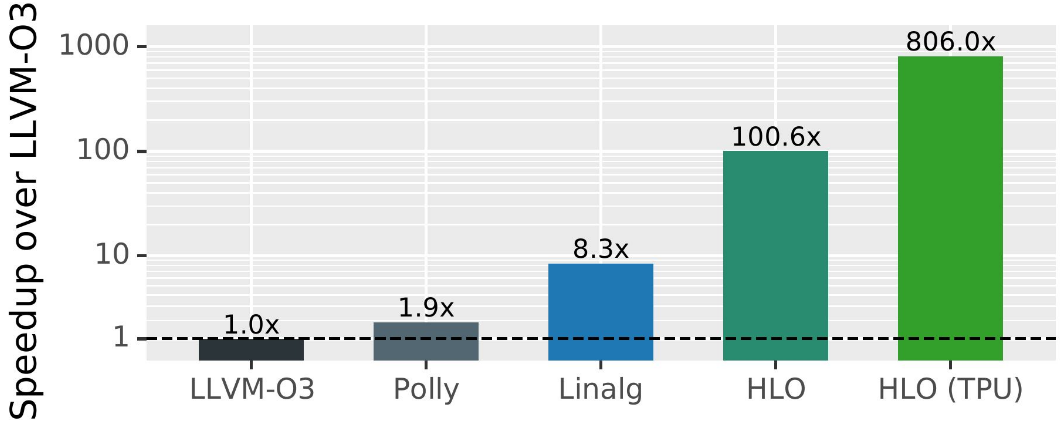
THE UNIVERSITY of EDINBURGH
informatics

MlirSynth: Automatic, Retargetable Program Raising in Multi-Level IR using Program Synthesis

Alexander Brauckmann, Elizabeth Polgreen,
Tobias Grosser, Michael O’Boyle

Abstraction raising enables significant performance



C Program

```
for (int r = 0; r < 150; r++) {
  for (int q = 0; q < 140; q++) {
    for (int p = 0; p < 160; p++) {
      sum[p] = 0.0;
      for (int s = 0; s < 160; s++)
        sum[p] += A[r][q][s] * C4[s][p];
    }
    for (int p = 0; p < 160; p++)
      A[r][q][p] = sum[p];
  }
}
```

Linalg IR

```
%0 = tensor.collapse_shape %arg0 [[0, 1], [2]]
  : tensor<150x140x160xf64>
  into tensor<2100x160xf64>
%1 = linalg.matmul
  ins(%0, %arg1 : tensor<21000x160xf32>,
          tensor<160x160xf32>)
  outs(%1 : tensor<21000x160xf32>)
  -> tensor<21000x160xf32>
%2 = tensor.expand_shape %1 [[0, 1], [2]]
  : tensor<2100x160xf64>
  into tensor<150x140x160xf64>
```

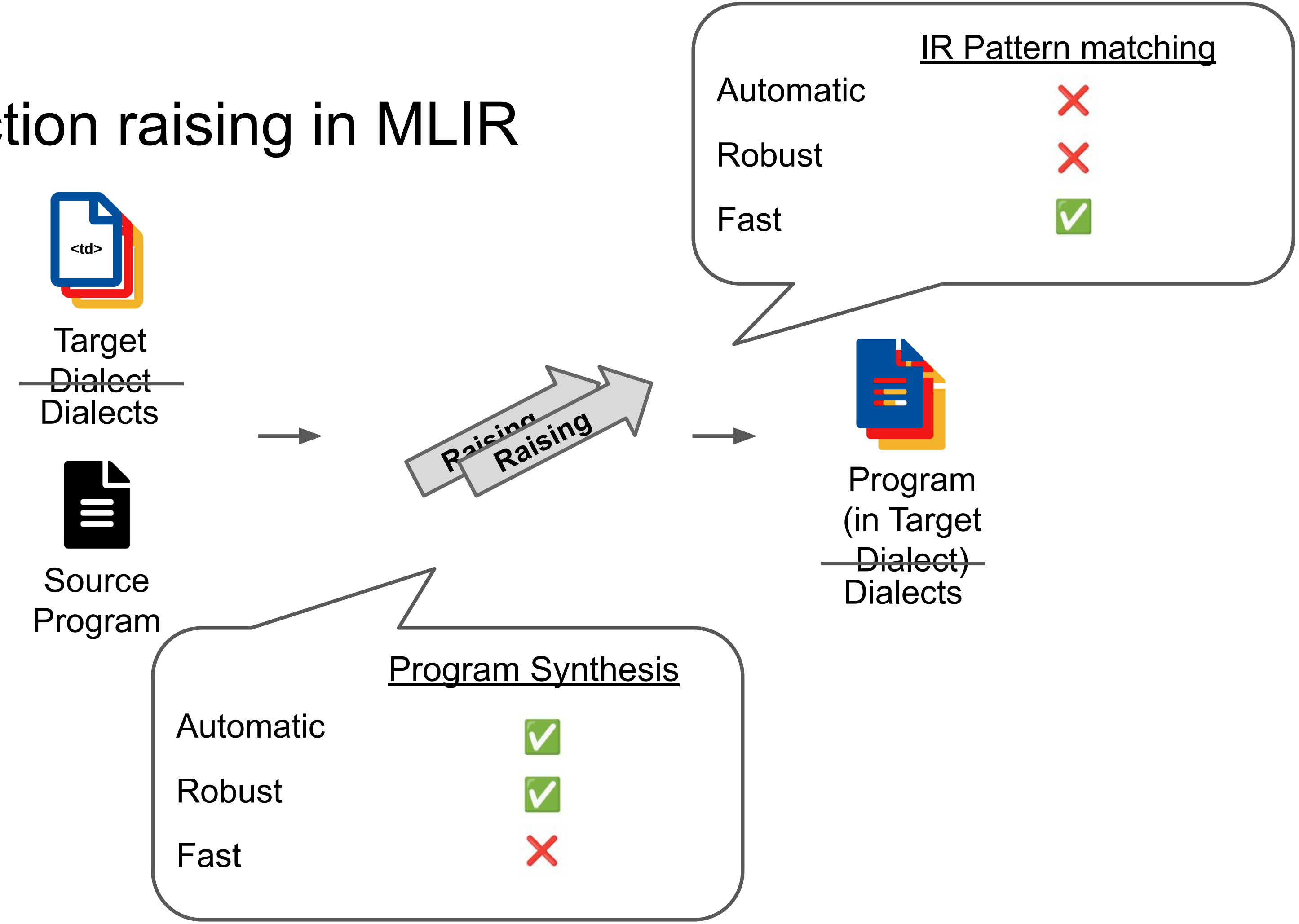
HLO IR

```
%0 = mhlo.dot_general (%arg0, %arg1) {
  dot_dimension_numbers = #mhlo.dot<
    lhs_contracting_dimensions = [2],
    rhs_contracting_dimensions = [0]>}
  : (tensor<150x140x160xf32>,
    tensor<160x160xf32>)
  -> tensor<150x140x160xf32>
```

CPU: AMD Ryzen 9 3900X
TPU: TPUv2

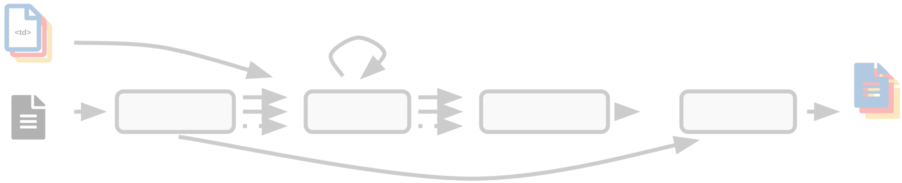
Motivation

Abstraction raising in MLIR

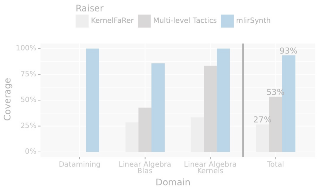


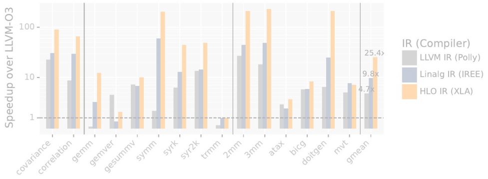
Overview

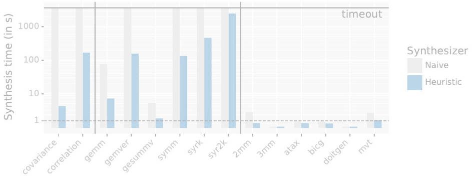
mlirSynth



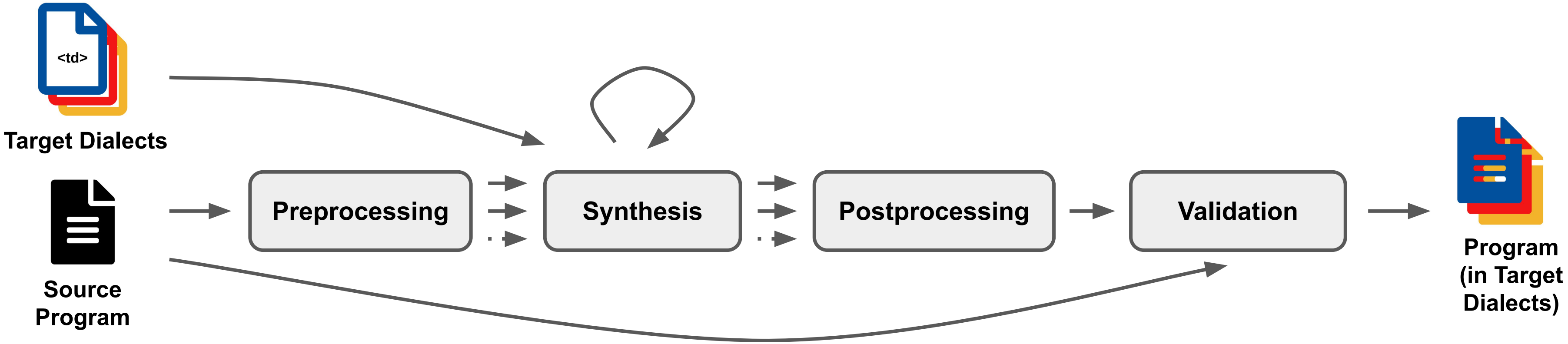
Evaluation



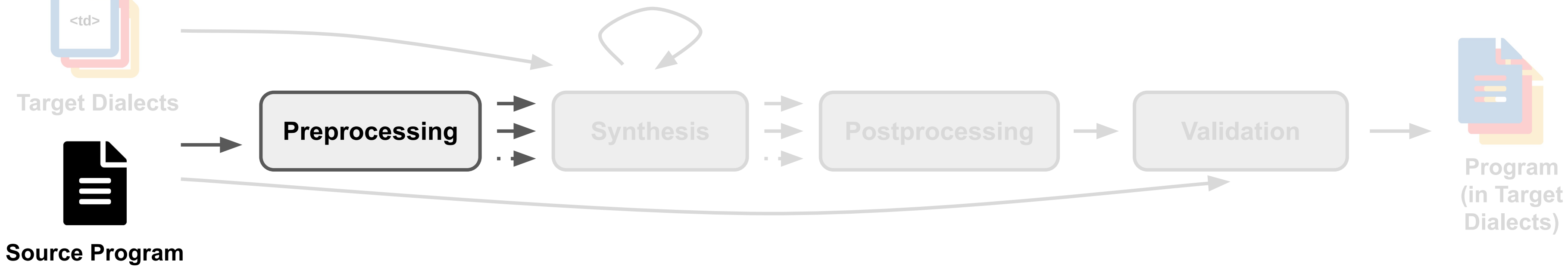




mlirSynth



<td>
Target Dialects
Source Program
Preprocessing
Synthesis
Postprocessing
Validation
Program (in Target Dialects)



C Code

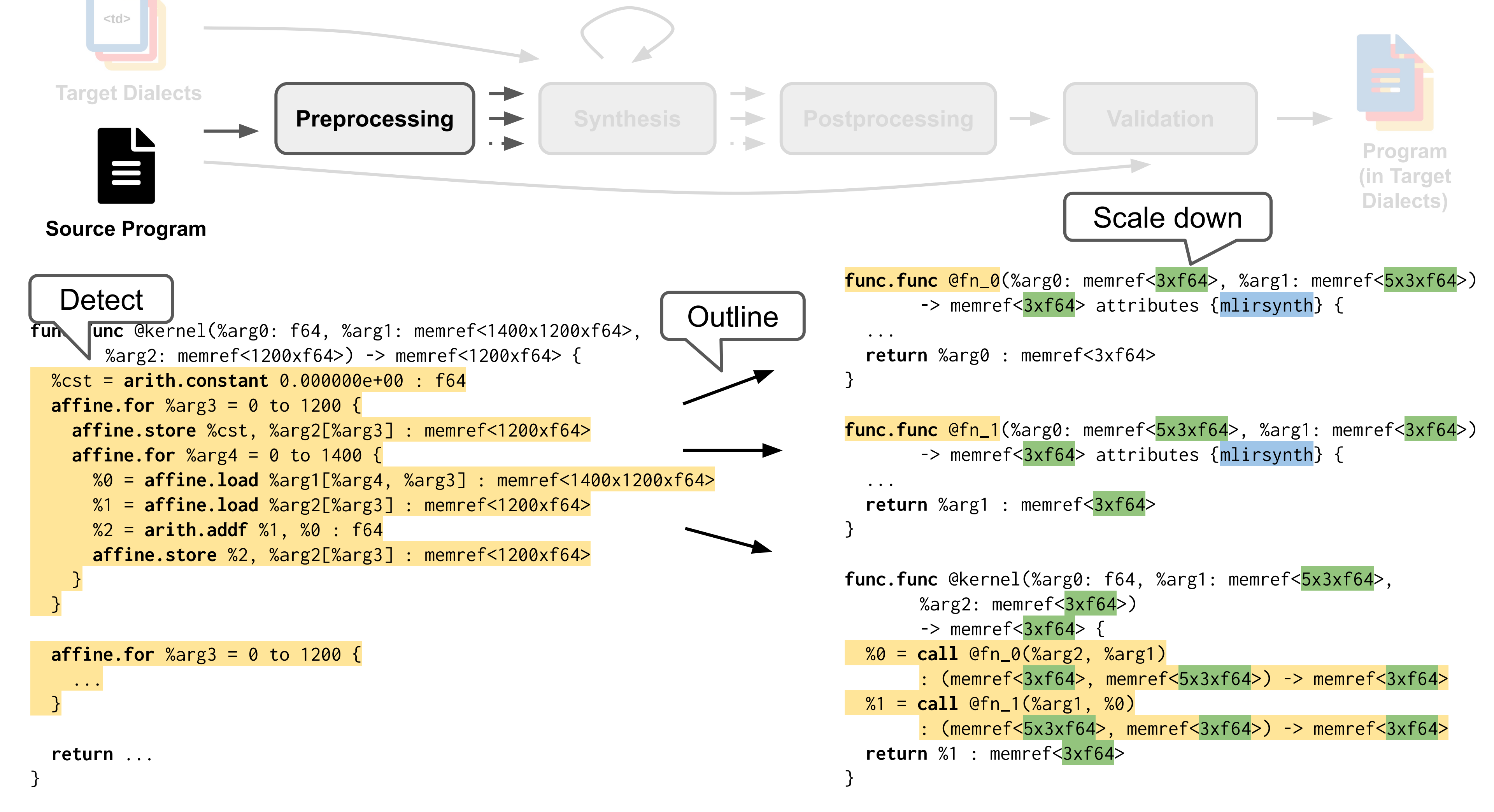
```c
void kernel(double arg0, double arg1[1400][1200],
            double arg2[1200]) {

  double cst = 0.0;
  for (int i=0; i<1200; i++) {
    arg2[i] = cst;

    for (int j=0; j<1400; j++) {
      arg2[i] += arg1[j][i];
    }
  }

  for (int i=0; i<1200; i++) {

  }
}
```

Polygeist C Frontend [1] →

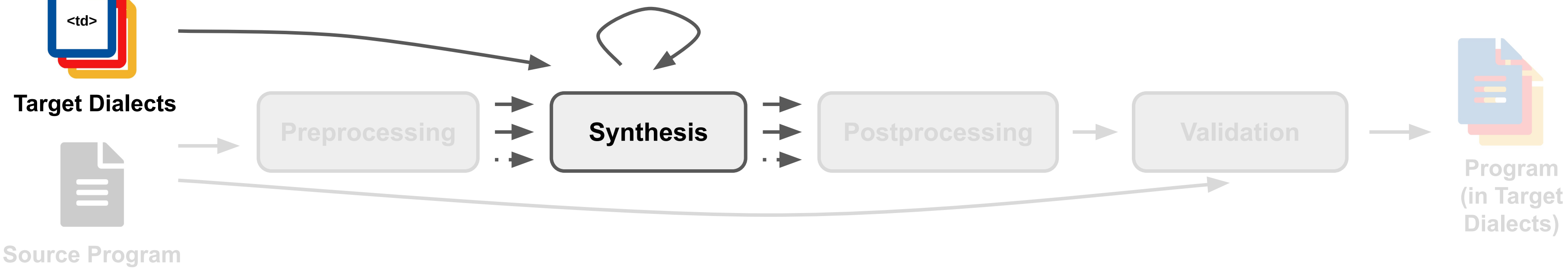
MLIR Code

```mlir
func.func @kernel(%arg0: f64, %arg1: memref<1400x1200xf64>,
       %arg2: memref<1200xf64>) -> memref<1200xf64> {
  %cst = arith.constant 0.000000e+00 : f64
  affine.for %arg3 = 0 to 1200 {
    affine.store %cst, %arg2[%arg3] : memref<1200xf64>
    affine.for %arg4 = 0 to 1400 {
      %0 = affine.load %arg1[%arg4, %arg3] : memref<1400x1200xf64>
      %1 = affine.load %arg2[%arg3] : memref<1200xf64>
      %2 = arith.addf %1, %0 : f64
      affine.store %2, %arg2[%arg3] : memref<1200xf64>
    }
  }

  affine.for %arg3 = 0 to 1200 {
    ...
  }

  return ...
}
```

[1] William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. Polygeist: Raising c to polyhedral mlir. In 2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 45–59. IEEE, 2021.



<td>
Target Dialects
Source Program
Preprocessing
Synthesis
Postprocessing
Validation
Program (in Target Dialects)
Detect
Outline
Scale down
fun unc @kernel(%arg0: f64, %arg1: memref<1400x1200xf64>,
%arg2: memref<1200xf64>) -> memref<1200xf64> {
%cst = arith.constant 0.000000e+00 : f64
affine.for %arg3 = 0 to 1200 {
affine.store %cst, %arg2[%arg3] : memref<1200xf64>
affine.for %arg4 = 0 to 1400 {
%0 = affine.load %arg1[%arg4, %arg3] : memref<1400x1200xf64>
%1 = affine.load %arg2[%arg3] : memref<1200xf64>
%2 = arith.addf %1, %0 : f64
affine.store %2, %arg2[%arg3] : memref<1200xf64>
}
}
affine.for %arg3 = 0 to 1200 {
...
}
return ...
}
func.func @fn_0(%arg0: memref<3xf64>, %arg1: memref<5x3xf64>)
-> memref<3xf64> attributes {mlirsynth} {
...
return %arg0 : memref<3xf64>
}
func.func @fn_1(%arg0: memref<5x3xf64>, %arg1: memref<3xf64>)
-> memref<3xf64> attributes {mlirsynth} {
...
return %arg1 : memref<3xf64>
}
func.func @kernel(%arg0: f64, %arg1: memref<5x3xf64>,
%arg2: memref<3xf64>)
-> memref<3xf64> {
%0 = call @fn_0(%arg2, %arg1)
: (memref<3xf64>, memref<5x3xf64>) -> memref<3xf64>
%1 = call @fn_1(%arg1, %0)
: (memref<5x3xf64>, memref<3xf64>) -> memref<3xf64>
return %1 : memref<3xf64>
}

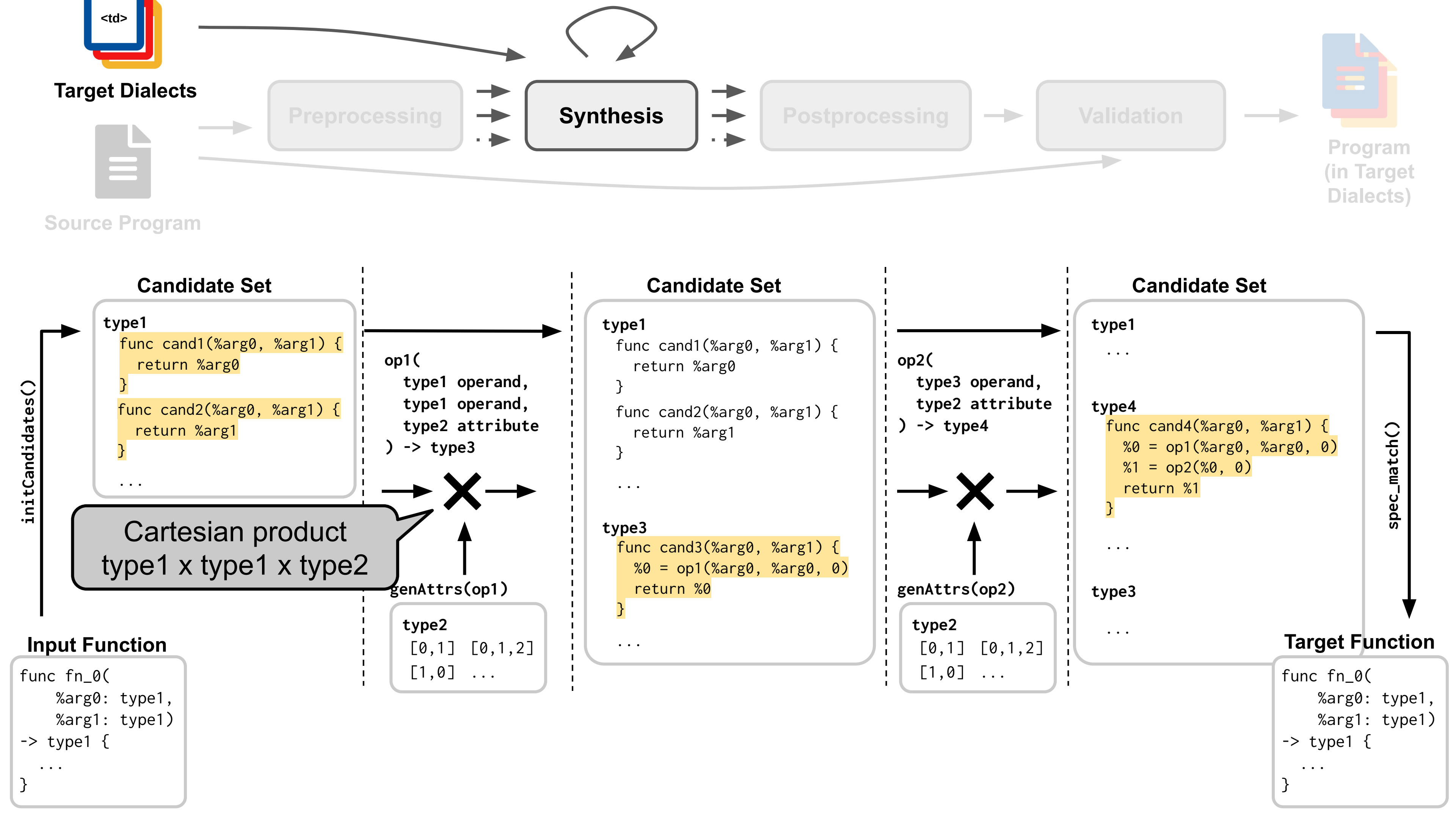


Algorithm 1 Core synthesis algorithm

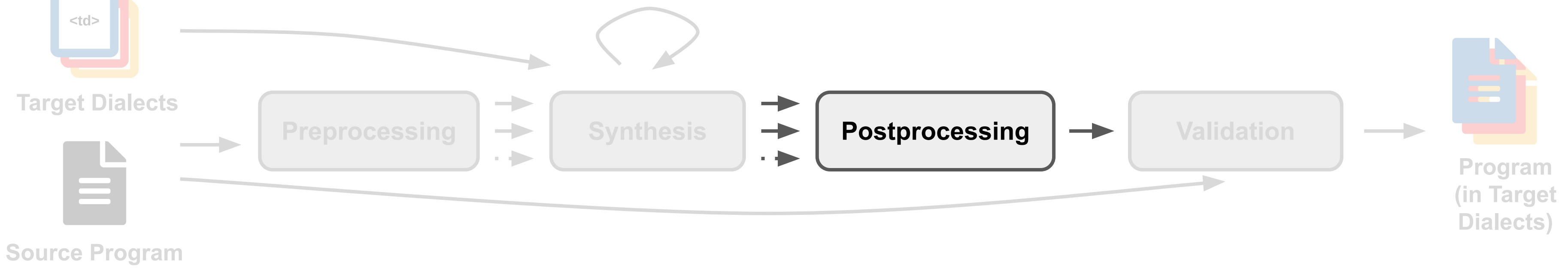
function SYNTHESIZE(f,G)
  $C \leftarrow \text{initCandidates}(f)$
  $I_n \leftarrow \text{genRandomInputs}(f, n)$
  $operations \leftarrow \text{pickOperations}(f, G)$
  while true **do**
    $f' \leftarrow \text{enumerate}(C, I_n, operations, f)$
    $I_N \leftarrow \text{genRandomInputs}(f, N)$
    if $\text{specCheck}(I_N, f, f')$ **then**
      return f'
    else
      $I_n \leftarrow \text{genRandomInputs}(f, n)$

Algorithm 2 Enumeration

function ENUMERATE(C, I_n, $operations$, f)
  while true **do**
    for op **in** $operations$ **do**
      $ops \leftarrow \text{filterTypes}(C, op)$
      $attr \leftarrow \text{genAttrs}(op)$
      $regs \leftarrow \text{genRegions}(op)$
      for f'**in** cartesianProduct(ops, $attr$, $regs$) **do**
        if not $\text{staticCheck}(f')$ **then**
          continue
        if $\text{observationallyUnique}(C, f')$ **then**
          $C \leftarrow C \cup f'$
        if $\text{specCheck}(I_n, f, f')$ **then**
          return f'



<td>
Target Dialects
Source Program
Preprocessing
Synthesis
Postprocessing
Validation
Program (in Target Dialects)
Candidate Set
type1
func cand1(%arg0, %arg1) {
return %arg0
}
func cand2(%arg0, %arg1) {
return %arg1
}
...
initCandidates()
Cartesian product
type1 x type1 x type2
op1(
type1 operand,
type1 operand,
type2 attribute
) -> type3
genAttrs(op1)
type2
[0,1] [0,1,2]
[1,0] ...
Candidate Set
type1
func cand1(%arg0, %arg1) {
return %arg0
}
func cand2(%arg0, %arg1) {
return %arg1
}
...
type3
func cand3(%arg0, %arg1) {
%0 = op1(%arg0, %arg0, 0)
return %0
}
...
op2(
type3 operand,
type2 attribute
) -> type4
genAttrs(op2)
type2
[0,1] [0,1,2]
[1,0] ...
Candidate Set
type1
...
type4
func cand4(%arg0, %arg1) {
%0 = op1(%arg0, %arg0, 0)
%1 = op2(%0, 0)
return %1
}
...
type3
...
spec_match()
Input Function
func fn_0(
%arg0: type1,
%arg1: type1)
-> type1 {
...
}
Target Function
func fn_0(
%arg0: type1,
%arg1: type1)
-> type1 {
...
}

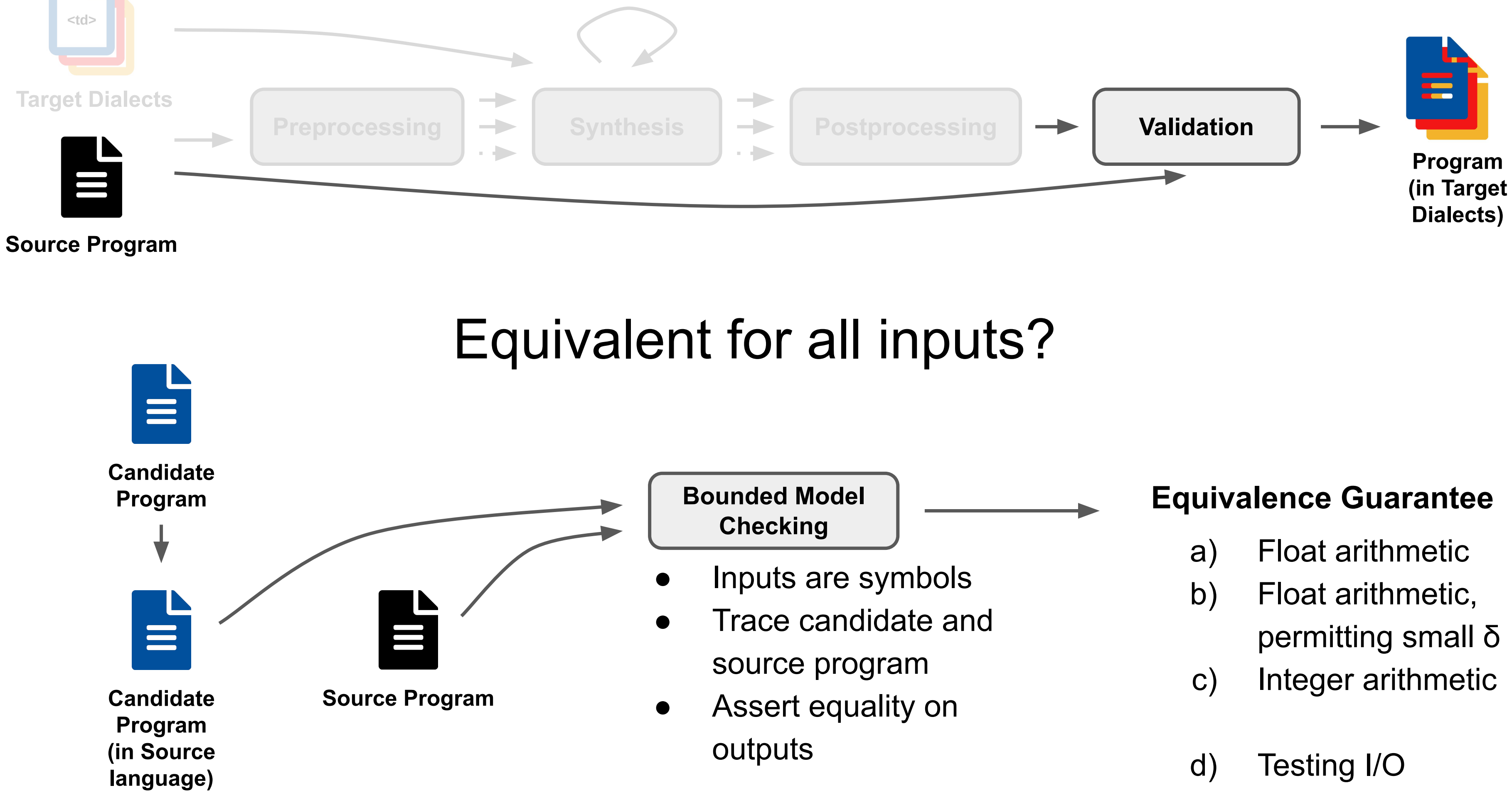



```
func.func @fn_0(%arg0: memref<3xf64>, %arg1: memref<5x3xf64>)
        -> memref<3xf64> attributes {mlirsynth} {
  %0 = op(%arg0, %arg1) : memref<3xf64>
  %1 = op(%0, %arg1) : memref<3xf64>
  return %1 : memref<3xf64>
}

func.func @fn_1(%arg0: memref<5x3xf64>, %arg1: memref<3xf64>)
        -> memref<3xf64> attributes {mlirsynth} {
  %0 = op(%arg0, %arg1) : memref<3xf64>
  return %0 : memref<3xf64>
}

func.func @kernel(%arg0: f64, %arg1: memref<5x3xf64>,
        %arg2: memref<3xf64>)
        -> memref<3xf64> {
  %0 = call @fn_0(%arg2, %arg1)
        : (memref<3xf64>, memref<5x3xf64>) -> memref<3xf64>
  %1 = call @fn_1(%arg1, %0)
        : (memref<5x3xf64>, memref<3xf64>) -> memref<3xf64>
  return %1 : memref<3xf64>
}
```

Inline

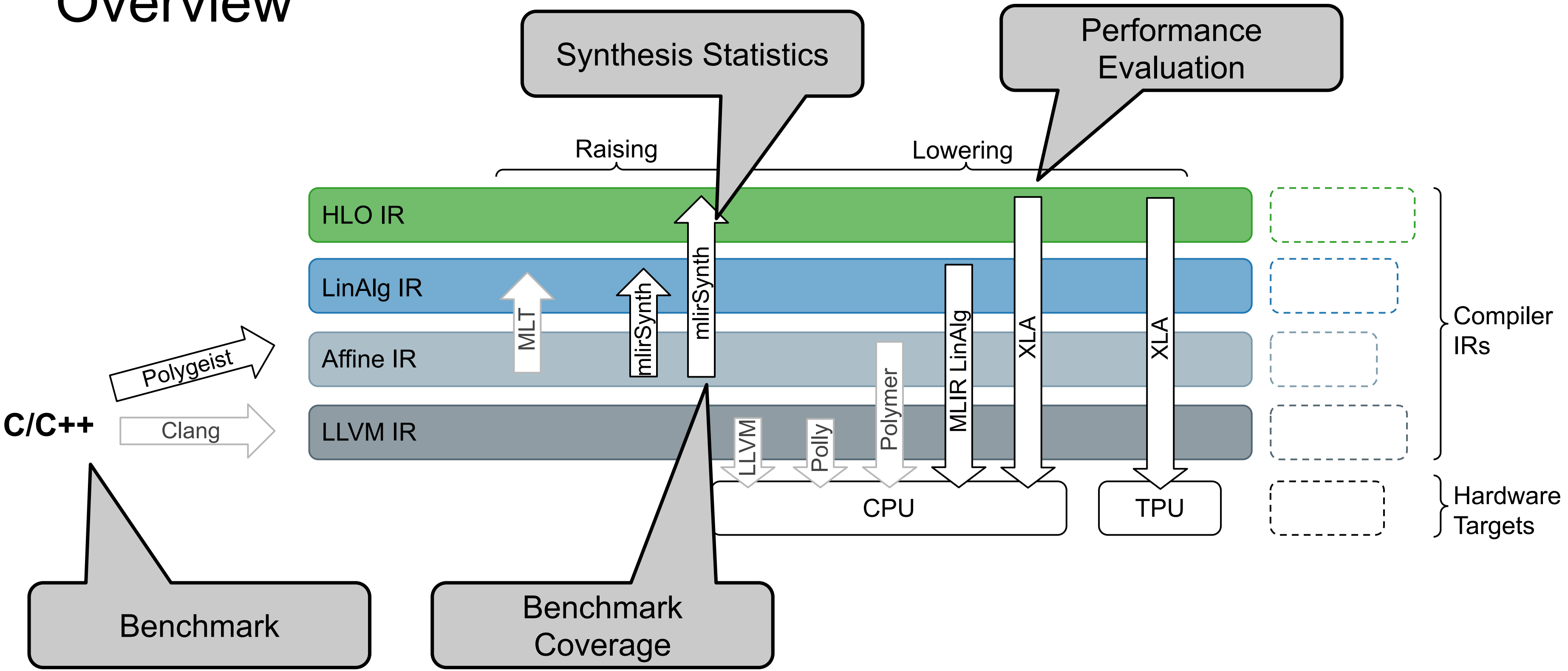
Scale up

```
func.func @kernel(%arg0: f64,
        %arg1: memref<1400x1200xf64>,
        %arg2: memref<1200xf64>)
      -> memref<1200xf64> {

  // fn_0
  %0 = op(%arg2, %arg1) : memref<1200xf64>
  %1 = op(%0, %arg1) : memref<1200xf64>

  // fn_1
  %2 = op(%arg1, %1) : memref<1200xf64>

  return %2 : memref<1200xf64>
}
```



<td>
Target Dialects
Source Program
Preprocessing
Synthesis
Postprocessing
Validation
Program (in Target Dialects)
Equivalent for all inputs?
Candidate Program
Candidate Program (in Source language)
Source Program
Bounded Model Checking
● Inputs are symbols
● Trace candidate and source program
● Assert equality on outputs
Equivalence Guarantee
a) Float arithmetic
b) Float arithmetic, permitting small δ
c) Integer arithmetic
d) Testing I/O

Overview

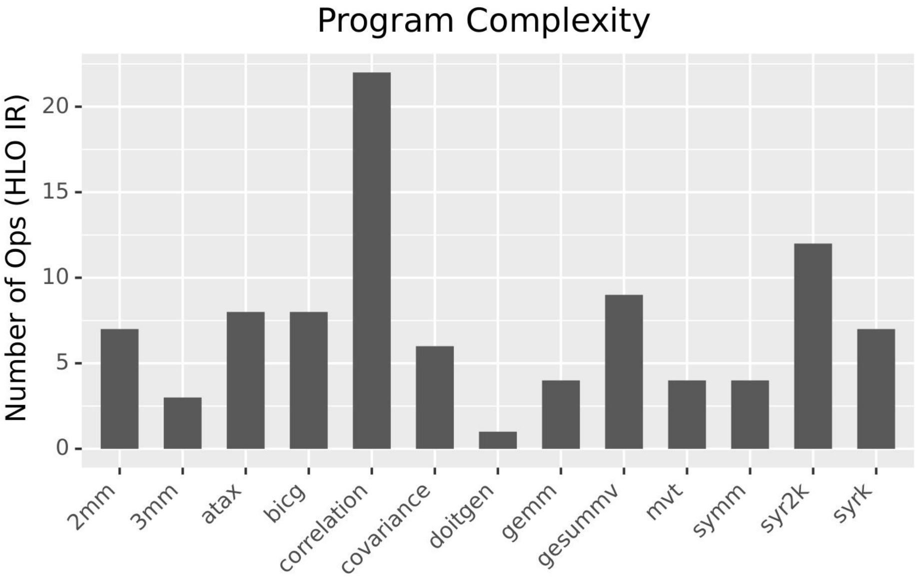

Synthesis Statistics
Performance Evaluation
Raising
Lowering
HLO IR
LinAlg IR
Affine IR
LLVM IR
MLT
mlirSynth
mlirSynth
Polygeist
Clang
C/C++
LLVM
Polly
Polymer
MLIR LinAlg
XLA
XLA
CPU
TPU
Compiler IRs
Hardware Targets
Benchmark
Benchmark Coverage

Evaluation

Benchmark

3 Categories from PolyBench
→ 14 Programs

- Solvers
- Data Mining
- Linear Algebra BLAS
- Linear Algebra Kernels
- Stencils
- Medley




```
for (j = 0; j < m; j++) {
  mean[j] = 0.0;
  for (i = 0; i < n; i++)
    mean[j] += data[i][j];
  mean[j] /= float_n;
}

for (j = 0; j < m; j++) {
  stddev[j] = 0.0;
  for (i = 0; i < n; i++)
    stddev[j] += (data[i][
               * (data[i
  stddev[j] /= float_n;
  stddev[j] = sqrt(stddev[

  stddev[j] = stddev[j] <=
}

for (i = 0; i < n; i++)
  for (j = 0; j < m; j++)
    data[i][j] -= mean[j];
    data[i][j] /= sqrt(flo
  }

for (i = 0; i < m - 1; i++
  corr[i][i] = 1.0;
  for (j = i + 1; j < m; j
    corr[i][j] = 0.0;
    for (k = 0; k < n; k++
      corr[i][j] += (data[
    corr[j][i] = corr[i][j
  }
}
corr[m - 1][m - 1] = 1.0;
```

```
for (i = 0; i < _PB_N; i++)
  y[i] = 0;
for (i = 0; i < _PB_M; i++) {
  tmp[i] = SCALAR_VAL(0.0);
  for (j = 0; j < _PB_N; j++)
    tmp[i] = tmp[i] + A[i][j] * x[j];
  for (j = 0; j < _PB_N; j++)
    y[j] = y[j] + A[i][j] * tmp[i];
}
```

```
for (j = 0; j < _PB_M; j++) {
  mean[j] = SCALAR_VAL(0.0);
  for (i = 0; i < _PB_N; i++)
    mean[j] += data[i][j];
  mean[j] /= float_n;
}

for (i = 0; i < _PB_N; i++)
  for (j = 0; j < _PB_M; j++)
    data[i][j] -= mean[j];
```

Synthesis Statistics

Static checks rule out ~95% candidates

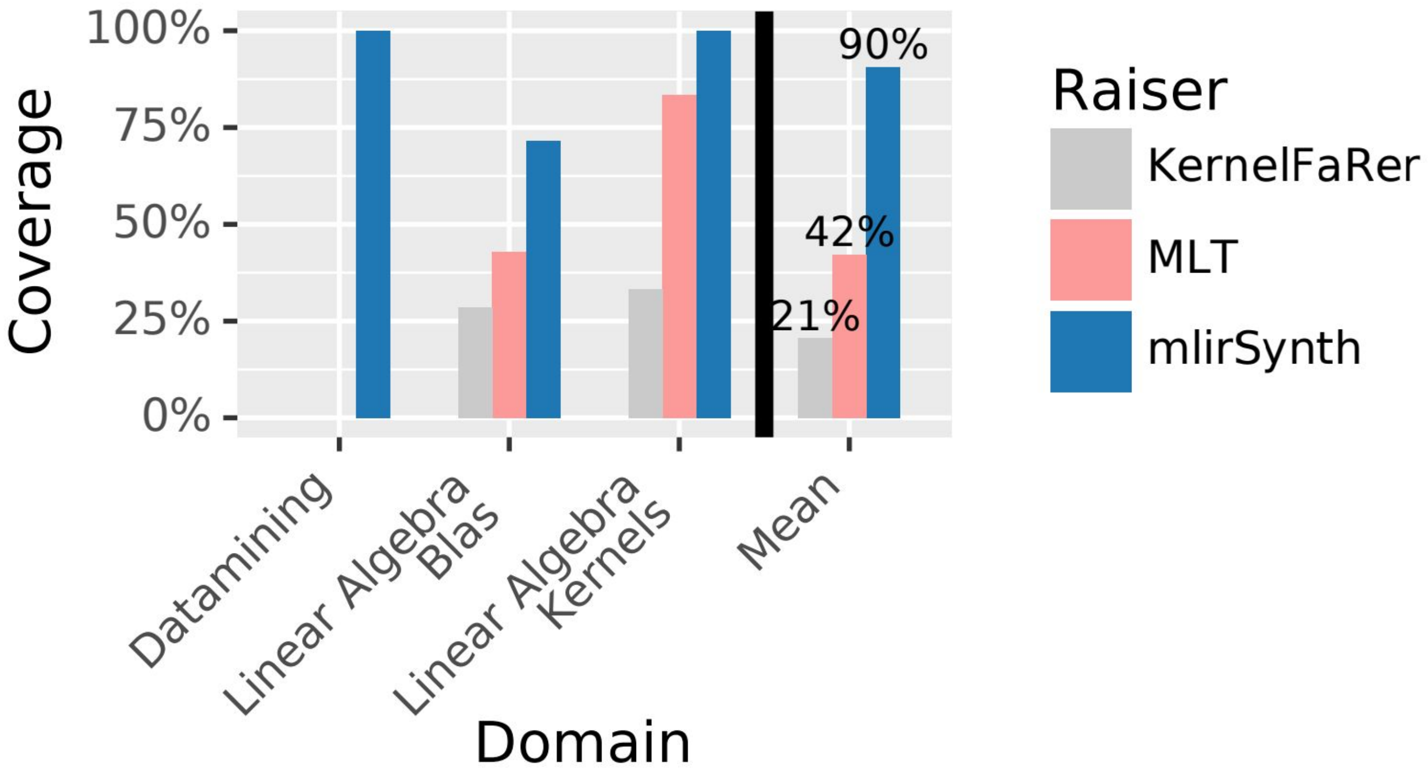
½ of benchmarks synthesize in < 2 seconds

Benchmark	Enumerated	Static filtered	Evaluated	Equiv filtered	Ops (max)	Synth time
2mm	49067	46504	1043	709	7 (3)	0.65s
3mm	2484	2409	3	0	3 (1)	0.14s
atax	18960	17042	1166	763	8 (3)	0.62s
bicg	18961	17046	1173	771	8 (3)	0.59s
correlation	1420241	1173035	188577	159679	22 (3)	174.11s
covariance	382674	374083	5799	2049	6 (3)	4.21s
doitgen	9972	9879	71	18	1 (1)	0.16s
gemm	607638	572798	13695	6745	4 (3)	7.26s
gesummv	29221	24566	3919	2333	9 (3)	1.37s
mvt	27977	24460	2855	1631	4 (2)	1.09s
symm	5353361	4943595	309752	163310	4 (4)	134.85s
syr2k	20820281	18547932	1467901	1022725	12 (5)	2438.69s
syrk	3532229	2954620	433798	297594	7 (5)	467.79s

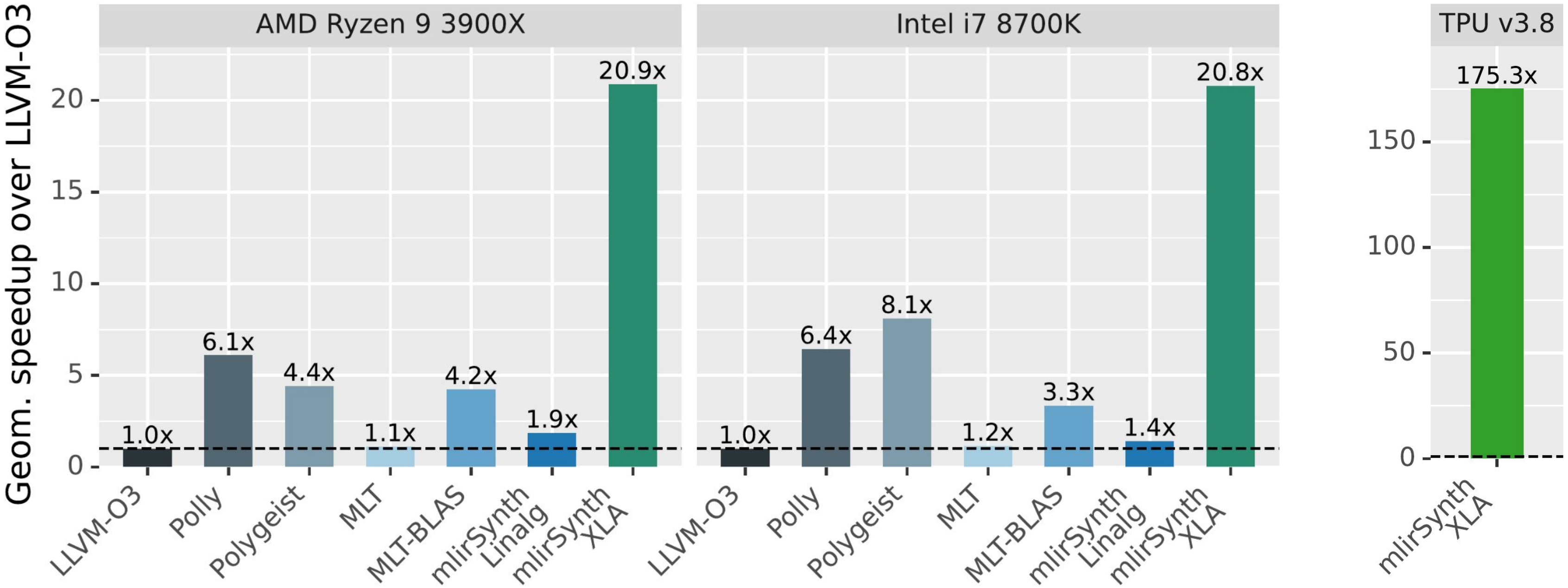
Equivalence filter rules out additional candidates

Synth time correlated to largest synth subproblem

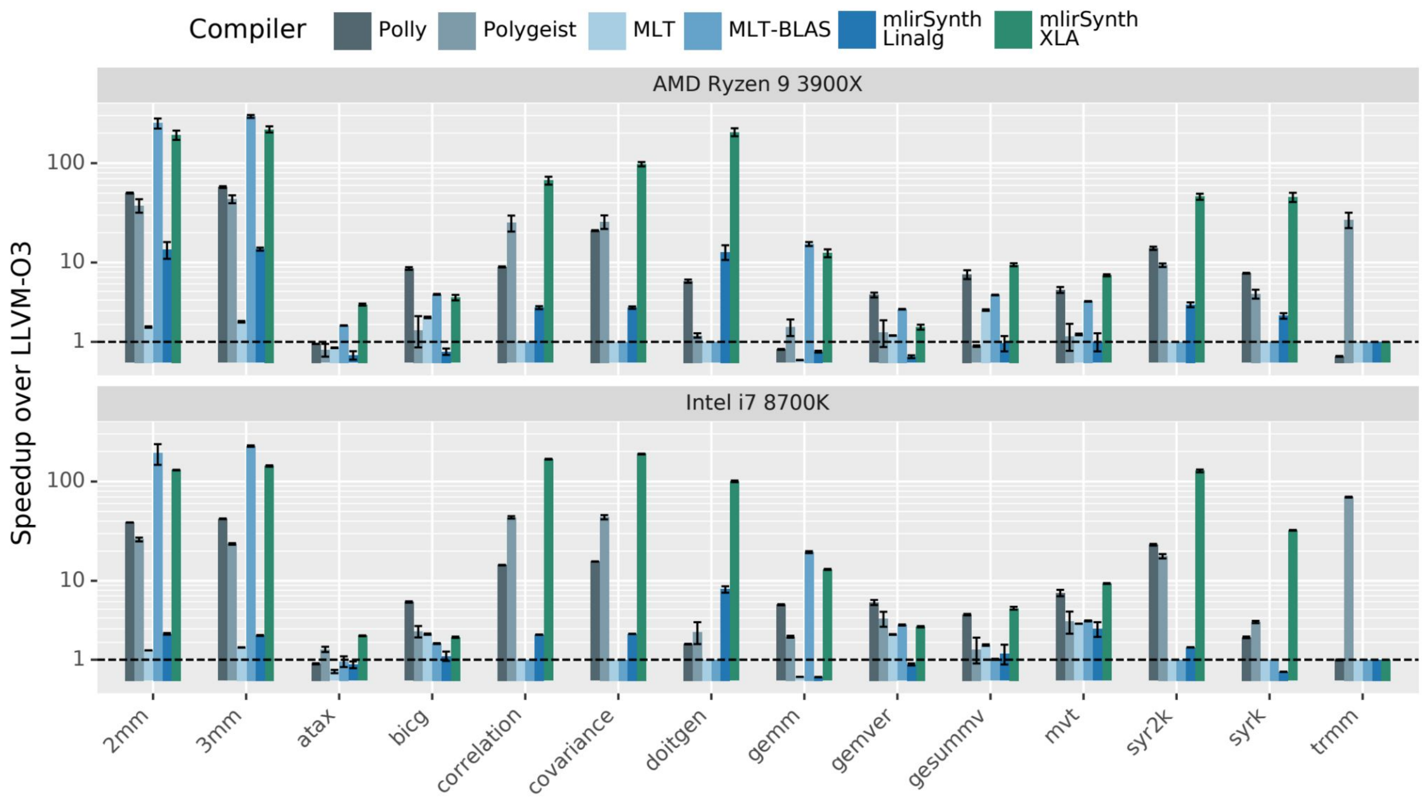
Benchmark Coverage



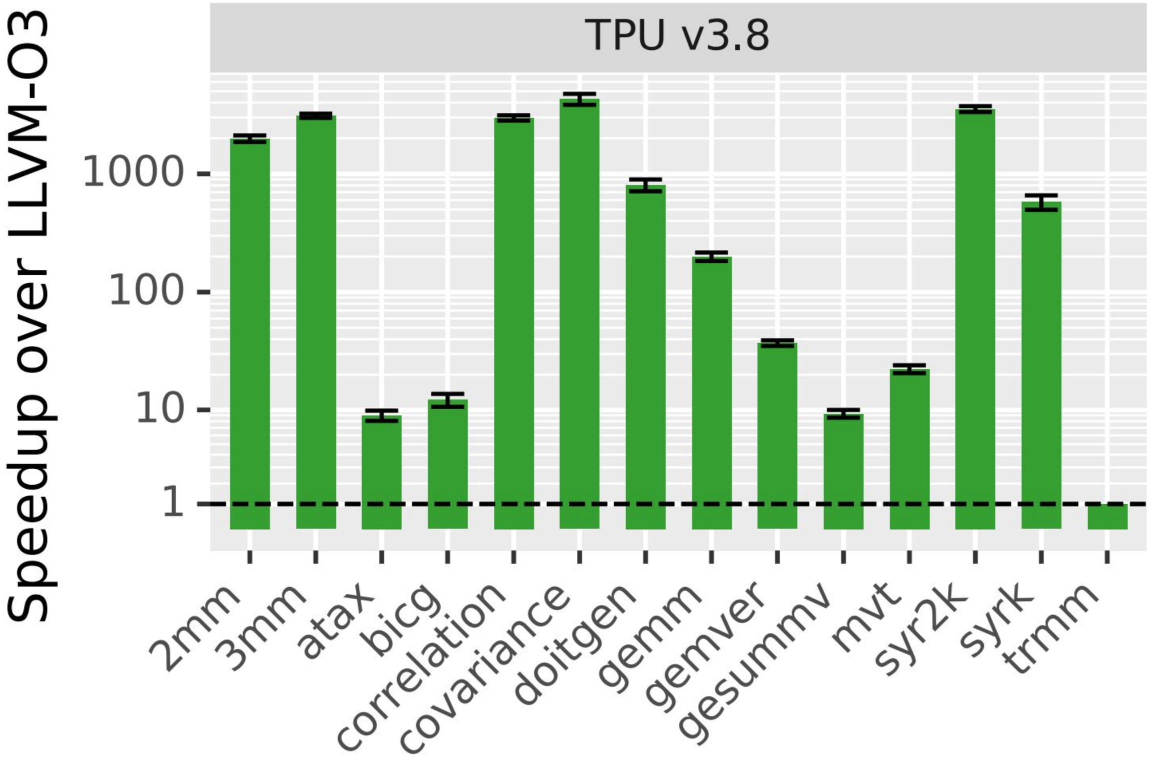
Performance



Performance (detailed)



Performance (detailed)



CPU: AMD Ryzen 9 3900X
TPU: TPU v2.8

Summary

- mlirSynth raises programs to high-level dialects using program synthesis
  - Automatic ✅
  - Robust ✅
  - Fast ❌

- Raised programs lead to significantly higher performance
  - Domain-specific optimizations
  - Kernel libraries
  - Hardware accelerators

Future Work

Method

- Speed up synthesis with neural guides

Applicability

- Support more source languages
- More target dialects / domains