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], SimT-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, 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

1Pydgin 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.
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 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 (green), and which variables are not part of the position key (red). 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 bytecode. 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 “jicodes” 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 jicode 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
using normal Python functions with the special signature

C. Instruction Semantics

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 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-

for simItARM in ...
tom DSL approach: increased language familiarity and a generating a DSL which uses a custom parser. From our experience, comparatively cleaner syntax while also providing compatibility 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 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 fields should be stored in memory. These worst-case assumptions reduce opportunities for JIT optimization and thus 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.

Figure 9. ADD Instruction Semantics: ArchC

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
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 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.
Etidable 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 etidable" 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. Etidable decode can potentially eliminate 20 JIT IR nodes on lines 6–18 in Figure 11. The combination of etidable instruction fetch and etidable 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 etidable instruction fetch and etidable 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 5x on average and up to 25x 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 addr 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 2x 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

![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.]
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, 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 to 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 its 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.

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.onnetpp 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.
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
outperform pydgin-arm-nojit by a factor of $2 \times$ 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 $1.44–1.52 \times$ 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 retransliteration, 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 Fijalkowski 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
