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

Derek Lockhart, Berkin Ilbeyi, and Christopher Batten
Motivation

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

```
armv5_binary  Instruction-Set Simulator
```

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

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

- Software Development
- Instruction-Set Simulator
- Hardware Design
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

Architectural Description Language

Performance

Instruction Set Interpreter in C with DBT

Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers
Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers
Productivity \[\rightarrow\] Performance

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

Architectural Description Language

[SimIt-ARM2006] [Wagstaff2013]

Instruction Set Interpreter in C with DBT

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

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

Productivity  \(\rightarrow\) Performance

Architectural Description Language

[SimIt-ARM2006] [Wagstaff2013]

Instruction Set Interpreter in C with DBT

Dynamic Language Interpreter in C with JIT Compiler
Key Insight:

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

Similar productivity-performance challenges for building high-performance interpreters of dynamic languages. (e.g. JavaScript, Python)
Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers
Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers

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

Meta-Tracing JIT: makes JIT generation generic across languages
Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers
Pydgin

- 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 ADL
State
Encoding
Semantics

Pydgin Interpreter
Loop

Pydgin JIT
Annotations

Python
ISS
Script

RPython
Translation
Toolchain

Pydgin
ISS
Executable

Pydgin
DBT-ISS
Executable
Pydgin Framework

Pydgin ADL
State
Encoding
Semantics

Pydgin Interpreter Loop
Pydgin JIT Annotations

Python ISS Script
RPython Translation Toolchain

Pydgin ISS Executable
Pydgin DBT-ISS Executable
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

```python
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

```python
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

        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', 'xxxx0000000xxxxxxxxxxxxxxx1001xxxx'],
    ['umull', 'xxxx000100xxxxxxxxxxxxxxx1001xxxx'],
    ['adc', 'xxxx00x0101xxxxxxxxxxxxxxx'],
    ['and', 'xxxx00x0000xxxxxxxxxxxxxxx'],
    ['b', 'xxxx1010xxxxxxxxxxxxxxxxxxx'],
    ['bl', 'xxxx1011xxxxxxxxxxxxxxxxxxx'],
    ['bic', 'xxxx00x1110xxxxxxxxxxxxxxx'],
    ['bkpt', '111000010010xxxxxxxxxxxx0111xxxx'],
    ['teq', 'xxxx00x10011xxxxxxxxxxxxxxx'],
    ['tst', 'xxxx00x10001xxxxxxxxxxxxxxx'],
]
```

Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers
Pydgin ADL: ARMv5 Encodings

encodings = [
    ['nop', '00000000000000000000000000000000'],
    ['mul', 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx1001xxxx'],
    ['umull', 'xxxxxxxxxxx0100xxxxxxxxxxxxxxxxxxxx1001xxxx'],
    ['adc', 'xxxxxxxxxx0101xxxxxxxxxxxxxxxxxxxx1001xxxx'],
    ['and', 'xxxxxxxxxxxx0000xxxxxxxxxxxxxxxxxxxx1001xxxx'],
    ['b', 'xxxxxxxxxxxx1010xxxxxxxxxxxxxxxxxx1001xxxx'],
    ['bl', 'xxxxxxxxxxxx1011xxxxxxxxxxxxxxxxxx1001xxxx'],
    ['bic', 'xxxxxxxxxxxx0000xxxxxxxxxxxxxxxxxxxx1001xxxx'],
    ['bkpt', '111000010010xxxxxxxxxxxx0111xxxx'],
    ['teq', 'xxxxxxxxxxxx10011xxxxxxxxxxxxxxxxxxxx'],
    ['tst', 'xxxxxxxxxxxx10001xxxxxxxxxxxxxxxxxxxx'],
    # ...
    ['teq', 'xxxxxxxxxxxx10011xxxxxxxxxxxxxxxxxxxx'],
    ['tst', 'xxxxxxxxxxxx10001xxxxxxxxxxxxxxxxxxxx']
]
Pydgin ADL: ARMv5 Encodings

encodings = [
    ['nop', '00000000000000000000000000000000'],
    ['mul', 'xxxx000000xxxxxxxxxxxxxxxxxxxx1001xxxx'],
    ['umull', 'xxxx000100xxxxxxxxxxxxxxxxxxxx1001xxxx'],
    ['adc', 'xxxx00x0101xxxxxxxxxxxxxxxxxxxx'],
    ['and', 'xxxx00x0000xxxxxxxxxxxxxxxxxxxx'],
    ['b', 'xxxx1010xxxxxxxxxxxxxxxxxxxx'],
    ['bl', 'xxxx1011xxxxxxxxxxxxxxxxxxxx'],
    ['bic', 'xxxx00x1110xxxxxxxxxxxxxxxxxxxx'],
    ['bkpt', '111000010010xxxxxxxxxxxx0111xxxx'],
    # ...
    ['teq', 'xxxx00x10011xxxxxxxxxxxxxxxxxxxx'],
    ['tst', 'xxxx00x10001xxxxxxxxxxxxxxxxxxxx']
]
Pydgin ADL: ARMv5 Encodings

```
encodings = [
    ['nop', '00000000000000000000000000000000'],
    ['mul', 'xxxx000000xxxxxxxxxxxxxxxxxxx1001xxxx'],
    ['umull', 'xxxx000100xxxxxxxxxxxxxxxxx1001xxxx'],
    ['adc', 'xxxx00x0101xxxxxxxxxxxxxxxxxxxx'],
    ['add', 'xxxx00x0100xxxxxxxxxxxxxxxxxxxx'],
    ['and', 'xxxx00x0000xxxxxxxxxxxxxxxxxxxx'],
    ['b', 'xxxx1010xxxxxxxxxxxxxxxxxxxx'],
    ['bl', 'xxxx1011xxxxxxxxxxxxxxxxxxxx'],
    ['bic', 'xxxx00x1110xxxxxxxxxxxxxxxxxxxx'],
    ['bkpt', '111000010010xxxxxxxxxxxx0111xxxx'],
    ['teq', 'xxxx00x10011xxxxxxxxxxxxxxxxxxxx'],
    ['tst', 'xxxx00x10001xxxxxxxxxxxxxxxxxxxx'],
]  # ...
```
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
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

```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
```

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

Pydgin ADL
  State
  Encoding
  Semantics

Pydgin Interpreter Loop

Pydgin JIT Annotations

Python ISS Script

RPython Translation Toolchain

Pydgin ISS Executable

Pydgin DBT-ISS Executable
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
```
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

```python
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
```
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

- Pydgin ADL
  - State
  - Encoding
  - Semantics

- Pydgin Interpreter Loop
  - Pydgin JIT Annotations

- Python ISS Script
  - RPython Translation Toolchain
  - Pydgin ISS Executable
  - Pydgin DBT-ISS Executable
The RPython Translation Toolchain

Pydgin ADL
- State
- Encoding
- Semantics

Pydgin Interpreter Loop

Pydgin

Python ISS
- Script

RPython Translation Toolchain

RPython Source
- Type Inference
  - Optimization
    - Code Generation
      - Compilation
        - Compiled Interpreter

Pydgin ISS
- Executable

Pydgin DBT-ISS
- Executable

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

Pydgin ADL
State
Encoding
Semantics

Pydgin Interpreter
Loop

Pydgin
JIT
Annotations

Python
ISS
Script

RPython
Translation
Toolchain

RPython Source
Type Inference

Optimization

Code Generation

Compilation

Compiled Interpreter

> ./pydgin-nojit arm_binary
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

Pydgin ADL
State
Encoding
Semantics

Pydgin Interpreter
Loop

Pydgin
JIT
Annotations

Pydgin
ISS
Script

RPython
Translation
Toolchain

Python
ISS
Executable

Pydgin
ISS
Executable

Pydgin
DBT-ISS
Executable

RPython Source

Type Inference

Optimization

JIT Generator

Code Generation

Compilation

Compiled Interpreter with JIT
The RPython Translation Toolchain

Pydgin ADL
- State
- Encoding
- Semantics

Pydgin
- Interpreter
- Loop

Pydgin
- JIT
- Annotations

RPython
- ISS
- Translation
- Toolchain

Python
- ISS
- Script

Pydgin
- ISS
- Executable

Pydgin
- DBT-ISS
- Executable

RPython Source
- Type Inference
- Optimization
- JIT Generator

Code Generation

Compilation

Compiled Interpreter with JIT

> ./pydgin-jit arm_binary

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

Pydgin ADL
  State
  Encoding
  Semantics

Pydgin Interpreter Loop

Pydgin JIT Annotations

Python ISS Script

RPython Translation Toolchain

Pydgin ISS Executable

Pydgin DBT-ISS Executable
Creating a competitive JIT requires additional RPython JIT hints:

- Pydgin ADL
  - State
  - Encoding
  - Semantics

- Pydgin Interpreter Loop

- Pydgin JIT Annotations

- Python ISS Script

- RPython Translation Toolchain

- Pydgin ISS Executable

- Pydgin DBT-ISS Executable
JIT Annotations

Creating a competitive JIT requires additional RPython JIT hints:

+ Minimal JIT Annotations

SPECINT2006
JIT Annotations

Creating a competitive JIT requires additional RPython JIT hints:

+ Minimal JIT Annotations
+ Elidable Instruction Fetch

SPECINT2006

Speedup

geomean
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

SPECINT2006

Speedup

geomean
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
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
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

Log Scale

MIPS

bzip2  mcf  gobmk  hmmr  sjeng  libquantum  h264ref  omnetpp  astrar

gem5
ISS Comparison: ARMv5

MIPS

bzip2    mcf     gobmk    hmmer    sjeng    libquantum    h264ref    omnetpp    astrar

gem5      pydgin-nojit
ISS Comparison: ARMv5

Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers
ISS Comparison: ARMv5

![Graph showing performance comparison between gem5, pydgin-nojit, and pydgin-jit for various benchmarks.](image)
ISS Comparison: ARMv5

Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers
ISS Comparison: ARMv5

![Graph showing comparison of different simulators for various benchmarks.](image)
ISS Comparison: ARMv5

Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers
ISS Comparison: ARMv5

Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers
ISS Comparison: ARMv5

poor pydgin-jit performance due to large number of trace aborts
Performance vs. Maximum Trace Length

Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers
Performance vs. Maximum Trace Length

speedup normalized to results in the paper

speedup

maximum trace length

Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers
Performance vs. Maximum Trace Length

speedup normalized to results in the paper
Performance vs. Maximum Trace Length

speedup normalized to results in the paper

- h264ref
- bzip2
Results: ARM with Longer Traces

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

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

700+ MIPS!
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.

15% improvement in 16 months
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.