Are We There Yet?

Simple Language-Implementation Techniques for the 21st Century

Stefan Marr, Tobias Pape, and Wolfgang De Meuter

Abstract—With the rise of domain-specific languages (DSLs), research in language implementation techniques regains importance. While DSLs can help to manage the domain's complexity, it is rarely affordable to build highly optimizing compilers or virtual machines, and thus, performance remains an issue. Ideally, one would implement a simple interpreter and still reach acceptable performance levels. RPython and Truffle are two approaches that promise to facilitate language implementation based on simple interpreters, while reaching performance of the same order of magnitude as highly optimizing virtual machines. In this case study, we compare the two approaches to identify commonalities, weaknesses, and areas for further research to improve their utility for language implementations.

Index Terms—Language Implementation, Virtual Machines, Compilers, Interpreters.

1 Introduction

Programming language implementation has been a major field of research since the beginning of computer science. Over the years, many advances were made leading to highly optimizing *compilers* and dynamic just-in-time (JIT) compilation techniques [1] allowing for efficient implementation of a wide range of programming languages. However, the implementation of compilers as well as JIT compiling virtual machines (VMs) require significant engineering effort, which often goes well beyond what is feasible for instance for domain-specific languages (DSLs). Thus, interpreters remain a simple alternative for the implementation of programming languages. Interpreters typically reflect directly the desired execution semantics and hence retain a close relation between the language and its execution. This makes them ideal for evolving languages, design experiments, and DSLs. The downside of interpreters however was so far that they are several orders of magnitude slower than highly optimizing VMs or compilers.

In this paper, we investigate language implementation techniques that enable the use of interpreters, while at the same time yielding performance that is supposed to reach the same order of magnitude as compilers and JIT compiling VMs. Specifically, we study RPython [2], which is know as the implementation platform for PyPy, a fast Python implementation, as well as Truffle [3], which is a new language implementation framework for the JVM that also aims to enable efficient language implementations.

- *S. Marr is with INRIA, Lille, France. E-mail: stefan.marr@inria.fr*
- *T. Pape is with the Hasso Plattner Institute, University of Potsdam, Germany. E-mail: Tobias.Pape@hpi.uni-potsdam.de*
- *W. De Meuter is with the Vrije Universiteit Brussels, Belgium.*

Note this is the authors' version, the final version of this paper is published in IEEE Software. DOI: 10.1109/MS.2014.98. http://stefan-marr.de/papers/ieee-soft-marr-et-al-are-we-there-yet/

2 SOM: Simple Object Machine

As running example and case study for this paper, we use SOM (Simple Object Machine) [4], a simple Smalltalk [5]. SOM is designed for teaching language implementation and VM techniques. Therefore, it is kept simple and includes only a small set of fundamental language concepts such as *objects*, *classes*, *closures*, *primitives*, and *non-local returns*. The following listing illustrates the SOM syntax:

```
1  Boolean = Object (
2    ifTrue: trueBlock ifFalse: falseBlock = (
3      self ifTrue:  [ ^trueBlock value  ].
4      self ifFalse: [ ^falseBlock value ].
5    )
6    "..."
7  )
8
9  True = Boolean (
10   ifTrue:  block = ( ^block value )
11   ifFalse: block = ( ^nil )
12   "..."
13 )
```

Here, the two classes `Boolean` and `True` are sketched. Line 1 defines the class `Boolean` as a

subclass of `Object`. In Smalltalk, control structures such as `if` or `while` are defined as polymorphic methods on objects and rely on mechanisms such as *closures* and *non-local returns*. Thus, `Boolean` defines the method `#ifTrue:ifFalse:` on line 2, which takes two blocks, i. e., closures, as arguments. When this message is sent to a boolean, line 3 is executed and will evaluate the `trueBlock` parameter by sending the message `#value`. The caret (`^`) is Smalltalk's return symbol, and causes a *non-local return* in this case. This means, if the boolean is an instance of the class `True`, the `trueBlock` is evaluated and the result is returned to the caller of `#ifTrue:ifFalse:`. Without the notion of non-local returns, execution would not return from `#ifTrue:ifFalse:` and line 4 would be executed as well.

3 Implementing Interpreters

Interpreters are commonly implemented using one of two approaches. The simpler is based on *abstract syntax trees* (ASTs) generated by a parser. The other approach uses a linearized program representation, typically referred to as *bytecodes*. For illustration of the difference between the two approaches, the following SOM listing shows a recursive factorial function.

```
factorial: n = (
  ^ n = 0
      ifTrue:  [1]
      ifFalse: [n * (self factorial: n - 1)])
```

The comparison `n = 0` results in a boolean object, which receives the message `#ifTrue:ifFalse:`. In case `n` is `0`, the `#factorial:` method will return the value of the first block, i. e., `1`. Otherwise, it evaluate the second block, which recursively computes the factorial for `n`.

3.1 AST Interpreters

The SOM parser compiles the example code to an AST similar to fig. 1, where each AST node is annotated with its type. An AST interpreter then implements for each node type the corresponding actions as sketched in the following listing:

```
class IntegerLiteral(ASTNode):
  def __init__(self, value):
    self._value = value

  def execute(self, frame):
    return self._value
```

```
class BinaryMessageSend(ASTNode):
  def __init__(self, selector, rcvr_ex, arg_ex):
    self._sel  = selector
    self._rcvr_ex = rcvr_ex
    self._arg_ex  = arg_ex

  def execute(self, frame):
    rcvr = self._rcvr_ex.execute(frame)
    arg  = self._arg_ex.execute(frame)

    method = rcvr.get_class().lookup(self._sel)
    return method.invoke(frame, rcvr, arg)
```

Note, we use Python pseudo-code to distinguish between the language implemented and its implementation language. The `IntegerLiteral` class illustrates how literals such as the integer `1` are handled. Since their value is known at compilation time, the `execute` method of `IntegerLiteral` just returns this stored value. In contrast, the `BinaryMessageSend` has two subexpressions, the receiver and the argument, that have to be evaluated first. Thus, `execute` first calls the `execute` methods on the subexpressions. Then, it retrieves the receiver's class and looks up the Smalltalk method that corresponds to the selector defined in the AST. In the factorial example, the arithmetic operations, the comparison, as well as `#factorial:` itself are such binary message sends.

While it is straightforward to build interpreters this way, the overhead of the dynamic dispatch for methods such as `execute` and the memory access patterns of the tree traversal cause performance issues. As a consequence, bytecode interpreters are advocated as the favorable alternative. For example, Ruby as well as WebKit JavaScript switched in 2007/2008 from AST-based to bytecode-based interpreters and gained better performance.

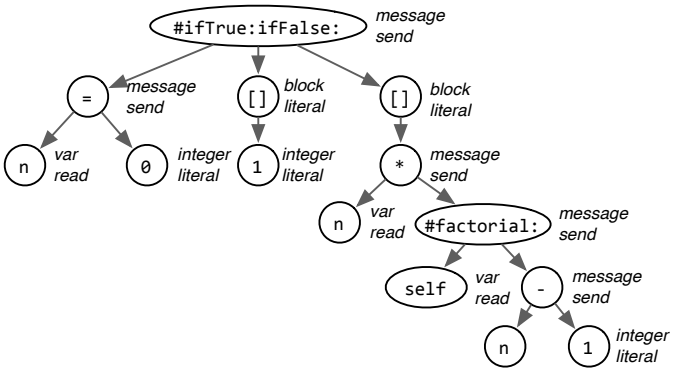

Fig. 1. Abstract syntax tree for the factorial function

3.2 Bytecode Interpreters

For bytecode interpreters, the AST is first *linearized* to represent the program as a sequence of instructions, typically called *bytecode*. This bytecode is executed by a stack- or register-based machine, which includes operations such as `push`, `pop`, `move`, `invoke method`, and `jump`. Often additional instructions for common program patterns are included to reduce the interpreter overhead. Examples include the VMs for Smalltalk-80 as well as Java.

For SOM, the bytecode set for a stack-based interpreter has 16 bytecodes. Compiling the factorial example from section 2 to this bytecode yields the following:

```
push_argument 1
push_constant Integer(0)
send #=
push_block Block([1])
push_block Block([n * self factorial...])
send #ifTrue:ifFalse:
return_local
```

While the AST interpreter evaluates subexpressions of a node directly, the bytecode interpreter uses a stack to communicate intermediate values. Thus, to invoke the `#=` method, the interpreter first pushes the argument `n` and then the integer constant `0` on the stack. The `#=` method consumes these arguments and pushes a boolean as result onto the stack. Such an interpreter could look like this pseudo-code:

```
def Interpreter.interpret(self, method, frame):
 bc_idx = 0
 while True:
  bc, arg = method.get_bytecode(bc_idx)
  switch bc:
   case push_argument:
     frame.push(frame.get_argument(arg))
   case push_constant:
     frame.push(method.get_const(arg))
   case send:
     select = method.get_const(arg)
     args = frame.pop_n_elems(select.arg_count)
     next_method = args[0].get_class().
                     lookup(select)
     next_method.invoke(self, args)
  bc_idx += 1
```

Basic bytecode interpreters use simple dispatch loops like these `switch/case` statements to implement the bytecode dispatch. First, it fetches the next bytecode from the method that is currently executed, including a possible argument to the bytecode. Then, it jumps to the corresponding bytecode implementation, which either pushes an argument to the message send onto the stack, pushes a constant encoded in the method, or performs a send.

As mentioned earlier, bytecode interpreters are more complex than AST interpreters. One reason is the explicit execution stack that requires the implementer to carefully track the stack balance, e. g., for method calls. However, this complexity seems to be accepted to reap the performance benefits.

Research on optimizations ignored AST interpreters for a long time [6] and only recently, with Truffle [3] reconsidered them actively. However, for bytecode interpreters, a number of optimizations have been developed. Examples are *threaded interpretation* [7] to reduce the bytecode dispatch overhead as well as *superinstructions* [8] and *quickening* [8, 9] to optimize the bytecode sets for better performance. These are only few examples, however, even when applied consequently, interpreters exhibit much lower performance than optimizing VMs with JIT compilers. RPython as well as Truffle promise to provide such optimizing VMs when provided with bytecode or AST interpreters.

4 RPython: Meta-Tracing JITs

In order to make language implementation simpler, the RPython toolchain [2] takes simple interpreters implemented in a high-level language and generates efficient VMs for them. For performance, it builds upon the notion of tracing JIT compilers [10]. Specific to RPython's approach is the use of meta tracing, i. e., an implementation of SOM does not trace the execution of a SOM program directly but rather the execution of the SOM *interpreter*. In addition to meta tracing, the RPython toolchain provides low-level services such as memory management and object layout. These services are added to the interpreter in several transformation steps that eventually generate C code for the final VM. The RPython language is a restricted subset of Python with a type system that enables the analysis and transformation steps of the interpreter into a VM with a meta-tracing JIT.

4.1 RPySOM: A Tracing JIT for SOM

The RPython-based SOM implementation, named RPySOM, is the bytecode interpreter sketched in section 3.2. To enable the meta-tracing JIT compiler, the bytecode loop needs to be annotated, as the tracing JIT has to be able to recognize the loop based on interpreter state. Thus, the interpreter loop needs to be changed slightly:

```
def Interpreter.interpret(self, method, frame):
 bc_idx = 0
 while True:
  bc, arg = method.get_bytecode(bc_idx)
  jit.jit_merge_point(bc_idx=bc_idx,
                      interp=self, method=method)
  switch bc:
   case push_argument:   #...
```

The `jit_merge_point` tells the VM that a loop occurred when the execution has reached a point where the bytecode index (`bc_idx`), method, and interpreter instance are the same objects as they have been at a previous point. When such a loop has been executed often enough, the VM starts to record a *trace* that is optimized subsequently to remove unnecessary operations such as temporary allocations and repeated side-effect-free function execution. Since the trace records all operations and disregards function boundaries, reordering of operations and removal of redundant operations should yield theoretically peak performance for a specific execution trace.

Since the optimizer has to make conservative assumptions even for such concrete traces, it is necessary to make certain properties explicit with annotations. For instance, functions can have side-effects that are not essential for the execution, e. g., for caching values. With RPython's `@elidable` annotation, the optimizer can be told that it is safe to elide repeated executions from a trace, because it is guaranteed that the function produces for the same input always the same return values. RPySOM uses it, e. g., for the lookup for globals, which use a dictionary that is normally not guaranteed to remain unchanged. Another chance for optimization comes from values that remain constant within the context of a trace, but where the optimizer cannot proof it. In these cases the value can be marked with the `promote` function. In RPySOM, the stack pointer is such a value that is constant within the specific context of a trace. A more in-depth discussion of these optimizations is provided by Bolz and Tratt [11].

4.2 Necessary Optimizations

To generate useful traces for JIT compilation. the tracer needs to be able to identify *hot* loops. This turned out to be a challenge for SOM. Smalltalk with its unified design and minimal set of concepts does not match very well with RPython's heuristics. One of the main issues is the way SOM implements loops. Originally, it used a primitive, i. e., a built-in function of the SOM interpreter, which resets the bytecode index of the loop frame to start over with the execution of the loop body. For the tracer, this approach makes it harder to detect loops.

Initially, the complete reification of the execution stack in the interpreter also puts extra burden on the optimizer because data dependencies become ambiguous.

To address the loop detection and to make data dependencies explicit, we made RPySOM a recursive interpreter. Originally, the `interpret` method of `Interpreter` was activated only once. In the recursive version, every SOM method invocation leads to an additional invocation of `interpret`. The relevant execution state like the frame object, the current method, and current bytecode index is thus kept in the `interpret` method. This makes data dependencies more explicit for the optimizer by decoupling logically independent copies of the variables.

Consequently, the implementation of non-local returns and loops changed. Non-local returns were implemented by walking the chain of frame objects and setting the corresponding fields in the interpreter object, which made it hard for the optimizer to determine independent changes. In the recursive interpreter, non-local returns use exceptions to walk the RPython stack of recursive `interpret` invocations. The SOM loop implementation changed from a primitive manipulating the bytecode index to explicit looping primitives for the `#whileTrue:` and `#whileFalse:` message. These two implement the loops by using RPython's `while` loop, which has the additional benefit that relevant trace merge points can be indicated to RPython using the `jit_merge_point` annotation.

5 Truffle: Self-Optimizing AST Interpreters

Truffle is a Java-based framework to implement efficient AST interpreters [3]. Based on the execution paths taken and the types observed, Truffle dynamically specializes the AST. In combination with a speculative optimizing compiler that uses partial evaluation, these ASTs are then JIT compiled to reach the performance of custom-built VMs [12]. The framework builds on top of a standard JVM and therefore benefits from its services, e. g., garbage collectors, native code optimizers, support for threads, or memory model. In contrast to RPython's approach, the language implementer has to explicitly

provide the AST specializations, and thus, the main mechanism for performance is in the hand of the developer instead of being a black-box provided by a toolchain.

5.1 TruffleSOM: A Self-Optimizing SOM

TruffleSOM is based on the AST interpreter outlined in section 3.1 using the Truffle framework. To reduce the execution overhead, we follow the general guideline for Truffle, i. e., to identify costly operations on the execution's fast path that can be performed optimistically only once as an initialization step. This allows later executions to rely on, e. g., a cached value and a guard that verifies the assumptions. In TruffleSOM, method lookup is one such optimization point, not only because of the avoided overhead of the lookup, but also because it enables later method inlining for context-specific specialization.

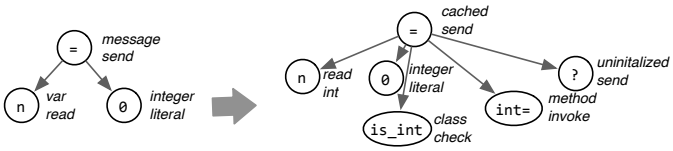

Fig. 2. Specialization of a Message Send

Figure 2 depicts a possible specialization for the messages send `n = 0` of the factorial function from fig. 1. The message send first evaluates its arguments. Thus, it determines the receiver as well as the operand. During the first execution of this AST node, the variable `n` contains an integer and, consequently, the variable read node can be specialized for reading integers. This is beneficial to avoid boxing [3]. The message send node can then be specialized based on the receiver's class, i. e., `Integer` in this case. Thus, the uninitialized message send node is replaced by a cached send node, which keeps the value of the lookup, i. e., the comparison method defined in `Integer`. As a guard, the cached send node has a special class check node as its child. This node is a specialized version of a class check and performs only a Java `instanceof` test on the receiver to determine whether the send can be performed. In the best case, this assumption holds and the cached comparison method can be invoked directly. In case the class check fails, execution is handed over to an uninitialized send node, which performs the method lookup and then specializes itself.

This caching strategy has a wide applicability for instance for the lookups of globals, which are a pair of key and value. The pair can be cached in a specialized node, which allows the optimizer to treat it as a constant. Compared to RPython, this technique is similar to the use of `promote()` and `@elidable`, but requires node specialization instead of being simple annotations.

Another example for specializations necessary to reach peak performance is the handling of non-local returns. In a naive implementation, each method handles non-local returns as follows:

```
1  def CatchNonLocalReturn.execute(self, frame):
2    marker = FrameOnStackMarker()
3    frame.set_object(self._marker_slot, marker)
4    try:
5      return self._method_body.execute(frame)
6    except ReturnException as e:
7      if e.reached_target(marker):
8        return e.result()
9      else:
10       raise e
11   finally:
12     marker.frame_no_longer_on_stack()
```

However, only few methods contain non-local returns, but the setup for handling them has a severe performance impact. Thus, TruffleSOM splits this handling into a `CatchNonLocalReturn` node and generates it only for methods that need it.

5.2 Building Specializations

While the optimization of a Truffle interpreter is left to the language implementer, Truffle's approach to it feels very natural. The tree structure facilitates optimizations such as special handling for Smalltalk loop message sends, or control structures, because it fits better with the semantics of these and other constructs than the linearized bytecode format in RPySOM. Taking the example of `#ifTrue:ifFalse:`, in TruffleSOM it is specialized when the receiver is a boolean and the arguments are blocks. The specialized node then performs a simple identity check on the receiver and evaluates the corresponding block. Compared to the library solution, this reduces the number of method and block activations significantly. In RPySOM, we had to introduce jump bytecodes, change the parser to generate them, and implement them in the interpreter. Thus, the changes were neither as localized nor as straightforward.

Another important part of the framework is TruffleDSL, a set of annotations, that facilitates the specification of node specializations. For instance, in dynamic languages such as Smalltalk, an addition operation has to account for the various possible

combinations of input types to implement the desired semantics. Since Truffle requires that each of these specializations are node objects, the language implementer would need to implement for each case a separated class. However, with TruffleDSL such classes are generated automatically. So that the language implementer only needs to define one method for each specialization and annotate it with the conditions that need to hold to apply the specialization.

6 Performance Evaluation

To assess whether RPySOM and TruffleSOM reach performance of the same order of magnitude as optimizing JIT compiling VMs, we compare them to Oracle's HotSpot JVM and PyPy. We chose DeltaBlue and Richards as two object-oriented benchmarks, which are used to tune JVM's, JavaScript VMs, PyPy, and Smalltalk VMs, as well as Mandelbrot as a numerical benchmark. Note, as with all benchmarks, the results are only indicative, but not predictive for the performance of actual applications.

Table 1 lists the results, indicating the absolute runtime in milliseconds, as well as the standard deviation. Furthermore, it lists the result normalized to Java, which means higher factors are slower. The benchmarks measure stable-state performance. Each benchmark was executed 100 times. The used machine has two quad-core Intel Xeons E5520, 2.26 GHz with 8 GB of memory and runs Ubuntu Linux with kernel 3.11. A full overview of the results and notes on their reproduction are available online.[1]

As a further reference point, we include the SOM++ interpreter, a SOM implemented in C++. It uses bytecodes and applies optimizations such as inline caching, threaded interpretation, and a generational garbage collector. However, it is 70 to 700× slower than Java. RPySOM and TruffleSOM on the other hand, reach more or less the same order of magnitude of performance. RPySOM reaches 1.7 to 10.6×, while TruffleSOM has more remaining optimization potential with being 1.4 to 16× slower. Nonetheless, both implementations manage to reach, depending on the benchmark, the same order of magnitude of performance as Java, without requiring custom VMs and hundreds of person years of engineering. Thus, we conclude, that RPython as well as Truffle live up to the expectations.

1. Appendix Performance Evaluation: http://stefan-marr.de/papers/ieee-soft-marr-et-al-appendix-performance-evaluation

For brevity, we omit a comparison based on microbenchmarks. However, TruffleSOM outperforms RPySOM on two out of three of the larger benchmarks and all but one of the microbenchmarks. Thus, we assume further specializations will allow TruffleSOM to reach the same performance level on the Richards benchmark as well.

7 Consequences for Language Implementations

RPython as well as Truffle overall deliver on their promise of providing a fast VM based on simple interpreter implementations, but with different trade-offs. The initial performance gains provided by RPython's meta-tracing can be more compelling, especially when engineering resources are constraint. However, Truffle's approach might be more beneficial in long-run when the black-box approach of meta-tracing fails and the flexibility of custom specializations is required. Performance aside, the Truffle framework initially guides the interpreter implementation sufficiently to quickly prototype ideas with low engineering effort, and thus, is a suitable platform for experiments as well.

To gain performance, both approaches turn out to be similar. They both rely on complex compiler technology combined either with tracing or partial evaluation. As a consequence, performance benefits significantly in both cases when data dependencies are made explicit, which often leads to a programming style that avoids unnecessary side effects. To further make certain properties explicit for the optimizer, they also provide roughly the same set of annotations.

Furthermore, both approaches steer a language implementer to avoid exposing implementation decisions to the language level. In a dynamic language such as Smalltalk, both approaches encourage to expose the dynamisms in a controlled manner only. For instance, it is easier to provide access to information such as the class of an object via a primitive than to expose it via an object field that is treated differently than other object fields. In both cases, the framework needs to be notified that the class structure has changed to invalidate compiled code.

8 Are we there yet?

The general impression of RPython's meta-tracing approach and Truffle's self-optimizing interpreters is that both have the potential to change the way languages are implemented. They enable language

TABLE 1
Performance Results

	DeltaBlue		Richards		Mandelbrot	
	ms	norm.	ms	norm.	ms	norm.
Java	118 ± 29.0	1.0	192 ± 10.9	1.0	146 ± 0.4	1.0
PyPy	606 ± 9.9	5.1	646 ± 2.6	4.1	217 ± 0.8	1.5
RPySOM	201 ± 0.5	1.7	1689 ± 2.9	10.6	786 ± 1.1	5.4
TruffleSOM	171 ± 45.2	1.4	2557 ± 72.7	16.0	344 ± 3.0	2.3
SOM++	8656 ± 12.8	73.0	87436 ± 170.2	547.4	104033 ± 184.2	710.4

implementations with low effort and good performance characteristics. Especially, in the field of DSL where the language implementations need to be maintainable and as flexible as the domain it requires, the engineering benefits of both approaches over classic compiler and VM implementations are significant. They free the language implementer from low-level concerns such as memory management, native code generation, and other typical VM services, and thus, require overall significantly less engineering than classic approaches.

Interestingly, neither of them provides advanced support for one of the main issues of today: concurrency and parallelism. Truffle allows the use of the JVM's capabilities, and RPython experiments with software transactional memory. Still, neither provides building blocks to language designers that can facilitate the implementation of DSLs for concurrent and parallel programming, even thought, programmers in this field could benefit from highly efficient DSL implementations to tackle the complexity of concurrency.

Furthermore, both approaches could benefit from each other. The initial performance advantage of the meta-tracing approach can be a benefit, especially for DSLs developed for narrow use case. A combination with the self-optimizing approach of Truffle might then provide benefits for long-term project to expose more optimization potential and reach the maximum possible performance, without having to rely solely on the black-box meta tracing.

For the future of programming languages and their implementation, RPython and Truffle both make valuable contributions. Combined with language design and implementation environments such as language workbenches, they can dramatically change the way languages are implemented in the future. Moreover, their optimization potential might reduce the cost of language features such as metaprogramming, which can have a positive impact on programmer productivity. That way, they could also widen the set of acceptable programming techniques in performance sensitive fields.

Acknowledgments

The authors would like to thank C. F. Bolz from the PyPy team as well as G. Duboscq, C. Humer, M. Haupt, C. Seaton, C. Wimmer, A. Wöß, T. Würthinger, and W. Zhang from the Truffle community for guidance and support. Their help was essential for enabling the SOM implementations to perform efficiently in the first place.

References

[1] J. Aycock, "A Brief History of Just-In-Time," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 97–113, Jun. 2003.

[2] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo, "Tracing the Meta-level: PyPy's Tracing JIT Compiler," in *Proc. of ICOOOLPS*. ACM, 2009, pp. 18–25.

[3] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer, "Self-Optimizing AST Interpreters," in *Proc. of DLS*, 2012, pp. 73–82.

[4] M. Haupt, R. Hirschfeld, T. Pape, G. Gabrysiak, S. Marr, A. Bergmann, A. Heise, M. Kleine, and R. Krahn, "The SOM Family: Virtual Machines for Teaching and Research," in *Proceedings of the 15th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM Press, June 2010, pp. 18–22.

[5] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[6] T. D'Hondt, "Are Bytecodes an Atavism?" in *Self-Sustaining Systems*, ser. LNCS, R. Hirschfeld and K. Rose, Eds. Springer, 2008, vol. 5146, pp. 140–155.

[7] J. R. Bell, "Threaded code," *Commun. ACM*, vol. 16, no. 6, pp. 370–372, Jun. 1973.

[8] K. Casey, M. A. Ertl, and D. Gregg, "Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 6, p. 37, 2007.

[9] S. Brunthaler, "Efficient Interpretation Using Quickening," in *Proceedings of the 6th Symposium on Dynamic Languages*, ser. DLS, no. 12. ACM, Oct. 2010, pp. 1–14.

[10] A. Gal, C. W. Probst, and M. Franz, "HotpathVM: An Effective JIT Compiler for Resource-constrained Devices," in *Proc. of VEE*. ACM, 2006, pp. 144–153.

[11] C. F. Bolz and L. Tratt, "The Impact of Meta-Tracing on VM Design and Implementation," *Science of Computer Programming*, 2013.

[12] 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 *Proc. of Onward!* ACM, 2013, pp. 187–204.