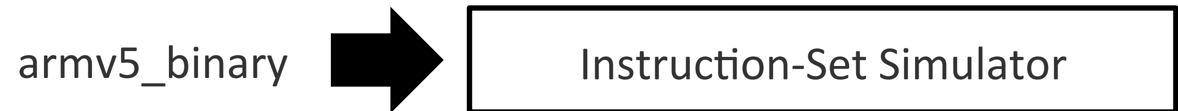
Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers

Derek Lockhart, Berkin Ilbeyi, and Christopher Batten

Cornell University
Computer Systems Laboratory

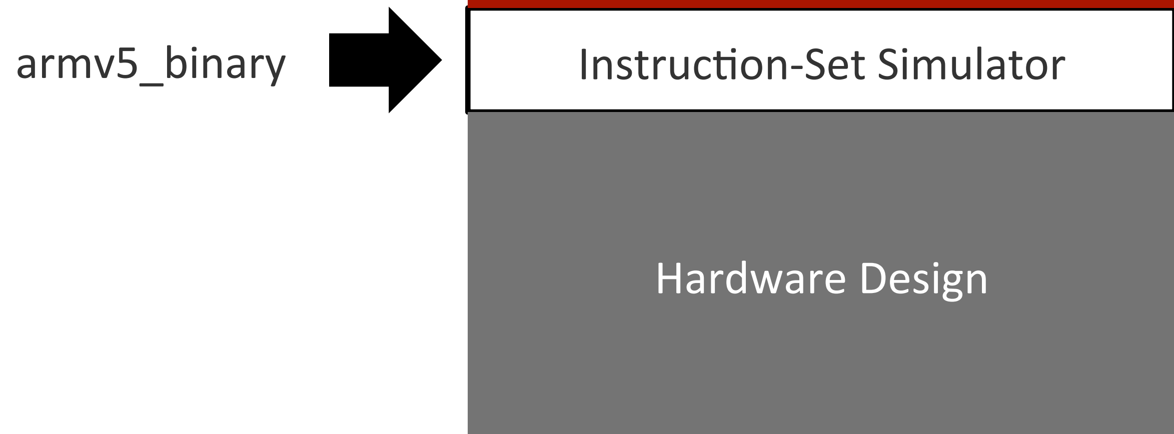
Motivation

Instruction set simulators perform **functional** simulation of a target architecture.



Motivation

Instruction set simulators perform **functional** simulation of a target architecture.

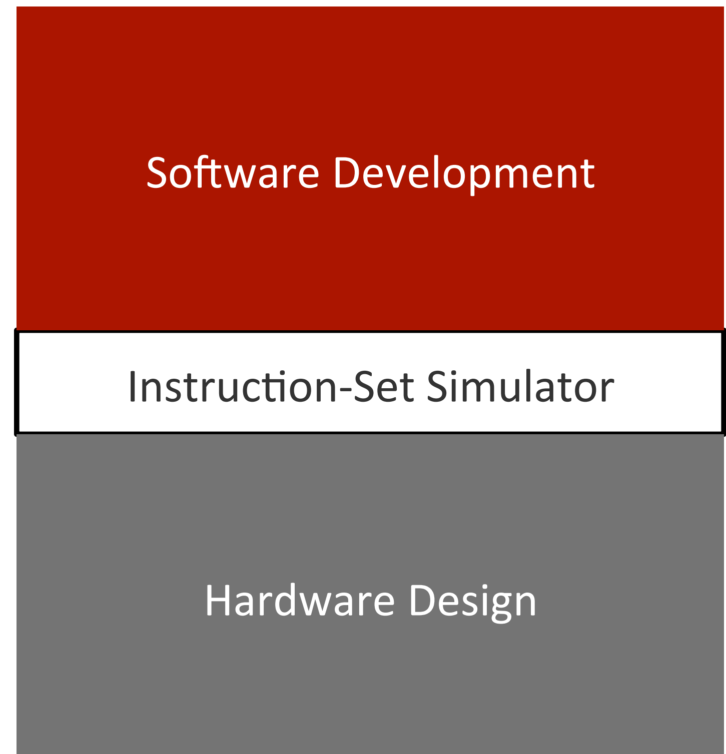


Motivation

Instruction set simulators perform **functional** simulation of a target architecture.

Instruction-Set Simulator Goals:

- Accuracy
- Observability
- Performance

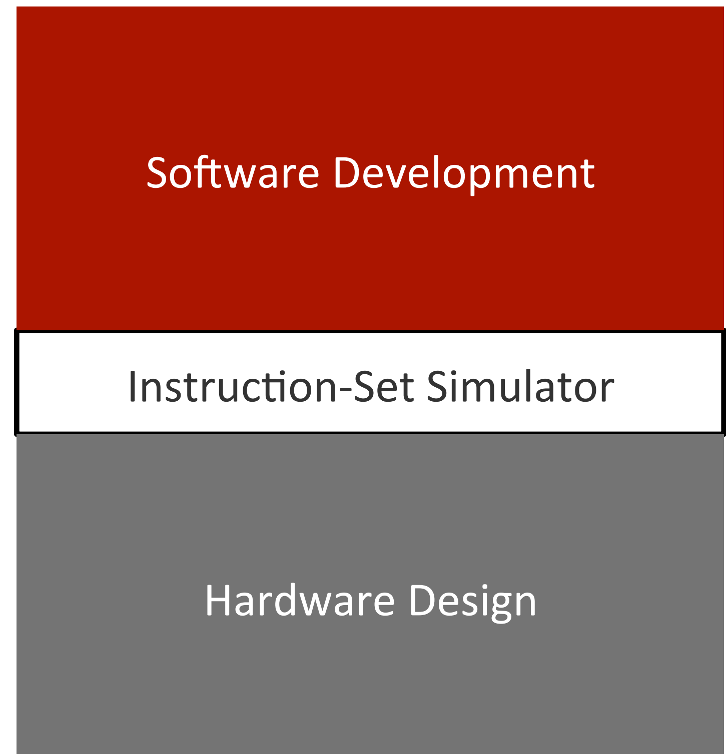


Motivation

Instruction set simulators perform **functional** simulation of a target architecture.

Instruction-Set Simulator Goals:

- Accuracy
- Observability
- Performance
- Productivity

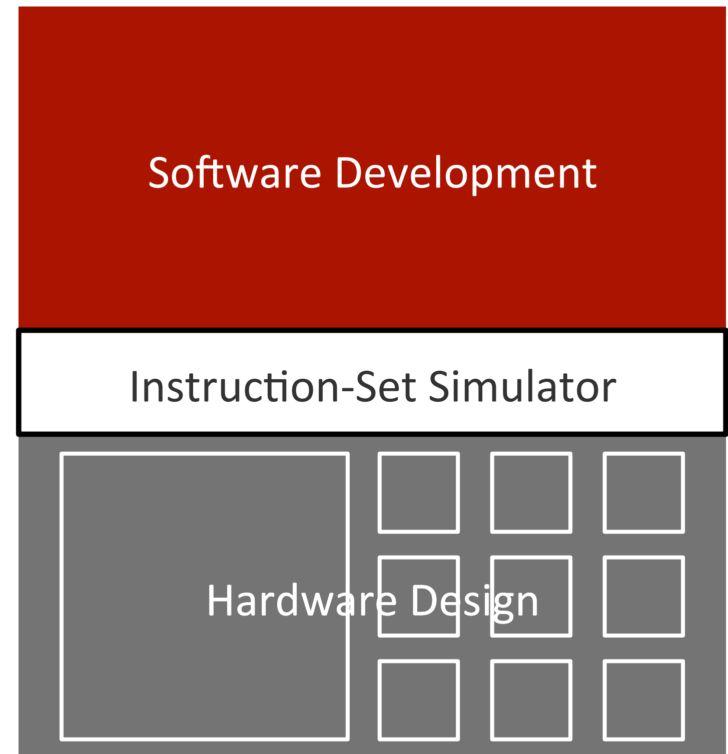


Motivation

Instruction set simulators perform **functional** simulation of a target architecture.

Instruction-Set Simulator Goals:

- Accuracy
- Observability
- Performance
- Productivity

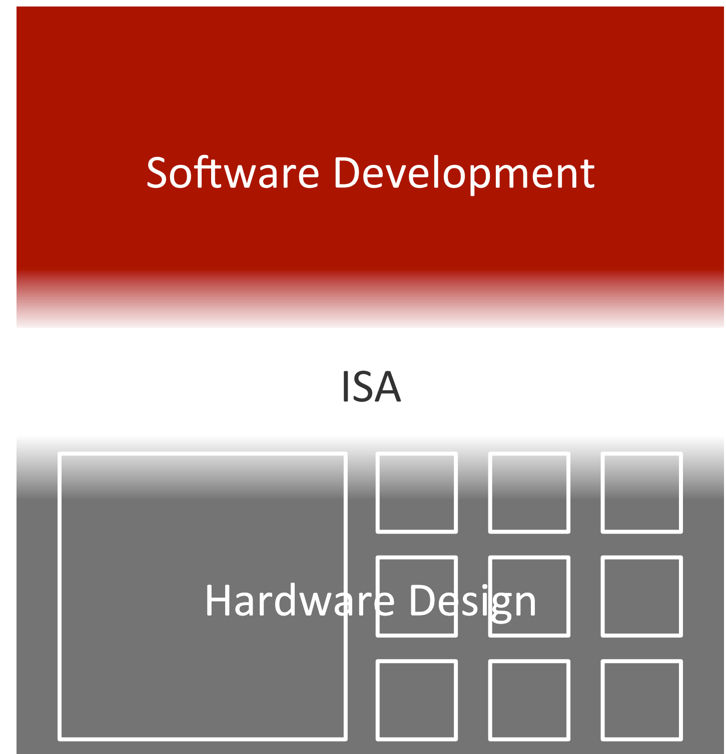


Motivation

Instruction set simulators perform **functional** simulation of a target architecture.

Instruction-Set Simulator Goals:

- Accuracy
- Observability
- Performance
- Productivity



Productivity Performance

Productivity Performance

Architectural Description Language

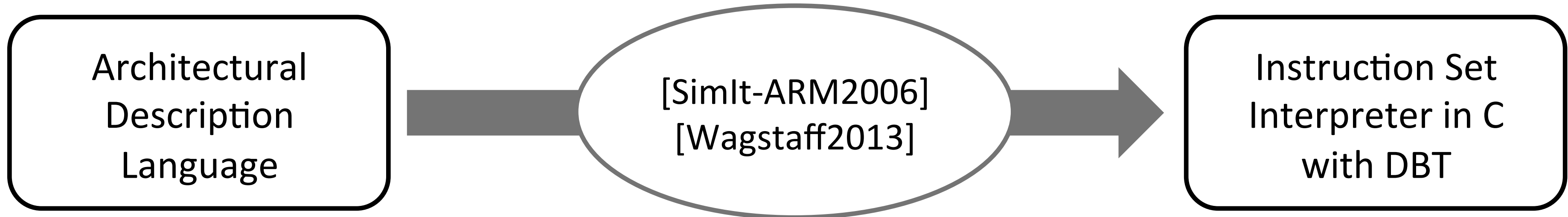
Instruction Set Interpreter in C with DBT

Productivity Performance

Architectural Description Language

Instruction Set Interpreter in C with DBT

Productivity Performance

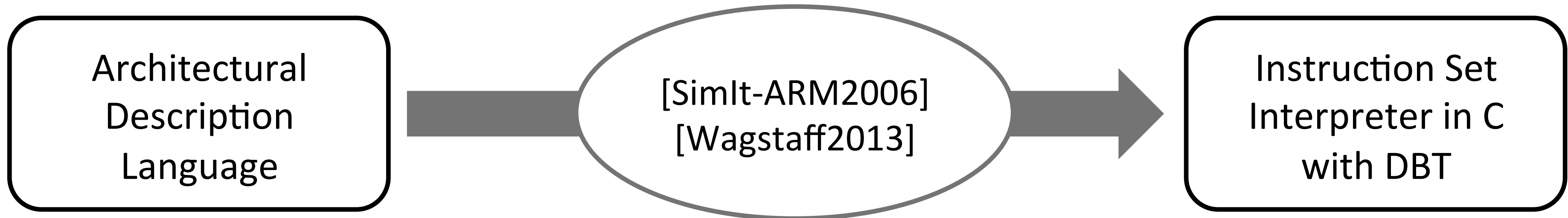


[Simit-ARM2006] J.D'Errico and W.Qin. Constructing Portable Compiled Instruction-Set Simulators — An ADL-Driven Approach. DATE'06.

[Wagstaff2013] H. Wagstaff, M. Gould, B. Franke, and N.Topham. Early Partial Evaluation in a JIT-Compiled, Retargetable Instruction Set Simulator Generated from a High-Level Architecture Description. DAC'13.

Productivity

Performance



[Simit-ARM2006]

+ Page-based JIT
- Ad-hoc ADL with custom parser
- Unmaintained

[Wagstaff2013]

+ Region-based JIT
+ Industry-supported ADL (ArchC)
- C++-based ADL is verbose
- Not Public

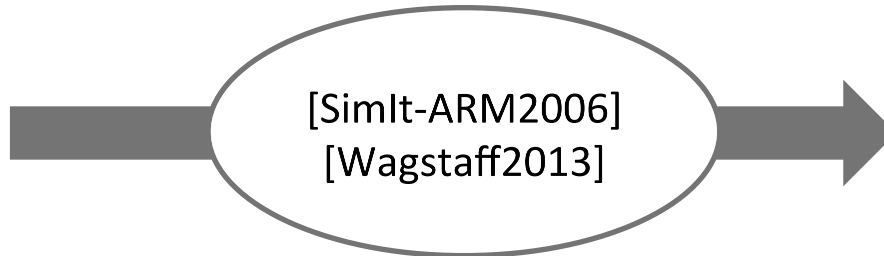
[Simit-ARM2006] J.D'Errico and W.Qin. Constructing Portable Compiled Instruction-Set Simulators — An ADL-Driven Approach. DATE'06.

[Wagstaff2013] H. Wagstaff, M. Gould, B. Franke, and N.Topham. Early Partial Evaluation in a JIT-Compiled, Retargetable Instruction Set Simulator Generated from a High-Level Architecture Description. DAC'13.

Productivity

Performance

Architectural Description Language



Instruction Set Interpreter in C with DBT

Dynamic Language Interpreter in C with JIT Compiler

Productivity Performance

Architectural Description Language

[SimIt-ARM2006]
[Wagstaff2013]

Instruction Set Interpreter in C with DBT

Key Insight:

Similar productivity-performance challenges for building high-performance interpreters of dynamic languages.
(e.g. JavaScript, Python)

Dynamic Language Interpreter in C with JIT Compiler

Productivity

Performance

Architectural Description Language

[SimIt-ARM2006]
[Wagstaff2013]

Instruction Set Interpreter in C with DBT

Key Insight:

Similar productivity-performance challenges for building high-performance interpreters of dynamic languages.
(e.g. JavaScript, Python)

Dynamic Language Interpreter in C with JIT Compiler

Productivity Performance

Architectural Description Language → [SimIt-ARM2006] [Wagstaff2013] → Instruction Set Interpreter in C with DBT

Dynamic-Language Interpreter in RPython → RPython Translation Toolchain → Dynamic Language Interpreter in C with JIT Compiler

Productivity

Performance

Architectural Description Language → [SimIt-ARM2006] [Wagstaff2013] → Instruction Set Interpreter in C with DBT

pypy

Dynamic-Language Interpreter in RPython → RPython Translation Toolchain → Dynamic Language Interpreter in C with JIT Compiler

Meta-Tracing JIT:
makes JIT generation generic across languages

Productivity

Performance

Architectural Description Language

Instruction Set Interpreter in C with DBT

RPython Translation Toolchain

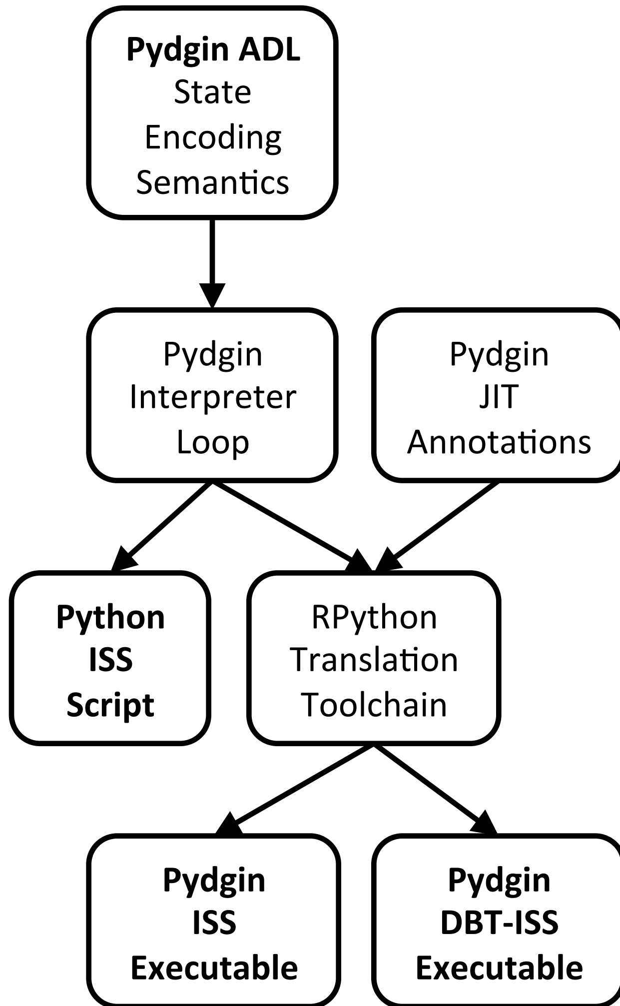
Productivity Performance

Architectural Description Language

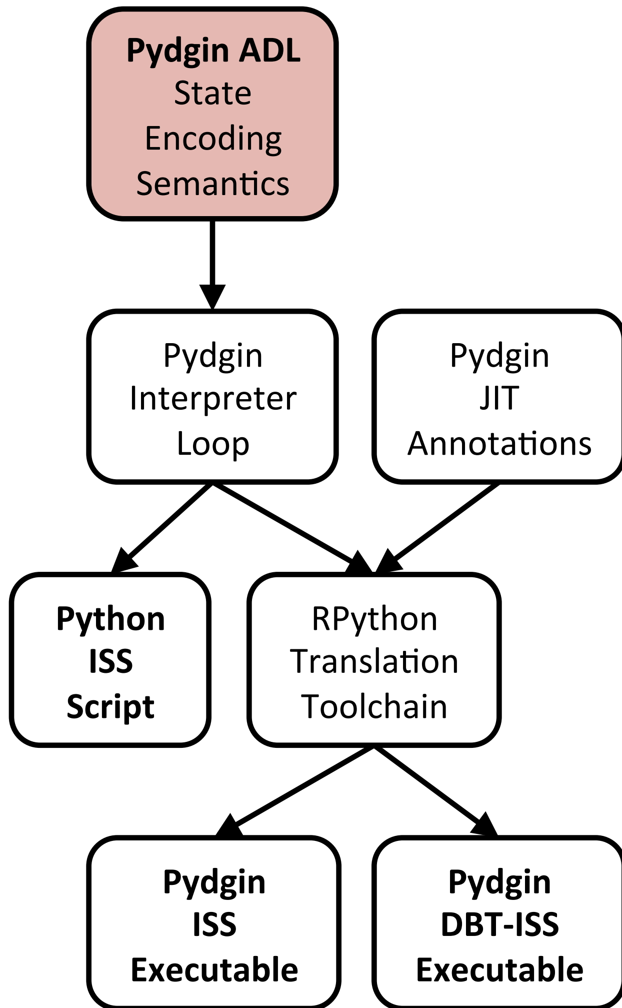
Instruction Set Interpreter in C with DBT

- Flexible, productive, pseudocode-like ADL syntax
- ADL embedded in a popular, general-purpose language
- Tracing-JIT generator applies across many different ISAs
- Leverages advancements from dynamic-language JIT research

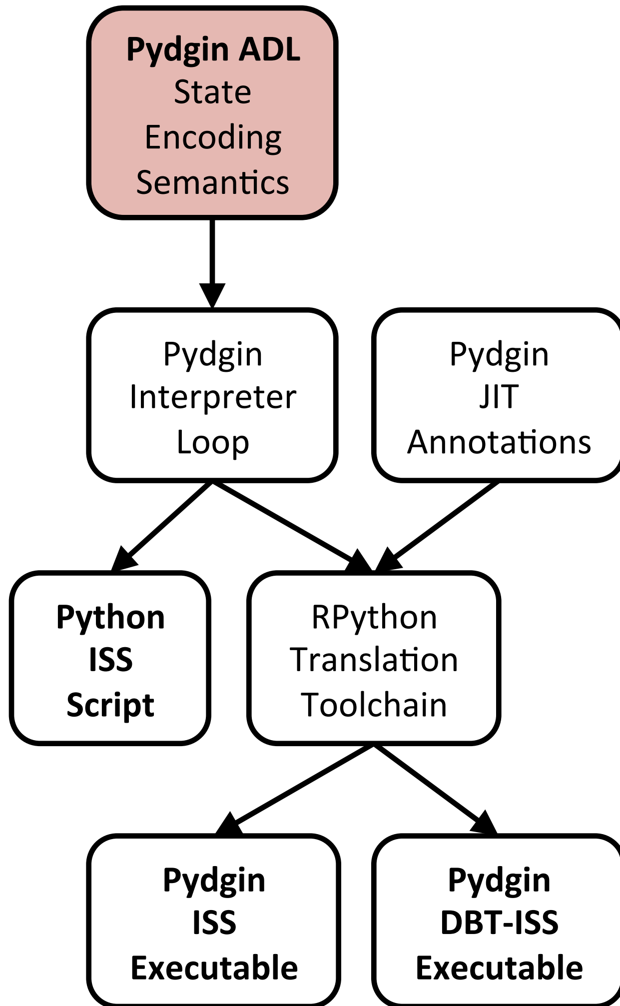
Pydgin Framework



Pydgin Framework

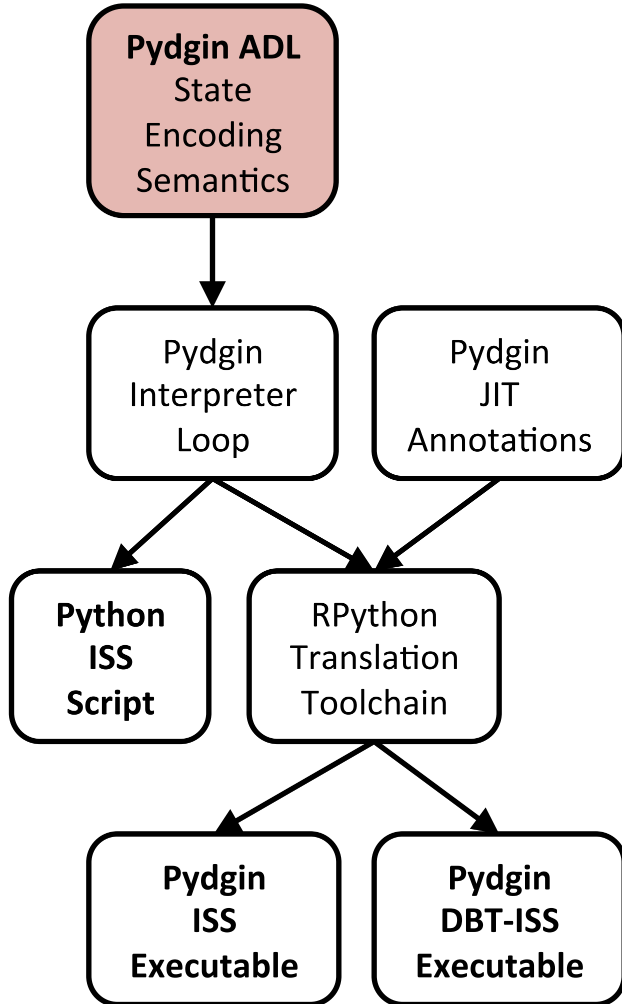


Pydgin ADL: ARMv5 Architectural State



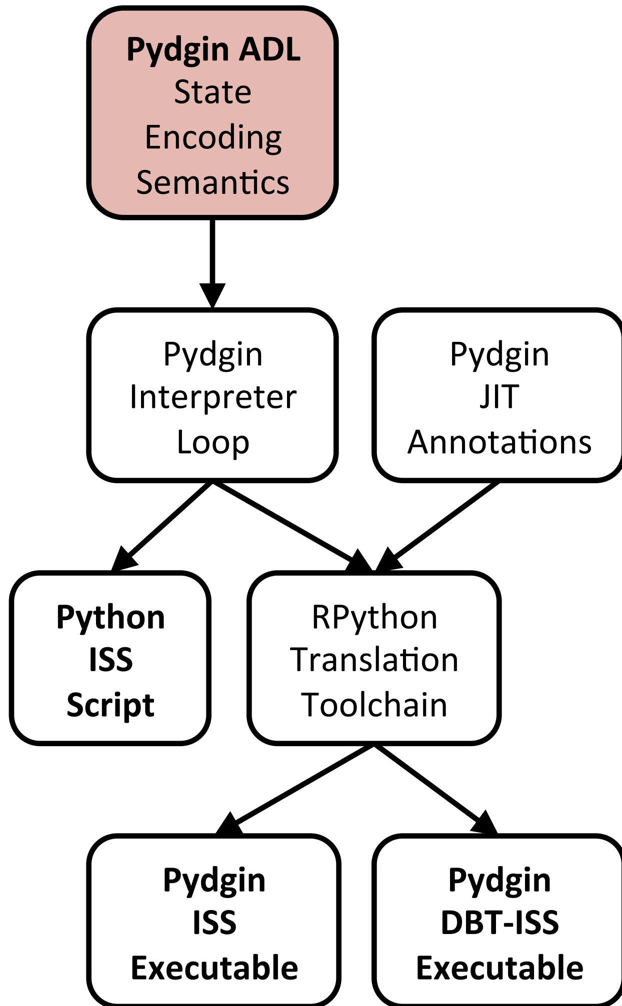

```
class State( object ):

  def __init__( self, memory, reset_addr=0x400 ):
    self.pc  = reset_addr
    self.rf  = ArmRegisterFile( self, num_regs=16 )
    self.mem = memory

    self.rf[ 15 ] = reset_addr

    # current program status register (CPSR)
    self.N    = 0b0      # Negative condition
    self.Z    = 0b0      # Zero condition
    self.C    = 0b0      # Carry condition
    self.V    = 0b0      # Overflow condition

  def fetch_pc( self ):
    return self.pc
```

Pydgin ADL: ARMv5 Architectural State



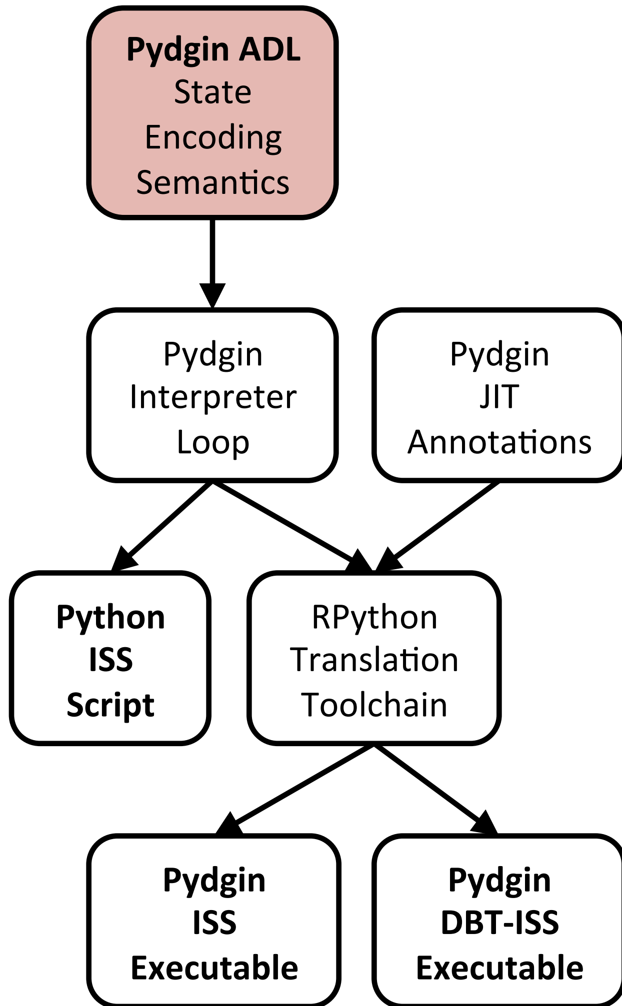

```
class State( object ):

  def __init__( self, memory, reset_addr=0x400 ):
    self.pc  = reset_addr
    self.rf  = ArmRegisterFile( self, num_regs=16 )
    self.mem = memory

    self.rf[ 15 ] = reset_addr

    # current program status register (CPSR)
    self.N    = 0b0      # Negative condition
    self.Z    = 0b0      # Zero condition
    self.C    = 0b0      # Carry condition
    self.V    = 0b0      # Overflow condition

  def fetch_pc( self ):
    return self.pc
```

Pydgin ADL: ARMv5 Architectural State



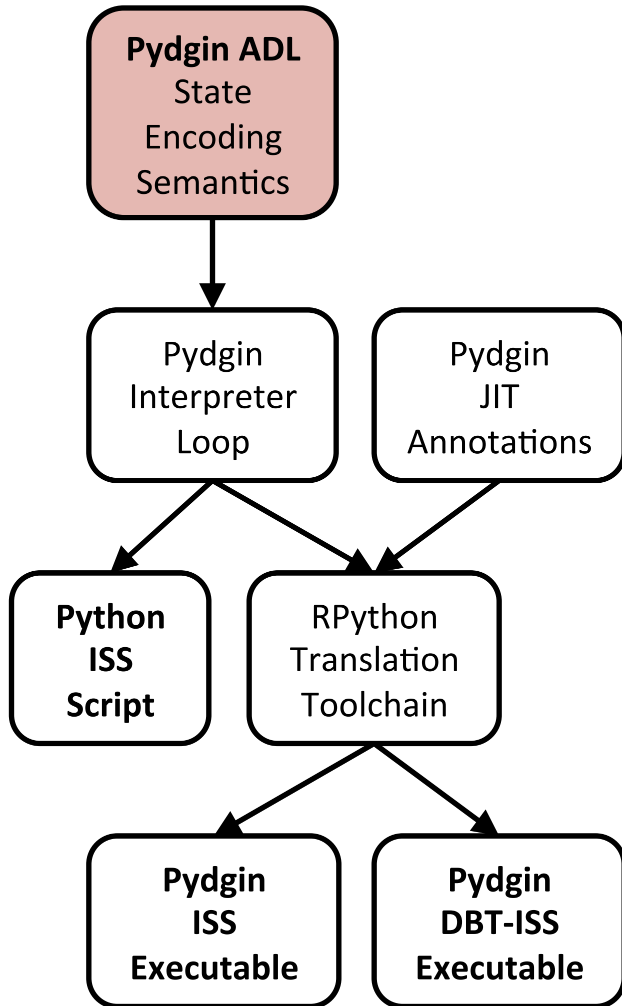

```
class State( object ):

  def __init__( self, memory, reset_addr=0x400 ):
    self.pc  = reset_addr
    self.rf  = ArmRegisterFile( self, num_regs=16 )
    self.mem = memory

    self.rf[ 15 ] = reset_addr

    # current program status register (CPSR)
    self.N    = 0b0      # Negative condition
    self.Z    = 0b0      # Zero condition
    self.C    = 0b0      # Carry condition
    self.V    = 0b0      # Overflow condition

  def fetch_pc( self ):
    return self.pc
```

Pydgin ADL: ARMv5 Architectural State



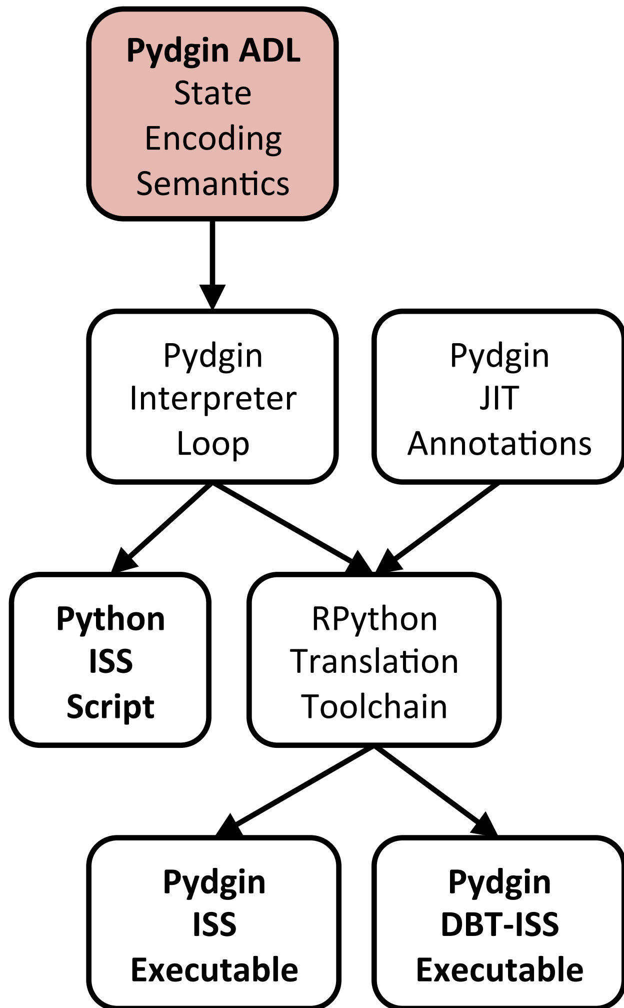

```
class State( object ):

  def __init__( self, memory, reset_addr=0x400 ):
    self.pc  = reset_addr
    self.rf  = ArmRegisterFile( self, num_regs=16 )
    self.mem = memory

    self.rf[ 15 ] = reset_addr

    # current program status register (CPSR)
    self.N    = 0b0      # Negative condition
    self.Z    = 0b0      # Zero condition
    self.C    = 0b0      # Carry condition
    self.V    = 0b0      # Overflow condition

  def fetch_pc( self ):
    return self.pc
```

Pydgin ADL: ARMv5 Encodings



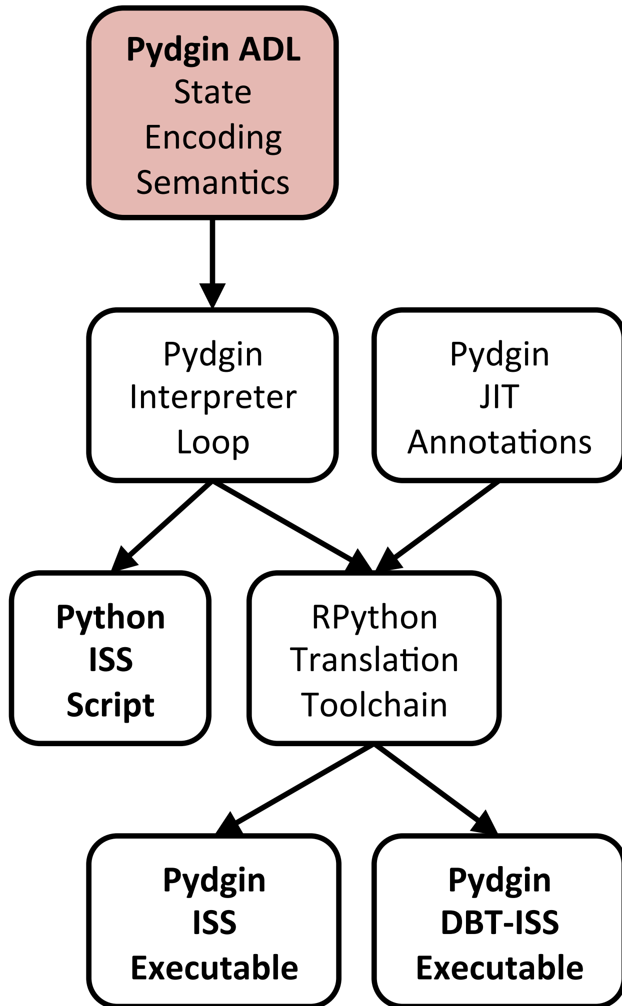

```
encodings = [

  ['nop',   '00000000000000000000000000000000'],
  ['mul',   'xxxx0000000xxxxxxxxxxxxx1001xxxx'],
  ['umull', 'xxxx0000100xxxxxxxxxxxxx1001xxxx'],
  ['adc',   'xxxx00x0101xxxxxxxxxxxxxxxxxxxxx'],

  ['and',   'xxxx00x0000xxxxxxxxxxxxxxxxxxxxx'],
  ['b',     'xxxx1010xxxxxxxxxxxxxxxxxxxxxxxx'],
  ['bl',    'xxxx1011xxxxxxxxxxxxxxxxxxxxxxxx'],
  ['bic',   'xxxx00x1110xxxxxxxxxxxxxxxxxxxxx'],
  ['bkpt',  '111000010010xxxxxxxxxxxx0111xxxx'],

  # ...

  ['teq',   'xxxx00x10011xxxxxxxxxxxxxxxxxxxx'],
  ['tst',   'xxxx00x10001xxxxxxxxxxxxxxxxxxxx'],

]
```

Pydgin ADL: ARMv5 Encodings



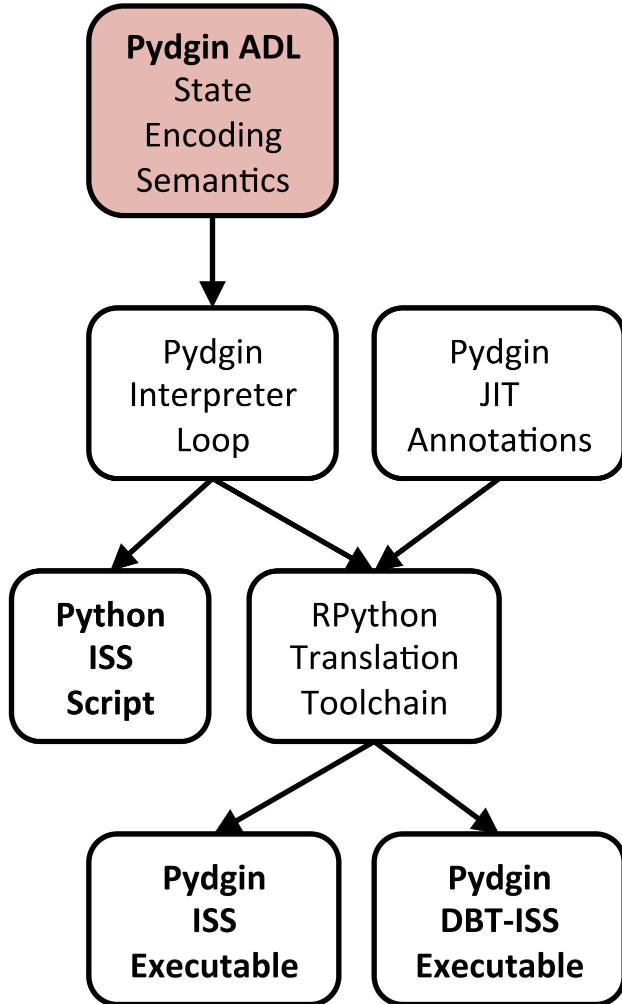

```
encodings = [

  ['nop',   '00000000000000000000000000000000'],
  ['mul',   'xxxx0000000xxxxxxxxxxxxx1001xxxx'],
  ['umull', 'xxxx0000100xxxxxxxxxxxxx1001xxxx'],
  ['adc',   'xxxx00x0101xxxxxxxxxxxxxxxxxxxxx'],

  ['and',   'xxxx00x0000xxxxxxxxxxxxxxxxxxxxx'],
  ['b',     'xxxx1010xxxxxxxxxxxxxxxxxxxxxxxx'],
  ['bl',    'xxxx1011xxxxxxxxxxxxxxxxxxxxxxxx'],
  ['bic',   'xxxx00x1110xxxxxxxxxxxxxxxxxxxxx'],
  ['bkpt',  '111000010010xxxxxxxxxxxx0111xxxx'],

  # ...

  ['teq',   'xxxx00x10011xxxxxxxxxxxxxxxxxxxx'],
  ['tst',   'xxxx00x10001xxxxxxxxxxxxxxxxxxxx'],

]
```

Pydgin ADL: ARMv5 Encodings



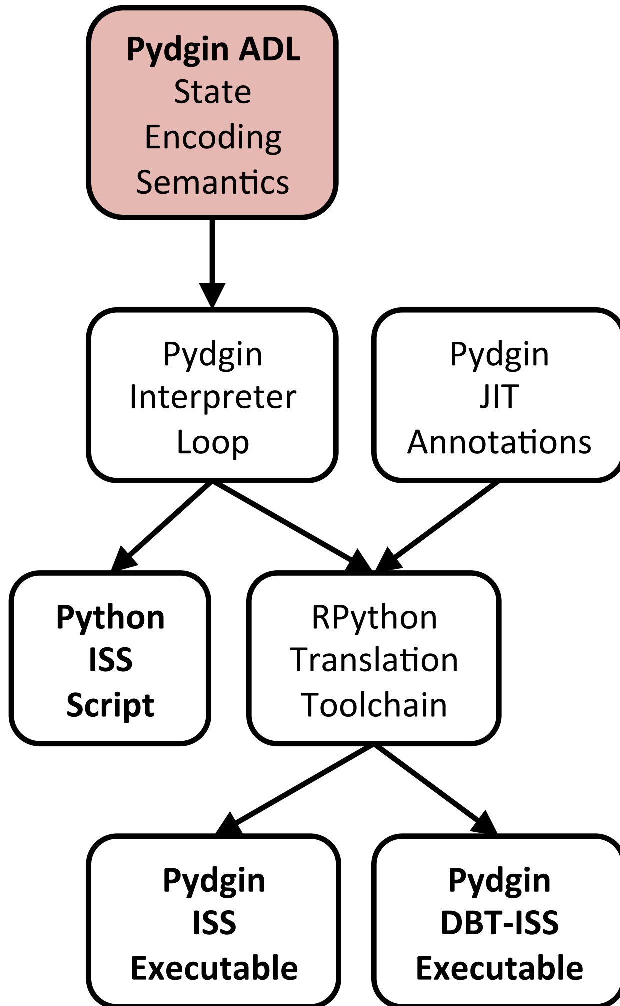

```
encodings = [

  ['nop',    '00000000000000000000000000000000'],
  ['mul',    'xxxx0000000xxxxxxxxxxxxx1001xxxx'],
  ['umull',  'xxxx0000100xxxxxxxxxxxxx1001xxxx'],
  ['adc',    'xxxx00x0101xxxxxxxxxxxxxxxxxxxxx'],

  ['and',    'xxxx00x0000xxxxxxxxxxxxxxxxxxxxx'],
  ['b',      'xxxx1010xxxxxxxxxxxxxxxxxxxxxxxx'],
  ['bl',     'xxxx1011xxxxxxxxxxxxxxxxxxxxxxxx'],
  ['bic',    'xxxx00x1110xxxxxxxxxxxxxxxxxxxxx'],
  ['bkpt',   '111000010010xxxxxxxxxxxx0111xxxx'],

  # ...

  ['teq',    'xxxx00x10011xxxxxxxxxxxxxxxxxxxx'],
  ['tst',    'xxxx00x10001xxxxxxxxxxxxxxxxxxxx'],

]
```

Pydgin ADL: ARMv5 Encodings



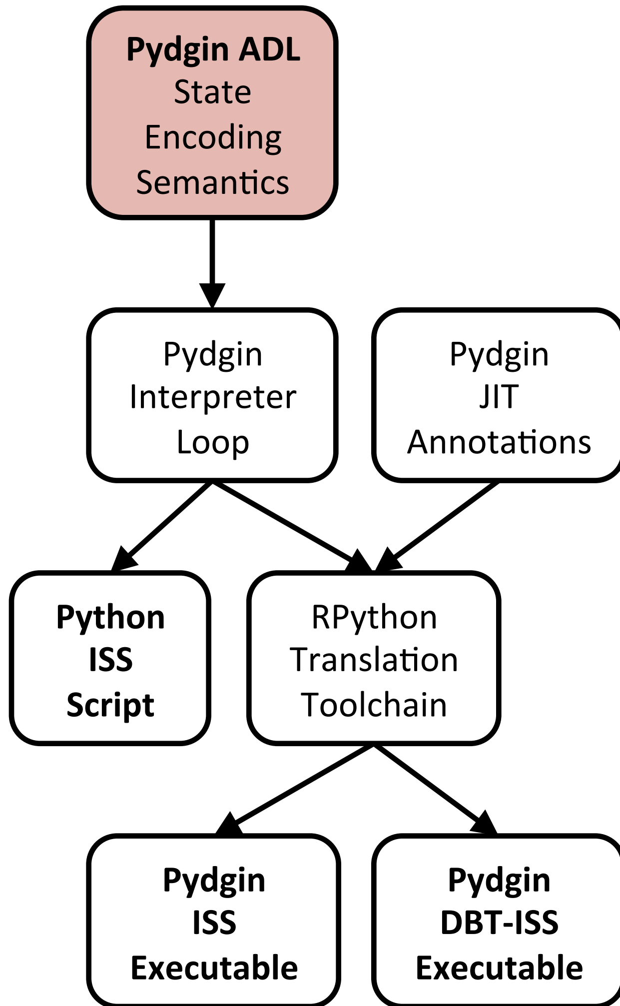

```
encodings = [

  ['nop',   '00000000000000000000000000000000'],
  ['mul',   'xxxx0000000xxxxxxxxxxxxx1001xxxx'],
  ['umull', 'xxxx0000100xxxxxxxxxxxxx1001xxxx'],
  ['adc',   'xxxx00x0101xxxxxxxxxxxxxxxxxxxxx'],
  ['add',   'xxxx00x0100xxxxxxxxxxxxxxxxxxxxx'],
  ['and',   'xxxx00x0000xxxxxxxxxxxxxxxxxxxxx'],
  ['b',     'xxxx1010xxxxxxxxxxxxxxxxxxxxxxxx'],
  ['bl',    'xxxx1011xxxxxxxxxxxxxxxxxxxxxxxx'],
  ['bic',   'xxxx00x1110xxxxxxxxxxxxxxxxxxxxx'],
  ['bkpt',  '111000010010xxxxxxxxxxxx0111xxxx'],

  # ...

  ['teq',   'xxxx00x10011xxxxxxxxxxxxxxxxxxxx'],
  ['tst',   'xxxx00x10001xxxxxxxxxxxxxxxxxxxx'],

]
```

Pydgin ADL: ARMv5 Instruction Semantics



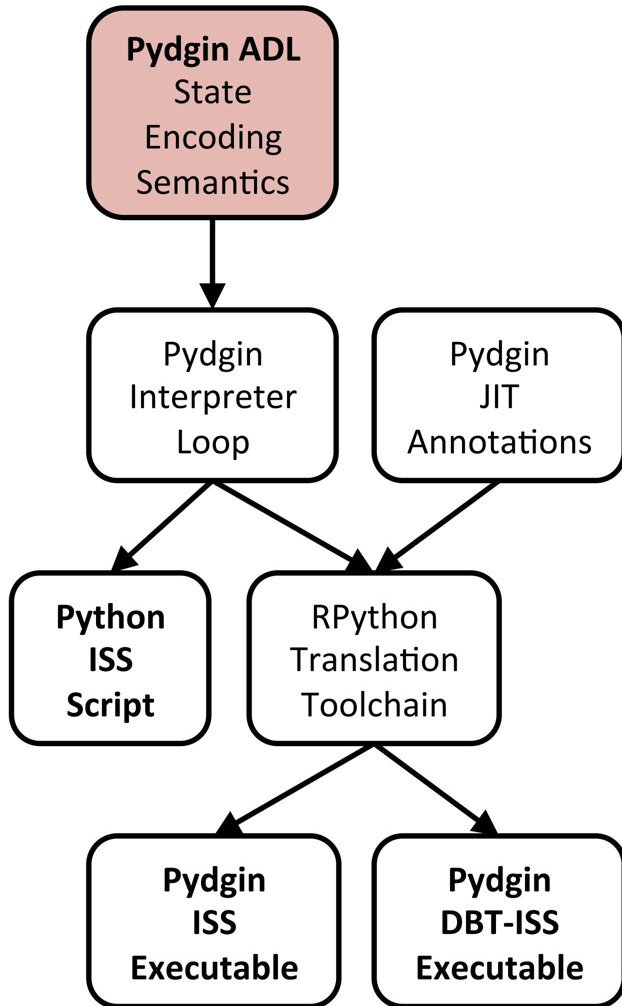

```
def execute_add( s, inst ):

  if condition_passed( s, inst.cond() ):
    a,   = s.rf[ inst.rn() ]
    b, _ = shifter_operand( s, inst )
    result = a + b
    s.rf[ inst.rd() ] = trim_32(result)

    if inst.S():
      # ...
      s.N = (result >> 31)&1
      s.Z = trim_32(result) == 0
      s.C = carry_from(result)
      s.V = overflow_from(a, b, result)

    if inst.rd() == 15:
      return
  s.rf[PC] = s.fetch_pc() + 4
```

Pydgin ADL: ARMv5 Instruction Semantics



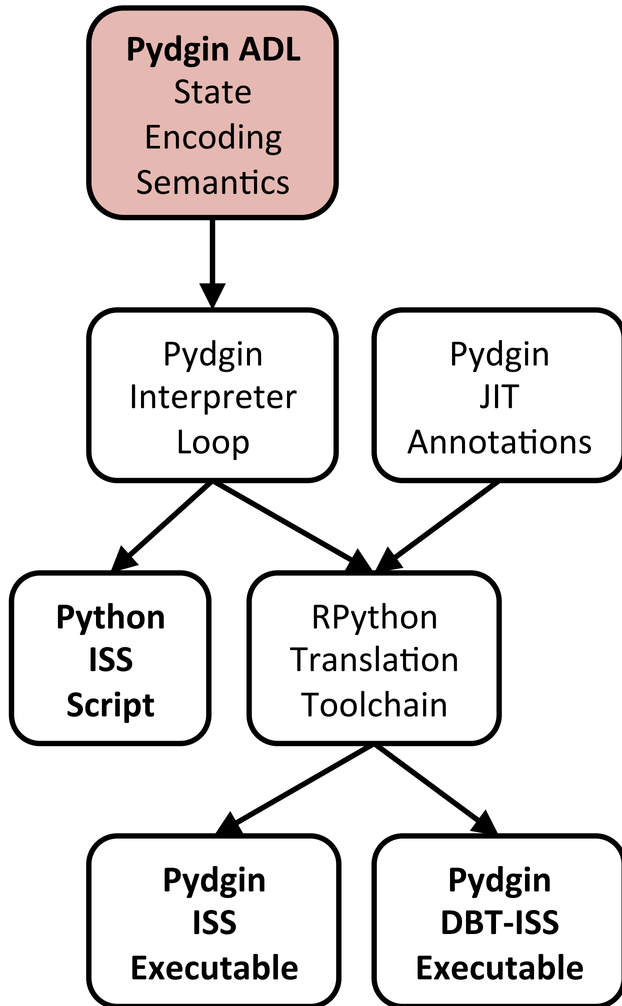

```
def execute_add( s, inst ):

  if condition_passed( s, inst.cond() ):
    a,   = s.rf[ inst.rn() ]
    b, _ = shifter_operand( s, inst )
    result = a + b
    s.rf[ inst.rd() ] = trim_32(result)

    if inst.S():
      # ...
      s.N = (result >> 31)&1
      s.Z = trim_32(result) == 0
      s.C = carry_from(result)
      s.V = overflow_from(a, b, result)

    if inst.rd() == 15:
      return
  s.rf[PC] = s.fetch_pc() + 4
```

Pydgin ADL: ARMv5 Instruction Semantics




```
def execute_add( s, inst ):

  if condition_passed( s, inst.cond() ):
    a,    = s.rf[ inst.rn() ]
    b, _ = shifter_operand( s, inst )
    result = a + b
    s.rf[ inst.rd() ] = trim_32(result)

    if inst.S():
      # ...
      s.N = (result >> 31)&1
      s.Z = trim_32(result) == 0
      s.C = carry_from(result)
      s.V = overflow_from(a, b, result)

    if inst.rd() == 15:
      return
  s.rf[PC] = s.fetch_pc() + 4
```

Pydgin ADL: ARMv5 Instruction Semantics




```
def execute_add( s, inst ):

  if condition_passed( s, inst.cond() ):
    a,    = s.rf[ inst.rn() ]
    b, _ = shifter_operand( s, inst )
    result = a + b
    s.rf[ inst.rd() ] = trim_32(result)

    if inst.S():
      # ...
      s.N = (result >> 31)&1
      s.Z = trim_32(result) == 0
      s.C = carry_from(result)
      s.V = overflow_from(a, b, result)

    if inst.rd() == 15:
      return
  s.rf[PC] = s.fetch_pc() + 4
```

Pydgin ADL: ARMv5 Instruction Semantics

Pydgin ADL
State

ARM ISA MANUAL SPEC

```
if ConditionPassed(cond) then

  Rd = Rn + shifter_operand

  if S == 1 and Rd == R15 then
    if CurrentModeHasSPSR() then
      CPSR = SPSR
    else UNPREDICTABLE

  else if S == 1 then
    N Flag = Rd[31]
    Z Flag = if Rd == 0 then 1 else 0
    C Flag = CarryFrom(Rn + shifter_operand)
    V Flag = OverflowFrom(Rn + shifter_operand)
```

Executable Executable

```
def execute_add( s, inst ):

  if condition_passed( s, inst.cond() ):
    a,    = s.rf[ inst.rn() ]
    b, _ = shifter_operand( s, inst )
    result = a + b
    s.rf[ inst.rd() ] = trim_32(result)

    if inst.S():
      # ...
      s.N = (result >> 31)&1
      s.Z = trim_32(result) == 0
      s.C = carry_from(result)
      s.V = overflow_from(a, b, result)

    if inst.rd() == 15:
      return
  s.rf[PC] = s.fetch_pc() + 4
```

Pydgin ADL: ARMv5 Instruction Semantics

Pydgin ADL
State

ARM ISA MANUAL SPEC

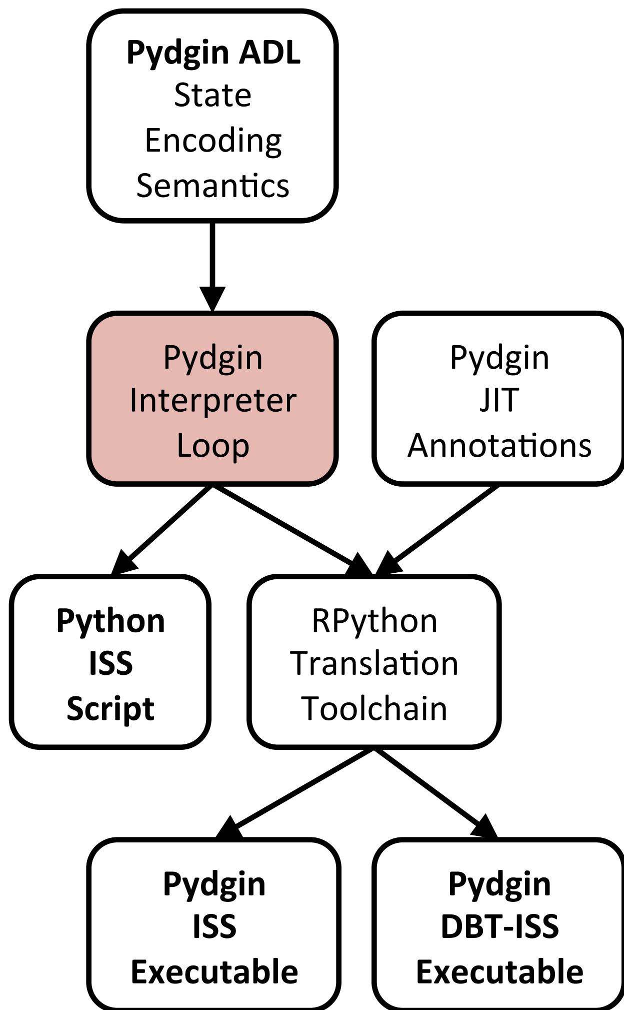
```
if ConditionPassed(cond) then

  Rd = Rn + shifter_operand

  if S == 1 and Rd == R15 then
    if CurrentModeHasSPSR() then
      CPSR = SPSR
    else UNPREDICTABLE

  else if S == 1 then
    N Flag = Rd[31]
    Z Flag = if Rd == 0 then 1 else 0
    C Flag = CarryFrom(Rn + shifter_operand)
    V Flag = OverflowFrom(Rn + shifter_operand)
```

Executable

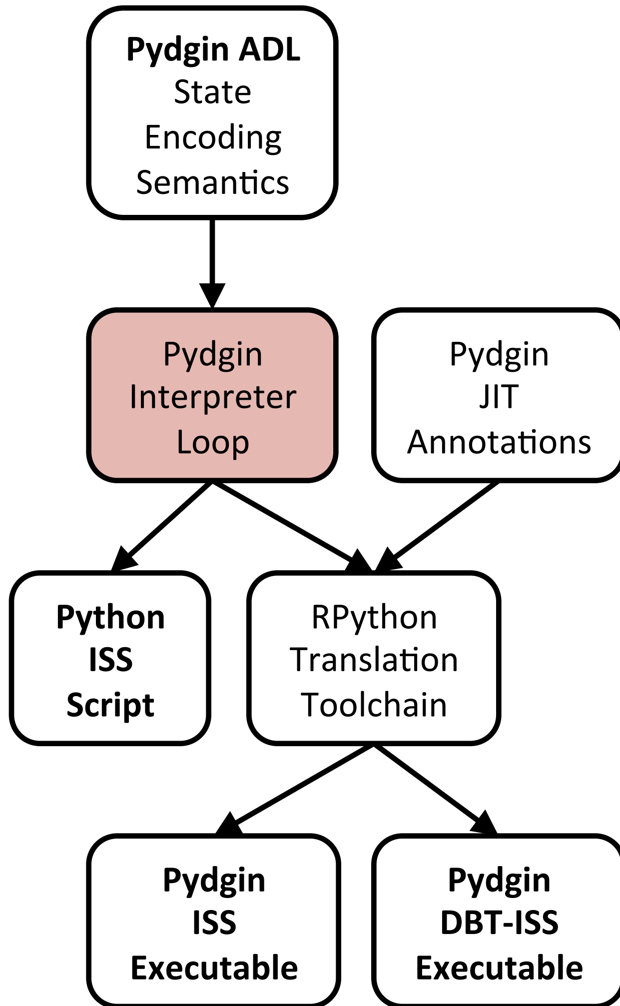
Executable

```python
def execute_add( s, inst ):

  if condition_passed( s, inst.cond() ):
    a,    = s.rf[ inst.rn() ]
    b, _ = shifter_operand( s, inst )
    result = a + b
    s.rf[ inst.rd() ] = trim_32(result)

    if inst.S():
      # ...
      s.N = (result >> 31)&1
      s.Z = trim_32(result) == 0
      s.C = carry_from(result)
      s.V = overflow_from(a, b, result)

    if inst.rd() == 15:
      return
  s.rf[PC] = s.fetch_pc() + 4
```

RPython ISS

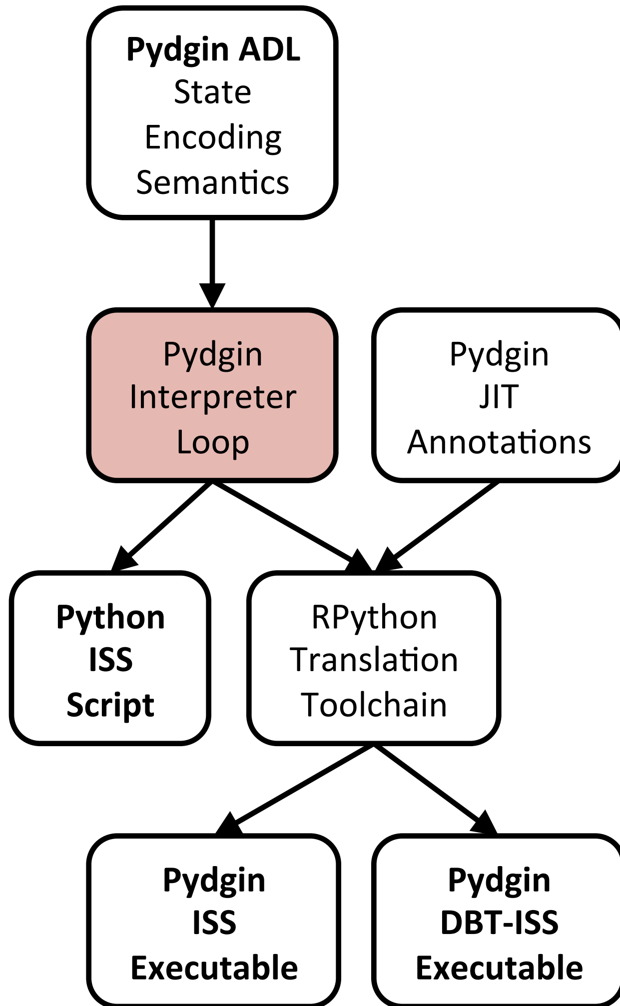


RPython ISS



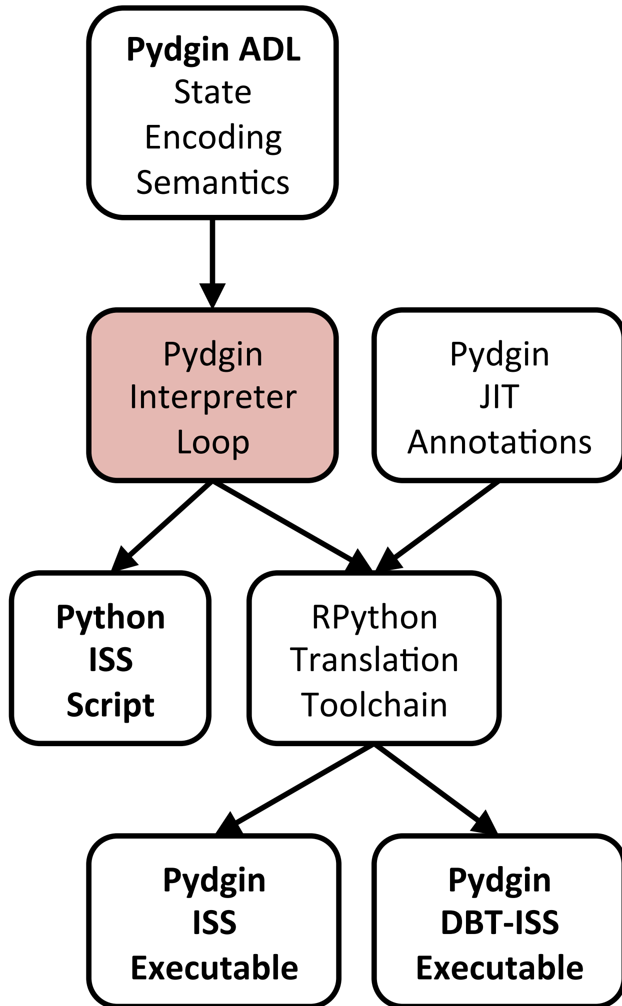

```
def instruction_set_interpreter( memory ):
  state = State( memory )

  while True:


    pc      = state.fetch_pc()

    inst    = memory[ pc ]        # fetch
    execute = decode( inst )      # decode
    execute( state, inst )        # execute
```

RPython ISS



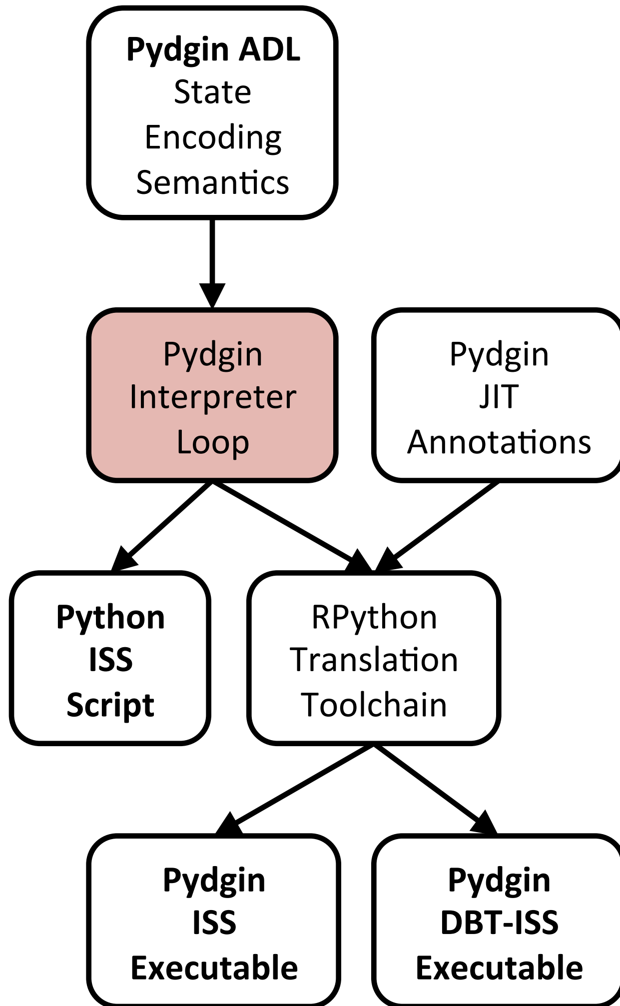

```
def instruction_set_interpreter( memory ):
  state = State( memory )

  while True:


    pc      = state.fetch_pc()

    inst    = memory[ pc ]        # fetch
    execute = decode( inst )      # decode
    execute( state, inst )        # execute
```

RPython ISS



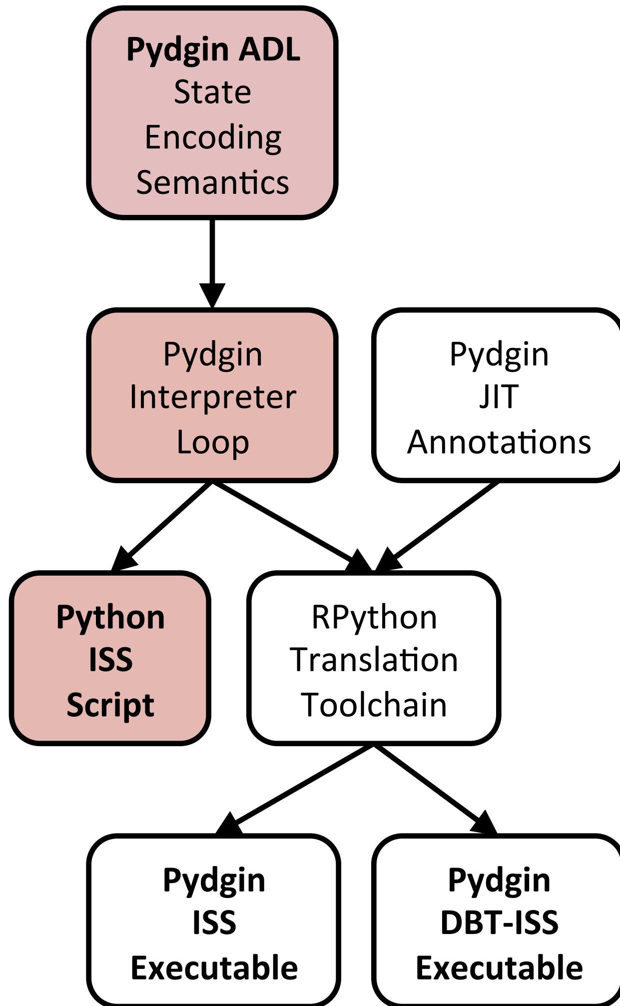

```
def instruction_set_interpreter( memory ):
  state = State( memory )

  while True:


    pc      = state.fetch_pc()

    inst    = memory[ pc ]        # fetch
    execute = decode( inst )      # decode
    execute( state, inst )        # execute
```

RPython ISS




```
def instruction_set_interpreter( memory ):
  state = State( memory )

  while True:


    pc      = state.fetch_pc()

    inst    = memory[ pc ]      # fetch
    execute = decode( inst )    # decode
    execute( state, inst )      # execute
```

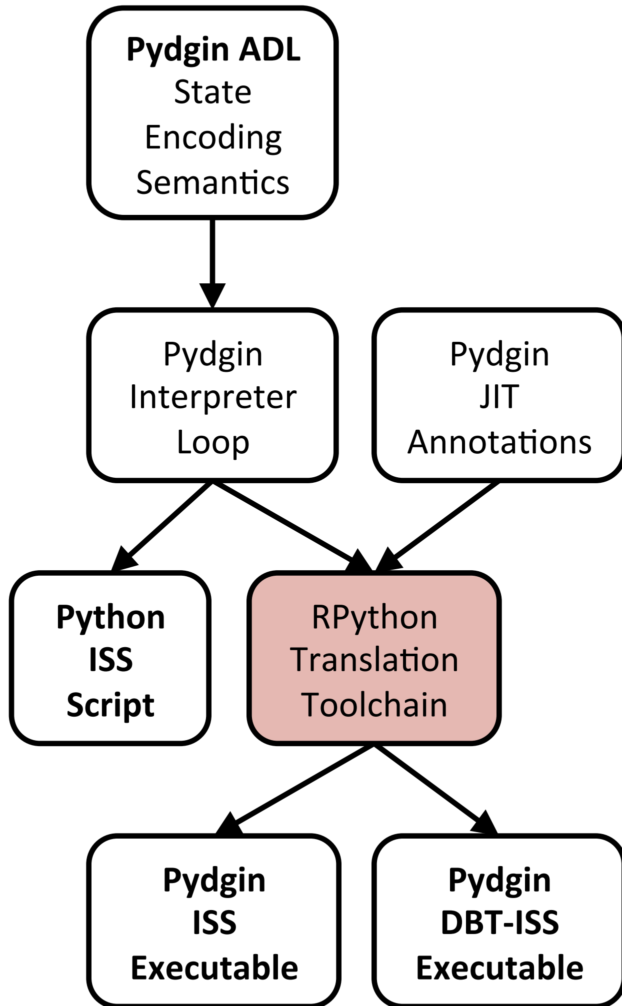
RPython ISS



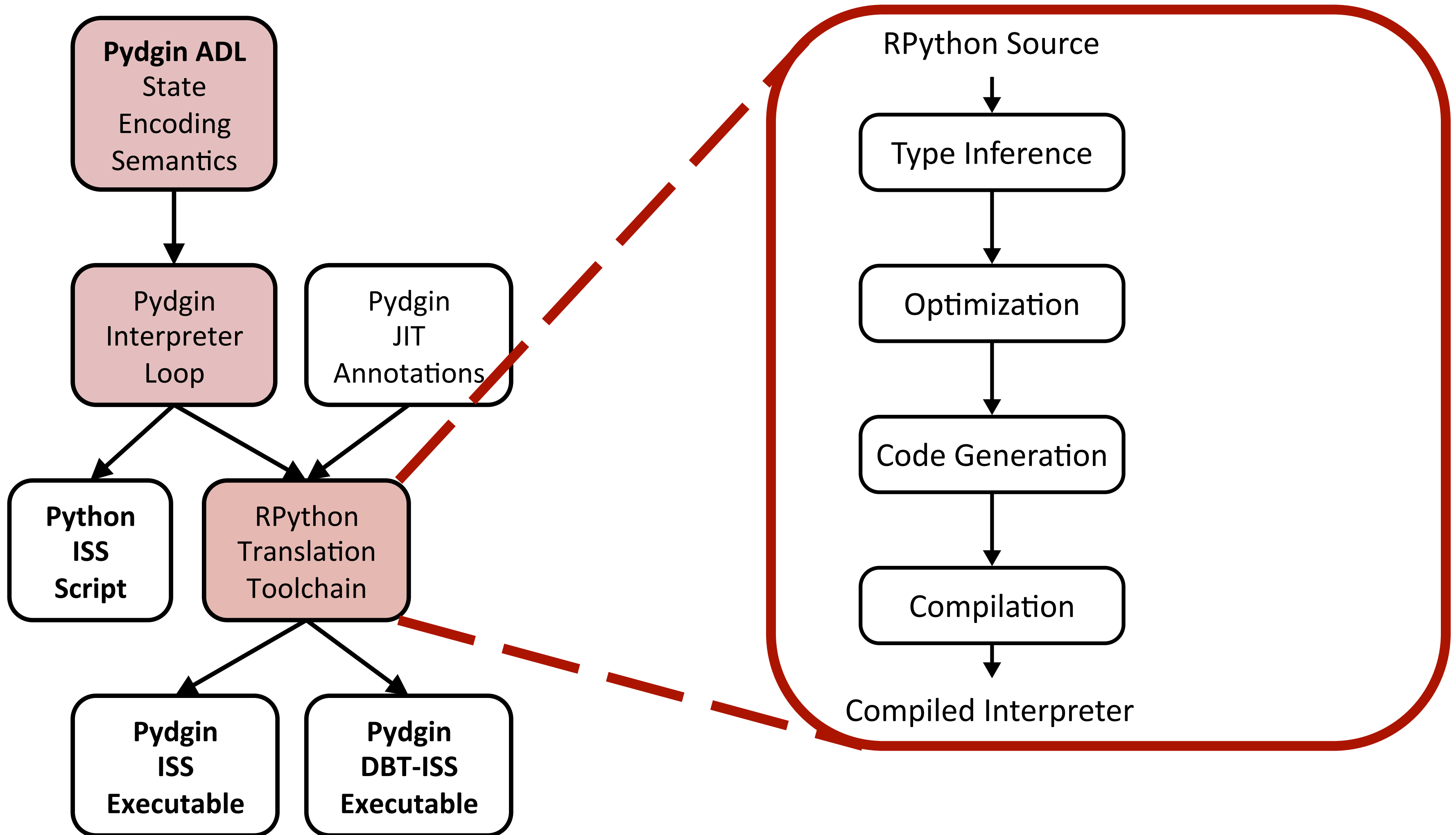

```
def instruction_set_interpreter( memory ):
  state = State( memory )

  while True:


    pc      = state.fetch_pc()

    inst    = memory[ pc ]      # fetch
    execute = decode( inst )    # decode
    execute( state, inst )      # execute
```

```
> python iss.py arm_binary
```

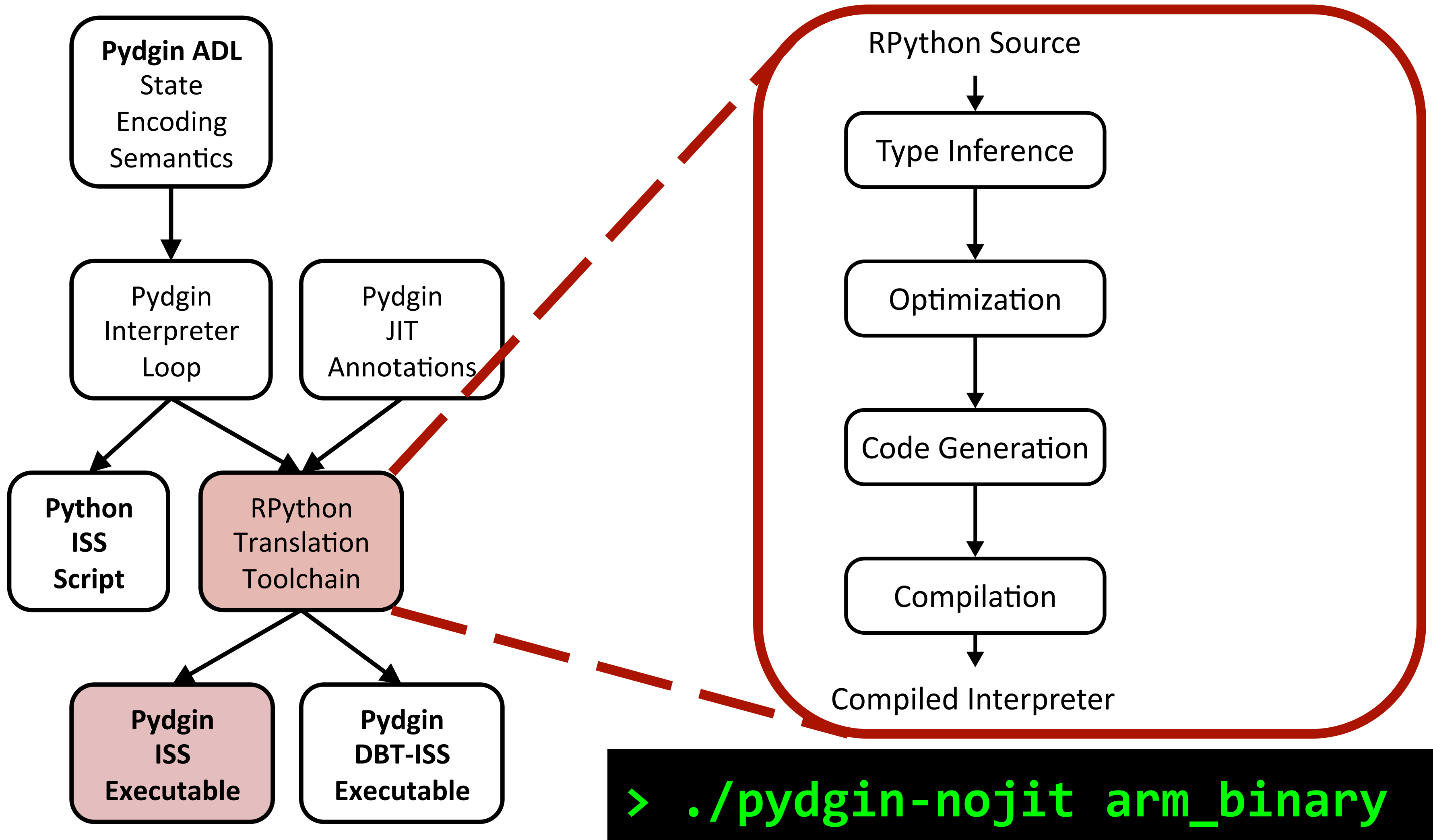
The RPython Translation Toolchain



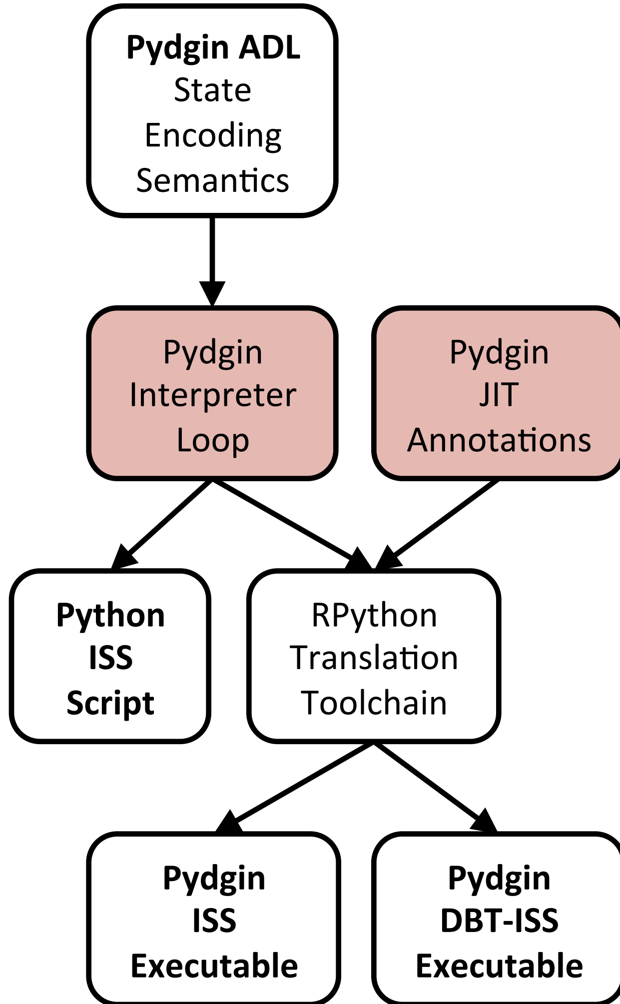
The RPython Translation Toolchain



The RPython Translation Toolchain

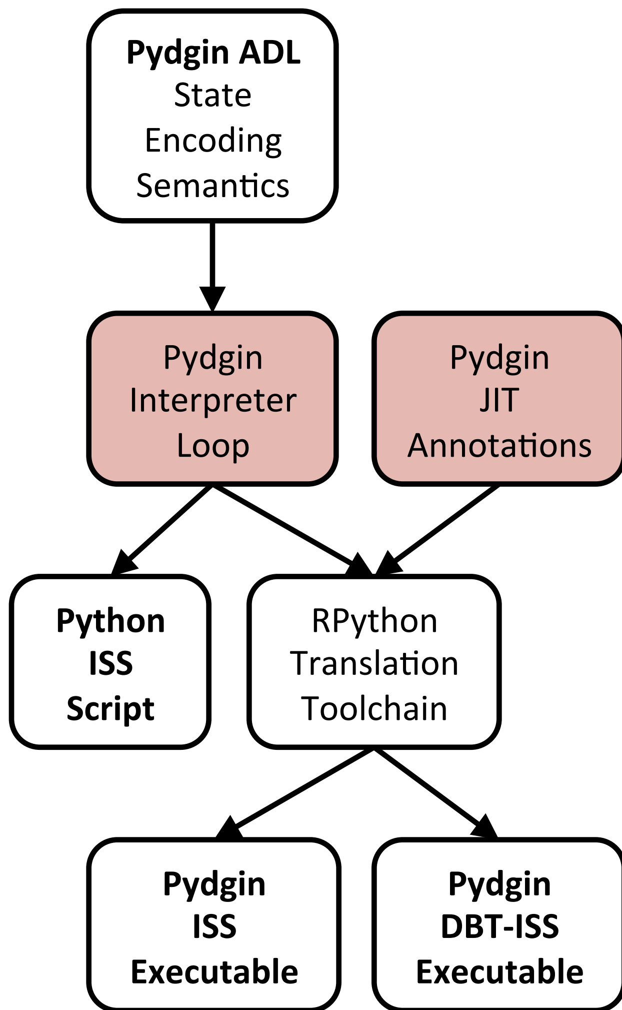


RPython ISS with JIT Annotations



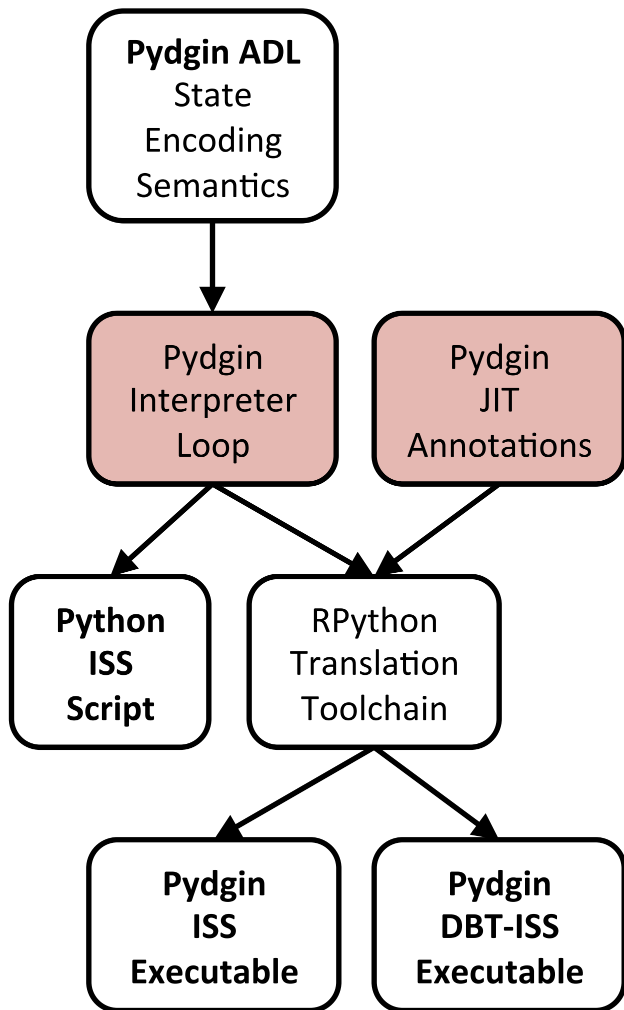

```
def instruction_set_interpreter( memory ):
  state = State( memory )

  while True:


    pc      = state.fetch_pc()

    inst    = memory[ pc ]       # fetch
    execute = decode( inst )     # decode
    execute( state, inst )       # execute
```

RPython ISS with JIT Annotations



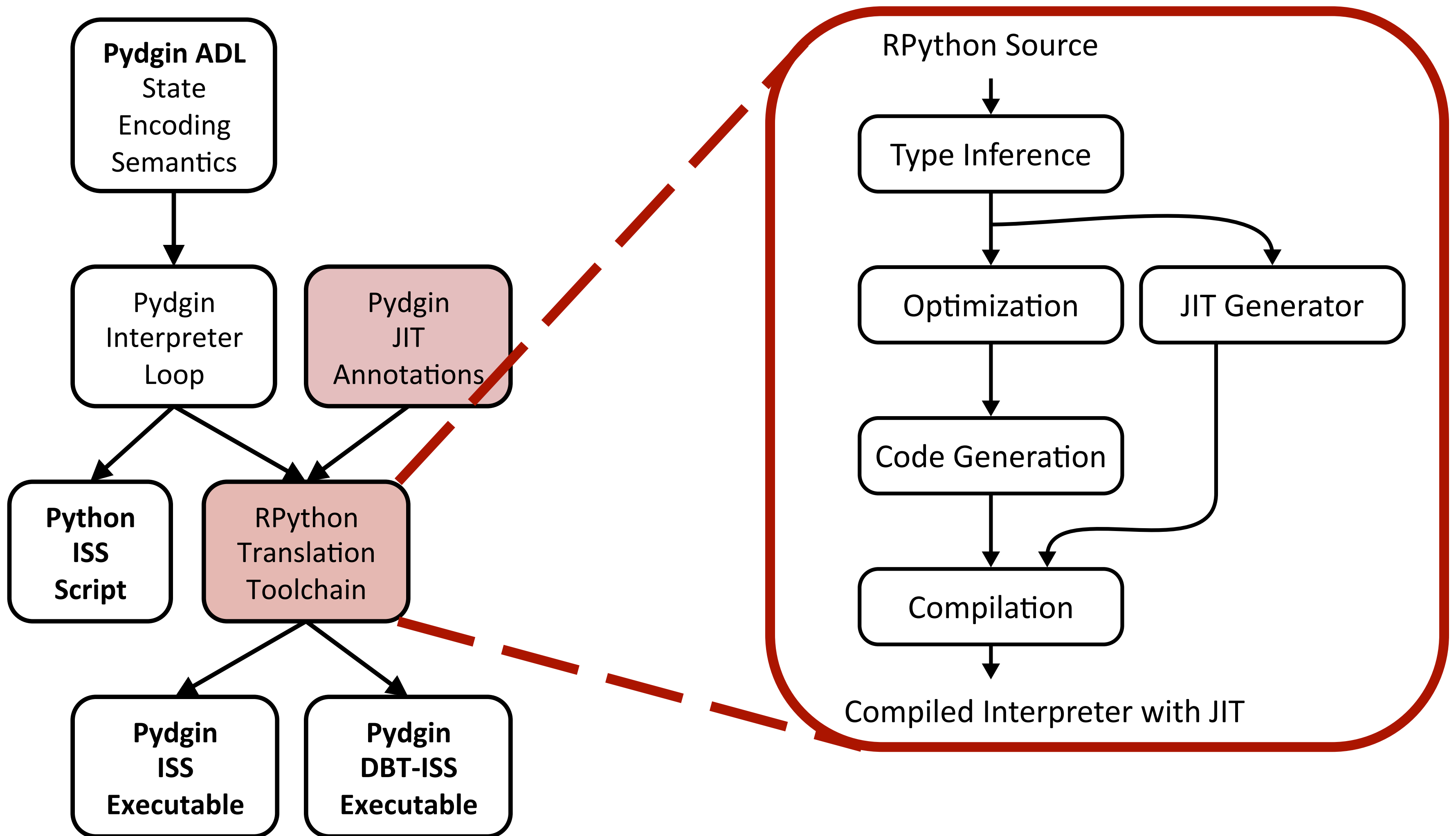

```
jd = JitDriver( greens = ['pc'],
                reds   = ['state'] )

def instruction_set_interpreter( memory ):
  state = State( memory )

  while True:
    jd.jit_merge_point( s.fetch_pc(), state )

    pc      = state.fetch_pc()

    inst    = memory[ pc ]       # fetch
    execute = decode( inst )     # decode
    execute( state, inst )       # execute

    if state.fetch_pc() < pc:
      jd.can_enter_jit( s.fetch_pc(), state )
```

RPython ISS with JIT Annotations

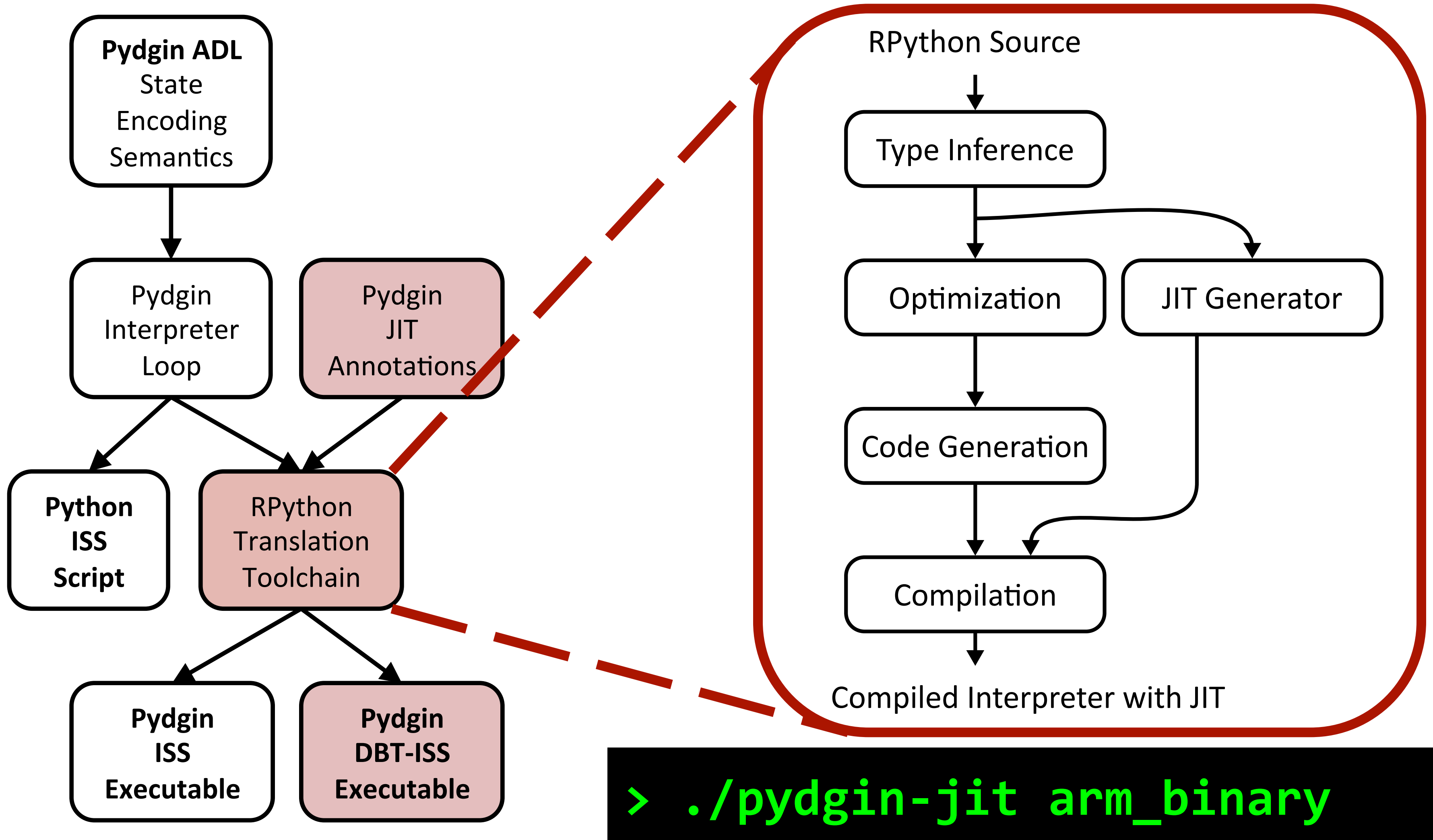



```
jd = JitDriver( greens = ['pc'],
                reds   = ['state'] )

def instruction_set_interpreter( memory ):
  state = State( memory )

  while True:
    jd.jit_merge_point( s.fetch_pc(), state )

    pc      = state.fetch_pc()

    inst    = memory[ pc ]       # fetch
    execute = decode( inst )     # decode
    execute( state, inst )       # execute

    if state.fetch_pc() < pc:
      jd.can_enter_jit( s.fetch_pc(), state )
```

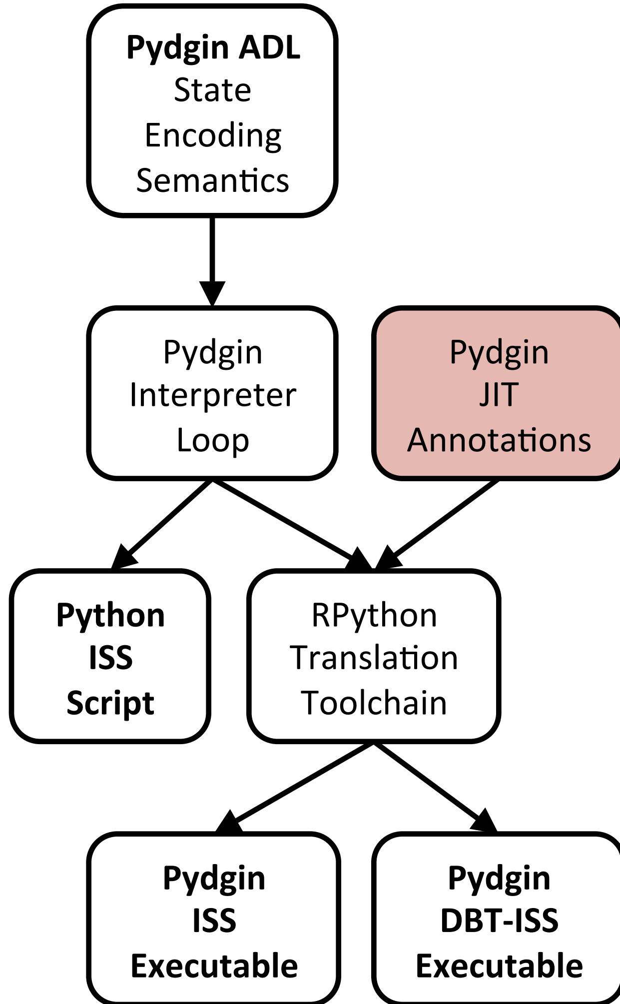
The RPython Translation Toolchain



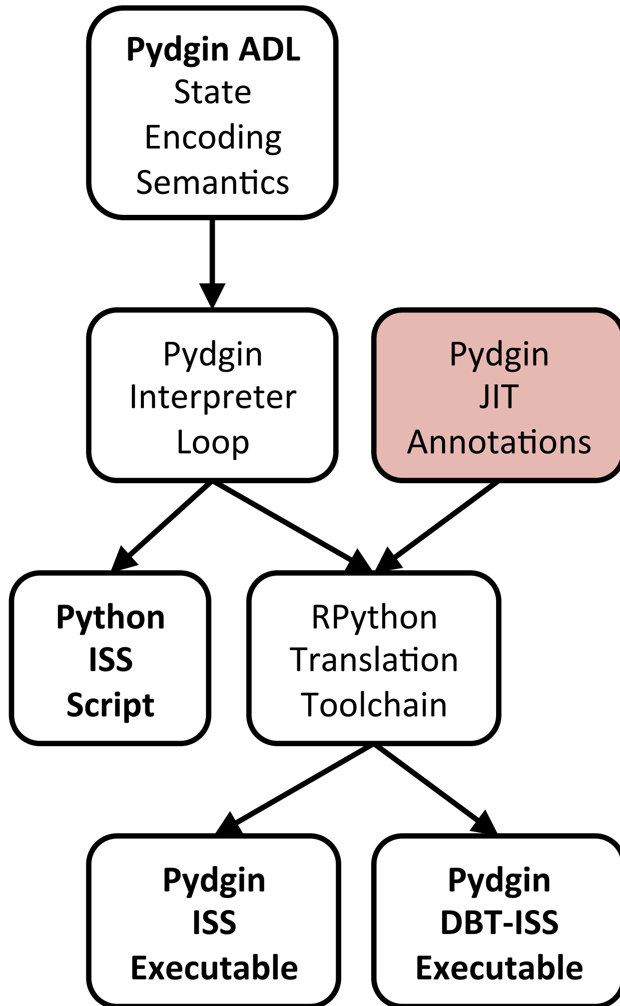
The RPython Translation Toolchain



JIT Annotations

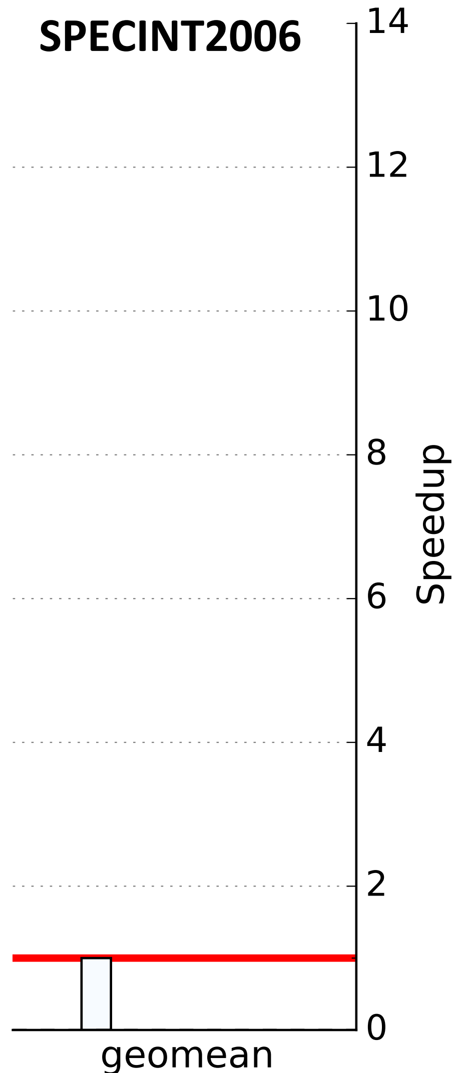


JIT Annotations



Creating a competitive JIT requires additional RPython JIT hints:

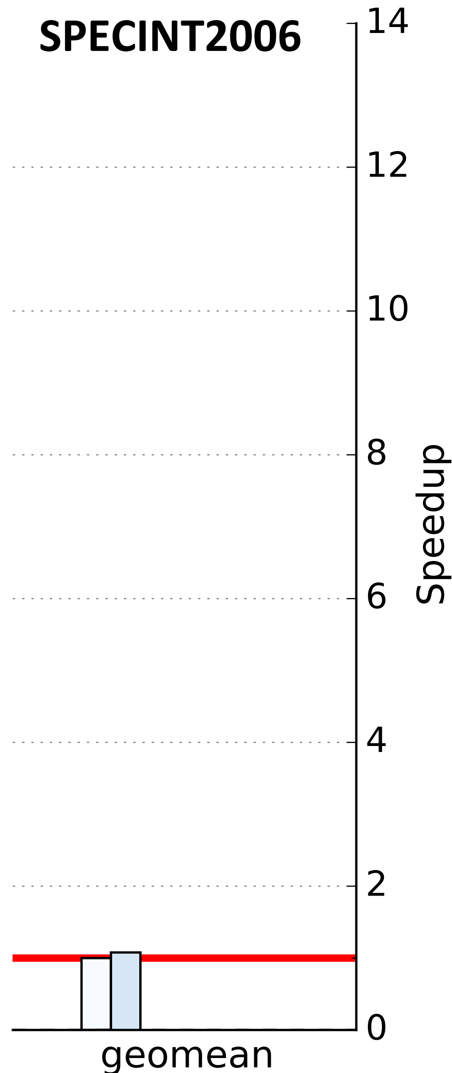
JIT Annotations



Creating a competitive JIT requires additional RPython JIT hints:

+ Minimal JIT Annotations

JIT Annotations

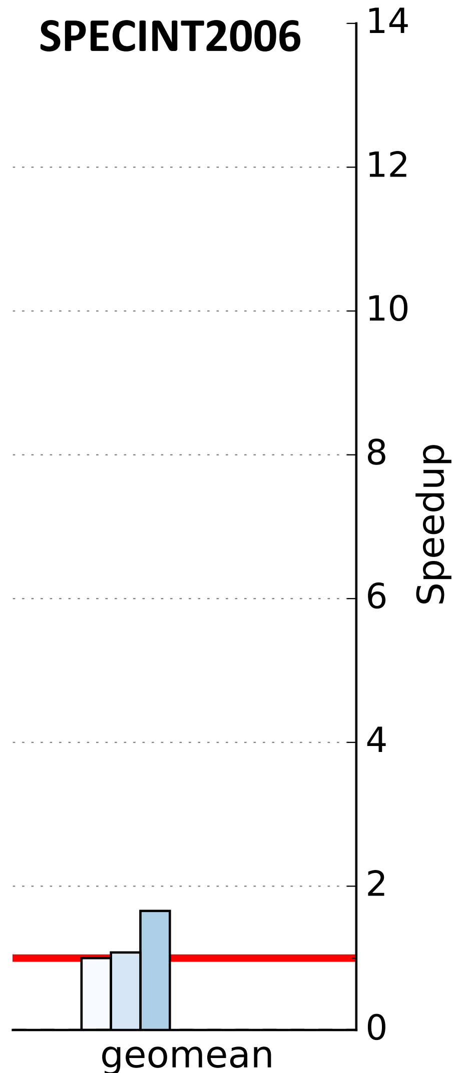


Creating a competitive JIT requires additional RPython JIT hints:

+ Minimal JIT Annotations

+ Elidable Instruction Fetch

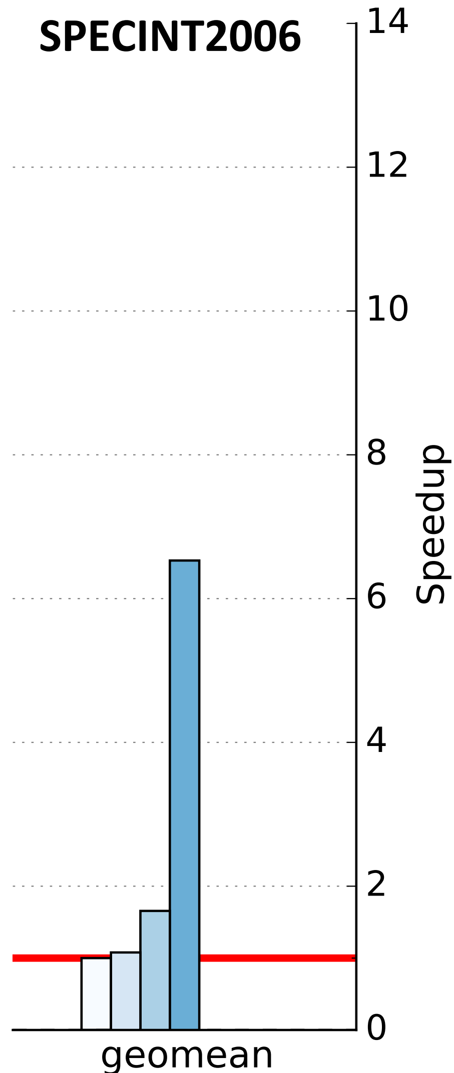
JIT Annotations



Creating a competitive JIT requires additional RPython JIT hints:

+ Minimal JIT Annotations
+ Elidable Instruction Fetch
+ Elidable Decode

JIT Annotations



Creating a competitive JIT requires additional RPython JIT hints:

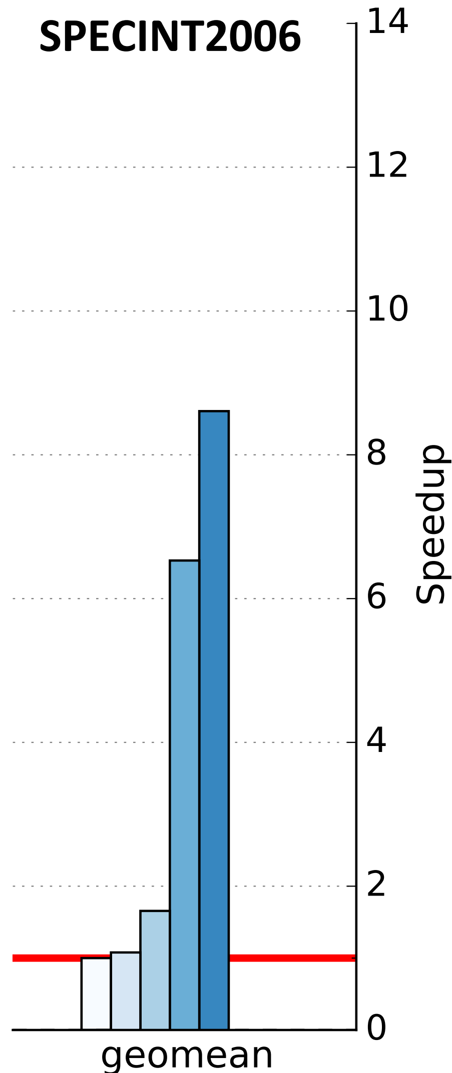
+ Minimal JIT Annotations

+ Elidable Instruction Fetch

+ Elidable Decode

+ Constant Promotion of PC and Memory

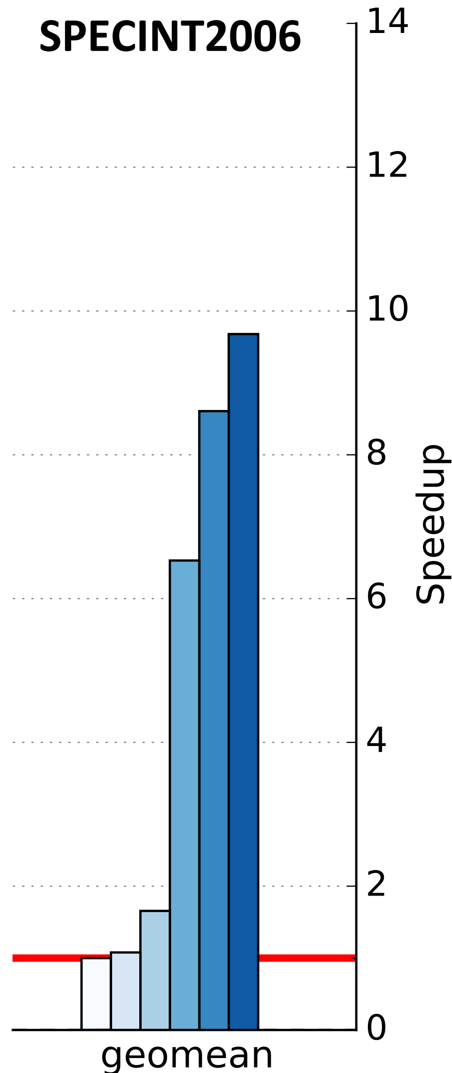
JIT Annotations



Creating a competitive JIT requires additional RPython JIT hints:

+ Minimal JIT Annotations
+ Elidable Instruction Fetch
+ Elidable Decode
+ Constant Promotion of PC and Memory
+ Word-Based Target Memory

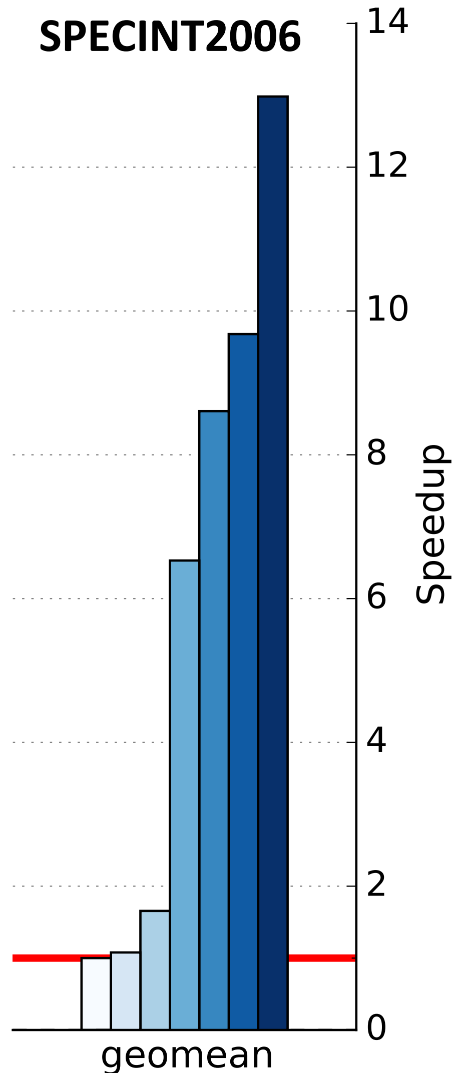
JIT Annotations



Creating a competitive JIT requires additional RPython JIT hints:

+ Minimal JIT Annotations
+ Elidable Instruction Fetch
+ Elidable Decode
+ Constant Promotion of PC and Memory
+ Word-Based Target Memory
+ Loop Unrolling in Instruction Semantics

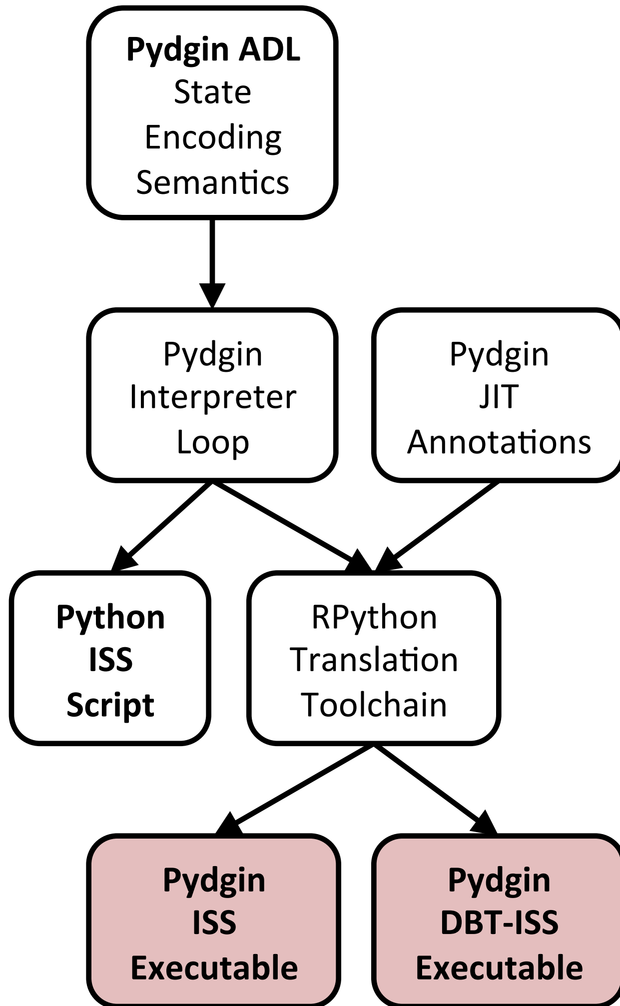
JIT Annotations



Creating a competitive JIT requires additional RPython JIT hints:

+ Minimal JIT Annotations
+ Elidable Instruction Fetch
+ Elidable Decode
+ Constant Promotion of PC and Memory
+ Word-Based Target Memory
+ Loop Unrolling in Instruction Semantics
+ Virtualizable PC and Statistics

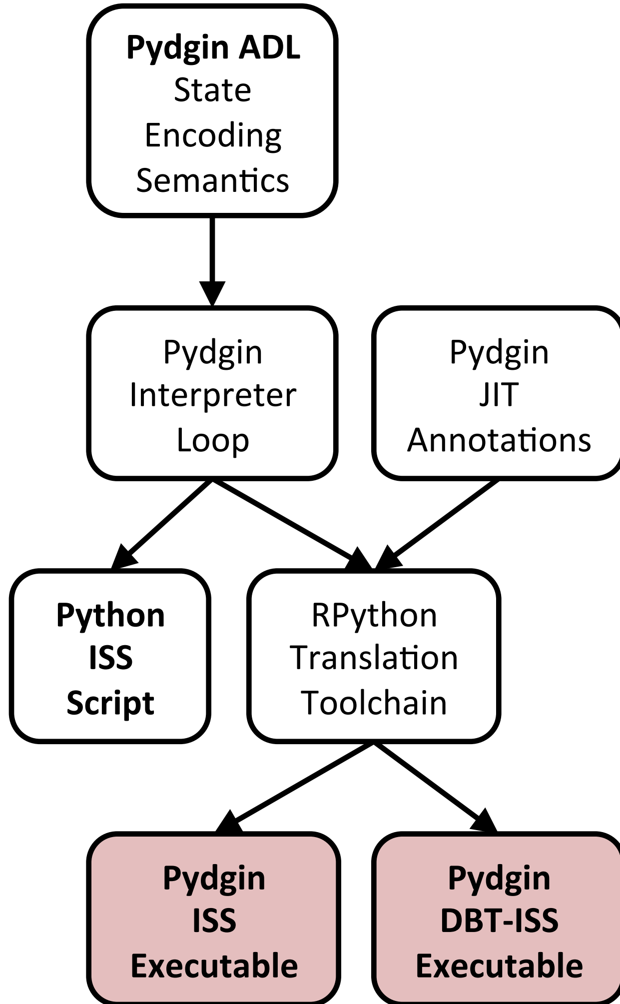
Pydgin ISS Evaluation



Two ISSs implemented in Pydgin

- Simplified-MIPS
- ARMv5

Pydgin ISS Evaluation



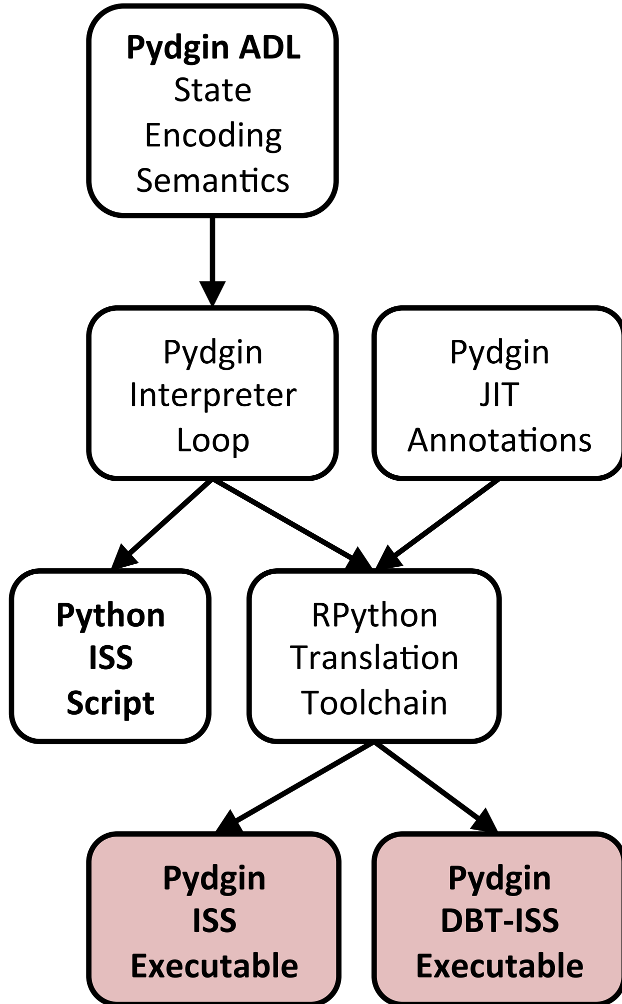
Two ISSs implemented in Pydgin

- Simplified-MIPS
- ARMv5

Simplifications

- GCC cross-compiler using newlib
- emulated system calls
- “bare-metal” system (no OS)

Pydgin ISS Evaluation



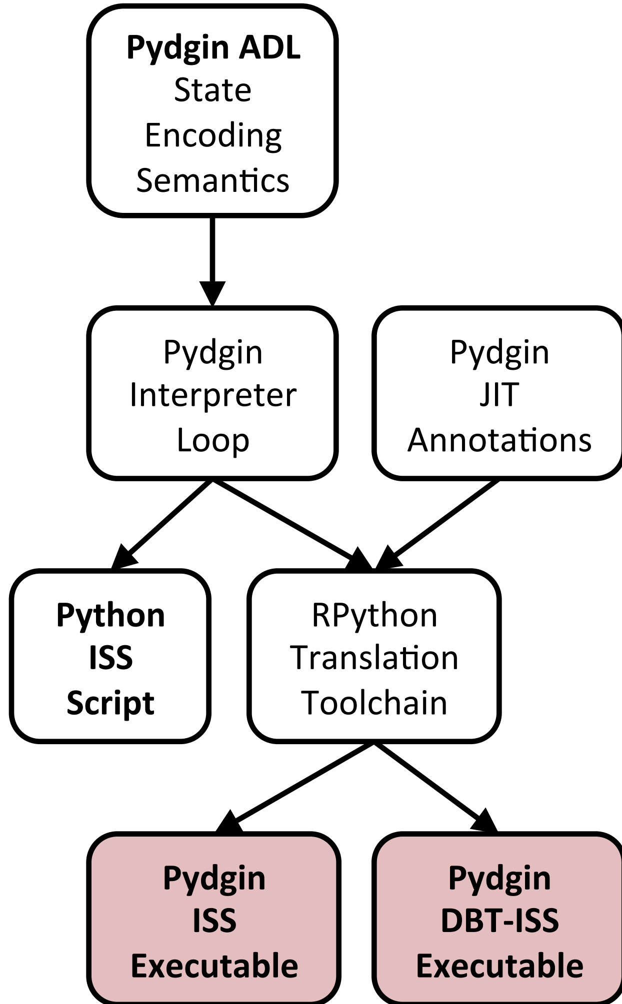
Two ISSs implemented in Pydgin

- Simplified-MIPS: 87-761 MIPS
- ARMv5

Simplifications

- GCC cross-compiler using newlib
- emulated system calls
- "bare-metal" system (no OS)

Pydgin ISS Evaluation: ARMv5



Two ISSs implemented in Pydgin

- Simplified-MIPS: 87-761 MIPS
- ARMv5

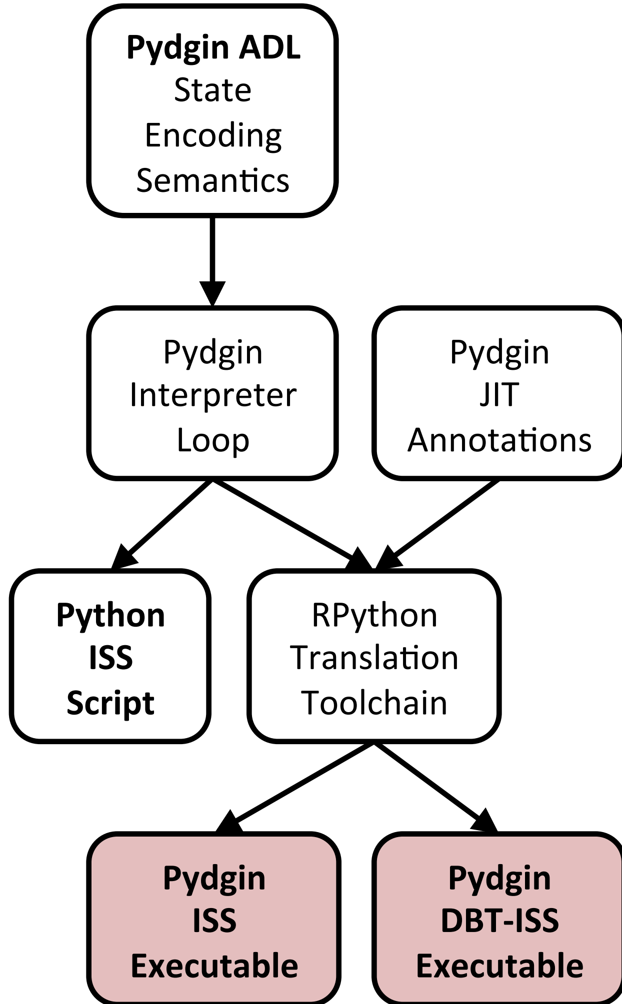
Simplifications

- GCC cross-compiler using newlib
- emulated system calls
- "bare-metal" system (no OS)

ARMv5 ISSs:

- **Interpretive**: gem5-atomic, pydgin-nojit

Pydgin ISS Evaluation: ARMv5



Two ISSs implemented in Pydgin

- Simplified-MIPS: 87-761 MIPS
- ARMv5

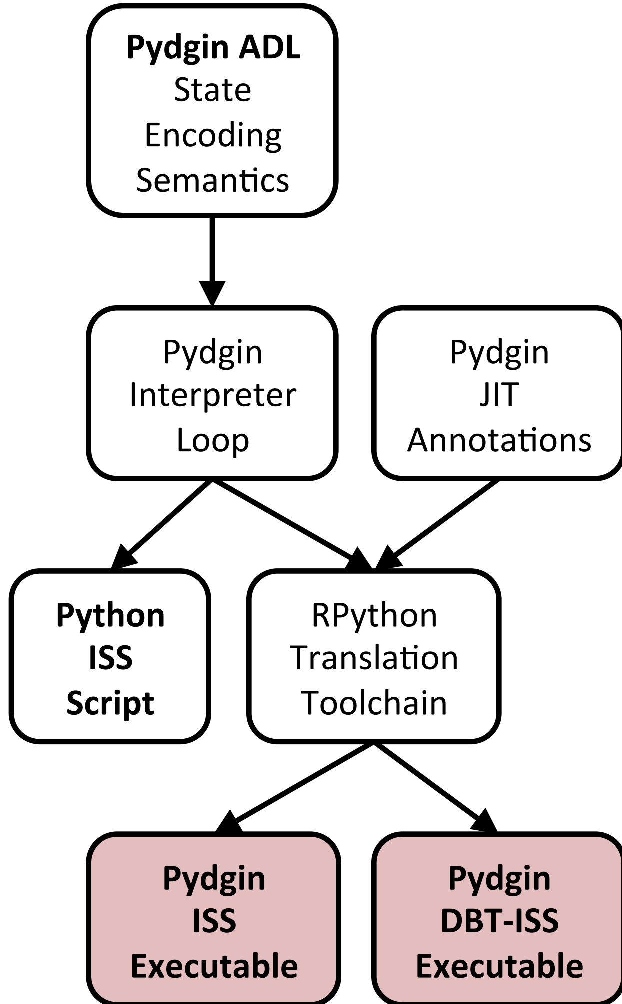
Simplifications

- GCC cross-compiler using newlib
- emulated system calls
- "bare-metal" system (no OS)

ARMv5 ISSs:

- **Interpretive**: gem5-atomic, pydgin-nojit
- **DBT**: simit-jit, pydgin-jit, qemu

Pydgin ISS Evaluation: ARMv5



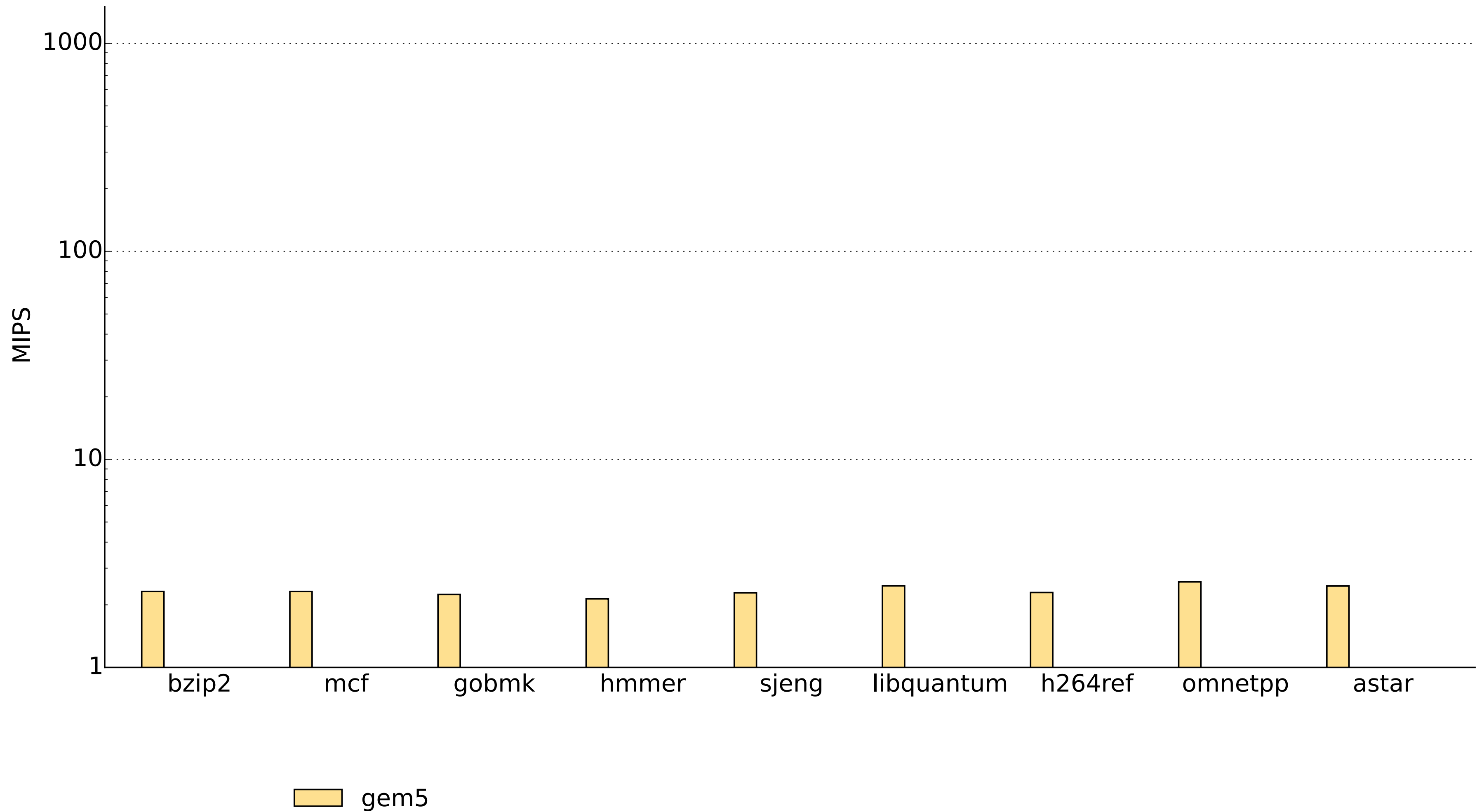
Two ISSs implemented in Pydgin
- Simplified-MIPS: 87-761 MIPS
- ARMv5

Simplifications
- GCC cross-compiler using newlib
- emulated system calls
- "bare-metal" system (no OS)

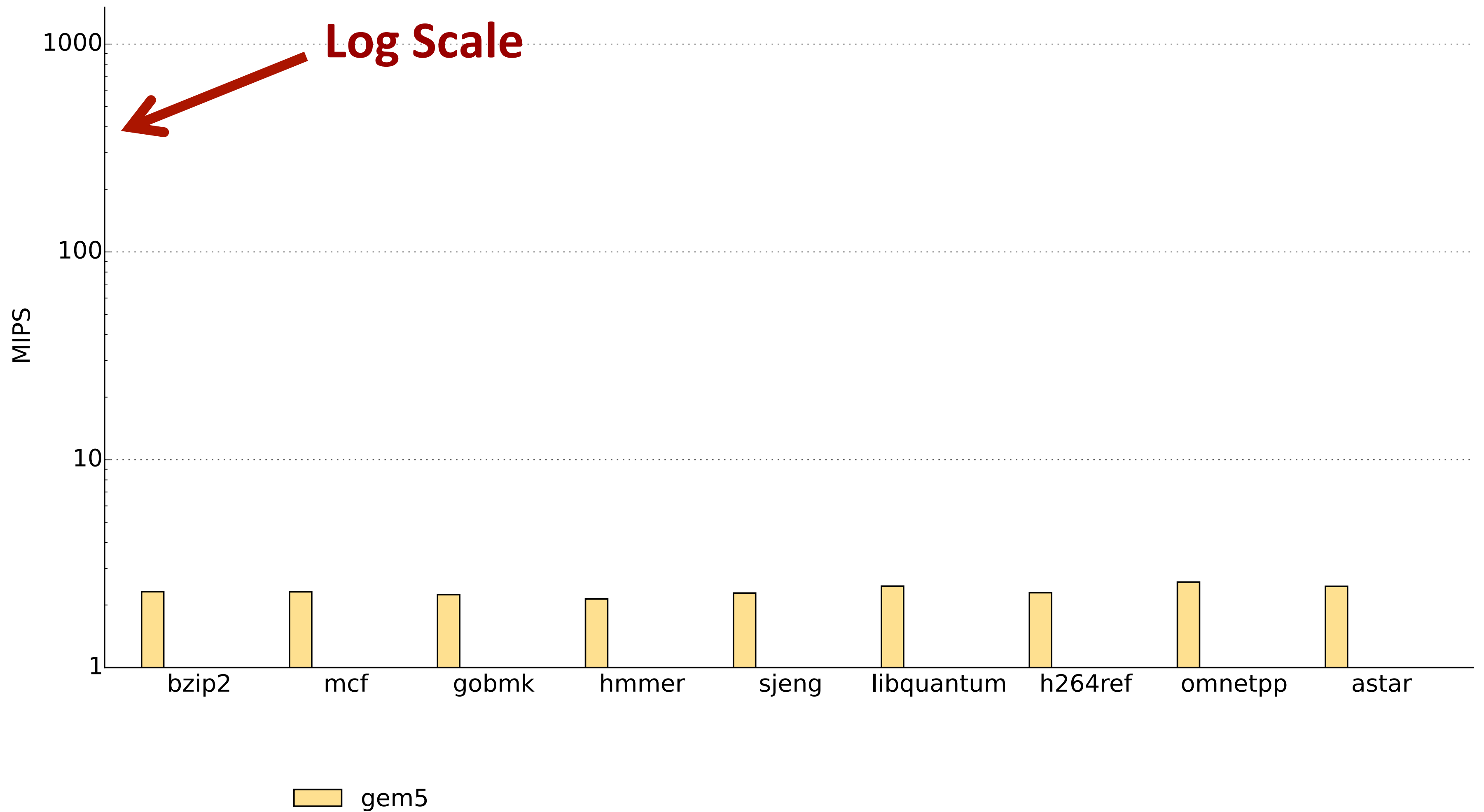
ARMv5 ISSs:
- **Interpretive**: gem5-atomic, pydgin-nojit
- **DBT**: simit-jit, pydgin-jit, qemu*

(* not fully observable)

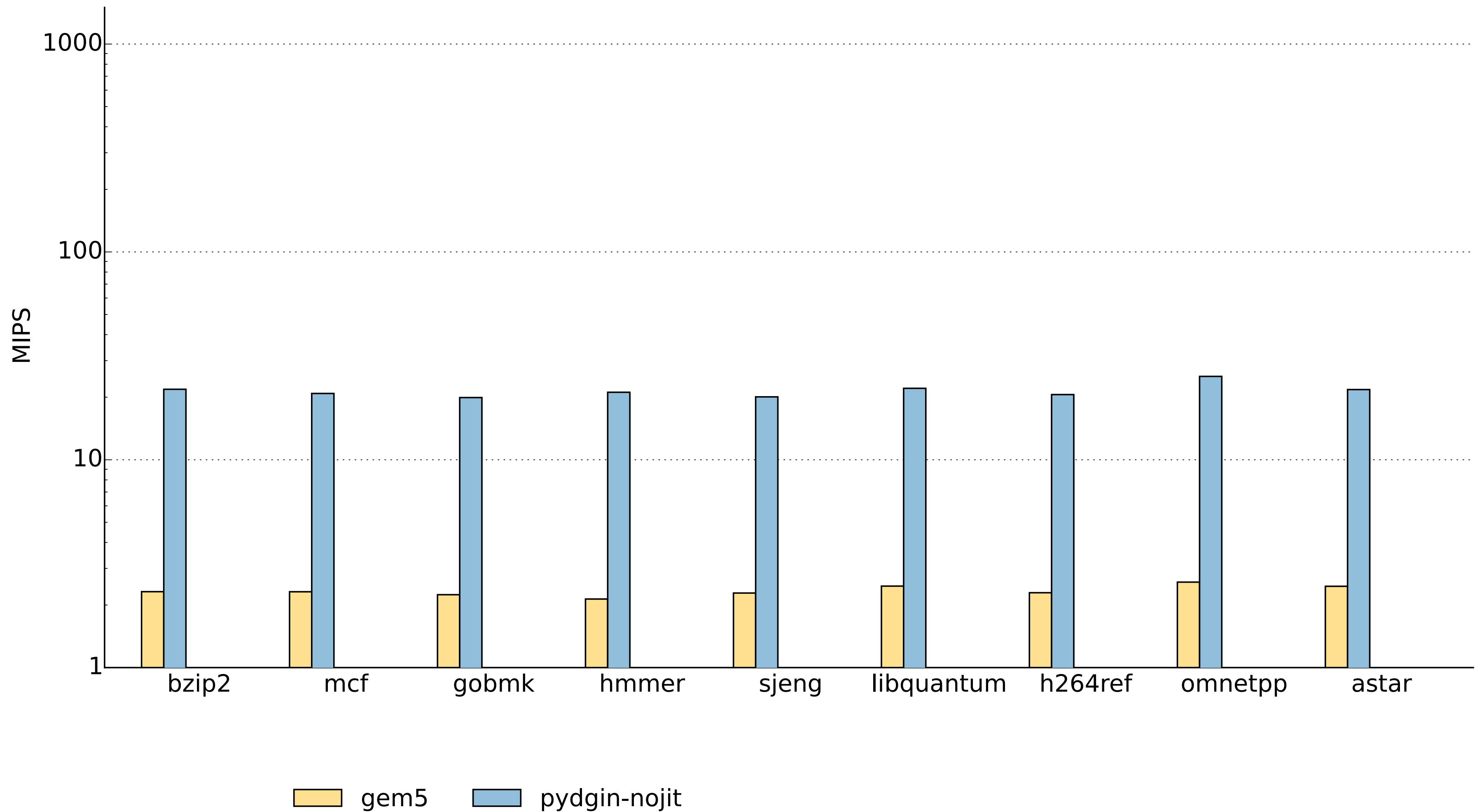
ISS Comparison: ARMv5



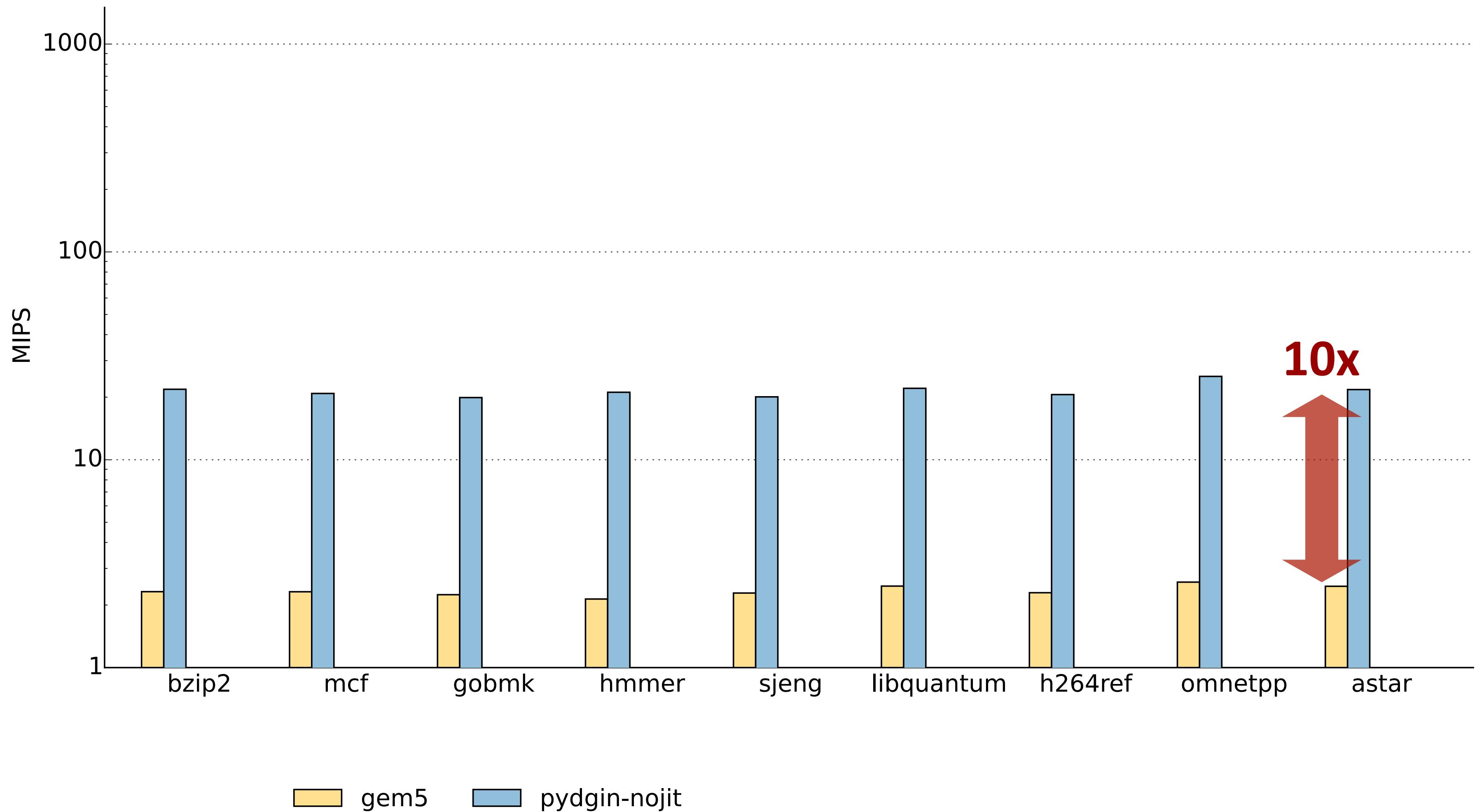
ISS Comparison: ARMv5



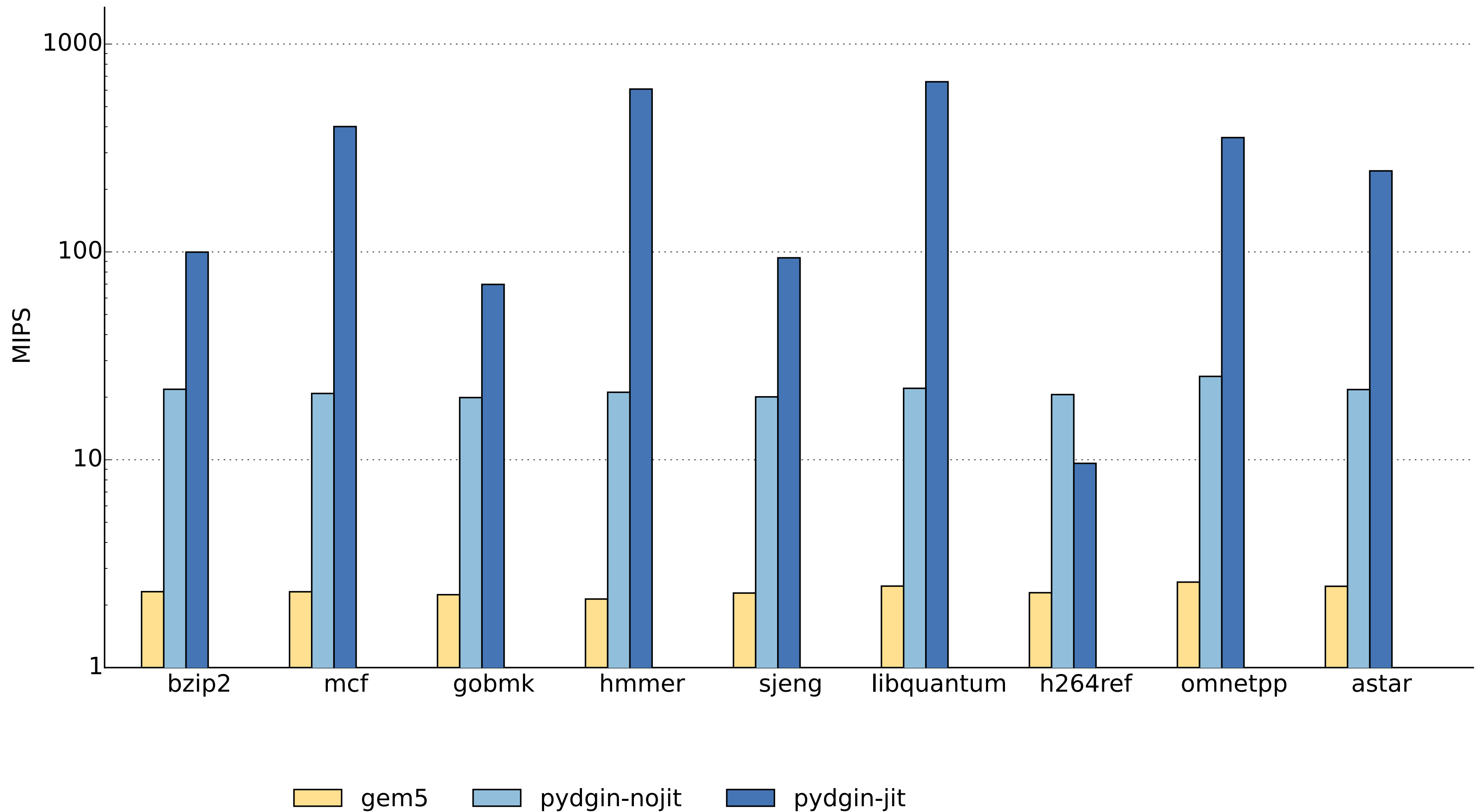
ISS Comparison: ARMv5



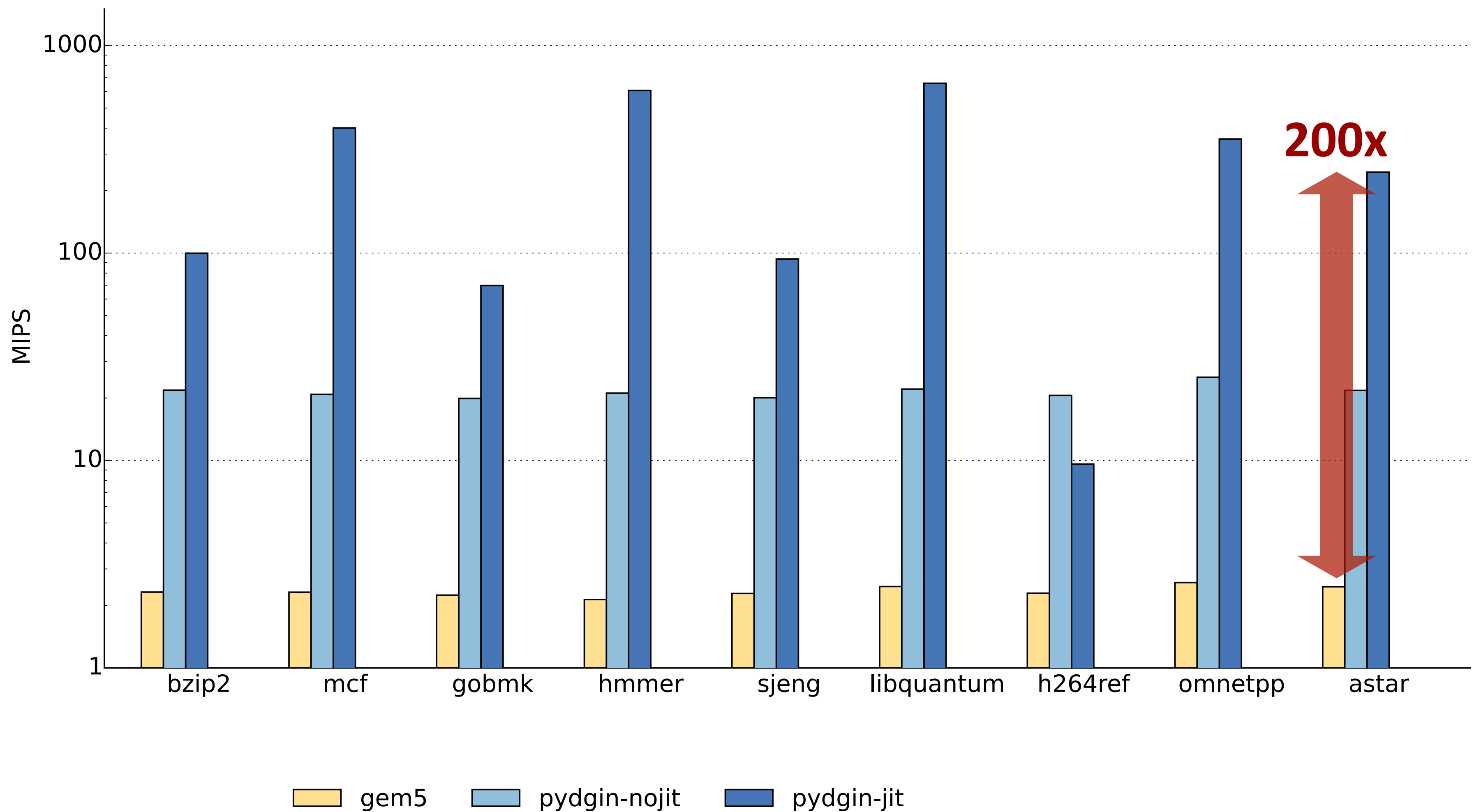
ISS Comparison: ARMv5



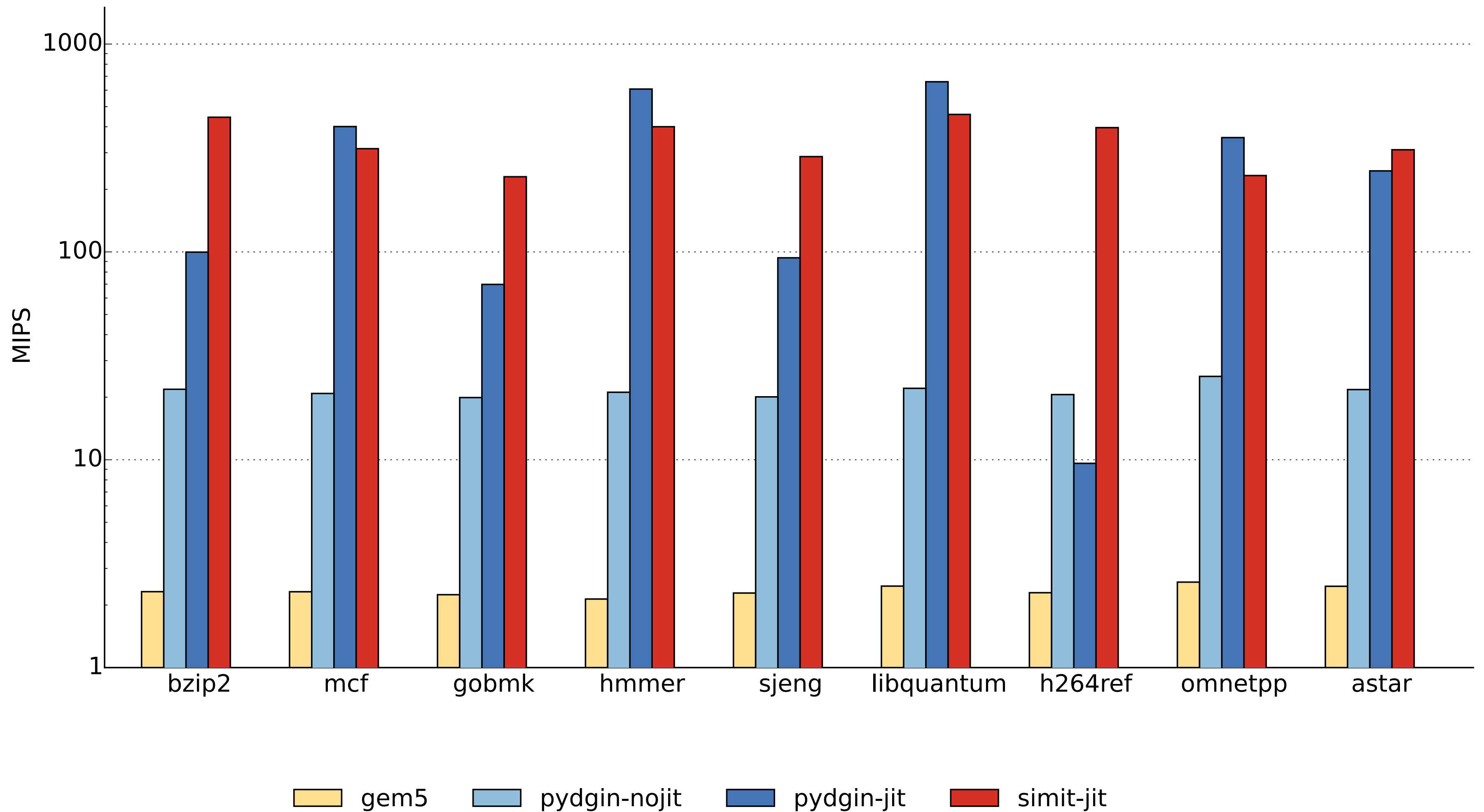
ISS Comparison: ARMv5



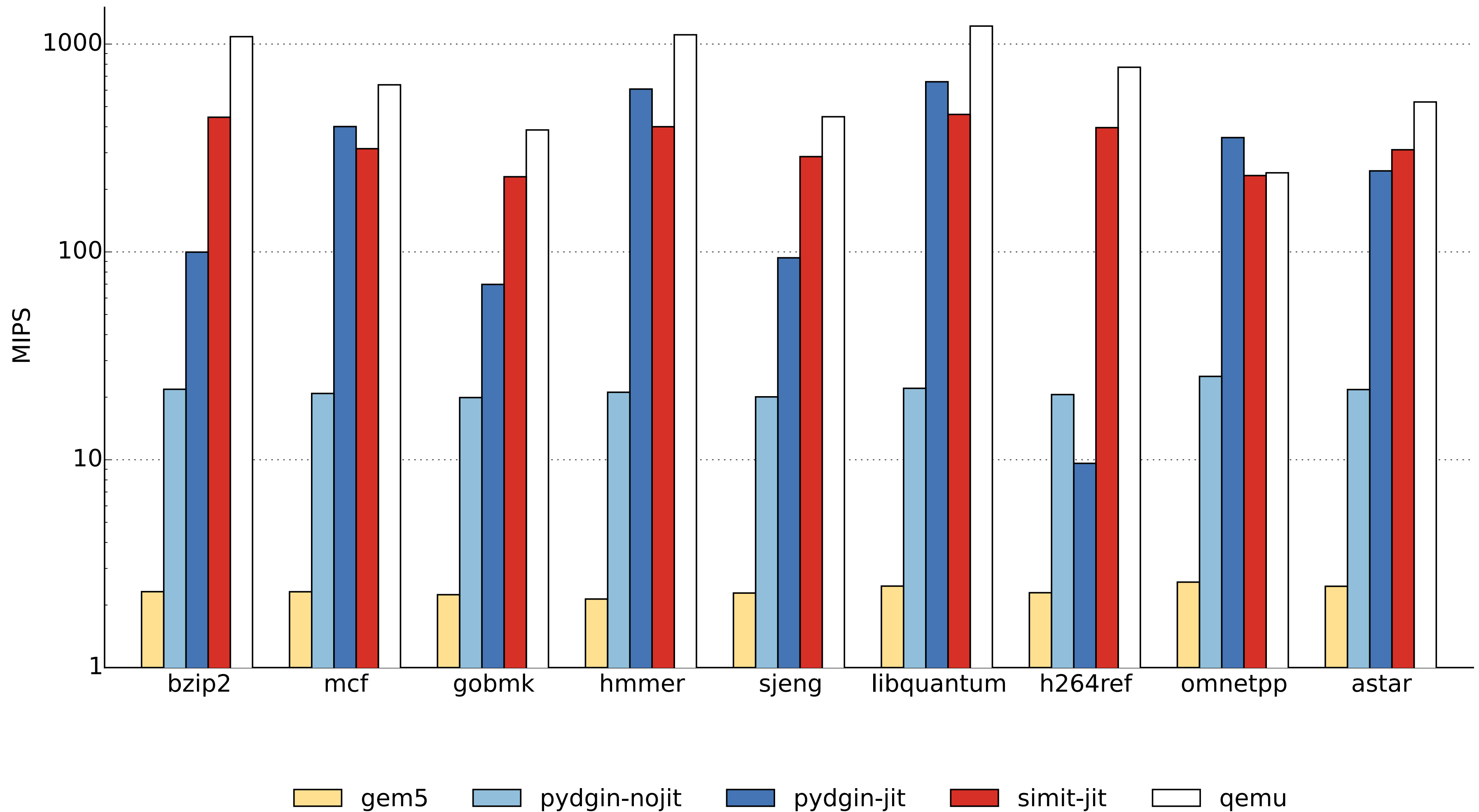
ISS Comparison: ARMv5



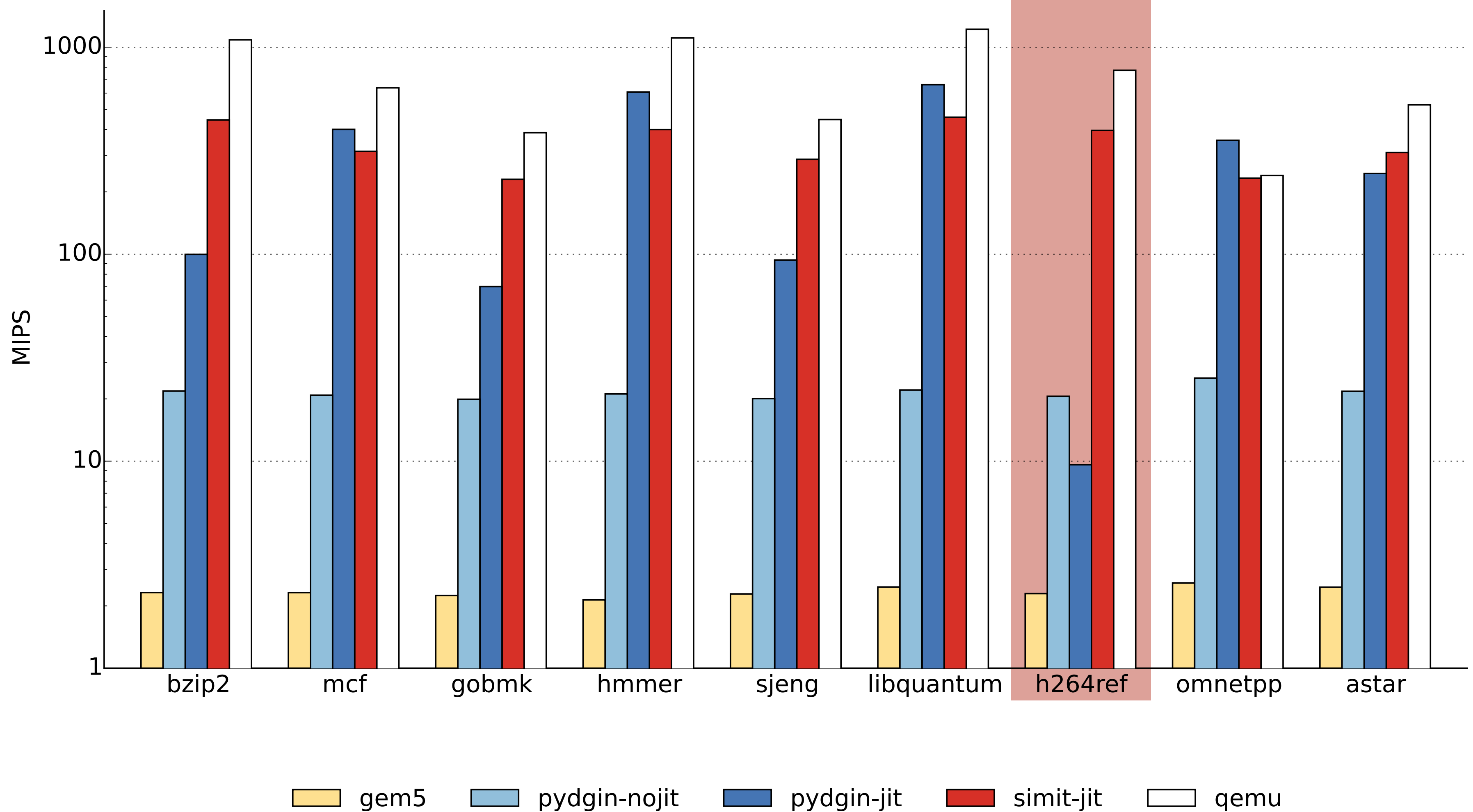
ISS Comparison: ARMv5



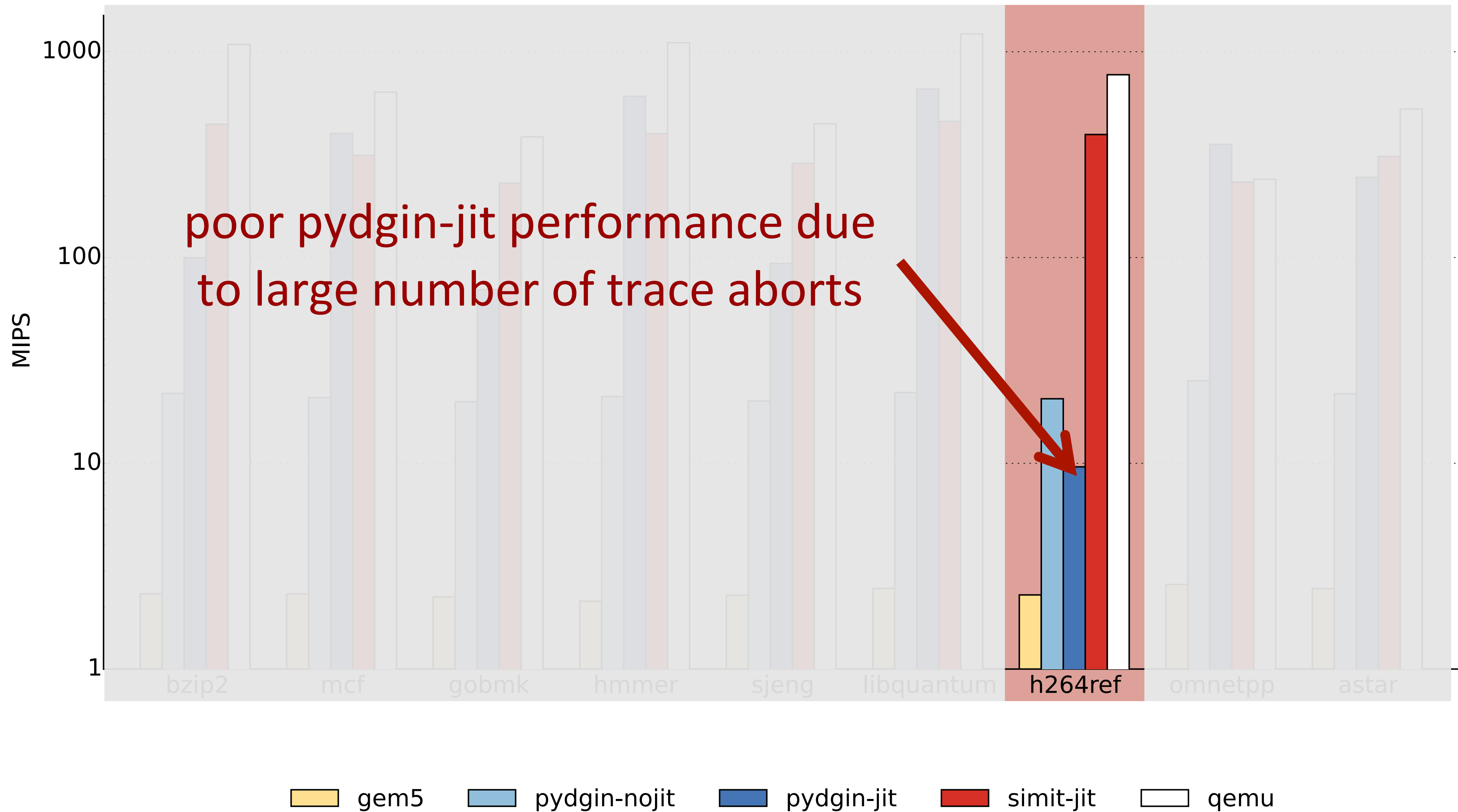
ISS Comparison: ARMv5



ISS Comparison: ARMv5



ISS Comparison: ARMv5

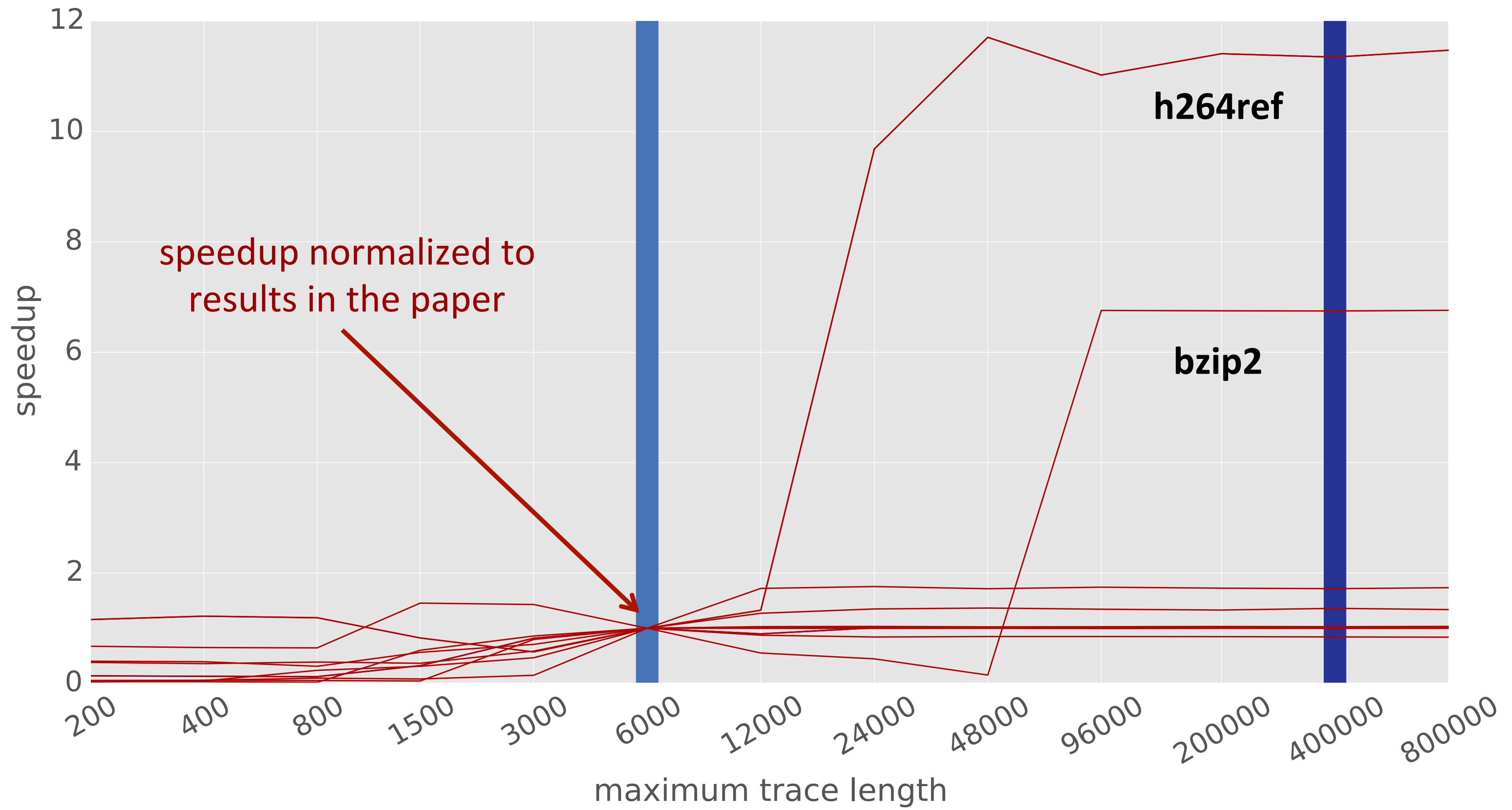


Performance vs. Maximum Trace Length

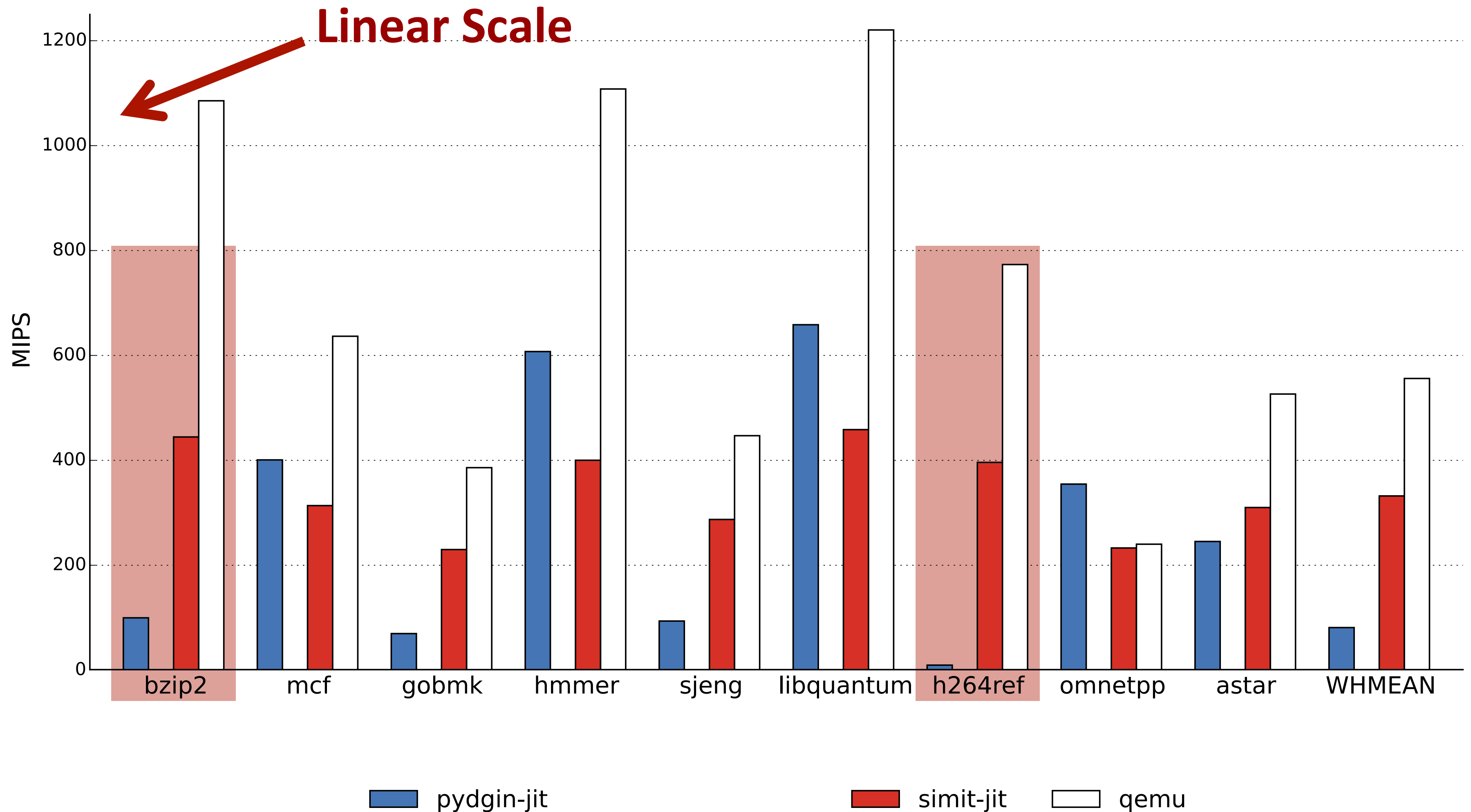
Performance vs. Maximum Trace Length

Performance vs. Maximum Trace Length

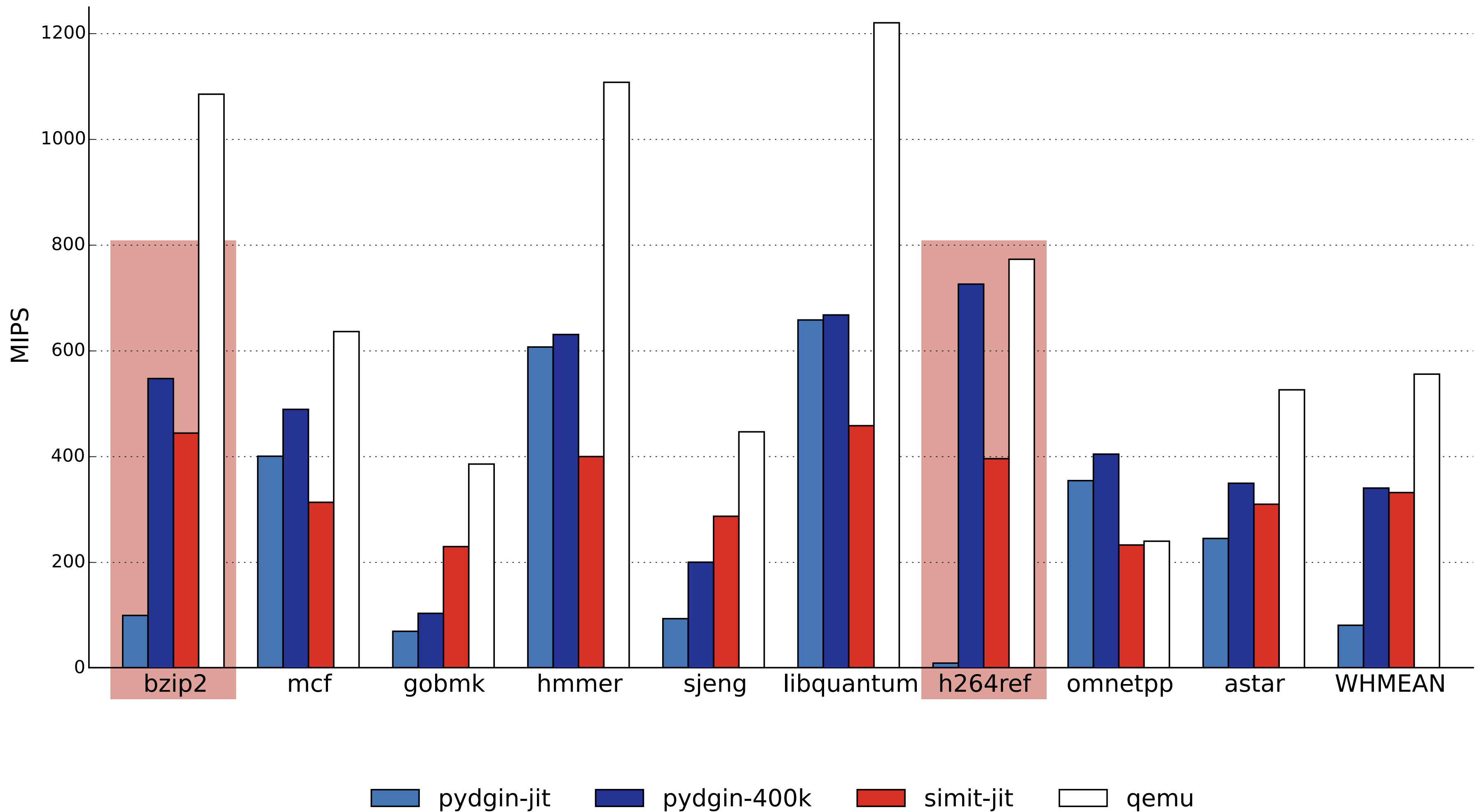
Performance vs. Maximum Trace Length



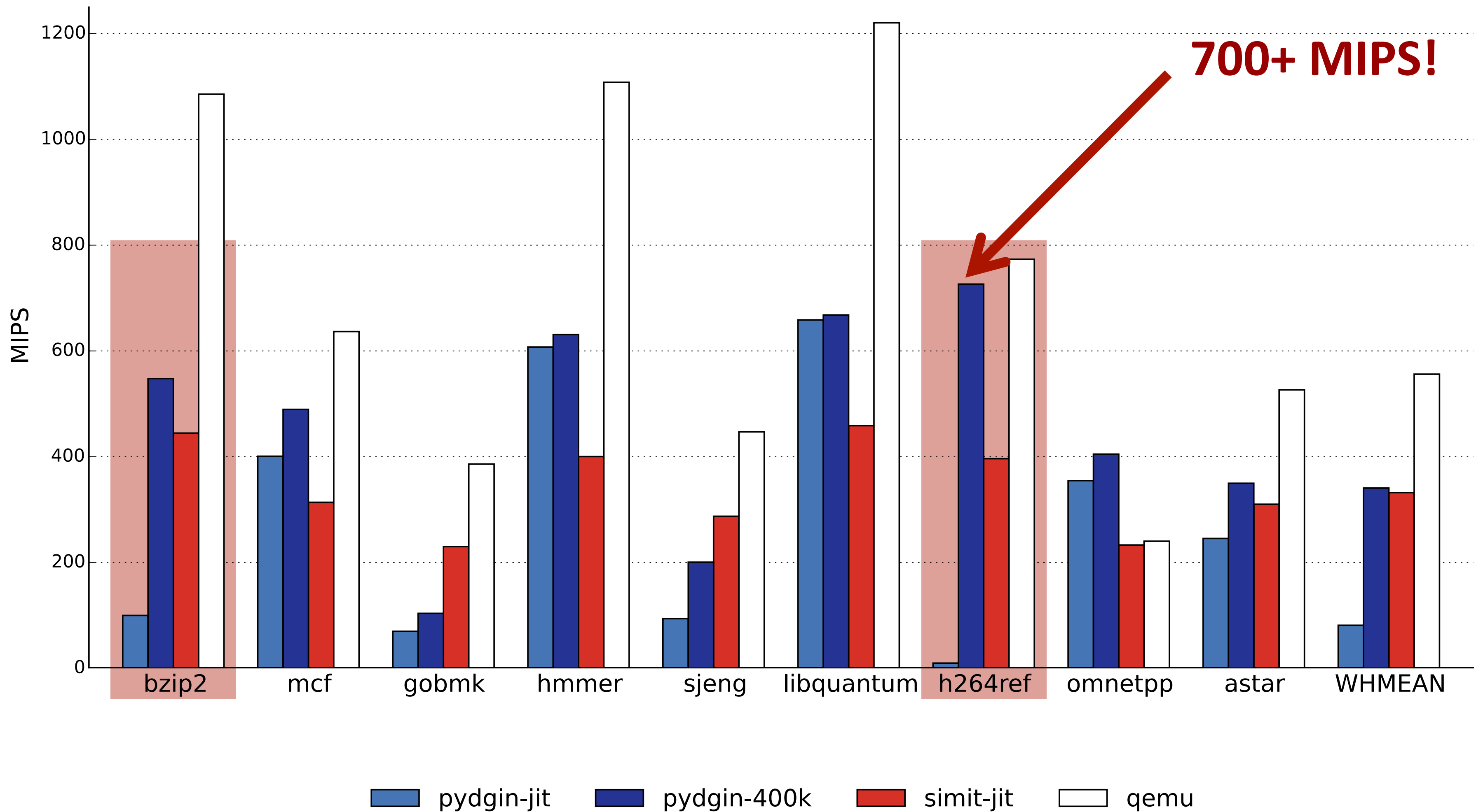
Results: ARM with Longer Traces



Results: ARM with Longer Traces



Results: ARM with Longer Traces

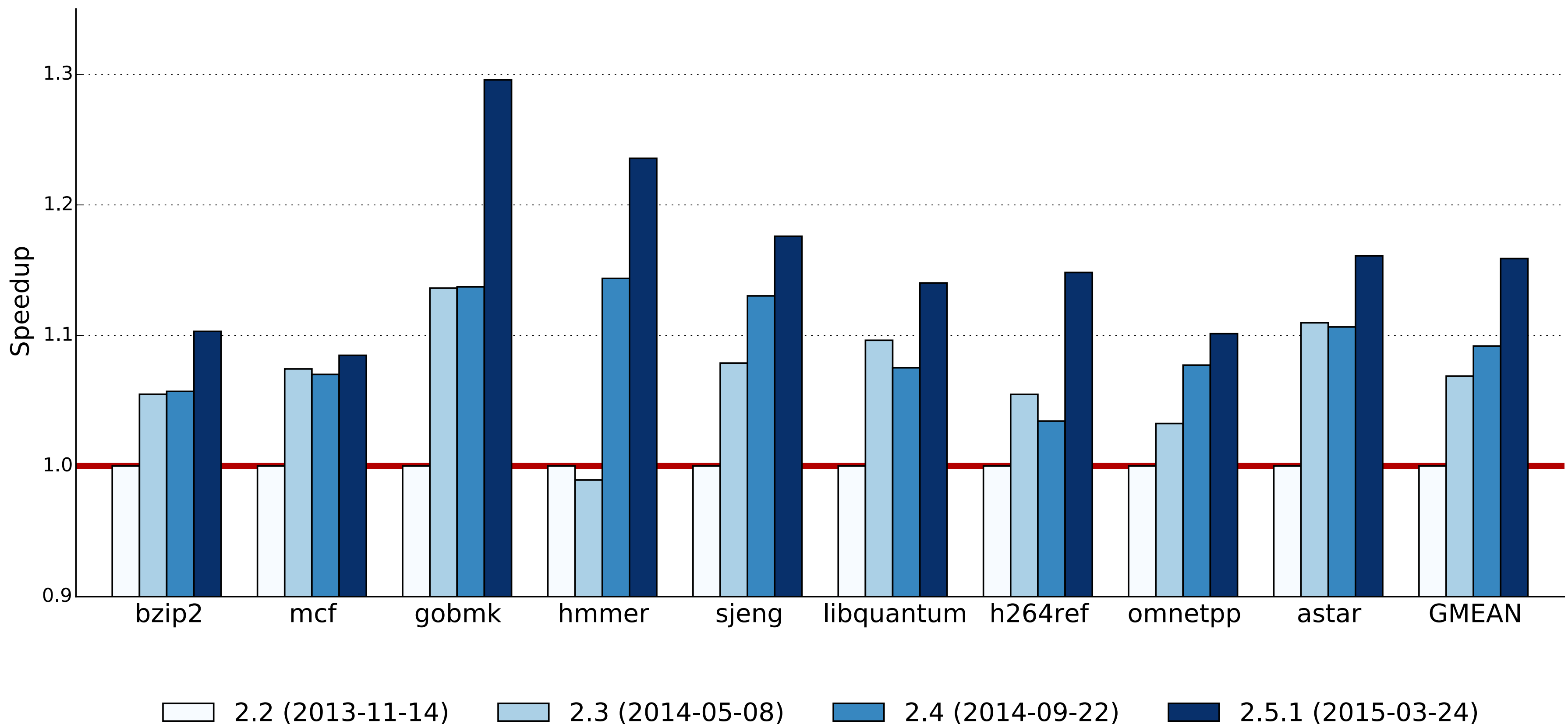


Improving with RPython

Pydgin ISSs benefit from performance improvements as newer versions of the RPython Translation Toolchain are released.

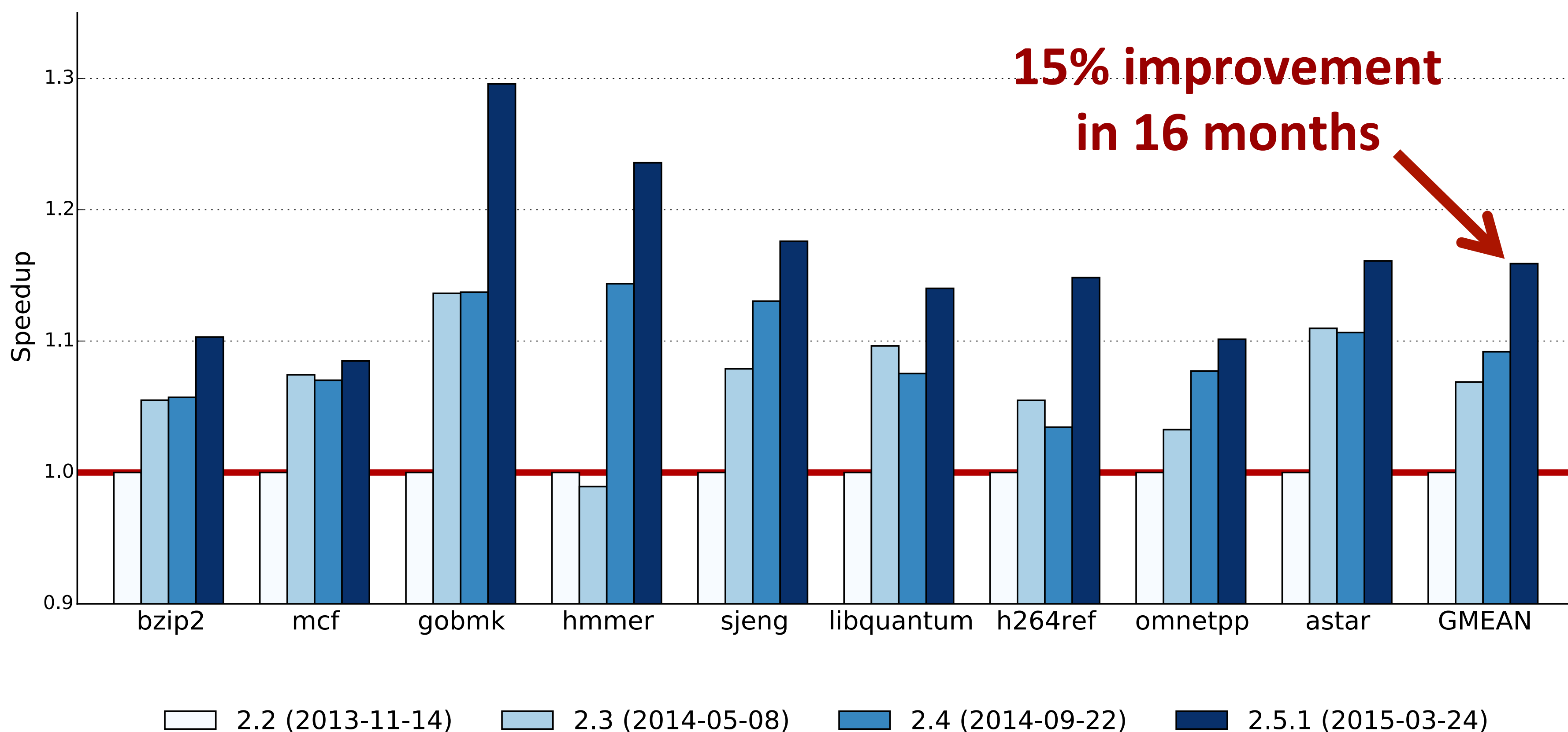
Improving with RPython

Pydgin ISSs benefit from performance improvements as newer versions of the RPython Translation Toolchain are released.



Improving with RPython

Pydgin ISSs benefit from performance improvements as newer versions of the RPython Translation Toolchain are released.



Conclusions

Pydgin provides a succinct, embedded-DSL within Python for rapid prototyping of ISAs for next-generation hardware.

Pydgin leverages the RPython translation toolchain to convert these descriptions into high-performance, JIT-enabled ISSs.

Conclusions

Pydgin provides a succinct, embedded-DSL within Python for rapid prototyping of ISAs for next-generation hardware.

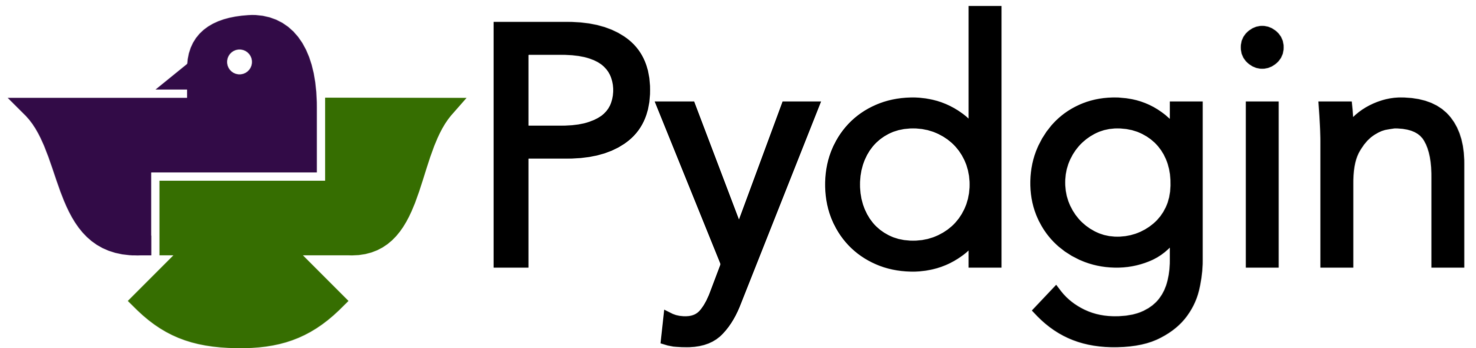
Pydgin leverages the RPython translation toolchain to convert these descriptions into high-performance, JIT-enabled ISSs.

Many opportunities for future improvements:

- more features and ISA implementations
- performance optimizations

Conclusion

Pydgin is a productive, **open-source** Python framework for creating fast instruction set simulators.



https://github.com/cornell-brg/pydgin

Thank you to our sponsors for their support:
NSF, DARPA, and donations from Intel Corporation and Synopsys, Inc.