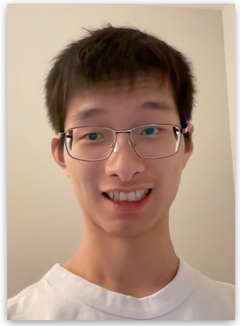
Poseidon: Automatic Profile-Guided Optimizations of Floating-Point Programs in Compilers





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
[2] Argonne National Laboratory

LLVM Developers' Meeting 2024, October 24, 2024

Numerical Analysis Primer

- A trend in Machine Learning: reduce precision to increase performance!

Numerical Analysis Primer

- A trend in Machine Learning: reduce precision to increase performance!
- Precision changes are not always the answer.
- Example: $1.0 \times 10^8 + 1.0 - 1.0 \times 10^8 = ?$
  - FP32: $(1.0 \times 10^8 + 1.0) - 1.0 \times 10^8 = 1.0 \times 10^8 - 1.0 \times 10^8 = 0.0$

Numerical Analysis Primer

- A trend in Machine Learning: reduce precision to increase performance!
- Precision changes are not always the answer.
- Example: $1.0 \times 10^8 + 1.0 - 1.0 \times 10^8 = ?$
  - FP32: $(1.0 \times 10^8 + 1.0) - 1.0 \times 10^8 = 1.0 \times 10^8 - 1.0 \times 10^8 = 0.0$
  - FP64: $(1.0 \times 10^8 + 1.0) - 1.0 \times 10^8 = 1.00000001 \times 10^8 - 1.0 \times 10^8 = 1.0$

Numerical Analysis Primer

- A trend in Machine Learning: reduce precision to increase performance!
- Precision changes are not always the answer.
- Example: $1.0 \times 10^8 + 1.0 - 1.0 \times 10^8 = ?$
  - FP32: $(1.0 \times 10^8 + 1.0) - 1.0 \times 10^8 = 1.0 \times 10^8 - 1.0 \times 10^8 = 0.0$
  - FP64: $(1.0 \times 10^8 + 1.0) - 1.0 \times 10^8 = 1.00000001 \times 10^8 - 1.0 \times 10^8 = 1.0$
  - **FP32 Reassociated:**
    $(1.0 \times 10^8 + 1.0) - 1.0 \times 10^8 = (1.0 \times 10^8 - 1.0 \times 10^8) + 1.0 = 1.0$

Existing Approaches

- Precision Tuning
  - `F32` → `BF16, F16, F64, F128, …`
- Floating-Point Expression Rewrites
  - $\sqrt{x+1} - \sqrt{x} \rightarrow \dfrac{1}{\sqrt{x+1} + \sqrt{x}}$
- Math Library Functions
  - $\sin(x) \approx x - \dfrac{x^3}{3!} + \dfrac{x^5}{5!}$

Existing Approaches

- Precision Tuning
  - `F32` → `BF16, F16, F64, F128, …`
- Floating-Point Expression Rewrites
  - $\sqrt{x+1} - \sqrt{x} \rightarrow \frac{1}{\sqrt{x+1} + \sqrt{x}}$
- Math Library Functions
  - $\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!}$

All require human intervention to determine if they are valid, let alone performant!

Can we automate existing FP optimization techniques in compilers?

How much performance are we sacrificing due to *poor* choices of FP precision and expressions?

Poseidon

- An end-to-end system which automatically transforms floating-point expressions in programs to maximize program accuracy subject to a computation cost budget
- Implemented as a PGO-like two-phase compilation within LLVM
  - First Compilation: **Instrumentation Pass**
  - Second Compilation
    - **FP Subgraph & Expression Transformations**
    - **Dynamic Programming Solver**

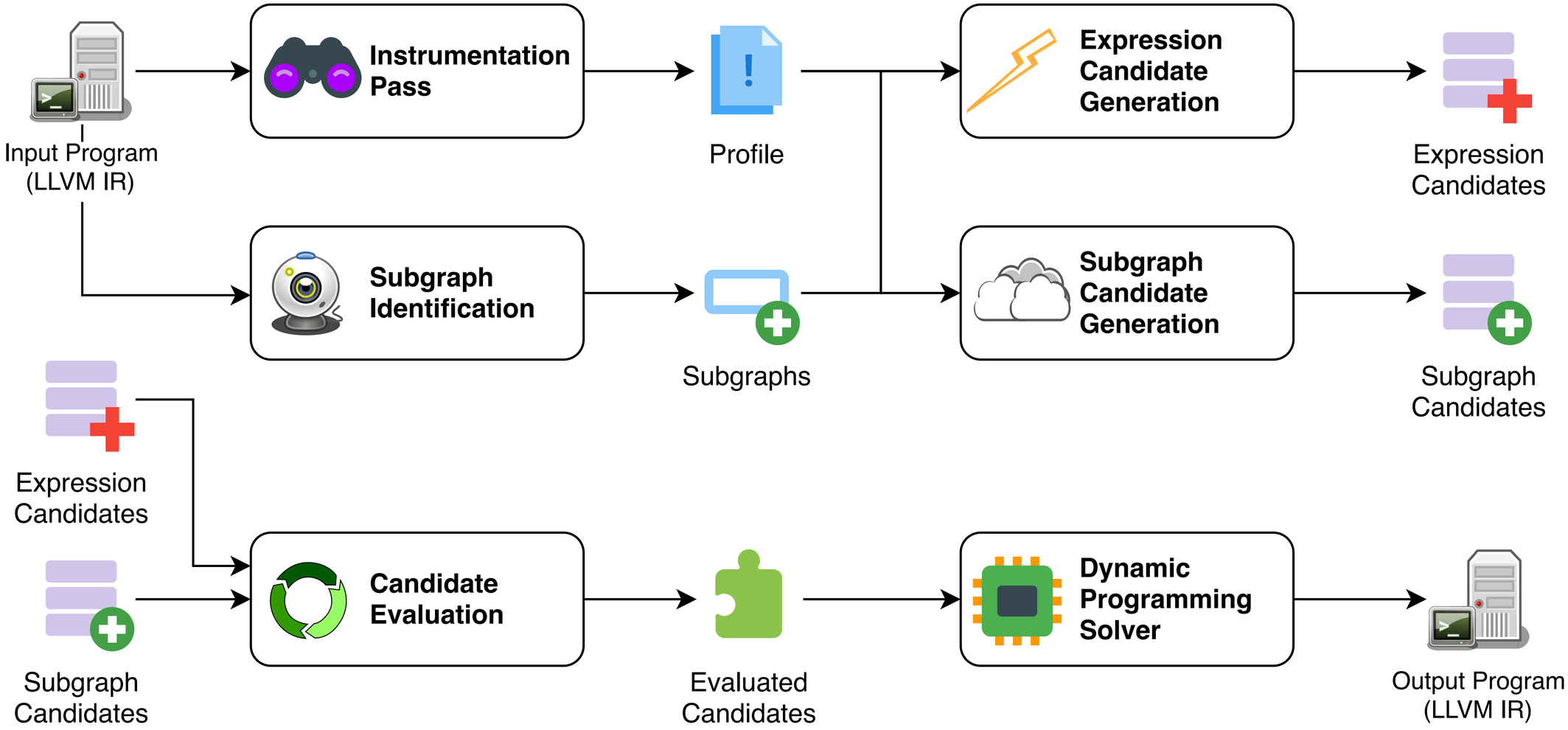
Floats…

Poseidon

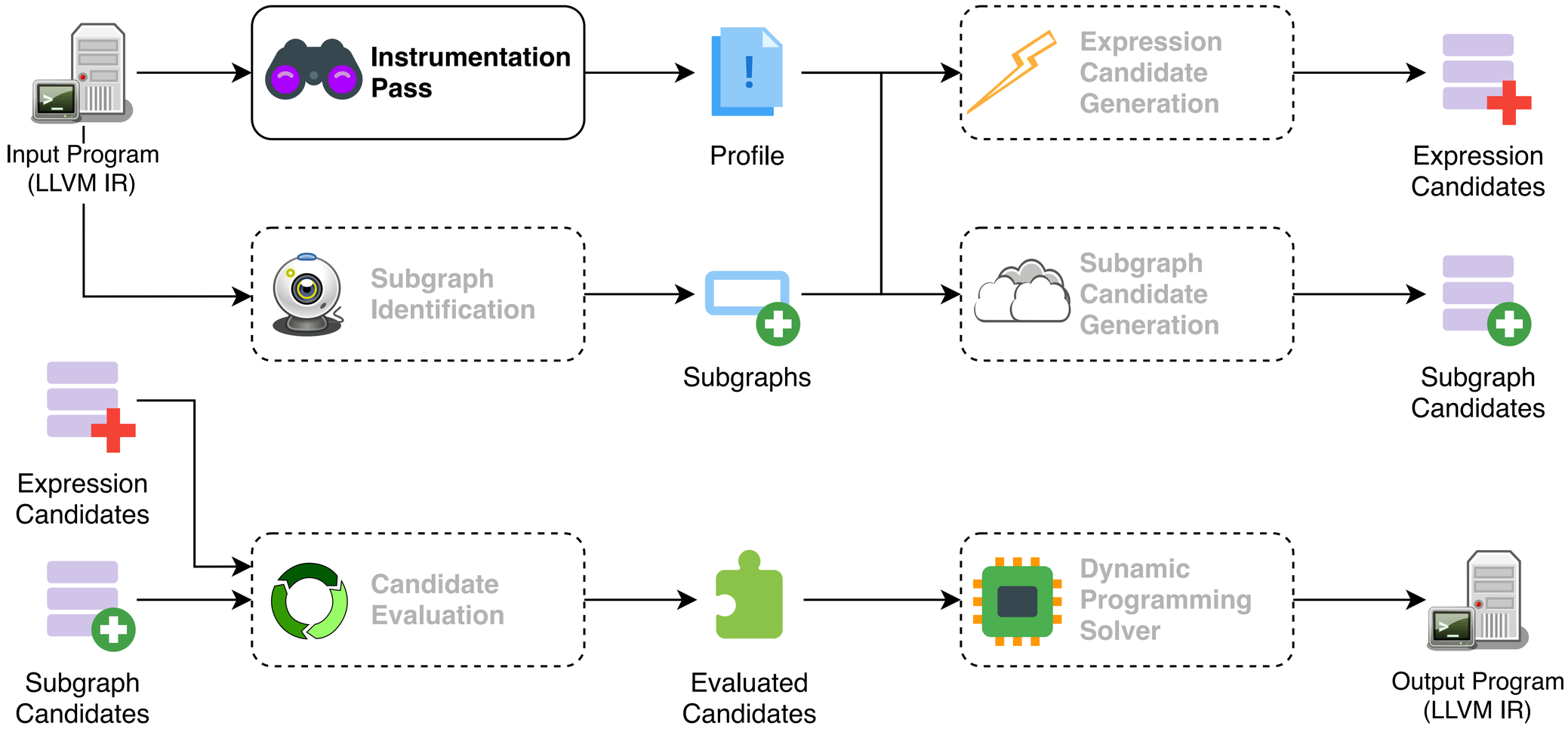
- An end-to-end system which automatically transforms floating-point expressions in programs to maximize program accuracy subject to a computation cost budget
- Implemented as a PGO-like two-phase compilation within LLVM
  - First Compilation: **Instrumentation Pass**
  - Second Compilation
    - **FP Subgraph & Expression Transformations**
    - **Dynamic Programming Solver**
- Implemented on top of **Enzyme**, a High-Performance Automatic Differentiator of LLVM and MLIR

Floats…

Poseidon Overview



Phase 1: Instrumentation



Phase 1: Instrumentation

- Augment the input program
  - Logger function calls embedded in Enzyme-synthesized gradient
- Run the augmented program to extract information of FP instructions

Phase 1: Instrumentation

- Augment the input program
  - Logger function calls embedded in Enzyme-synthesized gradient
- Run the augmented program to extract information of FP instructions

```
double example(…) {
  double tmp;
  if (a < b)
    tmp = sin(a + c * 3.0);
  else
    tmp = cos(b + c * 4.0);
  return tmp;
}
```

Original computation

```
define double @diffe_example(…) {
  …
  %tmp = call @llvm.sin.f64(…)
  call void @enzymeLogValue(double %tmp, …)
  %grad = call @llvm.cos.f64(…)
  call void @enzymeLogGrad(double %grad, …)
  …
}
```

Phase 1: Instrumentation

- Augment the input program
  - Logger function calls embedded in Enzyme-synthesized gradient
- Run the augmented program to extract information of FP instructions

```
double example(…) {
  double tmp;
  if (a < b)
    tmp = sin(a + c * 3.0);
  else
    tmp = cos(b + c * 4.0);
  return tmp;
}
```

Original computation

```
define double @diffe_example(…) {
  …
  %tmp = call @llvm.sin.f64(…)
  call void @enzymeLogValue(double %tmp, …)
  %grad = call @llvm.cos.f64(…)
  call void @enzymeLogGrad(double %grad, …)
  …
}
```

Enzyme-synthesized gradient

Phase 1: Instrumentation

- Augment the input program
  - Logger function calls embedded in Enzyme-synthesized gradient
- Run the augmented program to extract information of FP instructions

```
double example(…) {
  double tmp;
  if (a < b)
    tmp = sin(a + c * 3.0);
  else
    tmp = cos(b + c * 4.0);
  return tmp;
}
```

Original computation

```
define double @diffe_example(…) {
  …
  %tmp = call @llvm.sin.f64(…)
  call void @enzymeLogValue(double %tmp, …)
  %grad = call @llvm.cos.f64(…)
  call void @enzymeLogGrad(double %grad, …)
  …
}
```

Enzyme-synthesized gradient

Logger function calls

Phase 1: Instrumentation

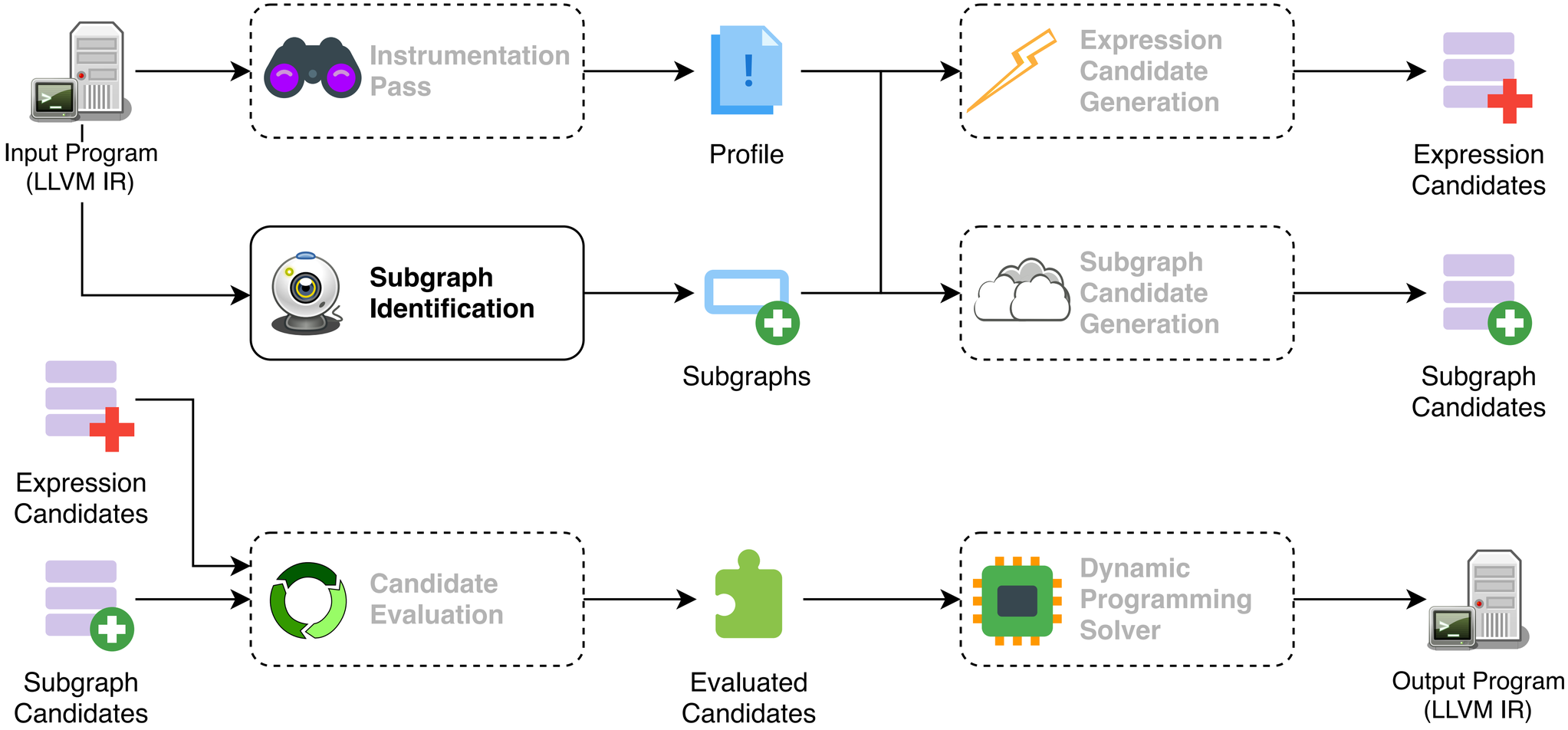
- Augment the input program
  - Logger function calls embedded in Enzyme-synthesized gradient
- Run the augmented program to extract information of FP instructions

```
double example(…) {
  double tmp;
  if (a < b)
    tmp = sin(a + c * 3.0);
  else
    tmp = cos(b + c * 4.0);
  return tmp;
}
```

Original computation

```
Value:example:1:2
    Min = -0.999…
    Max = 0.999…
    Executions = 50053
    Geometric Average = 6.017…e-01
// Other Values…
Grad:example:1:2
    Geometric Average = 9.999…e-01
// Other Gradients…
```

Phase 2A: Subgraph Identification



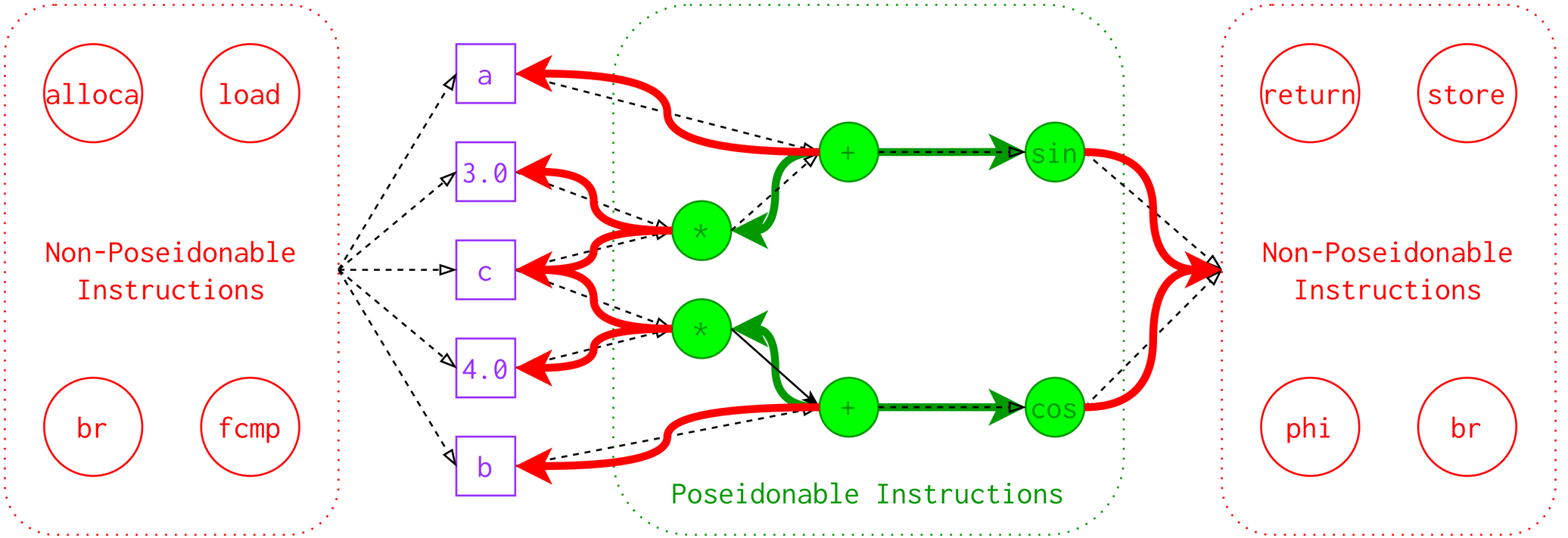
Phase 2A: Subgraph Identification

- Optimizable `llvm::Value` → *Poseidon-able* Values
- Poseidonable Values (FP Instructions)
  - `+, -, *, /, sin, cos, tan, exp, log, sqrt, cbrt, pow, fma, hypot, expm1, log1p, …`
- Non-Poseidonable Values
  - `load, store, br, ret, if*, ==*, <*, >*, <=*, >=*, !=*, and*, or*, not*, …`

* Opcode planned but not yet analyzable in Poseidon.

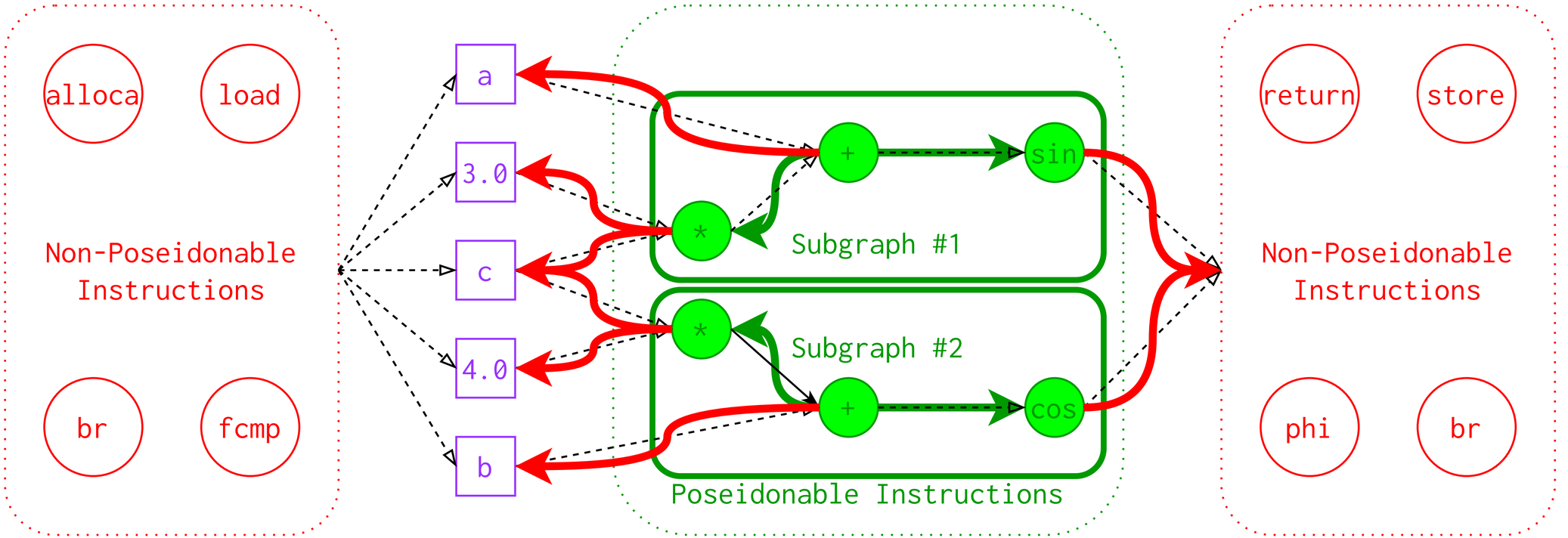
Phase 2A: Subgraph Identification

- Floodfill Algorithm → **DAGs of Poseidonable Instructions**

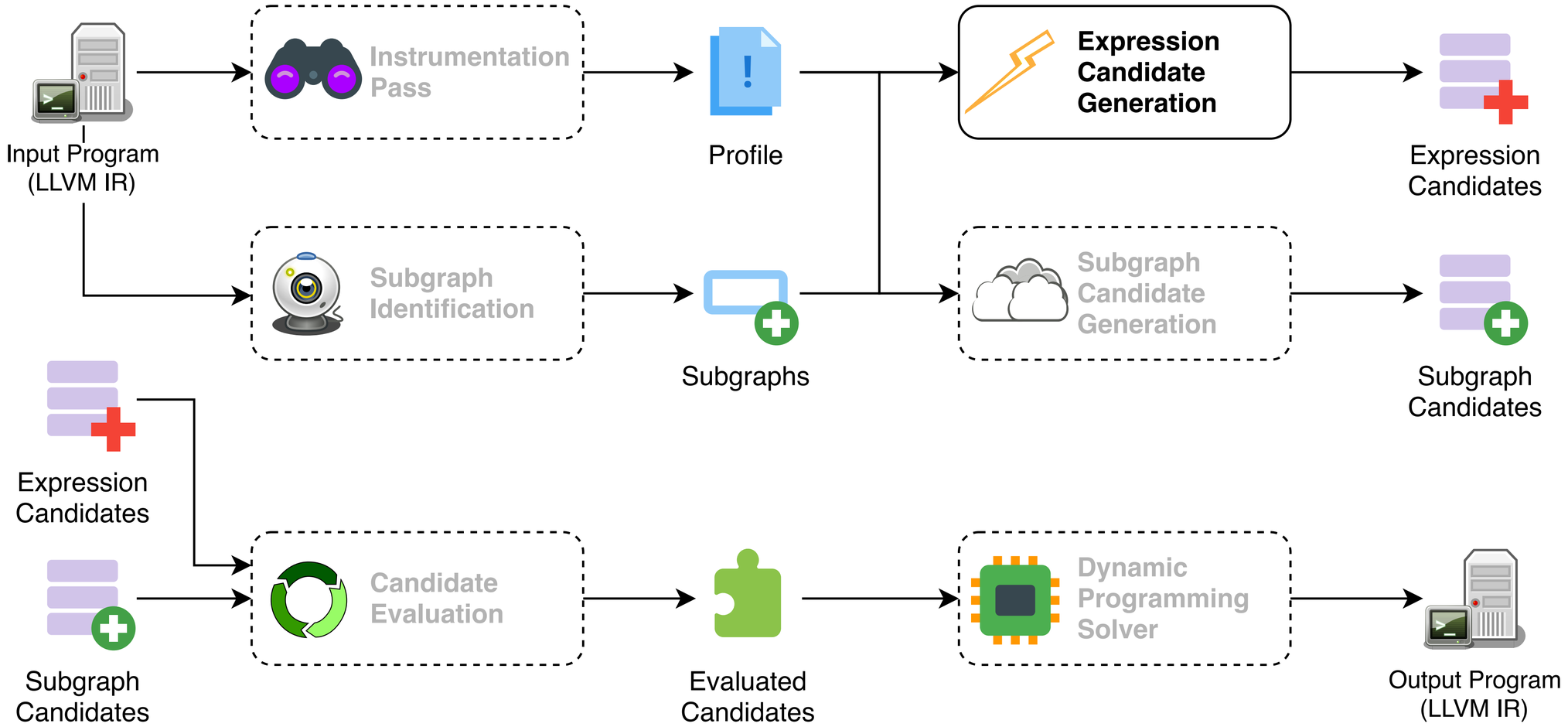


Phase 2A: Subgraph Identification

- Floodfill Algorithm → **DAGs of Poseidonable Instructions**

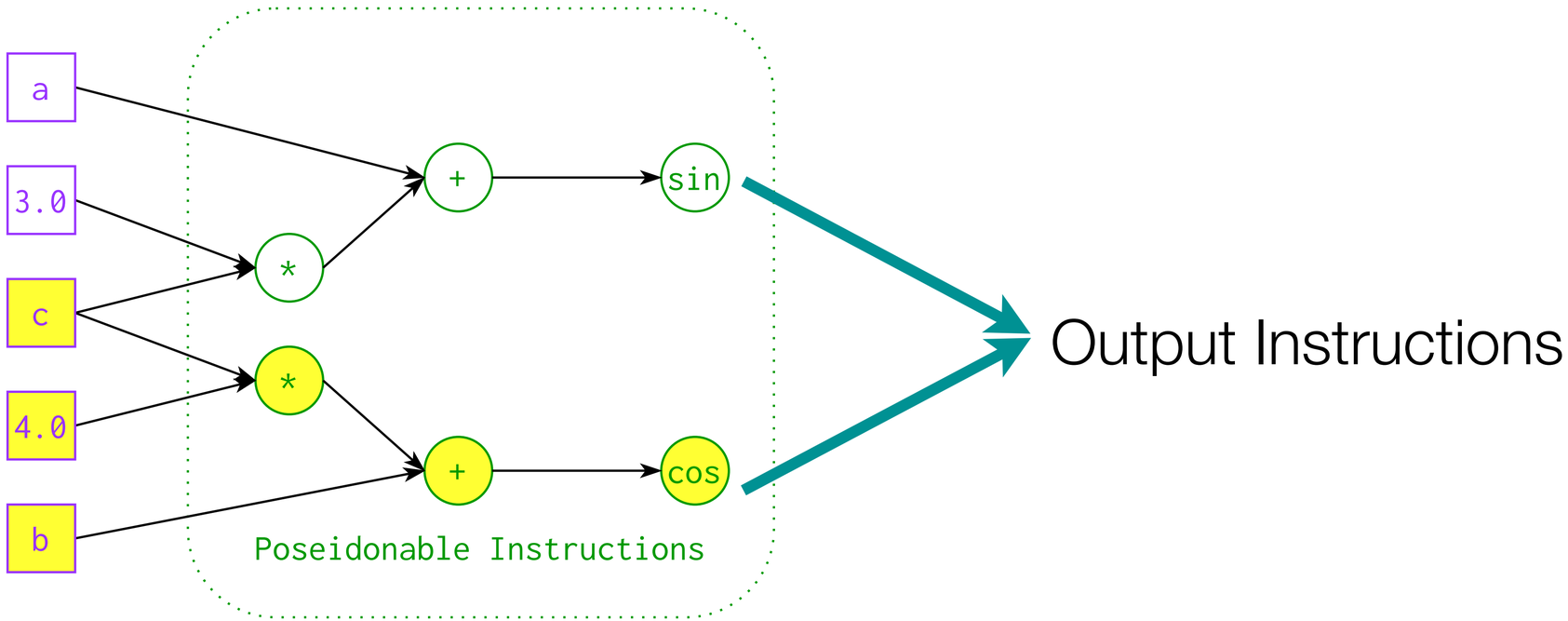


Phase 2B: Expression Candidate Generation



Phase 2B: Expression Candidate Generation

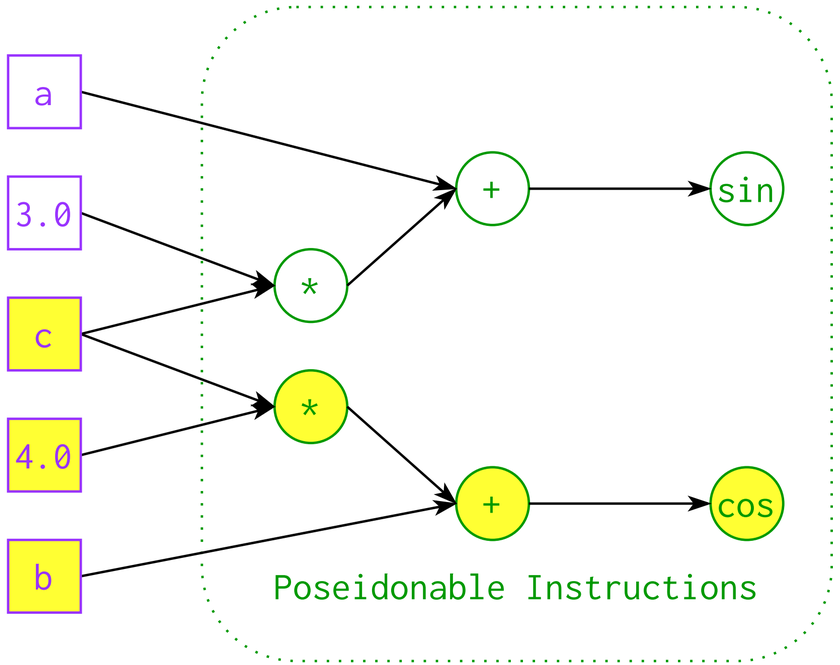
- *Output instructions* propagate values to non-poseidonable values



Phase 2B: Expression Candidate Generation

- *Output instructions* propagate values to non-poseidonable values

- Construct $\mathbb{F}$ expressions from *output instructions* and use existing tools to produce expression candidates




```
// Herbie* Input
(FPCore (v0 v2)
  :pre (and (<= 1.0…e+00 v0 8.9…e+00)
            (<= 1.0…e+00 v2 8.9…e+00))
  (cos (+ (* v0 4) v2)))
```

Bounds from *Phase 1: Instrumentation.*

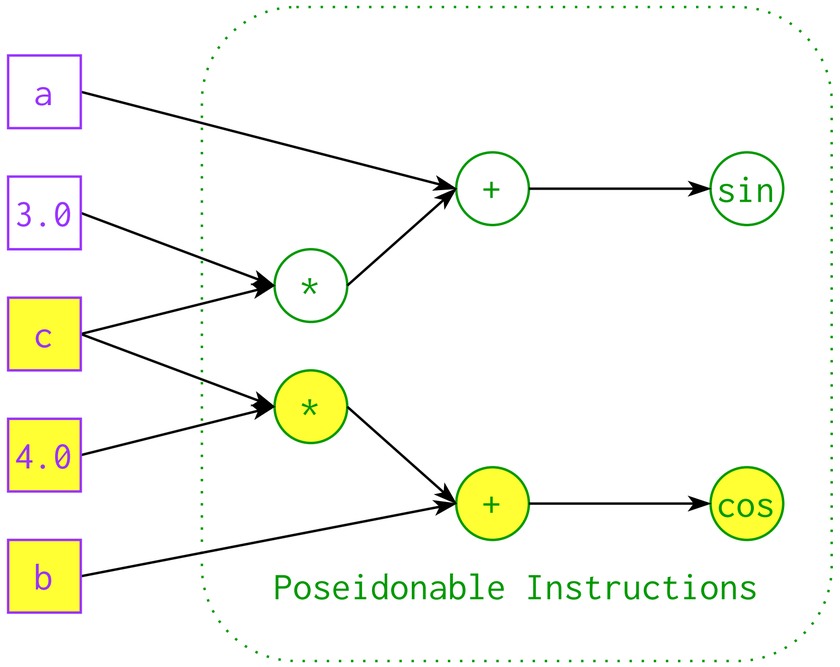
Constructed expression

* Panchekha et al. '15

Phase 2B: Expression Candidate Generation

- *Output instructions* propagate values to non-poseidonable values

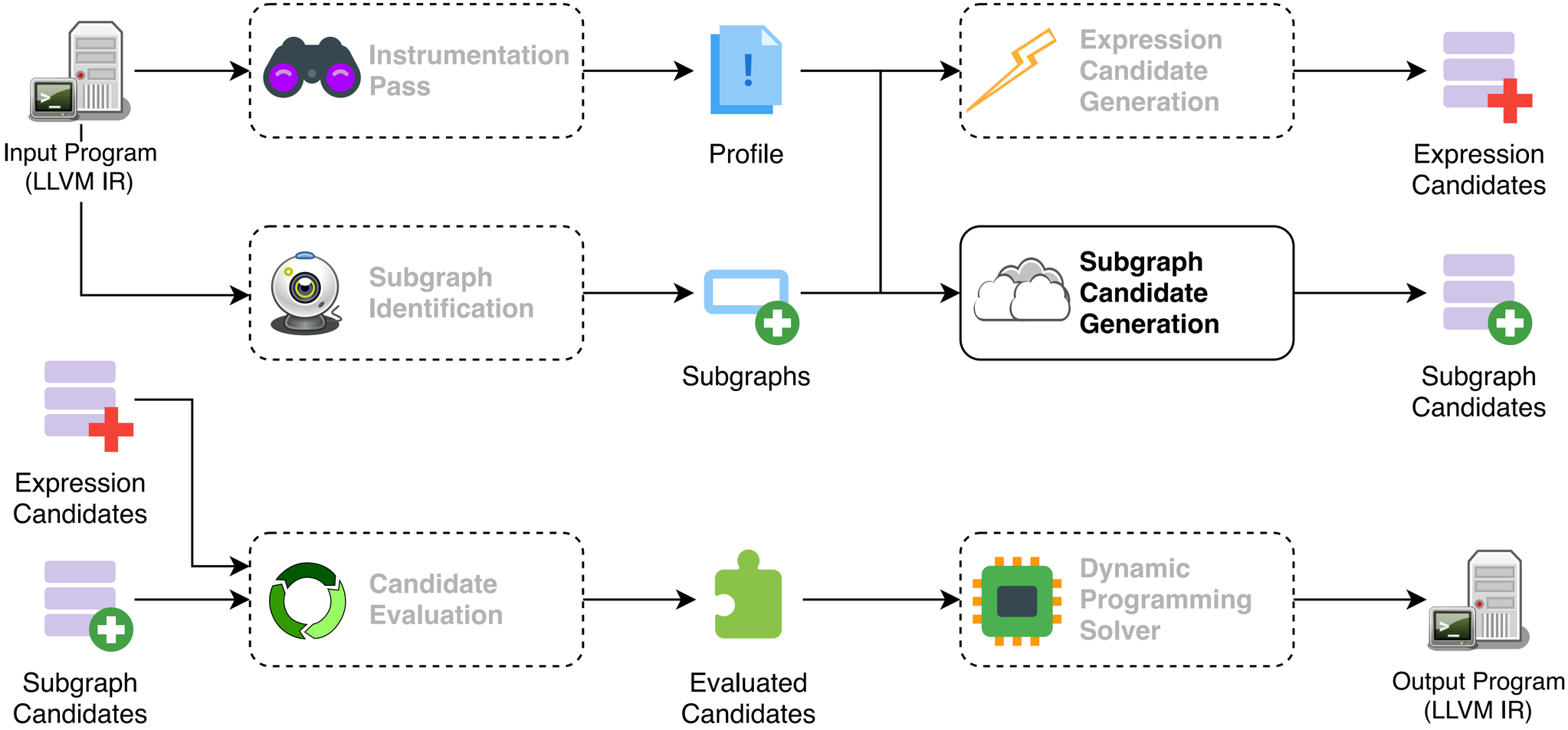
- Construct $\mathbb{F}$ expressions from *output instructions* and use existing tools to produce expression candidates




```
// Herbie* Input
(FPCore (v0 v2)
  :pre (and (<= 1.0…e+00 v0 8.9…e+00)
            (<= 1.0…e+00 v2 8.9…e+00))
  (cos (+ (* v0 4) v2)))
```

```
// Expression Candidate #1
(- (* (cos (* v0 4)) (cos v2))
  (* (sin (* v0 4)) (sin v2)))
// Expression Candidate #2
(fma (cos (* v0 4)) (cos v2) (*
  (sin (* v0 4)) (sin (neg v2))))
```

Phase 2C: Subgraph Candidate Generation



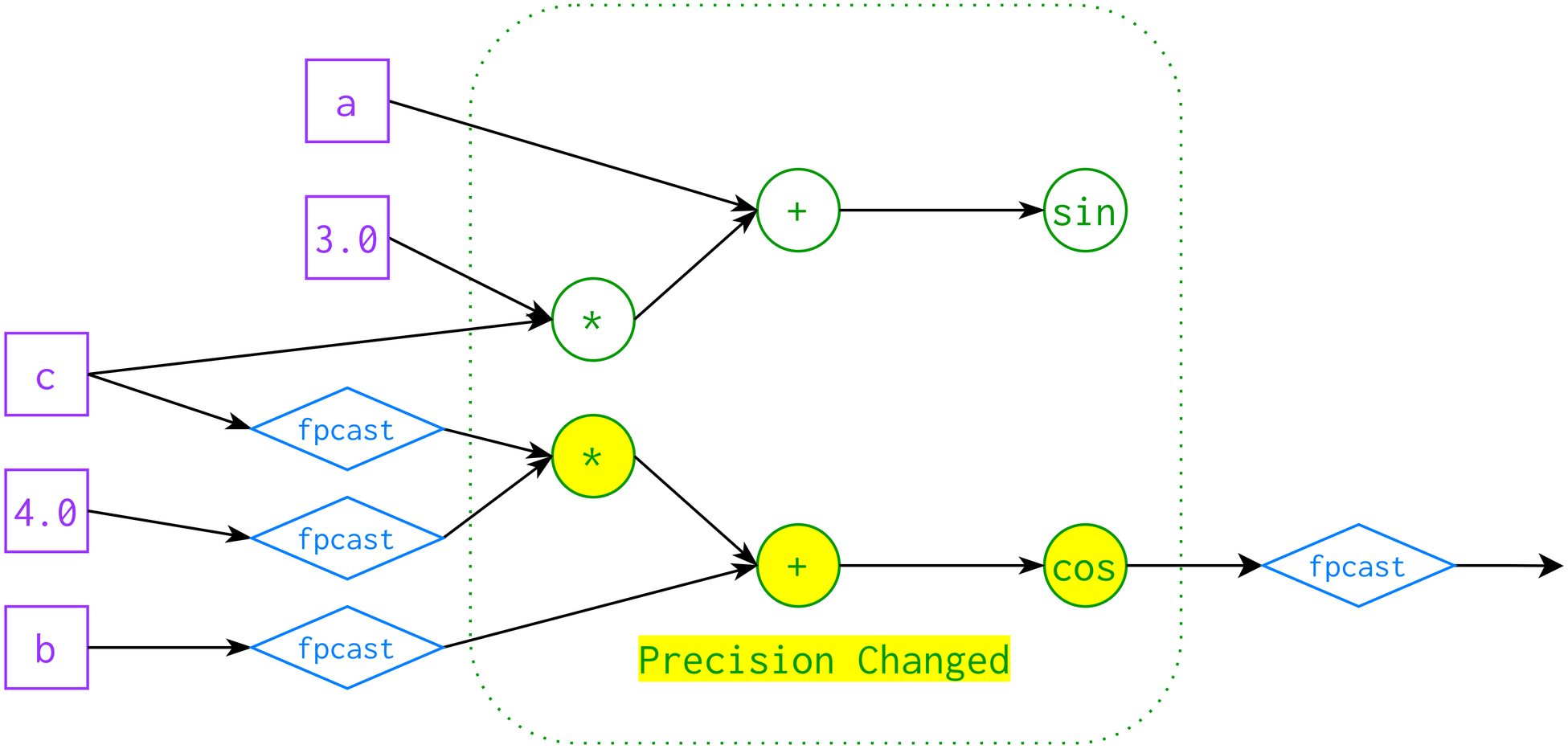
Phase 2C: Subgraph Candidate Generation

- **Sensitivity*** (How much changing an intermediate instruction impacts the final result)
  - Computed as $|x \cdot f'(x)|$
  - Values and gradients come from *Phase 1: Instrumentation*
- Gradient-Guided Precision Tuning

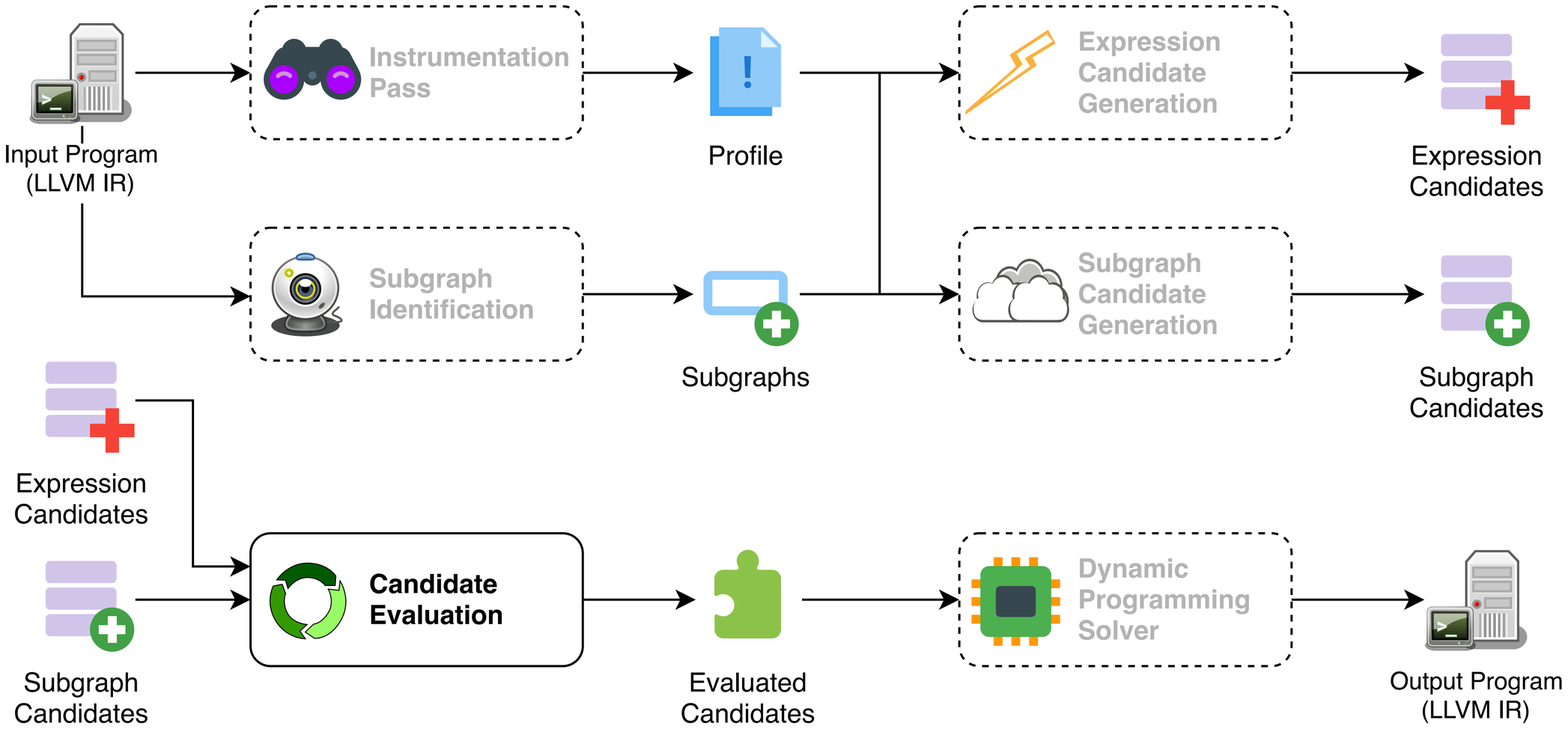
* From ADAPT (Menon et al. ’18).

Phase 2C: Subgraph Candidate Generation

- Gradient-Guided Precision Tuning



Phase 2C: Candidate Evaluation



Phase 2D: Candidate Evaluation

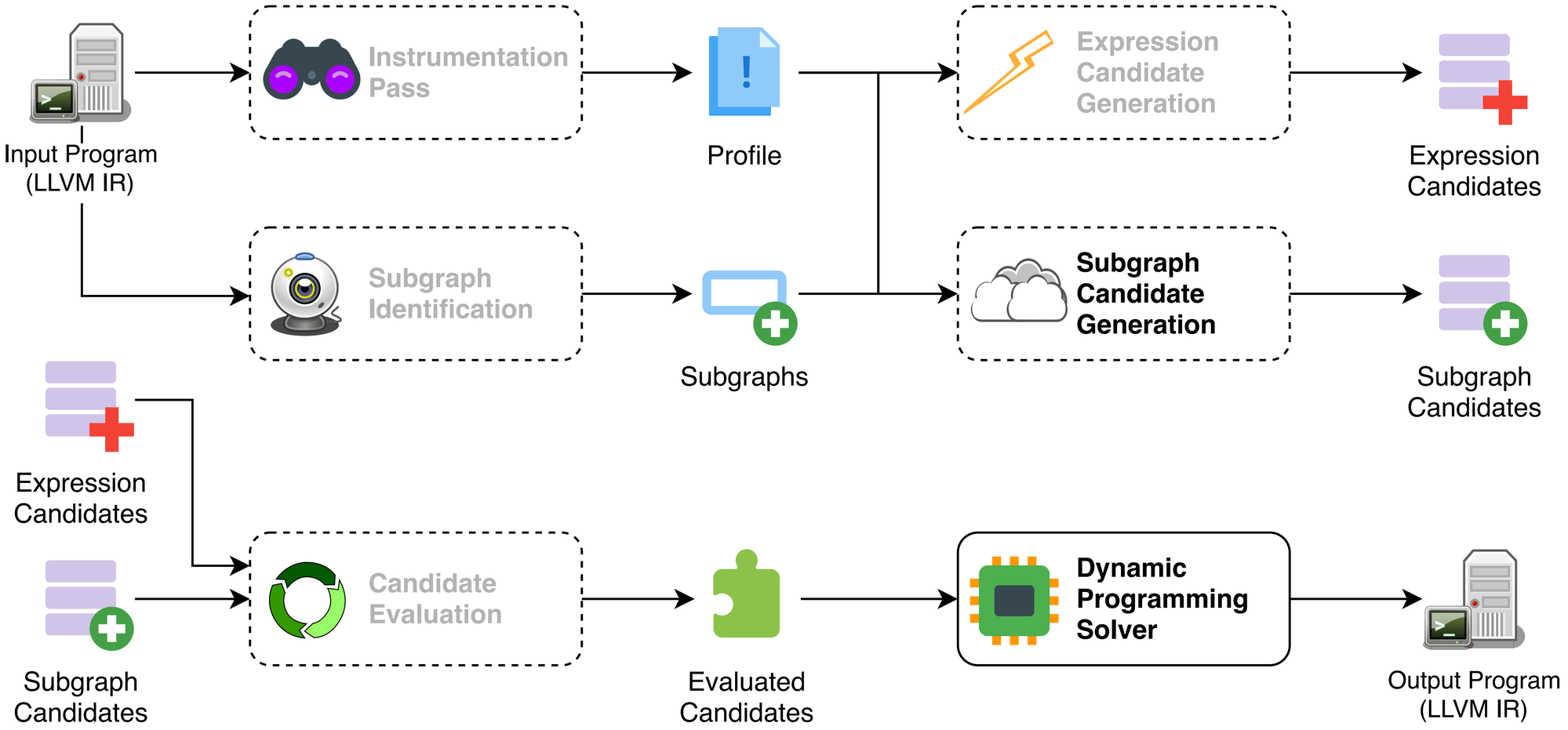
- How *local* transformations change *global* cost and accuracy?
- Cost Model
  - `llvm::TargetTransformInfo::getInstructionCost`
  - Custom Cost Model (Microbenchmarking LLVM Instructions)
  - $\texttt{Cost} = \sum_i \texttt{ExecutionCount}(i) \times \texttt{InstructionCost}(i)$

Phase 2D: Candidate Evaluation

- How *local* transformations change *global* cost and accuracy?
- Accuracy Model
  - Built-In MPFR Evaluator: **Ground Truth*** and **Emulated Results** of *Output Instructions*
  - $\texttt{Global Error} = \sum_{o} \left| \texttt{Gradient}(o) \times (\texttt{GroundTruth}(o) - \texttt{EmulatedResult}(o)) \right|$

* Similar to techniques in Herbie (Panchekha et al. '15).

Phase 3: Dynamic Programming Solver

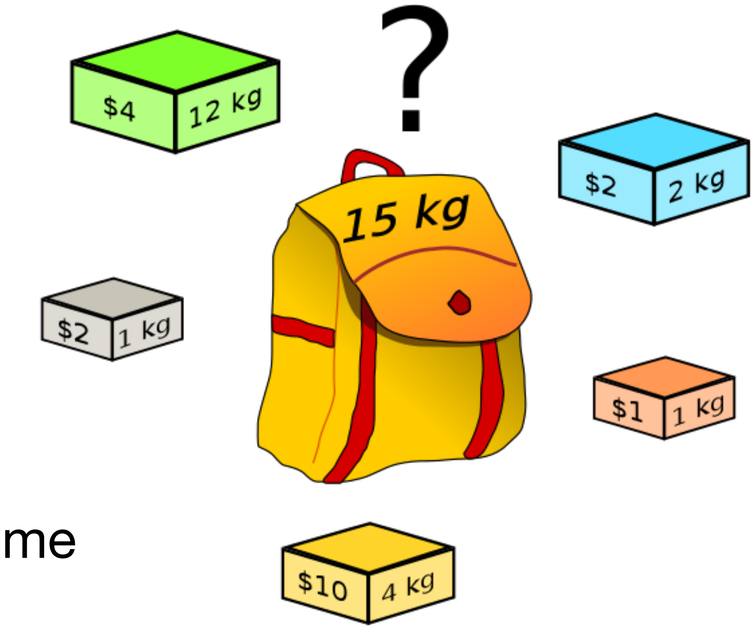


Phase 3: Dynamic Programming Solver

- Poseidon's Problem
  - Known: Computation Cost **Budget**, Expression Candidates (**Cost** & **Global Error**), Subgraph Candidates (**Cost** & **Global Error**).
  - Objective: $\operatorname{argmin}_{\phi} \sum_{\phi} \texttt{GlobalError}(\phi)$
  - Constraints:
    - $\sum_{\phi} \texttt{Cost}(\phi) \leq \textbf{ComputationCostBudget}$
    - Select only one candidate per $\mathbb{F}$ expression/subgraph

Phase 3: Dynamic Programming Solver

- 0/1 Knapsack Problem
  - Known: Knapsack **Capacity**, Items (**Weight** & **Profit**).
  - Objective: $\max_{\phi} \sum_{\phi} \texttt{Profit}(\phi)$
  - Constraint: $\sum_{\phi} \texttt{Weight}(\phi) \leq \textbf{Capacity}$
  - Can be solved using Dynamic Programming in polynomial time



Phase 3: Dynamic Programming Solver

- Poseidon's Problem (Rounding all costs to nearest integers)
  - Known: Computation Cost **Budget**, Expression Candidates (**Cost** & **Global Error**), Subgraph Candidates (**Cost*** & **Global Error***).
  - Objective: $\operatorname{argmin}_{\phi} \sum_{\phi} \texttt{GlobalError}(\phi)$
  - Constraints:
    - $\sum_{\phi} \texttt{Cost}(\phi) \leq \textbf{ComputationCostBudget}$
    - Select only one candidate per $\mathbb{F}$ expression/subgraph

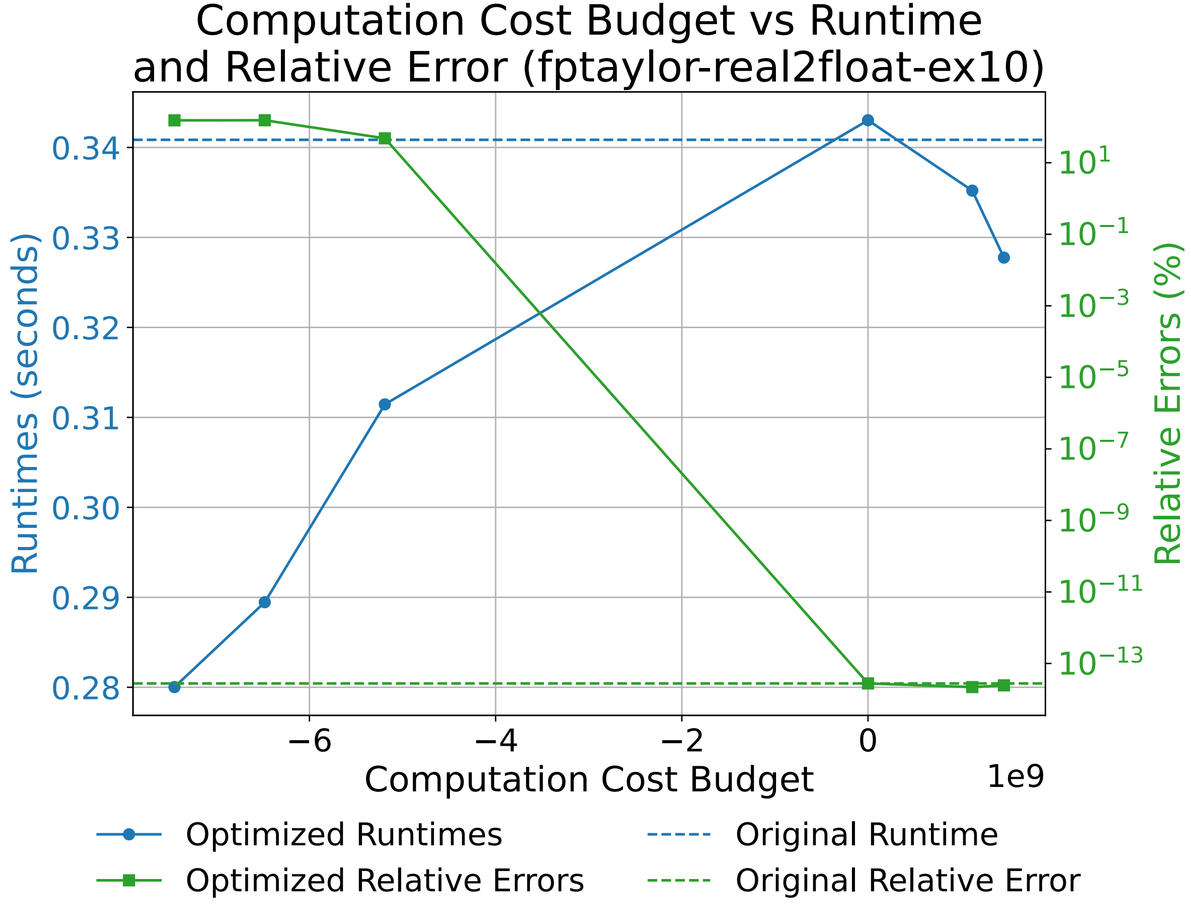
* Adjusted in the solving process.

Evaluation: FPBench Microbenchmarks

- Small FP functions (exported to C): 46 immediately optimizable functions
- Permitting 0.0001% Relative Error → **3.8%** Average Runtime Improvement
- Permitting 0.01% Relative Error → **5.3%** Average Runtime Improvement
- Permitting 10% Relative Error → **9.4%** Average Runtime Improvement
- Up to **5** bits of accuracy improvements (**0.23** bit on average)

Evaluation: FPBench Microbenchmarks

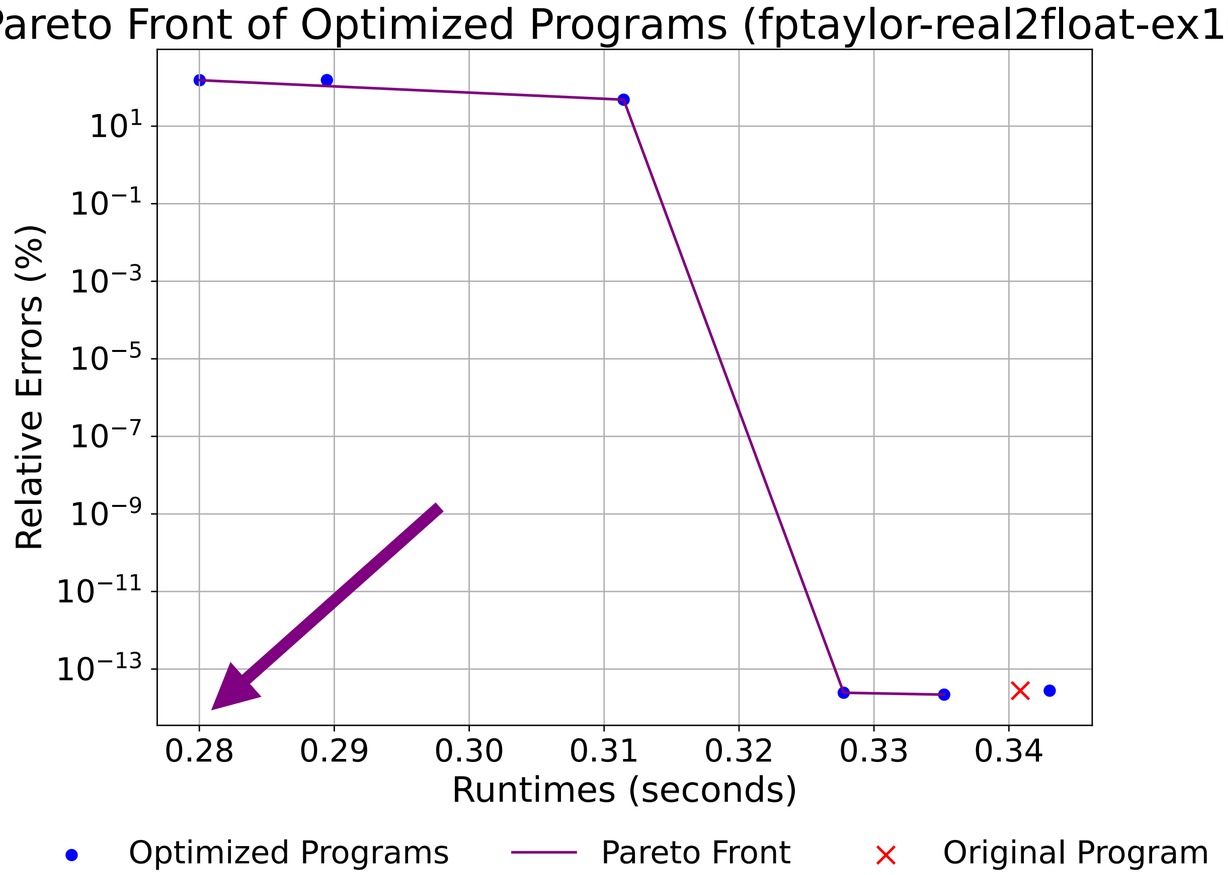
```
double ex10(double x1, double
x2, double x3, double x4,
double x5, double x6) {
  return ((((((((x1 * x4) *
(((((-x1 + x2) + x3) - x4) +
x5) + x6)) + ((x2 * x5) *
(((((x1 - x2) + x3) + x4) - x5)
+ x6))) + ((x3 * x6) * ((((x1
+ x2) - x3) + x4) + x5) - x6)))
- ((x2 * x3) * x4)) - ((x1 *
x3) * x5)) - ((x1 * x2) * x6))
- ((x4 * x5) * x6);
}
```



Lower is better

Evaluation: FPBench Microbenchmarks

```
double ex10(double x1, double
x2, double x3, double x4,
double x5, double x6) {
  return ((((((x1 * x4) *
(((((-x1 + x2) + x3) - x4) +
x5) + x6)) + ((x2 * x5) *
(((((x1 - x2) + x3) + x4) - x5)
+ x6))) + ((x3 * x6) * (((((x1
+ x2) - x3) + x4) + x5) - x6)))
- ((x2 * x3) * x4)) - ((x1 *
x3) * x5)) - ((x1 * x2) * x6))
- ((x4 * x5) * x6);
}
```



Lower-left is better

Poseidon (Open Source on GitHub: **EnzymeAD/Enzyme**)

- An end-to-end system that automatically performs profile-guided two-phase optimizations of floating-point programs in compilers

- **Instrumentation Pass**
- **Expression/Subgraph Transformation Candidate Generation & Evaluation**
- **Dynamic Programming Solver**
- Permitting 0.01% Relative Error → **5.3%** Average Runtime Improvement