AST vs. Bytecode: Interpreters in the Age of Meta-Compilation

OCTAVE LAROSE, University of Kent, UK
SOPHIE KALEBA, University of Kent, UK
HUMPHREY BURCHELL, University of Kent, UK
STEFAN MARR, University of Kent, UK

Thanks to partial evaluation and meta-tracing, it became practical to build language implementations that reach state-of-the-art peak performance by implementing only an interpreter. Systems such as RPython and GraalVM provide components such as a garbage collector and just-in-time compiler in a language-agnostic manner, greatly reducing implementation effort. However, meta-compilation-based language implementations still need to improve further to reach the low memory use and fast warmup behavior that custom-built systems provide. A key element in this endeavor is interpreter performance. Folklore tells us that bytecode interpreters are superior to abstract-syntax-tree (AST) interpreters both in terms of memory use and run-time performance.

This work assesses the trade-offs between AST and bytecode interpreters to verify common assumptions and whether they hold in the context of meta-compilation systems. We implemented four interpreters, each an AST and a bytecode one using RPython and GraalVM. We keep the difference between the interpreters as small as feasible to be able to evaluate interpreter performance, peak performance, warmup, memory use, and the impact of individual optimizations.

Our results show that both systems indeed reach performance close to Node.js/V8. Looking at interpreter-only performance, our AST interpreters are on par with, or even slightly faster than their bytecode counterparts. After just-in-time compilation, the results are roughly on par. This means bytecode interpreters do not have their widely assumed performance advantage. However, we can confirm that bytecodes are more compact in memory than ASTs, which becomes relevant for larger applications. However, for smaller applications, we noticed that bytecode interpreters allocate more memory because boxing avoidance is not as applicable, and because the bytecode interpreter structure requires memory, e.g., for a reified stack.

Our results show AST interpreters to be competitive on top of meta-compilation systems. Together with possible engineering benefits, they should thus not be discounted so easily in favor of bytecode interpreters.

CCS Concepts: • **Software and its engineering** → **Interpreters**; *Just-in-time compilers.*

Additional Key Words and Phrases: bytecode, abstract-syntax-tree, language implementation, just-in-time compilation, meta-tracing, partial evaluation, comparison, case study, interpreters

ACM Reference Format:
Octave Larose, Sophie Kaleba, Humphrey Burchell, and Stefan Marr. 2023. AST vs. Bytecode: Interpreters in the Age of Meta-Compilation. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 233 (October 2023), 29 pages. https://doi.org/10.1145/3622808

Authors' addresses: Octave Larose, O.Larose@kent.ac.uk, University of Kent, UK; Sophie Kaleba, S.Kaleba@kent.ac.uk, University of Kent, UK; Humphrey Burchell, H.Burchell@kent.ac.uk, University of Kent, UK; Stefan Marr, S.Marr@kent.ac.uk, University of Kent, UK.

2475-1421/2023/10-ART233
https://doi.org/10.1145/3622808

1 INTRODUCTION

Interpreters are often the first choice when implementing a new language and even established languages such as Java, JavaScript, Ruby, and Python rely on interpreters. This is because interpreters are relatively easy to implement, debug, and experiment with. Moreover, they can be memory efficient, and may execute code immediately, which becomes important when using a just-in-time compiler on applications with millions of lines of code.

With meta-compilation systems such as RPython [Bolz et al. 2009] and GraalVM [Würthinger et al. 2017], interpreters also became the main means of implementing a language when one wants to use existing just-in-time compilers, garbage collectors, multi-threading support, tooling [Van de Vanter et al. 2018], and language interoperability [Grimmer et al. 2018]. Being able to reuse the hundreds of person years of engineering effort that went into systems such as the HotSpot JVM, GraalVM, and V8 JavaScript, would benefit a large number of language projects.

While GraalVM, with its partial evaluation approach combined with just-in-time compilation, allows language implementations to reach peak performance that is comparable with V8 (see Section 5.3), it can not yet compete with custom-built systems when it comes to memory use and warmup performance. Given these memory and warmup issues, the use of abstract-syntax-tree (AST) interpreters for some of the language implementations on top of GraalVM has been criticized, because *"everyone knows that bytecode interpreters are faster."*

However, on meta-compilation systems, this generally accepted truth may not hold given the constraints and opportunities of such systems. To carefully study the trade-offs between interpreter designs, we implemented four interpreters: an AST and bytecode interpreter each on top of both RPython and GraalVM. The interpreters are as similar as possible to each other to facilitate direct comparison, while exploiting some minor performance opportunities where available (see Section 4.4). By investigating AST and bytecode interpreters on both RPython and GraalVM, our conclusions will more likely generalize to other meta-compilation systems, too.

The interpreters implement the SOM (Simple Object Machine) language [Haupt et al. 2010],[1] a class-based dynamic language sharing many commonalities with languages such as Python, Ruby, and Smalltalk (see Section 2.3). They implement many common interpreter-level optimizations, including polymorphic inline caching, inlining of control structures, self-optimization and bytecode quickening, supernodes and superinstructions, as well as optimizing object storage models and storage strategies for arrays. Together with evaluating the interpreters using the Are We Fast Yet benchmarks [Marr et al. 2016], the use of SOM allows us to study a common core shared by many languages, which makes it likely that our results generalize to other dynamic languages.

To assess the trade-offs between AST and bytecode interpreters, we compare interpreter performance as well as just-in-time-compiled peak performance. We find that AST interpreters are on par in run-time performance, and in some cases even moderately faster. Thus, bytecode interpreters do not have the generally expected advantage on top of meta-compilation systems. When looking at the overall memory use, we even find that bytecode interpreters have drawbacks because boxing avoidance is not as applicable, and their interpreter structure requires additional allocations. However, we can also confirm that bytecodes are more compact in memory, which is important for large codebases. But, this benefit narrows when the code is executed and collects profiling information.

For completeness, we also assess the warmup behavior and the performance impact of optimizations that improve AST or bytecode performance.

The contributions of this paper[2] are:

[1] https://som-st.github.io/

[2] A full version of this paper with appendix is available at https://kar.kent.ac.uk/102817/1/ast-vs-bc-paper.pdf

(1) the design of an experimental methodology to identify run-time performance and memory usage trade-offs between AST and bytecode interpreters
(2) the implementation of PySOM$_{\text{AST}}$, PySOM$_{\text{BC}}$, TSOM$_{\text{AST}}$ and TSOM$_{\text{BC}}$ to compare the interpreters on top of RPython as well as GraalVM
(3) an analysis that shows that bytecode interpreters cannot be assumed to be faster on top of current meta-compilation systems

2 BACKGROUND

This section sketches the techniques used for the AST and bytecode interpreters, meta-compilation, and SOM, i.e., the language used for our experiments.

2.1 Self-Optimizing AST Interpreters and Bytecode Interpreters

Interpreters are commonly distinguished based on the type of program representation they execute. The most common approaches include *abstract syntax trees* and *bytecodes*.

```
class Literal:
  def constructor(value):
    self.value = value
  def execute(frame):
    return self.value

class AddNode:
  def constructor(child1, child2):
    self.child1 = child1
    self.child2 = child2
  def execute(frame):
    return (child1.execute(frame)
          + child2.execute(frame))

tree = AddNode(Literal(1), Literal(2))
tree.execute(Frame())
```

(a) A basic direct-style abstract-syntax-tree interpreter. The `Literal` nodes represent values and the AddNode adds the results of two subexpressions.

```
bytecodes = [PUSH_LIT, 0,
             PUSH_LIT, 1, ADD]
literals = [1, 2]
stack = []
i = 0
while True:
  bc = bytecodes[i]
  i += 1
  switch bc:
    case PUSH_LIT:
      stack.push(literals[i])
      i += 1  # uses two bytes
    case ADD:
      val2 = stack.pop()
      val1 = stack.pop()
      stack.push(val1 + val2)
```

(b) A basic bytecode interpreter for a stack machine. The bytecodes encode pushing literal values onto the stack, and then add them with an ADD instruction.

Fig. 1. Pseudocode for interpreters adding the integers 1 and 2.

Abstract Syntax Tree Interpreters. An abstract syntax tree is a common program representation derived from the output of a parser. It represents the code structure, but omits details not relevant for a program's semantics, e.g., whitespace and grammar elements such as statement terminators.

Software engineering courses may introduce AST interpreters when discussing the visitor pattern [Gamma et al. 1994]. However, while vistor-style interpreters provides a lot of flexibility, e.g. to have a visitor for pretty printing, one for execution, and perhaps another for program analysis, the resulting performance when executing programs is not optimal.

For AST interpreters that aim for good performance, a more direct approach as sketched in Figure 1a is preferable. Each AST node type provides an `execute()` method, which implements the semantics of the language construct directly. For our addition example in Figure 1a, we have a

Literal node that represents, e.g., integer values, and an AddNode that implements the addition semantics by first evaluating its subexpressions and then computing the sum.

Bytecode Interpreters. Inspired by hardware instructions sets, bytecode sets are virtual instruction sets designed to be compact and linear program representations. An interpreter executes bytecodes by iterating over them one by one in a fetch-decode-execute cycle, at least conceptually.

Figure 1b illustrates the principle with a basic interpreter that can push values on a stack and add them. It shows a switch-case-style bytecode loop, where each case statement is a bytecode handler implementing the semantics of the bytecode.

Over the decades, bytecode interpreters have been highly optimized. The bytecode dispatch itself for instance is often optimized with direct or indirect threaded code [Bell 1973; Ertl and Gregg 2003], which aims to change the loop structure so that jumps to the next bytecode handler are at the end of each handler, increasing the likelihood that the CPU can correctly predict jump targets.

The machine design is also highly relevant for performance. Bytecode sets can choose a design on a spectrum between pure register and stack-based approaches [Shi et al. 2008]. While the example in Figure 1b is a classic stack machine with explicit stack operations, register-based designs may have performance benefits [Shi et al. 2008; Zhang et al. 2022]. By caching the top-of-stack values, one can design interpreters that are somewhere in-between the two approaches [Ertl 1995].

Selecting suitable instructions to be part of the bytecode set is performance-relevant as well. Taking instruction selection a step further gives us superoperators [Proebsting 1995], which are more commonly known as superinstructions [Piumarta and Riccardi 1998], combining multiple bytecodes into one, reducing dispatch overhead and enabling more optimal bytecode handlers.

To improve interpreter performance for dynamic languages, one can use run-time feedback to rewrite bytecodes, for instance based on the observed types [Brunthaler 2010a,b, 2021]. The basic operators such as addition often have complex semantics in dynamic languages. For instance, the ECMAScript specification[3] defines eight steps, which include possible method calls and type coercions. For such languages, quickening can replace a generic operation with a more specialized one, e.g., replacing a generic ADD bytecode with ADD_INT, which only handles integers but can fall back to the generic case when needed. This reduces the number of checks at run time and the code complexity of bytecode handlers, which may have secondary effects such as a more effective use of instruction caches.

AST versus Bytecode. It is standard practice that the first major optimization one can implement for an AST interpreter is to turn it into a bytecode interpreter. The linearized representation is more compact in memory than an AST, making iterating over bytecodes more efficient.

However, the work on self-optimizing AST interpreters [Würthinger et al. 2012] enables ASTs to benefit from run-time feedback, in a way similar to bytecode quickening. To optimize a complex and generic AddNode for a language such as JavaScript, the node would replace itself in the AST with one designed specifically for integer addition. In case something other than integers is added, it can then rewrite itself to the generic version to support the language correctly. Compared to bytecode quickening, self-optimizing AST interpreters have more flexibility. Since the AST is represented as a tree of node objects connected by pointers, rewriting can easily introduce a more complex structure if needed, for instance, to cache values. To simplify the implementation of such interpreters, GraalVM provides the TruffleDSL, which enables the concise implementation of many common patterns [Humer et al. 2014].

Many of the optimizations for such interpreters need to cheaply confirm a pre-condition at run time, for instance, that the values being added are integers. Qunaibit et al. [2018] perform a

[3] *Standard ECMA-262 Edition 5.1: The Addition operator*, Ecma International, https://262.ecma-international.org/5.1/#sec-11.6.1

high-level analysis that allows to combine such *guards*, i.e., pre-condition checks, into *megaguards*, which may reduce the need to repeatedly check the same precondition in an interpreter.

Not all classic interpreter optimizations are however directly useable when building on top of a meta-compilation framework, and Section 4.2 will detail the optimizations used in this study.

2.2 Meta-Compilation

While interpreters are easy to implement, many applications need just-in-time (JIT) compilers [Aycock 2003] to reach the desired performance. For many languages, one can likely build a baseline JIT compiler with moderate effort. However, highly optimizing JIT compilers, as known from Java and JavaScript virtual machines, take hundreds of engineering years of effort.

Meta-compilation reduces the individual implementation effort by making just-in-time compilers reusable and independent of any particular language. RPython and GraalVM are the main systems in this space. Both approach the problem by compiling the executing program together with the interpreter that is executing it. This combination provides the necessary specialization opportunities to produce efficient native code that rivals custom-built state-of-the-art JIT compilers.

2.2.1 RPython: A Tracing Meta-Compiler. While most JIT compilers choose frequently executed methods as their compilation units, *tracing* JIT compilers [Bala et al. 2000; Gal et al. 2006] use the *trace* of a concrete execution path as their compilation unit typically starting at a loop header or a method entry.

The RPython framework [Bolz et al. 2009; Bolz and Tratt 2013] uses tracing-based JIT compilation to enable the meta-compilation of interpreters. It is known for PyPy, a widely used Python implementation with JIT compilation. RPython interpreters use annotations to indicate for instance which loops are relevant for JIT compilation, which is then used by RPython to add a JIT compiler and garbage collection. While PyPy uses bytecodes, AST interpreters work also well on RPython.

To reach optimal performance, interpreter implementers can use annotations, typically to distinguish application-level from implementation-level loops. We would for instance mark application-level loops and the start of user functions as places where it is beneficial to start tracing. We can also indicate that a value in a trace is to be considered a constant, which allows subsequent optimizations. Other annotations include hints to unroll loops or to allow the compiler to assume that a function is side-effect-free.

From an engineering perspective, we found this approach to be highly effective [Marr and Ducasse 2015]. We were able to build an interpreter with comparably low effort and few code adaptations, the result being a language implementation with good peak performance. However, GraalVM, which we discuss next, provided better peak performance.

2.2.2 Graal: A Partial-Evaluation-based Meta-Compiler. Partial evaluation [Futamura 1971] uses partial input to derive an optimized program. The resulting program contains only computations that depend on the missing input. When the partial input is user code and our interpreter is the program to be partially evaluated, then the result is a compiled version of the user code, which can execute efficiently once the user input is provided. This is considered to be the first Futamura projection, effectively introducing the notion of meta-compilation.

GraalVM uses this approach to partially evaluate an interpreter for a user program, effectively just-in-time compiling it [Würthinger et al. 2017, 2013]. Instead of using tracing, the Graal compiler starts from methods and uses aggressive inlining to determine compilation units. This has the drawback that it cannot rely directly on the values of a concrete execution, as RPython can. Instead, GraalVM relies on interpreters to collect information about types, values, and taken branches while executing, and for them to encode it in the program representation, i.e., the AST, bytecode, and other profiling data structures. Interpreters would typically use self-optimizations, quickening, and similar forms of profiling. The Graal compiler will use the information during partial evaluation to compile an optimized program, eliminating most if not all of the interpreter code in the process.

GraalVM can be used either on top of the HotSpot JVM, or ahead-of-time-compiled with GraalVM Native Image [Wimmer et al. 2019]. The native image mode is similar to how RPython would typically be used, and results in a native binary of the interpreter.

In previous work [Marr and Ducasse 2015], we found that interpreters for partial-evaluation-base compilation required more engineering effort, since they need to collect profiling data explicitly. However, as mentioned, GraalVM also had higher peak performance compared to RPython.

2.3 Simple Object Machine (SOM)

Our experiments are based on the Simple Object Machine language (SOM), a class-based language and minimal Smalltalk dialect [Haupt et al. 2010]. It is dynamically typed, has class inheritance, closures, and non-local returns. It thus shares many implementation and especially optimization challenges of more commonly used languages such as JavaScript, Python, and Ruby, without reaching their complexity, which would make this study infeasible.

For this study, we adapted two existing SOM implementations PySOM and TSOM, which both support the full SOM language. PySOM is built on top of RPython and comes with a self-optimizing AST interpreter as well as a bytecode interpreter that supports quickening. TSOM is built on top of GraalVM and uses the TruffleDSL to implement a self-optimizing AST interpreter and has a custom bytecode interpreter that utilizes many of the self-optimization techniques, too.

All four interpreters reach peak performance similar to V8 using the Are We Fast Yet benchmarks [Marr et al. 2016] as we show in Section 5.3. Thus, the meta-compilation systems of RPython and GraalVM are effective, and the SOM implementations are fast enough to enable a study of trade-offs between bytecodes and ASTs in the context of such systems.

3 METHODOLOGY TO COMPARE AST AND BYTECODE INTERPRETERS

Our goal is to characterize the performance trade-offs of AST and bytecode interpreters in the context of meta-compilation systems. We focus on run-time performance and memory usage, since these are the two key aspects commonly named when preferring bytecode over AST interpreters. Since meta-tracing and partial-evaluation-based meta-compilation have different trade-offs [Marr and Ducasse 2015], we use both platforms to get a complete picture. Before detailing our SOM implementations in Section 4, we motivate the performance aspects we are going to compare. We will discuss our measurement methodology as part of the results in Section 5.

3.1 Interpreter Performance

The perhaps most important aspect for us is the run-time performance of the interpreters before just-in-time compilation. Here the interpreters have the highest impact on the observable performance.

For a wide range of workloads, the interpreter performance is crucial for the user experience. This includes highly interactive and often short-running applications such as scripts on websites, many short-running command-line scripts, large-scale applications with frequent redeployments many times a day [Ottoni and Liu 2021], and perhaps most important for the acceptance of new language implementations: the run time of unit test suites.

To assess interpreter performance, we compile the interpreters to stand-alone binaries without JIT compilation support. Both RPython and GraalVM are able to produce binaries with the interpreters ahead-of-time compiled for best interpreter performance.

3.2 Peak Performance

In the context of meta-compilation systems, we are also interested in the best possible performance that our language implementations can achieve. Thus, we want to assess the performance with JIT compilation enabled, and the benchmarks running long enough to be fully compiled.

This *peak performance* is especially relevant for long-running applications. However, even short-running applications may benefit when the executed code is compiled early on, and thus, can reduce the overall response time.

To assess the best possible, i.e., peak performance, we use both RPython and GraalVM with JIT compilation enabled. In the case of RPython, this is a native binary of the interpreter with the JIT compiler included as well. For GraalVM, we run TSOM on top of the HotSpot JVM using the Graal compiler for JIT compilation of the SOM programs. This configuration is recommended to gain the best performance overall, since for instance the garbage collector and other runtime elements of HotSpot are more sophisticated than in the ahead-of-time compiled binary we used to assess interpreter performance. However, using HotSpot may have a negative impact on warmup performance, because TSOM starts executing like any other Java program, which means first it is executed by HotSpot's Java interpreter and the TSOM interpreter code itself may be JIT compiled by HotSpot before the Graal compiler can JIT compile the running SOM program.

Both RPython and GraalVM rely on activation counters that once they reach a threshold trigger compilation. For RPython, we verified that the standard thresholds lead to a full compilation of the used benchmarks. The GraalVM's runtime system is more sophisticated, which however also makes it less predictable. To ensure full compilation, we adapted its settings by reducing the compilation threshold, disabling multi-tier compilation, and dynamic compilation thresholds.

While there is no guarantee that an arbitrary program can reach peak performance [Barrett et al. 2017], we adapted compilation settings and we execute the benchmarks long enough to ensure that all parts are compiled.

3.3 Warmup Performance

The phase between starting execution and reaching peak performance is the warmup phase.

In RPython, tracing and compilation happen on the same thread as the program execution. Thus, compilation stalls application progress and may be noticeable in the warmup behavior. RPython starts to trace and compile a loop after 1039 iterations. For functions, the threshold is higher at 1619 activations. This will give hot loops, which are likely more relevant for performance, a chance to compile before individual functions, typically resulting in better overall performance.

GraalVM supports background compilation, has a multi-tiered compiler, and uses a system of dynamic compilation thresholds, which is meant to carefully balance the compilation work with the resources needed by the application. In contrast to RPython, the background compilation ensures that the application can continue executing, while another thread compiles the function that reached the compilation threshold. To measure warmup on this system, we run both on top of Hotspot and the ahead-of-time compiled version. The goal here is to see whether the difference between AST and bytecode interpreters has an influence on the warmup behavior in either system. We also use the ahead-of-time compiled version, since here the interpreter performance is best, and has a higher impact during warmup.

3.4 Memory Usage

Since our AST interpreters represent programs as trees of objects, we need to assess the memory overhead compared to the bytecode interpreters. A bytecode interpreter encodes programs into an array of bytes, which avoids the cost of object headers and machine-word-sized pointers.

Consequently, bytecodes should be more compact. However, our interpreters also collect profiling data in form of lookup caches. TSOM's bytecode interpreter also uses the TruffleDSL-generated nodes for basic operations, which keep additional profiling information (see Section 4.4). Thus, in our bytecode interpreters designed for meta-compilation, the memory use of the program representation is not simply the number of bytes emitted by the bytecode compiler.

In addition to characterizing the difference between program representations, memory usage is also relevant in practice. When hosting applications in the cloud, memory is a direct cost. Moreover, since the program representation is subject to garbage collection, it may also have a secondary run-time cost by increasing the time garbage collection takes.

With meta-compilation and garbage collection in the mix, measuring all impact of the differences of the program representation is not straightforward. For instance, tracing and partial evaluation will be affected, and thus may use different amounts of memory.

To have a single consistent metric for both systems, we will use the maximum resident set size (RSS). RSS corresponds to the maximum heap size the language implementation requested. It thus is an overapproximation of all memory used by the interpreters. In programs that use large working memory, the impact of the program representation is naturally de-emphasized. However, for programs that load many lines of code, we will get a clearer picture of the cost of the program representation. Thus, while not precise, the maximum RSS reflects what users of a language implementation may care about.

Overall Impact. To get an impression of the high-level impact on memory usage, we measure the maximum RSS for the benchmark execution on the interpreter, as well as for peak performance. This will give us an intuition of the practical differences. However, especially for TSOM, the numbers are not only reflecting the difference in program representation, but also the difference between optimizations applied to the interpreters. As discussed in Section 4.2.1, TSOM_{AST} is able to perform boxing elimination to a higher degree than TSOM_{BC}, which is reflected in the memory usage.

Impact of the Program Representation. To more clearly measure the memory used by program representation, we need to dominate the overall memory use with it. Unit tests are a common extreme case for many language implementations, since much of the code is executed exactly once.

To measure the impact of the program representation, we thus compare the memory use between different sizes of unit test suites. Specifically, we measure the memory use of the interpreter for running 1 to 100 copies of the SOM unit tests. For 100 copies, this corresponds to 2,528 files with 197,756 LOC in total, excluding comments and blank lines. With this setup, memory use is dominated by the program representation. While the unit tests allocate memory during execution, these objects are garbage collected almost immediately, since the unit tests do not hold on to them.

A second version of this experiment merely loads the code. Thus, memory is allocated to represent the program, but neither run-time objects nor profiling data is allocated, providing a lower bound for code that is not executed.

With this approach, we can minimize the impact of other allocations, approximating the memory used to represent the program as AST or bytecode.

3.5 Impact of Optimizations

The optimizations we applied to the SOM interpreters fall into two broad categories: structural optimizations to essentially short-cut the generic language semantics in well-known cases, and optimizations that use run-time feedback and thus profiling to improve performance. This profiling can be beneficial even in the interpreter, or only for JIT compilation.

By assessing the performance impact of a number of optimizations, we want to gain a better understanding of how performance relates to the program representation and whether any possible additional memory use may translate into better run-time performance.

As a secondary goal, we also want to gain a better understanding of which optimizations are most important for performance, which may help fellow language implementers to focus their efforts in the context of meta-compilation systems and depending on whether they aim for an AST or bytecode interpreter.

However, measuring the impact of individual optimizations is nontrivial. Optimizations can interact in unpredictable ways. Some optimizations may even be relevant at the beginning, but become later less important because they are redundant in some way. Assuming the goal is overall best performance, we take our fully optimized interpreters and remove specific optimizations to identify their impact on the final performance of the systems. Thus, for each optimization under investigation, we have an interpreter that removes it completely to assess its impact.

3.6 The Are We Fast Yet Benchmarks

For the experiments, we will rely on the Are We Fast Yet (AWFY) benchmarks [Marr et al. 2016]. While they were designed for comparing the performance across languages, they are also known to exercise the common language abstractions of object-oriented languages. They include five macro-benchmarks, which behave similar to parts of applications, and nine micro-benchmarks, which focus on narrow aspects of runtime systems. This gives us sufficient breadth for the experiments.

Since they are designed for cross-language benchmarking, they also enable us to contextualize the performance by comparing to Node.js and OpenJDK. Their key limitation is that they are relatively small, and with most benchmarks, the results may not generalize to any particular application.

4 OVERVIEW OF THE SOM IMPLEMENTATIONS

The experimental subjects for this study are the four SOM interpreters that we will refer to for brevity as TSOM$_{\text{AST}}$, TSOM$_{\text{BC}}$, PySOM$_{\text{AST}}$, and PySOM$_{\text{BC}}$, distinguishing the abstract-syntax-tree (AST) and bytecode (BC) variants. Since TSOM is based on GraalVM using partial evaluation, and PySOM builds on RPython using meta-tracing, we cover both meta-compilation approaches.

In preparation of the experiments sketched in Section 3, we give an overview of the four interpreters, detail the optimizations, and discuss relevant differences between the implementations.

4.1 High-level Overview of the SOM Interpreter Implementations

For both TSOM and PySOM, the AST and bytecode interpreters are part of the same codebase, and the interpreter type can be chosen at compilation time. Since our experiments focus on the difference between the program representation, we aim to share as much as possible between the implementations that is independent of the program representation. This includes for instance the implementation of built-in functions such as loading of files and producing output. They also share the same object model and storage strategies as discussed in Section 4.2.5.

4.1.1 Abstract Syntax Tree. As previously mentioned, TSOM is built on top of the Truffle framework and uses the TruffleDSL to implement a self-optimizing AST interpreter. PySOM on the other hand, is implemented in RPython, which gives us a bit more freedom but also does not provide the conveniences of the TruffleDSL. In practice, this means we often keep the RPython interpreter simpler, but without sacrificing performance, as we will see in Section 5.2.

A basic AST interpreter for SOM would start out with the fine-grained AST nodes for the language features. This includes nodes for reading and writing method arguments, local variables, and object fields. It also includes nodes for reading global variables, literals, instantiating blocks, and sending messages, i.e., looking up and activating methods on a receiver object, as well as nodes for triggering and handling non-local returns. Each node has a single purpose and implements exactly one aspect of the language. This means, based on these fine-grained basic nodes, all programs can be represented. Back in Section 2.1 we sketched the `Literal` node in Figure 1a, which merely stores a literal values such as a number of string, and returns it when executed. A full list of nodes is given in the appendix.[4]

4.1.2 Bytecode. The original SOM interpreter had a bytecode set with merely 16 instructions[5] similar to the AST nodes mentioned above. This includes a bytecode instruction to duplicate the top of stack, different bytecodes read or write method arguments, local variables, and object fields. It also has bytecodes to push globals, literals, and new block objects. It has a bytecode for message sends with normal lookup semantics, and one with lexically-bound semantics to activate the method in the superclass. Local and non-local returns are the final two bytecodes.

To optimize the bytecode set, we introduced variants that directly encode certain parameters. To give two examples, pushing `self` on the stack, i.e., the first method argument is very common and thus, we introduced `PUSH_ARG0`. Pushing the `nil` value is also common, and done with a specialized bytecode `PUSH_NIL`. These specialized bytecodes are more compact, because they avoid the extra byte to encode the argument, and at the same time, they can be slightly more efficient, because for instance `PUSH_NIL` does not need to load the value from the literal array but pushes a constant directly to the stack. The full set of bytecodes is listed in the appendix.[4]

4.2 Optimizations Common to All Four Interpreters

We first compare optimizations present in all AST and bytecode interpreters, so that we may later analyze their individual effects in Section 5.6. Our general approach is to have the interpreters as similar as practical, which also includes optimizations.

4.2.1 Polymorphic Inline Caching. Hölzle et al. [1991] introduced polymorphic inline caches (PICs), which are still highly relevant today [Kaleba et al. 2022]. PICs store the lookup results at a given call site, avoiding the lookup cost in the interpreter, and for GraalVM, it enables inlining across guest-language methods. While this is not strictly needed for RPython with its meta-tracing, the PIC avoids unnecessary repeated lookups during tracing, too.

In both TSOM and PySOM, the maximum number of entries in the polymorphic cache is six. After this limit is exceeded, the call-site is considered megamorphic, the cache is discarded, and a lookup has to be done before each call.

4.2.2 Global Variable Caching. The SOM language has the notion of global variables, which are used for instance to represent class objects, but can also contain arbitrary values. Globals can be read like normal variables and are mutable via a hash-table-like reflective API. Furthermore, access to undefined globals triggers a reflective handler, which in the standard case will attempt to load a class with that name.

Since in most cases globals are only modified once, typically on loading a class or setting an initial value, we optimize them so that the compiler can treat them as constants, which is for instance important to efficiently instantiate objects. In case a global is mutated, compiled code using them will be invalidated to ensure correct semantics.

In the AST interpreter, access to globals is implemented with the `Global` nodes. For `true`, `false`, and `nil`, we have nodes with the corresponding value. For other globals, they start out in an uninitialized state, and rewrite themselves to a `CachedGlobal` after looking up the global.

[4] The full paper with appendix is available at https://kar.kent.ac.uk/102817/1/ast-vs-bc-paper.pdf

[5] https://github.com/SOM-st/som-java/blob/original/src/som/interpreter/Bytecodes.java

The bytecode is very similar and has `PUSH` operations. Though, we only added `PUSH_NIL` for literals, since `true` and `false` are very rare. For normal globals, `PUSH_GLOBAL` will rewrite itself to `Q_PUSH_GLOBAL`, which uses the result of the lookup.

4.2.3 Inlining Control Structures. Since SOM is a Smalltalk derivate, it uses classes and polymorphic methods to provide control structures. Conditional branches are realized as polymorphic methods named `#ifTrue:`, `#ifFalse:`, and `#ifTrue:ifFalse:` on the `True` and `False` classes, evaluating given lambdas when appropriate. Similarly, loops are methods on number and lambda classes. For instance, `Integer` has the `#to:do:` method, with the receiver being the lower bound, the first argument the upper bound, and the second argument the lambda representing the loop body.

While this design is a testament to the power of late binding, it makes achieving good performance more challenging. In PySOM and TSOM, we optimize control structures by inlining them on the AST or bytecode level. Thus, PySOM_{BC} and TSOM_{BC} will for instance translate calls to methods for conditional branches into bytecodes when they have literal lambdas, and inline the lambdas into the method with the call. This avoids method call overhead for the lambdas, and exposes the control flow more directly to the meta-compilers, which is expected to yield better performance. The bytecode interpreters support this inlining for the methods for conditional branches, the `#and:` and `#or:` methods on booleans, and while loops. The counting `#to:do:` loop was not included since it did not yield the desired performance. To support the inlining, we added the `JUMP` bytecodes listed in the appendix.[4]

The AST interpreters inline the same methods as the bytecode interpreters, but also `#to:do:`. During inlining, method call nodes are replaced by specific nodes for the control flow structures, e.g., an `InlinedIfElse` node (see appendix[4]). The lambda arguments are inlined into the calling method on the AST level, too.

4.2.4 Lowering of Basic Operations. As with control structures, basic operations such as arithmetic operations and comparison operators are provided as methods on classes. For example, the `Integer` and `Double` classes have methods for +, -, <, >= etc. Some of these methods are implemented as *primitives* directly in the interpreter. Others, including the >= methods, are implemented in the SOM code library as normal methods relying on primitive methods such as the one for <.

A basic SOM implementation would treat all these operations as normal polymorphic method calls, which is elegant but, because of the method call overhead, not particularly fast. PySOM and TSOM optimize these methods to avoid the call overhead, and they also provide primitive versions for methods implemented in SOM's core library to improve performance.

Since TSOM_{AST} is using the TruffleDSL, it also uses what we call *eager operations replacement*. Instead of relying on the polymorphic inline cache to resolve operations, we directly replace standard method call nodes for operations such as + and >= with AST nodes that specialize on argument types and implement the necessary semantics. This avoids the method call overhead and enables boxing elimination on the statement level TSOM_{AST}, because statements that work on integers can use code paths that are consistently typed for integers, without having to fall back on a representation that encapsulates the integer value in an object.

Since RPython does not have the TruffleDSL, and the eager replacement is not applicable to bytecode interpreters, $\text{PySOM}_{\text{AST}}$, PySOM_{BC}, and TSOM_{BC} do not use eager operations replacement. However, the extra flexibility of RPython allows us to handle activation of basic operations differently without extra overhead. Furthermore, all interpreters benefit from the primitive versions of SOM library methods.

4.2.5 Optimizations Orthogonal to Program Representation. PySOM and TSOM implement a number of other optimizations to reach the performance level of Node.js. Though, since they are orthogonal to the program representation, we will only briefly sketch the object model and use of storage strategies as the most important ones, and do not assess their performance impact.

Unboxed Number Storage in Object Fields. The object model takes inspiration from maps [Chambers et al. 1989] and the object model used by GraalVM [Wöß et al. 2014]. Its main benefit is unboxed storage of SOM integer and double values, which is especially beneficial in compiled code. Though, since SOM is a class-based language, the object model is simplified to avoid the need of a transition tree and reduce the problem of high map-polymorphism. Instead, each class tracks only the latest layout, i.e., mapping between logical fields and physical storage. Objects are transitioned to the latest layout lazily, avoiding a global impact when the layout changes.

Unbox Storage in Arrays. Based on the work on storage strategies [Bolz et al. 2013; Pape et al. 2015], PySOM and TSOM implement storage strategies for arrays. They currently support SOM integers, doubles, and booleans. Similar to the object model, this avoids storing values in their boxed form, which is especially beneficial in compiled code. In addition, they also support a strategy for empty arrays, i.e., arrays that store only `nil` values. In this case, the array only stores the length, which avoids allocating memory when not yet needed.

4.3 Optimizations Applied to a Subset of Interpreters

Because of the differences in implementation platform, and between AST and bytecode, the following optimizations are only available in some interpreters. We will detail key differences in Section 4.4. Note that superinstructions, quickening, and supernodes are nominally different optimizations. However, we apply them in a way that the resulting interpreters are as similar as possible to ensure fairness in the comparison.

4.3.1 Superinstructions. Superinstructions [Piumarta and Riccardi 1998], are combinations of multiple bytecode instructions into a single instruction that avoid dispatch overhead, and may have a more optimal implementation than the separate bytecodes could have.

In SOM, we use a few superinstructions that we identified as common based on bytecode profiles. This includes `INC` and `DEC` bytecodes, which increment or decrement the top of stack value, `INC_FIELD` and `INC_FIELD_PUSH`, which increment the value stored in a field, and possibly push the result onto the stack, as well as `RETURN_SELF` or `RETURN_FIELD_n`, which combines taking the `self` value or reading a field from `self` with returning either at the end of a method. Getters are perhaps the most common methods that end with reading a field.

As the above examples illustrate, superinstructions can be arbitrarily complex. For instance, `INC_FIELD_PUSH` would need to be expressed with a sequence of five bytecodes from the basic SOM bytecode set as can be seen in Figure 2.

```
PUSH_FIELD 0        # push the field value onto the stack
PUSH_CONSTANT 0     # push 1 by reading from the literal array at index 0
SEND  +             # send the + message to add the two values
DUP                 # duplicate the top of stack
POP_FIELD 0         # pop the top of stack, and write into the field
```

Fig. 2. Sequence of five basic bytecode instructions that is equivalent to the `INC_FIELD_PUSH` superinstruction.

4.3.2 Quickening. As mentioned in Section 2.1, quickening [Brunthaler 2010a] is a technique to incorporate run-time feedback into bytecode, which is especially useful for dynamic languages such as SOM. In PySOM_{BC} and TSOM_{BC}, we use it for instance for the method-call bytecodes.

The initial bytecodes initialize the polymorphic inline caches on first execution, and then rewrite themselves to quickened versions Q_SEND[_n], which merely use the caches. Similarly, as discussed in Section 4.2.2, we use quickening for access to global variables and introduced the Q_PUSH_GLOBAL bytecode. For these examples, quickening avoids run-time checks, and improve the performance of method calls and access to globals. In the AST interpreters, we use self-optimization [Würthinger et al. 2012], which is very similar to quickening in that it also relies on rewriting at run time, however, because of the tree structure of ASTs, it is more flexible.

4.3.3 Supernodes. This concept of a supernode [Larose et al. 2022] is our approach for replicating superinstructions in an AST interpreter. Thus, a supernode combines multiple AST nodes into one. The benefit over a superinstruction is the ability to merge arbitrary partial subtrees, giving a high degree of flexibility to parameterize a supernode. The main benefit is to reduce the number of node objects needed to express a program, which lowers memory use since we avoid a Java object header and a parent pointer. It also gives more opportunities for a compiler optimizing a node's code, and reduces the dispatch overhead among nodes.

We applied this optimization to TSOM$_{\text{AST}}$ and implemented supernodes based on performance-sensitive patterns identified in the interpreter. One of the supernodes we added is similar to the INC bytecode, the IntInc supernode. It is applied in situations where we have an addition of a literal integer to a value obtained from an arbitrary subexpression. This avoids the extra node for the literal value and specializes the addition to support only integers. It also avoids the virtual call to the execute() method of the literal node, as illustrated in Figure 3.

```
class IntInc:
  def constructor(value, child):
    self.child = child              # arbitrary subexpression
    self.value = value              # integer constant
  def execute(frame):
    return child.execute(frame) + value  # avoids one virtual call to execute()
```

Fig. 3. The IntInc node combines a Addition node, specializes it to integers, and stores an integer value for the addition. Handling of misspeculaltion and deoptimization is omitted for brevity.

Another set of supernodes fuses method call nodes with an argument read node for the receiver, e.g., with the BinaryArgSend node. We also added common comparisons with literal values, which simplifies the comparison logic since the literal value is statically known. Our supernodes for a few selected arithmetic expressions include squaring a value and squaring a value read and stored back into a field, which enables us to significantly reduce redundant type checking for this pattern.

4.4 Overview of Differences Between the Interpreters

Given the constraints and guidance provided by RPython and GraalVM with its TruffleDSL, there are differences between the interpreter implementations that may lead to different trade-offs.

GraalVM provides a more strict framework requiring the use of a predetermined calling convention between guest-language-level methods and a provided Frame abstraction that does type profiling of local variables, including boxing avoidance for primitive values. For the TSOM interpreters, this means calling between SOM methods has a relatively high overhead because an Object[] array needs to be allocated to pass method arguments, and the Frame structure needs to be constructed, which involves multiple objects and arrays.

Since RPython does not provide and require such a strict framework, we had more leeway, and represent method activations with a simple array. Similarly, there is no calling convention, which allowed us to specialize the calls to methods with one, two, and three arguments by providing explicit methods for it, and only requiring an allocation for an array to pass arguments when four or more arguments are needed, which happens much less frequently.

On the other hand, the TruffleDSL makes it straightforward to provide specializations for primitive values such as `long` and `double`, which enables us to avoid boxing of such values on the statement level in the TSOM_{AST} interpreter [Humer et al. 2014; Würthinger et al. 2012]. This enables nested expression nodes to pass temporary values between them as unboxed values.

In PySOM, we do not do any such specializations and all values are always boxed, i.e., stored in an object, except when storing a value in a SOM object or array (see Section 4.2.5).

The TSOM_{BC} interpreter does not benefit as much from avoiding of boxing since the bytecode structure does not directly benefit from the TruffleDSL-provided specializations, and temporary results of expressions are stored in an explicit stack, which is realized as an `Object[]` array, and thus, requires the boxing of `long` and `double` values. However, the TSOM_{BC} interpreter still uses the same node objects for basic operations that the TSOM_{AST} interpreter uses. This allows us to benefit from the profiling built into these nodes, to facilitate partial evaluation.

5 RESULTS OF COMPARING AST AND BYTECODE INTERPRETERS

We follow the high-level methodology outlined in Section 3 to identify the trade-offs between the different program representations. Before analyzing the results, we briefly sketch how we measure and which statistics we report.

5.1 Measurement Methodology and Reported Statistics

All experiments were run on a system with Ubuntu 20.04.6 LTS (kernel 5.4.0-146), with two 6-core Intel Xeon E5-2620 CPU at 2.40GHz, and with 16GB RAM. For the baseline comparisons, we use Node.js 17.9 and JDK 20. TSOM is based on GraalVM development branch. PySOM uses RPython from the version 7.3.11 release of PyPy.

We use ReBench [Marr 2023], in version 1.2, to configure the machine for benchmarking, and to collect the results. ReBench instructs Linux to use the performance governor, disables Turbo Boost, runs the benchmarks at the highest priority, and avoids scheduling other processes on the cores used for benchmarking (CPU shielding). This is intended to reduce the measurement noise.

To measure interpreter performance, just-in-time compilation is disabled, and we start the interpreter 100 times for a benchmark, and the benchmark harness measures how long one iteration of the benchmark takes.

We generally report the median run time factor over a baseline as well as the minimum and maximum to characterize the full range.

For measuring peak performance, the just-in-time compilers are enabled, and we start the interpreter once, and the benchmarking harness measures the time of 2000 iterations. Thus, all iterations execute in the same process, and benefit from the JIT compilation. We looked at the over-time performance of all benchmarks, and used the last 1000 measurements for the statistics. They are generally free from further compilation.

Since most experiments compare a set of benchmarks against a baseline, we report the median ratio over a baseline. The results are depicted as boxplots on a logarithmic scale, which leads to a more intuitive understanding of differences, since both a factor of 0.5 and 2 have the same physical distance from 1 on the resulting plot.

5.2 Interpreter Performance

As discussed in Section 3.1, we start by assessing the run-time performance of our interpreters with JIT compilation disabled. Figure 4 shows the results. We use Java as the baseline, as it performs best overall. However, most interpreters have at least one benchmark that is faster than on Java. For instance on Node.js, the Json parser benchmark takes only 33% of the time it takes on Java.

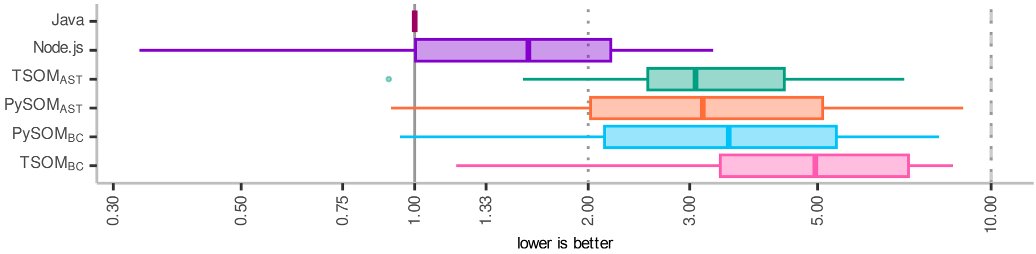


Fig. 4. Interpreter run-time performance, on a logarithmic scale, with Java as baseline. While TSOM and PySOM are overall slower than the Java and Node.js interpreters, we can also observe that PySOM$_{AST}$ and TSOM$_{AST}$ are faster than the bytecode versions. TSOM$_{BC}$ is the slowest interpreter overall.

When we focus on whether AST or bytecode interpreters on top of meta-compilation systems are faster, we see that the PySOM interpreters have very similar interpreter performance. The median slowdown of PySOM$_{AST}$ compared to Java is 3.16× (min. 0.91×, max. 8.94×), while the one for PySOM$_{BC}$ is 3.51× (min. 0.94×, max. 8.12×), and thus only slightly slower. However, the difference on top of GraalVM is larger for our interpreters. TSOM$_{AST}$ is 3.07× (min. 0.90×, max. 7.07×) slower than Java, while TSOM$_{BC}$ is 4.96× (min. 1.18×, max. 8.59×)

Based on these results, we conclude that AST interpreters can be on par with bytecode interpreters in run-time performance, and may even have an edge in some benchmarks and on some meta-compilation systems.

5.3 Peak Performance

Since one of the main reasons to use a meta-compilation system is to benefit from the JIT compilation, we look at the peak performance results in Figure 5. Here we omit the warmup phase, which we will come back to in Section 5.4.

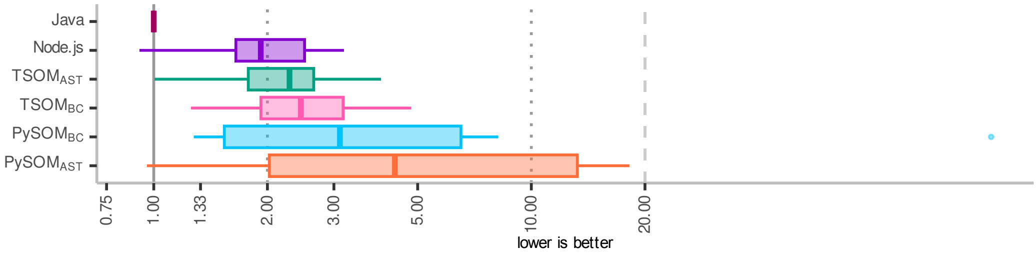


Fig. 5. Peak performance with just-in-time compilation enabled, a logarithmic scale, with Java as baseline. TSOM$_{AST}$ reaches the performance of Node.js, while the peak performance of the other implementations is a little further behind.

We use again Java as the baseline, since it has the best overall performance. The median run time of Node.js is about 1.92× (min. 0.92×, max. 3.19×) higher than Java.

From our implementations, TSOM$_{AST}$ is overall the fastest and takes about 2.29× (min. 1.00×, max. 4.00×) more time than Java, which is roughly at the same level as Node.js since the range over the benchmarks is well overlapping. TSOM$_{BC}$ is not far behind with 2.45× (min. 1.25×, max. 4.81×).

For PySOM, PySOM_{BC} is the fastest implementation, taking about 3.12× (min. 1.28×, max. 165.13×) more time than Java. This still overlaps well with Node.js, but has a few more benchmarks that do not quite reach the same level of performance. $\text{PySOM}_{\text{AST}}$ reaches 4.35× (min. 0.96×, max. 18.19×). A closer investigation shows that highly recursive benchmarks do not reach the same performance as on other systems. Here, RPython's trace-based compilation struggles to make the most optimal decision of when to stop tracing, which is a known issue.[6] While the bytecode version comes out faster, this result is dominated by a more suitable recursion inlining threshold.

Overall, there does not seem to be a clear benefit for either AST or bytecode interpreters in terms of peak performance. For TSOM, the results are arguably too close together, and for PySOM, the particularly slow recursive benchmarks make a strong conclusion seem undesirable. However, this in itself is a useful result. Even when focusing on peak performance, there does not seem to be a clear winner, which may allow us to make decisions in favor of one or the other program representation based on other factors.

5.4 Warmup Behavior During JIT Compilation

As discussed in Section 3.3, we collected and analyzed the warmup data for TSOM on Hotspot and the ahead-of-time-compiled interpreter, and PySOM. For brevity we only include plots for the ahead-of-time-compiled systems. However, the results are similar for TSOM on top of Hotspot.

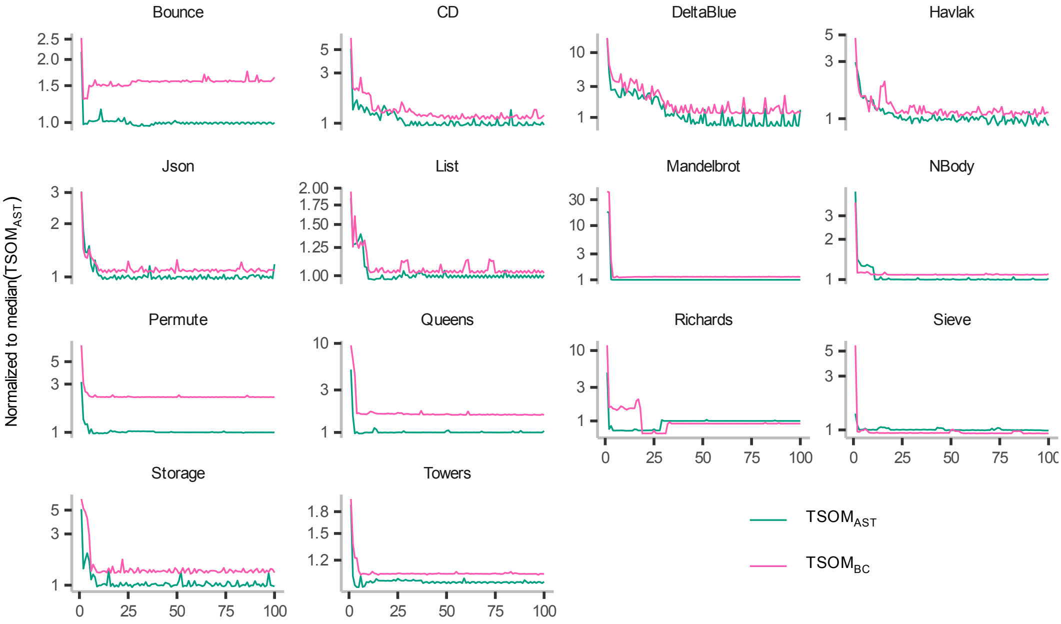


Fig. 6. Performance of benchmark iterations over time for the TSOM interpreters as measured on the ahead-of-time-compiled GraalVM Native Image. All data is normalized to the median of TSOM_{AST} and shown with a logarithmic y-axis. While we see performance difference between TSOM_{AST} and TSOM_{BC}, only the Richards benchmark seems to display a noticeable warmup delay on TSOM_{BC}.

We illustrate the data in Figure 6 and Figure 7, showing the iteration times for each benchmark normalized to the median value of the baseline. For consistency, we chose the AST interpreter as

[6] https://foss.heptapod.net/pypy/pypy/-/commit/928f810b36

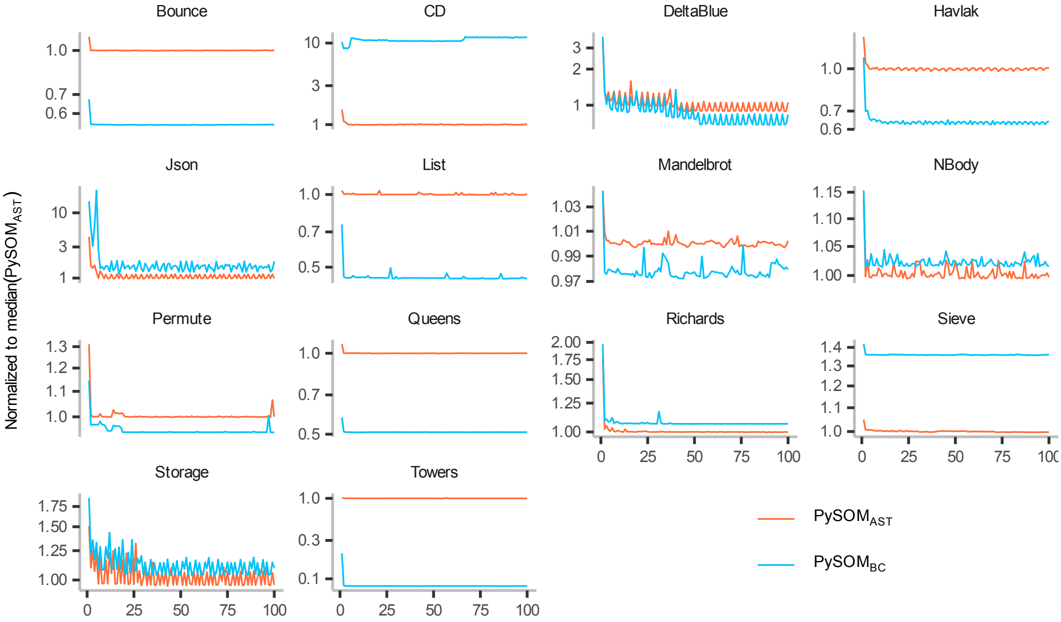


Fig. 7. Performance of benchmark iterations over time for the PySOM interpreters. All data is normalized to the median of PySOM$_{AST}$ and shown with a logarithmic y-axis. While we see performance difference between PySOM$_{AST}$ and PySOM$_{BC}$, there does not seem to be a major difference in warmup.

baseline. This approach preserves the over-time variations, while having a consistent *goal* value of 1 (or better), that implementations would want to achieve.

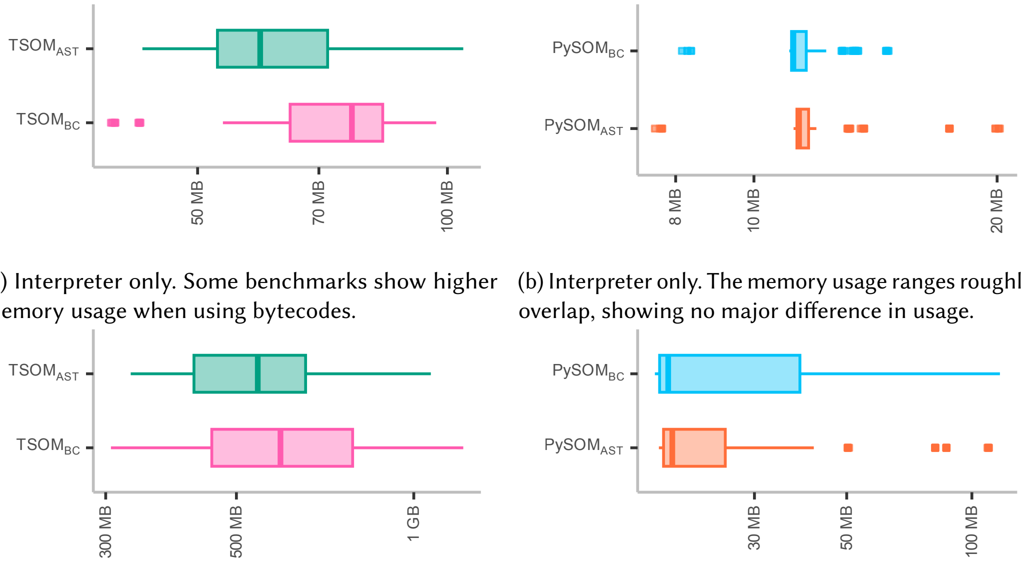
The main question we try to answer here is whether there is a notable difference in warmup behavior between AST and bytecode interpreters. As can be expected from the peak performance results (see Section 5.3), there are clear performance differences. However, we see neither for TSOM nor PySOM any systematic differences in warmup behavior. On TSOM$_{BC}$ in Figure 6, we see Richards has a longer delay before reaching the peak performance. However, none of the other benchmarks show it as pronounced. On PySOM in Figure 7, we may have more benchmarks that do not reach a similar performance, but this does not change after the shown 100 iterations either, and as such is not a warmup difference.

Based on the available data, we cannot identify any systematic differences. Though, warmup issues are typically more pronounced on larger applications, and the Are We Fast Yet benchmarks are not be large enough to trigger them.

5.5 Memory Usage

Besides run-time performance, memory use is the other major concern when it comes to program representation. As discussed in Section 3.4, we assess the overall impact on memory use, as well as on the program representation itself.

5.5.1 Overall Memory Impact. Figure 8 shows the maximum Resident Set Size (RSS) used by the benchmarks. We show the data separately for TSOM and PySOM, each for the interpreter and with JIT compilation. The numbers of the interpreter are expected to show a stronger effect if one is



(a) Interpreter only. Some benchmarks show higher memory usage when using bytecodes.

(b) Interpreter only. The memory usage ranges roughly overlap, showing no major difference in usage.

(c) With JIT compilation. Slightly higher memory use with bytecodes, but less than in the interpreter.

(d) With JIT compilation. The memory usage ranges rougly overlap, showing no major difference in usage.

Fig. 8. Maximum Resident Set Size (RSS) used by the interpreters, on a logarithmic scale.

there. The JIT compilers and their escape analysis may be able to reduce any additional allocation during optimization.

For TSOM, we can indeed see a difference in Figure 8a. The benchmarks on TSOM$_{BC}$ seem to use more memory. Figure 8c shows a similar trend, but not as clearly.

For PySOM, if there is any effect, the AST interpreter might use slightly more memory. However, this is much less clear from the data.

When investigating the allocation profiles on TSOM, we found that the bytecode interpreters allocate a significantly larger number of `Long` and `Double` values, because the TruffleDSL's approach to boxing avoidance does not translate directly to bytecode interpreters. We also noticed an increase in the number of allocations of `Object[]` arrays. Since we need to reify the stack for our stack-based bytecode, each method activation allocates an extra `Object[]` array, which for some benchmarks leads to a significant increase in allocations. The last significant difference is the allocation of `Object[]` arrays in the bytecode loop to pass arguments to method calls. In the AST interpreter these allocations happen inside of `execute()` methods, which in some cases allows the compiler to already avoid the allocation even in the interpreter. However, in the bytecode interpreter, this allocation happens in the bytecode loop, before passing it on to a polymorphic call site. This prevents compilers to avoid the allocation. Thus, on top of GraalVM, the AST interpreters may have some benefits in terms of memory usage, because boxing avoidance and code structure can lead to fewer allocations.

On PySOM, we do not see such effects because we do not have statement-level boxing avoidance and the SOM method calling convention does not require an array allocation for calls with fewer than four arguments, which are the most common ones.

While investigating the allocation behavior in detail, we also looked at precise memory allocation behavior. Though, the results were too similar to include and with garbage collection in the mix, maximum RSS seemed to be the more practically relevant metric.

5.5.2 Memory Usage for Program Representation. As detailed in Section 3.4, we approximate the differences in memory used by the program representation by using 1 to 100 copies of the SOM unit tests. One version of the benchmark executes the unit tests, covering every line, and thus, allocates memory to store profiling information, polymorphic inline caches, etc. However, it of course also allocates memory for the necessary run-time objects. Though, the run-time objects are garbage-collected almost immediately, since the unit tests do not hold onto them. Thus, the program representation dominates memory use more and more with an increasing number of copies of the unit tests. A second version of this benchmark merely loads the code. Thus, memory is allocated to represent the program, but neither run-time objects nor profiling data is allocated, providing a lower bound for code that is not executed.

With this approach, the AST on PySOM requires about 2.55× more memory than the bytecode when it is not executed and about 2.30× when it is executed. For TSOM, we see 2.89× more memory use for the AST when it is not executed, and only 1.21× more when it is executed. As mentioned in Section 4.4, the bytecode interpreter reused the node objects of the AST interpreter to benefit from the profiling they provide. The size of these nodes is comparably large, and can explain the increase in memory use during execution in the TSOM_{BC} interpreter, which we do not see in PySOM_{BC}, because there we do no use this technique. Thus, with increased run-time profiling, the difference between bytecode and AST interpreters in terms of memory use further narrows.

5.6 Impact of Optimizations

This section assesses the impact of our optimizations from Section 4.2 to better understand how they interact with the choice of program representation. This analysis is intended to help language implementers to prioritize optimizations.

5.6.1 Impact on Interpreter Performance. Figure 9 shows the impact of the various optimizations on interpreter performance, without JIT compilation. It shows how the run time increases when specific optimizations are removed.

Inlining of control structures (see Section 4.2.3) and inline caching (see Section 4.2.1) have the highest impact overall. On TSOM_{AST}, removing the inlining of control structures increases the median run time by 3.74× (min. 1.51×, max. 5.40×). On PySOM_{BC}, its impact is lower but still the most important one at 1.66× (min. 1.19×, max. 2.52×).

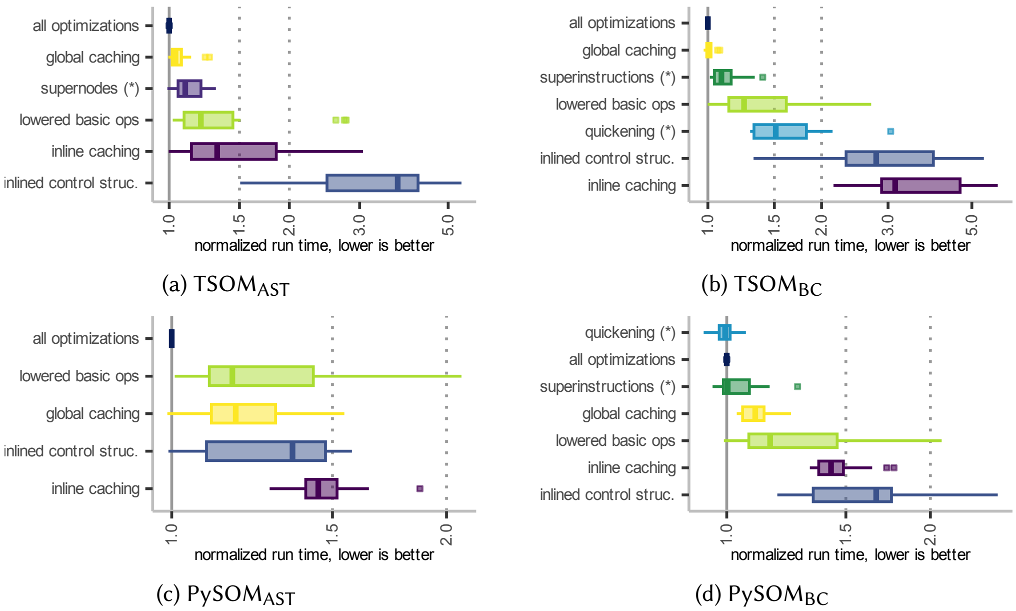
On TSOM_{BC} and $\text{PySOM}_{\text{AST}}$, inline caching is more important. On TSOM_{BC} it gives 3.13× (min. 2.15×, max. 5.86×) and on $\text{PySOM}_{\text{AST}}$ 1.45× (min. 1.28×, max. 1.87×).

As can be seen with the difference between PySOM and TSOM, inlining is more beneficial when the cost of method activation is high, which is the case for TSOM.

The caching of globals is not of high importance in the TSOM interpreters, but will be more important when we look at just-in-time-compiled performance.

Supernodes and superinstructions give roughly similar benefits on TSOM_{AST} and TSOM_{BC} respectively, which is to be expected, since they reduce the dispatch overhead between operations and open up a similar optimization potential. However, they are highly targeted and thus only give a modest gain.

When optimizing interpreter performance, we would recommend to start with inline caching, which is the most widely applicable optimization and highly important for large-scale applications. Afterwards, lowering of basic operations will give good benefits and can target specific benchmarks



(a) $\mathrm{TSOM}_{\mathrm{AST}}$

(b) $\mathrm{TSOM}_{\mathrm{BC}}$

(c) $\mathrm{PySOM}_{\mathrm{AST}}$

(d) $\mathrm{PySOM}_{\mathrm{BC}}$

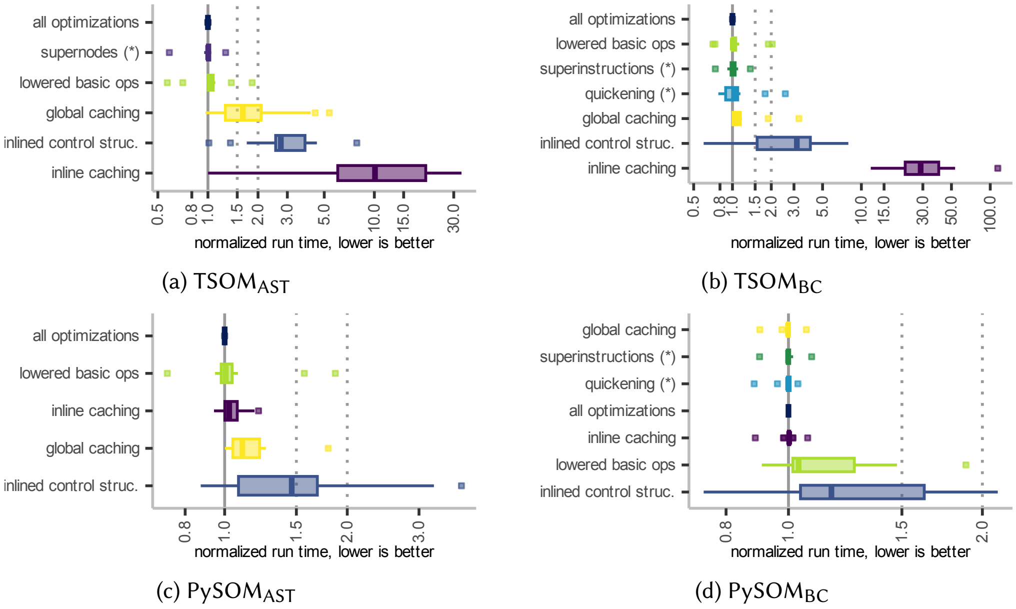
Fig. 9. Performance impact of individual optimizations on the interpreter-only speed of the fully optimized interpreters. Run time normalized to the implementation with all optimizations, on a logarithmic scale. Optimizations marked with (*) are specific to AST or bytecode interpreters. Overall, inlining of control structures, inline caching, and lowering of basic operations give the highest benefit.

of interests. Finally, inlining of control structures will also be important for interpreter performance. However, it can come with significant implementation complexity.

5.6.2 Impact on Peak Performance. Before looking into the impact of specific optimizations on peak performance, we can observe in Figure 10 that some optimizations are less important for it than others. Furthermore, one difference to the impact on interpreter performance is that caching of globals makes a good difference on TSOM for specific benchmarks, because with it the compiler can assume the value to be a constant, which enables it to optimize more. Also on TSOM, the lowering of basic operations, supernodes, superinstructions, and even quickening have a less noticeable impact, because the compiler can perform similar optimizations.

Similar to the interpreter performance, inline caching is important on TSOM. Removing it from $\mathrm{TSOM}_{\mathrm{AST}}$ results in 10.09× (min. 1.00×, max. 33.39×) increase in run time. On $\mathrm{TSOM}_{\mathrm{BC}}$, it results in 28.96× (min. 11.83×, max. 114.38×), making it the most important optimization. This is because inline caching enables the GraalVM's partial evaluator to inline across SOM methods, which is important to enable subsequent optimizations. This is in stark contrast to PySOM, where we see a difference of 1.03× (min. 0.94×, max. 1.21×) on $\mathrm{PySOM}_{\mathrm{AST}}$, and 1.00× (min. 0.89×, max. 1.07×), on $\mathrm{PySOM}_{\mathrm{BC}}$. Since RPython uses meta-tracing, inlining is done implicitly, and inline caching is not needed to enable it.

On PySOM, the most important optimization is the inlining of control structures. On $\mathrm{PySOM}_{\mathrm{AST}}$ it results in 1.46× (min. 0.87×, max. 3.81×) on $\mathrm{PySOM}_{\mathrm{AST}}$, and on $\mathrm{PySOM}_{\mathrm{BC}}$ it gives 1.17× (min. 0.74×, max. 2.11×). The inlining helps to reduce the overall trace length and makes optimization opportunities usually more evident to the compiler. However, on top of $\mathrm{PySOM}_{\mathrm{BC}}$ we do see also some benchmarks slowing down when applying the optimization. In the cases we observed, the



(a) TSOM_{AST}

(b) TSOM_{BC}

(c) $\text{PySOM}_{\text{AST}}$

(d) PySOM_{BC}

Fig. 10. Performance impact of individual optimizations on the peak performance of the fully optimized interpreters. Run time normalized to the implementation with all optimizations, on a logarithmic scale. Optimizations marked with (*) are specific to AST or bytecode interpreters. Inline caching gives the highest benefit on TSOM, because there it is needed for inlining across SOM methods. PySOM benefits more from inlining of control structures.

inlining in the bytecode leads to the compiler not having as precise information on the lifetime of variables, which can lead to extra boxing. Here, we notice that the AST structure and the use of the host-language stack communicates such details more precisely. The reified stack in the bytecode interpreter on the other hand can lead to such details being obscured. This effect is also noticable on TSOM_{BC} where removing inlining of control structures results in 3.16× (min. 0.60×, max. 7.93×). increase of run time. While generally beneficial, there are still some slowdowns observable.

Overall, we conclude from this analysis that there is a stronger difference between meta-tracing and partial evaluation than the choice of program representation. However, it remains to be noted that in some cases the explicit reified stack in a bytecode interpreter can obscure variable lifetimes, which makes it harder for a meta-compiler to yield optimal code.

6 RESULT DISCUSSION

In the following discussion, we will reflect on the results, their generalizability, the limitations of this study, and our perception of engineering differences.

6.1 Results and their Generalizability

Results. One might argue that the results are somehow inconclusive. However, this also means that in the context of meta-compilation systems there is no overwhelming evidence to prefer bytecode-based interpreters as one would expect based on folklore. Instead, our results provide some evidence that AST interpreters can be competitive.

Specifically, in Section 5.2 we have seen that the pure interpreter performance of AST interpreters can be better than that of their bytecode counterparts. The peak performance (Section 5.3) is also generally on par. Currently, the memory use is perhaps the most significant difference. There are some cases where the interpreter structure of bytecode interpreters has drawbacks. Here, techniques such as multi-level quickening [Brunthaler 2021] may provide solutions. On the other hand, it is still true that bytecodes are significantly more compact (see Section 5.5.2) than ASTs. Especially for established languages with large codebases, this can be a dominating factor. For new languages, perhaps in the context of research, small applications, or scripting, AST interpreters do not only seem to be *good enough*, but may have benefits over bytecode interpreters.

Generalizability Beyond SOM. As mentioned in Section 2.3, the SOM language shares the typical implementation challenges with other major dynamic languages. Specifically, it has objects, classes, collections, lambdas, and exception-like constructs. Furthermore, the Are We Fast Yet benchmarks [Marr et al. 2016] exercise specifically this common core. As such, the experimental setup is designed to evaluate the aspects common to a wide range of languages. With this, we believe our results to generalize to a wide range of other languages of a similar kind, which includes but is not limited to JavaScript, Python, and Ruby.

Generalizability to Other Meta-compilation Systems. Since this study is done in the context of meta-compilation systems, there are restrictions in the type of optimizations that are supported, as well as the code patterns the compilers are optimized for. For instance, bytecode interpreters currently cannot immediately benefit from optimizations such as threaded code [Bell 1973; Ertl and Gregg 2003]. While work is underway [Izawa et al. 2022] to enable language implementers to benefit from more such optimizations, it seems unlikely that one will have the same freedom as for completely custom-built language implementations. Thus, custom-built implementations may continue to use interpreters implemented in assembly, or assembly-level languages to manually optimize for the best possible performance. While this seems to be practical for industry projects such as V8 and JavaScriptCore VMs, and even research projects such as the Wizard WebAssembly interpreter [Titzer 2022], the benefits of meta-compilation systems are the overall reduction of engineering effort.

By exploring our research questions on top of two very different meta-compilation systems, our results are likely to generalize beyond these two specific systems. On the one hand, there is the difference in compilation approach. While RPython uses trace-based compilation, GraalVM uses method-based compilation, both having very different tradeoffs in terms of performance and engineering characteristics [Marr and Ducasse 2015]. On the other hand, they differ drastically in the degree of support they provide and the sophistication of the runtime system. While RPython uses a minimalistic approach with a few annotations, GraalVM uses the Truffle framework and TruffleDSL to enforce a much more strict way how languages are to be implemented. Similarly, the RPython system has a relatively straightforward incremental garbage collector, while GraalVM can use the HotSpot JVM and any of its garbage collectors. Since RPython compiles to C, our RPython interpreters are closer to system-language interpreters albeit with the restrictions of RPython. Thus, both systems arguably differ significantly from each other, which means our results are likely to generalize to other meta-compilation systems, too.

For meta-compilation systems, the results suggest that there may be further room to improve them for AST as well as bytecode interpreters. AST interpreters could benefit from techniques for more compact representations that more closely resemble bytecodes [Koparkar et al. 2021; Pape 2021; Pape et al. 2017; Vollmer et al. 2017]. Of course, taking the other perspective, support for bytecode interpreters could be further improved for instance with support for boxing avoidance as provided by the TruffleDSL [Humer et al. 2014] or multi-level quickening [Brunthaler 2021].

For meta-compilations, we believe that our results are transferable to other language implementations on the studied systems. To the best of our knowledge, TSOM and PySOM are currently the fastest interpreters on these platforms by a considerable margin.

Generalizability to System-level Interpreters. It is unclear to us whether parts of our results may generalize to interpreters implemented in system-level languages, i.e., C/C++, assembly, and perhaps Rust. Generally, we assume these languages to have more control over data layout and control flow, which results in a different performance tradeoff space.

We assume that more control over the code and code layout allows one to design interpreter loops that fit better into instruction caches, which can have a major impact on dispatch cost and might be much harder to achieve with an AST interpreter. While we can imagine more compact representations for ASTs, possibly in a partially serialized form [Koparkar et al. 2021; Vollmer et al. 2017], it is unclear that one can match the compactness of a well-designed bytecode set without arriving at a point in the design space that also reduces the flexibility in dynamic rewriting, removes the tree structure, or hinders co-location of instruction and caches. Thus, at the system level, bytecode interpreters may benefit from more compact representation and interpreter loop, which benefit data and instruction cache locality by avoiding pointer chasing in data structures and unpredictable control flow jumps.

6.2 Limitations of this Study

As discussed in Section 4.4, there are differences between the interpreters and their optimizations. These differences may have an impact on the accuracy and precision of our comparisons. While we believe that the most significant differences are related to the different underlying systems, i.e., GraalVM and RPython, there are also differences in the degree of optimization applied to AST and bytecode interpreters. Overall, we believe that we reached a similar level of optimization on all systems, enabling us to draw conclusions from this study. However for completeness, we will reiterate the main differences as limitations.

Boxing Avoidance. One of the major differences between TSOM_{AST} and TSOM_{BC} is the degree of boxing avoidance realized in the interpreter. The AST interpreter is able to do boxing avoidance for full statements, relying on the tree structure, which specializes itself to code that avoids boxing of `long` and `double` values. In the bytecode interpreter, this is not currently possible in the same way. The linear structure of bytecode does not provide the context for this optimization and the stack being reified as a `Object[]` array leads to boxing between bytecodes. Using more quickening [Brunthaler 2010a, 2021] and an additional stack for unboxed values may help to improve this and reduce some of the memory overhead seen in Section 5.5.1. However, we also saw the cost of the reified stack itself in the memory cost. Thus, boxing is not the only factor here, as the overall structure of bytecode interpreters also had an impact. We believe this difference may only lead us to slightly overestimating the difference between TSOM_{AST} and TSOM_{BC} on interpreter-only performance.

In the context of the overall study, we also believe this difference does not have a major impact since neither $\text{PySOM}_{\text{AST}}$ nor PySOM_{BC} do boxing avoidance, and thus, we have a consistent comparison along this axis.

Degree of Run-time Profiling. The TSOM interpreters both have a high degree of run-time profiling. This includes profiling the activated specialization in nodes as well as active branch and value profiling. Both TSOM_{AST} and TSOM_{BC} do almost the same amount of profiling. There are only minor differences caused by nodes that are only used in TSOM_{AST} since they are not applicable to the bytecode interpreter, because the operations are expressed as bytecodes. Given that the peak performance is fairly similar (see Section 5.3), we believe there is no fundamental issue.

PySOM does not do the same style of run-time profiling. While it does self-optimization and quickening in the interpreter, meta-tracing uses concrete values from the trace, which means we do not need to collect profiling information in the interpreter.

Bytecode-level Optimizations and Supernodes. Bytecodes and superinstructions were identified in a step-wise process based on profiles of bytecode sequences and run-time profiles of benchmarks. As such, the ones implemented are the ones that gave the highest likelihood for performance improvements. We used the same approach on AST nodes, identifying common patterns to construct supernodes.

We could have further expanded the bytecode set and increased the number of implemented supernodes. Similarly, we could continue to identify further opportunities for bytecode quickening and also add supernodes to $\text{PySOM}_{\text{AST}}$. Given that $\text{PySOM}_{\text{AST}}$ already outperforms PySOM_{BC}, any impact on the overall conclusions is likely not to be significant. We believe that overall, further performance gains on the used benchmarks would be minimal, because the overall benefit of these optimizations is limited as we saw from the results in Figure 9.

Our experience with larger codebases also suggests that any performance improvement potential is thwarted by the cost of method calls, which seem to dominate larger applications in the interpreter.

6.3 Engineering Tradeoffs

Since bytecode interpreters are traditionally considered to be the optimal solution when one wants to reach good performance, engineering considerations are typically relegated to a secondary concern. However, our results indicate that AST interpreters can be competitive and thus, we will briefly highlight the engineering tradeoffs that we noticed in this work.

With self-optimizing AST interpreters, the most important benefit is likely the use of the host-language stack. This means the interpreter code is much closer to regular application code. In contrast, a bytecode interpreter reifies the stack in its own data structure, which adds the complexity of reasoning about a stack machine and implies a tail of bugs around stack balancing issues. Furthermore, it requires code for stack printing or visualization to enable debugging, where we can rely on the standard tooling for an AST interpreter. This benefit should not be underestimated, especially when it comes to getting someone new to language implementations to start working on such an interpreter.

A similar benefit for reasoning is that AST nodes localize state more. In the object-oriented programming style, the encapsulation provides a clear separation simplifying reasoning. In contrast, profiling information in the bytecode interpreter needs to be stored in data structures that are accessed by a wide range of different bytecodes and thus, the interaction pattern is more complex and requires global instead of local invariants to be maintained.

When it comes to designing optimizations, as argued by Würthinger et al. [2012], the tree structure also provides additional flexibility. To optimize bytecode in similar ways, we may need to implement a pass over the bytecode to recognize a specific pattern, while in an AST the tree structure can make it trivial. An example would be the call to a built-in function with a constant and an argument for which we can speculate on a specific type. This tree structure gives us the opportunity to optimize something like addition, i.e., concatenation of strings right there. In a bytecode interpreter, the linearization removes this structure, making the optimization require a pass over the bytecodes or another non-local mechanism. However, the reverse can also be true, and the linearized structure may facilitate other optimizations, for instance in terms of simple peephole optimizations of instructions that are commonly adjacent. Thus, in the end, as with other intermediate representations, different representations can be beneficial for different sets of optimizations. However, for the typical set of optimizations that provide easy wins early on, the AST structure tents to be a good starting point. The authors personally also find AST interpreters more pleasant to work with.

7 RELATED WORK

We are not the first to compare between AST and bytecode interpreters. Most closely related to our work on top of meta-compilation systems is work by Niephaus et al. [2018] and Kalibera et al. [2014]. Niephaus et al. [2018] implement GraalSqueak and compare an AST with a bytecode-inspired interpreter on top of GraalVM. GraalSqueak is also a Smalltalk, which they evaluate with a prime number sieve and Fibonacci benchmark. However, what they call bytecodes does not leverage the benefits of a compact representation, because each of their "bytecodes" is represented by a node object which requires at least a Java header, typically 16 bytes, and a parent pointer of 4 or 8 bytes. Our work here uses a true bytecode encoding where each instruction is represented with one or two bytes (see Section 4.1). Furthermore, we compare the interpreters not just on top of GraalVM, but also RPython, and use the more comprehensive Are We Fast Yet benchmarks. Many of the mentioned GraalVM-specific optimizations are common to languages using the Truffle framework, and we apply them, too.

Kalibera et al. [2014] implemented FastR, an AST interpreter for a subset of the R language, and noted that they achieved better performance than the GNU R bytecode interpreter. While it is another data point suggesting the potential of AST interpreters, it is a comparison between worlds, i.e., across different systems and implementation languages with many confounding factors. Our comparison removes these factors and can examine the differences between program representations, since we have AST and bytecode interpreters in the same system. That being said, we apply many of the techniques they recommend in Kalibera et al. [2014] section 4. For instance, we apply optimizations as soon as possible on AST and bytecode to gain performance. We utilize aggressive specializations, cache lookups, and use storage strategies to specialize data structures to reach state-of-the-art performance.

Körner et al. [2021] compared the performance of AST and bytecode interpreters in the context of Prolog, and found AST interpreters to be more efficient in many cases, also noting their ease of implementation compared to bytecode. One aspect here is that Prolog is more amenable to being represented as an AST due to its recursive execution model, and attests to the strength of the more natural representation of a tree when compared to the linear nature of bytecode. These types of thoughts are not new. D'Hondt [2008] asked whether we should consider bytecodes to be an atavism, however, without a clear answer with respect to performance.

In the context of meta-compilation systems, we studied the difference between meta-tracing and partial evaluation before [Marr and Ducasse 2015]. While we still use GraalVM and RPython as the only practical meta-compilation systems, we now focused on the question of program representation.

8 CONCLUSION AND FUTURE WORK

Meta-compilation systems enable us to implement languages based on interpreters, while components such as just-in-time compilation and garbage collection are provided by the system. This means language implementers benefit not just from reduced engineering effort, but also all the know-how around interpreters. Part of this folklore is that bytecode interpreters are faster and more memory efficient than abstract-syntax-tree interpreters.

In our study, we showed that this is not generally the case on top of meta-compilation systems. Instead, AST interpreters are on par, and in some cases even faster than bytecode interpreters, *with and without* meta-compilation. We showed this based on PySOM and TSOM, for which we built both an AST and bytecode interpreter on top of both RPython and GraalVM. We evaluated performance and memory use based on the Are We Fast Yet benchmarks [Marr et al. 2016].

While we found that both AST and bytecode interpreters can be on par performance-wise, we could also confirm that bytecodes are, as expected, a more memory-efficient program representation. However, as soon as run-time profiling information comes into the mix, the difference becomes smaller. Furthermore, with the structure of bytecode interpreters, they may need more run-time memory allocations, which can have a negative impact on overall memory use. We also investigated the performance impact of common optimizations, which can help language implementers to prioritize their implementation.

We believe our results show that AST interpreters should not be so readily discounted when it comes to implementing efficient and fast language implementations. While bytecodes can be more compact, AST interpreters have engineering benefits such as direct utilization of the host-language stack, more localized and encapsulated state, and their tree structure facilitates early optimizations that can make them a suitable choice for some implementations.

In the future, we should reconsider self-optimizing AST interpreters also for languages implemented at the system level in C/C++ or assembly. Since these bytecode interpreters can benefit from threaded interpretation and other common bytecode interpreter optimizations, it would be good to confirm whether these techniques are still making bytecodes the better choice in terms of performance. Körner et al. [2021]'s work is an encouraging step in this direction.

Meta-compilation systems could also benefit from further improvements. Using techniques proposed by Koparkar et al. [2021], Pape et al. [2017], Pape [2021], and Vollmer et al. [2017] may enable more compact representations of ASTs that can still be interpreted and benefit from self-optimization. The work of Humer and Bebić [2022] on *Operation DSL* is also promising. It aims to make the implementation of bytecode interpreters on top of GraalVM as convenient as AST interpreters. Such an Operation DSL could solve the issues we observed around boxing avoidance and may enable the integration of multi-level quickening [Brunthaler 2021].

While we were not able to measure warmup differences with the benchmarks we used, we know that bytecode interpreters can suffer from longer compilation times on top of GraalVM. The partial evaluation system will first unroll the bytecode loop and duplicate it as many times as a method has bytecodes, and inline all called methods aggressively. This results in huge compiler graphs and is a known limitation with the current system. Future work should investigate how to effectively interleave classic optimizations with partial evaluation to minimize the compiler graph and the corresponding compilation time.

GraalVM is currently also limited in the optimizations it can apply to bytecode interpreters since the GraalIR only supports structured control flow. Thus, creative solutions to e.g. enable effective threaded interpretation would be needed.

DATA-AVAILABILITY STATEMENT

Our experimental setup to identify run-time performance and memory usage tradeoffs between AST and bytecode interpreters is available on Zenodo [Larose et al. 2023]. The artifact contains the PySOM and TSOM interpreters as well as the scripts and documentation to reproduce the figures in Section 5 and add new experiments.

ACKNOWLEDGMENTS

We thank the reviewers for their useful feedback. We also thank Nicolas Polomack for his contributions to the SOM language.

This work was supported by a grant from the Engineering and Physical Sciences Research Council (EP/V007165/1) and a Royal Society Industry Fellowship (INF\R1\211001).

REFERENCES

John Aycock. 2003. A Brief History of Just-In-Time. *ACM Comput. Surv.* 35, 2 (June 2003), 97–113. https://doi.org/10.1145/857076.857077

Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00).* ACM, 1–12. https://doi.org/10.1145/358438.349303

Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 52 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133876

James R. Bell. 1973. Threaded Code. *Commun. ACM* 16, 6 (June 1973), 370–372. https://doi.org/10.1145/362248.362270

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the Meta-level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09).* ACM, 18–25. https://doi.org/10.1145/1565824.1565827

Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. 2013. Storage Strategies for Collections in Dynamically Typed Languages. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13).* ACM, 167–182. https://doi.org/10.1145/2509136.2509531

Carl Friedrich Bolz and Laurence Tratt. 2013. The Impact of Meta-Tracing on VM Design and Implementation. *Science of Computer Programming* 98 (2013), 408–421. https://doi.org/10.1016/j.scico.2013.02.001

Stefan Brunthaler. 2010a. Efficient Interpretation Using Quickening. In *Proceedings of the 6th Symposium on Dynamic Languages (DLS'10, 12).* ACM, 1–14. https://doi.org/10.1145/1899661.1869633

Stefan Brunthaler. 2010b. Inline Caching Meets Quickening. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Vol. 6183. Springer, 429–451. https://doi.org/10.1007/978-3-642-14107-2_21

Stefan Brunthaler. 2021. Multi-Level Quickening: Ten Years Later. *CoRR* abs/2109.02958 (2021), 47 pages. arXiv:2109.02958 https://arxiv.org/abs/2109.02958

Craig Chambers, David Ungar, and Elgin Lee. 1989. An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA).* ACM, 49–70. https://doi.org/10.1145/74878.74884

Theo D'Hondt. 2008. Are Bytecodes an Atavism? In *Self-Sustaining Systems*, Robert Hirschfeld and Kim Rose (Eds.). Lecture Notes in Computer Science, Vol. 5146. Springer, 140–155. https://doi.org/10.1007/978-3-540-89275-5_8

M. Anton Ertl. 1995. Stack Caching for Interpreters. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI'95).* ACM, 315–327. https://doi.org/10.1145/207110.207165

M. Anton Ertl and David Gregg. 2003. The Structure and Performance of Efficient Interpreters. *Journal of Instruction-Level Parallelism* 5 (November 2003), 1–25. http://www.jilp.org/vol5/v5paper12.pdf

Yoshihiko Futamura. 1999, Originally published in 1971. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (1999, Originally published in 1971), 381–391. https://doi.org/10.1023/A:1010095604496

Andreas Gal, Christian W. Probst, and Michael Franz. 2006. HotpathVM: An Effective JIT Compiler for Resource-constrained Devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE'06).* ACM, 144–153. https://doi.org/10.1145/1134760.1134780

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software* (1st ed.). Addison-Wesley Professional.

Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, Mikel Luján, and Hanspeter Mössenböck. 2018. Cross-Language Interoperability in a Multi-Language Runtime. *ACM Transactions on Programming Languages and Systems* 40, 2 (June 2018), 1–43. https://doi.org/10.1145/3201898

Michael Haupt, Robert Hirschfeld, Tobias Pape, Gregor Gabrysiak, Stefan Marr, Arne Bergmann, Arvid Heise, Matthias Kleine, and Robert Krahn. 2010. The SOM Family: Virtual Machines for Teaching and Research. In *Proceedings of the 15th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'10).* ACM, 18–22. https://doi.org/10.1145/1822090.1822098

Christian Humer and Nikola Bebić. 2022. Operation DSL: How We Learned to Stop Worrying and Love Bytecodes again. https://2022.ecoop.org/details/truffle-2022/3/Operation-DSL-How-We-Learned-to-Stop-Worrying-and-Love-Bytecodes-again Truffle Workshop '22, Presentation.

Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. 2014. A Domain-Specific Language for Building Self-Optimizing AST Interpreters. In *Proceedings of the 13th International Conference on Generative*

Programming: Concepts and Experiences (GPCE '14). ACM, 123–132. https://doi.org/10.1145/2658761.2658776

Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP '91: European Conference on Object-Oriented Programming (LNCS, Vol. 512)*. Springer, 21–38. https://doi.org/10.1007/BFb0057013

Yusuke Izawa, Hidehiko Masuhara, Carl Friedrich Bolz-Tereick, and Youyou Cong. 2022. Threaded Code Generation with a Meta-Tracing JIT Compiler. *Journal of Object Technology* 21, 2 (2022), 2:1–11. https://doi.org/10.5381/jot.2022.21.2.a1

Sophie Kaleba, Octave Larose, Richard Jones, and Stefan Marr. 2022. Who You Gonna Call: Analyzing the Run-time Call-Site Behavior of Ruby Applications. In *Proceedings of the 18th Symposium on Dynamic Languages (DLS'22)*. ACM, 14. https://doi.org/10.1145/3563834.3567538

Tomas Kalibera, Petr Maj, Floreal Morandat, and Jan Vitek. 2014. A Fast Abstract Syntax Tree Interpreter for R. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'14)*. ACM, 89–102. https://doi.org/10.1145/2576195.2576205

Chaitanya Koparkar, Mike Rainey, Michael Vollmer, Milind Kulkarni, and Ryan R. Newton. 2021. Efficient Tree-Traversals: Reconciling Parallelism and Dense Data Representations. *Proceedings of the ACM on Programming Languages* 5, ICFP (Aug. 2021), 1–29. https://doi.org/10.1145/3473596

Philipp Körner, David Schneider, and Michael Leuschel. 2021. On the Performance of Bytecode Interpreters in Prolog. In *Functional and Constraint Logic Programming: 28th International Workshop, WFLP 2020 (WFLP'20, Vol. 12560)*. Springer, 41–56. https://doi.org/10.1007/978-3-030-75333-7_3

Octave Larose, Sophie Kaleba, Humphrey Burchell, and Stefan Marr. 2023. *AST vs. Bytecode: Interpreters in the Age of Meta-Compilation (Artifact)*. https://doi.org/10.5281/zenodo.8333815

Octave Larose, Sophie Kaleba, and Stefan Marr. 2022. Less Is More: Merging AST Nodes To Optimize Interpreters. MoreVMs '22 Workshop, Presentation.

Stefan Marr. 2023. *ReBench: Execute and Document Benchmarks Reproducibly*. Open Source Software. https://doi.org/10.5281/zenodo.1311762 Version 1.2 https://github.com/smarr/ReBench/.

Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-Language Compiler Benchmarking—Are We Fast Yet?. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS'16)*. ACM, 120–131. https://doi.org/10.1145/2989225.2989232

Stefan Marr and Stéphane Ducasse. 2015. Tracing vs. Partial Evaluation: Comparing Meta-Compilation Approaches for Self-Optimizing Interpreters. In *Proceedings of the 2015 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '15)*. ACM, 821–839. https://doi.org/10.1145/2814270.2814275

Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2018. GraalSqueak: A Fast Smalltalk Bytecode Interpreter Written in an AST Interpreter Framework. In *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'18)*. ACM, 30–35. https://doi.org/10.1145/3242947.3242948

G. Ottoni and B. Liu. 2021. HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 340–350. https://doi.org/10.1109/CGO51591.2021.9370314

Tobias Pape. 2021. *Efficient Compound Values in Virtual Machines*. Doctoral Thesis. Universität Potsdam. https://doi.org/10.25932/publishup-49913

Tobias Pape, Carl Friedrich Bolz, and Robert Hirschfeld. 2017. Adaptive Just-in-time Value Class Optimization for Lowering Memory Consumption and Improving Execution Time Performance. *Science of Computer Programming* 140 (2017), 17–29. https://doi.org/10.1016/j.scico.2016.08.003 Object-Oriented Programming and Systems (OOPS 2015).

Tobias Pape, Tim Felgentreff, Robert Hirschfeld, Anton Gulenko, and Carl Friedrich Bolz. 2015. Language-independent Storage Strategies for tracing-JIT-based Virtual Machines. In *Proceedings of the 11th Symposium on Dynamic Languages (DLS 2015)*. ACM, 104–113. https://doi.org/10.1145/2816707.2816716

Ian Piumarta and Fabio Riccardi. 1998. Optimizing Direct-threaded Code by Selective Inlining. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation - PLDI '98*. ACM, 291–300. https://doi.org/10.1145/277650.277743

Todd A. Proebsting. 1995. Optimizing an ANSI C interpreter with superoperators. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '95*. ACM Press, 322–332. https://doi.org/10.1145/199448.199526

Mohaned Qunaibit, Stefan Brunthaler, Yeoul Na, Stijn Volckaert, and Michael Franz. 2018. Accelerating Dynamically-Typed Languages on Heterogeneous Platforms Using Guards Optimization. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (ECOOP'18, Vol. 109)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 16:1–16:29. https://doi.org/10.4230/LIPIcs.ECOOP.2018.16

Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. 2008. Virtual Machine Showdown: Stack Versus Registers. *ACM Transactions on Architecture and Code Optimization* 4, 4 (Jan. 2008), 1–36. https://doi.org/10.1145/1328195.1328197

Ben L. Titzer. 2022. A Fast In-Place Interpreter for WebAssembly. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (Oct. 2022), 646–672. https://doi.org/10.1145/3563311

Michael L. Van de Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. 2018. Fast, Flexible, Polyglot Instrumentation Support for Debuggers and other Tools. *Programming Journal* 2, 3 (March 2018), 30. https://doi.org/10.22152/programming-journal.org/2018/2/14

Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, Chaitanya Koparkar, Milind Kulkarni, Sam Tobin-Hochstadt, and Ryan R. Newton. 2017. Compiling Tree Transforms to Operate on Packed Representations. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 74)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 26:1–26:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.26

Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize Once, Start Fast: Application Initialization at Build Time. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 1–29. https://doi.org/10.1145/3360610

Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. 2014. An Object Storage Model for the Truffle Language Implementation Framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '14)*. ACM, 133–144. https://doi.org/10.1145/2647508.2647517

Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*. ACM, 662–676. https://doi.org/10.1145/3062341.3062381

Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!'13)*. ACM, 187–204. https://doi.org/10.1145/2509578.2509581

Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Dynamic Languages Symposium (DLS'12)*. ACM, 73–82. https://doi.org/10.1145/2384577.2384587

Qiang Zhang, Lei Xu, and Baowen Xu. 2022. RegCPython: A Register-based Python Interpreter for Better Performance. *ACM Transactions on Architecture and Code Optimization* 20, 1 (Dec. 2022), 1–25. https://doi.org/10.1145/3568973

Received 2023-04-14; accepted 2023-08-27