Compiler 2.0

Why We Need to Modernize Our Compiler Stack and Some Ideas on What We Should Do

Saman Amarasinghe

Charith Mendis, Yishen Chen,
Ajay Brahmakshatriya, Alex Renda,
Cambridge Yang, Yewen Pu, Michael Carbin

Massachusetts Institute of Technology

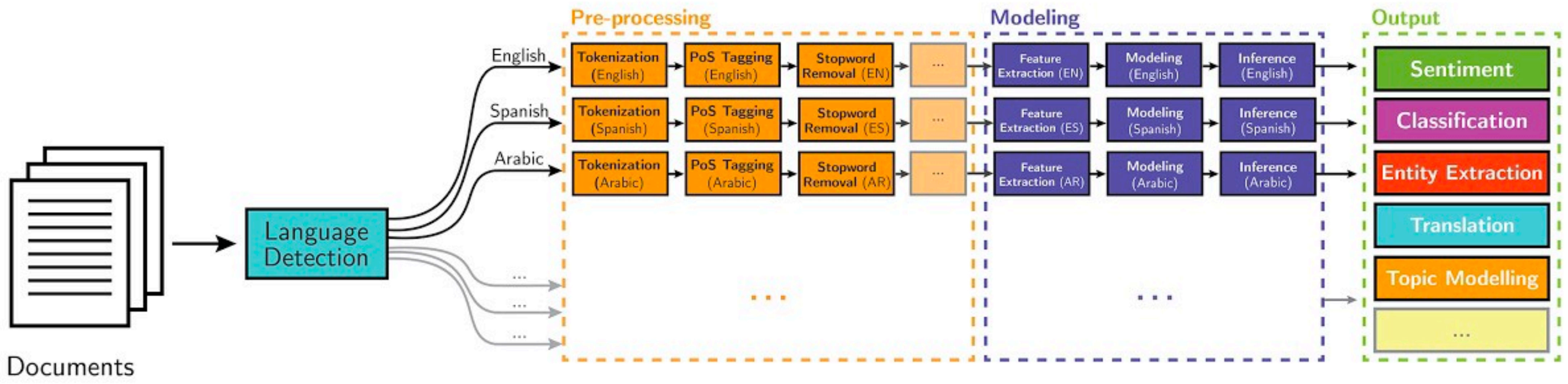
MIT CSAIL
compute. collaborate. create.

When I was a graduate student in the 1990's

SUIF Compiler Group at Stanford

Language Processing Software in the 1990's

Natural Language Processing

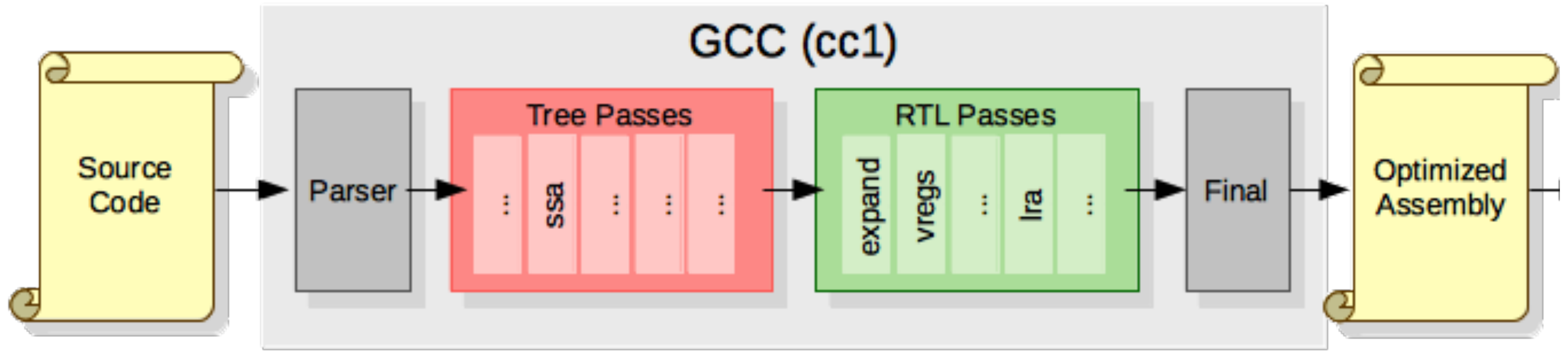


Rule-based Machine Translation (RBMT)

Components

- SL morphological analyser
- SL parser
- Translator
- TL morphological generator
- TL parser
- SL dictionary
- Bilingual dictionary
- TL dictionary

Programming Language Processing



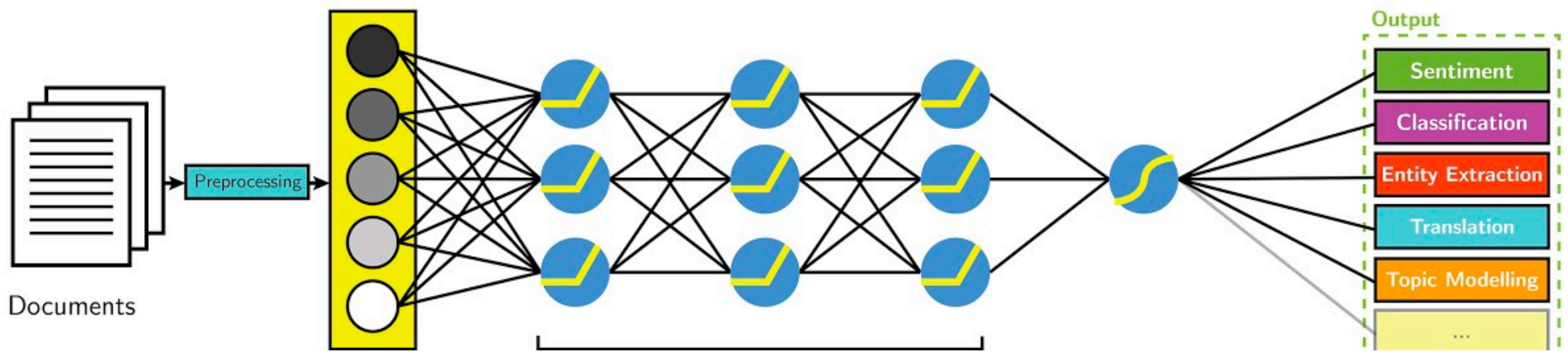
GCC Compiler Flow

Components

- Lexer
- Parser
- Semantic Analyser
- Intermediate Code Generator
- Code optimizer
- Low Level Code Generator

Language Processing Software in the 2020’s

Natural Language Processing



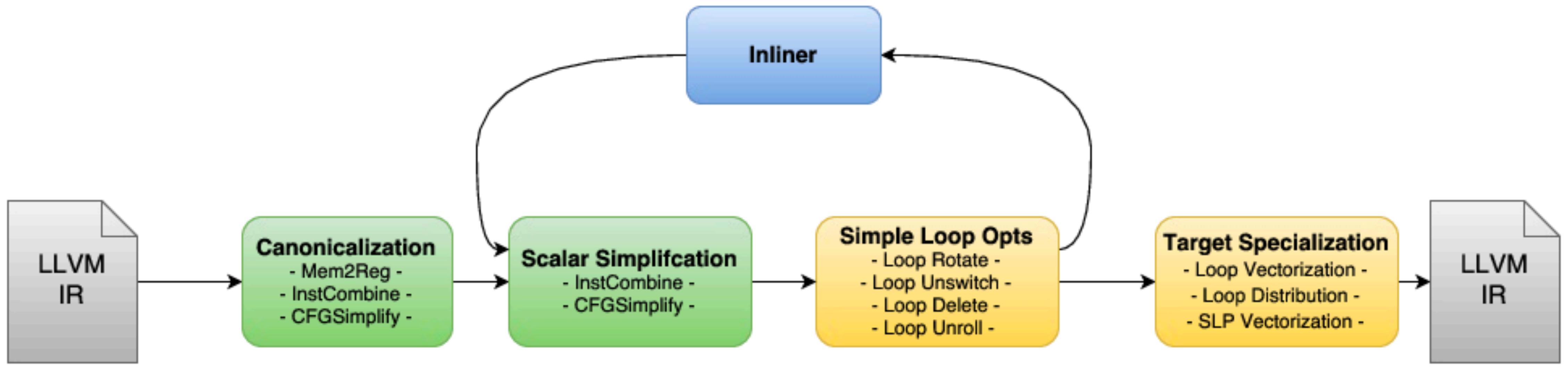
Neural Machine Translation (NMT)

Components
- Sequence to sequence model
  - Encoder
  - Decoder

Sequence to Sequence Learning with Neural Networks
Sutskever, et. al (NIPS 2014)

Attention is all you need
Vaswani, et. al (NIPS 2017)

Programming Language Processing



LLVM Compiler Flow

Components
- Lexer
- Parser
- Semantic Analyser
- Intermediate Code Generator
- Code optimizer
- Low Level Code Generator

Why haven't compilers changed?

Hypothesis: They are so good, no need to change

- ~~Compilers extract most performance from high-level programs~~
  - Matrix Multiply Example

```
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) {
    for (int k = 0; k < n; ++k) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup
1	C	1155.77	1.00	1
2	+ interchange loops	177.68	6.50	7
3	+ optimization flags	54.63	3.25	21
4	+ Parallel loops	3.04	17.97	380
5	+ tiling	1.79	1.70	646
6	+ Parallel divide-and-conquer	1.30	1.38	889
7	+ AVX intrinsics	0.39	1.76	2964

Why haven't compilers changed?

Hypothesis: They are so good, no need to change

- ~~Compilers extract most performance from high-level programs~~
- ~~Compilers have consistently contributed to performance~~

<u>Proebsting's Law</u>: Compiler Advances Double Computing Power Every 18 Years

<u>Moore's Law</u>: The # of Transistors that Fits on a Computer Chip will Double Every 18 Months

Why haven't compilers changed?

~~Hypothesis: They are so good, no need to change~~

- ~~Compilers extract most performance from high-level programs~~
- ~~Compilers have consistently contributed to performance~~
- ~~Compilers are relatively easy to create and maintain~~

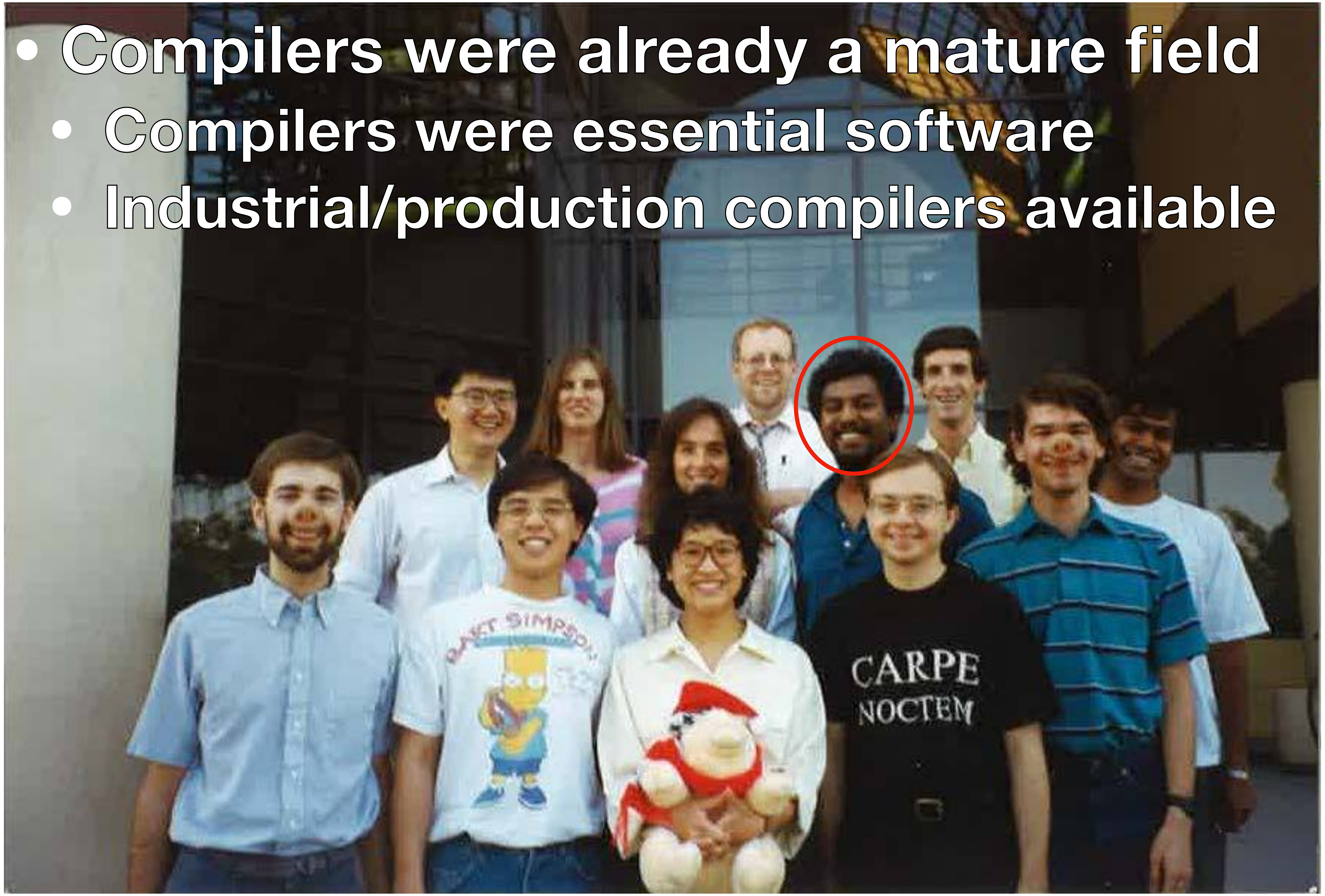
Compiler	Year Started	# of Developers	Lines of Code	Estimated Cost
GCC 9.2.0	[illegible]	[illegible]	[illegible]	$ [illegible]747,278
LLVM 8.0.1	2001	1,210	6,877,[illegible]	$ 529,894,190
OpenJDK 14+10	2007	883	7,955,827	$ 616,517,789
v8 7.8.112	2008	736	3,043,793	$ 225,195,832
Rust 1.37.0	2010	2,737	852,877	$ 59,109,425
Swift	2010	857	665,238	$ 45,535,689
Intel Graphics Compiler 1.0.10	2018	149	684,688	$ 46,934,626

It is High Time to Fundamentally Redesign our Compiler Stack

Compiled by Chris Cummins

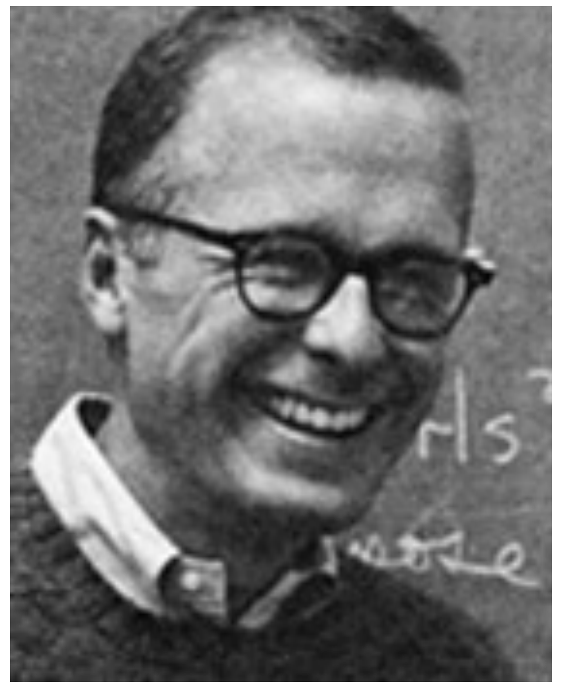
When I was a graduate student in the 1990's

- Compilers were already a mature field
  - Compilers were essential software
  - Industrial/production compilers available



SUIF Compiler Group at Stanford

Compilers became Essential Software at an Early Era



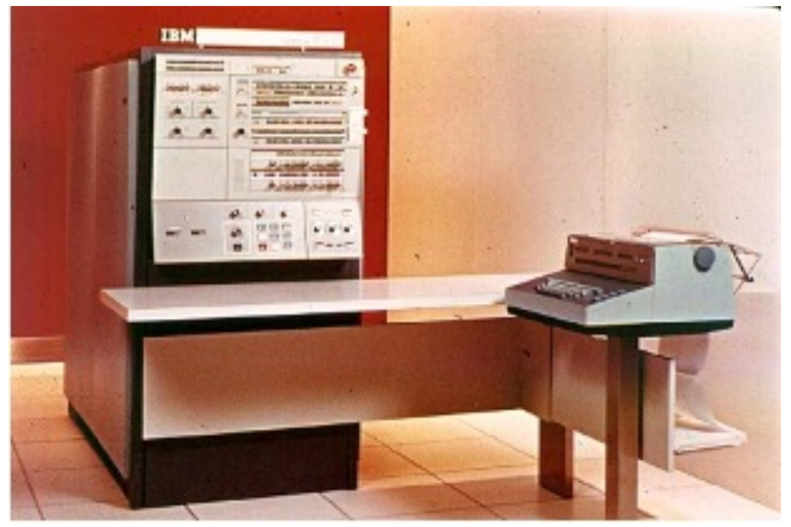
John
Backus

- FORTRAN language and compiler, invented in 1957, was extremely successful
  - High-level languages, compiled to assembly, became the norm
- Proliferation of ISAs in the 1970's and 1980's
  - IBM, Intel x86, Motorola 68000, Sun Sparc, SGI R4400, DEC Alpha
  - High-level languages such as C and Fortran was the only way forward.

Compilers became Essential Software at an Early Era

- But the computers compilers ran on were underpowered

IBM System/360



Launched: 1964
Clock rate: 33 KHz
Data path: 32 bits
Memory: 524 Kbytes
Cost: $250,000

DEC PDP-11

Launched: 1970
Clock rate: 1.25 MHz
Data path: 16 bits
Memory: 56 Kbytes
Cost: $20,000

SUN 100

Launched: 1982
Clock rate: 8 MHz
Data path: 32 bits
Memory: 256 Kbytes
Cost: $8,900

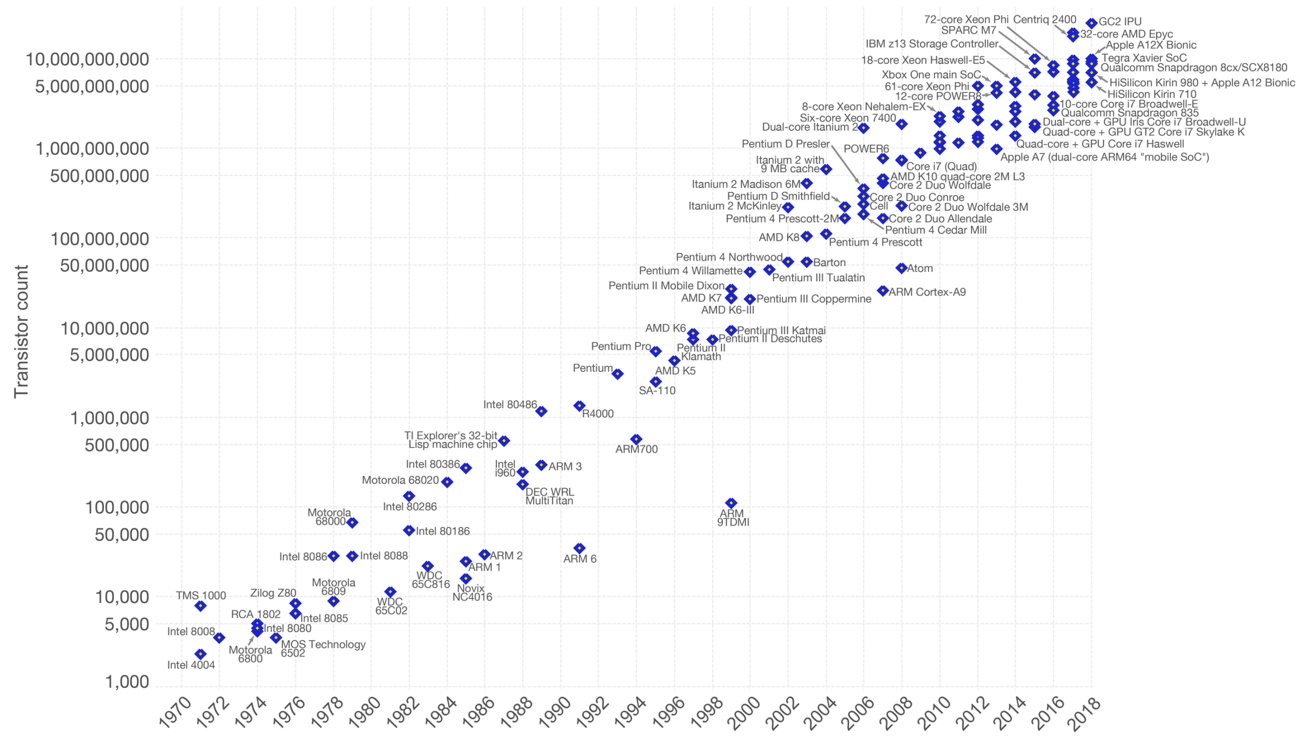
SGI IRIS Indigo

Launched: 1992
Clock rate: 100 MHz
Data path: 64 bits
Memory: 96 Mbytes
Cost: $7,995

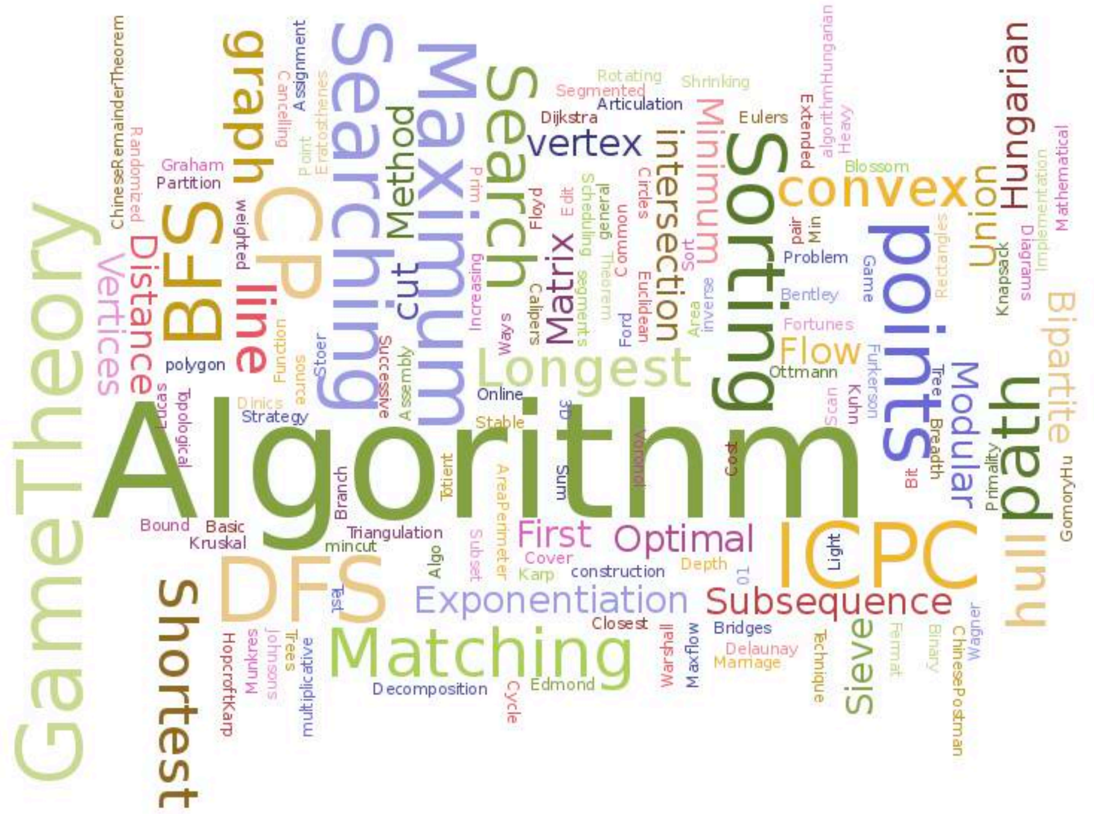
- Compilers were the biggest programs these machines ran
    - Compilers were designed to work in this paucity environment
- Many of those decisions still persist!

What Changed in 30 years?

- More Computing Power
  - Faster CPUs with multicores, GPUs & accelerators
  - More memory and storage
  - Cloud computing



- Better/Faster Algorithms
  - Integer Liner Programming
  - SMT solvers
  - Theorem provers
  - Deep Neural Networks



- More Data
  - Larger, better curated, and globally available data sets

Bringing the Compiler Technology to the 21st Century

- Use more compute power
  - Why not use parallelism, GPUs and the cloud?
- Use better algorithms
  - Complexity of compiler optimizations is due to search
    - Can we search better, faster, simpler?
- Use data better
  - From using data for testing and intuition to learning from data
    - From running SPEC benchmarks to Github mining

Compiler 2.0

- Build Compilers as a Service
- Automate Compiler Construction
- Use Machine Learning

Compiler 2.0

- **Build Compilers as a Service**
- Automate Compiler Construction
- Use Machine Learning

The Structure of a Modern Compiler

<u>Build with ancient technology</u>

- A command line tool
- Running on the developer's workstation (or a local cluster)
- With a single CPU thread
- Sequential execution of passes
  - Prog → AST → IR_1 →…→ IR_n → Assembly

<u>Impact</u>

- Compile time still matters
  - No expensive analyses
  - Limited to no global optimizations
- Memory footprint still matters
  - Highly optimized data structures
  - Limited to no global optimizations
- No path to learn and improve

CaaS: Compilation as a Service

- Access to unlimited processing power
- Access to accelerators
- Access to unlimited memory and storage
- Use of modern system building methods and frameworks
- Ability to learn from everyone and improve over time

- Build LLVM in 90 seconds (vs 10 minutes)
  - Using llama -- A CLI for outsourcing computation to AWS Lambda
  - Many related works of General Offloading
    Eg: “From Laptop to Lambda..” USENIX 2019

Analysis & Transformations with Serverless

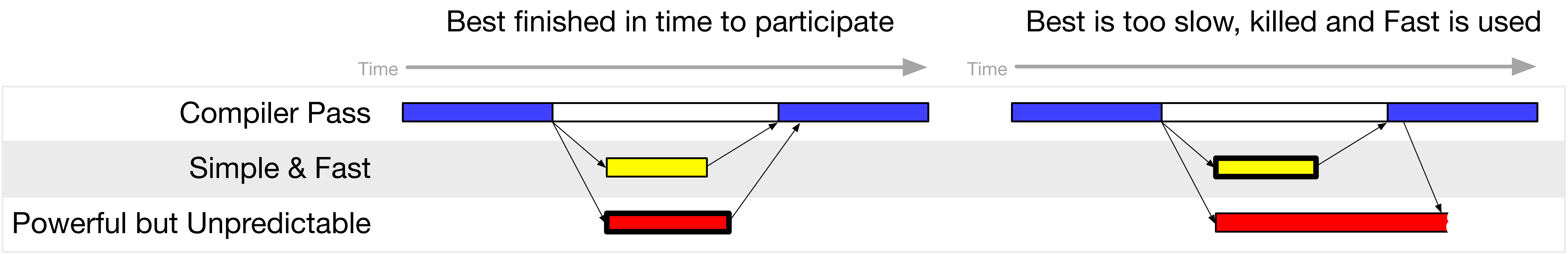
- Most of the compiler is parallel and stateless
  - Passes → Files → Functions → Basic Blocks → Statements
- Fits well to the serverless computing paradigm
- Scale-out for to match any program size
  - Size of functions and basic blocks are normally constant
  - Constant compile time for any size program!

Interprocedural Analysis with Distributed Graph Processing

- Compilers rarely/never do global analysis on real applications
  - Eg: Interprocedural type specialization, constant prop., inlining etc.
  - Too slow or too much memory consumption
  - Many papers written, never used in practice :(
- On the cloud, fits nicely to distributed graph processing
  - Many frameworks available, scales well, may even use GPUs

Expensive and Unpredictable Analysis using Redundancy Techniques used in Latency Reduction

- Production compilers don't use expensive analyses or analyses with unpredictable runtimes
  - Ex: Polyhedral analysis, program synthesis etc.
  - Many papers written, never used in practice :(
- Many modern systems use redundancy to hide tail latency
- Compilers can use redundancy to incorporate powerful but unpredictable analyses



Ansel et. al "SiblingRivalry: Online Autotuning Through Local Competitions." [CASES'12]

Less Optimize Data-structures leading to More Capabilities with Ease of Use

- Control Flow Graphs (CFGs) are a compact data-structure
- However, CFGs present many code motion obstacles
  - Ex: hard to hoist a loop-invariant branch out of a loop (Requires major surgery to rewire the CFG)
- Problem: CFG conflates two separate purposes
  - Code Layout (i.e., which instructions should execute together)
  - Control Dependence (i.e., whether they should execute)
- Solution: An IR with first-class Control Dependence

Flair: A Flat IR with First-Class Control Dependence

Chen et. al “All you need is SLP: Systematic Control-Flow Vectorization for Superword-Level Parallelism” [PLDI’22]

- symbolic boolean formula indicating execution condition
- tracks the control dependence for each instruction separately
- code motion is straightforward

```
int x = def();
int y = def();
int arr[n];
for (int i = 0; i < n; i++) {
   if (x < y) {
     int t = x + y;
     int t2 = t + arr[i];
     extern_func(t2);
   }
}
```

```
x = def()            ; true
y = def()            ; true
loop {
  i = phi (0, i’)    ; true
  c = x < y          ; true
  t = x + y          ; c
  ld = load &arr[i]  ; c
  t2 = t + ld        ; c
  extern_func(t2)    ; c
  i’ = i + 1         ; true
  lt_n = i < n
} while (lt_n) ; true
```

```
x = def()            ; true
y = def()            ; true
c = x < y            ; true
t = x + y            ; c
loop {
  i = phi (0, i’)    ; true
  ld = load &arr[i]  ; c
  t2 = t + ld        ; c
  extern_func(t2)    ; c
  i’ = i + 1         ; true
  lt_n = i < n
} while (lt_n) ; true
```

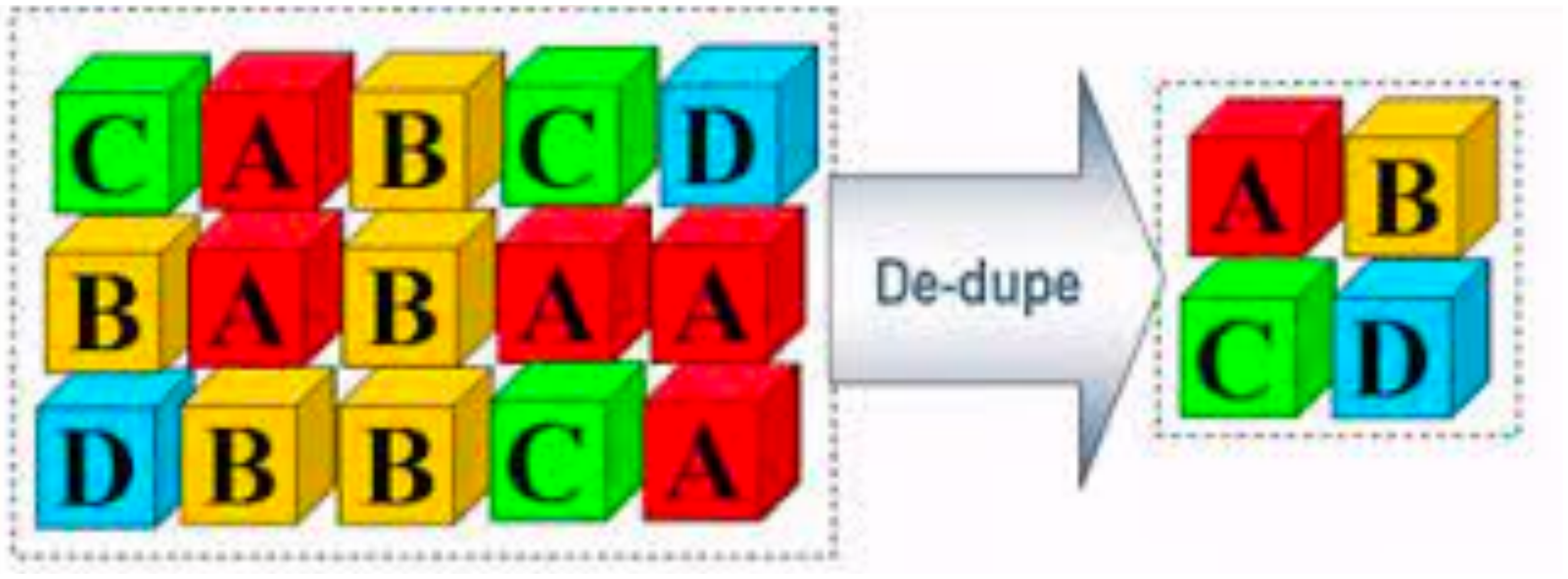
Flair: A Flat IR with First-Class Control Dependence

Chen et. al "All you need is SLP: Systematic Control-Flow Vectorization for Superword-Level Parallelism" [PLDI'22]

- Flair makes many transformations trivial
  - duplicating branches
  - partially duplicating basic blocks
- Transformation complexity is invariant w.r.t. control-flow nesting

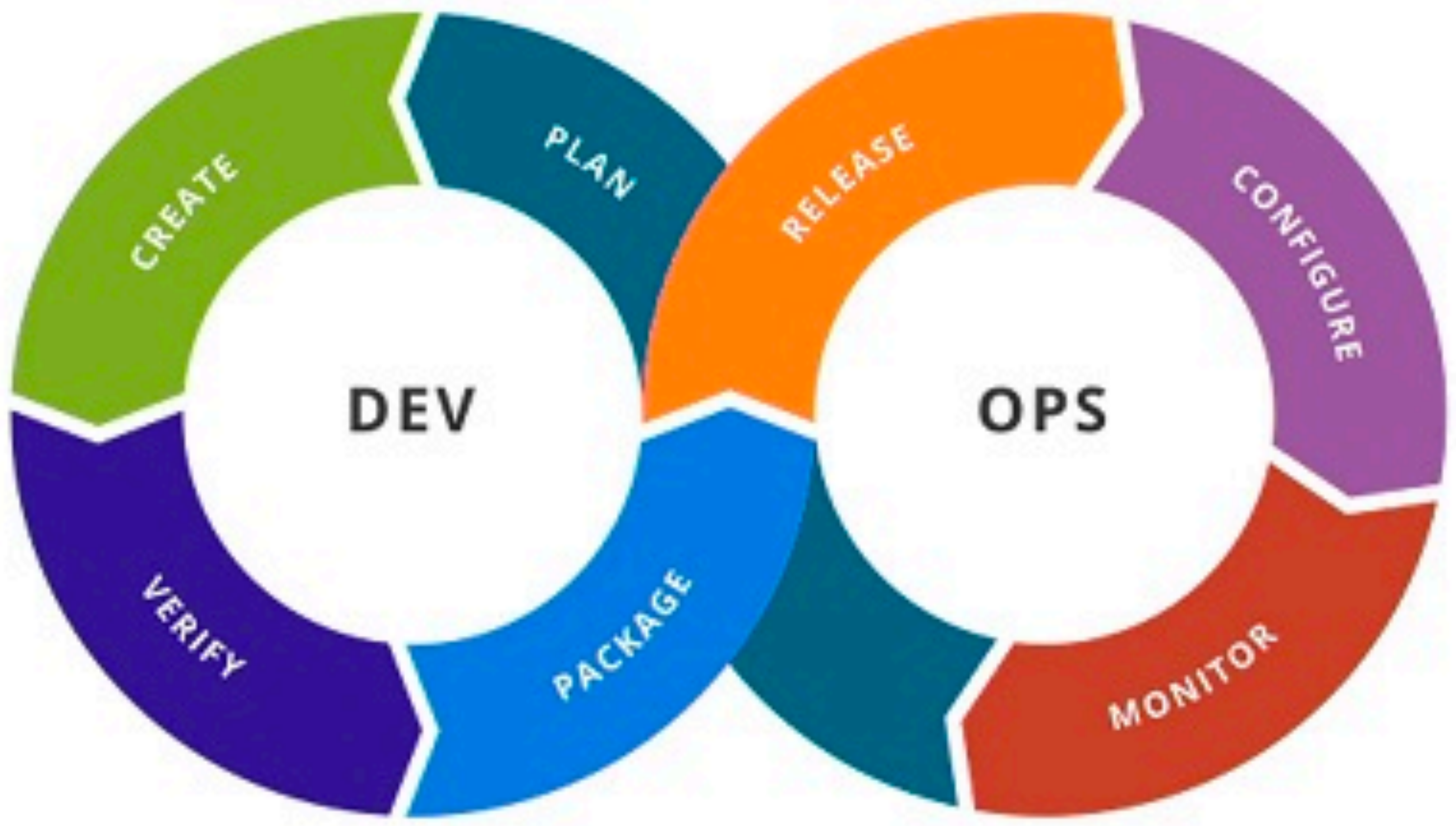
Overall Cost Reduction with Deduplication

- Reuse of compiled files is nothing new
  - Makefiles only compile changed files and their dependencies
- If most programmers use a single CaaS system for compiling
  - Each run is a small modification to a one seen before
  - Most probably exactly the same program as seen before
- Memoization can drastically reduce the cost of compilation
  - As done by many SaaS systems for storage



Centrally Collected Data for Continuous Improvement

- CaaS will see many programs
  - Usage is clear
  - Failures are obvious
- Can use the usage information for continuous improvement



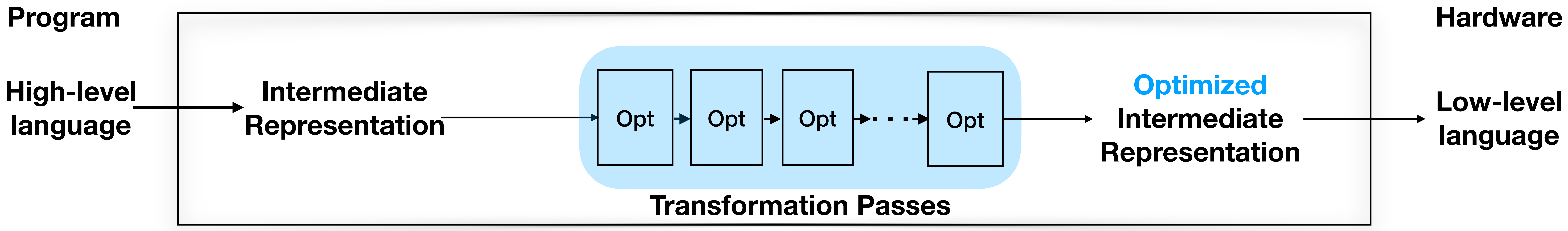
Compiler 2.0

- Build Compilers as a Service
- Automate Compiler Construction
- Use Machine Learning

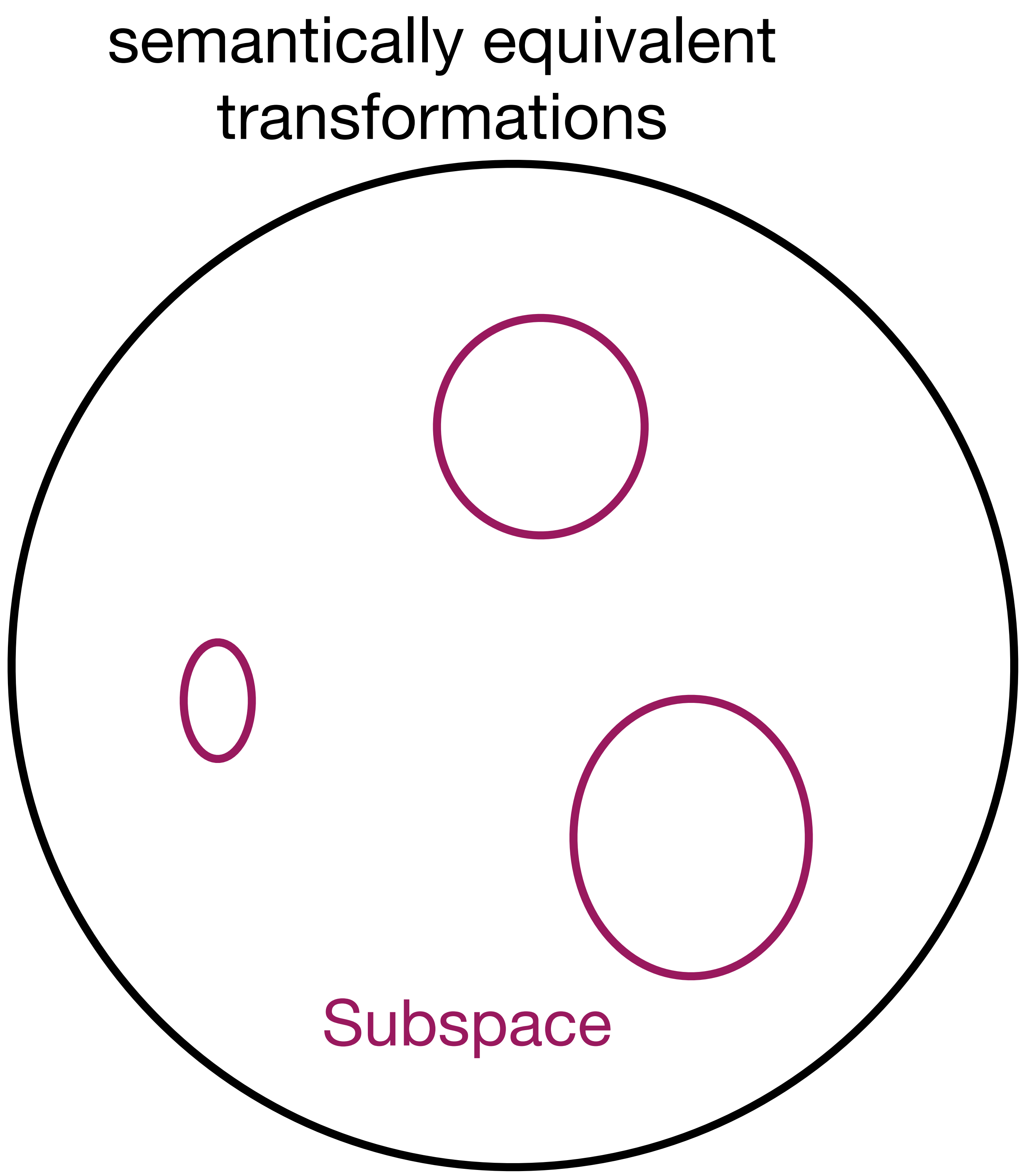
Compiler 2.0

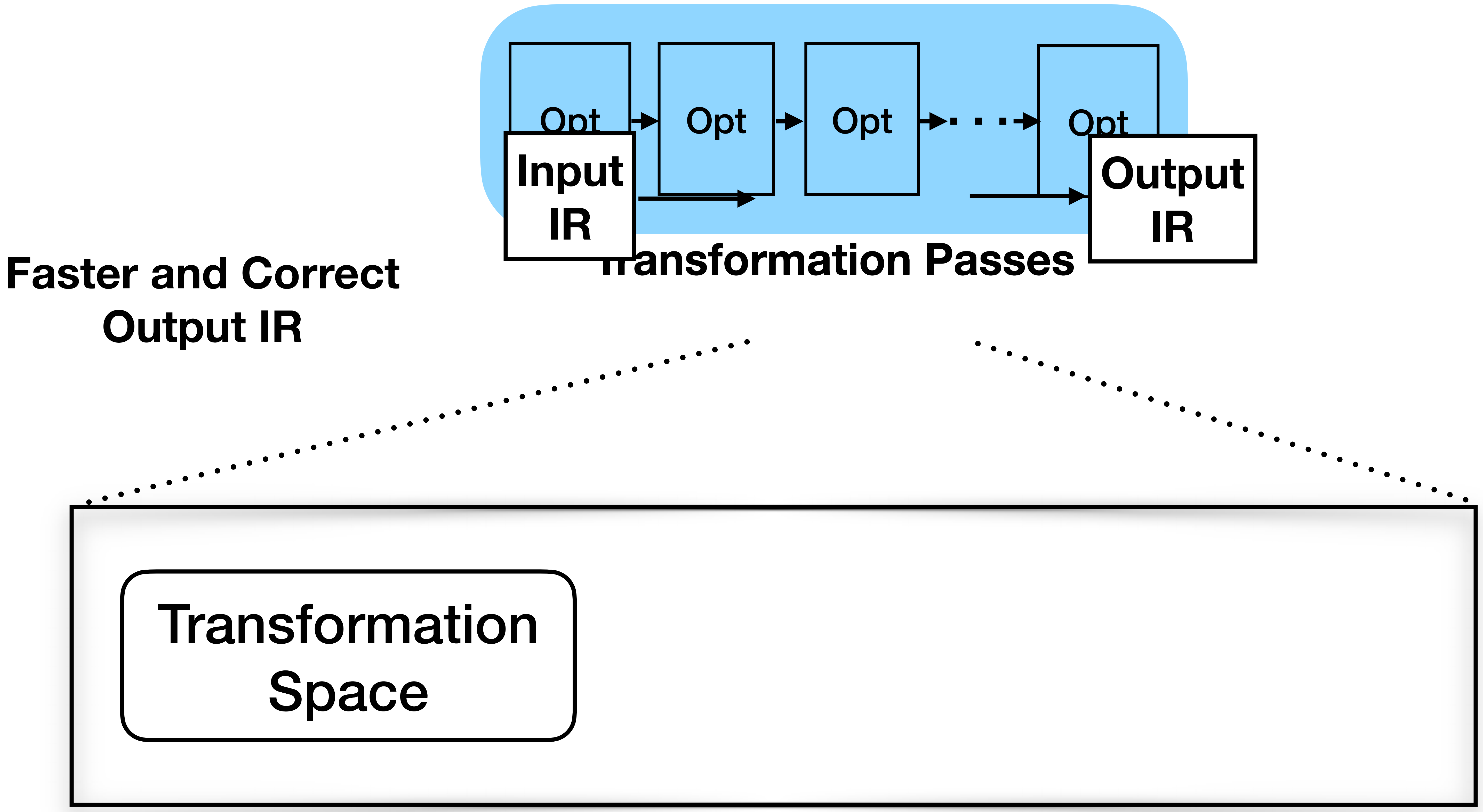
- Build Compilers as a Service
- **Automate Compiler Construction**
- **Use Machine Learning**

Mendis's Model of Compiler Optimization

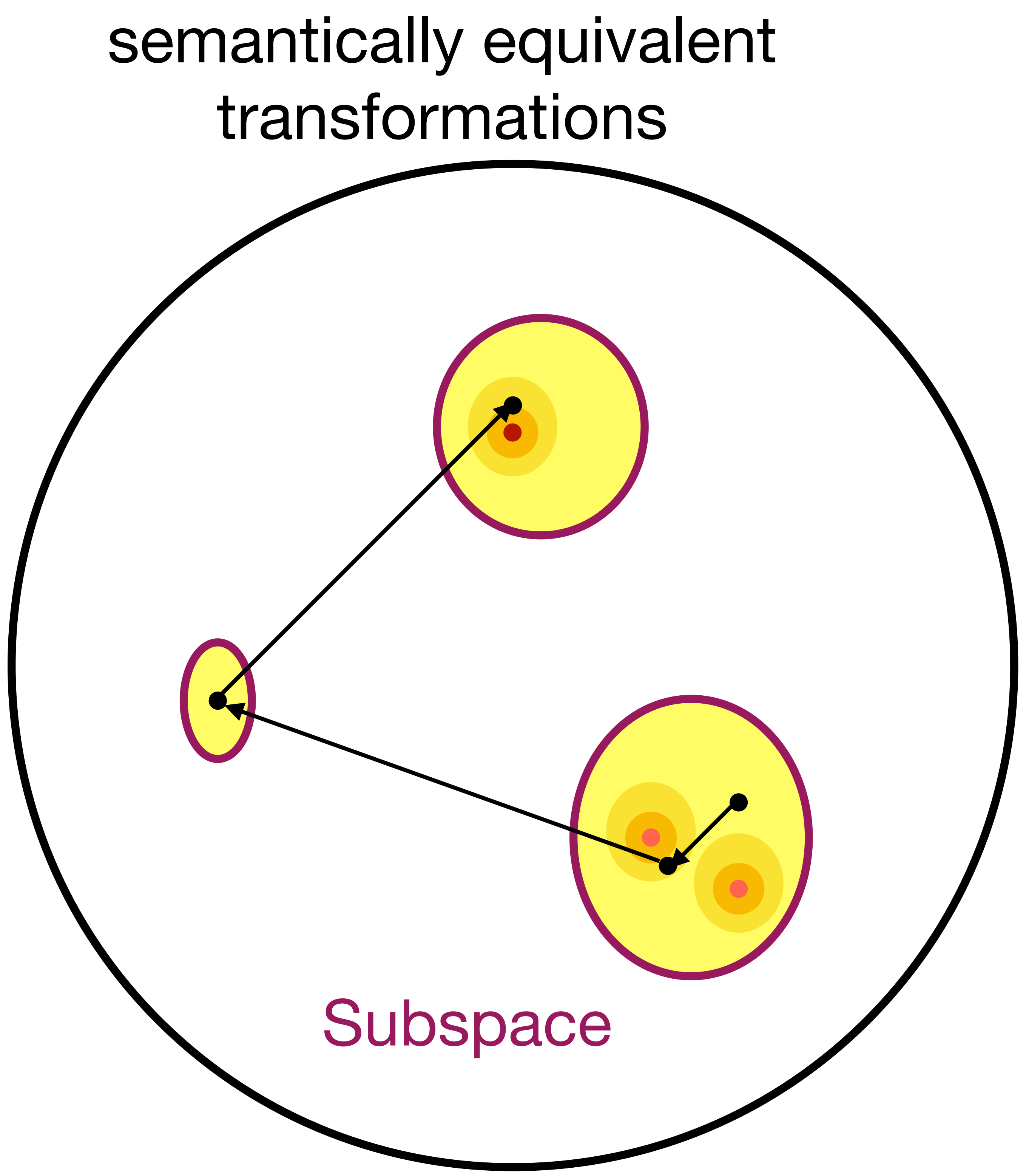


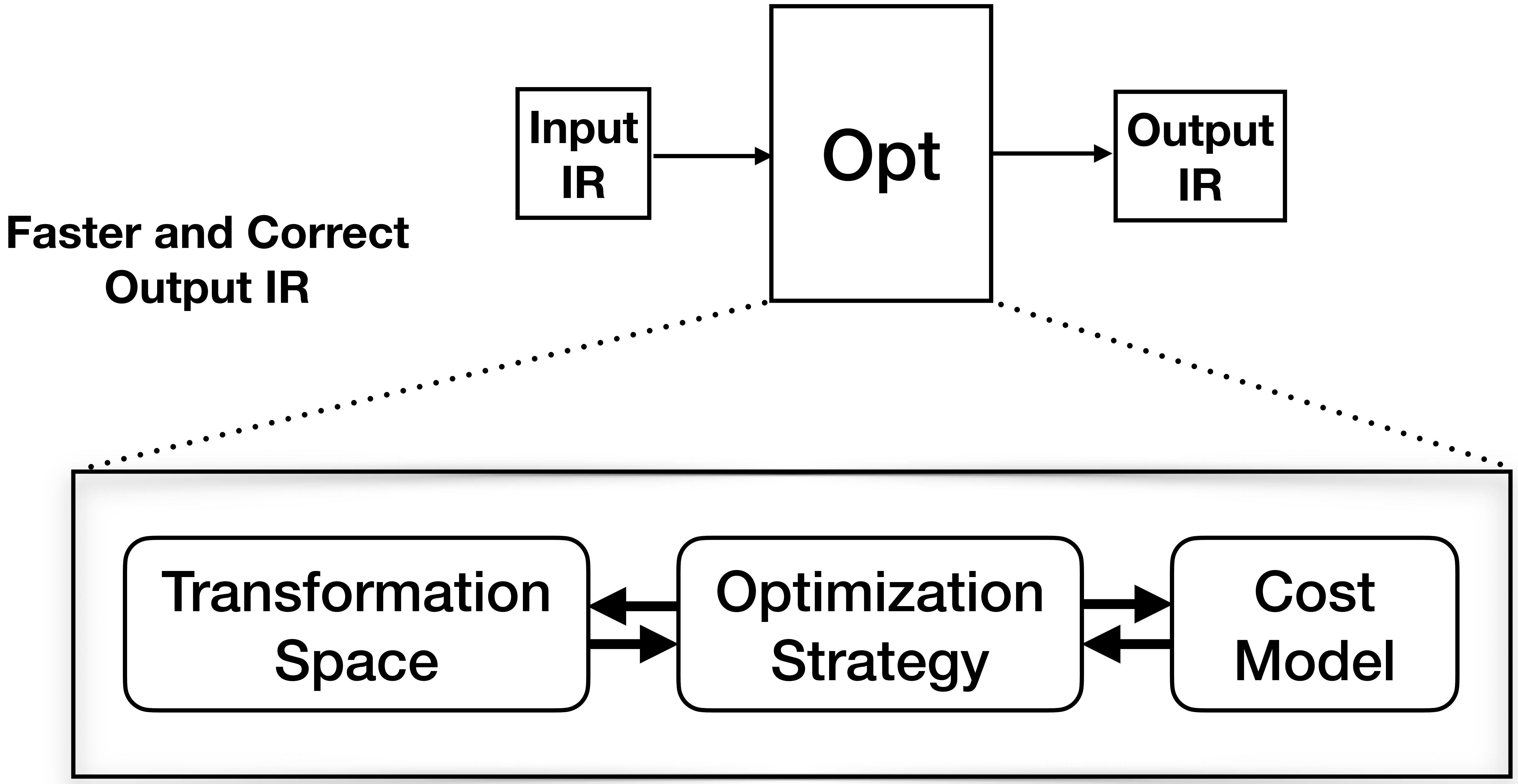
Mendis's Model of Compiler Optimization



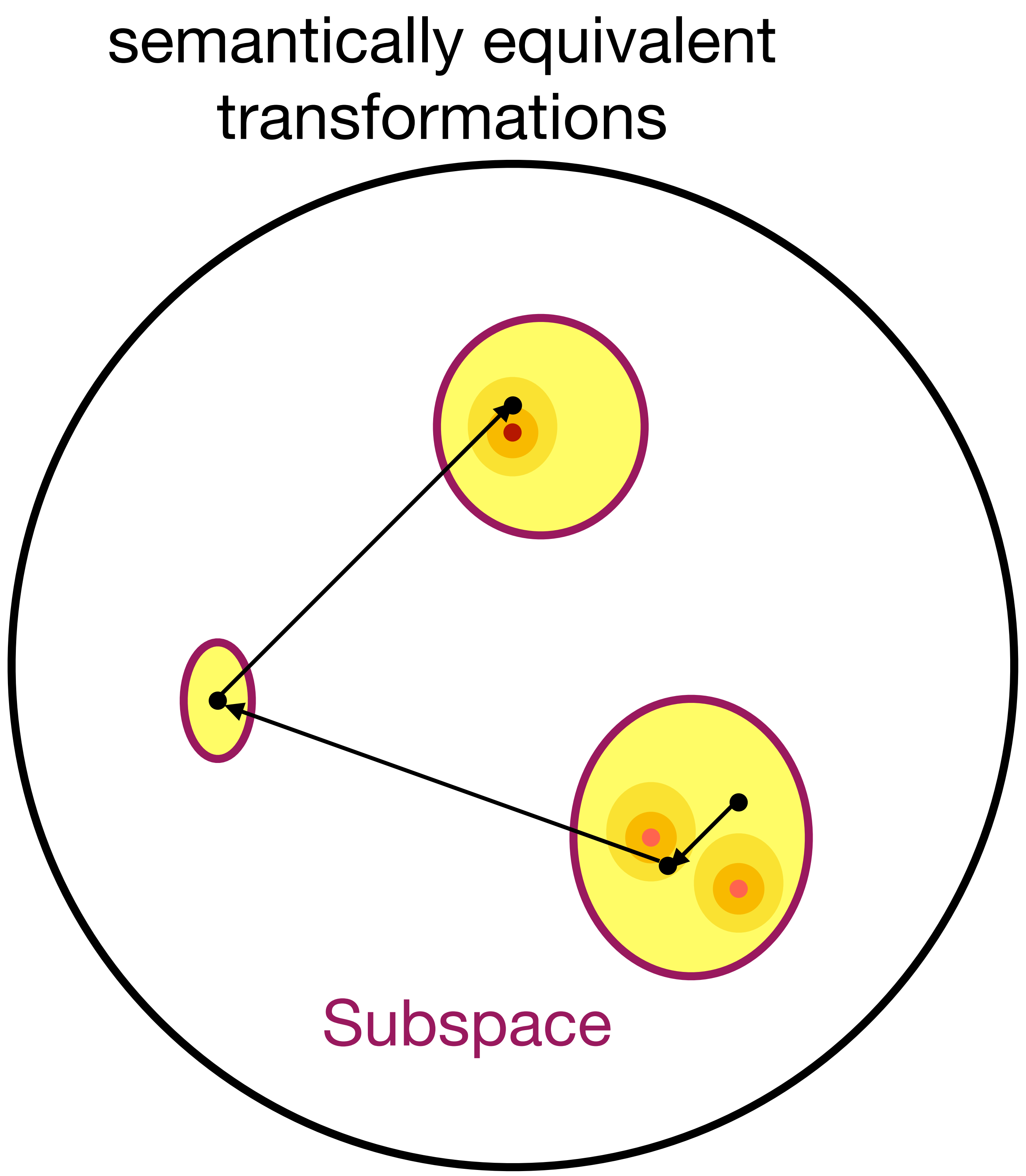


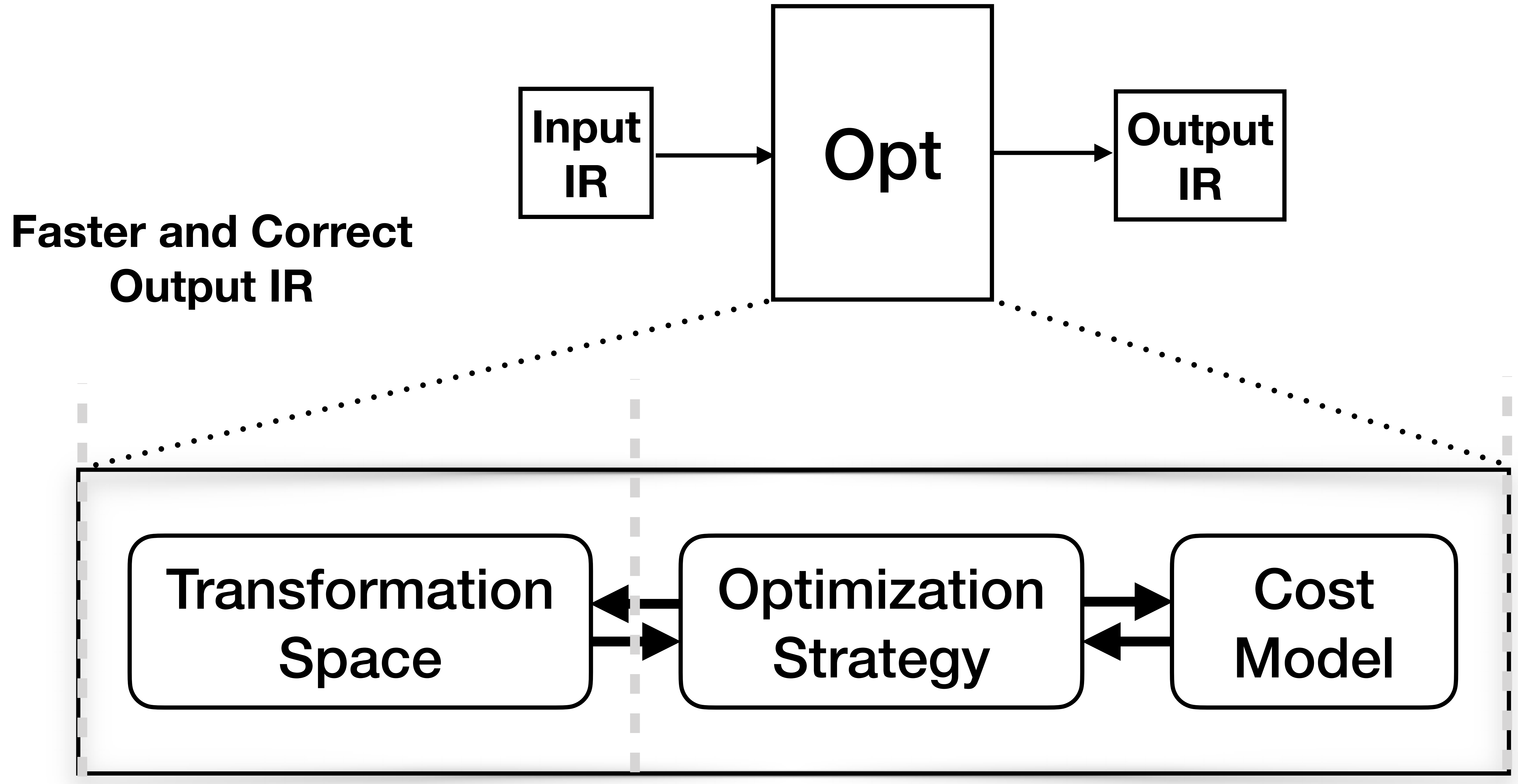
Mendis's Model of Compiler Optimization





Mendis's Model of Compiler Optimization





	Correctness	Optimization
Automate using :	**Program Logics**	**Machine Learning**

Mendis's Model applied to Vectorization

Vectorization	Transformation Space	Optimization Strategy	Cost Model
Manual solutions	Hand-written	Heuristics	Linear
Automated solutions State-of-the-art results Minimal development burden	**Code Generator Generator** Chen et. al [ASPLOS'21] **Program Logics**	**Data-driven** Mendis et. al [NeurIPS'19] **Machine Learning**	**Data-driven** Mendis et. al [ICML'19] Chen et. al [IISWC'19] **Machine Learning**

Today Transformations are Manually Created

Compiler Transformations:

- ❌ Are tedious to develop and maintain

Thousands of contributors and millions of lines of code
(e.g., LLVM: 1,115 contributors and 2.5 million lines)

- ❌ Can easily become stale

Out of over 3,600 x86 instructions, compilers organically generate <1000

Automatically Generate Transformations

- Many relevant Tools created by the PL community
  - Domain Specific Languages (DSLs)
  - Program Synthesis

- Pioneering compilers were early users of these tools
  - Eg: DSLs for lexer and parser generation

- Should aggressively use these tools
  - Generate compiler passes from high-level specifications

VeGen: A Vectorizer Generator

Chen et. al "VeGen: A Vectorizer Generator for SIMD and Beyond" [ASPLOS'21]

Instruction Description

```
<intrinsic tech="MMX" name="_m_pmaddwd">
    <type>Integer</type>
    <CPUID>MMX</CPUID>
    <category>Arithmetic</category>
    <return type="__m64" varname="dst" etype="FP32"/>
    <parameter type="__m64" varname="a" etype="SI64"/>
    <parameter type="__m64" varname="b" etype="SI64"/>
    <description>Multiply packed signed 16-bit integers in "a"
and "b", producing intermediate signed 32-bit integers.
Horizontally add adjacent pairs of intermediate 32-bit integers,
and pack the results in "dst".</description>
    <operation>
FOR j := 0 to 1
    i := j*32
    dst[i+31:i] := SignExtend32(a[i+31:i+16]*b[i+31:i+16]) +
SignExtend32(a[i+15:i]*b[i+15:i])
ENDFOR
    </operation>
    <instruction name="PMADDWD" form="mm, mm"
xed="PMADDWD_MMXq_MMXq"/>
    <header>mmintrin.h</header>
</intrinsic>
```

→ VEGEN → Target-specific Vectorizer

Transformation Space : Scalar → SIMD Vector ✓

Auto-generated — Scalar → non-SIMD Vector ✓

VeGen: A Vectorizer Generator

Chen et. al “VeGen: A Vectorizer Generator for SIMD and Beyond” [ASPLOS’21]

Scalar Program

```c
int16_t A[4], B[4];
int32_t C[2];
void dot_prod() {
    C[0] = A[0] * B[0] + A[1] * B[1];
    C[1] = A[2] * B[2] + A[3] * B[3];
}
```

Instruction Description

```xml
<intrinsic tech="MMX" name="_m_pmaddwd">
    <type>Integer</type>
    <CPUID>MMX</CPUID>
    <category>Arithmetic</category>
    <return type="__m64" varname="dst" etype="FP32"/>
    <parameter type="__m64" varname="a" etype="SI64"/>
    <parameter type="__m64" varname="b" etype="SI64"/>
    <description>Multiply packed signed 16-bit integers in "a"
and "b", producing intermediate signed 32-bit integers.
Horizontally add adjacent pairs of intermediate 32-bit integers,
and pack the results in "dst".</description>
    <operation>
FOR j := 0 to 1
    i := j*32
    dst[i+31:i] := SignExtend32(a[i+31:i+16]*b[i+31:i+16]) +
SignExtend32(a[i+15:i]*b[i+15:i])
ENDFOR
    </operation>
    <instruction name="PMADDWD" form="mm, mm"
xed="PMADDWD_MMXq_MMXq"/>
    <header>mmintrin.h</header>
</intrinsic>
```

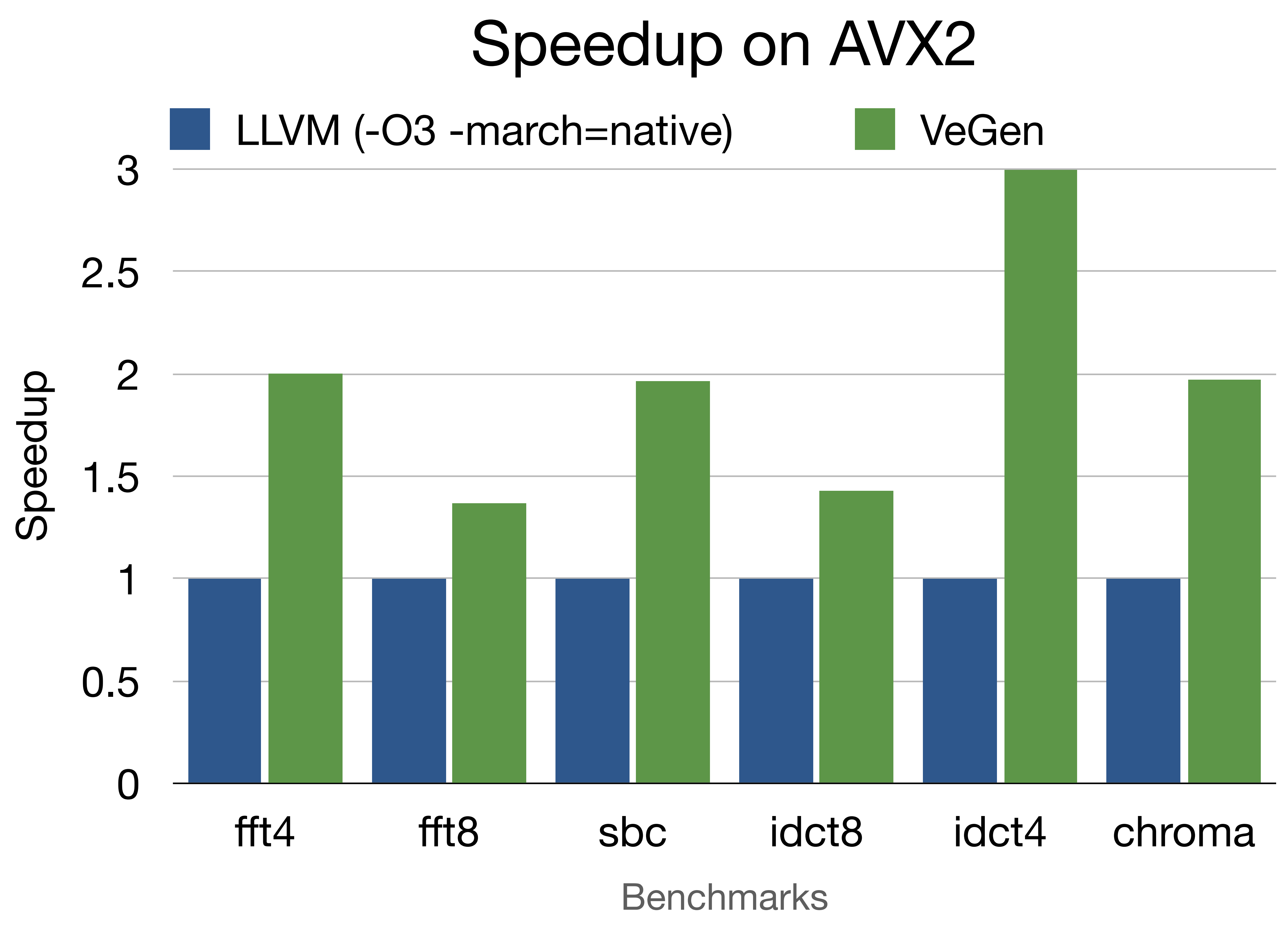
VEGEN

Target-specific Vectorizer

```
vmovd xmm0, [A]
vmovd xmm1, [B]
pmaddwd xmm0, xmm1, xmm0
vmovd [C], xmm0
```

VeGen is better than hand-created LLVM

DSP kernels from FFmpeg and x265



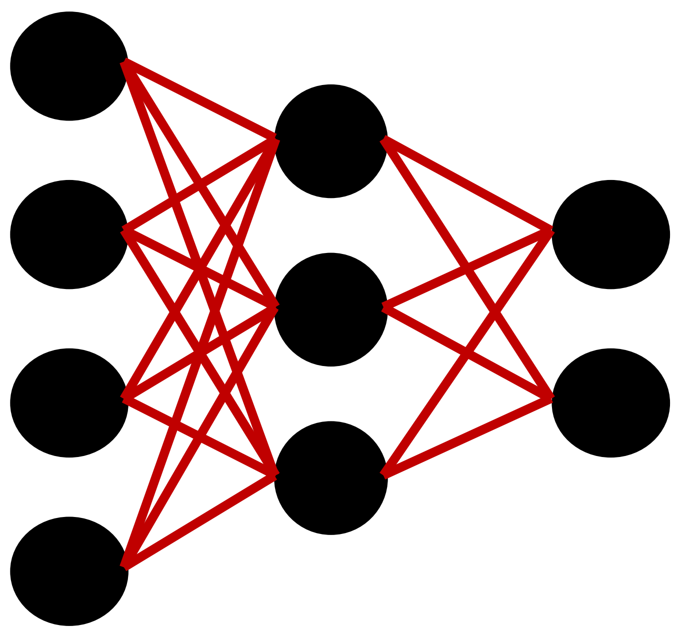
Mendis's Model of Compiler Optimization

Vectorization	Transformation Space	Optimization Strategy	Cost Model
Manual solutions	Hand-written	Heuristics	Linear
Automated solutions State-of-the-art results Minimal development burden	**Code Generator Generator** **Chen et. al [ASPLOS'21]** **Program Logics**	**Data-driven** **Mendis et. al [NeurIPS'19]** **Machine Learning**	**Data-driven** **Mendis et. al [ICML'19]** **Chen et. al [IISWC'19]** **Machine Learning**

Can we naively use ML?

Source Language

```
void dot_16x1x16_uint8_int8_int32(
   uint8_t data[restrict 4],
   int8_t kernel[restrict 16][4],
   int32_t output[restrict 16]) {
    for (int i = 0; i < 16; i++) {
      int32_t acc = output[i];
      for (int k = 0; k < 4; k++) {
          acc += data[k] * kernel[i][k];
      }
      output[i] = acc;}
 }
```



Target Language

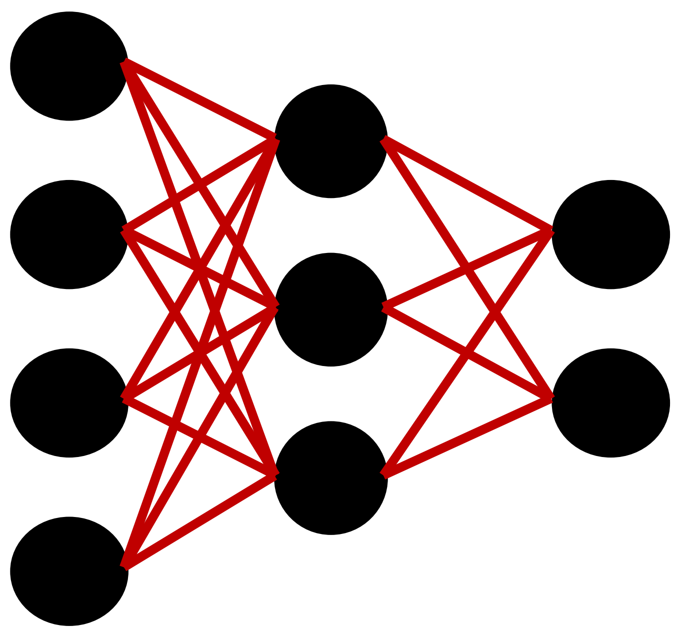
```
vmovdqu64     zmm0, [rdx]
vpbroadcastd zmm1, [rdi]
vpdpbusd      zmm0,  zmm0, [rsi]
vmovdqu64    [rdx],  zmm0
```

Use Neural Machine Translation?

Can we naively use ML?

Source Language

```c
void dot_16x1x16_uint8_int8_int32(
   uint8_t data[restrict 4],
   int8_t kernel[restrict 16][4],
   int32_t output[restrict 16]) {
    for (int i = 0; i < 16; i++) {
      int32_t acc = output[i];
      for (int k = 0; k < 4; k++) {
          acc += data[k] * kernel[i][k];
      }
      output[i] = acc;}
 }
```



Target Language

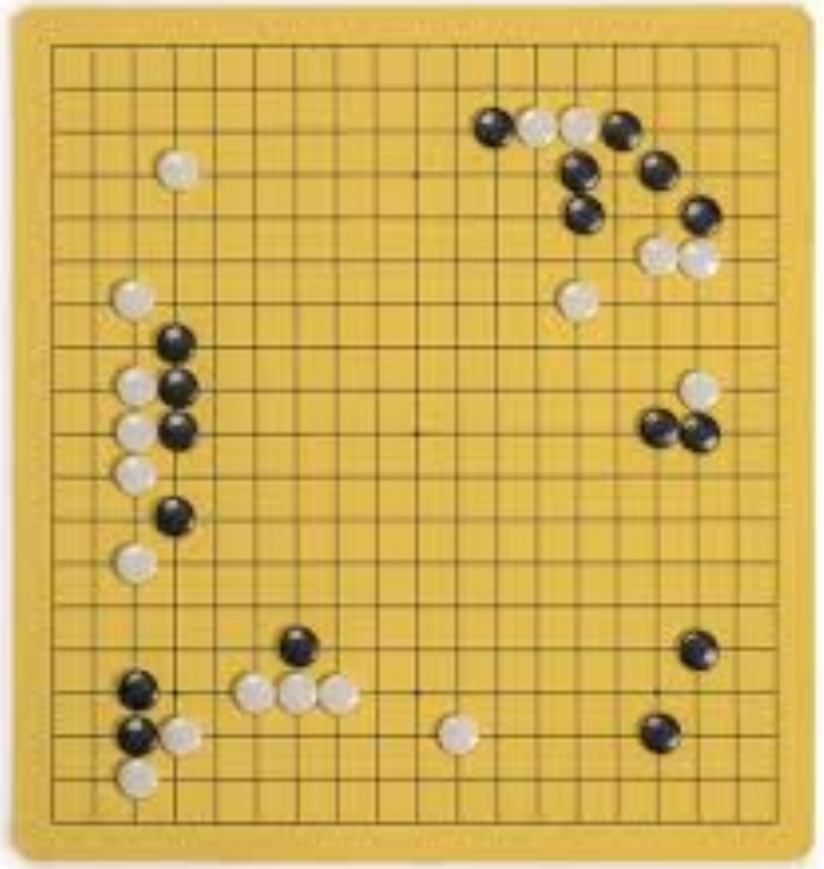
```asm
vmovdqu64    zmm0, [rdx]
vpbroadcastd zmm1, [rdi]
vpdpbusd     zmm0,  zmm0, [rsi]
vmovdqu64   [rdx],  zmm0
```

Use Neural Machine Translation?

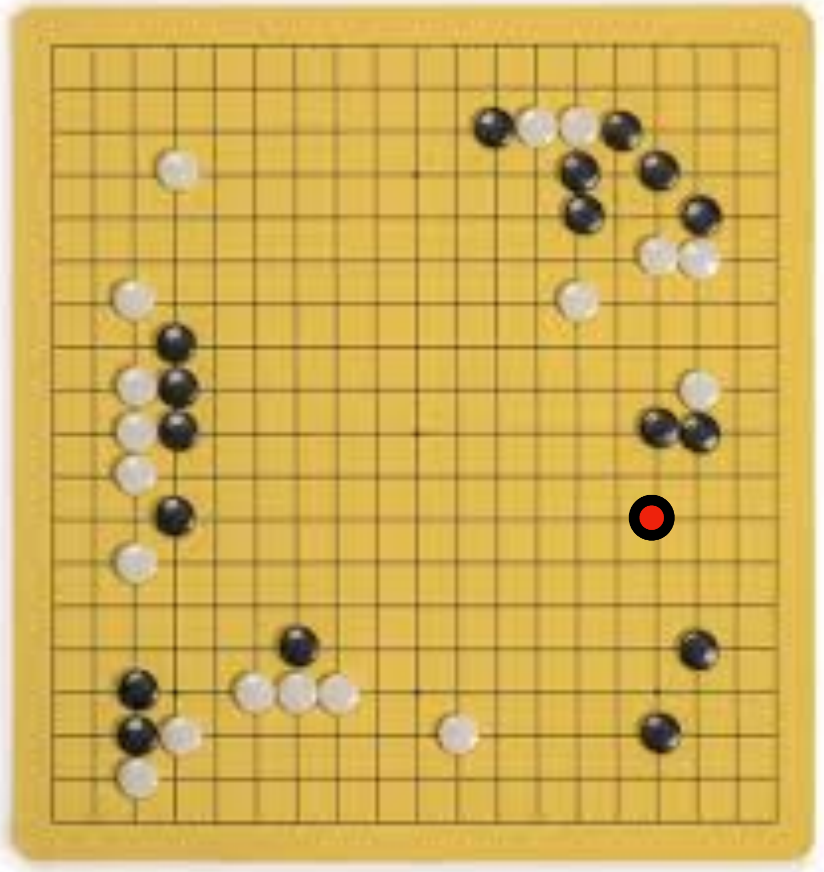
Machine Learning Systems do not guarantee “correctness”

The problem is too hard (search space too big)

What AlphaGo does



Choose a “valid” action



Iterate

State

New State

Get a Reward
(Win / Loss / Don’t know)

Markov Decision Process (MDP)

Vemal: Learnt Vectorization

Mendis et. al "Compiler Auto-Vectorization with Imitation Learning." [NeurIPS'19]

```
a[1] = b[1] + c[1]
a[2] = b[2] + c[2]
a[3] = b[3] + c[3]
a[4] = b[4] + c[4]
a[5] = b[5] * c[5]
```

State

Choose a "valid" action

{a[1],a[2]}, {a[2],a[3]}, {a[3],a[4]}


```
a[1:2] = b[1:2] + c[1:2]
```


```
a[1:2] = b[1:2] + c[1:2]
a[3:4] = b[3:4] + c[3:4]
a[5]   = b[5]   * c[5]
```

New State

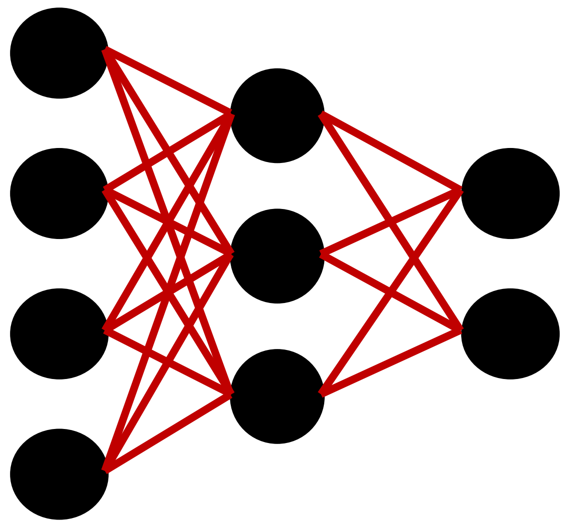
Iterate

Get a Reward
(Speed of execution)

How do you solve this MDP?

Vemal: Learnt Vectorization

Mendis et. al "Compiler Auto-Vectorization with Imitation Learning." [NeurIPS'19]



≈ Oracle goSLP

```
a[1] = b[1] + c[1]
a[2] = b[2] + c[2]
a[3] = b[3] + c[3]
a[4] = b[4] + c[4]
a[5] = b[5] * c[5]
```

Choose a "valid" action

{a[1],a[2]}, {a[2],a[3]}, {a[3],a[4]}

```
a[1:2] = b[1:2] + c[1:2]
a[3]   = b[3]   + c[3]
a[4]   = b[4]   + c[4]
a[5]   = b[5]   * c[5]
```

Iterate

State

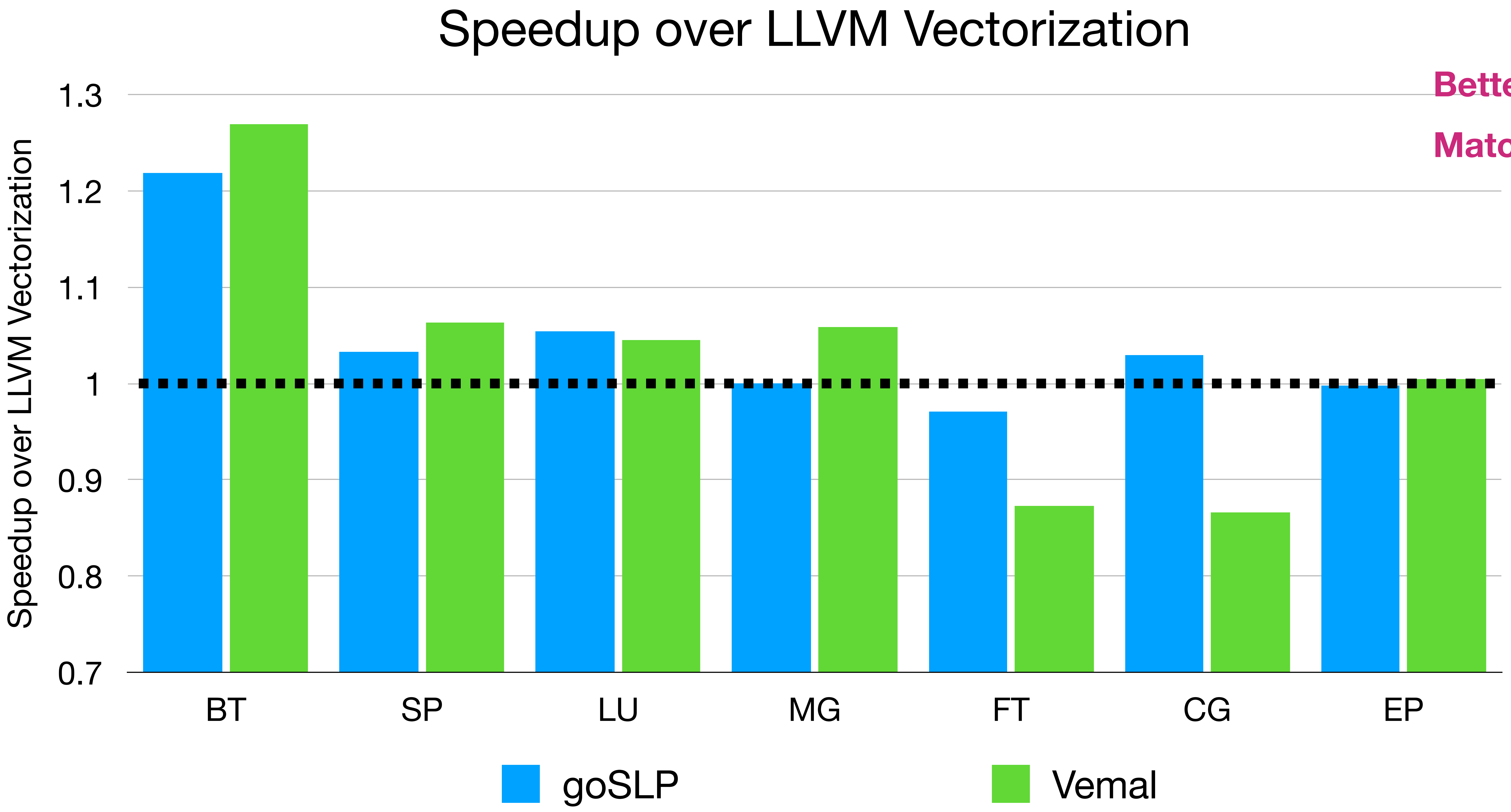
New State

Use Imitation Learning

The goal is to check learnability

Vemal: Learnt Vectorization

Mendis et. al "Compiler Auto-Vectorization with Imitation Learning." [NeurIPS'19]



Better than LLVM SLP

Matches or exceeds goSLP

Mendis's Model of Compiler Optimization

Vectorization	Transformation Space	Optimization Strategy	Cost Model
Manual solutions	Hand-written	Heuristics	Linear
Automated solutions State-of-the-art results Minimal development burden	**Code Generator Generator** Chen et. al [ASPLOS'21] **Program Logics**	**Data-driven** Mendis et. al [NeurIPS'19] **Machine Learning**	**Data-driven** Mendis et. al [ICML'19] Chen et. al [IISWC'19] **Machine Learning**

Accurate modeling of a processor is hard

≈

Analytical Model

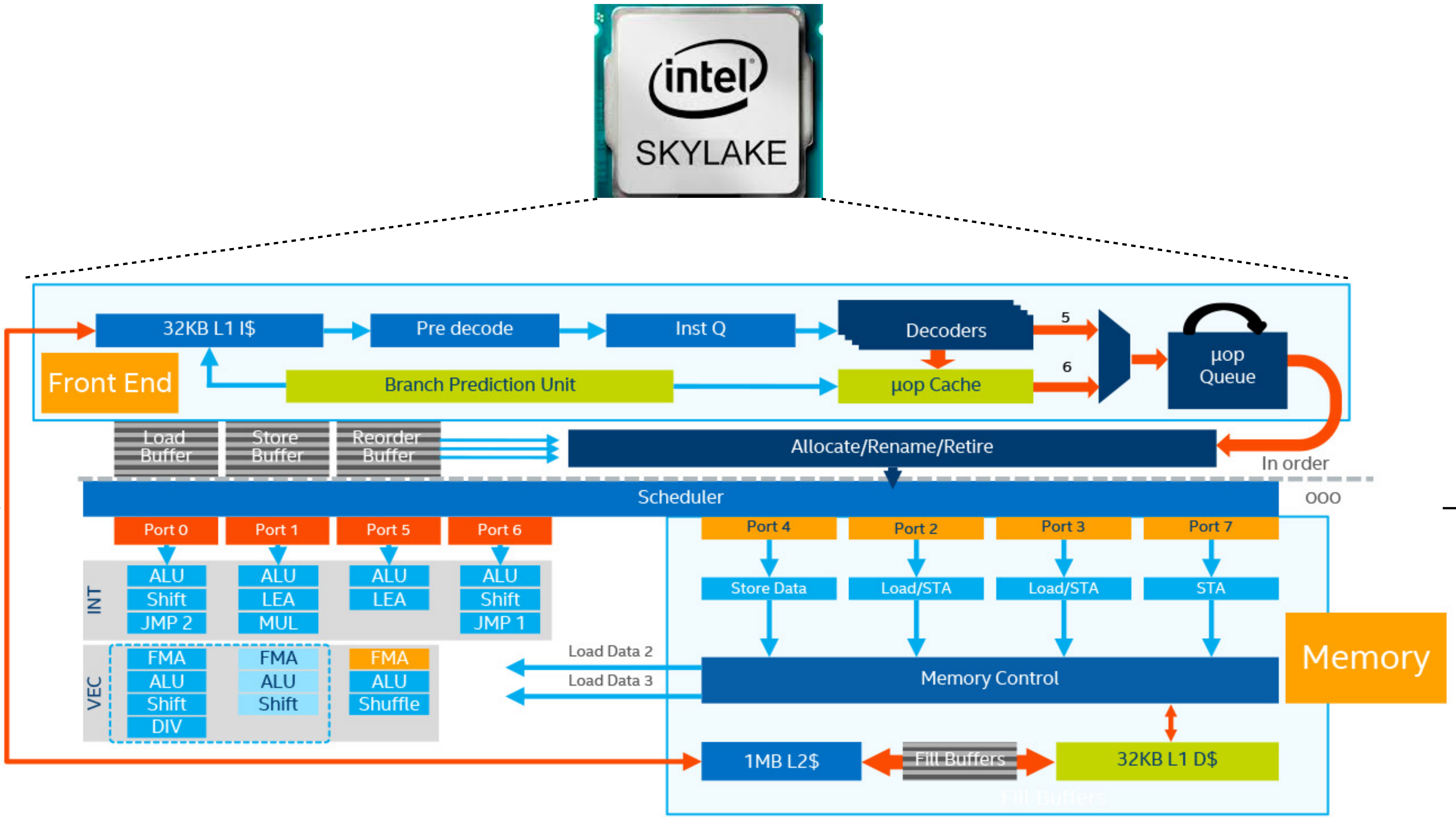
llvm-mca

IACA

~20% error

Accurate modeling of a processor is hard

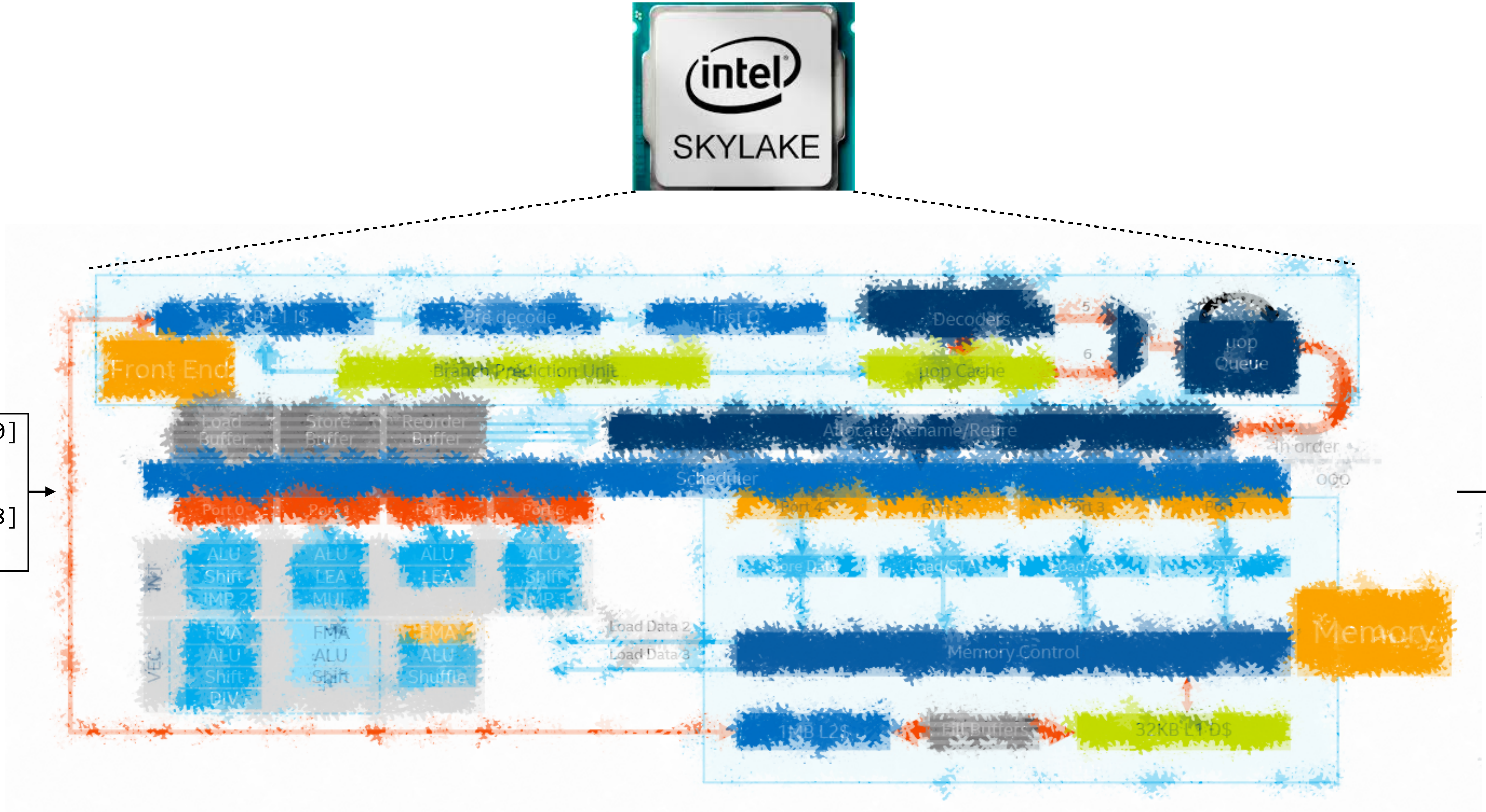
```
lea     r14, [rbx-0x40]
        ........
        ........
lea     rdx, [rbp+0x38]
cmp     rdi, rax
```



→ **44 cycles**

Accurate modeling of a processor is hard

```
lea     r14, [rbx-0x40]
        ........
        ........
lea     rdx, [rbp+0x38]
cmp     rdi, rax
```



→ **44 cycles**

Some details are proprietary

Accurate modeling of a processor is hard

2242 pages

intel

Intel® 64 and IA-32 Architectures
Software Developer's Manual

Volume 2 (2A, 2B, 2C & 2D):
Instruction Set Reference, A-Z

NOTE: The Intel 64 and IA-32 Architectures Software Developer's Manual consists of four volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-Z*, Order Number 325383; *System Programming Guide*, Order Number 325384; *Model-Specific Registers*, Order Number 335592. Refer to all four volumes when evaluating your design needs.

Order Number: 325383-070US
May 2019

672 pages

intel

Intel® 64 and IA-32 Architectures
Optimization Reference Manual

Order Number: 248966-033
June 2016

LLVM llvm-mca

~2000 lines

```
// BMI1 BEXTR/BLS, BMI2 BZHI
defm : HWWriteResPair<WriteBEXTR, [HWPort06,HWPort15], 2, [1,1], 2>;
defm : HWWriteResPair<WriteBLS,   [HWPort15], 1>;
defm : HWWriteResPair<WriteBZHI,  [HWPort15], 1>;

// TODO: Why isn't the HWDivider used?
defm : X86WriteRes<WriteDiv8,     [HWPort0,HWPort1,HWPort5,HWPort6], 22, [], 9>;
defm : X86WriteRes<WriteDiv16,    [HWPort0,HWPort1,HWPort5,HWPort6,HWPort01,HWPort0156], 98, [7,7,3,3,1,11], 32>;
defm : X86WriteRes<WriteDiv32,    [HWPort0,HWPort1,HWPort5,HWPort6,HWPort01,HWPort0156], 98, [7,7,3,3,1,11], 32>;
defm : X86WriteRes<WriteDiv64,    [HWPort0,HWPort1,HWPort5,HWPort6,HWPort01,HWPort0156], 98, [7,7,3,3,1,11], 32>;
defm : X86WriteRes<WriteDiv8Ld,   [HWPort0,HWPort23,HWDivider], 29, [1,1,10], 2>;
defm : X86WriteRes<WriteDiv16Ld,  [HWPort0,HWPort23,HWDivider], 29, [1,1,10], 2>;
defm : X86WriteRes<WriteDiv32Ld,  [HWPort0,HWPort23,HWDivider], 29, [1,1,10], 2>;
defm : X86WriteRes<WriteDiv64Ld,  [HWPort0,HWPort23,HWDivider], 29, [1,1,10], 2>;

defm : X86WriteRes<WriteIDiv8,    [HWPort0,HWPort1,HWPort5,HWPort6], 23, [], 9>;
defm : X86WriteRes<WriteIDiv16,   [HWPort0,HWPort1,HWPort5,HWPort6,HWPort06,HWPort0156], 112, [4,2,4,8,14,34], 66>;
defm : X86WriteRes<WriteIDiv32,   [HWPort0,HWPort1,HWPort5,HWPort6,HWPort06,HWPort0156], 112, [4,2,4,8,14,34], 66>;
defm : X86WriteRes<WriteIDiv64,   [HWPort0,HWPort1,HWPort5,HWPort6,HWPort06,HWPort0156], 112, [4,2,4,8,14,34], 66>;
defm : X86WriteRes<WriteIDiv8Ld,  [HWPort0,HWPort23,HWDivider], 29, [1,1,10], 2>;
defm : X86WriteRes<WriteIDiv16Ld, [HWPort0,HWPort23,HWDivider], 29, [1,1,10], 2>;
defm : X86WriteRes<WriteIDiv32Ld, [HWPort0,HWPort23,HWDivider], 29, [1,1,10], 2>;
defm : X86WriteRes<WriteIDiv64Ld, [HWPort0,HWPort23,HWDivider], 29, [1,1,10], 2>;

// Scalar and vector floating point.
defm : X86WriteRes<WriteFLD0,          [HWPort01], 1, [1], 1>;
defm : X86WriteRes<WriteFLD1,          [HWPort01], 1, [2], 2>;
defm : X86WriteRes<WriteFLDC,          [HWPort01], 1, [2], 2>;
defm : X86WriteRes<WriteFLoad,         [HWPort23], 5, [1], 1>;
defm : X86WriteRes<WriteFLoadX,        [HWPort23], 6, [1], 1>;
defm : X86WriteRes<WriteFLoadY,        [HWPort23], 7, [1], 1>;
defm : X86WriteRes<WriteFMaskedLoad,   [HWPort23,HWPort5], 8, [1,2], 3>;
defm : X86WriteRes<WriteFMaskedLoadY,  [HWPort23,HWPort5], 9, [1,2], 3>;
defm : X86WriteRes<WriteFStore,        [HWPort237,HWPort4], 1, [1,1], 2>;
defm : X86WriteRes<WriteFStoreX,       [HWPort237,HWPort4], 1, [1,1], 2>;
defm : X86WriteRes<WriteFStoreY,       [HWPort237,HWPort4], 1, [1,1], 2>;
```

```
defm : HWWriteResPair<WriteVarShuffleY,[HWPort5], 1, [1], 1, 7>;
defm : HWWriteResPair<WriteVarShuffleZ,[HWPort5], 1, [1], 1, 7>; // Unsupported = 1
defm : HWWriteResPair<WriteBlend,  [HWPort5], 1, [1], 1, 6>;
defm : HWWriteResPair<WriteBlendY, [HWPort5], 1, [1], 1, 7>;
defm : HWWriteResPair<WriteBlendZ, [HWPort5], 1, [1], 1, 7>; // Unsupported = 1
defm : HWWriteResPair<WriteShuffle256, [HWPort5], 3, [1], 1, 7>;
defm : HWWriteResPair<WriteVarShuffle256, [HWPort5], 3, [1], 1, 7>;
defm : HWWriteResPair<WriteVarBlend,  [HWPort5], 2, [2], 2, 6>;
defm : HWWriteResPair<WriteVarBlendY, [HWPort5], 2, [2], 2, 7>;
defm : HWWriteResPair<WriteVarBlendZ, [HWPort5], 2, [2], 2, 7>; // Unsupported = 1
defm : HWWriteResPair<WriteMPSAD,  [HWPort0, HWPort5], 7, [1, 2], 3, 6>;
defm : HWWriteResPair<WriteMPSADY, [HWPort0, HWPort5], 7, [1, 2], 3, 7>;
defm : HWWriteResPair<WriteMPSADZ, [HWPort0, HWPort5], 7, [1, 2], 3, 7>; // Unsupported = 1
defm : HWWriteResPair<WritePSADBW,  [HWPort0], 5, [1], 1, 5>;
defm : HWWriteResPair<WritePSADBWX, [HWPort0], 5, [1], 1, 6>;
defm : HWWriteResPair<WritePSADBWY, [HWPort0], 5, [1], 1, 7>;
defm : HWWriteResPair<WritePSADBWZ, [HWPort0], 5, [1], 1, 7>; // Unsupported = 1
defm : HWWriteResPair<WritePHMINPOS, [HWPort0],  5, [1], 1, 6>;
```

Accurate modeling of a processor is hard

2242 pages

672 pages

LLVM llvm-mca

~2000 lines

- ❌ tedious to develop and maintain
- ❌ human-error
- ❌ modeling-error

LLVM's Intel Performance Models are maintained by Sony with AMD hardware and are not validated

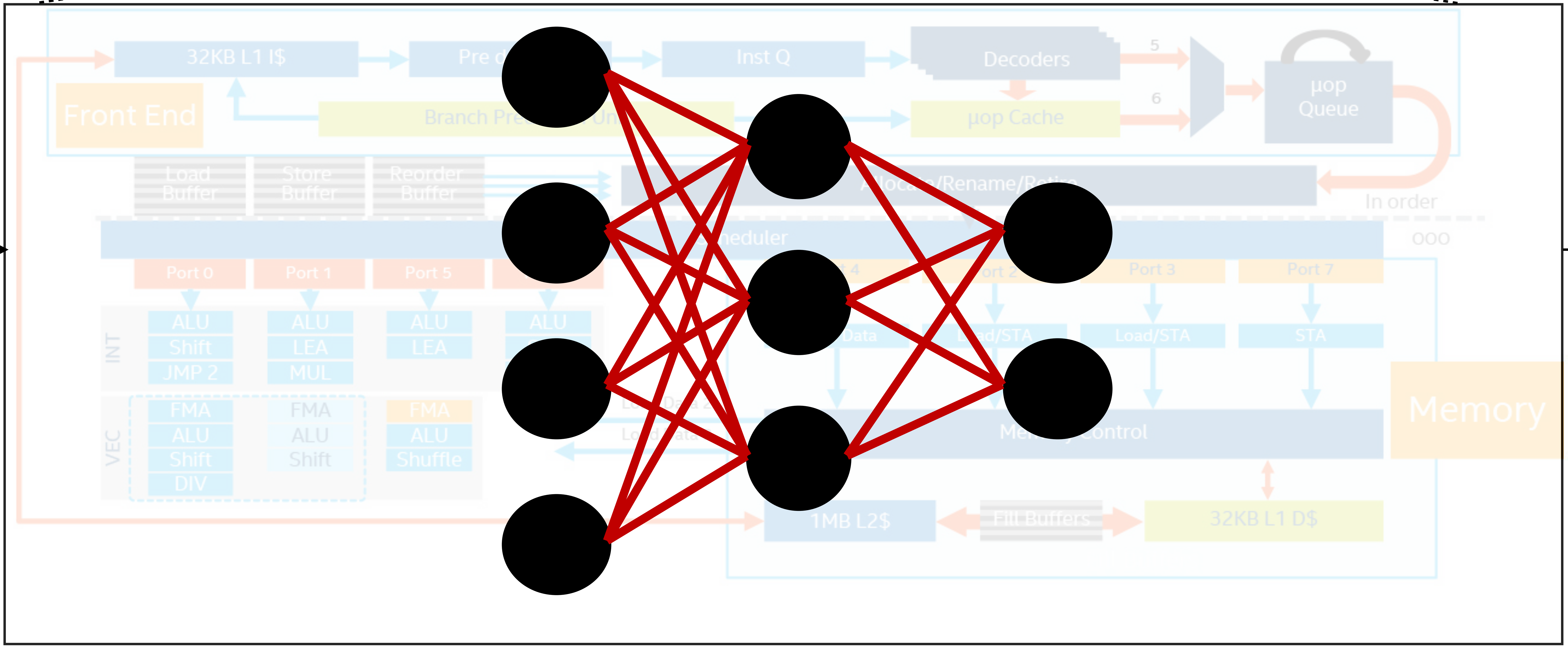
Learnt Cost Model - Ithemal

Mendis et. al [ICML'19]

✔ Fast
✔ Accurate

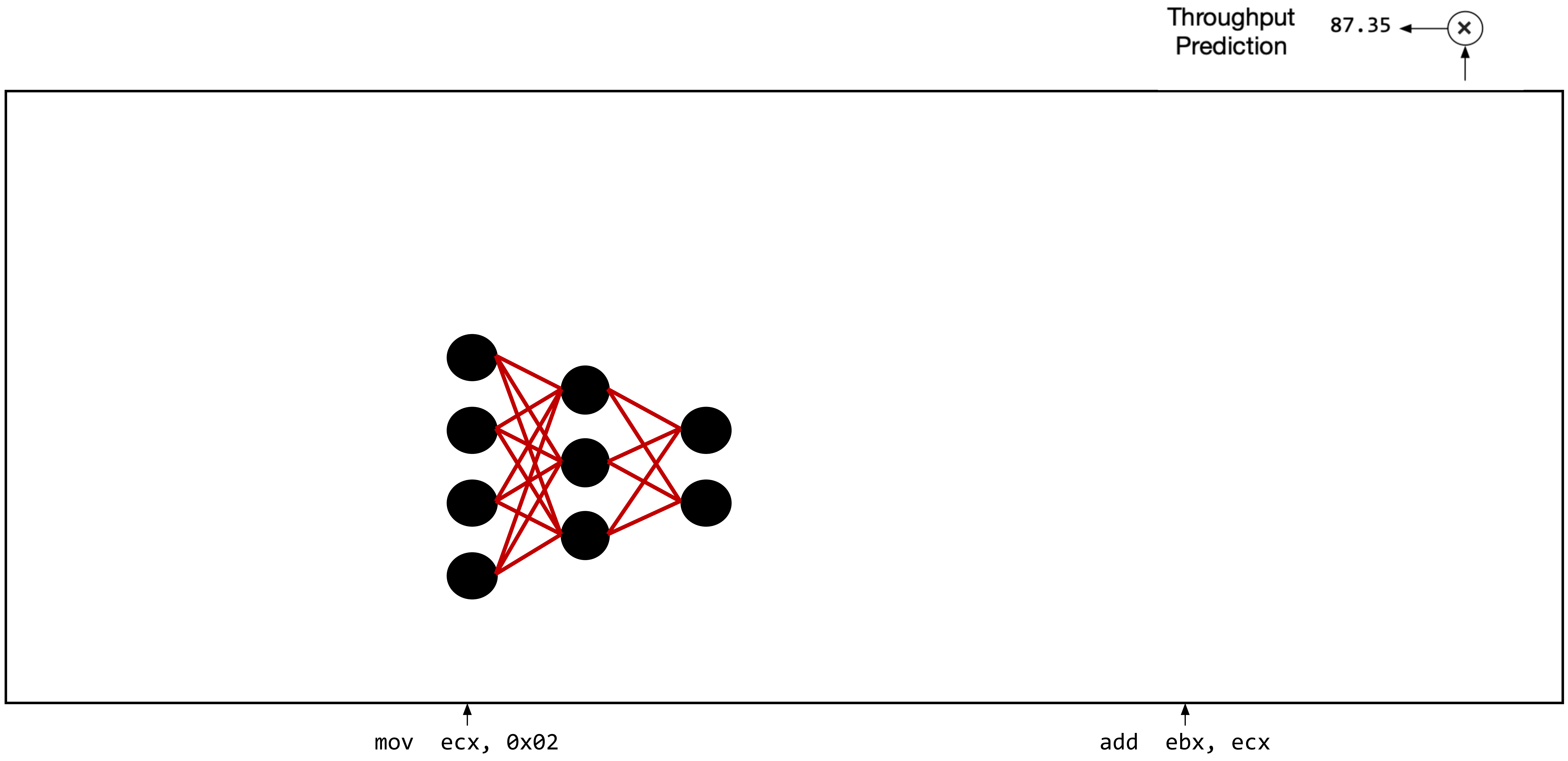
Ithemal

```
lea     r14, [rbx-0x40]
        ........
        ........
lea     rdx, [rbp+0x38]
cmp     rdi, rax
```



44 cycles

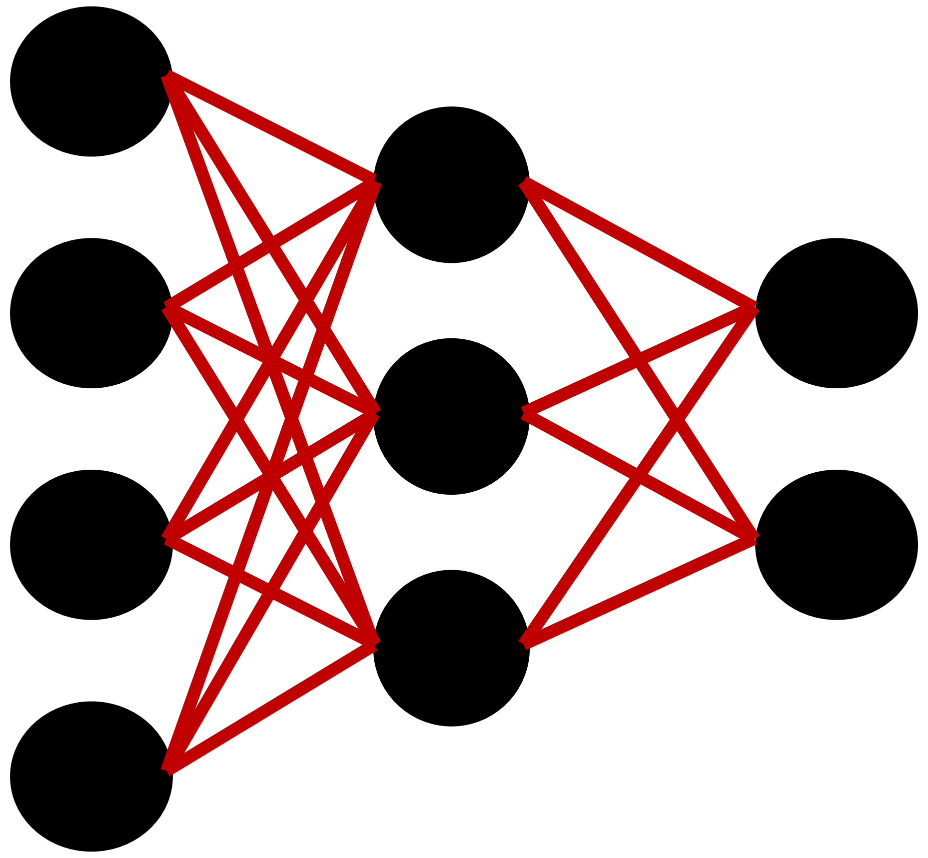
Neural Network Architecture



Neural Network Architecture

Throughput Prediction

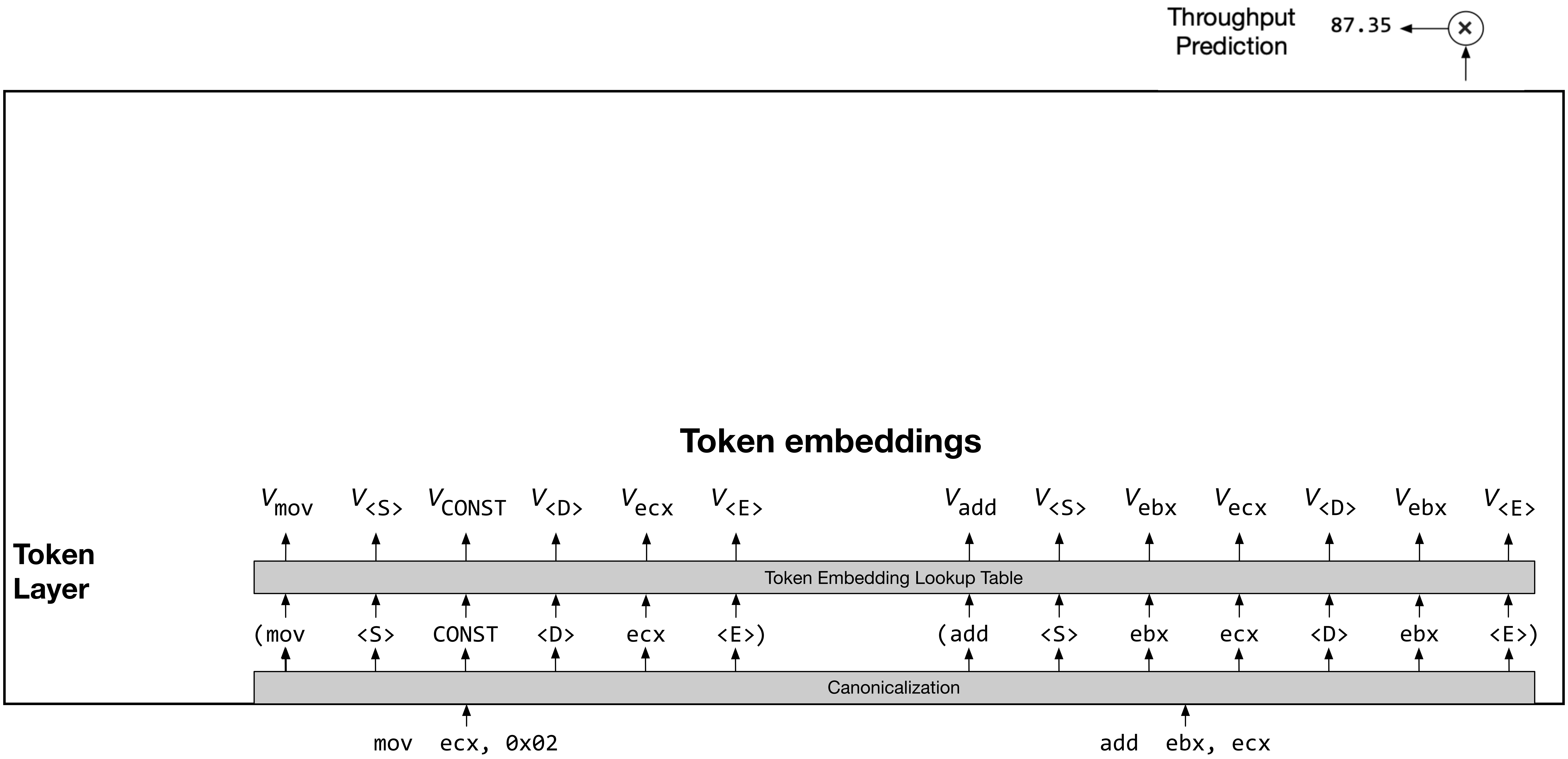
87.35



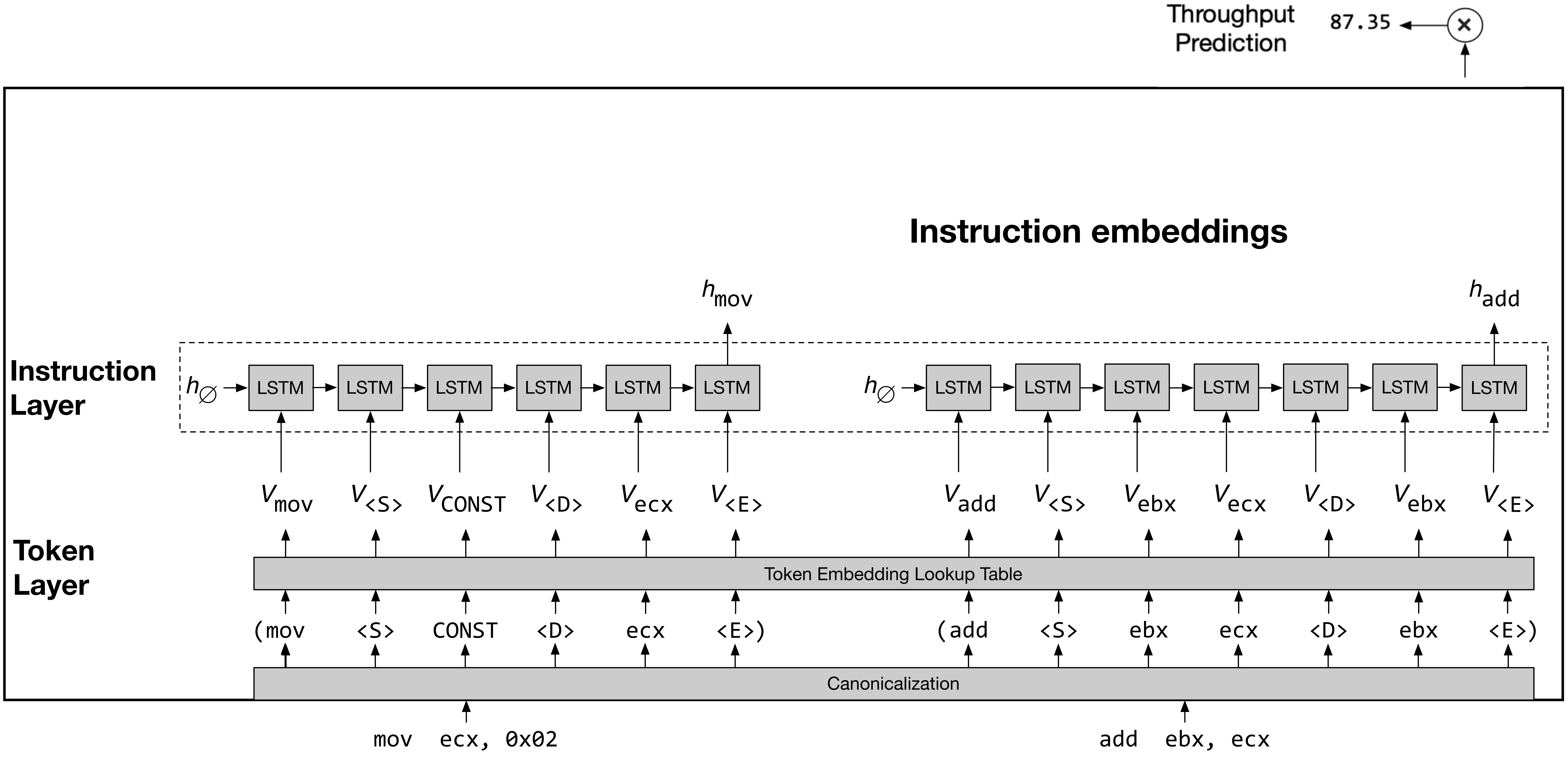
No featurization required

```
mov   ecx, 0x02
add   ebx, ecx
```

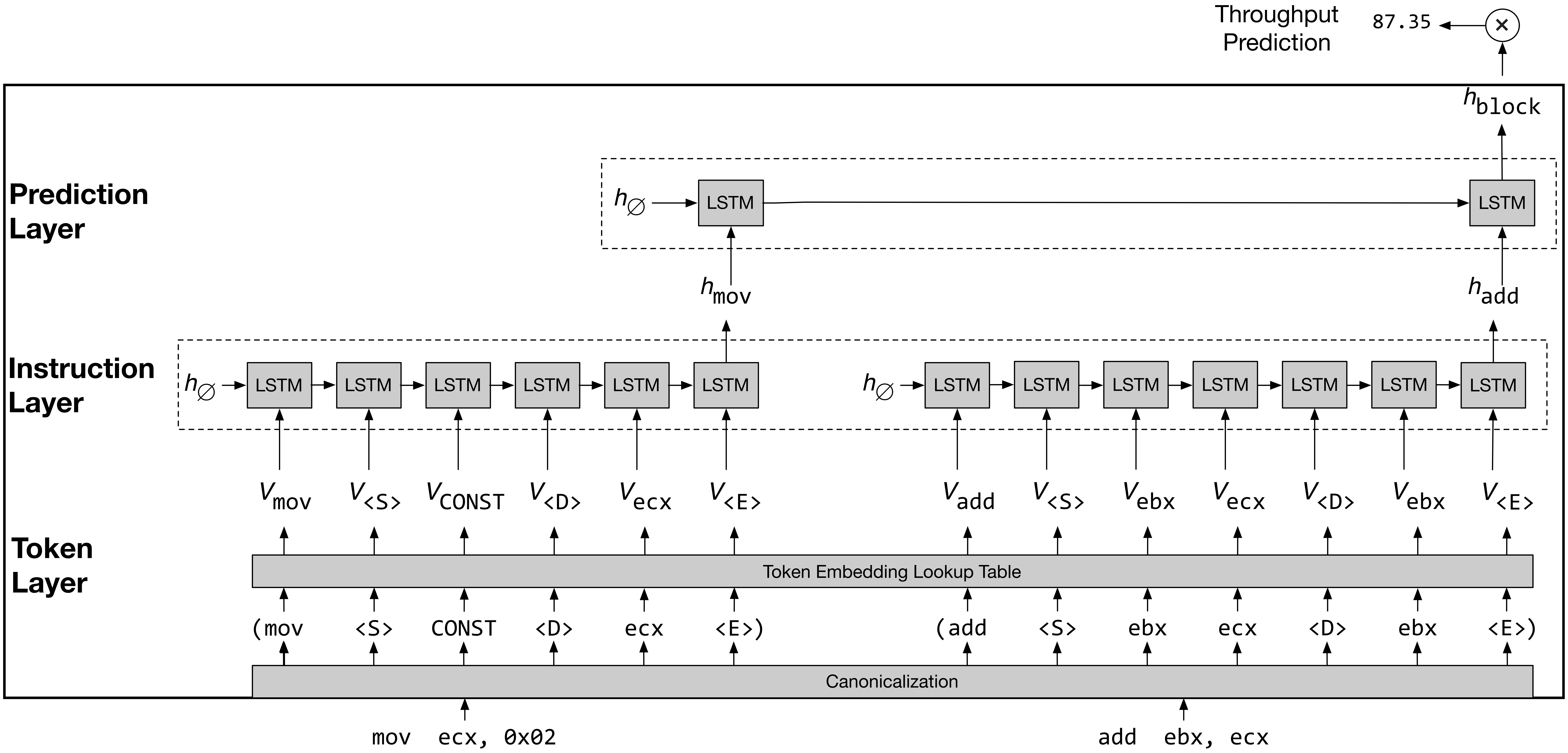
Neural Network Architecture



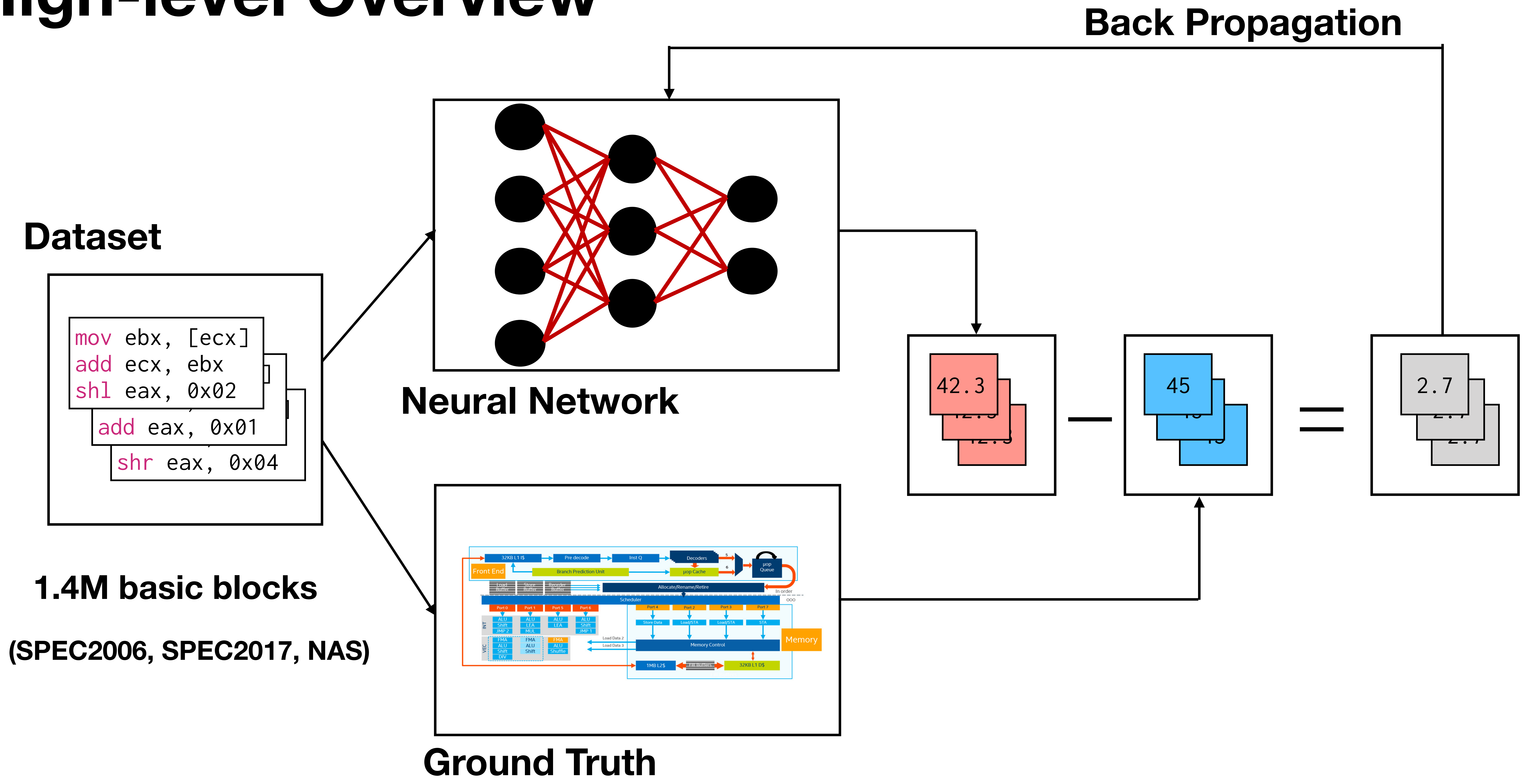
Neural Network Architecture



Neural Network Architecture



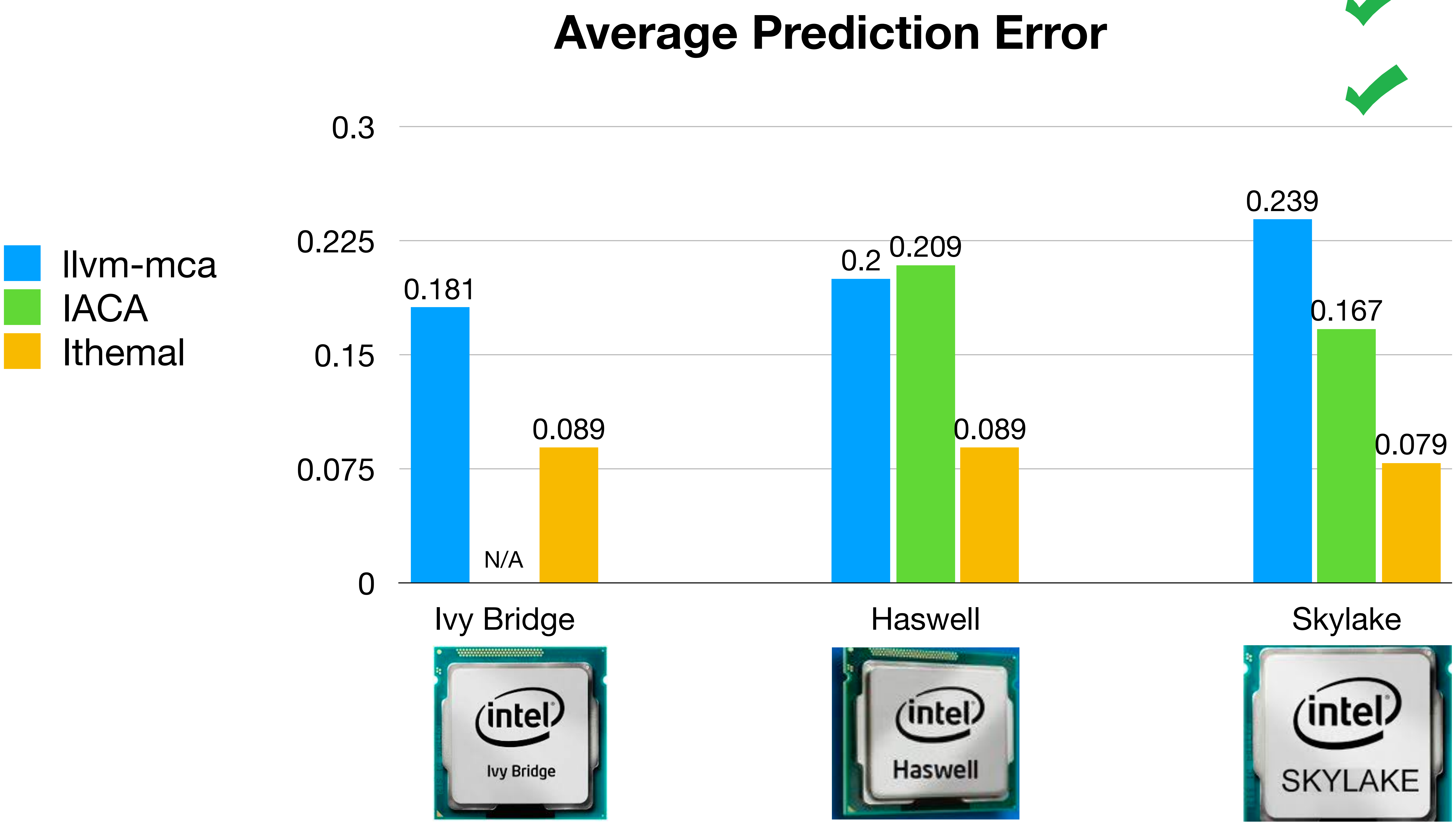
High-level Overview



Ithemal halves the error rate across multiple microarchitectures

Accurate

Portable

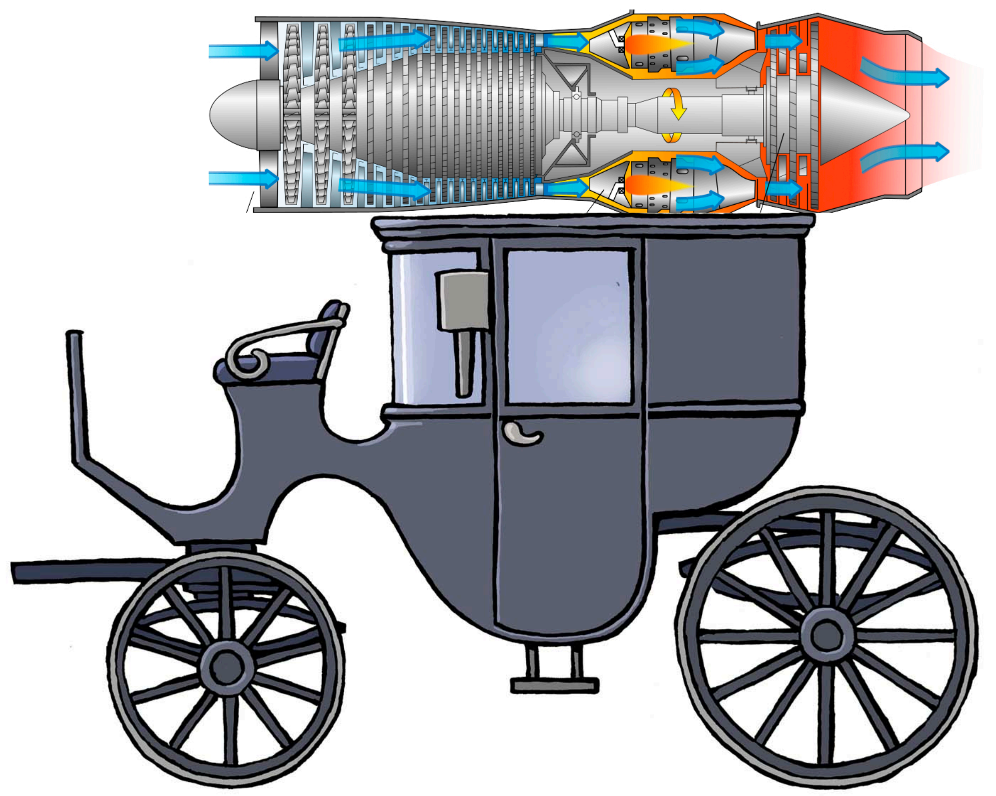


Compiler 2.0

- Build Compilers as a Service
- Automate Compiler Construction
- Use Machine Learning

Why new compiler infrastructure?

- We want to spend time doing cool research

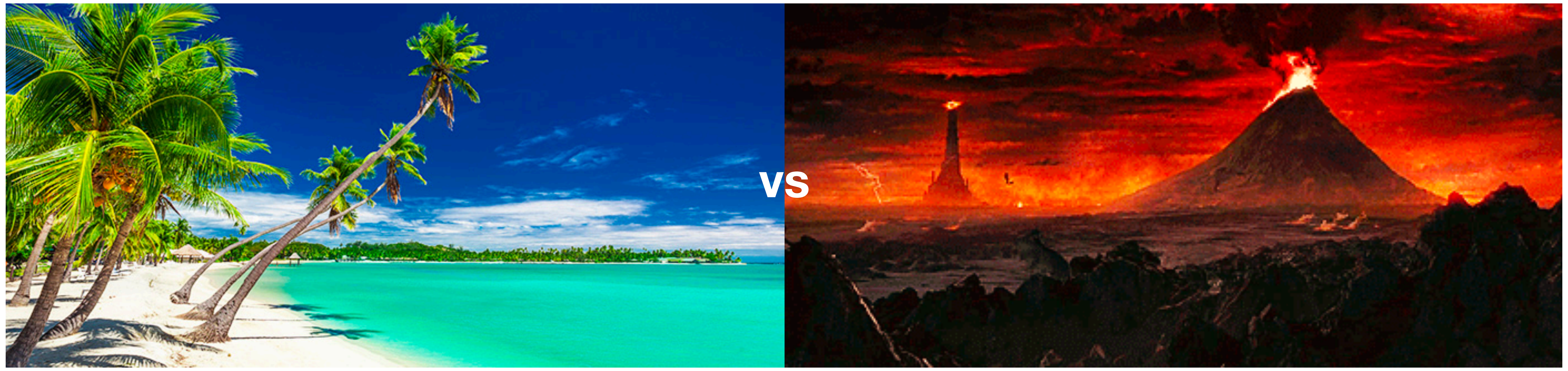


- Current infrastructure is too old to take the research to production
  - Many papers written, never used in practice :(
  - *“Prototypes are easy, production is hard”* -Elon Musk
- Researchers should put energy on building new production-quality compiler infrastructure using new technology

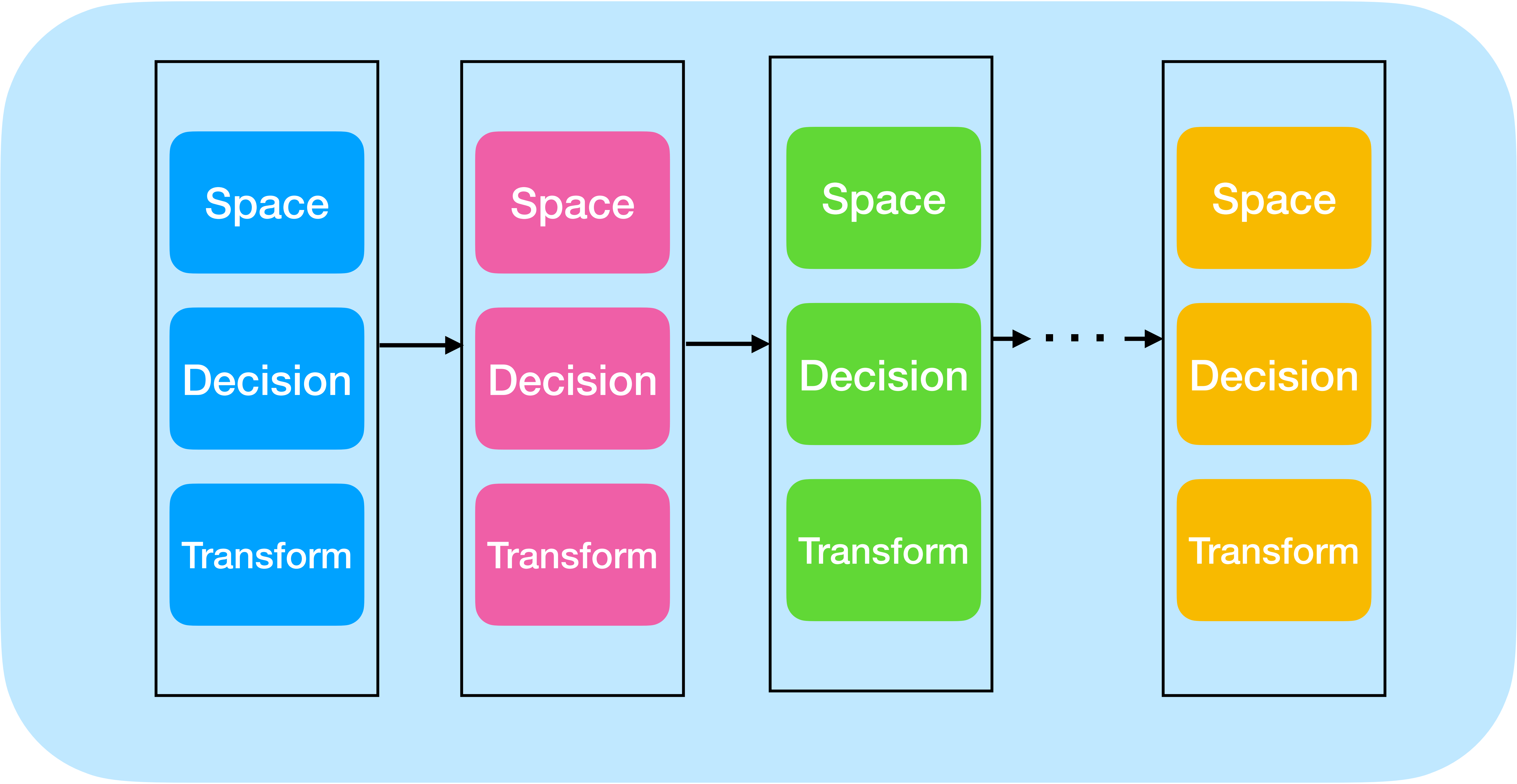
Can we Learn the Full Compiler Optimization Stack?

- Learn to search the space of optimizations

- Maintain program correctness
  Or dip into a scary place and recover
  (with verification)?



Break pass abstractions

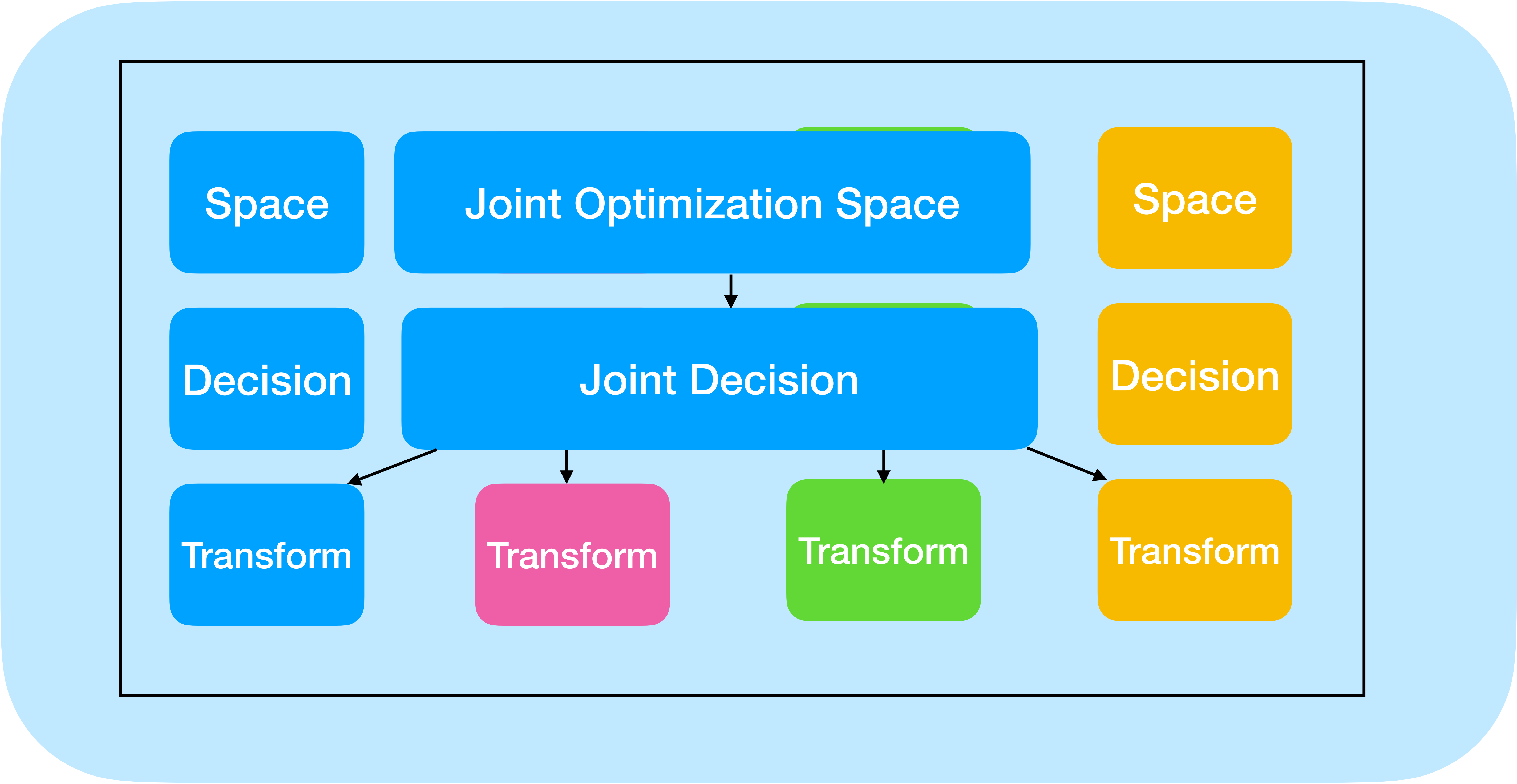


Transformation passes

Monolithic way to write optimization passes

Hypothesis : Better Optimization Decisions ❓

Break pass abstractions



Transformation passes

Monolithic way to write optimization passes

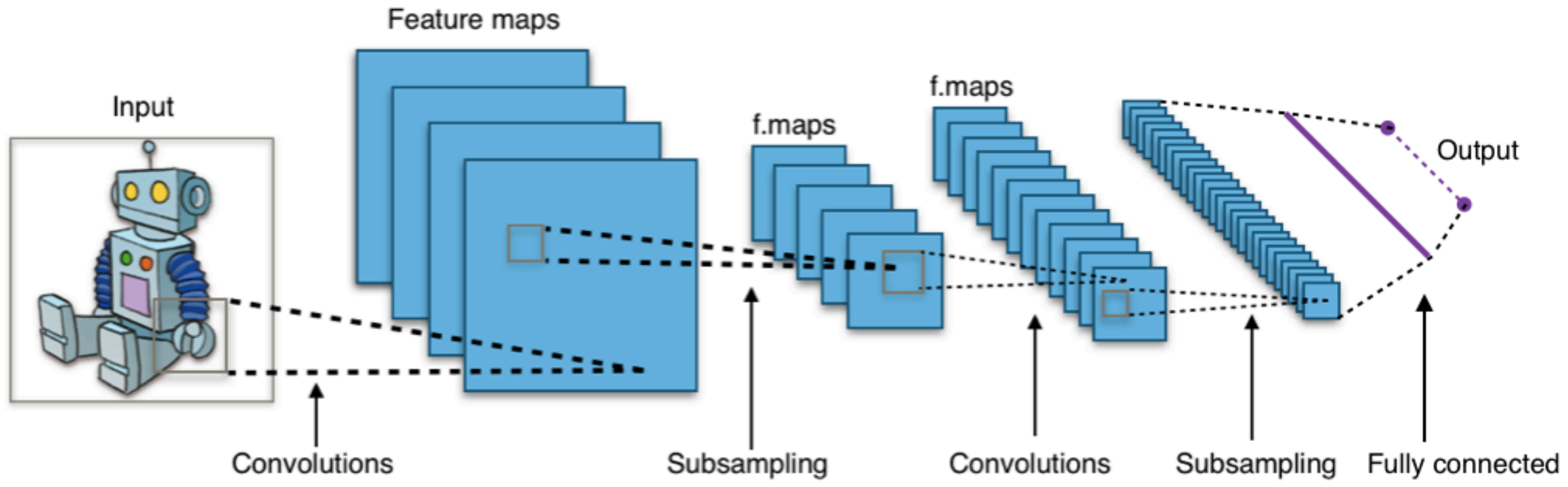
Joint Optimization Space

Joint Decision

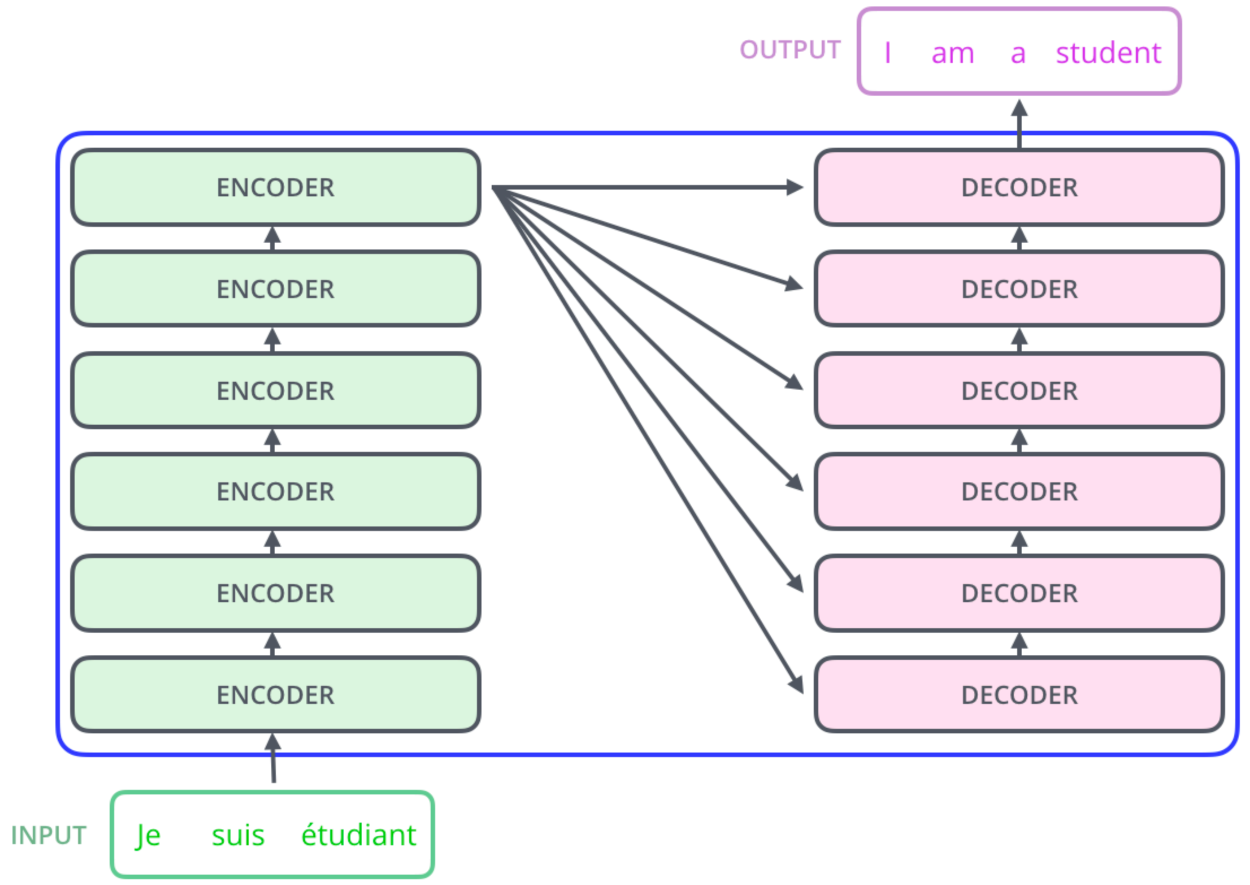
Delegate transformations

Hypothesis : Better Optimization Decisions ❓

Representation Matters

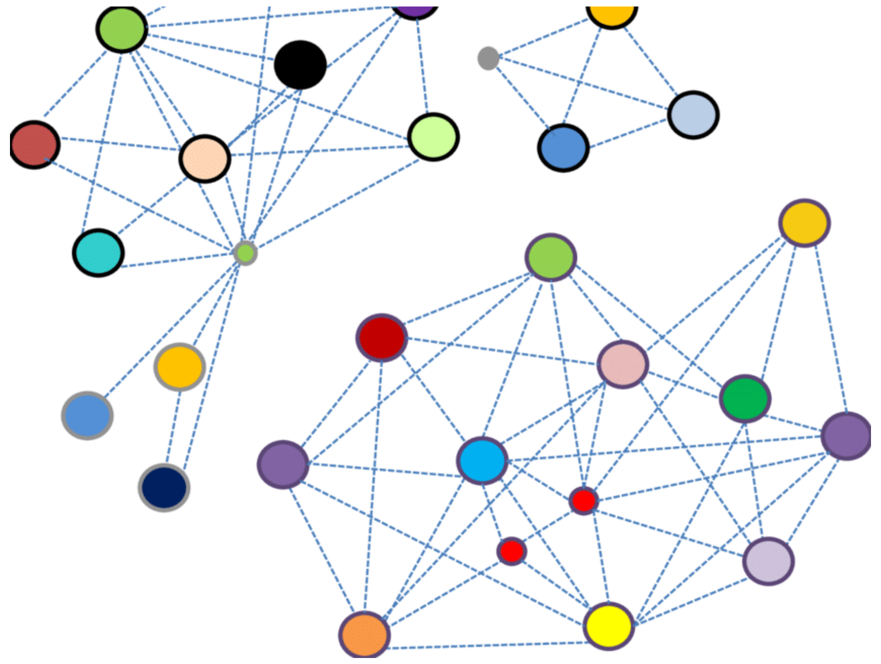


Images ➡ **Convolutional** Neural Networks **(CNN)**



Text ➡ **Recurrent** Neural Networks **(RNN)**

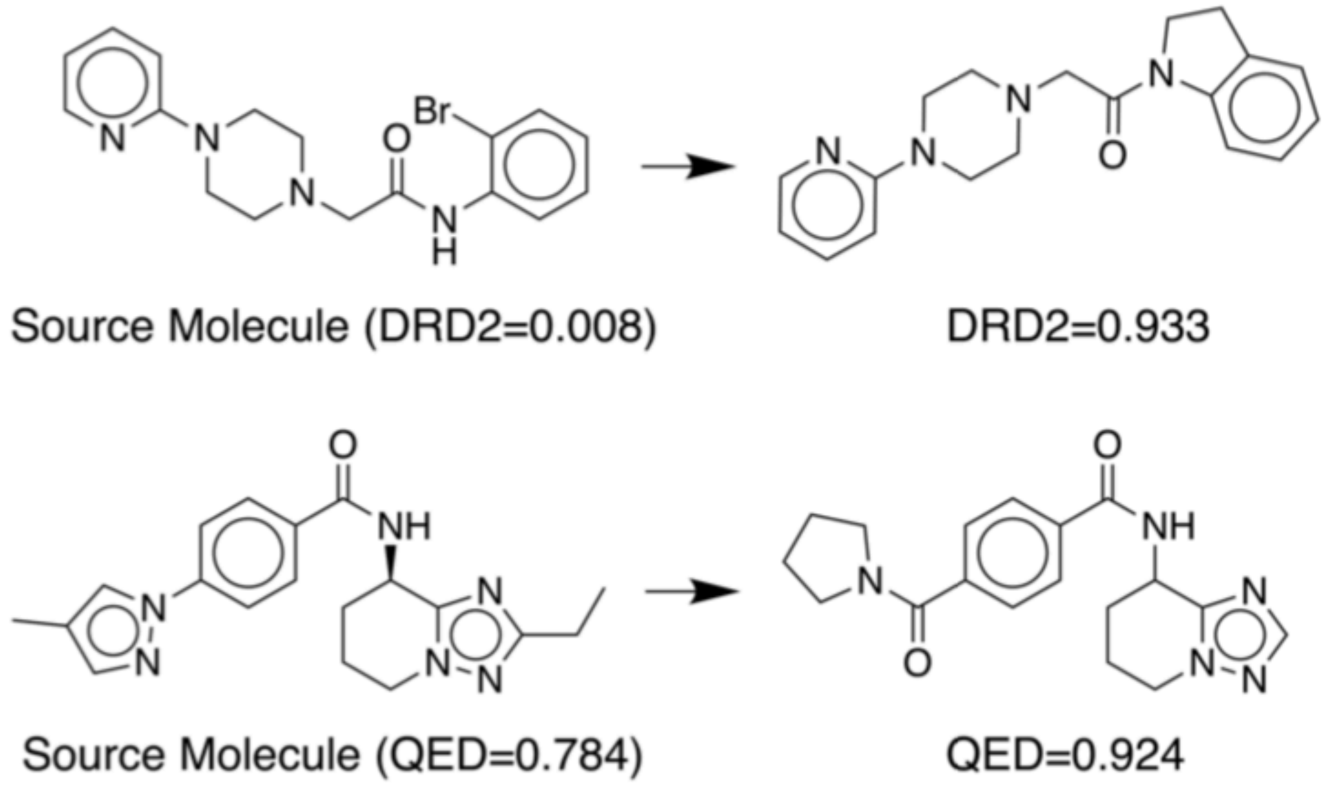
Transformers



Social Networks ➡ **Graph** Convolutional Networks **(GCN)**

Gated **Graph** Neural Networks **(GGNN)**

Graph Transformer Networks **(GTN)**

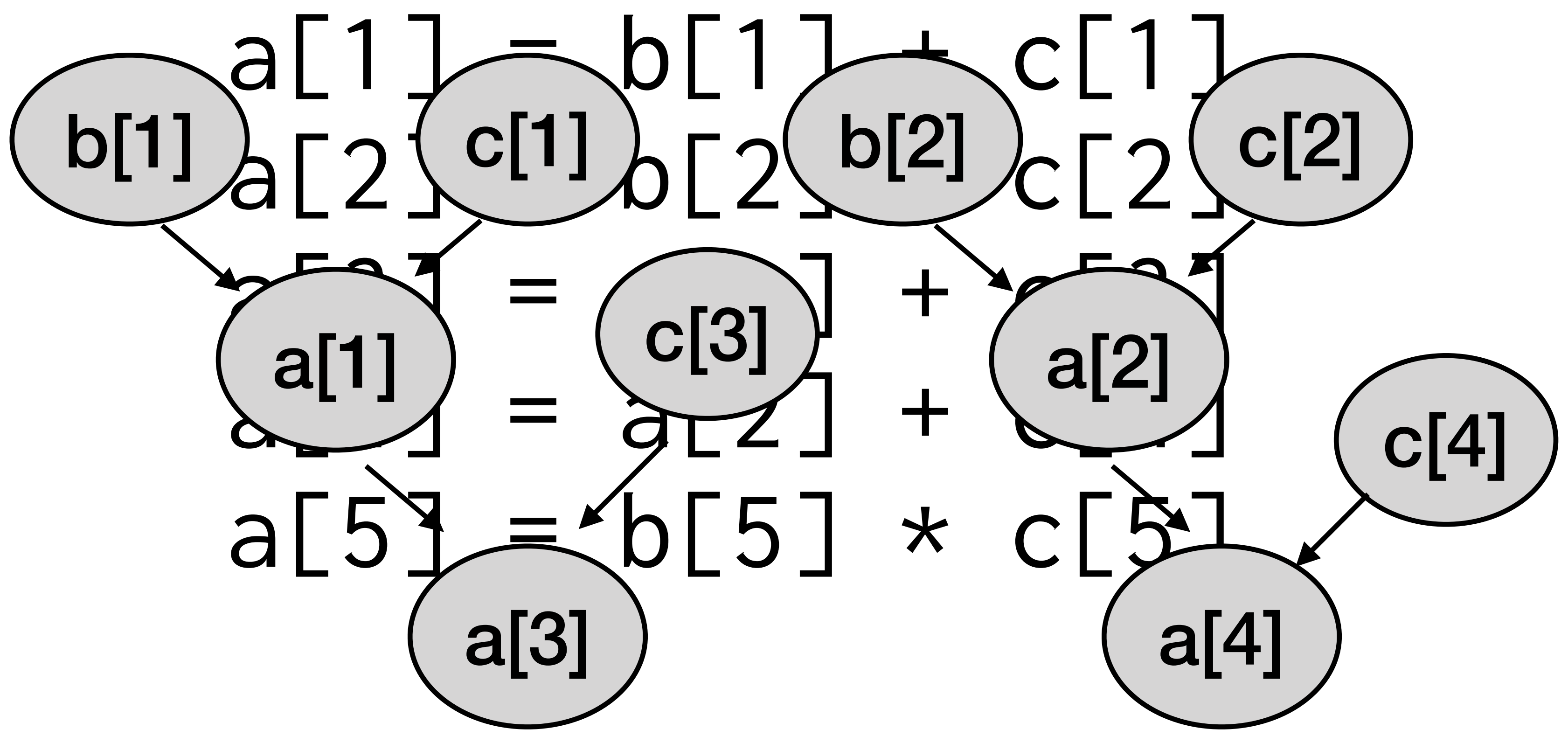


Molecules ➡ **Graph Encoder-Decoder**

Path-augmented

Graph Transformer Networks

Program Representation



Programs → **Graph Neural Networks** ?

Global Properties ?

Semantic Properties ?

Language-agnostic ?

Automatic Program Optimization

Bug finding

Automatic Patch Generation

Program Synthesis (automatic programming)

Precise program analysis

Dynamic Property Prediction

......

......

What is the best representation for programs?

Reasoning in continuous space

Thank You

- Current and recent projects in the Commit Compiler Group
  - TACO: A DSL for sparse tensor algebra
  - GraphIt: A DSL for graph analysts
  - Halide: A DSL dense array programming
  - SEQ: A DSL for bio informatics
  - BuildIt: A Multistage programming framework in C++
  - CoLa: A DSL for data compression
  - SimIt: A DSL for sparse systems
  - MILK: A DSL for Optimizing indirect memory references
  - Cimple: A DSL for Instruction and Memory Level Parallelism
  - Tiramisu: A polyhedral compiler for data parallel algorithms
  - Ithemal: Performance prediction using machine learning
  - Vemal: Vectorization using machine learning
  - goSLP & Revec: Modernizing vectorization technology
  - OpenTuner: An extensible framework for program autotuning

http://groups.csail.mit.edu/commit/

Commit Group is Supported By: