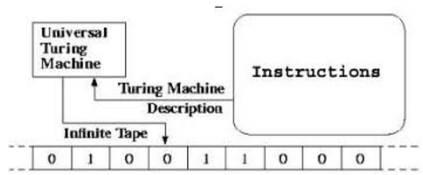
Rethinking how we build compilers: synthesis and neural machine translation

Michael O’Boyle
Senior EPSRC Research Fellow

Hardware/software contract breaking down

Technology trends means
- Hardware specialised or heterogenous

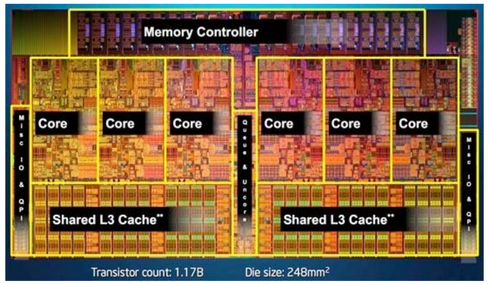
Software cannot fit on new hardware

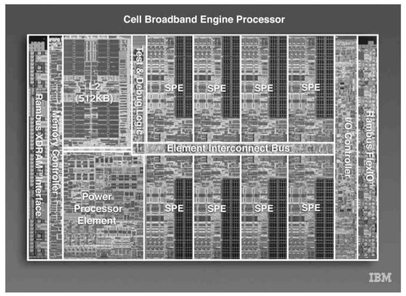


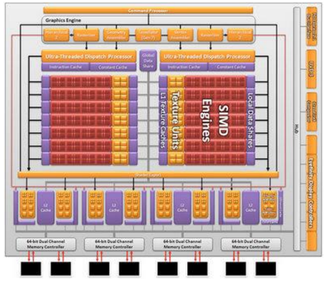
Heterogeneous crisis
- hardware stalls as software cannot fit

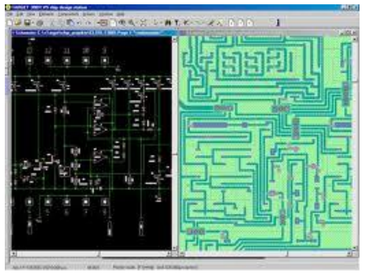
How to bridge the heterogeneity gap?

New Application/Legacy Code









DSL/API approach

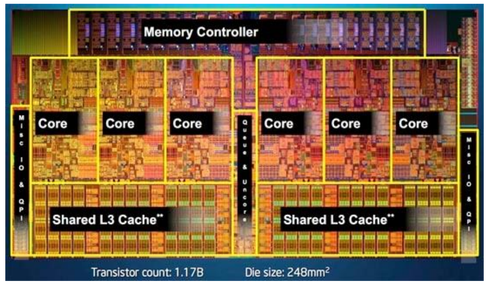
New Application/Legacy Code

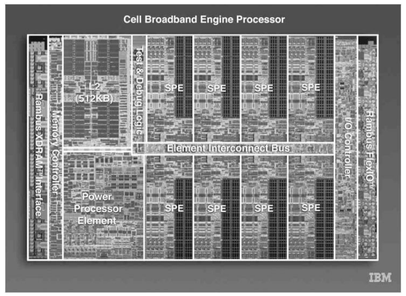
DSL/API

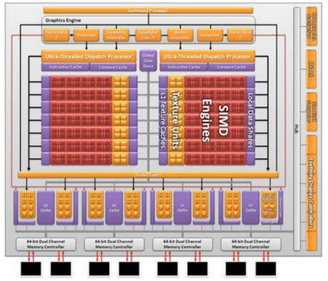
DSL/API

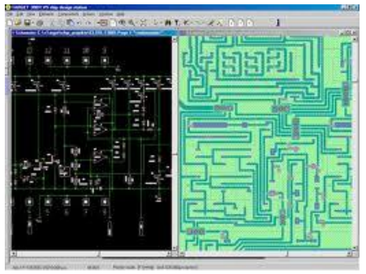
DSL/API

DSL/API







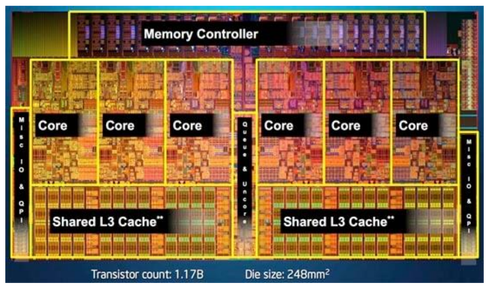


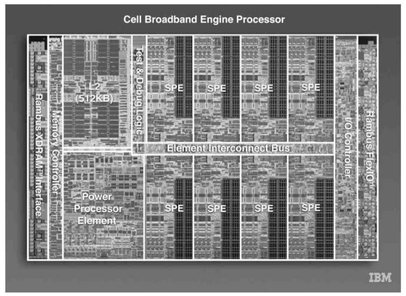
DSL/API approach

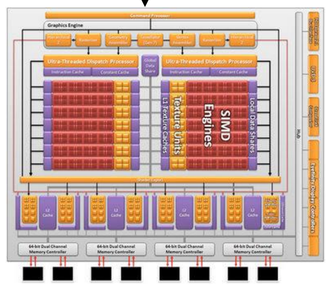
New Application/Legacy Code

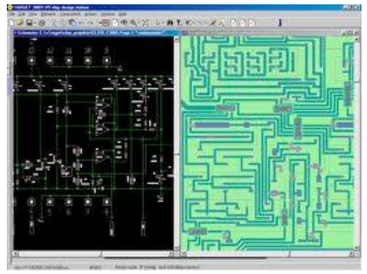
DSL/API DSL/API DSL/API DSL/API

Vendor Responsibility
Motivated to succeed!

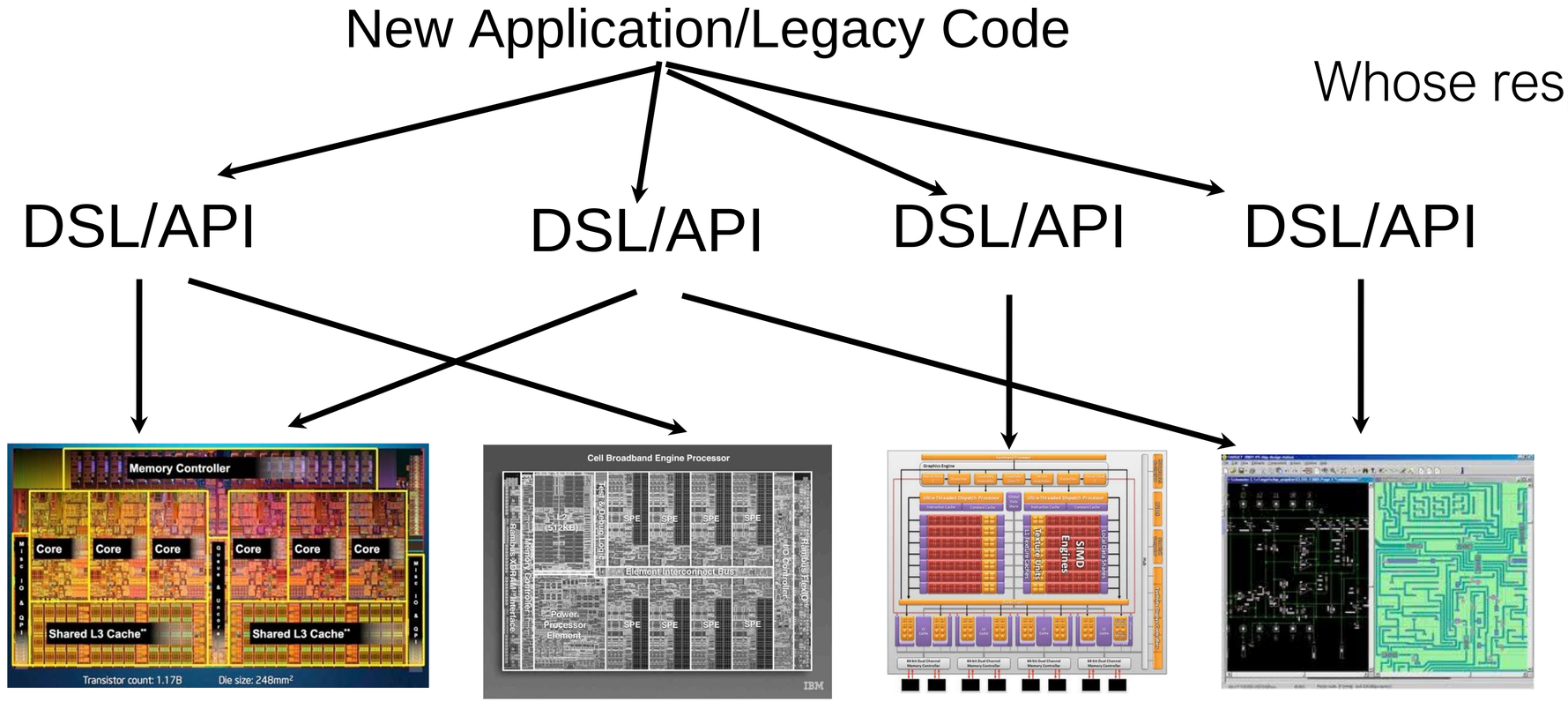






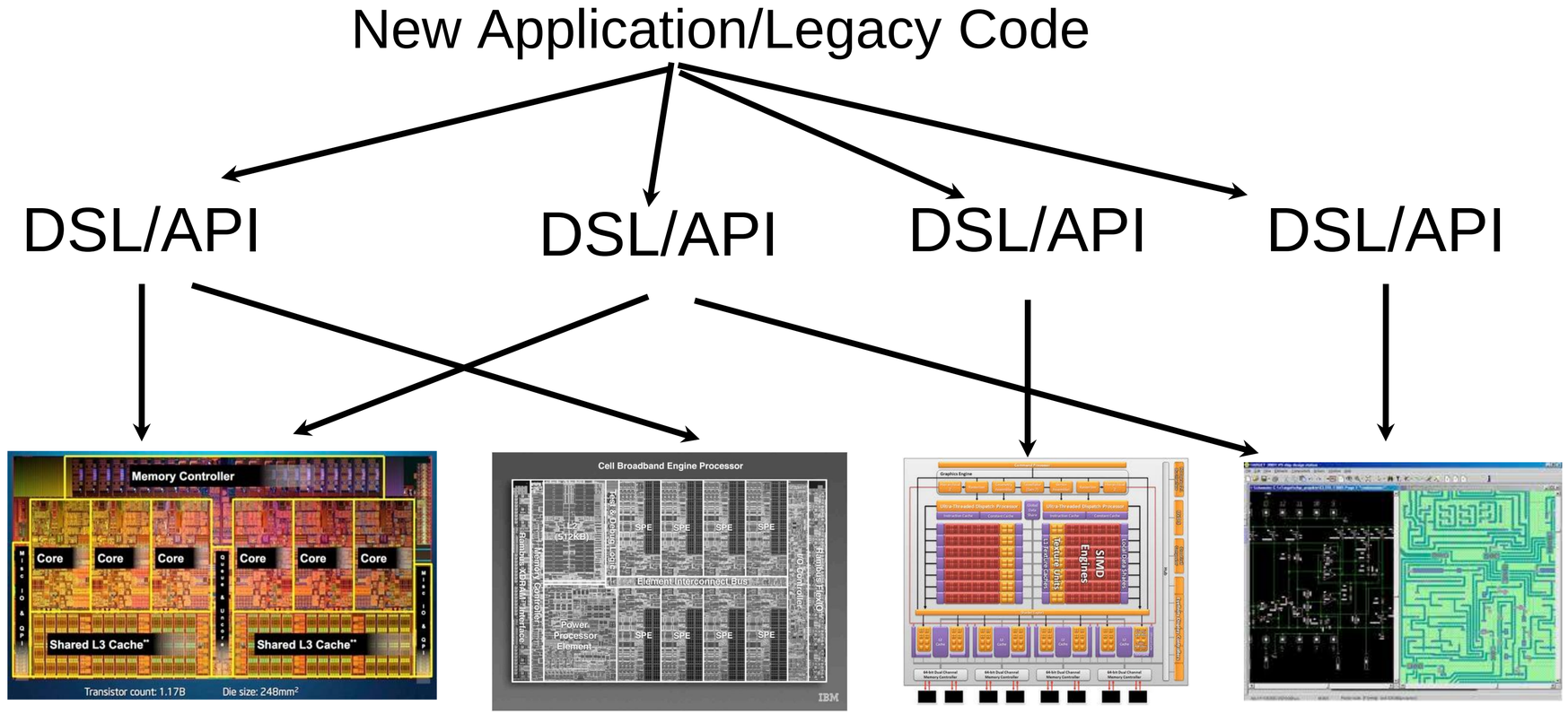


DSL/API approach



Whose responsibility?

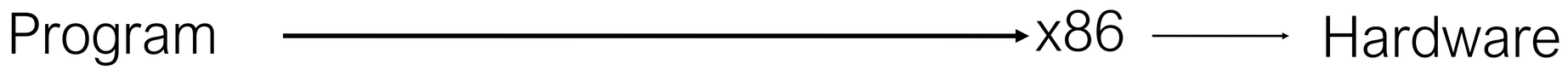
DSL/API approach

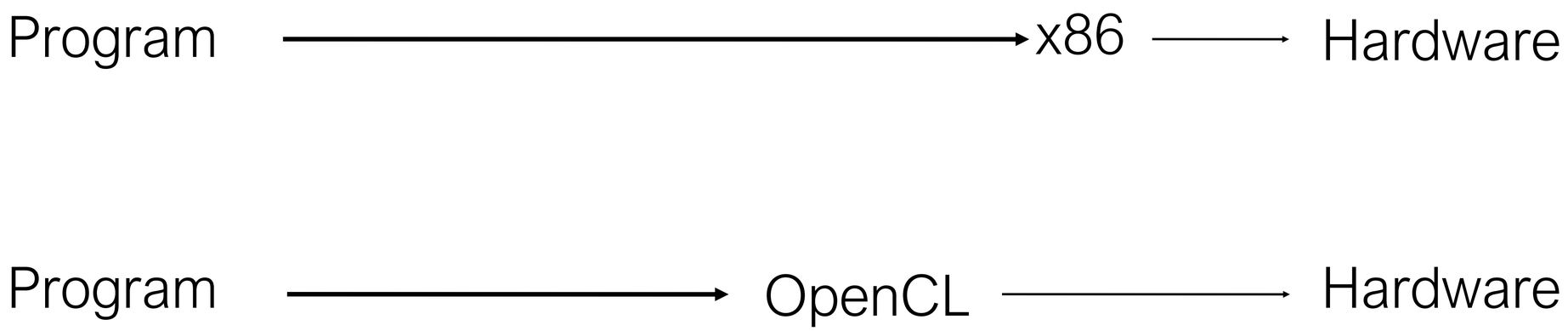


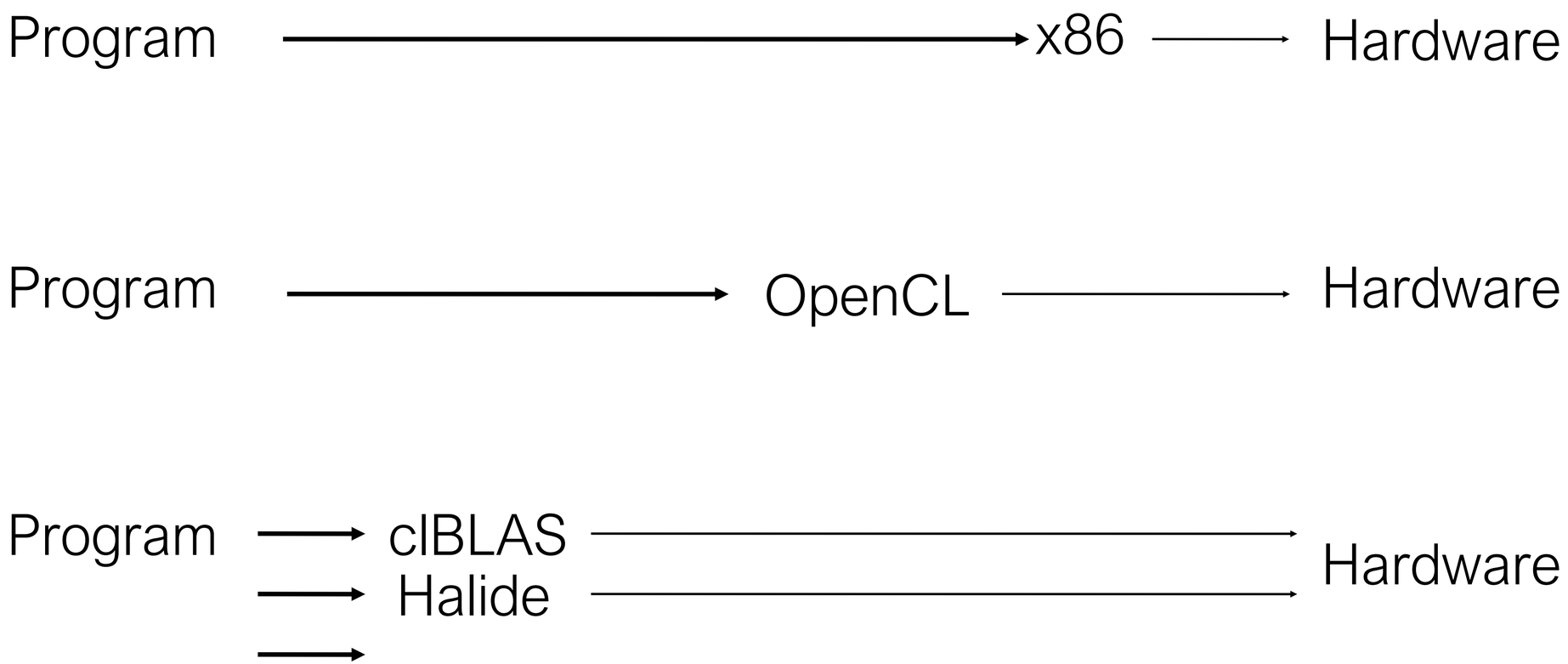
Many specialised languages

- programmers rewrite and hope it works on the next machine

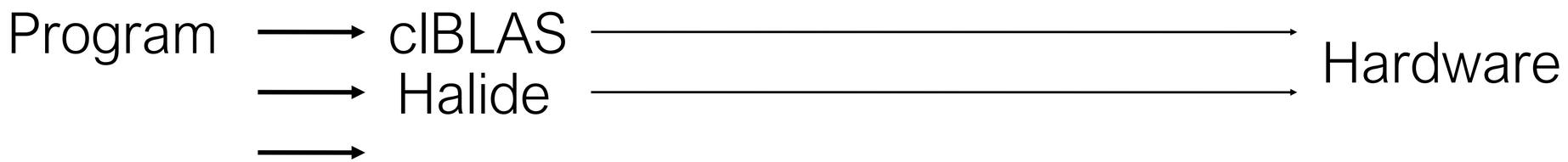
Rewriting programs for ever changing DSLs is not sustainable
Can compilers help?


Program
x86
Hardware

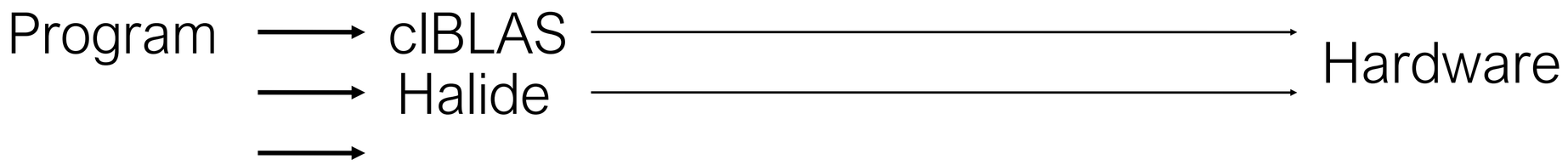

Program
x86
Hardware
Program
OpenCL
Hardware


Program
x86
Hardware
Program
OpenCL
Hardware
Program
clBLAS
Halide
Hardware

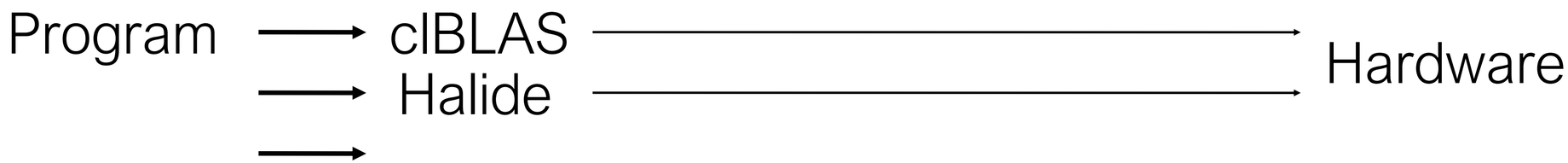
+ Target nearer to algorithm
+ Target willl always perform well



+ Target nearer to algorithm
+ Target willl always perform well
- Target complex and changeable



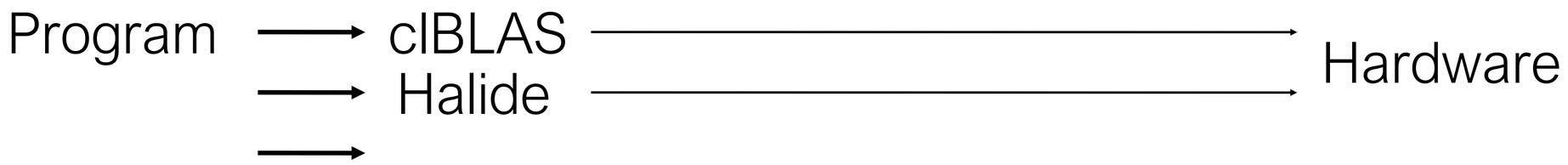
+ Target nearer to algorithm
+ Target willl always perform well
- Target complex and changeable



Constant change means any solution must work for any API, any DSL

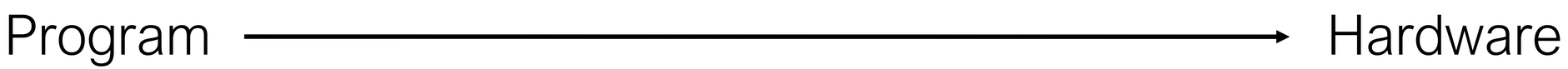
Need to automate

+ Target nearer to algorithm
+ Target willl always perform well
- Target complex and changeable
- Target may be at higher level

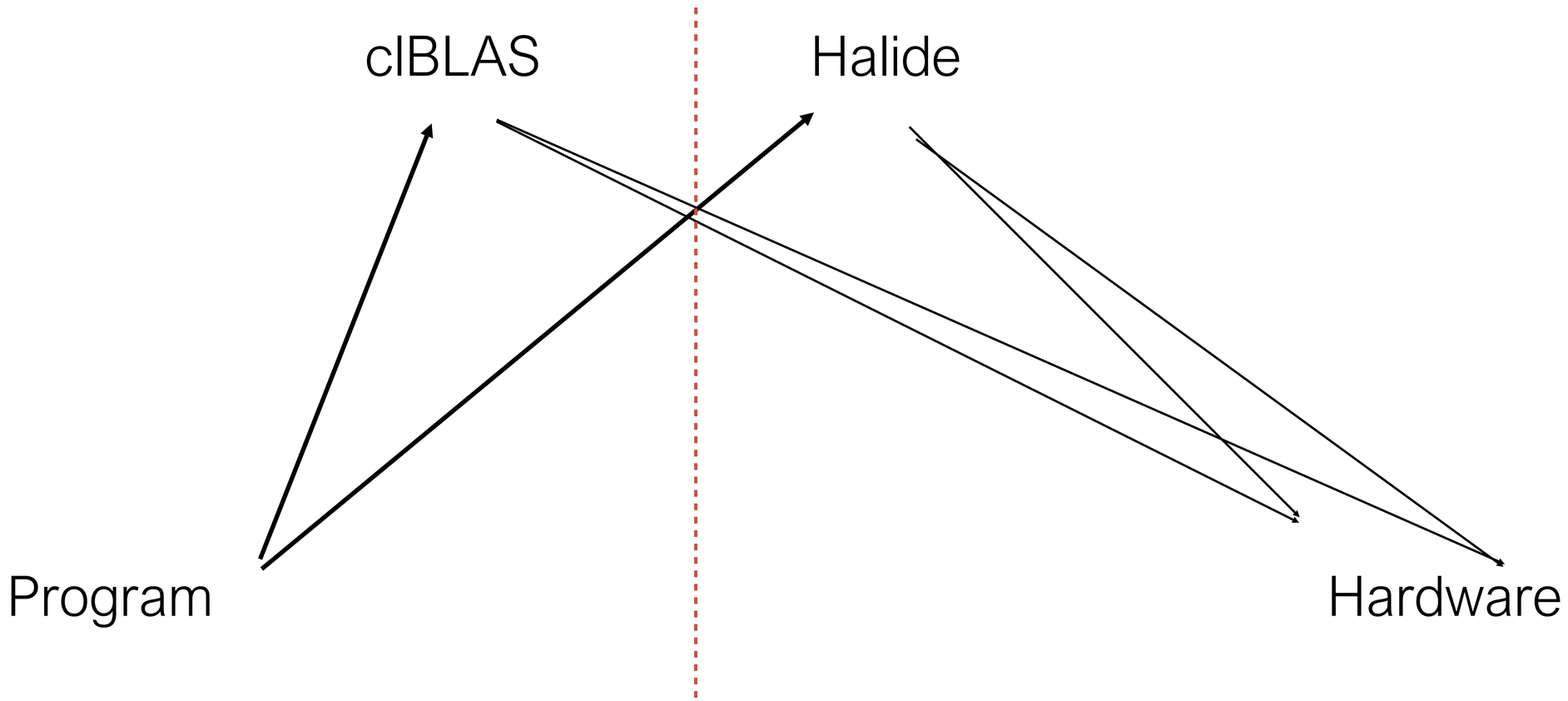


Rather than compile code to hardware

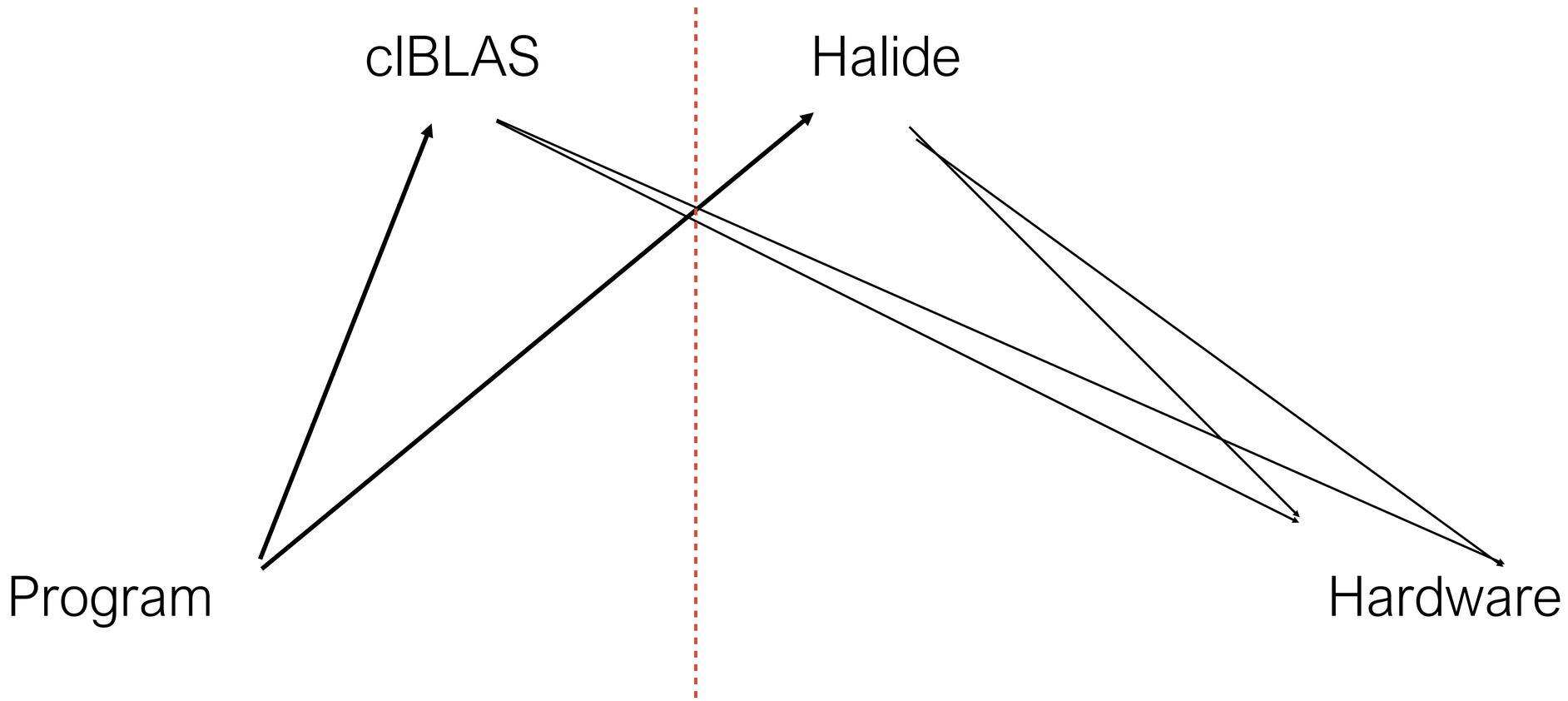
By lowering code for each language and each ISA



Instead LIFT code to API or DSL



LIFT code to API or DSL



Vendor responsibility to map API/DSL to hardware - already the case

Our job - automatically lift it to API/DSL enabling hardware utilisation

Two technologies for lifting

- Program synthesis: raising MLIR

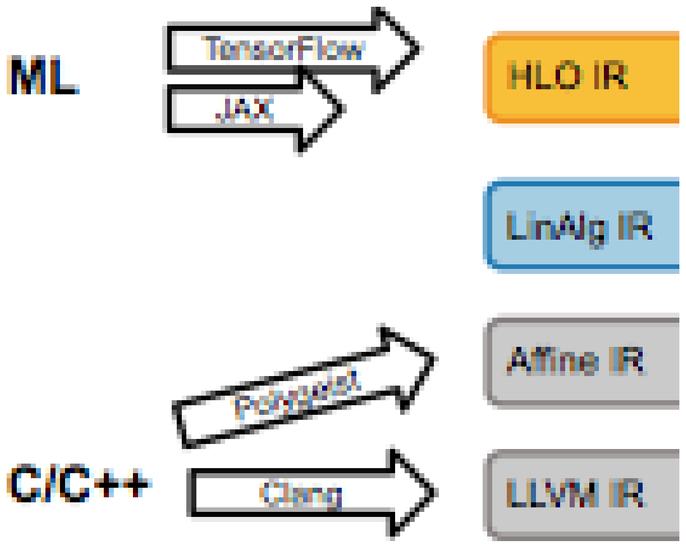
- Neural machine translation: decompilation

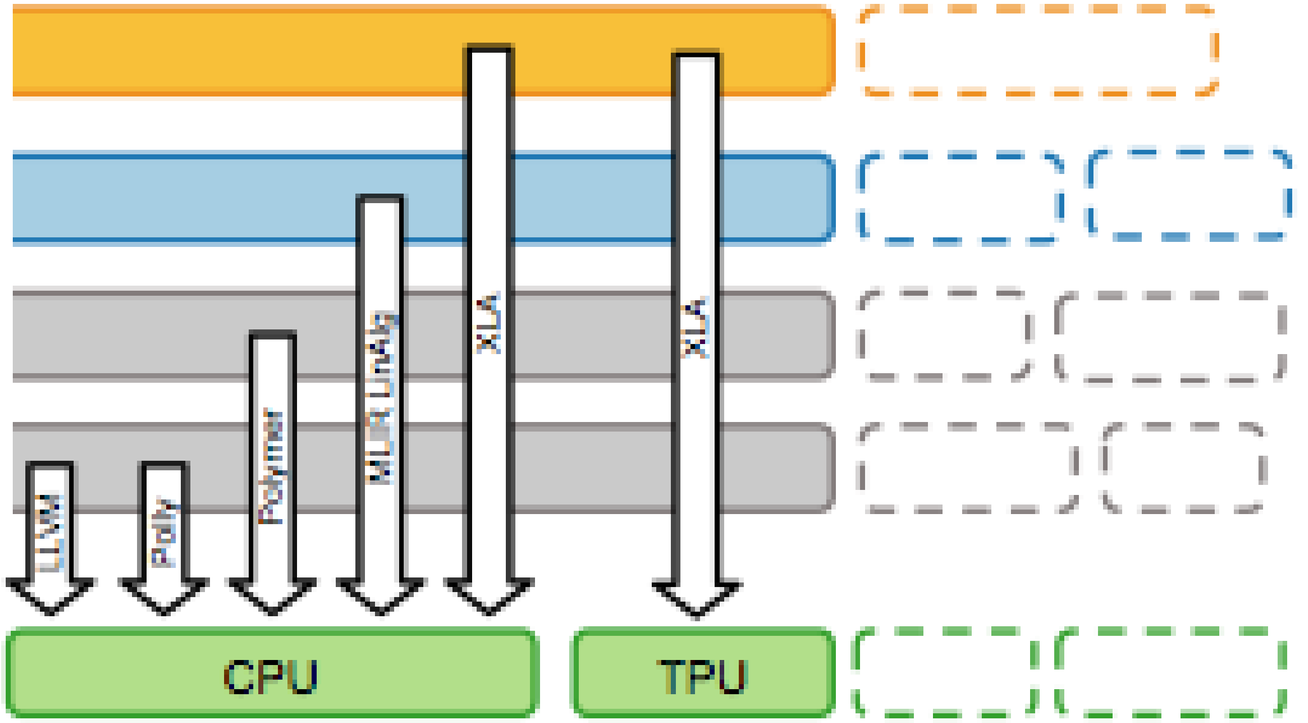
mlirSynth

Synthesizing Domain-Specific Programs in MLIR

Alexander Brauckmann

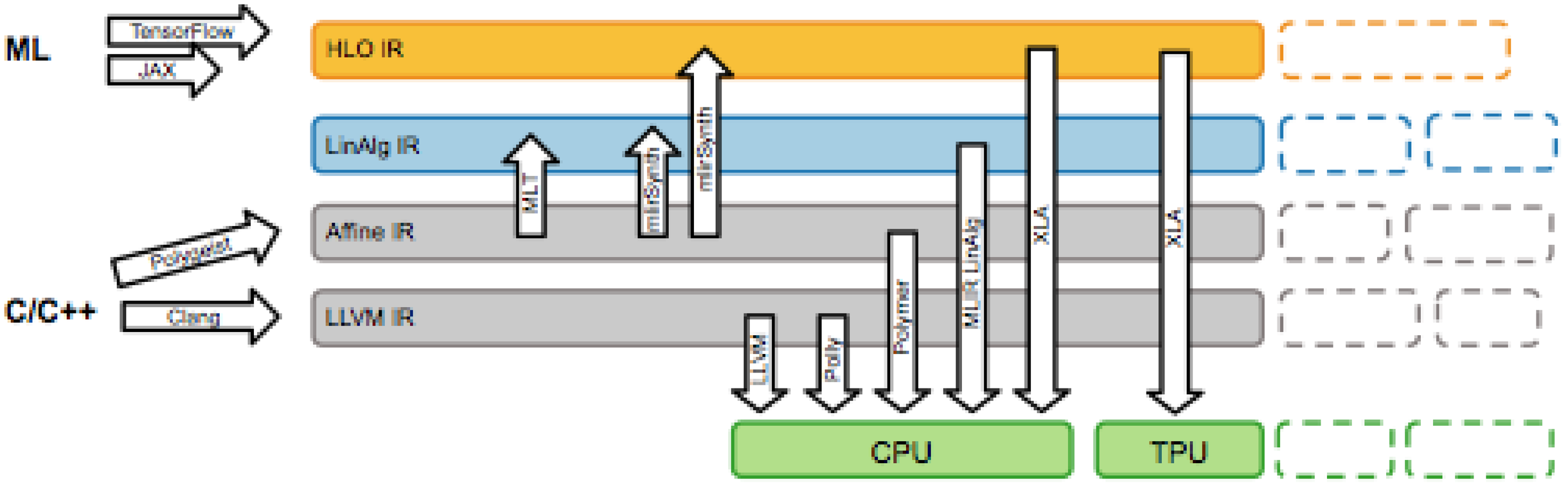
Elizabeth Polgreen
Tobias Grosser





High level dialects in MLIR allow compilers to exploit domain knowledge
- Benefits TensorFlow

Legacy code unable to exploit this
- At too low a level. Need to lift

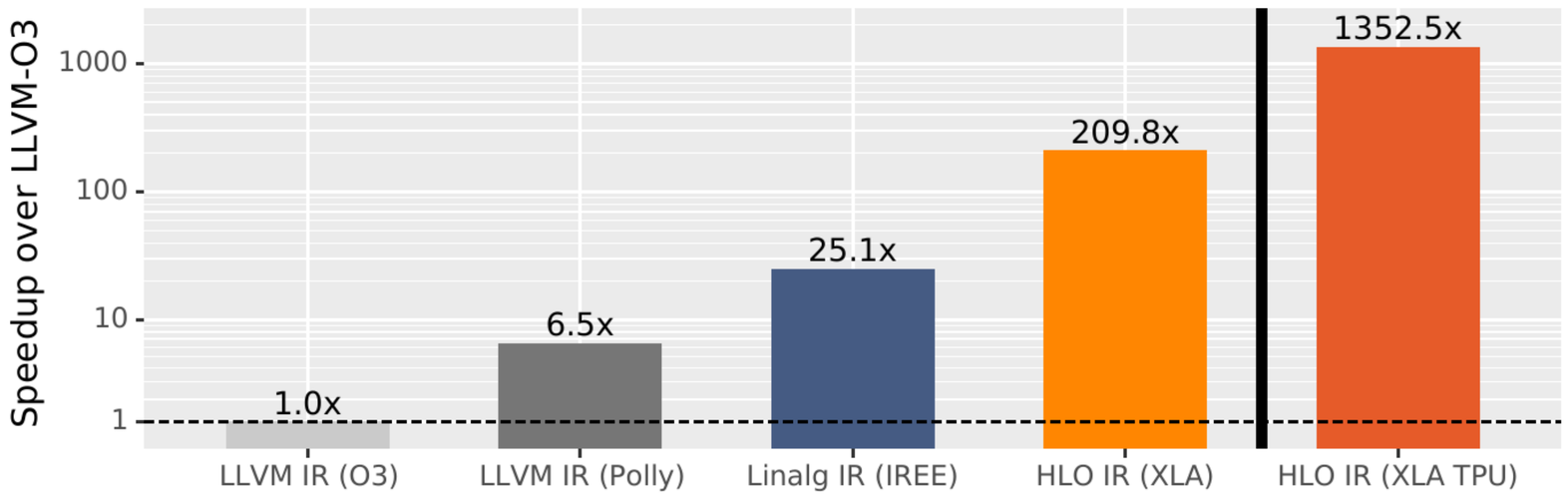


MLT: multi level tactics

- Pattern based lifter.
- Needs new rules for each pattern and dialect

MLIR synth - our automatic approach

mlirSynth delivers 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];
  }
}
```

Raising

Raising

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>
```

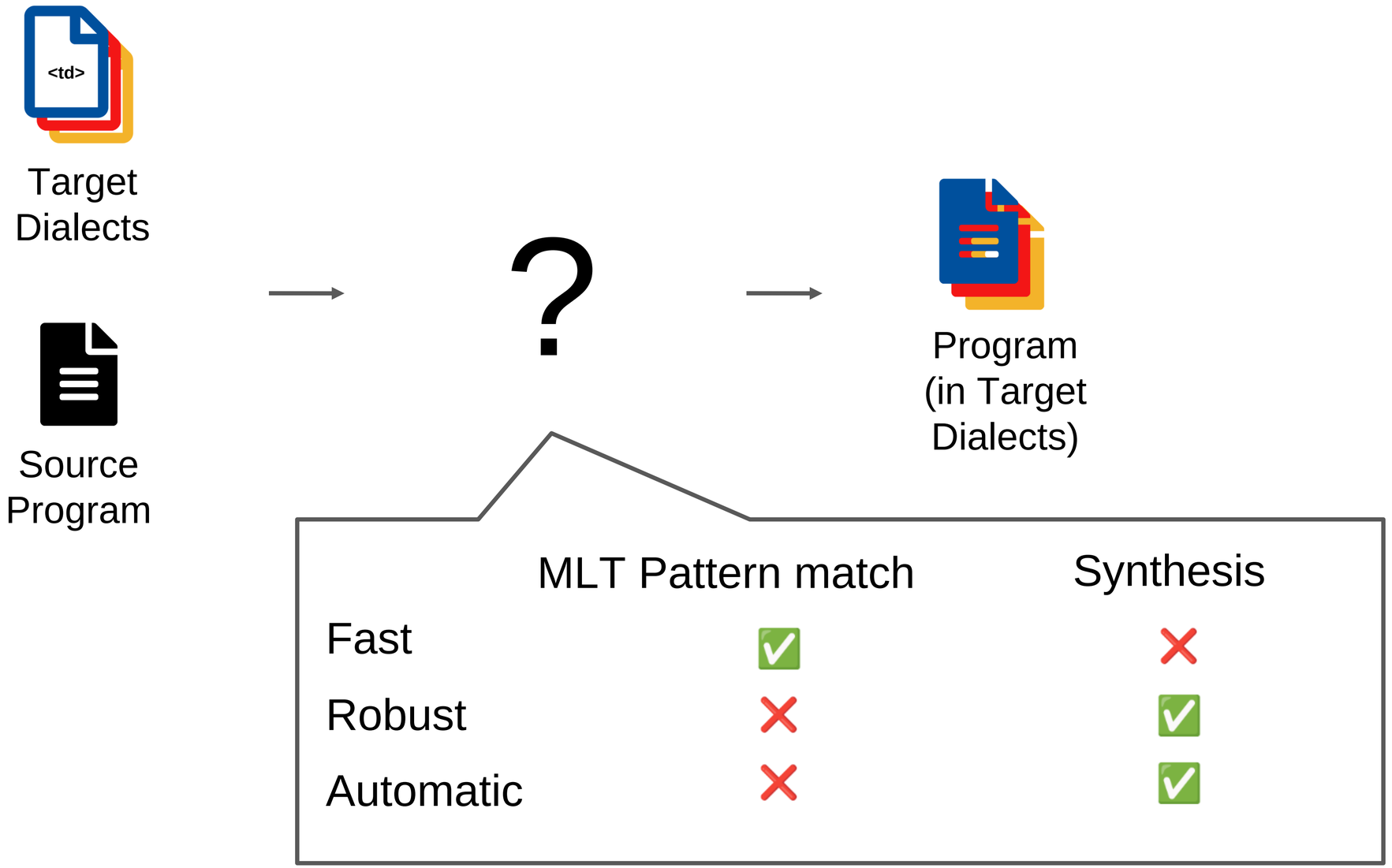
HLO IR

```
%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>
```

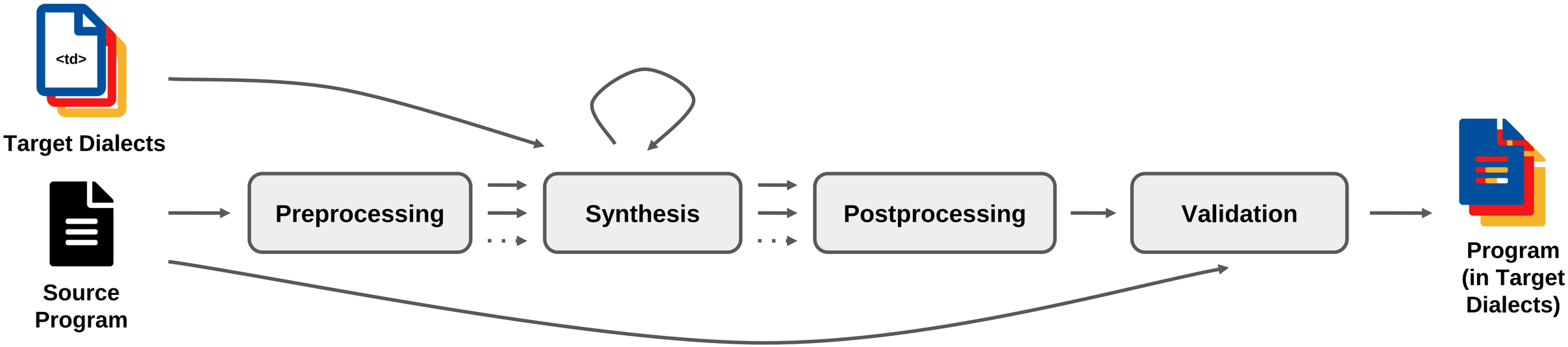
CPU: AMD Ryzen 9 3900X

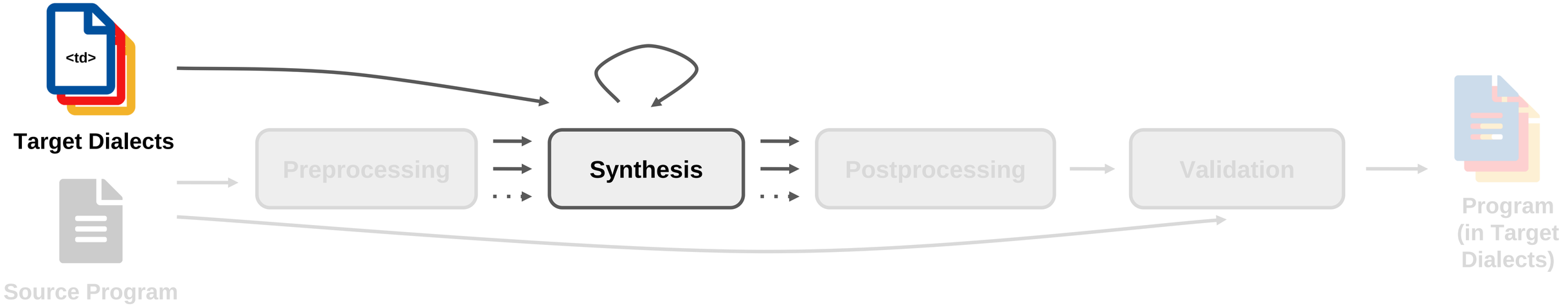
TPU: TPUv2

Raising with Synthesis



mlirSynth


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



Specification

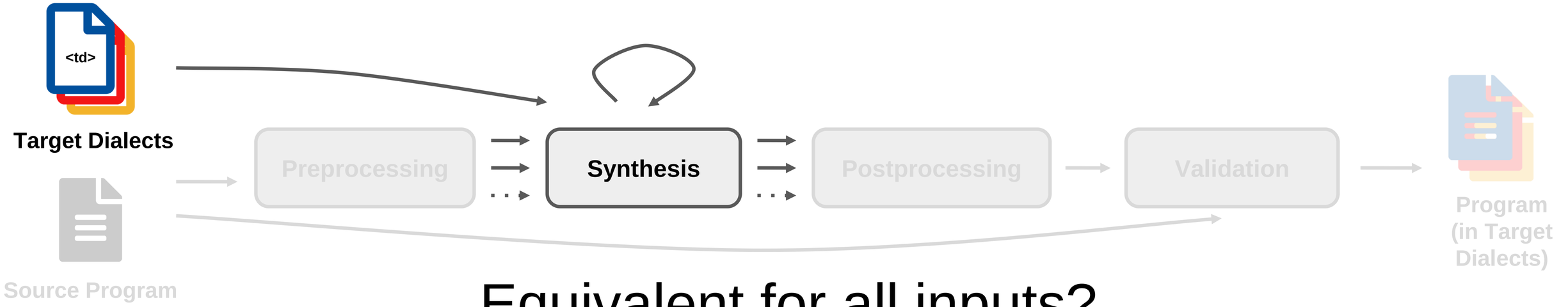
- Generate Input/Output example

Bottom-up enumerative search

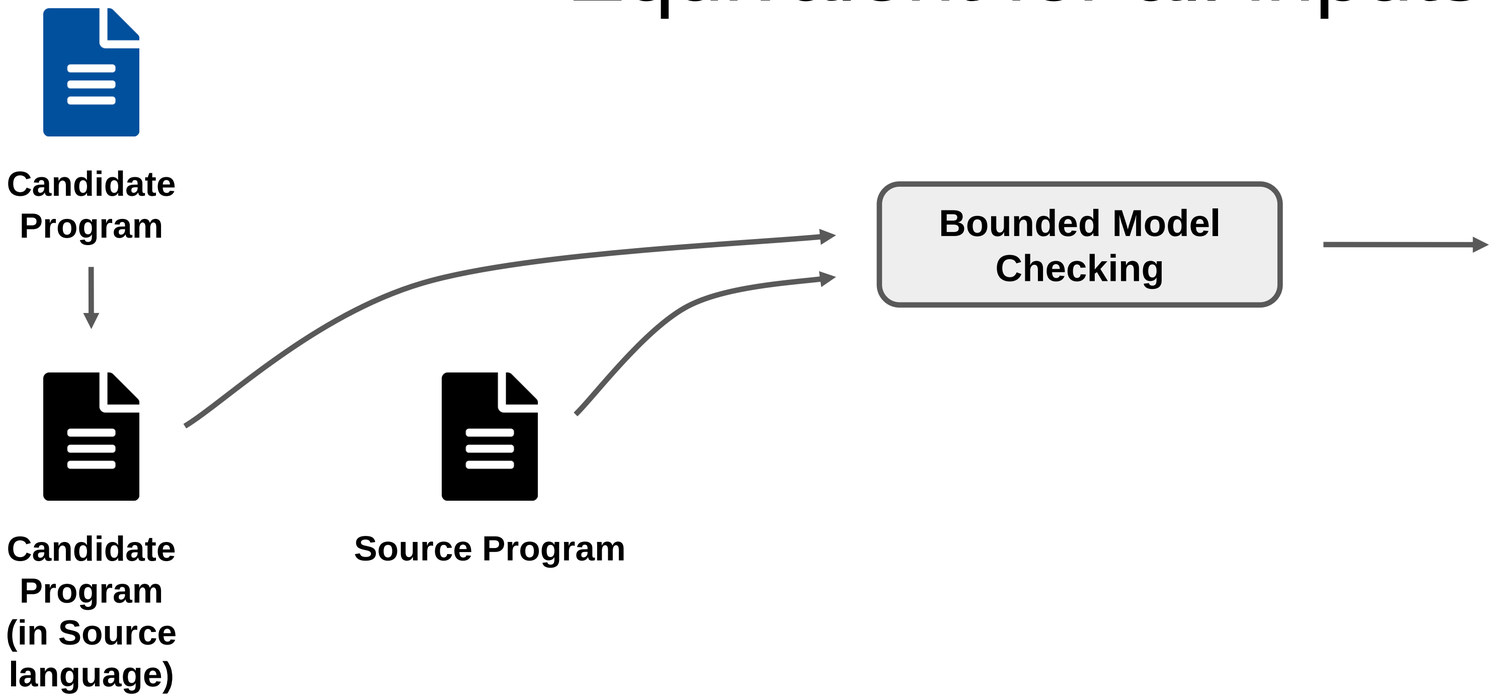
- Progressively grow a candidate set by combining simpler to more complex ones
- Initialization: Basic programs (returning arguments, constants)
- Terminate when specification matched

Optimization techniques

- Type correct by construction
- Identify classes of observationally equivalent candidates
- Polyhedral-based heuristics for guiding synthesis



Equivalent for all inputs?



Equivalence Guarantees

a) Float arithmetic
b) Float arithmetic, permitting small δ
c) Integer arithmetic

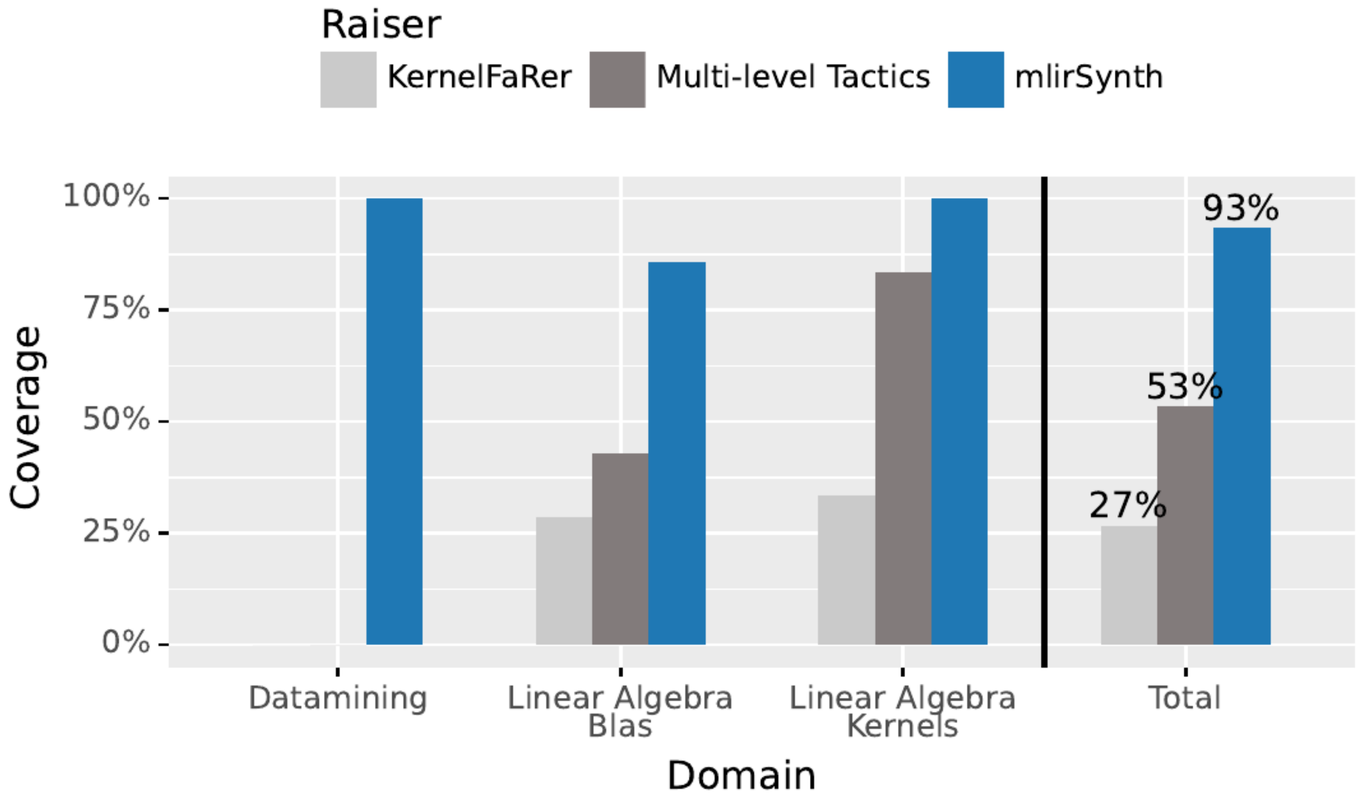
Testing I/O equivalence

Coverage

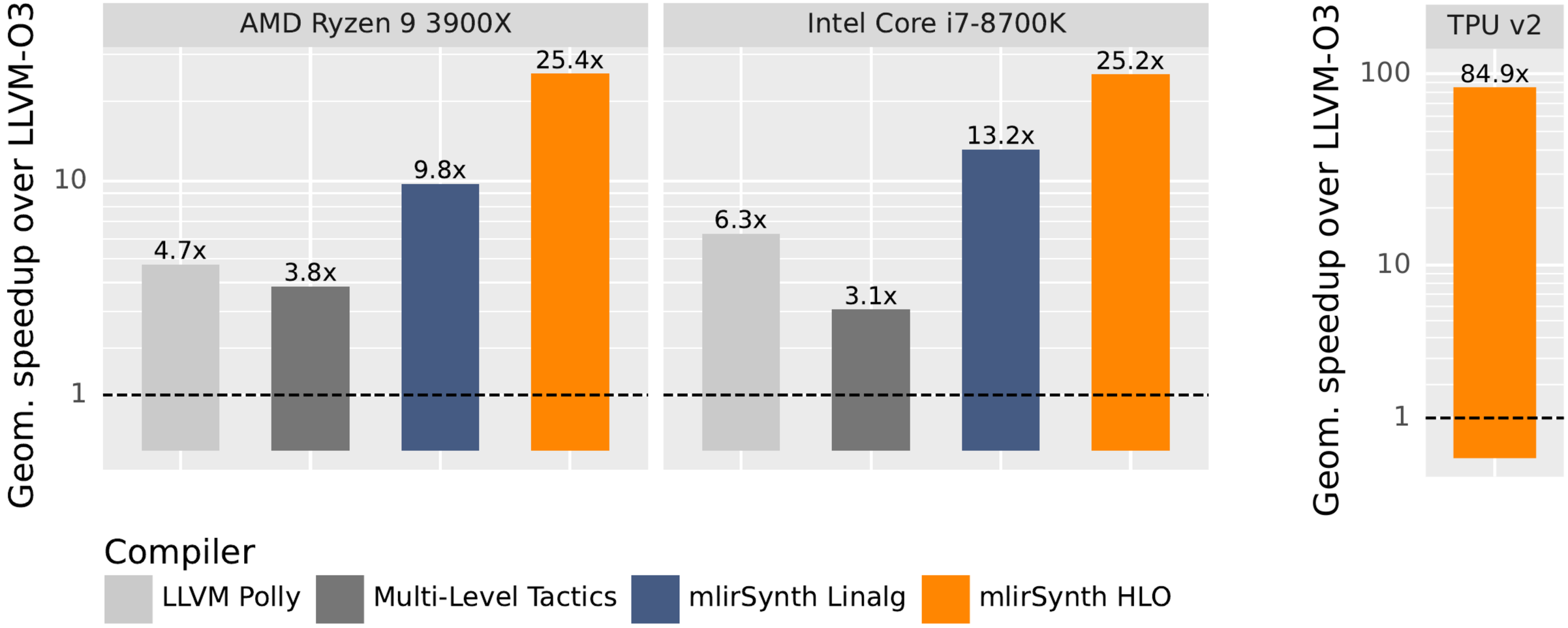
Benchmark: PolyBench

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

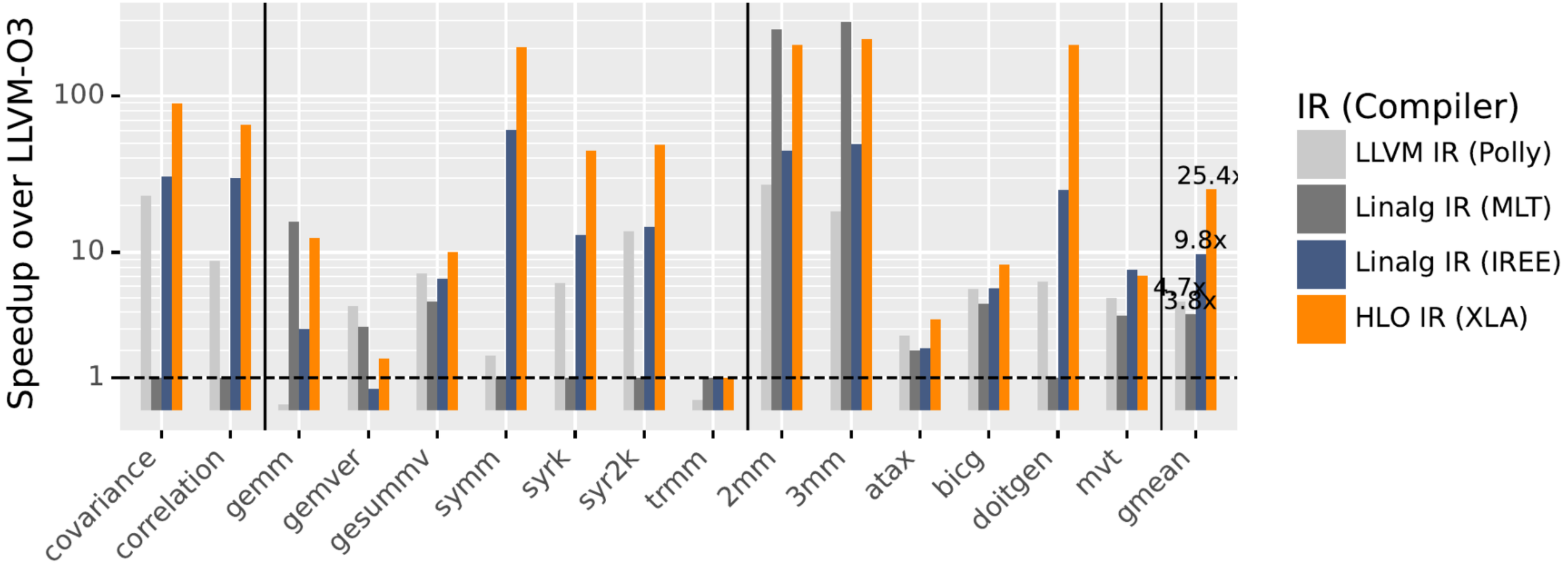
→Total: 15 Programs



Overall Performance

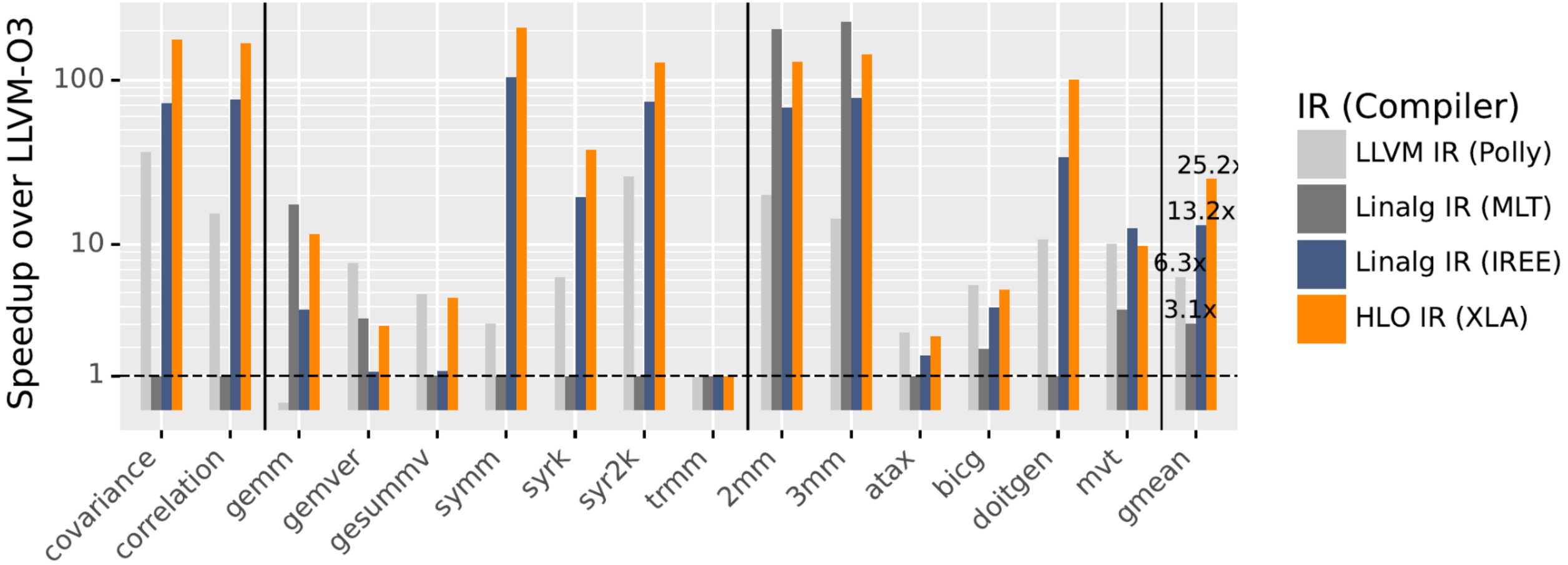


AMD: Per benchmark



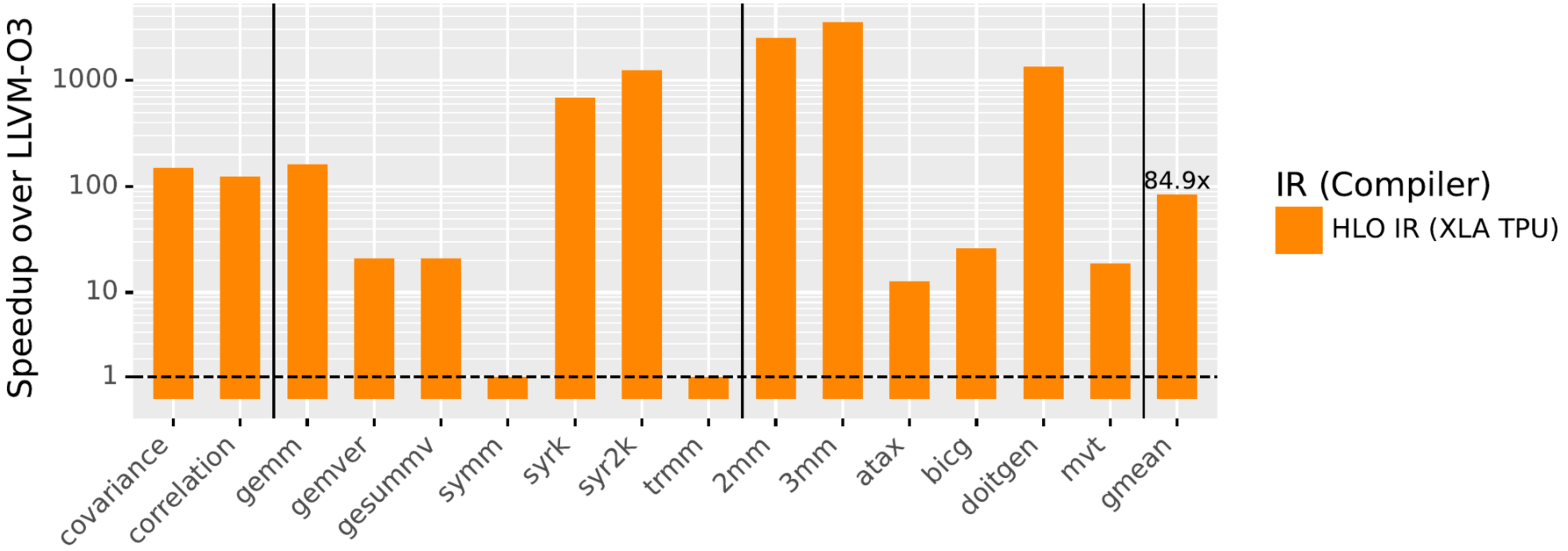
CPU: AMD Ryzen 9 3900X

Intel: Per benchmark



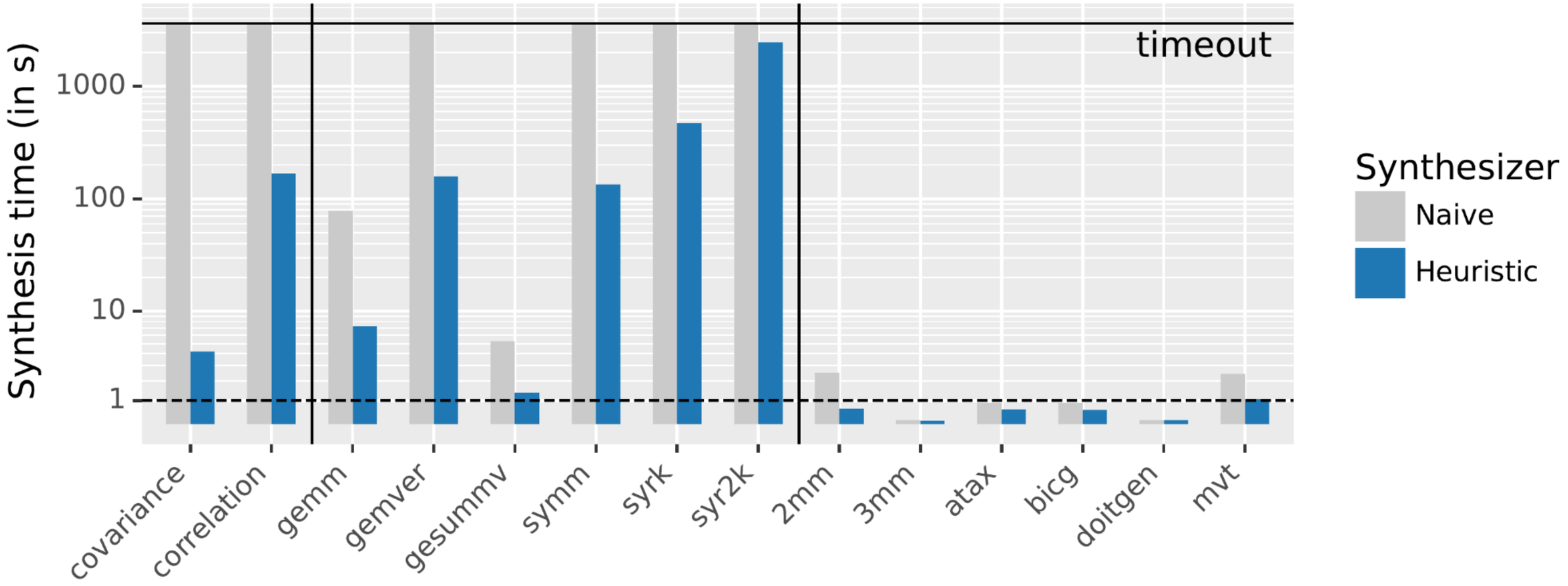
CPU: Intel Core i7-8700K

TPU: Per benchmark

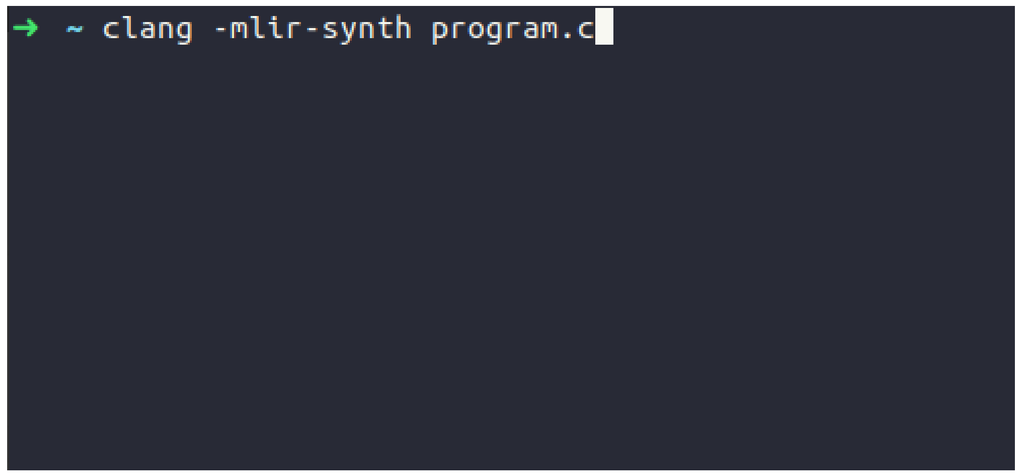


CPU: AMD Ryzen 9 3900X
TPU: TPU v2.8

Synthesis Time



Future Work



Method

- Detection of raisable code regions + their dialect
- Speed up search

Applicability

- More source dialects
- More target dialects

Two technologies for lifting

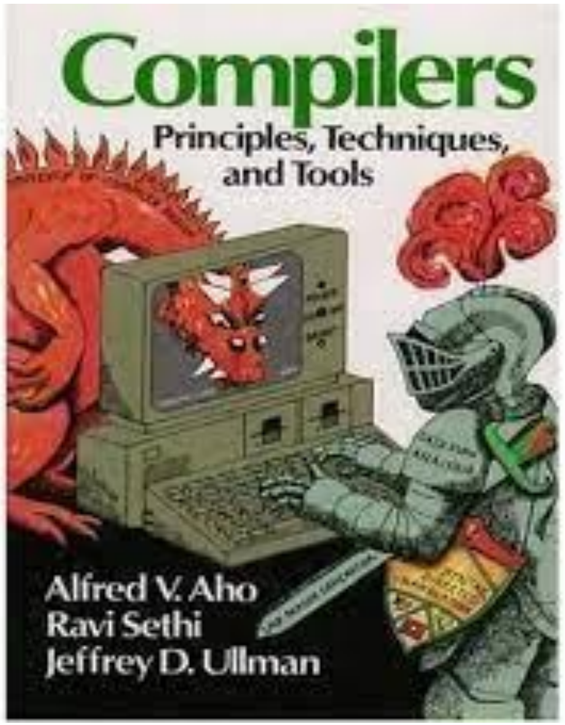
- Program synthesis: raising MLIR

- Neural machine translation: decompilation

SLaDe: A Portable Small Language Model Decompiler for Optimized Assembler

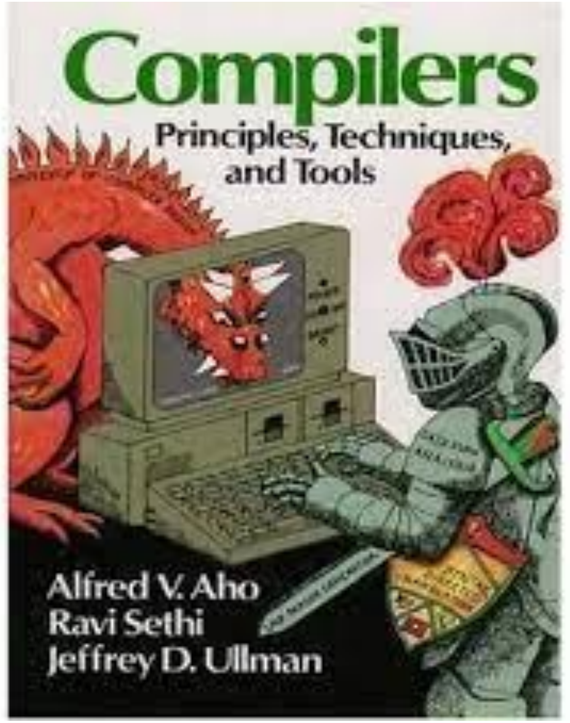
Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins

Compilation: C->x86

C → 

→ **x86**

Decompilation: x86->C

C ← 

← **x86**

Difficult to run compilers backwards

State-of-the-art GHIDRA

C ← 

← **x86**

Decompiler from NSA
Many person-years effort

Useful for:

- Security purposes!
- Porting legacy code.
- Lifting?

Excellent rule-based decompilers available…

… but they produce hard-to-read code!

What does this code do?

GCC O3:

```
 .globl add                                                    4
 .type add, @function
add:
.LFB0:                          .L3:
 .cfi_startproc                  movslq %edi, %rax
 endbr64                         leal 1(%rdi), %r8d
 movq %rdi, %rcx                 salq $2, %rax
 testl %edx, %edx                addl %esi, (%rcx,%rax)
 jle .L1                         cmpl %r8d, %edx
 leal -1(%rdx), %eax             jle .L1
 cmpl $2, %eax                   addl $2, %edi
 jbe .L6                         addl %esi, 4(%rcx,%rax)
 movq %rdi, %rax                 cmpl %edi, %edx
 movl %edx, %edi                 jle .L1
 movd %esi, %xmm2                addl %esi, 8(%rcx,%rax)
 shrl $2, %edi                  .L1:
 pshufd $0, %xmm2,               ret
       %xmm1                    .p2align 4,,10
 subl $1, %edi                   .p2align 3
 salq $4, %rdi                  .L9:
 leaq 16(%rcx,%rdi),%rdi         ret
 .p2align 4,,10                 .L6:
 .p2align 3                      xorl %edi, %edi
.L4:                             jmp .L3
 movdqu (%rax), %xmm0            .cfi_endproc
 addq $16, %rax
 paddd %xmm1, %xmm0
 movups %xmm0, -16(%rax)
 cmpq %rdi, %rax
 jne .L4
 movl %edx, %edi
 andl $-4, %edi
 testb $3, %dl
 je .L9
```

Generates structured code
- but difficult to read

Non-intuitive variable names

Use of shifts and masks

Follows O3 control-flow mangling

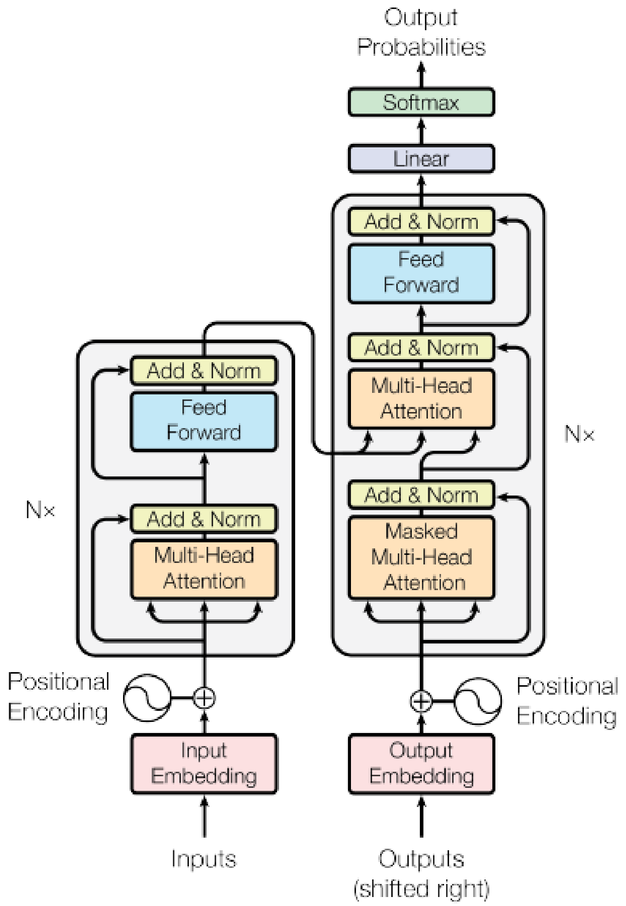
```
void add(int *param_1, int param_2,          1
         uint param_3) {
  uint uVar1;
  int *piVar2;
  int *piVar3;

  if (0 < (int)param_3) {
    if (param_3 - 1 < 3) {
      uVar1 = 0;
      do {
        param_1[(int)uVar1] =
          param_1[(int)uVar1] + param_2;
        uVar1 = uVar1 + 1;
      } while ((int)uVar1 < (int)param_3);
      return;
    } else {
      piVar2 = param_1;
      do {
        piVar3 = piVar2 + 4;
        *piVar2 = *piVar2 + param_2;
        piVar2[1] = piVar2[1] + param_2;
        piVar2[2] = piVar2[2] + param_2;
        piVar2[3] = piVar2[3] + param_2;
        piVar2 = piVar3;
      } while (piVar3 = param_1 +
        ((ulong)param_3 >> 2) * 4,
        piVar2 != piVar3);

      uVar1 = param_3 &
        0xfffffffc;
      if ((param_3 & 3) == 0) {
        return;
      }
      param_1[(int)uVar1] =
        param_1[(int)uVar1] + param_2;
      if ((int)(uVar1 + 1) <
          (int)param_3) {
        param_1[(int)(uVar1 + 1)] =
          param_1[(int)(uVar1 + 1)] +
          param_2;
        if ((int)(uVar1 + 2) <
            (int)param_3) {
          param_1[(int)(uVar1 + 2)] =
            param_1[(int)(uVar1 + 2)] +
            param_2;
          return;
        }
      }
    }
  }
}
```

The promise of neural decompilation

+ Neural networks could automate decompiling

+ Trained on human-written code, they should produce readable code

- Produce plausibly looking but semantically incorrect code



BTC

A recent neuro-decompiler
- works on code snippets
- not functions
- only O0

Generates readable code
- but does not compile
- is incorrect

```
int now = *len;                                          5
while (now > 1 && val > array[now >> 1]) {
    array[now] = array[now >> 1];
    now >>= 1;
    array[now] = val;
}
```

BTC
(Decompiled from O0)


ChatGPT

3

```
void add(int* dest, int* src, int n) {
  if (n <= 0) {
    return;
  }
  // If n is smaller than 2, use a loop
  // to add the values
  if (n < 2) {
    for (int i = 0; i < n; i++) {
      dest[i] += src[i];
    }
    return;
  }

  // Align dest pointer to
  // 16-byte boundary
  int* aligned_dest = (int*)
     ((uintptr_t) dest + 15) & -15);

  // Calculate the number of elements
  // before the aligned dest pointer
  int pre_count = (aligned_dest - dest);

  // Add the unaligned elements
  for (int i = 0; i < pre_count; i++) {
    dest[i] += src[i];
  }
```

```
  // Add the aligned elements
  int count = (n - pre_count) / 4;
  __m128i* sse_src =
        (__m128i*) (src + pre_count);
  __m128i* sse_dest =
        (__m128i*) (aligned_dest);
  __m128i xmm1 =
        _mm_loadu_si128(sse_src);
  for (int i = 0; i < count; i++) {
    __m128i xmm0 =
      _mm_load_si128(sse_dest);
    xmm0 = _mm_add_epi32(xmm0, xmm1);
      _mm_store_si128(sse_dest, xmm0);
    sse_src++;
    sse_dest++;
  }

  // Add the remaining elements
  int remaining = n -
    (pre_count + count * 4);
  for (int i = 0; i < remaining; i++) {
    dest[pre_count + count * 4 + i] +=
       src[pre_count + count * 4 + i];
  }
}
```

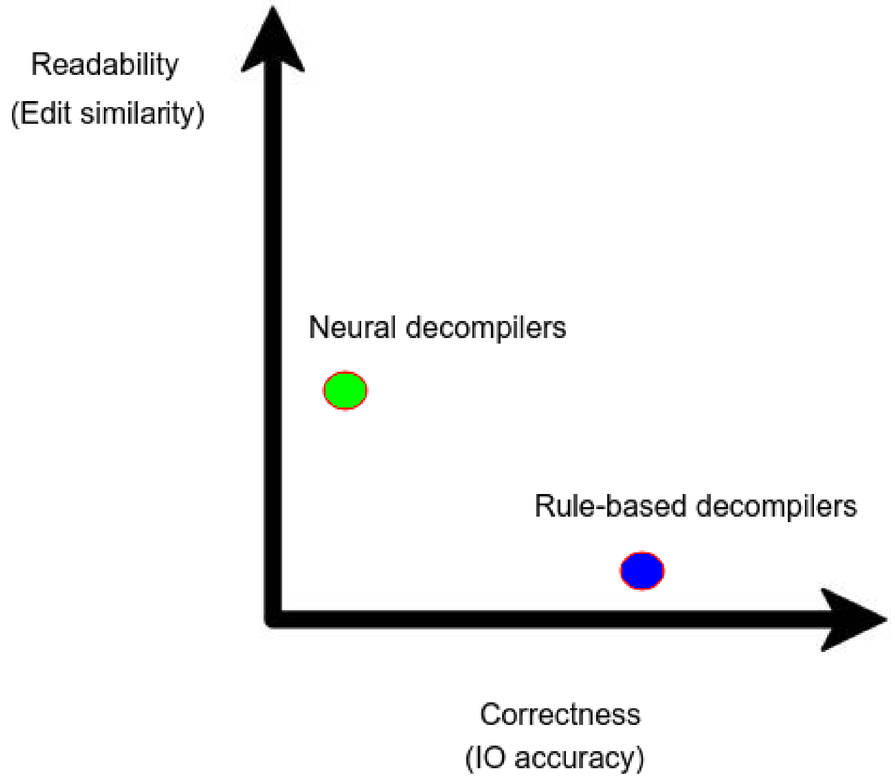
Compared to Ghidra and BTC
+ More readable code than Ghidra
+ More meaningful variable names
+ Even adds comments!
+ Compiles and executes

However
- complex code
- introduces x86 intrinsics
- gives incorrect results

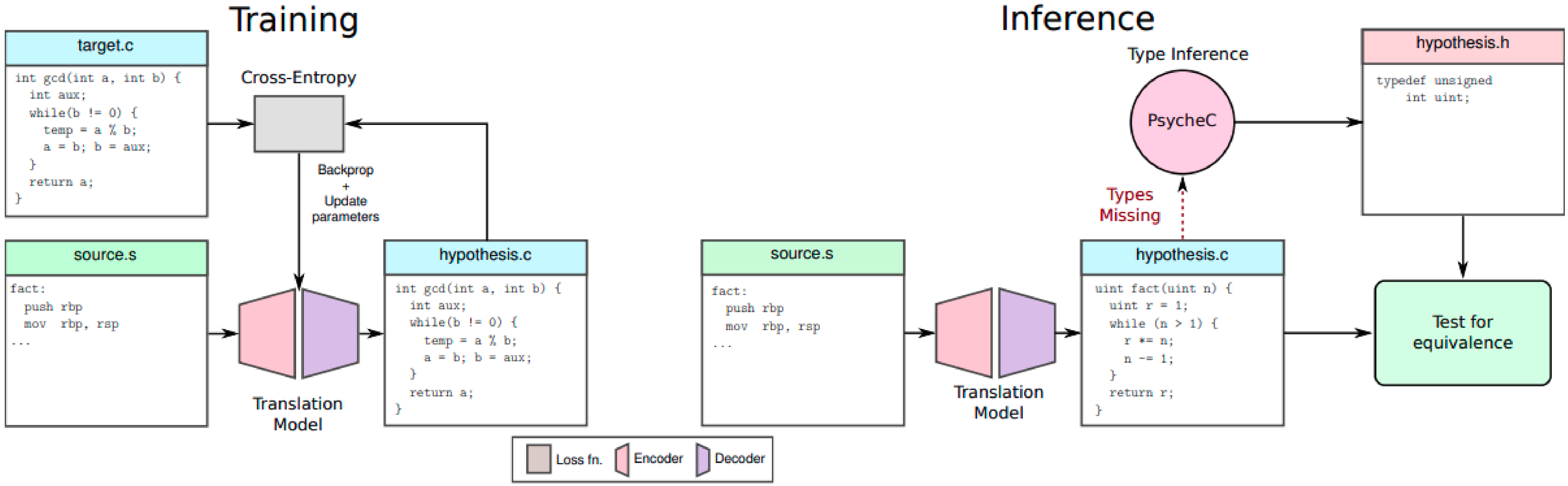
3

```
void add(int* dest, int* src, int n) {
  if (n <= 0) {
    return;
  }
  // If n is smaller than 2, use a loop
  // to add the values
  if (n < 2) {
    for (int i = 0; i < n; i++) {
      dest[i] += src[i];
    }
    return;
  }

  // Align dest pointer to
  // 16-byte boundary
  int* aligned_dest = (int*)
     (((uintptr_t) dest + 15) & -15);

  // Calculate the number of elements
  // before the aligned dest pointer
  int pre_count = (aligned_dest - dest);

  // Add the unaligned elements
  for (int i = 0; i < pre_count; i++) {
    dest[i] += src[i];
  }

  // Add the aligned elements
  int count = (n - pre_count) / 4;
  __m128i* sse_src =
        (__m128i*) (src + pre_count);
  __m128i* sse_dest =
        (__m128i*) (aligned_dest);
  __m128i xmm1 =
        _mm_loadu_si128(sse_src);
  for (int i = 0; i < count; i++) {
    __m128i xmm0 =
      _mm_load_si128(sse_dest);
    xmm0 = _mm_add_epi32(xmm0, xmm1);
      _mm_store_si128(sse_dest, xmm0);
    sse_src++;
    sse_dest++;
  }

  // Add the remaining elements
  int remaining = n -
    (pre_count + count * 4);
  for (int i = 0; i < remaining; i++) {
    dest[pre_count + count * 4 + i] +=
       src[pre_count + count * 4 + i];
  }
}
```

Can we escape the readability-correctness trade-off?



SLaDe architecture: small Transformer + Type Inference



SLaDe: both correct and readable!

Original Source

```
2
void add(int *list, int val, int n) {
  int i;
  for (i = 0; i < n; ++i) {
    list[i] += val;
  }
}
```

```
void add(int a[], int x, int y) {
  int i;
  for (i = 0; i < y; i++) {
    a[i] += x;
  }
}
6
```

SLaDe

Datasets: AnghaBench (large-scale training and evaluation)

AnghaBench: a Suite with One Million Compilable C Benchmarks for Code-Size Reduction

Anderson Faustino da Silva
Department of Informatics
UEM, Brazil
anderson@din.uem.br

Bruno Conde Kind
Department of Computer Science
UFMG, Brazil
condekind@dcc.ufmg.br

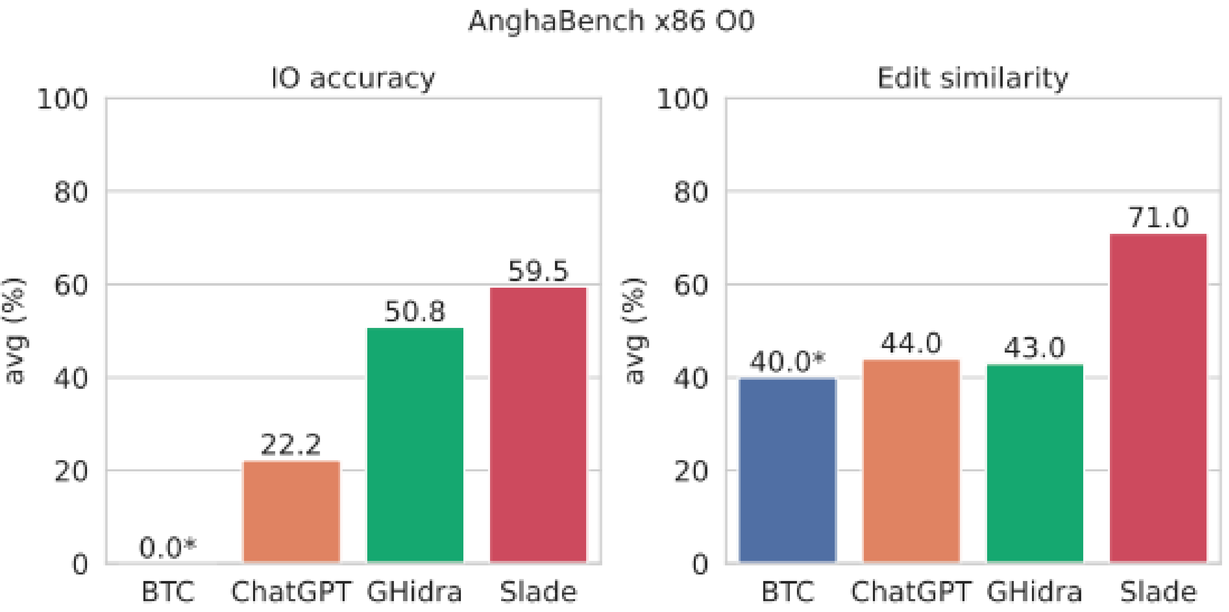
José Wesley de Souza Magalhães
Department of Computer Science
UFMG, Brazil
josewesleysouza@dcc.ufmg.br

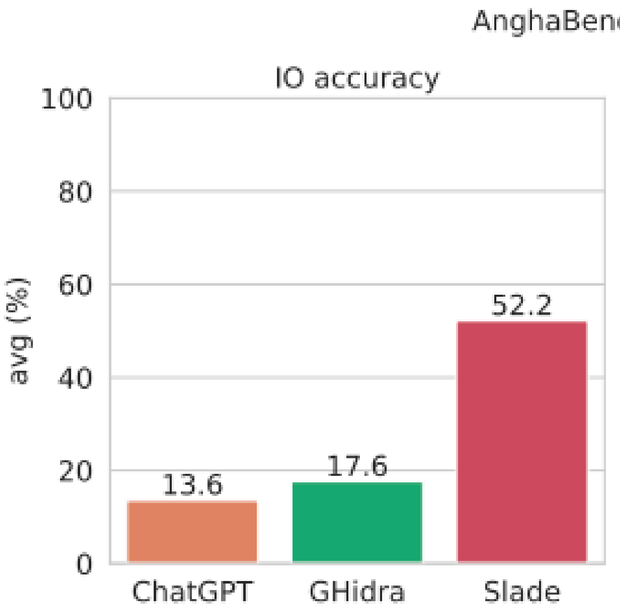
Jerônimo Nunes Rocha
Department of Computer Science
UFMG, Brazil
jeronimonunes@dcc.ufmg.br

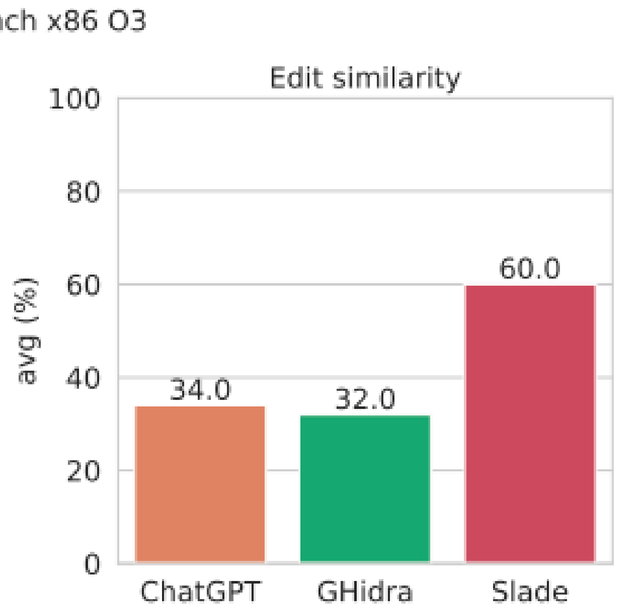
Breno Campos Ferreira Guimarães
Department of Computer Science
UFMG, Brazil
brenosfg@dcc.ufmg.br

Fernando Magno Quintão Pereira
Department of Computer Science
UFMG, Brazil
fernando@dcc.ufmg.br

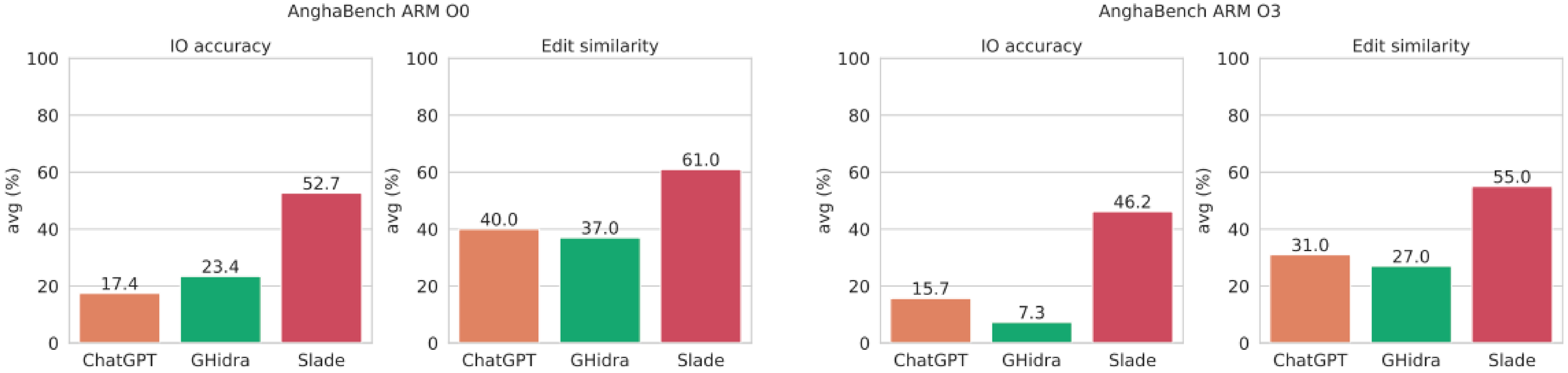
x86: 3x improvement on O3



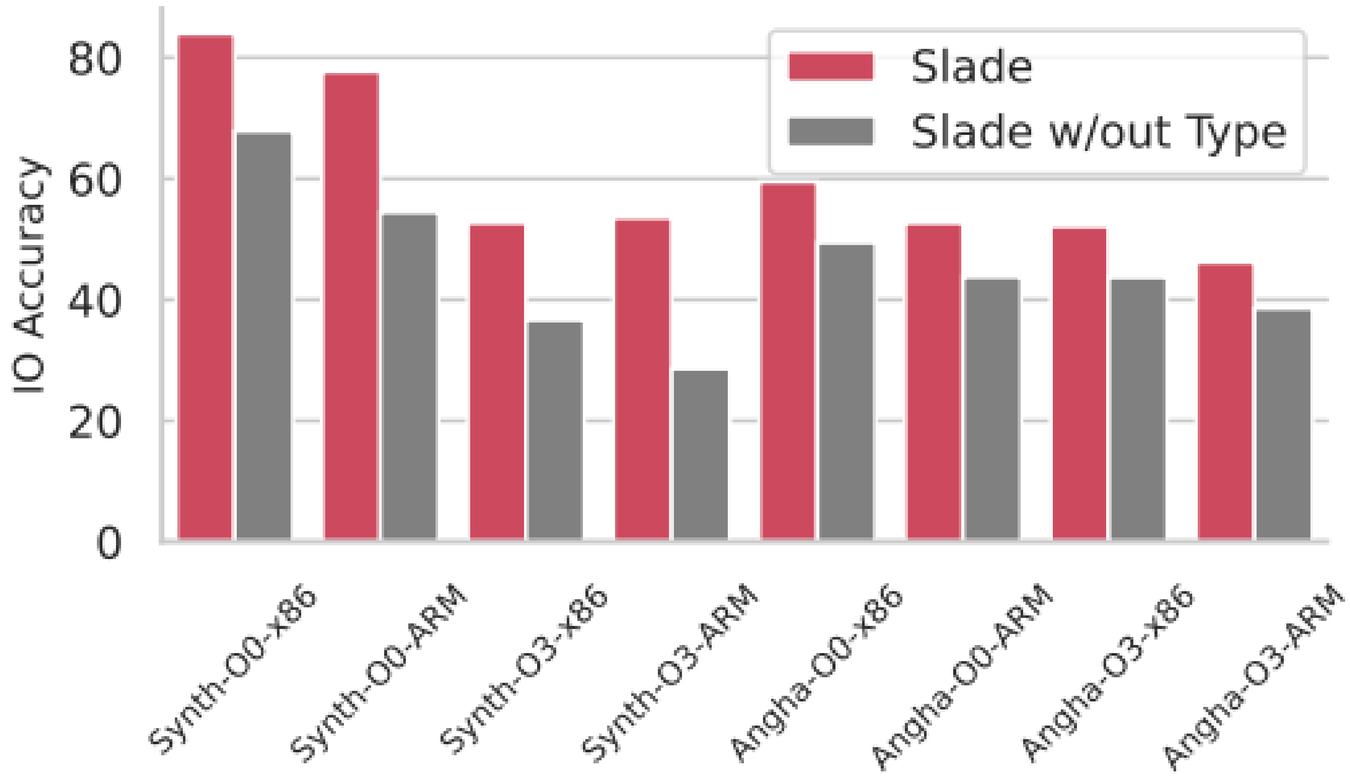




ARM: 3x to 6x improvement on O3



Effect of type inference: 14% average improvement



Conclusion and Future work

We can overcome readability vs correctness

Despite LLMs, specialized, small Transformer models have huge potential

Success will depend on combing program analysis and neural approaches

Ongoing and future work:

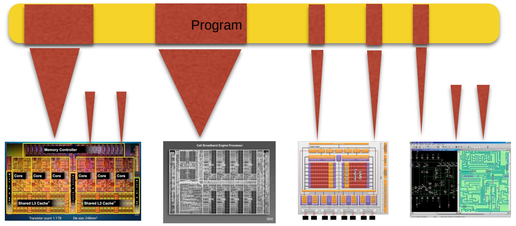
- multi-modal program synthesis

- neural compilation

- IO test generation, program evaluation

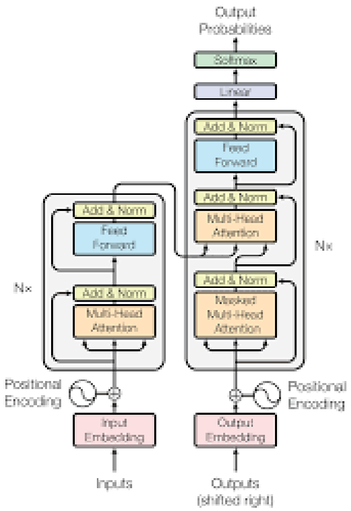
Summary

Matching Hardware to Software
- enables hardware innovation



Lifting to APIs and DSLs exploits compiler investment

Compilers need to invest in new technologies
- program synthesis
- neural machine translation



New technologies + endless automation = bridging software/hardware gap