A Compiler for Sound Floating-Point using Affine Arithmetic

Joao Rivera, ETH Zürich

Franz Franchetti, Carnegie Mellon University

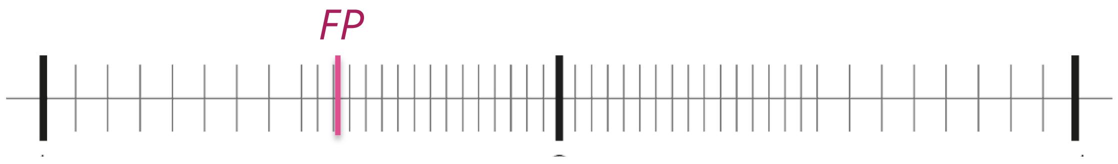
Markus Püschel, ETH Zürich

ETH zürich
Computer Science

```
double henon_map(double x , double y , int n) {
    double xi, yi;
    double a = 1.05;
    double b = 0.3;
    for (int i = 0; i < n ; i ++) {
        xi = x;
        yi = y;
        x = 1.0 - a*xi*xi + yi ;
        y = b*xi ;
    }
    return x;
}
```

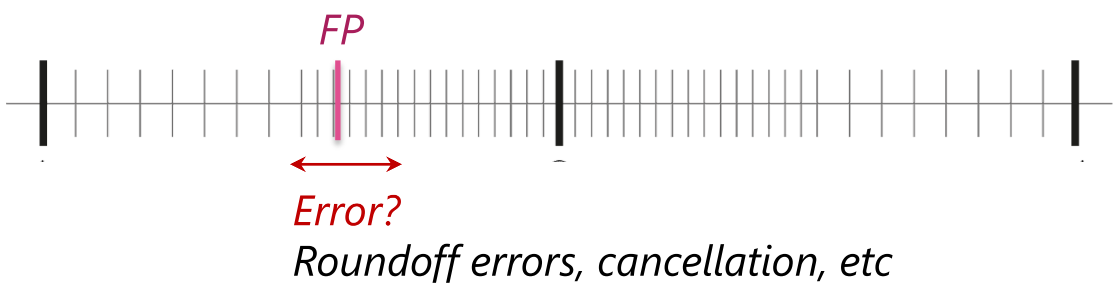
```
double henon_map(double x , double y , int n) {
   double xi, yi;
   double a = 1.05;
   double b = 0.3;
   for (int i = 0; i < n ; i ++) {
      xi = x;
      yi = y;
      x = 1.0 - a*xi*xi + yi ;
      y = b*xi ;
   }
   return x;
}
```

How accurate is the result?



```c
double henon_map(double x , double y , int n) {
   double xi, yi;
   double a = 1.05;
   double b = 0.3;
   for (int i = 0; i < n ; i ++) {
      xi = x;
      yi = y;
      x = 1.0 - a*xi*xi + yi ;
      y = b*xi ;
   }
   return x;
}
```

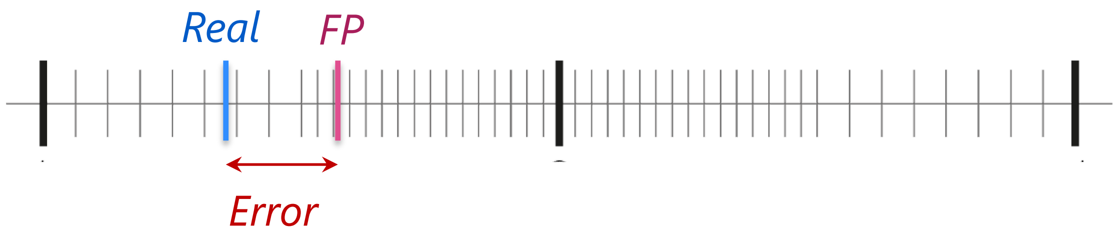
How accurate is the result?
We don't know…



FP is not **sound** with respect to real arithmetic.

```
double henon_map(double x , double y , int n) {
   double xi, yi;
   double a = 1.05;
   double b = 0.3;
   for (int i = 0; i < n ; i ++) {
      xi = x;
      yi = y;
      x = 1.0 - a*xi*xi + yi ;
      y = b*xi ;
   }
   return x;
}
```

How accurate is the result?
We don't know…



FP is not **sound** with respect to real arithmetic.

Sound Floating-Point

Accuracy guarantees of the final results.

How to make the code sound?

Ideal approach:

Static round-off error analysis.

```
double henon_map(double x , double y , int n) {
    double xi, yi;
    double a = 1.05;
    double b = 0.3;
    for (int i = 0; i < n ; i ++) {
        xi = x;  yi = y;
        x = 1.0 - a*xi*xi + yi ;
        y = b*xi ;
    }
    return x;
}
```

→ **Static Analyzer** → Absolute error

How to make the code sound?

Id

St

d

}

r

}

Certified Roundoff Error Bounds Using Semidefinite Programming

VICTOR MAGRON, CNRS Verimag
GEORGE CONSTANTINIDES and ALASTAIR DONALDSON, Imperial College London

Roundoff errors cannot be avoided when implementing numerical programs with finite precision. The ability to reason about rounding is especially important if one wants to explore a range of potential representations for instance, for FPGAs or custom hardware implementations. This problem becomes challenging when the program does not employ solely linear operations as non-linearities are inherent to many interesting

Towards a Compiler for Reals[1]

EVA DARULOVA[2], Max Planck Institute for Software Systems
VIKTOR KUNCAK[3], Ecole Polytechnique Federale de Lausann

Numerical software, common in scientific computing or emb
precision approximation of the real arithmetic in which most al
tions, the roundoff errors introduced by finite-precision arithmet
and measurement and other input errors further increase the uncer
tools are needed to help users select suitable data types and evalu
safety-critical applications.

We present a source-to-source compiler called Rosa which ta
error specifications and synthesizes code over an appropriate floa
main challenge of such a compiler is a fully automated, sound a
estimation. We introduce a unified technique for bounding roun
point arithmetic of various precisions. The technique can handle no
symbolic invariants for unbounded loops and quantify the effects
evaluate Rosa on a number of benchmarks from scientific computin
it to state-of-the-art in automated error estimation, show that it
accuracy and performance.

Static Analysis of Finite Precision Computations

Eric Goubault and Sylvie Putot

CEA LIST, Laboratory for the Modelling and Analysis of Interacting Systems,
Point courrier 94, Gif-sur-Yvette, F-91191 France, Firstname.Lastname@cea.fr

Abstract. We define several abstract semantics for the static analysis of finite precision computations, that bound not only the ranges of values taken by numerical variables of a program, but also the difference with the result of the same sequence of operations in an idealized real number semantics. These domains point out with more or less detail (control point, block, function for instance) sources of numerical errors in the program and the way they were propagated by further computations, thus
o sensitivity to
sses of abstrac-
akly relational

Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions

ALEXEY SOLOVYEV, MAREK S. BARANOWSKI, IAN BRIGGS, CHARLES JACOBSEN, ZVONIMIR RAKAMARIĆ, and GANESH GOPALAKRISHNAN, School of Computing, University of Utah, Salt Lake City, UT, USA

2

Rigorous estimation of maximum floating-point round-off errors is an important capability central to many formal verification tools. Unfortunately, available techniques for this task often provide very pessimistic over-estimates, causing unnecessary verification failure. We have developed a new approach called *Symbolic Taylor Expansions* that avoids these problems, and implemented a new tool called FPTaylor embodying this approach. Key to our approach is the use of rigorous global optimization, instead of the more familiar interval arithmetic, affine arithmetic, and/or SMT solvers. FPTaylor emits per-instance analysis certificates in the form of HOL Light proofs that can be machine checked.

In this article, we present the basic ideas behind Symbolic Taylor Expansions in detail. We also survey as well as thoroughly evaluate six tool families, namely, Gappa (two tool options studied), Fluctuat, PRECiSA, Real2Float, Rosa, and FPTaylor (two tool options studied) on 24 examples, running on the same machine, and

How to make the code sound?

Ideal approach:
Static round-off error analysis.

```
double henon_map(double x , double y , int n) {
    double xi, yi;
    double a = 1.05;
    double b = 0.3;
    for (int i = 0; i < n ; i ++) {
        xi = x;  yi = y;
        x = 1.0 - a*xi*xi + yi ;
        y = b*xi ;
    }
    return x;
}
```

→ **Static Analyzer** → Absolute error

Limitations

- Only for very simple programs.
- Loops are problematic.
- Overestimation.

How to make the code sound?

Analysis at runtime:

Rewrite code to account for ranges, e.g., using *interval arithmetic* (IA).

```
struct { double lo; double up; } ival_t ;
```

```
ival_t henon_map_ia(ival_t x , ival_t y , int n) {
   ival_t xi, yi;
   ival_t a = {1.05, 1.05};
   ival_t b = {0.3, 0.3};
   for (int i = 0; i < n ; i ++) {
      xi = x;  yi = y;
      ival_t t1 = ia_mul(xi, xi);
      ival_t t2 = ia_mul(a, t1);
      …
   }
   return x;
}
```

5x slower

How to make the code sound?

Analysis at runtime:

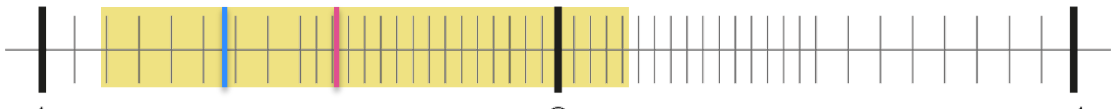
Rewrite code to account for ranges, e.g., using *interval arithmetic* (IA).

```
struct { double lo; double up; } ival_t ;
```

```
ival_t henon_map_ia(ival_t x , ival_t y , int n) {
   ival_t xi, yi;
   ival_t a = {1.05, 1.05};
   ival_t b = {0.3, 0.3};
   for (int i = 0; i < n ; i ++) {
      xi = x;  yi = y;
      ival_t t1 = ia_mul(xi, xi);
      ival_t t2 = ia_mul(a, t1);
     …
   }
   return x;
}
```

5x slower

The result is ***sound*** but it's ***not*** accurate.



Overapproximation due to the dependency problem.

Dependency problem: IA vs AA

Example:

Interval arithmetic (IA)

$\bar{a} = [0, 1]$

$\bar{b} = \bar{a} - \bar{a} = [0, 1] - [0, 1]$

Dependency problem: IA vs AA

Example:

Interval arithmetic (IA)

$\bar{a} = [0, 1]$

$\bar{b} = \bar{a} - \bar{a} = [0, 1] - [0, 1] = [-1,1]$

Correlation is lost

Dependency problem: IA vs AA

Example:

Interval arithmetic (IA)

$\bar{a} = [0, 1]$

$\bar{b} = \bar{a} - \bar{a} = [0, 1] - [0, 1] = [-1, 1]$

Correlation is lost

Affine arithmetic (AA)

$\hat{a} = 0.5 + 0.5\epsilon_1,$ where $\epsilon_i = [-1, 1]$

Dependency problem: IA vs AA

Example:

Interval arithmetic (IA)

$\bar{a} = [0, 1]$

$\bar{b} = \bar{a} - \bar{a} = [0, 1] - [0, 1] = [-1,1]$

Correlation is lost

Affine arithmetic (AA)

$\hat{a} = 0.5 + 0.5\epsilon_1$, where $\epsilon_i = [-1, 1]$

$\hat{b} = \hat{a} - \hat{a} = (0.5 + 0.5\epsilon_1) - (0.5 + 0.5\epsilon_1)$

$\hat{b} = \hat{a} - \hat{a} = 0$

Correlation is kept using error symbols

Using affine arithmetic

Using AA library

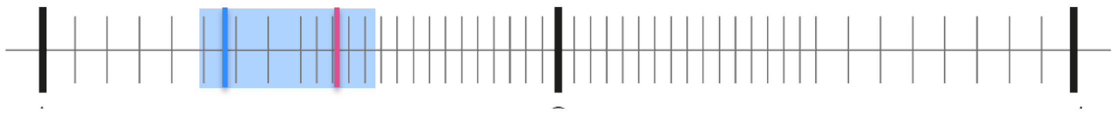
```
affine_t henon_map_aa(affine_t x , affine_t y , int m) {
   affine_t xi, yi;
   affine_t a = aa_set(1.05, 0.);
   affine_t b = aa_set(0.3, 0.);
   for (int i = 0; i < m ; i ++) {
      xi = x;  yi = y;
      affine_t t1 = aa_mul(xi, xi);
      affine_t t2 = aa_mul(a, t1);
      …
   }
   return x;
}
```

Using affine arithmetic

Using AA library

```
affine_t henon_map_aa(affine_t x , affine_t y , int m) {
   affine_t xi, yi;
   affine_t a = aa_set(1.05, 0.);
   affine_t b = aa_set(0.3, 0.);
   for (int i = 0; i < m ; i ++) {
      xi = x;  yi = y;
      affine_t t1 = aa_mul(xi, xi);
      affine_t t2 = aa_mul(a, t1);
      …
   }
   return x;
}
```

Very accurate!…

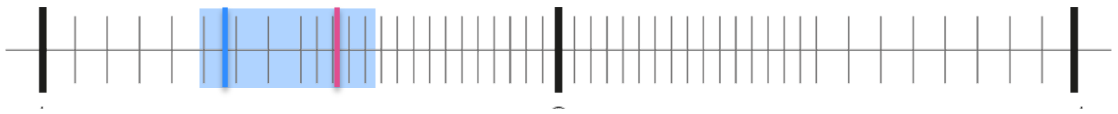


Using affine arithmetic

Using AA library

```
affine_t henon_map_aa(affine_t x , affine_t y , int m) {
   affine_t xi, yi;
   affine_t a = aa_set(1.05, 0.);
   affine_t b = aa_set(0.3, 0.);
   for (int i = 0; i < m ; i ++) {
      xi = x;  yi = y;
      affine_t t1 = aa_mul(xi, xi);
      affine_t t2 = aa_mul(a, t1);
      …
   }
   return x;
}
```

*Very accurate!… but **10K** slower*

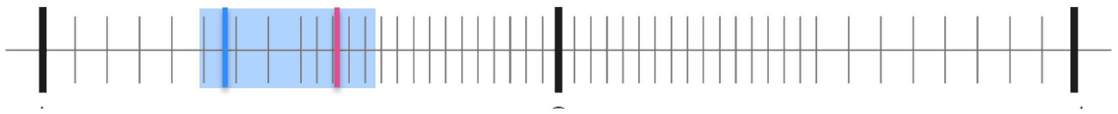


Using affine arithmetic

Using AA library

```
affine_t henon_map_aa(affine_t x , affine_t y , int m) {
   affine_t xi, yi;
   affine_t a = aa_set(1.05, 0.);
   affine_t b = aa_set(0.3, 0.);
   for (int i = 0; i < m ; i ++) {
      xi = x;  yi = y;
      affine_t t1 = aa_mul(xi, xi);
      affine_t t2 = aa_mul(a, t1);
      …
   }
   return x;
}
```

*Very accurate!… but **10K** slower*



Every operation introduces a new error symbol.

After *m* iterations:

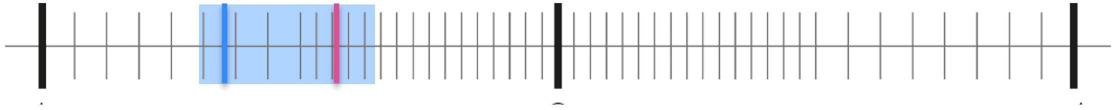
$$x = x_0 + x_1\epsilon_1 + x_2\epsilon_2 + x_3\epsilon_3 + x_4\epsilon_4 + \cdots + x_n\epsilon_n$$

Using affine arithmetic

Using AA library

```
affine_t henon_map_aa(affine_t x , affine_t y , int m) {
   affine_t xi, yi;
   affine_t a = aa_set(1.05, 0.);
   affine_t b = aa_set(0.3, 0.);
   for (int i = 0; i < m ; i ++) {
      xi = x;  yi = y;
      affine_t t1 = aa_mul(xi, xi);
      affine_t t2 = aa_mul(a, t1);
      …
   }
   return x;
}
```

*Very accurate!… but **10K** slower*



Every operation introduces a new error symbol.

After *m* iterations:

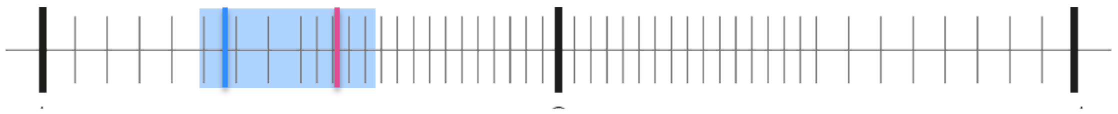
$$x = x_0 + x_1\epsilon_1 + x_2\epsilon_2 + x_3\epsilon_3 + x_4\epsilon_4 + \cdots + x_n\epsilon_n$$

Using affine arithmetic

Using AA library

```
affine_t henon_map_aa(affine_t x , affine_t y , int m) {
   affine_t xi, yi;
   affine_t a = aa_set(1.05, 0.);
   affine_t b = aa_set(0.3, 0.);
   for (int i = 0; i < m ; i ++) {
      xi = x;  yi = y;
      affine_t t1 = aa_mul(xi, xi);
      affine_t t2 = aa_mul(a, t1);
      …
   }
   return x;
}
```

*Very accurate!… but **10K** slower*



Every operation introduces a new error symbol.

After *m* iterations:

$$x = x_0 + x_1\epsilon_1 + x_2\epsilon_2 + x_{n+1}\epsilon_{n+1}$$

Accuracy vs performance tradeoff

Challenges to achieve soundness

- Get the AA or IA code with little effort
- Decide what symbols to keep in AA
- How to achieve high performance
- What if input code has SIMD intrinsics

```
affine_t henon_map_aa(affine_t x ,
                      affine_t y ,
                      int m) {
  affine_t xi, yi;
  affine_t a = aa_set(1.05, 0.);
  affine_t b = aa_set(0.3, 0.);
  for (int i = 0; i < m ; i ++) {
    xi = x;  yi = y;
    affine_t t1 = aa_mul(xi, xi);
    affine_t t2 = aa_mul(a, t1);
    ...
  }
  return x;
}
```

$$x = x_0 + x_1\epsilon_1 + x_2\epsilon_2 + \boxed{x_3\epsilon_3 + \cdots + x_n\epsilon_n}$$

$$x = x_0 + x_1\epsilon_1 + x_2\epsilon_2 + \boxed{x_{n+1}\epsilon_{n+1}}$$

```
__m256d _mm256_add_pd (__m256d a, __m256d b)
```

Synopsis

```
__m256d _mm256_add_pd (__m256d a, __m256d b)
#include <immintrin.h>
Instruction: vaddpd ymm, ymm, ymm
CPUID Flags: AVX
```

SIMD

Description

Add packed double-precision (64-bit) floating-point elements in a and b,

A compiler for sound floating-point computations

Input: Numerical C function (with SIMD)

```
double henon(double x, ...) {
    ...
    y = b * xi;
    ...
}
```

Target precision

Single, Double, Double-double

→ **Accuracy improvement**

Analysis to decide on symbols to keep

→ **Code transformation**

Decl (type a, foo, ..)

Stmt (for, if, ..)

Expr (a + b, sqrt, ..)

→ ***Output:*** Equivalent sound function using **AA** or **IA** (with SIMD)

```
f64a henon(f64a x, ...) {
    ...
    y = aa_mul_f64(b, xi);
    ...
}
```

- Result is sound.
- Fast implementation.
- Optionally using SIMD.

A compiler for sound floating-point computations

Input: Numerical C function (with SIMD)

```
double henon(double x, ...) {
    ...
    y = b * xi;
    ...
}
```

→ **Accuracy improvement** → **Code transformation** →

Output: Equivalent sound function using **AA** or **IA** (with SIMD)

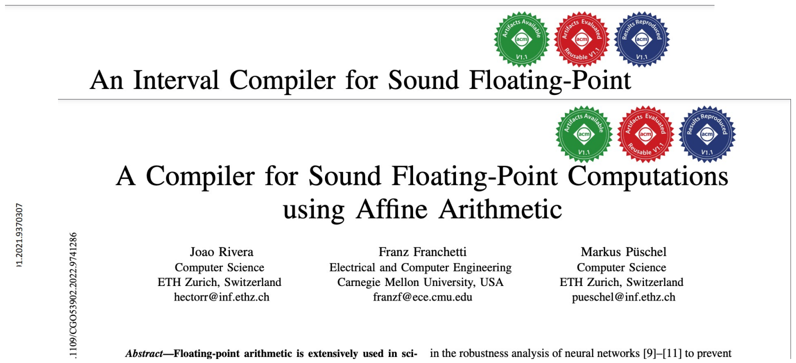
```
f64a henon(f64a x, ...) {
    ...
    y = aa_mul_f64(b, xi);
    ...
}
```

Target precision
Single, Double, Double-double

Analysis to decide on symbols to keep

Decl (type a, foo, ..)
Stmt (for, if, ..)
Expr (a + b, sqrt, ..)

- Result is sound.
- Fast implementation.
- Optionally using SIMD.


An Interval Compiler for Sound Floating-Point

A Compiler for Sound Floating-Point Computations using Affine Arithmetic

Joao Rivera, Computer Science, ETH Zurich, Switzerland, hectorr@inf.ethz.ch
Franz Franchetti, Electrical and Computer Engineering, Carnegie Mellon University, USA, franzf@ece.cmu.edu
Markus Püschel, Computer Science, ETH Zurich, Switzerland, pueschel@inf.ethz.ch

Abstract—Floating-point arithmetic is extensively used in sci- … in the robustness analysis of neural networks [9]–[11] to prevent

[1] *An interval compiler for sound floating-point computations*. CGO, 2021.
[2] *A compiler for sound floating-point computations using Affine Arithmetic*. CGO, 2022.

Transformations

Example

Original

```
double foo(double a, double b) {
    double c;
    c = a * b;
    c = c + 0.1;

    if (c > a) {
        ...
    }
    return c;
}
```

Generated

```
f64a foo(f64a a, f64a b) {
    f64a c;
    c = aa_mul_f64(a, b);
    f64a t1 = aa_set_f64(0.1, 1.38...e-17);
    c = aa_add_f64(c, t1);

    if (aa_cmpgt_f64(c, a)) {
        ...
    }
    return c;
}
```

Decl node
FP type → Affine type

Expr node
Sound operations

Transformations

Example

Original

```
double foo(double a, double b) {
    double c;
    c = a * b;
    c = c + 0.1;

    if (c > a) {
        ...
    }
    return c;
}
```

Generated

```
f64a foo(f64a a, f64a b) {
    f64a c;
    c = aa_mul_f64(a, b);
    f64a t1 = aa_set_f64(0.1, 1.38...e-17);
    c = aa_add_f64(c, t1);

    if (aa_cmpgt_f64(c, a)) {
        ...
    }
    return c;
}
```

Decl node

FP type → Affine type

Expr node

Sound operations

Sound constants

Limiting the number of symbols

Decide what symbols to keep to preserve accuracy

- at **runtime** based on a policy.
- at **static time** using DAG analysis.

$$x = x_0 + x_1\epsilon_1 + x_2\epsilon_2 + \boxed{x_3\epsilon_3 + \cdots + x_n\epsilon_n}$$

$$\Downarrow$$

$$x = x_0 + x_1\epsilon_1 + x_2\epsilon_2 + \boxed{x_{n+1}\epsilon_{n+1}}$$

Limiting the number of symbols

Decide what symbols to keep to preserve accuracy

- at **runtime** based on a policy.
- at **static time** using DAG analysis.

$$x = x_0 + x_1\epsilon_1 + x_2\epsilon_2 + x_3\epsilon_3 + \cdots + x_n\epsilon_n$$

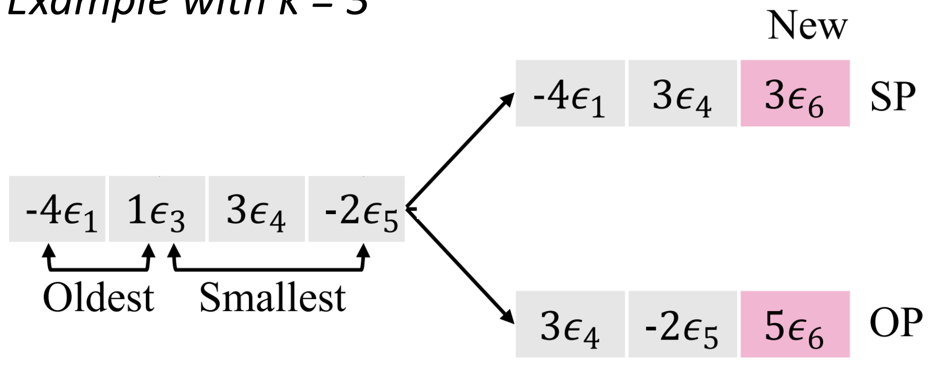
$$\Downarrow$$

$$x = x_0 + x_1\epsilon_1 + x_2\epsilon_2 + x_{n+1}\epsilon_{n+1}$$

Decide at runtime using fusion policy:

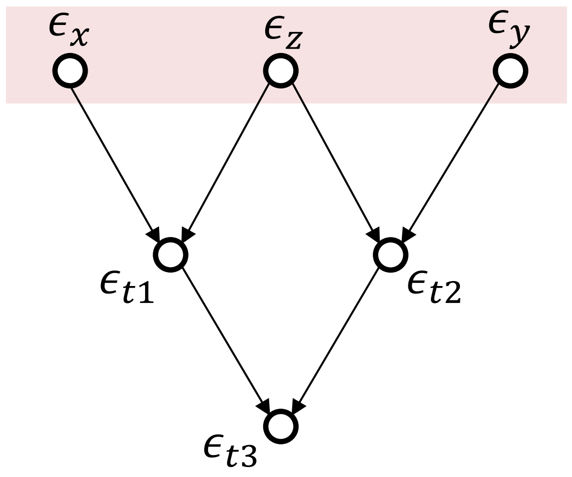
- *Random (RP)*
- *Oldest symbol (OP)*
- *Symbols with **smallest** absolute values (SP)*
- Below mean (MP)
- …

Example with k = 3



Preserving accuracy at static time: Using DAG analysis

$$x \cdot z - y \cdot z$$

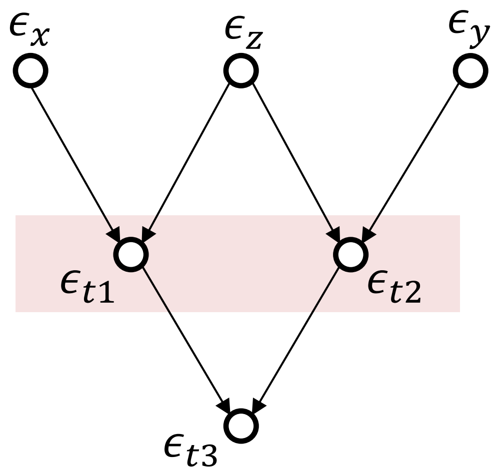


Example

$$x = 1 + \epsilon_x$$
$$y = 1 + \epsilon_y$$
$$\mathrm{z} = 1 + \epsilon_z$$

Preserving accuracy at static time: Using DAG analysis

$$x \cdot z - y \cdot z$$



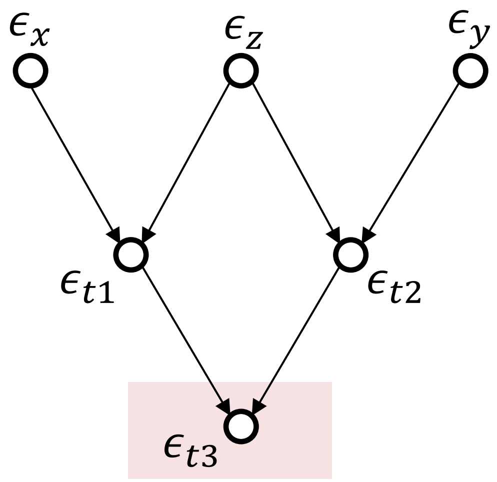
Example

$$x = 1 + \epsilon_x$$
$$y = 1 + \epsilon_y$$
$$z = 1 + \epsilon_z$$

$$t_1 = x \cdot z = 1 + \epsilon_x + \epsilon_z + \epsilon_{t1}$$
$$t_2 = y \cdot z = 1 + \epsilon_y + \epsilon_z + \epsilon_{t2}$$

Preserving accuracy at static time: Using DAG analysis

$x \cdot z - y \cdot z$



Example

$x = 1 + \epsilon_x$
$y = 1 + \epsilon_y$
$z = 1 + \epsilon_z$

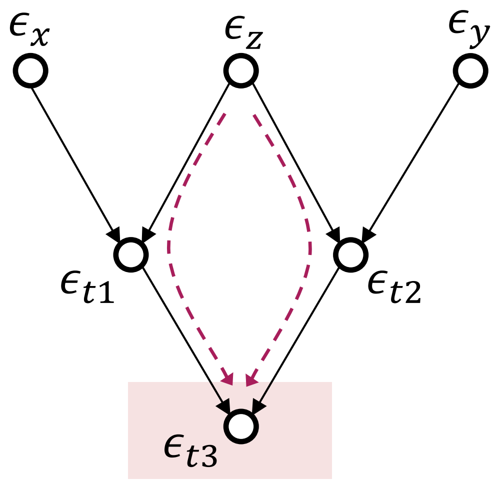
$t_1 = x \cdot z = 1 + \epsilon_x + \epsilon_z + \epsilon_{t1}$
$t_2 = y \cdot z = 1 + \epsilon_y + \epsilon_z + \epsilon_{t2}$

$t_3 = t_1 - t_2 = \epsilon_x + \epsilon_{t1} - \epsilon_y - \epsilon_{t2}$

ϵ_z cancels out

Preserving accuracy at static time: Using DAG analysis

$$x \cdot z - y \cdot z$$



ϵ_z should be kept to allow reuse.

Example

$$x = 1 + \epsilon_x$$
$$y = 1 + \epsilon_y$$
$$\mathrm{z} = 1 + \epsilon_z$$

$$t_1 = x \cdot z = 1 + \epsilon_x + \epsilon_z + \epsilon_{t1}$$
$$t_2 = y \cdot z = 1 + \epsilon_y + \epsilon_z + \epsilon_{t2}$$

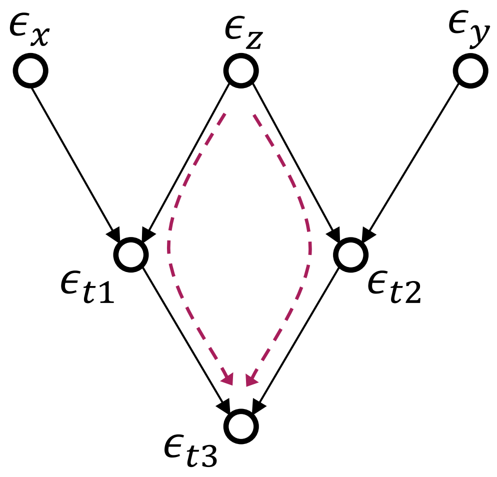
$$t_3 = t_1 - t_2 = \epsilon_x + \epsilon_{t1} - \epsilon_y - \epsilon_{t2}$$

ϵ_z cancels out

Reuse:
When a symbol arrives to a node from two different paths.
Symbols that are **reused** are likely to cancel out.

Preserving accuracy at static time: Using DAG analysis

$x \cdot z - y \cdot z$



ϵ_z should be kept to allow reuse.

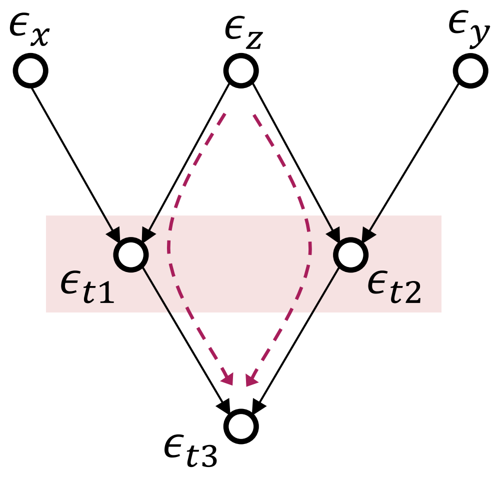
Example with <u>k = 2</u>

$x = 1 + \epsilon_x$
$y = 1 + \epsilon_y$
$\mathrm{z} = 1 + \epsilon_z$

$t_1 = x \cdot z = 1 + \epsilon_x + \epsilon_z + \epsilon_{t1}$ *Keep ϵ_z, i.e., fuse ϵ_x and ϵ_{t1}*
$t_2 = y \cdot z = 1 + \epsilon_y + \epsilon_z + \epsilon_{t2}$

Preserving accuracy at static time: Using DAG analysis

$x \cdot z - y \cdot z$



ϵ_z should be kept to allow reuse.

Reuse

When a symbol arrives to a node from two different paths.

Problem

Find the symbols to keep that maximizes reuse given that each node can keep at most k symbols.

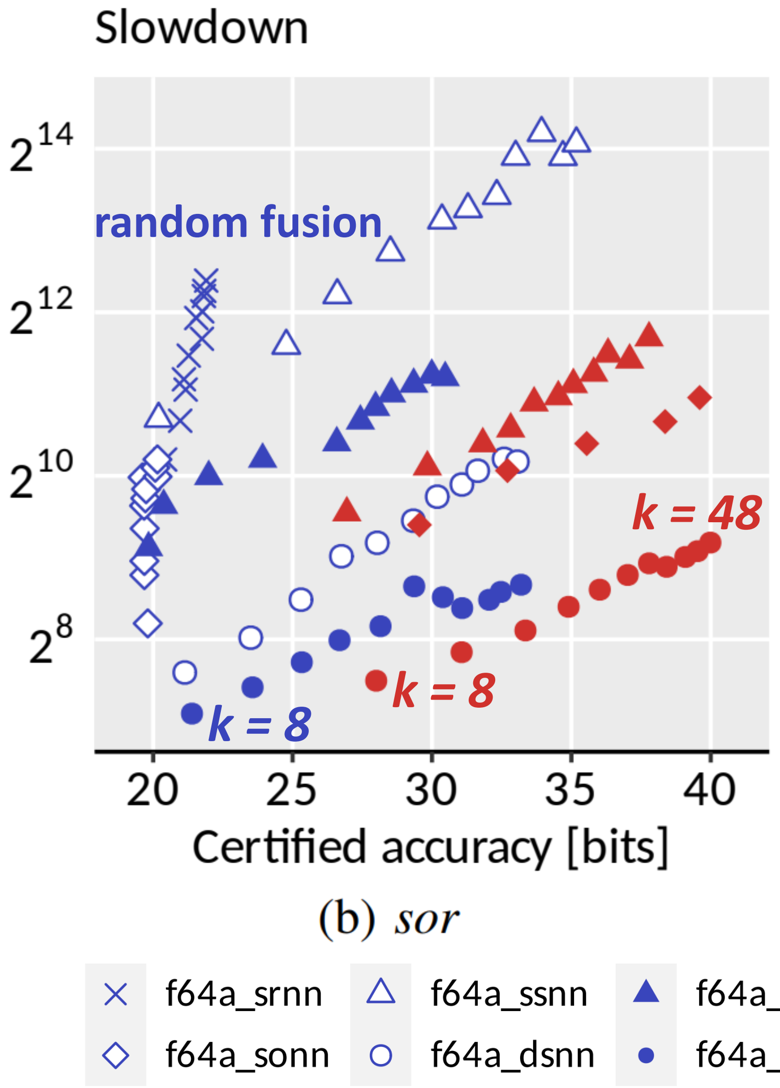
Modeled as an ILP program

$$\begin{aligned} \text{maximize} \quad & \sum_{s \in V} [\rho(s), \rho(s), \ldots, \rho(s)] \mathbf{q}_s \\ \text{subject to} \quad & \sum_{s \in V} \mathbf{p}_s \leq [k-1, k-1, \ldots, k-1]^{\mathrm{T}}, \\ & \mathbf{p}_s = \mathbf{R}_s \circ \mathbf{q}_s \quad \text{for all } s \in V, \end{aligned}$$

GUROBI OPTIMIZATION

Evaluation

Setup
Intel Xeon E-2176M @2.7GHz
Ubuntu 18.04, gcc 7.5

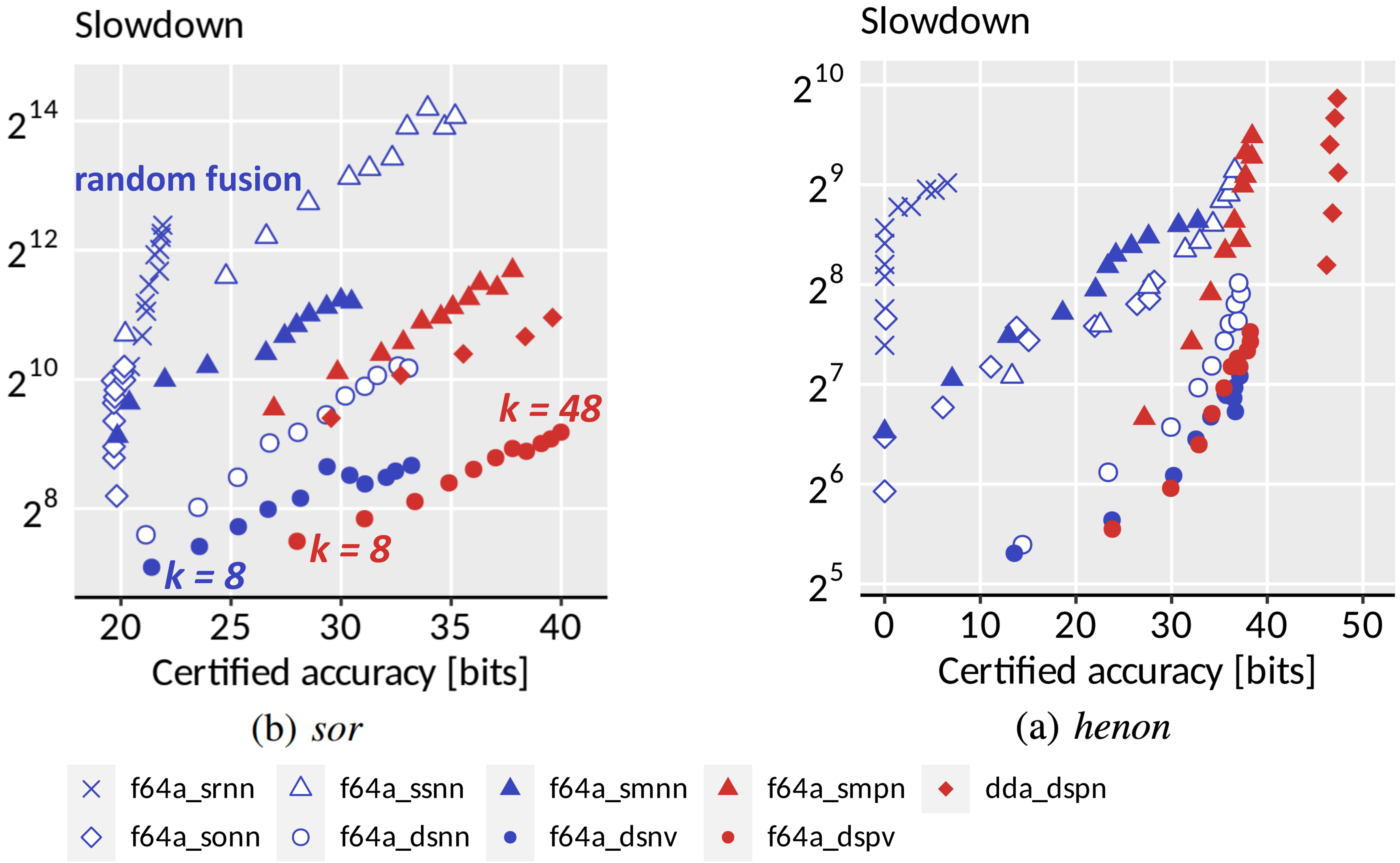


(b) *sor*

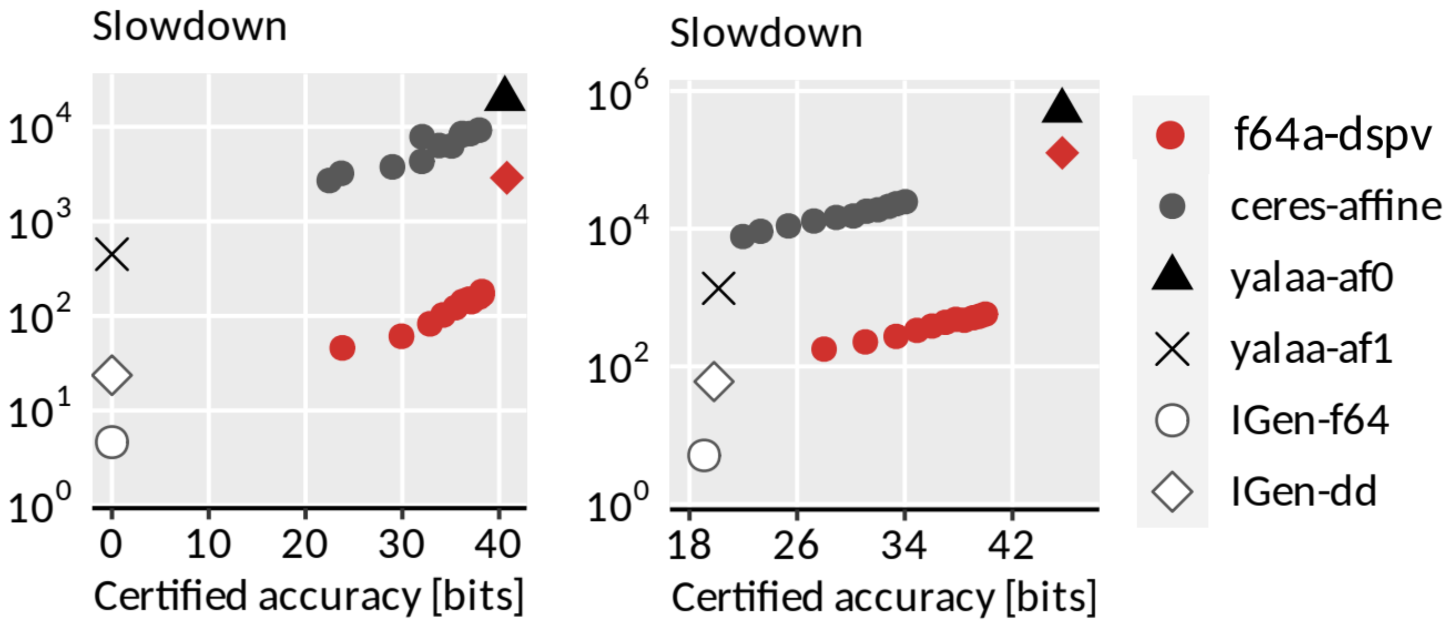
f64a_dspv:
smallest policy, **with DAG analysis** and vectorized.

DAG analysis:
4-8 bits improvement with 25% overhead.

Setup
Intel Xeon E-2176M @2.7GHz
Ubuntu 18.04, gcc 7.5



Comparison with IA and AA libraries



(a) *henon* (b) *sor*

f64a_dspv:
30x-70x faster than ceres-affine.
10x-36x slower than IGen (IA).

Compiler for Sound Floating-Point using AA

file.c

DAG Analysis

Transform node

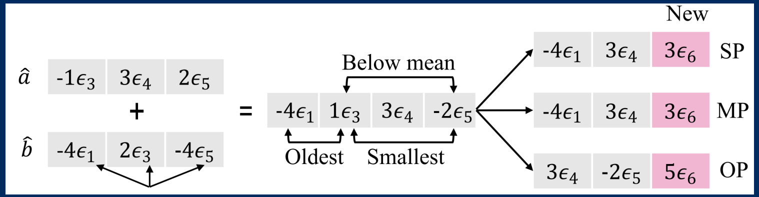
Decl, Stmt, Expr

safegen_file.c

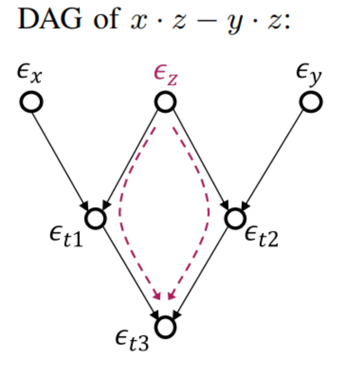
(optionally with SIMD)

```
f64a foo(f64a a, …) {
 …
 c = aa_add_f64(a, b);
 …
}
```

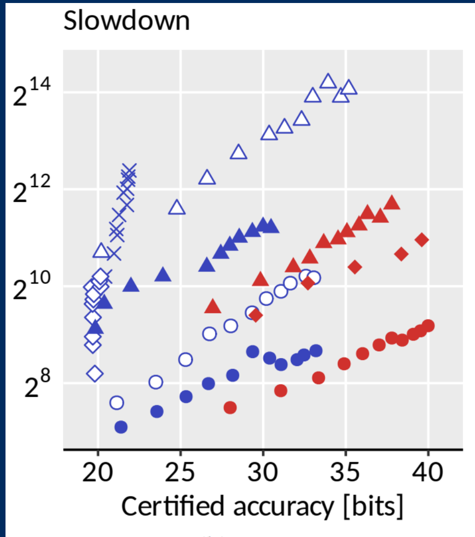
Fusion Policies



DAG Analysis



Evaluation



(b) *sor*

30x-70x faster libraries for AA
More accurate than IA

Prioritization improves accuracy by **4-8 bits**

[1] *An interval compiler for sound floating-point computations*. CGO, 2021.
[2] *A compiler for sound floating-point computations using Affine Arithmetic*. CGO, 2022.