GraalSqueak

A Fast Smalltalk Bytecode Interpreter Written in an AST Interpreter Framework

Fabio Niephaus
Hasso Plattner Institute,
University of Potsdam
Potsdam, Germany
fabio.niephaus@hpi.uni-potsdam.de

Tim Felgentreff
Oracle Labs
Potsdam, Germany
tim.felgentreff@oracle.com

Robert Hirschfeld
Hasso Plattner Institute,
University of Potsdam
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

Abstract

Language implementation frameworks aim to provide everything that is needed to build interpreters, simplify the process by making certain design decisions in advance, and suggest implementation strategies to virtual machine creators. Truffle, the language implementation framework for the GraalVM, is designed for building Abstract Syntax Tree interpreters and the process of doing so is well documented. However, although less documented, Truffle can also be used to implement bytecode interpreters. This approach requires additional hints to be passed into the compiler to gain good performance.

In this paper, we compare two Truffle interpreters for Squeak/Smalltalk, one using an AST implementation approach and the other executing bytecodes. While both run at roughly three times the speed of the standard Squeak/Smalltalk virtual machine, both represent different trade-offs in implementation strategies for interpreters in Truffle. We compare these trade-offs and discuss the advantages and disadvantages of the different approaches.

CCS Concepts • **Software and its engineering → Runtime environments**; *Interpreters*; *Integrated and visual development environments*;

Keywords Interpreters, Truffle, GraalVM, Squeak, Smalltalk, Language implementation frameworks, RPython

ACM Reference Format:
Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2018. GraalSqueak: A Fast Smalltalk Bytecode Interpreter Written in an AST Interpreter Framework. In *13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'18), July 17, 2018, Amsterdam, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3242947.3242948

ICOOOLPS'18, July 17, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5804-0/18/07…$15.00
https://doi.org/10.1145/3242947.3242948

1 Introduction and Background

Programming language implementation frameworks have become more and more popular as they allow language implementers to use another high-level language and useful components, such as garbage collectors or caching mechanisms, to implement virtual execution environments for dynamic languages. These frameworks, however, often enforce certain implementation styles and are usually designed to support a specific kind of interpretation model. RPython [1], a language implementation framework maintained as part of the PyPy project [10], for example, is mainly used to implement bytecode interpreters, because its tracing just-in-time (JIT) compiler is most suited to operate on bytecode. Oracle's Truffle framework [14], on the other hand, is designed for implementing Abstract Syntax Tree (AST) interpreters and its JIT compiler applies AST node rewriting and partial evaluation to significantly increase run-time performance of corresponding interpreters. Consequently, most languages implementations in Truffle are AST-based.

However, some language specifications include a well-defined bytecode set and are therefore designed to run on bytecode interpreters. For this reason, Truffle has optimization mechanisms specifically for building bytecode-based interpreters. These are used, for example, in Sulong [9], a Truffle-based interpreter for LLVM bitcode. The process and pitfalls of implementing bytecode interpreters in Truffle is not well documented which is one motivation for this paper.

Squeak/Smalltalk [6] is a Smalltalk dialect derived from the Smalltalk-80 language specification [5]. OpenSmalltalkVM [8], the default Virtual Machine (VM) for Squeak/Smalltalk, is bytecode-based and features a mostly handwritten JIT compiler. RSqueak/VM [3, 4] is an alternative VM for it written in RPython. With SOMns [7], a Smalltalk-like interpreter is already implemented in Truffle. However, it operates entirely on ASTs as it is not image-based like traditional Smalltalk-80 systems and can therefore be well optimized by the GraalVM [14], which in turn is the VM on which Truffle language interpreters are designed to run.

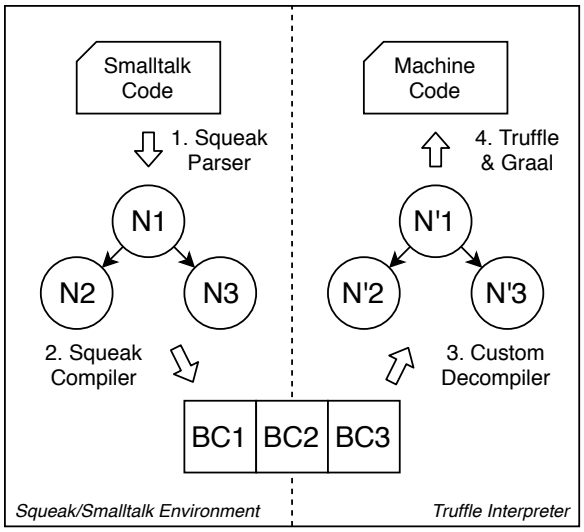

Figure 1. Architecture of an AST interpreter for Squeak

In this paper, we present and compare both the Truffle-based implementation of an AST interpreter as well as a bytecode interpreter for Squeak/Smalltalk. We report our experiences implementing the different interpretation models in Truffle, discuss implementation pitfalls, and compare the performance of the different interpreters with two benchmarks.

2 Implementations

In this section, we present two different approaches for implementing Squeak/Smalltalk interpreters in Truffle. The source code of the GraalSqueak interpreter variations is available on GitHub[1].

AST Interpreter Since Truffle and the Graal compiler operate on ASTs, the natural way of implementing a Squeak/Smalltalk interpreter in Truffle is to write an AST interpreter. However, Squeak/Smalltalk is traditionally bytecode-based, with its compiler written in Squeak itself and only the bytecodes, not the ASTs or sources stored in the image, are visible to the VM. Therefore, these bytecode streams need to be transformed into appropriate AST nodes again.

Figure 1 gives an overview of the architecture of this implementation. Each Smalltalk method is first parsed and compiled to bytecode inside the Smalltalk environment. Upon loading an image, the Truffle interpreter only has access to compiled code objects which hold Squeak/Smalltalk bytecode.

To transform the bytecode into Truffle AST nodes, we have ported Squeak/Smalltalk's decompiler to Truffle. This implementation approach was straightforward, but did not yield great performance. The key optimization to gain good

[1] https://github.com/hpi-swa/graalsqueak/releases

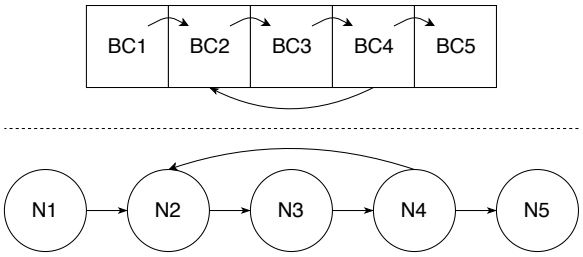

Figure 2. A bytecode stream and a sequence of AST nodes

performance (and the only deviation from the Smalltalk decompiler) was for us to reconstruct loop nodes. By default, the decompiler would turn every loop into a while node with the loop condition in the body and a break out of the loop if the condition becomes `false`. In Truffle, on the other hand, there is a dedicated `LoopNode` for which the runtime provides special optimization strategies. To optimize loops well, we had to modify the decompilation to split the condition from the body appropriately and use Truffle's `LoopNodes`. This significantly improved the performance of the interpreter on the GraalVM.

Bytecode Interpreter Implementing a bytecode interpreter in Truffle is not as intuitive and still requires generating AST nodes. However, these ASTs are simply linked lists with back pointers. We create one node for each bytecode and generate a chain of AST nodes as depicted in Figure 2. In Smalltalk, each of these nodes has either exactly one or, in the case of a conditional jump, two successors because these jumps are the only way to branch in Squeak/Smalltalk.

```
void executeLoop(VirtualFrame frame) {
  int pc = 0;
  while (pc >= 0) {
    pc = bytecodeNodes[pc].executeInt(frame);
  }
}
```

Listing 1. A simple loop for interpreting sequences of AST nodes

Consequently, we can implement an interpreter loop as shown in 1. However, running this loop with Truffle on the GraalVM gives rather low performance.

As Rigger et. al [9] have shown, the Graal compiler needs additional information to efficiently execute bytecode loops. Their optimization encodes the possible successor program counters in an immutable array of Java primitive integers on each bytecode node. This way, the compiler knows that most bytecode nodes have exactly one possible successor node, and can optimize these together.

However, Graal does not automatically detect control-flow cycles in such a loop. Instead, it reports escaping frame

```
1  @ExplodeLoop(kind = ExplodeLoop.LoopExplosionKind.MERGE_EXPLODE)
2  void executeLoop(VirtualFrame frame) {
3   CompilerAsserts.compilationConstant(bytecodeNodes.length);
4   int pc = 0; int backJumpCounter = 0;
5   try {
6    while (pc >= 0) {
7     CompilerAsserts.partialEvaluationConstant(pc);
8     AbstractBytecodeNode node = bytecodeNodes[pc];
9     if (node instanceof ConditionalJumpNode) {
10     ConditionalJumpNode jumpNode = (ConditionalJumpNode) node;
11     boolean condition = jumpNode.executeCondition(frame);
12     if (CompilerDirectives.injectBranchProbability(
13           jumpNode.getProbability(JUMP), condition)) {
14      int successor = jumpNode.getJumpSuccessor();
15      if (CompilerDirectives.inInterpreter()) {
16       jumpNode.increaseProbability(JUMP);
17       if (successor <= pc) backJumpCounter++;
18      }
19      pc = successor; continue;
20     } else {
21      int successor = jumpNode.getNoJumpSuccessor();
22      if (CompilerDirectives.inInterpreter()) {
23       jumpNode.increaseProbability(NO_JUMP);
24       if (successor <= pc) backJumpCounter++;
25      }
26      pc = successor; continue;
27     }
28    } else if (node instanceof UnconditionalJumpNode) {
29     UnconditionalJumpNode jumpNode = (UnconditionalJumpNode) node;
30     int successor = jumpNode.getJumpSuccessor();
31     if (CompilerDirectives.inInterpreter()) {
32      if (successor <= pc) backJumpCounter++;
33     }
34     pc = successor; continue;
35    } else { pc = node.executeInt(frame); }
36   }
37  } finally {
38   LoopNode.reportLoopCount(this, backJumpCounter);
39  }
40 }
```

Listing 2. Bytecode loop with hints for the Graal compiler

errors when trying to unroll the loop. This is also a problem for other bytecode interpreters such as Sulong, but was not further described by Rigger et. al. In order to inform Truffle about our bytecode interpreter, we had to add additional compiler annotations and hints. These are now explained in more detail.

Figure 2 shows the revised version of the interpreter loop with appropriate Truffle hints for the Graal compiler. The `@ExplodeLoop` annotation in line 1 instructs the compiler to unroll loops. In our case, it uses the `MERGE_EXPLODE` strategy which is designed especially for bytecode interpreters as it tries to explode all loops while merging copies of the loop body that have identical state. Then we assert that the number of bytecodes is constant per method instance (line 3) and ensure that the program counter is reduced to a constant during the partial evaluation phase (line 7). After fetching the next node for the current program counter, there are three options how control flow can continue.

First, if the node is a conditional jump node, the condition is being executed (line 11). Each conditional jump node maintains a probability value that represents how often the condition is `true` or `false`. Truffle provides an API to inject such values as a branch probability (line 12 and 13) which further supports the Graal compiler. Afterwards, the next program counter is determined depending on whether the condition was `true` or `false` (line 14 and 21). Additionally and only if executed in the interpreter, the corresponding branch probability is increased and if the successor is smaller than the current program counter, a `backJumpCounter` is incremented. This `backJumpCounter` is reported to the compiler through the `LoopNode::reportLoopCount` API on method exit as part of the `finally` block in line 38. This information is used in Truffle's optimization heuristics to further improve the compilation process. Lastly, the successor becomes the current program counter and the interpreter loop continues with the next bytecode.

Second and in the case of an unconditional jump, the next program counter is fetched and analyzed for backward jumps if running interpreted (line 29 to 34).

Otherwise, the current node is fully executed to determine the next program counter (line 35), just like it was in the simple version of the interpreter loop.

The Javadocs for the different Truffle hints provide further information on how they work or can be used, yet are unable to fully explain how to use them in combination with others or how exactly the Graal compiler benefits from them.

3 Evaluation

To assess the performance of our Truffle-based interpreters for Squeak/Smalltalk, we have implemented all bytecodes and primitives required to run Squeak's `tinyBenchmarks`. This micro-benchmark suite is often used to measure and compare the performance of different hardware platforms and Squeak VMs [3, 12] and consists of two benchmarks: The first one is bytecode-heavy as it allocates, fills, and reads from a large array. The other one is a recursive Fibonacci benchmark and therefore send-heavy. Additionally, `tinyBenchmarks` adjusts both benchmarks so that they run at least one second in order to produce more stable results. Although the results should be taken with a grain of salt as they do not represent a wide range of common operations, we believe they are a good indicator for the overall performance of our different interpreter approaches.

We ran the benchmarks on a 15-inch MacBook Pro from Mid 2015 (CPU: 2.5 GHz Intel Core i7; Memory: 16 GB 1600 MHz DDR3). For the AST interpreter, we used commit `8126c1b` of GraalSqueak and `ac530ac` for the bytecode interpreter with Truffle hints. Moreover, we copied the latter version of GraalSqueak and replaced `executeLoop(VirtualFrame frame)` defined on `SqueakMethodNode` with the code from 1 to remove all compiler hints. We also ran the benchmarks on other Squeak/Smalltalk VMs, a recent OpenSmalltalkVM (tag `201804030952`), the fastest stable VM for Squeak/Smalltalk, as well as RSqueak/VM (commit `d33005c`). Please note that, compared to these complete VM implementations, our interpreters do not have an interrupt handler and do not support Smalltalk context objects which might have a negative impact on performance when implemented. Furthermore, we used 100 iterations per run and all benchmarks took a little less than an hour to run in total. Nonetheless, we observed that all results stabilized within the first ten iterations.

The left half of Figure 3 shows the benchmark results of the bytecode-heavy micro-benchmark. The results of the OpenSmalltalkVM can be treated as the baseline as it is the default VM for Squeak/Smalltalk. It performs relatively stable at around three billion bytecodes per second while RSqueak/VM is able to process approximately 2.1 billion bytecodes per second. When looking at the results of the AST-based GraalSqueak implementation, we notice warmup behavior. After around two iterations, performance reaches a somewhat stable state. At the same time, it outperforms the OpenSmalltalkVM by approximately 3.06x. The bytecode interpreter, on the other hand, does not show this warmup behavior and is, with an average of ten billion bytecodes per second, the fastest VM. Without hints, however, Truffle is unable to perform its optimizations due to escaping frames and the otherwise identical interpreter performs very poorly.

The other half of Figure 3 shows the recursive Fibonacci benchmark. RSqueakVM performs much worse compared to the OpenSmalltalkVM. Although the GraalSqueak interpreters still outperform it, the relative difference is not as big as in the previous benchmark. The AST interpreter is approximately 1.42x faster and the bytecode interpreter with Truffle hints around 1.47x. Again, the performance of the bytecode interpreter without hints is very low compared to all other VMs.

Since the performance gap between the bytecode interpreter with and without Truffle hints is approximately three orders of magnitude large, we zoom in on the "GS-hints" results from Figure 4 and add two additional data series: the results of the bytecode interpreter with and without hints running fully interpreted on a standard JDK 1.8.0_144 and without the Graal compiler. Figure 4 shows the results and suggests that the hint-less GraalSqueak bytecode interpreter on the GraalVM performs roughly the same as the interpreted version with hints on the Java Virtual Machine (JVM). Compared to these two, the bytecode interpreter without Truffle hints performs better on the JVM. Therefore, these additional Truffle annotations introduce a measurable overhead which negatively impacts interpretation performance.

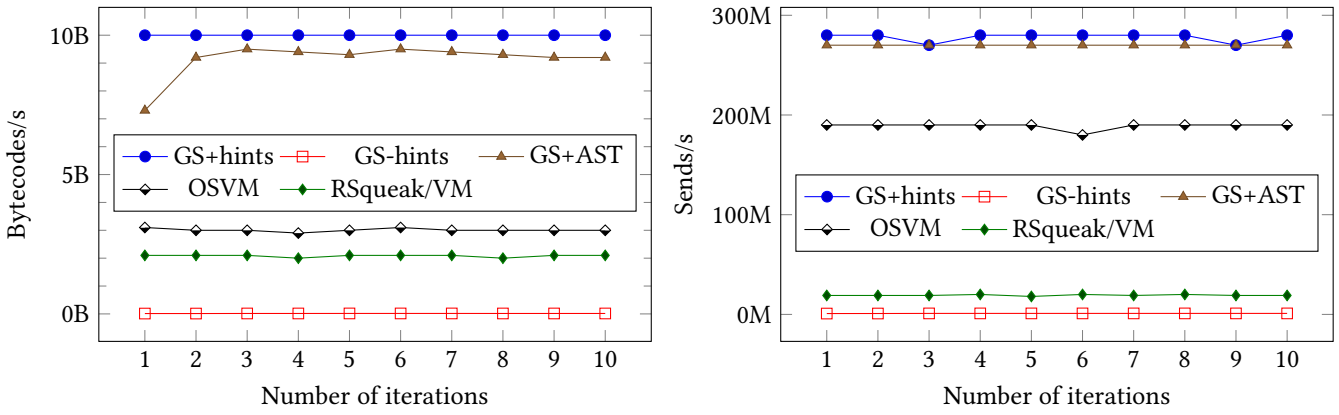

Figure 3. `tinyBenchmarks` results of different Squeak/Smalltalk VMs

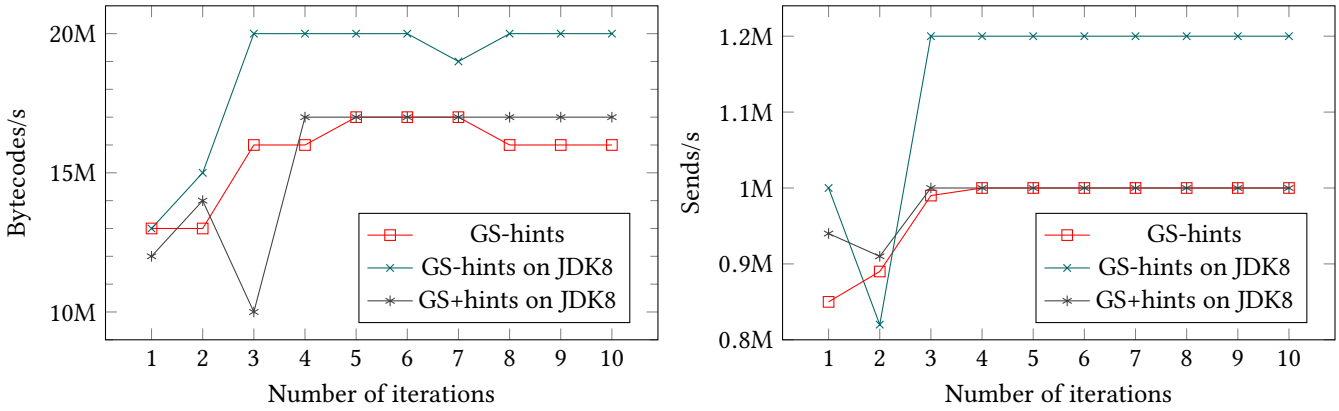

Figure 4. `tinyBenchmarks` results of the hint-less bytecode interpreter compared to pure interpreter performance on JDK8

4 Related Work

Sulong Sulong [9] is a bytecode-based interpreter for LLVM bitcode, written in Truffle, and maintained by Oracle Labs as part of the GraalVM project. Its bytecode loop employs similar Truffle hints to support the compiler in optimizing run-time performance.

TruffleRuby TruffleRuby [13] is a Truffle implementation of the Ruby programming language. Similar to our AST-based GraalSqueak implementation, TruffleRuby uses a custom parser for generating Truffle ASTs. Since Ruby 1.9, the default runtime for Ruby is YARV [11] which operates on bytecode.

OpenSmalltalkVM and Sista The OpenSmalltalkVM [8] is the default VM for Squeak/Smalltalk and variations of it include Sista [2] which stands for "Speculative Inlining SmallTalk Architecture". Instead of optimizing code purely on VM-level, the VM provides an API which can be used from inside a Smalltalk environment to retrieve profiling information. This information can then be used to apply optimizations on image-level which can also be persisted when saving the image.

RSqueak/VM RSqueak/VM [3] is an alternative interpreter for Squeak/Smalltalk and written in the language implementation framework RPython. Therefore, it leverages the same meta-tracing JIT compiler that is also used in PyPy to optimize its bytecode loop.

SOMns SOMns [7] is an implementation of the Newspeak language in Truffle. Since it is completely file-based, it does not provide compatibility with image-based Newspeak and traditional Smalltalk systems. Our GraalSqueak interpreters, on the other hand, are designed to be compatible with existing Squeak/Smalltalk images. This also includes language features such as sender modifications or support for various VM plugins.

5 Conclusion and Future Work

In this paper, we presented two different approaches for implementing interpreters in the Truffle language implementation framework given a language defined through and/or with bytecode as its interchange format such as Squeak/Smalltalk.

To create an AST interpreter for such a language requires a custom decompiler which is able to generate Truffle ASTs from bytecode. Besides the decompiler, the main requirement to achieve good performance on Truffle is that the decompiler needs to detect loops and emit appropriate Truffle `LoopNodes`.

A bytecode interpreter, on the other hand, does not need a decompiler, which reduces the complexity of the interpreter. To achieve good execution performance, however, the bytecode loop needs to be extended with several runtime hints to allow Truffle and the Graal compiler to successfully apply appropriate optimizations.

Our initial benchmarks show that a bytecode interpreter can be just as fast as an AST-based implementation in Truffle after warmup. Interpreted performance, on the other hand, is negatively impacted by additional Truffle hints, suggesting further optimization potential.

In the future, we want to extend GraalSqueak with full support for Squeak/Smalltalk context objects and more primitives as well as VM plugins, so that the entire programming environment can be used as intended. With a complete implementation of Squeak/Smalltalk in Truffle, the GraalVM also allows for interesting experiments in the area of ploygot programming.

Acknowledgments

We would like to thank Manuel Rigger and the anonymous reviewers for their feedback on this paper. We gratefully acknowledge the financial support of Oracle Labs[2], HPI's Research School[3], and the Hasso Plattner Design Thinking Research Program[4].

References

[1] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. 2007. RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *Proceedings of the 2007 Symposium on Dynamic Languages (DLS '07)*. ACM, New York, NY, USA, 53–64. https://doi.org/10.1145/1297081.1297091

[2] Clément Béra, Eliot Miranda, Tim Felgentreff, Marcus Denker, and Stéphane Ducasse. 2017. Sista: Saving Optimized Code in Snapshots for Fast Start-Up. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes (ManLang 2017)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/3132190.3132201

[3] Carl Friedrich Bolz, Adrian Kuhn, Adrian Lienhard, Nicholas D. Matsakis, Oscar Nierstrasz, Lukas Renggli, Armin Rigo, and Toon Verwaest. 2008. *Back to the Future in One Week — Implementing a Smalltalk VM in PyPy*. Springer-Verlag, Berlin, Heidelberg, 123–139. https://doi.org/10.1007/978-3-540-89275-5_7

[4] Tim Felgentreff, Tobias Pape, Patrick Rein, and Robert Hirschfeld. 2016. How to Build a High-Performance VM for Squeak/Smalltalk in Your Spare Time: An Experience Report of Using the RPython Toolchain. In *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies (IWST '16)*. ACM, New York, NY, USA, Article 21, 10 pages. https://doi.org/10.1145/2991041.2991062

[5] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman, Boston, MA, USA. The Blue Book.

[6] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. *SIGPLAN Not.* 32, 10 (10 1997), 318–326. https://doi.org/10.1145/263700.263754

[7] Stefan Marr, Carmen Torres Lopez, Dominik Aumayr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2017. A Concurrency-Agnostic Protocol for Multi-Paradigm Concurrent Debugging Tools. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Dynamic Languages (DLS'17)*. ACM, 12. https://doi.org/10.1145/3133841.3133842

[8] Eliot Miranda and contributors. 2017. OpenSmalltalkVM. (2017). https://github.com/OpenSmalltalk/opensmalltalk-vm

[9] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages (VMIL 2016)*. ACM, New York, NY, USA, 6–15. https://doi.org/10.1145/2998415.2998416

[10] Armin Rigo and Samuele Pedroni. 2006. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (OOPSLA '06)*. ACM, New York, NY, USA, 944–953. https://doi.org/10.1145/1176617.1176753

[11] Koichi Sasada. 2005. YARV: Yet Another RubyVM: Innovating the Ruby Interpreter. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 158–159. https://doi.org/10.1145/1094855.1094912

[12] Squeak/Smalltalk Community. 2018. FAQ: Speed. (2018). http://wiki.squeak.org/squeak/768

[13] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. *SIGPLAN Not.* 52, 6 (June 2017), 662–676. https://doi.org/10.1145/3140587.3062381

[14] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. 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)*. ACM, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[2] https://labs.oracle.com/

[3] https://hpi.de/en/research/research-school.html

[4] https://hpi.de/en/dtrp/