Photo by Sebastian Bruggisser

Towards an Optimizing Compiler for Numerical Kernels

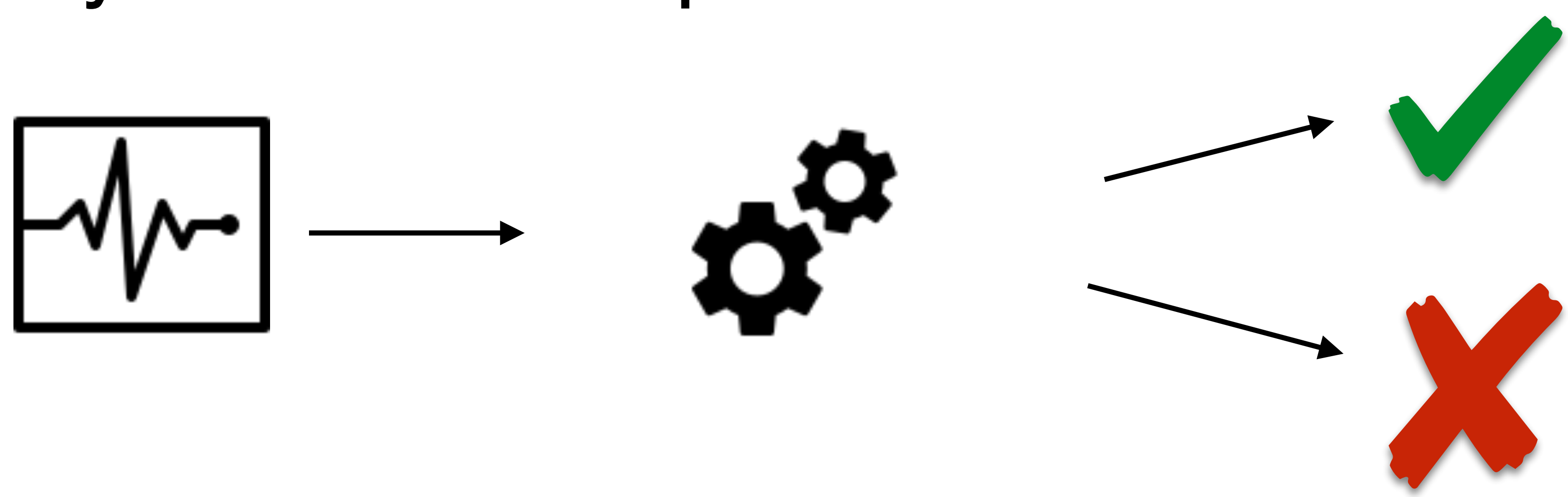
joint work with: Heiko Becker, Anastasiia Izycheva, Debasmita Lohar, Viktor Kuncak, Magnus Myreen, Sylvie Putot, Eric Goubault, Helmut Seidl

Eva Darulova
eva@mpi-sws.org

MAX PLANCK INSTITUTE FOR SOFTWARE SYSTEMS

Resources are Limited

Suppose you want to implement a heartbeat monitor:

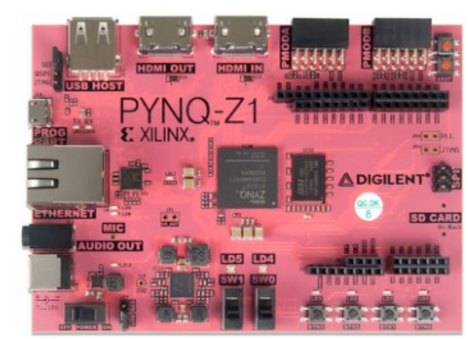


Design

infinite resources:

- perfect inputs
- continuous arithmetic

Implementation

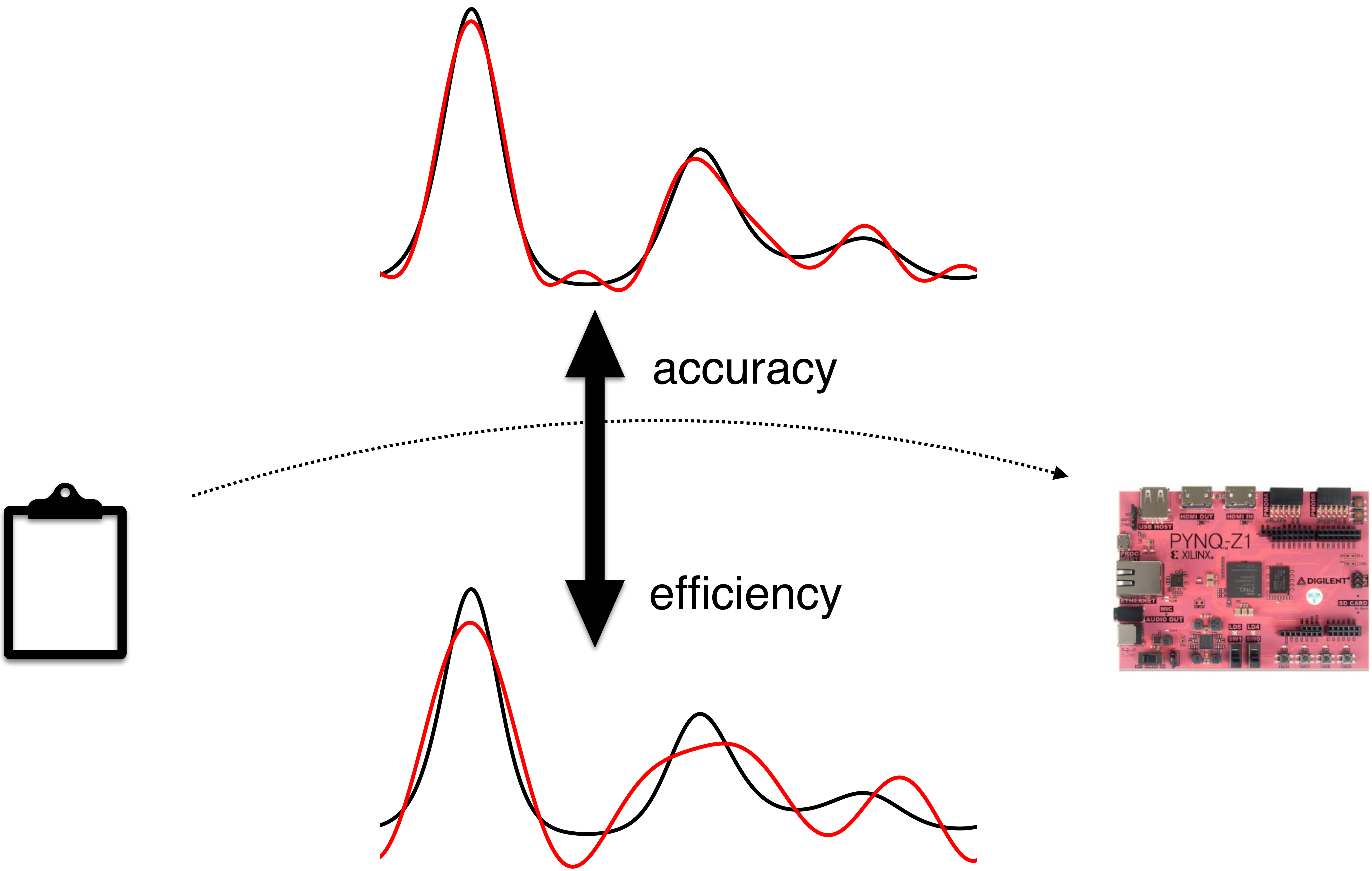


limited resources:

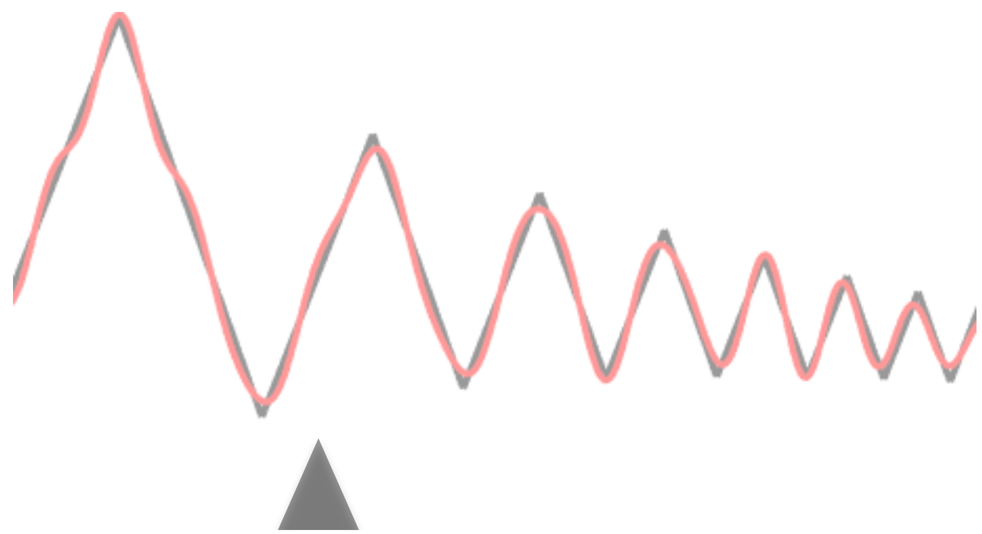
- noisy inputs
- finite-precision arithmetic

inspired from: A Methodology for Embedded Classification of Heartbeats Using Random Projections, DATE'13

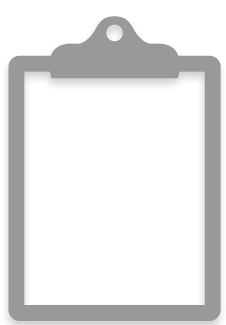
Approximations

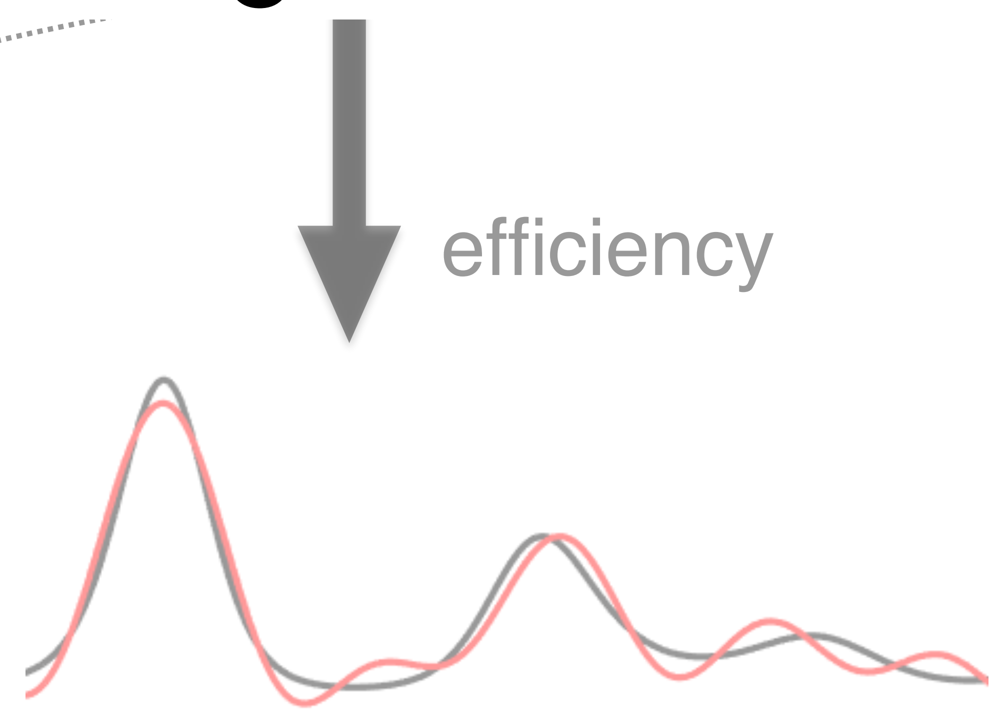


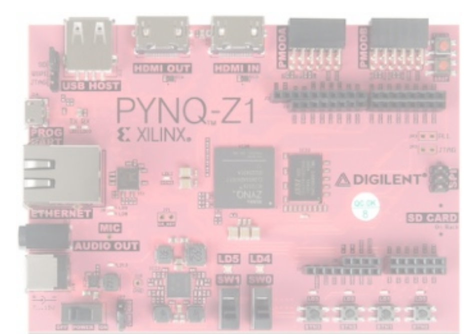
Approximations



Navigating the Tradeoff is Hard!







Programming with Approximations

state-of-the-art

Embedded systems and scientific computing

- manual
- costly
- error-prone

Programming languages

- automated
- sound
- limited point solutions

Vision: 'Approximating Compiler'

ideal real-valued program
with accuracy & resource spec

automatically

approximate finite-precision program
with correctness certificate

Today

real-valued specification
with transcendental functions

fixed-point/floating-point implementation
with polynomial approximations

Overview

real-valued specification
with transcendental functions

fixed-point/floating-point implementation
with polynomial approximations

Accuracy verification

- arithmetic
- conditionals

Optimization

- finite-precision
- elementary functions

Overview

real-valued specification
with transcendental functions

fixed-point/floating-point implementation
with polynomial approximations

Accuracy verification

- ***arithmetic***
- conditionals

Optimization

- finite-precision
- elementary functions

Daisy

real-valued specification

```
def sine(x: Real): Real = {
  require(-1.5 <= x && x <= 1.5 && x +/- 1e-11)

  x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0

} ensuring(res => res +/- 1.001e-11)
```

finite-precision implementation

floating-point arithmetic

```
def sine(x: Double): Double = {
  require(-1.5 <= x && x <= 1.5)

  x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0

}
```

fixed-point arithmetic

```
ap_fixed<64,3> sine(ap_fixed<64,2> x) {
  ap_fixed<64,4> _const0 = 6.0;
  ap_fixed<64,3> _tmp = (x * x);
  ap_fixed<64,3> _tmp1 = (_tmp * x);
  ap_fixed<64,1> _tmp2 = (_tmp1 / _const0);
  ap_fixed<64,3> _tmp3 = (x - _tmp2);
  ...
```

Worst-case Accuracy

for arithmetic expressions

```
def sine(x: Real): Real = {
  require(-1.5 <= x && x <= 1.5 && x +/- 1e-11)

  x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0

} ensuring(res => res +/- 1.001e-11)
```

absolute errors [TOPLAS'17]

- static data-flow analysis with interval & affine arithmetic
- interval subdivision

$$\max_{x \in I} |f(x) - \tilde{f}(\tilde{x})|$$

relative errors [FMCAD'17]

- global optimization
- for floating-points only

$$\max_{x \in I} \frac{|f(x) - \tilde{f}(\tilde{x})|}{|f(x)|}$$

Challenge: tight bounds for nonlinear arithmetic

Certificates [FMCAD'18, FM'19]

real-valued specification

```
def sine(x: Real): Real = {
  require(-1.5 <= x && x <= 1.5 && x +/- 1e-11)

  x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0

} ensuring(res => res +/- 1.001e-11)
```

formally verified finite-precision implementation

floating-point arithmetic

```
def sine(x: Double): Double = {
  require(-1.5 <= x && x <= 1.5)

  x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0

}
```

fixed-point arithmetic

```
ap_fixed<64,3> sine(ap_fixed<64,2> x) {
  ap_fixed<64,4> _const0 = 6.0;
  ap_fixed<64,3> _tmp = (x * x);
  ap_fixed<64,3> _tmp1 = (_tmp * x);
  ap_fixed<64,1> _tmp2 = (_tmp1 / _const0);
  ap_fixed<64,3> _tmp3 = (x - _tmp2);
  ...
```

Overview

real-valued specification
with transcendental functions

fixed-point/floating-point implementation
with polynomial approximations

Accuracy verification

- arithmetic
- ***conditionals***

Optimization

- finite-precision
- elementary functions

Conditionals: Continuous Case

```
    def sine(x: Real): Real = {
      require(-2.0 < x && x < 2.0 && x +/- 1e-11)

       if (x < 1.0) {
f1       0.95493 * x - 0.12901*(x*x*x)
      } else {
f2       x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0
      }

    } ensuring(res => res +/- 1.001e-11)
```

Control-flow may diverge:

reals

finite-precision

$$\max_{x \in I} |f_1(x) - \tilde{f}_2(\tilde{x})|$$

Challenge: complexity of constraint

Conditionals: Continuous Case

```
def sine(x: Real): Real = {
  require(-2.0 < x && x < 2.0 && x +/- 1e-11)

   if (x < 1.0) {
     0.95493 * x - 0.12901*(x*x*x)
   } else {
     x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0
   }

} ensuring(res => res +/- 1.001e-11)
```

f_1 — f_2

Control-flow may diverge:

reals

finite-precision

- break up total error into different manageable pieces [TOPLAS'17]

$$\max_{x \in I} |f_1(x) - \tilde{f}_2(\tilde{x})| \leq |f_1(x) - f_1(\tilde{x})| + |f_1(\tilde{x}) - f_2(\tilde{x})| + |f_2(\tilde{x}) - \tilde{f}_2(\tilde{x})|$$

Lipschitz const. real difference roundoff error

Challenge: complexity of constraint

Conditionals: Discrete Case

```
def rigidBody(x1: Real, x2: Real, x3: Real): Real = {
  require(-15.0 ≤ x1 ≤ 15 && -15.0 ≤ x2 ≤ 15.0 && -15.0 ≤ x3 ≤ 15)

  val res = -x1*x2 - 2*x2*x3 - x1 - x3

  if (res <= 0.0)
    raise_alarm()
  else
    continue()
}
```

reals

finite-precision

Conditionals: Discrete Case

```
def rigidBody(x1: Real, x2: Real, x3: Real): Real = {
  require(-15.0 ≤ x1 ≤ 15 && -15.0 ≤ x2 ≤ 15.0 && -15.0 ≤ x3 ≤ 15)

  val res = -x1*x2 - 2*x2*x3 - x1 - x3

  if (res <= 0.0)
    return 0
  else
    return 1
}
```

reals

finite-precision

worst-case analysis: maximum error = 1

Conditionals: Discrete Case

```
def rigidBody(x1: Real, x2: Real, x3: Real): Real = {
```

How often will the program return the wrong answer?

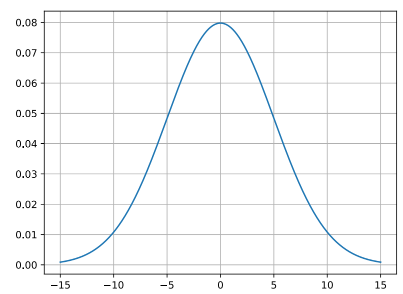
```
  val res = -x1*x2 - 2*x2*x3 - x1 - x3

  if (res <= 0.0)
    return 0
  else
    return 1
}
```

reals

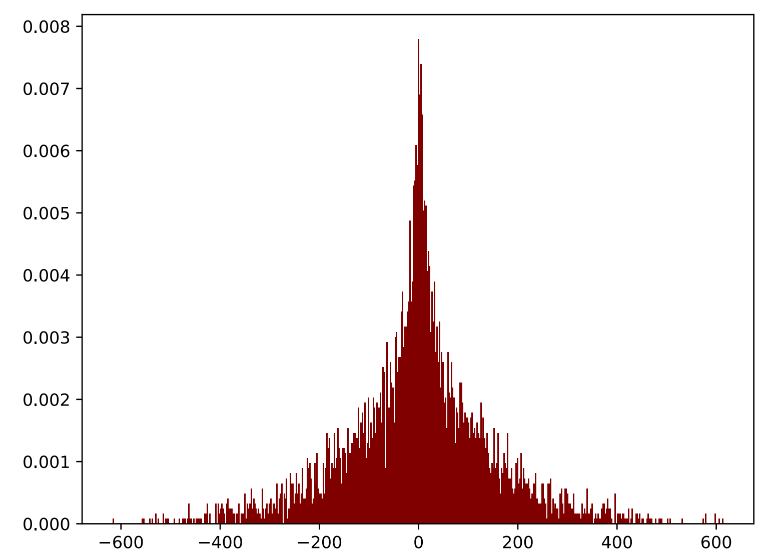
finite-precision

worst-case analysis: maximum error = 1

Probabilistic Analysis




```
def rigidBody(x1: Real, x2: Real, x3: Real): Real = {
  require(-15.0 ≤ x1 ≤ 15 && -15.0 ≤ x2 ≤ 15.0 && -15.0 ≤ x3 ≤ 15)

  val res = -x1*x2 - 2*x2*x3 - x1 - x3

  if (res <= 0.0)
    return 0
  else
    return 1

}
```



Goal: compute 'wrong path probability' (WPP)

- probability to compute the wrong answer

Exact Symbolic Inference

encode WPP as probabilistic program:

```
x1 := gauss(-15.0, 15.0);
x2 := gauss(-15.0, 15.0);
x3 := gauss(-15.0, 15.0);
res := -x1*x2 - 2*x2*x3 - x1 - x3;

error := 0.2042266; // worst-case error computed with Daisy
assert(0.0 - error <= res && res <= 0.0 + error);
```

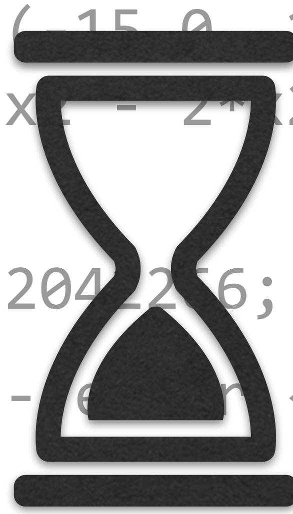
1. compute exact expression for WPP with PSI [1]
2. solve numerically with Mathematica

[1] PSI: Exact Symbolic Inference for Probabilistic Programs, CAV, 2016

Exact Symbolic Inference

encode WPP as probabilistic program:

```
x1 := gauss(-15.0, 15.0);
x2 := gauss(-15.0, 15.0);
x3 := gauss(-15.0, 15.0);
res := -x1*x[illegible] - 2*x2*x3 - x1 - x3;

error := 0.204[illegible]; // computed with Daisy
assert(0.0 - error <= res && res <= 0.0 + error);
```



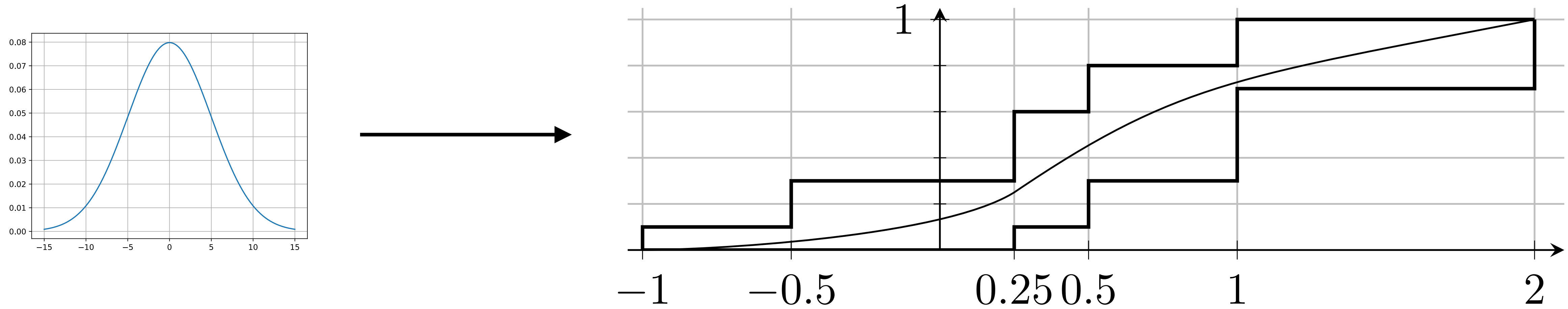
20min

1. compute exact expression for WPP with PSI [1]

2. solve numerically with Mathematica

[1] PSI: Exact Symbolic Inference for Probabilistic Programs, CAV, 2016

Probabilistic Range Analysis



- discretize input distribution

  $\{\langle[a_1, b_1], w_1\rangle, \langle[a_2, b_2], w_2\rangle, \ldots, \langle[a_n, b_n], w_n\rangle\}$

  - number of subdivisions determines accuracy
- propagation for independent variables: interval arithmetic
- propagation for dependent variables
  - LP problem
  - keep track of linear dependencies with affine arithmetic [2]

[2] A Generalization of P-boxes to Affine Arithmetic, Computing, vol. 94, no. 2-4, pp. 189–201, 2012.

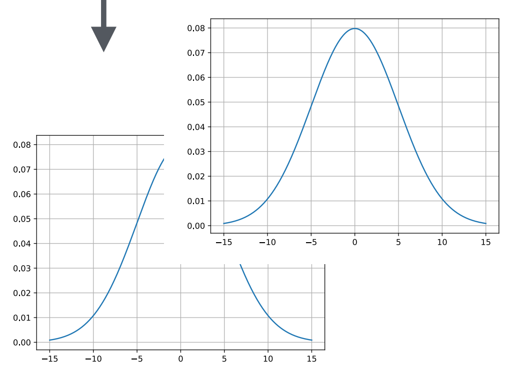
Computing WPP [EMSOFT'18]

input program

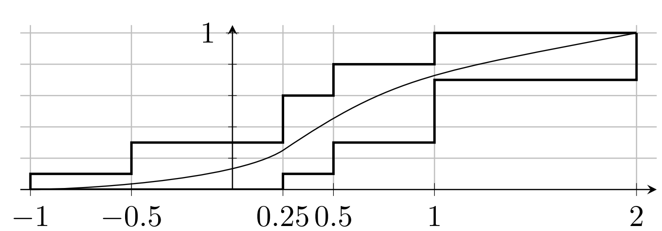
$f(\bar{x})$

roundoff analysis

$f(\bar{x})$



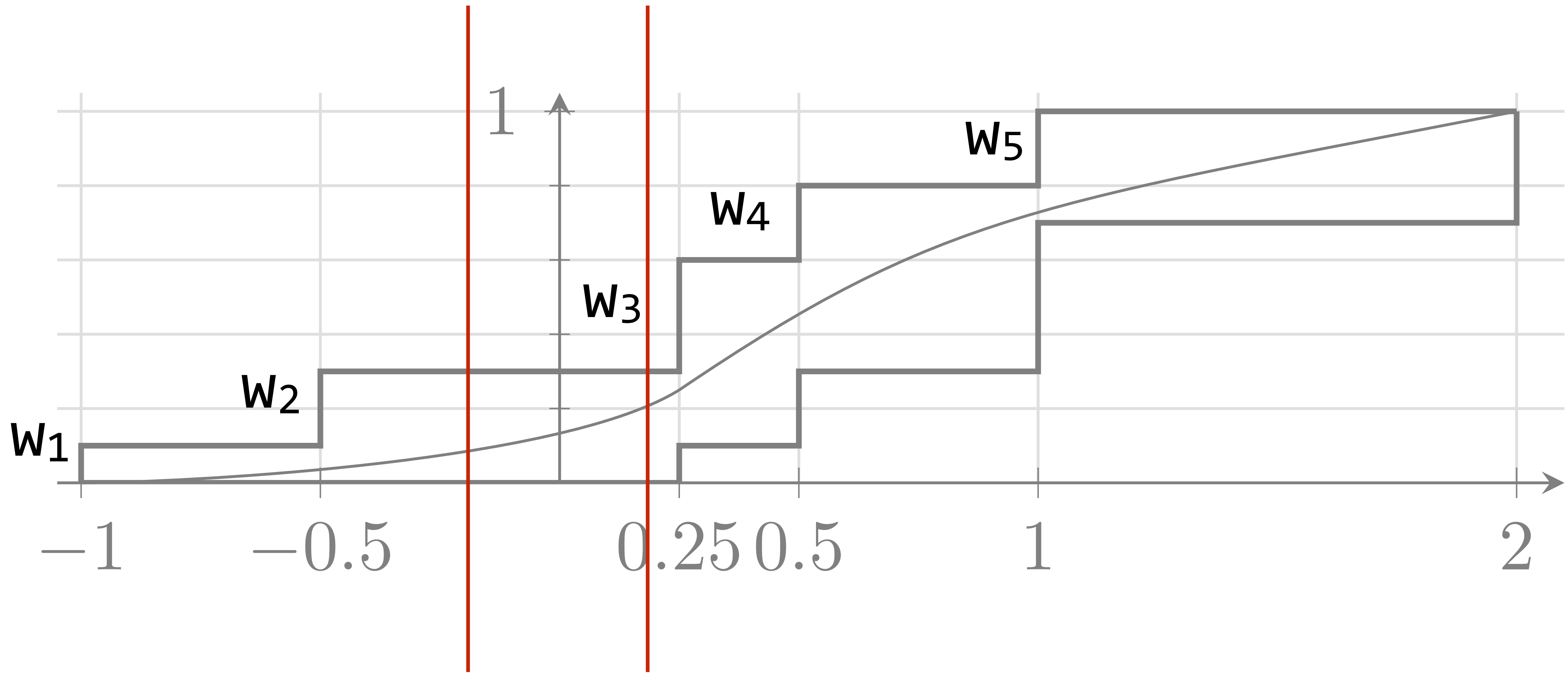
probabilistic analysis



`error := 0.2042266`

compute intersection (WPP)

Computing Intersection

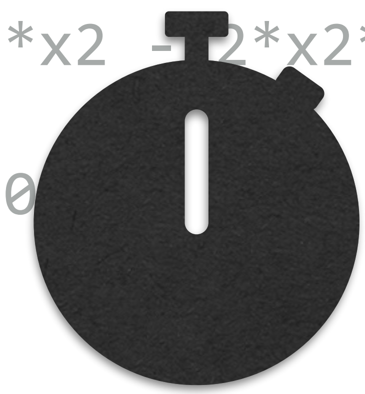



```
T := 0.0

error := 0.2042266
```

$\longrightarrow$ WPP = $w_1 + w_2$

Probabilistic Range Analysis

```
def rigidBody(x1: Real, x2: Real, x3: Real): Real = {
  require(-15.0 ≤ x1 ≤ 15 && -15.0 ≤ x2 ≤ 15.0 && -15.0 ≤ x3 ≤ 15)

  val res = -x1*x2 - 2*x2*x3 - x1 - x3

  if (res <= 0.0
    return 0
  else
    return 1

}
```

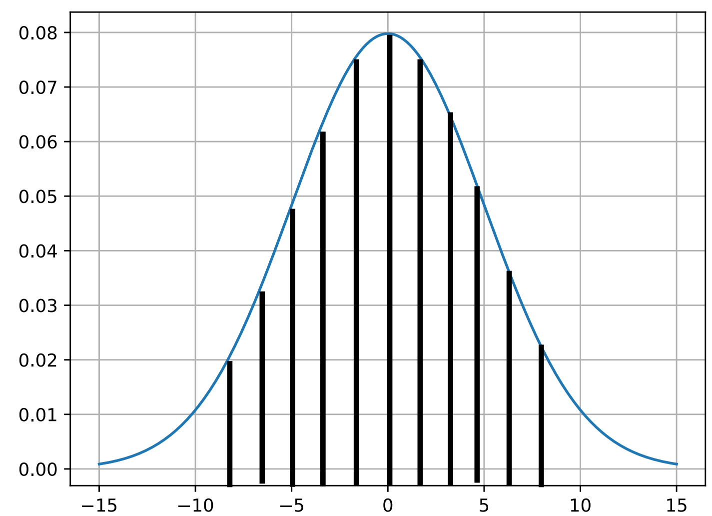


WPP = 1.0

Computing WPP II

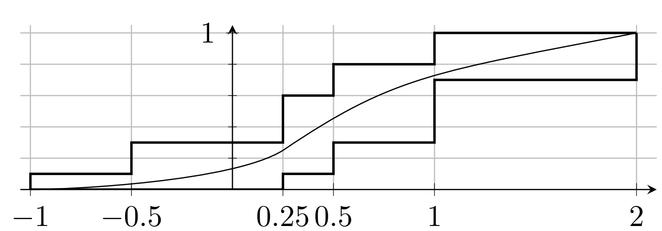
input program

roundoff analysis



interval subdivision

probabilistic analysis



`error := 0.2042266`

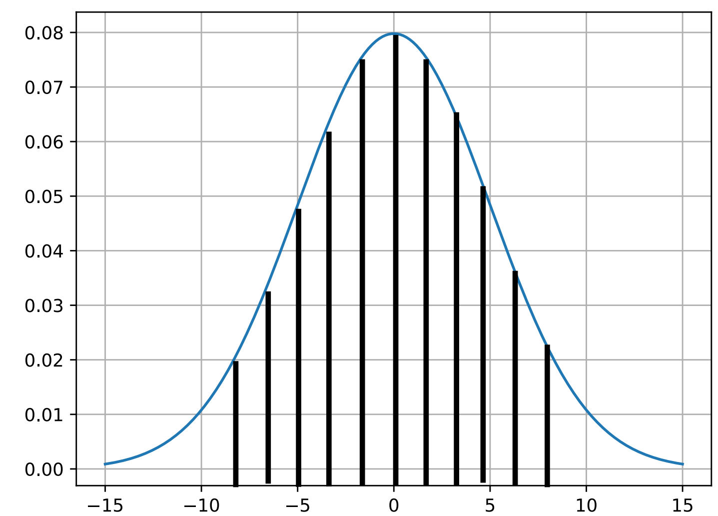
compute intersection (WPP)

Computing WPP III

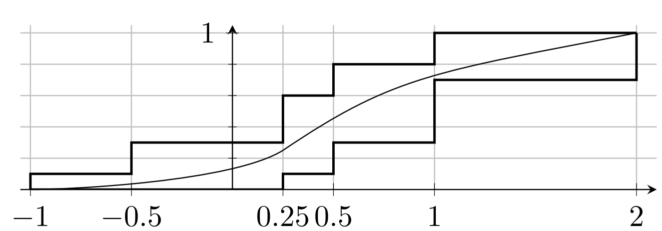
input program

roundoff analysis

interval subdivision with reachability check



probabilistic analysis




```
error := 0.2042266
```

compute intersection (WPP)

Probabilistic Range Analysis

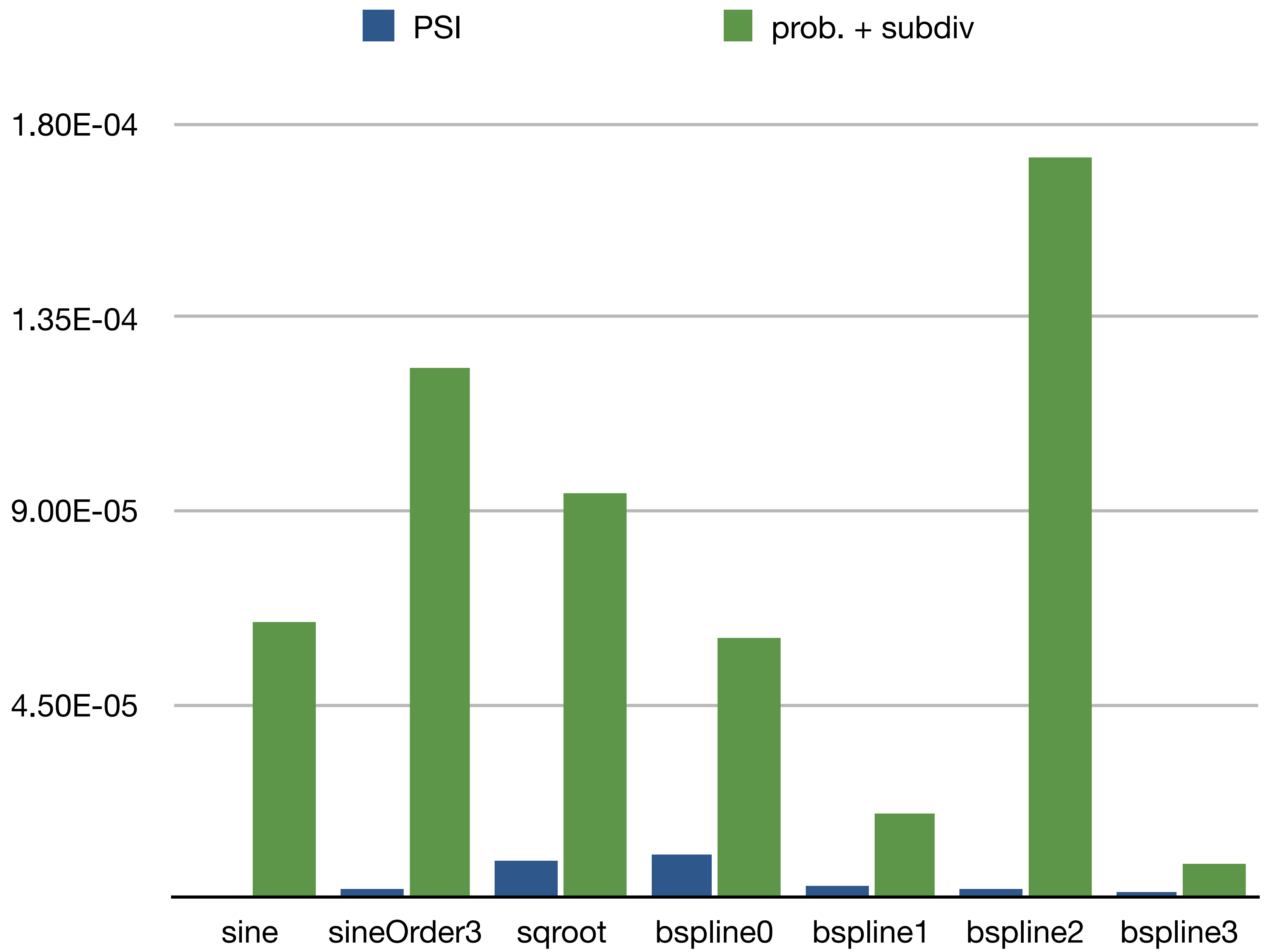
with subdivision

```
def rigidBody(x1: Real, x2: Real, x3: Real): Real = {
  require(-15.0 ≤ x1 ≤ 15 && -15.0 ≤ x2 ≤ 15.0 && -15.0 ≤ x3 ≤ 15)

  val res = -x1*x2 - 2*x2*x3 - x1 - x3

  if (res <= 0.0
    return 0
  else
    return 1

}
```

WPP = 0.07060

Experimental Results

- analysis runs on the order of minutes
- computes different WPP for gaussian and uniform inputs (as expected)
- over-approximation modest (about one order of magnitude):



Overview

real-valued specification
with transcendental functions

fixed-point/floating-point implementation
with polynomial approximations

Accuracy verification

- arithmetic
- conditionals

Optimization

- ***finite-precision***
- elementary functions

Assigning Precision

```
def rigidBody(x1: Real, x2: Real, x3: Real): Real = {
  require(-15.0 ≤ x1 ≤ 15 && -15.0 ≤ x2 ≤ 15.0 && -15.0 ≤ x3 ≤ 15)

   - x1 *  x2 -  (2*  x2) *  x3 -  x1 -  x3


} ensuring(res => res +/- 1.75e-13)
```

`Double` precision is *just* not enough: `3.5e-13`

`Quad` satisfies absolute error bound: `1.5e-28`
but is significantly slower than double precision

Assigning Precision

```
def rigidBody(x1: Real, x2: Real, x3: Real): Real = {
  require(-15.0 ≤ x1 ≤ 15 && -15.0 ≤ x2 ≤ 15.0 && -15.0 ≤ x3 ≤ 15)
```

$$-_D x1 \; *_D \; x2 \; -_Q \; (2 *_D x2) \; *_Q \; x3 \; -_Q \; x1 \; -_Q \; x3$$

```
} ensuring(res => res +/- 1.75e-13)
```

Double precision is *just* not enough: `3.5e-13`

Quad satisfies absolute error bound: `1.5e-28`
but is significantly slower than double precision

Mixed-precision satisfies absolute error bound
28% faster than uniform quad precision

Challenge: large, complex search space

Mixed-Precision Tuning

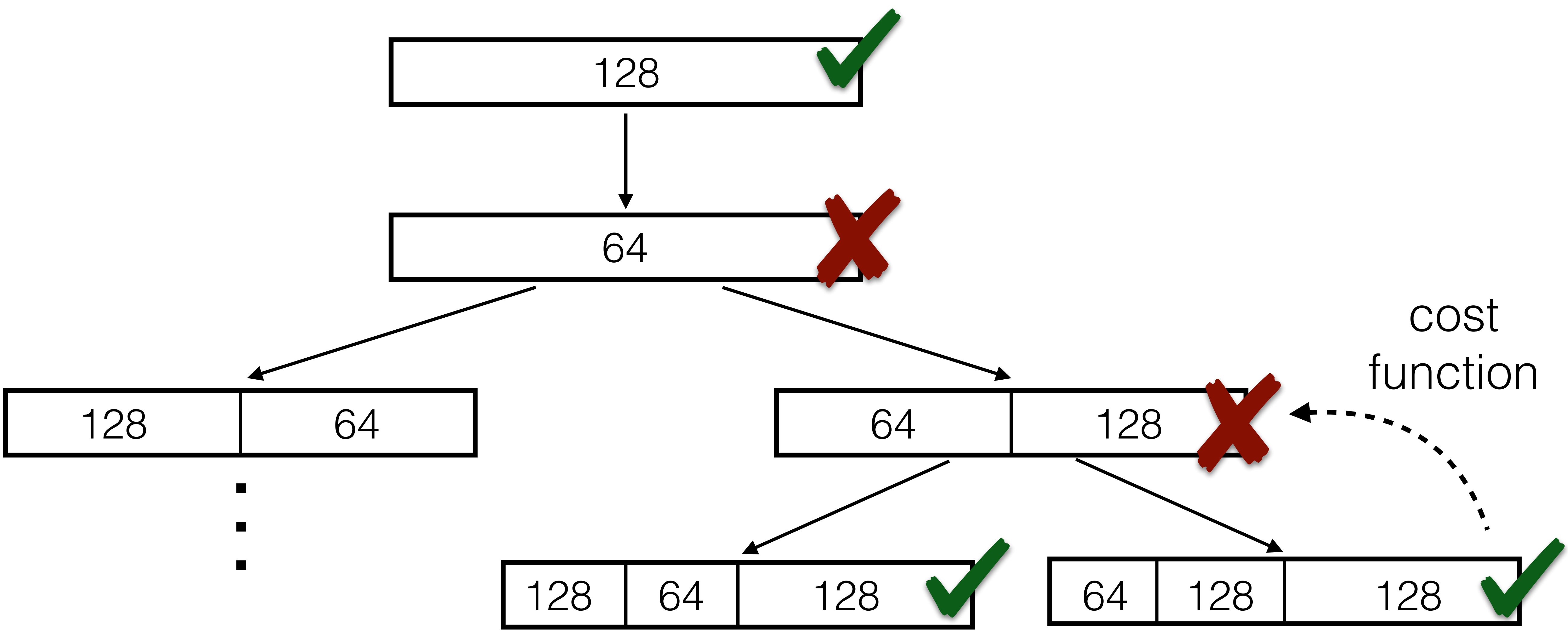
Our solution:

- incomplete search with static error analysis
- static cost function

Mixed-Precision Tuning

Our solution:

- incomplete search (delta debugging [3]) with static error analysis
- static cost function



[3] Precimonious: Tuning assistant for floating-point precision, SC, 2013

Mixed-Precision Tuning

Our solution:

- incomplete search with static error analysis
- static cost function

for floating-point arithmetic:

- **benchmarked** (best for Float, Double)
- **simple** (best for Float, Double, Quad)

for fixed-point arithmetic:

- **area-based**
- **machine-learning based** (for performance)

Rewriting

```
a + (b + c) ≠ (a + b) + c
```

Goal: find computation order which

- incurs *smallest* roundoff error (over input range)
- is equivalent under real-valued semantics

Our solution: genetic (heuristic) algorithm:

Iteratively evolve a population of expressions:

- mutate expression (associativity, distributivity etc. rules)
- evaluate fitness (static roundoff error)
- pick expr. from population (smaller roundoffs more likely)

significant (up to 70%) improvements in errors possible

Challenge: large, complex search space

Rewriting & Precision Tuning

[ICCPS'18]

rewriting (improves error)

+

mixed-precision (improves performance)

=

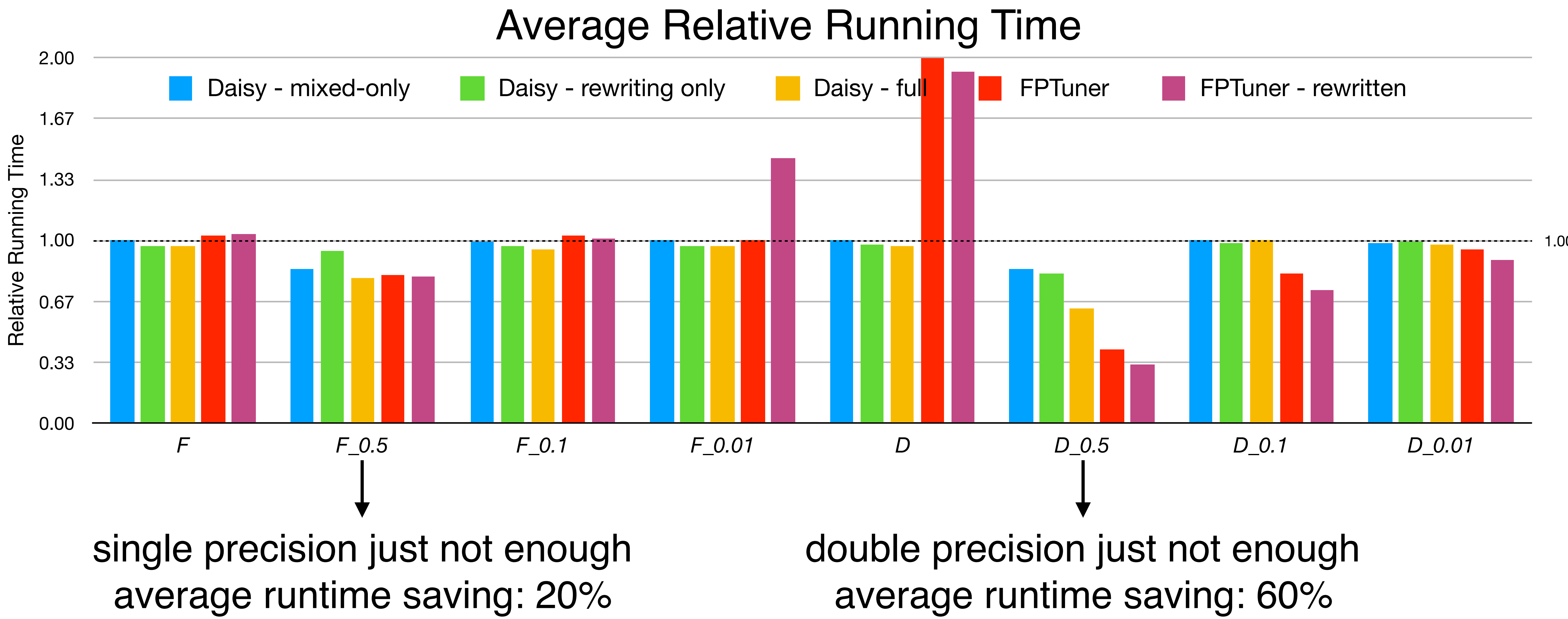
improves performance even more

def rigidBody(x1: **Quad**, x2: **Quad**, x3: **Double**): **Double** = {
($-_D$(x1 $*_Q$ x2) $-_D$ (x1 $+_Q$ x3)) $-_D$ ((x2 $*_Q$ 2.0f) $*_D$ x3)
}

satisfies absolute error bound

<u>43%</u> faster than uniform quad precision

Experimental Results



- rewriting generally helpful

	FPTuner	Daisy
doppler	12m 48s	5min 4s
kepler	1h 26m 3s	2m 9s
rigidBody	4m 45s	36s
traincar	17m 17s	2m 11s

Overview

real-valued specification
with transcendental functions

fixed-point/floating-point implementation
with polynomial approximations

Accuracy verification

- arithmetic
- conditionals

Optimization

- finite-precision
- ***elementary functions***

Elementary Functions

```
def axisRotationX(x: Real, y: Real, theta: Real): Real = {
  require(-2 ≤ x ≤ 2 && -2 ≤ y ≤ 2 && 0.01 ≤ theta ≤ 1.5)

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

} ensuring (res => res +/- 1.49e-6)
```

- library functions provide limited choice of precisions
- fixed-point implementations (for FPGAs) are inefficient

Goal: synthesize polynomial approximations

- efficient *specialized* implementation
- guaranteed *end-to-end* error bound
- *fixed-point* arithmetic implementation

Challenge: distribution of error budget

Elementary Function Synthesis [ATVA'19]

High-level Algorithm:

1. distribute global error budget
2. for each elementary function, distribute local error budget between:
   - polynomial approximation
   - fixed-point arithmetic of approximation

Global Error Distribution

High-level Algorithm:

1. distribute global error budget
2. for each elementary function, distribute local error budget between:
   - polynomial approximation
   - fixed-point arithmetic of approximation

Use *mixed-precision tuning* to assign precision to each

- arithmetic operation
- elementary function call
  - transform precision assigned to functions into local error
  - key idea: treat approximation errors as roundoff

- abstract cost function assigns 2x cost to elementary functions

Local Error Distribution

High-level Algorithm:

1. distribute global error budget
2. for each elementary function, distribute local error budget between:
   - polynomial approximation
   - fixed-point arithmetic of approximation

Feedback loop between

- start with equal split, estimate cost via cost function
- try increasing/decreasing approximation budget

Polynomial Approximation

High-level Algorithm:

1. distribute global error budget
2. for each elementary function, distribute local error budget between:
    - polynomial approximation
    - fixed-point arithmetic of approximation

Metalibm [4]: generator for piece-wise polynomial approximations

- Remez' algorithm for best polynomial approximation
- equal domain-splitting for piece-wise best approximation
- efficient double-precision floating-point implementation

[4] Metalibm: A Mathematical Functions Code Generator, ICMS 2014

Fixed-point Precision Assignment

[ATVA'19]

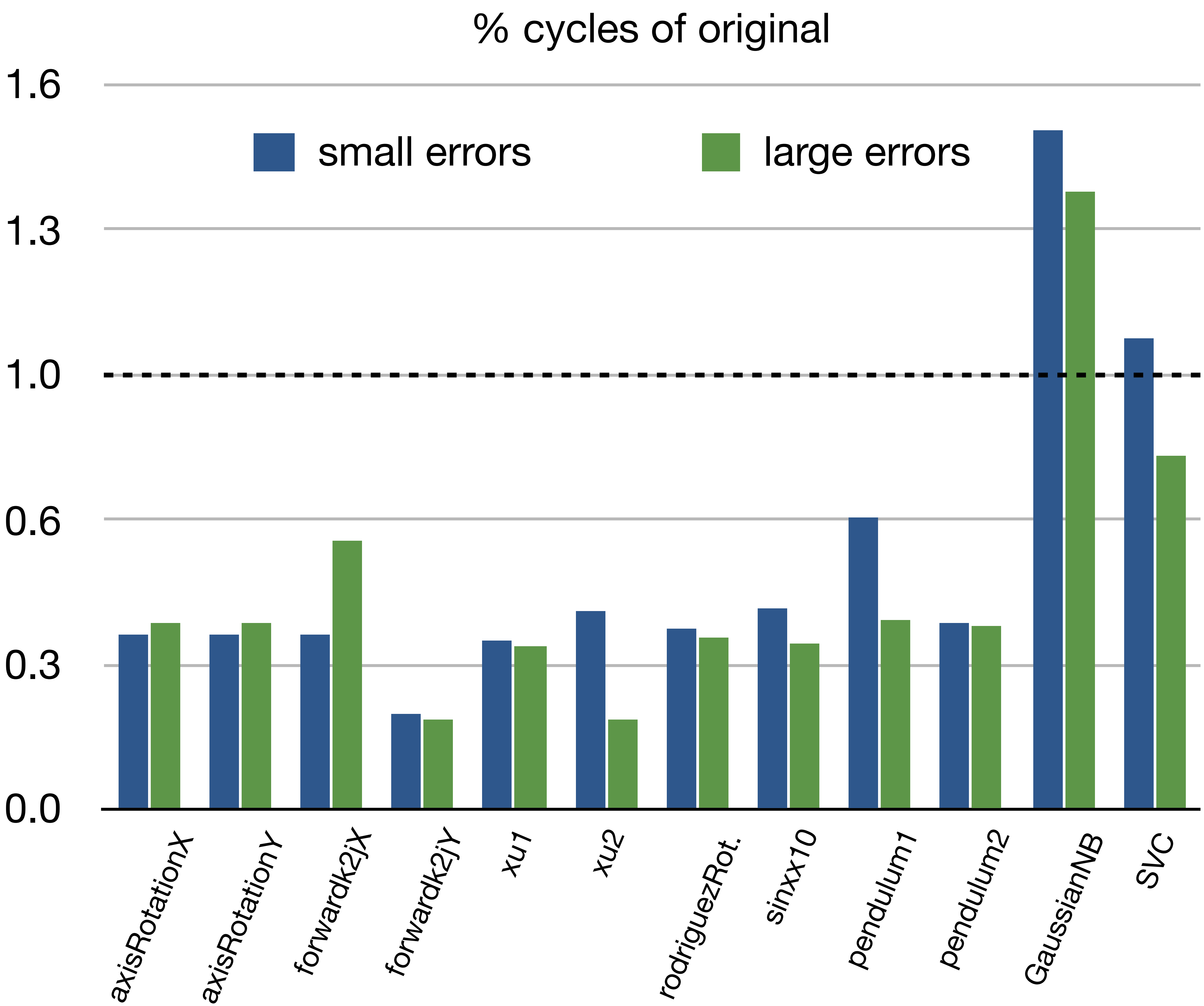
High-level Algorithm:

1. distribute global error budget
2. for each elementary function, distribute local error budget between:
    - polynomial approximation
    - fixed-point arithmetic of approximation

```
def sin_0_01to1_5(x: Real): Real = {
  if (x < 1.3021) {
    c0 +(c1 +((c3 +((c4 +((c5 +((c7 +(c8 * x)) * (x*x)))* x))* x)) * (x*x))) * x;
  } else {
    xh = x - s1
    b0 + b1 * (b2 + (b4 + (b6 + b7 * xh) * xh) * xh)
}
```

- assign mixed or uniform precision to each polynomial approximation

Experimental Results



average:
45%

Daisy [TACAS'18]

<u>real-valued</u> specification
with <u>transcendental</u> functions

<u>fixed-point/floating-point</u> implementation
with <u>polynomial approximations</u>

Accuracy verification

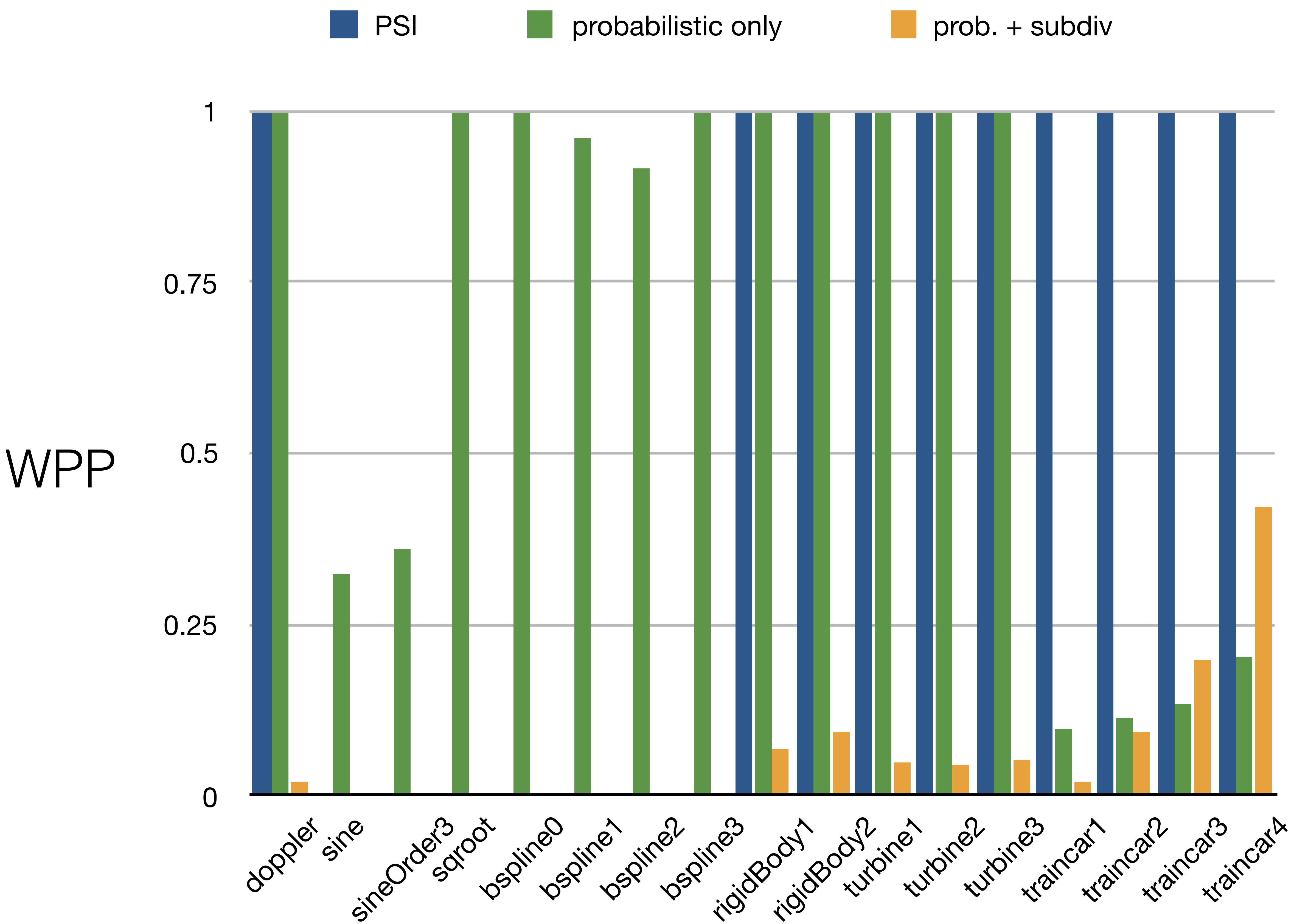
- arithmetic
- conditionals

Optimization

- finite-precision
- elementary functions

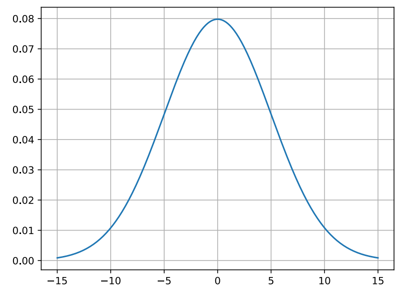
Next Big Challenge: Scalability

Experimental Results WPP



Worst-case is Pessimistic

Not all inputs are equally likely!



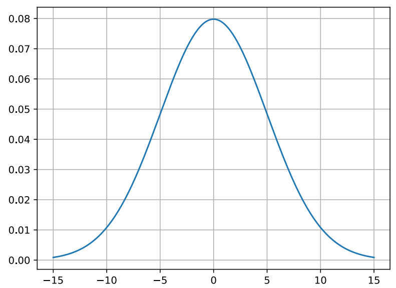
```
def sine(x: Real): Real = {
  require(-2.0 < x && x < 2.0)

    0.95493 * x - 0.12901*(x*x*x)
}
```

Worst-case error bounds can be too pessimistic:

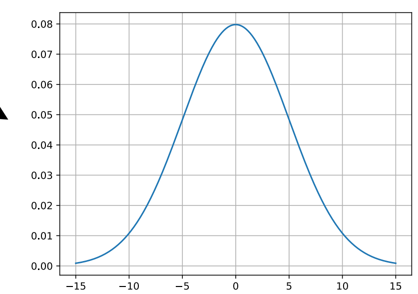
- not all errors are equally likely
- applications may tolerate an *occasional* large error

Probabilistic Analysis

Not all inputs are equally likely!




```
def sine(x: Real): Real = {
  require(-2.0 < x && x < 2.0)

    0.95493 * x - 0.12901*(x*x*x)
}
```



Alternative error specification: error bound with a probability

- probabilistic range analysis — 2.97e-7 with probability 0.85
- probabilistic interval subdivision — 2.67e-7 with probability 0.85