Appears in the Proceedings of the Int'l Symp. on Performance Analysis of Systems and Software (ISPASS-2015), March 2015

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

Derek Lockhart, Berkin Ilbeyi, and Christopher Batten

School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{dml257,bi45,cbatten}@cornell.edu

***Abstract*—Instruction set simulators (ISSs) remain an essential tool for the rapid exploration and evaluation of instruction set extensions in both academia and industry. Due to their importance in both hardware and software design, modern ISSs must balance a tension between developer productivity and high-performance simulation. Productivity requirements have led to "ADL-driven" toolflows that automatically generate ISSs from high-level architectural description languages (ADLs). Meanwhile, performance requirements have prompted ISSs to incorporate increasingly complicated dynamic binary translation (DBT) techniques. Construction of frameworks capable of providing both the productivity benefits of ADL-generated simulators and the performance benefits of DBT remains a significant challenge.**

We introduce Pydgin, a new approach to ISS construction that addresses the multiple challenges of designing, implementing, and maintaining ADL-generated DBT-ISSs. Pydgin uses a Python-based, embedded-ADL to succinctly describe instruction behavior as directly executable "pseudocode". These Pydgin ADL descriptions are used to automatically generate high-performance DBT-ISSs by creatively adapting an existing meta-tracing JIT compilation framework designed for general-purpose dynamic programming languages. We demonstrate the capabilities of Pydgin by implementing ISSs for two instruction sets and show that Pydgin provides concise, flexible ISA descriptions while also generating simulators with performance comparable to hand-coded DBT-ISSs.

I. Introduction

Recent challenges in CMOS technology scaling have motivated an increasingly fluid boundary between hardware and software. Examples include new instructions for managing fine-grain parallelism, new programmable data-parallel engines, programmable accelerators based on reconfigurable coarse-grain arrays, domain-specific co-processors, and application-specific instructions. This trend towards heterogeneous hardware/software abstractions combined with complex design targets is placing increasing importance on highly productive and high-performance instruction set simulators (ISSs).

Unfortunately, meeting the multitude of design requirements for a modern ISS (observability, retargetability, extensibility, support for self-modifying code, etc.) while also providing productivity and high performance has led to considerable ISS design complexity. Highly productive ISSs have adopted architecture description languages (ADLs) as a means to enable abstract specification of instruction semantics and simplify the addition of new instruction set features. The ADLs in these frameworks are domain specific languages constructed to be sufficiently expressive for describing traditional architectures, yet restrictive enough for efficient simulation (e.g., ArchC [3, 50], LISA [33, 55], LIS [34], MADL [43, 44], SimIt-ARM ADL [21, 40]). In addition, high-performance ISSs use dynamic binary translation (DBT) to discover frequently executed blocks of target instructions and convert these blocks into optimized sequences of host instructions. DBT-ISSs often require a deep understanding of the target instruction set in order to enable fast and efficient translation. However, promising recent work has demonstrated sophisticated frameworks that can automatically generate DBT-ISSs from ADLs [35, 42, 56].

Meanwhile, designers working on interpreters for general-purpose dynamic programming languages (e.g., Javascript, Python, Ruby, Lua, Scheme) face similar challenges balancing productivity of interpreter developers with performance of the interpreter itself. The highest performance interpreters use just-in-time (JIT) trace- or method-based compilation techniques. As the sophistication of these techniques have grown so has the complexity of interpreter codebases. For example, the WebKit Javascript engine currently consists of four distinct tiers of JIT compilers, each designed to provide greater amounts of optimization for frequently visited code regions [37]. In light of these challenges, one promising approach introduced by the PyPy project uses *meta-tracing* to greatly simplify the design of high-performance interpreters for dynamic languages. PyPy's meta-tracing toolchain takes traditional interpreters implemented in RPython, a restricted subset of Python, and automatically translates them into optimized, tracing-JIT compilers [2, 9, 10, 36, 39]. The RPython translation toolchain has been previously used to rapidly develop high-performance JIT-enabled interpreters for a variety of different languages [11–13, 24, 51, 52, 54]. We make the key observation that similarities between ISSs and interpreters for dynamic programming languages suggest that the RPython translation toolchain might enable similar productivity and performance benefits when applied to instruction set simulator design.

This paper introduces Pydgin[1], a new approach to ISS design that combines an embedded-ADL with automatically-generated meta-tracing JIT interpreters to close the productivity-performance gap for future ISA design. The Pydgin library provides an embedded-ADL within RPython for succinctly describing instruction semantics, and also provides a modular instruction set interpreter that leverages these user-defined instruction definitions. In addition to mapping closely to the pseudocode-like syntax of ISA manuals, Pydgin instruction descriptions are fully executable within the Python interpreter for rapid code-test-debug during ISA development. We adapt the RPython translation toolchain to take Pydgin ADL descriptions and automatically convert them into high-performance DBT-ISSs. Building the Pydgin framework required approximately three person-months worth of work, but implementing two different instruction sets (a simple MIPS-based instruction set and a more sophisticated ARMv5 instruction set) took just a few weeks and resulted in ISSs capable of executing many of the

[1] Pydgin loosely stands for [Py]thon [D]SL for [G]enerating [In]struction set simulators and is pronounced the same as "pigeon". The name is inspired by the word "pidgin" which is a grammatically simplified form of language and captures the intent of the Pydgin embedded-ADL.

```
jd = JitDriver( greens = ['bytecode', 'pc'],
                reds   = ['regs', 'acc'] )

def interpreter( bytecode ):
  regs = [0]*256 # vm state: 256 registers
  acc  = 0       # vm state: accumulator
  pc   = 0       # vm state: program counter

  while True:
    jd.jit_merge_point( bytecode, pc, regs, acc )
    opcode = ord(bytecode[pc])
    pc += 1

    if opcode == JUMP_IF_ACC:
      target = ord(bytecode[pc])
      pc += 1
      if acc:
        if target < pc:
          jd.can_enter_jit( bytecode, pc, regs, acc )
        pc = target

    elif opcode == MOV_ACC_TO_REG:
      rs = ord(bytecode[pc])
      regs[rs] = acc
      pc += 1

    # ... handle remaining opcodes ...
```

Figure 1. Simple Bytecode Interpreter Written in RPython – `bytecode` is string of bytes encoding instructions that operate on 256 registers and an accumulator. RPython enables succinct interpreter descriptions that can still be automatically translated into C code. Basic annotations (shown in blue) enable automatically generating a meta-tracing JIT compiler. Adapted from [10].

SPEC CINT2006 benchmarks at hundreds of millions of instructions per second.

This paper makes the following three contributions: (1) we describe the Pydgin embedded-ADL for productively specifying instruction set architectures; (2) we describe and quantify the performance impact of specific optimization techniques used to generate high-performance DBT-ISSs from the RPython translation toolchain; and (3) we evaluate the performance of Pydgin DBT-ISSs when running SPEC CINT2006 applications on two distinct ISAs.

II. The RPython Translation Toolchain

The increase in popularity of dynamic programming languages has resulted in a significant interest in high-performance interpreter design. Perhaps the most notable examples include the numerous JIT-optimizing JavaScript interpreters present in modern browsers today. Another example is PyPy, a JIT-optimizing interpreter for the Python programming language. PyPy uses JIT compilation to improve the performance of hot loops, often resulting in considerable speedups over the reference Python interpreter, CPython. The PyPy project has created a unique development approach that utilizes the *RPython translation toolchain* to abstract the process of language interpreter design from low-level implementation details and performance optimizations. The RPython translation toolchain enables developers to describe an interpreter in a restricted subset of Python (called RPython) and then automatically translate this RPython interpreter implementation into a C executable. With the addition of a few basic annotations, the RPython translation toolchain can also automatically insert a tracing-JIT compiler into the generated C-based interpreter. In this section, we briefly describe the RPython translation toolchain, which we leverage

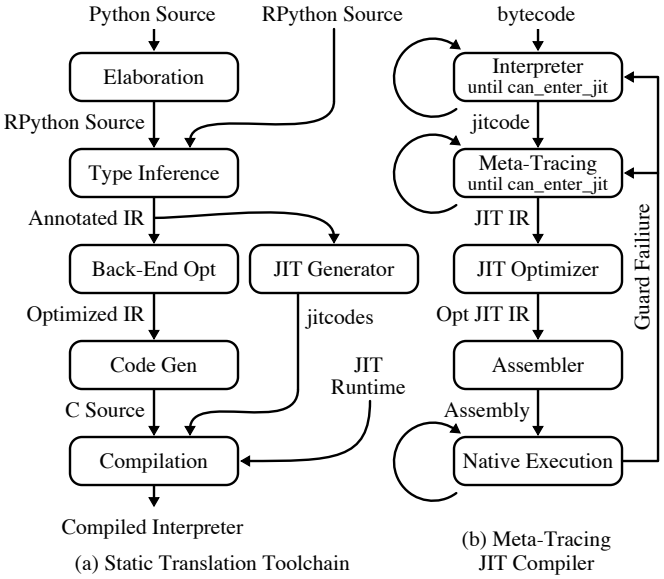

Figure 2. RPython Translation Toolchain – (a) the static translation toolchain converts an RPython interpreter into C code along with a generated JIT compiler; (b) the meta-tracing JIT compiler traces the interpreter (not the application) to eventually generate optimized assembly for native execution.

as the foundation for the Pydgin framework. More detailed information about RPython and the PyPy project in general can be found in [2, 8–10, 36, 39].

Python is a dynamically typed language with typed objects but untyped variable names. RPython is a carefully chosen subset of Python that enables static type inference such that the type of both objects and variable names can be determined at translation time. Even though RPython sacrifices some of Python's dynamic features (e.g., duck typing, monkey patching) it still maintains many of the features that make Python productive (e.g., simple syntax, automatic memory management, large standard library). In addition, RPython supports powerful metaprogramming allowing full-featured Python code to be used to generate RPython code at translation time.

Figure 1 shows a simple bytecode interpreter and illustrates how interpreters written in RPython can be significantly simpler than a comparable interpreter written in C (example adapted from [10]). The example is valid RPython because the type of all variables can be determined at translation time (e.g., `regs`, `acc`, and `pc` are always of type `int`; `bytecode` is always of type `str`). Figure 2(a) shows the RPython translation toolchain. The *elaboration phase* can use full-featured Python code to generate RPython source as long as the interpreter loop only contains valid RPython prior to starting the next phase of translation. The *type inference phase* uses various algorithms to determine high-level type information about each variable (e.g., integers, real numbers, user-defined types) before lowering this type information into an annotated intermediate representation (IR) with specific C datatypes (e.g., `int`, `long`, `double`, `struct`). The *back-end optimization phase* leverages standard optimization passes to inline functions, remove unnecessary dynamic memory allocation, implement exceptions efficiently, and manage garbage collection. The *code generation phase* translates the optimized

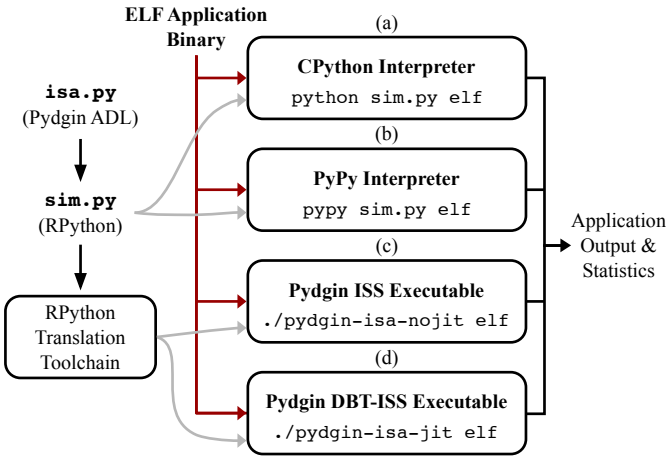

Figure 3. Pydgin Simulation – Pydgin ISA descriptions are imported by the Pydgin simulation driver which defines the top-level interpreter loop. The resulting Pydgin ISS can be executed directly using (a) the reference CPython interpreter or (b) the higher-performance PyPy JIT-optimizing interpreter. Alternatively, the interpreter loop can be passed to the RPython translation toolchain to generate a C-based executable implementing (c) an interpretive ISS or (d) a DBT-ISS.

IR into C source code, before the *compilation phase* generates the C-based interpreter.

The RPython translation toolchain also includes support for automatically generating a tracing JIT compiler to complement the generated C-based interpreter. To achieve this, the RPython toolchain uses a novel *meta-tracing* approach where the JIT compiler does not directly trace the bytecode but instead traces the *interpreter* interpreting the bytecode. While this may initially seem counter-intuitive, meta-tracing JIT compilers are the key to improving productivity and performance. This approach decouples the design of the interpreter, which can be written in a high-level dynamic language such as RPython, from the complexity involved in implementing a tracing JIT compiler for that interpreter. A direct consequence of this separation of concerns is that interpreters for different languages can all leverage the exact same JIT compilation framework as long as these interpreters make careful use of *meta-tracing annotations*.

Figure 1 highlights the most basic meta-tracing annotations required to automatically generate reasonable JIT compilers. The `JitDriver` object instantiated on lines 1–2 informs the JIT of which variables identify the interpreter's position within the target application's bytecode (`greens`), and which variables are not part of the position key (`reds`). The `can_enter_jit` annotation on line 19 tells the JIT where an *application-level loop* (i.e., a loop in the actual bytecode application) begins; it is used to indicate backwards-branch bytecodes. The `jit_merge_point` annotation on line 10 tells the JIT where it is safe to move from the JIT back into the interpreter; it is used to identify the top of the interpreter's dispatch loop. As shown in Figure 2(a), the JIT generator replaces the `can_enter_jit` hints with calls into the JIT runtime and then serializes the annotated IR for all code regions between the meta-tracing annotations. These serialized IR representations are called "jitcodes" and are integrated along with the JIT runtime into the C-based interpreter. Figure 2(b) illustrates how the meta-tracing JIT compiler operates at runtime. When the C-based interpreter reaches a `can_enter_jit` hint, it begins using the corresponding jitcode to build a meta-trace of the interpreter interpreting the bytecode application. When the same `can_enter_jit` hint is reached again, the JIT increments an internal per-loop counter. Once this counter exceeds a threshold, the collected trace is handed off to a JIT optimizer and assembler before initiating native execution. The meta-traces include guards that ensure the dynamic conditions under which the meta-trace was optimized still hold (e.g., the types of application-level variables remain constant). If at any time a guard fails or if the optimized loop is finished, then the JIT returns control back to the C-based interpreter at a `jit_merge_point`.

Figure 3 illustrates how the RPython translation toolchain is leveraged by the Pydgin framework. Once an ISA has been specified using the Pydgin embedded-ADL (described in Section III) it is combined with the Pydgin simulation driver, which provides a modular, pre-defined interpreter implementation, to create an executable Pydgin instruction set simulator. Each Pydgin ISS is valid RPython that can be executed in a number of ways. The most straightforward execution is direct interpretation using CPython or PyPy. Although interpreted execution provides poor simulation performance, it serves as a particularly useful debugging platform during early stages of ISA development and testing. Alternatively, the Pydgin ISS can be passed as input to the RPython translation toolchain in order to generate a compiled executable implementing either an interpretive ISS or a high-performance DBT-ISS (described in Section IV).

III. The Pydgin Embedded-ADL

To evaluate the capabilities of the Pydgin framework, we use Pydgin to implement instruction set simulators for two ISAs: a simplified version of MIPS32 called SMIPS, and the more complex ARMv5 ISA. This process involves using the Pydgin embedded-ADL to describe the architectural state, instruction encoding, and instruction semantics of each ISA. No special parser is needed to generate simulators from Pydgin ISA definitions, and in fact these definitions can be executed directly using a standard Python interpreter. In this section, we describe the various components of the Pydgin embedded-ADL using the ARMv5 ISA as an example.

A. Architectural State

Architectural state in Pydgin is implemented using Python classes. Figure 4 shows a simplified version of this state for the ARMv5 ISA. Library components provided by the Pydgin embedded-ADL such as `RegisterFile` and `Memory` classes can be used as provided or subclassed to suit the specific needs of a particular architecture. For example, the `ArmRegisterFile` on line 5 subclasses the `RegisterFile` component (not shown) and specializes it for the unique idiosyncrasies of the ARM architecture: within instruction semantic definitions register 15 must update the current `PC` when written but return `PC+8` when read. The `fetch_pc` accessor on line 10 is used to retrieve the current instruction address, which is needed for both instruction fetch and incrementing the `PC` in instruction semantic definitions (discussed in Section III-C). Users may also implement their own data structures, however, these data structures must conform to the restrictions

```
class State( object ):
  _virtualizable_ = ['pc', 'ncycles']
  def __init__( self, memory, debug, 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

    # simulator/stats info, not architecturally visible
    self.status  = 0
    self.ncycles = 0

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

Figure 4. Simplified ARMv5 Architectural State Description

```
encodings = [
  ['nop',      '00000000000000000000000000000000'],
  ['mul',      'xxxx0000000xxxxxxxxxxxx1001xxxx'],
  ['umull',    'xxxx0000100xxxxxxxxxxxx1001xxxx'],
  ['adc',      'xxxx00x0101xxxxxxxxxxxxxxxxxxxx'],
  ['add',      'xxxx00x0100xxxxxxxxxxxxxxxxxxxx'],
  ['and',      'xxxx00x0000xxxxxxxxxxxxxxxxxxxx'],
  ['b',        'xxxx1010xxxxxxxxxxxxxxxxxxxxxxx'],
  ...
  ['teq',      'xxxx00x10011xxxxxxxxxxxxxxxxxxx'],
  ['tst',      'xxxx00x10001xxxxxxxxxxxxxxxxxxx'],
]
```

Figure 5. Partial ARMv5 Instruction Encoding Table

imposed by the RPython translation toolchain. The class member `_virtualizable_` is an optional JIT annotation used by the RPython translation toolchain. We discuss this and other advanced JIT annotations in more detail in Section IV.

B. Instruction Encoding

Pydgin maintains encodings of all instructions in a centralized data structure for easy maintenance and quick lookup. A partial encoding table for the ARMv5 instruction set can be seen in Figure 5. While some ADLs keep this encoding information associated with the each instruction's semantic definition (e.g., SimIt-ARM's definitions in Figure 8), we have found that this distributed organization makes it more difficult to quickly assess available encodings for introducing ISA extensions. However, ISA designers preferring this distributed organization can easily implement it using Python decorators to annotate instruction definitions with their encoding. This illustrates the power of using an embedded-ADL where arbitrary Python code can be used for metaprogramming. Encodings and instruction names provided in the instruction encoding table are used by Pydgin to automatically generate decoders for the simulator. Unlike some ADLs, Pydgin does not require the user to explicitly specify instruction types or mask bits for field matching because Pydgin can automatically infer field locations from the encoding table.

C. Instruction Semantics

Pydgin instruction semantic definitions are implemented using normal Python functions with the special signature

```
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)
```

Figure 6. ADD Instruction Semantics: ARM ISA Manual

```
def execute_add( s, inst ):
  if condition_passed( s, inst.cond() ):
    a        = s.rf[ inst.rn() ]
    b, cout  = shifter_operand( s, inst )

    result   = a + b
    s.rf[ inst.rd() ] = trim_32( result )

    if inst.S():
      if inst.rd() == 15:
        raise Exception('Writing SPSR not implemented!')
      s.N = (result >> 31)&1
      s.Z = trim_32( result ) == 0
      s.C = carry_from( result )
      s.V = overflow_from_add( a, b, result )

    if inst.rd() == 15:
      return

  s.rf[PC] = s.fetch_pc() + 4
```

Figure 7. ADD Instruction Semantics: Pydgin

`execute_<inst_name>(s,inst)`. The function parameters `s` and `inst` refer to the *architectural state* and the *instruction bits*, respectively. An example of a Pydgin instruction definition is shown for the ARMv5 `ADD` instruction in Figure 7. Pydgin allows users to create helper functions that refactor complex operations common across many instruction definitions. For example, `condition_passed` on line 2 performs predication checks, while `shifter_operand` on line 4 encapsulates ARMv5's complex rules for computing the secondary operand `b` and computes a special carry out condition needed by some instructions (stored in `cout`). This encapsulation provides the secondary benefit of helping Pydgin definitions better match the instruction semantics described in ISA manuals. Note that the Pydgin definition for `ADD` is a fairly close match to the instruction specification pseudocode provided in the official ARM ISA manual, shown in Figure 6.

Figure 8 shows another description of the `ADD` instruction in the SimIt-ARM ADL, a custom, lightweight ADL used by the open source SimIt-ARM simulator to generate both interpretive and DBT ISSs [21]. In comparison to the SimIt-ARM ADL, Pydgin is slightly less concise as a consequence of using an embedded-ADL rather than implementing a custom parser. The SimIt-ARM description implements `ADD` as four separate instructions in order to account for the `S` and `I` instruction bits. These bits determine whether condition flags are updated and if a rotate immediate addressing mode should be used, respectively. This multi-instruction approach is presumably done for performance reasons as splitting the `ADD` definition into separate instructions results in simpler decode and less branching behavior during simulation. However, this approach incurs ad-

```
op add(----00001000:rn:rd:shifts) {
execute="
  WRITE_REG($rd$, READ_REG($rn$) + $shifts$);
"
}

op adds(----00001001:rn:rd:shifts) {
execute="
  tmp32 = $shifts$; val32 = READ_REG($rn$);
  rslt32 = val32 + tmp32;
    if ($rd$==15) WRITE_CPSR(SPSR);
    else ASGN_NZCV(rslt32, rslt32<val32,
           (val32^tmp32^-1) & (val32^rslt32));
  WRITE_REG($rd$, rslt32);
"
}

op addi(----00101000:rn:rd:rotate_imm32) {
execute="
  WRITE_REG($rd$, READ_REG($rn$) + $rotate_imm32$);
"
}

op addis(----00101001:rn:rd:rotate_imm32) {
execute="
  tmp32 = $rotate_imm32$; val32 = READ_REG($rn$);
  rslt32 = val32 + tmp32;
  if ($rd$==15) WRITE_CPSR(SPSR);
  else ASGN_NZCV(rslt32, rslt32<val32,
         (val32^tmp32^-1) & (val32^rslt32));
  WRITE_REG($rd$, rslt32);
"
}
```

Figure 8. ADD Instruction Semantics: SimIt-ARM

```
inline void ADD(arm_isa* ref, int rd, int rn, bool s,
    ac_regbank<31, arm_parms::ac_word,
                   arm_parms::ac_Dword>& RB,
    ac_reg<unsigned>& ac_pc) {

  arm_isa::reg_t RD2, RN2;
  arm_isa::r64bit_t soma;

  RN2.entire = RB_read(rn);
  if(rn == PC) RN2.entire += 4;
  soma.hilo = (uint64_t)(uint32_t)RN2.entire +
      (uint64_t)(uint32_t)ref->dpi_shiftop.entire;
  RD2.entire = soma.reg[0];
  RB_write(rd, RD2.entire);
  if ((s == 1)&&(rd == PC)) {
    #ifndef FORGIVE_UNPREDICTABLE
    ...
    ref->SPSRtoCPSR();
    #endif
  } else {
    if (s == 1) {
      ref->flags.N = getBit(RD2.entire,31);
      ref->flags.Z = ((RD2.entire==0) ? true : false);
      ref->flags.C = ((soma.reg[1]!=0) ? true : false);
      ref->flags.V = (((getBit(RN2.entire,31)
          && getBit(ref->dpi_shiftop.entire,31)
          && (!getBit(RD2.entire,31)))
          || ((!getBit(RN2.entire,31))
          && (!getBit(ref->dpi_shiftop.entire,31))
          && getBit(RD2.entire,31))) ? true : false);
    }
  }
  ac_pc = RB_read(PC);
}
```

Figure 9. ADD Instruction Semantics: ArchC

ditional overhead in terms of clarity and maintainability. Pydgin largely avoids the need for these optimizations thanks to its meta-tracing JIT compiler that can effectively optimize away branching behavior for hot paths. This works particularly well for decoding instruction fields such as the ARM conditional bits and the `S` and `I` flags: for non-self modifying code an instruction at a particular PC will always have the same instruction bits, enabling the JIT to completely optimize away this complex decode overhead.

An ArchC description of the `ADD` instruction can be seen in Figure 9. Note that some debug code has been removed for the sake of brevity. ArchC is an open source, SystemC-based ADL popular in system-on-chip toolflows [3, 50]. ArchC has considerably more syntactic overhead than both the SimIt-ARM ADL and Pydgin embedded-ADL. Much of this syntactic overhead is due to ArchC's C++-style description which requires explicit declaration of complex templated types. One significant advantage ArchC's C++-based syntax has over SimIt-ARM's ADL is that it is compatible with existing C++ development tools. Pydgin benefits from RPython's dynamic typing to produce a comparatively cleaner syntax while also providing compatibility with Python development tools.

D. Benefits of an Embedded-ADL

While the dynamic nature of Python enables Pydgin to provide relatively concise, pseudo-code-like syntax for describing instructions, it could be made even more concise by implementing a DSL which uses a custom parser. From our experience, embedded-ADLs provide a number of advantages over a custom DSL approach: increased language familiarity and a gentler learning curve for new users; access to better development tools and debugging facilities; and easier maintenance and extension by avoiding a custom parser. Additionally, we have found that the ability to directly execute Pydgin ADL descriptions in a standard Python interpreter such as CPython or PyPy significantly helps debugging and testing during initial ISA exploration.

IV. Pydgin JIT Generation and Optimizations

It is not immediately obvious that a JIT framework designed for general-purpose dynamic languages will be suitable for constructing fast instruction set simulators. In fact, a DBT-ISS generated by the RPython translation toolchain using only the basic JIT annotations shown in Figure 10 provides good but not exceptional performance. This is because the JIT must often use worst-case assumptions about interpreter behavior. For example, the JIT must assume that functions might have side effects, variables are not constants, loop bounds might change, and object fields should be stored in memory. These worst-case assumptions reduce opportunities for JIT optimization and thus reduce the overall JIT performance.

Existing work on the RPython translation toolchain has demonstrated the key to improving JIT performance is the careful insertion of advanced annotations that provide the JIT high-level hints about interpreter behavior [9, 10]. We use a similar technique by adding annotations to the Pydgin framework specifically chosen to provide ISS-specific hints. Most of these advanced JIT annotations are completely self-contained within the Pydgin framework itself. Annotations encapsulated in this way can be leveraged across any instruction set specified using

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

class State( object ):
  _virtualizable_ = ['pc', 'ncycles']
  def __init__( self, memory, reset_addr=0x400 ):
    self.pc       = reset_addr
    self.ncycles  = 0
    # ... more architectural state ...

class Memory( object ):
  def __init__( self, size=2**10 ):
    self.size  = size << 2
    self.data  = [0] * self.size

  def read( self, start_addr, num_bytes ):
    word = start_addr >> 2
    byte = start_addr &  0b11
    if   num_bytes == 4:
      return self.data[ word ]
    elif num_bytes == 2:
      mask = 0xFFFF << (byte * 8)
      return (self.data[ word ] & mask) >> (byte * 8)
    # ... handle single byte read ...

  @elidable
  def iread( self, start_addr, num_bytes ):
    return self.data[ start_addr >> 2 ]

  # ... rest of memory methods ...

def run( state, max_insts=0 ):
  s = state
  while s.status == 0:
    jd.jit_merge_point( s.fetch_pc(), max_insts, s )

    pc  = hint( s.fetch_pc(), promote=True )
    old = pc
    mem = hint( s.mem, promote=True )

    inst = mem.iread( pc, 4 )
    exec_fun = decode( inst )
    exec_fun( s, inst )

    s.ncycles += 1

    if s.fetch_pc() < old:
      jd.can_enter_jit( s.fetch_pc(), max_insts, s )
```

Figure 10. Simplified Instruction Set Interpreter Written in RPython – Although only basic annotations (shown in blue) are required by the RPython translation toolchain to produce a JIT, more advanced annotations (shown in red) are needed to successfully generate efficient DBT-ISSs.

the Pydgin embedded-ADL without any manual customization of instruction semantics by the user. Figure 10 shows a simplified version of the Pydgin interpreter with several of these advanced JIT annotations highlighted.

We use several applications from SPEC CINT2006 compiled for the ARMv5 ISA to demonstrate the impact of six advanced JIT annotations key to producing high-performance DBT-ISSs with the RPython translation toolchain. These advanced annotations include: (1) elidable instruction fetch; (2) elidable decode; (3) constant promotion of memory and PC; (4) word-based target memory; (5) loop unrolling in instruction semantics; and (6) virtualizable PC. Figure 13 shows the speedups achieved as these advanced JIT annotations are gradually added to the Pydgin framework. Speedups are normalized against a Pydgin ARMv5 DBT-ISS using only basic JIT annotations. Figures 11

```
# Byte accesses for instruction fetch
i1 = getarrayitem_gc(p6, 33259)
i2 = int_lshift(i1, 8)
# ... 8 more JIT IR nodes ...

# Decode function call
p1 = call(decode, i3)
guard_no_exception()
i4 = getfield_gc_pure(p1)
guard_value(i4, 4289648)

# Accessing instruction fields
i5 = int_rshift(i3, 28)
guard_value(i5, 14)
i6 = int_rshift(i3, 25)
i7 = int_and(i6, 1)
i8 = int_is_true(i7)
# ... 11 more JIT IR nodes ...

# Read from regfile
i10 = getarrayitem_gc(p2, i9)

# Register offset calculation
i11 = int_and(i10, 0xffffff)
i12 = int_rshift(i3, 16)
i13 = int_and(i12, 15)
i14 = int_eq(i13, 15)
guard_false(i14)

# Read from regfile
i15 = getarrayitem_gc(p2, i13)

# Addressing mode
i15 = int_rshift(i3, 23)
i16 = int_and(i15, 1)
i17 = int_is_true(i16)
# ... 13 more JIT IR nodes ...

# Access mem with byte reads
i19 = getarrayitem_gc(p6, i18)
i20 = int_lshift(i19, 8)
i22 = int_add(i21, 2)
# ... 13 more JIT IR nodes ...

# Write result to regfile
setarrayitem_gc(p2, i23, i24)

# Update PC
i25 = getarrayitem_gc(p2, 15)
i26 = int_add(i25, 4)
setarrayitem_gc(p2, 15, i26)
i27 = getarrayitem_gc(p2, 15)
i28 = int_lt(i27, 33256)
guard_false(i28)
guard_value(i27, 33260)

# Update cycle count
i30 = int_add(i29, 1)
setfield_gc(p0, i30)
```

Figure 11. Unoptimized JIT IR for ARMv5 LDR Instruction – When provided with only basic JIT annotations, the meta-tracing JIT compiler will translate the LDR instruction into 79 JIT IR nodes.

```
i1 = getarrayitem_gc(p2, 0)    # register file read
i2 = int_add(i1, i8)           # address computation
i3 = int_and(i2, 0xffffffff)   # bit masking
i4 = int_rshift(i3, 2)         # word index
i5 = int_and(i3, 0x00000003)   # bit masking
i6 = getarrayitem_gc(p1, i4)   # memory access
i7 = int_add(i8, 1)            # update cycle count
```

Figure 12. Optimized JIT IR for ARMv5 LDR Instruction – Pydgin's advanced JIT annotations enable the meta-tracing JIT compiler to optimize the LDR instruction to just seven JIT IR nodes.

and 12 concretely illustrate how the introduction of these advanced JIT annotations reduce the JIT IR generated for a single LDR instruction from 79 IR nodes down to only 7 IR nodes. In the rest of this section, we describe how each advanced annotation specifically contributes to this reduction in JIT IR nodes and enables the application speedups shown in Figure 13.

Elidable Instruction Fetch – RPython allows functions to be marked *trace elidable* using the `@elidable` decorator. This annotation guarantees a function will not have any side effects and therefore will always return the same result if given the same arguments. If the JIT can determine that the arguments to a trace elidable function are likely constant, then the JIT can use constant folding to replace the function with its result and a series of guards to verify that the arguments have not changed. When executing programs without self-modifying code, the Pydgin ISS benefits from marking instruction fetches as trace elidable since the JIT can then assume the same instruction bits will always be returned for a given PC value. While this annotation, seen on line 26 in Figure 10, can potentially eliminate 10 JIT IR nodes on lines 1–4 in Figure 11, it shows negligible performance benefit in Figure 13. This is because the benefits of elidable instruction fetch are not realized until combined with other symbiotic annotations like elidable decode.

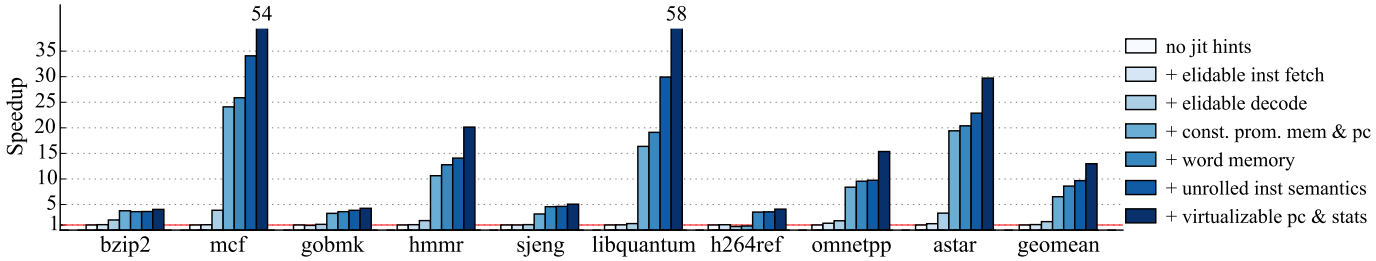

Figure 13. Impact of JIT Annotations – Including advanced annotations in the RPython interpreter allows our generated ISS to perform more aggressive JIT optimizations. However, the benefits of these optimizations varies from benchmark to benchmark. Above we show how incrementally combining several advanced JIT annotations impacts ISS performance when executing several SPEC CINT2006 benchmarks. Speedups are normalized against a Pydgin ARMv5 DBT-ISS using only basic JIT annotations.

Elidable Decode – Previous work has shown efficient instruction decoding is one of the more challenging aspects of designing fast ISSs [23, 27, 41]. Instruction decoding interprets the bits of a fetched instruction in order to determine which execution function should be used to properly emulate the instruction's semantics. In Pydgin, marking decode as *trace elidable* allows the JIT to optimize away all of the decode logic since a given set of instruction bits will always map to the same execution function. Elidable decode can potentially eliminate 20 JIT IR nodes on lines 6–18 in Figure 11. The combination of elidable instruction fetch and elidable decode shows the first performance increase for many applications in Figure 13.

Constant Promotion of PC and Target Memory – By default, the JIT cannot assume that the pointers to the PC and the target memory within the interpreter are constant, and this results in expensive and potentially unnecessary pointer dereferences. *Constant promotion* is a technique that converts a variable in the JIT IR into a constant plus a guard, and this in turn greatly increases opportunities for constant folding. The constant promotion annotations can be seen on lines 37–39 in Figure 10. Constant promotion of the PC and target memory is critical for realizing the benefits of the elidable instruction fetch and elidable decode optimizations mentioned above. When all three optimizations are combined the entire fetch and decode logic (i.e., lines 1–18 in Figure 11) can truly be removed from the optimized trace. Figure 13 shows how all three optimizations work together to increase performance by 5× on average and up to 25× on *429.mcf*. Only *464.h264ref* has shown no performance improvements up to this point.

Word-Based Target Memory – Because modern processors have byte-addressable memories the most intuitive representation of this target memory is a byte container, analogous to a `char` array in C. However, the common case for most user programs is to use full 32-bit word accesses rather than byte accesses. This results in additional access overheads in the interpreter for the majority of load and store instructions. As an alternative, we represent the target memory using a word container. While this incurs additional byte masking overheads for sub-word accesses, it makes full word accesses significantly cheaper and thus improves performance of the common case. Lines 11–24 in Figure 10 illustrates our target memory data structure which is able to transform the multiple memory accesses and 16 JIT IR nodes in lines 38–42 of Figure 11 into the single memory access on line 6 of Figure 12. The number and kind of memory accesses performed influence the benefits of this optimization. In Figure 13 most applications see a small benefit, outliers include *401.bzip2* which experiences a small performance degradation and *464.h264ref* which receives a large performance improvement.

Loop Unrolling in Instruction Semantics – The RPython toolchain conservatively avoids inlining function calls that contain loops since these loops often have different bounds for each function invocation. A tracing JIT attempting to unroll and optimize such loops will generally encounter a high number of guard failures, resulting in significant degradation of JIT performance. The `stm` and `ldm` instructions of the ARMv5 ISA use loops in the instruction semantics to iterate through a register bitmask and push or pop specified registers to the stack. Annotating these loops with the `@unroll_safe` decorator allows the JIT to assume that these loops have static bounds and can safely be unrolled. One drawback of this optimization is that it is specific to the ARMv5 ISA and currently requires modifying the actual instruction semantics, although we believe this requirement can be removed in future versions of Pydgin. The majority of applications in Figure 13 see only a minor improvement from this optimization, however, both *462.libquantum* and *429.mcf* receive a significant improvement from this optimization suggesting that they both include a considerable amount of stack manipulation.

Virtualizable PC and Statistics – State variables in the interpreter that change frequently during program execution (e.g., the PC and statistics counters) incur considerable execution overhead because the JIT conservatively implements object member access using relatively expensive loads and stores. To address this limitation, RPython allows some variables to be annotated as *virtualizable*. Virtualizable variables can be stored in registers and updated locally within an optimized JIT trace without loads and stores. Memory accesses that are needed to keep the object state synchronized between interpreted and JIT-compiled execution is performed only when entering and exiting a JIT trace. The virtualizable annotation (lines 2 and 5 of Figure 10) is able to eliminate lines 47–58 from Figure 11 resulting in an almost 2× performance improvement for *429.mcf* and *462.libquantum*. Note that even greater performance improvements can potentially be had by also making the register file virtualizable, however, a bug in the RPython translation toolchain prevented us from evaluating this optimization.

V. Evaluation

We evaluate Pydgin by implementing two ISAs using the Pydgin embedded-ADL: a simplified version of MIPS32 (SMIPS) and ARMv5. These embedded-ADL descriptions are com-

bined with RPython optimization annotations, including those described in Section IV, to generate high-performance, JIT-enabled DBT-ISSs. Traditional interpretive ISSs without JITs are also generated using the RPython translation toolchain in order to help quantify the performance benefit of the meta-tracing JIT. We compare the performance of these Pydgin-generated ISSs against several reference ISSs.

To quantify the simulation performance of each ISS, we collected total simulator execution time and simulated MIPS metrics from the ISSs running SPEC CINT2006 applications. All applications were compiled using the recommended SPEC optimization flags (`-O2`) and all simulations were performed on unloaded host machines; compiler and host-machine details can be found in Table I. Three applications from SPEC CINT2006 (*400.perlbench*, *403.gcc*, and *483.xalancbmk*) would not build successfully due to limited system call support in our Newlib-based cross-compilers. When evaluating the high-performance DBT-ISSs, target applications were run to completion using datasets from the SPEC reference inputs. Simulations of the interpretive ISSs were terminated after 10 billion simulated instructions since the poor performance of these simulators would require many hours, in some cases days, to run these benchmarks to completion. Total application runtimes for the truncated simulations (labeled with *Time** in Tables II and III) were extrapolated using MIPS measurements and dynamic instruction counts. Experiments on a subset of applications verified the simulated MIPS computed from these truncated runs provided a good approximation of MIPS measurements collected from full executions. This matches prior observations that interpretive ISSs demonstrate very little performance variation across program phases. Complete information on the SPEC CINT2006 application input datasets and dynamic instruction counts can be found in Tables II and III.

Reference simulators for SMIPS include a slightly modified version of the gem5 MIPS atomic simulator (*gem5-smips*) and a hand-written C++ ISS used internally for teaching and research purposes (*cpp-smips*). Both of these implementations are purely interpretive and do not take advantage of any JIT-optimization strategies. Reference simulators for ARMv5 include the gem5 ARM atomic simulator (*gem5-arm*), interpretive and JIT-enabled versions of SimIt-ARM (*simit-arm-nojit* and *simit-arm-jit*), as well as QEMU. Atomic models from the gem5 simulator [5] were chosen for comparison due their wide usage amongst computer architects. SimIt-ARM [21, 40] was selected because it is currently the highest performance ADL-generated DBT-ISS publicly available. QEMU has long been held as the gold-standard for DBT simulators due to its extremely high performance [4]. Note that QEMU achieves it's excellent performance at the cost of observability. Unlike QEMU, all other simulators in this study faithfully track architectural state at an instruction level rather than block level.

TABLE I. SIMULATION CONFIGURATIONS

Simulation Host		
CPU		Intel Xeon E5620
Frequency		2.40GHz
RAM		48GB @ 1066 MHz
Target Hosts		
ISA	Simplified MIPS32	ARMv5
Compiler	Newlib GCC 4.4.1	Newlib GCC 4.3.3
Executable	Linux ELF	Linux ELF
System Calls	Emulated	Emulated

All experiments were performed on an unloaded target machine described above. Both the ARMv5 and Simplified MIPS (SMIPS) ISAs used system call emulation. SPEC CINT2006 benchmarks were cross-compiled using SPEC recommended optimization flags (`-O2`).

A. SMIPS

Table II shows the complete performance evaluation results for each SMIPS ISS while Figure 14 shows a plot of simulator performance in MIPS. Pydgin's generated interpretive and DBT-ISSs are able to outperform *gem5-smips* and *cpp-smips* by a considerable margin: around a factor of 8–9× for *pydgin-smips-nojit* and a factor of 25–200× for *pydgin-smips-jit*. These speedups translate into considerable improvements in simulation times for large applications in SPEC CINT2006. For example, whereas *471.omnetpp* would have taken eight days to simulate on *gem5-smips*, this runtime is drastically reduced down to 21.3 hours on *pydgin-smips-nojit* and an even more impressive 1.3 hours on *pydgin-smips-jit*. These improvements significantly increase the kind of applications researchers can experiment with when performing design space exploration.

The interpretive ISSs tend to demonstrate relatively consistent performance across all benchmarks: 3–4 MIPS for *gem5-smips* and *cpp-smips*, 28–36 MIPS for *pydgin-smips-nojit*. Unlike DBT-ISSs which that optimize away many overheads for frequently encountered instruction paths, interpretive ISSs must perform both instruction fetch and decode for every instruction simulated. These overheads limit the amount of simulation time variability, which is primarily caused by complexity differences between instruction implementations.

Also interesting to note are the different implementation approaches used by each of these interpretive simulators. The *cpp-smips* simulator is completely hand-coded with no generated components, whereas the *gem5-smips* decoder and instruction classes are automatically generated from what the gem5 documentation describes as an "ISA description language" (effectively an ad-hoc and relatively verbose ADL). As mentioned previously, *pydgin-smips-nojit* is generated from a high-level embedded-ADL. Both the generated *gem5-smips* and *pydgin-smips-nojit* simulators are able to outperform the hand-coded *cpp-smips*, demonstrating that generated simulator approaches can provide both productivity and performance advantages over simple manual implementations.

In addition to providing significant performance advantages over *gem5-smips*, both Pydgin simulators provide considerable productivity advantages as well. Because the gem5 instruction descriptions have no interpreter, they must be first generated into C++ before testing. This leaves the user to deduce whether the source of an implementation bug resides in the instruction definition, the code generator, or the gem5 simulator framework. In comparison, Pydgin's embedded-ADL is fully compliant Python that requires no custom parsing and can be executed directly in a standard Python interpreter. This allows Pydgin ISA implementations to be tested and verified using Python debugging tools prior to RPython translation into a fast C implementation, leading to a much more user-friendly debugging experience.

TABLE II. SMIPS PERFORMANCE RESULTS

			gem5		cpp			pydgin nojit			pydgin jit		
Benchmark	**Dataset**	**Inst (B)**	**Time***	**MIPS**	**Time***	**MIPS**	**vs. g5**	**Time***	**MIPS**	**vs. g5**	**Time**	**MIPS**	**vs. g5**
401.bzip2	`chicken.jpg`	198	15.1h	3.7	17.2h	3.2	0.87	1.6h	34	9.3	19.3m	171	47
429.mcf	`inp.in`	337	1.1d	3.7	1.2d	3.2	0.87	3.3h	28	7.8	15.0m	373	102
445.gobmk	`13x13.tst`	290	21.7h	3.7	1.1d	3.1	0.83	2.6h	31	8.4	29.0m	167	45
456.hmmer	`nph3.hmm`	1212	3.8d	3.7	4.5d	3.2	0.84	10.4h	32	8.7	26.5m	761	204
458.sjeng	`ref.txt`	2757	8.5d	3.7	10.2d	3.1	0.83	1.0d	31	8.4	2.3h	337	90
462.libquantum	`1397 8`	2917	8.9d	3.8	10.9d	3.1	0.81	23.3h	35	9.1	1.3h	629	165
464.h264ref	`foreman_ref`	679	2.2d	3.5	2.5d	3.2	0.90	5.7h	33	9.4	2.2h	87	25
471.omnetpp	`omnetpp.ini`	2708	8.3d	3.8	10.0d	3.1	0.84	21.2h	36	9.4	1.3h	572	152
473.astar	`BigLakes2048.cfg`	472	1.5d	3.8	1.7d	3.2	0.85	4.1h	32	8.4	16.5m	476	127

Benchmark datasets taken from the SPEC CINT2006 reference inputs. Time is provided in either minutes (m), hours (h), or days (d) where appropriate. Time* indicates runtime estimates that were extrapolated from simulations terminated after 10 billion instructions. DBT-ISSs (*pydgin-smips-jit*) were simulated to completion. vs. g5 = simulator performance normalized to gem5.

TABLE III. ARMV5 PERFORMANCE RESULTS

		gem5		simit nojit			simit jit			pydgin nojit				pydgin jit				qemu	
Benchmark	**Inst (B)**	**Time***	**MIPS**	**Time***	**MIPS**	**vs. g5**	**Time**	**MIPS**	**vs. g5**	**Time***	**MIPS**	**vs. g5**	**vs. s0**	**Time**	**MIPS**	**vs. g5**	**vs. sJ**	**Time**	**MIPS**
401.bzip2	195	23.4h	2.3	52.2m	62	27	7.3m	445	192	2.5h	22	9.4	0.35	32.6m	100	43	0.22	3.0m	1085
429.mcf	374	1.9d	2.3	2.0h	52	23	19.9m	314	135	5.0h	21	9.0	0.40	15.5m	401	173	1.28	9.8m	637
445.gobmk	324	1.7d	2.2	1.8h	50	22	23.5m	230	103	4.5h	20	8.9	0.40	1.3h	70	31	0.30	14.0m	386
456.hmmer	1113	6.0d	2.1	5.9h	52	24	46.3m	400	187	14.6h	21	9.9	0.41	30.5m	607	284	1.52	16.7m	1108
458.sjeng	2974	15.1d	2.3	16.7h	49	22	2.9h	287	126	1.7d	20	8.8	0.41	8.8h	94	41	0.33	1.8h	447
462.libquantum	3070	14.4d	2.5	15.6h	55	22	1.9h	459	186	1.6d	22	8.9	0.40	1.3h	659	267	1.44	41.9m	1220
464.h264ref	753	3.8d	2.3	4.2h	50	22	31.7m	396	173	10.2h	21	9.0	0.42	21.8h	9.6	4.2	0.02	16.2m	773
471.omnetpp	1282	5.8d	2.6	5.3h	68	26	1.5h	233	90	14.1h	25	9.8	0.37	1.0h	355	138	1.52	1.5h	240
473.astar	434	2.0d	2.5	2.2h	55	22	23.3m	310	126	5.5h	22	8.8	0.40	29.5m	245	100	0.79	13.8m	526

Benchmark datasets taken from the SPEC CINT2006 reference inputs (shown in Table II). Time is provided in either minutes (m), hours (h), or days (d) where appropriate. Time* indicates runtime estimates that were extrapolated from simulations terminated after 10 billion instructions. DBT-ISSs (*simit-arm-jit*, *pydgin-arm-jit*, and QEMU) were simulated to completion. vs. g5 = simulator performance normalized to gem5. vs. s0 = simulator performance normalized to *simit-arm-nojit*. vs. sJ = simulator performance normalized to *simit-arm-jit*.

Enabling JIT optimizations in the RPython translation toolchain results in a considerable improvement in Pydgin-generated ISS performance: from 28–36 MIPS for *pydgin-smips-nojit* up to 87–761 MIPS for *pydgin-smips-jit*. Compared to the interpretive ISSs, *pydgin-smips-jit* demonstrates a much greater range of performance variability that depends on the characteristics of the application being simulated. The RPython generated meta-tracing JIT is designed to optimize hot loops and performs best on applications that execute large numbers of frequently visited loops with little branching behavior. As a result, applications with large amounts of irregular control flow cannot be optimized as well as more regular applications. For example, although *464.h264ref* shows decent speedups on *pydgin-smips-jit* when compared to the interpretive ISSs, its performance in MIPS lags that of other applications by a wide margin. Improving DBT-ISS performance on challenging applications such as *464.h264ref* remains important future work.

B. ARMv5

The ARMv5 ISA demonstrates significantly more complex instruction behavior than the relatively simple SMIPS ISA. Although still a RISC ISA, ARMv5 instructions include a number of interesting features that greatly complicate instruction processing such as pervasive use of conditional instruction flags and fairly complex register addressing modes. This additional complexity makes ARMv5 instruction decode and execution much more difficult to emulate efficiently when compared to SMIPS. This is demonstrated in the relative performance of the two gem5 ISA models shown in Tables II and III: *gem5-arm* performance never exceeds 2.6 MIPS whereas *gem5-smips* averages 3.7 MIPS. Note that this trend is also visible when comparing *pydgin-arm-nojit* (20–25 MIPS) and *pydgin-smips-nojit* (28–36 MIPS). Complete performance results for all ARMv5 ISSs can be found in Table III and Figure 15.

To help mitigate some of the additional decode complexity of the ARMv5 ISA, ISS implementers can create more optimized instruction definitions that deviate from the pseudo-code form described in the ARMv5 ISA manual (as previously discussed in Section III). These optimizations and others enable the SimIt-ARM ISS to achieve simulation speeds of 49–68 MIPS for *simit-arm-nojit* and 230–459 MIPS for *simit-arm-jit*. In comparison, Pydgin's more straightforward ADL descriptions of the ARMv5 ISA result in an ISS performance of 20–25 MIPS for *pydgin-arm-nojit* and 9–659 MIPS for *pydgin-arm-jit*.

Comparing the interpretive versions of the SimIt-ARM and Pydgin generated ISSs reveals that *simit-arm-nojit* is able to

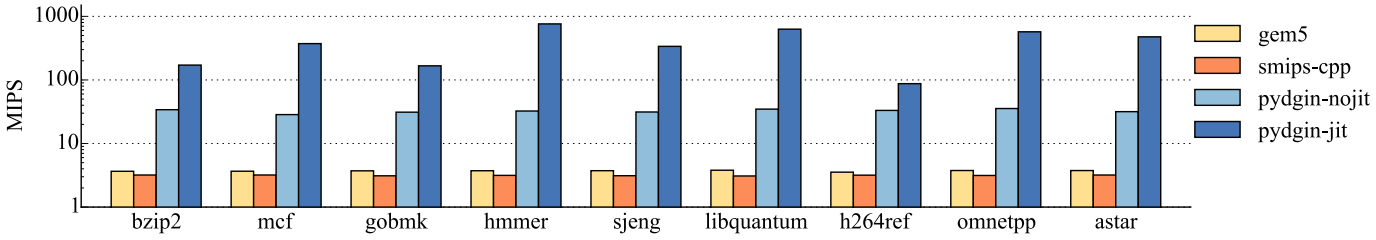

Figure 14. SMIPS ISS Performance

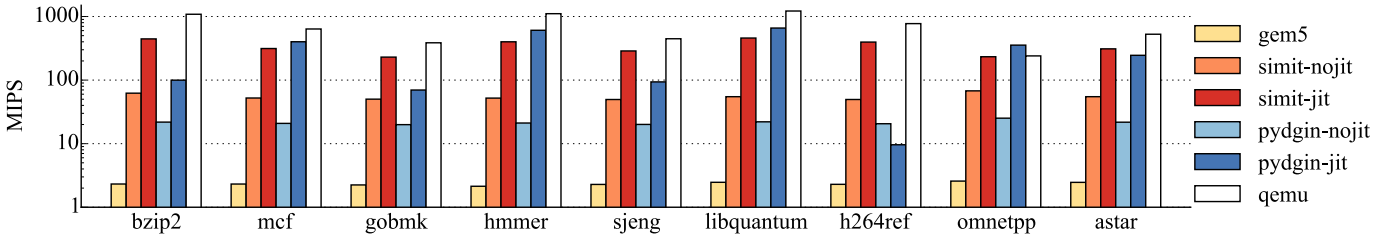

Figure 15. ARMv5 ISS Performance

outperform *pydgin-arm-nojit* by a factor of 2× on all applications. The fetch and decode overheads of interpretive simulators make it likely much of this performance improvement is due to SimIt-ARM's decode optimizations. However, decode optimizations should have less impact on DBT-ISSs which are often able to eliminate decode entirely.

The DBT-ISS versions of SimIt-ARM and Pydgin exhibit comparatively more complex performance characteristics: *simit-arm-jit* is able to consistently provide good speedups across all applications while *pydgin-arm-jit* has a much greater range of variability. Overall *pydgin-arm-jit* is able to outperform *simit-arm-jit* on approximately half of the applications, including considerable performance improvements of 1.44–1.52× for the applications *456.hmmer*, *462.libquantum*, and *471.omnetpp*. However, *pydgin-arm-jit* performs relatively poorly on *445.gobmk*, *458.sjeng*, and especially *464.h264ref* (all under 100 MIPS), while *simit-arm-jit* never does worse than 230 MIPS on any benchmark.

The variability differences displayed by these two DBT-ISSs is a result of the distinct JIT architectures employed by Pydgin and SimIt-ARM. Unlike *pydgin-arm-jit*'s meta-tracing JIT which tries to detect hot loops and highly optimize frequently taken paths through them, *simit-arm-jit* uses a page-based approach to JIT optimization that partitions an application binary into equal sized bins, or *pages*, of sequential program instructions. Once visits to a particular page exceed a preset threshold, all instructions within that page are compiled together into a single optimized code block. A page-based JIT provides two important advantages over a tracing JIT: first, pages are constrained to a fixed number of instructions (on the order of 1000) which prevents unbounded trace growth for irregular code; second, pages enable JIT-optimization of code that does not contain loops. While this approach to JIT design prevents SimIt-ARM from reaching the same levels of optimization as a trace-based JIT on code with regular control flow, it allows for more consistent performance across a range of application behaviors.

One particularly bad example of pathological behavior in Pydgin's tracing JIT is *464.h264ref*, the only application to perform worse on *pydgin-arm-jit* than *pydgin-arm-nojit* (9.6 MIPS vs. 21 MIPS). The cause of this performance degradation is a large number of tracing aborts in the JIT due to traces growing too long, most likely due to irregular code with complex function call chains. Tracing aborts cause *pydgin-arm-jit* to incur the overheads of tracing without the ability to amortize these overheads by executing optimized JIT-generated code. A similar problem is encountered by tracing JITs for dynamic languages and is currently an active area of research. We hope to look into into potential approaches to mitigate this undesirable JIT behavior in future work.

QEMU also demonstrates a wide variability in simulation performance depending on the application (240–1220 MIPS), however it achieves a much higher maximum performance and manages to outperform *simit-arm-jit* and *pydgin-arm-jit* on nearly every application except for *471.omnetpp*. Although QEMU has exceptional performance, it has a number of drawbacks that impact its usability. Retargeting QEMU simulators for new instructions requires manually writing blocks of low-level code in the tiny code generator (TCG) intermediate representation, rather than automatically generating a simulator from a high-level ADL. Additionally, QEMU sacrifices observability by only faithfully tracking architectural state at the block level rather than at the instruction level. These two limitations impact the productivity of researchers interested in rapidly experimenting with new ISA extensions.

VI. Related Work

A considerable amount of prior work exists on improving the performance of instruction set simulators through dynamic optimization. Foundational work on simulators leveraging dynamic binary translation (DBT) provided significant performance benefits over traditional interpretive simulation [20, 30, 31, 57]. These performance benefits have been further enhanced by optimizations that reduce overheads and improve code generation quality of JIT compilation [26, 29, 53]. Current state-of-the-art

ISSs incorporate parallel JIT-compilation task farms [7], multicore simulation with JIT-compilation [1, 40], or both [28]. These approaches generally require hand modification of the underlying DBT engine in order to achieve good performance for user-introduced instruction extensions.

In addition, significant research has been spent on improving the usability and retargetability of ISSs, particularly in the domain of application-specific instruction-set processor (ASIP) toolflows. Numerous frameworks have proposed using a high-level architectural description language (ADL) to generate software development artifacts such as cross compilers [3,14,18,19, 22, 25] and software decoders [23, 27, 41]. Instruction set simulators generated using an ADL-driven approach [3, 43, 44], or even from definitions parsed directly from an ISA manual [6], provide considerable productivity benefits but suffer from poor performance when using a purely interpretive implementation strategy. ADL-generated ISSs have also been proposed that incorporate various JIT-compilation strategies, including just-in-time cache-compiled (JIT-CCS) [16, 32], instruction-set compiled (ISCS) [45, 47, 48], hybrid-compiled [46, 49], dynamic-compiled [15, 38, 40], multicore and distributed dynamic-compiled [21], and parallel DBT [56].

Penry et al. introduced the *orthogonal-specification principle* as an approach to functional simulator design that proposes separating simulator specification from simulator implementation [34, 35]. This work is very much in the same spirit as Pydgin, which aims to separate JIT implementation details from architecture implementation descriptions by leveraging the RPython translation toolchain. RPython has previously been used for emulating hardware in the PyGirl project [17]. PyGirl is a whole-system VM (WSVM) that emulates the processor, peripherals, and timing-behavior of the Game Boy and had no JIT, whereas our work focuses on JIT-enabled, timing-agnostic instruction-set simulators.

VII. Conclusions

In an era of rapid development of increasingly specialized system-on-chip platforms, instruction set simulators can sacrifice neither designer productivity nor simulation performance. However, constructing ISS toolchains that are both highly productive and high performance remains a significant research challenge. To address these multiple challenges, we have introduced Pydgin: a novel approach to the automatic generation of high-performance DBT-ISSs from a Python-based embedded-ADL. Pydgin creatively adapts an existing meta-tracing JIT compilation framework designed for general-purpose dynamic programming languages towards the purpose of generating ISSs.

Pydgin opens up a number of interesting directions for future research. Further performance optimizations are certainly possible, including: using Python meta-programming during translator elaboration to further specialize instruction definitions (e.g., automatically generating variants for ARM S and I instruction bits as SimIt-ARM does manually); inserting additional RPython annotations in the embedded-ADL libraries; or possibly even modifying the RPython translation toolchain itself. One significant benefit of the Pydgin approach is that any improvements applied to the actively developed RPython translation toolchain immediately benefit Pydgin ISSs after a simple software download and retranslation, allowing Pydgin to track ongoing advances in the JIT research community. Additionally, we believe Pydgin's meta-tracing JIT compiler approach suggests a potential opportunity to use RPython to add simple timing models (e.g., simple pipeline or cache models) to an ISS, then using the RPython translation toolchain to produce a JIT which can optimize the performance of *both* the instruction execution and the timing model.

The Pydgin framework along with the Pydgin SMIPS and ARMv5 ISSs have been released under an open source software license. Our hope is that the productivity and performance benefits of Pydgin will make it a useful framework for the broader research community.

Acknowledgments

This work was supported in part by NSF CAREER Award #1149464, a DARPA Young Faculty Award, and a donation from Intel Corporation. We would like to sincerely thank Carl Friedrich Bolz and Maciej Fijałkowski for their assistance in performance tuning Pydgin as well as their valuable feedback on the paper. We would also like to thank Kai Wang for his help debugging the ARMv5 Pydgin ISS and Shreesha Srinath for his user feedback as an early adopter of Pydgin in his research.

References

[1] O. Almer et al. Scalable Multi-Core Simulation Using Parallel Dynamic Binary Translation. *Int'l Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, Jul 2011.

[2] D. Ancona et al. RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages. *Symp. on Dynamic Languages*, Oct 2007.

[3] R. Azevedo et al. The ArchC Architecture Description Language and Tools. *Int'l Journal of Parallel Programming (IJPP)*, 33(5):453–484, Oct 2005.

[4] F. Bellard. QEMU, A Fast and Portable Dynamic Translator. *USENIX Annual Technical Conference (ATEC)*, Apr 2005.

[5] N. Binkert et al. The gem5 Simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, Aug 2011.

[6] F. Blanqui et al. Designing a CPU Model: From a Pseudo-formal Document to Fast Code. *CoRR arXiv:1109.4351*, Sep 2011.

[7] I. Böhm, B. Franke, and N. Topham. Generalized Just-In-Time Trace Compilation Using a Parallel Task Farm in a Dynamic Binary Translator. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun 2011.

[8] C. F. Bolz. *Meta-Tracing Just-In-Time Compilation for RPython*. PhD thesis, Mathematisch-Naturwissenschaftliche Fakultät, Heinrich Heine Universität Düsseldorf, 2012.

[9] C. F. Bolz et al. Allocation Removal by Partial Evaluation in a Tracing JIT. *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, Jan 2011.

[10] C. F. Bolz et al. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. *Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS)*, Jul 2009.

[11] C. F. Bolz et al. Back to the Future in One Week — Implementing a Smalltalk VM in PyPy. *Workshop on Self-Sustaining Systems (S3)*, May 2008.

[12] C. F. Bolz, M. Leuschel, and D. Schneider. Towards a Jitting VM for Prolog Execution. *Int'l Symp. on Principles and Practice of Declarative Programming (PPDP)*, Jul 2010.

[13] C. F. Bolz et al. Meta-Tracing Makes a Fast Racket. *Workshop on Dynamic Languages and Applications (DYLA)*, Jun 2014.

[14] F. Brandner, D. Ebner, and A. Krall. Compiler Generation from Structural Architecture Descriptions. *Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, Sep 2007.

[15] F. Brandner et al. Fast and Accurate Simulation using the LLVM Compiler Framework. *Workshop on Rapid Simulation and Performance Evalution: Methods and Tools (RAPIDO)*, Jan 2009.

[16] G. Braun et al. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, Jun 2004.

[17] C. Bruni and T. Verwaest. PyGirl: Generating Whole-System VMs from High-Level Prototypes Using PyPy. *Int'l Conf. on Objects, Components, Models, and Patterns (TOOLS-EUROPE)*, Jun 2009.

[18] J. Ceng et al. Modeling Instruction Semantics in ADL Processor Descriptions for C Compiler Retargeting. *Int'l Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, Jul 2004.

[19] J. Ceng et al. C Compiler Retargeting Based on Instruction Semantics Models. *Design, Automation, and Test in Europe (DATE)*, Mar 2005.

[20] B. Cmelik and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, May 1994.

[21] J. D'Errico and W. Qin. Constructing Portable Compiled Instruction-Set Simulators — An ADL-Driven Approach. *Design, Automation, and Test in Europe (DATE)*, Mar 2006.

[22] S. Farfeleder et al. Effective Compiler Generation by Architecture Description. *International Conference on Languages, Compilers, Tools, and Theory for Embedded Systems (LCTES)*, Jun 2006.

[23] N. Fournel, L. Michel, and F. Pétrot. Automated Generation of Efficient Instruction Decoders for Instruction Set Simulators. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2013.

[24] HippyVM PHP Implementation. Online Webpage, 2014 (accessed Jan 14, 2015), `http://www.hippyvm.com`.

[25] M. Hohenauer et al. A Methodology and Tool Suite for C Compiler Generation from ADL Processor Models. *Design, Automation, and Test in Europe (DATE)*, Feb 2004.

[26] D. Jones and N. Topham. High Speed CPU Simulation Using LTU Dynamic Binary Translation. *Int'l Conf. on High Performance Embedded Architectures and Compilers (HiPEAC)*, Jan 2009.

[27] R. Krishna and T. Austin. Efficient Software Decoder Design. *Workshop on Binary Translation (WBT)*, Sep 2001.

[28] S. Kyle et al. Efficiently Parallelizing Instruction Set Simulation of Embedded Multi-Core Processors Using Region-Based Just-In-Time Dynamic Binary Translation. *International Conference on Languages, Compilers, Tools, and Theory for Embedded Systems (LCTES)*, Jun 2012.

[29] Y. Lifshitz et al. Zsim: A Fast Architectural Simulator for ISA Design-Space Exploration. *Workshop on Infrastructures for Software/Hardware Co-Design (WISH)*, Apr 2011.

[30] C. May. Mimic: A Fast System/370 Simulator. *ACM Sigplan Symp. on Interpreters and Interpretive Techniques*, Jun 1987.

[31] W. S. Mong and J. Zhu. DynamoSim: A Trace-Based Dynamically Compiled Instruction Set Simulator. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2004.

[32] A. Nohl et al. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. *Design Automation Conf. (DAC)*, Jun 2002.

[33] S. Pees, A. Hoffmann, and H. Meyr. Retargetable Compiled Simulation of Embedded Processors Using a Machine Description Language. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, Oct 2000.

[34] D. A. Penry. A Single-Specification Principle for Functional-to-Timing Simulator Interface Design. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2011.

[35] D. A. Penry and K. D. Cahill. ADL-Based Specification of Implementation Styles for Functional Simulators. *Int'l Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, Jul 2011.

[36] B. Peterson. PyPy. In A. Brown and G. Wilson, editors, *The Architecture of Open Source Applications, Volume II*. LuLu.com, 2008.

[37] F. Pizio. Introducing the WebKit FTL JIT. Online Article (accessed Sep 26, 2014), May 2014, `https://www.webkit.org/blog/3362/introducing-the-webkit-ftl-jit`.

[38] Z. Přikryl et al. Fast Just-In-Time Translated Simulator for ASIP Design. *Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, Apr 2011.

[39] PyPy. Online Webpage, 2014 (accessed Sep 26, 2014), `http://www.pypy.org`.

[40] W. Qin, J. D'Errico, and X. Zhu. A Multiprocessing Approach to Accelerate Retargetable and Portable Dynamic-Compiled Instruction-Set Simulation. *Intl'l Conf. on Hardware/Software Codesign and System Synthesis (CODES/ISSS)*, Oct 2006.

[41] W. Qin and S. Malik. Automated Synthesis of Efficient Binary Decoders for Retargetable Software Toolkits. *Design Automation Conf. (DAC)*, Jun 2003.

[42] W. Qin and S. Malik. Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation. *Design, Automation, and Test in Europe (DATE)*, Jun 2003.

[43] W. Qin and S. Malik. A Study of Architecture Description Languages from a Model-based Perspective. *Workshop on Microprocessor Test and Verification (MTV)*, Nov 2005.

[44] W. Qin, S. Rajagopalan, and S. Malik. A Formal Concurrency Model Based Architecture Description Language for Synthesis of Software Development Tools. *International Conference on Languages, Compilers, Tools, and Theory for Embedded Systems (LCTES)*, Jun 2004.

[45] M. Reshadi et al. An Efficient Retargetable Framework for Instruction-Set Simulation. *Intl'l Conf. on Hardware/Software Codesign and System Synthesis (CODES/ISSS)*, Oct 2003.

[46] M. Reshadi and N. Dutt. Reducing Compilation Time Overhead in Compiled Simulators. *Int'l Conf. on Computer Design (ICCD)*, Oct 2003.

[47] M. Reshadi, N. Dutt, and P. Mishra. A Retargetable Framework for Instruction-Set Architecture Simulation. *IEEE Trans. on Embedded Computing Systems (TECS)*, May 2006.

[48] M. Reshadi, P. Mishra, and N. Dutt. Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation. *Design Automation Conf. (DAC)*, Jun 2003.

[49] M. Reshadi, P. Mishra, and N. Dutt. Hybrid-Compiled Simulation: An Efficient Technique for Instruction-Set Architecture Simulation. *IEEE Trans. on Embedded Computing Systems (TECS)*, Apr 2009.

[50] S. Rigo et al. ArchC: A SystemC-Based Architecture Description Language. *Int'l Symp. on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct 2004.

[51] E. W. Thomassen. Trace-Based Just-In-Time Compiler for Haskell with RPython. Master's thesis, Department of Computer and Information Science, Norwegian University of Science and Technology, 2013.

[52] Topaz Ruby Implementation. Online Webpage, 2014 (accessed Jan 14, 2015), `http://github.com/topazproject/topaz`.

[53] N. Topham and D. Jones. High Speed CPU Simulation using JIT Binary Translation. *Workshop on Modeling, Benchmarking and Simulation (MOBS)*, Jun 2007.

[54] L. Tratt. Compile-Time Meta-Programming in a Dynamically Typed OO Language. *Dynamic Languages Symposium (DLS)*, Oct 2005.

[55] V. Živojnović, S. Pees, and H. Meyr. LISA - Machine Description Language and Generic Machine Model for HW/ SW CO-Design. *Workshop on VLSI Signal Processing*, Oct 1996.

[56] H. Wagstaff et al. Early Partial Evaluation in a JIT-Compiled, Retargetable Instruction Set Simulator Generated from a High-Level Architecture Description. *Design Automation Conf. (DAC)*, Jun 2013.

[57] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, May 1996.