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

Carl Friedrich Bolz[a] Antonio Cuni[a] 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

Abstract

A meta-tracing JIT is a JIT that is applicable to a variety of different languages without explicitly encoding language semantics into the compiler. So far, meta-tracing JITs lacked a way to feed back runtime information into the compiler, which 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 flexible and language-specific ways. These mechanisms are flexible enough to implement classical VM techniques such as maps and polymorphic inline caches.

1. Introduction

One of the hardest parts of implementing a dynamic language efficiently is to optimize its object model. This is made harder by the fact that many recent languages such as Python, JavaScript or Ruby have rather complex core object semantics. For them, implementing just an interpreter is already a complex task. Implementing them 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 recent renaissance of this idea around the approach of tracing just-in-time compilers. A number of projects have attempted this approach. SPUR [2] is a tracing JIT for .NET together with a JavaScript implementation in C#. PyPy [19] contains a tracing JIT for Python (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 [24].

ICOOOLPS '11 Lancaster, UK

These projects have in common that they work one meta-level down, providing a tracing JIT for the implementation language used to implement the dynamic language, and not for the dynamic language itself. The tracing JIT then will 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.

Conceptually the significant speed-ups that can be achieved with dynamic compilation depend on feeding into compilation and exploiting values observed at runtime that are slow-varying in practice. To exploit the runtime feedback, the implementation code and data structures need to be structured so that many such values are at hand. The hints that we present allow exactly to implement such feedback and exploitation in a meta-tracing context.

Concretely these hints are used to control how the optimizer of the tracing JIT can improve the traces of the object model. More specifically, these hints influence the constant folding optimization. The first hint makes it possible to turn arbitrary variables in the trace into constant by feeding back runtime values. The second hint allows the definition of additional foldable operations.

Together these two hints can be used to express many classic implementation techniques used for object models of dynamic languages, such as maps and polymorphic inline caches.

The contributions of this paper are:

- A hint to turn arbitrary variables into constants in the trace, that means the feedback of runtime information into compilation.
- A way to define new pure operations which the constant folding optimization then recognizes.
- A worked-out example of a simple object model of a dynamic language and how it can be improved using these hints.

The paper is structured as follows: Section 2 gives an introduction to the PyPy project and meta-tracing and presents an example 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 [19] 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 [7] and a Smalltalk VM [5].

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.

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

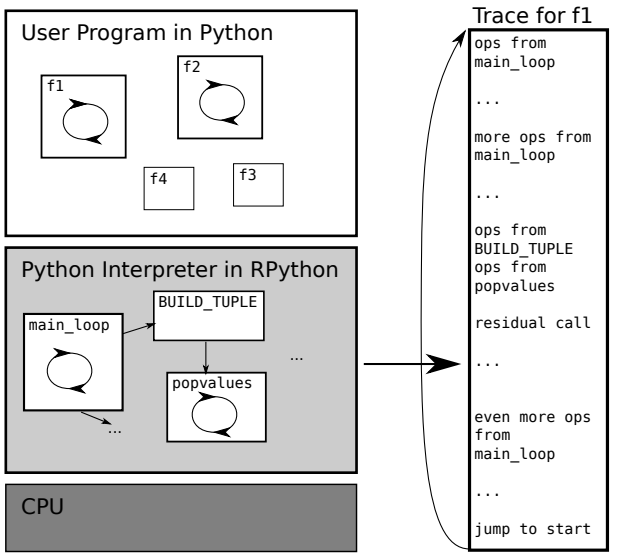

Figure 1. The levels involved in tracing

2.2 PyPy's Meta-Tracing JIT Compilers

A recently popular approach to JIT compilers is that of tracing JITs. Tracing JITs have their origin in the Dynamo project, which used one of them for dynamic assembler optimization [1]. Later they were used to implement a lightweight JIT for Java [13] and for dynamic languages such as JavaScript [11].

A tracing JIT works by recording traces of concrete execution paths through the program. Those traces are therefore linear list 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 too.

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 turned into machine code. 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 [12].

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.

One disadvantage of (tracing) JITs which makes them not directly applicable to PyPy is that they need to encode the language semantics of the language they are tracing. Since PyPy wants to be a general framework, we want to reuse our tracer for different languages. Therefore PyPy's JIT is a meta-tracer [4]. 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 don't contain the bytecodes of the language in question, but RPython-level operations that the interpreter did to execute the program.

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.

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. This means that the tracer stops tracing after one iteration of the loop in the user function that is being considered. At this point, it can have 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:

- constant folding
- common subexpression elimination
- allocation removal [3]
- 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 fancy features. The model has classes, which contain methods. Instances have a class. Instances have their own attributes (or fields). When looking up an attribute on an instance, the instances attributes are searched. If the attribute is not found there, the class' methods are searched.

To implement this object model, we could use the RPython code in Figure 2 as part of the interpreter source code. In this straightforward implementation the methods and attributes are just

```
class Class(object):                                          1
    def __init__(self, name):                                 2
        self.name = name                                      3
        self.methods = {}                                     4
                                                              5
    def instantiate(self):                                    6
        return Instance(self)                                 7
                                                              8
    def find_method(self, name):                              9
        return self.methods.get(name, None)                   10
                                                              11
    def write_method(self, name, value):                      12
        self.methods[name] = value                            13
                                                              14
                                                              15
class Instance(object):                                       16
    def __init__(self, cls):                                  17
        self.cls = cls                                        18
        self.attributes = {}                                  19
                                                              20
    def getfield(self, name):                                 21
        return self.attributes.get(name, None)                22
                                                              23
    def write_attribute(self, name, value):                   24
        self.attributes[name] = value                         25
                                                              26
    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

stored in dictionaries (hash maps) on the classes and instances, respectively. While this object model is very simple it already contains all the 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 huge amount of time will be spent doing lookups in these dictionaries. Let's 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 indeed 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 somehow. How to achieve this will be topic of Section 4.

3. Hints for Controlling Optimization

In this section we will describe how to add 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, thus making the resulting program actually slower.

For constant folding to work, two conditions need to be met:

```
# inst1.getattr("a")                                          27
attributes1 = inst1.attributes                                22
result1 = dict.get(attributes1, "a")                          22
guard(result1 is not None)                                    29

# inst1.getattr("b")                                          27
attributes2 = inst1.attributes                                22
v1 = dict.get(attributes2, "b")                               22
guard(v1 is None)                                             29
cls1 = inst1.cls                                              30
methods1 = cls.methods                                        10
result2 = dict.get(methods1, "b")                             10
guard(result2 is not None)                                    31
v2 = result1 + result2                                        -1

# inst1.getattr("c")                                          27
attributes3 = inst1.attributes                                22
v3 = dict.get(attributes3, "c")                               22
guard(v3 is None)                                             29
cls1 = inst1.cls                                              30
methods2 = cls.methods                                        10
result3 = dict.get(methods2, "c")                             10
guard(result3 is not None)                                    31

v4 = v2 + result3                                             -1
return(v4)                                                    -1
```

Figure 3. Trace Through the Object Model

- the arguments of an operation actually need to all be constant, i.e. statically known by the optimizer
- the operation needs to be *pure*, i.e. always yield the same result given the same arguments.

The PyPy JIT generator automatically detects the majority of these conditions. However, for the cases in which the automatic detection does not work, the interpreter author can apply **hints** to improve the optimization opportunities. There is one kind of hint for both of the conditions above.

3.1 Where Do All the Constants Come From

It is worth clarifying what is a "constant" 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. For example, if in the RPython source code we have a line like `y = x + 1`, the second operand will be a constant in the trace.

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:

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

If the fragment is traced with x_1 being `4`, the following trace is produced:

```
guard(x1 == 4)
y2 = y1 + x1
```

In the trace above, the value of x_1 is statically known thanks to the guard. Remember that 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.

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" [17] there). Promotion is also heavily used by Psyco [18] and by all older versions of PyPy's JIT. Promotion is a technique that only works well in JIT compilers; in static compilers it is significantly less applicable.

Promotion is essentially a tool for trace specialization. In some places in the interpreter it would be very useful if a variable were constant, even though it could have different values in practice. In such a place, promotion is used. The typical reason to do that is if there is a lot of computation depending on the value of that variable.

Let's make this more concrete. If we trace a call to the following function:

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

We get a trace that looks like this:

```
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's assume that the value of `x` in the Python 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):
    x = hint(x, promote=True)
    z = x * 2 + 1
    return z + y
```

The meaning of this hint is that the tracer should pretend that `x` is a constant in the code that follows. When just running the code, the function has no effect, as it simply returns its first argument. When tracing, some extra work is done. Let's 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:

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

The promotion is turned into a `guard` operation in the trace. The guard captures the value of x_1 as it was at runtime. From the point of view of the optimizer, this guard is not any different than the one produced by the `if` statement in the example above. After the guard, the rest of the trace can assume 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 don't vary much. However, adding a promotion hint will never produce wrong results. It might just lead to too much assembler code.

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. Even though in principle the argument to a Python function could be any Python type, in practice the argument types tend not to vary often. Therefore it is possible to promote the types. The next section will present a complete example of how this works.

3.2 Declaring New Pure Operations

In the previous section we saw a way to turn arbitrary variables into constants. All pure operations on these constants can be constant-folded. This works great for constant folding of simple types, e.g. integers. Unfortunately, in the context of an interpreter for a dynamic language, most operations actually manipulate objects, not simple types. The operations on objects are often not pure 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.

As an example, take the following class:

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

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

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

Tracing the call `a.f(10)` of some instance of `A` yields the following trace (note how the call to `compute` is inlined):

```
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 `compute` is a pure function. To communicate this, there is a `purefunction` decorator. If the code in `compute` 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):
        self = hint(self, promote=True)
        self.y = self.compute() + val

    @purefunction
    def compute(self):
        return self.x * 2 + 1
```

Now the trace will look like this:

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

Here, `0xb73984a8` is the address of the instance of `A` that was used during tracing. The call to `compute` is not inlined, so that the optimizer has a chance to see it. Since the `compute` function is marked as pure, and its argument is a constant reference, the call will be removed by the optimizer. The final trace looks like this:

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

(assuming that the `x` field's value is `4`).

On the one hand, the `purefunction` 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 mess things up. If a function is annotated to be pure, but is not really, the optimizer can produce subtly wrong code. Therefore, a lot of care has to be taken when using this annotation.

3.2.1 Observably Pure Functions

Why can't we simply write an analysis to find out that the `x` fields of the `A` instances is immutable and deduce that `compute` is a pure function, since it only reads the `x` field and does not have side effects? This might be possible in this particular case, but in practice the functions that are annotated with the `purefunction` decorator are usually more complex. The easiest example for this is that of a function that uses memoization to cache its results. If this function is analyzed, it looks like the function has side effects, because it changes the memoizing dictionary. However, because this side effect is not externally visible, the function from the outside is pure. This is a property that is not easily detectable by analysis. Therefore, the purity of this function needs to be annotated.

3.2.2 Immutable Fields

One of the most common cases of pure functions is reading immutable values out of objects. Since this is so common, we have special syntactic sugar for it. A RPython class can have a class attribute `_immutable_fields_` set to a list of strings, listing the fields that cannot be changed. This is equivalent to using getters and annotating them with `purefunction`.

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 the 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 we have looked at in the two previous sections don't seem to help with the current object model. There is no pure 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 pure 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 [8]. 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. The maps are shared between objects with the same layout. Therefore they have to be immutable, which means that their `getindex` method is a pure 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 (which is also pure). 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.

With this changed instance implementation, the trace we had above changes to the following that of see 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.

The calls to `Map.getindex` can be optimized away, because they are calls to a pure function and they have constant arguments. That means that `index1/2/3` 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.

4.2 Versioning of Classes

Instances were optimized making the assumption 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 totally reasonable (sometimes classes contain counters or similar things) but for this simple example it is good enough.

What we would really like is if the `Class.find_method` method were pure. 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 content of the `methods` dictionary changes). This means that the result of `methods.get()` for a given `(name, version)` pair will always be the same, i.e. it is a pure operation. To help the JIT to detect this case, we factor it out in a helper method which is explicitly marked as `@purefunction`. 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 pure, because when the version object changes, the result of the call might be different than the previous one.

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

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

    @purefunction
    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 = hint(self.map, promote=True)
        index = map.getindex(name)
        if index != -1:
            return self.storage[index]
        return None

    def write_attribute(self, name, value):
        map = hint(self.map, promote=True)
        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

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` array 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.

```
# 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")
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")
guard(result3 is not None)

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

Figure 5. Unoptimized Trace After the Introduction of Maps

```
class VersionTag(object):
    pass

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

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

    @purefunction
    def _find_method(self, name, version):
        return self.methods.get(name)

    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

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 object model here.

A downside of the versioning of classes that we haven't yet fixed in PyPy, is that some classes *do* change a lot. An example would be a class that keeps a counter of how many instances have been created so far. This is very slow right now, but we have ideas about how to fix it in the future.

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 act somewhat more cleverly. The class of an instance is not stored on the instance itself, but on the 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 two benchmarks, a port of the classical Richards benchmark in Python and a Python version of the Telco decimal benchmark[1], using a pure Python decimal floating point implementation. The results we see in these two benchmarks seem to repeat themselves in other benchmarks using object-oriented code, for purely numerical algorithms the speedups are a lot lower.

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 various Python implementations on the benchmarks. As a baseline, we used the standard Python implementation in C, CPython 2.6.6[2], which uses a bytecode-based interpreter. We compare it against four versions of PyPy's Python interpreter, all of them with JIT enabled. The PyPy baseline does not enable maps or type versions. Then we have a version each where maps and versions are enabled alone and finally a version with 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 [14]. The results are reported in Figure 9.

Versioned types speed up both benchmarks by a significant factor of around 7. The speed advantage of maps alones is a lot less clear. Maps also have a memory advantage which we did not measure here. By themselves, maps improved the Richards benchmark slightly, but made the Telco benchmark slower. Enabling both maps and versioned types together yields a significant improvement over just versioned types for Richards. XXX good explanation. For Telco, enabling both does not change much.

[1] http://speleotrove.com/decimal/telco.html

[2] http://python.org

	richards[ms]	telco[ms]
CPython	357.79 ± 1.32	1209.67 ± 2.20
speedup	$1.00 \times$	$1.00 \times$
JIT baseline	421.87 ± 0.48	738.18 ± 3.29
speedup	$0.85 \times$	$1.64 \times$
JIT map	382.88 ± 4.40	834.19 ± 4.91
speedup	$0.93 \times$	$1.45 \times$
JIT version	49.87 ± 0.29	157.88 ± 1.79
speedup	$7.17 \times$	$7.66 \times$
JIT full	17.89 ± 1.15	153.48 ± 1.86
speedup	$20.00 \times$	$7.88 \times$

Figure 9. Benchmark Results

6. Related Work

The very first meta-tracer is described by Sullivan et. al. [22]. They used Dynamo RIO, the successor of Dynamo [1] to trace through a small synthetic interpreter. As in Dynamo, tracing happens on the machine code level. The tracer is instructed by some hints in the tiny interpreter where the main interpreter loop is and for how long to trace to match loops in the user-level functions. These hints are comparable to the one PyPy uses for the same reasons [4]. Their approach suffers mostly from the low abstraction level that machine code provides.

Yermolovich et. al. 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 interpreter 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 [2] is a tracing JIT for CIL bytecode, which is then used to trace through an 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 [17] tries to automatically transform interpreters into compilers using the second futamura projection [10]. Given that classical partial evaluation works strictly ahead of time, it inherently cannot support runtime feedback.

An early attempt at building a general environment for implementing languages efficiently is described by Wolczko et. al. [23]. 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 [21] 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 language implementor control over how the JIT optimizes the language's features and when optimized code needs to be deoptimized. XXX

We already explored promotion in other context, such as earlier versions of PyPy's JIT [20] as well as a Prolog partial evaluator [6]. Promotion is quite similar to (polymorphic) inline caching and runtime type feedback techniques which were first used in Smalltalk [9] and SELF [15, 16] implementations. Promotion is more general because any information can be cached in line, not just classes of method receivers.

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

References

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

[2] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: a trace-based JIT compiler for CIL. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 708–725, Reno/Tahoe, Nevada, USA, 2010. ACM.

[3] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo. Allocation removal by partial evaluation in a tracing JIT. *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, page 43–52, 2011. ACM ID: 1929508.

[4] C. F. Bolz, A. Cuni, M. Fijałkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, Genova, Italy, 2009. ACM.

[5] C. F. Bolz, A. Kuhn, A. Lienhard, N. Matsakis, O. Nierstrasz, L. Renggli, A. Rigo, and T. Verwaest. Back to the future in one week — implementing a smalltalk VM in PyPy. In *Self-Sustaining Systems*, pages 123–139. 2008.

[6] C. F. Bolz, M. Leuschel, and A. Rigo. Towards Just-In-Time partial evaluation of prolog. In *Logic-based Program Synthesis and Transformation (LOPSTR'2009)*, LNCS 6037 to appear. Springer-Verlag.

[7] C. F. Bolz, M. Leuschel, and D. Schneider. Towards a jitting VM for prolog execution. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 99–108, Hagenberg, Austria, 2010. ACM.

[8] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Not.*, 24(10):49–70, 1989.

[9] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302, Salt Lake City, Utah, United States, 1984. ACM.

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

[11] 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 *ACM SIGPLAN Notices*, PLDI '09, page 465–478, New York, NY, USA, 2009. ACM. ACM ID: 1542528.

[12] 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.

[13] A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the*

2nd international conference on Virtual execution environments, pages 144–153, Ottawa, Ontario, Canada, 2006. ACM.

[14] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, 2007.

[15] U. Hölzle, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38. Springer-Verlag, 1991.

[16] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 326–336, Orlando, Florida, United States, 1994. ACM.

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

[18] A. Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26, Verona, Italy, 2004. ACM.

[19] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 944–953, Portland, Oregon, USA, 2006. ACM.

[20] A. Rigo and S. Pedroni. JIT compiler architecture. Technical Report D08.2, PyPy, May 2007.

[21] J. R. Rose. Bytecodes meet combinators: invokedynamic on the JVM. *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, page 2:1–2:11, 2009. ACM ID: 1711508.

[22] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic native optimization of interpreters. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 50–57, San Diego, California, 2003. ACM.

[23] 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.

[24] A. Yermolovich, C. Wimmer, and M. Franz. Optimization of dynamic languages using hierarchical layering of virtual machines. In *Proceedings of the 5th symposium on Dynamic languages*, pages 79–88, Orlando, Florida, USA, 2009. ACM.