

Polygeist: Affine C in MLIR



William S. Moses
wmoses@mit.edu



Lorenzo Chelini
l.chelini@tue.nl



Ruizhe Zhao
rz3515@ic.ac.uk

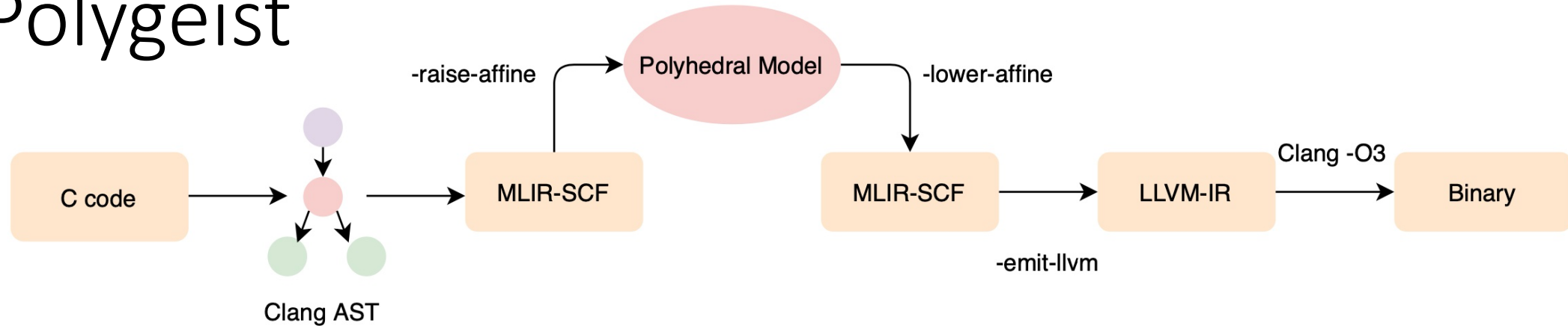


Alex Zinenko
zinenko@google.com

Motivation

- ✓ The compiler research has recently been enamored by the MLIR framework, whose first-class polyhedral representation may provide benefits on a variety of codes
- ✓ We can fully leverage decades of polyhedral research by connecting MLIR with existing polyhedral tools first.
- ✓ Without MLIR-versions of standard polyhedral benchmarks, one cannot perform a fair assessment
- ⚠ Goal of this work is not to use polyhedral tools to speedup MLIR, but to provide a fair baseline for subsequent work

Polygeist



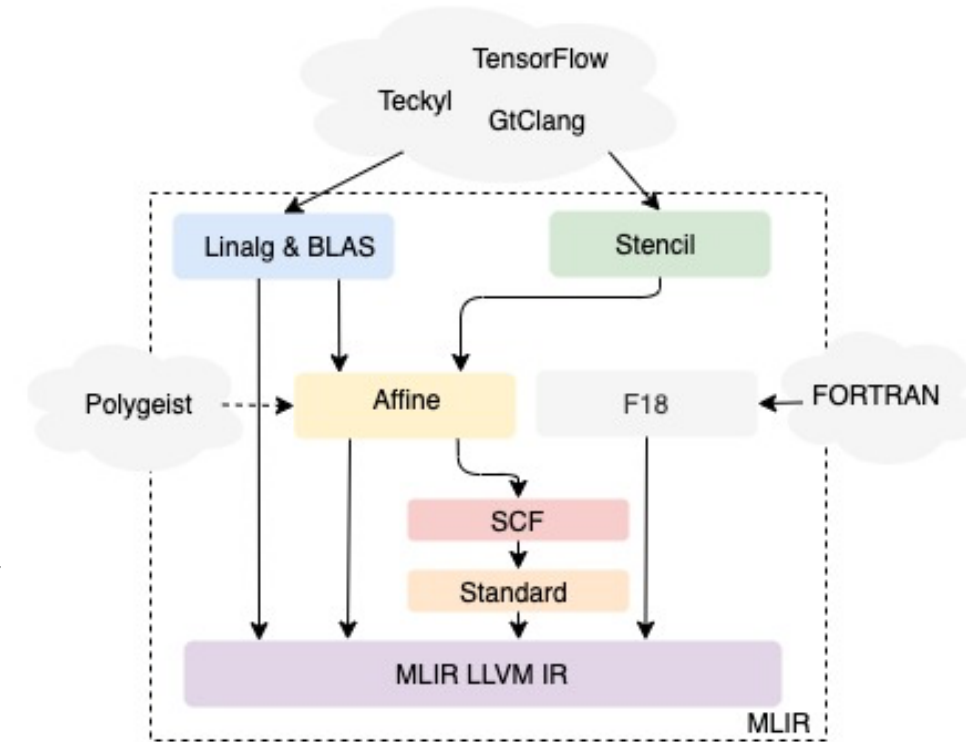
A platform to establish baselines for polyhedral transformations within MLIR

- Generic C or C++ frontend that generates "standard" MLIR
- Raising transformations for transforming "standard" MLIR to polyhedral MLIR (Affine)
- Embedding of existing polyhedral tools (Pluto, CLoog) into MLIR
- Polyhedral benchmarks for MLIR based off of Polybench
- End-to-end evaluation on standard polyhedral benchmarks

The MLIR Framework

- A toolkit for representing and transforming "code"
 - Modular and extensible via dialects (namespaces of operations/types and attributes)
 - Non-opinionated – choose the level of abstraction that is right for you
 - State-of-the-art SSA-based compiler technology

```
%result = "dialect.operation"(%operand, %operand)
    {attribute = #dialect<"value">} ({
^basic_block(%block_argument: !dialect.type):
    "another.operation"() : () -> ()
}) : (!dialect.type) -> !dialect.result_type
```

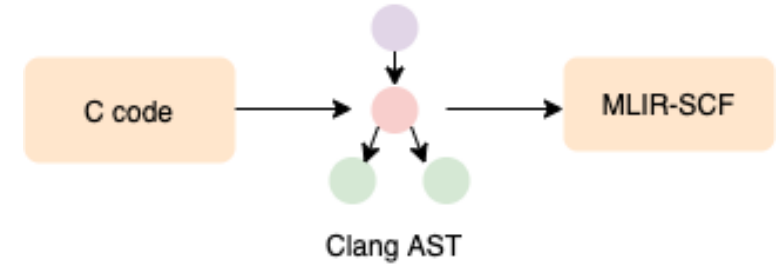


The Affine dialect

- Represent SCoP with polyhedral-friendly loops and conditions
- Core Affine representation
 - Symbols - parameters
 - Dimensions - symbol extension that accepts induction variables
 - Maps - multi-dimensional function of symbols and dimensions
 - Sets - integer tuples constrained by a conjunction

```
%c0 = constant 0 : index
%0 = dim %A, %c0 : memref<?xf32>
%1 = dim %B, %c0 : memref<?xf32>
affine.for %i = 0 to affine_map<() [s0] -> (s0)>()[%0] {
  affine.for %j = 0 to affine_map<() [s0] -> (s0)>()[%1] {
    %2 = affine.load %A[%i] : memref<?xf32>
    %3 = affine.load %B[%j] : memref<?xf32>
    %4 = mulf %2, %3 : f32
    %5 = affine.load %C[%i + %j] : memref<?xf32>
    %6 = addf %4, %5 : f32
    affine.store %6, %C[%i + %j] : memref<?xf32>
  }
}
```

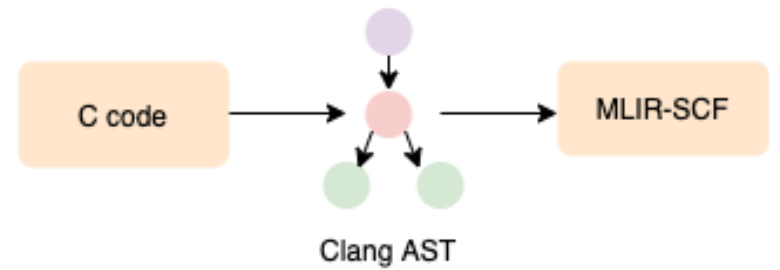
Polygeist Frontend



- Built a generic C or C++ frontend for MLIR, based off of Clang
- C control flow directly lowered to MLIR for, if, etc..
- Variables and arrays represented by MLIR memref (memory reference) construct

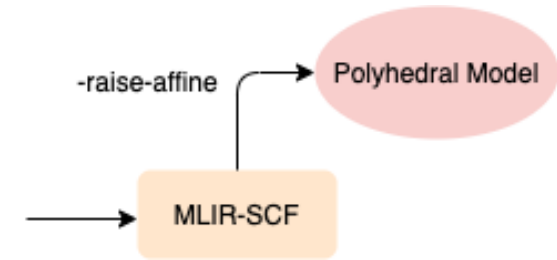
Polygeist Frontend

```
void set(int *arr, int val) {  
    #pragma scop  
    for(int i=0; i<10; i++){  
        arr[2*i] = val;  
    }  
    #pragma endscop  
}
```



```
func @set(%arg0: memref<?xi32>, %arg1: i32) {  
    %c0 = constant 0 : index  
    %0 = alloca() : memref<1xmemref<?xi32>>  
    store %arg0, %0[%c0] : memref<1xmemref<?xi32>>  
    %1 = alloca() : memref<1xi32>  
    store %arg1, %1[%c0] : memref<1xi32>  
    %c0_i32 = constant 0 : i32  
    %c2_i32 = constant 2 : i32  
    %c10_i32 = constant 10 : i32  
    %2 = index_cast %c10_i32 : i32 to index  
    affine.for %arg2 = %c0_i32 to %2 {  
        %3 = index_cast %arg2 : index to i32  
        %4 = alloca() : memref<1xi32>  
        store %3, %4[%c0] : memref<1xi32>  
        %5 = load %0[%c0] : memref<1xmemref<?xi32>>  
        %6 = load %4[%c0] : memref<1xi32>  
        %7 = muli %c2_i32, %6 : i32  
        %8 = index_cast %7 : i32 to index  
        %9 = load %1[%c0] : memref<1xi32>  
        store %9, %5[%8] : memref<?xi32>  
    }  
    return  
}
```

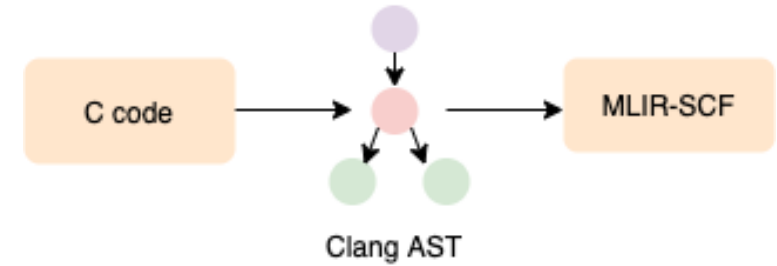
Polygeist Raising



- Directly lowered constructs are not valid polyhedral programs
- Local variables eliminated, if possible, by new MLIR mem2reg pass
- Loads and stores are raised to affine loads, if possible
 - Detect if index calculation is a valid affine expression
 - Progressively fold index calculation into an affine operation
- if statements are changed to affine if their condition can be raised

Polygeist Raising

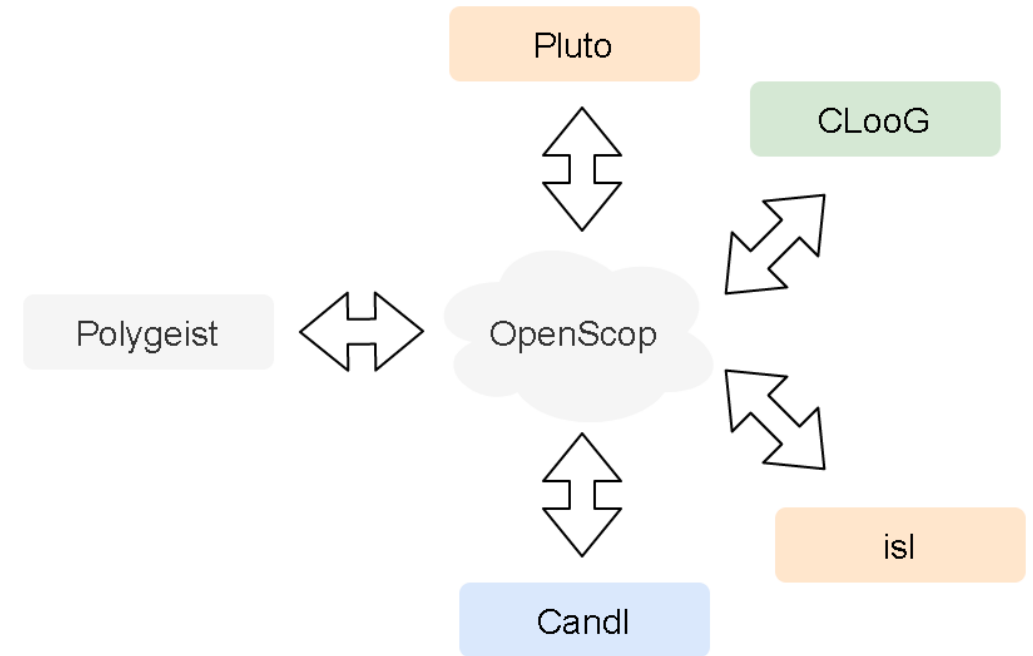
```
func @set(%arg0: memref<?xi32>, %arg1: i32) {  
  %c0 = constant 0 : index  
  %0 = alloca() : memref<1xmemref<?xi32>>  
  store %arg0, %0[%c0] : memref<1xmemref<?xi32>>  
  %1 = alloca() : memref<1xi32>  
  store %arg1, %1[%c0] : memref<1xi32>  
  %c0_i32 = constant 0 : i32  
  %c10_i32 = constant 10 : i32  
  %2 = index_cast %c10_i32 : i32 to index  
  affine.for %arg2 = %c0_i32 to %2 {  
    %3 = index_cast %arg2 : index to i32  
    %4 = alloca() : memref<1xi32>  
    store %3, %4[%c0] : memref<1xi32>  
    %5 = load %0[%c0] : memref<1xmemref<?xi32>>  
    %c2_i32 = constant 2 : i32  
    %6 = load %4[%c0] : memref<1xi32>  
    %7 = muli %c2_i32, %6 : i32  
    %8 = index_cast %7 : i32 to index  
    %9 = load %1[%c0] : memref<1xi32>  
    store %9, %5[%8] : memref<?xi32>  
  }  
  return  
}
```



```
func @set(%arg0: memref<?xi32>, %arg1: i32) {  
  affine.for %arg2 = 0 to 10 {  
    affine.store %arg1, %arg0[%arg2 * 2]  
      : memref<?xi32>  
  }  
  return  
}
```

Connecting MLIR to Polyhedral Tools

- Polygeist can obtain polyhedral representation in MLIR Affine
- But it is difficult to leverage existing polyhedral tools
- OpenScop is the interchangeable format among polyhedral tools
- How to translate between MLIR code and OpenScop representation?

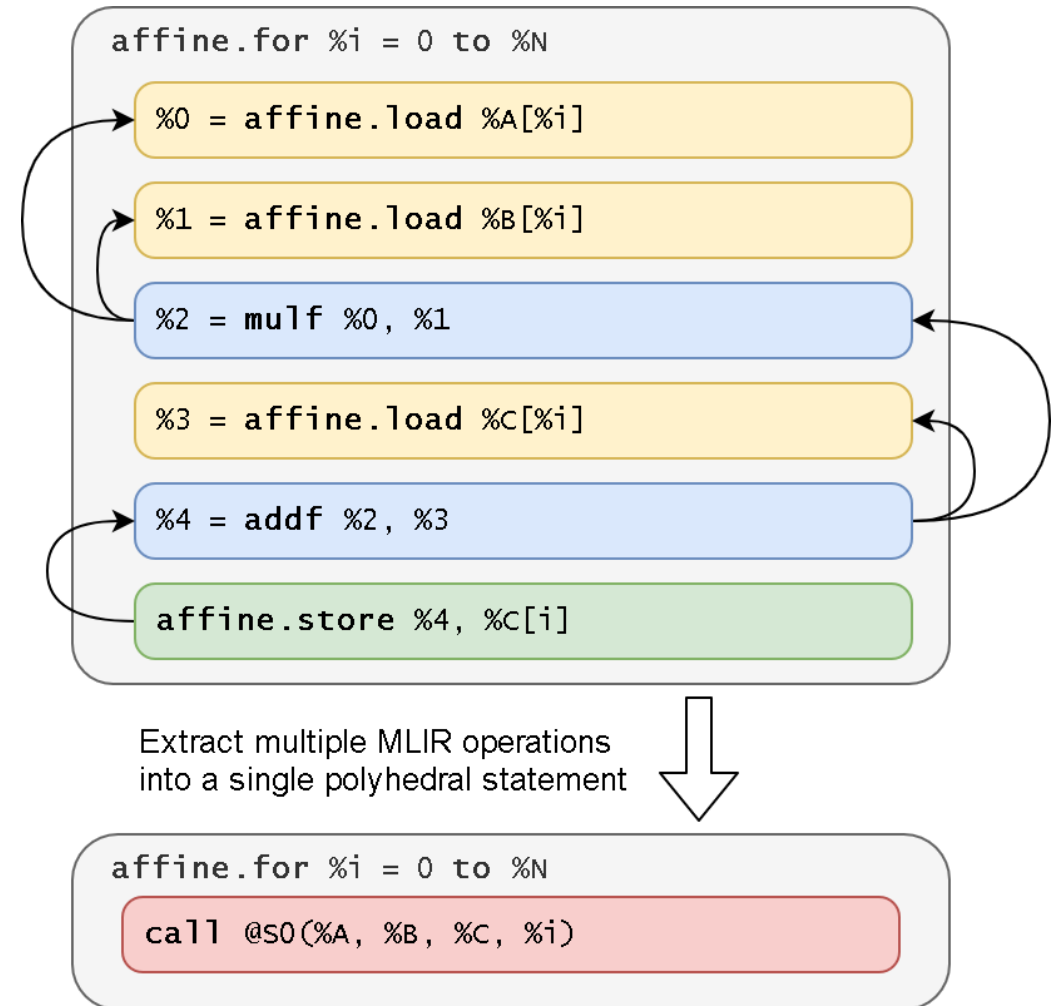


Polyhedral Statement

- OpenScop expects C-like statements:

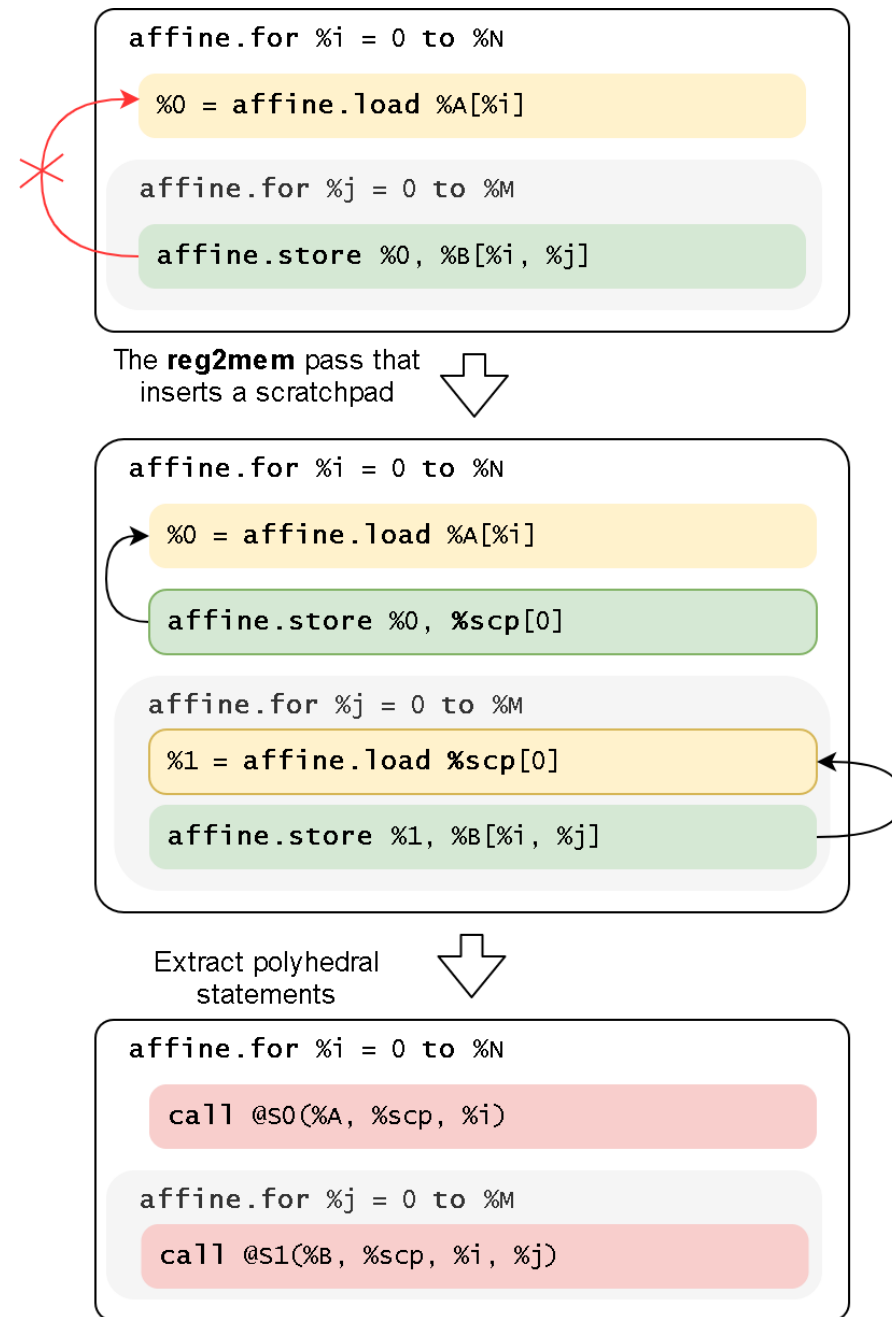
```
C[i][j] += A[i][k] * B[k][j]
```

- MLIR is lower level and a store instruction alone does not specify how to compute the stored operand
- 1 OpenScop statement may correspond to N MLIR operations
- To match C-like statements:
 - Extract 1 MLIR memory write
 - Traverse SSA use-def chains
 - Continue until all operations are loads or symbols



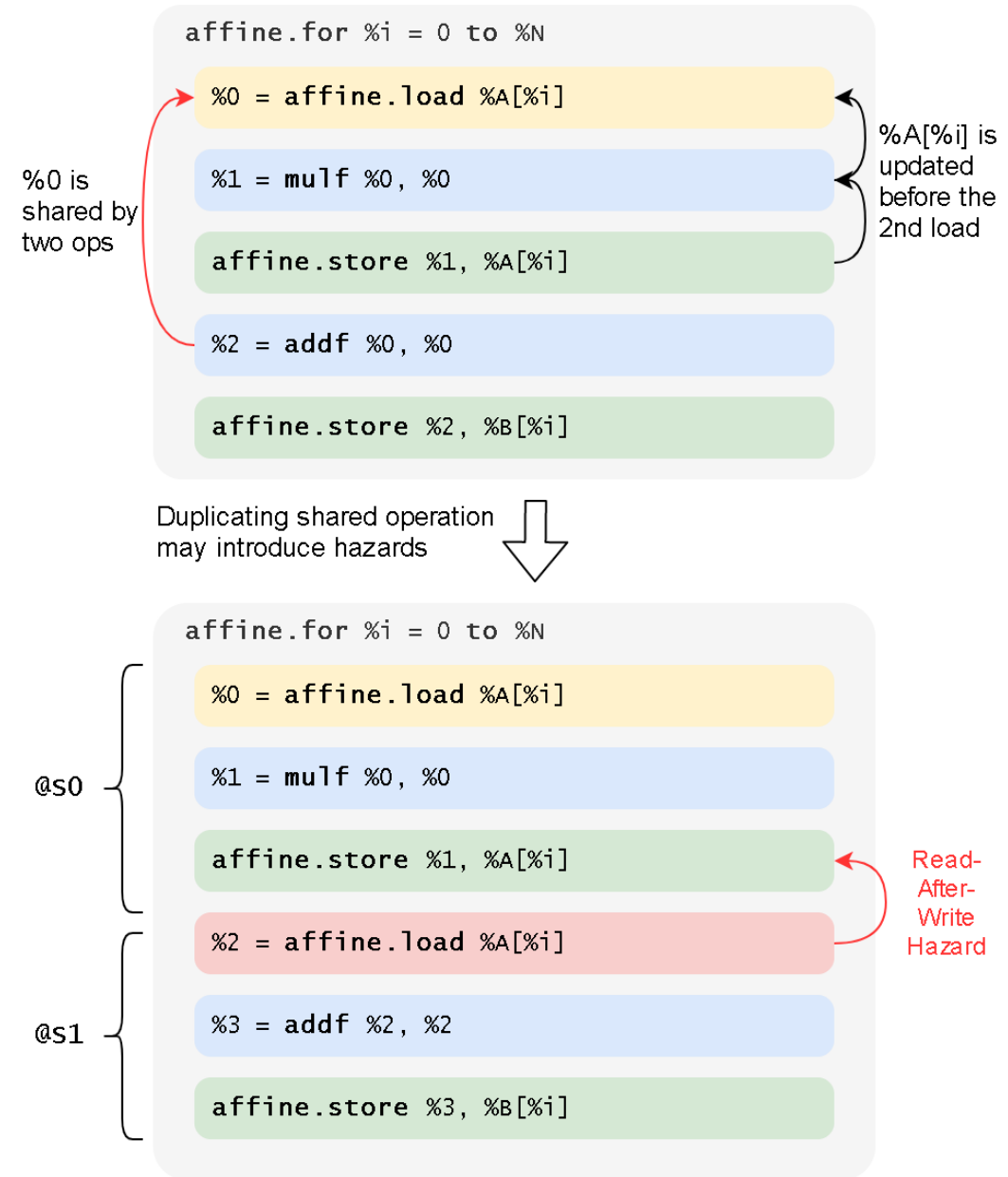
Region-Spanning Problem

- A use-def chain may span multiple loops (regions).
 - e.g., A load op defines a register used by other ops in inner loops.
- Statement nesting in loops is ambiguous
- Difficult to reconstruct when converting back to MLIR
- Reg2mem pass: insert a scratchpad for each use-def across regions



Avoid RAW Hazard

- The RAW hazard problem:
 - A load op is duplicated for use in multiple statements
 - Intermediate writes may clobber
 - After extraction, later statements may load wrong values
- Simplified value analysis to detect
- Insert scratchpads

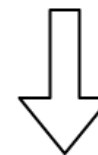


Outlining

- We outline statements into functions
- Opaque calls with known memory footprints
- Lift local stack allocations and symbol definitions

```
func @S0(%A: memref<?xf32>) {  
    %c0 = constant 0 : index  
    %s0 = dim %A, %c0 : index  
  
    %1 = affine.load %A[0]  
    affine.store %1, %A[symbol(%s0) - 1]  
    return  
}
```

Lift local symbols to the
function interface



```
func @S0(%A: memref<?xf32>, %s0: index) {  
    %0 = affine.load %A[0]  
    affine.store %0, %A[%s0 - 1]  
    return  
}
```

Translate to OpenScop

- First pre-process MLIR Affine code by previous passes
- For each extracted polyhedral statement:
 - Domain: get constraints from affine.for/if
 - Initial Schedule: derive from region nesting and operation order
 - Access: extract from affine load/stores
- Store symbols in OpenScop extensions

Translate to OpenScop

```
affine.for %i = 0 to %N
```

```
  affine.for %j = 0 to %N
```

```
    call @S0(%A, %i, %j)
```

```
func @S0(%A: memref<?x?xf32>, %i: index,
        %j: index) {
  %0 = affine.load %A[%i, %j]
  %1 = mul f %0, %0
  affine.store %1, %A[%i, %j]
  return
}
```

Domain

#	e/i	%i	%j	%N	1	
1	1	0	0	0	0	## %i >= 0
1	-1	0	1	-1	-1	## -%i+%N-1 >= 0
1	0	1	0	0	0	## %j >= 0
1	0	-1	1	-1	-1	## -%j+%N-1 >= 0

Scattering

#	e/i	s1	s2	s3	s4	s5	%i	%j	%N	1
0	-1	0	0	0	0	0	0	0	0	0
0	0	-1	0	0	0	0	1	0	0	0
0	0	0	-1	0	0	0	0	0	0	0
0	0	0	0	-1	0	0	0	1	0	0
0	0	0	0	0	-1	0	0	0	0	0

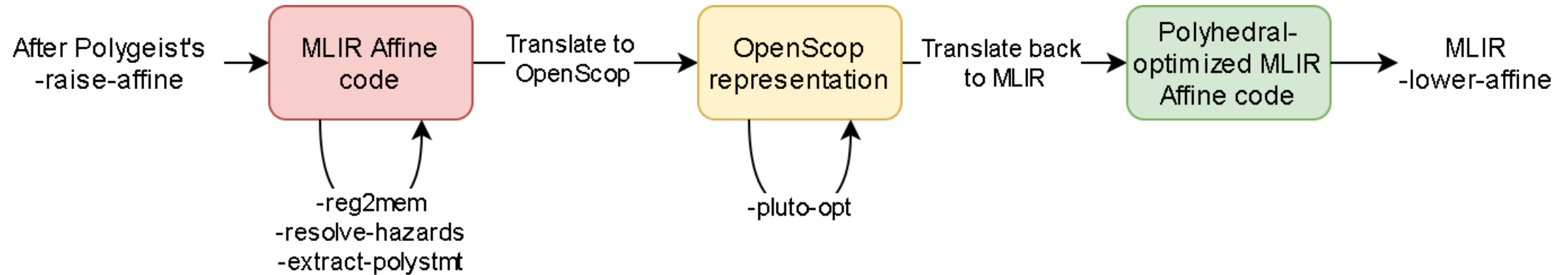
READ/WRITE Accesses

#	e/i	Arr	[1]	[2]	%i	%j	%N	1	
0	-1	0	0	0	0	0	0	0	## %A
0	0	-1	0	1	0	0	0	0	## %i
0	0	0	-1	0	1	0	0	0	## %j

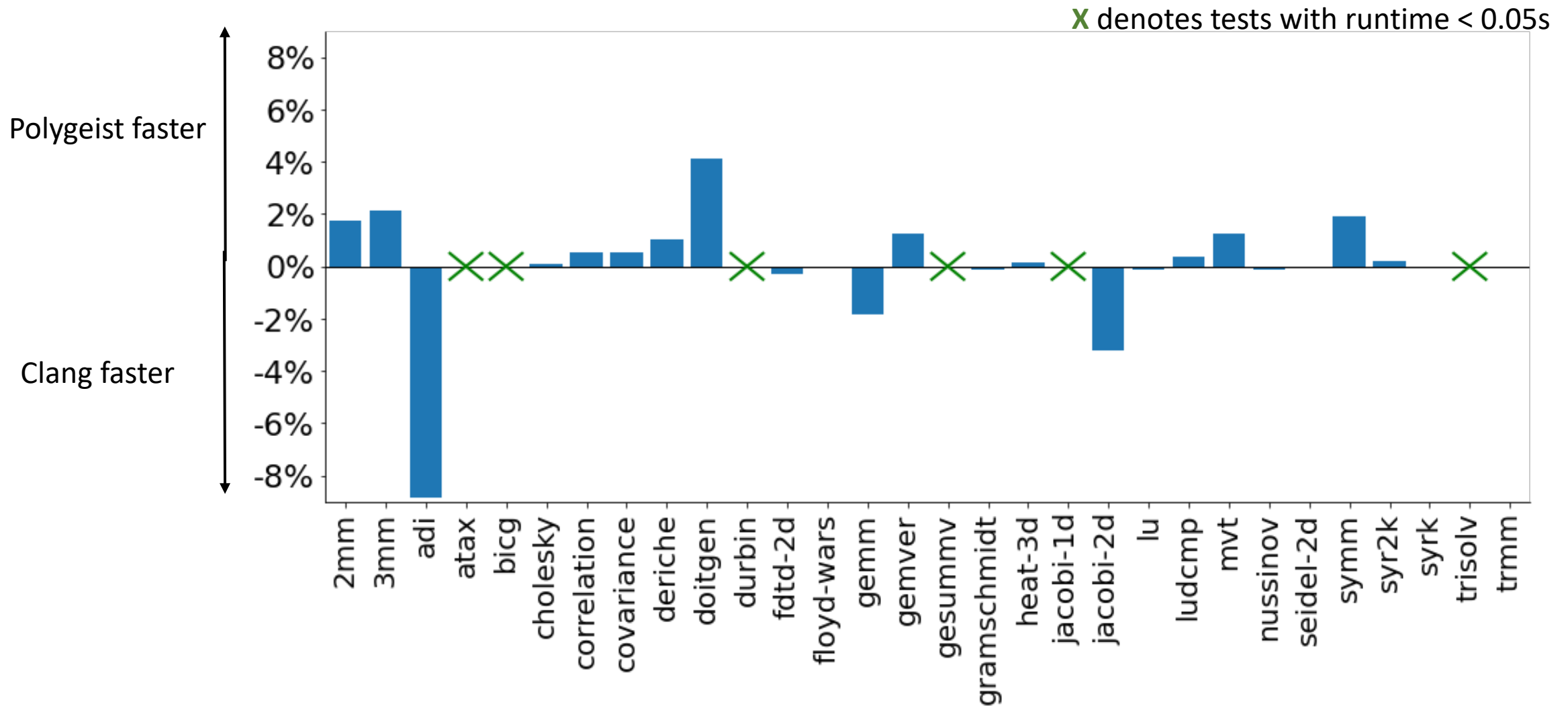
Regenerate MLIR Code

- Obtain a CLooG AST from an optimized OpenScop representation
- Regenerate MLIR code by traversing AST
- OpenScop symbols will be translated to MLIR values or operations based on a maintained symbol table.

Polyhedral Optimization Pipeline



Frontend Comparison with Clang



Frontend Performance Differences

- Solved differences (removed prior to benchmarking):
 - 8% performance boost on Floyd-Warshall occurs if Polygeist generates a single MLIR module for both benchmarking and timing code by default
 - MLIR doesn't properly generate LLVM datalayout, preventing vectorization for MLIR-generated code (patched in our lowering)

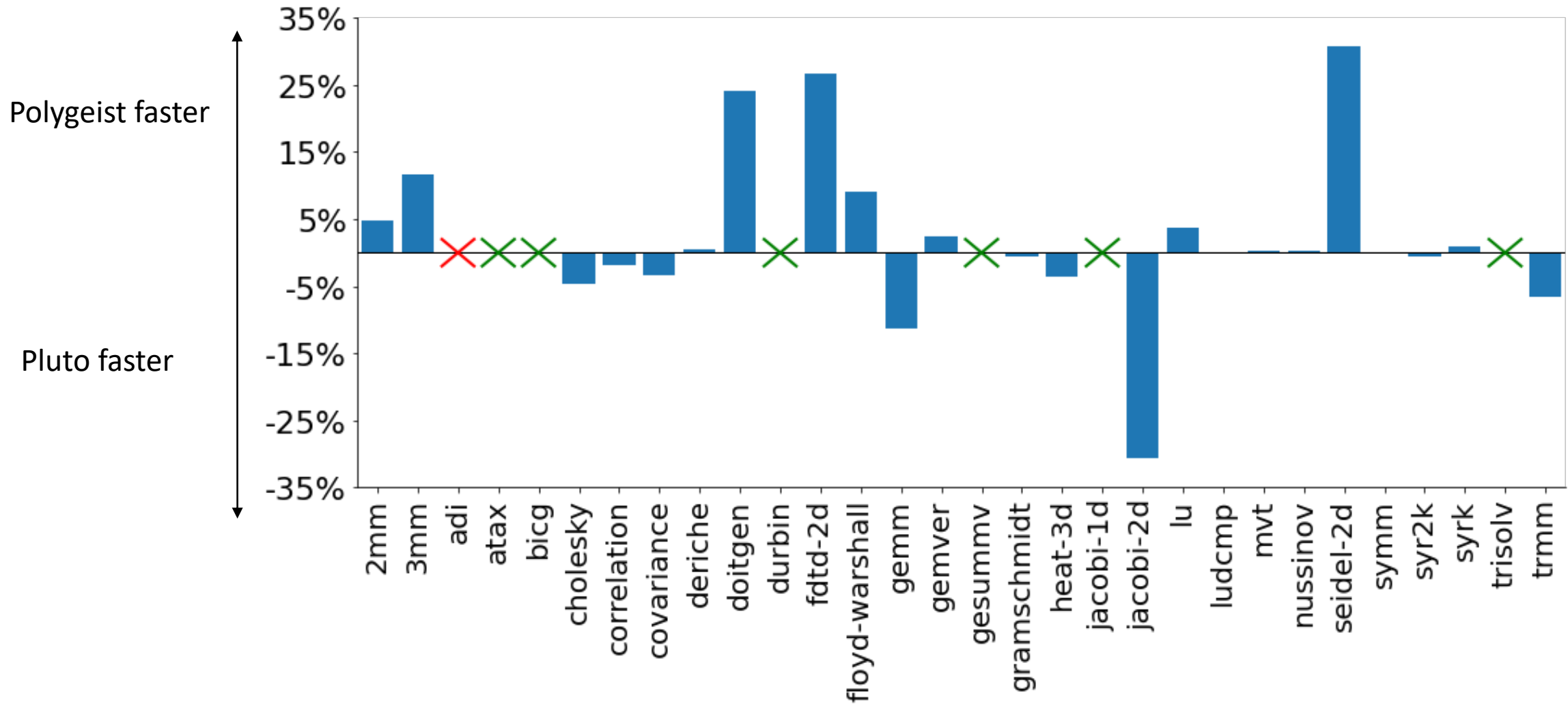
Frontend Performance Differences

- Remaining gaps:
 - Different memory allocation function
 - ~48% of gap in adi benchmark
 - LLVM strength-reduction is fragile and sometimes misses reversed loop induction variable (remaining gap in adi)
 - Type of induction variables (MLIR index vs C int32) make it easier for LLVM loop analyses to analyze code generated from MLIR.

Polygeist vs Pluto

Red X denotes test incompatible with Pluto

Green X denotes tests with runtime < 0.05s



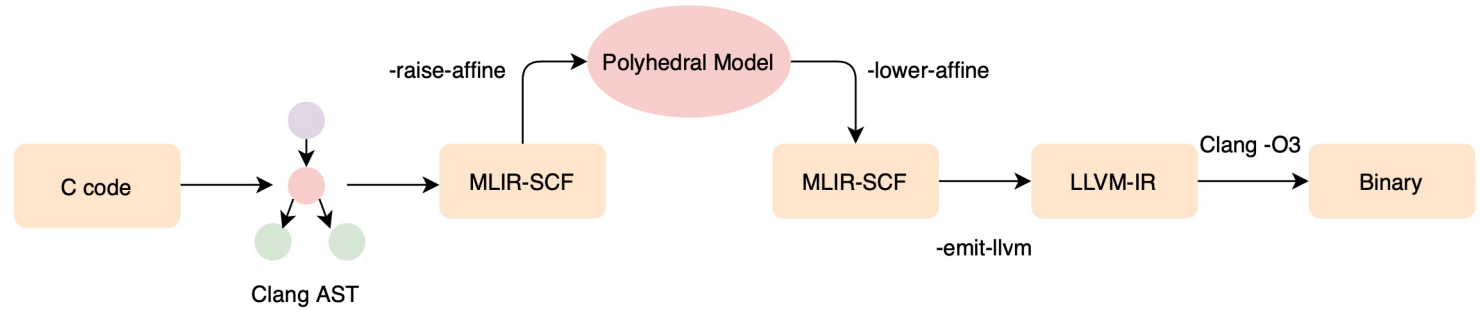
Polyhedral Performance Differences

Besides previously mentioned issues:

- CLooG AST generation
 - We test Pluto by its CLI tool (polycc)
 - Polygeist uses libpluto's `pluto_schedule_prog` API together with CLooG
 - Pluto configure options & optimized schedules are identical between them
 - Different CLooG AST, e.g., 579 (Pluto) vs 78 (Polygeist) lines for jacobi-2d
 - Pluto CLI has finer-grained control over CLooG AST generation
- Induction variable types (Pluto int vs MLIR i64)
- Auto-vectorization triggered differently

More details in the paper

Conclusion

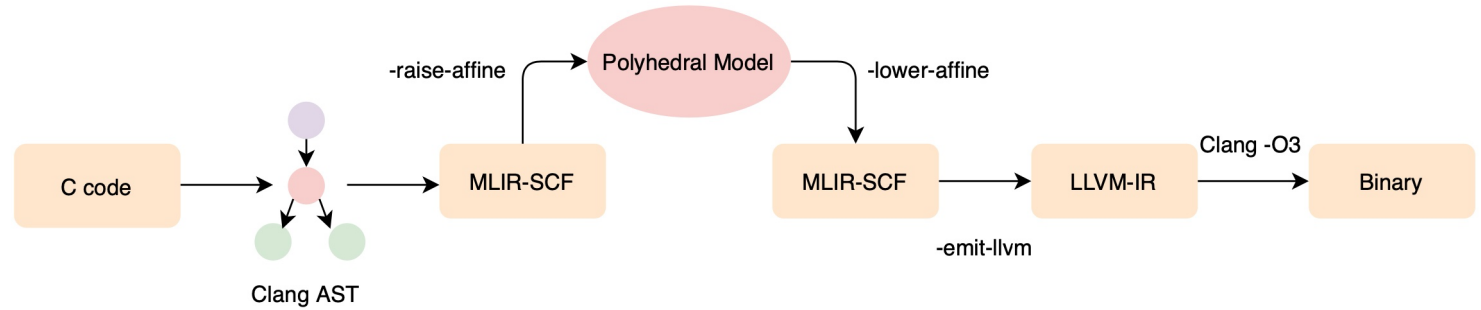


- Polygeist provides tools to fairly compare MLIR-based polyhedral flows with prior Polyhedral tools
 - C/C++ frontend for (Affine) MLIR
 - Integration of existing polyhedral tools for transforming MLIR (via OpenScop)
 - End-to-end comparison using existing Polyhedral benchmarks (Polybench)
- Polygeist enables future research on polyhedral MLIR transformations
- MLIR-based frontend differs from Clang by 1.25%
- Polygeist's polyhedral optimized code differs from Pluto by 7.76%

Acknowledgements

- Thanks to Valentin Churavy, Albert Cohen, Henk Corporaal, Tobias Grosser, and Charles Leiserson for thoughtful discussions on this work.
- William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship, in part by Los Alamos National Laboratories, and in part by the United States Air Force Research Laboratory.
- Lorenzo Chelini is partially supported by the European Commission Horizon 2020
- Ruizhe Zhao is sponsored by UKRI and Corerain Technologies Ltd. The support of the UK EPSRC is also gratefully acknowledged.

Conclusion

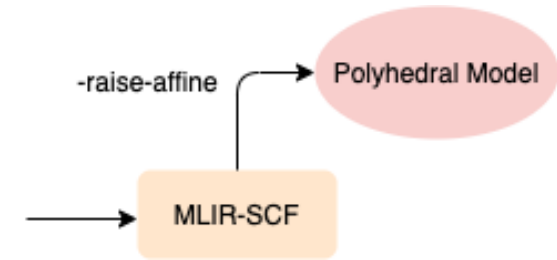


- Polygeist provides tools to fairly compare MLIR-based polyhedral flows with prior Polyhedral tools
 - C/C++ frontend for (Affine) MLIR
 - Integration of existing polyhedral tools for transforming MLIR (via OpenScop)
 - End-to-end comparison using existing Polyhedral benchmarks (Polybench)
- Polygeist enables future research on polyhedral MLIR transformations
- MLIR-based frontend differs from Clang by 1.25%
- Polygeist's polyhedral optimized code differs from Pluto by 7.76%

Backup Slides

```
func @set(%arg0: memref<?xi32>, %arg1: i32) {  
    affine.for %arg2 = 0 to 10 {  
        affine.store %arg1, %arg0[%arg2 * 2] : memref<?xi32>  
    }  
    return  
}
```

Polygeist Raising



- Select statements must be represented by a C ternary operator
 - C ternaries have lazy-evaluation semantics which are replicated in the generated MLIR
 - Mem2Reg and code motion attempt to remove unnecessary loads within if's to generate a valid select.