Appears in the Proceedings of the 2017 IEEE Int'l Symp. on Workload Characterization (IISWC), October 2017

Cross-Layer Workload Characterization of Meta-Tracing JIT VMs

Berkin Ilbeyi[1], Carl Friedrich Bolz-Tereick[2], and Christopher Batten[1]
[1] School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
[2] Heinrich-Heine-Universität Düsseldorf, Germany
bi45@cornell.edu, cfbolz@gmx.de, cbatten@cornell.edu

***Abstract*—Dynamic programming languages are becoming increasingly popular, and this motivates the need for just-in-time (JIT) compilation to close the productivity/performance gap. Unfortunately, developing custom JIT-optimizing virtual machines (VMs) requires significant effort. Recent work has shown the promise of meta-JIT frameworks, which abstract the language definition from the VM internals. Meta-JITs can enable automatic generation of high-performance JIT-optimizing VMs from high-level language specifications. This paper provides a detailed workload characterization of meta-tracing JITs for two different dynamic programming languages: Python and Racket. We propose a new cross-layer methodology, and then we use this methodology to characterize a diverse selection of benchmarks at the application, framework, interpreter, JIT-intermediate-representation, and microarchitecture level. Our work is able to provide initial answers to important questions about meta-tracing JITs including the potential performance improvement over optimized interpreters, the source of various overheads, and the continued performance gap between JIT-compiled code and statically compiled languages.**

I. Introduction

Dynamic programming languages are growing in popularity across the computing spectrum from smartphones (e.g., JavaScript for mobile web clients), to servers (e.g., Node.js, Ruby on Rails), to supercomputers (e.g., Julia for numerical computing). Four out of the top-ten most popular programming languages are now dynamic [12]. Dynamic languages typically include: lightweight syntax; dynamic typing of variables; managed memory and garbage collection; rich standard libraries; interactive execution environments; and advanced introspection and reflection capabilities. The careful use of these features can potentially enable more productive programming.

However, the very features that make dynamic languages popular and productive also result in lower performance. These languages traditionally use interpreters to implement a virtual machine that closely aligns with the language semantics, but interpreted code can be orders-of-magnitude slower than statically compiled code. To address this performance/productivity gap, dynamic languages are using just-in-time (JIT) optimizing virtual machines (VMs) to apply traditional ahead-of-time compiler techniques at run-time [6,10,17,22,24,26,43,46,47]. Such JIT optimizations include removing the (bytecode) fetch and decode overhead, generating type-specialized code for the observed types, (partial) escape analysis [5,39], constant propagation, and dead-code elimination. It is well known that developing state-of-the-art JIT-optimizing VMs is very challenging due to the overall complexity (e.g., the need for profiling, recording a trace, compiling, deoptimizations on type misspeculation, garbage collection, etc.), performance requirements (e.g., the JIT optimization process itself must be fast since it is on the critical path), and development process (e.g., debugging dynamic code generation). With many programmer-decades of engineering effort, some JIT-optimizing VMs (e.g., Google V8 [22,43], Mozilla IronMonkey [24]) can begin to achieve performance that is within 10× of statically compiled code.

Unfortunately, many emerging and experimental languages simply cannot afford the effort required to implement a custom state-of-the-art JIT-optimizing VM. This has motivated work on meta-JIT optimizing VMs (or "meta-JITs") which abstract the language definition from the VM internals such that language implementers need not worry about the complexity typically associated with JIT optimizations. There are currently two production-ready meta-JITs: the Truffle framework for rapidly developing method-based JIT-optimizing VMs [45–47] and the RPython framework for rapidly developing trace-based JIT-optimizing VMs [6, 10] (see [29] for a detailed comparison of these frameworks). The *Truffle framework* enables language implementers to define abstract-syntax-tree- (AST-) based interpreters for their language and also specify JIT-optimization opportunities to the Graal compiler. Truffle automatically identifies "hot" target-language methods and then applies the previously specified JIT optimizations along with aggressive partial evaluation before targeting the HotSpot JVM. The *RPython framework* enables language implementers to build AST- or bytecode-based interpreters in a high-level language. An interpreter communicates to the framework its dispatch loop, target-language loops, and additional run-time hints. RPython automatically identifies "hot" target-language loops and then generates a trace, optimizes the trace, and lowers the trace into assembly. Meta-JITs can significantly reduce the effort involved in building JIT-optimizing VMs, and so it is not surprising that many language interpreters are now either using or experimenting with meta-JITs (e.g., Python [6], Ruby [37, 40, 42], JavaScript [47], R [41], Racket [9], PHP [20], Prolog [8], Smalltalk [7]). To narrow the scope of this work, we focus on the RPython meta-tracing JIT, and Section II provides additional background on this framework.

We anticipate the trend towards meta-JITs will continue for new, research, and domain-specific languages, and this motivates our interest in performing a multi-language workload characterization of the RPython meta-tracing JIT. Section III describes our baseline characterization methodology. One of the key challenges in performing such a characterization is the many layers of abstraction used in a meta-tracing JIT including: the target dynamic programming language; the target language AST or bytecode; the high-level language used to implement the interpreter; the intermediate representation (IR) used in the meta-trace; and the final assembly instructions. In Section IV, we describe a new cross-layer characterization methodology that enables inserting cross-layer annotations at a higher

layer, and then intercepting these annotations at a lower layer. In Section V, we use this new cross-layer methodology to characterize a diverse selection of benchmarks written in Python and Racket at the application, framework, interpreter, JIT-IR, and microarchitecture level. In Section VI, we use this characterization data to answer the following nine key questions:

1. Can meta-tracing JITs significantly improve the performance of multiple dynamic languages?
2. Does generating and optimizing traces in a meta-tracing JIT add significant overhead?
3. Does deoptimization in a meta-tracing JIT consume a significant fraction of the total execution time?
4. Does garbage collection in a meta-tracing JIT consume a significant fraction of the total execution time?
5. Is all JIT-compiled code equally used in a meta-tracing JIT?
6. Does a meta-tracing JIT need to spend most of its time in the JIT-compiled code to achieve good performance?
7. What fraction of the time in the JIT-compiled code is overhead due to the meta-tracing JIT?
8. What is the microarchitectural behavior (e.g., instruction throughput, branch prediction) of JIT-compiled code?
9. Why are meta-tracing JITs for dynamic programming languages still slower than statically compiled languages?

This paper makes three technical contributions: (1) we describe a new cross-layer methodology that enables detailed characterization of meta-JITs; (2) we present a cross-layer workload characterization of the RPython meta-tracing JIT for two different programming languages; and (3) we provide initial answers to the above nine questions. Our work can help the broader community and meta-JIT developers understand the difference between traditional JITs and meta-JITs and focus on key performance bottlenecks.

II. Background on Meta-Tracing and RPython

The RPython toolchain uses a novel approach where the tracing JIT compiler is actually a *meta-tracing JIT compiler*, meaning that the JIT does not directly trace the application, but instead traces the *interpreter* interpreting the application. The interpreter is written in RPython, a statically typed subset of Python. The interpreter uses a dispatch loop to continuously fetch a quantum of execution (e.g., bytecode) and dispatch to a corresponding execution function. The RPython framework has its own standard library (similar to Python's standard library) and an API so that the language implementer can inform the framework of the interpreter's program counter, dispatch loop, and application-level loops. The framework also supports hints to indicate which variables can be treated as constants in the trace, which interpreter-level functions are pure, and when type-specialized versions of functions should be generated.

Since RPython is a proper subset of Python, interpreters written in RPython can be executed using a standard Python interpreter. However, for good performance, these interpreters are automatically translated into C using the RPython translation toolchain (see dashed lines in Figure 1). At run-time, the application (e.g., Python code) is compiled into bytecode, and the bytecode executes on the C-based interpreter (see solid lines

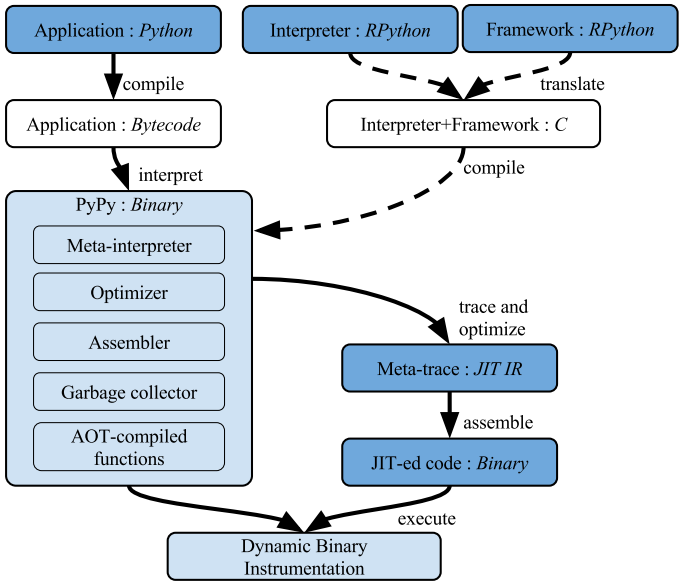

Figure 1. RPython Framework – Dashed lines indicate ahead-of-time operations; solid lines indicate execution-time operations. The language each block is written in is indicated after the colon. Dark blue blocks are where we can insert cross-layer annotations, and light blue blocks are where we can intercept cross-layer annotations.

in Figure 1). When the interpreter reaches an application-level loop, the framework increments an internal per-loop counter. Once this counter exceeds a threshold, the execution is transfered to a meta-interpreter. The meta-interpreter builds the meta-trace by recording the operations performed by the interpreter until the application-level loop is completed. The trace is then passed on to a JIT optimizer and assembler before initiating native execution. A meta-trace includes 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 a guard fails or if the optimized loop is finished, the JIT returns control back to the C-based interpreter using a process called *deoptimization*. Deoptimization transforms the intermediate state in a JIT-compiled trace to the precise state required to start execution of the interpreter. If a guard fails often, it is converted into a *bridge*, which is a direct branch from one JIT-compiled trace to another, separately JIT-compiled trace. More on the RPython toolchain can be found in [1, 4, 36].

Meta-tracing JITs have some important differences compared to traditional tracing JITs. Tracing JITs trace the execution of the target program directly (e.g., recording the trace of executed bytecodes), while meta-tracing JITs trace the execution of the interpreter as it executes the target program. These extra levels of abstraction can potentially result in increased overhead during tracing, deoptimization, garbage collection, and JIT-compiled execution. Also note that the meta-interpreter opportunistically inlines interpreter-level function calls. However, if these functions contain loops with data-dependent bounds, they are excluded from the trace to avoid numerous guard failures. These functions are ahead-of-time- (AOT-) compiled (and so not dynamically optimized) and then called from within the JIT-compiled code. While traditional tracing JITs may also include calls to the JIT framework from within JIT-compiled

code, there is the potential for many more calls in a meta-tracing JIT because such calls are particularly easy to use in RPython.

III. Baseline Characterization Methodology

To characterize meta-tracing JITs across multiple languages, we will study Python (a well-known, general-purpose programming language) and Racket (a general-purpose programming language in the Lisp-Scheme family). As a baseline, we will use the "reference" interpreters for both Python (i.e., CPython) and Racket. Technically, Racket does not use an interpreter but instead uses a custom JIT-optimizing VM. Both Python and Racket also have highly optimized meta-tracing JITs implemented using the RPython framework. For Python, we will use the very popular PyPy meta-tracing JIT [6] and for Racket, we will use the Pycket meta-tracing JIT [9]. For each language, we will also explore an interpreter generated from the RPython translation toolchain but without the meta-tracing JIT enabled as another example of a traditional interpreter. Finally, we also include results for C/C++ implementations of some benchmarks to provide a reference for statically compiled languages.

We use benchmarks from two suites: the PyPy Benchmark Suite [31] and the Computer Languages Benchmarks Game (CLBG) [19]. We use the PyPy Benchmark Suite because it is widely used for benchmarking Python programs [5, 36]. We use the CLBG to compare the performance of different languages and interpreters on highly optimized implementations of the same benchmarks. The benchmarks in the PyPy Benchmark Suite are single-threaded and single-process, whereas many implementations in the CLBG make use of multi-programming and multi-threading. CPython and the RPython framework currently make use of a Global Interpreter Lock (GIL) [3], preventing parallelism in multi-threaded programs. Because the focus of the paper is on the performance characterization of meta-tracing JITs, and not of GILs and parallelism, we restrict all benchmarks, including many parallel implementations in the CLBG, to use a single hardware thread. Because CLBG provides multiple implementations of the same benchmark and language, we pick the fastest implementation for each benchmark and language combination. Due to some missing features of the Racket language in Pycket, a number of the CLBG benchmarks did not execute correctly.

When characterizing at the JIT IR level, we make use of the PyPy Log facility, which is part of the RPython framework. The PyPy Log contains information about each JIT-compiled trace including the bytecode operations, JIT IR nodes, and assembly instructions contained in each trace, along with the number of times each trace was executed. Enabling the PyPy Log slightly degrades performance (<10%), so we disabled this feature when comparing the overall execution time of the interpreters.

We use two mechanisms to collect microarchitectural measurements through performance counters. The first uses the performance application programming interface (PAPI) to record performance counters on certain cross-layer annotations [30]. We implemented this mechanism in the RPython-based interpreters. To compare microarchitectural characteristics of meta-tracing to other interpreters and statically compiled code, we use Linux's `perf` tool to periodically read the performance counters.

IV. Cross-Layer Characterization Methodology

The baseline characterization methodology described in Section III can enable initial analysis, but the many layers involved in a meta-tracing JIT make it difficult to gain insight into cross-layer interactions between: the target dynamic programming language; the target language AST or bytecode; the interpreter including the RPython standard library; the JIT IR used in the meta-trace; and the final assembly instructions. In this section, we describe a new cross-layer characterization methodology based on using *cross-layer annotations*.

Cross-layer annotations are a unified mechanism to annotate events of interest at one level of meta-tracing execution, and collect these annotations at different level. For instance, a Python application might annotate when a particular function is called, the Python interpreter might annotate every time the dispatch loop is executed, the RPython framework might annotate when loops are being traced or when garbage collection occurs, and generated machine code from the JIT compiler might annotate when a particular assembly sequence corresponding to a particular IR node is being executed. Figure 1 shows the blocks where the cross-layer annotations can be inserted in dark blue.

These annotations can be collected at different levels. At the assembly instruction level, annotations can be observed by using a carefully selected instruction which does not change the program behavior but also includes a tag to indicate the corresponding annotation. Our methodology uses the `nop` instruction in the x86 ISA. Although the `nop` instruction supports the usual addressing modes of the x86 ISA, the architecture only considers the opcode and ignores the corresponding address. Our methodology uses a unique address to serve as the tag for each cross-layer annotation. Other ISAs can use other instructions or sequences of instructions to achieve a similar effect (e.g., `add r1, r1, #145`; `sub r1, r1, #145` in ARM). The execution target that executes these machine instructions (e.g., a dynamic instrumentation tool, an ISA simulator, or a soft-core processor on an FPGA) can perform some action when one of these cross-layer annotations is executed. Cross-layer annotations can also be collected at higher levels. For example, the timestamps or microarchitectural measurements can be logged to a file every time a cross-layer annotation is called. Figure 1 shows the blocks where the cross-layer annotations can be collected in light blue.

In this paper, we have modified the RPython framework and inserted cross-layer annotations at various events of interest in the framework (e.g., when a minor garbage collection starts and ends, when tracing starts and ends, when execution starts on the JIT-compiled code, etc.). This allows us to know exactly what the framework is doing at a given point of time. We can use this information to get detailed breakdowns of time spent in different parts of the framework. We also added cross-layer annotations at the interpreter level at the beginning of the dispatch loop. This allows us to have an independent measure of "work" (e.g., number of bytecodes in PyPy) regardless of whether the interpreter is being used (if the JIT is off or has not warmed up yet), the tracing interpreter is being used, or the JIT-compiled code is being executed. This enables precisely finding the JIT warmup break-even point. Finding the break-even point using other techniques is likely very difficult because counting

the number of bytecodes executed in the JIT would likely introduce significant performance overheads that would skew the results. We added application-level APIs to our interpreters so that cross-layer hints can also be emitted from the application level. Finally, we can also emit cross-layer annotations when each JIT IR node is lowered to assembly instructions. This enables tracking the connection between traces, JIT IR nodes, and assembly instructions. Each cross-layer annotation can be enabled/disabled from the command line.

We use the Pin dynamic binary instrumentation tool [28] as the primary mechanism for intercepting cross-layer annotations. We have developed a custom PinTool that detects the `nop` instructions and can track information on the phase, the bytecode execution rate, AOT-compiled functions, and JIT IR nodes.

V. Cross-Layer Workload Characterization

This section presents cross-layer workload characterization studies that will help us answer the key questions regarding meta-tracing JITs.

A. Application-Level Characterization

We first compare the overall application performance of dynamic languages running on different VMs. Table I compares the application performance of the PyPy Benchmark Suite using CPython, PyPy without a meta-tracing JIT, and PyPy with a meta-tracing JIT. We can see that CPython is consistently faster than PyPy with its meta-tracing JIT disabled for almost all of the benchmarks, usually by 2× or more. The first reason for this is because CPython is written directly in C, whereas PyPy is written in a high-level language (RPython) translated to C. The other reason is because CPython is designed to only be an interpreter, and thus it includes some modest interpreter-focused optimizations. The performance benefit of PyPy with the meta-tracing JIT over CPython is much more varied and usually much higher: from 0.7–51×.

To compare different languages, Table II shows the overall execution time of the CLBG benchmarks. Similar trends hold here between PyPy with the meta-tracing JIT enabled and CPython, except for a few cases (`chameneosredux`, `pidigits`, `revcomp`) where CPython performs much better. These Python programs use external libraries, which are often written in C using an API that exposes CPython implementation details. There is ongoing work on a PyPy C-compatibility layer which could enable similar performance benefits. We see that the other RPython-based meta-tracing JIT, Pycket, has similar performance as Racket with a range of 0.3–2×. This is due to: (1) Pycket is less mature compared to PyPy; and (2) unlike CPython, Racket uses a custom JIT-optimizing VM. Racket- and Python-language implementations, even with a meta-tracing JIT, tend to perform very poorly compared to C and C++.

B. Framework-Level Characterization: Phases

Tracing JITs typically have different phases inherent to the way they execute and optimize code. Initially, execution starts in the *interpreter phase*. When hot loops are detected, these loops are traced and compiled in the *tracing phase*. The JIT-compiled code is executed during the *JIT phase*. Occasionally, there are calls to AOT-compiled functions in the runtime from

Table I. PyPy Benchmark Suite Performance

	CPython			PyPy w/o JIT				PyPy with JIT			
Benchmark	**t (s)**	**IPC**	**M**	**t (s)**	**vC**	**IPC**	**M**	**t (s)**	**vC**	**IPC**	**M**
richards	0.2	1.65	5.9	0.5	0.5	1.32	6.4	0.004	51.2	1.38	3.5
crypto_pyaes	2	1.94	3.1	4	0.4	1.55	3.2	0.06	30.2	1.62	0.8
chaos	0.3	1.49	5.5	0.7	0.4	1.03	6.7	0.01	27.2	1.31	1.9
telco	0.9	1.24	7.4	2	0.4	0.88	7.3	0.03	27.1	1.11	4.0
spectral-norm	0.3	1.93	3.3	0.9	0.3	1.44	3.6	0.01	25.9	1.90	0.8
django	0.7	1.24	5.7	1	0.5	0.88	6.6	0.04	18.2	1.37	2.5
twisted_iteration	0.09	1.41	4.5	0.4	0.2	0.95	7.1	0.006	15.0	1.27	0.8
spitfire_cstringio	10	1.92	1.4	9	1.1	1.46	3.0	0.9	11.4	2.00	0.5
raytrace-simple	2	1.54	5.4	4	0.5	1.13	5.8	0.2	10.4	1.22	2.7
hexiom2	149	1.88	2.5	442	0.3	1.38	4.2	14	10.1	1.91	1.2
float	0.4	1.62	2.5	0.8	0.5	1.22	4.6	0.05	7.1	1.38	2.9
ai	0.3	1.44	3.7	1	0.3	1.05	4.7	0.04	7.0	1.79	1.6
nbody_modified	0.3	2.06	2.9	0.9	0.3	1.58	3.0	0.04	6.9	1.50	1.1
twisted_pb	0.05	1.12	7.9	0.1	0.4	0.86	7.0	0.007	6.4	0.68	4.1
fannkuch	1	1.70	3.8	2	0.6	1.36	6.5	0.2	5.2	1.59	5.5
genshi_text	0.1	1.29	6.2	0.4	0.3	0.94	7.1	0.02	5.2	1.30	1.8
pyflate-fast	2	1.68	4.4	4	0.5	1.29	5.6	0.4	4.8	1.62	2.4
bm_mako	0.1	1.46	2.3	0.3	0.4	0.89	5.1	0.02	4.8	1.41	2.3
twisted_names	0.008	0.74	13.9	0.02	0.5	0.67	9.5	0.002	4.1	0.51	9.4
json_bench	3	1.54	4.6	31	0.1	1.17	6.2	0.9	3.9	1.91	0.7
genshi_xml	0.2	1.11	7.6	0.8	0.3	0.78	8.6	0.06	3.9	1.09	1.4
bm_chameleon	0.07	1.35	5.4	0.2	0.4	1.05	5.6	0.02	3.5	1.39	2.5
pypy_interp	0.3	1.15	7.0	0.6	0.5	0.89	6.5	0.1	3.3	0.91	6.4
twisted_tcp	0.6	0.68	10.3	1	0.5	0.54	9.2	0.2	3.0	0.48	3.4
html5lib	11	0.93	9.7	27	0.4	0.77	7.0	4	2.5	0.89	4.6
meteor-contest	0.2	1.51	7.0	0.6	0.4	1.32	3.4	0.1	2.4	1.64	3.9
sympy_sum	1	1.25	5.6	5	0.3	0.80	6.6	0.6	2.3	0.99	5.6
spitfire	5	1.82	2.4	12	0.4	1.37	2.7	2	2.1	1.55	1.1
spambayes	0.2	1.59	4.3	0.6	0.4	1.35	4.2	0.1	2.0	0.99	6.4
rietveld	0.6	0.99	9.1	1	0.5	0.76	7.8	0.3	1.8	0.76	8.4
deltablue	0.02	1.74	3.5	0.05	0.4	1.25	4.2	0.01	1.7	0.98	6.1
eparse	0.8	1.35	5.5	1	0.6	0.96	6.3	0.5	1.5	0.77	5.8
sympy_expand	1	1.17	7.1	3	0.4	0.83	6.7	0.8	1.4	0.86	6.4
slowspitfire	0.4	1.86	2.1	1	0.4	1.52	2.2	0.3	1.3	1.39	0.5
sympy_integrate	4	1.30	5.5	15	0.2	0.88	6.3	3	1.2	0.86	6.8
pidigits	12	2.45	0.1	11	1.1	1.83	0.1	10	1.1	1.84	0.1
bm_mdp	10	1.34	8.6	70	0.2	1.19	5.2	10	1.1	1.39	2.1
sympy_str	0.5	1.14	7.7	1	0.4	0.83	6.9	0.7	0.7	0.88	7.2
Average		1.46	5.4		**0.5**	1.10	5.6		**1.7**	1.27	3.4

Overall execution times ordered by PyPy with JIT speedup over CPython. vC = speedup compared to CPython. IPC = instructions per cycle. M = branch misses per 1000 instructions.

Table II. CLBG Performance

	C	CPython		PyPy		Racket		Pycket	
Benchmark	**IPC**	**Sdn**	**IPC**	**Sdn**	**IPC**	**Sdn**	**IPC**	**Sdn**	**IPC**
binarytrees	2.16	37	1.95	4.5	1.37	5.7	1.80	11	1.26
chameneosredux	1.19	87	1.13	1374	0.86	111	0.99	–	–
fannkuchredux	1.16	89	1.85	25	1.43	13	1.97	7.2	1.16
fasta	1.89	8.3	1.93	5.6	1.23	2.0	1.09	2.1	1.09
knucleotide	1.74	15	1.69	8.8	1.65	4.4	2.20	–	–
mandelbrot	1.99	115	2.23	29	1.29	9.0	1.52	6.7	1.21
meteor	1.18	78	1.78	30	1.04	12	1.63	31	1.06
nbody	2.23	97	2.16	12	1.33	5.3	1.68	2.8	1.20
pidigits	1.65	1.8	1.48	5.1	0.96	13	0.93	7.4	1.89
regexdna	1.36	3.8	1.78	3.1	1.12	6.7	2.01	–	–
revcomp	1.43	4.0	1.96	6.8	1.35	5.1	2.10	4.0	1.21
spectralnorm	1.20	104	1.97	10	1.25	6.4	1.56	4.2	1.16
threadring	1.03	20	0.97	16	0.94	16	0.82	–	–

IPC and slowdowns (Sdn) compared to C/C++. Meta-tracing JIT is enabled for PyPy and Pycket.

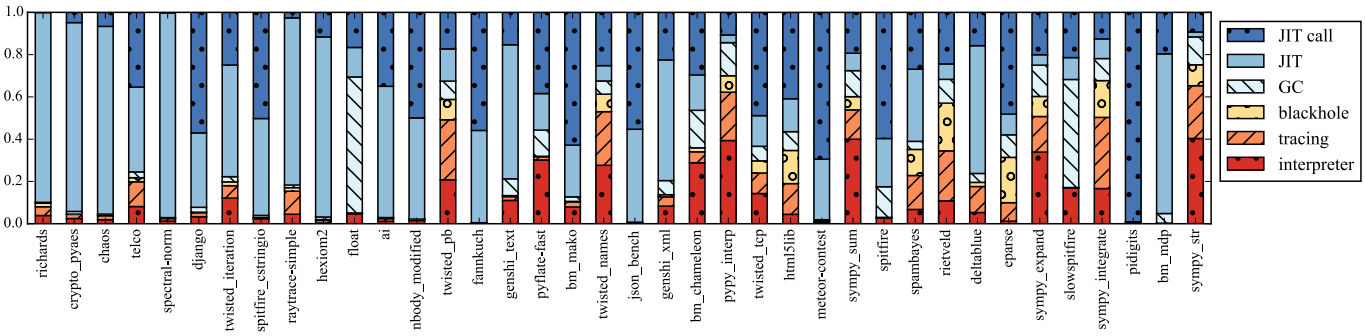

Figure 2. RPython Framework Breakdown – Shows the breakdown of time spend in various components of the RPython framework.

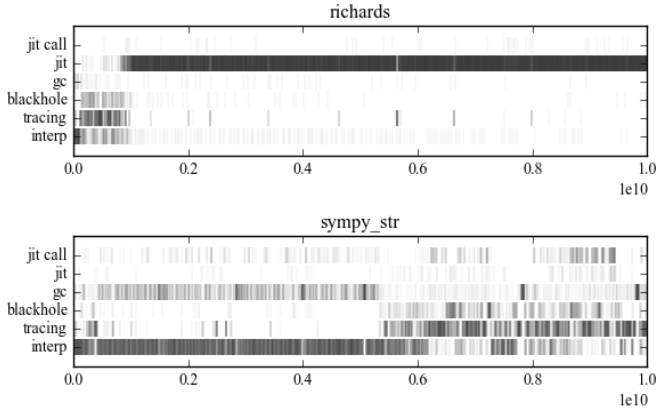

Figure 3. Framework Phases in Time – Each line indicates which phase the framework currently is in during the first 10 billion instructions of the best-performing (`richards`) and worst-performing (`sympy_str`) benchmarks.

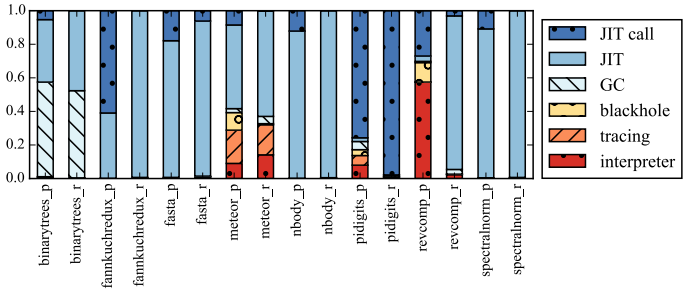

Figure 4. RPython Framework Breakdown – Shows the breakdown of time spent in various components of the RPython framework in CLBG. The PyPy and Pycket implementations are suffixed with `_p` and `_r` respectively.

JIT-compiled code, which is the *JIT call phase*. Finally, there is the *GC* phase for garbage collection and the *blackhole* phase deoptimization (due to the "blackhole interpreter" used to implement deoptimization in RPython).

Using cross-layer annotations and a custom PinTool that intercepts them, we can tease apart how much of the execution time is spent in each of these phases. Figure 3 shows the phases for the best- and worst-performing (compared to CPython) benchmarks. As expected, the framework initially spends most of its time in the interpreter, tracing, and blackhole phases until the meta-tracing JIT warms up, then time in JIT phase dominates. Interestingly, garbage collection is used more heavily before the JIT phase. This is most likely due to escape analysis in the JIT which removes many object allocations. Figure 2 shows the breakdown of time spent in each phase by benchmark. For some benchmarks, the *JIT* and *JIT call* phases dominate the execution (e.g., `ai`, `json_bench`), while others spend most of their time in the interpreter (e.g., `sympy_str`). With the exception of *blackhole*, every different phase dominates the execution time of at least one benchmark. This shows that none of these phases can be ignored from an optimization perspective, and aggressive improvements in one of these phases unfortunately translates to modest improvements of execution times on average. Figure 4 compares the phase breakdown for the meta-tracing JITs on the CLBG benchmarks. For the most part, different interpreters show similar trends when running the same program: large usage of GC in `binarytrees`, large usage of the JIT in `fasta` and `spectralnorm`, heavy use of JIT calls in `pidigits`, and large warmup overheads in `meteor`. The main exception to this is `revcomp` where PyPy spends most of its time in the interpreter while Pycket is able to compile and use JIT traces quickly.

C. Framework-Level Characterization: JIT Calls

Figures 2 and 4 show that for many benchmarks, the framework is mostly in the *JIT call phase*. These calls to AOT-compiled functions typically happen at a very fine granularity, unlike our cross-layer methodology, many other measurement methodologies typically lump this phase into the JIT phase. These calls arise from functions in the interpreter or the meta-tracing framework that cannot be inlined into the trace (typically because they contain unbounded loops).

To determine which AOT-compiled functions are being called from JIT-compiled meta-traces, we tracked the target addresses when our PinTool observed a call from JIT-compiled code. Table III shows the functions that constitute at least 10% of the *overall* execution time. Note that if these functions call other functions, the time spent in the called functions is also counted as part of these entry points. Some of the functions are part of RPython-level type intrinsics which implement the operations over standard Python types (e.g., lists, strings, dictionaries) used in the interpreter and framework. Another source is the RPython standard library, which provides a subset of Python's standard library for use by the interpreter and framework. There are also external C functions usually part of the

TABLE III. SIGNIFICANT AOT-COMPILED FUNCTIONS FROM META-TRACES

Benchmark	%	Src	Function
ai	19.4	I	setobject.get_storage_from_list
bm_chameleon	17.9	R	rordereddict.ll_call_lookup_function
bm_mako	26.1	L	runicode.unicode_encode_ucs1_helper
bm_mako	12.8	R	rordereddict.ll_call_lookup_function
bm_mdp	16.8	R	rordereddict.ll_call_lookup_function
django	16.6	L	rstring.replace
django	14.8	R	rordereddict.ll_call_lookup_function
eparse	12.3	R	rstr.ll_join
fannkuch	20.0	I	IntegerListStrategy._setslice
fannkuch	15.9	I	IntegerListStrategy._fill_in_with_sliced...
genshi_xml	12.4	R	rordereddict.ll_call_lookup_function
hexiom2	10.8	I	IntegerListStrategy._safe_find
html5lib	13.1	I	W_UnicodeObject._descr_translate
json_bench	18.5	M	_pypyjson.raw_encode_basestring_ascii
json_bench	10.6	R	rbuilder.ll_append
meteor-contest	35.4	I	BytesSetStrategy.difference_unwrapped
meteor-contest	22.2	I	BytesSetStrategy.issubset_unwrapped
nbody_modified	44.6	C	pow
pidigits	36.1	L	rbigint.divmod
pidigits	33.2	L	rbigint.int_mul
pidigits	13.0	L	rbigint.lshift
pidigits	12.6	L	rbigint.add
pyflate-fast	16.1	R	rstr.ll_find_char
pyflate-fast	11.7	I	BytesListStrategy.setslice
spitfire	22.1	R	rstr.ll_join
spitfire	14.4	R	rstr._ll_strhash
spitfire_cstringio	14.6	R	rbuilder.ll_append
spitfire_cstringio	14.1	R	ll_str.ll_int2dec
telco	13.4	L	rarithmetic.string_to_int
twisted_tcp	16.6	C	memcpy

Significant (>10% of overall execution) functions. The percentages are the time spent in AOT-compiled functions in overall execution. Src is where the functions are defined: R = RPython type system intinsics; L = RPython's std lib; C = external stdlib call; I = interpeter; M = PyPy module.

C standard library. In addition, some of these functions are defined by the interpreter or Python modules. In particular we can see that many benchmarks spend a significant amount of time in `rordereddict.ll_call_lookup_function`, which is the hashmap lookup function of RPython's dictionary data structure.

D. Interpreter-Level Characterization: JIT Warmup

Warmup can have important performance implications: compiling traces too eagerly results in wasted work, and compiling traces too lazily results in wasted opportunity. Traditional characterization methods can struggle to capture detailed warmup behavior. This is because the overhead associated with the process of measuring might alter the measured performance. We insert cross-layer annotations at the interpreter level at the beginning of each iteration of the dispatch loop. This enables accurately measuring the bytecode execution rate using a PinTool, and enables a precise definition of completed work per unit time. This data can enable finding the break-even points where JIT-compiling can compile efficient code that amortizes the overhead of tracing and compiling.

Figure 5 shows the warmup curves of the benchmarks, normalized to CPython. These plots show the number of bytecodes executed per unit time compared to CPython over the first 10 billion assembly instructions executed. It also shows the PyPy warmup break-even points for the point in time where the number of bytecodes executed thus far on PyPy matches that on CPython (dashed vertical lines) and PyPy without JIT (dotted vertical lines). In PyPy, it is surprising that the meta-tracing JIT compilation incurs negligible slowdowns compared to running the code on a PyPy interpreter without a meta-tracing JIT. The break-even point for PyPy compared to PyPy without JIT is usually reached very early on in the programs. There is more variability for the break-even points of reaching CPython performance. The programs where PyPy's performance advantage is smaller tends to have break-even points that are later. These benchmarks tend to be more complicated and have many different traces, so the warmup tends to take longer.

E. JIT IR-Level Characterization: Compilation Burden and Usefulness of Meta-Traces

One potential drawback of tracing-based JITs as opposed to method-based JITs is long and redundant traces. Whereas the unit of compilation in a method-based JIT is typically an application-level method, for tracing-based JITs, it is a particular trace taken as an application-level loop is executed. Different functions called as the loop executes are inlined into the trace, and different control paths taken result in different traces. Tracing-based JITs therefore tend to perform poorly when the application-level code has many alternative control flow paths that are taken in similar probabilities. Such code can result in the compilation of many traces, most of them infrequently used. JIT compilation of unused traces can be a compilation burden and hurt the performance, especially for long traces. Loops with long bodies (or loops that call many functions) result in long traces. The main drawback of long and infrequently used traces is the time it takes to compile them may not be amortized especially since some compiler passes have superlinear complexities with respect to the size of the code. The secondary drawback is increased memory usage to store the generated code.

Figure 6(a) shows the number of IR nodes that are JIT-compiled, which shows a large variability, ranging from less than 1000 (`float`, `nbody_modified`, `slowspitfire`, and `pidigits`) to 370,000 (`sympy_integrate`). Compiling a small number of IR nodes can indicate either the program does not have many branches due to its arithmetically heavy nature, or in the case of `pidigits`, spends most of its time in calls to AOT-compiled functions for arithmetic operations. While there is a large variability across different benchmarks, the best performing benchmarks typically compile between 2000-20,000 IR nodes. Figure 6(b) shows the percentage of JIT IR nodes that are executed 95% of the time spent in the JIT. For some benchmarks (e.g., `spectral-norm`, `spitfire_cstringio`, `slowspitfire`, and `bm_mdp`), only 5% of the compiled IR nodes are executed 95% of the time, indicating these benchmarks have exceptionally "hot" regions within the JIT-compiled code. For these benchmarks, multi-tiered JIT compilation might be beneficial even though this is not currently supported in RPython. If a large number of JIT-compiled IR nodes are used in the 95th percentile, this shows that many traces are used equally (e.g., in `sympy_integrate` and `sympy_str`) indicating a very branchy application and large compiler burden (as Figure 2 also shows large tracing and blackhole overheads). Figure 6(c) shows the number of executed IR nodes per one million dynamic assembly instructions. This data mostly matches the

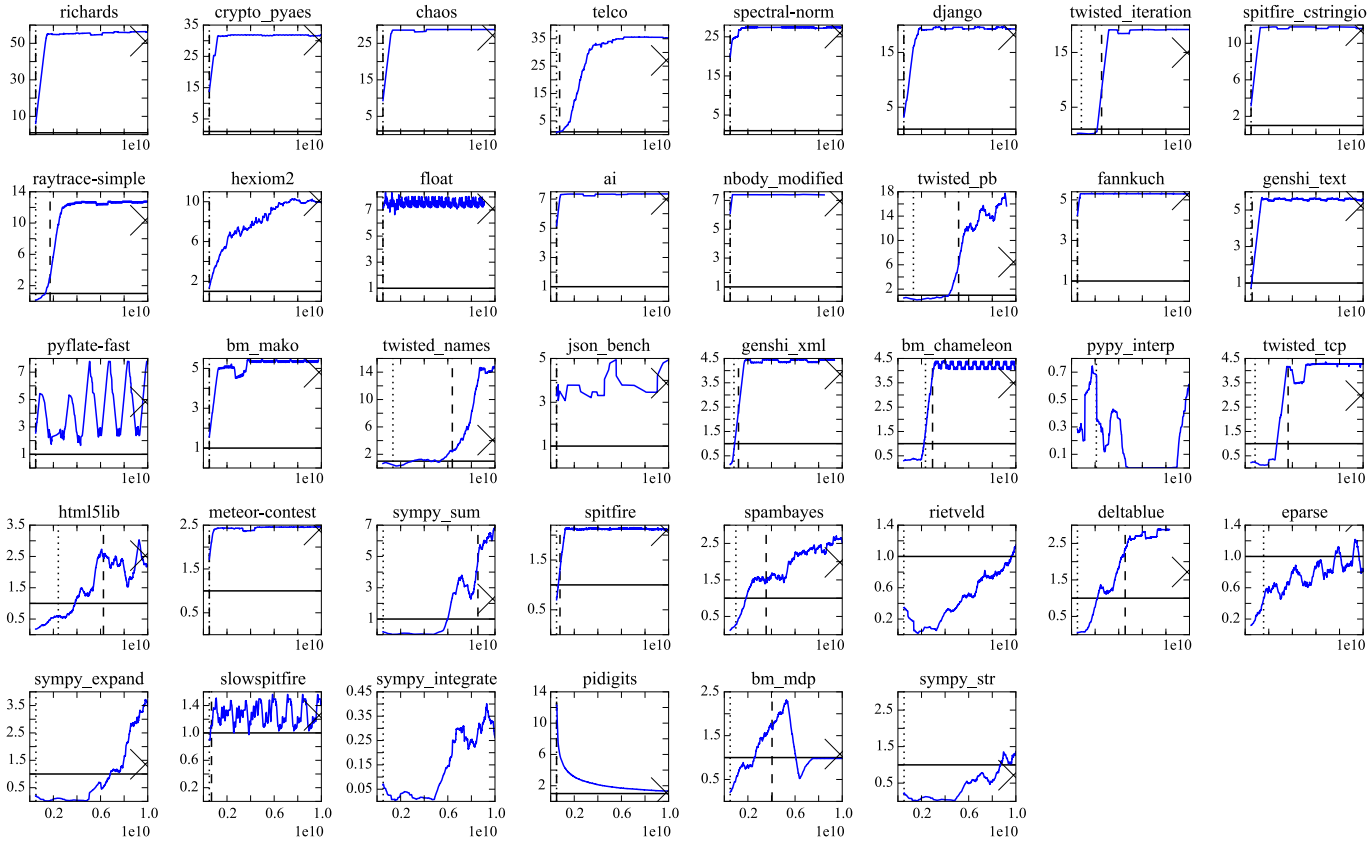

Figure 5. PyPy Warmup – PyPy bytecode execution rate normalized to CPython for the first 10 billion instructions. The dashed vertical lines indicate the break-even point with respect to CPython: in this point in time, both PyPy and CPython have executed the same number of bytecodes. The dotted vertical lines indicate the break-even point with respect to PyPy without JIT. The cross indicates the eventual (at the end of the execution) speedup of PyPy compared to CPython. The benchmarks are sorted in the order of speedup over CPython.

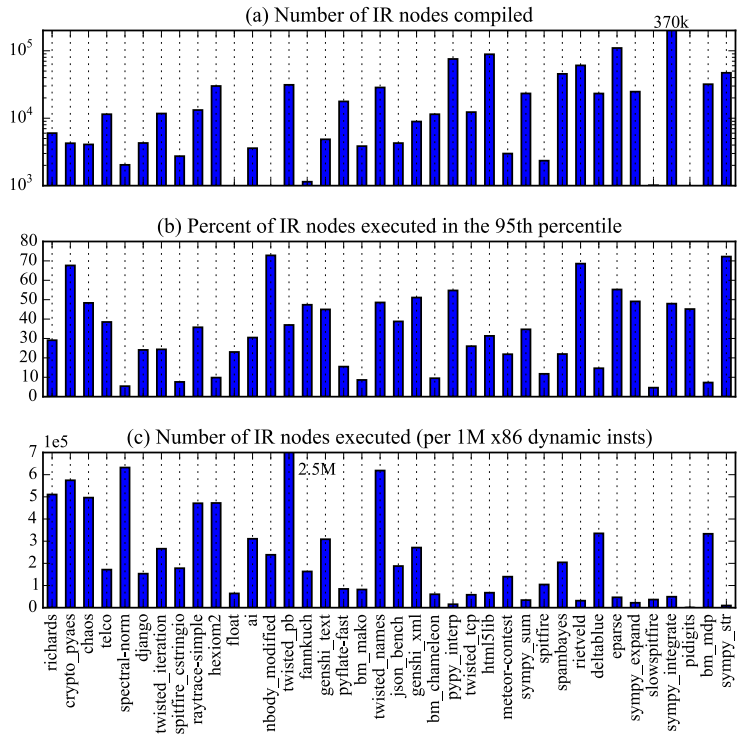

Figure 6. JIT IR Node Compilation and Execution Statistics – (a) Total number of JIT IR Nodes compiled throughout the benchmarks (every benchmark executed for 10B instructions), shown in log scale. (b) Most commonly executed JIT IR nodes (95% of the time spent in JIT-compiled code) shown as percentage of all IR nodes compiled. (c) Total (dynamic) number of IR nodes executed for every one million of assembly instructions executed. The benchmarks are sorted in the order of speedups over CPython.

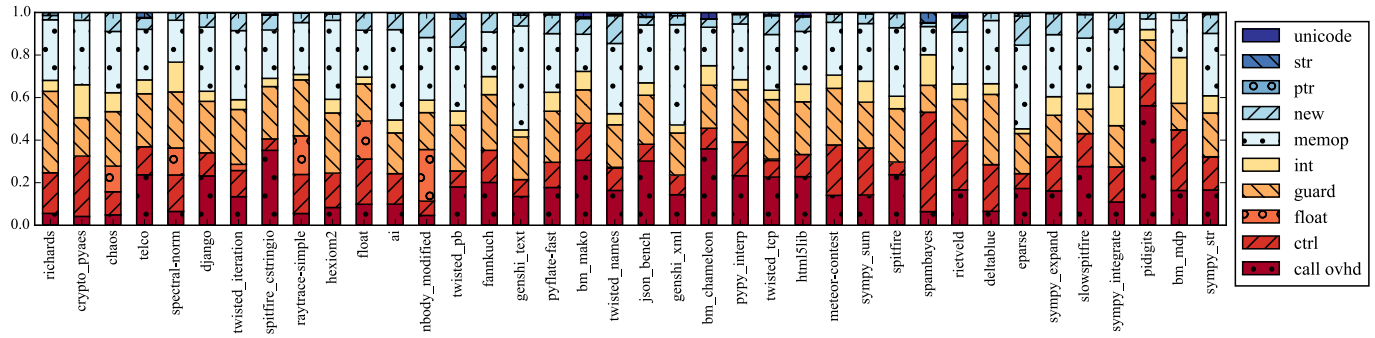

Figure 7. Categorized weighted IR node frequency by benchmark – Shows categorized breakdown of roughly the ratio of time spend in each class of IR nodes.

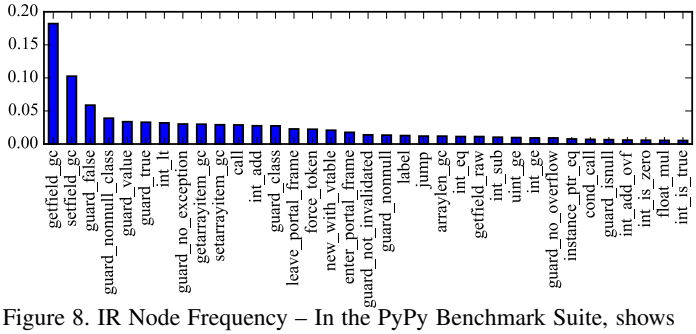

Figure 8. IR Node Frequency – In the PyPy Benchmark Suite, shows the frequency of the most common 35 IR node types.

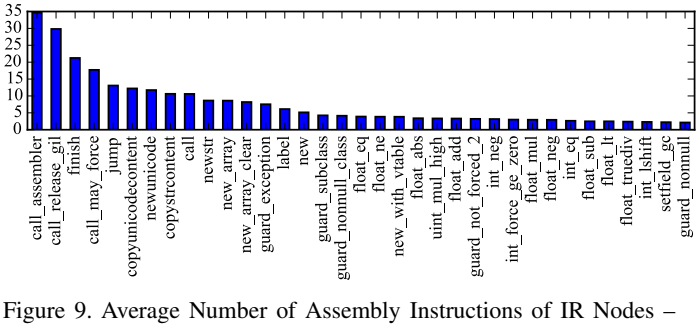

Figure 9. Average Number of Assembly Instructions of IR Nodes – In the PyPy Benchmark Suite, shows the average number of assembly instructions to implement the top 35 most expensive IR node types.

time spent in JIT-compiled code in Figure 2. The variations are mostly due to some IR nodes mapping to different number of x86 assembly instructions. It can also be seen that the best performing benchmarks on PyPy tend to have higher than average dynamic number of executed IR nodes.

F. JIT-IR-Level Characterization: Composition of Meta-Traces

Figure 8 shows the *dynamic* frequency histogram of different IR node types encountered in the PyPy Benchmark Suite, based on how many times these nodes are executed. Interestingly, 80% of IR node types constitute less than 1% of the overall execution in the JIT-compiled traces (mostly for uncommon use cases). Two IR nodes, `getfield_gc` and `setfield_gc`, constitute more than 18% and 10% respectively of all JIT traces. Implied by their name, these two operations get or set a field from a pointer, resulting in a memory load or a store after pointer arithmetic.

However, the frequency alone of these IR node types does not indicate how expensive they are. For this, Figure 9 shows on average, how many x86 assembly instructions are required to implement each IR node type. We can see that the top IR node type is `call_assembler` which maps to more than 30 assembly instructions. Other types of calls also take up more than 15 assembly instructions. The `call_assembler` calls another JIT trace from this trace, while the other `call_` nodes call AOT-compiled functions. These values are the call *overheads*, not the time spent in the called functions. However, most IR nodes, including the common `getfield_gc` and `setfield_gc`, require only one or two assembly instructions.

Figure 7 shows the breakdowns with IR nodes categorized as: memory operations (memop), guards, call overheads, control flow (ctrl), integer operations (int), memory allocation (new), floating-point operations (float), string operations (str), pointer manipulations (ptr), and unicode operations. Across all benchmarks, memory operation IR nodes are the most common, around 26%, followed by guards at 22%, call overheads at 18%, and control flow at 16%. Memory operations are the biggest part of meta-traces, followed by guards, which are unique to JIT-compiled code. Call overheads are also a major part of meta-traces.

Looking at how these categories break down per benchmark, we see that memory operations are the most significant part of meta-traces for most benchmarks, which likely is due to Python code that makes it easy to work with complex data structures. Exceptions to this are `bm_chameleon`, `bm_mako`, `bm_mdp`, `fannkuch`, `pidigits`, `spambayes`, `spectralnorm`. We see that `bm_chameleon`, `pidigits`, and `spitfire_cstringio` have large call overheads, as these benchmarks cause many calls to AOT-compiled functions. As an example, `pidigits` is making very heavy usage of `bignum` arithmetic, which is all implemented in AOT-compiled code that the meta-traces call into. We see that guard percentages stay similar across different benchmarks except for `richards` where it constitutes most of the execution. We see that even in the arithmetic-intensive benchmarks (e.g., `float`, `nbody_modified`, `chaos`), integer and floating point operations do not constitute a significant portion of meta-traces.

G. Microarchitecture-Level Characterization

Table I shows the instructions per cycle (IPC) and branch misses per 1000 instructions (MPKI) for the benchmarks. There

is a high variance (standard deviation is 0.37, 0.30, and 0.41 respectively for CPython, PyPy without JIT, and PyPy with JIT) in IPC across all benchmarks, indicating the application has a major impact on the IPC. However, CPython has a better IPC than PyPy with JIT by 15%, and PyPy without JIT by 32%. The IPC difference between the two interpreters without the JIT is surprising and partially accounts for the 2× performance gap between the two. This is again likely because RPython is not optimized for use as an interpreter without a meta-tracing JIT.

The JIT-compiled code includes numerous guards to ensure the observed control paths still hold true. This might lead to an increase in the number of branches. However, the results show that the branch rate across all interpreters is almost identical and different benchmarks do not show much variation. The MPKI of CPython and PyPy without JIT is very similar, however when the JIT is enabled, the MPKI drops by 35%. This is likely due to the more specialized and denser code produced by the JIT helping the processor to better predict the control flow [34].

The cross-layer annotations also allow us to study microarchitectural characteristics by phases. Table IV shows the mean and standard deviation of IPC, branches per instruction, and branch miss rate for each phase. As the overall results suggested, the interpreter IPC tends to be low with a relatively large misprediction rate. This is partially due to the interpreter being used only at the beginning for a short amount of time. The JIT phase (in this table, this also includes calls to AOT-compiled functions from the JIT-compiled code) has the largest IPC mean and the largest variation. It also has the lowest miss rate. The higher variation in the microarchitectural values is due to the application-specific nature of JIT-compiled code. The blackhole interpreter has the worst IPC among the phases, making the observations in Section V-B regarding the expense of this phase even more significant. Finally, the GC phase has a relatively high IPC compared to the other phases, perhaps because the same collection code is executed over and over, allowing the predictors to warm up sufficiently.

TABLE IV. MICROARCHITECTURAL CHARACTERIZATION OF PHASES

Phase	IPC	Branch / inst	Branch miss rate
interpreter	0.76 (0.26)	0.15 (0.019)	0.06 (0.021)
tracing	1.05 (0.07)	0.15 (0.003)	0.05 (0.005)
JIT	1.24 (0.53)	0.16 (0.038)	0.02 (0.026)
blackhole	0.48 (0.10)	0.13 (0.009)	0.09 (0.019)
GC	1.18 (0.30)	0.20 (0.024)	0.04 (0.016)

Microarchitectural means (and standard deviation in parentheses) by RPython phase in the PyPy Benchmark Suite.

VI. DISCUSSION

In this section, we use the results from Section V to derive initial answers to the nine key questions listed in Section I.

1. Can meta-tracing JITs significantly improve the performance of multiple dynamic languages? While it is well known that JIT compilation can significantly improve the performance of dynamic languages, the performance of meta-tracing JITs across different languages is still an active research area. As Table I showed, the PyPy meta-tracing JIT was indeed able to outperform the CPython interpreter on almost all benchmarks (up to 51× on `richards` and 30× on `crpyto_pyaes`). Looking at the CLBG results in Table II, the Pycket meta-tracing JIT had comparable performance to Racket, a custom JIT-optimizing VM. Even though there are some exceptions, and certainly room for improvement, our results seem to confirm the promise of meta-tracing JITs.

2. Does generating and optimizing traces in a meta-tracing JIT add significant overhead? The conventional wisdom is that JIT warmup can add a significant overhead, which might be problematic especially when interactivity is important. We see in Figure 2 that tracing can consume a large percentage of the total execution time for certain benchmarks. However, Figure 5 shows that the break-even point where meta-tracing JIT performance exceeds PyPy without the meta-tracing JIT occurs early in the benchmark execution. This suggests that even for short running applications, enabling the meta-tracing JIT does not significantly reduce performance. So compared to the basic PyPy interpreter, the overhead is not as significant as the conventional wisdom might suggest. CPython's interpreter is faster than PyPy without the meta-tracing JIT, so the corresponding break-even point is somewhat later in the execution.

3. Does deoptimization in a meta-tracing JIT consume a significant fraction of the total execution time? The conventional wisdom is that deoptimization should be relatively inexpensive [23, 27, 36]. Figure 2 shows that deoptimization (implemented using the "blackhole interpreter" in RPython) can consume more than 10% of the total execution time for some benchmarks. The phase diagrams of both fast-warming and slow-warming benchmarks in Figure 3 illustrate that the blackhole phase is an essential part of the warmup process. Meta-traces compiled when the control-flow coverage is insufficient require many deoptimizations to fall back to the interpreter and compile additional meta-traces. Furthermore, Table IV suggests that the blackhole phase performs poorly on modern hardware. Our results suggest that deoptimization presents a more significant overhead than perhaps the conventional wisdom might suggest.

4. Does garbage collection in a meta-tracing JIT consume a significant fraction of the total execution time? While garbage collection (GC) used to be a significant worry, modern JIT-optimizing VMs are carefully constructed to ensure that GC only consumes a small fraction of the total execution time (e.g., Wilson writes it "should cost roughly ten percent of running time" [44], which Jones and Lins call a "not unreasonable [figure] ... for a well-implemented system" [25, p. 13]). The breakdowns in Figure 2 confirm that RPython's GC does indeed consume a reasonable fraction of the total execution time with the exception of a few memory-intensive benchmarks.

5. Is all JIT-compiled code equally used in a meta-tracing JIT? The conventional wisdom states that there is different levels of "hotness" of frequently executed code leading to multi-tiered JITs, where the "hotter" a code region is, the more the compiler will try to optimize. However, this effect might be less pronounced in a tracing JIT due to each trace of execution getting compiled separately, so there might be many JIT-compiled traces which are executed roughly equally. Figure 6 shows that there is indeed a large "hotness" variability where in some benchmarks only 5% of compiled IR nodes are executed

95% of the time. For these cases, multi-tiered JIT compilation indeed should help.

6. Does a meta-tracing JIT need to spend most of its time in the JIT-compiled code to achieve good performance? Because JIT-compiled code is optimized using run-time feedback, the conventional wisdom suggests that most of the time should be spent executing JIT-compiled instructions. However, Figure 2 showed that many benchmarks (including several high-performance benchmarks) spend more of their time in AOT-compiled functions than JIT-compiled code. Because irregular control flow can potentially reduce performance, there is a delicate balance between code which should be JIT-compiled (to benefit from run-time feedback), and code which should be AOT-compiled (to avoid generating many bridges).

7. What fraction of the time in the JIT-compiled code is overhead due to the meta-tracing JIT? The conventional wisdom is that it is difficult to optimize dynamic languages, so overheads such as pointer chasing to access high-level data structures cannot be easily optimized away [13]. If we look at the breakdown in Figure 7, we see that memory operations indeed constitute the most significant time of JIT-compiled code. In addition, most guards also represent a JIT-specific overhead. Finally, calling AOT-compiled functions brings in a substantial overhead. While it is hard to name an exact percentage, it is likely that more than half of the JIT-compiled code is overhead.

8. What is the microarchitectural behavior of JIT-compiled code? The conventional wisdom states that JIT-compiled code might be inefficient compared to AOT-compiled code or a pure interpreter due to additional overheads such as guards. Comparing the performance to statically compiled languages and CPython in Table II, we see that meta-tracing JITs indeed have lower IPC. However, the IPC is usually within 0.5 of C code, meaning the gap is not as significant as one might assume. Furthermore, comparing the microarchitectural behavior of the RPython interpreter with and without the meta-tracing JIT in Table II and the phase microarchitectural breakdown in Table IV, we can see that the JIT-compiled code has better microarchitectural behavior than other phases of the RPython framework.

9. Why are meta-tracing JITs for dynamic programming languages still slower than statically compiled languages? Holistically considering our workload characterization, it is clear that there is no single reason. The meta-tracing framework has many components that different benchmarks and languages stress in different ways. The microarchitecture-level results do suggest that the primary problem is less that the JIT-compiled code is difficult to execute efficiently on modern architectures, and more that meta-tracing JITs still do significantly more work than statically compiled languages. Overall, there is no single "silver bullet" to improve meta-tracing JIT performance, meaning there is a wide array of opportunities for future researchers from both the VM and architecture communities.

VII. Related Work

To our knowledge, this is the first multi-language, cross-layer workload characterization that focuses on meta-tracing JITs.

Sarimbekov et al. study the workload characteristics of various dynamic languages on the JVM using the CLBG benchmarks [35]. They find that even though the implementations use many polymorphic callsites, most runtime method invocations are actually monomorphic. They also find that dynamic languages allocate significantly more objects than Java, most of which are short-lived, due to the additional boxing/unboxing that is common in dynamic languages.

Rohou et al. study branch misprediction in switch-based interpreters on Haswell-generation Intel hardware [34]. They find that counter to folklore [16], the cost of branch misprediction is very low on modern hardware, so techniques like jump threading are no longer necessary. However the statement of Rohou et al. that the "principal overhead of interpreters comes from the execution of the dispatch loop" is itself called into question by the work of Brunthaler [11]. He argues that this "is specifically not true for the interpreter of the Python programming language." The reason he gives is that Python bytecodes do a lot of work, so the overhead of bytecode dispatch is relatively lower. Castanos et al. observe this as well [13]. They write: "[in Jython] a single Python bytecode involves between 160 and 300 Java bytecodes, spans more than 20 method invocations, and performs many heap-related operations. Similar path lengths are also observed in the CPython implementation."

Anderson et al. introduce the "Checked Load" ISA extensions to offload type checking overhead to hardware in dynamic language interpreters [2]. Choi et al. propose a similar hardware mechanism for object access [14]. Gope et al. identify calls to AOT-compiled functions from the PHP JIT as significant overhead and propose hardware accelerators for common framework-level functions [18]. Dot et al. present a steady-state performance analysis of the V8 JavaScript engine using the builtin sampling profiler and present hardware mechanisms to provide a 6% performance speedup [15]. Southern and Renau use real systems to characterize the overhead of deoptimization checks (guards) in V8 [38]. Like our findings, they also find that the cost of such checks is lower than the conventional wisdom might suggest in the context of V8.

Würthinger et al. provide an overview of the Truffle system and the concept of runtime calls (AOT-compiled calls) from the JIT-compiled code, but do not quantify the overheads of such calls [45].

Holkner and Harland evaluate the dynamic behaviour of Python applications [21]. They find that all the tested programs use some dynamic reflective features that are hard to compile statically, and a large fraction of the tested programs execute code that is dynamically generated at runtime (20%).

There have been some studies to characterize the performance of JavaScript, e.g., [33] study the dynamic features typically used in JavaScript, and [32] study the correlation between benchmarks and their real-world counterparts.

VIII. Conclusion

We have presented a cross-layer workload characterization of meta-tracing JIT VMs. To make this study possible, we first introduced a new cross-layer annotation methodology, which allows inserting annotations at different abstraction levels of a multi-level VM (e.g., the RPython framework), and observing these at different levels of execution (e.g., the binary level to get microarchitectural statistics, or the assembly layer to

intercept using a dynamic binary instrumentation tool). We then used two RPython interpreters, PyPy and Pycket, for application-level, framework-level, interpreter-level, JIT IR-level, and microarchitecture-level characterization. We finally provided initial answers to nine key questions regarding meta-tracing JIT VM performance. One main takeaway is the performance characteristics are highly varied across different applications, bottlenecked by different parts of the meta-tracing JIT VM, and there is likely no single "silver bullet" that could give significant speedups by a small change in software or hardware.

ACKNOWLEDGMENTS

This work was supported in part by NSF CRI Award #1512937, NSF SHF Award #1527065, and donations from Intel. The authors acknowledge and thank Alex Katz for his early work on automating data collection for the workload characterization.

REFERENCES

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

[2] O. Anderson et al. Checked Load: Architectural Support for JavaScript Type-Checking on Mobile Processors. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb 2011.

[3] D. Beazley. Understanding the Python GIL. *PyCon*, Feb 2010.

[4] C. F. Bolz. *Meta-Tracing Just-In-Time Compilation for RPython*. Ph.D. Thesis, Mathematisch-Naturwissenschaftliche Fakultät, Heinrich-Heine-Universität Düsseldorf, 2012.

[5] C. F. Bolz et al. Allocation Removal by Partial Evaluation in a Tracing JIT. *Workshop on Partial Evaluation and Prog. Manipulation*, Jan 2011.

[6] 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*, Jul 2009.

[7] C. F. Bolz et al. Back to the Future in One Week: Implementing a Smalltalk VM in PyPy. *Workshop on Self-Sustaining Sys.*, May 2008.

[8] 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*, Jul 2010.

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

[10] C. F. Bolz and L. Tratt. The Impact of Meta-Tracing on VM Design & Implementation. *Science of Computer Prog.*, 98:408–421, Aug 2015.

[11] S. Brunthaler. Virtual-Machine Abstraction & Optimization Techniques. *Electronic Notes in Theoretical Computer Science*, Dec 2009.

[12] S. Cass. 2016 Top Programming Languages. *IEEE Spectrum*, Jul 2016.

[13] J. Castanos et al. On the benefits and pitfalls of extending a statically typed language JIT compiler for dynamic scripting languages. *SIGPLAN Not.*, Oct 2012.

[14] J. Choi et al. ShortCut: Architectural Support for Fast Object Access in Scripting Languages. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2017.

[15] G. Dot, A. Martínez, and A. González. Analysis and Optimization of Engines for Dynamically Typed Languages. *Int'l Symp. on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct 2015.

[16] M. Ertl and D. Gregg. The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures. *Euro-Par 2001*, Aug 2001.

[17] A. Gal et al. Trace-based Just-in-Time Type Specialization for Dynamic Languages. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2009.

[18] D. Gope, D. J. Schalis, and M. H. Lipasti. Architectural Support for Server-Side PHP Processing. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2017.

[19] I. Gouy. The Computer Language Benchmarks Game. `http://benchmarksgame.alioth.debian.org/`.

[20] HippyVM PHP. `https://github.com/hippyvm/hippyvm`.

[21] A. Holkner and J. Harland. Evaluating the dynamic behaviour of Python applications. *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, 2009.

[22] M. Hölttä. Crankshafting from the Ground Up. *Google Technical Report*, 2013.

[23] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 1992.

[24] Mozilla IonMonkey. `https://wiki.mozilla.org/IonMonkey`.

[25] R. Jones and R. D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Sep 1996.

[26] Julia. `http://julialang.org`.

[27] T. Kotzmann and H. Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. *ACM/USENIX Int'l Conf. on Virtual Execution Environments (VEE)*, 2005.

[28] C.-K. Luk et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2005.

[29] S. Marr and S. Ducasse. Tracing vs. Partial Evaluation: Comparing Meta-Compilation Approachess for Self-Optimizing Interpreters. *Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, Dec 2015.

[30] P. J. Mucci et al. PAPI: A Portable Interface to Hardware Performance Counters. *DoD HPCMP Users Gp. Conf.*, 1999.

[31] PyPy Benchmark Suite. `https://bitbucket.org/pypy/benchmarks`.

[32] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. *WebApps 2010*, Jun 2010.

[33] G. Richards et al. An Analysis of the Dynamic Behavior of JavaScript Programs. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2010.

[34] E. Rohou, B. N. Swamy, and A. Seznec. Branch Prediction and the Performance of Interpreters: Don't Trust Folklore. *Int'l Symp. on Code Generation and Optimization*, Feb 2015.

[35] A. Sarimbekov et al. Characteristics of Dynamic JVM Languages. *Workshop on Virtual Machines and Intermediate Lang.*, Oct 2013.

[36] D. Schneider and C. F. Bolz. The Efficient Handling of Guards in the Design of RPython's Tracing JIT. *Workshop on Virtual Machines and Intermediate Lang.*, 2012.

[37] C. Seaton. *Specialising Dynamic Techniques for Implementing the Ruby Programming Language*. Ph.D. Thesis, School of Computer Science, University of Manchester, 2015.

[38] G. Southern and J. Renau. Overhead of Deoptimization Checks in the V8 JavaScript Engine. *Int'l Symp. on Workload Characterization (IISWC)*, Sep 2016.

[39] L. Stadler, T. Würthinger, and H. Mössenböck. Partial Escape Analysis and Scalar Replacement for Java. *Int'l Symp. on Code Generation and Optimization*, Feb 2014.

[40] Topaz Ruby. `http://github.com/topazproject/topaz`.

[41] FastR. `https://github.com/graalvm/truffleruby`.

[42] TruffleRuby. `https://github.com/graalvm/truffleruby`.

[43] V8 JavaScript Engine. `https://code.google.com/p/v8`.

[44] P. R. Wilson. Uniprocessor Garbage Collection Techniques. *Int'l Workshop on Memory Management*, 1992.

[45] T. Würthinger et al. Practical Partial Evaluation for High-Performance Dynamic Language Runtimes. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun 2017.

[46] T. Würthinger et al. One VM to Rule Them All. *Int'l Symp. on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, Oct 2013.

[47] T. Würthinger et al. Self-Optimizing AST Interpreters. *Symp. on Dynamic Languages*, Oct 2012.