Tracing vs. Partial Evaluation

Comparing Meta-Compilation Approaches
for Self-Optimizing Interpreters

Stefan Marr
INRIA, Lille, France
mail@stefan-marr.de

Stéphane Ducasse
INRIA, Lille, France
stephane.ducasse@inria.fr

Abstract

Tracing and partial evaluation have been proposed as meta-compilation techniques for interpreters to make just-in-time compilation language-independent. They promise that programs executing on simple interpreters can reach performance of the same order of magnitude as if they would be executed on state-of-the-art virtual machines with highly optimizing just-in-time compilers built for a specific language. Tracing and partial evaluation approach this meta-compilation from two ends of a spectrum, resulting in different sets of tradeoffs.

This study investigates both approaches in the context of self-optimizing interpreters, a technique for building fast abstract-syntax-tree interpreters. Based on RPython for tracing and Truffle for partial evaluation, we assess the two approaches by comparing the impact of various optimizations on the performance of an interpreter for SOM, an object-oriented dynamically-typed language. The goal is to determine whether either approach yields clear performance or engineering benefits. We find that tracing and partial evaluation both reach roughly the same level of performance. SOM based on meta-tracing is on average 3x slower than Java, while SOM based on partial evaluation is on average 2.3x slower than Java. With respect to the engineering, tracing has however significant benefits, because it requires language implementers to apply fewer optimizations to reach the same level of performance.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers, Interpreters, Optimizations

General Terms Languages, Performance

Keywords language implementation, just-in-time compilation, meta-tracing, partial evaluation, comparison, case study, self-optimizing interpreters

1. Introduction

Interpretation is one of the simplest approaches to language implementation. However, interpreters lost some of their appeal because highly optimizing virtual machines (VMs) such as the Java Virtual Machine (JVM) or Common Language Runtime deliver performance that is multiple orders of magnitude better. Nevertheless, interpreters stand out for their simplicity, maintainability, and portability.

The development effort for highly optimizing static ahead-of-time or dynamic just-in-time compilers makes it often infeasible to build more than a simple interpreter. A recent example is JavaScript. In the last decade, its performance was improved by several orders of magnitude, but it required major industrial investments. Unfortunately, such investments are rarely justified, especially for research projects or domain-specific languages (DSLs) with narrow use cases.

In recent years, tracing and partial evaluation became suitable meta-compilation techniques that alleviate the problem. RPython [5, 6] and Truffle [27, 28] are platforms for implementing (dynamic) languages based on simple interpreters that can reach the performance of state-of-the-art VMs. RPython uses trace-based just-in-time (JIT) compilation [2, 14], while Truffle uses partial evaluation [12] to guide the JIT compilation.

The PyPy[1] and Truffle/JS[2] projects show that general purpose languages can be implemented with good performance. For instance Truffle/JS reaches the performance of V8 and SpiderMonkey on a set of selected benchmarks.[3] However,

[1] *PyPy*, a fast Python, access date: 2014-12-18 http://pypy.org/

[2] *Truffle/JS, a JavaScript for the JVM*, Oracle Labs, access date: 2014-12-18 http://www.oracle.com/technetwork/oracle-labs/program-languages/javascript/index.html

[3] *Performance: JavaScript, Slide 86*, Graal Tutorial, Christian Wimmer, CGO, 2015, access date: 2015-07-19 http://lafo.ssw.uni-linz.ac.at/papers/2015_CGO_Graal.pdf

for language implementers and implementation technology researchers, it remains the question of what the concrete tradeoffs between the two meta-compilation approaches are. When considering possible use cases and varying maturity of language designs, the available engineering resources and the desired performance properties require different tradeoffs. For instance for a language researcher, it is most important to be able to experiment and change a language's semantics. For the implementation of a standardized language however, the focus is typically on performance, and thus the best mechanisms to realize optimizations are required. For implementation research, a good understanding of the tradeoffs between both meta-compilation approaches might lead to further improvements that simplify language implementation for either of the scenarios.

In this study, we compare tracing and partial evaluation as meta-compilation techniques for self-optimizing interpreters to determine whether either of the two has clear advantages with respect to performance or engineering properties. To characterize the tradeoffs between the two, we investigate the impact of a set of interpreter optimizations. This allows us to determine whether an optimization is necessary depending on the approach. We use RPython and Truffle as concrete representations of these two approaches. To compare them in a meaningful way, we implement SOM [15], a dynamic object-oriented language with closures, as identical as possible on top of both. Section 3 details the practical constraints and the requirements for a conclusive comparison. The contributions of this paper are:[4]

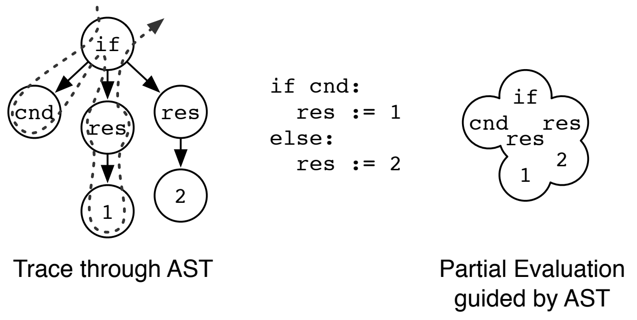
- a comparison of tracing and partial evaluation as meta-compilation techniques for self-optimizing interpreters.
- an assessment of the performance impact and implementation size of optimizations in self-optimizing interpreters.
- a performance assessment of RPython and Truffle with respect to interpreter performance, peak performance, whole-program behavior, and memory utilization.

We find that neither of the two approaches has a fundamental advantage for the reached peak-performance. However, meta-tracing has significant benefits from the engineering perspective. With tracing, the optimizer uses directly observed runtime information. In the case of partial evaluation on the other hand, it is up to the language implementer to capture much of the same information and expose it to the optimizer based on specializations.

2. Background

This section gives a brief overview of meta-tracing, partial evaluation, and self-optimizing interpreters as background for the remainder of this paper.

[4] Artifacts: http://stefan-marr.de/papers/oopsla-marr-ducasse-meta-tracing-vs-partial-evaluation-artifacts/



```
if cnd:
  res := 1
else:
  res := 2
```

Figure 1. Selecting JIT Compilation Units for AST Interpreters. To select a compilation unit, meta-tracing (left) records the operations performed by the interpreter for the execution of one specific path through a program. Partial evaluation (right) uses the AST structure to determine which interpreter-level code to include in a compilation unit.

2.1 Meta-Tracing and Partial Evaluation

While interpreters are a convenient and simple implementation technique, they are inherently slow. Hence, researchers tried to find ways to generate efficient native code from them without having to build custom JIT compilers. With the appearance of trace-based JIT compilation [14], trace-based meta-compilation, i. e., *meta-tracing* was the first practical solution for general interpreters [5, 6] that also works for dynamic languages such as JavaScript, Python, or Ruby. The main idea is to trace the execution of the interpreter instead of tracing the concrete program it executes, and thus, make the JIT compiler a reusable *meta-compiler* that can be used for different language implementations. The resulting traces are the units of compilation in such a system. Based on frequently executed loops on the application level, the interpreter records a concrete path through the program, which then can be heavily optimized and compiled to native code. Since traces span across many interpreter operations (cf. fig. 1), the interpreter overhead can be eliminated completely and only the relevant operations of the application remain.

Partial evaluation [12] of interpreters has been discussed as a potential meta-compilation technique for interpreters as well [1, 7, 24, 25]. However, only very recently, Würthinger et al. [28] were able to show that it is a practical meta-compilation technique for abstract-syntax-tree-based (AST) interpreters for dynamic languages. Instead of selecting the compilation unit by tracing, the unit is determined by using a program's AST to guide a partial evaluator. The evaluator resolves all parts of the program that do not depend on unknown runtime information. With the knowledge of the AST and values embedded in it, the evaluator can resolve otherwise highly polymorphic method calls, perform aggressive constant propagation, and inlining. Thereby it identifies the relevant elements of the interpreter implementation (cf. fig. 1), which need to be included in a compilation unit.

In contrast to tracing, partial evaluation preserves the control flow of the interpreter and the user program that cannot be resolved statically. Since interpreters needs to handle every case of a language, the resulting control flow is generally too complex for partial evaluation and compiler optimizations to generate efficient native code. However, combined with the idea of self-optimizing interpreters, partial evaluation became finally practical for a wide range of languages.

2.2 Self-Optimizing Interpreters

The main idea of a self-optimizing interpreter is that an executing AST rewrites itself at runtime, e. g., based on observed types and values [27]. Typical optimizations speculate for instance that observed types do not change in the future. In case of an addition operation, a generic node that handles all possible types can be replaced by one that is specialized for integers. With such optimizations, an AST can specialize itself for exactly the way the program uses the language. This is beneficial for the interpreter, because it can avoid unnecessarily generic runtime operations, and at the same time the control flow is simplified, which leads to better compilation results when partial-evaluation-based meta-compilation is used [28]. In case of the addition operation, the type-based specialization avoids generic checks at runtime as well as boxing of primitive integer values to reduce overhead and complexity of the operations.

Self-optimizations can also have other benefits. One common technique is to cache runtime values for later use. It enables for instance polymorphic inline caches for method lookups [19]. Starting out from a generic AST, the first execution of a method invocation node does the normal lookup and then rewrites itself to a simpler node that caches the lookup result and associates it with a predicate that confirms whether the cached value is valid in subsequent invocations. Thus, instead of having to include the complex lookup logic, the node only performs a check, and if it succeeds, the actual method invocation.

3. Study Setup and Practical Constraints

The goal of this study is to compare tracing and partial evaluation as meta-compilation techniques with respect to the achievable performance as well as the required engineering effort for interpreters. This section discusses how these two techniques can be compared based on concrete existing systems. It further discusses the design for the experimental setup, the concrete experiments, and the implications for the generalizability of the results. It also provides the required background on the SOM language, for which we implement interpreters for this study.

3.1 How to Compare Tracing and Partial Evaluation?

As discussed above, partial evaluation for dynamic languages has only recently been shown to be practical and so far only in the context of self-optimizing interpreters. Meta-tracing has been successfully applied to AST interpreters as well [5], thus, we compare both approaches based on self-optimizing AST interpreters.

To the best of our knowledge RPython[5] is the only meta-tracing toolchain for interpreters. Similarly, Truffle[6] is the only framework with partial-evaluation-based meta-compilation for interpreters. Thus, we chose these two systems for this experiment.

The goal of this study is to assess the conceptual as well as the practical difference of tracing and partial evaluation. Hence, it stands to question what the generalizable insights of an empirical comparison are. From our perspective, both systems reached sufficient maturity and sophistication to represent the state of the art in tracing as well as partial evaluation technology. Furthermore, RPython with PyPy and Truffle with Truffle/JS implement complex widely used languages with the goal to optimize the peak performance as much as possible, and indeed reach the performance levels of dedicated JIT compiling VMs. Thus, we expect a performance comparison to reflect the general capabilities of the two approaches. However, both systems implement different sets of optimizations, and have different approaches for generating native code. Therefore, minor performance difference between both systems are expected and will not allow for conclusions with respect to the general approaches. Nonetheless, we think the general order of magnitude is representative for both approaches.

In order to compare both approaches fairly, we need a language implementation based on RPython as well as Truffle. With PyPy and ZipPy [26], there exist Python implementations for both systems. However, PyPy is a bytecode-interpreter and ZipPy a self-optimizing interpreter. Thus, a comparison would not only compare tracing with partial evaluation, but also include bytecode vs. ASTs, which would make a study inconclusive with respect to our question. The situation is the same for the Ruby implementations JRuby+Truffle[7] and Topaz. Moreover, they all differ in many other aspects, e. g., differences in the implemented optimizations, which makes a comparison generally inconclusive. Hence, for a fair comparison we need language implementations for both systems that are as identical as possible, and enables us to compare tracing and partial evaluation instead of other aspects. For this study we use SOM, which is discussed in section 3.3.

[5] *RPython Documentation*, The PyPy Project, access date: 2015-03-18 `http://rpython.readthedocs.org/`

[6] *The Truffle Language Implementation Framework*, SSW JKU Linz, access date: 2015-03-18 `http://www.ssw.uni-linz.ac.at/Research/Projects/JVM/Truffle.html`

[7] *JRuby+Truffle - a High-Performance Truffle Backend for JRuby*, JRuby Project, access date: 2015-03-18 `https://github.com/jruby/jruby/wiki/Truffle`

3.2 RPython and Truffle

In the previous section, we discussed meta-tracing and partial evaluation from the conceptual perspective only. Since this study compares the two approaches empirically, this section provides a few technical details on RPython and Truffle and discusses the theoretical differences between the two meta-compilation approaches.

RPython is a toolchain for language implementation that uses meta-tracing. It is also a restricted subset of Python that uses type inference and code transformations to add low-level services such as memory management and JIT compilation to interpreters to generate complete VMs. RPython's meta-tracing has been shown to work well for a wide range of different languages including Pyrolog (Prolog), Pycket (Racket), and Topaz (Ruby), of which some are bytecode interpreters, e. g., PyPy and Topaz, and others are AST interpreters, e. g., Pyrolog and Pycket.

With a set of annotations, language implementers can communicate high-level knowledge about the implemented language to the toolchain. Since trace-based compilation works best on loops, one of the main annotation is the so-called *trace merge point*, which indicates potential starting points for traces and defines how to recognize application-level loops. Other language-specific properties, for instance about mostly-constant values such as method lookup results can be communicated similarly. For instance, functions can have side-effects that are not essential for the execution, e. g., for caching the result of method lookups. With RPython's `@elidable` annotation, the optimizer can be told that it is safe to elide repeated executions within the context of a trace. Another example are values that are runtime constants. Those can be explicitly *promoted* to enable the compiler to optimize based on them. In general, these annotations are useful in cases where an optimizer alone needs to make conservative assumptions, but the specific language usage patterns allow for additional optimizations, which can be used to generate specialized native code. A more detailed discussion of RPython is provided by Bolz and Tratt [5].

RPython's Meta-Tracing As mentioned earlier, RPython traces the execution of an interpreter instead of tracing the program the interpreter is executing. The resulting trace is the compilation unit on which the optimizer works to produce efficient native code that can be executed instead of the slow interpreter.

The tracing process is started based on *trace merge points* in the interpreter. It is triggered when a merge point has been visited with the same interpreter state for a predefined number of times. During tracing, the interpreter continues executing as usual but also records each of the RPython-level operations it performs. As for regular tracing, control-flow operations are not directly recorded. Instead, for conditional branches, the observed result of the conditional expression is recorded as a *guard* in the trace. Afterwards, the interpreter continues in the corresponding branch. Similarly, for dynamic dispatches, the actual call is not recorded. To ensure that the trace is only used when the dispatch goes to the same function, a guard is recorded, e. g., to check that the function object is the expected one.

Generally, the tracer records all operations the interpreter performance, but does not consider the concrete values. However, as discussed earlier, it can be desirable to do so based on `@elidable` and `promote()`. In case of a lookup for instance, the results are likely constant and a repeated lookup can be avoided in compiled code.

Once the tracer reached again the merge point, the trace, i. e., the resulting compilation unit is completed and can be optimized and compiled to native code. Note that this means that the compilation unit is determined strictly during interpretation and contains concrete values observed during a single execution. Furthermore, it is a completely linear list of instructions and does not contain control flow. This simplifies optimization significantly. On the other hand, all change in control-flow conditions and dynamic-dispatch targets lead to guard failures. If a guard fails, execution returns to the interpreter, or if the guard failed repeatedly can start tracing of a side-trace. Thus, the approach assumes that control flow is relatively stable, which seems to be the case in interpreters since the control flow is governed by the user program.

Truffle is Würthinger et al.'s Java framework for self-optimizing interpreters and uses partial evaluation as meta-compilation technique. It integrates with the Graal JIT compiler for the partial evaluation of ASTs and the subsequent native code generation. Truffle in combination with Graal is built on top of the HotSpot JVM, and thus, guest languages benefit from the garbage collectors, memory model, thread support, as well as the general Java ecosystem.

For language implementers, Truffle has an annotation-based DSL [17], which avoids much of the boilerplate code for self-optimizations. For instance, the DSL provides simple means to build specialized nodes for different argument types of operations. Instead of manually defining various node classes, with the DSL only the actual operations need to be defined. The corresponding node classes as well as the node rewriting and argument checking logic are generated.

In addition to the DSL, there are other differences to RPython. For instance, runtime constants, and more generally any form of *profiling* information, are exposed by providing node specializations instead of using a `promote`-like operation. Thus, the value is cached in the AST instead of relying on a trace context as RPython does. Another difference is that Truffle relies on explicit indications to determine the boundaries of compilation units. While RPython relies mostly on tracing, Truffle uses the `@TruffleBoundary` annotation to indicate that methods should not be included in the compilation unit. This is necessary, because Truffle's partial evaluation *greedily* inlines all possible control-flow paths, which would lead to too large compilation units without these explicit cutoffs. In practice, boundaries are placed on complex operations that are not on the fast path, e. g., lookup operations and complex library functionality such as string or hashtable operations. Also related is Truffle's `transferToInterpreter` operation, which results in a deoptimization point [16] in the native code. This excludes the code of that branch from compilation and can avoid the generation of excessive amounts of native code and enable optimizations, because the constraints of that branch do not have to be considered.

Truffle's Partial Evaluation In contrast to RPython's meta-tracing, Truffle's partial evaluation works on a method level. Similar to classic JIT compilers, the method invocation count is used as a heuristic to start compilation. When a certain threshold is reached, the AST root node of such a method is given to the partial evaluator, which then starts processing the `execute()` method of that node. Based on the actual AST and all constants referenced by the code, the Java code is partially evaluated. In a classic JIT compiler without such partial evaluation, the highly polymorphic calls to the `execute()` methods of subexpressions are problematic, but with the knowledge of the AST and the concrete code corresponding to the `execute()` methods for its nodes, aggressive inlining can be performed to construct a compilation unit that contains all of the interpreter's behavior for a user-level method. Furthermore, the use of inline caches on the AST level exposes inlining opportunities on the user-language level, which further increases the opportunity for optimization. As mentioned earlier, this greedy inlining can be controlled by placing `@TruffleBoundary` annotations and calls to `transferToInterpreter()` to avoid code explosion.

Compared to RPython's meta-tracing, this approach has two fundamental differences. On the one hand, a compilation unit contains the complete control flow that cannot be resolved by compiler optimizations. Thus, the approach has the known tradeoffs between method-based and trace-based compilation. On the other hand, the compilation units are determine strictly independent of a concrete execution. This means, a language implementer needs to accumulate profiling information to guide optimistic optimizations, whereas tracing considers one set of concrete values gathered during the tracing. We discuss the impact of this based on our experiments in section 5.2.

From a conceptual perspective, both approaches are instances of the first Futamura projection [13], i. e., they specialize an interpreter based on a given source program to an executable. However, while partial evaluation is restricted by the knowledge at compilation time, tracing deliberately chooses which knowledge to use to avoid over-specializing code, which would then only work for a subset of inputs.

3.3 The Case Study: SOM (Simple Object Machine)

As discussed in section 3.1, for a meaningful comparison of the meta-compilation approaches, we need close to identical language implementations on top of RPython and Truffle. We chose to implement the SOM language as case study. It is an object-oriented class-based language [15] designed for teaching. Therefore, it is kept simple and includes only fundamental language concepts such as *objects*, *classes*, *closures*, and *non-local returns*. With these concepts, SOM represents a wide range of dynamic languages. Its implementation solves the same performance challenges more complex languages face, for instance for implementing exceptions, specializing object layouts, and avoiding the overhead for dynamic method invocation semantics, to name but a few.

While its size makes it a good candidate for this study, its low complexity raises the question of how generalizable the results of this study are for other languages. From our perspective, SOM represents the core concepts and thus solves many of the challenges common to more complex languages. What we do not investigate here is however the *scalability* of the meta-compilation approaches to more complex languages. Arguably, projects such as PyPy, Pycket, Topaz, JRuby+Truffle, and Truffle/JS demonstrate this scalability already. Furthermore, even though SOM is simple, it is a complete language. It supports classic object-oriented VM benchmarks such as DeltaBlue, Richards, and numeric ones such as Mandelbrot set computation and n-body simulations. The benchmark set further includes a JSON parser, a page rank algorithm, and a graph search to cover a wide range of use cases server programs might face.

Implementation Differences of SOM_{MT} and SOM_{PE}. Subsequently, we refer to the two SOM implementations as SOM_{MT} for the version with RPython's meta-tracing, and SOM_{PE} for one with Truffle's partial evaluation. SOM_{PE} builds on the Truffle framework with its TruffleDSL [17]. SOM_{MT} however is built with ad hoc techniques to realize a self-optimizing interpreter, which are kept as comparable to SOM_{PE} as possible. Generally, the structure of the AST is the same for both interpreters. Language functionality such as method invocation, field access, or iteration constructs are represented in the same way as AST nodes.

Some aspects of the interpreters are different however. SOM_{PE} uses the TruffleDSL to implement basic operations such as arithmetics and comparisons. TruffleDSL significantly simplifies self-optimization based on types observed at runtime and ensures that arithmetic operations work directly on Java's primitive types `long` and `double` without requiring boxing. Boxing means that primitive values are stored in specifically allocated objects. With Java's unboxed versions of primitive types, we avoid the additional allocation for the object and the pointer indirection when operating on the values.

SOM_{MT} on RPython relies however on uniform boxing of all primitive values as objects. With the absence of TruffleDSL for RPython, the minimal boxing approach used in SOM_{PE} was not practical because the RPython type system requires a common root type but does not support Java's implicit boxing of primitive types. Since tracing compilation eliminates the boxing within a compilation unit, it makes only a difference in the interpreted execution. Since Truffle, and therefore SOM_{PE} uses a method calling convention based on `Object` arrays, boxing is not eliminated completely either. Thus, we consider this difference acceptable (cf. sections 4.3 and 4.4).

3.4 Assessing the Impact of the Meta-Compilation Strategies

To assess the benefits and drawbacks of meta-tracing and partial evaluation from the perspective of language implementers, we determine the impact of a number of interpreter optimizations on interpretation and peak performance. Furthermore, we assess the implementation sizes to gain an intuition of how the required engineering effort compares for both approaches.

Optimizations. To use a representative set of optimizations, we identify tree main categories. *Structural optimizations* are applied based on information that can be derived at parse time. *Dynamic optimizations* require runtime knowledge to specialize execution based on observed values or types. *Lowerings* reimplement performance critical standard library functionality in the interpreter. These three groups cover a wide range of possible optimizations. For each category, we pick representative optimizations. They are listed in table 1 and detailed in appendix A.

Performance Evaluation. For the performance evaluation, we consider the pure interpreted performance and the compiled peak performance. Both aspects can be important. While interpreter speed can be negligible for long-running server applications, it is critical for short-lived programs such as shell scripts. We assess the impact of the optimizations for both modes to also determine whether they are equally beneficial for interpretation and peak performance, or whether they might have a negative effect on one of them.

Implementation Size of Optimizations. To gain some *indication* for potential differences in engineering effort, we assess the implementation size of the applied optimizations. However, this is not a systematic study of the engineering effort. On the one hand RPython and Java are two very different languages making a proper comparison hard, and on the other hand, implementation size is only a weak predictor for effort. Nonetheless, implementation size gives an intuition and enables us to position the two approaches also with respect to the size of other language implementation projects. For instance in a research setting, an interpreter prototype might be implemented in 2.5K lines of code (LOC). A maturing interpreter might be 10 KLOC in size, but a state-of-the-art VM is usually larger than 100 KLOC.

Structural Optimizations	
opt. local vars	distinguish variable accesses in local and non-local scopes
catch-return nodes	handle non-local returns only in methods including them
min. escaping vars	expose variables in scope only if accessed (SOM_{MT} only)
min. escaping closures	avoid letting unused lexical scopes escape
Dynamic Optimizations	
cache globals	cache lookup of global values
inline caching	cache method lookups and block invocations
typed vars	type-specialize variable accesses (SOM_{PE} only)
typed args	type-specialize argument accesses (SOM_{PE} only)
typed fields	specialize object field access and object layout
array strategies	type-specialize array storage
inline basic ops.	specialize basic operations (SOM_{PE} only)
Lowerings	
lower control structures	lower control structures from library into interpreter
lower common ops	lower common operations from library into interpreter

Table 1. The set of optimizations applied to the SOM_{MT} and SOM_{PE} interpreters (cf. appendix A).

4. Comparing Tracing and Partial Evaluation

Before discussing the results of the comparisons, we detail the methodology used to obtain and assess the performance and give a brief characterization of the used benchmarks.

4.1 Methodology

With the non-determinism in modern systems, JIT compilation, and garbage collection, we need to account for the influence of variables outside of our control. Thus, we execute each benchmark at least 500 times within the same VM instance. This guarantees that we have at least 100 continuous measurements for assessing steady state performance. The steady state is determined informally by examining plots of the measurements for each benchmark to confirm that the last 100 measurements do not show signs of compilation.

The benchmarks are executed on a system with two quad-core Intel Xeons E5520 processors at 2.26 GHz with 8 GB of memory and runs Ubuntu Linux with kernel 3.11, PyPy 2.4-dev, and Java 1.8.0_11 with HotSpot 25.11-b03.

Measurement Setup. Pure interpretation performance for SOM_{MT} is measured with executables without meta-tracing

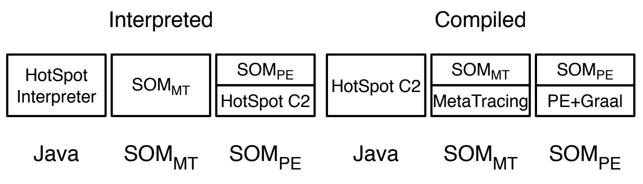


Figure 2. Experimental setup for interpreted as well as compiled, i. e., peak performance measurements.

support. Similarly, we measure the pure interpretation performance of SOM$_{PE}$ on Hotspot without the partial evaluation and compilation support of Truffle. Thus, in both cases, there is no additional overhead, e. g., for compiler related bookkeeping. However, SOM$_{PE}$ still benefits from the HotSpot's normal Java JIT compilation, while SOM$_{MT}$ is a simple interpreter executing directly without any underlying JIT compilation. We chose this setup to avoid measuring overhead from the meta-JIT compiler infrastructure and focus on the interpreter-related optimizations. Since we report results after warmup, the results for SOM$_{PE}$ and SOM$_{MT}$ represent the ideal interpreter performance in both cases.

Figure 2 depicts the setup for the measurements including only the elements that are relevant for the interpreter or peak performance.

For measuring the peak performance, we enable meta-compilation in both cases. Thus, execution starts first in the interpreter, and after completing a warmup phase, the benchmarks execute solely in optimized native code. To assess the capability of the used meta-compilation approach, we report only the measurements after warmup is completed, i. e., ideal peak performance. For this experiment, Truffle is configured to avoid parallel compilation to be more comparable with RPython, which does not have any parallel execution. Furthermore, for peak performance measurements, SOM$_{PE}$ uses a minimum heap size of 2GB to reduce noise from the GC. Still, measurement errors for SOM$_{PE}$ are generally higher than for SOM$_{MT}$, because the JVM performs various operations in parallel and the operating system can reschedule the benchmark thread on other cores. RPython's runtime system on the other hand is completely sequential and is therefore less exposed to rescheduling, which leads to lower measurement errors.

For measuring whole program and warmup behavior in section 4.5, the VMs use their standard unchanged garbage collection settings and Truffle uses parallel compilation. We chose to rely for the experiments on the standard settings to reflect the experience a normal user would have, assuming that the parameters are tuned for a wide range of applications. We use the same settings for determining the memory usage in section 4.6.

Benchmark Suite. The used benchmarks cover various aspects of VMs. DeltaBlue and Richards test among other things how well polymorphic method invocations are optimized. Json is a parser benchmark measuring string operations and object creation. PageRank and GraphSearch traverse large data structures of objects and arrays. Mandelbrot and n-body are classic numerical ones focusing on floating point performance. Fannkuch, n-queens, sieve of Eratosthenes, array permutations, bubble sort, and quick sort measure array access and logical operations. The storage benchmark is a stress test for garbage collectors. A few microbenchmarks test the performance, e. g., of loops, field access, and integer addition. While these benchmarks are comparably small and cannot compete with application benchmark suites such as DaCapo [4], they test a relevant range of features and indicate the order of magnitude the discussed optimizations have on interpretation and peak performance.

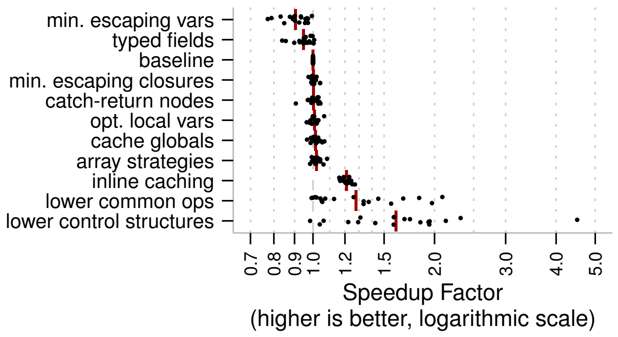


Figure 3. Impact of optimizations on SOM$_{MT}$'s interpreter performance. Experiments are ordered by geometric mean of the speedup over all benchmarks, compared to the baseline. Each dot represents a benchmark. The red vertical bar indicates the geometric mean. The results show that the optimization for minimizing escaping variables slows the interpreter down. Inline caching and lowering of library functionality give substantial benefits.

Assessing Optimization Impact. As in classic compilers, optimizations interact with each other, and varying the order in which they are applied can have significant implications on the observed gains they provide. To minimize the impact of these interdependencies, we assess the optimizations by comparing against a *baseline* that includes all optimizations. Thus, the obtained results indicate the gain of a specific optimization for the scenario where all the other optimizations have been applied already. While this might lead to underestimating the value of an optimization for gradually improving the performance of a system, we think it reflects more accurately the expected gains in optimized systems.

4.2 Impact on Interpreter

Before assessing the impact of the meta-compilation approach, we discuss the optimization's impact on interpretation performance.

Figure 3 depicts for each of the optimizations the benchmark results as separate points representing the average speedup over the baseline version of SOM$_{MT}$. All dots on the right of the 1-line indicate speedup, while all dots left of the line indicate slowdowns. Furthermore, the optimizations are ordered by the geometric mean over all benchmarks, which is indicated for each optimization with a red bar. Based on this ordering, all optimizations listed above the baseline cause on average a slowdown, while all optimizations listed below the baseline result in a speedup. Note, the x-axis uses a logarithmic scale.

The optimization for minimizing escaping of variables causes on average a slowdown of 9.6%. This is not surprising, since the interpreter has to allocate additional data structures for each method call and the optimization can only benefit the JIT compiler. Similarly, typed fields cause a slowdown of 5.3%. Since SOM_{MT} uses uniform boxing, the interpreter creates the object after reading from a field, and thus, the optimization is not beneficial. Instead, the added complexity of the type-specialization nodes causes a slowdown. The optimizations to separate catch-return nodes (0.2%), minimizing escaping of closures (0.2%), and the extra nodes for accessing local variables (0.8%) do not make a significant difference for the interpreter's performance. The dynamic optimizations for caching the association object of globals (1.4%) and array strategies (2%) do not provide a significant improvement either.

The remaining optimizations more clearly improve the interpreter performance of SOM_{MT}. The largest gains for interpreter performance come from the lowering of control structures. Here we see an average gain of 1.6x (min. -1.6%, max. 4.5x). This is expected because their implementation in the standard library rely on polymorphic method invocations and the loop implementations all map onto the basic `while` loop in the interpreter. Especially for `for`-loops, the runtime overhead is much smaller when they are implemented directly in the interpreter because it avoids multiple method invocations and the counting is done in RPython instead of requiring language-level operations. Inline caching for methods and blocks (21%) gives also significant speedup based on runtime feedback.

For SOM_{PE}, fig. 4 shows that the complexity introduced for the type-related specializations leads to overhead during interpretation. The typed arguments optimization makes the interpreter on average 18.3% slower. For typed variables, we see 8.9% overhead. Thus, if only interpreter speed is relevant, these optimizations are better left out. For typed object fields, the picture is less clear. On average, they cause a slowdown of 4.1%, but range from 16% slowdown to 4.5% speedup. The effect for SOM_{PE} is more positive than for SOM_{MT} because of the differences in boxing, but overall the optimization is not beneficial for interpreted execution.

Caching of globals (0.4%), optimizing access to local variables (3%), and inline caching (4.6%) give only minimal average speedups for the interpreter. The low gains from inline caching are somewhat surprising. However, SOM_{MT} did not inline basic operations as SOM_{PE} does. Thus, we assume that inlining of basic operations, which gives in itself a major speedup of 1.9x, hides the gains that inline caching of blocks and methods gives on an interpreter without it.

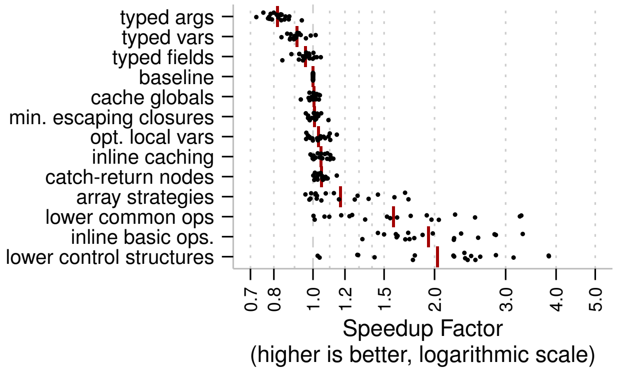


Figure 4. SOM_{PE} optimization impact on interpreter performance. Type-based specialization introduce overhead. Lowering of library functionality and direct inlining of basic operations on the AST-level are highly beneficial.

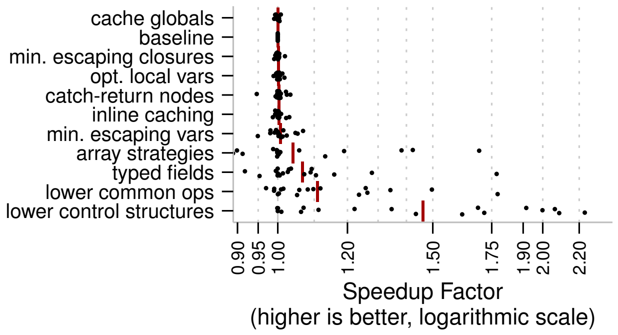


Figure 5. SOM_{MT} optimization impact on peak performance. Most optimizations do not affect average performance. Only lowering of library functionality gives substantial performance gains.

Array strategies give a speedup of 17.6% (min. -4.2%, max. 72.4%) and is with the different boxing strategy of SOM_{PE} more beneficial for the interpreter. Similar to SOM_{MT}, lowering library functionality to the interpreter level gives large improvements. Lowering common operations gives an average speedup of 1.6x and lowering control structures gives 2.1x, confirming the usefulness of these optimizations for interpreters in general.

4.3 Peak Performance

While some of the studied optimizations improve interpreted performance significantly, others cause slowdowns. However, especially the ones causing slowdowns are meant to improve peak performance for the meta-compilation with tracing or partial evaluation.

Meta-Tracing. Figure 5 shows the results for SOM_{MT} with meta-tracing enabled. The first noticeable result is that 6 out of 10 optimizations have barely any effect on the optimized peak performance. The optimizations to cache globals (0%), minimize escaping closures (0.1%), optimize local variable access (0.2%), the separate nodes to catch returns (0.2%), inline caching (0.2%), and minimize escaping variables (0.7%) affect average performance only minimally.

For the optimization of local variable access and inline caching, this result is expected. The trace optimizer eliminate tests on compile-time constants and other unnecessary operations. Furthermore, inline caching is only useful for the interpreter, because SOM_{MT} uses RPython's `@elidable` (cf. section 3.2) to enable method lookup optimization. The lookup is marked as `@elidable` so that the optimizer knows its results can be considered runtime constants to avoid lookup overhead.

The optimization to minimize escaping of variables shows variability from a 5.1% slowdown to a to 6.8% speedup. Thus, there is some observable benefit, but overall it is not worth the added complexity, especially since the interpreter performance is significantly reduced.

Array strategies gives an average speedup of 4.7% (min. −29.9%, max. 69.3%). The additional complexity can have a negative impact, but also gives a significant speedup on benchmarks that use integer arrays, e. g., bubble and quick sort. For typed fields, the results are similar with an average speedup of 7% (min. −8.2%, max. 77.3%). For benchmarks that use object fields for integers and doubles, we see speedups, while others show small slowdowns from the added complexity.

The lowering of library functionality is not only beneficial for the interpreter but also for meta-tracing. For common operations, we see a speedup of 11.5% (min. −21.6%, max. 1.8x). The lowering provides two main benefits. On the one hand, the intended functionality is expressed more directly in the recorded trace. For instance for simple comparisons this can make a significant difference, because instead of building, e. g., a *larger or equal* comparison with *smaller than* and negation, the direct comparison can be used. When layering abstractions on top of each other, these effects accumulate, especially since trace guards might prevent further optimizations. On the other hand, lowering typically reduce the number of operations that are in a trace and thus need to be optimized. Since RPython uses trace length as a criterion for compilation, lowering functionality from the library into the interpreter can increase the size of user programs that are acceptable for compilation.

For the lowering of control structures, we see a speedup of 1.5x (min. −0.1%, max. 4.1x). These speedups are based on the effects for common operations, but also on the additional trace merge points introduced for loop constructs. With these merge points, we communicate directly to RPython where user-level loops are and thereby provide more precise information for compilation.

Generally, we can conclude that only few optimizations have a significant positive impact when meta-tracing is used.

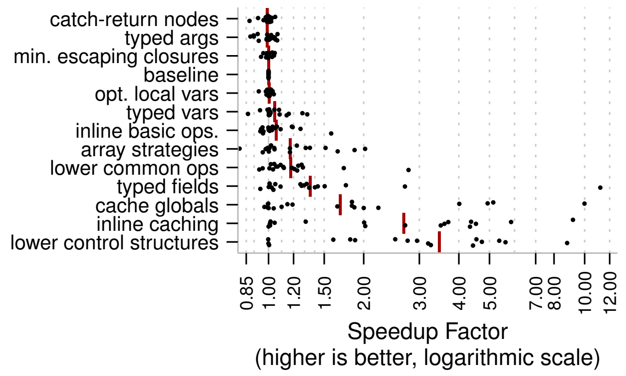


Figure 6. SOM_{PE} optimization impact on peak performance. Overall, the impact of optimizations in case of partial evaluation is larger. Lowering of control structures and inline caching are the most beneficial optimizations.

Specifically, the lowering of library functionality into the interpreter helps to expose more details about the execution semantics, which enables better optimizations. The typing of fields and array strategies are useful, but highly specific to the language usage.

Partial Evaluation. The first observation based on fig. 6 is that compared to SOM_{MT}, more of SOM_{PE}'s optimizations have a positive effect on performance, which is also larger on average. Added catch-return node (−1.1%), typed arguments (−1.1%), minimization of escaping closures (−0.1%)), and direct access to variables in local scope (0.3%) have only insignificant effect on peak performance.

Typed variables give an average speedup of (4.6%) (min. −13.9%, max. 32.6%). Thus, there is some speedup, however, in most situations partial evaluation is able to achieve the same effect without the type specialization.

Inlining of basic operations, which avoids full method calls, e. g., for arithmetic operations, shows a speedup of 5.8% (min. −5.8%, max. 1.6x). It shows that in many cases the optimizer is able to remove the overhead of method calls. However, the optimization provides significant speedup in other cases as for instance complex loop conditions.

Array strategies give a speedup of 18.1% (min. −19%, max. 2x), which is comparable to the speedup for SOM_{MT}, but slightly higher.

The lowering of common operations gives an average speedup of 18.7% (min. −6.5%, max. 2.8x). The results are similar to the ones for SOM_{MT}, indicating the general usefulness of these optimization independent of the technique to determine compilation units. Furthermore, the benefit of the optimization here is again higher for SOM_{PE}.

The optimization for object fields improves performance significantly. For the SOM_{PE} interpreter, it was causing a slowdown. With the partial evaluation and subsequent compilation however, we see a speedup of 41.1% (min. −5.8%, max. 11.2x). Thus, typed object fields contribute significantly to the overall peak performance, despite their negative impact on interpreter performance. The benefit of typing variables and arguments seems to be minimal. Here the optimizer has already sufficient information to generate efficient code regardlessly.

The caching of globals gives an average speedup of 79.9% (min. -3%, max. 10x). Compared to RPython, on Truffle this form of node specialization is the only way to communicate runtime constants to the optimizer and as the results show, it is important for the overall performance.

Custom inline caching at method call sites and block invocations is the second most beneficial optimization. It results on average in a speedup of 3x (min. 0%, max. 19.6x). On SOM_{MT}, this optimization did not give any improvements because RPython offers annotations that communicate the same information to the compiler. With Truffle however, inline caching is only done by chaining nodes with the cached data to the call site AST node. While tracing intrinsically inlines across methods, Truffle needs these caching nodes to see candidates for inlining. Since inlining enables many other classic compiler optimizations, it is one of the the most beneficial optimizations for SOM_{PE}.

The lowering of control structures is the most beneficial optimization for SOM_{PE}. It gives an average speedup of 4.3x (min. -0.2%, max. 232.6x). Similar to SOM_{MT}, expressing the semantics of loops and other control flow structures results in significant performance improvements. In Truffle, similar to RPython, the control structures communicate additional information to the compilation backend. In SOM_{PE}, loops record loop counts to direct the adaptive compilation. Similarly, branching constructs record branch profiles to enable optimizations based on branch probabilities.

Conclusion. Considering all optimizations that are beneficial on average, and show for at least one benchmark larger gains, we find that array strategies, typed fields, and lowering of common operations and control structures are highly relevant for both meta-compilation approaches.

Inline caching and caching of globals is realized with annotations in RPython's meta-tracing and thus, does not require the optimizations based on node specialization, even so, they are beneficial for the interpreted mode. However, with partial evaluation, the node specializations for these two optimizations provide significant speedup. Inlining of basic operations is beneficial for partial evaluation. While we did not apply this optimization to SOM_{MT}, it is unlikely that it provides benefits, since the same result is already achieved with the annotations that are used for basic inline caching. The typing of variables was also only applied to SOM_{PE}. Here it improves peak performance. For SOM_{MT}, it might in some cases also improve performance, but the added complexity might lead to a result like, e. g., for the minimizing of escaping variables, which does not improve peak performance on average.

Thus, overall we conclude that partial evaluation benefits more from the optimizations in our experiments by generating higher speedups. Furthermore, we conclude that more optimizations are beneficial, because partial evaluation cannot provide the same implicit specialization based on runtime information that meta-tracing provides implicitly.

4.4 SOM_{MT} vs. SOM_{PE}

To compare the overall performance of SOM_{MT} and SOM_{PE}, we use their respective baseline version, i. e., including all optimizations. Furthermore, we compare their performance to Java. The compiled performance is compared to the results for the HotSpot server compiler and the interpreted performance to the Java interpreter (`-Xint`). Note, the results for the compiled and interpreted modes are not comparable. Since the performance difference is at least one order of magnitude, the benchmarks were run with different parameters. Furthermore, cross-language benchmarking is inherently problematic. While the benchmarks are very similar, they are not identical and, the VMs executing them are tuned based on how the constructs are typically used, which differ between languages. Thus, the reported comparison to Java is merely an indication for the general order of magnitude one can expect, but no reliable predictor.

Figure 7 shows that SOM_{MT}'s peak performance is on this benchmark set on average 3x (min. 1.5x, max. 11.5x) slower than Java 8 on HotSpot. SOM_{PE} is about 2.3x (min. 3.9%, max. 4.9x) slower. Thus, overall both SOMs reach within 3x of Java performance, even so they are simple interpreters running on top of generic JIT compilation frameworks. This means both meta-compilation approaches achieve the goal of reaching good performance. However, SOM_{MT} is slower than SOM_{PE}. At this point, we are not able to attribute this performance difference to any conceptual differences between meta-tracing and partial evaluation as underlying technique. Instead, when investigating the performance differences, we see indications that the performance differences are more likely an indication of the amount of engineering that went into the RPython and Truffle projects, which results in Truffle and Graal producing more efficient machine code, while RPython has remaining optimization opportunities. For instance, GraphSearch is much slower on SOM_{MT} than on SOM_{PE}. The main reason is that RPython currently does not optimize the transition between traces. The benchmark has many nested loops and therefore trace transitions. But instead of passing only the needed values when transferring to another trace, it constructs a frame object with all argument and local variable structures. RPython could optimize this by transitioning directly to the loop body and passing only the values that are needed.

The performance of SOM_{MT} being only interpreted is about 5.6x (min. 1.6x, max. 15.7x) lower than that of the Java 8 interpreter. Similarly, SOM_{PE} is about 6.3x (min. 1.9x, max. 15.7x) slower than the Java 8 interpreter. Here we see some benchmarks being more than an order of magnitude slower. Such high overhead can become problematic when applications have short runtimes and very irregular be-

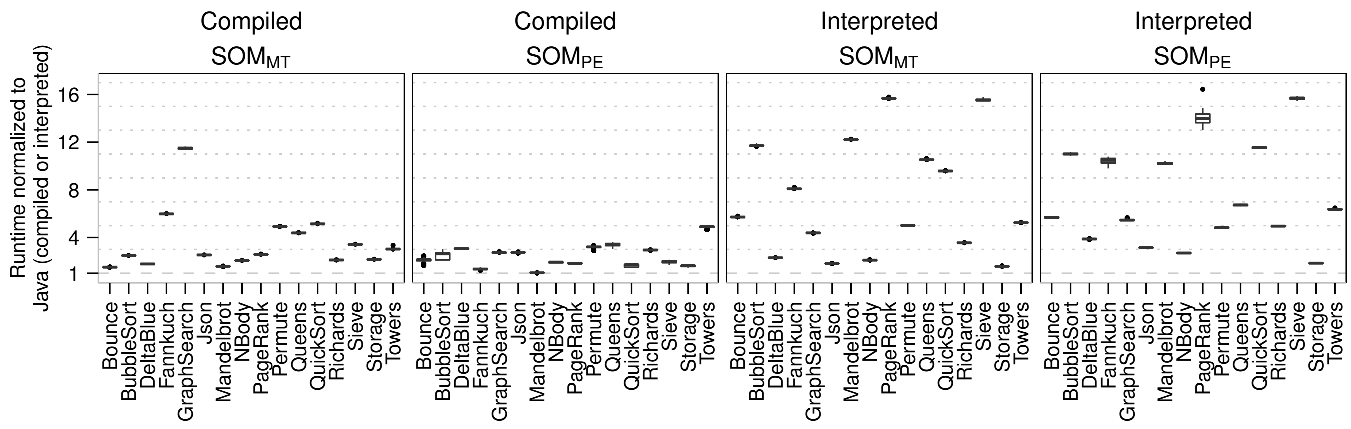


Figure 7. SOM performance compared to Java. The *compiled* performance are the SOMs with JIT compiler compared to HotSpot's peak performance. The *interpreted* performance is compared to the HotSpot interpreter (`-Xint`).

havior, because only parts of the application are executed as compiled code with good performance.

4.5 Whole Program and Warmup Behavior

In addition to interpreter and peak performance, the perceived performance for users is also a relevant indicator. Typically, it is influenced by the warmup behavior, i. e., the time it takes to reach peak performance and the overall time it takes to execute a program. To characterize RPython and Truffle more closely in this respect, we measure the time it takes to execute a given benchmark n times. The measured time is wall-clock time and includes process start and shutdown.[8] By varying n, we can approximate the warmup behavior. By using wall-clock time, we further abstract from the concrete time a single iteration takes by accounting for garbage collection, compilation, and other miscellaneous runtime overheads. In contrast to the previous measuments, we have only a single measurement for each n for each of the benchmarks. Because of the long runtimes, it was impractical to collect more. However, the graphs indicate that the measurement errors are acceptable since the lines are relatively smooth and the results correspond to the other measurements.

Figure 8 depicts the results for our benchmark set. To emphasize the warmup behavior, the results are normalized with $f(n) = time_{\mathrm{VM}}(n)/(time_{\mathrm{Java}}(1000)/1000 * n)$ that represents an idealized behavior based on Java's peak performance. This means, each result is normalized by the the n-th fraction of the result for Java with 1000 iterations. This approach results in a plot that shows the warmup behavior for all three systems and allows us to compare them visually. At each point, the plot shows the factor by which SOM$_{\mathrm{MT}}$, SOM$_{\mathrm{PE}}$, and Java are slower than a hypothetical VM with Java peak performance.

[8] It is measured with the common Unix utility `/usr/bin/time`.

For the first benchmark, Bounce, we see that SOM$_{\mathrm{MT}}$ starts out to be minimally faster than Java, but then Java warms up faster and SOM$_{\mathrm{MT}}$ eventually cannot keep up with it. SOM$_{\mathrm{PE}}$ however starts out being significantly slower and then warms up slowly. On this particular benchmark, SOM$_{\mathrm{MT}}$ remains faster so that the high warmup cost of SOM$_{\mathrm{PE}}$ is not compensated by higher peak performance. For benchmarks such as Fannkuch or GraphSearch on the other hand, SOM$_{\mathrm{PE}}$ warms up faster and compensates for its warmup cost early on. Averaging these results over all benchmarks, we find that SOM$_{\mathrm{PE}}$ starts out to be about 16.3x slower than Java and after 1000 benchmark iterations reaches 2.4x. SOM$_{\mathrm{MT}}$ starts out with about 1.5x slower and is after 1000 iterations 3.1x slower than Java. Compared to SOM$_{\mathrm{MT}}$, it takes SOM$_{\mathrm{PE}}$ about 200 iterations to break even and reach a performance of 3x slower than Java.

In its current state, Truffle does not optimize startup performance. On the one hand, it builds on the standard HotSpot JVM and all interpreter code as well as the code of the Graal JIT compiler are first compiled by HotSpot, which increases the warmup time. On the other hand, the Graal JIT compiler itself is designed to be a top-tier compiler optimizing for peak performance, which makes it comparably slow. RPython on the other hand does create a static binary of the interpreter, which does not need to warmup and therefore is initially faster. From the conceptual perspective, this difference is not related to the meta-compilation approaches, but merely an artifact of the concrete systems.

4.6 Memory Usage

With the differences in how objects are represented between Java and our SOM implementations, as well as the question of how effective optimizations such as escape analyses are, it is interesting to investigate the memory usage of programs executing on RPython and Truffle. Especially for programs with large data sets, memory usage can have a major per-

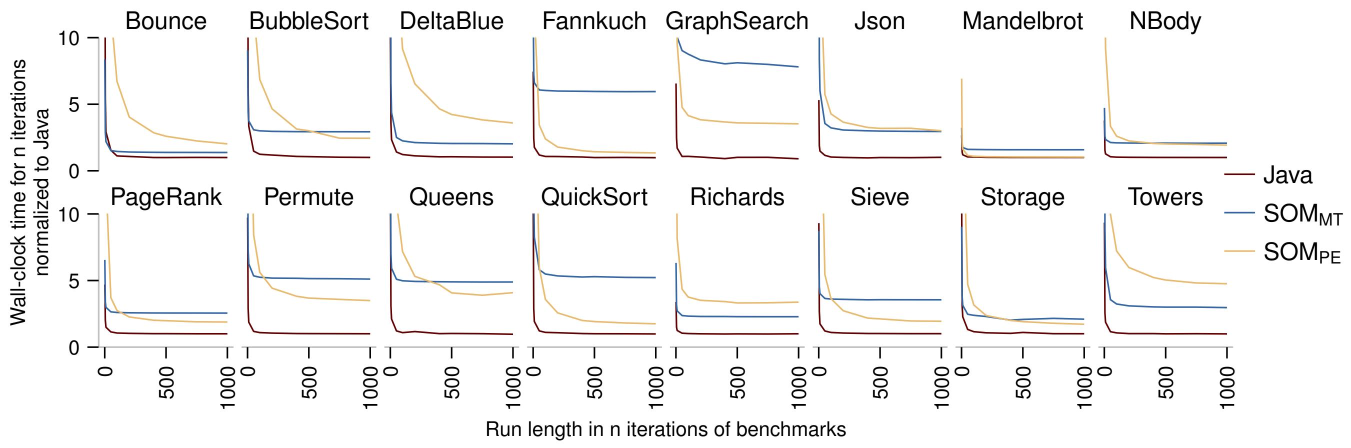


Figure 8. Whole program behavior of SOM compared to Java. Each benchmark is execute n times within the same VM processes and we measure the overall wall-clock time for the execution. For each benchmark, the result for n iterations is normalized to the n-th fraction of Java with 1000 iterations.

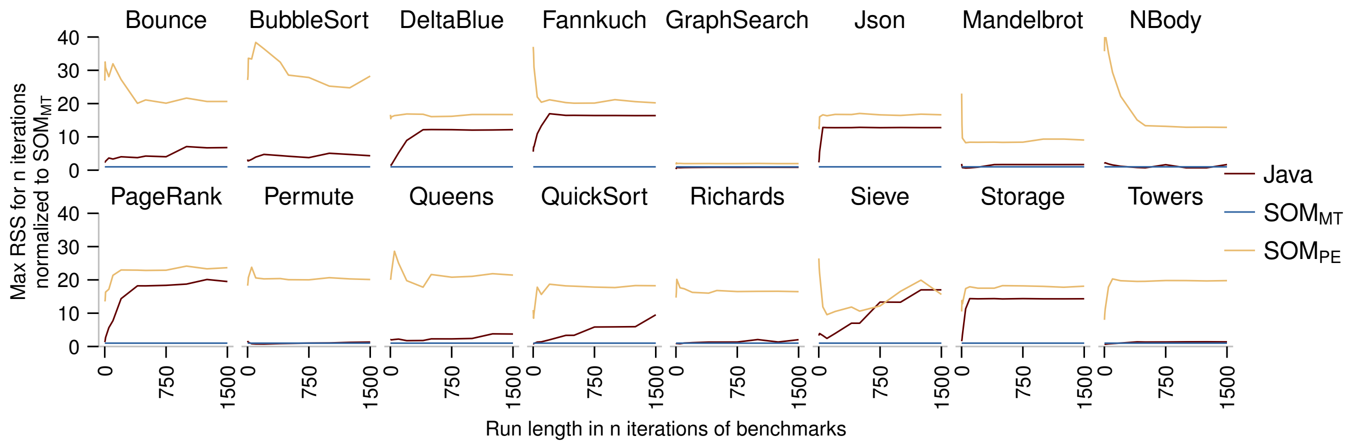


Figure 9. Maximum resident set size, i. e., maximum memory usage of SOM and Java normalized to SOM_{MT}. Each benchmark is execute n times within the same VM processes and we measure the max. RSS for the execution. For each benchmark, the result for n iterations is normalized to SOM_{MT}.

formance impact. For this comparison we can unfortunately not rely on precise information since RPython does not provide access to the current heap usage or statistics from the garbage collector. Thus, we measure the maximum resident set size (RSS) as reported by the Unix `time` utility. This number only gives a rough indication of the maximal memory pressure during program execution. Thus, we measure it for different number of iterations of the benchmarks. However, this numbers is also interesting, because it includes all memory used by the systems. It includes the garbage collected heap memory as well as memory that is used by the VM for instance for the generated machine code.

The results are depicted in fig. 9. The measurements are normalized based on SOM_{MT}, because it has the smallest overall resident set size, and the resulting graph shows more details than if it would be normalized to Java. Note that the direct comparison between SOM_{MT} and Java or SOM_{PE} is not allowing any conclusion with respect to the meta-compilation approaches, because the systems are too different. However, a comparison of SOM_{PE} with Java is possible.

Averaged over all benchmarks, SOM_{PE} has at the first iteration an 9.6x higher max. RSS than Java. After 1500 iterations, the difference is down to 3.2x. This means, that SOM_{PE} has a higher initial footprint than Java. The dynamic overhead seems to be still higher than Java's but significantly less then the initial factor of 9.6x.

Currently, neither SOM_{PE} nor SOM_{MT} use precise allocation, i. e., minimize the allocated memory for objects based on the knowledge of their layout. Instead, they use an object representation with 5 fields for primitive values (longs or doubles), 5 fields for object values, and optional extension arrays for primitive and object values. In praxis, this means that small objects use more space then needed. Arrays on the other hand use storage strategies and thus, do not use more memory than necessary.

Since the garbage collectors of RPython and HotSpot are so different, we cannot draw conclusions from this data with respect to the meta-compilation approaches.

4.7 Implementation Sizes

In addition to the achievable performance, engineering aspects can be of importance for language implementations as well. To gain some insight of how partial evaluation and meta-tracing compare in that regard, we determine the implementation sizes of the experiments. However, in addition to the weak insights measurement of implementation size provides, it needs to be noted that the obtained numbers are only directly comparable for experiments with the same SOM implementation. Since Java and RPython have significant syntactical and semantic differences, a direct comparison is not possible. Instead, we compare the relative numbers with respect to the corresponding baseline implementation. The reported percentages are based on the implementation without an optimization as denominator so that the percentage indicates the change needed to add the optimization.

As first indication, we compare the minimal versions of the SOM interpreters without optimizations with the baseline versions. SOM_{MT} has 3455 lines of code (LOC, excluding blank lines and comments) with all optimizations added it grows to 5414 LOC which is a 57% increase. The minimal version of SOM_{PE} has 5424 LOC and grows to 11037 LOC with all optimizations, which is an increase of 103%. Thus, SOM_{PE} is overall larger, which is expected since we apply more optimizations.

Table 2 lists the data for all experiments incl. absolute numbers. Comparing the relative increases of implementation sizes for SOM_{MT} and SOM_{PE} indicates that the optimizations are roughly of the same size in both cases. The only outlier is the implementation of inline caching which is larger for SOM_{PE}. Here the language differences between RPython and Java are becoming apparent and causes the SOM_{PE} implementation to be much more concise.

Conclusion. Considering performance and implementation sizes combined, we see for SOM_{MT} an overall peak performance increase of 1.8x (min. -10.5%, max. 5.4x) for going from the minimal to the baseline version. The interpreter performance improves by 2.4x (min. 41.5%, max. 3.9x). Note, the minimal version includes one trace merge point in the `while` loop to enable trace compilation (cf. section 3.2). For SOM_{PE}, the peak performance improves by 78.1x (min. 22.8x, max. 342.4x) from the minimal to the baseline version. SOM_{PE}'s interpreter speed improves by 4x (min. 2.1x, max. 7.3x). SOM_{PE} also implements `while` in the interpreter, but it does not provide the same benefits for the partial evaluator as it does for the meta-tracer.

We conclude that for partial evaluation the optimizations are essential to gain performance. For meta-tracing however, they are much less essential and can be used more gradually to improve the performance for specific use cases.

5. Discussion

This sections discusses technical questions and possible alternative design choices for RPython and Truffle. The general design of this study, its conclusiveness, and the generalizability of the results are also discussed as part of section 3.

5.1 Performance Results

After studying the impact of various optimization on SOM_{MT} and SOM_{PE}, the question arrises whether the observed performance effects are generalizable to other languages. Without further experiments, it needs to be assumed that they are not directly transferable. To give but a single example, for SOM_{PE} we observed no benefit for peak performance from specializing method argument access based on their types. On the contrary, the interpreter showed clear performance drawbacks. However, in SOM, arguments are not assignable and methods are generally short. The usage pattern for arguments can thus be different in languages with assignable argument variables such as Java. Thus, other languages potentially benefit from this optimization. Nonetheless, the observations made here can provide initial guidance for other language implementations to prioritize the optimization effort. For instance, the relevance of inline caching is not only widely acknowledge in literature but is also very relevant in other Truffle languages such as JRuby+Truffle [22].

Since the performance of the SOM implementations is compared with Java, the result of being in the range of 2.3-3x slower leads to the question of whether it is a limitation of the meta-compilation approaches. From our perspective, the reason for the performance difference is based on the language differences and further optimization potential in RPython and Truffle as well as in our interpreters. Since Java and SOM have different language characteristics, even highly optimizing compilers cannot produce identical code for both languages. Operations on Java's integer types for instance do not require any dynamic checks. On the other hand, SOM requires dynamic dispatch of operators as well as the promotion to arbitrary precision integers on overflow. Compilers can reduce the impact of such differences, for instance by moving checks out of loops, however, the checks cannot be eliminated completely without changing the language's semantics. Nonetheless, RPython and Truffle seem to be able to deliver performance comparable with classic JIT compilation approaches, indicated for instance by Truf-

	SOM_{MT}	SOM_{PE}		SOM_{MT}			SOM_{PE}	
	LOC %	LOC %	LOC	ins.	del.	LOC	ins.	del.
baseline	0.0	0.0	5414	0	0	11037	0	0
array strategies	11.6	9.0	4851	37	829	10125	126	1233
cache globals	0.5	1.7	5386	2	41	10853	14	239
catch-return nodes	0.3	0.4	5397	12	36	10995	54	107
inline basic ops.		3.7				10647	0	430
inline caching	2.0	7.9	5307	1	158	10231	95	1095
lower common ops	10.2	9.1	4912	2	678	10115	1	1083
lower control structures	12.2	9.9	4824	8	790	10045	9	1160
min. escaping closures	0.4	0.9	5394	5	30	10943	42	152
min. escaping vars	1.7		5322	20	130			
opt. local vars	1.0	1.6	5359	49	135	10863	70	284
typed args		1.4				10886	204	383
typed fields	10.2	11.1	4912	18	698	9933	39	1393
typed vars		1.1				10915	9	161

Table 2. Implementation sizes of the implementations without the optimization. LOC: Lines of code excluding comments and empty lines, LOC %: increase of LOC to add optimization, ins./del.: inserted and deleted lines as reported by `git`

fle/JS performing in the range of V8 and SpiderMonkey.[9] Remaining optimization potential in both systems is for instance in the inter-compilation-unit calling convention. Currently, both systems use a simplified approach that requires boxing all arguments and pass them in an argument object or array. Since both system however need to know the types of these arguments in either case, they could generate code with more efficient signatures. Furthermore, in the SOM interpreters other optimizations could be added, for instance precise object allocation to reduce memory usage, using storage strategies for other common data structures beside arrays to avoid boxing overhead, and more optimizations for standard library functionality.

Another aspect this study does not discuss in detail is the impact of self-optimizations on memory usage. While we see that the maximal memory usage of Java and SOM_{PE} become more similar with increasing runtime (cf. section 4.6), we did not assess the concrete memory usage of the ASTs. However, the general requirement for self-optimizing interpreters is that the AST stabilizes at some point [27]. This implies that self-modification should only introduce an upper bound of nodes, which limits the additional memory requirements. Whether this can lead to excessive memory consumption on large applications remains an open question, but since AST nodes are generally small objects with only few fields, it seems unlikely.

5.2 Meta-Tracing vs. Partial Evaluation

A major difference between the two approaches is their overhead during interpretation. Partial evaluation requires the interpreter to record information about the executed code for instance branch probabilities and unused code paths. This information is used by the compiler to guide optimization together with heuristics, e. g., to avoid compilation of exception handling in the standard case. While sampling might reduce the overhead of collecting the runtime feedback, Truffle does currently use a precise approach that is active at all times, leading to a high overhead during interpretation.

With meta-tracing, the interpreter tracks execution only at the trace merge points. Only during tracing, which happens very infrequently, it records additional information needed for optimization. Thus, in a tracing system, interpreter performance might have conceptual advantages over a system with partial evaluation.

From a language implementers perspective, it can be argued that the meta-tracing approach as exemplified by RPython also is a conceptually purer approach in the sense that it requires only to reason about interpretation behavior. With partial evaluation on the other hand, the language implementer needs to reason about *compilation time* as well. Since partial evaluation is performed strictly independent of actual execution, profiling information and value caches need to be collected separately during execution to facilitate the later partial evaluation and optimization. This comes with the consequence that not only a single value has to be regarded as during the concrete tracing execution, but multiple values, i. e., general polymorphism has to be handled directly. Note, the explicit reasoning about *compilation time* is not necessarily a drawback, since it makes performance relevant polymorphism explicit.

5.3 RPython vs. Truffle

The main difference observed between RPython and Truffle is the performance difference between unoptimized interpreters. With RPython's meta-tracing, the performance is

[9] *Performance: JavaScript, Slide 86*, Graal Tutorial, Christian Wimmer, CGO, 2015, access date: 2015-07-19 `http://lafo.ssw.uni-linz.ac.at/papers/2015_CGO_Graal.pdf`

already in the same order of magnitude, while Truffle's partial evaluation results in one order difference. While much of the difference can be attributed to the missing compile-time knowledge of method calls, and thus, the missing support for inlining on language level, another important difference between the two systems is the chosen language in which interpreters are implemented. Truffle uses standard Java with full Java semantics. This comes for the partial evaluation with additional restrictions. For instance, Java gives certain guarantees with respect to object identities, which restricts for instance optimizations avoid boxing. Another relevant restriction is that interfaces are not sufficient to optimize in all cases, which requires the use of concrete value profiles to enable the optimizer to know that certain objects are of a specific class and optimize accordingly. Such profiling information can be provided with the Truffle framework. However, compared to RPython, it requires additional work from the language implementer.

The benefit Truffle gains from the use of Java is that existing Java code can be easily integrated into an interpreter. It can even become part of a Truffle compilation unit and thus be highly optimized on the fast path. From our perspective, there are however sufficient indications that it restricts the partial evaluation and optimizations consequently requiring language implementers to provide more self-optimizations in their interpreters than ideally would be required.

From the perspective of how knowledge is communicate to the optimizers, both RPython and Truffle turn out to be very similar. With RPython's `@elidable` and `promote()`, the compiler can be told about runtime constants. Very similar, Truffle's `ValueProfile` fulfills the same purpose. A second concept is explicit loop unrolling for instance for the processing of a constant number of method arguments. In RPython, the `@unroll_safe` annotation is used for this, and in Truffle the equivalent `@ExplodeLoop` annotation. A third relevant concept is global optimistic speculation. RPython has the notion of *quasi-immutable* fields, which do not leave a runtime check in the code, but instead writes to such fields cause an invalidation of all compiled code that depended on the field's value. In Truffle, this is handled by the `Assumption` class, which also causes an invalidation on all code that depends on it incase it is invalidated. Since these are the major mechanisms offered by the two systems, and offered in very similar ways, there does not seem to be an immediate opportunity for either of the systems to add a missing mechanism.

When implementing a language, tooling can be a relevant deciding factor for RPython or Truffle. When optimizing an implementation, tools need to make it easy to understand and relate the optimizations done by the respective toolchains to an input program in the language that is implemented. Based on the current status of the tools provided with both systems, there seems to be some benefit for meta-tracing. Since all optimizations are based on traces that linearize control flow, the tools are able to attribute relatively accurately the optimized instructions in a trace to the elements of the language implementation they originate from. In practice, this means that a program is relatively easily recognized in a trace, which supports the understandability of the results. For Truffle on the other hand, the available tool for inspecting the control- and data-flow graph of a program does not maintain the connection to the language implementation. Part of the issue is that some of Graal's compiler optimizations can duplicate or merge nodes, which complicates the mapping to the input program.

Another practical aspect are the platforms' capabilities and their ecosystems. Since Truffle builds on the JVM, support for threads, a memory model, and a wide range of software is implicitly give. Furthermore, the use of JVM-based software does not introduce a compilation boundary and thus, just-in-time compilation can optimize a Truffle-based language together with other libraries. RPython on the other hand does not come with comprehensive support for threads. Furthermore, its integration into the surrounding ecosystem is based on a foreign function interface (rffi), which is a compilation boundary for the tracing compiler.

6. Related Work

As far as we are aware, there is no other study comparing meta-tracing and partial evaluation in detail. In previous work, we studied whether both approaches deliver on their performance promise [21]. However, we compared a bytecode-based with a self-optimizing AST interpreter limiting the explanatory value of the results. In this study, we compare two self-optimizing AST interpreters and further detail the impact of optimizations, overall performance, whole program behavior, and memory usage.

Interpreters and Optimizations Related to Würthinger et al. [27]'s self-optimizing interpreters is for instance quickening and superinstructions focused on bytecode-based interpreters [9, 10, 23].

The optimizations proposed for self-optimizing interpreters cover a wide range of topics and the optimizations discussed in this paper are either directly based on the literature or small variations. Würthinger et al. [27] initially discussed operation specialization by type, dynamic data type specialization, type specialization of local variable and field accesses, boxing elimination, and polymorphic inline caching (cf. also Würthinger et al. [28]). Later, Wöß et al. [29] detailed the strategy for field access optimization with an object storage model. Kalibera et al. [20] discussed the challenges of a self-optimizing interpreter for the R language to address the dynamic and lazy nature of R. They detail a number of structural optimizations similar to the ones discussed here, dynamic operation and variable specialization, inline caching, data type specializations, as well as a profiling-based optimization of R's view feature, which is a complex language feature that has different tradeoffs depending on the size of vectors it is used on. A similarly complex language feature that has been optimized in this context is Python's generators [30].

Meta-Tracing Bolz and Tratt [5] discuss the impact of meta-tracing on the VM design and implementation. They detail how an implementation needs to expose for instance data dependencies, compile-time constants, and elidable computations clearly to the tracer for best optimization results. Generally, they advise to expose runtime constants also on the level of the used data structures. Thus, to prefer fixed sized arrays over variable sized lists, and to use known techniques such as *maps* [11] to optimize objects to provide the tracer and subsequent optimization with as much information about runtime constants as possible. In this study, we find that these general suggestions apply to both compilation techniques, meta-tracing as well as partial evaluation.

Beside RPython, SPUR is another system that uses meta-tracing just-in-time compilation for dynamic languages [3]. We did not investigate it in this study since it requires that the language is compiled to the *Common Intermediate Language* (CIL), and thus, has a different and not directly compatible approach with RPython and Truffle. The general benefit of the system we study is that language implementers build simple interpreters, without requiring an additional compilation step.

7. Conclusion and Future Work

This study compares tracing and partial evaluation as meta-compilation techniques for self-optimizing AST interpreters. The results indicate that both techniques enable language implementations to reach average performance within 3x of Java. A major difference between meta-tracing and partial evaluation is the amount of optimization a language implementer needs to apply to reach the same level of performance. Our experiments with SOM, a dynamic class-based language, indicates that meta-tracing performs well even without adding optimizations. With the additional optimizations it is on average only 3x (min. 1.5x, max. 11.5x) slower than Java. SOM_{MT} reaches this results with 5414 LOC. For partial evaluation on the other hand, we find that many of the optimization are essential to reach good performance. With all optimizations, SOM_{PE} is on average only 2.3x (min. 3.9%, max. 4.9x) slower than Java. SOM_{PE} reaches this result with 11037 LOC. We conclude overall that meta-tracing and partial evaluation can reach the same level of performance. However, meta-tracing has significant benefits from the engineering perspective, because the optimizations provide generally fewer performance benefits and thus are less critical to be applied.

Since this study uses with Truffle and RPython two independent systems, we consider the observed difference in absolute performance as insignificant. We find that tracing and partial evaluation are equally suited for meta-compilation, and that the observed performance differences are merely an artifact of the different amounts of engineering that went into Truffle and RPython. Future work could verify this by studying both techniques on top of the same optimization infrastructure. For instance a tracing JIT compiler on top of HotSpot [18] could be used to verify whether the observed engineering benefits are a consequence of tracing. On the other hand, if the partial evaluated language would be more geared towards it than Java, it might also reduce the self-optimizations that are necessary to reach peak performance.

The interpreted performance of self-optimizing interpreters could still benefit from significant improvements. Possible research directions include approaches similar to superinstructions [10] on the AST level to avoid costly polymorphic method invocations. Another direction could be to attempt the generation of bytecode interpreters potentially in highly efficient machine code to reach interpretive performance competitive with for instance Java's bytecode interpreter.

A. Evaluated Optimization Techniques

This appendix details the interpreter optimizations used for the study of this paper. The optimizations are grouped into structural and dynamic optimizations as well as lowering of language and library functionality.

A.1 Structural Optimizations

Literature discusses many optimizations that can be performed after parsing a program, without requiring dynamic information. We chose a few to determine their impact in the context of meta-compilation. Note, each optimization has a shorthand by which we refer to it throughout the paper.

Distinguish Variable Accesses in Local and Non-Local Lexical Scopes (opt. local vars) In SOM, closures can capture variables of their lexical scope. A variable access thus needs to determine in which lexical scope the variable is to be found, then traverse the scope chain, and finally do the variable access. SOM's compiler can statically determine whether a variable access is in the local scope. At runtime, it might be faster to avoid the tests and branches of the generic variable access implementation. Thus, in addition to the generic AST node for variable access, this optimization introduces an AST node to access local variables directly.

Handle Non-Local Returns Only in Methods including Them (catch-return nodes) In recursive AST interpreters such as SOM_{PE} and SOM_{MT}, non-local returns are implemented using exceptions that unwind the stack until the method is found from which the non-local return needs to exit. A naive implementation handles the return exception simply in every method and checks whether it was the target. However, the setup for exception handlers as well as catching and checking an exception has a runtime cost on most platforms, and the handling is only necessary in methods that actually contain lexically embedded non-local returns.

Thus, it might be beneficial to do the handling only in methods that need it. Since it is known after parsing a method whether it contains any non-local returns, the handling can be represented as an extra AST node that wraps the body of the method and is only added when necessary.

Expose Variables in Lexical Scope Only if Accessed (min. escaping vars, SOM$_{MT}$ only) Truffle relies on a rigid framework that realizes temporary variables of methods with `Frame` objects. The partial evaluator checks that these *frames* do not escape the compilation unit, so that they do not need to be allocated. However, for lexical scoping, frame objects can *escape* as part of a closure object. In Truffle, such escaping frames need to be *materialized* explicitly. Instead of using such a strict approach, RPython works the other way around. An object can be marked as potentially *virtual*, so that its allocation is more likely to be avoided depending on its use in a trace.

To help the implicit approach of RPython in SOM$_{\text{MT}}$, the frames can be structured to minimize the elements that need to escape to realize closures. At method compilation time, it is determined which variables are accessed from an inner lexical scope and only those are kept in an array that can escape. The optimizer then ideally sees that the frame object itself does not need to be allocated. Since Truffle fixes the structure of frames, this optimization is not applicable to SOM$_{\text{PE}}$.

Avoid Letting Unused Lexical Scopes Escape (min. escaping closures) While the previous optimization tries to minimize the escaping of frames by separating variables, this optimization determines for the whole lexical scope whether it is needed in an inner scope or not. When the scope is not used, the frame object is not passed to the closure object and therefore will not escape. The optimization is realized by using a second AST node type that creates the closure object with `null` instead of the frame object.

A.2 Dynamic Optimizations

While the discussed static optimizations can also be applied to other types of interpreters, the dynamic optimizations are self-optimizations that require runtime information.

Cache Lookup of Global Values (cache globals) In SOM, values that are globally accessible in the language are stored in a hash table. Since classes as well as the literals `true`, `false`, and `nil` are globals, accessing the hash table is a common operation. To avoid the hash table lookup at runtime, globals are represented as association objects that can be cached after the initial lookup in a specialized AST node. The association object is necessary, because globals can be changed. For `true`, `false`, and `nil`, we optimistically assume that they are not changed and specialize the access to a node that returns the corresponding constants directly.

Cache Method Lookups and Block Invocations (inline caching) In dynamic languages, inline caching of method lookups is common to avoid the overhead of traversing the class hierarchy at runtime for each method invocation. In self-optimizing interpreters, this is represented as a chain of nodes, which encodes the lookup results for different kinds of objects as part of the caller's AST. In addition to avoiding the lookup, this technique also exposes the target method as a constant to the compiler which in turn can decide to inline a method to enable further optimizations. Similar to caching method lookups, it is beneficial to cache the code of closures at their call sites to enable inlining.

In both cases, the node chains are structured in a way that each node checks whether its cached value applies to the current object or closure, and if that is not the case, it delegates to the next node in the chain. At the end of the chain, an uninitialized node either does the lookup operation or in case the chain grows too large, it is replaced by a fallback node that always performs the lookup.

Type-Specialize Variable Accesses (typed vars, SOM$_{PE}$ only) As mentioned earlier, Truffle [27] uses `Frame` objects to implement local variables. For optimization, it tracks the types stored in a frame's *slots*, i. e., of local variables. For SOM$_{\text{PE}}$, Truffle thus stores `long` and `double` values as primitives, which avoids the overhead of boxing. Furthermore, SOM$_{\text{PE}}$'s variable access nodes specialize themselves based on this type information to ensure that all operations in this part of an AST work directly with unboxed values.

Since SOM$_{\text{MT}}$ uses uniform boxing, this optimization is not applied.

Type-Specialize Argument Accesses (typed args, SOM$_{PE}$ only) With the type specialization of SOM$_{\text{PE}}$'s access to local variables, it might be beneficial to type specialize also the access to a method's arguments. In Truffle, arguments to method invocations are passed as an `Object` array. Thus, this optimization takes the arguments passed in the object array and stores them into the frame object to enable type specialization. While this does not avoid the boxing of primitive values on method call boundaries, it ensures that they are unboxed and operations on these arguments are type specialized in the body of a method.

Note, since the variable access optimization is not applicable to SOM$_{\text{MT}}$, this optimization is not applicable either.

Specialize Object Field Access and Object Layout (typed fields) To avoid boxing, it is desirable to store unboxed values into object fields as well. Truffle provides support for a general object storage model [29] that is optimized for class-less languages such as JavaScript, and is similar to maps in Self [11]. To have identical strategies, SOM$_{\text{PE}}$ and SOM$_{\text{MT}}$ use a simplified solution that keeps track of how object fields are used at runtime, so that `long` and `double` values can be stored directly in the primitive slots of an object. For each SOM class, an object layout is maintained that maps the observed field types to either a storage slot for primitive values or to a slot for objects. The field access nodes in the AST specialize themselves according to the object layout that is determined at runtime to enable direct access to the corresponding slot.

Type-Specialize Array Storage (array strategies) Similar to other dynamic languages, SOM only has generic object arrays. To avoid the overhead of boxing, we implement strategies [8] for arrays. It is similar to the idea of specializing the access and layout of object fields. However, here the goal is to avoid boxing for arrays that are used only for either `long`, `double`, or `boolean` values. In these cases, we specialize the storage to an array of the primitive type. In the case of booleans, it also reduces the size of the array from a 64-bit pointer to a byte per element.

Specialize Basic Operations (inline basic ops., SOM$_{PE}$ only) As in other dynamic languages, SOM's basic operations such as arithmetics and comparisons are normal method invocations on objects. Thus for instance the expression `1 + 2` causes the *plus* method to be invoked on the `1` object. While this allows developers to define for instance addition for arbitrary classes, in most cases arithmetics on numbers still use the built-in method. To avoid unnecessary method lookups and the overhead of method invocation, we specialize the AST nodes of basic operations directly to the built-in semantics when the type information obtained at runtime indicate that it is possible.

Note, since this relies on TruffleDSL and its handling of the possible polymorphism for such specializations, this optimization is not applied to SOM$_{MT}$.

A.3 Lowerings

The last category of optimizations covers the reimplementation of standard library functionality as part of the interpreter to gain performance.

Control Structures (lower control structures) Similar to specializing basic operations, we specialize control structures for conditionals and loops. In SOM, conditional structures are realized as polymorphic methods on boolean objects and loops are polymorphic methods on closures. An optimization of these constructs is of special interest because they are often used with lexically defined closures. Thus, in the context of one method, the closures reaching a control structure are statically known. Thus, specializing the control structures on the AST level does not only avoid overhead for method invocations done in the language-level implementation, but also utilizes directly the static knowledge about the program structure and exposes the closure code directly for further compiler optimizations such as inlining.

In SOM$_{MT}$, such specializations have the benefit of exposing the language-level loops to the implementation by communicating them directly to the meta-tracer with trace merge points (cf. section 3.2).

Common Library Operations (lower common ops) In addition to generic control structures, the SOM library provides many commonly used operations. We selected boolean, numeric, array copying, and array iteration operations for implementation at the interpreter level.

Similar to the specialization of basic operations and control structures, these optimizations are applied optimistically on the AST nodes that do the corresponding method invocation if the observed runtime types permit it.

B. Artifact Overview

This paper is supplemented with an online appendix that includes the experiments and the source artifacts on which this research is based. The artifacts and how to execute the experiments are detailed as at: `http://stefan-marr.de/papers/oopsla-marr-ducasse-meta-tracing-vs-partial-evaluation-artifacts/`.

The artifacts include the following elements:

- a VirtualBox image with the complete experiment set up for experimentation
- the raw data set on which section 4 is based
- R scripts to process the raw data and produce the graphs and numbers used in section 4
- a complete source tar ball containing the snapshot of the used sources
- a ReBench[10] configuration file to execute the benchmarks with the parameters used in this paper

The experiment setup is also accessible via our GitHub repository `https://github.com/smarr/selfopt-interp-performance` on the branch `papers/metatracing-vs-partialevaluation`.

Acknowledgments

The authors would like to thank C. F. Bolz, M. Fijałkowski, and A. Rigo from the PyPy team as well as B. Daloze, G. Duboscq, M. Grimmer, C. Humer, M. Haupt, C. Seaton, L. Stadler C. Wimmer, A. Wöß, T. Würthinger, and W. Zhang from the Truffle community for guidance, support, and discussions on the topic of this research. Their help was essential for enabling the SOM implementations to perform efficiently in the first place. We would further like to thank E. Barrett, C. F. Bolz, T. D'Hondt, L. Tratt, and the anonymous reviewers for their input on this paper.

References

[1] L. Augustsson. Partial Evaluation in Aircraft Crew Planning. In *Proc. of PEPM*, pages 127–136. ACM, 1997.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proc. of PLDI*, pages 1–12. ACM, 2000. ISBN 1-58113-199-2.

[10] *ReBench*, Execute and document benchmarks reproducibly, access date: 2015-07-12 `https://github.com/smarr/ReBench`

[3] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. Spur: A trace-based jit compiler for cil. In *Proc. of OOPSLA*, pages 708–725. ACM, 2010.

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proc. of OOPSLA*, pages 169–190. ACM, 2006.

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

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

[7] C. F. Bolz, M. Leuschel, and D. Schneider. Towards a Jitting VM for Prolog Execution. In *Proc. of PPDP*, pages 99–108. ACM, 2010. ISBN 978-1-4503-0132-9.

[8] C. F. Bolz, L. Diekmann, and L. Tratt. Storage Strategies for Collections in Dynamically Typed Languages. In *Proc. of OOPSLA*, pages 167–182. ACM, 2013.

[9] S. Brunthaler. Efficient Interpretation Using Quickening. In *Proc. of DLS*, pages 1–14. ACM, Oct. 2010.

[10] K. Casey, M. A. Ertl, and D. Gregg. Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters. *ACM Trans. Program. Lang. Syst.*, 29(6):37, 2007.

[11] C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proc. of OOPSLA*, pages 49–70. ACM, 1989. ISBN 0-89791-333-7.

[12] Y. Futamura. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1971/1999.

[13] Y. Futamura. Partial Computation of Programs. In E. Goto, K. Furukawa, R. Nakajima, I. Nakata, and A. Yonezawa, editors, *RIMS Symposia on Software Science and Engineering*, volume 147 of *LNCS*, pages 1–35. Springer, 1983.

[14] A. Gal, C. W. Probst, and M. Franz. HotpathVM: An Effective JIT Compiler for Resource-constrained Devices. In *Proc. of VEE*, pages 144–153. ACM, 2006. ISBN 1-59593-332-6.

[15] 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 *Proc. of ITiCSE*, pages 18–22. ACM Press, June 2010.

[16] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proc. of PLDI*, pages 32–43. ACM, 1992. ISBN 0-89791-475-9.

[17] C. Humer, C. Wimmer, C. Wirth, A. Wöß, and T. Würthinger. A Domain-Specific Language for Building Self-Optimizing AST Interpreters. In *Proc. of GPCE*, pages 123–132. ACM, 2014.

[18] C. Häubl, C. Wimmer, and H. Mössenböck. Context-sensitive Trace Inlining for Java. *Computer Languages, Systems & Structures*, 39(4):123–141, 2013.

[19] U. Hölzle, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proc. of ECOOP*, volume 512 of *LNCS*, pages 21–38. Springer, 1991. ISBN 3-540-54262-0.

[20] T. Kalibera, P. Maj, F. Morandat, and J. Vitek. A Fast Abstract Syntax Tree Interpreter for R. In *Proc. of VEE*, pages 89–102. ACM, 2014. ISBN 978-1-4503-2764-0.

[21] S. Marr, T. Pape, and W. De Meuter. Are we there yet? simple language implementation techniques for the 21st century. *IEEE Software*, 31(5):60–67, September 2014.

[22] S. Marr, C. Seaton, and S. Ducasse. Zero-overhead metaprogramming: Reflection and metaobject protocols fast and without compromises. In *Proc. of PLDI*, pages 545–554. ACM, 2015.

[23] T. A. Proebsting. Optimizing an ANSI C Interpreter with Superoperators. In *Proc. of POPL*, pages 322–332. ACM, 1995.

[24] A. Rigo and S. Pedroni. PyPy's Approach to Virtual Machine Construction. In *Proc. of DLS*, pages 944–953. ACM, 2006.

[25] G. Sullivan. Dynamic Partial Evaluation. In *Programs as Data Objects*, volume 2053 of *LNCS*, pages 238–256. Springer, 2001.

[26] C. Wimmer and S. Brunthaler. ZipPy on Truffle: A Fast and Simple Implementation of Python. In *Proc. of OOPSLA Workshops*, SPLASH '13, pages 17–18. ACM, 2013.

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

[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 *Proc. of Onward!*, pages 187–204. ACM, 2013. ISBN 978-1-4503-2472-4.

[29] A. Wöß, C. Wirth, D. Bonetta, C. Seaton, C. Humer, and H. Mössenböck. An Object Storage Model for the Truffle Language Implementation Framework. In *Proc. of PPPJ*, pages 133–144. ACM, 2014. ISBN 978-1-4503-2926-2.

[30] W. Zhang, P. Larsen, S. Brunthaler, and M. Franz. Accelerating Iterators in Optimizing AST Interpreters. In *Proc. of OOPSLA*, pages 727–743. ACM, 2014.