Building Efficient and Highly Run-Time Adaptable Virtual Machines

Guido Chari, Diego Garbervetsky
Departamento de Computación, FCEyN, UBA.
CONICET, Argentina
{gchari,diegog}@dc.uba.ar

Stefan Marr
Johannes Kepler University Linz, Austria
stefan.marr@jku.at

Abstract

Programming language virtual machines (VMs) realize language semantics, enforce security properties, and execute applications efficiently. Fully Reflective Execution Environments (EEs) are VMs that additionally expose their whole structure and behavior to applications. This enables developers to observe and adapt VMs at run time. However, there is a belief that reflective EEs are not viable for practical usages because such flexibility would incur a high performance overhead.

To refute this belief, we built a reflective EE on top of a highly optimizing dynamic compiler. We introduced a new optimization model that, based on the conjecture that variability of low-level (EE-level) reflective behavior is low in many scenarios, mitigates the most significant sources of the performance overheads related to the reflective capabilities in the EE. Our experiments indicate that reflective EEs can reach peak performance in the order of standard VMs. Concretely, that a) if reflective mechanisms are not used the execution overhead is negligible compared to standard VMs, b) VM operations can be redefined at language-level without incurring in significant overheads, c) for several software adaptation tasks, applying the reflection at the VM level is not only lightweight in terms of engineering effort, but also competitive in terms of performance in comparison to other ad-hoc solutions.

Categories and Subject Descriptors D.3.4 [*Processors*]: Run-time Environments, Optimization

General Terms Languages, Performance, Experimentation

Keywords Reflection, Virtual Machines, Metaobject Protocols, Performance

1. Introduction

VMs for dynamic programming languages provide a wide range of heterogeneous functionalities including the language semantics, memory management, native code compilation, security enforcement and the interaction with the operating system. The low-level nature and interdependency of these features cause industrial strength VMs to be complex artifacts. Consequently, adapting VM features at run time is challenging and usually only possible in an ad-hoc manner.

A Fully Reflective Execution Environment (EEs) [5] enables VMs to be modified at run time using reflective APIs. Fully reflective EEs are based on metaobject protocols (MOPs) [11] and enable adaptation scenarios ranging from simple changes in the semantic of individual operations up to the complete reimplementation of its components. Consequently, fully reflective EEs have an enormous potential as a general platform for developing flexible and adaptable applications. To support such adaptation capabilities, fully reflective EEs must expose their structure and behavior to the language level.

In stark contrast to other reflective approaches, fully reflective EEs provide reflective capabilities in every component of the VM such as the interpreter, memory manager, garbage collector, object layout, etc. This flexibility implies a significant proliferation of indirections and conditions (guards) that the EE needs to check at run time to realize the proper semantics. The same applies also for (partially) reflective EEs, which follow the same approach but implement reflection in a subset of the components and functionalities. Recent work has shown that speculative compilers can remove the run-time overhead for common usage patterns of reflection and even for simple MOPs [14]. Although these performance results do not apply directly to fully reflective EEs they show a promising path for exploring ways to make them efficient.

In this paper we show preliminary empirical evidence that reflective EEs can reach a peak performance similar to standard non-reflective VMs. To do so, we first built an ex-

perimental reflective EE named TruffleMate,[1] which is an extension of TruffleSOM,[2] an optimized implementation of SOM Smalltalk. TruffleMate supports an adapted version of Mate v1's MOP, a prototypical reflective EE presented in [5]. Based on the conjecture that the variability of actually observed metaobjects is low for consecutive executions of the VM semantics, we designed and implemented an optimization model that minimizes indirections of the meta-level by speculating that its behavior is stable.

We assess the peak performance of TruffleMate with VM benchmarks and measure the overhead incurred by the Mate MOP for individual VM operations. Furthermore, we evaluate TruffleMate in on-the-fly adaptation scenarios that need intervention at the EE level: introducing immutable (read-only) references [2] and monitoring method activations. The evaluation shows that our reflective EE has on average zero-overhead at peak performance compared to TruffleSOM when no adaptations are needed. The same applies for the redefinition of individual VM operations at language-level. We finally show that for more elaborated scenarios the reflective solutions using TruffleMate introduce low overheads while its MOP-based implementation is lightweight in terms of engineering effort.

In summary, the contributions of this work are:

- An approach to optimize MOP-based reflective EEs based on the assumption that the actually observed local meta-variability at run time is low.
- TruffleMate: an EE with reflective capabilities for execution semantics and object layouts that implements the aforementioned optimizations.
- Empirical evidence showing that TruffleMate can run efficiently in terms of peak performance.

2. Background

To set up the remainder of the paper, this section describes the main characteristics of reflective EEs following the Mate approach. Then we introduce the Graal dynamic compiler and Truffle. Finally we describe TruffleSOM, an implementation of SOM in Truffle.

2.1 Mate Approach to Virtual Machines

In Fully Reflective EEs every VM component provides comprehensive reflective capabilities, *i.e.*, it is possible to introspect and intercede its structure and behavior from the language level [5]. This characteristic enables flexible adaptations at run time, *e.g.*, to implement new language features, security properties, monitoring facilities, or to optimize an object's low-level representation. The Mate approach uses a MOP to mediate between the application and the VM reflective capabilities.

[1] https://github.com/charig/TruffleMATE

[2] https://github.com/SOM-st/TruffleSOM

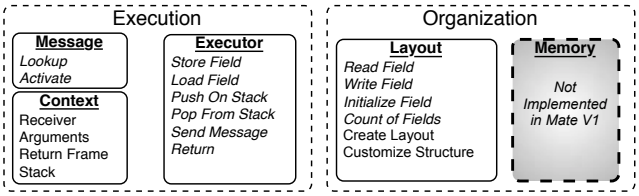

Figure 1. Mate V1's MOP (based on Figure 2 of [5]).

Mate v1 is a prototypical implementation of the Mate approach to demonstrate its feasibility and experimenting with its applicability. As such, it supports reflective capabilities for a subset of the VM's components. However, it does not include reflective capabilities for the memory manager. That is the reason why it is considered a reflective but not fully reflective EE. Concretely, Mate v1 consists mainly of an interpreter for the Smalltalk-80 bytecode set [8] extended with hooks (intercession handlers) in the reflective VM components. Figure 1 shows the MOP representing Mate v1's reflective capabilities at the VM level with the reflective operations clustered by their corresponding VM component. The operations highlighted with italic fonts represent behavioral operations while the others represent structural features. The remainder of this paper refers to the Mate v1's MOP simply as the Mate MOP.

2.2 Example: Tracing Program Execution with Mate

In Mate v1 metaobjects can be installed either to individual objects or to method activation frames. When installed, they govern the semantics of the interceded entity. To illustrate their usage, below we show how to install a metaobject that monitors the number of method invocation that an individual object is making (in Section 6 we generalize this to all application objects using metaobjects installed in frames). Consider this to be part of a server application scenario, where the requirement is to dynamically enable the aforementioned monitoring in a module without shutting down the application. On top of Mate we only need to create a metaobject with the monitoring semantics and install it into the required object:

```
1  class TracingMessageMO extends Metaobject {
2    def activateWithArgs(subject, aMethod, args){
3      Counter.addMethodActivation();
4      super.activateWithArgs(subject, aMethod, args);
5    }
6  }
7  tracingMO = new TracingMessageMO();
8  monitoredObject.installSemantics(tracingMO);
```

Listing 1. Metaobject to trace method activations

The first step consists of subclassing the *Message* metaobject and redefining the `activateWithArgs` method so it increments a global counter (Lines 1-3). Then the method activation delegates the execution to the *superclass* method that defines the standard activation. Finally, it becomes a matter

of simply installing the metaobject into the required object (Lines 7-8).

2.3 Truffle and Graal

The Truffle framework allows developers to build programming languages by expressing their semantics as abstract syntax trees (ASTs). To realize self-optimizing AST interpreters [25], Truffle provides a framework to specify specializations. Specializations express special cases for a language's execution that need to do less work at run time than a generic implementation (*e.g.*: use primitive operations for integers instead of language methods). This approach minimizes run-time checks and interpreter code executed for a specific program.

In combination with the Graal JIT compiler [24], self-optimizing interpreters can reach performance of the same order of magnitude as Java on top of HotSpot [13]. To reach this performance, Truffle applies partial evaluation on the specialized ASTs to determine the compilation unit that corresponds to a relevant part of a guest-language's program, which is then optimized and compiled to native code by Graal using classic compiler optimizations, inlining, and escape analysis. According to Simon et al. [19], the Graal compiler produces native code for Java benchmarks (DaCapo, SPECjvm2008, and others) that is on average 8% slower than the code produced by HotSpot's highly optimizing C2 compiler.

Truffle Object Storage Model. Truffle includes an object storage model (OSM) [23] that provides language implementers with an efficient representation for objects. The OSM maintains a so-called object *shape*. Shapes are immutable representations of a fixed set of fields and the types that have been observed for an object at one point in time. The OSM optimistically optimizes objects of the same class or prototype based on the assumption that the potential dynamicity that is allowed by a language's object model is rarely used.

Objects are represented by a set of memory locations for fields and a pointer to a shape describing their memory layout. When a field is accessed at run time, its actual memory location is determined by inspecting the object's shape and then accessed directly. The OSM speculates on the memory location information for a specific shape to avoid repeated lookup at run time. With this approach, the run-time overhead is a simple comparison between the shape of the current object, and the one used for speculation.

In the context of Mate, these shapes are an implementation of the *Layout* component of the MOP. We use them to realize layouts, and for consistency refer to them as layouts as well.

2.4 TruffleSOM

The Simple Object Machine (SOM) [9] is a Smalltalk implementation designed to avoid inessential complexity. It includes fundamental language concepts such as objects, classes, closures, and non-local returns. Following the Smalltalk tradition, control structures such as `if` or `while` are defined as polymorphic methods on objects and rely on closures and non-local returns for realizing their expected behavior.

TruffleSOM [12, 14] is a SOM implementation using Truffle. When used in combination with Graal it is just 2.7x slower than Java for a set of classic VM benchmarks (cf. Section 6.2). TruffleSOM relies on Truffle and the Truffle OSM to run efficiently by speculating on the observed run-time types and using primitive operations such as integer additions or String access operations.

2.4.1 Dispatch Chains

Dispatch chains are a generalization of polymorphic inline caches (PICs)[10, 14]. PICs record type information and cache methods to minimize their lookup overhead. Dispatch chains generalize PICs to generic operations such as object field accesses and metaprogramming, caching arbitrary values [14, 23]. Like PICs, which are based on the observation that the number of invoked methods at a call site in a program is typically small [6, 10], dispatch chains rely on the stability and low variability of the run-time behavior. They are structured so that the last element implements the *fallback* behavior (*i.e.*, the behavior for the most general case).

TruffleSOM uses dispatch chains to implement its optimizations. Figure 2 shows an example: on the left-hand side is the unoptimized AST of an expression that accesses object field `foo` and invokes the method `bar` with a constant integer parameter. The right-hand side shows the optimized AST. The optimized state is reached after executing the expression at least once. The *Literal Node* is rewritten to an *Integer Literal Node* and the *Message Node* is identified as a generic message. The bottom compound nodes are an example of a dispatch chain for optimizing a field read. The first node of the list already cached the value of the only observed layout until now and the location where the corresponding field is located for that layout. In further executions of the field read, the VM first checks if the layout of the receiver is the same as the one cached. In case of a hit, the location for the field does not have to be computed. In case of a miss, the chain executes the following node. The last node of the chain (in this case *Uninitialized Read* node) must always decide if it adds a new cached entry to the chain or if it executes the generic, and thus more expensive in terms of performance, read operation.

3. The Overhead of Supporting VM Reflective Capabilities

Traditionally, supporting reflective capabilities comes with a significant run-time overhead [14]. Before executing operations such as invoking a method, reading a field, or accessing a local variable, a reflective EE must verify whether the operation is redefined at the language level, and if so, execute the language-level code. This is known as *intercession handling* (IH). Instead of using highly optimized code, IH requires

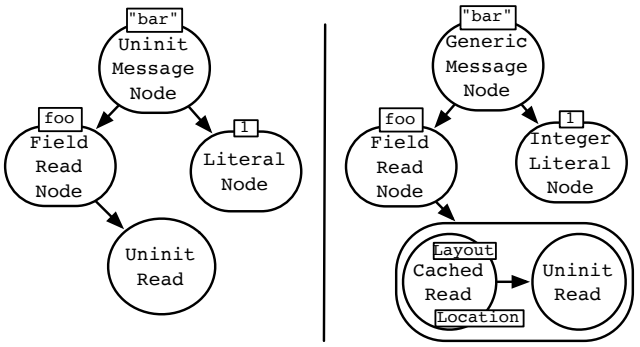

Figure 2. Parser output (left) and Optimized (right) ASTs for an expression `foo.bar(1)` in TruffleSOM.

run-time checks and relies on language-level code to realize basic operations.

As a preliminary evaluation, we added the Mate MOP naively without any kind of optimizations to TruffleSOM, an optimized Smalltalk VM. We observed on average an overhead of 4.6x when analyzing a set of macro benchmarks that do not redefine semantics (cf. Section 6.3). Considering the massive proliferation of IH sites in a reflective EE 4.6x on peak performance can be seen as a great achievement of the Graal compiler. At the same time, it also exposes that several extra indirections are still being executed, even when the MOP is not being used. Furthermore, we saw an overhead of at least 100x when we started using metaobjects. This is prohibitive and blocks adoption of reflective EEs.

The most significant additional behavior introduced by our reflective EE are the ubiquitous IHs. Analogously to call sites for method invocations, we give the name *IH site* to any independent place where the IH is actually triggered. Mate v1 requires the installation of an IH site just before the VM execution of any of the behavioral operations included in Figure 1. Furthermore, Mate supports the intercession of every VM operation at both, the object and method levels. The following algorithm defines the common substrate:

```
1  def IH(frame, operation){
2    result = NOMETAOBJECT.
3    metaobject = frame.getMetaobject();
4    if (metaobject != null){
5      result = metaobject.activateFor(operation);
6    }
7    if (metaobject == null or result == null) {
8      metaobject = getReceiver().getMetaobject();
9      if (metaobject != null){
10       result = metaobject.activateFor(operation);
11     }
12   }
13   return result;
14 }
```

Listing 2. Algorithm describing Mate's IH.

First, the IH checks for the existence of a metaobject associated with the current *frame* of execution. In that case the IH activates (dispatches to) the language-level redefinition of the current operation in the corresponding metaobject. In case there is no metaobject for the frame, or it does not redefine the VM operation being executed, the IH checks for a metaobject in the *subject* of the current VM operation (*e.g.*, in a variable read operation, the concrete object owning the variable). Again, if there is a metaobject the IH delegates the execution to the operation redefined in the metaobject. Otherwise, the IH returns the `NOMETAOBJECT` constant and the VM executes the default behavior for the current operation.

Performance Analysis of IH. Metaobjects are objects themselves. Therefore, interacting with them requires accessing memory. As a consequence, in addition to the performance overhead caused by the tests in each branch, the IH incorporates several indirections (memory accesses) to the execution of each VM operation. However, we expect that both, receiver of the operations and the corresponding frames for these operations are used repeatedly in consecutive executions of the IH. Therefore, a dynamic compiler should be able to factor out most of the repetitive indirections introduced by subsequent IH sites. This can be illustrated in the context of the execution of a method. A method is composed of several operations such as field reads, variable accesses, and other operations, each defining a different IH site in the context of a reflective EE. We conjecture that the subject of most of these operations, in this case the receiver or frame of the method, is the same.

The Challenge of Meta-Variability. We observe that the main difficulty of compilers in the context of reflective EEs is dealing with the variability of metaobjects. To enable arbitrary dynamic adaptation, reflective EEs allow users to freely change metaobjects for subjects or frames. For a compiler, this means it cannot make any assumptions about the meta-level operations, which in turn prevents many optimizations. For example, let us consider the compilation of a block of code that accesses the metaobject of the same object multiple times. Since the metaobject could change at any point in time, a compiler would only optimize out the accesses and the corresponding meta-level operations if it is able to prove that the metaobject does not change. If not, the compiled code needs to include all reads of metaobjects and the corresponding operations on them.

As a result of the meta-variability problem, several operations relating to metaobjects cannot be remove. Even worse, compiler optimizations such as inlining may not be generally applicable any longer, which means that several of the indirections introduced by IH cannot be optimized out. This leads in the end to what we call the proliferation of indirections at run time resulting in a significant impact on application performance.

4. Optimizing Mate

This section details the strategy to minimize the overhead introduced by Mate's VM-level reflection.

4.1 Conjectures about the Dynamic Usage of the Mate MOP

Reflective EEs are a very recent approach for building flexible systems. As a consequence, there are no large applications from which common usage patterns could be derived. Instead, we conjecture that the usage patterns will be similar to what is observed for dynamic languages [14], *i.e.*, the applications use only a minimal degree of the potential dynamicity the language provides, showing very stable behaviors. This conjecture is also informed by literature on unanticipated dynamic adaptation approaches [16], where the most common scenarios described require small dynamicity at run time. Mate's MOP fits well as an expressive medium for introducing additional stable behavior into an application. Our conjectures can be detailed as follows:

Stable semantics: We expect users of the Mate MOP to freely define and combine metaobjects describing (novel) adaptations to the EE at run time. However, we do not expect those adaptations to be changed often and, thus, show a stable behavior eventually. The main reason is that it is hard to reason about frequently changing behavior. This means, we assume users are not going to constantly create new metaobjects with different behavior within the same application.

Low local meta-variability: While a single IH site can potentially observe many different metaobjects, we expect the behavior to be similar to that of dynamic method dispatches [10]. This means that the large majority of IH sites will be monomorphic, *i.e.*, they observe only a single metaobject. Few IH sites will be polymorphic with multiple metaobjects observed. Only in very rare cases many observed metaobjects will make an IH site megamorphic. We further assume this behavior correlates strongly with method polymorphism. A megamorphic call site is likely to have a higher probability to observe more metaobjects than a monomorphic call site that is used only in very specific cases.

Overall, this means that for a wide variety of use cases, the local meta-variability will be minimal and most IH sites will be monomorphic.

4.2 Optimizing IH Sites

This section describes the optimization model based on our conjectures of low local meta-variability.

Speculate on metaobjects: We propose to cache, at each IH site, a predefined number of the metaobjects that has been seen at that site. Furthermore, for each metaobject we propose to cache the entry point of the (language-level) method reimplementing the corresponding VM operation at the current IH site. This way, at run time, we only need to test whether the current metaobject is in the list of the cached ones. In case of a cache hit, we can directly call the corresponding cached method saving all the indirections that the lookup of the method in the metaobject consume. In case of a miss, we have to execute all checks and lookups. If the stable behavior assumption holds misses will be infrequent. Any dynamic change of the metaobject attached to a frame or object will trigger a deoptimization at run time, eventually stabilizing again.

Speculate on method activations: Similarly, metaobjects assigned to frames are cached at each IH site.

Combine metaobjects with object layouts: Most IH sites are triggered by VM operations that may need to access the state of objects. For instance, reading/writing fields or obtaining object's meta information like its class, *e.g.*, for method lookups. Hence, we designed an object model where the metaobjects are coupled with the object's layout. The dispatch chain can then store object layouts and the entry point of the redefined method for the corresponding metaobject. At run time, the dispatch chain guard only needs to check the object layout with pointer equality. Since for most VM operations the layout needs to be accessed, this check does not introduce overhead. A possible drawback of this strategy is that objects with the same structure but with different metaobjects have distinct layouts. As a consequence, the size of any dispatch chain guarding object layouts may increase, incurring additional overheads. Nevertheless, we do not expect a significant impact due to the low meta variability.

We expect that an aggressive dynamic compiler in combination with the proposed optimizations will remove most of the indirections caused by the IHs. We validate this hypothesis in Section 6.

5. Implementing Mate Efficiently

TruffleMate is our prototypical implementation of a reflective EE supporting the Mate MOP (cf. Section 2.1). It is implemented as an extension of TruffleSOM and incorporates all the optimizations described in Section 4. We chose a Smalltalk-like language as target because it already features advanced reflective capabilities at the language level that eases the implementation of the VM-level reflective API. Nevertheless, our contributions focus mainly on supporting reflection at the VM level and we expect them to be language independent.

For supporting the Mate MOP, TruffleMate mainly incorporates: reflective layouts, reflective execution contexts and IH sites for all the behavioral operations of the MOP. In Mate, the semantics of each VM operation can be redefined at the object or method level (cf. Section 3). To support the former, every object is able to refer to a metaobject that describes the semantics of a set of VM operations for itself. To support the latter, we modified the calling convention so that every method receives an extra parameter with the metaobject that

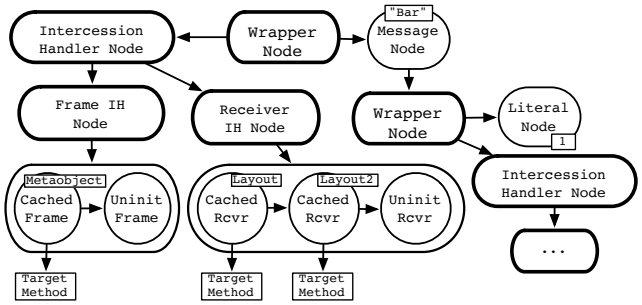

Figure 3. Truffle Mate's AST version of foo.bar(1) statement without including the field read node but with dispatch chains already filled. The bold circle nodes are metalevel nodes exclusive for Mate. Simple circled nodes were inherited from TruffleSOM.

governs the semantics of its execution. This implies that each frame includes the metaobject describing how the activated method must behave.

5.1 Meta-level Nodes

We refer to nodes that are part of TruffleSOM, *e.g.*, the nodes presented in Figure 2, as the *base-level nodes*. In TruffleMate, the MOP semantics are implemented with new AST nodes. We refer to nodes related to Mate as *meta-level nodes*. The base-level nodes for VM operations that can be customized by the MOP are wrapped with meta-level nodes that perform the IH. If the execution of the IH in the meta-level node returns that the operation is not redefined, then the meta-level node delegates the execution to the base-level node.

Figure 3 shows the expression of Figure 2 after wrapping the message send node. Bold thick nodes represent meta-level nodes. The IH is composed of several meta-levels nodes. Each node implements a fragment of the IH algorithm presented in Listing 2 (cf. Section 3), or is part of a dispatch chain (compound nodes) implementing the optimizations described in Section 4.2. We introduce the meta-level nodes below, and leave the description of the dispatch chains for the following section.

Wrapper Node: the entry point of every VM operation that can be redefined. It wraps a base-level node (a VM operation) and connects it with its corresponding meta-level nodes implementing the IH. Its role is to delegate first to the IH to execute the meta-level behavior if the operation is redefined or delegate to the base-level node otherwise.

Intercession Handler Node: orchestrates the intercession handling relying on the Receiver and Frame IH nodes.

Receiver and Frame IH Node: checks whether the environment/receiver has a metaobject redefining the current operation. In that case delegates the invocation of the corresponding language-level method to its dispatch chain.

5.2 Optimizations

Below we explain how we implement the optimization model presented in Section 4 in order two optimize IHs in general, and the dispatch chains for subjects and frames, in particular.

Speculate on the Metaobject of Subjects. As explained in Section 4, we expect the variability of metaobjects at a specific IH site to be low. Thus, we cache the observed metaobjects and the corresponding language-level operations within a dispatch chain. This optimization avoids to lookup for the redefinition of an operation in a metaobject whenever the metaobject is the same. Furthermore, it enables a compiler to inline the language-level operation. In the dispatch chain hanging from the *Receiver IH Node* (Figure 3), each *Cached Frame Node* caches a *Layout* and the target method reimplementing the VM operation. In case the assumption of a low meta-variability fails, the dispatch chain falls back to using a generic implementation without caching.

Speculate on the Metaobject of Frames. Analogously to the previous case, we implemented a similar dispatch chain speculating on the metaobjects attached to frames. This dispatch chain hangs from the *Frame IH Node* of Figure 3. In contrast to the previous case, this dispatch chain caches the actual metaobject because frames do not have a layout.

Optimize Metaobject Guards. We combine object layouts with metaobjects as a mean to reduce an extra indirection in the IHs (cf. Section 4.2). Concretely, we add to every object layout an extra field denoting its metaobject. Then, by just testing two layouts we can determine any change in the metaobject assigned to an object. Accordingly, we store in the dispatch chains that caches metaobjects, not only them, but also a pointer to the layout. Therefore, at run time when each *Receiver IH* node walks through the *Cached Rcvr* nodes of the dispatch chain, it first compares the current subject's layout with the stored layout to determine whether it can execute the corresponding optimized node. In case of hit, it saves the extra indirection of reading the actual metaobject assigned to the current subject.

Speculate on IH branches. Due to the low variability it is likely that each IH will execute only one of its branches (*i.e.*, receiver or frame). However, the compiler is not always able to figure that out and cannot optimize accordingly. We thus profile branches related to the usage of metaobjects to help the compiler to speculate on which branches to compile and optimize.

6. Validation

This section evaluates the impact of our optimizations on TruffleMate's peak performance.

6.1 Research Questions

We based our evaluation on these research questions:

RQ1: What is the *inherent* peak performance overhead of running programs in a reflective EE?

RQ2: What is the peak performance overhead of redefining *individual VM operations* at language level?

RQ3: What is the peak performance overhead of *running applications* that dynamically adapt by redefining VM operations at language level?

6.2 Methodology

To account for the non-determinism in modern systems as well as the adaptive optimization techniques in TruffleMate, Graal, and HotSpot, each reported result is based on 100 measurements after a steady state has been reached, thus measuring *peak performance*. To determine when a steady state is reached, we run several iterations of the same benchmark in a single process execution and manually select a threshold since where measurements does not show signs of compilation. We report geometric means. The benchmarking machine is a quad-core Intel Core i7-3770, 3.40 GHz with 16 GB RAM, running Ubuntu with Linux kernel 4.2, and Java 1.8.0_91 with HotSpot 25.91-b14.

Subjects. The benchmarks for the experiments were collected from various sources and include classic VM benchmarks used for Smalltalk VMs, JVMs, JavaScript VMs (e.g. Octane and JetStream suites), as well as own additions. They have been translated to Java 8 and Smalltalk respectively idiomatically, so that they behave strictly identical on both languages, and to avoid code patterns that are unexpected by the compilers. This allows for an assessment of the effectiveness of the compiler optimizations and gives as much comparability between languages as possible. The benchmarks for assessing the performance of using reflection at the VM level are described in their particular sections.

Base Performance. TruffleSOM, our baseline for the experimentation, is on average only about 2.7x slower than Java 8 on the HotSpot JVM. This is comparable with the performance of the V8 VM.

Warmup. To assess the warmup behavior of TruffleMate and TruffleSOM, we analyzed time series plots for the micro and macro benchmarks reported in Section 6.3. The plots are not included because of space constraints.[3] Overall, TruffleMate and TruffleSOM show very similar warmup patterns, *i.e.*, compilation happens in the same phases, and the curves have the same shape.

To characterize warmup, we measure the first iteration for which execution time differs by at most 10% from the mean peak performance, and whose next 5 subsequent iterations maintain the same condition. Based on this proxy, it takes TruffleSOM, on average, 42 iterations and 3.8 seconds to warmup, while 46 iterations and 8.3 seconds to TruffleMate. Considering only the first iteration where the code is mostly interpreted, the mean execution time in TruffleSOM is 0.4 seconds while 0.7 in TruffleMate respectively. As a conclusion, the IH does not increase significantly the warming up iterations while it has some impact on the overall execution time because more code needs to be executed and compiled.

[3] Warmup behavior of TruffleSOM is reported in [12].

6.3 Inherent Overhead

For assessing the inherent overhead of our reflective VM we evaluate a set of benchmarks that do not execute meta-level behavior, in both, TruffleSOM and TruffleMate. Therefore, the performance difference is determined solely by the inherent overhead of TruffleMate.

Overhead without optimizing the meta level. Figure 4 shows that, even if no metaobjects are being used, there is an inherent overhead from executing IH sites that the compiler could not optimize. The mean of this overhead is 3.38x for micro benchmarks, 5.3x for macro benchmarks, and the worst cases are FieldLoop with 12.9x and Richards with 8.53x correspondingly.

Overhead when optimizing the meta level. Figure 4 also shows the results when our optimizations are enabled. The mean overhead is 0.97x and 1.02x for micro and macro benchmarks, with worst cases of 1.13x for TreeSort and 1.21x for DeltaBlue. Notice that some runs are even slightly faster in TruffleMate. This may be influenced by the use of different memory layouts, and cache effects triggered by the additional IH code. Considering all the benchmarks, the mean overhead is 0.99x. This shows that our optimization strategy (cf. Section 4.2) reduced the performance overhead significantly in a scenario when the flexibility of the VM is available, but not used.

Answering RQ1, we conclude from these results that reflective EEs can reach the performance of state-of-the-art VMs when metaobjects are not activated.

6.4 Individual Operations of the MOP

To analyze the impact of redefining VM operations at language level, we use mircobenchmarks to asses the overhead of redefining field reads/writes, method dispatches, returns, and all operations together. The redefinitions in Mate realize the standard behavior, but use language-level reflective operations. We also compare the cost of reading a field on a mono and megamorphic IH site. Concretely, we pseudo-randomly select an element from a list of 20 objects and read a fixed field. In the monomorphic benchmark (mono), all the objects use the same metaobject redefining the read operation. In the megamorphic (mega), we simulate a megamorphic site by assigning a distinct metaobject, redefining equally the reading operation, to each object.

Overhead without optimizing the meta level. The top quadrants of Figure 5 shows the results using TruffleMate without dispatch chains. The micro benchmarks show a mean overhead of 1532x. The worst case is the 'All' benchmark

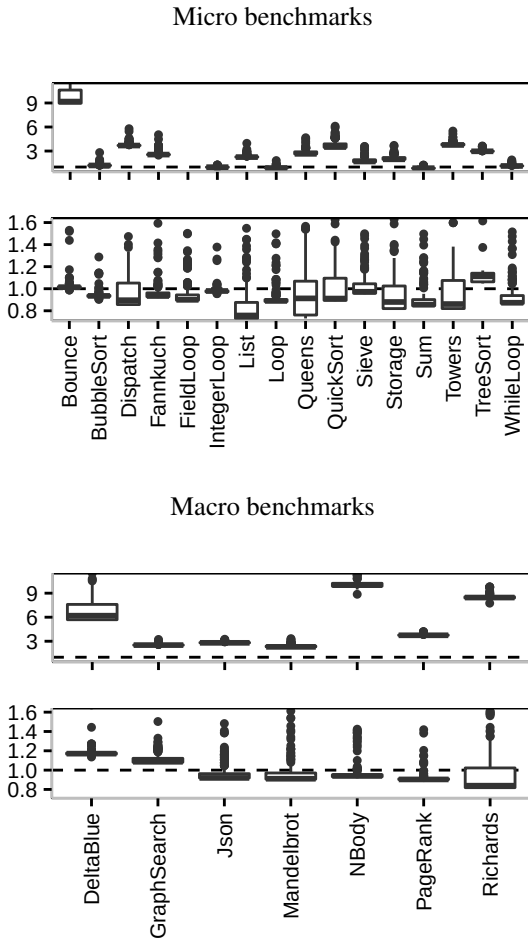

Figure 4. Overhead of TruffleMate normalized to TruffleSOM, clustered by micro and macro benchmarks, without (above) and with (below) optimizations.

with 13563x overhead since it is the one that redefines more operations. The mono/mega morphic analysis in the top-right figure presents a mean overhead of 31.36x, with similar results in both cases.

Overhead when optimizing the meta level. The boxplot in the bottom-left quadrant of Figure 5 indicates that there is no considerable overhead in executing a redefined VM operation at language level in terms of peak performance. The mean overhead for this micro benchmarks is zero while the worst case is the redefinition of the read operation with 1.035x.

In the top-right quadrant the monomorphic site exposes a mean overhead of 1.10x. The only difference with the individual operation case (with zero overhead) is that the monomorphic site observes much more objects. This suggests that, in some cases, the compiler is still not able to remove all indirections when dealing with a IH site that executes multiple subjects with the same metaobject. On the other hand, the megamorphic site is 18.5x slower showing that low

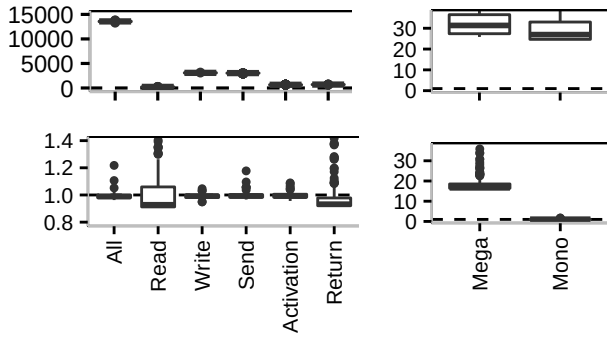

Figure 5. Left-hand side. Above: the overhead of running VM operations redefined at language-level, normalized to the same operation at the VM level. Below: the same but without dispatch chains. Right-hand side images show the overhead of an IH site that observe only one (mono) or several metaobjects (mega).

meta-variability is a requirement to optimize the Mate MOP successfully.

To answer RQ2, we conclude from this experiment that for cases with low meta-variability there is no perceptible overhead in redefining a VM operation at language-level. The excessive overhead of the unoptimized version expose the impact of our optimizations.

6.5 Using the MOP

To answer RQ3, we evaluate a more comprehensive usage of the Mate MOP. To assess the peak performance overhead of creating and activating metaobjects at run time, we analyze one implementation of read-only references and a case of instrumentation of method activations. These two scenarios intend to represent use cases of run-time adaptation for enforcing security properties (read-only references) or monitoring application's behavior (tracing). The former evaluates a more in-depth usage of metaobjects attached to subjects while the latter to frames. Both install metaobjects dynamically.

6.5.1 Read-only References

A read-only reference ensures that all objects reachable transitively through itself cannot be modified. Arnaud et al. [2] proposed an implementation of read-only references based on the idea of *handles*, a kind of proxy protecting the read-only reference. To enforce the transparency of handles, this approach requires the modification of the underlying VM. Later, an approach based on *delegation proxies* [22] proposed a language-level implementation. It uses hidden classes to instrument all the code that will eventually be called by a proxy. Using Mate's MOP we dynamically introduce handles using metaobjects with less than 50 lines of code:

```
1  class HandleMO extends Metaobject (
2    def read(subject, anIndex){
3      return new Handle(subject.instanceVarAt(anIndex));
4    }
5    def write(subject, anIndex, aValue){// ignore write}
6    def lookup(subject, aMethodName){
7      class = subject.target().getClass();
8      return super.lookup(
9                 subject.getTarget(), aMethodName);
10   }
11   def activateWithArgs(subject, aMethod, args){
12     if(aMethod.name().equals("=="))
13               args["receiver"] = subject.getTarget();
14     }
15 )
16 class Handle extends Object = (
17   fields: target;
18   static fields: semantics = new HandleMO();
19
20   def getTarget(){ return target;}
21   def static getSemantics(){return semantics;}
22   constructor Handle(anObject){
23     target = anObject;
24     this.installEnvironment(Handle.getSemantics());
25   }
26 )
```

Listing 3. Metaobject for immutable references.

The Handle class is a proxy to the readonly object (the *target* field) that installs a metaobject on its instances at initialization time. This metaobject enforces the readonly behavior by redefining the semantics of read, write, lookup and activation. The write operation is redefined to do nothing. The read enforces that any access to a field of an object referenced by a read-only reference returns a handle wrapping the reference. The lookup enforces that any method invoked in a Handle is looked up in the target reference (line 8). The activation ensures that the identity of read-only references is preserved. This way, clients always perceive that they interact directly with objects and not with handles.

For the sake of the analysis, we implemented a version of delegation proxies that *do not* use metaobjects. To compare both implementations, we ran a program that creates a (read-only) reference to the head of a linked list and traverses the list attempting to write some of its elements. We use TruffleMate with standard mutable references as baseline. Note that the version with handles attaches metaobjects to handles whenever a reference is read (Lines 3 and 24, Listing 3). In our setting, changing metaobjects is a slow operation since it requires to modify the object's layout. To avoid this slow operation during the traversal, we exploit TruffleMate's reflective capabilities on layouts. Concretely, we cache a predefined layout with a metaobject defining the read-only behavior already assigned, and instantiate handles using this layout.

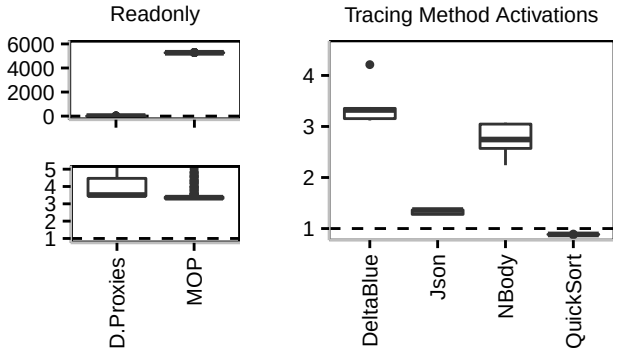

Figure 6. Left-hand side: Overhead of a traversing and intending to modify a readonly list in TruffleMate with two different approaches, DelegationProxies and the Mate MOP. The baseline is the same traversal but with standard mutable references. Above are the results without dispatch chains and below with the optimized version. Right-hand side: Overhead of monitoring method activations normalized to the same benchmarks without the monitoring.

Overhead without optimizing the meta level. The top left-hand side of Figure 6 shows that the overhead for delegation proxies is 4.1x while for Mate is a prohibitively 5946x.

Overhead when optimizing the meta level. The bottom left-hand side of the figure shows the optimized version results. Delegation proxies overhead does not vary since it does not use any kind of meta behavior, and as such, the overhead is mainly a result of the extra allocations and its double dispatching usage. Mate outperforms delegation proxies with a mean overhead of 3.5x. Furthermore, the implementation of handles in Mate is simpler and transparent to the language level. Moreover, it does not require an extensive framework, which might impede integration into existing code bases. It is worth noting that both approaches, delegation proxies and handles, need to wrap each field read during the traversal with a proxy or handle. This instantiation of considerably more objects than the baseline version is a plausible explanation of their overheads.

6.5.2 Tracing

In this experiment we asses the overhead of TruffleMate when a large number of IHs trigger the execution of metaobjects. This happens, for instance, when metaobjets are attached to the execution context of the VM operations (*i.e.*, the frame in the IH). Therefore, we introduce a metaobject that redefines the activation of methods to account for the number of activations made by the program. The redefined method activation increments a counter and invokes the original activation logic. It is intentionally simple because we are interesting in measuring the overhead of the IHs rather than the "tracing" itself. The actual metaobject is similar to the one presented in 2.2. The main difference is that it is attached

to frames instead of objects to ensure the monitoring is propagated in subsequents activations.

We ran the experiment on the DeltaBlue, NBody and Json macro benchmarks and the Quicksort micro benchmark. In an attempt to simulate the execution of more realistic applications we increased the input size for the benchmarks, and used a 55MB file for the Json parser instead of the original of 90KB.

Overhead without optimizing the meta level. Running this experiment without optimizations was not possible because the runs did not finish within a day.

Overhead when optimizing the meta level. Right-hand side of Figure 6 shows the results. QuickSort presents zero overhead, Json 1.33x, NBody 2.74x, and DeltaBlue 3.43x. Recall that this overhead includes the cost of the counting logic in all activations, including even activations on primitive types like integers and strings (*e.g.*, sum, multiplication, etc.).

DeltaBlue and NBody perform a significant amount of computation on primitive types. Moreover, we noticed that they are more sensitive to the inlining parameter of the Graal compiler. This may also increase their running times because the cost of invoking a method is usually higher than inlining the callee.

To answer RQ3, our experiments suggest that the peak performance overhead when dynamically adapting applications can be reasonably low in TruffleMate, specially when compared with language-level approaches.

6.5.3 Conclusions

Our experiments analyze the peak performance of Mate for scenarios that activate a small number of metaobjects at multiple places. Overall, Mate ran efficiently, being competitive with the alternative approaches. This indicates that our optimizations help the compiler to eliminate redundant indirections introduced by the ubiquitous IH sites. On the other hand, the 3.43x slowdown on DeltaBlue for the tracing experiment indicates remaining potential for performance improvements.

7. Discussion

Our empirical evaluation showed that the proposed optimization model removes most of the indirections introduced by the Mate IHs that normally a dynamic compiler cannot remove. The experiments also illustrate that a high local meta-variability leads to a severe degradation of the performance, implying that the flexibility of the MOP comes with a price and must be used with care. Nevertheless, it is worth mentioning that the experiments with megamorphic IH sites are purely synthetic. Currently, we do not anticipate concrete use cases using the MOP to this extent. As discussed in Section 4.1, we assume a low local meta-variability.

While our original work aimed to make all components of a VM reflective [5], TruffleMate focuses on the components for language semantics and the representation of objects. TruffleMate does not currently provide reflective capabilities for the memory manager, the garbage collector, or other (lower level) components. Consequently, our work does not assess the impact of our optimizations on IH sites concerning, *e.g.*, memory operations. The main reason is that building an efficient VM is an extremely time demanding task. We use the Truffle framework to implement a language efficiently, without having to implement low-level features such as memory management and compilation ourselves. However, for the next step of reflective capabilities, this means we need to adapt Truffle and Graal to support run-time interaction with memory allocation, garbage collection, and just-in-time compilation. We consider the corresponding effort out of the scope of this paper.

A potential threat to the validity of our evaluation is the selection of benchmarks. We may lead to conclusions that may not generalize to other programs. To mitigate this threat we selected a set of benchmarks included in the literature of the field [12, 14]. We also evaluated the MOP in use cases appearing in previous works (read-only [2], tracing [15]). It is worth noting however, that the field of reflective EEs is comparatively unexplored and the work on applications that harness the capabilities of reflective EEs has just started. Concerning the readonly experiment, we implemented Delegation Proxies following the guidelines of the paper [22]. Additional benchmarks challenging our optimization model (*e.g.*, more meta-variability) will provide more information about performance impact when our assumptions do not hold. However, we gathered preliminary insights about this kind of scenarios in the dynamic instantiation of metaobjects (readonly experiment) and the evaluation of mono and megamorphic IH sites.

8. Related Work

8.1 Reflective VMs

Ungar et. al. [4] introduced several optimizations for improving the performance of Self programs. Based on this work, Ungar and others worked on Klein, a metacircular VM for Self [20]. To the best of our knowledge, Klein is the only VM that tries to provide comprehensive reflective capabilities similar to Mate. Unfortunately, there are neither optimizations nor performance evaluations documented that could be used for a comparison.

Pinocchio [21] is a research prototype of a metacircular VM for Smalltalk that also shares some reflective characteristics with Mate. Unlike Mate, Pinocchio features reflective capabilities only for the execution component. However, Pinocchio does not impose a limit in the number of metalevels. With respect to performance, Pinocchio is only implemented as interpreter, which according to Verwaest [1] performs 250 times slower than a native Smalltalk interpreter.

CLOS is an object-oriented layer for LISP that features an advanced MOP, which is regarded as one of the most com-

plete in terms of introspection and intercession reflective capabilities on the language level [11]. In terms of performance, the CLOS MOP employs currying to facilitate memoization of lookups. However, this is not sufficient to eliminate all runtime overhead. Furthermore, by exposing memoization in the API, it shifts the burden of optimization from the runtime system to the developer, which we avoid in TruffleMate.

8.2 Efficient Run-time Adaptability

Based on Self-Optimizing Interpreters. Marr et al. [14] showed that dynamic compilation and dispatch chains can be used to remove the run-time overhead of reflective operations and IH for language level concepts. They showed that speculating on stable behavior for IH of field access or method invocation provides the compiler with enough information to optimize the operation to a normal base-level access or invocation.

Although we use the same fundamental technique (dispatch chains) for caching meta-level information, our work has significant differences with theirs: 1) The Mate approach aims at providing reflection at VM level, *i.e.*, targeting low-level operations of VM components instead of including only language-level operations. 2) In this context, Marr et al. consider only field accesses and method execution. Mate's MOP in addition allows to customize object layout, separates IH for method lookup and activation, and provides fine-grained IH for variable and argument access as well as a MOP for stack frames. This results in a much higher degree of flexibility and consequently a much higher number of IH sites exacerbating any inefficiencies in the runtime system. 3) We designed and described a new optimization model targeted to our particular IH that combines object layouts and meta-level semantics to minimize run-time cost. 4) Our evaluation also considers more comprehensive usages of the MOP. For instance, we evaluated dynamic adaptation scenarios of more realistic applications to support the claims that the ubiquitous IH sites can be optimized effectively.

Seaton et al. [17] optimizes out the overhead of debugging when it is not activated. This is achieved by wrapping operations with a kind of IH that checks if debugging for the current operation is activated. We use a similar approach for wrapping Mate's base-level nodes. In contrast to Mate, the intercession handling is simpler because it does not depend on metaobjects, and is fixed at compilation time. Thus, they do not need to handle the meta-variability problem.

General Approaches Several recent works delegate the burden of optimizing performance to the developer. Such approaches provide language abstractions to communicate hints to the compilers so that it can optimize execution. For instance, Shali and Cook [18] proposed the notion of hybrid partial evaluation, which is a combination of partial evaluation and compile-time metaprogramming. DeVito et al. [7] proposed Exotypes, which are similar in spirit but used staged programming, which is a form of run-time code generation. Finally, Asai [3] explores the usage of staged programming for improving the performance of a tower of meta interpreters with a powerful metaobject protocol.

As a common denominator, these approaches impose restrictions on the supported dynamicity, *i.e.*, meta-variability. In contrast, TruffleMate does not need any modification to the source code of the programs for doing the optimizations, and while high meta-variability can lead to slower execution, the expressivity is not restricted by the system.

9. Conclusions and Future Work

This work presents preliminary evidence suggesting that a reflective EE can run programs efficiently. This contradicts the belief that reflection at VM level is impractical, and opens the door to further research in this area. The main optimizations applied to TruffleMate, our reflective EE, are the speculation on low local meta-variability and the combination of the meta behavior with object layouts. The experiments showed that when the MOP is not activated, or when it only redefines individual VM operations, the mean peak performance overhead is zero. Furthermore, the overhead is at most 3.5x when dealing with elaborated, and on the fly, adaptation requirements.

With these results as a foundation, future work could explore how the Mate approach could be applied to further low-level VM components. The next step could be to explore incorporating reflective capabilities into the compiler. This could enable dynamic optimizations for specific use cases such as the customization of compilation/inlining thresholds, or the adaptation of the structure and length of dispatch chains. It could also further improve performance by leaving the choice of whether an optimization should be based on run-time or compile-time checks to the user. In performance critical parts, a user could explicitly disable or lower the degree of dynamicity supported by a MOP.

Acknowledgments

We would like to thank the reviewers of this paper for their constructive feedback which contributed to improve it. This work was partially supported by the projects ANPCYT PICT 2012-0724, ANPCYT PICT 2013-2341, ANPCYT PICT 2014-1656, ANPCYT PICT 2015-1718, UBACYT 384, CONICET PIP 2014/16 Nº11220130100688CO, CONICET PIP 2015/17 Nº11220150100931CO. Stefan Marr was funded by a grant of the Austrian Science Fund (FWF), project number I2491-N31.

References

[1] *Bridging the Gap between Machine and Language using First-Class Building Blocks*. PhD thesis, University of Bern, 2012.

[2] J.-B. Arnaud, M. Denker, S. Ducasse, D. Pollet, A. Bergel, and M. Suen. Read-only execution for dynamic languages. In *TOOLS*, pages 117–136. Springer, 2010.

[3] K. Asai. Compiling a reflective language using metaocaml. In *GPCE*, pages 113–122. ACM, 2014.

[4] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *OOPSLA*, pages 49–70. ACM, October 1989.

[5] G. Chari, D. Garbervetsky, S. Marr, and S. Ducasse. Towards fully reflective environments. In *Onward!* ACM, 2015.

[6] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *POPL*, Jan. 1984.

[7] Z. DeVito, D. Ritchie, M. Fisher, A. Aiken, and P. Hanrahan. First-class runtime generation of high-performance types using exotypes. In *PLDI*, pages 77–88. ACM, 2014.

[8] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley, 1983.

[9] M. Haupt, R. Hirschfeld, T. Pape, G. Gabrysiak, S. Marr, A. Bergmann, A. Heise, M. Kleine, and R. Krahn. The som family: Virtual machines for teaching and research. In *ITiCSE*, pages 18–22. ACM, 2010.

[10] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP*, pages 21–38. Springer, 1991.

[11] G. Kiczales and J. D. Rivieres. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[12] S. Marr and S. Ducasse. Tracing vs. partial evaluation: Comparing meta-compilation approaches for self-optimizing interpreters. In *OOPSLA*, pages 821–839. ACM, 2015.

[13] S. Marr, T. Pape, and W. De Meuter. Are We There Yet? Simple Language Implementation Techniques for the 21st Century. *IEEE Software*, 31(5):60–67, 2014.

[14] S. Marr, C. Seaton, and S. Ducasse. Zero-Overhead Metaprogramming: Reflection and Metaobject Protocols Fast and Without Compromises. In *PLDI*, pages 545–554. ACM, 2015.

[15] D. Rothlisberger, M. Harry, A. Villazon, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. Augmenting static source views in ides with dynamic metrics. In *ICSM '09*, pages 253–262, Sept 2009.

[16] G. Salvaneschi, C. Ghezzi, and M. Pradella. An analysis of language-level support for self-adaptive software. *TAAS*, 8(2):7, 2013.

[17] C. Seaton, M. L. Van De Vanter, and M. Haupt. Debugging at full speed. In *DLYA*, pages 1–13. ACM, 2014.

[18] A. Shali and W. R. Cook. Hybrid partial evaluation. In *OOPSLA*, pages 375–390. ACM, 2011.

[19] D. Simon, C. Wimmer, B. Urban, G. Duboscq, L. Stadler, and T. Würthinger. Snippets: Taking the High Road to a Low Level. *ACM Trans. Archit. Code Optim.*, 12(2):20:1–20:25, June 2015.

[20] D. Ungar, A. Spitz, and A. Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *OOPSLA*, pages 11–20. ACM, 2005.

[21] T. Verwaest, C. Bruni, D. Gurtner, A. Lienhard, and O. Nierstrasz. Pinocchio: Bringing reflection to life with first-class interpreters. In *OOPSLA*, pages 774–789. ACM, 2010.

[22] E. Wernli, O. Nierstrasz, C. Teruel, and S. Ducasse. Delegation proxies: The power of propagation. In *Modularity*, pages 1–12. ACM, 2014.

[23] A. Wöß, C. Wirth, D. Bonetta, C. Seaton, C. Humer, and H. Mössenböck. An object storage model for the truffle language implementation framework. In *PPPJ*, pages 133–144. ACM, 2014.

[24] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One vm to rule them all. In *Onward!*, pages 187–204. ACM, 2013.

[25] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing ast interpreters. In *DLS*, pages 73–82. ACM, 2012.