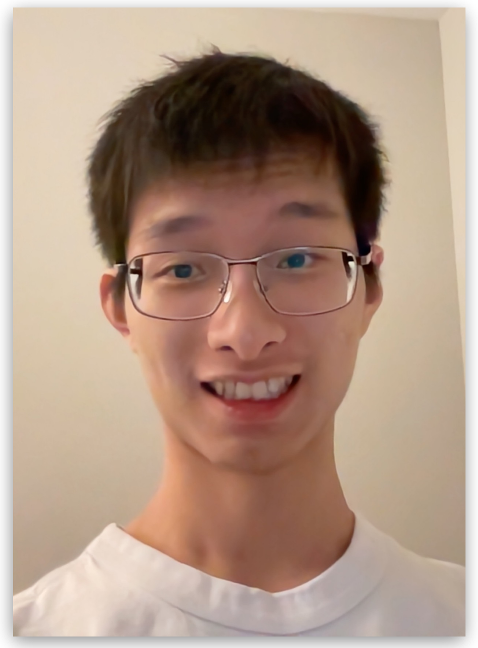
Thinking Fast and Correct: Automated Rewriting of Numerical Code through Compiler Augmentation



Siyuan Brant Qian[1] Vimarsh Sathia[1] Jan Hückelheim[2] Paul Hovland[2] William S. Moses[1]

siyuanq4@illinois.edu

[1] University of Illinois Urbana-Champaign, USA
[2] Argonne National Laboratory, USA

CGO 2026
Feb. 3, 2026

Optimizing numerical code is challenging at three levels:
Context, **Scope**, and **Scale**.
Profiling + **Compiler Optimization** are the key.

Context: Where You Change Matters

```
double sum = 0.0;
for (double x : xs) {
    double val = sigmoid(exp(x));
    sum += val;
}
```

Context: Where You Change Matters

```
double sum = 0.0;
for (double x : xs) {
    double val = sigmoid(exp(x));
    sum += val;
}
```

```
double sum = 0.0;
for (double x : xs) {
    float val = sigmoidf(expf(x));
    sum += val;
}
```

Lower Activation:
1.6× speedup
`error +≈ 1e-12 (safe!)`

Context: Where You Change Matters

```
double sum = 0.0;
for (double x : xs) {
    double val = sigmoid(exp(x));
    sum += val;
}
```

```
double sum = 0.0;
for (double x : xs) {
    float val = sigmoidf(expf(x));
    sum += val;
}
```

Lower Activation:
1.6× speedup
error +≈ 1e-12 (safe!)

```
float sum = 0.0f;
for (double x : xs) {
    float val = sigmoidf(expf(x));
    sum += val;
}
```

Lower Reduction:
error +≈ 1e-4 (danger!)

Context: Where You Change Matters

```
double sum = 0.0;
for (double x : xs) {
    double val = sigmoid(exp(x));
    sum += val;
}
```

Profiling just 10 iterations:
Reduction is $10^4\times$ more sensitive*!

```
double sum = 0.0;
for (double x : xs) {
    float val = sigmoidf(expf(x));
    sum += val;
}
```

Lower Activation:
1.6× speedup
`error +≈ 1e-12 (safe!)`

```
float sum = 0.0f;
for (double x : xs) {
    float val = sigmoidf(expf(x));
    sum += val;
}
```

Lower Reduction:
`error +≈ 1e-4 (danger!)`

* ADAPT metric (Menon et al. '18).

Context: *Structured Critical Value Hypothesis*

```
double sum = 0.0;
for (double x : xs) {
    double val = sigmoid(exp(x));
    sum += val;
}
```

Profiling just 10 iterations:
Reduction is $10^4 \times$ more sensitive*!

```
double sum = 0.0;
for (double x : xs) {
    float val = sigmoidf(expf(x));
    sum += val;
}
```

```
float sum = 0.0f;
for (double x : xs) {
    float val = sigmoidf(expf(x));
    sum += val;
}
```

Accuracy-critical values are often ***structural*** (e.g., reduction, cancellation, one-hot, thresholds). A **small surrogate profile** reveals key information.

Scope: Precision Isn’t the Only Knob

FP32 (Original Order) ⚡ Fast ❌ Wrong

```
(1e8 + 1) - 1e8 → 1.0000000e8 - 1e8 → 0
```

+1 rounded away

Scope: Precision Isn’t the Only Knob

FP32 (Original Order) ⚡ Fast ❌ Wrong

`(1e8 + 1) - 1e8 → 1.0000000e8 - 1e8 → 0`

+1 rounded away

FP64 (Same Order) 🐢 Costly ✅ Correct

`(1e8 + 1) - 1e8 → 1.00000001e8 - 1e8 → 1`

Use FP64 (Tradeoff!)

Scope: Precision Isn’t the Only Knob

FP32 (Original Order) ⚡ Fast ❌ Wrong

`(1e8 + 1) - 1e8 → 1.0000000e8 - 1e8 → 0`

+1 rounded away

FP64 (Same Order) 🐢 Costly ✅ Correct

`(1e8 + 1) - 1e8 → 1.00000001e8 - 1e8 → 1`

Use FP64 (Tradeoff!)

FP32 (Reassociated) ⚡ Fast ✅ Correct

`(1e8 - 1e8) + 1 → 0 + 1 → 1`

Algebraic Rewrite (circumvents tradeoff but requires context)

Scale: The Math Is in the Mess

```
double foo(double x) {
  double y = 0.0;
  if (x > 0.0)
    y = pow(x, 3);
  return y;
}
```

Math behind control flow,
memory I/O,
function calls, …;
existing source/binary-level
tools fail.

Scale: The Math Is in the Mess

```
double foo(double x) {
  double y = 0.0;
  if (x > 0.0)
    y = pow(x, 3);
  return y;
}
```

Math behind control flow,
memory I/O,
function calls, ...;
existing source/binary-level
tools fail.

A production compiler makes math
explicit through optimizations
(`SimplifyCFG`, `mem2reg`, inlining, ...)

Scale: The Math Is in the Mess

```c
double foo(double x) {
  double y = 0.0;
  if (x > 0.0)
    y = pow(x, 3);
  return y;
}
```

Math behind control flow, memory I/O, function calls, …; existing source/binary-level tools fail.

```llvm
define double @foo(double %x) {
entry:
  %p = call double @powi(
    double %x, i32 3)
  %r = call double @max(
    double 0.0, double %p)
  ret double %r
}
```

A production compiler makes math *explicit* through optimizations (`SimplifyCFG`, `mem2reg`, inlining, …)

Scale: The Math Is in the Mess

```
double foo(double x) {
  double y = 0.0;
  if (x > 0.0)
    y = pow(x, 3);
  return y;
}
```

Math behind control flow, memory I/O, function calls, …; existing source/binary-level tools fail.

A production compiler makes math *explicit* through optimizations (`SimplifyCFG`, `mem2reg`, inlining, …)

```
define double @foo(double %x) {
entry:
  %p = call double @powi(
    double %x, i32 3)
  %r = call double @max(
    double 0.0, double %p)
  ret double %r
}
```

Aha, caller wants $\max(0, x^3)$!

Can we **automate** numerical rewriting techniques in compilers?

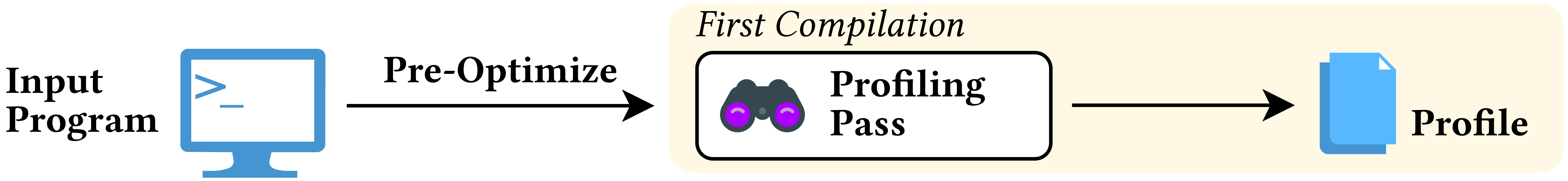
How much performance are we **leaving on the table** due to *suboptimal* choices of precision and expressions?

Our Answer — Poseidon

- The first framework that automates numerical rewrites within a production compiler

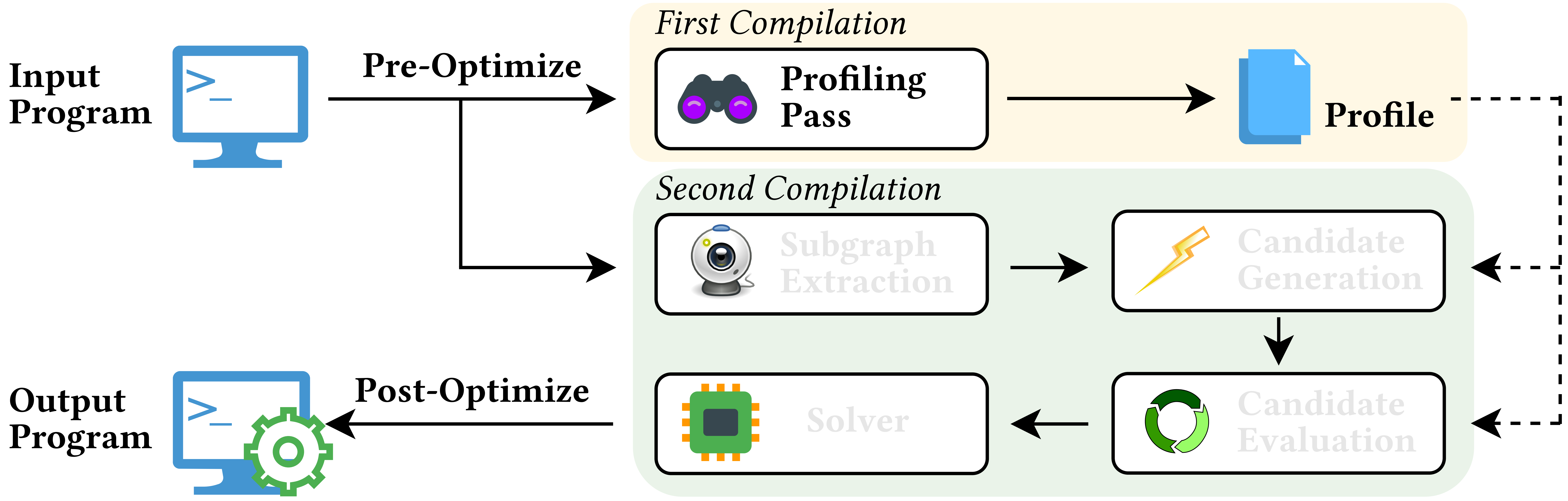
Our Answer — Poseidon

- The first framework that automates numerical rewrites within a production compiler
- **First Compile:** Instrument user program to collect numerical context



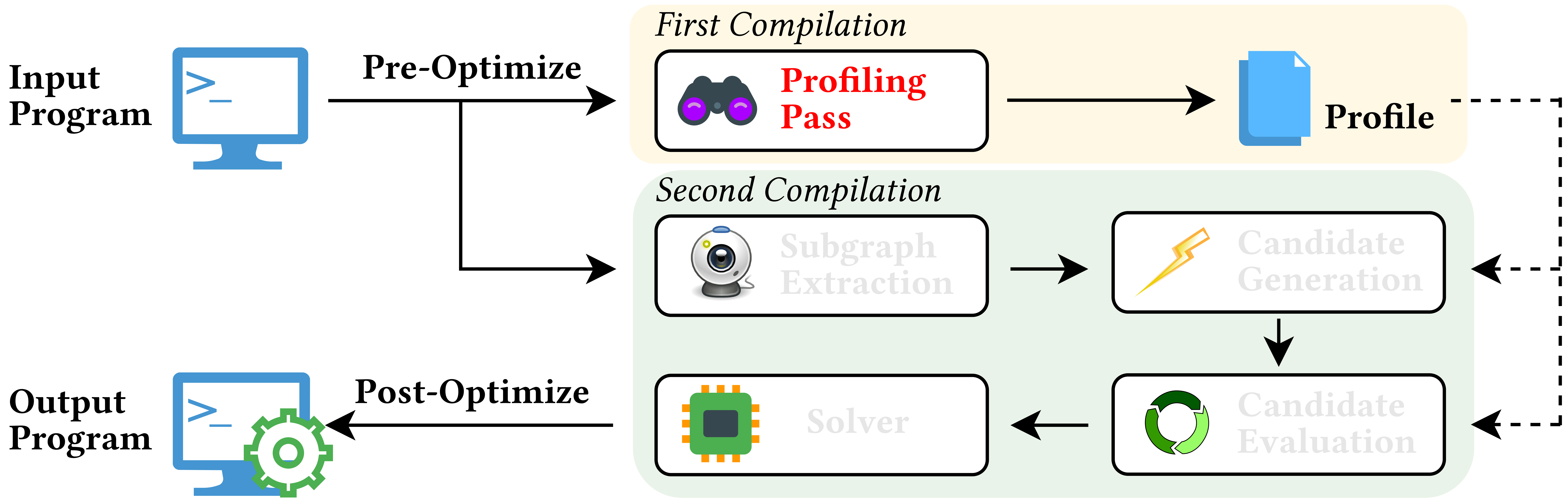
Our Answer — Poseidon

- The first framework that automates numerical rewrites within a production compiler
- **Second Compile:** Perform *full-application scale* numerical rewrites



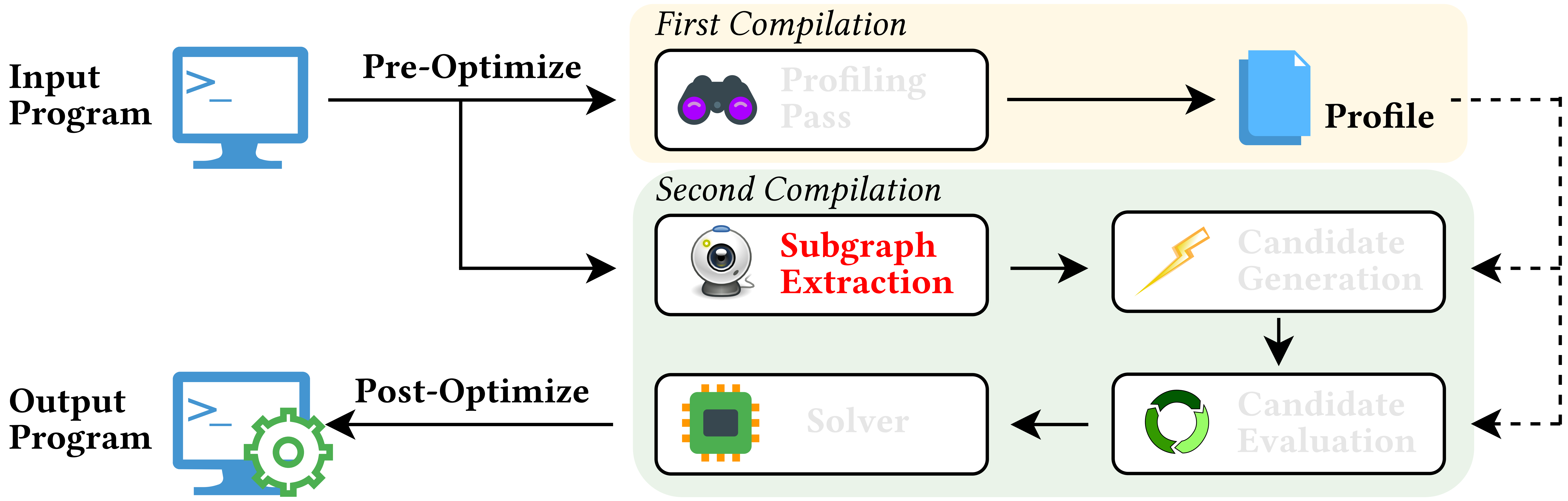
Profiling Pass

- Perform **pre-optimizations** to expose underlying math
- Emit profiles: execution counts, running sums of values/gradients per instruction



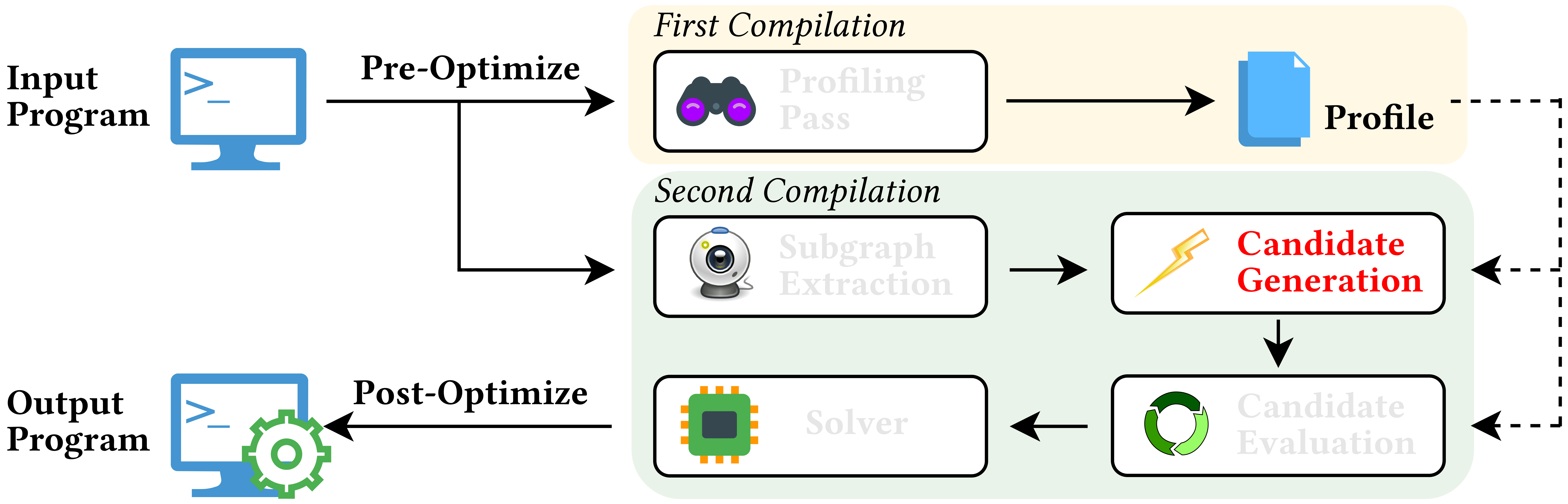
Subgraph Extraction

- Similarly, perform **pre-optimizations** to expose underlying math
- **Rewrite Regions**: floating-point def–use subgraph (linear-time flood fill)



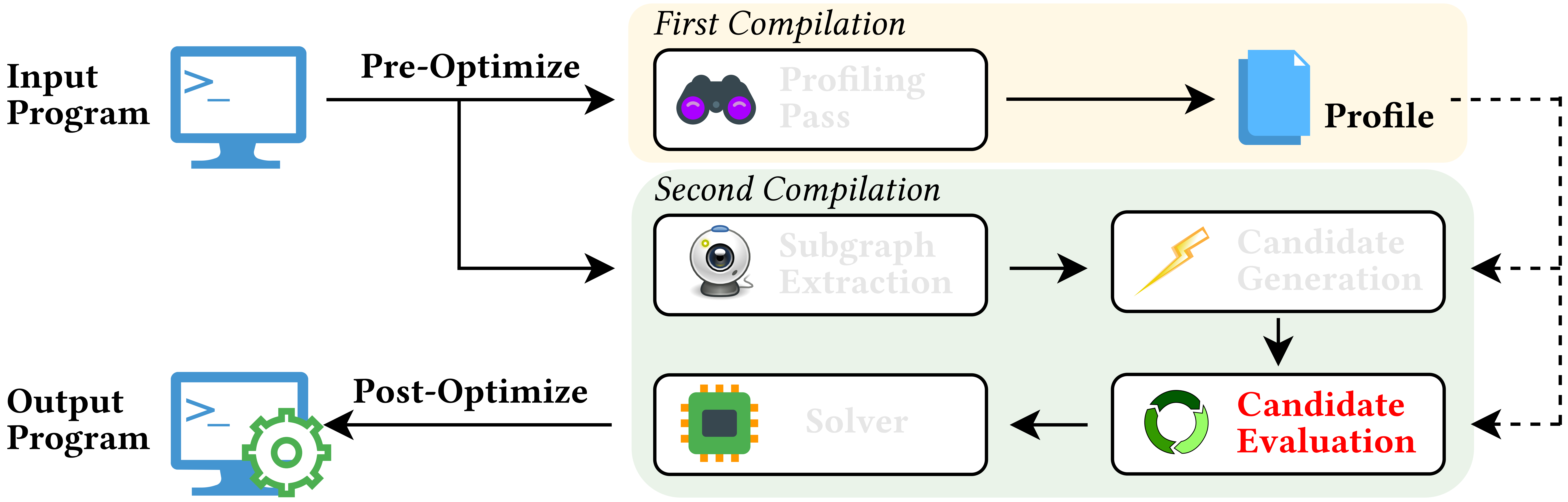
Candidate Generation

- Algebraic rewrites from external tools (e.g., Herbie)
- Profile-guided precision changes
- Fully extensible!



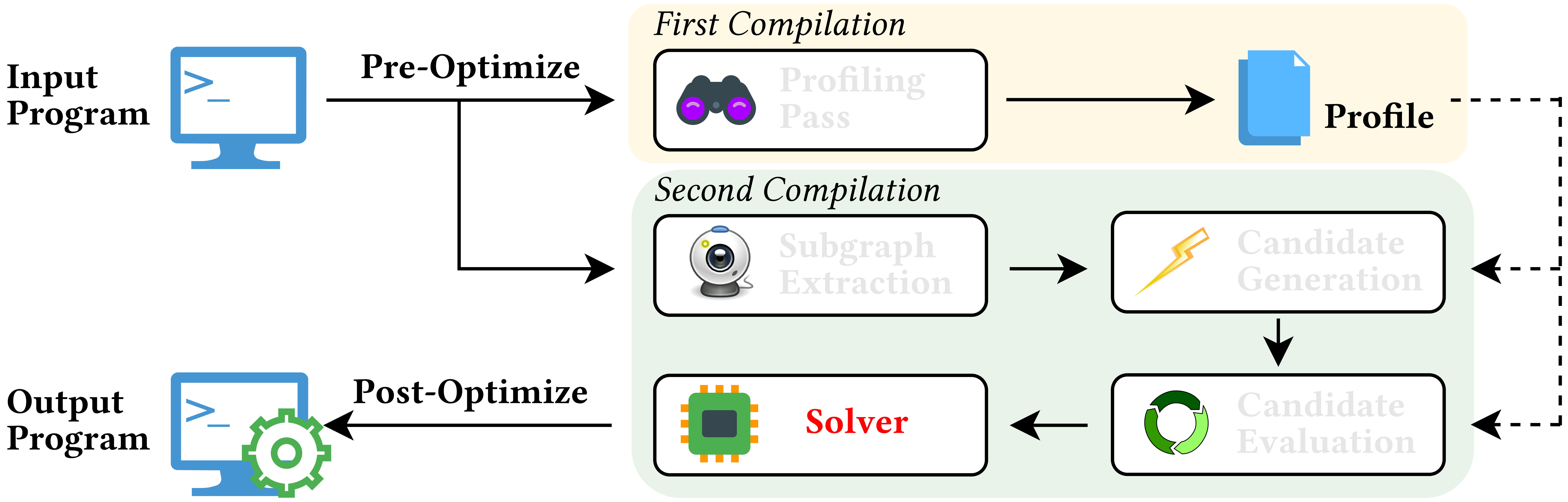
Candidate Evaluation

- Goal: **predict** candidate's performance/accuracy impact using the profile
- Cost: Sum over instructions of *(cost × execution count)*
- Accuracy: Sum over instructions of *(local error × global sensitivity)*



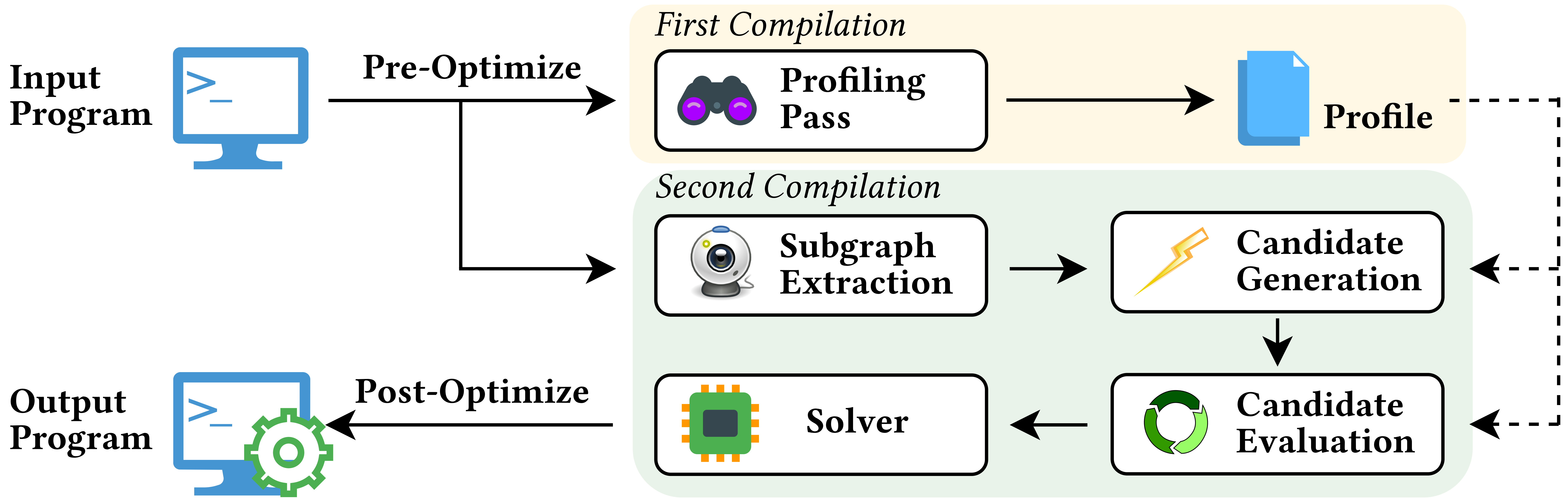
Solver (Global Selection of Rewrites)

- Objective: **minimize predicted global error** given a set of cost-error pairs
- Similar to *knapsack*: dynamic programming builds a **cost–error frontier**



Can we **automate** numerical rewriting techniques in compilers?

How much performance are we **leaving on the table** due to *suboptimal* choices of precision and expressions?

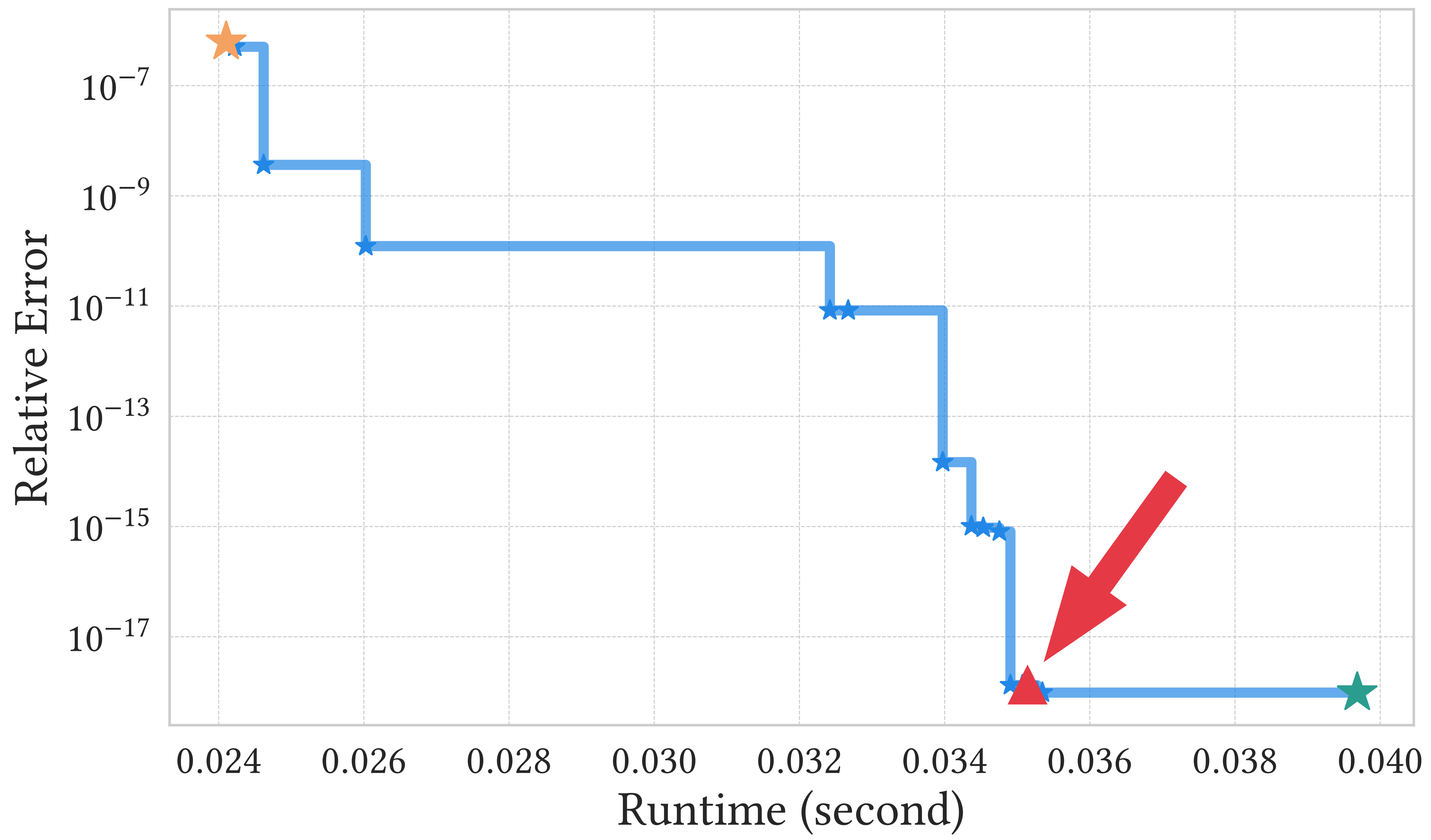


Evaluation

- Structured Critical Value Hypothesis: ***small surrogate profiles*** reveal numerical structure
- Coverage (FPBench): Accuracy improvements on **58%** of benchmarks;
  Maximum speedup of **1.82×** (error < 1e-6)
- Quaternion Differentiator: **1.46×** speedup (error < 1e-6)
- LULESH: **Bitwise-identical to MPFR-512** in FP64; no substantial slowdown

Evaluation: Quaternion Differentiator

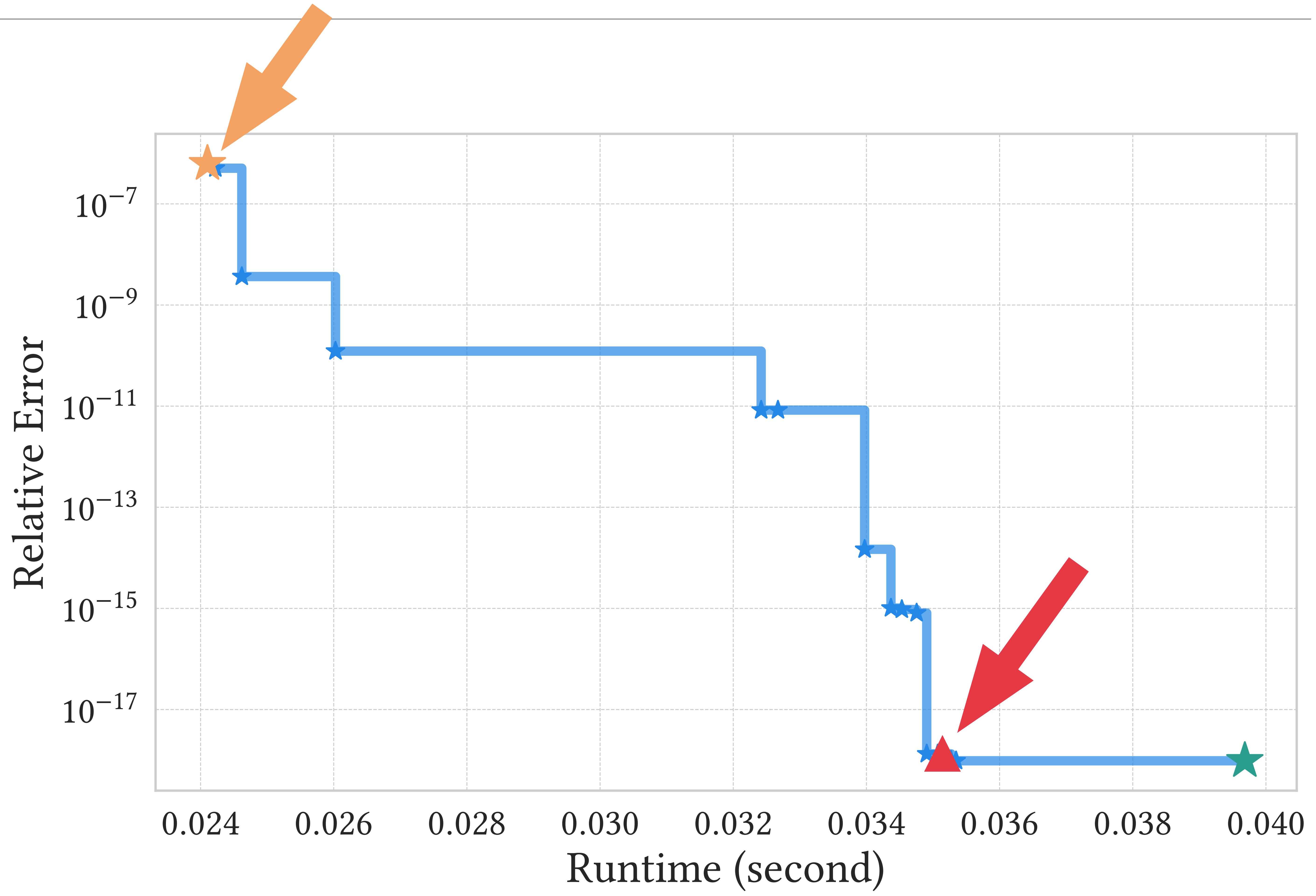
- Given the **original** program, Poseidon produces tradeoffs



Lower is more accurate; left is faster

Evaluation: Quaternion Differentiator

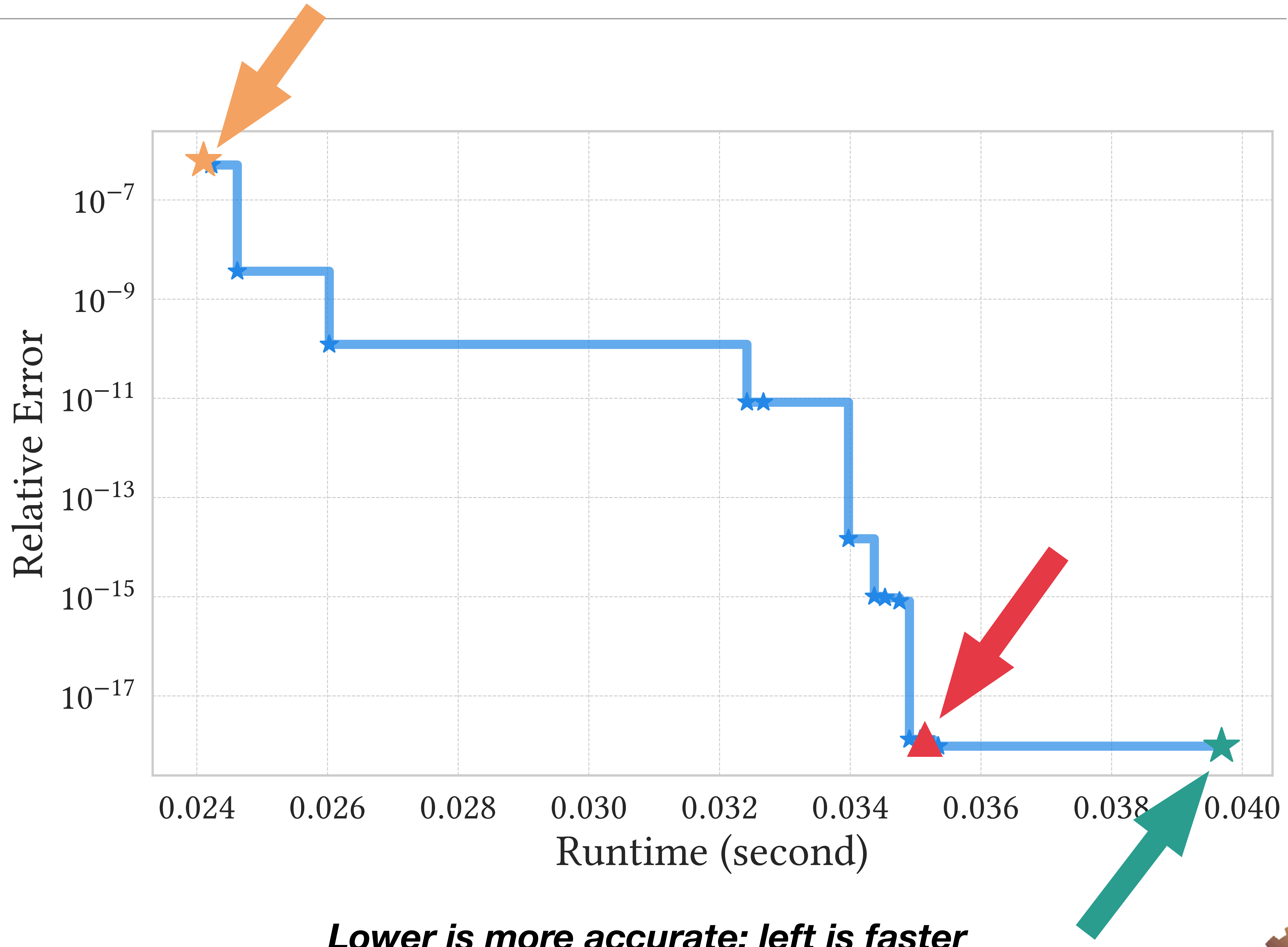
- Given the **original** program, Poseidon produces tradeoffs
- Trade accuracy for performance
  - **1.46×** with error of 6e-7



Lower is more accurate; left is faster

Evaluation: Quaternion Differentiator

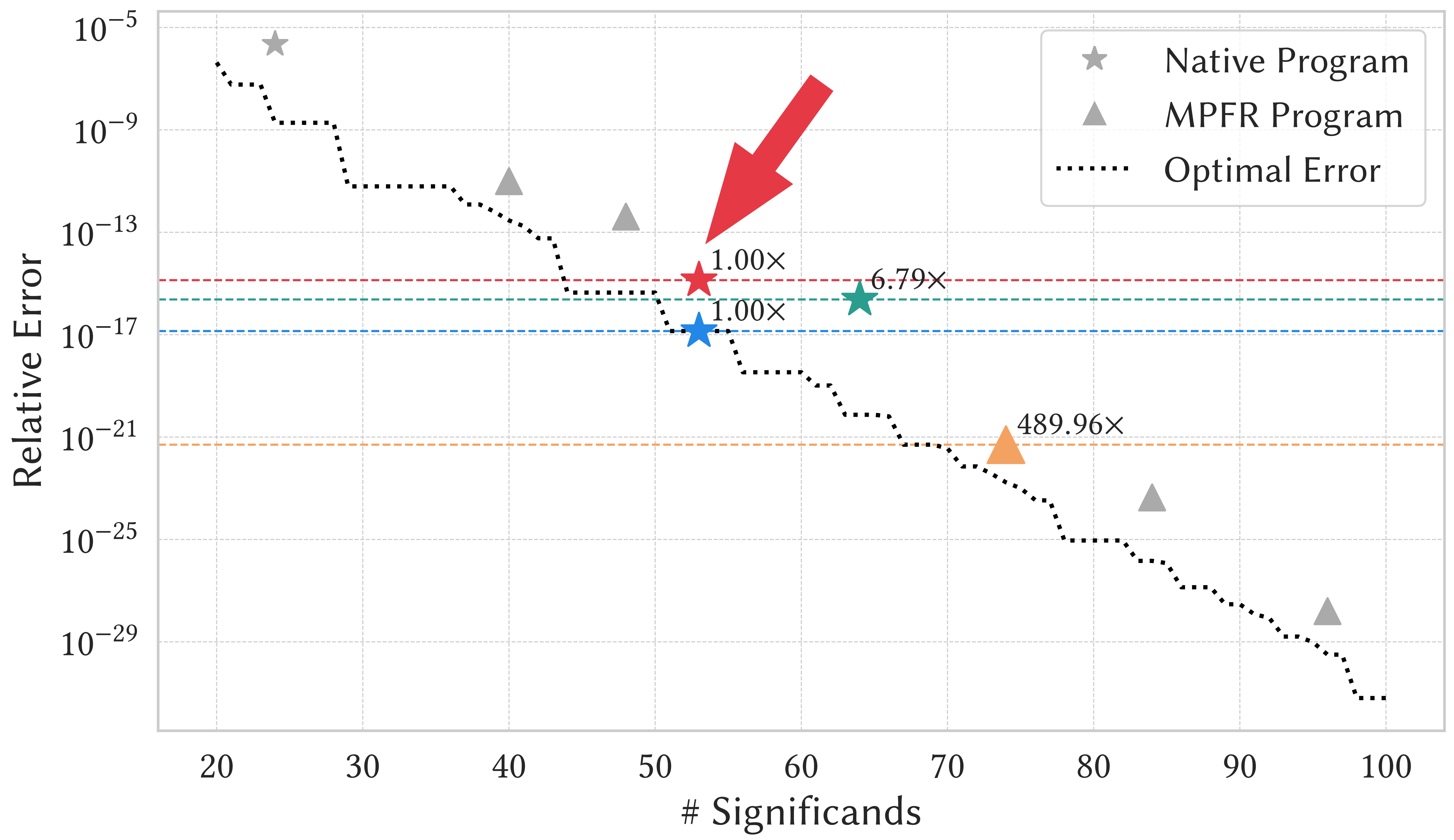
- Given the **original** program, Poseidon produces tradeoffs
- Trade accuracy for performance
    - **1.46×** with error of 6e-7
- Trade performance for accuracy
    - Error reduced by **27%** with 1.13× more compute time



Lower is more accurate; left is faster

Evaluation: LULESH

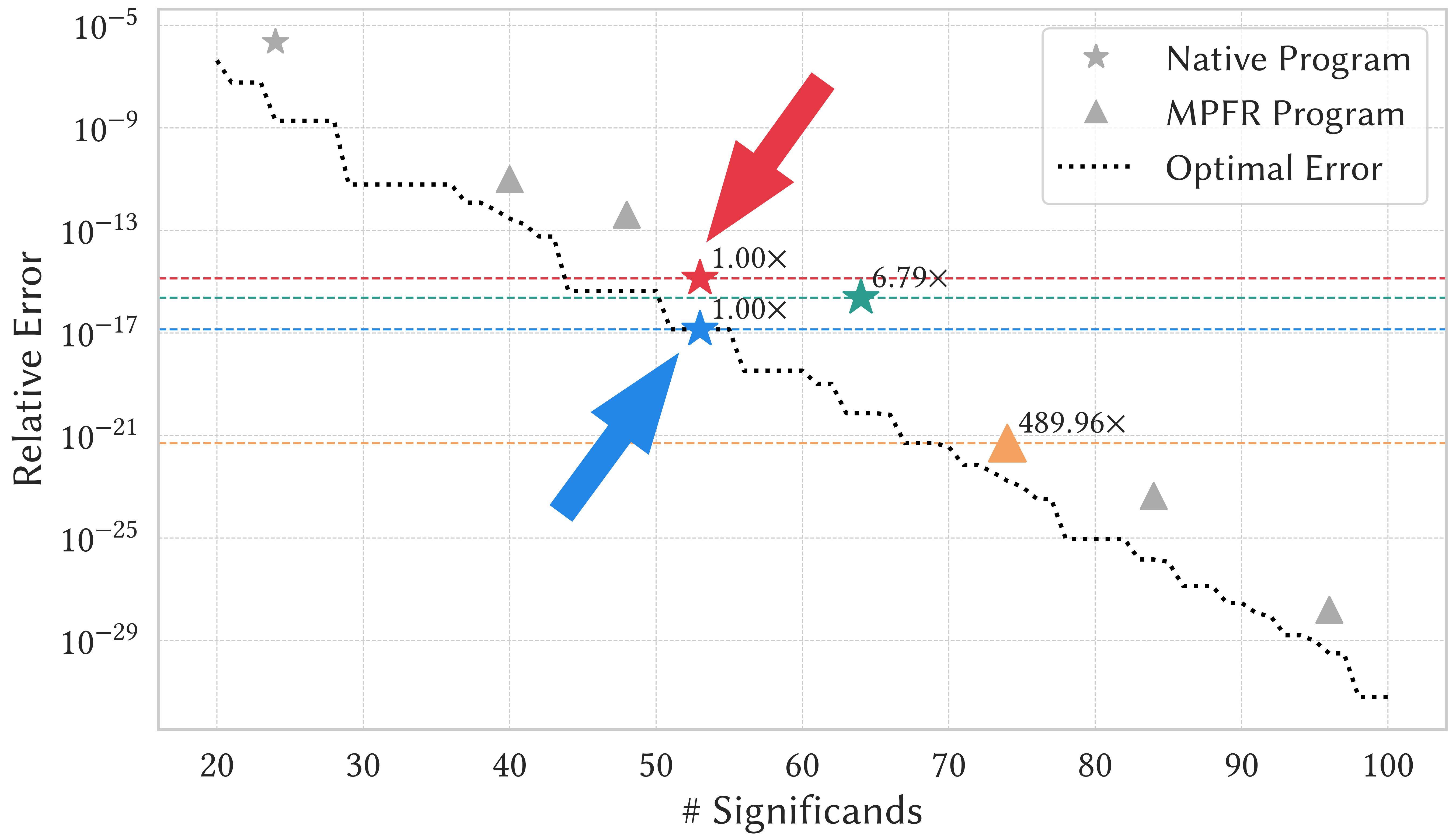
- Large DOE proxy application that simulates Lagrangian hydrodynamics
- 5600+ LOC with over 200 loops
- Original (FP64): **12 ULPs*** off



* Ground truth: 64-bit rounding of MPFR-512 computation.

Evaluation: LULESH

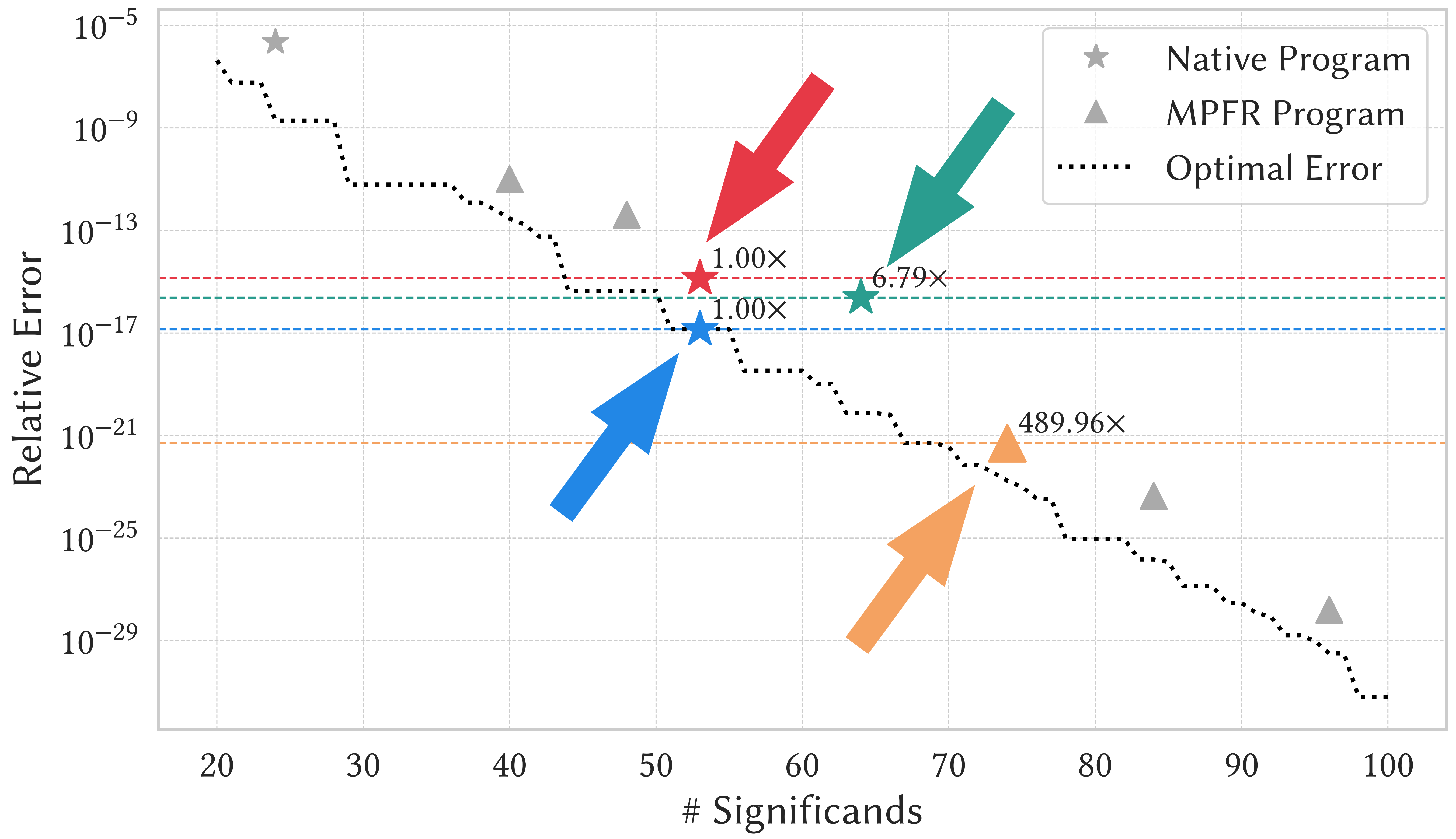
- Large DOE proxy application that simulates Lagrangian hydrodynamics
- 5600+ LOC with over 200 loops
- Original (FP64): **12 ULPs*** off
- Poseidon (FP64): **Optimal* FP64 error**; little slowdown



* Ground truth: 64-bit rounding of MPFR-512 computation.

Evaluation: LULESH

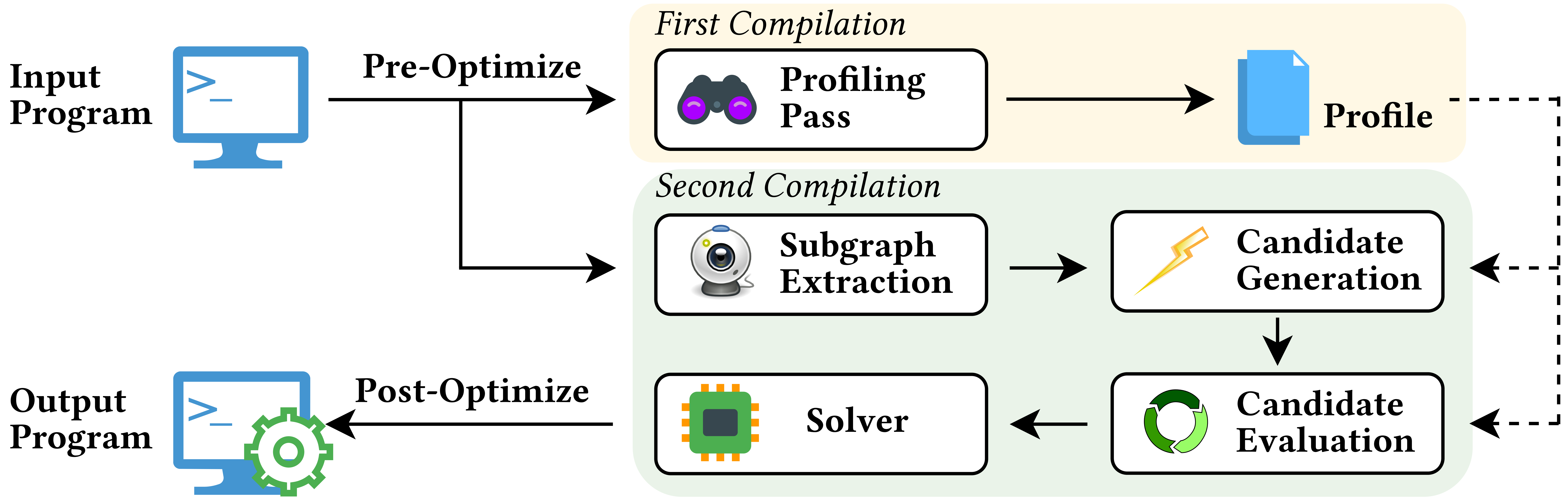
- Large DOE proxy application that simulates Lagrangian hydrodynamics
- 5600+ LOC with over 200 loops
- Original (FP64): **12 ULPs*** off
- Poseidon (FP64): **Optimal* FP64 error**; little slowdown
- FP80: **6.79×** slower; still **2 ULPs*** off
- MPFR-74: **~500×** slower



* Ground truth: 64-bit rounding of MPFR-512 computation.

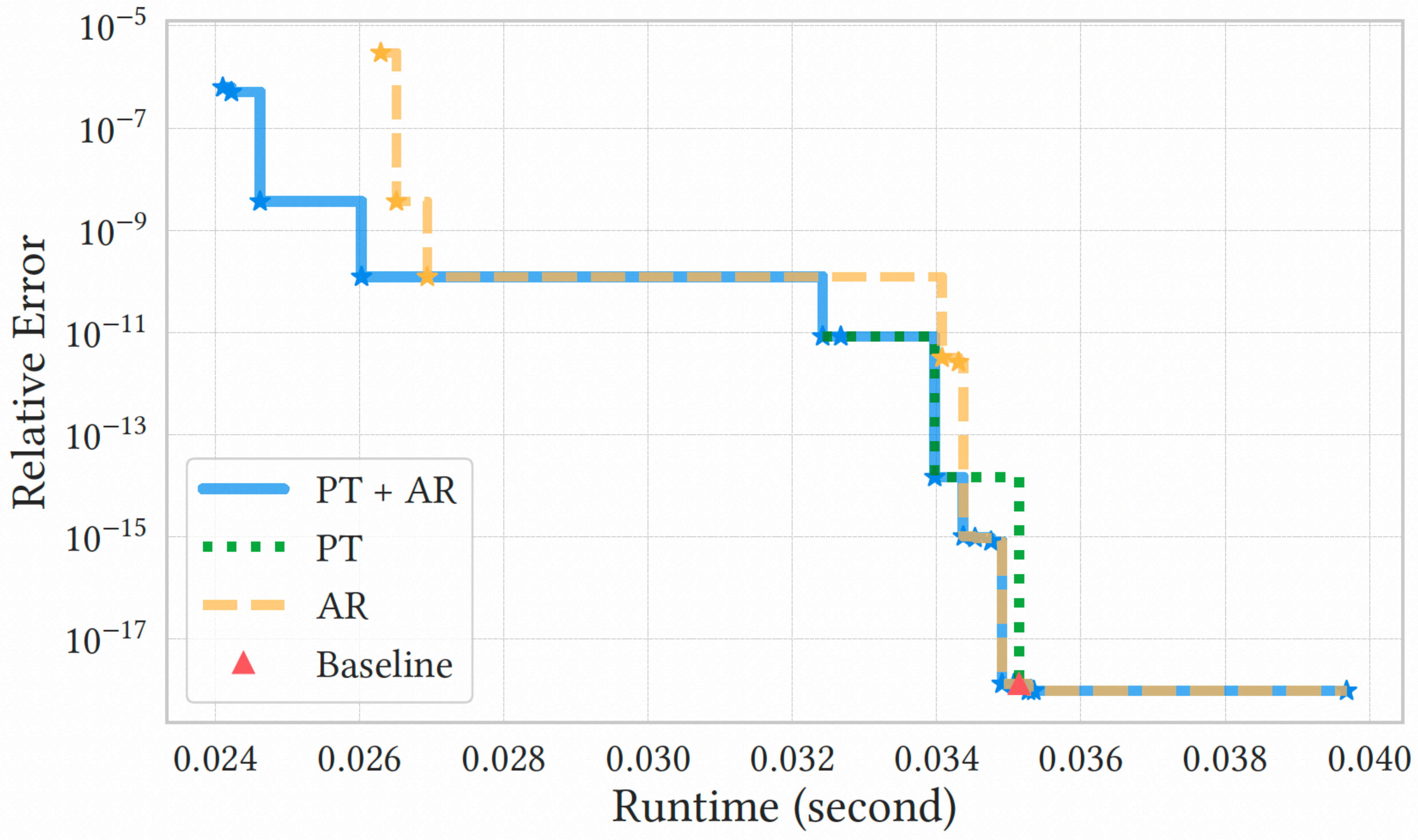
Poseidon (Open Source on GitHub: **PRONTOLab/Poseidon**)

- The first framework that automates numerical rewrites within a production compiler
- **Profiling** + **Compiler Optimization** provides context, scope, and scale
- Outsized performance benefit without sacrificing the accuracy, and *vice versa*



Evaluation: Quaternion Differentiator

- PT + AR: Precision Tuning + Algebraic Rewrites
- PT: Precision Tuning only
- AR: Algebraic Rewrites only
- **PT + AR has the best frontier!**



Lower-left is better