An Efficient and Generic Event-based Profiler Framework for Dynamic Languages

Gülfem Savrun-Yeniçeri
University of California, Irvine
gsavruny@uci.edu

Michael L. Van de Vanter
Oracle Labs
michael.van.de.vanter@oracle.com

Per Larsen
University of California, Irvine
perl@uci.edu

Stefan Brunthaler
University of California, Irvine
s.brunthaler@uci.edu

Michael Franz
University of California, Irvine
franz@uci.edu

ABSTRACT

Profilers help programmers analyze their programs and identify performance bottlenecks. We implement a profiler framework that helps to compare and analyze programs implementing the same algorithms written in different languages. Profiler implementers replicate common functionalities in their language profilers. We focus on building a generic profiler framework for dynamic languages to minimize the recurring implementation effort. We implement our profiler in a framework that optimizes abstract syntax tree (AST) interpreters using a just-in-time (JIT) compiler. We evaluate it on ZipPy and JRuby+Truffle, Python and Ruby implementations in this framework, respectively. We show that our profiler runs faster than the existing profilers in these languages and requires modest implementation effort. Our profiler serves three purposes: 1) helps users to find the bottlenecks in their programs, 2) helps language implementers to improve the performance of their language implementation, 3) helps to compare and evaluate different languages on cross-language benchmarks.

CCS Concepts

•Software and its engineering → Interpreters; Runtime environments; Just-in-time compilers;

Keywords

Profiling, dynamic languages, abstract syntax tree interpreters, Python, Ruby, PyPy, JRuby, Java virtual machine

1. MOTIVATION

A profiler is a program analysis tool that helps programmers identify frequently executed parts of the program and detects performance bottlenecks. Typically, there are two profiling modes: event-based profiling and sampling-based profiling. Event-based profilers track every occurrence of certain events, such as method calls and returns, whereas sampling-based profilers sample the current program state at regular intervals. Event-based profilers are typically more accurate, but add more overhead than sampling-based profilers.

Many popular dynamic languages such as Python, Ruby, and Perl are typically interpreted. Since there is already an active interpreter during execution, it is straightforward to implement an event-based profiler for these languages. Therefore, the reference implementations typically provide an event-based profiler. However, these languages and their profilers are slow due to interpretation.

As the number and popularity of dynamic language applications grow, optimizing their performance and building efficient analysis tools for them become increasingly important. Programmers now use dynamic languages to build large scale applications. For example, YouTube, Dropbox, Yelp, and Quora use Python, and Twitter and GitHub use Ruby in their implementation. Similarly, Django and Ruby on Rails are popular high-level web application frameworks in Python and Ruby, respectively.

Truffle [28, 11] is a new framework for creating high-performance implementations of dynamic languages. Language implementers write an AST interpreter, then the framework applies dynamic optimizations such as type specialization, and just-in-time compiles the AST to machine code. There are Truffle implementations for JavaScript, Python, Ruby, Smalltalk, and R, which show superior performance compared to existing implementations [10, 30, 15].

We develop a profiler framework for Truffle language implementations and evaluate it on two Truffle language implementations: ZipPy (Python implementation in Truffle) and JRuby+Truffle (Ruby implementation in Truffle).

Our contributions are:

- We provide a comprehensive event-based profiler framework that profiles various events to analyze dynamic language programs in more detail. Our profiler framework helps to compare and evaluate the programs implementing the same algorithms across different languages. We compare and evaluate two different languages, Python and Ruby, on cross-language benchmarks implementing the same algorithms with our profiler framework, and report our findings.
- We show how to implement a generic profiler framework for languages running on the Truffle framework that minimizes profiler implementation effort.

PPPJ '15, September 08-11, 2015, Melbourne, FL, USA

DOI: http://dx.doi.org/10.1145/2807426.2807435

- We compare our profiler against existing profilers in Python and Ruby, and report an average speedup of 10× and 1.1× over CPython and PyPy, and an average speedup 208× and 12× over Matz's Ruby Interpreter (MRI), a.k.a. CRuby, and JRuby respectively. We show that with modest implementation effort, the Truffle framework lets us construct a high-performance profiler for languages running on top of it.

- Our profiler framework helps both the language implementer and user. Existing profilers in Python and Ruby target the programming language users, i.e. they help programmers to find the performance bottlenecks in their programs. However, we also target the language implementers in our profiler framework. We give feedback to the language implementers to improve the performance of their Truffle language implementation.

2. BACKGROUND

2.1 Truffle Framework

Dynamic languages provide flexibility such as dynamic typing and heterogeneously typed high-level data structures to programmers. With dynamic typing, variables get the type of the object assigned to them at run time. For example, "a + b" can perform integer addition, string concatenation, or a user-defined operation based on its input operands at runtime. The Truffle framework optimizes dynamic languages by collecting type information, and speculatively replacing generic AST nodes with type-specialized nodes [29]. This node-rewriting strategy relies on an important observation called type stability [6] that optimizes dynamic typing. The observation is that the operand types of an operation are unlikely to change during program execution. Therefore, Truffle speculatively rewrites generic nodes to type-specialized versions, and it replaces nodes back to generic nodes when speculation fails.

ZipPy [30] is the Python implementation developed with the Truffle framework. Similarly, JRuby+Truffle [10] is the Ruby implementation implemented with the Truffle framework.

2.2 Existing Dynamic Language Profilers

Python [23] has a profiler called *cProfile* [24] implemented as a C extension module. It is an event-based profiler which records all method calls and returns, and measures time intervals between these events. Another Python profiler is *profile*. It is a pure Python module that has the same interface as cProfile. The disadvantage of the *profile* module is that it adds significant overhead to profiled programs.

Ruby [16] has a *profile* library that is part of the standard library. Since it is a pure Ruby library, profiled programs run slow. An alternative profiler is *ruby-prof* [25] which is a C extension and therefore is many times faster than the standard Ruby profiler. JRuby [12] is the implementation of Ruby on top of the Java virtual machine (JVM). It uses the new `invokedynamic` bytecode [27], which improves the costly invocation semantics of dynamic programming languages targeting the JVM. JRuby has a built-in profiler which is a clone of ruby-prof built into JRuby [13]. It produces similar output to ruby-prof, and users need to enable the *–profile* flag to use it.

3. IMPLEMENTATION

An AST interpreter is a natural way to implement a language. A parser typically builds an AST. Then, the language implementer adds `execute` methods to the AST nodes that implement an AST interpreter. Every node in the AST has a list of child nodes, which it executes before returning a result to its parent. Truffle language implementers write an AST interpreter. Each language function has a root node, and the execution starts by executing the root node. Listing 1 shows a Python function that adds two variables, and Figure 1 displays the generated AST for this function on the left.

Listing 1: Example add function.

```
1 def add(a, b):
2     return a + b
3
4 add(10, 20)
5 add("hello", "world")
```

3.1 Instrumentation Framework

The Truffle platform includes multi-purpose instrumentation support for building program analyzers and other tools. An experimental built-in debugger in JRuby+Truffle demonstrated that extremely low runtime overheads could be achieved through tight integration with the underlying Truffle platform [26]. A generalized instrumentation framework is now in place, supporting development of a fully functional multi-language debugging service along with other tools such as code coverage. Here we focus on building an efficient profiler using Truffle instrumentation.

The instrumentation framework works by inserting additional AST nodes, essentially rewriting a program to include instrumentation functionality. Instrumentation nodes are no different from other Truffle AST nodes with respect to optimization, so inactive instrumentation nodes add zero overhead when running fully optimized. Instrumentation provided by client tools can be added and removed dynamically using core Truffle tree-rewriting mechanisms.

An AST node becomes *instrumentable* by insertion of a *wrapper node* between the node and its parent. A wrapper node is language-specific, so it must be implemented separately by every language implementer. Each wrapper has an attached language-independent *probe* node, which represents the association with the particular piece of source code represented by the instrumentable node. The probe node also manages the *attachment* and *detachment* of tool-provided *instrument nodes*. When the execution reaches a wrapper node, it notifies the probe node both before and after executing its child; the probe node passes each of these *execution events* to every attached instrument node.

Figure 1 shows the AST after inserting a wrapper node and attaching a profiler instrument node to the add node on the right. Listing 2 and Listing 3 show the implementation of a wrapper and a profiler instrument node.

3.2 Cross-language Comparison

Existing profilers detect performance bottlenecks in a program written in a specific language. For example, a Python profiler analyzes a Python program, and identifies performance bottlenecks in that program. Truffle focuses on implementing high-performance implementations of multiple

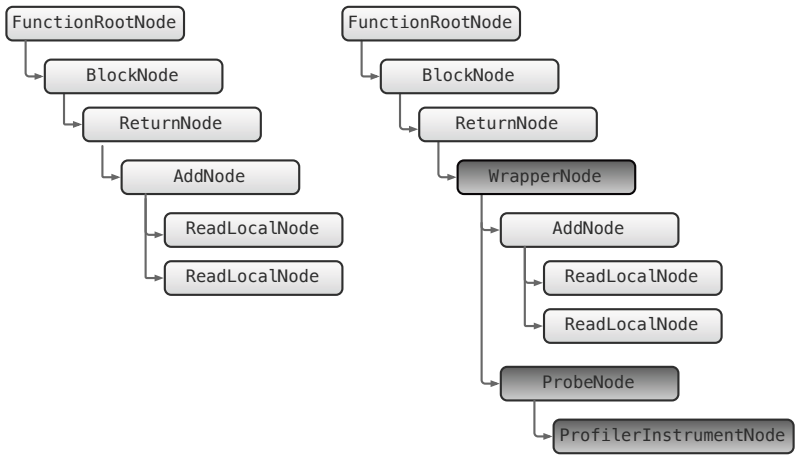

Figure 1: AST of an add function before and after inserting instrumentation nodes to the add operation event.

Listing 2: Wrapper Node.

```
class Wrapper extends PythonNode {
    @Child Node child;
    ProbeNode probe;

    @Override
    Object execute(VirtualFrame frame) {
        probe.enter(child, frame);
        Object result = child.execute(frame);
        probe.leave(child, frame);
        return result;
    }
}
```

Listing 3: Profiler Instrument Node.

```
class ProfilerInstrument extends Instrument {

    @Override
    void enter(Node astNode, VirtualFrame frame) {
        ...
    }

    @Override
    void leave(Node astNode, VirtualFrame frame) {
        ...
    }
}
```

dynamic languages. Implementing a profiler on this framework lets us compare programs across different languages. Languages have their unique features, and using these unique features results in executing different events. Profiling different languages on cross-language benchmarks that implement the same algorithms allows us to show common and different features in languages. To the best of our knowledge, our profiler is the first profiler that makes it possible to compare and analyze the dynamic language programs implementing the same algorithms across different languages.

3.3 Event-based Profiling

Event-based profilers track certain execution events. For instance, method profilers count the number of times each method executes at runtime. Counting AST nodes provides information about the execution of a program, so we count how many times each node executes in our profiler. We divide collected events into five categories: control-flow, operation, collection, variable access, and type distribution.

Control-flow: Profiles statement nodes that change the control-flow such as loops, iterations, `if`, `continue`, `next`, and `break` statements. For example, this category shows a user how many times each loop or `if` statement executes.

Operation: Profiles operation nodes such as arithmetic, comparison, and logical operations in the program.

Collection: Profiles collections such as lists and arrays in the given program. Profiled collection operations include reading an element from a collection, adding an element to a collection, deleting an element from a collection, and slicing a collection.

Variable access: Profiles nodes performing any kind of variable access, such as a local, global, and an instance variable access.

Type distribution: Profiles distribution of types of nodes in a program.

We insert a wrapper node to the node that we want to profile, and attach a profiler instrument node to the wrapper node. The profiler instrument node simply increments the counter whenever the instrumented node executes. We report how many times each node executes, and the source code location of the node in the program. We can simply extend our profiler to collect execution time of each event. Although we divide the collected events info five categories, the implementer can simply extend these categories based on her language.

Dynamic typing adds a significant performance overhead to language execution. Truffle uses type-specialized nodes to reduce this overhead. Demonstrating the distribution of types of nodes in a program is useful for two reasons. First, the language implementer can verify whether type-specializations are performed correctly in her implementation. Second, the language user could monitor the types in the program. In the example showed in Listing 1, the function executes two times: with integer arguments and string arguments, so the profiler reports two types for the add operation.

For type distribution profiling, we insert a wrapper node, and attach an instrument node that maps types to counters. Whenever a node executes, we check whether we have observed this type before. If this is a new type, we add it to the map. In ZipPy, type distribution profiling profiles the types for operations and variable accesses. It only profiles the types for variable accesses in JRuby+Truffle because Ruby represents operations as method calls.

3.4 Method Profiling

Both Python and Ruby implementations have method profilers that measure the number of method invocations and the time spent in each method. Since detecting frequently executed methods is a useful feature, we also add this functionality to our profiler. We insert a wrapper node to call nodes and attach an instrument node that collects invocation counter and time. We record the time before and after a call node executes and calculate the elapsed time. Listing 4 shows the implementation of our method profiler instrument.

In ZipPy, our method profiler output is similar to cProfile for Python. We show the total number of calls and execution time for each user-defined and built-in function in the running application. We show the total time spent in each function by excluding time made in calls to sub-functions, and cumulative time spent in this and all sub-functions. On the other hand, Ruby represents many of its internals as methods calls such as operators, and property accesses. For example, Ruby translates "a + b" statement into "a.+(b)", which is a call to a method named "+". We are interested in profiling user-defined methods in our method profiler. Our method profiler in JRuby+Truffle produces output similar to the built-in profiler in JRuby. We show the total time spent in each method, and break the total time into self and children time. Self time shows the time spent in the method itself, excluding calls to child methods. Children time shows the time spent in calls to the child methods.

4. EVALUATION

In this section, we present a detailed evaluation of our profiler in ZipPy and JRuby+Truffle.

4.1 Experimental Setup

We include all the benchmarks that are available in both ZipPy and JRuby+Truffle implementations. There are six benchmarks (binarytrees, fannkuchredux, mandelbrot, nbody, pidigits, spectralnorm) from the Computer Language Benchmarks Game [7], and one benchmark (richards) from the V8 benchmark suite [8], which was originally developed for BCPL.

- binarytrees: recursive calls to allocate and deallocate binary trees
- fannkuchredux: repeatedly access a tiny integer-sequence
- mandelbrot: generates a Mandelbrot set
- nbody: perform an N-body simulation of the Jovian planets
- pidigits: streams arbitrary-precision arithmetic
- richards: performs an OS kernel simulation
- spectralnorm: calculate an eigenvalue using the power method

Listing 4: Method Profiler Instrument Node.

```
class MethodProfilerInstrument extends Instrument {
    long counter;
    long startTime;
    long totalElapsedTime;

    @Override
    public void enter(Node astNode, VirtualFrame frame) {
        counter++;
        startTime = System.nanoTime();
    }

    @Override
    public void leave(Node astNode, VirtualFrame frame) {
        long endTime = System.nanoTime();
        long elapsedTime = endTime - startTime;
        totalElapsedTime = totalElapsedTime + elapsedTime;
    }
}
```

Our system configuration is as follows:

- CPython 3.3.2.
- PyPy 3.2.1.
- MRI (CRuby) 1.9.3.
- JRuby 1.7.13.
- Truffle 0.5
- Intel Xeon E5-2660 running at a frequency of 2.20 GHz, the Ubuntu Linux 3.2.0-64 kernel and `gcc` 4.6.3.

We execute each benchmark ten times and report individual and average execution times.

4.2 Profiler Implementation Effort Comparison

We implement and evaluate our profiler in ZipPy and JRuby+Truffle, but our profiler is reusable among other implementations of Truffle such as JavaScript, Smalltalk, and R. Table 1 lists the number of lines of generic profiler framework code shared by ZipPy and JRuby+Truffle, and language-dependent code implemented separately in ZipPy and JRuby+Truffle. Although language-dependent parts are similar in both ZipPy and JRuby+Truffle, the language implementers must implement them separately for their own language. It also shows the number of lines of code used to implement Python cProfile profiler and JRuby's built-in profiler. As a result, we implement a generic profiler framework with modest implementation effort.

Implementation	Number of Lines of Code
Generic	1073
ZipPy	508
JRuby+Truffle	516
Python cProfile profiler	1650
JRuby built-in profiler	2223

Table 1: Profiler implementation effort comparison details.

4.3 Cross-language Comparison

Our profiler helps to compare and evaluate the programs implementing the same algorithms written in different languages. As an experiment, we compare and evaluate two different languages, Python and Ruby, on cross-language benchmarks implementing the same algorithms with our profiler. Table 2 and Table 3 list the total number of executed nodes in ZipPy and JRuby+Truffle in various categories. They execute different numbers of nodes in each profiling category for the same benchmarks. We highlight the categories that show significantly different results in two languages on cross-language benchmarks.

Python and Ruby have their unique features, and using these unique features results in executing different numbers of nodes. For example, mandelbrot operates on complex numbers. Python has a built-in complex type, whereas Ruby does not provide a built-in complex type. Therefore, Ruby performs many more operations to do complex number arithmetic. Python has a builtin method called `abs` which operates on complex numbers, and mandelbrot intensively uses this function, so it performs more methods calls than Ruby. In nbody, Ruby defines a class and creates objects for every Jovian planet, whereas Python uses a collection to hold Jovian planets. Therefore, Ruby performs many instance method calls, and Python performs more collection operations in nbody. As a result, our profiler makes it possible to compare and analyze the programs implementing the same algorithms across different languages.

4.4 Event-based Profiling Performance

Figure 2 demonstrates the contribution of each event counting category to the total execution time in ZipPy respectively. Control-flow, operation, collection, variable access, and type distribution profiling add an average 4%, 7%, 3%, 25%, 34% overhead in ZipPy. Control-flow, operation, and collection profiling add a low overhead in ZipPy. However, variable access and type distribution profiling add a higher overhead. For example, fannkuchredux performs a signifi-

ZipPy	Call	Control-flow	Operation	Collection	VariableAccess
binarytrees	329×10^6	495×10^6	410×10^6	1	2×10^9
fannkuchredux	739	12×10^9	13×10^9	9×10^9	55×10^9
mandelbrot	$\mathbf{416 \times 10^6}$	905×10^6	1×10^9	1	2×10^9
nbody	**387**	116×10^6	1×10^9	$\mathbf{1 \times 10^9}$	3×10^9
pidigits	1×10^6	8×10^6	27×10^6	0	61×10^6
richards	1×10^9	3×10^9	2×10^9	165×10^6	11×10^9
spectralnorm	470×10^6	481×10^6	5×10^9	480×10^6	8×10^9

Table 2: Total number of event counters in ZipPy.

JRuby+Truffle	Call	Control-flow	Operation	Collection	Variable Access
binarytrees	658×10^6	495×10^6	575×10^6	0	2×10^9
fannkuchredux	216	5×10^9	10×10^9	10×10^9	53×10^9
mandelbrot	**34**	920×10^6	4×10^9	0	9×10^9
nbody	$\mathbf{573 \times 10^6}$	116×10^6	2×10^9	$\mathbf{109 \times 10^6}$	6×10^9
pidigits	4×10^3	8×10^6	27×10^6	0	68×10^6
richards	2×10^9	2×10^9	761×10^6	29×10^6	5×10^9
spectralnorm	932×10^6	481×10^6	4×10^9	462×10^6	7×10^9

Table 3: Total number of event counters in JRuby+Truffle.

cant number of variable accesses as shown in Table 2, so variable access profiling adds a significant overhead in that benchmark. Type distribution profiling collects types for operations and variable accesses, therefore, it also adds a high overhead in fannkuchredux benchmark.

Figure 3 demonstrates the contribution of each event counting category to the total execution time in JRuby+Truffle. Control-flow, operation, collection operation, variable access, and type distribution profiling add an average 4%, 13%, 4%, 41%, 45% overhead in JRuby+Truffle respectively. Similar to ZipPy, control-flow, operation and collection operation profiling only add a small overhead, but variable access and type distribution profiling add a higher overhead in JRuby+Truffle.

Control-flow profiling is especially useful for language users because they can observe which loops or statements are hot in their program. Statement-granularity profiling might be more insightful than method-granularity profiling to the programmers in some cases. Collection profiling is also useful for language users. Python and Ruby each provide a rich set of collections, and users heavily use them in their programs. To the best of our knowledge, there is no Python or Ruby profiler that provides information about collection usage. Therefore, our unique collection profiling gives hints to users about how to optimize their collection usage. For example, tuples are immutable collections and lists are mutable collections in Python. By looking at the collection profiling output, users might learn that they never modify the list, therefore, they can use a tuple instead. As future work, we plan to give feedback and suggestions about collections based on the collected profiling information in our profiler. Type distribution is an important category for both users and implementers. When the types are not stable in a given program, Truffle is not able to optimize that program well. Our type distribution profiling draws users attention to the nodes that frequently change their type during execution. Then, the user might modify their program for better performance.

Our event-based profiler is not limited to the event categories shown here. It is extensible, so the profiler implementer can add more events to capture her language's characteristics.

4.5 Method Profiling Performance

Figure 4 shows method profiling performance across different Python implementations. We compare our method profiler in ZipPy against CPython [23] and PyPy [22] using cProfile module to profile methods. CPython is the reference Python implementation that has an interpreter written in C. PyPy is the Python implementation written in a subset of Python, and uses a tracing JIT compiler to optimize frequently executed parts of the program [5]. The x axis labels benchmarks, and y axis displays the execution time in a logarithmic scale. On average, our profiler runs 10× faster than CPython and 1.1× faster than PyPy profiler in ZipPy. PyPy performs slightly better than ZipPy in nbody and pidigits benchmarks without profiling. When profiling is enabled, ZipPy performs better than PyPy in binarytrees, fannkuchredux, mandelbrot and richards, and PyPy performs better than ZipPy in spectralnorm. Among our benchmarks, binarytrees, mandelbrot, richards, and spectralnorm perform the highest number of calls as displayed in Table 2. As a result, profiling methods adds a higher overhead in these four benchmarks.

PyPy is a fast alternative implementation of Python that uses a tracing JIT compiler for producing optimized code, however, it is not able to perform well when profiling is enabled. The reason is that cProfile is a module written in C, and PyPy is not able to optimize C extension modules. The PyPy community states that they support C extension modules just to provide basic functionality. They advise users to use a native Python implementation instead of a C module for better performance. We also use profile module which is a pure Python module to profile our benchmarks

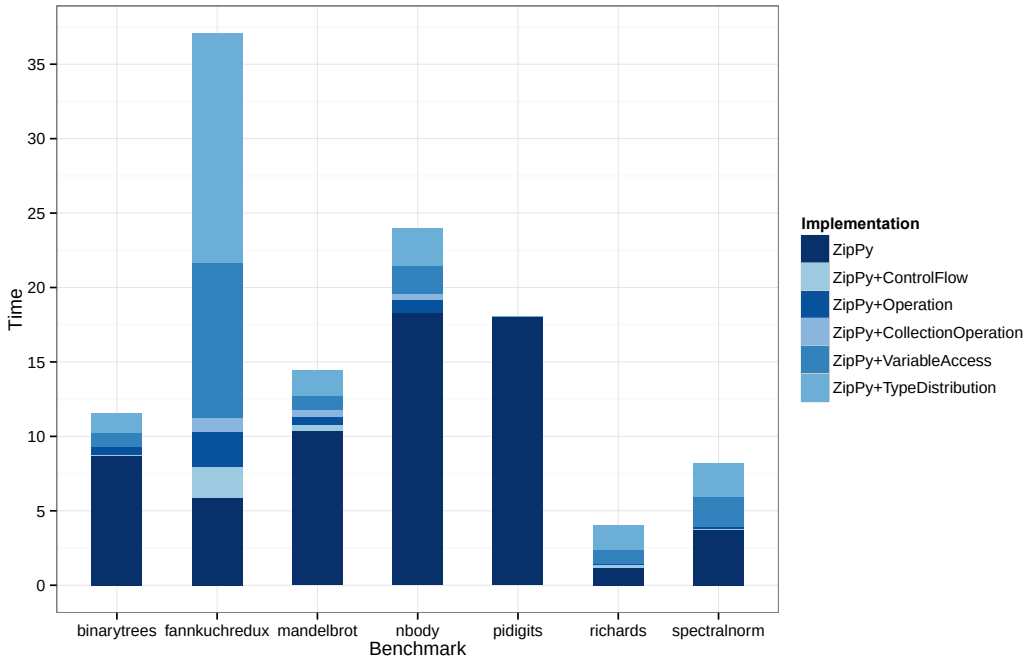

Figure 2: Event-based profiling overhead in ZipPy.

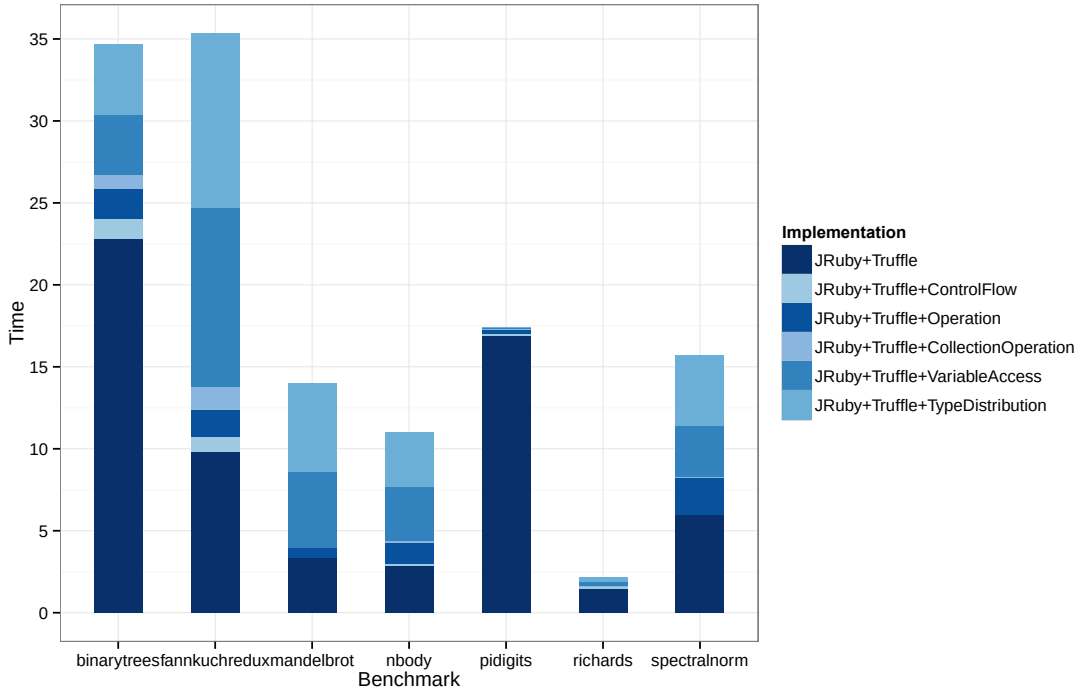

Figure 3: Event-based profiling overhead in JRuby+Truffle.

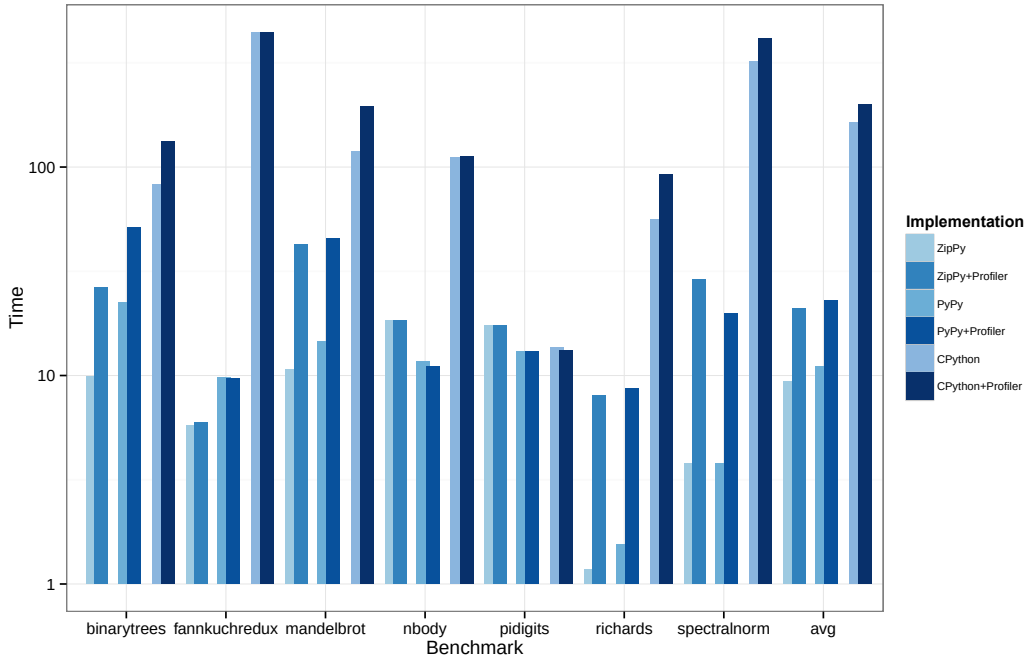

Figure 4: Method profiling comparisons for Python.

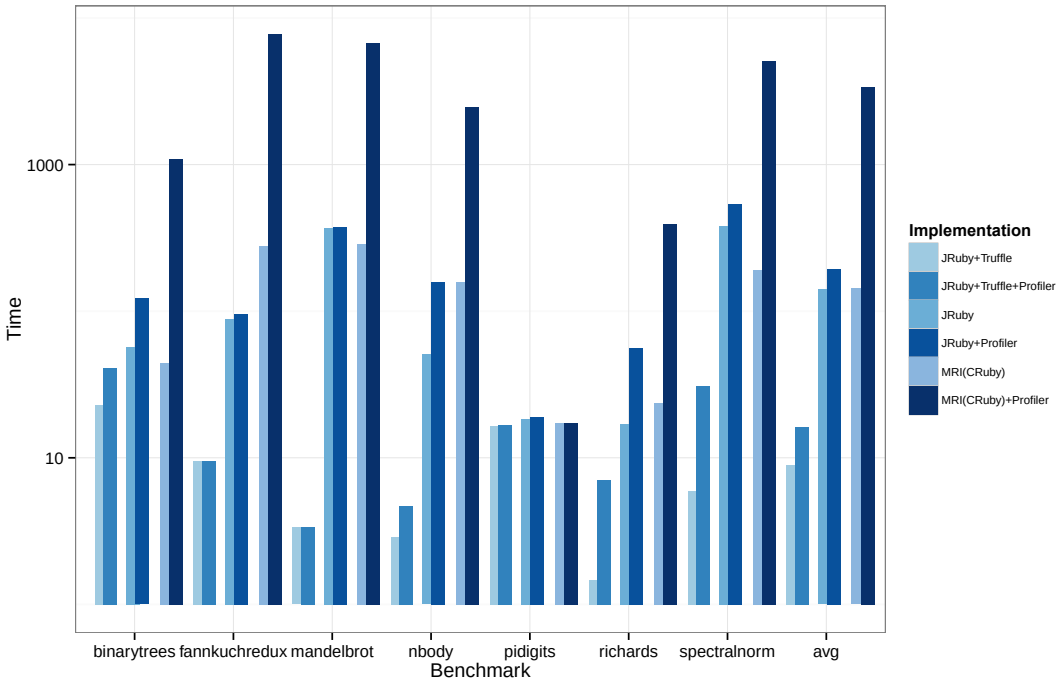

Figure 5: Method profiling comparisons for Ruby.

in PyPy. However, PyPy performs worse with this module, so we only include the numbers from PyPy while using the cProfile module.

Figure 5 shows method profiling performance among different Ruby implementations. We compare our method profiler against MRI (CRuby) using ruby-prof and JRuby's built-in profiler. We report an average speedup of 208× over MRI profiler, and an average speedup of 12× over JRuby's built-in profiler. With and without profiling, JRuby+Truffle performs better than JRuby and MRI in all the benchmarks.

As mentioned before, Ruby represents many of its internals as method calls. Since we are interested in profiling user-defined methods, we exclude operation method calls in MRI profiler. Similarly, JRuby specializes certain basic operation calls, and does not profile them.

We verify the results of our method profilers by comparing the number of method invocations across different profilers. We find out that our method profilers give the same results for the number of method invocations as the profilers that we compare them against.

5. RELATED WORK

Measuring execution frequencies to guide programmers to apply optimizations has a long history. Knuth [14] reports an early study of frequency counts of each statement in Fortran programs in 1971. Graham et al. [9] describe the implementation of the *gprof* performance analysis tool for Unix applications. They developed the tool for Berkeley Unix in 1982, and it has been part of the GNU project since 1988. The gprof tool gathers execution counts and execution time for called subroutines. It uses a combination of instrumentation and sampling. When the compiler compiles the profiled program, it augments the code at the prologue of each subroutine that counts the number of times of each subroutine. The gprof tool also samples the program counter at fixed intervals to collect execution time, so the resulting data from execution time is a statistical approximation.

Ball et al. [1, 2] introduce path profiling that records execution frequency of basic blocks or control-flow edges during an execution for purposes like performance tuning, profile-directed compilation, and test coverage. They describe an algorithm to insert less instrumentation code, and place it in less frequently executed parts of the program to reduce the instrumentation overhead. Unlike gprof, their technique records exact execution frequency, not a statistical sample. In 1999, Melski et al. [17] extend this technique to collect information about interprocedural paths.

Traditional profilers perform offline, that is, they collect information in a separate run, and then use the collected information afterwards. However, dynamic compilation systems such as JIT compilers need to collect information and consume it online. Oren et al. [21] describe an online path profiling in 2002.

The Java Virtual Machine Profiling Interface (JVMTI) [20] is an interface to tools that need an access to VM state, such as profiling, debugging, monitoring, and thread analysis. The interface provides support for bytecode instrumentation to modify the Java bytecode in a Java program. It can maintain exact counters or statistically sample events.

Binder et al. [4] describe their Java Profiler tool JP that instruments the bytecode of Java programs to profile the number of executed methods and bytecode in a given program. JP is an event-based profiler implemented in pure Java, and its implementation does not rely on JVMTI.

Microsoft provides Visual Studio Profiler [18] for the Windows .NET platform. It supports a single profiling environment for the languages running on this platform such as C, C++, Visual Basic, C♯, etc. It provides cross-language support which allows interaction with code written in a different programming language.

Bergel [3] introduces Compteur, which is the message-based code profiler for the Pharo implementation of the Smalltalk language. Pharo is an object-oriented programming language and environment in the tradition of Smalltalk, and relies on message passing. Compteur uses message counting as a profiling metric, and shows more stable measurements than execution sampling.

Morandat et al. [19] describe ProfileR, a profiling tool for the VM they implement for R language. It is an event-based profiler measuring the time spent in operations such as memory management, I/O, and foreign calls.

Our technique differs from existing profilers in several aspects: First, we instrument the AST nodes generated from the source code whereas they instrument the compiled binary code, or managed bytecode. Second, the inserted nodes in our implementation are subject to full runtime optimizations, and could be activated or deactivated at runtime. Third, some of the existing profilers use sampling, whereas our profiler is event-based tracking every occurrence of events. Fourth, some of the existing techniques require significant implementation effort, but our profiler requires little implementation. Fifth, our profiler makes it possible to compare the programs implementing the same algorithms across different languages. Lastly, our technique benefits both the language implementer and the user.

6. CONCLUSION

We implement a profiler framework that makes it possible to compare languages on cross-language benchmarks implementing the same algorithms. Our profiler is an event-based profiler that provides a more comprehensive profiling to further investigate dynamic language programs. We implement our high-performance profiler for dynamic languages in the context of the Truffle framework that optimizes AST interpreters with a JIT compiler. Our generic profiler framework minimizes profiler implementation effort. It benefits both the language user and implementer, and compares the programs implementing the same algorithms across different languages. We evaluate our profiler framework on Python and Ruby implementations, however, it is applicable to any language running on top of Truffle. Our profiler runs faster than existing profilers on average and requires modest effort.

As future work, we plan to add a user-friendly visual interface to our profiler. For instance, we plan to add syntax highlighting to show the frequently executed parts of the program in the source code. Similarly, we plan to display the distribution of types in the source program.

7. ACKNOWLEDGMENTS

We thank Chris Seaton for his help on the JRuby+Truffle implementation, as well as the anonymous reviewers for their insightful feedback and suggestions.

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts D11PC20024 and N660001-1-2-4014, by the

National Science Foundation (NSF) under grant No. CCF-1117162, and by a gift from Oracle Labs.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.

8. REFERENCES

[1] T. Ball and J. R. Larus. Optimally Profiling and Tracing Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, July 1994.

[2] T. Ball and J. R. Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[3] A. Bergel. Counting Messages As a Proxy for Average Execution Time in Pharo. In *Proceedings of the 25th European Conference on Object-Oriented Programming*, ECOOP'11, pages 533–557, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] W. Binder, J. Hulaas, P. Moret, and A. VillazŮn. Platform-independent Profiling in a Virtual Execution Environment. *Software: Practice and Experience*, 39(1), 2009.

[5] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the Meta-level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[6] L. P. Deutsch and A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

[7] B. Fulgham. The Computer Language Benchmarks Game. http://benchmarksgame.alioth.debian.org.

[8] Google. V8 Benchmark Suite. http://v8.googlecode.com/svn/data/benchmarks/current/run.html.

[9] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM.

[10] M. Grimmer, C. Seaton, T. Würthinger, and H. Mössenböck. Dynamically Composing Languages in a Modular Way: Supporting C Extensions for Dynamic Languages. In *Proceedings of the 14th International Conference on Modularity*, MODULARITY 2015, pages 1–13, New York, NY, USA, 2015. ACM.

[11] C. Humer, C. Wimmer, C. Wirth, A. Wöß, and T. Würthinger. A Domain-Specific Language for Building Self-Optimizing AST Interpreters. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 123–132, New York, NY, USA, 2014. ACM.

[12] JRuby. JRuby. http://jruby.org.

[13] JRuby. JRuby Built-in Profiler. https://github.com/jruby/jruby/wiki/Profiling-jruby.

[14] D. E. Knuth. An Empirical Study of FORTRAN Programs. In *Software - Practice and Experience*, volume 1, 1971.

[15] S. Marr, C. Seaton, and S. Ducasse. Zero-Overhead Metaprogramming: Reflection and Metaobject Protocols Fast and Without Compromises. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 545–554, New York, NY, USA, 2015. ACM.

[16] Y. Matsumoto. Ruby. https://www.ruby-lang.org/en/.

[17] D. Melski and T. W. Reps. Interprocedural Path Profiling. In *Proceedings of the 8th International Conference on Compiler Construction, Held As Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, CC '99, pages 47–62, London, UK, UK, 1999. Springer-Verlag.

[18] Microsoft. Visual Studio Profiler. http://msdn.microsoft.com/en-us/magazine/cc337887.aspx.

[19] F. Morandat, B. Hill, L. Osvald, and J. Vitek. Evaluating the Design of the R Language: Objects and Functions for Data Analysis. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 104–131, Berlin, Heidelberg, 2012. Springer-Verlag.

[20] Oracle Corporation. Java Virtual Machine Tool Interface. http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html.

[21] D. Oren, Y. Matias, and S. Sagiv. Online Subpath Profiling. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 78–94, London, UK, UK, 2002. Springer-Verlag.

[22] PyPy. PyPy. http://pypy.org.

[23] Python Software Foundation. Python. http://www.python.org.

[24] Python Software Foundation. The Python Profilers. https://docs.python.org/3.3/library/profile.html.

[25] Ruby. Fast Code Profiler for Ruby. https://github.com/ruby-prof/ruby-prof.

[26] C. Seaton, M. L. Van De Vanter, and M. Haupt. Debugging at Full Speed. In *Proceedings of the Workshop on Dynamic Languages and Applications*, Dyla'14, pages 2:1–2:13, New York, NY, USA, 2014. ACM.

[27] Sun Microsystems Inc. JSR 292: Supporting Dynamically Typed Languages on the Java Platform. http://www.bibsonomy.org/bibtex/2c5990e5bf854a87a7d441e129cef8193/gron.

[28] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. ACM.

[29] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM.

[30] W. Zhang, P. Larsen, S. Brunthaler, and M. Franz. Accelerating Iterators in Optimizing AST Interpreters. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 727–743, New York, NY, USA, 2014. ACM.