Runtime Feedback in a Meta-Tracing JIT for Efficient Dynamic Languages

Carl Friedrich Bolz[a] Antonio Cuni[a,c] Maciej Fijałkowski[b] Michael Leuschel[a]
Samuele Pedroni[c] Armin Rigo[a]

[a]Heinrich-Heine-Universität Düsseldorf, STUPS Group, Germany
[b]merlinux GmbH, Hildesheim, Germany
[c]Open End, Göteborg, Sweden

cfbolz@gmx.de anto.cuni@gmail.com fijal@merlinux.eu leuschel@cs.uni-duesseldorf.de
samuele.pedroni@gmail.com arigo@tunes.org

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—code generation, incremental compilers, interpreters, run-time environments

General Terms Languages, Performance, Experimentation

Keywords Tracing JIT, Runtime Feedback, Interpreter, Code Generation, Meta-Programming

Abstract

Meta-tracing JIT compilers can be applied to a variety of different languages without explicitly encoding language semantics into the compiler. So far, they lacked a way to give the language implementor control over runtime feedback. This restricted their performance. In this paper we describe the mechanisms in PyPy's meta-tracing JIT that can be used to control runtime feedback in language-specific ways. These mechanisms are flexible enough to express classical VM techniques such as maps and runtime type feedback.

1. Introduction

One of the hardest parts of implementing an object-oriented dynamic language well is to optimize its object model. This is made harder by the complexity of the core object semantics of many recent languages such as Python, JavaScript or Ruby. For them, even implementing just an interpreter is already a difficult task. Implementing these languages efficiently with a just-in-time compiler (JIT) is extremely challenging, because of their many corner-cases.

It has long been an objective of the partial evaluation community to automatically produce compilers from interpreters. There has been a renaissance of this idea around the approach of tracing just-in-time compilers. A number of projects have attempted this approach. SPUR [3] is a tracing JIT for .NET together with a JavaScript implementation in C#. PyPy [18] contains a tracing JIT for RPython [1] (a restricted subset of Python). This JIT is then used to trace a number of languages implementations written in RPython. A number of other experiments in this directions were done, such as an interpreter for Lua in JavaScript, which is run on and optimized with a tracing JIT for JavaScript [22].

These projects have in common that they work one meta-level down, providing a tracing JIT for the language used to implement the dynamic language, and not for the dynamic language itself. The tracing JIT will then trace through the object model of the dynamic language implementation. This makes the object model transparent to the tracer and its optimizations. Therefore the semantics of the dynamic language does not have to be replicated in a JIT. We call this approach *meta-tracing*. Another commonality of these approaches is that they allow some annotations (or hints) in the dynamic language implementation to guide the meta-tracer. This makes the process not completely automatic but can give good speedups over bare meta-tracing.

In this paper we present two of these hints that are extensively used in the PyPy project to improve the performance of its Python interpreter, particularly of the object model.

Conceptually, the significant speed-ups that can be achieved with dynamic compilation depend on feeding into compilation values observed at runtime and exploiting them. In particular, if there are values which vary very slowly, it is possible to compile multiple specialized versions of the same code, one for each actual value. To exploit the runtime feedback, the implementation code and data structures need to be structured so that many such slow-varying values are at hand. The hints that we present precisely allow us to implement such feedback and exploitation in a meta-tracing context.

Together these hints can be used to express many classic implementation techniques used for object models of dynamic languages, such runtime type feedback and maps.

The contributions of this paper are:

- A hint to turn arbitrary variables into constants in the trace by feeding back runtime information into compilation.
- A way to annotate operations which the constant folding optimization then recognizes and exploits.
- A worked-out example of a simple object model of a dynamic language and how it can be improved using these hints.
- This example also exemplifies general techniques for refactoring code to expose constant folding opportunities of likely runtime constants.

The paper is structured as follows: Section 2 gives an introduction to the PyPy project and meta-tracing and presents an example

ICOOOLPS'11, July 26, 2011, Lancaster, UK.

of a tiny dynamic language object model. Section 3 presents the hints, what they do and how they are applied. Section 4 shows how the hints are applied to the tiny object model and Section 5 presents benchmarks.

2. Background

2.1 The PyPy Project

The PyPy project [18] strives to be an environment where complex dynamic languages can be implemented efficiently. The approach taken when implementing a language with PyPy is to write an interpreter for the language in *RPython*. RPython is a restricted subset of Python chosen in such a way that it is possible to perform type inference on it. The interpreters in RPython can therefore be translated to efficient C code.

A number of languages have been implemented with PyPy, most importantly a full Python implementation, but also a Prolog interpreter [6] and some less mature experiments.

The translation of the interpreter to C code adds a number of implementation details into the final executable that are not present in the interpreter implementation, such as a garbage collector. The interpreter can therefore be kept free from low-level implementation details. Another aspect of the final VM that is added semi-automatically to the generated VM is a tracing JIT compiler.

The advantage of this approach is that writing an interpreter is much easier and less error prone than manually writing a JIT compiler. Similarly, writing in a high level language such as RPython is easier than writing in C.

We call the code that runs on top of a VM implemented with PyPy the *user code* or *user program*.

2.2 PyPy's Meta-Tracing JIT Compilers

A recently popular approach to JIT compilers is that of tracing JITs. Tracing JITs were popularized by the Dynamo project, which used the technique for dynamic machine code optimization [2]. Later they were used to implement a lightweight JIT for Java [12] and for dynamic languages such as JavaScript [10].

A tracing JIT works by recording traces of concrete execution paths through the program. Those traces are linear lists of operations, which are optimized and then get turned into machine code. This recording automatically inlines functions: when a function call is encountered the operations of the called functions are simply put into the trace of the caller too. The tracing JIT tries to produce traces that correspond to loops in the traced program, but most tracing JITs now also have support for tracing non-loops [11].

Because the traces always correspond to a concrete execution they cannot contain any control flow splits. Therefore they encode the control flow decisions needed to stay on the trace with the help of *guards*. Those are operations that check that the assumptions are still true when the compiled trace is later executed with different values.

To be able to do this recording, VMs with a tracing JIT typically contain an interpreter. After a user program is started the interpreter is used; only the most frequently executed paths through the user program are traced and turned into machine code. The interpreter is also used when a guard fails to continue the execution from the failing guard.

Since PyPy wants to be a general framework, we want to reuse our tracer for different languages, which makes classical tracers inapplicable, because they encode language semantics. Therefore PyPy's JIT is a *meta-tracer* [5]. It does not trace the execution of the user program, but instead traces the execution of the *interpreter* that is running the program. This means that the traces it produces do not contain the bytecodes of the language in question, but RPython-level operations that the interpreter did to execute the program.

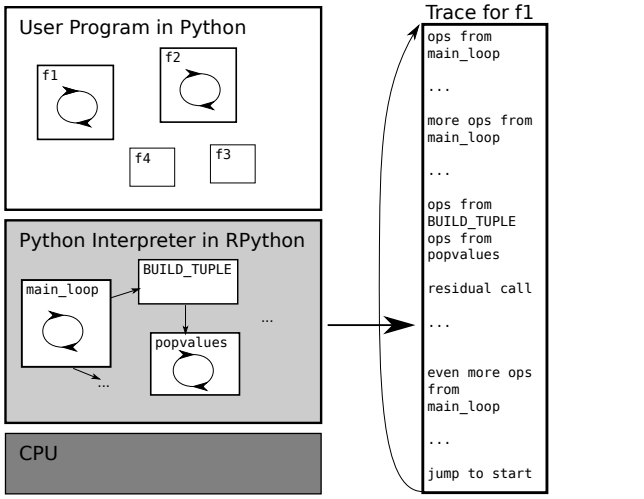

Figure 1. The levels involved in tracing

Tracing through the execution of an interpreter has many advantages. It makes the tracer, its optimizers and backends reusable for a variety of languages. The language semantics do not need to be encoded into the JIT. Instead the tracer just picks them up from the interpreter. This also means that the JIT by construction supports the full language as correctly as the interpreter.

While the operations in a trace are those of the interpreter, the loops that are traced by the tracer are the loops in the user program. To achieve this the tracer stops tracing after one iteration of the loop in the user function that is being considered; at this point, it probably traced many iterations of the interpreter main loop.

Figure 1 shows a diagram of the process. On the left are the levels of execution. The CPU executes the binary of PyPy's Python interpreter, which consists of RPython functions that have been compiled first to C, then to machine code. The interpreter runs a Python program written by a programmer (the user). If the tracer is used, it traces operations on the level of the interpreter. However, the extent of the trace is determined by the loops in the user program.

2.3 Optimizing Traces

Before sending the trace to the backend to produce actual machine code, it is optimized. The optimizer applies a number of techniques to remove or simplify the operations in the trace. Most of these are well known compiler optimization techniques, with the difference that it is easier to apply them in a tracing JIT because it only has to deal with linear traces. Among the techniques are constant folding, common subexpression elimination, allocation removal [4], store/load propagation, loop invariant code motion.

In some places it turns out that if the interpreter author rewrites some parts of the interpreter with these optimizations in mind the traces that are produced by the optimizer can be vastly improved.

2.4 Running Example

As the running example of this paper we will use a very simple and bare-bones object model that just supports classes and instances, without any inheritance or other advanced features. In the model classes contain methods. Instances have a class. Instances have their own attributes (or fields). When looking up an attribute of an instance, the instance's attributes are searched. If the attribute is not found there, the class' methods are searched.

To implement this object model, we use the RPython code in Figure 2 as part of the interpreter source code. In this straightfor-

```
class Class(object):                                        1
    def __init__(self, name):                               2
        self.name = name                                    3
        self.methods = {}                                   4

    def instantiate(self):                                  6
        return Instance(self)                               7

    def find_method(self, name):                            9
        return self.methods.get(name, None)                10

    def write_method(self, name, value):                    12
        self.methods[name] = value                          13


class Instance(object):                                     16
    def __init__(self, cls):                                17
        self.cls = cls                                      18
        self.attributes = {}                                19

    def getfield(self, name):                               21
        return self.attributes.get(name, None)              22

    def write_attribute(self, name, value):                 24
        self.attributes[name] = value                       25

    def getattr(self, name):                                27
        result = self.getfield(name)                        28
        if result is None:                                  29
            result = self.cls.find_method(name)             30
            if result is None:                              31
                raise AttributeError                        32
        return result                                       33
```

Figure 2. Original Version of a Simple Object Model

ward implementation the methods and attributes are just stored in dictionaries (hash maps) on the classes and instances, respectively. While this object model is very simple it already contains most hard parts of Python's object model. Both instances and classes can have arbitrary fields, and they are changeable at any time. Moreover, instances can change their class after they have been created.

When using this object model in an interpreter, a large amount of time will be spent doing lookups in these dictionaries. Let us assume we trace through code that sums three attributes, such as:

```
inst.getattr("a") + inst.getattr("b") + inst.getattr("c")
```

The trace would look like in Figure 3. In this example, the attribute `a` is found on the instance, but the attributes `b` and `c` are found on the class. The line numbers in the trace correspond to the line numbers in Figure 2 where the traced operations come from. The trace is in SSA form. Note how all the guards in trace correspond to a condition in the original code. The trace contains five calls to `dict.get`, which is slow. To make the language efficient using a tracing JIT, we need to find a way to get rid of these dictionary lookups. How to achieve this will be the topic of Section 4.

3. Hints for Controlling Optimization

In this section we will describe two hints that allow the interpreter author to increase the optimization opportunities for constant folding. If applied correctly these techniques can give really big speedups by pre-computing parts of what happens at runtime. On the other hand, if applied incorrectly they might lead to code bloat,

```
# inst_1.getattr("a")                                       27
attributes_1 = inst_1.attributes                            22
result_1 = dict.get(attributes_1, "a")                      22
guard(result_1 is not None)                                 29

# inst_1.getattr("b")                                       27
attributes_2 = inst_1.attributes                            22
v_1 = dict.get(attributes_2, "b", None)                     22
guard(v_1 is None)                                          29
cls_1 = inst_1.cls                                          30
methods_1 = cls.methods                                     10
result_2 = dict.get(methods_1, "b", None)                   10
guard(result_2 is not None)                                 31
v_2 = result_1 + result_2                                   -1

# inst_1.getattr("c")                                       27
attributes_3 = inst_1.attributes                            22
v_3 = dict.get(attributes_3, "c", None)                     22
guard(v_3 is None)                                          29
cls_1 = inst_1.cls                                          30
methods_2 = cls.methods                                     10
result_3 = dict.get(methods_2, "c", None)                   10
guard(result_3 is not None)                                 31

v_4 = v_2 + result_3                                        -1
return(v_4)                                                 -1
```

Figure 3. Trace Through the Object Model

thus making the resulting program actually slower. Note that these hints are *never* put into the user program, only into the interpreter.

For constant folding to work, two conditions need to be met: the arguments of an operation actually need to all be constant, i.e. statically known by the optimizer and the operation needs to be *constant-foldable*, i.e. always yield the same result given the same arguments. There is one kind of hint for both of these conditions.

3.1 Where Do All the Constants Come From?

It is worth clarifying what a "constant" is in this context. A variable of the trace is said to be constant if its value is statically known by the optimizer. The simplest example of constants are literal values, such as `1`. However, the optimizer can statically know the value of a variable even if it is not a constant in the original source code. For example, consider the following fragment of RPython code on the left. If the fragment is traced with `x` being `4`, the trace on the right is produced:

```
if x == 4:
    y = y + x
```

```
guard(x_1 == 4)
y_2 = y_1 + x_1
```

A guard is a runtime check. The above trace will run to completion when x_1 `== 4`. If the check fails, execution of the trace is stopped and the interpreter continues to run. Therefore, the value of x_1 is statically known to be `4` after the guard.

There are cases in which it is useful to turn an arbitrary variable into a constant value. This process is called *promotion* and it is an old idea in partial evaluation (it's called "The Trick" [16] there). The technique is substantially more powerful in a JIT compiler than in the static setting of classic partial evaluation.

Promotion is essentially a tool for trace specialization. There are places in the interpreter where it would open a lot of optimization opportunities if a variable were constant, even though it could have different values in practice. In such a place, promotion can be used.

The typical reason to do that is if there is a lot of computation depending on the value of one variable.

Let us make this more concrete. If we trace a call to the function (written in RPython) on the left, we get the trace on the right:

```
def f1(x, y):
    z = x * 2 + 1
    return z + y
```

```
v1 = x1 * 2
z1 = v1 + 1
v2 = z1 + y1
return(v2)
```

Observe how the first two operations could be constant-folded if the value of x_1 were known. Let us assume that the value of x in the RPython code can vary, but does so rarely, i.e. only takes a few different values at runtime. If this is the case, we can add a hint to promote x, like this:

```
def f1(x, y):
    promote(x)
    z = x * 2 + 1
    return z + y
```

```
guard(x1 == 4)
v1 = x1 * 2
z1 = v1 + 1
v2 = z1 + y1
return(v2)
```

The hint indicates that x is likely a runtime constant and the JIT should try to perform runtime specialization on it in the code that follows. When just running the code, the promote function has no effect. When tracing, some extra work is done. Let us assume that this changed function is traced with the arguments 4 and 8. The trace will be the same, except for one operation at the beginning.

The promotion is turned into a guard operation in the trace. The guard captures the runtime value of x as it was during tracing, which can then be exploited by the compiler. The introduced guard specializes the trace, because it only works if the value of x_1 is 4. From the point of view of the optimizer, this guard is not different frome the one produced by the if statement in the first example. After the guard, it can be assumed that x_1 is equal to 4, meaning that the optimizer will turn this trace into:

```
guard(x1 == 4)
v2 = 9 + y1
return(v2)
```

Notice how the first two arithmetic operations were constant folded. The hope is that the guard is executed quicker than the multiplication and the addition that was now optimized away.

If this trace is executed with values of x_1 other than 4, the guard will fail, and execution will continue in the interpreter. If the guard fails often enough, a new trace will be started from the guard. This other trace will capture a different value of x_1. If it is e.g. 2, then the optimized trace looks like this:

```
guard(x1 == 2)
v2 = 5 + y1
return(v2)
```

This new trace will be attached to the guard instruction of the first trace. If x_1 takes on even more values, a new trace will eventually be made for all of them, linking them into a chain. This is clearly not desirable, so we should promote only variables that do not vary much. However, adding a promotion hint will never produce wrong results. It might just lead to too much machine code being generated.

Promoting integers, as in the examples above, is not used that often. However, the internals of dynamic language interpreters often have values that are variable but vary little in the context of parts of a user program. An example would be the types of variables in a user function, which rarely change in a dynamic language in practice (even though they could). In the interpreter, these user-level types are values. Thus promoting them will lead to type-specialization on the level of the user program. Section 4 will present a complete example of how this works.

3.2 Declaring New Foldable Operations

In the previous section we saw a way to turn arbitrary variables into constants. All foldable operations on these constants can be constant-folded. This works well for constant folding of primitive types, e.g. integers. Unfortunately, in the context of an interpreter for a dynamic language, most operations actually manipulate objects, not primitive types. The operations on objects are often not foldable and might even have side-effects. If one reads a field out of a constant reference to an object this cannot necessarily be folded away because the object can be mutated. Therefore, another hint is needed.

This hint can be used to mark functions as *trace-elidable*. A function is termed trace-elidable if, during the execution of the program, successive calls to the function with identical arguments always return the same result. In addition the function needs to have no side effects or idempotent side effects[1]. From this definition follows that a call to a trace-elidable function with constant arguments in a trace can be replaced with the result of the call seen during tracing.

As an example, take the class on the left. Tracing the call a.f(10) of some instance of A yields the trace on the right (note how the call to c is inlined):

```
class A(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def f(self, val):
        self.y = self.c() + val

    def c(self):
        return self.x * 2 + 1
```

```
x1 = a1.x
v1 = x1 * 2
v2 = v1 + 1
v3 = v2 + val1
a1.y = v3
```

In this case, adding a promote of self in the f method to get rid of the computation of the first few operations does not help. Even if a_1 is a constant reference to an object, reading the x field does not necessarily always yield the same value. To solve this problem, there is another annotation, which lets the interpreter author communicate invariants to the optimizer. In this case, she could decide that the x field of instances of A is immutable, and therefore c is a trace-elidable function. To communicate this, there is an @elidable decorator. If the code in c should be constant-folded away, we would change the class as follows:

```
class A(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def f(self, val):
        promote(self)
        self.y = self.c() + val

    @elidable
    def c(self):
        return self.x * 2 + 1
```

```
guard(a1 ==
      0xb73984a8)
v1 = A.c(a1)
v2 = v1 + val1
a1.y = v2
```

[1] This property is less strict than that of a pure function, because it is only about actual calls during execution. All pure functions are trace-elidable though.

Here, `0xb73984a8` is the address of the instance of `A` that was used during tracing. The call to `A.c` is not inlined, so that the optimizer has a chance to see it. Since the `A.c` method is marked as trace-elidable, and its argument is a constant reference, the call will be removed by the optimizer. The final trace looks like this (assuming that the `x` field's value is 4):

```
guard(a1 == 0xb73984a8)
v2 = 9 + val1
a1.y = v2
```

On the one hand, the `@elidable` annotation is very powerful. It can be used to constant-fold arbitrary parts of the computation in the interpreter. However, the annotation also gives the interpreter author ample opportunity to introduce bugs. If a function is annotated to be trace-elidable, but is not really, the optimizer can produce subtly wrong code. Therefore, a lot of care has to be taken when using this annotation[2]. We hope to introduce a debugging mode which would (slowly) check whether the annotation is applied incorrectly to mitigate this problem.

4. Putting It All Together

In this section we describe how the simple object model from Section 2.4 can be made efficient using the hints described in the previous section. The object model there is typical for many current dynamic languages (such as Python, Ruby and JavaScript) as it relies heavily on hash-maps to implement its objects.

4.1 Making Instance Attributes Faster Using Maps

The first step in making `getattr` faster in our object model is to optimize away the dictionary lookups on the instances. The hints of the previous section do not seem to help with the current object model. There is no trace-elidable function to be seen, and the instance is not a candidate for promotion, because there tend to be many instances.

This is a common problem when trying to apply hints. Often, the interpreter needs a small rewrite to expose the trace-elidable functions and nearly-constant objects that are implicitly there. In the case of instance fields this rewrite is not entirely obvious. The basic idea is as follows. In theory instances can have arbitrary fields. In practice however many instances share their layout (i.e. their set of keys) with many other instances.

Therefore it makes sense to factor the layout information out of the instance implementation into a shared object, called the *map*. Maps are a well-known technique to efficiently implement instances and come from the SELF project [7]. They are also used by many JavaScript implementations such as V8. The rewritten `Instance` class using maps can be seen in Figure 4.

In this implementation instances no longer use dictionaries to store their fields. Instead, they have a reference to a map, which maps field names to indexes into a storage list. The storage list contains the actual field values. Maps are shared between different instances, therefore they have to be immutable, which means that their `getindex` method is a trace-elidable function. When a new attribute is added to an instance, a new map needs to be chosen, which is done with the `add_attribute` method on the previous map. This function is also trace-elidable, because it caches all new instances of `Map` that it creates, to make sure that objects with the same layout have the same map, which makes its side effects idempotent. Now that we have introduced maps, it is safe to promote the map everywhere, because we assume that the number of different instance layouts is small.

[2] The most common use case of the `@elidable` annotation is indeed to declare the immutability of fields. Because it is so common, we have special syntactic sugar for it.

```
class Map(object):
    def __init__(self):
        self.indexes = {}
        self.other_maps = {}

    @elidable
    def getindex(self, name):
        return self.indexes.get(name, -1)

    @elidable
    def add_attribute(self, name):
        if name not in self.other_maps:
            newmap = Map()
            newmap.indexes.update(self.indexes)
            newmap.indexes[name] = len(self.indexes)
            self.other_maps[name] = newmap
        return self.other_maps[name]

EMPTY_MAP = Map()

class Instance(object):
    def __init__(self, cls):
        self.cls = cls
        self.map = EMPTY_MAP
        self.storage = []

    def getfield(self, name):
        map = self.map
        promote(map)
        index = map.getindex(name)
        if index != -1:
            return self.storage[index]
        return None

    def write_attribute(self, name, value):
        map = self.map
        promote(map)
        index = map.getindex(name)
        if index != -1:
            self.storage[index] = value
            return
        self.map = map.add_attribute(name)
        self.storage.append(value)

    def getattr(self, name):
        ... # as before
```

Figure 4. Simple Object Model With Maps

With this adapted instance implementation, the trace we saw in Section 2.4 changes to that of Figure 5. There `0xb74af4a8` is the memory address of the `Map` instance that has been promoted. Operations that can be optimized away are grayed out, their results will be replaced with fixed values by the constant folding.

The calls to `Map.getindex` can be optimized away, because they are calls to a trace-elidable function and they have constant arguments. That means that $index_{1/2/3}$ are constant and the guards on them can be removed. All but the first guard on the map will be optimized away too, because the map cannot have changed in between. This trace is already much better than the original one. Now we are down from five dictionary lookups to just two.

The technique to make instance lookups faster is applicable in more general cases. A more abstract view of maps is that of splitting a data-structure into an immutable part (e.g., the map) and a part

```
# inst1.getattr("a")
map1 = inst1.map
guard(map1 == 0xb74af4a8)
index1 = Map.getindex(map1, "a")
guard(index1 != -1)
storage1 = inst1.storage
result1 = storage1[index1]

# inst1.getattr("b")
map2 = inst1.map
guard(map2 == 0xb74af4a8)
index2 = Map.getindex(map2, "b")
guard(index2 == -1)
cls1 = inst1.cls
methods1 = cls1.methods
result2 = dict.get(methods1, "b", None)
guard(result2 is not None)
v2 = result1 + result2

# inst1.getattr("c")
map3 = inst1.map
guard(map3 == 0xb74af4a8)
index3 = Map.getindex(map3, "c")
guard(index3 == -1)
cls2 = inst1.cls
methods2 = cls2.methods
result3 = dict.get(methods2, "c", None)
guard(result3 is not None)

v4 = v2 + result3
return(v4)
```

Figure 5. Unoptimized Trace After the Introduction of Maps

that changes (e.g., the storage list). All the computation on the immutable part is trace-elidable so that only the manipulation of the quick-changing part remains in the trace after optimization.

4.2 Versioning of Classes

Above we assumed that the total number of different instance layouts is small compared to the number of instances. For classes we will make an even stronger assumption. We simply assume that it is rare for classes to change at all. This is not always reasonable (sometimes classes contain counters or similar things) but for this simple example it is good enough.[3]

What we would really like that the `Class.find_method` method is trace-elidable. But it cannot be, because it is always possible to change the class itself. Every time the class changes, `find_method` can potentially return a new value.

Therefore, we give every class a version object, which is changed every time a class gets changed (i.e., the `methods` dictionary changes). This means that the result of calls to `methods.get()` for a given `(name, version)` pair will always be the same, i.e. it is a trace-elidable operation. To help the JIT to detect this case, we factor it out in a helper method `_find_method` which is marked as `@elidable`. The refactored `Class` can be seen in Figure 6

What is interesting here is that `_find_method` takes the `version` argument but it does not use it at all. Its only purpose is to make the call trace-elidable, because when the version object changes, the result of the call might be different from the previous one.

[3] There is a more complex variant of the presented technique that can accommodate quick-changing class fields a lot better.

```
class VersionTag(object):
    pass

class Class(object):
    def __init__(self, name):
        self.name = name
        self.methods = {}
        self.version = VersionTag()

    def find_method(self, name):
        promote(self)
        version = self.version
        promote(version)
        return self._find_method(name, version)

    @elidable
    def _find_method(self, name, version):
        assert version is self.version
        return self.methods.get(name, None)

    def write_method(self, name, value):
        self.methods[name] = value
        self.version = VersionTag()
```

Figure 6. Versioning of Classes

```
# inst1.getattr("a")
map1 = inst1.map
guard(map1 == 0xb74af4a8)
index1 = Map.getindex(map1, "a")
guard(index1 != -1)
storage1 = inst1.storage
result1 = storage1[index1]

# inst1.getattr("b")
map2 = inst1.map
guard(map2 == 0xb74af4a8)
index2 = Map.getindex(map2, "b")
guard(index2 == -1)
cls1 = inst1.cls
guard(cls1 == 0xb7aaaaf8)
version1 = cls1.version
guard(version1 == 0xb7bbbb18)
result2 = Class._find_method(cls1, "b", version1)
guard(result2 is not None)
v2 = result1 + result2

# inst1.getattr("c")
map3 = inst1.map
guard(map3 == 0xb74af4a8)
index3 = Map.getindex(map3, "c")
guard(index3 == -1)
cls2 = inst1.cls
guard(cls2 == 0xb7aaaaf8)
version2 = cls2.version
guard(version2 == 0xb7bbbb18)
result3 = Class._find_method(cls2, "c", version2)
guard(result3 is not None)

v4 = v2 + result3
return(v4)
```

Figure 7. Unoptimized Trace After Introduction of Versioned Classes

```
# inst1.getattr("a")
map1 = inst1.map
guard(map1 == 0xb74af4a8)
storage1 = inst1.storage
result1 = storage1[0]

# inst1.getattr("b")
cls1 = inst1.cls
guard(cls1 == 0xb7aaaaf8)
version1 = cls1.version
guard(version1 == 0xb7bbbb18)
v2 = result1 + 41

# inst1.getattr("c")
v4 = v2 + 17
return(v4)
```

Figure 8. Optimized Trace After Introduction of Versioned Classes

The trace with this new class implementation can be seen in Figure 7. The calls to `Class._find_method` can now be optimized away, also the promotion of the class and the version, except for the first one. The final optimized trace can be seen in Figure 8.

The index `0` that is used to read out of the `storage` list is the result of the constant-folded `getindex` call. The constants `41` and `17` are the results of the folding of the `_find_method` calls. This final trace is now very good. It no longer performs any dictionary lookups. Instead it contains several guards. The first guard checks that the map is still the same. This guard will fail if the same code is executed with an instance that has another layout. The second guard checks that the class of `inst` is still the same. It will fail if the trace is executed with an instance of another class. The third guard checks that the class did not change since the trace was produced. It will fail if somebody calls the `write_method` method on the class.

4.3 Real-World Considerations

The techniques used above for the simple object model are used for the object model of PyPy's Python interpreter too. Since Python's object model is considerably more complex, some additional work needs to be done.

The first problem that needs to be solved is that Python supports (multiple) inheritance. Therefore looking up a method in a class needs to consider all the classes in the whole method resolution order. This makes the versioning of classes more complex. If a class is changed its version changes. At the same time, the versions of all the classes inheriting from it need to be changed as well, recursively. This makes class changes expensive, but they should be rare. On the other hand, a method lookup in a complex class hierarchy is as optimized in the trace as in our simple object model above.

Another optimization is that in practice the shape of an instance is correlated with its class. In our code above, we allow both to vary independently. In PyPy's Python interpreter we store the class of an instance on its map. This means that we get one fewer promotion and thus one fewer guard in the trace, because the class doesn't need to be promoted after the map has been.

5. Evaluation

For space reasons we cannot perform a full evaluation here, but still want to present some benchmark numbers. We chose to present just some benchmarks: The templating engine of the Django web framework[4]; a Monte-Carlo Go AI[5]; a BZ2 decoder; a port of the classical Richards benchmark to Python; a Python version of the Telco decimal benchmark[6], using a pure Python decimal floating point implementation. The results we see in these benchmarks seem to repeat themselves in other benchmarks using object-oriented code; for purely numerical algorithms the speedups introduced by the techniques in this paper are much smaller because they are already fast.

	CPython	JIT baseline	JIT full
django[ms]	988.67 ± 0.49	405.62 ± 4.80	149.31 ± 1.37
	6.62 ×	2.72 ×	1.00 ×
go[ms]	947.43 ± 1.30	525.53 ± 7.67	174.32 ± 7.78
	5.44 ×	3.01 ×	1.00 ×
pyflate[ms]	3209.20 ± 3.65	2884.26 ± 21.11	1585.48 ± 5.22
	2.02 ×	1.82 ×	1.00 ×
richards[ms]	357.79 ± 1.32	421.87 ± 0.48	17.89 ± 1.15
	20.00 ×	23.58 ×	1.00 ×
telco[ms]	1209.67 ± 2.20	738.18 ± 3.29	153.48 ± 1.86
	7.88 ×	4.81 ×	1.00 ×

Figure 9. Benchmark Results

The benchmarks were run on an otherwise idle Intel Core2 Duo P8400 processor with 2.26 GHz and 3072 KB of cache on a machine with 3GB RAM running Linux 2.6.35. We compared the performance of two Python implementations on the benchmarks. As a baseline, we used the standard Python implementation in C, CPython 2.6.6[7], which uses a bytecode-based interpreter. We compare it against two versions of PyPy's Python interpreter, both of them with JIT enabled. The PyPy baseline does not enable maps or type version, the full JIT enables both.

All benchmarks were run 50 times in the same process, to give the JIT time to produce machine code. The arithmetic mean of the times of the last 30 runs were used as the result. The errors were computed using a confidence interval with a 95% confidence level [13]. The results are reported in Figure 9, together with the same numbers normalized to those of the full JIT.

The optimizations give a speedup between 80% and almost 20 times. The Richards benchmark is a particularly good case for the optimizations as it makes heavy uses of object-oriented features. Pyflate uses mostly imperative code, so does not benefit as much. Together with the optimization, PyPy outperforms CPython in all benchmarks, which is not surprising because CPython is a simple bytecode-based interpreter.

6. Related Work

The very first meta-tracer is described by Sullivan et. al. [20]. They used Dynamo RIO, the successor of Dynamo [2] to trace through a small synthetic interpreter. As in Dynamo, tracing happens on the machine code level. The system needs some hints to mark the main interpreter loop and where the backward jumps in user programs are. PyPy uses similar hints to achieve this [5]. Their approach suffers mostly from the low abstraction level that machine code provides.

Yermolovich et. al. [22] describe the use of the Tamarin JavaScript tracing JIT as a meta-tracer for a Lua interpreter. They compile the normal Lua interpreter in C to ActionScript bytecode. Again, the interpreter is annotated with some hints that indicate the main inter-

[4] `http://www.djangoproject.com/`

[5] `http://shed-skin.blogspot.com/2009/07/disco-elegant-python-go-player.html`

[6] `http://speleotrove.com/decimal/telco.html`

[7] `http://python.org`

preter loop to the tracer. No further hints are described in the paper. There is no comparison of their system to the original Lua VM in C, which makes it hard to judge the effectiveness of the approach.

SPUR [3] is a tracing JIT for CIL bytecode, which is then used to trace through a JavaScript implementation written in C#. The JavaScript implementation compiles JavaScript to CIL bytecode together with an implementation of the JavaScript object model. The object model uses maps and inline caches to speed up operations on objects. The tracer traces through the compiled JavaScript functions and the object model. SPUR contains two hints that can be used to influence the tracer: one to prevent tracing of a C# function and one to force unrolling of a loop (PyPy has equivalent hints, but they were not described in this paper).

Partial evaluation [16] tries to automatically transform interpreters into compilers using the second futamura projection [9]. Given that classical partial evaluation works strictly ahead of time, it inherently cannot support runtime feedback. Some partial evaluators work at runtime, such as DyC [14], which also supports a concept similar to promotion (called dynamic-to-static promotion).

An early attempt at building a general environment for implementing languages efficiently is described by Wolczko et. al. [21]. They implement Java and Smalltalk on top of the SELF VM by compiling the languages to SELF. The SELF JIT is good enough to optimize the compiled code very well. We believe the approach to be restricted to languages that are similar enough to SELF as there were no mechanisms to control the underlying compiler.

Somewhat relatedly, the proposed "invokedynamic" bytecode [19] that will be added to the JVM is supposed to make the implementation of dynamic languages on top of JVMs easier. The bytecode gives the user access to generalized inline caches. It requires of course compilation to JVM bytecode instead of writing an interpreter.

We already explored promotion in other contexts, such as earlier versions of PyPy's JIT. Promotion is also heavily used by Psyco [17] (promotion is called "unlifting" in this paper) a method-based JIT compiler for Python written by one of the authors. Promotion is quite similar to runtime type feedback (and also inline caching) techniques which were first used in Smalltalk [8] and SELF [15] implementations. Promotion is more general because any information can be fed back into compilation, not just types.

7. Conclusion

In this paper we presented two hints that can be used in the source code of an interpreter written with PyPy. They give control over runtime feedback and optimization to the language implementor. They are expressive enough for building well-known virtual machine optimization techniques, such as maps and inline caches. We believe that they are flexible enough to express a wide variety of language semantics efficiently.

Acknowledgements

The authors would like to thank Peng Wu, David Edelsohn and Laura Creighton for encouragement, fruitful discussions and feedback during the writing of this paper. This research was partially supported by the BMBF funded project PyJIT (nr. 01QE0913B; Eureka Eurostars). We also want to thank the anonymous reviewers for their feedback.

References

[1] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *DLS*, Montreal, Quebec, Canada, 2007. ACM.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[3] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: a trace-based JIT compiler for CIL. In *OOPSLA*, Reno/Tahoe, Nevada, USA, 2010. ACM.

[4] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo. Allocation removal by partial evaluation in a tracing JIT. In *PEPM*, PEPM '11, 2011.

[5] C. F. Bolz, A. Cuni, M. Fijałkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *ICOOOLPS*, pages 18–25, Genova, Italy, 2009. ACM.

[6] C. F. Bolz, M. Leuschel, and D. Schneider. Towards a jitting VM for prolog execution. In *PPDP*, Hagenberg, Austria, 2010. ACM.

[7] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In *OOPSLA*, volume 24, 1989.

[8] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *POPL*, Salt Lake City, Utah, 1984. ACM.

[9] Y. Futamura. Partial evaluation of computation process - an approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[10] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI*, PLDI '09, New York, New York, 2009. ACM. ACM ID: 1542528.

[11] A. Gal and M. Franz. Incremental dynamic code generation with trace trees. Technical Report ICS-TR-06-16, Donald Bren School of Information and Computer Science, University of California, Irvine, Nov. 2006.

[12] A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *VEE*, Ottawa, Ontario, Canada, 2006. ACM.

[13] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Notices*, 42(10):57–76, 2007.

[14] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: an expressive annotation-directed dynamic compiler for c. *Theoretical Computer Science*, 248:147–199, Oct. 2000. ACM ID: 357493.

[15] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *PLDI*, pages 326–336, Orlando, Florida, United States, 1994. ACM.

[16] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, 1993.

[17] A. Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *PEPM*, Verona, Italy, 2004. ACM.

[18] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *DLS*, Portland, Oregon, USA, 2006. ACM.

[19] J. R. Rose. Bytecodes meet combinators: invokedynamic on the JVM. *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, 2009.

[20] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic native optimization of interpreters. In *Workshop on Interpreters, virtual machines and emulators*, San Diego, California, 2003. ACM.

[21] M. Wolczko, O. Agesen, and D. Ungar. Towards a universal implementation substrate for Object-Oriented languages. In *OOPSLA workshop on Simplicity, Performance, and Portability in Virtual Machine Design*, 1999.

[22] A. Yermolovich, C. Wimmer, and M. Franz. Optimization of dynamic languages using hierarchical layering of virtual machines. In *DLS*, pages 79–88, Orlando, Florida, USA, 2009. ACM.