MAX PLANCK INSTITUTE FOR SOFTWARE SYSTEMS

ANASTASIA VOLKOVA, EVA DARULOVA

SOUND APPROXIMATION OF PROGRAMS WITH ELEMENTARY FUNCTIONS

TRADING ACCURACY FOR PERFORMANCE

Elementary functions sin, cos, exp, log, …

- essential to scientific and financial computations
- may be a performance bottleneck (~75% execution time for SPICE simulator)
- evaluated using standard libm (math.h) in single or double precision

TRADING ACCURACY FOR PERFORMANCE

Elementary functions sin, cos, exp, log, …

- essential to scientific and financial computations
- may be a performance bottleneck (~75% execution time for SPICE simulator)
- evaluated using standard libm (math.h) in single or double precision

Goal:

improve performance at the cost of accuracy

TRADING ACCURACY FOR PERFORMANCE

Elementary functions sin, cos, exp, log, …

- essential to scientific and financial computations
- may be a performance bottleneck (~75% execution time for SPICE simulator)
- evaluated using standard libm (math.h) in single or double precision

Goal:

improve performance at the cost of *guaranteed* accuracy

TRADING ACCURACY FOR PERFORMANCE

Elementary functions sin, cos, exp, log, …

- essential to scientific and financial computations
- may be a performance bottleneck (~75% execution time for SPICE simulator)
- evaluated using standard libm (math.h) in single or double precision

Goal:

Automatically improve performance at the cost of *guaranteed* accuracy

OVERVIEW OF THE TOOL

Input: program over reals

```
def axisRotationX(x: Real, y: Real, theta: Real): Real =  {
    require(-2 <= x && x <= 2 && -4 <= y && y <= 4 && -5 <= theta && theta <= 5)

    x * cos(theta) + y * sin(theta)
  }
```

Output: C code with float64 & worst-case absolute error

Assuming libm:

- Absolute error 5.77e-15
- Roughly 38% of overall time for elementary functions

OVERVIEW OF THE TOOL

Input: program over reals

```
def axisRotationX(x: Real, y: Real, theta: Real): Real =  {
    require(-2 <= x && x <= 2 && -4 <= y && y <= 4 && -5 <= theta && theta <= 5)

    x * cos(theta) + y * sin(theta)
  }
```

Output: C code with float64 & worst-case absolute error

Assuming libm:

- Absolute error 5.77e-15 — Allow to increase
- Roughly 38% of overall time for elementary functions — Want to reduce

OVERVIEW OF THE TOOL

Input: program over reals

```
def axisRotationX(x: Real, y: Real, theta: Real): Real =  {
    require(-2 <= x && x <= 2 && -4 <= y && y <= 4 && -5 <= theta && theta <= 5)

    x * cos(theta) + y * sin(theta)
  } ensuring(res => res +/- 1e-13)
```

Output: C code with float64 & worst-case absolute error

Assuming libm:

- Absolute error 5.77e-15 — Allow to increase
- Roughly 38% of overall time for elementary functions — Want to reduce

OVERVIEW OF THE TOOL

Input: program over reals

```
def axisRotationX(x: Real, y: Real, theta: Real): Real =  {
    require(-2 <= x && x <= 2 && -4 <= y && y <= 4 && -5 <= theta && theta <= 5)

    x * cos(theta) + y * sin(theta)
  } ensuring(res => res +/- 1e-13)
```

Output: C code with float64 & worst-case absolute error

Assuming libm:

- Absolute error 5.77e-15 — Allow to increase
- Roughly 38% of overall time for elementary functions — Want to reduce

With our tool:

- Improve performance using custom approximations with guaranteed accuracy

User requirement	Overall speedup	Elem. func. speedup
1e-13	2.9%	7.6%
1e-12	13.4%	35.3%
1e-11	17.6%	46.3%

FLOATING-POINT ANALYSIS TOOLS AND CODE GENERATION

- IEEE 754-2008 standard (formats, operations, exceptions,…)
- Rounding errors must be modeled, analyzed and bounded:

$$\circ(x \text{ op } y) = (x \text{ op } y)(1+\delta), \quad |\delta| \leq u, \text{ op} = +, -, \times, /$$

$$\max_{x \in [a;b]} |f(x) - \tilde{f}(\tilde{x})|$$

FLOATING-POINT ANALYSIS TOOLS AND CODE GENERATION

- IEEE 754-2008 standard (formats, operations, exceptions,…)
- Rounding errors must be modeled, analyzed and bounded:

$$\circ(x \text{ op } y) = (x \text{ op } y)(1+\delta), \quad |\delta| \leq u, \text{ op} = +, -, \times, /$$

$$\max_{x \in [a;b]} |f(x) - \tilde{f}(\tilde{x})|$$

- Automated tool support
  - Certified error bounds (Gappa, FPTaylor, **Daisy**, PRECiSA, Real2Float,…)
  - Rewriting (SALSA) and mixed-precision tuning (Herbie)
  - Approximate computing (STOKE)
  - Code generators for small numerical kernels (**Metalibm**)

DAISY

- Static analysis of numerical codes
- Rewriting techniques
- Mixed-precision tuning
- Code generation in floating- and fixed-point by ensuring user-given error

Two-step data flow static analysis:

RANGE ANALYSIS	ROUNDOFF ERROR ANALYSIS
Interval and Affine Arithmetic	Affine Arithmetic

Arithmetic operations and common elementary functions (sin, cos, exp,…) assuming libm

https://github.com/malyzajko/daisy

METALIBM – CODE GENERATOR FOR MATH FUNCTIONS

INPUT

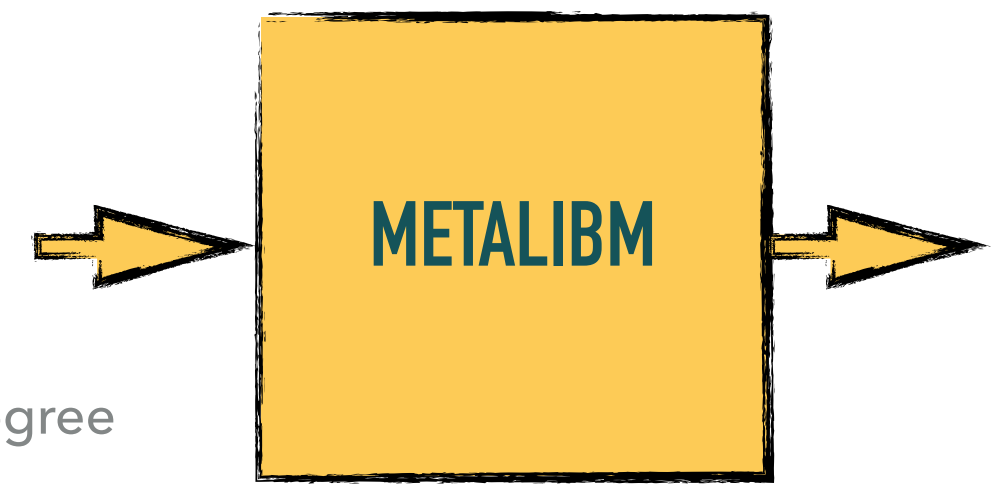
Function

Domain

Target error

Max approx degree

…



OUTPUT

C code

Gappa certificate

Three-stages of function evaluation:

PROPERTIES DETECTION

Symmetry, period, …

ARG REDUCTION / DOMAIN SPLITTING

uniform/arbitrary splitting

(PIECE-WISE) POLYNOMIAL APPROXIMATION

fpminimax

http://www.metalibm.org/lutetia.html

EMPOWERING DAISY BY USING METALIBM

Analyses errors and, given error budget, determines the room for improvement

Provides guaranteed implementations of elementary functions

https://github.com/malyzajko/daisy

http://www.metalibm.org/lutetia.html

KEY IDEA: ERROR BUDGET REPARTITION

Our example: f(x) = x * cos(theta) + y * sin(theta)

$$|f(x) - \tilde{f}(\tilde{x})| \leq |f(x) - \hat{f}_1(x)| + |\hat{f}_1(x) - \hat{f}_2(x)| + |\hat{f}_2(x) - \tilde{f}(\tilde{x})|$$

only cos() approximated | both cos() and sin() approximated | arithmetic approximated

When satisfying a priori error bound…

Step 1: bound the arithmetic errors

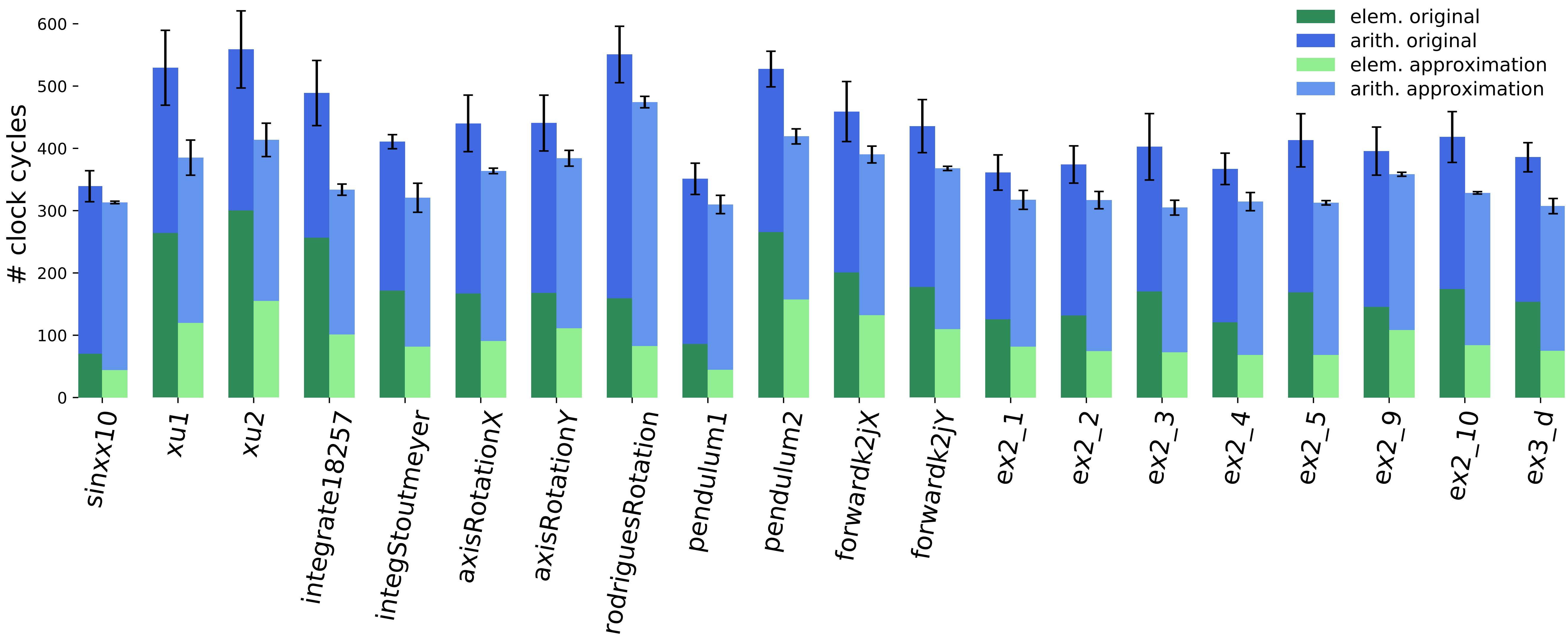
Step 2: repartition the remaining error budget among $\hat{f}_1$ and $\hat{f}_2$

Technique: estimate the sensitivity of a program wrt $\hat{f}_1$ and $\hat{f}_2$

OVERALL STRUCTURE

- Reading and decomposing the program
- Range and roundoff error analysis (float64 arithmetic + libm)
- Error budget repartition
- Code generation via Metalibm
- Computing final error bounds (always tighter than the target)
- Final C code generation

PERFORMANCE IMPROVEMENTS



Target errors: 4 orders of magnitude larger than libm-based
Compound functions: maximum depth

Average overall speedup: 18.1%
Average elem. function speedup: 54% (2x faster!)

CONCLUSION

- Automatic performance improvements even for non-experts
- Flexible tool for expert scientific computing developers
- Efficient heuristic to select suitable approximation parameters

https://github.com/malyzajko/daisy

Research report available on https://avolkova.org