Pydgin for RISC-V: A Fast and Productive Instruction-Set Simulator

Berkin Ilbeyi, Derek Lockhart, and Christopher Batten
School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{bi45,dml257,cbatten}@cornell.edu

Abstract

RISC-V is a new instruction-set architecture that encourages users to design new domain-specific extensions for their needs. This necessitates RISC-V instruction-set simulators that allow productive development, productive extension, and productive instrumentation. In addition, these simulators need to be high-performance to allow simulating real-world benchmarks. There is a productivity-performance gap due to lack of tools that achieve both aspects. Pydgin is a new instruction-set simulator that aims to bridge this gap by its many productivity features and performance that makes it the fastest currently available open-source RISC-V simulator.

1. Introduction

Instruction-set simulators (ISSs) are used to functionally simulate instruction-set architecture (ISA) semantics. ISSs play an important role in aiding software development for experimental hardware targets that do not exist yet. ISSs can be used to help design brand new ISAs or ISA extensions for existing ISAs. ISSs can also be complemented with performance counters to aid in the initial design-space exploration of novel hardware/software interfaces.

Performance is one of the most important qualities for an ISS. High-performance ISSs allow real-world benchmarks (many minutes of simulated time) to be simulated in a reasonable amount of time (hours of simulation time). For the simplest ISS type, an interpretive ISS, typical simulation times are between 1 to 10 millions of instructions per second (MIPS) on a contemporary server-class host CPU. For a trillion-instruction-long benchmark, a typical length found in SPEC CINT2006, this would take many days of simulation time. Instructions in an interpretive ISS need to be fetched, decoded, and executed in program order. To improve the performance, a much more sophisticated technique called dynamic binary translation (DBT) is used by more advanced ISSs. In DBT, the target instructions are *dynamically* translated to host instructions and cached for future use. Whenever possible, these already-translated and cached host instructions are used to amortize much of the target instruction fetch and decode overhead. DBT-based ISSs typically achieve performance levels in the range of 100s of MIPS, lowering the simulation time to a few hours. QEMU is a widely used DBT-based ISS because of its high performance, which can achieve 1000 MIPS on certain benchmarks, or less than an hour of simulation time [2].

Productivity is another important quality for ISSs. A productive ISS allows *productive development* of new ISAs; *productive extension* for ISA specialization; and *productive custom instrumentation* to quantify the performance benefits of new ISAs and extensions. These productivity features might not be a high priority for the users of proprietary ISAs. Proprietary ISAs tend to be specified by a single vendor, and aim to satisfy the use cases of all users. These users might not need their ISS to be productive for extensions and instrumentation because non-standard ISA extensions are uncommon. Contrary to this, RISC-V is a new ISA that embraces the idea of specifying a minimalist standard ISA and encouraging users to use its numerous mechanisms for extensions for their domain-specific needs [20]. The users of RISC-V would likely need their ISS to be productive for extension and instrumentation while still needing the high-performance features to allow them to run real-world benchmarks. There is a lack of tools to bridge this productivity-performance gap, and the RISC-V community is especially affected by it.

There has been previous ISSs that aim to achieve both productivity and performance. Highly productive ISSs typically use high-level architecture description languages (ADLs) to represent the ISA semantics [6, 11, 14, 15, 18]. However, to achieve high performance, ISSs use low-level languages such as C, with a custom DBT, which requires in-depth and low-level knowledge of the DBT internals [5, 7, 9, 10, 17, 21]. To bridge this gap, previous research have focused on techniques to automatically translate high-level ADLs into a low-level language where the custom DBT can optimize the performance [12, 13, 19]. However, these approaches tend to suffer because the ADLs are too close to low-level C, not very well supported, or not open source.

A similar productivity-performance tension exists in the programming languages community. Interpreters for highly productive dynamic languages (e.g., JavaScript and Python) need to be written in low-level C/C++ with very complicated custom just-in-time compilers (JITs). A notable exception to this is the PyPy project, the JIT-optimized interpreter for the Python language, which was written in a reduced typeable subset of Python, called RPython [1, 3, 4]. To translate the interpreter source written in RPython to a native binary, the PyPy community also developed the RPython translation toolchain. The RPython translation toolchain translates the interpreter source from the RPython language, adds a JIT, and generates C to be compiled with a standard C compiler. Pydgin is an ISS written in the RPython language, and uses the RPython translation toolchain to generate a fast JIT-optimized interpreter (JIT and DBT in the context of ISSs are very similar; we use both terms interchangeably in this paper) from high-level architectural descriptions in Python [8]. This unique development approach of Pydgin and the productivity features built into the Pydgin framework make this an ideal candidate to fill in the ISS productivity-performance gap.

In the remainder of this paper, Section 2 provides an introduction to the Pydgin ADL and Pydgin framework, Section 3

```
class State( object ):

  def __init__( self, memory, reset_addr=0x400 ):

    self.pc       = reset_addr
    self.rf       = RiscVRegisterFile()
    self.mem      = memory

    # optional state if floating point is enabled
    if ENABLE_FP:
      self.fp   = RiscVFPRegisterFile()
      self.fcsr = 0

    # used for instrumentation, not arch. visible
    self.num_insts = 0

    # state for custom instrumentation (number of
    # additions, number of misaligned memory operations,
    # list of all loops)
    self.num_adds        = 0
    self.num_misaligned = 0
    # a dict that returns 0 if the key is not found
    self.loop_dict      = DefaultDict(0)
```

Figure 1: Simplified RISC-V Architectural State Description – State for instrumentation can also be added here. The last three fields, `num_adds`, `num_aligned`, and `loops_dict` are examples of custom instrumentation state.

provides examples of Pydgin productivity, Section 4 highlights the performance of Pydgin, and Section 5 concludes.

2. Pydgin ADL and Framework

Pydgin uses the RPython language as the basis of its embedded ADL. RPython has minimal restrictions compared to Python, such as dynamic typing not being allowed, however it still inherits many of the productivity features of full Python: simple syntax, automatic memory management, and a large standard library. Unlike other embedded ADL approaches that use a low-level host language such as C, architecture descriptions in Pydgin can make use of many of the high-level features of the Python host language. The ADL in Pydgin consists of architectural state, instruction encodings, and instruction semantics. The architectural state uses a plain Python class as shown in Figure 1. The Pydgin framework provides templates for various data structures such as memories and register files, and ISA implementers can either use these components directly or extend them to specialize for their architecture. For example, the RISC-V register file special-cases `x0` to always contain the value 0. Figure 2 shows how instruction encodings are defined in Pydgin using a user-friendly Python list of instruction name and a bitmask consisting of `0`, `1`, or `x` (for not care). Using this list, the Pydgin framework automatically generates an instruction decoder. When encountering a match, the decoder calls an RPython function with the name `execute_<inst_name>`. Figure 3 shows examples of execute functions that implement the instruction semantics. The execute functions are plain Python functions with two arguments: the state object and an instruction object that provides convenient access to various instruction fields such as the immediate values. The execute function can access and manipulate the fields in the state and instruction to implement the expected behavior.

```
encodings = [
  # ...
  [ 'xori',    'xxxxxxxxxxxxxxxxx100xxxxx0010011' ],
  [ 'ori',     'xxxxxxxxxxxxxxxxx110xxxxx0010011' ],
  [ 'andi',    'xxxxxxxxxxxxxxxxx111xxxxx0010011' ],
  [ 'slli',    '000000xxxxxxxxxxx001xxxxx0010011' ],
  [ 'srli',    '000000xxxxxxxxxxx101xxxxx0010011' ],
  [ 'srai',    '010000xxxxxxxxxxx101xxxxx0010011' ],
  [ 'add',     '0000000xxxxxxxxxx000xxxxx0110011' ],
  [ 'sub',     '0100000xxxxxxxxxx000xxxxx0110011' ],
  [ 'sll',     '0000000xxxxxxxxxx001xxxxx0110011' ],
  # ...

  # new instruction encodings can be added by for
  # example re-using custom2 primary opcode reserved
  # for extensions
  #['custom2', 'xxxxxxxxxxxxxxxxx000xxxxx1011011' ],
  [ 'gcd',     'xxxxxxxxxxxxxxxxx000xxxxx1011011' ],
]
```

Figure 2: RISC-V Instruction Encoding – Examples of instruction encodings showing some of the integer subset instructions. Encodings are defined using the instruction name and a bitmask consisting of `0`, `1`, and `x` (for not care). Reserved primary opcodes such as `custom2` can be used for instruction set extensions.

The Pydgin framework provides the remaining ISA-independent components to complete the ISS definition. In addition to providing various templates for storage data structures, bit manipulation, system-call implementations, and ELF-file loading facilities, the framework most importantly includes the simulation loop. Figure 4 shows a simplified version of the simulation loop, which includes the usual fetch, decode, and execute calls. The Pydgin framework is designed so that ISA implementers and users would not need to modify the framework for most use cases. Since the ADL and the framework are written in RPython (a valid subset of Python), popular interpreters, debugging, and testing tools for Python can be used out of the box for Pydgin. This makes Pydgin development highly productive. Pydgin running on top of a Python interpreter is unsurprisingly very slow (around 100 thousand instructions per second), so this usage should be limited to testing and debugging with short programs.

The performance benefits of Pydgin comes from using the RPython translation toolchain. RPython is a typeable subset of Python, and given an RPython source (e.g., PyPy or Pydgin), the RPython translation toolchain performs type inference, optimization, and code generation steps to produce a statically typed C-language translation of the interpreter. This C-language interpreter can be compiled using a standard C compiler to produce a native binary of an interpretive ISS. This interpretive version of Pydgin manages to reach 10 MIPS, which is much faster than interpreting Pydgin on top of a Python interpreter. However, the real strength of the RPython translation toolchain comes from its ability to couple a JIT compiler alongside the interpreter to be translated. The translated interpreter and the JIT compiler are compiled together to produce a DBT-based interpreter. The DBT-based interpreter can have significant speedups over the one without DBT by optimizing for the common paths and removing unnecessary computation. However, without communicating to the framework what are the common paths, usually-true conditions, and always-true conditions, enabling the JIT compiler

```
1 # add-immediate semantics
2 def execute_addi( s, inst ):
3   s.rf[inst.rd] = s.rf[inst.rs1] + inst.i_imm
4   s.pc += 4
5   # count number of adds
6   s.num_adds += 1
7
8 # store-word semantics
9 def execute_sw( s, inst ):
10   addr = trim_xlen( s.rf[inst.rs1] + inst.s_imm )
11   s.mem.write( addr, 4, trim_32( s.rf[inst.rs2] ) )
12   s.pc += 4
13   # count misaligned stores
14   if addr % 4 != 0:  s.num_misaligned += 1
15
16 # branch-equals semantics
17 def execute_beq( s, inst ):
18   old_pc = s.pc
19   if s.rf[inst.rs1] == s.rf[inst.rs2]:
20     s.pc = trim_xlen( s.pc + inst.sb_imm )
21     # record and count all executed loops
22     if s.pc <= old_pc:  s.loop_dict[(s.pc, old_pc)] += 1
23   else:
24     s.pc += 4
25
26 # extension example: greatest common divisor
27 def execute_gcd( s, inst ):
28   a, b = s.rf[inst.rs1], s.rf[inst.rs2]
29   while b:
30     a, b = b, a%b
31   s.rf[inst.rd] = a
32   s.pc += 4
```

Figure 3: RISC-V Instruction Semantics – Examples of three RISC-V instructions: `addi`, `sw`, and `beq`; and `gcd` as an example of instruction-set extension implementing the greatest common divisor algorithm. Instructions in gray are examples of custom instrumentation that can be added and are optional.

```
1 def instruction_set_interpreter( memory ):
2   state = State( memory )
3   while True:
4     inst    = memory[ state.pc ] # fetch
5     execute = decode( inst )     # decode
6     execute( state, inst )       # execute
```

Figure 4: Simplified Pydgin Interpreter Loop

usually causes a slowdown due to additional overheads. The assumptions and hints about the interpreter are added in the form of annotations. Examples of these hints include the location in the interpreted code (the PC), when a "loop" happens in the interpreted code (backwards branches), and constant PC-to-instruction guarantee for non-self-modifying code. These hints are used by the JIT generator in the RPython translation toolchain to optimize away unnecessary computation, primarily for instruction fetch and decode and produces a DBT-based ISS. Please see [8] for more details about the particular optimizations we have used in Pydgin. The DBT-based Pydgin ISS manages to improve the performance to 100s of MIPS. Section 4 describes the performance results of Pydgin running large RISC-V benchmarks.

3. Pydgin Productivity

Pydgin uses the RPython subset of standard Python in its ADL, and can use off-the-shelf Python testing and debugging tools. This makes developing implementations for new ISAs productive. Even though it is hard to quantify these productivity benefits, as a rough anecdote, the first two authors of this paper added RISC-V support to Pydgin over the course of nine days, three of which were during a busy conference. Implementing a simple (and slow) ISS in this duration is possible, however implementing one that achieves 100+ MIPS of performance is a testament to the productivity and performance features of Pydgin.

Facilitating domain-specific extensions to the existing ISA is a strength of RISC-V. These extensions require adding the encoding and semantics of the new instructions to the ISS, and the software stack to take advantage of these new instructions. Pydgin makes the ISS-side modifications productive. Line 18 in Figure 2 shows an example of adding a new domain-specific greatest common divisor (`gcd`) instruction by reusing the `custom2` primary opcode reserved for extensions such as this. Lines 26–32 in Figure 3 show adding the semantics for the `gcd` instruction. Note that this particular example does not require any new architectural state to be added.

Adding custom instrumentation productively to the ISS is an important feature of Pydgin. Hardware and software designers are often interested in how well a new ISA or instruction-set extension performs running realistic benchmarks. Software designers would be interested in knowing how often these new instructions are used to ensure the software stack is fully making use of the new features. Hardware designers would be interested in knowing how software tends to use these new instructions, with which arguments, in order to design optimized hardware for the common case. Custom instrumentation in Pydgin often involves adding a new state variable as shown between Lines 14–23 in Figure 1. Here, we illustrate adding counters or more sophisticated data structures to count number of executed instructions, number of executed addition instructions, number of misaligned memory operations, and a histogram of all loops in the program. Figure 3 shows (in gray) how the new custom instrumentation can be added directly in the instruction semantics with only a few lines of code.

Productive development, extensibility, and instrumentation in Pydgin are complemented with the performance benefits of JIT compilation. Most of the JIT annotations necessary to use the full power of the RPython JIT are in the Pydgin framework, so the additions of the ADL automatically make use of these. The additional instrumentation code will automatically be JIT compiled. However, depending on the computational intensiveness and frequency of the instrumentation code, there will be a graceful degradation in performance.

We use Pydgin extensively in our research group. We are often interested in different program phases and how do each of these phases perform on a new hardware design. To enable phase-specific instrumentation, we can extend our benchmarks to communicate with Pydgin the currently executing phase and leverage Pydgin's support for per-phase statistics. We have also used Pydgin to experiment with instruction-set specializations for novel algorithm and data-structure accelerators. For another project, we have used Pydgin to keep track of control- and memory-divergence statistics of SIMD and vector engines with different hardware vector lengths. We

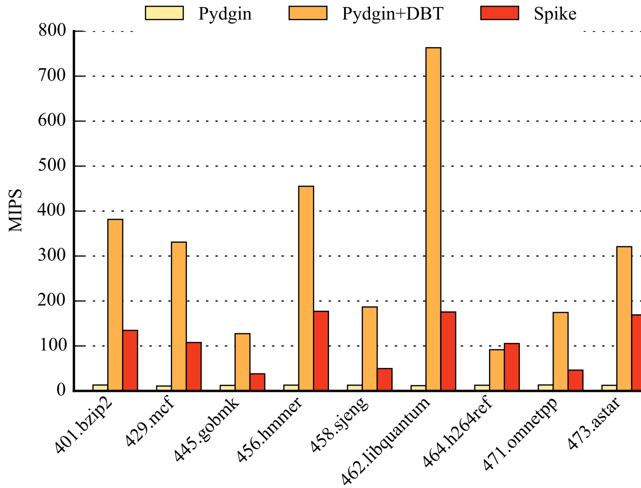

Figure 5: Performance Results – Pydgin without DBT, Pydgin with DBT, and Spike running SPEC CINT2006 benchmarks.

have also used Pydgin to generate basic block vectors used by the SimPoint algorithm [16] to determine samples to be simulated in detail using a cycle-level microarchitectural simulator. Finally, we have used Pydgin to study interpreter, tracing, garbage collection, and JIT-optimized phases of dynamic language interpreters.

4. Pydgin Performance

We evaluated the performance of Pydgin, with and without enabling the JIT, on SPEC CINT2006 benchmarks. We used a Newlib-based GCC 4.9.2 RISC-V cross-compiler to compile the benchmarks. We used system-call emulation in Pydgin to implement the system calls. Some of the SPEC CINT2006 benchmarks (`400.perlbench`, `403.gcc`, and `483.xalancbmk`) did not compile due to limited system-call and standard C-library support in Newlib, so we omitted these. Figure 5 shows the performance results. As expected, Pydgin without JIT achieves a relatively low performance of 10 MIPS across the benchmarks. The performance is fairly consistent regardless of the benchmark, and this is a characteristic of interpreter-based ISSs. Turning on the JIT in Pydgin increases the performance to 90–750 MIPS range. This performance point is high enough to allow running the reference datasets of these benchmarks (trillions of instructions) over a few hours of simulation time. Different patterns in the target code is responsible for the highly varied performance, and this is a characteristic of DBT-based ISSs.

To compare the performance of Pydgin to other ISSs for RISC-V, Figure 5 shows the performance of Spike, the reference RISC-V ISS. Despite being an interpreter-based ISS, Spike manages to achieve 40–200 MIPS, much faster than non-DBT Pydgin. This is because Spike is heavily optimized to be a very fast interpretive ISS, and employs some optimization techniques that are typical of DBT-based interpreters. Examples of these optimizations include a software instruction cache that stores pre-decoded instructions, a software TLB, and an unrolled PC-indexed interpreter loop to improve host branch prediction. These optimizations also cause the performance to be similarly varied.

At the time of writing, the QEMU [2] port of RISC-V was not available. We anticipate the eventual QEMU port will be faster than Pydgin, however the fact that an up-to-date QEMU RISC-V port does not yet exist is indicative of lack of productivity in performance-oriented ISSs. Pydgin's strength lies in combining productivity and performance.

5. Conclusions

We have introduced Pydgin from the perspective of its RISC-V support and how it answers the unique requirements of the RISC-V community. Extensibility is a major feature in RISC-V for the domain-specific needs of its users. This makes it necessary for the RISC-V ISSs to be highly productive to develop, extend, and instrument, while also high-performance to allow real-world benchmarks to be simulated in reasonable amount of time. We believe Pydgin is the right tool to uniquely combine productivity and performance and be of use to many users of the RISC-V community.

Acknowledgments

This work was supported in part by NSF XPS Award #1337240, NSF CRI Award #1512937, NSF SHF Award #1527065, DARPA Young Faculty Award, and donations from Intel Corporation, Synopsys, Inc, and Xilinx, Inc. We would also like to thank Yunsup Lee for his initial help in developing RISC-V support for Pydgin.

References

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

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] D. Lockhart, B. Ilbeyi, and C. Batten. Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-Tracing JIT Compilers. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Mar 2015.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] T. Sherwood et al. Automatically Characterizing Large Scale Program Behavior. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Feb 2002.

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

[18] 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.

[19] 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.

[20] A. Waterman et al. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0. Technical report, EECS Department, University of California, Berkeley, May 2014.

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