,t) = s \cdot a(t) + 2^{-p-1}$$

   $\longrightarrow$ $a(t)$: univariate polynomial approximant of $1/(1+t)$

   $\longrightarrow$ approximation error E_{approx}

3. Evaluate $P(s,t)$ by a well-chosen efficient evaluation program $\mathcal{P}$

$$v = \mathcal{P}(s^*, t^*)$$

   $\longrightarrow$ evaluation error E_{eval}

How to ensure that $|(\ell + 2^{-p-1}) - v| < 2^{-p-1}$?

Sufficient error bounds

- To ensure $\quad |(\ell + 2^{-p-1}) - v| < 2^{-p-1}$

  it suffices to ensure that $\quad \mu \cdot E_{\text{approx}} + E_{\text{eval}} < 2^{-p-1}$,

  since

$$|(\ell + 2^{-p-1}) - v| \leq \mu \cdot E_{\text{approx}} + E_{\text{eval}} \qquad \text{with} \qquad \mu = 4 - 2^{3-p}$$

Sufficient error bounds

- To ensure $\quad |(\ell + 2^{-p-1}) - v| < 2^{-p-1}$

  it suffices to ensure that $\quad \mu \cdot E_{\text{approx}} + E_{\text{eval}} < 2^{-p-1}$,

  since

  $$|(\ell + 2^{-p-1}) - v| \leq \mu \cdot E_{\text{approx}} + E_{\text{eval}} \qquad \text{with} \qquad \mu = 4 - 2^{3-p}$$

- This gives the following sufficient conditions

$$E_{\text{approx}} < 2^{-p-1}/\mu \qquad \Rightarrow \qquad E_{\text{eval}} < 2^{-p-1} - \mu \cdot E_{\text{approx}}$$

Sufficient error bounds

- To ensure $\quad |(\ell + 2^{-p-1}) - v| < 2^{-p-1}$

  it suffices to ensure that $\quad \mu \cdot E_{\text{approx}} + E_{\text{eval}} < 2^{-p-1}$,

  since

$$|(\ell + 2^{-p-1}) - v| \leq \mu \cdot E_{\text{approx}} + E_{\text{eval}} \qquad \text{with} \qquad \mu = 4 - 2^{3-p}$$

- This gives the following sufficient conditions

$$E_{\text{approx}} \leq \theta \quad \text{with} \quad \theta < 2^{-p-1}/\mu \qquad \Rightarrow \qquad E_{\text{eval}} < \eta = 2^{-p-1} - \mu \cdot \theta$$

Example for the *binary32* division

- Sufficient conditions with $\mu = 4 - 2^{-21}$

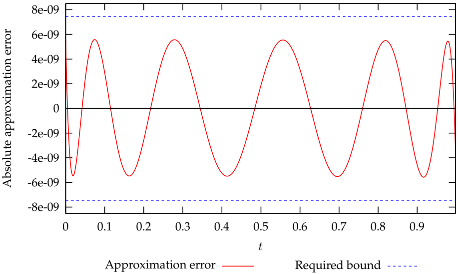
$$E_{\text{approx}} \leq \theta \quad \text{with} \quad \theta < 2^{-25}/\mu \qquad \text{and} \qquad E_{\text{eval}} < \eta = 2^{-25} - \mu \cdot \theta$$

Example for the *binary32* division

- Sufficient conditions with $\mu = 4 - 2^{-21}$

$$E_{\text{approx}} \leq \theta \quad \text{with} \quad \theta < 2^{-25}/\mu \qquad \text{and} \qquad E_{\text{eval}} < \eta = 2^{-25} - \mu \cdot \theta$$

- Approximation of $1/(1+t)$ by a Remez-like polynomial of degree 10



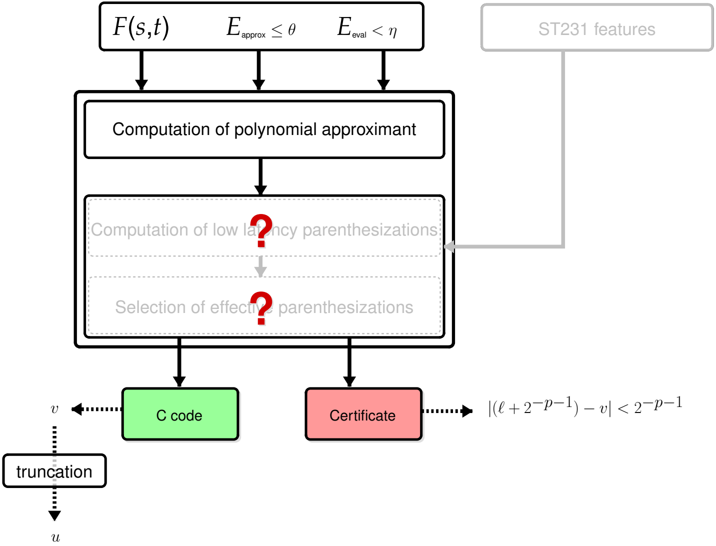
- $E_{\text{approx}} \leq \theta$,

  with $\theta = 3 \cdot 2^{-29} \approx 6 \cdot 10^{-9}$

- $E_{\text{eval}} < \eta$,

  with $\eta \approx 7.4 \cdot 10^{-9}$

Flowchart for generating efficient and certified C codes

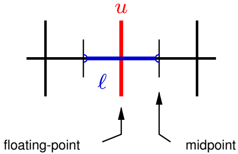


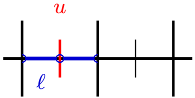
Rounding condition: definition

- Approximation u of ℓ with

$$\ell = 2^{1-c} \cdot m_x / m_y$$

- The exact value ℓ may have an infinite number of bits
  - $\longrightarrow$ the sticky bit cannot always be computed



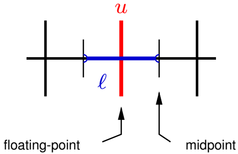


Rounding condition: definition

- Approximation u of ℓ with

$$\ell = 2^{1-c} \cdot m_x / m_y$$

- The exact value ℓ may have an infinite number of bits
  - $\longrightarrow$ the sticky bit cannot always be computed



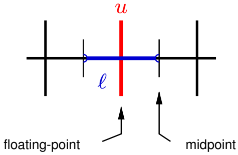
- Compute RN(ℓ) requires to be able to decide whether $u \geq \ell$
  - $\longrightarrow$ ℓ cannot be a midpoint

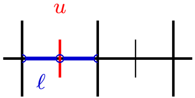
Rounding condition: definition

- Approximation u of ℓ with

$$\ell = 2^{1-c} \cdot m_x / m_y$$

- The exact value ℓ may have an infinite number of bits
  - $\longrightarrow$ the sticky bit cannot always be computed





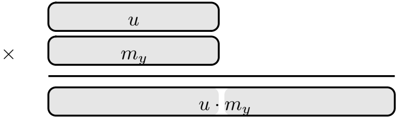
- Compute RN(ℓ) requires to be able to decide whether $u \geq \ell$
  - $\longrightarrow$ ℓ cannot be a midpoint
- Rounding condition: $u \geq \ell$

$$u \geq \ell \quad \Longleftrightarrow \quad u \cdot m_y \geq 2^{1-c} \cdot m_x$$

Rounding condition: implementation in integer arithmetic

- Rounding condition: $u \cdot m_y \geq 2^{1-c} \cdot m_x$

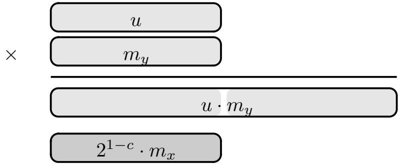
- Approximation u and m_y: representable with 32 bits



  - $u \cdot m_y$ is exactly representable with 64 bits

Rounding condition: implementation in integer arithmetic

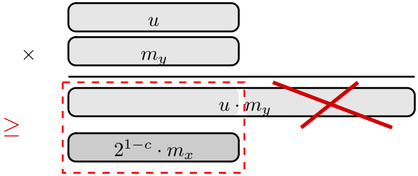
- Rounding condition: $u \cdot m_y \geq 2^{1-c} \cdot m_x$
- Approximation u and m_y: representable with 32 bits



  - $u \cdot m_y$ is exactly representable with 64 bits
  - $2^{1-c} \cdot m_x$ is representable with 32 bits since $c \in \{0, 1\}$

Rounding condition: implementation in integer arithmetic

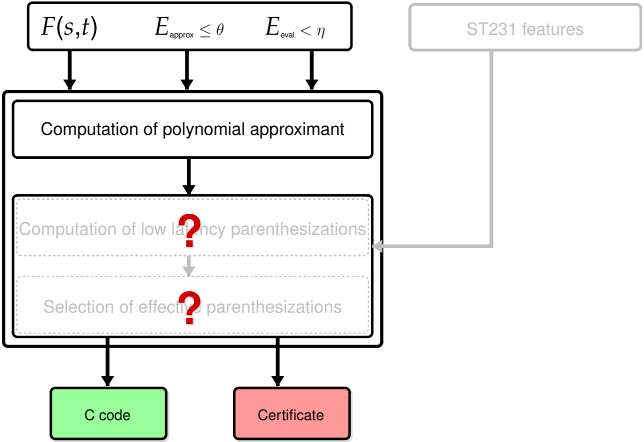
- Rounding condition: $u \cdot m_y \geq 2^{1-c} \cdot m_x$
- Approximation u and m_y: representable with 32 bits



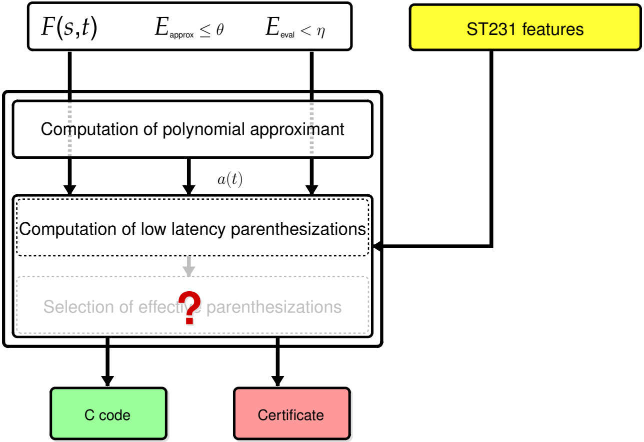
  - $u \cdot m_y$ is exactly representable with 64 bits
  - $2^{1-c} \cdot m_x$ is representable with 32 bits since $c \in \{0, 1\}$

$\Rightarrow$ one $32 \times 32 \rightarrow 32$-bit multiplication and one comparison

Flowchart for generating efficient and certified C codes



Flowchart for generating efficient and certified C codes



Outline of the talk

1. Design and implementation of floating-point operators

2. Low latency parenthesization computation
   - Classical evaluation methods
   - Computation of all parenthesizations
   - Towards low evaluation latency

3. Selection of effective evaluation parenthesizations

4. Numerical results

5. Conclusions

6. And now in ParLab: Debugging of floating-point programs

Objectives

- Compute an efficient parenthesization for evaluating $P(s, t)$

  $\longrightarrow$ reduces the evaluation latency on unbounded parallelism

Objectives

- Compute an efficient parenthesization for evaluating $P(s,t)$

  $\longrightarrow$ reduces the evaluation latency on unbounded parallelism

- Evaluation program $\mathcal{P}$ = main part of the full software implementation

  $\longrightarrow$ dominates the cost

Objectives

- Compute an efficient parenthesization for evaluating $P(s,t)$

  $\longrightarrow$ reduces the evaluation latency on unbounded parallelism

- Evaluation program $\mathcal{P}$ = main part of the full software implementation

  $\longrightarrow$ dominates the cost

- Two families of algorithms
  - algorithms with coefficient adaptation: Knuth and Eve (60's), Paterson and Stockmeyer (1964), ...

    $\longrightarrow$ ill-suited in the context of fixed-point arithmetic

  - algorithms without coefficient adaptation

Objectives

- Compute an efficient parenthesization for evaluating $P(s,t)$

  $\longrightarrow$ reduces the evaluation latency on unbounded parallelism

- Evaluation program $\mathcal{P}$ = main part of the full software implementation

  $\longrightarrow$ dominates the cost

- Two families of algorithms
  - algorithms with coefficient adaptation: Knuth and Eve (60's), Paterson and Stockmeyer (1964), ...

    $\longrightarrow$ ill-suited in the context of fixed-point arithmetic

  - algorithms without coefficient adaptation

Classical parenthesizations for *binary32* division

$$P(s,t) = 2^{-25} + s \cdot \sum_{0 \le i \le 10} a_i \cdot t^i$$

- Horner's rule: $(3+1) \times 11 = 44$ cycles
  - $\longrightarrow$ no ILP exposure

Classical parenthesizations for *binary32* division

$$P(s,t) = 2^{-25} + s \cdot \sum_{0 \le i \le 10} a_i \cdot t^i$$

- Horner's rule: $(3+1) \times 11 = 44$ cycles
  - $\longrightarrow$ no ILP exposure
- Second-order Horner's rule: 27 cycles
  - $\longrightarrow$ evaluation of odd and even parts independently with Horner, more ILP

Classical parenthesizations for *binary32* division

$$P(s,t) = 2^{-25} + s \cdot \sum_{0 \le i \le 10} a_i \cdot t^i$$

- Horner's rule: $(3+1) \times 11 = 44$ cycles
  - $\longrightarrow$ no ILP exposure
- Second-order Horner's rule: 27 cycles
  - $\longrightarrow$ evaluation of odd and even parts independently with Horner, more ILP
- Estrin's method: 19 cycles
  - $\longrightarrow$ evaluation of high and low parts in parallel, even more ILP
  - $\longrightarrow$ distributing the multiplication by s in the evaluation of $a(t) \rightarrow$ 16 cycles

Classical parenthesizations for *binary32* division

$$P(s,t) = 2^{-25} + s \cdot \sum_{0 \le i \le 10} a_i \cdot t^i$$

- Horner's rule: $(3+1) \times 11 = 44$ cycles
  - $\longrightarrow$ no ILP exposure
- Second-order Horner's rule: 27 cycles
  - $\longrightarrow$ evaluation of odd and even parts independently with Horner, more ILP
- Estrin's method: 19 cycles
  - $\longrightarrow$ evaluation of high and low parts in parallel, even more ILP
  - $\longrightarrow$ distributing the multiplication by s in the evaluation of $a(t) \rightarrow$ 16 cycles
- ...

We can do better.

How to explore the solution space of parenthesizations?

Algorithm for computing all parenthesizations

$$a(x,y) = \sum_{0 \le i \le n_x} \sum_{0 \le j \le n_y} a_{i,j} \cdot x^i \cdot y^j \qquad \text{with} \quad n = n_x + n_y, \quad \text{and} \quad a_{n_x,n_y} \neq 0$$

Example

Let $a(x,y) = a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y$. Then

$a_{1,0} + a_{1,1} \cdot y$ is a valid expression, while $a_{1,0} \cdot x + a_{1,1} \cdot x$ is not.

Algorithm for computing all parenthesizations

$$a(x,y) = \sum_{0\le i\le n_x} \sum_{0\le j\le n_y} a_{i,j} \cdot x^i \cdot y^j \qquad \text{with} \quad n = n_x + n_y, \quad \text{and} \quad a_{n_x,n_y} \neq 0$$

Example

Let $a(x,y) = a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y$. Then

$a_{1,0} + a_{1,1} \cdot y$ is a valid expression, while $a_{1,0} \cdot x + a_{1,1} \cdot x$ is not.

- Exhaustive algorithm: iterative process
  - $\longrightarrow$ step k = computation of all the valid expressions of total degree k
- 3 building rules for computing all parenthesizations

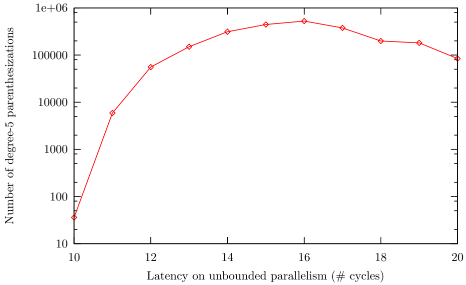
Number of parentheszations

	$n_x = 1$	$n_x = 2$	$n_x = 3$	$n_x = 4$	$n_x = 5$	$n_x = 6$
$n_y = 0$	1	7	163	11602	2334244	1304066578
$n_y = 1$	51	67467	1133220387	207905478247998	$\cdots$	$\cdots$
$n_y = 2$	67467	106191222651	10139277122276921118	$\cdots$	$\cdots$	$\cdots$

Number of generated parenthesizations for evaluating a bivariate polynomial

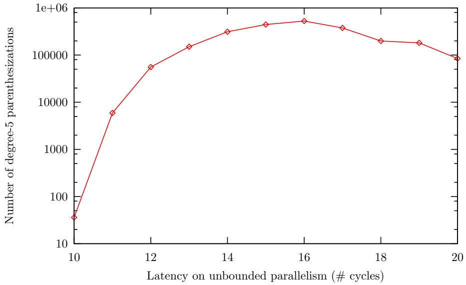
- Timings for parenthesization computation
  - $\longrightarrow$ for univariate polynomial of degree 5 $\approx$ 1h on a 2.4 GHz core
  - $\longrightarrow$ for bivariate polynomial of degree (2,1) $\approx$ 30s
  - $\longrightarrow$ for $P(s,t)$ of degree (3,1) $\approx$ 7s (88384 schemes)
- Optimization for univariate polynomial and $P(s,t)$
  - $\longrightarrow$ univariate polynomial of degree 5 $\approx$ 4min
  - $\longrightarrow$ for $P(s,t)$ of degree (3,1) $\approx$ 2s (88384 schemes)

Number of parenthesizations



→ minimal latency for univariate polynomial of degree 5: 10 cycles (36 schemes)

Number of parenthesizations



→ minimal latency for univariate polynomial of degree 5: 10 cycles (36 schemes)

How to compute only parenthesizations of low latency?

Determination of a *target* latency

- Target latency = minimal cost for evaluating

$$a_{0,0} + a_{n_x,n_y} \cdot x^{n_x} y^{n_y}$$

  - if no scheme satisfies τ then increase τ and restart

Determination of a *target* latency

- Target latency = minimal cost for evaluating

$$a_{0,0} + a_{n_x,n_y} \cdot x^{n_x} y^{n_y}$$

  - if no scheme satisfies τ then increase τ and restart

- Static target latency τ_{static}
  - as general as evaluating $a_{0,0} + x^{n_x+n_y+1}$

$$\tau_{\text{static}} = A + M \times \lceil \log_2(n_x + n_y + 1) \rceil$$

Determination of a *target* latency

- Target latency = minimal cost for evaluating

$$a_{0,0} + a_{n_x,n_y} \cdot x^{n_x} y^{n_y}$$

  - if no scheme satisfies τ then increase τ and restart

- Static target latency τ_{static}
  - as general as evaluating $a_{0,0} + x^{n_x+n_y+1}$

$$\tau_{\text{static}} = A + M \times \lceil \log_2(n_x + n_y + 1) \rceil$$

- Dynamic target latency τ_{dynamic}
  - cost of operator on a_{n_x,n_y} and delay on intederminates
  - dynamic programming

Optimized search of *best* parenthesizations

Example

Let $a(x,y)$ be a degree-2 bivariate polynomial

$$a(x,y) = a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y.$$

$\Rightarrow$ find a best splitting of the polynomial $\longrightarrow$ low latency

Optimized search of *best* parenthesizations

Example

Let $a(x,y)$ be a degree-2 bivariate polynomial

$$a(x,y) = a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y.$$

$\Rightarrow$ find a best splitting of the polynomial $\longrightarrow$ low latency

$$\Big(a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y\Big) + \Big(a_{1,1} \cdot x \cdot y\Big)$$

Optimized search of *best* parenthesizations

Example

Let $a(x, y)$ be a degree-2 bivariate polynomial

$$a(x,y) = a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y.$$

$\Rightarrow$ find a best splitting of the polynomial $\longrightarrow$ low latency

$$\Big(\big(a_{0,0} + a_{1,0} \cdot x \big) + a_{0,1} \cdot y \Big) + \Big(a_{1,1} \cdot x \cdot y \Big)$$

Optimized search of *best* parenthesizations

Example

Let $a(x, y)$ be a degree-2 bivariate polynomial

$$a(x,y) = a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y.$$

$\Rightarrow$ find a best splitting of the polynomial $\longrightarrow$ low latency

$$\Big(a_{0,0} + \big(a_{1,0} \cdot x + a_{0,1} \cdot y\big)\Big) + \Big(a_{1,1} \cdot x \cdot y\Big)$$

Optimized search of *best* parenthesizations

Example

Let $a(x, y)$ be a degree-2 bivariate polynomial

$$a(x,y) = a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y.$$

$\Rightarrow$ find a best splitting of the polynomial $\longrightarrow$ low latency

$$\Big(a_{0,0} + a_{1,0} \cdot x\Big) + \Big(a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y\Big)$$

Optimized search of *best* parenthesizations

Example

Let $a(x, y)$ be a degree-2 bivariate polynomial

$$a(x, y) = a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y.$$

$\Rightarrow$ find a best splitting of the polynomial $\longrightarrow$ low latency

$$a_{0,0} + \left(a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y\right)$$

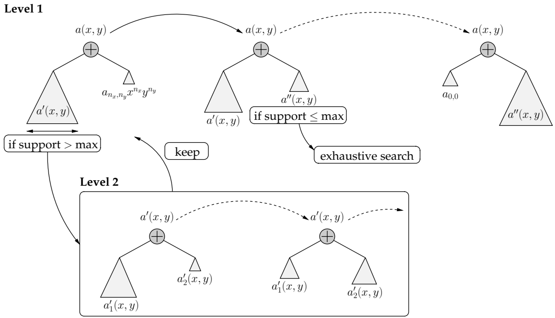
Optimized search of *best* parenthesizations

Example

Let $a(x, y)$ be a degree-2 bivariate polynomial

$$a(x, y) = a_{0,0} + a_{1,0} \cdot x + a_{0,1} \cdot y + a_{1,1} \cdot x \cdot y.$$

$\Rightarrow$ find a best splitting of the polynomial $\longrightarrow$ low latency



Efficient evaluation parenthesization generation

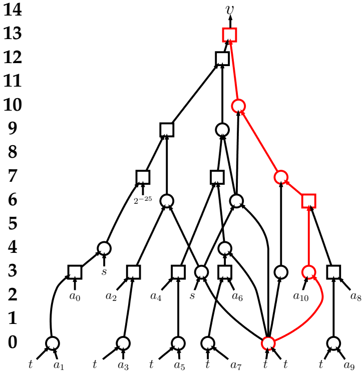
$$P(s,t) = 2^{-25} + s \cdot \sum_{0 \le i \le 10} a_i \cdot t^i$$

- First target latency $\tau = 13$
  - $\longrightarrow$ no parenthesization found

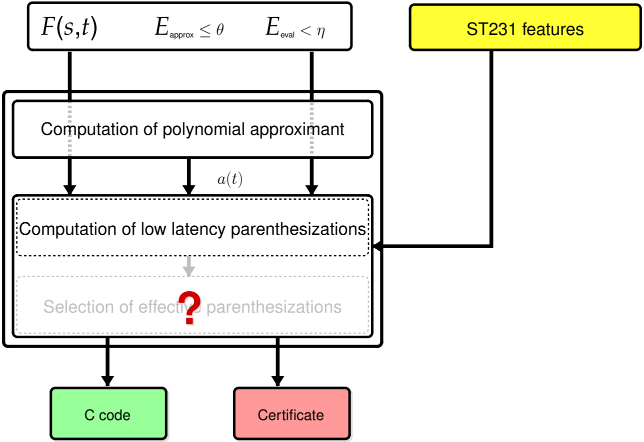
Efficient evaluation parenthesization generation

$$P(s,t) = 2^{-25} + s \cdot \sum_{0 \le i \le 10} a_i \cdot t^i$$

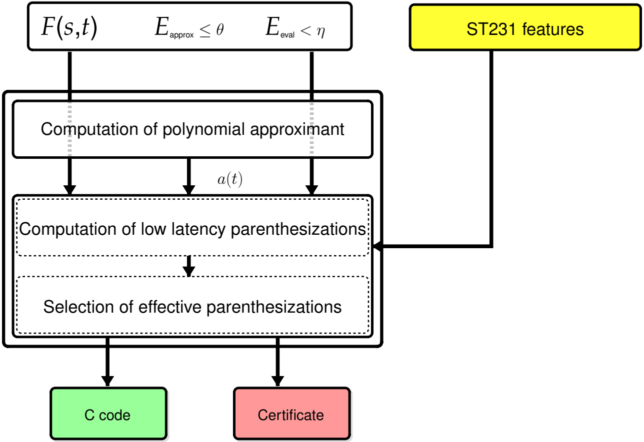
- First target latency $\tau = 13$
  - $\longrightarrow$ no parenthesization found
- Second target latency $\tau = 14$
  - $\longrightarrow$ obtained in about 10 sec.
- Classical methods
  - Horner: 44 cycles,
  - Estrin: 19 cycles,
  - Estrin by distributing s: 16 cycles



Flowchart for generating efficient and certified C codes



Flowchart for generating efficient and certified C codes



Outline of the talk

1. Design and implementation of floating-point operators

2. Low latency parenthesization computation

3. Selection of effective evaluation parenthesizations
   - General framework
   - Automatic certification of generated C codes

4. Numerical results

5. Conclusions

6. And now in ParLab: Debugging of floating-point programs

Selection of effective parenthesizations

1. Arithmetic Operator Choice
   - all intermediate variables are of constant sign

Selection of effective parenthesizations

1. Arithmetic Operator Choice
   - all intermediate variables are of constant sign
2. Scheduling on a simplified model of the ST231
   - constraints of architecture: cost of operators, instructions bundling, ...
   - delays on indeterminates

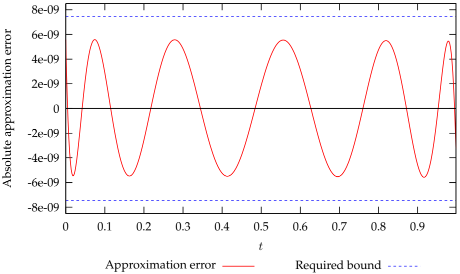
Selection of effective parenthesizations

1. Arithmetic Operator Choice
   - all intermediate variables are of constant sign
2. Scheduling on a simplified model of the ST231
   - constraints of architecture: cost of operators, instructions bundling, ...
   - delays on indeterminates
3. Certification of generated C code
   - straightline polynomial evaluation program
   - "certified C code": we can bound the evaluation error in integer arithmetic

Certification of evaluation error for *binary32* division

- Sufficient conditions with $\mu = 4 - 2^{-21}$

$$E_{\text{approx}} \leq \theta \quad \text{with} \quad \theta < 2^{-25}/\mu \qquad \text{and} \qquad E_{\text{eval}} < \eta = 2^{-25} - \mu \cdot \theta$$



- $E_{\text{approx}} \leq \theta$,

  with $\quad \theta = 3 \cdot 2^{-29} \approx 6 \cdot 10^{-9}$

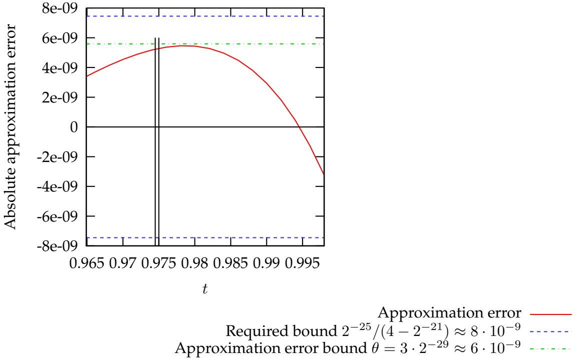
- $E_{\text{eval}} < \eta$,

  with $\quad \eta \approx 7.4 \cdot 10^{-9}$

Certification of evaluation error for *binary32* division

- Case 1: $m_x \geq m_y \rightarrow$ condition satisfied
- Case 2: $m_x < m_y \rightarrow$ condition not satisfied: $E_{\text{eval}} \geq \eta$

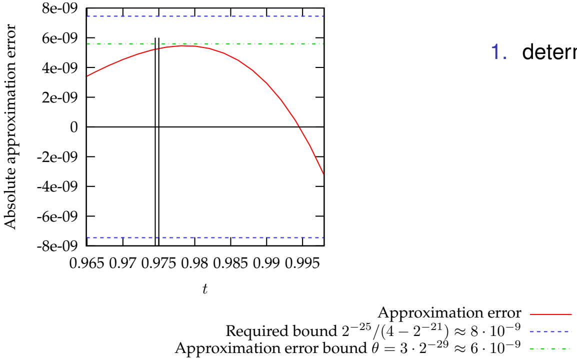
$s^* = 3.935581684112548828125$ and $t^* = 0.974904417991638183593755$



Certification of evaluation error for *binary32* division

- Case 1: $m_x \geq m_y \rightarrow$ condition satisfied
- Case 2: $m_x < m_y \rightarrow$ condition not satisfied: $E_{\text{eval}} \geq \eta$

$s^* = 3.935581684112548828125$ and $t^* = 0.974904417991638183359375$

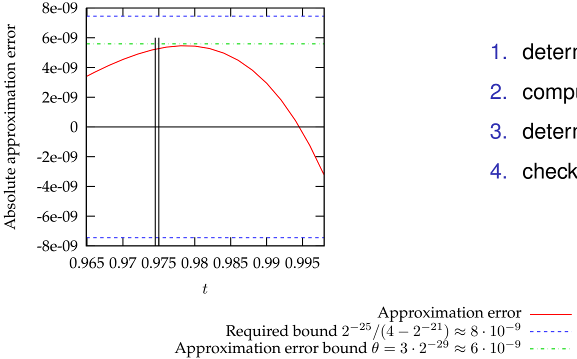


1. determine an interval I around this point

Certification of evaluation error for *binary32* division

- Case 1: $m_x \geq m_y \rightarrow$ condition satisfied
- Case 2: $m_x < m_y \rightarrow$ condition not satisfied: $E_{\text{eval}} \geq \eta$

$$s^* = 3.935581684112548828125 \text{ and } t^* = 0.974904417991638183593 75$$

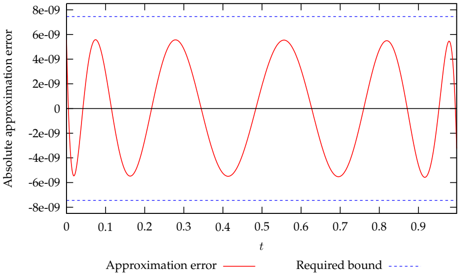


1. determine an interval I around this point
2. compute E_{approx} over I
3. determine an evaluation error bound η
4. check if $E_{\text{eval}} < \eta$?

Certification of evaluation error for *binary32* division

- Sufficient conditions for each subinterval, with $\mu = 4 - 2^{-21}$

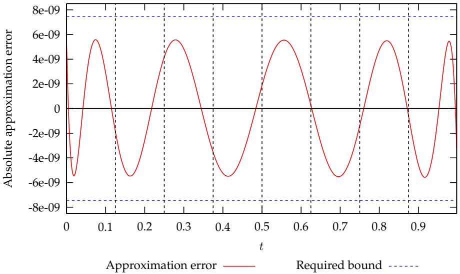
$$E_{\text{approx}}^{(i)} \leq \theta^{(i)} \quad \text{with} \quad \theta^{(i)} < 2^{-25}/\mu \qquad \text{and} \qquad E_{\text{eval}}^{(i)} < \eta^{(i)} = 2^{-25} - \mu \cdot \theta^{(i)}$$



Certification of evaluation error for *binary32* division

- Sufficient conditions for each subinterval, with $\mu = 4 - 2^{-21}$

$$E_{\text{approx}}^{(i)} \leq \theta^{(i)} \quad \text{with} \quad \theta^{(i)} < 2^{-25}/\mu \qquad \text{and} \qquad E_{\text{eval}}^{(i)} < \eta^{(i)} = 2^{-25} - \mu \cdot \theta^{(i)}$$



- $E_{\text{approx}}^{(i)} \leq \theta^{(i)}$
- $E_{\text{eval}}^{(i)} < \eta^{(i)}$

Certification using a dichotomy-based strategy

- Implementation of the splitting by dichotomy
  - for each $\mathcal{T}^{(i)}$
    1. compute a certified approximation error bound $\theta^{(i)}$
    2. determine an evaluation error bound $\eta^{(i)}$
    3. check this bound: $E_{\text{eval}}^{(i)} < \eta^{(i)}$

  $\Rightarrow$ if this bound is not satisfied, $\mathcal{T}^{(i)}$ is split up into 2 subintervals

Certification using a dichotomy-based strategy

- Implementation of the splitting by dichotomy
  - for each $\mathcal{T}^{(i)}$
    1. compute a certified approximation error bound $\theta^{(i)}$ *Sollya*
    2. determine an evaluation error bound $\eta^{(i)}$ *Sollya*
    3. check this bound: $E_{\text{eval}}^{(i)} < \eta^{(i)}$ *Gappa*

  $\Rightarrow$ if this bound is not satisfied, $\mathcal{T}^{(i)}$ is split up into 2 subintervals

Certification using a dichotomy-based strategy

- Implementation of the splitting by dichotomy
  - for each $\mathcal{T}^{(i)}$
    1. compute a certified approximation error bound $\theta^{(i)}$ *Sollya*
    2. determine an evaluation error bound $\eta^{(i)}$ *Sollya*
    3. check this bound: $E_{\text{eval}}^{(i)} < \eta^{(i)}$ *Gappa*

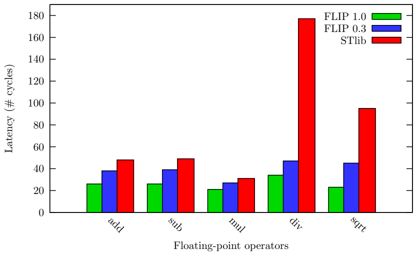
  $\Rightarrow$ if this bound is not satisfied, $\mathcal{T}^{(i)}$ is split up into 2 subintervals

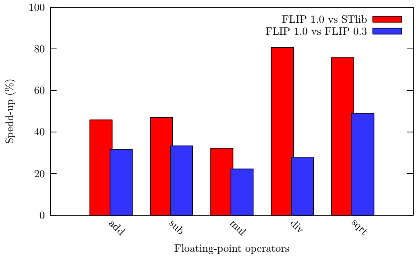
- Example of *binary32* implementation
  - $\longrightarrow$ launched on a 64 processor grid
  - $\longrightarrow$ 36127 subintervals found in several hours ($\approx$ 5h.)

Outline of the talk

1. Design and implementation of floating-point operators
2. Low latency parenthesization computation
3. Selection of effective evaluation parenthesizations
4. Numerical results
5. Conclusions
6. And now in ParLab: Debugging of floating-point programs

Performances of FLIP on ST231

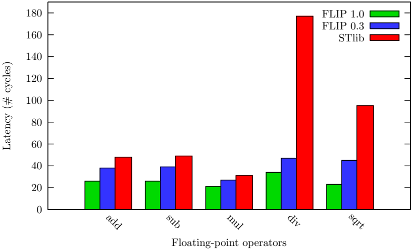


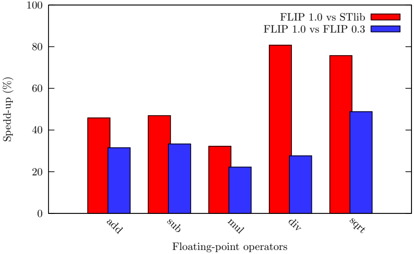


Performances on ST231, in RoundTiesToEven

$\Rightarrow$ Speed-up between 20 and 50 %

Performances of FLIP on ST231





Performances on ST231, in RoundTiesToEven

$\Rightarrow$ Speed-up between 20 and 50 %

- Implementations of other operators

x^{-1}	$x^{-1/2}$	$x^{1/3}$	$x^{-1/3}$	$x^{-1/4}$
25	29	34	40	42

Performances on ST231, in RoundTiesToEven (in number of cycles)

Outline of the talk

1. Design and implementation of floating-point operators

2. Low latency parenthesization computation

3. Selection of effective evaluation parenthesizations

4. Numerical results

5. Conclusions

6. And now in ParLab: Debugging of floating-point programs

Conclusions

- Design and implementation of floating-point operators
  - uniform approach for correctly-rounded roots and their reciprocals
  - extension to correctly-rounded division

Conclusions

- Design and implementation of floating-point operators
  - uniform approach for correctly-rounded roots and their reciprocals
  - extension to correctly-rounded division
  - polynomial evaluation-based method, very high ILP exposure

$\Rightarrow$ new, much faster version of FLIP

Conclusions

- Design and implementation of floating-point operators
  - uniform approach for correctly-rounded roots and their reciprocals
  - extension to correctly-rounded division
  - polynomial evaluation-based method, very high ILP exposure

  $\Rightarrow$ new, much faster version of FLIP

- Code generation for efficient and certified polynomial evaluation
  - methodologies and tools for automating polynomial evaluation implementation
  - heuristics and techniques for generating quickly efficient and certified C codes

  $\Rightarrow$ CGPE: allows to write and certify automatically $\approx$ 50 % of the codes of FLIP

Outline of the talk

1. Design and implementation of floating-point operators
2. Low latency parenthesization computation
3. Selection of effective evaluation parenthesizations
4. Numerical results
5. Conclusions
6. And now in ParLab: Debugging of floating-point programs

Debugging of floating-point programs

- Tool for detecting and remedying anomalies in floating-point programs
  - $\longrightarrow$ either at C code level or at run-time

- What are the usual anomalies?
  - rounding error accumulations
  - conditional branches involving floating-point comparisons
    - $\longrightarrow$ may fail due to the subtleties of floating-point arithmetic
  - difficulties of programming languages
    - $\longrightarrow$ Fortran: constants converted in full double precision accuracy if written with the `dX` notation
  - overflows, resolution of ill-conditioned problems
    - $\longrightarrow$ returned result may be completely wrong
  - benign / catastrophic cancellation, ...

Debugging of floating-point programs

- Tool for detecting and remedying anomalies in floating-point programs

  $\longrightarrow$ either at C code level or at run-time

- How to detect these usual anomalies?
  - altering rounding mode of floating-point arithmetic hardware

    $\longrightarrow$ may not be used for remedying problems
  - extending precision of floating-point computation

    $\longrightarrow$ run time may increase significantly (due to the use of software interface)
  - using interval arithmetic

    $\longrightarrow$ produces a certificate, but run time cost is the greatest

Debugging of floating-point programs

- Tool for detecting and remedying anomalies in floating-point programs

  $\longrightarrow$ either at C code level or at run-time

- How to detect these usual anomalies?
  - altering rounding mode of floating-point arithmetic hardware

    $\longrightarrow$ may not be used for remedying problems
  - extending precision of floating-point computation

    $\longrightarrow$ run time may increase significantly (due to the use of software interface)
  - using interval arithmetic

    $\longrightarrow$ produces a certificate, but run time cost is the greatest

How to detect quickly the most sensitive part of a C program?

Detection using *delta-debugging*

- Principle: find a minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, double precision, ...)

  $\longrightarrow$ implementation by binary search

```c
#include <math.h>
#include <stdio.h>

int
main( void )
{
  float a = 1e15f;
  float b = 1.0f;
  float c = a + b;
  float d = c - a;        // d = 0.0

  printf("The value of d is: %1.19e\n", d);

  return 0;
}
```

- What is the value of d?
  - Using *binary32* floating-point arithmetic

    $\longrightarrow$ $d = 0.0$
  - Using *binary64* floating-point arithmetic

    $\longrightarrow$ $d = 1.0$

Detection using *delta-debugging*

- Principle: find a minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, double precision, ...)

  $\longrightarrow$ implementation by binary search

```
#include <math.h>
#include <stdio.h>

int
main( void )
{
  double a = 1e15f;
  double b = 1.0f;
  float c = a + b;
  float d = c - a;       // d = 0.0

  printf("The value of d is: %1.19e\n", d);

  return 0;
}
```

- What is the value of *d*?
  - Using *binary32* floating-point arithmetic

    $\longrightarrow$ $d = 0.0$
  - Using *binary64* floating-point arithmetic

    $\longrightarrow$ $d = 1.0$

Detection using *delta-debugging*

- Principle: find a minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, double precision, ...)

  $\longrightarrow$ implementation by binary search

```c
#include <math.h>
#include <stdio.h>

int
main( void )
{
  float a = 1e15f;
  float b = 1.0f;
  double c = a + b;
  double d = c - a;      // d = 0.0

  printf("The value of d is: %1.19e\n", d);

  return 0;
}
```

- What is the value of d?
  - Using *binary32* floating-point arithmetic

    $\longrightarrow$ $d = 0.0$
  - Using *binary64* floating-point arithmetic

    $\longrightarrow$ $d = 1.0$

Detection using *delta-debugging*

- Principle: find a minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, double precision, ...)

  $\longrightarrow$ implementation by binary search

```
#include <math.h>
#include <stdio.h>

int
main( void )
{
  double a = 1e15f;
  float b = 1.0f;
  float c = a + b;
  float d = c - a;      // d = 0.0

  printf("The value of d is: %1.19e\n", d);

  return 0;
}
```

- What is the value of d?
  - Using *binary32* floating-point arithmetic

    $\longrightarrow$ $d = 0.0$
  - Using *binary64* floating-point arithmetic

    $\longrightarrow$ $d = 1.0$

Detection using *delta-debugging*

- Principle: find a minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, double precision, ...)

  $\longrightarrow$ implementation by binary search

```
#include <math.h>
#include <stdio.h>

int
main( void )
{
  float a = 1e15f;
  double b = 1.0f;
  float c = a + b;
  float d = c - a;      // d = 0.0

  printf("The value of d is: %1.19e\n", d);

  return 0;
}
```

- What is the value of d?
  - Using *binary32* floating-point arithmetic

    $\longrightarrow$ $d = 0.0$
  - Using *binary64* floating-point arithmetic

    $\longrightarrow$ $d = 1.0$

Detection using *delta-debugging*

- Principle: find a minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, double precision, ...)

  $\longrightarrow$ implementation by binary search

```
#include <math.h>
#include <stdio.h>

int
main( void )
{
  float a = 1e15f;
  float b = 1.0f;
  double c = a + b;
  float d = c - a;       // d = 0.0

  printf("The value of d is: %1.19e\n", d);

  return 0;
}
```

- What is the value of d?
  - Using *binary32* floating-point arithmetic

    $\longrightarrow$ $d = 0.0$
  - Using *binary64* floating-point arithmetic

    $\longrightarrow$ $d = 1.0$

Detection using *delta-debugging*

- Principle: find a minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, double precision, ...)

  $\longrightarrow$ implementation by binary search

```
#include <math.h>
#include <stdio.h>

int
main( void )
{
  float a = 1e15f;
  float b = 1.0f;
  float c = a + b;
  double d = c - a;      // d = 0.0

  printf("The value of d is: %1.19e\n", d);

  return 0;
}
```

- What is the value of d?
  - Using *binary32* floating-point arithmetic

    $\longrightarrow$ $d = 0.0$
  - Using *binary64* floating-point arithmetic

    $\longrightarrow$ $d = 1.0$

Detection using *delta-debugging*

- Principle: find a minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, double precision, ...)

  $\longrightarrow$ implementation by binary search

```c
#include <math.h>
#include <stdio.h>

int
main( void )
{
  float a = 1e15f;
  double b = 1.0f;
  double c = a + b;
  double d = c - a;      // d = 1.0

  printf("The value of d is: %1.19e\n", d);

  return 0;
}
```

- What is the value of d?
  - Using *binary32* floating-point arithmetic

    $\longrightarrow$ $d = 0.0$
  - Using *binary64* floating-point arithmetic

    $\longrightarrow$ $d = 1.0$

Detection using *delta-debugging*

- Principle: find a minimal set of changes on a C code, so that the returned result remains at a given threshold of a known and more accurate result (exact, double precision, ...)

  $\longrightarrow$ implementation by binary search

```c
#include <math.h>
#include <stdio.h>

int
main( void )
{
  float a = 1e15f;
  double b = 1.0f;
  double c = a + b;
  float d = c - a;      // d = 1.0

  printf("The value of d is: %1.19e\n", d);

  return 0;
}
```

- What is the value of *d*?
  - Using *binary32* floating-point arithmetic

    $\longrightarrow$ $d = 0.0$
  - Using *binary64* floating-point arithmetic

    $\longrightarrow$ $d = 1.0$

Current work

- *Delta-debugging*
  - how to determine initial set of changes?
  - implementation of other transformations
- Implementation of an exception handler
  - may be useful for building initial set of *delta-debugging* algorithm
- Detection of infinite loops, ...

BeBOP meeting (ParLab, EECS, UC Berkeley)

Berkeley, CA, USA - March 30, 2010

Implementation of binary floating-point arithmetic on embedded integer processors

Polynomial evaluation-based algorithms
and
certified code generation

Guillaume Revy

ParLab EECS University of California, Berkeley

Ph.D. thesis' work done under the direction of Claude-Pierre Jeannerod and Gilles Villard
Arénaire INRIA project-team (LIP, Ens Lyon, France)