HAL
archives-ouvertes.fr

Zero-Overhead Metaprogramming

Stefan Marr, Chris Seaton, Stéphane Ducasse

▶ **To cite this version:**

Stefan Marr, Chris Seaton, Stéphane Ducasse. Zero-Overhead Metaprogramming: Reflection and Metaobject Protocols Fast and without Compromises. Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Jun 2015, Portland, OR, USA, France. <10.1145/2737924.2737963>. <hal-01141135>

HAL Id: hal-01141135
https://hal.inria.fr/hal-01141135

Submitted on 25 Sep 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

Zero-Overhead Metaprogramming

Reflection and Metaobject Protocols Fast and without Compromises

Stefan Marr
RMoD, Inria, Lille
France
stefan.marr@inria.fr

Chris Seaton
Oracle Labs / University of Manchester
United Kingdom
chris.seaton@oracle.com

Stéphane Ducasse
RMoD, Inria, Lille
France
stephane.ducasse@inria.fr

Abstract

Runtime metaprogramming enables many useful applications and is often a convenient solution to solve problems in a generic way, which makes it widely used in frameworks, middleware, and domain-specific languages. However, powerful metaobject protocols are rarely supported and even common concepts such as reflective method invocation or dynamic proxies are not optimized. Solutions proposed in literature either restrict the metaprogramming capabilities or require application or library developers to apply performance improving techniques.

For overhead-free runtime metaprogramming, we demonstrate that *dispatch chains*, a generalized form of polymorphic inline caches common to self-optimizing interpreters, are a simple optimization at the language-implementation level. Our evaluation with self-optimizing interpreters shows that unrestricted metaobject protocols can be realized for the first time without runtime overhead, and that this optimization is applicable for just-in-time compilation of interpreters based on meta-tracing as well as partial evaluation. In this context, we also demonstrate that optimizing common reflective operations can lead to significant performance improvements for existing applications.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]; D.3.4 [*Processors*]: Optimization

Keywords Metaprogramming, Reflection, Proxies, Metaobject Protocols, Meta-tracing, Partial Evaluation, Virtual Machines, Just-in-Time Compilation

1. Introduction

Reflection, dynamic proxies, and metaobject protocols provide developers with mechanisms to provide generic solutions that abstract from concrete programs. They are widely used to build frameworks for functionality such as persistence and unit testing, or as a foundation for so-called internal domain-specific languages (DSLs) [Fowler 2010]. In dynamic languages such as Python, Ruby, or Smalltalk, the runtime metaprogramming facilities enable DSLs designers to taylor the host language to enable more concise programs. Metaobject protocols (MOPs), as in Smalltalk or CLOS, go beyond more common metaprogramming techniques and enable for example DSL designers to change the language's behavior from within the language [Kiczales et al. 1991]. With these additional capabilities, DLSs can for instance have more restrictive language semantics than the host language they are embedded in.

However, such runtime metaprogramming techniques still exhibit severe overhead on most language implementations. For example, for Java's dynamic proxies we see 6.5x overhead, even so the HotSpot JVM has one of the best just-in-time compilers. Until now, solutions to reduce the runtime overhead either reduce expressiveness of metaprogramming techniques [Masuhara et al. 1995, Chiba 1996, Asai 2014] or burden the application and library developers with applying optimizations [Shali and Cook 2011, DeVito et al. 2014]. One of the most promising solutions so far is trace-based compilation, because it can optimize reflective method invocation or field accesses. However, we found that MOPs still require additional optimizations. Furthermore, trace-based compilation [Bala et al. 2000, Gal et al. 2006] has issues, which have prevented wider adoption so far. For instance, programs with many unbiased branches can cause trace explosion and very bimodal performance profiles. Another issue is the complexity of integrating tracing into multi-tier just-in-time compilers, which has being investigated only recently [Inoue et al. 2012].

To remove the overhead of reflection, metaobject protocols, and other metaprogramming techniques, we explore the applicability of *dispatch chains* as a simple and sufficient optimization technique for language implementations. Dispatch chains are a generalization of polymorphic inline caches [Hölzle et al. 1991] and allow optimization of metaprogramming at the virtual machine level. In essence, they allow the language implementation to resolve the additional variability introduced by runtime metaprogramming and expose runtime stability to the compiler to enable optimization. We demonstrate that they can be applied to remove all runtime overhead of metaprogramming. Furthermore, we show that they are applicable to both major directions for the generation of just-in-time (JIT) compilers for interpreters, i. e., meta-tracing [Bolz et al. 2009] and partial evaluation [Würthinger et al. 2013].

The contributions of this paper are as follows:[1]

- For the first time, we demonstrate the optimization of an unrestricted metaobject protocol so that all reflective overhead is removed.

[1] Artifact available: `http://stefan-marr.de/papers/pldi-marr-et-al-zero-overhead-metaprogramming-artifacts/`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PLDI'15, June 13–17, 2015, Portland, OR, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3468-6/15/06…$15.00.
http://dx.doi.org/10.1145/2737924.2737963

```
1 class ActorDomain : Domain {
2   fn writeToField(obj, fieldIdx, value) {
3     if (Domain.current() == this) {
4       obj.setField(fieldIdx, value);
5     } else {
6       throw new IsolationError(obj);
7     }
8   }
9   /* ... */ }
```

Listing 1. `ActorDomain` defines a metaobject that ensures that actors write only to objects that belong to them. The example is given in pseudo code for some dynamic object-oriented language.

- We identify dispatch chains as a simple and sufficient technique to remove the runtime overhead of metaprogramming.
- We evaluate them in the context of just-in-time meta-compilers for interpreters that identify their compilation units based on either meta-tracing or partial evaluation.
- We show that existing Ruby libraries can benefit directly without developer intervention.

2. Background

This section motivates the usefulness of metaobject protocols and assesses the cost of related runtime metaprogramming techniques on modern VMs. Furthermore, it briefly introduces self-optimizing interpreters as the foundation for our experiments.

2.1 Metaobject Protocols and DSLs

Metaobject protocols (MOPs) enable the adaptation of a language's behavior [Kiczales et al. 1991] by providing an interface that can be used by application and library developers. Among other use-cases, MOPs are useful as a foundation for internal DSLs. For example, a DSL to simplify concurrent programming might try to free programmers from low-level issues such as race conditions. Frameworks built in systems without a MOP such as Akka[2] do not guarantee isolation between actors, and so expose the risk of race conditions. This leaves the burden of correctness with the programmer.

The *ownership-based metaobject protocol* (OMOP) proposed by Marr and D'Hondt [2012] is an example of such a MOP. It provides framework and DSL implementers with abstractions to guarantee, e. g., isolation between actors. It provides an interface to change the semantics of state access and method invocation so that one can guarantee that an actor accesses only objects that belong to it. With the OMOP, an actor framework can associate each actor with an instance of the class `ActorDomain` (cf. listing 1). On line 2, this metaobject implements the `writeToField()` handler that is called for each field write and ensures that an object can only be written if it belongs to the current domain. If the test on line 3 succeeds, line 4 performs the write to the object's field, otherwise an exception is raised.

This example illustrates the conciseness and power of such runtime MOPs. Furthermore, it indicates that removing the runtime overhead of reflective operations such as `setField()` and the overall cost of the `writeToField()` handler is crucial for performance. Otherwise, every field access would have a significant performance cost.

2.2 State of the Art in Metaprogramming

Reflection in Common Systems. As mentioned earlier, reflective operations are used for a wide range of use cases. In dynamic languages such as Python, Ruby, or Smalltalk, they are widely used to solve problems in a concise and generic way. For instance, idiomatic Ruby embraces metaprogramming so that reflective method invocation and `#method_missing` are used in common libraries. As concrete example, in the Ruby on Rails web framework, the `TimeWithZone` class wraps a standard time value and uses `#method_missing` to delegate method calls that do not involve the time zone to the standard `Time` object. This delegation pattern is also common to Smalltalk. In existing implementations of both languages this imposes a performance overhead compared to direct method calls. A work-around often used is to have `#method_missing` generate methods on the fly to perform the delegation directly on subsequent calls, but this obfuscates the more simple underlying logic—that of redirecting a method call.

Another example from the Ruby ecosystem is psd.rb,[3] a widely used library for processing Photoshop images. The Photoshop file format supports multiple layers that are then composed using one of about 14 functions that accepts two colors and produces a single composed color. As the particular compose operation to run is determined by the file contents and not statically, a reflective call is used to invoke the correct method. Without optimization for reflective operations, this is a severe performance bottleneck.

In the case of psd.rb, this has led to the development of `psd_native`, which replaces performance critical parts of psd.rb with C extensions. However this raises the barrier to contribution to the library and reduces maintainability. Other approaches to avoid the reflective call include creating a class for each compose operation and having each define a compose method. However, since this leads to very repetitive code, metaprogramming is generally preferred in the Ruby community and considered idiomatic [Brown 2009, Olsen 2011].

Performance. Workarounds such as `psd_native` and patterns to avoid metaprogramming foster the intuition that runtime metaprogramming is slow. To verify this intuition, we investigated reflective method invocation and dynamic proxies in Java 8, with its highly optimizing HotSpot JVM, and PyPy 2.3, with its meta-tracing just-in-time compiler. We chose those two to compare their different approaches and get an impression of the metaprogramming overhead in widely available VMs.

For Java, we used the generic `Method.invoke()` API from `java.lang.reflect` and the `MethodHandle` API from `java.lang.invoke`, which was introduced as part of the `invokedynamic` support for dynamic languages in Java 7 [Rose 2009]. Our microbenchmarks add the value of an object field to the same object field accessed via a getter method. The getter method is then either invoked directly or via one of the reflective APIs. The methodology is detailed in section 4.2.

Compared to the direct method invocation, the invocation via a `MethodHandle` causes up to 7x overhead. Only when the `MethodHandle` is stored in a `static final` field, we see no overhead. Thus, the usability of method handles is rather limited in normal applications. Language implementations, for which method handles were designed, use bytecode generation to satisfy this strict requirement. The `java.lang.reflect` API is about 6x slower than the direct method invocation. Its performance is independent of how the `Method` object is stored.

Java's dynamic proxies (`reflect.Proxy`) have an overhead of 6.5x. Our microbenchmark uses an object with an `add(int b)` method, which reads an object field and then adds the parameter. Thus, the overhead indicates the cost of reflection compared to a situation where a JIT compiler can inline the involved operations normally, and thus eliminate all non-essential operations.

[2] *akka*, Typesafe, access date: 28 October 2014 `http://akka.io/`

[3] *psd.rb*. Its 180 forks on GitHub indicate wide adoption. Access date: 05 November 2014 `https://github.com/layervault/psd.rb`

The PyPy experiments were structured similarly. However, since Python's object model is designed based on the notion of *everything is a field read*, it is simpler to perform reflective method invocations. In addition, the meta-tracing approach has some advantages over method-based compilation when it comes to runtime metaprogramming.

Concretely, neither reflective method invocation nor dynamic proxies have runtime overhead in PyPy. To identify the limits of RPython's meta-tracing [Bolz et al. 2009, Bolz and Tratt 2013], we experimented with a simple version of the OMOP (cf. section 2.1). Implemented with proxies, we restricted the MOP to enable redefinition of method invocations based on the metaobject. While proxies alone do not show overhead, in this setting we see an overhead of 49%. The resulting traces show that read operations on the metaobjects and related guards are not removed from the most inner benchmark loop. Such optimization would require value-based specialization, e. g., by communicating constants via an API to the compiler.[4] Without such mechanisms, guards remain for all base-level operations that interact with the MOP and their overhead becomes an issue not only for microbenchmarks.

From these experiments, we conclude that current VMs require better support for runtime metaprogramming, independent of the underlying JIT compilation technique. While PyPy's meta-tracing fares better than HotSpot's method-based compilation and removes the overhead of reflective operations and dynamic proxies, it still requires better support for metaobject protocols such as the OMOP.

2.3 Self-optimizing Interpreters

Self-optimizing interpreters [Würthinger et al. 2012] are an approach to language implementation that uses abstract-syntax trees (ASTs). We use them for our experiments detailed in section 4.1. In such interpreters, AST nodes are designed to specialize themselves during execution based on the observed values. Similar to bytecode quickening [Casey et al. 2007, Brunthaler 2010], the general idea of self-optimizing interpreters is to replace a generic operation with one that is specialized and optimal to the context in which it is used. For example, one can speculate on future executions doing similar operations, and thus, specialize an operation on the types of its arguments or cache the result of method or field lookups in dynamic languages. Speculating on types of values has multiple benefits. On the one hand it avoids boxing or tagging of primitive values, and on the other hand it enables the specialization of generic operations to optimal versions for the observed types. When applied consistently, local variables can be specialized as well as object slots in the object layout [Wöß et al. 2014]. Complex language constructs can also be optimized. For instance Zhang et al. [2014] demonstrate how to optimize Python's generators based on AST specialization and peeling of generators. Similarly, Kalibera et al. [2014] use views on R vectors to realize R's complex data semantics with good performance.

While this technique enables the optimization of interpreter performance based on the high-level AST, it enables also the generation of efficient native code by meta JIT compilers. Truffle is a Java framework for these interpreters. Combined with the Graal JIT compiler [Würthinger et al. 2013], interpreters can reach performance that is of the same order of magnitude as Java on top of HotSpot [Marr et al. 2014]. 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. This approach is somewhat similar to RPython's meta-tracing, because it also works on the level of the executing interpreter instead of the level of the executed program as with traditional JIT compilers. With either meta-tracing or partial evaluation as JIT compiler techniques, self-optimizing interpreters can be used to build fast language implementations.

[4] RPython provides the promote() operation for this purpose, but it is not exposed to the Python-level of PyPy.

3. Using Dispatch Chains for Zero-Overhead Metaprogramming

This section discusses how *dispatch chains* are applied so that runtime metaprogramming has zero overhead. As a first step, we detail how simple reflective operations are optimized. Only afterwards, we discuss how more complex metaobject protocols are optimized. Throughout, we contrast partial evaluation and meta-tracing to clarify their different needs.

3.1 Reflective Operations

For this study, we show how reflective method invocation, object field access, global variable access, and dynamic handling of undefined methods can be realized. These operations are common to dynamic languages such as JavaScript, PHP, Ruby, and Smalltalk. A subset of these operations is also found in more static languages such as C# or Java.

To emphasize that the presented techniques are language agnostic, we use pseudo code of a dynamic object-oriented language. In this language, methods can be invoked reflectively by calling `obj.invoke(symbol, args)`, which use the symbol of the method name to look up the method and invoke it with the given arguments array. In case a method is invoked on an object that does not implement it, the `methodMissing(symbol, args)` method is called instead, which for instance allows to implement dynamic proxies.

For reflective field access, objects have `getField(idx)` and `setField(idx, value)` methods, which allows a program to read and write fields based on an the field's index. Global variables are accessed with the `getGlobal(symbol)` and `setGlobal(symbol, value)` methods. To simplify the discussion in this paper, we assume that reflective methods on objects are not polymorphic themselves. However, this is not a requirement and in our case studies, reflective methods can be overridden like any other method.

Optimizing Reflective Method Invocation with Dispatch Chains. Direct method calls are done based on the method name that is fixed in the program text, and consequently represented as a constant in the AST. For such calls, the number of methods invoked at a specific call site in a program is typically small [Deutsch and Schiffman 1984, Hölzle et al. 1991]. The reflective call with `invoke()` takes however a variable argument. For optimization, we assume that the number of different method names used at a reflective call site are also small. Thus, the idea is to apply a generalization of polymorphic inline caches [Hölzle et al. 1991] for the method name, too. This generalization is called a *dispatch chain*.

Dispatch chains are a common pattern in self-optimizing interpreters. They generalize polymorphic inline caches from a mechanism to record type information and avoid method lookup overhead in dynamic languages to a mechanism to cache arbitrary values as part of AST nodes in a program. Therefore, in addition to call sites, they apply to a wide range of possible operation sites. For instance, the Truffle DSL [Humer et al. 2014] uses dispatch chains to resolve the polymorphism of different node specializations, e. g., for basic operations such as arithmetics. The object storage model [Wöß et al. 2014] uses dispatch chains to resolve the polymorphism of different shapes. Here, we explore their applicability for reflective operations to resolve their extra degree of variability and expose information about stable runtime behavior to enable JIT compiler optimizations.

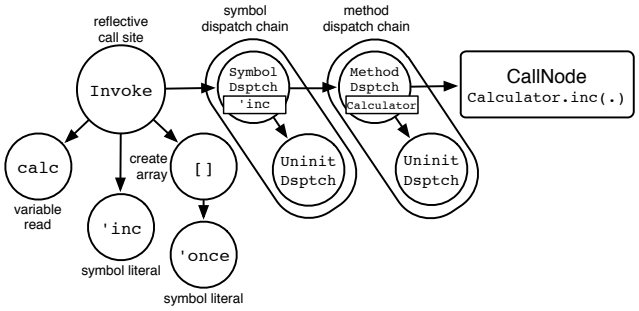

Figure 1. AST for: `calc.invoke('inc, ['once])` with dispatch chain at the `invoke()` call site. The first level of the dispatch chain records the observed symbols for method names as indicated with the `'inc` symbol as part of the cached symbol node. The second level then is a classic polymorphic inline cache for direct method dispatch, which caches the objects class (`Calculator`) for comparison and refers to the `CallNode` representing the actual method for direct invocation.

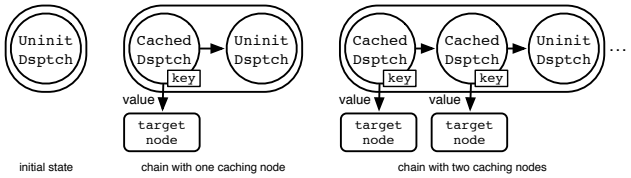

Figure 2. Buildup of a dispatch chain. A chain starts with only an uninitialized node, which specializes itself to a cached node and a new uninitialized node. A cached node determines whether it applies based on a key value kept in the node. If it applies, the target node is executed, otherwise it delegates to the following node in the chain.

For `invoke(symbol, args)`, we need to embed the dispatch chain for the reflective invocation as part of the call site of `invoke()` in the AST. Figure 1 depicts the AST for a reflective invocation of `inc()` on a `Calculator` object with an argument array containing the symbol `'once`. The `Invoke` node represents the call site as part of the method in which `invoke()` is used. Beside having the subexpression nodes to determine the object, the symbol for the method name, and the argument array, the `Invoke` node also has a nested dispatch chain. On the first level, we record the method name symbols and on the second level, the dispatch chain records the object's classes observed at runtime and cache the corresponding lookup results, i. e., the methods to be invoked.

Generally, a dispatch chain consists of a small number of linked nodes (cf. fig. 2). Except the last node, each one keeps a key and a target node. The key is checked whether it currently applies, e. g., the key is the method name and it is compared to the symbol used for reflective invocation. The target node can be an arbitrary node for instance a nested dispatch chain, a basic operation, or a method call node that is execute in case the check succeeded. Otherwise, the next node in the chain is executed. The last node is an uninitialized node that specializes itself on execution. Usually, the number of nodes in a chain are limited and if the limit is reached the chain is either terminated by or completely replaced with a generic node that can handle all dynamic values.

In the context of reflective method invocation, the nested dispatch chain fulfills two purposes. On the one hand, it allows an interpreter to cache method lookups also for reflective invocation. By nesting the dispatch chains, it becomes possible to first resolve the variability of the symbol representing the method name, and then use a classic polymorphic inline cache. On the other hand, the dispatch chain directly exposes runtime constants to an optimizing compiler. In the ideal situation, a call site records only a single method name symbol as in our example, and is used only with one class of objects, which makes it a monomorphic call site. This enables compilers to use the constant values of the dispatch chain nodes to for instance inline methods and benefit from the optimization potential enabled by it.

Optimizing `methodMissing()`. Since dispatch chain nodes can do arbitrary operations, `methodMissing(symbol,args)` can be handled by a special node for the method dispatch chain. This means, in case the method lookup fails, a `MethodMissing` node is added to the chain, and `methodMissing()` becomes part of the normal dispatch and benefits from the related optimizations. The `dispatch(.)` method sketched in listing 2 is part of this node. First, line 8 checks whether the node is applicable by testing that the class of the object is the expected one. If it is the expect class, the `methodMissing()` handler needs to be called. On line 9, the argument array is constructed to pass the object, the method name, and the original arguments of the method call to the handler, which is finally called on line 10. In case the object is of another class, execution is delegated to the next node in the chain.

```
1  class MethodMissing(DispatchNode):
2    final _expected_class  # cache key
3    final _method_name     # name of missing method
4    final _mth_missing     # methodMissing() handler
5    child _next            # next node in chain
6
7    def dispatch(frame, obj, args):
8      if obj.get_class() == _expected_class):
9        args_arr = [_method_name, args]
10       return _mth_missing.call(frame,obj,args_arr)
11     else:
12       return _next.dispatch(frame, obj, args)
```

Listing 2. `methodMissing()` handler. It is a special node class for the standard method dispatch chain. The cache key is the receiver's class. The cache's value is the handler method, which can then be invoked directly.

With this implementation approach, the failing lookup that traverses the whole superclass chain is avoided and the lookup of the `methodMissing()` handler is avoided as well. Furthermore, the handler is exposed as a constant to facilitate the same optimizations as for direct method calls.

Reflective Field and Global Access Operations. To optimize field accesses as well as accesses to global variables, we also rely on dispatch chains. First a dispatch chain is used to resolve the indirection used by the reflective operation and then the actual operation node can specialize itself in that context. This means, for the field access operations `getField(idx)` and `setField(idx, value)`, we use a dispatch chain on the field index, which works similar to the dispatch chain for method names. The target nodes, i. e., the field read or write node can then specialize itself, for instance based on the type stored in the object field.

For accessing globals with the `getGlobal(symbol)` and `setGlobal(symbol, value)` methods, the dispatch chain is on the symbol and thus corresponds to the one for reflective method invocation. The target nodes in this case are the nodes for accessing the global, which for instance can cache the association to avoid a runtime lookup in the hash table of globals.

Partial Evaluation versus Meta-Tracing. Independent of whether partial evaluation or meta-tracing is used, the interpreter performance benefits from the use of dispatch chains. Similar to classic

```
1  class DirectInvokeNode(AstNode):
2    final _method_name  # name of method
3    child _dispatch     # dispatch chain
4
5    def execute(frame, obj, args):
6      if jit.is_tracing(): # only used while tracing
7        method = _lookup_method(obj)
8        if method:
9          return method.call(frame, obj, args)
10       else:
11         return _method_missing(frame, obj, args)
12     else: # uses the chain while interpreting
13       return _dispatch.dispatch(frame, obj, args)
14
15   @jit.elidable
16   def _lookup_method(obj):
17     return obj.get_class().lookup(_method_name)
18
19   def _method_missing(frame, obj, args):
20     hdlr = obj.get_class().lookup("methodMissing")
21     args_arr = [_method_name, args]
22     return hdlr.call(frame, obj, args_arr)
```

Listing 3. Sketch of direct method invocation and `methodMissing()` handling during tracing. During normal interpretation, the dispatch chain is used, only during tracing a simple direct path is used that relies on `@jit.elidable` for optimization.

polymorphic inline caches, their use avoids for instance repeated lookup overheads.

When compilation is based on partial evaluation to determine the compilation unit, the dispatch chains are one of the main mechanisms to communicate optimization opportunities to the compiler. Since the chains expose the cached keys and values as constants to the optimizer, they enable further optimizations such as inlining.

For meta-tracing, the situation is slightly different. While dispatch chains speed up the interpreter, it can be beneficial to by-pass them during the trace-recording and rely for instance on RPython's annotations to communicate constant values and elidable function invocations to the optimizer. While this slows down trace recording, it can result in fewer guards in optimized traces.

Considering the `methodMissing()` example, during tracing it is beneficial to avoid the dispatch chain and instead use a fast path for direct method invocation, including explicit handling of the `methodMissing()` case.

Listing 3 sketches the implementation of direct method invocation. The `execute()` method on line 5 first checks whether the interpreter currently records a trace for compilation, or executes normally. If it is recording a trace, it performs a direct lookup of the method to be invoked on line 7. The function `_lookup_method()` is marked as elidable (cf. line 15), so that the tracer knows that the function returns the same value for the same parameters. Thus, in the same trace, the lookup does not need to be repeated. Assuming the lookup failed, `_method_missing()` is called and the `methodMissing()` handler is invoked (cf. line 19). This corresponds to the implementation in the dispatch chain sketched in listing 2. The difference here is that the trace does not need to record the chain traversal, and thus results in fewer runtime guards.

Generally this means that if interpreter performance is not relevant, dispatch chains are not always necessary to optimize reflective operations for meta-tracing. This confirms also the brief assessment that tracing eliminates the overhead of reflective method invocation and dynamic proxies in PyPy (cf. section 2.2).

3.2 Optimizing the OMOP

To optimize metaobject protocols such as the OMOP, dispatch chains can be used to resolve the runtime-variability of metaobjects, and thus, to expose the desired optimization opportunities. On the base level, all operations for method invocation, field, and global accesses trigger the intercession handlers of the metaobject,[5] which is linked to a base-level object. With dispatch chains over metaobjects, the intercession handler corresponding to a specific operation can be exposed as constants to the optimizers. Thus, similar to the reflective operations that use dispatch chains over method names or field indexes, the base-level operations dispatch over metaobjects. The main conjecture here is that a program uses only a fraction of the possible dynamicity, which leads to similar opportunities as classic polymorphic inline caches provide for method calls.

In addition to exposing intercession handlers as runtime constants, the use of dispatch chains provides a simple mechanism to handle the standard semantics for method invocations, field, and global accesses, too. In the case where an intercession handler has not been customized, the dispatch chain can hold the node implementing the standard semantics instead of having to dispatch to the intercession handler implementing those reflectively. Thereby, the overhead of the MOP is reduced also during interpretation and without requiring compiler optimizations. This solution corresponds to implementing `methodMissing()` as part of the dispatch chain. A brief example is given and detailed in fig. 3.

Partial Evaluation versus Meta-Tracing. To optimize metaobject protocols such as the OMOP, the compilation technique does not change the requirements for the implementation. In our experience, the dispatch chains work well for partial evaluation and meta-tracing. They give the necessary flexibility to determine the runtime constants and thereby enable various compiler optimization.

4. Evaluation

To assess whether the use of dispatch chains is sufficient to remove the runtime overhead of metaprogramming, we evaluate the perfor-

[5] Since the OMOP does not support meta recursion, code execute on the meta level can use separate ASTs containing only the direct operations that are not changeable by metaobjects. This avoids runtime overhead at the meta level.

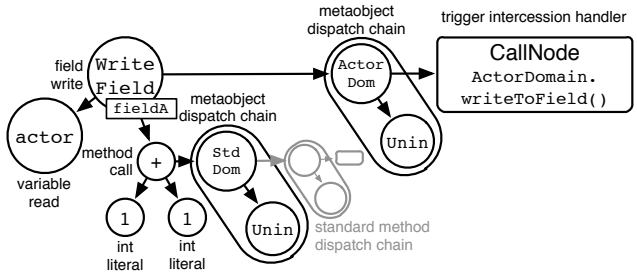

Figure 3. AST for `actor.fieldA := 1 + 1`, a field write on an actor object. The method invocation and field write operation are subject to the OMOP. Assuming that the actor object is owned by the `ActorDomain` (cf. listing 1), the dispatch chain caches the corresponding `writeToField()` intercession handler and calls it instead of writing to the object field. The + method invocation however is done on the `1` literal, which is owned by the standard metaobject. Thus, the dispatch chain can directly nest the standard method dispatch chain for the invocation.

mance of a Smalltalk extended with the OMOP (cf. section 2.1), measure the performance impact on JRuby and the psd.rb image processing library, and examine the native code generated for reflective operations and dynamic proxies. The evaluation focuses on the peak performance, i. e., stable state performance for a given benchmark.

4.1 SOM Smalltalk and JRuby+Truffle

For our experiments, we rely on SOM, a Smalltalk designed for teaching and research on VM techniques [Haupt et al. 2010], and JRuby+Truffle [Seaton et al. 2014], a Ruby implementation for the JVM that leverages the Truffle framework to outperform the other Ruby implementations.

SOM: Simple Object Machine. SOM is designed to avoid inessential complexity, but includes fundamental language concepts such as *objects*, *classes*, *closures*, and *non-local returns*. In 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 the experiments it also implements common reflective operations such as method invocation and field access, as well as the OMOP.

To cover meta-tracing as well as partial evaluation as JIT compilation techniques, we use two SOM implementations. SOM_{MT} is implemented in RPython and the corresponding tool-chain generates a JIT compiler based on meta-tracing. SOM_{PE} is implemented in Java on top of the Truffle framework and the Graal compiler with its partial evaluation approach. Both are self-optimizing interpreters and reach performance of the same order of magnitude as Java on top of the HotSpot JVM [Marr et al. 2014], and thus, are suitable to assess the performance of runtime metaprogramming.

JRuby+Truffle. Since SOM is rather academic, we also investigate the performance potential in the context of JRuby+Truffle, which aims to be a fully compliant Ruby implementation, and thus is significantly more complex. Being part of the JRuby code base, it is comparable with other industrial-strength language implementations from the perspective of complexity. On a set of numerical and semi-numerical benchmarks, JRuby+Truffle outperforms Ruby 2.1 and other Ruby implementations on most benchmarks, often by more than an order of magnitude.[6]

To gain performance that is of the same order of magnitude as Java, JRuby+Truffle uses Truffle and self-optimization for instance to type-specialize basic operations such as arithmetics and comparisons [Humer et al. 2014], to optimize object field access [Wöß et al. 2014], or to remove the overhead of debugging related functionality [Seaton et al. 2014].

Similar to other dynamic languages such as Smalltalk, it offers a wide range of metaprogramming facilities including reflective method invocation with `#send`, checks whether a method is implemented with `#respond_to?`, and `#method_missing` to handle the case that a method is not implemented. These operations are optimized with dispatch chains based on the approach discussed in section 3.

4.2 Methodology

To account for the non-determinism in modern systems as well as the adaptive compilation techniques in RPython and Truffle combined with Graal and HotSpot, each reported result is based on at least 100 measurements after a steady state has been reached. To determine when a steady state is reached, each benchmark is executed between 350 and 500 times within the same VM instance. The steady state is determined informally by examining plots of the measurements for each benchmark and selecting a suitable range of measurements that does not show signs of compilation.

[6] Please see the supplement material with the performance numbers for JRuby+Truffle.

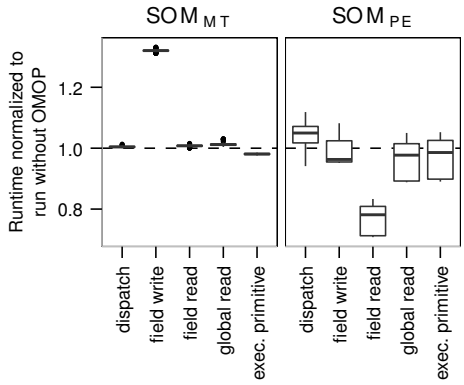

Figure 4. Microbenchmarks to assess the OMOP's overhead

The same approach was used for the PyPy results reported in section 2.2, while the Java results were determined using JMH,[7] which reports the number of operations per second after warmup. The reported result is an average over 100 reported measurements.

The benchmark machine used has two quad-core Intel Xeons E5520, 2.26 GHz with 8 GB of memory and runs Ubuntu Linux with kernel 3.11, PyPy 2.3.1, and Java 1.8.0_11 with HotSpot 25.11-b03.

4.3 Performance of an Unrestricted Metaobject Protocol

One of the main goals of this work is to make unrestricted metaobject protocols such as the OMOP (cf. section 2.1) efficient. Thus, despite the ability to change metaobjects, i. e., the language semantics of base-level objects at runtime, the cost to change semantics of field accesses or method invocations should be reduced to the cost of the involved base-level operations and avoid any overhead for the involved metaobject handlers and reflection.

As explained in section 3.2, we speculate on the metaobject of a base-level object staying the same to be able to eliminate overhead of reflection and the involved metaobject handlers. In the evaluation, we want to assess the overhead in the best-case scenario that the metaobject does not change so that we can see the impact of the residual guards and checks on the peak performance.

Overhead for Metaprogramming. To assess the scenario where the OMOP is used to change the language's behavior, we measure the runtime overhead on simple microbenchmarks. In each of them, one aspect of the language's behavior is changed for a selected object. Thus, either method dispatch, field read, field write, reading a global value, or executing a primitive function provided by the interpreter. To see whether any unnecessary overhead remains, the changed behavior increments the result of, e. g., the field read operation simply by adding one. The baseline benchmark for the comparison does the same operations, i. e., a field read and adding one executes without triggering the MOP. Thus, the benchmarks measure the cost of moving a base-level operation to the meta level. Ideally, this does not incur any cost even so that it requires calling the metaobject's handlers and doing the customized operation reflectively.

Figure 4 shows the measured results as boxplots indicating median, 25th, and 75th percentiles. Measurement errors aside, the

[7] *JMH is a Java harness for building, running, and analyzing micro benchmarks*, OpenJDK, access date: 28 August 2014 `http://openjdk.java.net/projects/code-tools/jmh/`

results show that moving between base and meta level does not incur any overhead, and thus, dispatch chains are sufficient to optimize MOPs as dynamic as the OMOP.

On SOM_{MT} the benchmark for field writes sticks out. An analysis of the resulting traces shows that the optimizer was able to remove the overhead of the reflective operations as in the other microbenchmarks. The differences are another type of guard and a different type of add instruction being used. However, the overall number of guards and operations is identical, but on the used hardware, the selected instructions have different performance. We attribute that to the fact that RPython's backend does currently not take the different cost of instructions into account. For the other benchmarks, RPython as well as Graal produce the same machine code. The only difference are in memory offsets. Ignoring the anomaly, we conclude that the proposed optimizations are sufficient on top of meta-tracing as well as partial evaluation to enable the optimizers to remove the reflective overhead and minimize the necessary runtime guards.

Note, compared to the experiment with PyPy, which showed an overhead of 49% (cf. section 2.2), our approach eliminates the overhead completely.

Inherent Overhead. While we were able to show that moving operations from the base level to the meta level does not incur any overhead, the question remains what the inherent overhead for the support of the OMOP is. Since metaobjects can change at runtime, some guards need to remain in the program to ensure the correct semantics of the OMOP. To measure the inherent overhead these guards cause, we assess the runtime impact on benchmarks that are executed with the OMOP but without using it to change the language's behavior. Thus, we assess whether the proposed optimizations are sufficient to remove the overhead of metaoperations and minimization of guards. We use 22 benchmarks. DeltaBlue and Richards have been used by various VM implementers to represent polymorphic object-oriented programs. Mandelbrot and NBody measure whether the optimizer can reduce object-oriented programs to the basic numerical operations. The other benchmarks are kernels that stress a wide range of VM aspects such as garbage collection, string performance, data movement, as well as basic language features such as method invocation, loop constructs, and recursion.

Figure 5 shows that in practice there is an overhead to ensure the OMOPs semantics, i. e., possibly changing metaobjects at runtime. On average, the overhead is 3.6% (min. −0.9%, max. 19.2%) on SOM_{MT} and 8.6% (min. −7.2%, max. 38.2%) on SOM_{PE}.

Smaller benchmarks are generally as fast as their versions running without the OMOP. Larger benchmarks such as DeltaBlue and Richards exhibit about 10–25% runtime overhead, which comes solely from remaining guards. All overhead such as extra allocations for reflective argument passing is removed. In case reflective overhead would remain, it be at least in the range of 5–10x.

To conclude, dispatch chains are sufficient to remove the overhead of reflective operations completely. For MOPs such as the OMOP, there can be a small inherent runtime cost, since removing the remaining guards would compromise its correctness.

4.4 Performance Benefits for JRuby+Truffle

To evaluate the impact of optimized dispatch chains on real code used in production applications, we measure the impact of using dispatch chains to optimize reflective operations in JRuby+Truffle. We use 18 image composition kernels from the psd.rb library as benchmarks. The compose operations that produce a single color value from multiple inputs are key for performance as they are run for every pixel in an image, and in psd.rb these are implemented using multiple metaprogramming operations. Following the Ruby philosophy of choosing convenient implementations over creating extra abstraction with classes, the library developers chose to pass the name of the composition operation as an argument, which is then used by the reflective method invocation `#send`. Within each composition operation, color value manipulation methods are called that are not part of the object. These are caught via `#method_missing`, filtered with `#respond_to?` and delegated with `#send`, in a form of ad hoc modularity. In an extreme case for each pixel in the image there are 7 calls to `#send` and 6 each to `#method_missing` and `#respond_to?`. Although this structure may not be optimal for other implementations of Ruby, and could be alternatively expressed using existing tools for modularity, this is the code that the psd.rb developers found was most clear for their purpose and it is common for Ruby programs.

To assess the benefit of the optimization, the benchmarks compare the performance with and without the use of dispatch chains. Thus, we assess the effectiveness of the optimization in the context of complex interactions of reflective operations on real code. Without the optimization, the optimizer is not able to cache method lookups and inline method calls to enable further optimizations.

Figure 6 shows that using dispatch chains gives between 10x and 20x speedup over unoptimized calls. Thus, they give a significant performance benefit without requiring the developer to change the implementation style or to use native extensions.

4.5 Compilation of Reflection and Dynamic Proxies

Finally, we investigate the compilation result for the use of reflective operations and compare it with the result for the direct operations. To assess the results for meta-tracing as well as partial evaluation, we use microbenchmarks on top of SOM_{MT} and SOM_{PE}. The microbenchmarks use a counter object that implements a `increment` method to increment an integer. The baseline for comparison calls `increment` directly. The `PerformAdd` benchmark calls the increment method reflectively. For `DnuAdd` the counter does not implement `increment` and instead uses SOM's missing-method handler (`#doesNotUnderstand:`) to do the integer increment. `DnuPerformAdd` combines missing-method handling with a reflective method call. The `ProxyAdd` benchmark combines missing-method handling with reflective calls to assess the overhead of dynamic proxies built with it.

For each of these benchmarks, we assessed the generated machine code for SOM_{MT} as well as SOM_{PE}. In either case, the compilation results for each of the benchmarks is identical to the non-reflective counter part, leaving memory offsets aside. Thus, the generated instructions are the same and the optimizers were able to completely remove the overhead of reflection. Neither reflective method invocation nor missing-method handling with its additional allocation of arrays for argument passing resulted in additional machine code instructions. Thus, we conclude that the dispatch chains expose the essential information to the optimizers that enable them to generate the same machine code they generate for non-reflective operations. The performance measurements of the benchmarks are depicted in fig. 7 as boxplots. The expected result is that these benchmarks perform identical to their non-reflective counter parts and thus results are on the 1-line. Since the machine code is identical, we attribute the measured difference to memory offset differences, measurement inaccuracies, garbage collection, and other influences outside of our experimental control.

5. Discussion

Our evaluation showed that dispatch chains are sufficient to eliminate the overhead of metaprogramming. It enables the optimizers in the context of meta-tracing as well as partial evaluation to eliminate the cost of reflective operations. However, the dynamic semantics of the MOP might require runtime checks for correctness, which cannot be eliminated.

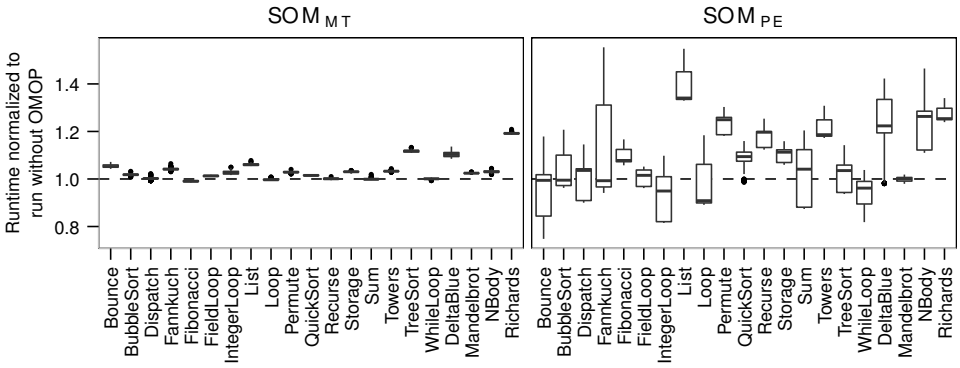

Figure 5. Overhead of running benchmarks with the OMOP, but without changing language behavior.

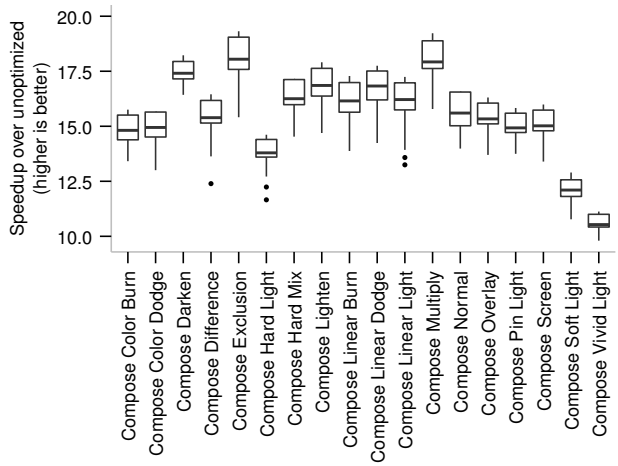

Figure 6. Speedup on psd.rb image composition kernels from optimizing reflective operations.

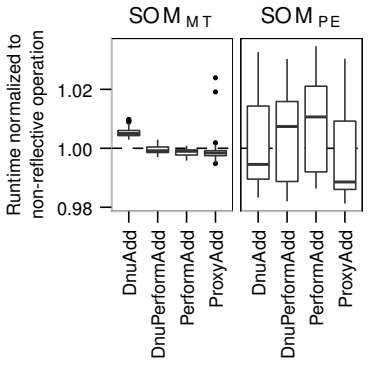

Figure 7. Performance of using reflective operations and proxies.

In this work, we used dispatch chains together with self-optimizing interpreters. However, we believe that the notion of generalized polymorphic inline caches is applicable to VMs in general, but demonstrating their effectiveness remains future work. When building self-optimizing interpreters on top of meta-tracing or partial evaluation, we did not notice any particularities that were specific to one of the compilation techniques. However, we saw indications that their heuristics might require adaptations. Meta-tracing uses trace length as criterium for compilation and partial evaluation also relies on size-based heuristics. Since dispatch chains introduce additional operations, such heuristics might require further fine tuning. For the benchmarks used in this paper, the standard heuristics were however sufficient.

Since the evaluation of dispatch chains focused on peak performance, we did not evaluate their impact on interpreted performance or on other dimensions of performance beyond running time, such as memory consumption. Generally, they are caching data structures and thus consume additional memory. However, since the length of dispatch chains is typically bounded to avoid the overhead of long linear searches, they introduce only a memory overhead with an upper bound per AST. Furthermore, we expect them to also improve interpreter performance. On the one hand, traversing a dispatch chain is less complex than the lookup operations of languages such as Ruby and Smalltalk, and on the other hand, we assume that most dispatch points are monomorphic. Similar to the observations made for method dispatches [Hölzle et al. 1991], we expect programs to exhibit much less dynamic behavior than that which the language allows for. Thus, reflective operations have a high chance to be monomorphic and caching will be effective. Highly polymorphic or megamorphic usage of reflective operations can also be solved by inlining to contextualize these operations. Thus, methods that access various fields reflectively can be inlined into the callers to expose that each caller only accesses a small number of fields. On top of Truffle, such AST-based inlining is done by the framework and in RPython the meta-tracing also provides the necessary contextualization. In the future, it should however be verified that reflective operations and MOPs exhibit similar restricted variability at runtime.

6. Related Work

As mentioned before, dispatch chains are a common pattern in self-optimizing interpreters and have been used for other optimizations such as an efficient object storage model [Wöß et al. 2014] or a fast C [Grimmer et al. 2014]. They are similar to the method handle infrastructure introduced with Java's `invokedynamic` [Rose 2009]. Method handles can also be used to implement polymorphic

inline caches. Most relevant for this paper is however the insight that they can be used to remove the overhead of reflective operations and complex metaobject protocols, which to our knowledge has not been demonstrated before. On the contrary, below we discuss a number of approaches that all restrict the reflective power or burden the application or library level with performance optimizations.

Compile-time Metaprogramming. Compile-time metaprogramming techniques try to preserve the expressiveness of their runtime counterparts but improve performance by applying analyses and optimizations on the metaprograms, which then are statically compiled to obtain performance properties that are ideally on a par with programs that do not use metaprogramming. Chiba [1996] proposed Open C++, a compile-time MOP, to enable changes to the language's behavior. Unfortunately, to enable the optimization of reflective operations, the MOP needs to be severely restricted and for instance metaobjects cannot change at runtime. Beside MOPs, compile-time metaprogramming can also take forms similar to templates or Lisp-like macro systems. Examples include MetaML [Taha and Sheard 1997], Converge [Tratt 2005], and Racket [Tobin-Hochstadt et al. 2011]. These approaches typically give programmers the power to interact safely with the compiler to produce optimized program fragments. However, the programming model is usually different from the normal language and requires good understanding of which parts are executed at compile time and which at runtime. Furthermore, most incarnations are not as powerful as MOPs in that they cannot redefine the language's semantics.

Runtime Metaprogramming. The CLOS MOP [Kiczales et al. 1991] employs currying to facilitate memoization of lookups. However, it is not sufficient to eliminate all runtime overhead. Furthermore, to enable memoization, the MOP design compromises expressiveness for performance by restricting the API.

To preserve the flexibility of runtime metaprogramming but to reduce its overhead, Masuhara et al. [1995] proposed to use partial evaluation. However, this early work treats the connection between base-level objects meta level as fixed, so that partial evaluation could be applied on methods ahead of time, again compromising expressiveness for performance.

The metaXa system [Golm and Kleinöder 1999] focuses on performance of method interception in the context of an early JVM. While the JIT compiler optimized the transition from base to meta level, the system was very basic. For instance, it enabled inlining only for very small methods, and required hints that objects are used only locally.

Sullivan [2001] proposed to use dynamic partial evaluation to reduce the overhead of MOPs. He uses partial evaluation based on observed runtime types as well as values. Methods can be specialized to observed type signatures, which then can be selected at runtime to avoid the restrictions of optimization based on purely static analyses. Unfortunately, the work remains mostly theoretical.

Another approach to gain performance is partial behavioral reflection [Tanter et al. 2003], which restricts the application of a MOP in spatial and temporal manner. Instead of applying to all program parts equally, the developer optimizes the MOP's application to only the places where it is needed to avoid unnecessary overhead. However, this does not reduce the cost of reflective operations used. Furthermore, it burdens again the programmer with the optimization while dispatch chains are part of the language implementation and thus optimize reflective operations without programmer intervention.

Recent Hybrid Approaches. Shali and Cook [2011] proposed more recently the notion of hybrid partial evaluation, which is a combination of PE and compile-time metaprogramming. The main idea is that programmers can indicate that certain expressions are to be evaluated at compile time so that they can be precomputed via partial evaluation. While this leads to major performance benefits, the burden is on programmers and for instance forbids the use of compile-time objects at runtime, which might be a to strong restriction for some use cases.

Exotypes proposed by DeVito et al. [2014] are similar in spirit. Their idea is to give programmers language abstractions to use staged programming, which allows the generation of specialized implementations for a given problem such as serialization at runtime. Their implementation is then able to use the known types to generate efficient code that can outperforms similar custom implementations. However, as with hybrid partial evaluation, it is up to the programmers to implement these kind of staged programs.

Similar techniques have also been investigated by Asai [2014] to improve performance of a tower of meta interpreters with a powerful metaobject protocols. However, the staging approach limits the expressiveness of the meta interpreters and the ability to change language behavior is restricted under compilation. Since our approach of using dispatch chains integrates with JIT compilation and dynamic code invalidation, expressiveness is not restricted and language behavior can be changed dynamically.

7. Conclusion

This work shows that the overhead of reflective operations and metaobject protocols can be eliminated based on a generalized notion of polymorphic inline caches called *dispatch chains*. Dispatch chains resolve the dynamicity introduced by meta-operations at runtime. They expose the stable runtime behavior and cache, e. g., lookup results to enable compiler optimizations such as inlining.

We demonstrate the effectiveness of the approach in the context of self-optimizing interpreters on top of meta-tracing and partial-evaluation-based compilers, which are both able to remove the indirections introduced by metaprogramming. We showed that the overhead of reflection can be removed completely, and that the cost of a MOP can be minimized to runtime-checks that are required to preserve the semantics of the MOP. In the context of JRuby, we further demonstrate that there is huge potential for performance improvements for real production code that embraces metaprogramming to improve programmer productivity.

While the presented solution is simple and in hindsight obvious, it enables us for the first time to make MOPs perform well and have reflection without overhead, restrictions, or forcing programmers to optimize manually as solutions proposed so far. We hope that the simplicity of the approach encourages language implementers to optimize runtime metaprogramming to remove the still common but unnecessary performance penalty.

For future work, we intend to investigate how dispatch chains can be constructed as generalized forms of polymorphic inline caches in the context of native code generation. In the current form, dispatch chains have been evaluated for self-optimizing interpreters. However, modern VMs typically use multi-tier compilation. Baseline compilers could support generalized polymorphic inline caches in a similar way to determine the necessary runtime values for optimization in later compilation tiers.

Furthermore, we will reinvestigate metaobject protocols with the now possible performance properties. For a long time, MOPs have lost the attention of the community, which favored techniques such as aspect-oriented programming to avoided some of the performance cost. However, MOPs provide a greater power, e. g., by enabling internal domain-specific languages to enforce behavioral restrictions and semantics, which might help to solve challenges in areas with high complexity such as concurrent programming.

Acknowledgments

We would like to thank Carl Friedrich Bolz, Maciej Fijałkowski, and Armin Rigo from the PyPy community for their support with optimizing SOM_{MT}. Likewise, we would like to thank Christian Humer, Lukas Stadler, Andreas Wöß, Thomas Würthinger, and the wider Truffle community for their help with optimizing SOM_{PE}. We would also like to thank Clément Bera, Marcus Denker, and Laurence Tratt for comments on early drafts of this paper.

References

K. Asai. Compiling a Reflective Language Using MetaOCaml. In *Proc of the Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 113–122. ACM, 2014.

V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proc. of the Conference on Programming Language Design and Implementation*, PLDI '00, pages 1–12. ACM, 2000.

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

C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the Meta-level: PyPy's Tracing JIT Compiler. In *Proc. of the Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25. ACM, 2009.

G. T. Brown. *Ruby Best Practices*. O'Reilly, Sebastopol, CA, June 2009.

S. Brunthaler. Efficient Interpretation Using Quickening. In *Proc. of the Symposium on Dynamic Languages*, number 12 in DLS, pages 1–14. ACM, Oct. 2010.

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

S. Chiba. *A Study of Compile-time Metaobject Protocol*. Phd thesis, University of Tokyo, November 1996.

L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 297–302. ACM, 1984.

Z. DeVito, D. Ritchie, M. Fisher, A. Aiken, and P. Hanrahan. First-class Runtime Generation of High-performance Types Using Exotypes. In *Proc. of the Conference on Programming Language Design and Implementation*, PLDI '14, pages 77–88. ACM, 2014.

M. Fowler. *Domain-Specific Languages*. Addison-Wesley, October 2010.

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

M. Golm and J. Kleinöder. Jumping to the Meta Level. In P. Cointe, editor, *Meta-Level Architectures and Reflection*, volume 1616 of *LNCS*, pages 22–39. Springer, 1999.

M. Grimmer, M. Rigger, R. Schatz, L. Stadler, and H. Mössenböck. TruffleC: Dynamic Execution of C on a Java Virtual Machine. In *Proc of the Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 17–26. ACM, 2014.

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

C. Humer, C. Wimmer, C. Wirth, A. Wöß, and T. Würthinger. A Domain-Specific Language for Building Self-Optimizing AST Interpreters. In *Proc. of the Conference on Generative Programming: Concepts and Experiences*, GPCE '14. ACM, 2014.

U. Hölzle, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proc. of the European Conference on Object-Oriented Programming*, volume 512 of *LNCS*, pages 21–38. Springer, 1991.

H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani. Adaptive multi-level compilation in a trace-based java jit compiler. In *Proc. of the Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 179–194, New York, NY, USA, 2012. ACM.

T. Kalibera, P. Maj, F. Morandat, and J. Vitek. A Fast Abstract Syntax Tree Interpreter for R. In *Proc of the Conference on Virtual Execution Environments*, VEE'14, pages 89–102. ACM, 2014.

G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

S. Marr and T. D'Hondt. Identifying A Unifying Mechanism for the Implementation of Concurrency Abstractions on Multi-Language Virtual Machines. In *Objects, Models, Components, Patterns, 50th International Conference, TOOLS 2012*, volume 7304 of *LNCS*, pages 171–186. Springer, May 2012.

S. Marr, T. Pape, and W. De Meuter. Are We There Yet? Simple Language Implementation Techniques for the 21st Century. *IEEE Software*, 31(5): 60–67, September 2014. ISSN 0740-7459.

H. Masuhara, S. Matsuoka, K. Asai, and A. Yonezawa. Compiling Away the Meta-level in Object-oriented Concurrent Reflective Languages Using Partial Evaluation. In *Proc. of the Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '95, pages 300–315. ACM, 1995.

R. Olsen. *Eloquent Ruby*. Professional Ruby. Addison-Wesley, Upper Saddle River, NJ, February 2011.

J. R. Rose. Bytecodes meet Combinators: invokedynamic on the JVM. In *Proc. of the Workshop on Virtual Machines and Intermediate Languages*, pages 1–11. ACM, Oct. 2009.

C. Seaton, M. L. Van De Vanter, and M. Haupt. Debugging at Full Speed. In *Proc. of the Workshop on Dynamic Languages and Applications*, Dyla'14, pages 2:1–2:13. ACM, 2014.

A. Shali and W. R. Cook. Hybrid Partial Evaluation. In *Proc. of the Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 375–390. ACM, 2011.

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

W. Taha and T. Sheard. Multi-stage Programming with Explicit Annotations. In *Proc. of the Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '97, pages 203–217. ACM, 1997.

E. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. In *Proc. of the Conference on Object-oriented Programing, Systems, Languages, and Applications*, pages 27–46. ACM, October 2003.

S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proc. of the Conference on Programming Language Design and Implementation*, PLDI '11, pages 132–141. ACM, 2011.

L. Tratt. Compile-time Meta-programming in a Dynamically Typed OO Language. In *Proc. of the Symposium on Dynamic Languages*, DLS '05, pages 49–63. ACM, 2005.

T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-Optimizing AST Interpreters. In *Proc of the Dynamic Languages Symposium*, DLS'12, pages 73–82, October 2012.

T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to Rule Them All. In *Proc. of the Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward!'13, pages 187–204. ACM, 2013.

A. Wöß, C. Wirth, D. Bonetta, C. Seaton, C. Humer, and H. Mössenböck. An Object Storage Model for the Truffle Language Implementation Framework. In *Proc. of the Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14. ACM, 2014.

W. Zhang, P. Larsen, S. Brunthaler, and M. Franz. Accelerating Iterators in Optimizing AST Interpreters. In *Proc. of the Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 727–743. ACM, 2014.